Fundamental Data Structures
# Contents

## Articles

### Introduction
- Abstract data type 1
- Data structure 9
- Analysis of algorithms 11
- Amortized analysis 16
- Accounting method 18
- Potential method 20

### Sequences
- Array data type 22
- Array data structure 26
- Dynamic array 31
- Linked list 34
- Doubly linked list 50
- Stack (abstract data type) 54
- Queue (abstract data type) 82
- Double-ended queue 85
- Circular buffer 88

### Dictionaries
- Associative array 103
- Association list 106
- Hash table 107
- Linear probing 120
- Quadratic probing 121
- Double hashing 125
- Cuckoo hashing 126
- Hopscotch hashing 130
- Hash function 131
- Perfect hash function 140
- Universal hashing 141
- K-independent hashing 146
- Tabulation hashing 147
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptographic hash function</td>
<td>150</td>
</tr>
<tr>
<td><strong>Sets</strong></td>
<td>157</td>
</tr>
<tr>
<td>Set (abstract data type)</td>
<td>157</td>
</tr>
<tr>
<td>Bit array</td>
<td>161</td>
</tr>
<tr>
<td>Bloom filter</td>
<td>166</td>
</tr>
<tr>
<td>MinHash</td>
<td>176</td>
</tr>
<tr>
<td>Disjoint-set data structure</td>
<td>179</td>
</tr>
<tr>
<td>Partition refinement</td>
<td>183</td>
</tr>
<tr>
<td><strong>Priority queues</strong></td>
<td>185</td>
</tr>
<tr>
<td>Priority queue</td>
<td>185</td>
</tr>
<tr>
<td>Heap (data structure)</td>
<td>190</td>
</tr>
<tr>
<td>Binary heap</td>
<td>192</td>
</tr>
<tr>
<td>$d$-ary heap</td>
<td>198</td>
</tr>
<tr>
<td>Binomial heap</td>
<td>200</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>205</td>
</tr>
<tr>
<td>Pairing heap</td>
<td>210</td>
</tr>
<tr>
<td>Double-ended priority queue</td>
<td>213</td>
</tr>
<tr>
<td>Soft heap</td>
<td>218</td>
</tr>
<tr>
<td><strong>Successors and neighbors</strong></td>
<td>221</td>
</tr>
<tr>
<td>Binary search algorithm</td>
<td>221</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>228</td>
</tr>
<tr>
<td>Random binary tree</td>
<td>238</td>
</tr>
<tr>
<td>Tree rotation</td>
<td>241</td>
</tr>
<tr>
<td>Self-balancing binary search tree</td>
<td>244</td>
</tr>
<tr>
<td>Treap</td>
<td>246</td>
</tr>
<tr>
<td>AVL tree</td>
<td>249</td>
</tr>
<tr>
<td>Red–black tree</td>
<td>253</td>
</tr>
<tr>
<td>Scapegoat tree</td>
<td>268</td>
</tr>
<tr>
<td>Splay tree</td>
<td>272</td>
</tr>
<tr>
<td>Tango tree</td>
<td>286</td>
</tr>
<tr>
<td>Skip list</td>
<td>308</td>
</tr>
<tr>
<td>B-tree</td>
<td>314</td>
</tr>
<tr>
<td>B+ tree</td>
<td>325</td>
</tr>
<tr>
<td><strong>Integer and string searching</strong></td>
<td>330</td>
</tr>
<tr>
<td>Trie</td>
<td>330</td>
</tr>
</tbody>
</table>
Radix tree 337
Directed acyclic word graph 339
Suffix tree 341
Suffix array 346
van Emde Boas tree 349
Fusion tree 353

References

Article Sources and Contributors 354
Image Sources, Licenses and Contributors 359

Article Licenses

License 362
Introduction

Abstract data type

In computing, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.[1]

For example, an abstract stack data structure could be defined by three operations: push, that inserts some data item onto the structure, pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and peek, that allows data on top of the structure to be examined without removal. When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

The term abstract data type can also be regarded as a generalised approach of a number of algebraic structures, such as lattices, groups, and rings.[2] This can be treated as part of subject area of Artificial intelligence. The notion of abstract data types is related to the concept of data abstraction, important in object-oriented programming and design by contract methodologies for software development.

Defining an abstract data type (ADT)

An Abstract Data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Imperative abstract data type definitions

In the "imperative" view, which is closer to the philosophy of imperative programming languages, an abstract data structure is conceived as an entity that is mutable — meaning that it may be in different states at different times. Some operations may change the state of the ADT; therefore, the order in which operations are evaluated is important, and the same operation on the same entities may have different effects if executed at different times — just like the instructions of a computer, or the commands and procedures of an imperative language. To underscore this view, it is customary to say that the operations are executed or applied, rather than evaluated. The imperative style is often used when describing abstract algorithms. This is described by Donald E. Knuth and can be referenced from here The Art of Computer Programming.
Abstract data type

Imperative ADT definitions often depend on the concept of an *abstract variable*, which may be regarded as the simplest non-trivial ADT. An abstract variable $V$ is a mutable entity that admits two operations:

- $\text{store}(V,x)$ where $x$ is a *value* of unspecified nature; and
- $\text{fetch}(V)$, that yields a value;

with the constraint that

- $\text{fetch}(V)$ always returns the value $x$ used in the most recent $\text{store}(V,x)$ operation on the same variable $V$.

As in so many programming languages, the operation $\text{store}(V,x)$ is often written $V \leftarrow x$ (or some similar notation), and $\text{fetch}(V)$ is implied whenever a variable $V$ is used in a context where a value is required. Thus, for example, $V \leftarrow V + 1$ is commonly understood to be a shorthand for $\text{store}(V,\text{fetch}(V) + 1)$.

In this definition, it is implicitly assumed that storing a value into a variable $U$ has no effect on the state of a distinct variable $V$. To make this assumption explicit, one could add the constraint that

- if $U$ and $V$ are distinct variables, the sequence $\{ \text{store}(U,x); \text{store}(V,y) \}$ is equivalent to $\{ \text{store}(V,y); \text{store}(U,x) \}$.

More generally, ADT definitions often assume that any operation that changes the state of one ADT instance has no effect on the state of any other instance (including other instances of the same ADT) — unless the ADT axioms imply that the two instances are connected (aliased) in that sense. For example, when extending the definition of abstract variable to include abstract records, the operation that selects a field from a record variable $R$ must yield a variable $V$ that is aliased to that part of $R$.

The definition of an abstract variable $V$ may also restrict the stored values $x$ to members of a specific set $X$, called the *range* or *type* of $V$. As in programming languages, such restrictions may simplify the description and analysis of algorithms, and improve their readability.

Note that this definition does not imply anything about the result of evaluating $\text{fetch}(V)$ when $V$ is *un-initialized*, that is, before performing any $\text{store}$ operation on $V$. An algorithm that does so is usually considered invalid, because its effect is not defined. (However, there are some important algorithms whose efficiency strongly depends on the assumption that such a $\text{fetch}$ is legal, and returns some arbitrary value in the variable's range.)

**Instance creation**

Some algorithms need to create new instances of some ADT (such as new variables, or new stacks). To describe such algorithms, one usually includes in the ADT definition a $\text{create}()$ operation that yields an instance of the ADT, usually with axioms equivalent to

- the result of $\text{create}()$ is distinct from any instance $S$ in use by the algorithm.

This axiom may be strengthened to exclude also partial aliasing with other instances. On the other hand, this axiom still allows implementations of $\text{create}()$ to yield a previously created instance that has become inaccessible to the program.
Preconditions, postconditions, and invariants

In imperative-style definitions, the axioms are often expressed by preconditions, that specify when an operation may be executed; postconditions, that relate the states of the ADT before and after the execution of each operation; and invariants, that specify properties of the ADT that are not changed by the operations.

Example: abstract stack (imperative)

As another example, an imperative definition of an abstract stack could specify that the state of a stack \( S \) can be modified only by the operations

- \( \text{push}(S,x) \), where \( x \) is some value of unspecified nature; and
- \( \text{pop}(S) \), that yields a value as a result;

with the constraint that

- For any value \( x \) and any abstract variable \( V \), the sequence of operations \( \{ \text{push}(S,x); V \leftarrow \text{pop}(S) \} \) is equivalent to \( \{ V \leftarrow x \} \);

Since the assignment \( V \leftarrow x \), by definition, cannot change the state of \( S \), this condition implies that \( \{ V \leftarrow \text{pop}(S) \} \) restores \( S \) to the state it had before the \( \{ \text{push}(S,x) \} \). From this condition and from the properties of abstract variables, it follows, for example, that the sequence

\[
\{ \text{push}(S,x); \text{push}(S,y); U \leftarrow \text{pop}(S); \text{push}(S,z); V \leftarrow \text{pop}(S); W \leftarrow \text{pop}(S); \}
\]

where \( x, y, \) and \( z \) are any values, and \( U, V, W \) are pairwise distinct variables, is equivalent to

\[
\{ U \leftarrow y; V \leftarrow z; W \leftarrow x \}
\]

Here it is implicitly assumed that operations on a stack instance do not modify the state of any other ADT instance, including other stacks; that is,

- For any values \( x, y, \) and any distinct stacks \( S \) and \( T \), the sequence \( \{ \text{push}(S,x); \text{push}(T,y) \} \) is equivalent to \( \{ \text{push}(T,y); \text{push}(S,x) \} \).

A stack ADT definition usually includes also a Boolean-valued function \( \text{empty}(S) \) and a \( \text{create}() \) operation that returns a stack instance, with axioms equivalent to

- \( \text{create}() \neq S \) for any stack \( S \) (a newly created stack is distinct from all previous stacks)
- \( \text{empty}(\text{create}()) \) (a newly created stack is empty)
- \( \text{not empty}(\text{push}(S,x)) \) (pushing something into a stack makes it non-empty)

Single-instance style

Sometimes an ADT is defined as if only one instance of it existed during the execution of the algorithm, and all operations were applied to that instance, which is not explicitly notated. For example, the abstract stack above could have been defined with operations \( \text{push}(x) \) and \( \text{pop}() \), that operate on "the" only existing stack. ADT definitions in this style can be easily rewritten to admit multiple coexisting instances of the ADT, by adding an explicit instance parameter (like \( S \) in the previous example) to every operation that uses or modifies the implicit instance.

On the other hand, some ADTs cannot be meaningfully defined without assuming multiple instances. This is the case when a single operation takes two distinct instances of the ADT as parameters. For an example, consider augmenting the definition of the stack ADT with an operation \( \text{compare}(S,T) \) that checks whether the stacks \( S \) and \( T \) contain the same items in the same order.
Functional ADT definitions

Another way to define an ADT, closer to the spirit of functional programming, is to consider each state of the structure as a separate entity. In this view, any operation that modifies the ADT is modeled as a mathematical function that takes the old state as an argument, and returns the new state as part of the result. Unlike the "imperative" operations, these functions have no side effects. Therefore, the order in which they are evaluated is immaterial, and the same operation applied to the same arguments (including the same input states) will always return the same results (and output states).

In the functional view, in particular, there is no way (or need) to define an "abstract variable" with the semantics of imperative variables (namely, with fetch and store operations). Instead of storing values into variables, one passes them as arguments to functions.

Example: abstract stack (functional)

For example, a complete functional-style definition of a stack ADT could use the three operations:

- **push**: takes a stack state and an arbitrary value, returns a stack state;
- **top**: takes a stack state, returns a value;
- **pop**: takes a stack state, returns a stack state;

with the following axioms:

- \( \text{top}(\text{push}(s, x)) = x \) (pushing an item onto a stack leaves it at the top)
- \( \text{pop}(\text{push}(s, x)) = s \) (pop undoes the effect of push)

In a functional-style definition there is no need for a create operation. Indeed, there is no notion of "stack instance". The stack states can be thought of as being potential states of a single stack structure, and two stack states that contain the same values in the same order are considered to be identical states. This view actually mirrors the behavior of some concrete implementations, such as linked lists with hash cons.

Instead of create(), a functional definition of a stack ADT may assume the existence of a special stack state, the empty stack, designated by a special symbol like Λ or "()"; or define a bottom() operation that takes no arguments and returns this special stack state. Note that the axioms imply that

- \( \text{push}(\Lambda, x) \neq \Lambda \)

In a functional definition of a stack one does not need an empty predicate: instead, one can test whether a stack is empty by testing whether it is equal to Λ.

Note that these axioms do not define the effect of top(s) or pop(s), unless s is a stack state returned by a push. Since push leaves the stack non-empty, those two operations are undefined (hence invalid) when \( s = \Lambda \). On the other hand, the axioms (and the lack of side effects) imply that \( \text{push}(s, x) = \text{push}(t, y) \) if and only if \( x = y \) and \( s = t \).

As in some other branches of mathematics, it is customary to assume also that the stack states are only those whose existence can be proved from the axioms in a finite number of steps. In the stack ADT example above, this rule means that every stack is a finite sequence of values, that becomes the empty stack (Λ) after a finite number of pops.

By themselves, the axioms above do not exclude the existence of infinite stacks (that can be popped forever, each time yielding a different state) or circular stacks (that return to the same state after a finite number of pops). In particular, they do not exclude states \( s \) such that \( \text{pop}(s) = s \) or \( \text{push}(s, x) = s \) for some \( x \). However, since one cannot obtain such stack states with the given operations, they are assumed "not to exist".
Advantages of abstract data typing

• Encapsulation
Abstraction provides a promise that any implementation of the ADT has certain properties and abilities; knowing these is all that is required to make use of an ADT object. The user does not need any technical knowledge of how the implementation works to use the ADT. In this way, the implementation may be complex but will be encapsulated in a simple interface when it is actually used.

• Localization of change
Code that uses an ADT object will not need to be edited if the implementation of the ADT is changed. Since any changes to the implementation must still comply with the interface, and since code using an ADT may only refer to properties and abilities specified in the interface, changes may be made to the implementation without requiring any changes in code where the ADT is used.

• Flexibility
Different implementations of an ADT, having all the same properties and abilities, are equivalent and may be used somewhat interchangeably in code that uses the ADT. This gives a great deal of flexibility when using ADT objects in different situations. For example, different implementations of an ADT may be more efficient in different situations; it is possible to use each in the situation where they are preferable, thus increasing overall efficiency.

Typical operations
Some operations that are often specified for ADTs (possibly under other names) are

• \texttt{compare}\(s,t\), that tests whether two structures are equivalent in some sense;
• \texttt{hash}\(s\), that computes some standard hash function from the instance's state;
• \texttt{print}\(s\) or \texttt{show}\(s\), that produces a human-readable representation of the structure's state.

In imperative-style ADT definitions, one often finds also

• \texttt{create}\(), that yields a new instance of the ADT;
• \texttt{initialize}\(s\), that prepares a newly-created instance \(s\) for further operations, or resets it to some "initial state";
• \texttt{copy}\(s,t\), that puts instance \(s\) in a state equivalent to that of \(t\);
• \texttt{clone}\(t\), that performs \(s \leftarrow \texttt{new}\(), \texttt{copy}\(s,t\), and returns \(s\);
• \texttt{free}\(s\) or \texttt{destroy}\(s\), that reclaims the memory and other resources used by \(s\);

The \texttt{free} operation is not normally relevant or meaningful, since ADTs are theoretical entities that do not "use memory". However, it may be necessary when one needs to analyze the storage used by an algorithm that uses the ADT. In that case one needs additional axioms that specify how much memory each ADT instance uses, as a function of its state, and how much of it is returned to the pool by \texttt{free}.

Examples
Some common ADTs, which have proved useful in a great variety of applications, are

• Container
• Deque
• List
• Map
• Multimap
• Multiset
• Priority queue
• Queue
Abstract data type

• Set
• Stack
• String
• Tree

Each of these ADTs may be defined in many ways and variants, not necessarily equivalent. For example, a stack ADT may or may not have a `count` operation that tells how many items have been pushed and not yet popped. This choice makes a difference not only for its clients but also for the implementation.

Implementation

Implementing an ADT means providing one procedure or function for each abstract operation. The ADT instances are represented by some concrete data structure that is manipulated by those procedures, according to the ADT's specifications.

Usually there are many ways to implement the same ADT, using several different concrete data structures. Thus, for example, an abstract stack can be implemented by a linked list or by an array.

An ADT implementation is often packaged as one or more modules, whose interface contains only the signature (number and types of the parameters and results) of the operations. The implementation of the module — namely, the bodies of the procedures and the concrete data structure used — can then be hidden from most clients of the module. This makes it possible to change the implementation without affecting the clients.

When implementing an ADT, each instance (in imperative-style definitions) or each state (in functional-style definitions) is usually represented by a handle of some sort. Modern object-oriented languages, such as C++ and Java, support a form of abstract data types. When a class is used as a type, it is a abstract type that refers to a hidden representation. In this model an ADT is typically implemented as class, and each instance of the ADT is an object of that class. The module's interface typically declares the constructors as ordinary procedures, and most of the other ADT operations as methods of that class. However, such an approach does not easily encapsulate multiple representational variants found in an ADT. It also can undermine the extensibility of object-oriented programs. In a pure object-oriented program that uses interfaces as types, types refer to behaviors not representations.

Example: implementation of the stack ADT

As an example, here is an implementation of the stack ADT above in the C programming language.

**Imperative-style interface**

An imperative-style interface might be:

```c
typedef struct stack_Rep stack_Rep;    /* Type: instance representation (an opaque record). */
typedef stack_Rep *stack_T;             /* Type: handle to a stack instance (an opaque pointer). */
typedef void *stack_Item;               /* Type: value that can be stored in stack (arbitrary address). */

stack_T stack_create(void);            /* Create new stack instance, initially empty. */
void stack_push(stack_T s, stack_Item e); /* Add an item at the top of the stack. */
stack_Item stack_pop(stack_T s);        /* Remove the top item from the stack and return it. */
```
This implementation could be used in the following manner:

```c
#include <stack.h> /* Include the stack interface. */
stack_T t = stack_create(); /* Create a stack instance. */
int foo = 17; /* An arbitrary datum. */
t = stack_push(t, &foo); /* Push the address of 'foo' onto the stack. */
...
void *e = stack_pop(t); /* Get the top item and delete it from the stack. */
if (stack_empty(t)) { ... } /* Do something if stack is empty. */
...
```

This interface can be implemented in many ways. The implementation may be arbitrarily inefficient, since the formal definition of the ADT, above, does not specify how much space the stack may use, nor how long each operation should take. It also does not specify whether the stack state `t` continues to exist after a call `s ← pop(t)`.

In practice the formal definition should specify that the space is proportional to the number of items pushed and not yet popped; and that every one of the operations above must finish in a constant amount of time, independently of that number. To comply with these additional specifications, the implementation could use a linked list, or an array (with dynamic resizing) together with two integers (an item count and the array size).

**Functional-style interface**

Functional-style ADT definitions are more appropriate for functional programming languages, and vice-versa. However, one can provide a functional style interface even in an imperative language like C. For example:

```c
typedef struct stack_Rep stack_Rep; /* Type: stack state representation (an opaque record). */
typedef stack_Rep *stack_T; /* Type: handle to a stack state (an opaque pointer). */
typedef void *stack_Item; /* Type: item (arbitrary address). */
stack_T stack_empty(void); /* Returns the empty stack state. */
stack_T stack_push(stack_T s, stack_Item x); /* Adds x at the top of s, returns the resulting state. */
stack_Item stack_top(stack_T s); /* Returns the item currently at the top of s. */
stack_T stack_pop(stack_T s); /* Remove the top item from s, returns the resulting state. */
```

The main problem is that C lacks garbage collection, and this makes this style of programming impractical; moreover, memory allocation routines in C are slower than allocation in a typical garbage collector, thus the performance impact of so many allocations is even greater.
ADT libraries

Many modern programming languages, such as C++ and Java, come with standard libraries that implement several common ADTs, such as those listed above.

Built-in abstract data types

The specification of some programming languages is intentionally vague about the representation of certain built-in data types, defining only the operations that can be done on them. Therefore, those types can be viewed as "built-in ADTs". Examples are the arrays in many scripting languages, such as Awk, Lua, and Perl, which can be regarded as an implementation of the Map or Table ADT.

References


Further


External links

• Abstract data type (http://www.nist.gov/dads/HTML/abstractDataType.html) in NIST Dictionary of Algorithms and Data Structures
Data structure

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.\[^1\] [\(^2\)]

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Data structures are used in almost every program or software system. Data structures provide a means to manage huge amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Overview

- An array stores a number of elements of the same type in a specific order. They are accessed using an integer to specify which element is required (although the elements may be of almost any type). Arrays may be fixed-length or expandable.
- Record (also called tuple or struct) Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called fields or members.
- A hash or dictionary or map is a more flexible variation on a record, in which name-value pairs can be added and deleted freely.
- Union. A union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float and an integer; whereas, in a union, there is only one value at a time.
- A tagged union (also called a variant, variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.
- A set is an abstract data structure that can store certain values, without any particular order, and no repeated values. Values themselves are not retrieved from sets, rather one tests a value for membership to obtain a boolean "in" or "not in".
- An object contains a number of data fields, like a record, and also a number of program code fragments for accessing or modifying them. Data structures not containing code, like those above, are called plain old data structure.

Many others are possible, but they tend to be further variations and compounds of the above.

Basic principles

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address—a bit string that can be itself stored in memory and manipulated by the program. Thus the record and array data structures are based on computing the addresses of data items with arithmetic operations; while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways (as in XOR linking).

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).
Language support

Most assembly languages and some low-level languages, such as BCPL, lack support for data structures. Many high-level programming languages, and some higher-level assembly languages, such as MASM, on the other hand, have special syntax or other built-in support for certain data structures, such as vectors (one-dimensional arrays) in the C language or multi-dimensional arrays in Pascal.

Most programming languages feature some sorts of library mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. Examples are the C++ Standard Template Library, the Java Collections Framework, and Microsoft's .NET Framework.

Modern languages also generally support modular programming, the separation between the interface of a library module and its implementation. Some provide opaque data types that allow clients to hide implementation details. Object-oriented programming languages, such as C++, Java and .NET Framework use classes for this purpose.

Many known data structures have concurrent versions that allow multiple computing threads to access the data structure simultaneously.

References


Further readings

• Peter Brass, Advanced Data Structures, Cambridge University Press, 2008.

External links

• UC Berkeley video course on data structures (http://academicearth.org/courses/data-structures)
• Descriptions (http://nist.gov/dads/) from the Dictionary of Algorithms and Data Structures
• CSE.unr.edu (http://www.cse.unr.edu/~bebis/CS308/)
• Data structures course with animations (http://www.cs.auckland.ac.nz/software/AlgAnim/ds_ToC.html)
• Data structure tutorials with animations (http://courses.cs.vt.edu/~csonline/DataStructures/Lessons/index.html)
• An Examination of Data Structures from .NET perspective (http://msdn.microsoft.com/en-us/library/aa289148(VS.71).aspx)
• Schaffer, C. Data Structures and Algorithm Analysis (http://people.cs.vt.edu/~shafffer/Book/C++3e20110915.pdf)
Analysis of algorithms

To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, omega notation and theta notation are used to this end. For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in $O(\log(n))$, colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted list to which we apply binary search has $n$ elements, and we can guarantee that each lookup of an element in the list can be done in unit time, then at most $\log_2 n + 1$ time units are needed to return an answer.

Cost models

Time efficiency estimates depend on what we define to be a step. For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bounded above by a constant. One must be careful here; for instance, some analyses count an addition of two numbers as one step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.

Two cost models are generally used,[1][2][3][4][5]

• the uniform cost model, also called uniform-cost measurement (and similar variations), assigns a constant cost to every machine operation, regardless of the size of the numbers involved

• the logarithmic cost model, also called logarithmic-cost measurement (and variations thereof), assigns a cost to every machine operation proportional to the number of bits involved

The latter is more cumbersome to use, so it's only employed when necessary, for example in the analysis of arbitrary-precision arithmetic algorithms, like those used in cryptography.

A key point which is often overlooked is that published lower bounds for problems are often given for a model of computation that is more restricted than the set of operations that you could use in practice and therefore there are algorithms that are faster than what would naively be thought possible.[6]
Run-time analysis
Run-time analysis is a theoretical classification that estimates and anticipates the increase in running time (or run-time) of an algorithm as its input size (usually denoted as \(n\)) increases. Run-time efficiency is a topic of great interest in computer science: A program can take seconds, hours or even years to finish executing, depending on which algorithm it implements (see also performance analysis, which is the analysis of an algorithm's run-time in practice).

Shortcomings of empirical metrics
Since algorithms are platform-independent (i.e. a given algorithm can be implemented in an arbitrary programming language on an arbitrary computer running an arbitrary operating system), there are significant drawbacks to using an empirical approach to gauge the comparative performance of a given set of algorithms.

Take as an example a program that looks up a specific entry in a sorted list of size \(n\). Suppose this program were implemented on Computer A, a state-of-the-art machine, using a linear search algorithm, and on Computer B, a much slower machine, using a binary search algorithm. Benchmark testing on the two computers running their respective programs might look something like the following:

<table>
<thead>
<tr>
<th>(n) (list size)</th>
<th>Computer A run-time (in nanoseconds)</th>
<th>Computer B run-time (in nanoseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>7 ns</td>
<td>100,000 ns</td>
</tr>
<tr>
<td>65</td>
<td>32 ns</td>
<td>150,000 ns</td>
</tr>
<tr>
<td>250</td>
<td>125 ns</td>
<td>200,000 ns</td>
</tr>
<tr>
<td>1,000</td>
<td>500 ns</td>
<td>250,000 ns</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1,000,000</td>
<td>500,000 ns</td>
<td>500,000 ns</td>
</tr>
<tr>
<td>4,000,000</td>
<td>2,000,000 ns</td>
<td>550,000 ns</td>
</tr>
<tr>
<td>16,000,000</td>
<td>8,000,000 ns</td>
<td>600,000 ns</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63,072 \times 10^{12}</td>
<td>31,536 \times 10^{12} ns, or 1 year</td>
<td>1,375,000 ns, or 1.375 milliseconds</td>
</tr>
</tbody>
</table>

Based on these metrics, it would be easy to jump to the conclusion that Computer A is running an algorithm that is far superior in efficiency to that of Computer B. However, if the size of the input-list is increased to a sufficient number, that conclusion is dramatically demonstrated to be in error:

Computer A, running the linear search program, exhibits a linear growth rate. The program's run-time is directly proportional to its input size. Doubling the input size doubles the run time, quadrupling the input size quadruples the run-time, and so forth. On the other hand, Computer B, running the binary search program, exhibits a logarithmic
growth rate. Doubling the input size only increases the run time by a constant amount (in this example, 25,000 ns). Even though Computer A is ostensibly a faster machine, Computer B will inevitably surpass Computer A in run-time because it’s running an algorithm with a much slower growth rate.

Orders of growth

Informally, an algorithm can be said to exhibit a growth rate on the order of a mathematical function if beyond a certain input size \( n \), the function \( f(n) \) times a positive constant provides an upper bound or limit for the run-time of that algorithm. In other words, for a given input size \( n \) greater than some \( n_0 \) and a constant \( c \), the running time of that algorithm will never be larger than \( c \times f(n) \). This concept is frequently expressed using Big O notation. For example, since the run-time of insertion sort grows quadratically as its input size increases, insertion sort can be said to be of order \( O(n^2) \).

Big O notation is a convenient way to express the worst-case scenario for a given algorithm, although it can also be used to express the average-case — for example, the worst-case scenario for quicksort is \( O(n^2) \), but the average-case run-time is \( O(n \log n) \).[7]

Evaluating run-time complexity

The run-time complexity for the worst-case scenario of a given algorithm can sometimes be evaluated by examining the structure of the algorithm and making some simplifying assumptions. Consider the following pseudocode:

```plaintext
1   get a positive integer from input
2   if n > 10
3       print "This might take a while..."
4   for i = 1 to n
5       for j = 1 to i
6           print i * j
7   print "Done!"
```

A given computer will take a discrete amount of time to execute each of the instructions involved with carrying out this algorithm. The specific amount of time to carry out a given instruction will vary depending on which instruction is being executed and which computer is executing it, but on a conventional computer, this amount will be deterministic.[8] Say that the actions carried out in step 1 are considered to consume time \( T_1 \), step 2 uses time \( T_2 \), and so forth.

In the algorithm above, steps 1, 2 and 7 will only be run once. For a worst-case evaluation, it should be assumed that step 3 will be run as well. Thus the total amount of time to run steps 1-3 and step 7 is:

\[
T_1 + T_2 + T_3 + T_7.
\]

The loops in steps 4, 5 and 6 are trickier to evaluate. The outer loop test in step 4 will execute \( (n + 1) \) times (note that an extra step is required to terminate the for loop, hence \( n + 1 \) and not \( n \) executions), which will consume \( T_4(n + 1) \) time. The inner loop, on the other hand, is governed by the value of \( i \), which iterates from 1 to \( n \). On the first pass through the outer loop, \( j \) iterates from 1 to 1: The inner loop makes one pass, so running the inner loop body (step 6) consumes \( T_6 \) time, and the inner loop test (step 5) consumes \( 2T_5 \) time. During the next pass through the outer loop, \( j \) iterates from 1 to 2: the inner loop makes two passes, so running the inner loop body (step 6) consumes \( 2T_6 \) time, and the inner loop test (step 5) consumes \( 3T_5 \) time.

Altogether, the total time required to run the inner loop body can be expressed as an arithmetic progression:

\[
T_6 + 2T_6 + 3T_6 + \cdots + (n - 1)T_6 + nT_6
\]

which can be factored[9] as
The total time required to run the inner loop test can be evaluated similarly:

\[
T_6 [1 + 2 + 3 + \cdots + (n - 1) + n] = T_6 \left[ \frac{1}{2} (n^2 + n) \right]
\]

which can be factored as

\[
T_6 [1 + 2 + 3 + \cdots + (n - 1) + n + (n + 1)] - T_6 = T_6 \left[ \frac{1}{2} (n^2 + 3n + 2) \right] - T_6
\]

Therefore the total running time for this algorithm is:

\[
f(n) = T_1 + T_2 + T_3 + T_7 + (n+1)T_4 + T_6 \left[ \frac{1}{2} (n^2 + n) \right] T_5 + T_6 \left[ \frac{1}{2} (n^2 + 3n + 2) \right] T_5 - T_5
\]

As a rule-of-thumb, one can assume that the highest-order term in any given function dominates its rate of growth and thus defines its run-time order. In this example, \(n^2\) is the highest-order term, so one can conclude that \(f(n) = O(n^2)\). Formally this can be proven as follows:

Prove that

\[
\left[ \frac{1}{2} (n^2 + n) \right] T_6 + \left[ \frac{1}{2} (n^2 + 3n) \right] T_5 + (n+1)T_4 + T_6 \left[ \frac{1}{2} (n^2 + 3n + 2) \right] T_5 - T_5 \leq cn^2, \quad n \geq n_0
\]

\[
\left[ \frac{1}{2} (n^2 + n) \right] T_6 + \left[ \frac{1}{2} (n^2 + 3n) \right] T_5 + (n+1)T_4 + T_6 \left[ \frac{1}{2} (n^2 + 3n + 2) \right] T_5 - T_5 \leq (n^2 + n)T_6 + (n^2 + 3n)T_5 + (n+1)T_4 + T_6 \left[ \frac{1}{2} (n^2 + 3n + 2) \right] T_5 - T_5 \quad \text{(for } n \geq 0)\]

Let \(k\) be a constant greater than or equal to \([T_1..T_7]\)

\[
T_6(n^2+n)+T_5(n^2+3n)+(n+1)T_4+T_6 \left[ \frac{1}{2} (n^2 + 3n + 2) \right] T_5 - T_5 \leq k(n^2+n)+k(n^2+3n)+kn+5k
\]

\[
= 2kn^2 + 5kn + 5k \leq 2kn^2 + 5kn^2 + 5kn^2 \quad \text{(for } n \geq 1) = 12kn^2
\]

Therefore

\[
\left[ \frac{1}{2} (n^2 + n) \right] T_6 + \left[ \frac{1}{2} (n^2 + 3n) \right] T_5 + (n+1)T_4 + T_6 \left[ \frac{1}{2} (n^2 + 3n + 2) \right] T_5 - T_5 \leq cn^2, \quad n \geq n_0
\]

for \(c = 12k; n_0 = 1\)

A more elegant approach to analyzing this algorithm would be to declare that \([T_1..T_7]\) are all equal to one unit of time greater than or equal to \([T_1..T_7]\). This would mean that the algorithm's running time breaks down as follows:

\[
4 + \sum_{i=1}^{n} i \leq 4 + \sum_{i=1}^{n} n = 4 + n^2 \leq 5n^2 \quad \text{(for } n \geq 1) = O(n^2).
\]

Growth rate analysis of other resources

The methodology of run-time analysis can also be utilized for predicting other growth rates, such as consumption of memory space. As an example, consider the following pseudocode which manages and reallocates memory usage by a program based on the size of a file which that program manages:

```plaintext
while (file still open)
    let n = size of file
    for every 100,000 kilobytes of increase in file size
        double the amount of memory reserved
```

In this instance, as the file size \(n\) increases, memory will be consumed at an exponential growth rate, which is order \(O(2^n)\).
Relevance

Algorithm analysis is important in practice because the accidental or unintentional use of an inefficient algorithm can significantly impact system performance. In time-sensitive applications, an algorithm taking too long to run can render its results outdated or useless. An inefficient algorithm can also end up requiring an uneconomical amount of computing power or storage in order to run, again rendering it practically useless.

Notes


[7] The term \( \lg \) is often used as shorthand for \( \log_2 \)

[8] However, this is not the case with a quantum computer

[9] It can be proven by induction that \( 1 + 2 + 3 + \cdots + (n - 1) + n = \frac{n(n + 1)}{2} \)

[10] This approach, unlike the above approach, neglects the constant time consumed by the loop tests which terminate their respective loops, but it is trivial to prove that such omission does not affect the final result

[11] Note that this is an extremely rapid and most likely unmanageable growth rate for consumption of memory resources

References


• Knuth, Donald. The Art of Computer Programming. Addison-Wesley.


Amortized analysis

In computer science, amortized analysis is a method of analyzing algorithms that considers the entire sequence of operations of the program. It allows for the establishment of a worst-case bound for the performance of an algorithm irrespective of the inputs by looking at all of the operations. At the heart of the method is the idea that while certain operations may be extremely costly in resources, they cannot occur at a high-enough frequency to weigh down the entire program because the number of less costly operations will far outnumber the costly ones in the long run, "paying back" the program over a number of iterations.\(^1\) It is particularly useful because it guarantees worst-case performance while accounting for the entire set of operations in an algorithm.

History

Amortized analysis initially emerged from a method called aggregate analysis, which is now subsumed by amortized analysis. However, the technique was first formally introduced by Robert Tarjan in his paper *Amortized Computational Complexity*, which addressed the need for a more useful form of analysis than the common probabilistic methods used. Amortization was initially used for very specific types of algorithms, particularly those involving binary trees and union operations. However, it is now ubiquitous and comes into play when analyzing many other algorithms as well.\(^1\)

Method

The method requires knowledge of which series of operations are possible. This is most commonly the case with data structures, which have state that persists between operations. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give the same answers, and their usage difference is primarily circumstantial and due to individual preference.\(^2\)

- Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of $n$ operations, then calculates the average cost to be $T(n) / n$.\(^2\)
- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.\(^2\)
- The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.\(^2\)

Examples

As a simple example, in a specific implementation of the dynamic array, we double the size of the array each time it fills up. Because of this, array reallocation may be required, and in the worst case an insertion may require $O(n)$. However, a sequence of $n$ insertions can always be done in $O(n)$ time, because the rest of the insertions are done in constant time, so $n$ insertions can be completed in $O(n)$ time. The amortized time per operation is therefore $O(n) / n = O(1)$.

Another way to see this is to think of a sequence of $n$ operations. There are 2 possible operations: a regular insertion which requires a constant $c$ time to perform (assume $c = 1$), and an array doubling which requires $O(j)$ time (where $j$ is the size of the array at the time of the doubling). Clearly the time to perform these operations is less than the time needed to perform $n$ regular insertions in addition to the number of array doublings that would have taken place in the original sequence of $n$ operations. There are only as many array doublings in the sequence as there are...
powers of 2 between 0 and \( n (\lg(n)) \). Therefore the cost of a sequence of \( n \) operations is strictly less than the below expression:[3]

\[
n + \sum_{j=0}^{\lfloor \lg(n) \rfloor} 2^j = 3n
\]

The *amortized* time per operation is the worst-case time bound on a series of \( n \) operations divided by \( n \). The *amortized* time per operation is therefore \( O(3n)/n = O(n)/n = O(1) \).

**Comparison to other methods**

Notice that average-case analysis and probabilistic analysis of probabilistic algorithms are not the same thing as amortized analysis. In average-case analysis, we are averaging over all possible inputs; in probabilistic analysis of probabilistic algorithms, we are averaging over all possible random choices; in amortized analysis, we are averaging over a sequence of operations. Amortized analysis assumes worst-case input and typically does not allow random choices.

An average-case analysis for an algorithm is problematic because the user is dependent on the fact that a given set of inputs will not trigger the worst case scenario. A worst-case analysis has the property of often returning an overly pessimistic performance for a given algorithm when the probability of a worst-case operation occurring multiple times in a sequence is 0 for certain programs.

**Common use**

- In common usage, an "amortized algorithm" is one that an amortized analysis has shown to perform well.
- Online algorithms commonly use amortized analysis.

**References**

Accounting method

In the field of analysis of algorithms in computer science, the accounting method is a method of amortized analysis based on accounting. The accounting method often gives a more intuitive account of the amortized cost of an operation than either aggregate analysis or the potential method. Note, however, that this does not guarantee such analysis will be immediately obvious; often, choosing the correct parameters for the accounting method requires as much knowledge of the problem and the complexity bounds one is attempting to prove as the other two methods.

The accounting method is most naturally suited for proving an O(1) bound on time. The method as explained here is for proving such a bound.

The method

Preliminarily, we choose a set of elementary operations which will be used in the algorithm, and arbitrarily set their cost to 1. The fact that the costs of these operations may in reality differ presents no difficulty in principle. What is important, is that each elementary operation has a constant cost.

Each aggregate operation is assigned a "payment". The payment is intended to cover the cost of elementary operations needed to complete this particular operation, with some of the payment left over, placed in a pool to be used later.

The difficulty with problems that require amortized analysis is that, in general, some of the operations will require greater than constant cost. This means that no constant payment will be enough to cover the worst case cost of an operation, in and of itself. With proper selection of payment, however, this is no longer a difficulty; the expensive operations will only occur when there is sufficient payment in the pool to cover their costs.

Examples

A few examples will help to illustrate the use of the accounting method.

Table expansion

It is often necessary to create a table before it is known how much space is needed. One possible strategy is to double the size of the table when it is full. Here we will use the accounting method to show that the amortized cost of an insertion operation in such a table is O(1).

Before looking at the procedure in detail, we need some definitions. Let T be a table, E an element to insert, num(T) the number of elements in T, and size(T) the allocated size of T. We assume the existence of operations create_table(n), which creates an empty table of size n, for now assumed to be free, and elementary_insert(T,E), which inserts element E into a table T that already has space allocated, with a cost of 1.

The following pseudocode illustrates the table insertion procedure:

```plaintext
function table_insert(T,E)
    if num(T) = size(T)
        U := create_table(2 × size(T))
        for each F in T
            elementary_insert(U,F)
        T := U
        elementary_insert(T,E)
```

Without amortized analysis, the best bound we can show for n insert operations is O(n^2) — this is due to the loop at line 4 that performs num(T) elementary insertions.
For analysis using the accounting method, we assign a payment of 3 to each table insertion. Although the reason for this is not clear now, it will become clear during the course of the analysis.

Assume that initially the table is empty with size(T) = m. The first m insertions therefore do not require reallocation and only have cost 1 (for the elementary insert). Therefore, when num(T) = m, the pool has (3 - 1)×m = 2m.

Inserting element m + 1 requires reallocation of the table. Creating the new table on line 3 is free (for now). The loop on line 4 requires m elementary insertions, for a cost of m. Including the insertion on the last line, the total cost for this operation is m + 1. After this operation, the pool therefore has 2m + 3 - (m + 1) = m + 2.

Next, we add another m - 1 elements to the table. At this point the pool has m + 2 + 2×(m - 1) = 3m. Inserting an additional element (that is, element 2m + 1) can be seen to have cost 2m + 1 and a payment of 3. After this operation, the pool has 3m + 3 - (2m + 1) = m + 2. Note that this is the same amount as after inserting element m + 1. In fact, we can show that this will be the case for any number of reallocations.

It can now be made clear why the payment for an insertion is 3. 1 goes to inserting the element the first time it is added to the table, 1 goes to moving it the next time the table is expanded, and 1 goes to moving one of the elements that was already in the table the next time the table is expanded.

We initially assumed that creating a table was free. In reality, creating a table of size n may be as expensive as O(n). Let us say that the cost of creating a table of size n is n. Does this new cost present a difficulty? Not really; it turns out we use the same method to show the amortized O(1) bounds. All we have to do is change the payment.

When a new table is created, there is an old table with m entries. The new table will be of size 2m. As long as the entries currently in the table have added enough to the pool to pay for creating the new table, we will be all right.

We cannot expect the first $\frac{m}{2}$ entries to help pay for the new table. Those entries already paid for the current table.

We must then rely on the last $\frac{m}{2}$ entries to pay the cost 2m. This means we must add $\frac{2m}{m/2} = 4$ to the payment for each entry, for a total payment of $3 + 4 = 7$.

References

Potential method

In computational complexity theory, the potential method is a method used to analyze the amortized time and space complexity of a data structure, a measure of its performance over sequences of operations that smooths out the cost of infrequent but expensive operations.\cite{1} \cite{2}

Definition of amortized time

In the potential method, a function $\Phi$ is chosen that maps states of the data structure to non-negative numbers. If $S$ is a state of the data structure, $\Phi(S)$ may be thought of intuitively as an amount of potential energy stored in that state;\cite{1} \cite{2} alternatively, $\Phi(S)$ may be thought of as representing the amount of disorder in state $S$ or its distance from an ideal state. The potential value prior to the operation of initializing a data structure is defined to be zero.

Let $o$ be any individual operation within a sequence of operations on some data structure, with $S_{\text{before}}$ denoting the state of the data structure prior to operation $o$ and $S_{\text{after}}$ denoting its state after operation $o$ has completed. Then, once $\Phi$ has been chosen, the amortized time for operation $o$ is defined to be

$$T_{\text{amortized}}(o) = T_{\text{actual}}(o) + C \cdot (\Phi(S_{\text{after}}) - \Phi(S_{\text{before}})),$$

where $C$ is a non-negative constant of proportionality (in units of time) that must remain fixed throughout the analysis. That is, the amortized time is defined to be the actual time taken by the operation plus $C$ times the difference in potential caused by the operation.\cite{1} \cite{2}

Relation between amortized and actual time

Despite its artificial appearance, the total amortized time of a sequence of operations provides a valid upper bound on the actual time for the same sequence of operations. That is, for any sequence of operations $o_0, o_1, \ldots$, the total amortized time $\sum_i T_{\text{amortized}}(o_i)$ is always at least as large as the total actual time $\sum_i T_{\text{actual}}(o_i)$. In more detail,

$$\sum_i T_{\text{amortized}}(o_i) = \sum_i (T_{\text{actual}}(o_i) + C \cdot (\Phi(S_i) - \Phi(S_{i+1}))) = \left(\sum_i T_{\text{actual}}(o_i)\right) + C \cdot (\Phi(S_{\text{final}}) - \Phi(S_{\text{initial}})) \geq \sum_i T_{\text{actual}}(o_i),$$

where the sequence of potential function values forms a telescoping series in which all terms other than the initial and final potential function values cancel in pairs, and where the final inequality arises from the assumptions that $\Phi(S_{\text{final}}) \geq 0$ and $\Phi(S_{\text{initial}}) = 0$. Therefore, amortized time can be used to provide accurate predictions about the actual time of sequences of operations, even though the amortized time for an individual operation may vary widely from its actual time.

Amortized analysis of worst-case inputs

Typically, amortized analysis is used in combination with a worst case assumption about the input sequence. With this assumption, if $X$ is a type of operation that may be performed by the data structure, and $n$ is an integer defining the size of the given data structure (for instance, the number of items that it contains), then the amortized time for operations of type $X$ is defined to be the maximum, among all possible sequences of operations on data structures of size $n$ and all operations $o_i$ of type $X$ within the sequence, of the amortized time for operation $o_i$.

With this definition, the time to perform a sequence of operations may be estimated by multiplying the amortized time for each type of operation in the sequence by the number of operations of that type.
Example
A dynamic array is a data structure for maintaining an array of items, allowing both random access to positions within the array and the ability to increase the array size by one. It is available in Java as the "ArrayList" type and in Python as the "list" type. A dynamic array may be implemented by a data structure consisting of an array $A$ of items, of some length $N$, together with a number $n \leq N$ representing the positions within the array that have been used so far. With this structure, random accesses to the dynamic array may be implemented by accessing the same cell of the internal array $A$, and when $n < N$ an operation that increases the dynamic array size may be implemented simply by incrementing $n$. However, when $n = N$, it is necessary to resize $A$, and a common strategy for doing so is to double its size, replacing $A$ by a new array of length $2n$.\[3\]

This structure may be analyzed using a potential function $\Phi = 2n - N$. Since the resizing strategy always causes $A$ to be at least half-full, this potential function is always non-negative, as desired. When an increase-size operation does not lead to a resize operation, $\Phi$ increases by 2, a constant. Therefore, the constant actual time of the operation and the constant increase in potential combine to give a constant amortized time for an operation of this type. However, when an increase-size operation causes a resize, the potential value of $n$ prior to the resize decreases to zero after the resize. Allocating a new internal array $A$ and copying all of the values from the old internal array to the new one takes $O(n)$ actual time, but (with an appropriate choice of the constant of proportionality $C$) this is entirely cancelled by the decrease of $n$ in the potential function, leaving again a constant total amortized time for the operation. The other operations of the data structure (reading and writing array cells without changing the array size) do not cause the potential function to change and have the same constant amortized time as their actual time.\[2\]

Therefore, with this choice of resizing strategy and potential function, the potential method shows that all dynamic array operations take constant amortized time. Combining this with the inequality relating amortized time and actual time over sequences of operations, this shows that any sequence of $n$ dynamic array operations takes $O(n)$ actual time in the worst case, despite the fact that some of the individual operations may themselves take a linear amount of time.\[2\]

Applications
The potential function method is commonly used to analyze Fibonacci heaps, a form of priority queue in which removing an item takes logarithmic amortized time, and all other operations take constant amortized time.\[4\] It may also be used to analyze splay trees, a self-adjusting form of binary search tree with logarithmic amortized time per operation.\[5\]

References
Arrays

Array data type

In computer science, an array type is a data type that is meant to describe a collection of elements (values or variables), each selected by one or more indices that can be computed at run time by the program. Such a collection is usually called an array variable, array value, or simply array. By analogy with the mathematical concepts of vector and matrix, an array type with one or two indices is often called a vector type or matrix type, respectively.

Language support for array types may include certain built-in array data types, some syntactic constructions (array type constructors) that the programmer may use to define such types and declare array variables, and special notation for indexing array elements. For example, in the Pascal programming language, the declaration type MyTable: array [1..4,1..2] of integer, defines a new array data type called MyTable. The declaration var A: MyTable then defines a variable A of that type, which is an aggregate of eight elements, each being an integer variable identified by two indices. In the Pascal program, those elements are denoted A[1,1], A[1,2], A[2,1], ..., A[4,2]. Special array types are often defined by the language's standard libraries.

Array types are distinguished from record types mainly because they allow the element indices to be computed at run time, as in the Pascal assignment A[I,J] := A[N-I,2*J]. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array variable.

In more theoretical contexts, especially in type theory and in the description of abstract algorithms, the terms "array" and "array type" sometimes refer to an abstract data type (ADT) also called abstract array or may refer to an associative array, a mathematical model with the basic operations and behavior of a typical array type in most languages — basically, a collection of elements that are selected by indices computed at run-time.

Depending on the language, array types may overlap (or be identified with) other data types that describe aggregates of values, such as lists and strings. Array types are often implemented by array data structures, but sometimes by other means, such as hash tables, linked lists, or search trees.

History

Assembly languages and low-level languages like BCPL generally have no syntactic support for arrays.

Because of the importance of array structures for efficient computation, the earliest high-level programming languages, including FORTRAN (1957), COBOL (1960), and Algol 60 (1960), provided support for multi-dimensional arrays.

Abstract arrays

An array data structure can be mathematically modeled as an abstract data structure (an abstract array) with two operations

get(A, I): the data stored in the element of the array A whose indices are the integer tuple I.
set(A,I,V): the array that results by setting the value of that element to V.

These operations are required to satisfy the axioms

get(set(A,I, V), I) = V

get(set(A,I, V), J) = get(A, J) if I ≠ J

for any array state A, any value V, and any tuples I, J for which the operations are defined.
The first axiom means that each element behaves like a variable. The second axiom means that elements with distinct indices behave as disjoint variables, so that storing a value in one element does not affect the value of any other element.

These axioms do not place any constraints on the set of valid index tuples $I$, therefore this abstract model can be used for triangular matrices and other oddly-shaped arrays.

**Implementations**

In order to effectively implement variables of such types as array structures (with indexing done by pointer arithmetic), many languages restrict the indices to integer data types (or other types that can be interpreted as integers, such as bytes and enumerated types), and require that all elements have the same data type and storage size. Most of those languages also restrict each index to a finite interval of integers, that remains fixed throughout the lifetime of the array variable. In some compiled languages, in fact, the index ranges may have to be known at compile time.

On the other hand, some programming languages provide more liberal array types, that allow indexing by arbitrary values, such as floating-point numbers, strings, objects, references, etc.. Such index values cannot be restricted to an interval, much less a fixed interval. So, these languages usually allow arbitrary new elements to be created at any time. This choice precludes the implementation of array types as array data structures. That is, those languages use array-like syntax to implement a more general associative array semantics, and must therefore be implemented by a hash table or some other search data structure.

**Language support**

**Multi-dimensional arrays**

The number of indices needed to specify an element is called the *dimension*, *dimensionality*, or rank of the array type. (This nomenclature conflicts with the concept of dimension in linear algebra, where it is the number of elements. Thus, an array of numbers with 5 rows and 4 columns (hence 20 elements) is said to have dimension 2 in computing contexts, but 20 in mathematics. Also, the computer science meaning of "rank" is similar to its meaning in tensor algebra but not to the linear algebra concept of rank of a matrix.)

Many languages support only one-dimensional arrays. In those languages, a multi-dimensional array is typically represented by an iliffe vector, a one-dimensional array of references to arrays of one dimension less. A two-dimensional array, in particular, would be implemented as a vector of pointers to its rows. Thus an element in row $i$ and column $j$ of an array $A$ would be accessed by double indexing ($A[i][j]$ in typical notation). This way of emulating multi-dimensional arrays allows the creation of *ragged* or *jagged* arrays, where each row may have a different size — or, in general, where the valid range of each index depends on the values of all preceding indices.

This representation for multi-dimensional arrays is quite prevalent in C and C++ software. However, C and C++ will use a linear indexing formula for multi-dimensional arrays that are declared as such, e.g. by `int A[10][20]` or `int A[m][n]`, instead of the traditional `int * * A`.\[5\]:p.81
Indexing notation

Most programming languages that support arrays support the store and select operations, and have special syntax for indexing. Early languages used parentheses, e.g. A(i, j), as in FORTRAN; others choose square brackets, e.g. A[i, j] or A[i][j], as in Algol 60 and Pascal.

Index types

Array data types are most often implemented as array structures: with the indices restricted to integer (or totally ordered) values, index ranges fixed at array creation time, and multilinear element addressing. This was the case in most "third generation" languages, and is still the case of most systems programming languages such as Ada, C, and C++. In some languages, however, array data types have the semantics of associative arrays, with indices of arbitrary type and dynamic element creation. This is the case in some scripting languages such as Awk and Lua, and of some array types provided by standard C++ libraries.

Bounds checking

Some languages (like Pascal and Modula) perform bounds checking on every access, raising an exception or aborting the program when any index is out of its valid range. Compilers may allow these checks to be turned off to trade safety for speed. Other languages (like FORTRAN and C) trust the programmer and perform no checks. Good compilers may also analyze the program to determine the range of possible values that the index may have, and this analysis may lead to bounds-checking elimination.

Index origin

Some languages, such as C, provide only zero-based array types, for which the minimum valid value for any index is 0. This choice is convenient for array implementation and address computations. With a language such as C, a pointer to the interior of any array can be defined that will symbolically act as a pseudo-array that accommodates negative indices. This works only because C does not check an index against bounds when used.

Other languages provide only one-based array types, where each index starts at 1; this is the traditional convention in mathematics for matrices and mathematical sequences. A few languages, such as Pascal, support n-based array types, whose minimum legal indices are chosen by the programmer. The relative merits of each choice have been the subject of heated debate. Zero-based indexing has a natural advantage to one-based indexing in avoiding off-by-one or fencepost errors.[6]

See comparison of programming languages (array) for the base indices used by various languages.

The 0-based/1-based debate is not limited to just programming languages. For example, the elevator button for the ground-floor of a building is labeled "0" in France and many other countries, but "1" in the USA.

Highest index

The relation between numbers appearing in an array declaration and the index of that array's last element also varies by language. In many languages (such as C), languages one should specify the number of elements contained in the array; whereas in others (such as Pascal and Visual Basic .NET) one should specify the numeric value of the index of the last element. Needless to say, this distinction is immaterial in languages where the indices start at 1.

Array algebra

Some programming languages (including APL, Matlab, and newer versions of Fortran) directly support array programming, where operations and functions defined for certain data types are implicitly extended to arrays of elements of those types. Thus one can write A+B to add corresponding elements of two arrays A and B. The multiplication operation may be merely distributed over corresponding elements of the operands (APL) or may be
interpreted as the matrix product of linear algebra (Matlab).

**String types and arrays**

Many languages provide a built-in string data type, with specialized notation ("string literals") to build values of that type. In some languages (such as C), a string is just an array of characters, or is handled in much the same way. Other languages, like Pascal, may provide vastly different operations for strings and arrays.

**Array index range queries**

Some programming languages provide operations that return the size (number of elements) of a vector, or, more generally, range of each index of an array. In C and C++ arrays do not support the size function, so programmers often have to declare separate variable to hold the size, and pass it to procedures as a separate parameter.

Elements of a newly created array may have undefined values (as in C), or may be defined to have a specific "default" value such as 0 or a null pointer (as in Java).

In C++ a std::vector object supports the store, select, and append operations with the performance characteristics discussed above. Vectors can be queried for their size and can be resized. Slower operations like inserting an element in the middle are also supported.

**Slicing**

An array slicing operation takes a subset of the elements of an array-typed entity (value or variable) and then assembles them as another array-typed entity, possibly with other indices. If array types are implemented as array structures, many useful slicing operations (such as selecting a sub-array, swapping indices, or reversing the direction of the indices) can be performed very efficiently by manipulating the dope vector of the structure. The possible slicings depend on the implementation details: for example, FORTRAN allows slicing off one column of a matrix variable, but not a row, and treat it as a vector; whereas C allow slicing off a row from a matrix, but not a column.

On the other hand, other slicing operations are possible when array types are implemented in other ways.

**Resizing**

Some languages allow dynamic arrays (also called resizable, growable, or extensible): array variables whose index ranges may be expanded at any time after creation, without changing the values of its current elements.

For one-dimensional arrays, this facility may be provided as an operation "append(A,x)" that increases the size of the array A by one and then sets the value of the last element to x. Other array types (such as Pascal strings) provide a concatenation operator, which can be used together with slicing to achieve that effect and more. In some languages, assigning a value to an element of an array automatically extends the array, if necessary, to include that element. In other array types, a slice can be replaced by an array of different size" with subsequent elements being renumbered accordingly — as in Python's list assignment "A[5:5] = [10,20,30]", that inserts three new elements (10,20, and 30) before element "A[5]". Resizable arrays are conceptually similar to lists, and the two concepts are synonymous in some languages.

An exensible array can be implemented as a fixed-size array, with a counter that records how many elements are actually in use. The append operation merely increments the counter; until the whole array is used, when the append operation may be defined to fail. This is an implementation of a dynamic array with a fixed capacity, as in the string type of Pascal. Alternatively, the append operation may re-allocate the underlying array with a larger size, and copy the old elements to the new area.
Array data structure

In computer science, an array data structure or simply array is a data structure consisting of a collection of elements (values or variables), each identified by at least one index. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula.\[1\] [2] [3]

For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, … 2036, so that the element with index \(i\) has the address 2000 + 4 \(\times i\).\[4\]

Arrays are analogous to the mathematical concepts of the vector, the matrix, and the tensor. Indeed, arrays with one or two indices are often called vectors or matrices, respectively. Arrays are often used to implement tables, especially lookup tables; the word table is sometimes used as a synonym of array.

Arrays are among the oldest and most important data structures, and are used by almost every program and are used to implement many other data structures, such as lists and strings. They effectively exploit the addressing logic of computers. In most modern computers and many external storage devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially vector processors, are often optimized for array operations.

Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually,\[3\] [5] but not always,\[2\] fixed while the array is in use.

The term array is often used to mean array data type, a kind of data type provided by most high-level programming languages that consists of a collection of values or variables that can be selected by one or more indices computed at run-time. Array types are often implemented by array structures; however, in some languages they may be implemented by hash tables, linked lists, search trees, or other data structures.

The term is also used, especially in the description of algorithms, to mean associative array or "abstract array", a theoretical computer science model (an abstract data type or ADT) intended to capture the essential properties of arrays.

References

[4] Lukham, Suzuki (1979), "Verification of array, record, and pointer operations in Pascal". ACM Transactions on Programming Languages and Systems 1(2), 226–244.

External links

Array data structure

History
The first digital computers used machine-language programming to set up and access array structures for data tables, vector and matrix computations, and for many other purposes. Von Neumann wrote the first array-sorting program (merge sort) in 1945, during the building of the first stored-program computer. Array indexing was originally done by self-modifying code, and later using index registers and indirect addressing. Some mainframes designed in the 1960s, such as the Burroughs B5000 and its successors, had special instructions for array indexing that included index-bounds checking.

Assembly languages generally have no special support for arrays, other than what the machine itself provides. The earliest high-level programming languages, including FORTRAN (1957), COBOL (1960), and ALGOL 60 (1960), had support for multi-dimensional arrays, and so has C (1972). In C++ (1983), class templates exist for multi-dimensional arrays whose dimension is fixed at runtime as well as for runtime-flexible arrays.

Applications
Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.

Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists.

One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably.

Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive), multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array. The array may contain subroutine pointers (or relative subroutine numbers that can be acted upon by SWITCH statements) that direct the path of the execution.

Addressing formulas
The number of indices needed to specify an element is called the dimension, dimensionality, or rank of the array.

In standard arrays, each index is restricted to a certain range of consecutive integers (or consecutive values of some enumerated type), and the address of an element is computed by a "linear" formula on the indices.

One-dimensional arrays
A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

As an example consider the C declaration auto int new[10];

In the given example the array starts with auto storage class and is of integer type named new which can contain 10 elements in it i.e. 0-9. It is not necessary to declare the storage class as the compiler initializes auto storage class by default to every data type After that the data type is declared which is followed by the name i.e. new which can contain 10 entities.

For a vector with linear addressing, the element with index i is located at the address B + c · i, where B is a fixed base address and c a fixed constant, sometimes called the address increment or stride.

If the valid element indices begin at 0, the constant B is simply the address of the first element of the array. For this reason, the C programming language specifies that array indices always begin at 0; and many programmers will call that element "zeroth" rather than "first".
However, one can choose the index of the first element by an appropriate choice of the base address $B$. For example, if the array has five elements, indexed 1 through 5, and the base address $B$ is replaced by $B = 30c$, then the indices of those same elements will be 31 to 35. If the numbering does not start at 0, the constant $B$ may not be the address of any element.

**Two-dimensional arrays**

For a two-dimensional array, the element with indices $i, j$ would have address $B + c \cdot i + d \cdot j$, where the coefficients $c$ and $d$ are the row and column address increments, respectively.

More generally, in a $k$-dimensional array, the address of an element with indices $i_1, i_2, \ldots, i_k$ is

$$B + c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + c_k \cdot i_k.$$  

This formula requires only $k$ multiplications and $k-1$ additions, for any array that can fit in memory. Moreover, if any coefficient is a fixed power of 2, the multiplication can be replaced by bit shifting.

The coefficients $c_k$ must be chosen so that every valid index tuple maps to the address of a distinct element.

If the minimum legal value for every index is 0, then $B$ is the address of the element whose indices are all zero. As in the one-dimensional case, the element indices may be changed by changing the base address $B$. Thus, if a two-dimensional array has rows and columns indexed from 1 to 10 and 1 to 20, respectively, then replacing $B$ by $B + c_1 - 3c_1$ will cause them to be renumbered from 0 through 9 and 4 through 23, respectively. Taking advantage of this feature, some languages (like FORTRAN 77) specify that array indices begin at 1, as in mathematical tradition; while other languages (like Fortran 90, Pascal and Algol) let the user choose the minimum value for each index.

**Dope vectors**

The addressing formula is completely defined by the dimension $d$, the base address $B$, and the increments $c_1, c_2, \ldots, c_k$. It is often useful to pack these parameters into a record called the array's descriptor or stride vector or dope vector. The size of each element, and the minimum and maximum values allowed for each index may also be included in the dope vector. The dope vector is a complete handle for the array, and is a convenient way to pass arrays as arguments to procedures. Many useful array slicing operations (such as selecting a sub-array, swapping indices, or reversing the direction of the indices) can be performed very efficiently by manipulating the dope vector.

**Compact layouts**

Often the coefficients are chosen so that the elements occupy a contiguous area of memory. However, that is not necessary. Even if arrays are always created with contiguous elements, some array slicing operations may create non-contiguous sub-arrays from them.

There are two systematic compact layouts for a two-dimensional array. For example, consider the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}. $$

In the row-major order layout (adopted by C for statically declared arrays), the elements in each row are stored in consecutive positions and all of the elements of a row have a lower address than any of the elements of a consecutive row:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} $$

In Column-major order (traditionally used by Fortran), the elements in each column are consecutive in memory and all of the elements of a columns have a lower address than any of the elements of a consecutive column:
For arrays with three or more indices, "row major order" puts in consecutive positions any two elements whose index tuples differ only by one in the last index. "Column major order" is analogous with respect to the first index.

In systems which use processor cache or virtual memory, scanning an array is much faster if successive elements are stored in consecutive positions in memory, rather than sparsely scattered. Many algorithms that use multidimensional arrays will scan them in a predictable order. A programmer (or a sophisticated compiler) may use this information to choose between row- or column-major layout for each array. For example, when computing the product $A \cdot B$ of two matrices, it would be best to have $A$ stored in row-major order, and $B$ in column-major order.

**Array resizing**

Static arrays have a size that is fixed at allocation time and consequently do not allow elements to be inserted or removed. However, by allocating a new array and copying the contents of the old array to it, it is possible to effectively implement a *dynamic* version of an array; see dynamic array. If this operation is done infrequently, insertions at the end of the array require only amortized constant time.

Some array data structures do not reallocate storage, but do store a count of the number of elements of the array in use, called the count or size. This effectively makes the array a dynamic array with a fixed maximum size or capacity; *Pascal strings* are examples of this.

**Non-linear formulas**

More complicated (non-linear) formulas are occasionally used. For a compact two-dimensional triangular array, for instance, the addressing formula is a polynomial of degree 2.

**Efficiency**

Both *store* and *select* take (deterministic worst case) constant time. Arrays take linear ($O(n)$) space in the number of elements $n$ that they hold.

In an array with element size $k$ and on a machine with a cache line size of $B$ bytes, iterating through an array of $n$ elements requires the minimum of $\lceil nk/B \rceil$ cache misses, because its elements occupy contiguous memory locations. This is roughly a factor of $B/k$ better than the number of cache misses needed to access $n$ elements at random memory locations. As a consequence, sequential iteration over an array is noticeably faster in practice than iteration over many other data structures, a property called locality of reference (this does *not* mean however, that using a perfect hash or trivial hash within the same (local) array, will not be even faster - and achievable in constant time). Libraries provide low-level optimized facilities for copying ranges of memory (such as memcpy) which can be used to move contiguous blocks of array elements significantly faster than can be achieved through individual element access. The speedup of such optimized routines varies by array element size, architecture, and implementation.

Memory-wise, arrays are compact data structures with no per-element overhead. There may be a per-array overhead, e.g. to store index bounds, but this is language-dependent. It can also happen that elements stored in an array require *less* memory than the same elements stored in individual variables, because several array elements can be stored in a single word; such arrays are often called *packed* arrays. An extreme (but commonly used) case is the bit array, where every bit represents a single element. A single octet can thus hold up to 256 different combinations of up to 8 different conditions, in the most compact form.

Array accesses with statically predictable access patterns are a major source of data parallelism.
Array data structure

### Efficiency comparison with other data structures

<table>
<thead>
<tr>
<th></th>
<th>Linked list</th>
<th>Array</th>
<th>Dynamic array</th>
<th>Balanced tree</th>
<th>Random access list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Indexing</strong></td>
<td>Θ(n)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(log n)</td>
<td>Θ(log n)</td>
</tr>
<tr>
<td><strong>Insert/delete at beginning</strong></td>
<td>Θ(1)</td>
<td>N/A</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td><strong>Insert/delete at end</strong></td>
<td>Θ(1)</td>
<td>N/A</td>
<td>Θ(1) amortized</td>
<td>Θ(log n)</td>
<td>Θ(log n) updating</td>
</tr>
<tr>
<td><strong>Insert/delete in middle</strong></td>
<td>Θ(1)</td>
<td>N/A</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>Θ(log n) updating</td>
</tr>
<tr>
<td><strong>Wasted space (average)</strong></td>
<td>Θ(n)</td>
<td>0</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>

Growable arrays are similar to arrays but add the ability to insert and delete elements; adding and deleting at the end is particularly efficient. However, they reserve linear (Θ(n)) additional storage, whereas arrays do not reserve additional storage.

Associative arrays provide a mechanism for array-like functionality without huge storage overheads when the index values are sparse. For example, an array that contains values only at indexes 1 and 2 billion may benefit from using such a structure. Specialized associative arrays with integer keys include Patricia tries, Judy arrays, and van Emde Boas trees.

Balanced trees require $O(\log n)$ time for indexed access, but also permit inserting or deleting elements in $O(\log n)$ time,[9] whereas growable arrays require linear (Θ(n)) time to insert or delete elements at an arbitrary position.

Linked lists allow constant time removal and insertion in the middle but take linear time for indexed access. Their memory use is typically worse than arrays, but is still linear.

An Iliffe vector is an alternative to a multidimensional array structure. It uses a one-dimensional array of references to arrays of one dimension less. For two dimensions, in particular, this alternative structure would be a vector of pointers to vectors, one for each row. Thus an element in row $i$ and column $j$ of an array $A$ would be accessed by double indexing ($A[i][j]$ in typical notation). This alternative structure allows ragged or jagged arrays, where each row may have a different size — or, in general, where the valid range of each index depends on the values of all preceding indices. It also saves one multiplication (by the column address increment) replacing it by a bit shift (to index the vector of row pointers) and one extra memory access (fetching the row address), which may be worthwhile in some architectures.

### Meaning of dimension

The dimension of an array is the number of indices needed to select an element. Thus, if the array is seen as a function on a set of possible index combinations, it is the dimension of the space of which its domain is a discrete subset. Thus a one-dimensional array is a list of data, a two-dimensional array a rectangle of data, a three-dimensional array a block of data, etc.

This should not be confused with the dimension of the set of all matrices with a given domain, that is, the number of elements in the array. For example, an array with 5 rows and 4 columns is two-dimensional, but such matrices form a 20-dimensional space. Similarly, a three-dimensional vector can be represented by a one-dimensional array of size three.
## Dynamic array

In computer science, a **dynamic array**, **growable array**, **resizable array**, **dynamic table**, or **array list** is a random access, variable-size list data structure that allows elements to be added or removed. It is supplied with standard libraries in many modern mainstream programming languages.

A dynamic array is not the same thing as a dynamically-allocated array, which is a fixed-size array whose size is fixed when the array is allocated, although a dynamic array may use such a fixed-size array as a back end.[1]

### Bounded-size dynamarrays and capacity

The simplest dynamic array is constructed by allocating a fixed-size array and then dividing it into two parts: the first stores the elements of the dynamic array and the second is reserved, or unused. We can then add or remove elements at the end of the dynamic array in constant time by using the reserved space, until this space is completely consumed. The number of elements used by the dynamic array contents is its **logical size** or **size**, while the size of the underlying array is called the dynamic array's **capacity**, which is the maximum possible size without relocating data.

In applications where the logical size is bounded, the fixed-size data structure suffices. This may be short-sighted, when problems with the array filling up turn up later. It is best to put resize code into any array, to respond to new conditions. Then choosing initial capacity is optimization, not getting the program to run. Resizing the underlying array is an expensive task, typically involving copying the entire contents of the array.
Geometric expansion and amortized cost

To avoid incurring the cost of resizing many times, dynamic arrays resize by a large amount, such as doubling in size, and use the reserved space for future expansion. The operation of adding an element to the end might work as follows:

```c
function insertEnd(dynarray a, element e)
    if (a.size = a.capacity)
        // resize a to twice its current capacity:
        a.capacity ← a.capacity * 2
        // (copy the contents to the new memory location here)
        a[a.size] ← e
        a.size ← a.size + 1
```

As \(n\) elements are inserted, the capacities form a geometric progression. Expanding the array by any constant proportion ensures that inserting \(n\) elements takes \(O(n)\) time overall, meaning that each insertion takes amortized constant time. The value of this proportion \(a\) leads to a time-space tradeoff: the average time per insertion operation is about \(a/(a-1)\), while the number of wasted cells is bounded above by \((a-1)n\). The choice of \(a\) depends on the library or application: some textbooks use \(a = 2\) \[2\] \[3\] but Java's ArrayList implementation uses \(a = 3/2\) \[1\] and the C implementation of Python's list data structure uses \(a = 9/8\). \[4\]

Many dynamic arrays also deallocate some of the underlying storage if its size drops below a certain threshold, such as 30% of the capacity. This threshold must be strictly smaller than \(1/a\) in order to support mixed sequences of insertions and removals with amortized constant cost.

Dynamic arrays are a common example when teaching amortized analysis. \[2\] \[3\]

Performance

<table>
<thead>
<tr>
<th>Service</th>
<th>Linked list</th>
<th>Array</th>
<th>Dynamic array</th>
<th>Balanced tree</th>
<th>Random access list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td>Insert/delete at beginning</td>
<td>(\Theta(1))</td>
<td>N/A</td>
<td>(\Theta(n))</td>
<td>(\Theta(log n))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>Insert/delete at end</td>
<td>(\Theta(1))</td>
<td>N/A</td>
<td>(\Theta(1)) amortized</td>
<td>(\Theta(log n))</td>
<td>(\Theta(log n)) updating</td>
</tr>
<tr>
<td>Insert/delete in middle</td>
<td>search time (\Theta(1)) [5]</td>
<td>N/A</td>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
<td>(\Theta(log n)) updating</td>
</tr>
<tr>
<td>Wasted space (average)</td>
<td>(\Theta(n))</td>
<td>0</td>
<td>(\Theta(n)) [6]</td>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
</tr>
</tbody>
</table>

The dynamic array has performance similar to an array, with the addition of new operations to add and remove elements from the end:

- Getting or setting the value at a particular index (constant time)
- Iterating over the elements in order (linear time, good cache performance)
- Inserting or deleting an element in the middle of the array (linear time)
- Inserting or deleting an element at the end of the array (constant amortized time)

Dynamic arrays benefit from many of the advantages of arrays, including good locality of reference and data cache utilization, compactness (low memory use), and random access. They usually have only a small fixed additional overhead for storing information about the size and capacity. This makes dynamic arrays an attractive tool for building cache-friendly data structures.
Compared to linked lists, dynamic arrays have faster indexing (constant time versus linear time) and typically faster iteration due to improved locality of reference; however, dynamic arrays require linear time to insert or delete at an arbitrary location, since all following elements must be moved, while linked lists can do this in constant time. This disadvantage is mitigated by the gap buffer and tiered vector variants discussed under Variants below. Also, in a highly-fragmented memory region, it may be expensive or impossible to find contiguous space for a large dynamic array, whereas linked lists do not require the whole data structure to be stored contiguously.

A balanced tree can store a list while providing all operations of both dynamic arrays and linked lists reasonably efficiently, but both insertion at the end and iteration over the list are slower than for a dynamic array, in theory and in practice, due to non-contiguous storage and tree traversal/manipulation overhead.

### Variants

Gap buffers are similar to dynamic arrays but allow efficient insertion and deletion operations clustered near the same arbitrary location. Some deque implementations use array deques, which allow amortized constant time insertion/removal at both ends, instead of just one end.

Goodrich\(^7\) presented a dynamic array algorithm called Tiered Vectors that provided $O(n^{1/2})$ performance for order preserving insertions or deletions from the middle of the array.

Hashed Array Tree (HAT) is a dynamic array algorithm published by Sitarski in 1996.\(^8\) Hashed Array Tree wastes order $n^{1/2}$ amount of storage space, where $n$ is the number of elements in the array. The algorithm has $O(1)$ amortized performance when appending a series of objects to the end of a Hashed Array Tree.

In a 1999 paper,\(^6\) Brodnik et al. describe a tiered dynamic array data structure, which wastes only $n^{1/2}$ space for $n$ elements at any point in time, and they prove a lower bound showing that any dynamic array must waste this much space if the operations are to remain amortized constant time. Additionally, they present a variant where growing and shrinking the buffer has not only amortized but worst-case constant time.

Bagwell (2002)\(^9\) presented the VList algorithm, which can be adapted to implement a dynamic array.

### Language support

C++'s `std::vector` is an implementation of dynamic arrays, as are the `ArrayList`\(^10\) classes supplied with the Java API and the .NET Framework. The generic `List<>` class supplied with version 2.0 of the .NET Framework is also implemented with dynamic arrays. Python's `list` datatype implementation is a dynamic array. Delphi and D implement dynamic arrays at the language's core. Many scripting languages such as Perl and PHP offer dynamic arrays as a built-in primitive data type.

### References

1. See, for example, the source code of java.util.ArrayList class from OpenJDK 6 (http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/ e0e25ac28560/src/share/classes/java/util/ArrayList.java).
6. Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (Technical Report CS-99-09), *Resizable Arrays in Optimal Time and Space* (http://www.cs.uwaterloo.ca/research/tr/1999/09/CS-99-09.pdf), Department of Computer Science, University of Waterloo,
In computer science, a **linked list** is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a datum and a reference (in other words, a link) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.

A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data structures, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require scanning most or all of the list elements.

**History**

Linked lists were developed in 1955-56 by Allen Newell, Cliff Shaw and Herbert Simon at RAND Corporation as the primary data structure for their Information Processing Language. IPL was used by the authors to develop several early artificial intelligence programs, including the Logic Theory Machine, the General Problem Solver, and a computer chess program. Reports on their work appeared in IRE Transactions on Information Theory in 1956, and several conference proceedings from 1957 to 1959, including Proceedings of the Western Joint Computer Conference in 1957 and 1958, and Information Processing (Proceedings of the first UNESCO International Conference on Information Processing) in 1959. The now-classic diagram consisting of blocks representing list
nodes with arrows pointing to successive list nodes appears in "Programming the Logic Theory Machine" by Newell and Shaw in Proc. WJCC, February 1957. Newell and Simon were recognized with the ACM Turing Award in 1975 for having "made basic contributions to artificial intelligence, the psychology of human cognition, and list processing". The problem of machine translation for natural language processing led Victor Yngve at Massachusetts Institute of Technology (MIT) to use linked lists as data structures in his COMIT programming language for computer research in the field of linguistics. A report on this language entitled "A programming language for mechanical translation" appeared in Mechanical Translation in 1958.

LISP, standing for list processor, was created by John McCarthy in 1958 while he was at MIT and in 1960 he published its design in a paper in the Communications of the ACM, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". One of LISP's major data structures is the linked list. By the early 1960s, the utility of both linked lists and languages which use these structures as their primary data representation was well established. Bert Green of the MIT Lincoln Laboratory published a review article entitled "Computer languages for symbol manipulation" in IRE Transactions on Human Factors in Electronics in March 1961 which summarized the advantages of the linked list approach. A later review article, "A Comparison of list-processing computer languages" by Bobrow and Raphael, appeared in Communications of the ACM in April 1964.

Several operating systems developed by Technical Systems Consultants (originally of West Lafayette Indiana, and later of Chapel Hill, North Carolina) used singly linked lists as file structures. A directory entry pointed to the first sector of a file, and succeeding portions of the file were located by traversing pointers. Systems using this technique included Flex (for the Motorola 6800 CPU), mini-Flex (same CPU), and Flex9 (for the Motorola 6809 CPU). A variant developed by TSC for and marketed by Smoke Signal Broadcasting in California, used doubly linked lists in the same manner.

The TSS/360 operating system, developed by IBM for the System 360/370 machines, used a double linked list for their file system catalog. The directory structure was similar to Unix, where a directory could contain files and/or other directories and extend to any depth. A utility flea was created to fix file system problems after a crash, since modified portions of the file catalog were sometimes in memory when a crash occurred. Problems were detected by comparing the forward and backward links for consistency. If a forward link was corrupt, then if a backward link to the infected node was found, the forward link was set to the node with the backward link. A humorous comment in the source code where this utility was invoked stated "Everyone knows a flea collar gets rid of bugs in cats".

Basic concepts and nomenclature

Each record of a linked list is often called an element or node.

The field of each node that contains the address of the next node is usually called the next link or next pointer. The remaining fields are known as the data, information, value, cargo, or payload fields.

The head of a list is its first node. The tail of a list may refer either to the rest of the list after the head, or to the last node in the list. In Lisp and some derived languages, the next node may be called the cdr (pronounced could-er) of the list, while the payload of the head node may be called the car.
**Linked list**

Post office box analogy

The concept of a linked list can be explained by a simple analogy to real-world post office boxes. Suppose Alice is a spy who wishes to give a codebook to Bob by putting it in a post office box and then giving him the key. However, the book is too thick to fit in a single post office box, so instead she divides the book into two halves and purchases two post office boxes. In the first box, she puts the first half of the book and a key to the second box, and in the second box she puts the second half of the book. She then gives Bob a key to the first box. No matter how large the book is, this scheme can be extended to any number of boxes by always putting the key to the next box in the previous box.

In this analogy, the boxes correspond to elements or nodes, the keys correspond to pointers, and the book itself is the data. The key given to Bob is the head pointer, while those stored in the boxes are next pointers. The scheme as described above is a singly linked list (see below).

Linear and circular lists

In the last node of a list, the link field often contains a null reference, a special value used to indicate the lack of further nodes. A less common convention is to make it point to the first node of the list; in that case the list is said to be circular or circularly linked; otherwise it is said to be open or linear.

A circular linked list

Singly, doubly, and multiply linked lists

Singly linked lists contain nodes which have a data field as well as a next field, which points to the next node in the linked list.

A singly linked list whose nodes contain two fields: an integer value and a link to the next node

In a doubly linked list, each node contains, besides the next-node link, a second link field pointing to the previous node in the sequence. The two links may be called forward(s) and backwards, or next and previous.

A doubly linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

A technique known as XOR-linking allows a doubly linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.
In a **multiply linked list**, each node contains two or more link fields, each field being used to connect the same set of data records in a different order (e.g., by name, by department, by date of birth, etc.). (While doubly linked lists can be seen as special cases of multiply linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.)

In the case of a circular doubly linked list, the only change that occurs is that end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

**Sentinel nodes**

In some implementations, an extra sentinel or dummy node may be added before the first data record and/or after the last one. This convention simplifies and accelerates some list-handling algorithms, by ensuring that all links can be safely dereferenced and that every list (even one that contains no data elements) always has a "first" and "last" node.

**Empty lists**

An empty list is a list that contains no data records. This is usually the same as saying that it has zero nodes. If sentinel nodes are being used, the list is usually said to be empty when it has only sentinel nodes.

**Hash linking**

The link fields need not be physically part of the nodes. If the data records are stored in an array and referenced by their indices, the link field may be stored in a separate array with the same indices as the data records.

**List handles**

Since a reference to the first node gives access to the whole list, that reference is often called the address, pointer, or handle of the list. Algorithms that manipulate linked lists usually get such handles to the input lists and return the handles to the resulting lists. In fact, in the context of such algorithms, the word "list" often means "list handle". In some situations, however, it may be convenient to refer to a list by a handle that consists of two links, pointing to its first and last nodes.

**Combining alternatives**

The alternatives listed above may be arbitrarily combined in almost every way, so one may have circular doubly linked lists without sentinels, circular singly linked lists with sentinels, etc.

**Tradeoffs**

As with most choices in computer programming and design, no method is well suited to all circumstances. A linked list data structure might work well in one case, but cause problems in another. This is a list of some of the common tradeoffs involving linked list structures.

**Linked lists vs. dynamic arrays**
A **dynamic array** is a data structure that allocates all elements contiguously in memory, and keeps a count of the current number of elements. If the space reserved for the dynamic array is exceeded, it is reallocated and (possibly) copied, an expensive operation.

Linked lists have several advantages over dynamic arrays. Insertion or deletion of an element at a specific point of a list, assuming that we have a pointer to the node (before the one to be removed, or before the insertion point) already, is a constant-time operation, whereas insertion in a dynamic array at random locations will require moving half of the elements on average, and all the elements in the worst case. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", this causes fragmentation that impedes the performance of iteration.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while a dynamic array will eventually fill up its underlying array data structure and have to reallocate — an expensive operation (although the cost of the reallocation can be averaged over insertions, and the cost of insertions would still be amortized $O(1)$, the same as for linked lists), one that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may have to be resized in order to avoid wasting too much space.

On the other hand, dynamic arrays (as well as fixed-size array data structures) allow constant-time random access, while linked lists allow only sequential access to elements. Singly linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays and dynamic arrays is also faster than on linked lists on many machines, because they have optimal locality of reference and thus make good use of data caching.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values, because the storage overhead for the links may exceed by a factor of two or more the size of the data. In contrast, a dynamic array requires only the space for the data itself (and a very small amount of control data).\[^3\] It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

Some hybrid solutions try to combine the advantages of the two representations. Unrolled linked lists store several elements in each list node, increasing cache performance while decreasing memory overhead for references. CDR coding does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record.

A good example that highlights the pros and cons of using dynamic arrays vs. linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle $n$ times. Once you reach the $n$th person, take them out of the circle and have the members close the circle. Then count around the circle the same $n$ times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. a dynamic array, because if you view the people as connected nodes in a
circular linked list then it shows how easily the linked list is able to delete nodes (as it only has to rearrange the links to the different nodes). However, the linked list will be poor at finding the next person to remove and will need to search through the list until it finds that person. A dynamic array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the $n$th person in the circle by directly referencing them by their position in the array.

The list ranking problem concerns the efficient conversion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a parallel algorithm is complicated and has been the subject of much research.

A balanced tree has similar memory access patterns and space overhead to a linked list while permitting more efficient indexing. However, insertion and deletion operations are more expensive due to the overhead of tree manipulations to maintain balance.

**Singly linked linear lists vs. other lists**

While doubly linked and/or circular lists have advantages over singly linked linear lists, linear lists offer some advantages that make them preferable in some situations.

For one thing, a singly linked linear list is a recursive data structure, because it contains a pointer to a smaller object of the same type. For that reason, many operations on singly linked linear lists (such as merging two lists, or enumerating the elements in reverse order) often have very simple recursive algorithms, much simpler than any solution using iterative commands. While one can adapt those recursive solutions for doubly linked and circularly linked lists, the procedures generally need extra arguments and more complicated base cases.

Linear singly linked lists also allow tail-sharing, the use of a common final portion of sub-list as the terminal portion of two different lists. In particular, if a new node is added at the beginning of a list, the former list remains available as the tail of the new one — a simple example of a persistent data structure. Again, this is not true with the other variants: a node may never belong to two different circular or doubly linked lists.

In particular, end-sentinel nodes can be shared among singly linked non-circular lists. One may even use the same end-sentinel node for every such list. In Lisp, for example, every proper list ends with a link to a special node, denoted by nil or (), whose CAR and CDR links point to itself. Thus a Lisp procedure can safely take the CAR or CDR of any list.

Indeed, the advantages of the fancy variants are often limited to the complexity of the algorithms, not in their efficiency. A circular list, in particular, can usually be emulated by a linear list together with two variables that point to the first and last nodes, at no extra cost.

**Doubly linked vs. singly linked**

Double-linked lists require more space per node (unless one uses XOR-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In a doubly linked list, one can insert or delete a node in a constant number of operations given only that node’s address. To do the same in a singly linked list, one must have the address of the pointer to that node, which is either the handle for the whole list (in case of the first node) or the link field in the previous node. Some algorithms require access in both directions. On the other hand, doubly linked lists do not allow tail-sharing and cannot be used as persistent data structures.
**Circularly linked vs. linearly linked**

A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. the corners of a polygon, a pool of buffers that are used and released in FIFO order, or a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list.

With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer, instead of two.

A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes in two distinct lists joins the two list into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge.

The simplest representation for an empty circular list (when such a thing makes sense) is a null pointer, indicating that the list has no nodes. With this choice, many algorithms have to test for this special case, and handle it separately. By contrast, the use of null to denote an empty linear list is more natural and often creates fewer special cases.

**Using sentinel nodes**

Sentinel node may simplify certain list operations, by ensuring that the next and/or previous nodes exist for every element, and that even empty lists have at least one node. One may also use a sentinel node at the end of the list, with an appropriate data field, to eliminate some end-of-list tests. For example, when scanning the list looking for a node with a given value $x$, setting the sentinel's data field to $x$ makes it unnecessary to test for end-of-list inside the loop.

Another example is the merging two sorted lists: if their sentinels have data fields set to $+\infty$, the choice of the next output node does not need special handling for empty lists.

However, sentinel nodes use up extra space (especially in applications that use many short lists), and they may complicate other operations (such as the creation of a new empty list).

However, if the circular list is used merely to simulate a linear list, one may avoid some of this complexity by adding a single sentinel node to every list, between the last and the first data nodes. With this convention, an empty list consists of the sentinel node alone, pointing to itself via the next-node link. The list handle should then be a pointer to the last data node, before the sentinel, if the list is not empty; or to the sentinel itself, if the list is empty.

The same trick can be used to simplify the handling of a doubly linked linear list, by turning it into a circular doubly linked list with a single sentinel node. However, in this case, the handle should be a single pointer to the dummy node itself.[4]

**Linked list operations**

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudocode for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use null to refer to an end-of-list marker or sentinel, which may be implemented in a number of ways.

**Linearly linked lists**

**Singly linked lists**

Our node data structure will have two fields. We also keep a variable firstNode which always points to the first node in the list, or is null for an empty list.
Linked list

```
record Node {
    data; // The data being stored in the node
    Node next; // A reference to the next node, null for last node
}

record List {
    Node firstNode; // points to first node of list; null for empty list
}
```

Traversal of a singly linked list is simple, beginning at the first node and following each `next` link until we come to the end:

```plaintext
node := list.firstNode
while node not null
    (do something with node.data)
    node := node.next
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done directly; instead, one must keep track of the previous node and insert a node after it.

```
function insertAfter(Node node, Node newNode) // insert newNode after node
    newNode.next := node.next
    node.next := newNode
```

Inserting at the beginning of the list requires a separate function. This requires updating `firstNode`.

```
function insertBeginning(List list, Node newNode) // insert node before current first node
    newNode.next := list.firstNode
    list.firstNode := newNode
```

Similarly, we have functions for removing the node after a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.
function removeAfter(node node) // remove node past this one
    obsoleteNode := node.next
    node.next := node.next.next
    destroy obsoleteNode

function removeBeginning(List list) // remove first node
    obsoleteNode := list.firstNode
    list.firstNode := list.firstNode.next // point past deleted node
    destroy obsoleteNode

Notice that removeBeginning() sets list.firstNode to null when removing the last node in the list.

Since we can't iterate backwards, efficient insertBefore or removeBefore operations are not possible.

Appending one linked list to another can be inefficient unless a reference to the tail is kept as part of the List structure, because we must traverse the entire first list in order to find the tail, and then append the second list to this. Thus, if two linearly linked lists are each of length $n$, list appending has asymptotic time complexity of $O(n)$. In the Lisp family of languages, list appending is provided by the append procedure.

Many of the special cases of linked list operations can be eliminated by including a dummy element at the front of the list. This ensures that there are no special cases for the beginning of the list and renders both insertBeginning() and removeBeginning() unnecessary. In this case, the first useful data in the list will be found at list.firstNode.next.
Circularly linked list

In a circularly linked list, all nodes are linked in a continuous circle, without using null. For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The next node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Circularly linked lists can be either singly or doubly linked.

Both types of circularly linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing firstNode and lastNode, although if the list may be empty we need a special representation for the empty list, such as a lastNode variable which points to some node in the list or is null if it's empty; we use such a lastNode here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

Algorithms

Assuming that someNode is some node in a non-empty circular singly linked list, this code iterates through that list starting with someNode:

```plaintext
function iterate(someNode)
    if someNode ≠ null
        node := someNode
        do
            do something with node.value
            node := node.next
        while node ≠ someNode
```

Notice that the test "while node ≠ someNode" must be at the end of the loop. If the test was moved to the beginning of the loop, the procedure would fail whenever the list had only one node.

This function inserts a node "newNode" into a circular linked list after a given node "node". If "node" is null, it assumes that the list is empty.

```plaintext
function insertAfter(Node node, Node newNode)
    if node = null
        newNode.next := newNode
    else
        newNode.next := node.next
        node.next := newNode
```

Suppose that "L" is a variable pointing to the last node of a circular linked list (or null if the list is empty). To append "newNode" to the end of the list, one may do

```
insertAfter(L, newNode)
L := newNode
```

To insert "newNode" at the beginning of the list, one may do

```
insertAfter(L, newNode)
if L = null
    L := newNode
```
Linked lists using arrays of nodes

Languages that do not support any type of reference can still create links by replacing pointers with array indices. The approach is to keep an array of records, where each record has integer fields indicating the index of the next (and possibly previous) node in the array. Not all nodes in the array need be used. If records are also not supported, parallel arrays can often be used instead.

As an example, consider the following linked list record that uses arrays instead of pointers:

```
record Entry {
  integer next; // index of next entry in array
  integer prev; // previous entry (if double-linked)
  string name;
  real balance
}
```

By creating an array of these structures, and an integer variable to store the index of the first element, a linked list can be built:

```
integer listHead
Entry Records[1000]
```

Links between elements are formed by placing the array index of the next (or previous) cell into the Next or Prev field within a given element. For example:

<table>
<thead>
<tr>
<th>Index</th>
<th>Next</th>
<th>Prev</th>
<th>Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>Jones, John</td>
<td>123.45</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>Smith, Joseph</td>
<td>234.56</td>
</tr>
<tr>
<td>2 (listHead)</td>
<td>4</td>
<td>-1</td>
<td>Adams, Adam</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>Ignore, Ignatius</td>
<td>999.99</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>Another, Anita</td>
<td>876.54</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the above example, ListHead would be set to 2, the location of the first entry in the list. Notice that entry 3 and 5 through 7 are not part of the list. These cells are available for any additions to the list. By creating a ListFree integer variable, a free list could be created to keep track of what cells are available. If all entries are in use, the size of the array would have to be increased or some elements would have to be deleted before new entries could be stored in the list.

The following code would traverse the list and display names and account balance:

```
i := listHead
while i ≥ 0 // loop through the list
  print i, Records[i].name, Records[i].balance // print entry
  i := Records[i].next
```

When faced with a choice, the advantages of this approach include:

- The linked list is relocatable, meaning it can be moved about in memory at will, and it can also be quickly and directly serialized for storage on disk or transfer over a network.
• Especially for a small list, array indexes can occupy significantly less space than a full pointer on many architectures.
• Locality of reference can be improved by keeping the nodes together in memory and by periodically rearranging them, although this can also be done in a general store.
• Naïve dynamic memory allocators can produce an excessive amount of overhead storage for each node allocated; almost no allocation overhead is incurred per node in this approach.
• Seizing an entry from a pre-allocated array is faster than using dynamic memory allocation for each node, since dynamic memory allocation typically requires a search for a free memory block of the desired size.

This approach has one main disadvantage, however: it creates and manages a private memory space for its nodes. This leads to the following issues:
• It increase complexity of the implementation.
• Growing a large array when it is full may be difficult or impossible, whereas finding space for a new linked list node in a large, general memory pool may be easier.
• Adding elements to a dynamic array will occasionally (when it is full) unexpectedly take linear (O(n)) instead of constant time (although it's still an amortized constant).
• Using a general memory pool leaves more memory for other data if the list is smaller than expected or if many nodes are freed.

For these reasons, this approach is mainly used for languages that do not support dynamic memory allocation. These disadvantages are also mitigated if the maximum size of the list is known at the time the array is created.

Language support
Many programming languages such as Lisp and Scheme have singly linked lists built in. In many functional languages, these lists are constructed from nodes, each called a cons or cons cell. The cons has two fields: the car, a reference to the data for that node, and the cdr, a reference to the next node. Although cons cells can be used to build other data structures, this is their primary purpose.

In languages that support abstract data types or templates, linked list ADTs or templates are available for building linked lists. In other languages, linked lists are typically built using references together with records.

Internal and external storage
When constructing a linked list, one is faced with the choice of whether to store the data of the list directly in the linked list nodes, called internal storage, or merely to store a reference to the data, called external storage. Internal storage has the advantage of making access to the data more efficient, requiring less storage overall, having better locality of reference, and simplifying memory management for the list (its data is allocated and deallocated at the same time as the list nodes).

External storage, on the other hand, has the advantage of being more generic, in that the same data structure and machine code can be used for a linked list no matter what the size of the data is. It also makes it easy to place the same data in multiple linked lists. Although with internal storage the same data can be placed in multiple lists by including multiple next references in the node data structure, it would then be necessary to create separate routines to add or delete cells based on each field. It is possible to create additional linked lists of elements that use internal storage by using external storage, and having the cells of the additional linked lists store references to the nodes of the linked list containing the data.

In general, if a set of data structures needs to be included in multiple linked lists, external storage is the best approach. If a set of data structures need to be included in only one linked list, then internal storage is slightly better, unless a generic linked list package using external storage is available. Likewise, if different sets of data that can be stored in the same data structure are to be included in a single linked list, then internal storage would be fine.
Another approach that can be used with some languages involves having different data structures, but all have the initial fields, including the next (and prev if double linked list) references in the same location. After defining separate structures for each type of data, a generic structure can be defined that contains the minimum amount of data shared by all the other structures and contained at the top (beginning) of the structures. Then generic routines can be created that use the minimal structure to perform linked list type operations, but separate routines can then handle the specific data. This approach is often used in message parsing routines, where several types of messages are received, but all start with the same set of fields, usually including a field for message type. The generic routines are used to add new messages to a queue when they are received, and remove them from the queue in order to process the message. The message type field is then used to call the correct routine to process the specific type of message.

**Example of internal and external storage**

Suppose you wanted to create a linked list of families and their members. Using internal storage, the structure might look like the following:

```plaintext
record member { // member of a family
    member next;
    string firstName;
    integer age;
}
record family { // the family itself
    family next;
    string lastName;
    string address;
    member members // head of list of members of this family
}
```

To print a complete list of families and their members using internal storage, we could write:

```plaintext
aFamily := Families // start at head of families list
while aFamily ≠ null // loop through list of families
    print information about family
    aMember := aFamily.members // get head of list of this family's members
    while aMember ≠ null // loop through list of members
        print information about member
        aMember := aMember.next
    aFamily := aFamily.next
```

Using external storage, we would create the following structures:

```plaintext
record node { // generic link structure
    node next;
    pointer data // generic pointer for data at node
}
record member { // structure for family member
    string firstName;
    integer age
}
record family { // structure for family
```
To print a complete list of families and their members using external storage, we could write:

```plaintext
famNode := Families // start at head of families list
while famNode ≠ null // loop through list of families
    aFamily := (family) famNode.data // extract family from node
    print information about family
    memNode := aFamily.members // get list of family members
    while memNode ≠ null // loop through list of members
        aMember := (member)memNode.data // extract member from node
        print information about member
        memNode := memNode.next
    famNode := famNode.next
```

Notice that when using external storage, an extra step is needed to extract the record from the node and cast it into the proper data type. This is because both the list of families and the list of members within the family are stored in two linked lists using the same data structure (node), and this language does not have parametric types.

As long as the number of families that a member can belong to is known at compile time, internal storage works fine. If, however, a member needed to be included in an arbitrary number of families, with the specific number known only at run time, external storage would be necessary.

### Speeding up search

Finding a specific element in a linked list, even if it is sorted, normally requires O(n) time (linear search). This is one of the primary disadvantages of linked lists over other data structures. In addition to the variants discussed above, below are two simple ways to improve search time.

In an unordered list, one simple heuristic for decreasing average search time is the move-to-front heuristic, which simply moves an element to the beginning of the list once it is found. This scheme, handy for creating simple caches, ensures that the most recently used items are also the quickest to find again.

Another common approach is to "index" a linked list using a more efficient external data structure. For example, one can build a red-black tree or hash table whose elements are references to the linked list nodes. Multiple such indexes can be built on a single list. The disadvantage is that these indexes may need to be updated each time a node is added or removed (or at least, before that index is used again).

### Random access lists

A random access list is a list with support for fast random access to read or modify any element in the list.\[5\] One possible implementation is a skew-binary random access list using the skew-binary number system, which involves a list of trees with special properties; this allows worst-case constant time head/cons operations, and worst-case logarithmic time random access to an element by index.\[5\] Random access lists can be implemented as persistent data structures.\[5\]

Random access lists can be viewed as immutable linked lists in that they likewise support the same O(1) head and tail operations.\[5\]
A simple extension to random access lists is the min-list, which provides an additional operation that yields the minimum element in the entire list in constant time (without mutation complexities).[5]

## Related data structures

Both stacks and queues are often implemented using linked lists, and simply restrict the type of operations which are supported.

The skip list is a linked list augmented with layers of pointers for quickly jumping over large numbers of elements, and then descending to the next layer. This process continues down to the bottom layer, which is the actual list.

A binary tree can be seen as a type of linked list where the elements are themselves linked lists of the same nature. The result is that each node may include a reference to the first node of one or two other linked lists, which, together with their contents, form the subtrees below that node.

An unrolled linked list is a linked list in which each node contains an array of data values. This leads to improved cache performance, since more list elements are contiguous in memory, and reduced memory overhead, because less metadata needs to be stored for each element of the list.

A hash table may use linked lists to store the chains of items that hash to the same position in the hash table.

A heap shares some of the ordering properties of a linked list, but is almost always implemented using an array. Instead of references from node to node, the next and previous data indexes are calculated using the current data's index.

A self-organizing list rearranges its nodes based on some heuristic which reduces search times for data retrieval by keeping commonly accessed nodes at the head of the list.

## Notes

2. Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (Technical Report CS-99-09), Resizable Arrays in Optimal Time and Space (http://www.cs.uwaterloo.ca/research/tr/1999/09/CS-99-09.pdf), Department of Computer Science, University of Waterloo,
3. The amount of control data required for a dynamic array is usually of the form $K + B \cdot n$, where $K$ is a per-array constant, $B$ is a per-dimension constant, and $n$ is the number of dimensions. $K$ and $B$ are typically on the order of 10 bytes.

## References

 Linked list


**External links**

- Description (http://nist.gov/dads/HTML/linkedList.html) from the Dictionary of Algorithms and Data Structures
- Some linked list materials are available from the Stanford University Computer Science department:
  - Introduction to Linked Lists (http://cslibrary.stanford.edu/103/)
  - Linked List Problems (http://cslibrary.stanford.edu/105/)
- Linked lists are a bad structure for modern computer systems. (http://www.futurechips.org/thoughts-for-researchers/quick-post-linked-lists.html)
- Patent for the idea of having nodes which are in several linked lists simultaneously (http://www.google.com/patents?vid=USPAT7028023) (note that this technique was widely used for many decades before the patent was granted)
Doubly linked list

In computer science, a doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

![A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.](image)

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

**Nomenclature and implementation**

The first and last nodes of a doubly linked list are immediately accessible (i.e., accessible without traversal, and usually called **head** and **tail**) and therefore allow traversal of the list from the beginning or end of the list, respectively: e.g., traversing the list from beginning to end, or from end to beginning, in a search of the list for a node with specific data value. Any node of a doubly linked list, once gotten, can be used to begin a new traversal of the list, in either direction (towards beginning or end), from the given node.

The link fields of a doubly linked list node are often called **next** and **previous** or **forward** and **backward**. The references stored in the link fields are usually implemented as pointers, but (as in any linked data structure) they may also be address offsets or indices into an array where the nodes live.

**Basic algorithms**

**Open doubly linked lists**

**Data type declarations**

```plaintext
record DoublyLinkedNode {
    prev // A reference to the previous node
    next // A reference to the next node
    data // Data or a reference to data
}

record DoublyLinkedList {
    Node firstNode // points to first node of list
    Node lastNode  // points to last node of list
}
```
Traversing the list

Traversal of a doubly linked list can be in either direction. In fact, the direction of traversal can change many times, if desired. **Traversal** is often called **iteration**, but that choice of terminology is unfortunate, for **iteration** has well-defined semantics (e.g., in mathematics) which are not analogous to **traversal**.

**Forwards**

```plaintext
node := list.firstNode
while node ≠ null
    <do something with node.data>
    node := node.next
```

**Backwards**

```plaintext
node := list.lastNode
while node ≠ null
    <do something with node.data>
    node := node.prev
```

Inserting a node

These symmetric functions insert a node either after or before a given node, with the diagram demonstrating after:

```
function insertAfter(List list, Node node, Node newNode)
    newNode.prev := node
    newNode.next := node.next
    if node.next == null
        list.lastNode := newNode
    else
        node.next.prev := newNode
        node.next := newNode
```

```
function insertBefore(List list, Node node, Node newNode)
    newNode.prev := node.prev
    newNode.next := node
    if node.prev == null
        list.firstNode := newNode
    else
        node.prev.next := newNode
        node.prev := newNode
```

We also need a function to insert a node at the beginning of a possibly empty list:
Doubly linked list

```plaintext
function insertBeginning(List list, Node newNode)
    if list.firstNode == null
        list.firstNode := newNode
        list.lastNode := newNode
        newNode.prev := null
        newNode.next := null
    else
        insertBefore(list, list.firstNode, newNode)

A symmetric function inserts at the end:

function insertEnd(List list, Node newNode)
    if list.lastNode == null
        insertBeginning(list, newNode)
    else
        insertAfter(list, list.lastNode, newNode)
```

Removing a node

Removal of a node is easier than insertion, but requires special handling if the node to be removed is the `firstNode` or `lastNode`:

```plaintext
function remove(List list, Node node)
    if node.prev == null
        list.firstNode := node.next
    else
        node.prev.next := node.next
    if node.next == null
        list.lastNode := node.prev
    else
        node.next.prev := node.prev
    destroy node
```

One subtle consequence of the above procedure is that deleting the last node of a list sets both `firstNode` and `lastNode` to `null`, and so it handles removing the last node from a one-element list correctly. Notice that we also don’t need separate "removeBefore" or "removeAfter" methods, because in a doubly linked list we can just use "remove(node.prev)" or "remove(node.next)" where these are valid. This also assumes that the node being removed is guaranteed to exist. If the node does not exist in this list, then some error handling would be required.

Circular doubly linked lists

Traversing the list

Assuming that `someNode` is some node in a non-empty list, this code traverses through that list starting with `someNode` (any node will do):

```plaintext
Forwards

node := someNode
do
    do something with node.value
    node := node.next
while node ≠ someNode
```
Backwards

node := someNode

\[
\text{do }
\begin{align*}
& \text{do something with node.value} \\
& \text{node := node.prev} \\
& \text{while node \neq someNode}
\end{align*}
\]

Notice the postponing of the test to the end of the loop. This is important for the case where the list contains only the single node someNode.

Inserting a node

This simple function inserts a node into a doubly linked circularly linked list after a given element:

\[
\text{function insertAfter(Node node, Node newNode)} \\
\text{newNode.next := node.next} \\
\text{newNode.prev := node} \\
\text{node.next.prev := newNode} \\
\text{node.next := newNode}
\]

To do an "insertBefore", we can simply "insertAfter(node.prev, newNode)". Inserting an element in a possibly empty list requires a special function:

\[
\text{function insertEnd(List list, Node node)} \\
\text{if list.lastNode == null} \\
\text{node.prev := node} \\
\text{node.next := node} \\
\text{else} \\
\text{insertAfter(list.lastNode, node)} \\
\text{list.lastNode := node}
\]

To insert at the beginning we simply "insertAfter(list.lastNode, node)". Finally, removing a node must deal with the case where the list empties:

\[
\text{function remove(List list, Node node)} \\
\text{if node.next == node} \\
\text{list.lastNode := null} \\
\text{else} \\
\text{node.next.prev := node.prev} \\
\text{node.prev.next := node.next} \\
\text{if node == list.lastNode} \\
\text{list.lastNode := node.prev;} \\
\text{destroy node}
\]

References
Stack (abstract data type)

In computer science, a stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only three fundamental operations: push, pop and stack top. The push operation adds a new item to the top of the stack, or initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items, or results in an empty stack, but if the stack is empty then it goes into underflow state (It means no items are present in stack to be removed). The stack top operation gets the data from the top-most position and returns it to the user without deleting it. The same underflow state can also occur in stack top operation if stack is empty.

A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest.[1]

History

The stack was first proposed in 1955, and then patented in 1957, by the German Friedrich L. Bauer.[2] The same concept was developed independently, at around the same time, by the Australian Charles Leonard Hamblin.

Abstract definition

A stack is a fundamental computer science data structure and can be defined in an abstract, implementation-free manner, or it can be generally defined as, Stack is a linear list of items in which all additions and deletion are restricted to one end that is Top.

This is a VDM (Vienna Development Method) description of a stack:[3]

Function signatures:

init: -> Stack
push: N x Stack -> Stack
top: Stack -> (N U ERROR)
remove: Stack -> Stack
isempty: Stack -> Boolean

(where N indicates an element (natural numbers in this case), and U indicates set union)

Semantics:

top(init()) = ERROR
top(push(i,s)) = i
remove(init()) = init()
remove(push(i, s)) = s
isempty(init()) = true
isempty(push(i, s)) = false
Inessential operations

In modern computer languages, the stack is usually implemented with more operations than just "push", "pop" and "Stack Top". Some implementations have a function which returns the current number of items on the stack. Alternatively, some implementations have a function that just returns if the stack is empty. Another typical helper operation stack top\(^4\) (also known as peek) can return the current top element of the stack without removing it.

Software stacks

Implementation

In most high level languages, a stack can be easily implemented either through an array or a linked list. What identifies the data structure as a stack in either case is not the implementation but the interface: the user is only allowed to pop or push items onto the array or linked list, with few other helper operations. The following will demonstrate both implementations, using C.

Array

The array implementation aims to create an array where the first element (usually at the zero-offset) is the bottom. That is, array[0] is the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack. The stack itself can therefore be effectively implemented as a two-element structure in C:

```c
typedef struct {
    size_t size;
    int items[STACKSIZE];
} STACK;
```

The push() operation is used both to initialize the stack, and to store values to it. It is responsible for inserting (copying) the value into the ps->items[] array and for incrementing the element counter (ps->size). In a responsible C implementation, it is also necessary to check whether the array is already full to prevent an overrun.

```c
void push(STACK *ps, int x)
{
    if (ps->size == STACKSIZE) {
        fputs("Error: stack overflow\n", stderr);
        abort();
    } else
    ps->items[ps->size++] = x;
}
```

The pop() operation is responsible for removing a value from the stack, and decrementing the value of ps->size. A responsible C implementation will also need to check that the array is not already empty.

```c
int pop(STACK *ps)
{
    if (ps->size == 0){
        fputs("Error: stack underflow\n", stderr);
        abort();
    } else
    return ps->items[--ps->size];
}
```
If we use a dynamic array, then we can implement a stack that can grow or shrink as much as needed. The size of the stack is simply the size of the dynamic array. A dynamic array is a very efficient implementation of a stack, since adding items to or removing items from the end of a dynamic array is amortized O(1) time.

**Linked list**

The **linked-list** implementation is equally simple and straightforward. In fact, a simple singly linked list is sufficient to implement a stack—it only requires that the head node or element can be removed, or popped, and a node can only be inserted by becoming the new head node.

Unlike the array implementation, our structure typedef corresponds not to the entire stack structure, but to a single node:

```c
typedef struct stack {
    int data;
    struct stack *next;
} STACK;
```

Such a node is identical to a typical singly linked list node, at least to those that are implemented in C.

The `push()` operation both initializes an empty stack, and adds a new node to a non-empty one. It works by receiving a data value to push onto the stack, along with a target stack, creating a new node by allocating memory for it, and then inserting it into a linked list as the new head:

```c
void push(STACK **head, int value) {
    STACK *node = malloc(sizeof(STACK)); /* create a new node */

    if (node == NULL) {
        fputs("Error: no space available for node\n", stderr);
        abort();
    } else { /* initialize node */
        node->data = value;
        node->next = empty(*head) ? NULL : *head; /* insert new head if any */
        *head = node;
    }
}
```

A `pop()` operation removes the head from the linked list, and assigns the pointer to the head to the previous second node. It checks whether the list is empty before popping from it:

```c
int pop(STACK **head) {
    if (empty(*head)) { /* stack is empty */
        fputs("Error: stack underflow\n", stderr);
        abort();
    } else { /* pop a node */
        STACK *top = *head;
        int value = top->data;
        *head = top->next;
        free(top);
    }
}
```
Stacks and programming languages

Some languages, like LISP and Python, do not call for stack implementations, since push and pop functions are available for any list. All Forth-like languages (such as Adobe PostScript) are also designed around language-defined stacks that are directly visible to and manipulated by the programmer.

C++’s Standard Template Library provides a "stack" templated class which is restricted to only push/pop operations. Java’s library contains a Stack class that is a specialization of Vector---this could be considered a design flaw, since the inherited get() method from Vector ignores the LIFO constraint of the Stack. PHP has an SplStack [5] class.

Hardware stacks

A common use of stacks at the architecture level is as a means of allocating and accessing memory.

Basic architecture of a stack
A typical stack is an area of computer memory with a fixed origin and a variable size. Initially the size of the stack is zero. A stack pointer, usually in the form of a hardware register, points to the most recently referenced location on the stack; when the stack has a size of zero, the stack pointer points to the origin of the stack.

The two operations applicable to all stacks are:
- a push operation, in which a data item is placed at the location pointed to by the stack pointer, and the address in the stack pointer is adjusted by the size of the data item;
- a pop or pull operation: a data item at the current location pointed to by the stack pointer is removed, and the stack pointer is adjusted by the size of the data item.

There are many variations on the basic principle of stack operations. Every stack has a fixed location in memory at which it begins. As data items are added to the stack, the stack pointer is displaced to indicate the current extent of the stack, which expands away from the origin.

Stack pointers may point to the origin of a stack or to a limited range of addresses either above or below the origin (depending on the direction in which the stack grows); however, the stack pointer cannot cross the origin of the stack. In other words, if the origin of the stack is at address 1000 and the stack grows downwards (towards addresses 999, 998, and so on), the stack pointer must never be incremented beyond 1000 (to 1001, 1002, etc.). If a pop operation on the stack causes the stack pointer to move past the origin of the stack, a stack underflow occurs. If a push operation causes the stack pointer to increment or decrement beyond the maximum extent of the stack, a stack overflow occurs.

Some environments that rely heavily on stacks may provide additional operations, for example:
- **Dup(licate):** the top item is popped, and then pushed again (twice), so that an additional copy of the former top item is now on top, with the original below it.
- **Peek:** the topmost item is inspected (or returned), but the stack pointer is not changed, and the stack size does not change (meaning that the item remains on the stack). This is also called top operation in many articles.
- **Swap or exchange:** the two topmost items on the stack exchange places.
- **Rotate or Roll:** the n topmost items are moved on the stack in a rotating fashion. For example, if n=3, items 1, 2, and 3 on the stack are moved to positions 2, 3, and 1 on the stack, respectively. Many variants of this operation are possible, with the most common being called left rotate and right rotate.
Stacks are either visualized growing from the bottom up (like real-world stacks), or, with the top of the stack in a fixed position (see image [note in the image, the top (28) is the stack 'bottom', since the stack 'top' is where items are pushed or popped from]), a coin holder, a Pez dispenser, or growing from left to right, so that "topmost" becomes "rightmost". This visualization may be independent of the actual structure of the stack in memory. This means that a right rotate will move the first element to the third position, the second to the first and the third to the second. Here are two equivalent visualizations of this process:

<table>
<thead>
<tr>
<th>apple</th>
<th>banana</th>
</tr>
</thead>
<tbody>
<tr>
<td>banana</td>
<td>===right rotate==&gt; cucumber</td>
</tr>
<tr>
<td>cucumber</td>
<td>apple</td>
</tr>
<tr>
<td>cucumber</td>
<td>apple</td>
</tr>
<tr>
<td>banana</td>
<td>===left rotate==&gt; cucumber</td>
</tr>
<tr>
<td>apple</td>
<td>banana</td>
</tr>
</tbody>
</table>

A stack is usually represented in computers by a block of memory cells, with the "bottom" at a fixed location, and the stack pointer holding the address of the current "top" cell in the stack. The top and bottom terminology are used irrespective of whether the stack actually grows towards lower memory addresses or towards higher memory addresses.

Pushing an item on to the stack adjusts the stack pointer by the size of the item (either decrementing or incrementing, depending on the direction in which the stack grows in memory), pointing it to the next cell, and copies the new top item to the stack area. Depending again on the exact implementation, at the end of a push operation, the stack pointer may point to the next unused location in the stack, or it may point to the topmost item in the stack. If the stack points to the current topmost item, the stack pointer will be updated before a new item is pushed onto the stack; if it points to the next available location in the stack, it will be updated after the new item is pushed onto the stack.

Popping the stack is simply the inverse of pushing. The topmost item in the stack is removed and the stack pointer is updated, in the opposite order of that used in the push operation.

**Hardware support**

**Stack in main memory**

Most CPUs have registers that can be used as stack pointers. Processor families like the x86, Z80, 6502, and many others have special instructions that implicitly use a dedicated (hardware) stack pointer to conserve opcode space. Some processors, like the PDP-11 and the 68000, also have special addressing modes for implementation of stacks, typically with a semi-dedicated stack pointer as well (such as A7 in the 68000). However, in most processors, several different registers may be used as additional stack pointers as needed (whether updated via addressing modes or via add/sub instructions).

**Stack in registers or dedicated memory**

The x87 floating point architecture is an example of a set of registers organised as a stack where direct access to individual registers (relative the current top) is also possible. As with stack-based machines in general, having the top-of-stack as an implicit argument allows for a small machine code footprint with a good usage of bus bandwidth and code caches, but it also prevents some types of optimizations possible on processors permitting random access to the register file for all (two or three) operands. A stack structure also makes superscalar implementations with register renaming (for speculative execution) somewhat more complex to implement, although it is still feasible, as exemplified by modern x87 implementations.

Sun SPARC, AMD Am29000, and Intel i960 are all examples of architectures using register windows within a register-stack as another strategy to avoid the use of slow main memory for function arguments and return values.
There are also a number of small microprocessors that implements a stack directly in hardware and some microcontrollers have a fixed-depth stack that is not directly accessible. Examples are the PIC microcontrollers, the Computer Cowboys MuP21, the Harris RTX line, and the Novix NC4016. Many stack-based microprocessors were used to implement the programming language Forth at the microcode level. Stacks were also used as a basis of a number of mainframes and mini computers. Such machines were called stack machines, the most famous being the Burroughs B5000.

**Applications**

Stacks have numerous applications. We see stacks in everyday life, from the books in our library, to the sheaf of papers that we keep in our printer tray. All of them follow the *Last In First Out* (LIFO) logic, that is when we add a book to a pile of books, we add it to the top of the pile, whereas when we remove a book from the pile, we generally remove it from the top of the pile.

Given below are a few applications of stacks in the world of computers:

**Converting a decimal number into a binary number**

The logic for transforming a decimal number into a binary number is as follows:

1. Read a number
2. Iteration (while number is greater than zero)
   1. Find out the remainder after dividing the number by 2
   2. Print the remainder
   3. Divide the number by 2
3. End the iteration

However, there is a problem with this logic. Suppose the number, whose binary form we want to find is 23. Using this logic, we get the result as 11101, instead of getting 10111.

To solve this problem, we use a stack. We make use of the LIFO property of the stack. Initially we push the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we pop one digit at a time from the stack and print it. Therefore we get the decimal number is converted into its proper binary form.

**Algorithm:**

```java
function outputInBinary(Integer n)
    Stack s = new Stack
    while n > 0 do
        Integer bit = n modulo 2
        s.push(bit)
    endwhile
    while s is not empty do
        print s.pop()
    endwhile
```

Stack of books
if s is full then
    return error
end if
n = floor(n / 2)
end while
while s is not empty do
    output(s.pop())
end while
end function

Towers of Hanoi

One of the most interesting applications of stacks can be found in solving a puzzle called Tower of Hanoi. According to an old Brahmin story, the existence of the universe is calculated in terms of the time taken by a number of monks, who are working all the time, to move 64 disks from one pole to another. But there are some rules about how this should be done, which are:

1. You can move only one disk at a time.
2. For temporary storage, a third pole may be used.
3. You cannot place a disk of larger diameter on a disk of smaller diameter.[7]

For algorithm of this puzzle see Tower of Hanoi.

Here we assume that A is first tower, B is second tower & C is third tower.
Stack (abstract data type)

Towers of Hanoi step 1

Initial step

After moving from A to C

Towers of Hanoi step 2

After moving from A to B

After moving from C to B

Towers of Hanoi step 3

After moving from A to C

After moving from B to A
Output: (when there are 3 disks)
Let 1 be the smallest disk, 2 be the disk of medium size and 3 be the largest disk.
The C++ code for this solution can be implemented in two ways:

First Implementation (Without using Stacks)

```cpp
void TowersofHanoi(int n, int a, int b, int c)
{
    if(n > 0)
    {
        TowersofHanoi(n-1, a, c, b); //recursion
        cout << " Move top disk from tower " <<
        a << " to tower " << b << endl ;
        TowersofHanoi(n-1, c, b, a); //recursion
    }
}
```

Second Implementation (Using Stacks)

```cpp
// Global variable , tower [1:3] are three towers
arrayStack<int> tower[4];

void TowerofHanoi(int n)
{
    // Preprocessor for moveAndShow.
    // initialize
    for (int d = n; d > 0; d--)
                    //add disk d to tower 1
            tower[1].push(d);
    moveAndShow(n, 1, 2, 3); /*move n disks from tower 1 to
tower 3 using
    tower 2 as intermediate tower*/
}

void moveAndShow(int n, int a, int b, int c)
{
    // Move the top n disks from tower a to tower b showing states.
    // Use tower c for intermediate storage.
    if(n > 0)
    {
```
However complexity for above written implementations is $O(2^n)$. So it’s obvious that problem can only be solved for small values of $n$ (generally $n \leq 30$). In case of the monks, the number of turns taken to transfer 64 disks, by following the above rules, will be 18,446,744,073,709,551,615; which will surely take a lot of time!\(^7\)\(^8\)

### Expression evaluation and syntax parsing

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations and conversion from one form to another may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most programming languages are context-free languages, allowing them to be parsed with stack based machines.

**Evaluation of an Infix Expression that is Fully Parenthesized**

**Input:** $(((2 \times 5) - (1 \times 2)) / (11 - 9))$

**Output:** 4

**Analysis:** Five types of input characters

<table>
<thead>
<tr>
<th>* Opening bracket</th>
<th>* Numbers</th>
<th>* Operators</th>
<th>* Closing bracket</th>
<th>* New line character</th>
</tr>
</thead>
</table>

**Data structure requirement:** A character stack

**Algorithm**

1. Read one input character
2. Actions at end of each input
   - **Opening brackets** (2.1) Push into stack and then Go to step (1)
   - **Number** (2.2) Push into stack and then Go to step (1)
   - **Operator** (2.3) Push into stack and then Go to step (1)
   - **Closing brackets** (2.4) Pop from character stack
     - (2.4.1) if it is closing bracket, then discard it, Go to step (1)
     - (2.4.2) Pop is used three times
       - The first popped element is assigned to op2
       - The second popped element is assigned to op
       - The third popped element is assigned to op1
       - Evaluate op1 op op2
       - Convert the result into character and
Result: The evaluation of the fully parenthesized infix expression is printed as follows:

Input String: $(((2 * 5) - (1 * 2)) / (11 - 9))$

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Stack (from bottom to top)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>( 10</td>
<td>$2 * 5 = 10$ and push</td>
</tr>
<tr>
<td>-</td>
<td>( 10 -</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>( 10 - (</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( 10 - ( 1</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>( 10 - ( 1 *</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( 10 - ( 1 * 2</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>( 10 - 2</td>
<td>$1 * 2 = 2 \text{ &amp; Push}$</td>
</tr>
<tr>
<td>)</td>
<td>( 8</td>
<td>$10 - 2 = 8 \text{ &amp; Push}$</td>
</tr>
<tr>
<td>/</td>
<td>( 8 /</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>( 8 /</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>( 8 / ( 11</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>( 8 / ( 11 -</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>( 8 / ( 11 - 9</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>( 8 / 2</td>
<td>$11 - 9 = 2 \text{ &amp; Push}$</td>
</tr>
<tr>
<td>)</td>
<td>4</td>
<td>$8 / 2 = 4 \text{ &amp; Push}$</td>
</tr>
<tr>
<td>New line</td>
<td>Empty</td>
<td>Push &amp; Print</td>
</tr>
</tbody>
</table>

Evaluation of Infix Expression which is not fully parenthesized

Input: $(2 * 5 - 1 * 2) / (11 - 9)$

Output: 4

Analysis: There are five types of input characters which are:

* Opening brackets
* Numbers
* Operators
* Closing brackets
* New line character (\n)
We do not know what to do if an operator is read as an input character. By implementing the priority rule for operators, we have a solution to this problem.

The **Priority rule**: we should perform a comparative priority check if an operator is read, and then push it. If the stack *top* contains an operator of priority higher than or equal to the priority of the input operator, then we *pop* it and print it. We keep on performing the priority check until the *top* of stack either contains an operator of lower priority or if it does not contain an operator.

**Data Structure Requirement for this problem**: A character stack and an integer stack

**Algorithm**:

1. Read an input character

2. Actions that will be performed at the end of each input
   - **Opening brackets**
     - Push it into character stack and then Go to step (1)
   - **Digit**
     - Push into integer stack, Go to step (1)
   - **Operator**
     - Do the comparative priority check
     - (2.3.1) if the character stack's *top* contains an operator with equal or higher priority, then *pop* it into *op*
       - Pop a number from integer stack into *op2*
       - Pop another number from integer stack into *op1*
       - Calculate *op1* *op* *op2* and *push* the result into the integer stack
   - **Closing brackets**
     - Pop from the character stack
     - (2.4.1) if it is an opening bracket, then discard it and Go to step (1)
     - (2.4.2) To *op*, assign the popped element
       - Pop a number from integer stack and assign it *op2*
       - Pop another number from integer stack and assign it to *op1*
       - Calculate *op1* *op* *op2* and push the result into the integer stack
       - Convert into character and *push* into stack
     - Go to the step (2.4)
   - **New line character**
     - Print the result after popping from the stack

**Result**: The evaluation of an infix expression that is not fully parenthesized is printed as follows:

**Input String**: \((2 * 5 - 1 * 2) / (11 - 9)\)
Evaluation of Prefix Expression

Input: / - 2 5 * 1 2 - 11 9

Output: 4

Analysis: There are three types of input characters

* Numbers
* Operators
* New line character (\n)

Data structure requirement: A character stack and an integer stack

Algorithm:

1. Read one character input at a time and keep pushing it into the character stack until the new line character is reached
2. Perform pop from the character stack. If the stack is empty, go to step (3)
   - Number
     2.1) Push in to the integer stack and then go to step (1)
   - Operator
     2.2) Assign the operator to op
          Pop a number from integer stack and assign it to op1
          Pop another number from integer stack and assign it to op2
          Calculate op1 op op2 and push the output into the integer stack

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Character Stack (from bottom to top)</th>
<th>Integer Stack (from bottom to top)</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>(*</td>
<td></td>
<td>Push as * has higher priority</td>
</tr>
<tr>
<td>5</td>
<td>(*</td>
<td>2 5</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>(*</td>
<td></td>
<td>Since - has less priority, we do 2 * 5 = 10</td>
</tr>
<tr>
<td>1</td>
<td>(-</td>
<td>10</td>
<td>We push 10 and then push -</td>
</tr>
<tr>
<td>*</td>
<td>(- *)</td>
<td>10 1</td>
<td>Push * as it has higher priority</td>
</tr>
<tr>
<td>2</td>
<td>(- *)</td>
<td>10 1 2</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>(-</td>
<td>10 2</td>
<td>Perform 1 * 2 = 2 and push it</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>8</td>
<td>Pop - and 10 - 2 = 8 and push, Pop (</td>
</tr>
<tr>
<td>/</td>
<td>/(</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>/</td>
<td>8 11</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>/(-</td>
<td>8 11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>/(-</td>
<td>8 11 9</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>/</td>
<td>8 2</td>
<td>Perform 11 - 9 = 2 and push it</td>
</tr>
<tr>
<td>New line</td>
<td>4</td>
<td></td>
<td>Perform 8 / 2 = 4 and push it</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>Print the output, which is 4</td>
</tr>
</tbody>
</table>

[9]
3. Pop the result from the integer stack and display the result

**Result:** The evaluation of prefix expression is printed as follows:

**Input String:** / - * 2 5 * 1 2 - 11 9

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Character Stack (from bottom to top)</th>
<th>Integer Stack (from bottom to top)</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>/ - *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>/ - * 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>/ - * 25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>/ - * 25 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>/ - * 25 * 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>/ - * 25 * 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>/ - * 25 * 12 -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>/ - * 25 * 12 - 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>/ - * 25 * 12 - 11 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ln</td>
<td>/ - * 25 * 12 - 11 9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Evaluation of postfix expression

The calculation: 1 + 2 * 4 + 3 can be written down like this in postfix notation with the advantage of no precedence rules and parentheses needed:

1 2 4 * + 3 +

The expression is evaluated from the left to right using a stack:

1. when encountering an operand: push it
2. when encountering an operator: pop two operands, evaluate the result and push it.

Like the following way (the Stack is displayed after Operation has taken place):
Stack (abstract data type)

<table>
<thead>
<tr>
<th>Input</th>
<th>Operation</th>
<th>Stack (after op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Push operand</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Push operand</td>
<td>2, 1</td>
</tr>
<tr>
<td>4</td>
<td>Push operand</td>
<td>4, 2, 1</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>8, 1</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Push operand</td>
<td>3, 9</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
<td>12</td>
</tr>
</tbody>
</table>

The final result, 12, lies on the top of the stack at the end of the calculation.

Example in C

```c
#include<stdio.h>

int main()
{
    int a[100], i;
    printf("To pop enter -1\n\n");
    for(i = 0;;)
    {
        printf("Push ");
        scanf("%d", &a[i]);
        if(a[i] == -1)
        {
            if(i == 0)
            {
                printf("Underflow\n");
            }
            else
            {
                printf("pop = %d\n", a[--i]);
            }
        }
        else
        {
            i++;
        }
    }
}
```
Evaluation of postfix expression (Pascal)

This is an implementation in Pascal, using marked sequential file as data archives.

```pascal
{ programmer : clx321
file : stack.pas
unit : Pstack.tpu
}
program TestStack;
{this program uses ADT of Stack, I will assume that the unit of ADT of Stack has already existed}

uses
  PStack;  {ADT of STACK}
{dictionary}
const
  mark = '.';
var
  data : stack;
  f : text;
  cc : char;
  ccInt, cc1, cc2 : integer;
{functions}
  IsOperand (cc : char) : boolean;  {JUST Prototype}
      {return TRUE if cc is operand}
  ChrToInt (cc : char) : integer;  {JUST Prototype}
      {change char to integer}
  Operator (cc1, cc2 : integer) : integer;  {JUST Prototype}
      {operate two operands}
{algorithms}
begin
  assign (f, cc);
  reset (f);
  read (f, cc);  {first elmt}
  if (cc = mark) then
    begin
      writeln ('empty archives !');
    end
  else
    begin
      repeat
        if (IsOperand (cc)) then
          begin
            ccInt := ChrToInt (cc);
            end
        cc1 := Operator (cc, cc1);
        end
  end;
```


Conversion of an Infix expression that is fully parenthesized into a Postfix expression

Input: (((8 + 1) - (7 - 4)) / (11 - 9))

Output: 8 1 + 7 4 - - 11 9 - /

Analysis: There are five types of input characters which are:

* Opening brackets
* Numbers
* Operators
* Closing brackets
* New line character (\n)

Requirement: A character stack

Algorithm:

1. Read an character input
2. Actions to be performed at end of each input
   - Opening brackets: (2.1) Push into stack and then Go to step (1)
   - Number: (2.2) Print and then Go to step (1)
   - Operator: (2.3) Push into stack and then Go to step (1)
   - Closing brackets: (2.4) Pop it from the stack
     - (2.4.1) If it is an operator, print it, Go to step (1)
     - (2.4.2) If the popped element is an opening bracket, discard it and go to step (1)
   - New line character: (2.5) STOP

Therefore, the final output after conversion of an infix expression to a postfix expression is as follows:
<table>
<thead>
<tr>
<th>Input</th>
<th>Operation</th>
<th>Stack (after op)</th>
<th>Output on monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(2.2) Print it</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>+</td>
<td>(2.3) Push operator into stack</td>
<td>( ( +</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>(2.2) Print it</td>
<td></td>
<td>8 1</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '+' print it</td>
<td>(</td>
<td>8 1 +</td>
</tr>
<tr>
<td></td>
<td>(2.4) Pop from the stack: Since popped element is ')' we ignore it and read next character</td>
<td>(</td>
<td>8 1 +</td>
</tr>
<tr>
<td>-</td>
<td>(2.3) Push operator into stack</td>
<td>( -</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push operand into stack</td>
<td>( - (</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(2.2) Print it</td>
<td></td>
<td>8 1 + 7</td>
</tr>
<tr>
<td>-</td>
<td>(2.3) Push the operator in the stack</td>
<td>( - ( -</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(2.2) Print it</td>
<td></td>
<td>8 1 + 7 4</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '-' print it</td>
<td>( - (</td>
<td>8 1 + 7 4 -</td>
</tr>
<tr>
<td></td>
<td>(2.4) Pop from the stack: Since popped element is ')' we ignore it and read next character</td>
<td>(</td>
<td>8 1 + 7 4 -</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '-' print it</td>
<td>(</td>
<td>8 1 + 7 4 -</td>
</tr>
<tr>
<td></td>
<td>(2.4) Pop from the stack: Since popped element is ')' we ignore it and read next character</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>(2.3) Push the operand into the stack</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(2.1) Push into the stack</td>
<td>/ (</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(2.2) Print it</td>
<td></td>
<td>8 1 + 7 4 - 11</td>
</tr>
<tr>
<td>-</td>
<td>(2.3) Push the operand into the stack</td>
<td>/ ( -</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(2.2) Print it</td>
<td></td>
<td>8 1 + 7 4 - 11 9</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '-' print it</td>
<td>/ (</td>
<td>8 1 + 7 4 - 11 9 -</td>
</tr>
<tr>
<td></td>
<td>(2.4) Pop from the stack: Since popped element is ')' we ignore it and read next character</td>
<td>/</td>
<td>8 1 + 7 4 - 11 9 -</td>
</tr>
<tr>
<td>)</td>
<td>(2.4) Pop from the stack: Since popped element is '/' print it</td>
<td>(</td>
<td>8 1 + 7 4 - 11 9 -</td>
</tr>
<tr>
<td></td>
<td>(2.4) Pop from the stack: Since popped element is ')' we ignore it and read next character</td>
<td>Stack is empty</td>
<td></td>
</tr>
</tbody>
</table>

New line character | (2.5) STOP | Stack is empty |
**Rearranging railroad cars**

**Problem Description**
This is one useful application of stacks. Consider that a freight train has $n$ railroad cars, each to be left at different station. They're numbered 1 through $n$ and freight train visits these stations in order $n$ through 1. Obviously, the railroad cars are labeled by their destination. To facilitate removal of the cars from the train, we must rearrange them in ascending order of their number (i.e. 1 through $n$). When cars are in this order, they can be detached at each station. We rearrange cars at a shunting yard that has *input track*, *output track* and $k$ holding tracks between input & output tracks (i.e. *holding track*).

**Solution Strategy**
To rearrange cars, we examine the cars on the input from front to back. If the car being examined is next one in the output arrangement, we move it directly to *output track*. If not, we move it to the *holding track* & leave it there until it's time to place it to the *output track*. The holding tracks operate in a LIFO manner as the cars enter & leave these tracks from top. When rearranging cars only following moves are permitted:

- A car may be moved from front (i.e. right end) of the input track to the top of one of the *holding tracks* or to the left end of the output track.
- A car may be moved from the top of *holding track* to left end of the *output track*.

The figure shows a shunting yard with $k = 3$, holding tracks $H1$, $H2$ & $H3$, also $n = 9$. The $n$ cars of freight train begin in the input track & are to end up in the output track in order 1 through $n$ from right to left. The cars initially are in the order 5,8,1,7,4,2,9,6,3 from back to front. Later cars are rearranged in desired order.
A Three Track Example

- Consider the input arrangement from figure, here we note that the car 3 is at the front, so it can't be output yet, as it to be preceded by cars 1 & 2. So car 3 is detached & moved to holding track H1.
- The next car 6 can't be output & it is moved to holding track H2. Because we have to output car 3 before car 6 & this will not possible if we move car 6 to holding track H1.
- Now it's obvious that we move car 9 to H3.

The requirement of rearrangement of cars on any holding track is that the cars should be preferred to arrange in ascending order from top to bottom.

- So car 2 is now moved to holding track H1 so that it satisfies the previous statement. If we move car 2 to H2 or H3, then we've no place to move cars 4,5,7,8. The least restrictions on future car placement arise when the new car λ is moved to the holding track that has a car at its top with smallest label Ψ such that λ < Ψ. We may call it an assignment rule to decide whether a particular car belongs to a specific holding track.

- When car 4 is considered, there are three places to move the car H1,H2,H3. The top of these tracks are 2,6,9. So using above mentioned Assignment rule, we move car 4 to H2.
- The car 7 is moved to H3.
- The next car 1 has the least label, so it's moved to output track.
- Now it's time for car 2 & 3 to output which are from H1 (in short all the cars from H1 are appended to car 1 on output track).

The car 4 is moved to output track. No other cars can be moved to output track at this time.

- The next car 8 is moved to holding track H1.
- Car 5 is output from input track. Car 6 is moved to output track from H2, so is the 7 from H3,8 from H1 & 9 from H3.

Quicksort

Sorting means arranging the list of elements in a particular order. In case of numbers, it could be in ascending order, or in the case of letters, alphabetic order.

Quicksort is an algorithm of the divide and conquer type. In this method, to sort a set of numbers, we reduce it to two smaller sets, and then sort these smaller sets.

This can be explained with the help of the following example:

Suppose A is a list of the following numbers:
In the reduction step, we find the final position of one of the numbers. In this case, let us assume that we have to find the final position of 48, which is the first number in the list.

To accomplish this, we adopt the following method. Begin with the last number, and move from right to left. Compare each number with 48. If the number is smaller than 48, we stop at that number and swap it with 48.

In our case, the number is 24. Hence, we swap 24 and 48.

The numbers 96 and 72 to the right of 48, are greater than 48. Now beginning with 24, scan the numbers in the opposite direction, that is from left to right. Compare every number with 48 until you find a number that is greater than 48.

In this case, it is 60. Therefore we swap 48 and 60.

Note that the numbers 12, 24 and 36 to the left of 48 are all smaller than 48. Now, start scanning numbers from 60, in the right to left direction. As soon as you find lesser number, swap it with 48.

In this case, it is 44. Swap it with 48. The final result is:

Now, beginning with 44, scan the list from left to right, until you find a number greater than 48. Such a number is 84. Swap it with 48. The final result is:

Now, beginning with 84, traverse the list from right to left, until you reach a number lesser than 48. We do not find such a number before reaching 48. This means that all the numbers in the list have been scanned and compared with 48. Also, we notice that all numbers less than 48 are to the left of it, and all numbers greater than 48, are to its right.
The final partitions look as follows:

![Partition Diagram]

First partition  Second partition

24 36 12 44 48 98 77 65 108 60 96 72

Therefore, 48 has been placed in its proper position and now our task is reduced to sorting the two partitions. This above step of creating partitions can be repeated with every partition containing 2 or more elements. As we can process only a single partition at a time, we should be able to keep track of the other partitions, for future processing. This is done by using two stacks called LOWERBOUND and UPPERBOUND, to temporarily store these partitions. The addresses of the first and last elements of the partitions are pushed into the LOWERBOUND and UPPERBOUND stacks respectively. Now, the above reduction step is applied to the partitions only after it's boundary values are popped from the stack.

We can understand this from the following example:

Take the above list A with 12 elements. The algorithm starts by pushing the boundary values of A, that is 1 and 12 into the LOWERBOUND and UPPERBOUND stacks respectively. Therefore the stacks look as follows:

<table>
<thead>
<tr>
<th>LOWERBOUND:  1</th>
<th>UPPERBOUND:  12</th>
</tr>
</thead>
</table>

To perform the reduction step, the values of the stack top are popped from the stack. Therefore, both the stacks become empty.

<table>
<thead>
<tr>
<th>LOWERBOUND:  {empty}</th>
<th>UPPERBOUND:  {empty}</th>
</tr>
</thead>
</table>

Now, the reduction step causes 48 to be fixed to the 5th position and creates two partitions, one from position 1 to 4 and the other from position 6 to 12. Hence, the values 1 and 6 are pushed into the LOWERBOUND stack and 4 and 12 are pushed into the UPPERBOUND stack.

<table>
<thead>
<tr>
<th>LOWERBOUND:  1, 6</th>
<th>UPPERBOUND:  4, 12</th>
</tr>
</thead>
</table>

For applying the reduction step again, the values at the stack top are popped. Therefore, the values 6 and 12 are popped. Therefore the stacks look like:

<table>
<thead>
<tr>
<th>LOWERBOUND:  1</th>
<th>UPPERBOUND:  4</th>
</tr>
</thead>
</table>

The reduction step is now applied to the second partition, that is from the 6th to 12th element.
After the reduction step, 98 is fixed in the 11th position. So, the second partition has only one element. Therefore, we push the upper and lower boundary values of the first partition onto the stack. So, the stacks are as follows:

<table>
<thead>
<tr>
<th>LOWERBOUND</th>
<th>UPPERBOUND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 6</td>
<td>4, 10</td>
</tr>
</tbody>
</table>

The processing proceeds in the following way and ends when the stacks do not contain any upper and lower bounds of the partition to be processed, and the list gets sorted.

The Stock Span Problem

In the stock span problem, we will solve a financial problem with the help of stacks.

Suppose, for a stock, we have a series of $n$ daily price quotes, the span of the stock’s price on a particular day is defined as the maximum number of consecutive days for which the price of the stock on the current day is less than or equal to its price on that day.

An algorithm which has Quadratic Time Complexity

**Input:** An array $P$ with $n$ elements

**Output:** An array $S$ of $n$ elements such that $P[i]$ is the largest integer $k$ such that $k \leq i + 1$ and $P[y] \leq P[i]$ for $j = i - k + 1, \ldots, i$

**Algorithm:**

1. Initialize an array $P$ which contains the daily prices of the stocks
2. Initialize an array $S$ which will store the span of the stock
3. for $i = 0$ to $i = n - 1$
   3.1 Initialize $k$ to zero
   3.2 Done with a false condition
   3.3 repeat
      3.3.1 if ($P[i - k] \leq P[i]$) then
         Increment $k$ by 1
      3.3.2 else
         Done with true condition
   3.4 Till ($k > i$) or done with processing
Assign value of k to S[i] to get the span of the stock

4. Return array S

Now, analyzing this algorithm for running time, we observe:

- We have initialized the array S at the beginning and returned it at the end. This is a constant time operation, hence takes $O(n)$ time.
- The repeat loop is nested within the for loop. The for loop, whose counter is $i$, is executed $n$ times. The statements which are not in the repeat loop, but in the for loop are executed $n$ times. Therefore, these statements and the incrementing and condition testing of $i$ take $O(n)$ time.
- In repetition of $i$ for the outer for loop, the body of the inner repeat loop is executed maximum $i + 1$ times. In the worst case, element $S[i]$ is greater than all the previous elements. So, testing for the if condition, the statement after that, as well as testing the until condition, will be performed $i + 1$ times during iteration $i$ for the outer for loop. Hence, the total time taken by the inner loop is $O(n(n + 1)/2)$, which is $O(n^2)$.

The running time of all these steps is calculated by adding the time taken by all these three steps. The first two terms are $O(n)$ while the last term is $O(n^2)$. Therefore, the total running time of the algorithm is $O(n^2)$.

**An algorithm which has Linear Time Complexity**

In order to calculate the span more efficiently, we see that the span on a particular day can be easily calculated if we know the closest day before $i$, such that the price of the stocks on that day was higher than the price of the stocks on the present day. If there exists such a day, we can represent it by $h(i)$ and initialize $h(i)$ to be -1.

Therefore the span of a particular day is given by the formula, $s = i - h(i)$.

To implement this logic, we use a stack as an abstract data type to store the days $i$, $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to $i$, we pop the days when the price of the stock was less than or equal to $p(i)$ and then push the value of day $i$ back into the stack.

Here, we assume that the stack is implemented by operations that take $O(1)$ that is constant time. The algorithm is as follows:

**Input:** An array $P$ with $n$ elements and an empty stack $N$

**Output:** An array $S$ of $n$ elements such that $P[i]$ is the largest integer $k$ such that $k <= i + 1$ and $P[y] <= P[i]$ for $j = i - k + 1, ...., i$

**Algorithm:**

1. Initialize an array $P$ which contains the daily prices of the stocks
2. Initialize an array $S$ which will store the span of the stock
3. for $i = 0$ to $i = n - 1$
   3.1 Initialize $k$ to zero
   3.2 Done with a false condition
   3.3 while not (Stack $N$ is empty or done with processing)
      3.3.1 if ( $P[i] >= P[N.top()]$) then
         Pop a value from stack $N$
      3.3.2 else
         Done with true condition
      3.4 if Stack $N$ is empty
         3.4.1 Initialize $h$ to -1
      3.5 else
         3.5.1 Initialize stack top to $h$
         3.5.2 Put the value of $h - i$ in $S[i]$


3.5.3 Push the value of i in N

4. Return array S

Now, analyzing this algorithm for running time, we observe:

- We have initialized the array S at the beginning and returned it at the end. This is a constant time operation, hence takes $O(n)$ time.
- The while loop is nested within the for loop. The for loop, whose counter is $i$, is executed $n$ times. The statements which are not in the repeat loop, but in the for loop are executed $n$ times. Therefore these statements and the incrementing and condition testing of $i$ take $O(n)$ time.
- Now, observe the inner while loop during $i$ repetitions of the for loop. The statement done with a true condition is done at most once, since it causes an exit from the loop. Let us say that $t(i)$ is the number of times statement Pop a value from stack N is executed. So it becomes clear that while not (Stack N is empty or done with processing) is tested maximum $t(i) + 1$ times.
- Adding the running time of all the operations in the while loop, we get:
  \[
  \sum_{i=0}^{n-1} t(i) + 1
  \]
  - An element once popped from the stack N is never pushed back into it. Therefore, 
  \[
  \sum_{i=1}^{n-1} t(i)
  \]
  So, the running time of all the statements in the while loop is $O(n)$.

The running time of all the steps in the algorithm is calculated by adding the time taken by all these steps. The runtime of each step is $O(\eta)$. Hence the running time complexity of this algorithm is $O(\eta)$.

[11]

Runtime memory management

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack.

Forth uses two stacks, one for argument passing and one for subroutine return addresses. The use of a return stack is extremely commonplace, but the somewhat unusual use of an argument stack for a human-readable programming language is the reason Forth is referred to as a stack-based language.

Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine.

Almost all calling conventions -- computer runtime memory environments—use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. The functions follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements (or their equivalents) and is not manipulated directly by the programmer.

Some programming languages use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, and is deallocated when the procedure exits. The C programming language is typically implemented in this way. Using the same stack for both data and procedure calls has important security implications (see below) of which a programmer must be aware in order to avoid introducing serious security bugs into a program.
Security

Some computing environments use stacks in ways that may make them vulnerable to security breaches and attacks. Programmers working in such environments must take special care to avoid the pitfalls of these implementations.

For example, some programming languages use a common stack to store both data local to a called procedure and the linking information that allows the procedure to return to its caller. This means that the program moves data into and out of the same stack that contains critical return addresses for the procedure calls. If data is moved to the wrong location on the stack, or an oversized data item is moved to a stack location that is not large enough to contain it, return information for procedure calls may be corrupted, causing the program to fail.

Malicious parties may attempt a stack smashing attack that takes advantage of this type of implementation by providing oversized data input to a program that does not check the length of input. Such a program may copy the data in its entirety to a location on the stack, and in so doing it may change the return addresses for procedures that have called it. An attacker can experiment to find a specific type of data that can be provided to such a program such that the return address of the current procedure is reset to point to an area within the stack itself (and within the data provided by the attacker), which in turn contains instructions that carry out unauthorized operations.

This type of attack is a variation on the buffer overflow attack and is an extremely frequent source of security breaches in software, mainly because some of the most popular programming languages (such as C) use a shared stack for both data and procedure calls, and do not verify the length of data items. Frequently programmers do not write code to verify the size of data items, either, and when an oversized or undersized data item is copied to the stack, a security breach may occur.

References

[8] Data structures, Algorithms and Applications in C++ by Sartaj Sahni

• Stack implementation on goodsoft.org.ua (http://goodsoft.org.ua/en/data_struct/stack.html)
• A Templated Stack Data Structure Example at AssignmentExpert.com (http://www.assessmentexpert.com/blog/)
Further reading


External links

- Stack Machines - the new wave (http:/:www.ece.cmu.edu/~koopman/stack_computers/index.html)
- Bounding stack depth (http://www.cs.utah.edu/~regehr/stacktool)
- Libsafe - Protecting Critical Elements of Stacks (http://research.avayalabs.com/project/libsafe/)
- VBScript implementation of stack, queue, deque, and Red-Black Tree (http://www.ludvikjerabek.com/downloads.html)
- Stack Size Analysis for Interrupt-driven Programs (http://www.cs.ucla.edu/~palsberg/paper/sas03.pdf) (322 KB)
- Pointers to stack visualizations (http://web-cat.cs.vt.edu/AlgovizWiki/Stacks)

Queue (abstract data type)

A **queue** (prONounced English pronunciation: /ˈkjuː/ kew) is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.
Operations

Common operations from the C++ Standard Template Library include the following:

bool empty()
    Returns True if the queue is empty, and False otherwise.

T & front()
    Returns a reference to the value at the front of a non-empty queue. There is also a constant version of this function, const T & front().

void pop()
    Removes the item at the front of a non-empty queue.

void push(const T &foo)
    Inserts the argument foo at the back of the queue.

size_type size()
    Returns the total number of elements in the queue.

Representing a queue

In each of the cases, the customer or object at the front of the line was the first one to enter, while at the end of the line is the last to have entered. Every time a customer finishes paying for their items (or a person steps off the escalator, or the machine part is removed from the assembly line, etc.) that object leaves the queue from the front. This represents the queue “dequeue” function. Every time another object or customer enters the line to wait, they join the end of the line and represent the “enqueue” function. The queue “size” function would return the length of the line, and the “empty” function would return true only if there was nothing in the line.

Queue implementation

Theoretically, one characteristic of a queue is that it does not have a specific capacity. Regardless of how many elements are already contained, a new element can always be added. It can also be empty, at which point removing an element will be impossible until a new element has been added again.

Fixed length arrays are limited in capacity, and inefficient because items need to be copied towards the head of the queue. However conceptually they are simple and work with early languages such as FORTRAN and BASIC which did not have pointers or objects. Most modern languages with objects or pointers can implement or come with libraries for dynamic lists. Such data structures may have not specified fixed capacity limit besides memory constraints. Queue overflow results from trying to add an element onto a full queue and queue underflow happens when trying to remove an element from an empty queue.

A bounded queue is a queue limited to a fixed number of items.

There are several efficient implementations of FIFO queues. An efficient implementation is one that can perform the operations -- enqueuing and dequeuing -- in O(1) time.

• Linked list
  • A doubly linked list has O(1) insertion and deletion at both ends, so is a natural choice for queues.
  • A regular singly linked list only has efficient insertion and deletion at one end. However, a small modification -- keeping a pointer to the last node in addition to the first one -- will enable it to implement an efficient queue.
  • A deque implemented using a modified dynamic array
Queues and programming languages

Some languages, like Perl and Ruby, already have operations for pushing and popping an array from both ends, so one can use `push` and `shift` functions to enqueue and dequeue a list (or, in reverse, one can use `unshift` and `pop`), although in some cases these operations are not efficient.

C++’s Standard Template Library provides a "queue" templated class which is restricted to only push/pop operations. Since J2SE5.0, Java's library contains a `Queue` interface that specifies queue operations; implementing classes include `LinkedList` and (since J2SE 1.6) `ArrayDeque`. PHP has an `SplQueue` class.

References

General


Citations


External links

- STL Quick Reference (http://www.halpernwightsoftware.com/stdlib-scratch/quickref.html#containers14)
- VBScript implementation of stack, queue, deque, and Red-Black Tree (http://www.ludvikjerabek.com/downloads.html)

Double-ended queue

In computer science, a double-ended queue (deque, often abbreviated to deque, pronounced deck) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

Naming conventions

Deque is sometimes written dequeue, but this use is generally deprecated in technical literature or technical writing because dequeue is also a verb meaning "to remove from a queue". Nevertheless, several libraries and some writers, such as Aho, Hopcroft, and Ullman in their textbook Data Structures and Algorithms, spell it dequeue. John Mitchell, author of Concepts in Programming Languages, also uses this terminology. DEQ and DQ are also used.

Distinctions and sub-types

This differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

• An input-restricted deque is one where deletion can be made from both ends, but insertion can only be made at one end.
• An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of deques, and can be implemented using deques.

Operations

The following operations are possible on a deque:

<table>
<thead>
<tr>
<th>operation</th>
<th>Ada</th>
<th>C++</th>
<th>Java</th>
<th>Perl</th>
<th>PHP</th>
<th>Python</th>
<th>Ruby</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert element at back</td>
<td>Append</td>
<td>push_back</td>
<td>offerLast</td>
<td>push</td>
<td>array_push</td>
<td>append</td>
<td>push</td>
<td>push</td>
</tr>
<tr>
<td>insert element at front</td>
<td>Prepend</td>
<td>push_front</td>
<td>offerFirst</td>
<td>unshift</td>
<td>array_unshift</td>
<td>appendleft</td>
<td>unshift</td>
<td>unshift</td>
</tr>
<tr>
<td>remove last element</td>
<td>Delete_Last</td>
<td>pop_back</td>
<td>pollLast</td>
<td>pop</td>
<td>array_pop</td>
<td>pop</td>
<td>pop</td>
<td>pop</td>
</tr>
<tr>
<td>remove first element</td>
<td>Delete_First</td>
<td>pop_front</td>
<td>pollFirst</td>
<td>shift</td>
<td>array_shift</td>
<td>popleft</td>
<td>shift</td>
<td>shift</td>
</tr>
<tr>
<td>examine last element</td>
<td>Last_Element</td>
<td>back</td>
<td>peekLast</td>
<td>$array[-1]</td>
<td>end</td>
<td>&lt;obj&gt;[−1]</td>
<td>last</td>
<td>&lt;obj&gt;[&lt;obj&gt;.length − 1]</td>
</tr>
<tr>
<td>examine first element</td>
<td>First_Element</td>
<td>front</td>
<td>peekFirst</td>
<td>$array[0]</td>
<td>reset</td>
<td>&lt;obj&gt;[0]</td>
<td>first</td>
<td>&lt;obj&gt;[0]</td>
</tr>
</tbody>
</table>
Implementations

There are at least two common ways to efficiently implement a deque: with a modified dynamic array or with a doubly linked list.

The dynamic array approach uses a variant of a dynamic array that can grow from both ends, sometimes called array deques. These array deques have all the properties of a dynamic array, such as constant time random access, good locality of reference, and inefficient insertion/removal in the middle, with the addition of amortized constant time insertion/removal at both ends, instead of just one end. Three common implementations include:

• Storing deque contents in a circular buffer, and only resizing when the buffer becomes completely full. This decreases the frequency of resizings, but requires an expensive branch instruction for indexing.
• Allocating deque contents from the center of the underlying array, and resizing the underlying array when either end is reached. This approach may require more frequent resizings and waste more space, particularly when elements are only inserted at one end.
• Storing contents in multiple smaller arrays, allocating additional arrays at the beginning or end as needed.

Indexing is implemented by keeping a dynamic array containing pointers to each of the smaller arrays.

Language support

Ada's containers provides the generic packages Ada.Containers.Vectors and Ada.Containers.Doubly_Linked_Lists, for the dynamic array and linked list implementations, respectively.

C++'s Standard Template Library provides the class templates std::deque and std::list, for the multiple array and linked list implementations, respectively.

As of Java 6, Java's Collections Framework provides a new Deque interface that provides the functionality of insertion and removal at both ends. It is implemented by classes such as ArrayDeque (also new in Java 6) and LinkedList, providing the dynamic array and linked list implementations, respectively. However, the ArrayDeque, contrary to its name, does not support random access.

As of PHP 5.3, PHP's SPL extension contains the 'SplDoublyLinkedList' class that can be used to implement Deque datastructures. Previously to make a Deque structure the array functions array_shift/unshift/pop/push had to be used instead.

GHC's Data.Sequence module implements an efficient, functional deque structure in Haskell. The implementation uses 2-3 finger trees annotated with sizes. There are other (fast) possibilities to implement purely functional (thus also persistent) double queues (most using heavily lazy evaluation), see references [3], [4], [5].

Complexity

• In a doubly linked list implementation and assuming no allocation/deallocation overhead, the time complexity of all deque operations is O(1). Additionally, the time complexity of insertion or deletion in the middle, given an iterator, is O(1); however, the time complexity of random access by index is O(n).
• In a growing array, the amortized time complexity of all deque operations is O(1). Additionally, the time complexity of random access by index is O(1); but the time complexity of insertion or deletion in the middle is O(n).
Applications

One example where a deque can be used is the A-Steal job scheduling algorithm.[6] This algorithm implements task scheduling for several processors. A separate deque with threads to be executed is maintained for each processor. To execute the next thread, the processor gets the first element from the deque (using the "remove first element" deque operation). If the current thread forks, it is put back to the front of the deque ("insert element at front") and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. its deque is empty), it can "steal" a thread from another processor: it gets the last element from the deque of another processor ("remove last element") and executes it.

References


External links

- Code Project: An In-Depth Study of the STL Deque Container (http://www.codeproject.com/KB/stl/vector_vs_deque.aspx)
- Diagram of a typical STL deque implementation (http://pages.cpsc.ucalgary.ca/~kremer/STL/1024x768/deque.html)
- Deque implementation in C (http://www.martinbroadhurst.com/articles/deque.html)
- VBScript implementation of stack, queue, deque, and Red-Black Tree (http://www.ludvikjerabek.com/downloads.html)
Circular buffer

A circular buffer, cyclic buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

Uses

An example that could possibly use an overwriting circular buffer is with multimedia. If the buffer is used as the bounded buffer in the producer-consumer problem then it is probably desired for the producer (e.g., an audio generator) to overwrite old data if the consumer (e.g., the sound card) is unable to momentarily keep up. Another example is the digital waveguide synthesis method which uses circular buffers to efficiently simulate the sound of vibrating strings or wind instruments.

The "prized" attribute of a circular buffer is that it does not need to have its elements shuffled around when one is consumed. (If a non-circular buffer were used then it would be necessary to shift all elements when one is consumed.) In other words, the circular buffer is well suited as a FIFO buffer while a standard, non-circular buffer is well suited as a LIFO buffer.

Circular buffering makes a good implementation strategy for a Queue that has fixed maximum size. Should a maximum size be adopted for a queue, then a circular buffer is a completely ideal implementation; all queue operations are constant time. However, expanding a circular buffer requires shifting memory, which is comparatively costly. For arbitrarily expanding queues, a Linked list approach may be preferred instead.

How it works

A circular buffer first starts empty and of some predefined length. For example, this is a 7-element buffer:

Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a circular buffer):

Then assume that two more elements are added — 2 & 3 — which get appended after the 1:

If two elements are then removed from the buffer, the oldest values inside the buffer are removed. The two elements removed, in this case, are 1 & 2 leaving the buffer with just a 3:

If the buffer has 7 elements then it is completely full:
A consequence of the circular buffer is that when it is full and a subsequent write is performed, then it starts overwriting the oldest data. In this case, two more elements — A & B — are added and they *overwrite* the 3 & 4:

![Circular buffer with elements overwritten](image)

Alternatively, the routines that manage the buffer could prevent overwriting the data and return an error or raise an exception. Whether or not data is overwritten is up to the semantics of the buffer routines or the application using the circular buffer.

Finally, if two elements are now removed then what would be returned is **not** 3 & 4 but 5 & 6 because A & B overwrote the 3 & the 4 yielding the buffer with:

![Circular buffer with elements removed](image)

### Circular buffer mechanics

What is not shown in the example above is the mechanics of how the circular buffer is managed.

#### Start / End Pointers

Generally, a circular buffer requires three pointers:

- one to the actual buffer in memory
- one to point to the start of valid data
- one to point to the end of valid data

Alternatively, a fixed-length buffer with two integers to keep track of indices can be used in languages that do not have pointers.

Taking a couple of examples from above. (While there are numerous ways to label the pointers and exact semantics can vary, this is one way to do it.)

This image shows a partially-full buffer:

![Partially-full buffer](image)

This image shows a full buffer with two elements having been overwritten:

![Full buffer with elements overwritten](image)

What to note about the second one is that after each element is overwritten then the start pointer is incremented as well.
Difficulties

Full / Empty Buffer Distinction

A small disadvantage of relying on pointers or relative indices of the start and end of data is, that in the case the buffer is entirely full, both pointers point to the same element:

![Diagram showing full buffer with pointers at the same location]

This is exactly the same situation as when the buffer is empty:

![Diagram showing empty buffer with pointers at the same location]

To solve this confusion there are a number of solutions:

- Always keep one slot open.
- Use a fill count to distinguish the two cases.
- Use read and write counts to get the fill count from.
- Use absolute indices.

Always Keep One Slot Open

This simple solution always keeps one slot unallocated. A full buffer has at most \((\text{size} - 1)\) slots. If both pointers are pointing at the same location, the buffer is empty. If the end (write) pointer, plus one, equals the start (read) pointer, then the buffer is full.

The advantages are:

- Very simple and robust.
- You need only the two pointers.

The disadvantages are:

- You can never use the entire buffer.
- You might only be able to access one element at a time, since you won't easily know how many elements are next to each other in memory..

An example implementation in C: (Keep One Slot Open)

```c
#include <stdio.h>
#include <string.h>
#include <malloc.h>

/*! * Circular Buffer Example (Keep one slot open) * Compile: gcc cbuf.c -o cbuf.exe */

/**< Buffer Size */
#define BUFFER_SIZE 10
#define NUM_OF_ELEMS (BUFFER_SIZE-1)

/**< Circular Buffer Types */
typedef unsigned char INT8U;
typedef INT8U KeyType;
```
typedef struct
{
    INT8U writePointer; /**< write pointer */
    INT8U readPointer;  /**< read pointer */
    INT8U size;         /**< size of circular buffer */
    KeyType keys[0];   /**< Element of circular buffer */
} CircularBuffer;

/** Init Circular Buffer */
CircularBuffer* CircularBufferInit(CircularBuffer** pQue, int size)
{
    int sz = size*sizeof(KeyType)+sizeof(CircularBuffer);
    *pQue = (CircularBuffer*) malloc(sz);
    if(*pQue)
    {
        printf("Init CircularBuffer: keys[%d] (%d)\n", size, sz);
        (*pQue)->size=size;
        (*pQue)->writePointer = 0;
        (*pQue)->readPointer = 0;
    }
    return *pQue;
}

inline int CircularBufferIsFull(CircularBuffer* que)
{
    return ((que->writePointer + 1) % que->size) == que->readPointer;
}

inline int CircularBufferIsEmpty(CircularBuffer* que)
{
    return (que->readPointer == que->writePointer);
}

inline int CircularBufferEnque(CircularBuffer* que, KeyType k)
{
    int isFull = CircularBufferIsFull(que);
    if(!isFull)
    {
        que->keys[que->writePointer] = k;
        que->writePointer++;
        que->writePointer %= que->size;
    }
    return isFull;
}

inline int CircularBufferDeque(CircularBuffer* que, KeyType* pK)
{
int isEmpty = CircularBufferIsEmpty(que);
if(!isEmpty)
{
    *pK = que->keys[que->readPointer];
    que->readPointer++;
    que->readPointer %= que->size;
}
return(isEmpty);

inline int CircularBufferPrint(CircularBuffer* que)
{
    int i=0;
    int isEmpty = CircularBufferIsEmpty(que);
    int isFull = CircularBufferIsFull(que);
    printf("\n==Q: w:%d r:%d f:%d e:%d\n",
        que->writePointer, que->readPointer,  isFull, isEmpty);
    for(i=0; i< que->size; i++)
    {
        printf("%d ", que->keys[i]);
    }
    printf("\n");
    return(isEmpty);
}

int main(int argc, char *argv[])
{
    CircularBuffer* que;
    KeyType a = 101;
    int isEmpty, i;

    CircularBufferInit(&que, BUFFER_SIZE);
    CircularBufferPrint(que);

    for(i=1; i<=3; i++)
    {
        a=10*i;
        printf("\n\nTest: Insert %d-%d\n", a, a+NUM_OF_ELEMS-1);
        while(! CircularBufferEnque(que, a++));
        //CircularBufferPrint(que);
        printf("RX%d:\n", i);
        a=0;
        isEmpty = CircularBufferDeque(que, &a);
        while (!isEmpty)
        {
            printf("%02d ", a);
        }
    }
}
a=0;
isEmpty = CircularBufferDeque(que, &a);

//CircularBufferPrint(que);

free(que);
return 0;

An example implementation in C: (Use all slots) (but is dangerous - an attempt to insert items on a full queue will yield success, but will, in fact, overwrite the queue)

#include <stdio.h>
#include <string.h>
#include <malloc.h>

/*! Circular Buffer Example
* Compile: gcc cbuf.c -o cbuf.exe */

/**< Buffer Size */
#define BUFFER_SIZE 16

/**< Circular Buffer Types */
typedef unsigned char INT8U;
typedef INT8U KeyType;
typedef struct
{
  INT8U writePointer;  /**< write pointer */
  INT8U readPointer;  /**< read pointer */
  INT8U size;  /**< size of circular buffer */
  KeyType keys[0];  /**< Element of circular buffer */
} CircularBuffer;

/**< Init Circular Buffer */
CircularBuffer* CircularBufferInit(CircularBuffer** pQue, int size)
{
  int sz = size*sizeof(KeyType)+sizeof(CircularBuffer);
  *pQue = (CircularBuffer*) malloc(sz);
  if(*pQue)
  {
    printf("Init CircularBuffer: keys[%d] (%d)\n", size, sz);
    (*pQue)->size=size;
    (*pQue)->writePointer = 0;
    (*pQue)->readPointer = 0;
  }
}
return *pQue;

inline int CircularBufferIsFull(CircularBuffer* que)
{
    return ((que->writePointer + 1) % que->size == que->readPointer);
}

inline int CircularBufferIsEmpty(CircularBuffer* que)
{
    return (que->readPointer == que->writePointer);
}

inline int CircularBufferEnque(CircularBuffer* que, KeyType k)
{
    int isFull = CircularBufferIsFull(que);
    que->keys[que->writePointer] = k;
    que->writePointer++;
    que->writePointer %= que->size;
    return isFull;
}

inline int CircularBufferDeque(CircularBuffer* que, KeyType* pK)
{
    int isEmpty = CircularBufferIsEmpty(que);
    *pK = que->keys[que->readPointer];
    que->readPointer++;
    que->readPointer %= que->size;
    return isEmpty;
}

int main(int argc, char *argv[])
{
    CircularBuffer* que;
    KeyType a = 0;
    int isEmpty;
    CircularBufferInit(&que, BUFFER_SIZE);

    while(! CircularBufferEnque(que, a++));

    do {
        isEmpty = CircularBufferDeque(que, &a);
        printf("%02d ", a);
    } while (!isEmpty);

    printf("\n");
    free(que);
    return 0;
Use a Fill Count

The second simplest solution is to use a fill count. The fill count is implemented as an additional variable which keeps the number of readable items in the buffer. This variable has to be increased if the write (end) pointer is moved, and to be decreased if the read (start) pointer is moved.

In the situation if both pointers pointing at the same location, you consider the fill count to distinguish if the buffer is empty or full.

- Note: When using semaphores in a Producer-consumer model, the semaphores act as a fill count.

The advantages are:
- Simple.
- Needs only one additional variable.

The disadvantage is:
- You need to keep track of a third variable. This can require complex logic, especially if you are working with different threads.

Alternately, you can replace the second pointer with the fill count and generate the second pointer as required by incrementing the first pointer by the fill count, modulo buffer size.

The advantages are:
- Simple.
- No additional variables.

The disadvantage is:
- Additional overhead when generating the write pointer.

Read / Write Counts

Another solution is to keep counts of the number of items written to and read from the circular buffer. Both counts are stored in signed integer variables with numerical limits larger than the number of items that can be stored and are allowed to wrap freely.

The unsigned difference (write_count - read_count) always yields the number of items placed in the buffer and not yet retrieved. This can indicate that the buffer is empty, partially full, completely full (without waste of a storage location) or in a state of overrun.

The advantage is:
- The source and sink of data can implement independent policies for dealing with a full buffer and overrun while adhering to the rule that only the source of data modifies the write count and only the sink of data modifies the read count. This can result in elegant and robust circular buffer implementations even in multi-threaded environments.

The disadvantage is:
- You need two additional variables.
**Record last operation**

Another solution is to keep a flag indicating whether the most recent operation was a read or a write. If the two pointers are equal, then the flag will show whether the buffer is full or empty: if the most recent operation was a write, the buffer must be full, and conversely if it was a read, it must be empty.

The advantages are:
- Only a single bit needs to be stored (which may be particularly useful if the algorithm is implemented in hardware)
- The test for full/empty is simple

The disadvantage is:
- You need an extra variable

**Absolute indices**

If indices are used instead of pointers, indices can store read/write counts instead of the offset from start of the buffer. This is similar to the above solution, except that there are no separate variables, and relative indices are obtained on the fly by division modulo the buffer's length.

The advantage is:
- No extra variables are needed.

The disadvantages are:
- Every access needs an additional modulo operation.
- If counter wrap is possible, complex logic can be needed if the buffer's length is not a divisor of the counter's capacity.

On binary computers, both of these disadvantages disappear if the buffer's length is a power of two—at the cost of a constraint on possible buffers lengths.

**Multiple Read Pointers**

A little bit more complex are multiple read pointers on the same circular buffer. This is useful if you have \( n \) threads, which are reading from the same buffer, but one thread writing to the buffer.

**Chunked Buffer**

Much more complex are different chunks of data in the same circular buffer. The writer is not only writing elements to the buffer, it also assigns these elements to chunks.

The reader should not only be able to read from the buffer, it should also get informed about the chunk borders.

Example: The writer is reading data from small files, writing them into the same circular buffer. The reader is reading the data, but needs to know when and which file is starting at a given position.

**Optimization**

A circular-buffer implementation may be optimized by mapping the underlying buffer to two contiguous regions of virtual memory. (Naturally, the underlying buffer's length must then equal some multiple of the system's page size.) Reading from and writing to the circular buffer may then be carried out with greater efficiency by means of direct memory access; those accesses which fall beyond the end of the first virtual-memory region will automatically wrap around to the beginning of the underlying buffer. When the read offset is advanced into the second virtual-memory region, both offsets—read and write—are decremented by the length of the underlying buffer.
Circular buffer

Optimized POSIX Implementation

```
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>

#define report_exceptional_condition() abort ()

struct ring_buffer
{
    void *address;

    unsigned long count_bytes;
    unsigned long write_offset_bytes;
    unsigned long read_offset_bytes;
};

// Warning: order should be at least 12 for Linux
void ring_buffer_create (struct ring_buffer *buffer, unsigned long order)
{
    char path[] = "/dev/shm/ring-buffer-XXXXXX";
    int file_descriptor;
    void *address;
    int status;

    file_descriptor = mkstemp (path);
    if (file_descriptor < 0)
        report_exceptional_condition ();

    status = unlink (path);
    if (status)
        report_exceptional_condition ();

    buffer->count_bytes = 1UL << order;
    buffer->write_offset_bytes = 0;
    buffer->read_offset_bytes = 0;

    status = ftruncate (file_descriptor, buffer->count_bytes);
    if (status)
        report_exceptional_condition ();

    buffer->address = mmap (NULL, buffer->count_bytes << 1, PROT_NONE,
                              MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    if (buffer->address == MAP_FAILED)
        report_exceptional_condition ();
```
Circular buffer

```c
address =
    mmap (buffer->address, buffer->count_bytes, PROT_READ | PROT_WRITE,
          MAP_FIXED | MAP_SHARED, file_descriptor, 0);

if (address != buffer->address)
    report_exceptional_condition();

address = mmap (buffer->address + buffer->count_bytes,
                buffer->count_bytes, PROT_READ | PROT_WRITE,
                MAP_FIXED | MAP_SHARED, file_descriptor, 0);

if (address != buffer->address + buffer->count_bytes)
    report_exceptional_condition();

status = close (file_descriptor);
if (status)
    report_exceptional_condition();
}

void
ring_buffer_free (struct ring_buffer *buffer)
{
    int status;

    status = munmap (buffer->address, buffer->count_bytes << 1);
    if (status)
        report_exceptional_condition();
}

void *
ring_buffer_write_address (struct ring_buffer *buffer)
{
    /*** void pointer arithmetic is a constraint violation. ***/
    return buffer->address + buffer->write_offset_bytes;
}

void
ring_buffer_write_advance (struct ring_buffer *buffer,
                           unsigned long count_bytes)
{
    buffer->write_offset_bytes += count_bytes;
}

void *
ring_buffer_read_address (struct ring_buffer *buffer)
{
    return buffer->address + buffer->read_offset_bytes;
```
```c
void ring_buffer_read_advance (struct ring_buffer *buffer, 
    unsigned long count_bytes) 
{
    buffer->read_offset_bytes += count_bytes;

    if (buffer->read_offset_bytes >= buffer->count_bytes) 
    {
        buffer->read_offset_bytes -= buffer->count_bytes;
        buffer->write_offset_bytes -= buffer->count_bytes;
    }
}

unsigned long ring_buffer_count_bytes (struct ring_buffer *buffer) 
{
    return buffer->write_offset_bytes - buffer->read_offset_bytes;
}

unsigned long ring_buffer_count_free_bytes (struct ring_buffer *buffer) 
{
    return buffer->count_bytes - ring_buffer_count_bytes (buffer);
}

void ring_buffer_clear (struct ring_buffer *buffer) 
{
    buffer->write_offset_bytes = 0;
    buffer->read_offset_bytes = 0;
}
```
public:
    Queue()
    {
        qsize= 32;
        qbuf= new T[qsize];
        head= tail= 0;
    }

    Queue(const int size): qsize(size), qbuf(0), head(0), tail(0)
    {
        if ((size <= 0) || (size & (size - 1)))
        {
            throw "Value is not power of two";
        }

        qsize= size;
        qbuf= new T[qsize];
        head= tail= 0;
    }

    ~Queue()
    {
        Free();
    }

    void Enqueue(const T &p)
    {
        if (IsFull())
        {
            throw "Queue is full";
        }

        qbuf[tail]= p;
        tail= (tail + 1) & (qsize - 1);
    }

    // Retrieve the item from the queue
    void Dequeue(T &p)
    {
        if (IsEmpty())
{ 
    throw "Queue is empty";
}

p = qbuf[head];
head = (head + 1) & (qsize - 1);

// Get i-element with not delete
void Peek(const int i, T &p) const
{
    int j = 0;
    int k = head;
    while (k != tail)
    {
        if (j == i) break;
        j++;

        k = (k + 1) & (qsize - 1);
    }
    if (k == tail) throw "Out of range";
    p = qbuf[k];
}

// Size must by: 1, 2, 4, 8, 16, 32, 64, ..
void Resize(const int size)
{
    if ((size <= 0) || (size & (size - 1))
    {
        throw "Value is not power of two";
    }

    Free();
    qsize = size;
    qbuf = new T[qsize];
    head = tail = 0;
}

inline void Clear(void) { head = tail = 0; }

inline int GetCapacity(void) const { return (qsize - 1); }

// Count elements
inline int GetBusy(void) const { return ((head > tail) ? qsize : 0) + tail - head; }

// true - if queue if empty
inline bool IsEmpty(void) const { return (head == tail); }

// true - if queue is full
inline bool IsFull(void) const { return (((tail + 1) & (qsize - 1)) == head); }

// Use:
Queue <int> Q;
Q.Enqueue(5);
Q.Enqueue(100);
Q.Enqueue(13);
int len = Q.GetBusy();
int val;
Q.Dequeue(val);

External links

- Boost: Templated Circular Buffer Container [1]
- http://www.dspguide.com/ch28/2.htm

References

Dictionaries

Associative array

In computer science, an associative array (also called a map or a dictionary) is an abstract data type composed of a collection of (key,value) pairs, such that each possible key appears at most once in the collection.

Operations associated with this data type allow:

- the addition of pairs to the collection,
- the removal of pairs from the collection,
- the modification of the values of existing pairs, and
- the lookup of the value associated with a particular key.\[1\] [2]

The dictionary problem is the task of designing a data structure that implements an associative array. A standard solution to the dictionary problem is a hash table; in some cases it is also possible to solve the problem using directly addressed arrays, binary search trees, or other more specialized structures.\[1\] [2] [3]

Many programming languages include associative arrays as primitive data types, and they are available in software libraries for many others. Content-addressable memory is a form of direct hardware-level support for associative arrays.

Associative arrays have many applications including such fundamental programming patterns as memoization and the decorator pattern.\[4\]

Operations

In an associative array, the association between a key and a value is often known as a "binding", and the same word "binding" may also be used to refer to the process of creating a new association.

The operations that are usually defined for an associative array are:\[1\] [2]

- **Add** or **insert**: add a new (key,value) pair to the collection, binding the new key to its new value. The arguments to this operation are the key and the value.
- **Reassign**: replace the value in one of the (key,value) pairs that are already in the collection, binding an old key to a new value. As with an insertion, the arguments to this operation are the key and the value.
- **Remove** or **delete**: remove a (key,value) pair from the collection, unbinding a given key from its value. The argument to this operation is the key.
- **Lookup**: find the value (if any) that is bound to a given key. The argument to this operation is the key, and the value is returned from the operation. If no value is found, some associative array implementations raise an exception.

In addition, associative arrays may also include other operations such as determining the number of bindings or constructing an iterator to loop over all the bindings. Usually, for such an operation, the order in which the bindings are returned may be arbitrary.

A multimap generalizes an associative array by allowing multiple values to be associated with a single key.\[5\] A bidirectional map is a related abstract data type in which the bindings operate in both directions: each value must be associated with a unique key, and a second lookup operation takes a value as argument and looks up the key associated with that value.
**Example**

Suppose that the set of loans made by a library is to be represented in a data structure. Each book in a library may be checked out only by a single library patron at a time. However, a single patron may be able to check out multiple books. Therefore, the information about which books are checked out to which patrons may be represented by an associative array, in which the books are the keys and the patrons are the values. For instance (using the notation from Python in which a binding is represented by placing a colon between the key and the value), the current checkouts may be represented by an associative array

```
{
    "Great Expectations": "John",
    "Pride and Prejudice": "Alice",
    "Wuthering Heights": "Alice"
}
```

A lookup operation with the key "Great Expectations" in this array would return the name of the person who checked out that book, John. If John returns his book, that would cause a deletion operation in the associative array, and if Pat checks out another book, that would cause an insertion operation, leading to a different state:

```
{
    "Pride and Prejudice": "Alice",
    "The Brothers Karamazov": "Pat",
    "Wuthering Heights": "Alice"
}
```

In this new state, the same lookup as before, with the key "Great Expectations", would raise an exception, because this key is no longer present in the array.

**Implementation**

For dictionaries with very small numbers of bindings, it may make sense to implement the dictionary using an association list, a linked list of bindings. With this implementation, the time to perform the basic dictionary operations is linear in the total number of bindings; however, it is easy to implement and the constant factors in its running time are small. Another very simple implementation technique, usable when the keys are restricted to a narrow range of integers, is direct addressing into an array: the value for a given key \( k \) is stored at the array cell \( A[k] \), or if there is no binding for \( k \) then the cell stores a special sentinel value that indicates the absence of a binding. As well as being simple, this technique is fast: each dictionary operation takes constant time. However, the space requirement for this structure is the size of the entire keyspace, making it impractical unless the keyspace is small.

The most frequently used general purpose implementation of an associative array is with a hash table: an array of bindings, together with a hash function that maps each possible key into an array index. The basic idea of a hash table is that the binding for a given key is stored at the position given by applying the hash function to that key, and that lookup operations are performed by looking at that cell of the array and using the binding found there. However, hash table based dictionaries must be prepared to handle collisions that occur when two keys are mapped by the hash function to the same index, and many different collision resolution strategies have been developed for dealing with this situation, often based either on open addressing (looking at a sequence of hash table indices instead of a single index, until finding either the given key or an empty cell) or on hash chaining (storing a small association list instead of a single binding in each hash table cell).

Dictionaries may also be stored in binary search trees or in data structures specialized to a particular type of keys such as radix trees, tries, Judy arrays, or van Emde Boas trees, but these implementation methods are less efficient than hash tables as well as placing greater restrictions on the types of data that they can handle. The advantages of
these alternative structures come from their ability to handle operations beyond the basic ones of an associative array, such as finding the binding whose key is the closest to a queried key, when the query is not itself present in the set of bindings.

**Language support**

Associative arrays can be implemented in any programming language as a package and many language systems provide them as part of their standard library. In some languages, they are not only built into the standard system, but have special syntax, often using array-like subscripting.

Built-in syntactic support for associative arrays was introduced by SNOBOL4, under the name "table". MUMPS made multi-dimensional associative arrays, optionally persistent, its key data structure. SETL supported them as one possible implementation of sets and maps. Most modern scripting languages, starting with AWK and including Perl, Tcl, JavaScript, Python, Ruby, and Lua, support associative arrays as a primary container type. In many more languages, they are available as library functions without special syntax.

In Smalltalk, Objective-C, .NET, Python, and REALbasic they are called dictionaries; in Perl and Ruby they are called hashes; in C++, Java, and Go they are called maps (see map (C++), unordered_map (C++), and Map); in Common Lisp and Windows PowerShell, they are called hash tables (since both typically use this implementation). In PHP, all arrays can be associative, except that the keys are limited to integers and strings. In JavaScript, all objects behave as associative arrays. In Lua, they are called tables, and are used as the primitive building block for all data structures. In Visual FoxPro, they are called Collections.

**References**


**External links**

Association list

In computer programming and particularly in Lisp, an association list, often referred to as an alist, is a linked list in which each list element (or node) comprises a key and a value. The association list is said to associate the value with the key. In order to find the value associated with a given key, each element of the list is searched in turn, starting at the head, until the key is found. Duplicate keys that appear later in the list are ignored. It is a simple way of implementing an associative array.

The disadvantage of association lists is that the time to search is $O(n)$, where $n$ is the length of the list. And unless the list is regularly pruned to remove elements with duplicate keys multiple values associated with the same key will increase the size of the list, and thus the time to search, without providing any compensatory advantage. One advantage is that a new element can be added to the list at its head, which can be done in constant time. For quite small values of $n$ it is more efficient in terms of time and space than more sophisticated strategies such as hash tables and trees.

In the early development of Lisp, association lists were used to resolve references to free variables in procedures.\[1\]

Many programming languages, including Lisp, Scheme, OCaml, and Haskell have functions for handling association lists in their standard library.

References

In computer science, a **hash table** or **hash map** is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number). Thus, a hash table implements an associative array. The hash function is used to transform the key into the index (the *hash*) of an array element (the *slot* or *bucket*) where the corresponding value is to be sought.

Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that *hash collisions*—different keys that map to the same hash value—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average (indeed, amortized\[3\]) cost per operation.\[4\] [5]

In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.
**Hash function**

At the heart of the hash table algorithm is a simple array of items; this is often simply called the *hash table*. Hash table algorithms calculate an index from the data item's key and use this index to place the data into the array. The implementation of this calculation is the hash function, \( f \):

\[
\text{index} = f(\text{key}, \text{arrayLength})
\]

The hash function calculates an *index* within the array from the data *key*. *arrayLength* is the size of the array. For assembly language or other low-level programs, a trivial hash function can often create an index with just one or two inline machine instructions.

**Choosing a good hash function**

A good hash function and implementation algorithm are essential for good hash table performance, but may be difficult to achieve. Poor hashing usually degrades hash table performance by a constant factor, but hashing is often only a small part of the overall computation.

A basic requirement is that the function should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions, and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests, e.g. a Pearson's chi-squared test for uniform distributions \[6\], \[7\].

The distribution needs to be uniform only for table sizes \( s \) that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of \( s \), the hash function needs to be uniform only when \( s \) is a power of two. On the other hand, some hashing algorithms provide uniform hashes only when \( s \) is a prime number.\[8\]

For open addressing schemes, the hash function should also avoid *clustering*, the mapping of two or more keys to consecutive slots. Such clustering may cause the lookup cost to skyrocket, even if the load factor is low and collisions are infrequent. The popular multiplicative hash\[4\] is claimed to have particularly poor clustering behavior.\[8\]

Cryptographic hash functions are believed to provide good hash functions for any table size \( s \), either by modulo reduction or by bit masking. They may also be appropriate, if there is a risk of malicious users trying to sabotage a network service by generating a large number of collisions in the server's hash tables. However, these presumed qualities are hardly worth their much larger computational cost and algorithmic complexity, and the risk of sabotage can be avoided by cheaper methods (such as applying a secret salt to the data, or using a universal hash function).

Some authors claim that good hash functions should have the avalanche effect; that is, a single-bit change in the input key should affect, on average, half the bits in the output. Some popular hash functions do not have this property.

**Perfect hash function**

If all keys are known ahead of time, a perfect hash function can be used to create a perfect hash table that has no collisions. If minimal perfect hashing is used, every location in the hash table can be used as well.

Perfect hashing allows for constant time lookups in the worst case. This is in contrast to most chaining and open addressing methods, where the time for lookup is low on average, but may be very large (proportional to the number of entries) for some sets of keys.
Collision resolution

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,500 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is a 95% chance of at least two of the keys being hashed to the same slot. Therefore, most hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

Load factor

The performance of most collision resolution methods does not depend directly on the number \( n \) of stored entries, but depends strongly on the table's load factor, the ratio \( n/s \) between \( n \) and the size \( s \) of its array of buckets. Sometimes this is referred to as the fill factor, as it represents the portion of the \( s \) buckets in the structure that are filled with one of the \( n \) stored entries. With a good hash function, the average lookup cost is nearly constant as the load factor increases from 0 up to 0.7 (about 2/3 full) or so. Beyond that point, the probability of collisions and the cost of handling them increases.

On the other hand, as the load factor approaches zero, the proportion of the unused areas in the hash table increases but there is not necessarily any improvement in the search cost, resulting in wasted memory.

Separate chaining

In the strategy known as separate chaining, direct chaining, or simply chaining, each slot of the bucket array is a pointer to a linked list that contains the key-value pairs that hashed to the same location. Lookup requires scanning the list for an entry with the given key. Insertion requires adding a new entry record to either end of the list belonging to the hashed slot. Deletion requires searching the list and removing the element. (The technique is also called open hashing or closed addressing, which should not be confused with 'open addressing' or 'closed hashing'.)

Chained hash tables with linked lists are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods.

The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the average cost of a lookup depends only on the average number of keys per bucket—that is, on the load factor.

Chained hash tables remain effective even when the number of table entries \( n \) is much higher than the number of slots. Their performance degrades more gracefully (linearly) with the load factor. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.
For separate-chaining, the worst-case scenario is when all entries were inserted into the same bucket, in which case
the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the
lookup procedure may have to scan all its entries; so the worst-case cost is proportional to the number \( n \) of entries in
the table.

The bucket chains are often implemented as ordered lists, sorted by the key field; this choice approximately halves
the average cost of unsuccessful lookups, compared to an unordered list. However, if some keys are much more
likely to come up than others, an unordered list with move-to-front heuristic may be more effective. More
sophisticated data structures, such as balanced search trees, are worth considering only if the load factor is large
(about 10 or more), or if the hash distribution is likely to be very non-uniform, or if one must guarantee good
performance even in the worst-case. However, using a larger table and/or a better hash function may be even more
effective in those cases.

Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space
overhead of the `next` pointer in each entry record can be significant. An additional disadvantage is that traversing a
linked list has poor cache performance, making the processor cache ineffective.

### Separate chaining with list heads

Some chaining implementations store the first record of each chain in the slot array itself.\[5\] The purpose is to
increase cache efficiency of hash table access. To save memory space, such hash tables often have about as many
slots as stored entries, meaning that many slots have two or more entries.

### Separate chaining with other structures

Instead of a list, one can use any other data structure that supports the required operations. For example, by using a self-balancing tree, the theoretical worst-case time of common hash table operations (insertion, deletion, lookup) can be brought down to \( \mathcal{O}(\log n) \) rather than \( \mathcal{O}(n) \). However, this approach is only worth the trouble and extra memory cost if long delays must be
avoided at all costs (e.g. in a real-time application), or if one expects to have many entries hashed to the same slot
(e.g. if one expects extremely non-uniform or even malicious key distributions).

The variant called array hash table uses a dynamic array to store all the entries that hash to the same slot.\[9\] \[10\] \[11\] Each newly inserted entry gets appended to the end of the dynamic array that is assigned to the slot. The dynamic
array is resized in an exact-fit manner, meaning it is grown only by as many bytes as needed. Alternative techniques
such as growing the array by block sizes or pages were found to improve insertion performance, but at a cost in
space. This variation makes more efficient use of CPU caching and the translation lookaside buffer (TLB), because
slot entries are stored in sequential memory positions. It also dispenses with the `next` pointers that are required by
linked lists, which saves space. Despite frequent array resizing, space overheads incurred by operating system such
as memory fragmentation, were found to be small.

An elaboration on this approach is the so-called dynamic perfect hashing,\[12\] where a bucket that contains \( k \) entries is
organized as a perfect hash table with \( k^2 \) slots. While it uses more memory (\( n^2 \) slots for \( n \) entries, in the worst case),
this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion.
Open addressing

In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called closed hashing; it should not be confused with "open hashing" or "closed addressing" that usually mean separate chaining.)

Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by another hash function

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. Thus a more aggressive resize scheme is needed. Separate linking works correctly with any load factor, although performance is likely to be reasonable if it is kept below 2 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the same hash value; whether they are adjacent or nearby is completely irrelevant.

Even experienced programmers may find such clustering hard to avoid.

Open addressing only saves memory if the entries are small (less than 4 times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.
Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (that is, usually the only one). It also has better locality of reference, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups.

Hash tables with open addressing are also easier to serialize, because they do not use pointers.

On the other hand, normal open addressing is a poor choice for large elements, because these elements fill entire CPU cache lines (negating the cache advantage), and a large amount of space is wasted on large empty table slots. If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records that can be stored within the table (internal storage) and fit in a cache line. They are particularly suitable for elements of one word or less. If the table is expected to have a high load factor, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

 Ultimately, used sensibly, any kind of hash table algorithm is usually fast enough; and the percentage of a calculation spent in hash table code is low. Memory usage is rarely considered excessive. Therefore, in most cases the differences between these algorithms are marginal, and other considerations typically come into play.

**Coalesced hashing**

A hybrid of chaining and open addressing, coalesced hashing links together chains of nodes within the table itself.\[13\] Like open addressing, it achieves space usage and (somewhat diminished) cache advantages over chaining. Like chaining, it does not exhibit clustering effects; in fact, the table can be efficiently filled to a high density. Unlike chaining, it cannot have more elements than table slots.

**Robin Hood hashing**

One interesting variation on double-hashing collision resolution is Robin Hood hashing.\[14\] The idea is that a new key may displace a key already inserted, if its probe count is larger than that of the key at the current position. The net effect of this is that it reduces worst case search times in the table. This is similar to Knuth's ordered hash tables except that the criterion for bumping a key does not depend on a direct relationship between the keys. Since both the worst case and the variation in the number of probes is reduced dramatically, an interesting variation is to probe the table starting at the expected successful probe value and then expand from that position in both directions.\[15\] External Robin Hashing is an extension of this algorithm where the table is stored in an external file and each table position corresponds to a fixed-sized page or bucket with \(B\) records.\[16\]
Cuckoo hashing

Another alternative open-addressing solution is cuckoo hashing, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions. It uses two or more hash functions, which means any key/value pair could be in two or more locations. For lookup, the first hash function is used; if the key/value is not found, then the second hash function is used, and so on. If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets, at which point the table is resized. By combining multiple hash functions with multiple cells per bucket, very high space utilisation can be achieved.

Hopscotch hashing

Another alternative open-addressing solution is hopscotch hashing,¹⁷ which combines the approaches of cuckoo hashing and linear probing, yet seems in general to avoid their limitations. In particular it works well even when the load factor grows beyond 0.9. The algorithm is well suited for implementing a resizable concurrent hash table.

The hopscotch hashing algorithm works by defining a neighborhood of buckets near the original hashed bucket, where a given entry is always found. Thus, search is limited to the number of entries in this neighborhood, which is logarithmic in the worst case, constant on average, and with proper alignment of the neighborhood typically requires one cache miss. When inserting an entry, one first attempts to add it to a bucket in the neighborhood. However, if all buckets in this neighborhood are occupied, the algorithm traverses buckets in sequence until an open slot (an unoccupied bucket) is found (as in linear probing). At that point, since the empty bucket is outside the neighborhood, items are repeatedly displaced in a sequence of hops. (This is similar to cuckoo hashing, but with the difference that in this case the empty slot is being moved into the neighborhood, instead of items being moved out with the hope of eventually finding an empty slot.) Each hop brings the open slot closer to the original neighborhood, without invalidating the neighborhood property of any of the buckets along the way. In the end, the open slot has been moved into the neighborhood, and the entry being inserted can be added to it.

Dynamic resizing

To keep the load factor under a certain limit, e.g. under 3/4, many table implementations expand the table when items are inserted. For example, in Java's HashMap¹⁸ class the default load factor threshold for table expansion is 0.75. Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays.

Resizing is accompanied by a full or incremental table rehash whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table—followed by a rehash—when items are deleted. From the point of space-time tradeoffs, this operation is similar to the deallocation in dynamic arrays.
**Resizing by copying all entries**

A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold $r_{\text{max}}$. Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold $r_{\text{min}}$, all entries are moved to a new smaller table.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries $n$ and of the number $m$ of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If $m$ elements are inserted into that table, the total number of extra re-insertions that occur in all dynamic resizings of the table is at most $m-1$. In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

**Incremental resizing**

Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually:

- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move $r$ elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least $(r+1)/r$ during resizing.

**Monotonic keys**

If it is known that key values will always increase monotonically, then a variation of consistent hashing can be achieved by keeping a list of the single most recent key value at each hash table resize operation. Upon lookup, keys that fall in the ranges defined by these list entries are directed to the appropriate hash function—and indeed hash table—both of which can be different for each range. Since it is common to grow the overall number of entries by doubling, there will only be $O(\lg(N))$ ranges to check, and binary search time for the redirection would be $O(\lg(\lg(N)))$.

**Other solutions**

Linear hashing\(^{[19]}\) is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible look-up functions.

Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hashes, where rehashing is prohibitively costly.
Performance analysis

In the simplest model, the hash function is completely unspecified and the table does not resize. For the best possible choice of hash function, a table of size $n$ with open addressing has no collisions and holds up to $n$ elements, with a single comparison for successful lookup, and a table of size $n$ with chaining and $k$ keys has the minimum max(0, $k-n$) collisions and $O(1 + k/n)$ comparisons for lookup. For the worst choice of hash function, every insertion causes a collision, and hash tables degenerate to linear search, with $\Omega(k)$ amortized comparisons per insertion and up to $k$ comparisons for a successful lookup.

Adding rehashing to this model is straightforward. As in a dynamic array, geometric resizing by a factor of $b$ implies that only $k/b^i$ keys are inserted $i$ or more times, so that the total number of insertions is bounded above by $bk/(b-1)$, which is $O(k)$. By using rehashing to maintain $k < n$, tables using both chaining and open addressing can have unlimited elements and perform successful lookup in a single comparison for the best choice of hash function.

In more realistic models, the hash function is a random variable over a probability distribution of hash functions, and performance is computed on average over the choice of hash function. When this distribution is uniform, the assumption is called "simple uniform hashing" and it can be shown that hashing with chaining requires $\Theta(1 + k/n)$ comparisons on average for an unsuccessful lookup, and hashing with open addressing requires $\Theta(1/(1 - k/n))$.[20]

Both these bounds are constant, if we maintain $k/n < c$ using table resizing, where $c$ is a fixed constant less than 1.

Features

Advantages

The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more). Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

If the set of key-value pairs is fixed and known ahead of time (so insertions and deletions are not allowed), one may reduce the average lookup cost by a careful choice of the hash function, bucket table size, and internal data structures. In particular, one may be able to devise a hash function that is collision-free, or even perfect (see below). In this case the keys need not be stored in the table.

Drawbacks

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree. Thus hash tables are not effective when the number of entries is very small. (However, in some cases the high cost of computing the hash function can be mitigated by saving the hash value together with the key.)

For certain string processing applications, such as spell-checking, hash tables may be less efficient than tries, finite automata, or Judy arrays. Also, if each key is represented by a small enough number of bits, then, instead of a hash table, one may use the key directly as the index into an array of values. Note that there are no collisions in this case.

The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. Therefore, there is no efficient way to locate an entry whose key is nearest to a given key. Listing all $n$ entries in some specific order generally requires a separate sorting step, whose cost is proportional to $\log(n)$ per entry. In comparison, ordered search trees have lookup and insertion cost proportional to $\log(n)$, but allow finding the nearest key at about the same cost, and ordered enumeration of all entries at constant cost per entry.

If the keys are not stored (because the hash function is collision-free), there may be no easy way to enumerate the keys that are present in the table at any given moment.
Although the average cost per operation is constant and fairly small, the cost of a single operation may be quite high. In particular, if the hash table uses dynamic resizing, an insertion or deletion operation may occasionally take time proportional to the number of entries. This may be a serious drawback in real-time or interactive applications.

Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays. Compact data structures such as arrays searched with linear search may be faster, if the table is relatively small and keys are integers or other short strings. According to Moore's Law, cache sizes are growing exponentially and so what is considered "small" may be increasing. The optimal performance point varies from system to system.

Hash tables become quite inefficient when there are many collisions. While extremely uneven hash distributions are extremely unlikely to arise by chance, a malicious adversary with knowledge of the hash function may be able to supply information to a hash that creates worst-case behavior by causing excessive collisions, resulting in very poor performance (e.g., a denial of service attack). In critical applications, either universal hashing can be used or a data structure with better worst-case guarantees may be preferable.[21]

**Uses**

**Associative arrays**

Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects), especially in interpreted programming languages like AWK, Perl, and PHP.

When storing a new item into a multimap and a hash collision occurs, the multimap unconditionally stores both items.

When storing a new item into a typical associative array and a hash collision occurs, but the actual keys themselves are different, the associative array likewise stores both items. However, if the key of the new item exactly matches the key of an old item, the associative array typically erases the old item and overwrites it with the new item, so every item in the table has a unique key.

**Database indexing**

Hash tables may also be used as disk-based data structures and database indices (such as in dbm) although B-trees are more popular in these applications.

**Caches**

Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually erasing the old item that is currently stored in the table and overwriting it with the new item, so every item in the table has a unique hash value.
Sets
Besides recovering the entry that has a given key, many hash table implementations can also tell whether such an entry exists or not.
Those structures can therefore be used to implement a set data structure, which merely records whether a given key belongs to a specified set of keys. In this case, the structure can be simplified by eliminating all parts that have to do with the entry values. Hashing can be used to implement both static and dynamic sets.

Object representation
Several dynamic languages, such as Perl, Python, JavaScript, and Ruby, use hash tables to implement objects. In this representation, the keys are the names of the members and methods of the object, and the values are pointers to the corresponding member or method.

Unique data representation
Hash tables can be used by some programs to avoid creating multiple character strings with the same contents. For that purpose, all strings in use by the program are stored in a single hash table, which is checked whenever a new string has to be created. This technique was introduced in Lisp interpreters under the name hash consing, and can be used with many other kinds of data (expression trees in a symbolic algebra system, records in a database, files in a file system, binary decision diagrams, etc.)

Implementations

In programming languages
Many programming languages provide hash table functionality, either as built-in associative arrays or as standard library modules. In C++11, for example, the unordered_map class provides hash tables for keys and values of arbitrary type.
In PHP 5, the Zend 2 engine uses one of the hash functions from Daniel J. Bernstein to generate the hash values used in managing the mappings of data pointers stored in a HashTable. In the PHP source code, it is labelled as "DJBX33A" (Daniel J. Bernstein, Times 33 with Addition).
Python's built-in hash table implementation, in the form of the dict type, as well as Perl's hash type (%) are highly optimized as they are used internally to implement namespaces.
In the .NET Framework, support for hash tables is provided via the non-genericHashtable and genericDictionary classes, which store key-value pairs, and the genericHashSet class, which stores only values.

Independent packages
• Google Sparse Hash[22] The Google SparseHash project contains several C++ hash-map implementations in use at Google, with different performance characteristics, including an implementation that optimizes for memory use and one that optimizes for speed. The memory-optimized one is extremely memory-efficient with only 2 bits/entry of overhead.
• SunriseDD[23] An open source C library for hash table storage of arbitrary data objects with lock-free lookups, built-in reference counting and guaranteed order iteration. The library can participate in external reference counting systems or use its own built-in reference counting. It comes with a variety of hash functions and allows the use of runtime supplied hash functions via callback mechanism. Source code is well documented.
• uthash[24] This is an easy-to-use hash table for C structures.
Hash table

History

The idea of hashing arose independently in different places. In January 1953, H. P. Luhn wrote an internal IBM memorandum that used hashing with chaining.\(^{25}\) G. N. Amdahl, E. M. Boehme, N. Rochester, and Arthur Samuel implemented a program using hashing at about the same time. Open addressing with linear probing (relatively prime stepping) is credited to Amdahl, but Ershov (in Russia) had the same idea.\(^{25}\)

References


[2] (O(1) delete time assuming doubly-linked list chaining collision reduction technique)


[6] Karl Pearson (1900). "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling". *Philosophical Magazine, Series 5* 50 (302): pp. 157–175.


Further reading


External links

- A Hash Function for Hash Table Lookup (http://www.burtleburtle.net/bob/hash/doobs.html) by Bob Jenkins.
- Hash Tables (http://www.sparknotes.com/cs/searching/hashtables/summary.html) by SparkNotes—explanation using C
- Hash functions (http://www.azillionmonkeys.com/qed/hash.html) by Paul Hsieh
- Libhashish (http://libhashish.sourceforge.net/) hash library
- NIST entry on hash tables (http://www.nist.gov/dads/HTML/hashtab.html)
- A basic explanation of how the hash table works by Reliable Software (http://www.relisoft.com/book/lang/pointer/8hash.html)
- Lecture on Hash Tables (http://compgeom.cs.uiuc.edu/~jeffe/teaching/373/notes/06-hashing.pdf)
- Hash-tables in C (http://task3.cc/308/hash-maps-with-linear-probing-and-separate-chaining/)—two simple and clear examples of hash tables implementation in C with linear probing and chaining
- How to sort a HashMap (Java) and keep the duplicate entries (http://www.lampos.net/sort-hashmap)
Linear probing

Linear probing is a scheme in computer programming for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location.[1] This is accomplished using two values - one as a starting value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the stepsize, is repeatedly added to the starting value until a free space is found, or the entire table is traversed.

\[
\text{newLocation} = (\text{startingValue} + \text{stepSize}) \mod \text{arraySize}
\]

This algorithm, which is used in open-addressed hash tables, provides good memory caching (if stepsize is equal to one), through good locality of reference, but also results in clustering, an unfortunately high probability that where there has been one collision there will be more. The performance of linear probing is also more sensitive to input distribution when compared to double hashing.

Given an ordinary hash function \(H(x)\), a linear probing function \(H(x, i)\) would be:

\[
H(x, i) = (H(x) + i) \mod n.
\]

Here \(H(x)\) is the starting value, \(n\) the size of the hash table, and the stepsize is \(i\) in this case.

Dictionary operation in constant time

Using linear probing, dictionary operation can be implemented in constant time. In other words, insert, remove and find operations can be implemented in \(O(1)\), as long as the load factor of the hash table is a constant strictly less than one.[2] This analysis makes the (unrealistic) assumption that the hash function is completely random, but can be extended also to 5-independent hash functions.[3] Weaker properties, such as universal hashing, are not strong enough to ensure the constant-time operation of linear probing,[4] but one practical method of hash function generation, tabulation hashing, again leads to a guaranteed constant expected time performance despite not being 5-independent.[5]

References


External links

Quadratic probing

Quadratic probing is a scheme in computer programming for resolving collisions in hash tables. It is an open addressing method to handle overflows after a collision takes place in some bucket of a hash table. Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. The form of the equation is \( f(k) = c_1k^2 + c_2k + c_3 \). The function used might even be \( f(k) = c_1k^2 \) if \( c_2 \) and \( c_3 \) are taken as zero. In this case, suppose a cell \( H \) is reached but is occupied, then the next sequence of cells to be examined would be \( H + 1^2, H + 2^2, H + 3^2, H + 4^2, \ldots, H + k^2 \). Linear Probing, instead, would examine the sequence \( H + 1, H + 2, H + 3, H + 4, \ldots, H + k \). This would result in primary clustering and the larger the cluster grows, lesser will be the search efficiency for those items. Quadratic probing can be a more efficient algorithm in a closed hash table, since it better avoids the clustering problem that can occur with linear probing, although it is not immune. It also provides good memory caching because it preserves some locality of reference; however, linear probing has greater locality and, thus, better cache performance. Quadratic probing is used in the Berkeley Fast File System to allocate free blocks. The allocation routine chooses a new cylinder-group when the current is nearly full using quadratic probing, because of the speed it shows in finding unused cylinder-groups.

Quadratic Function

Let \( h(k) \) be a hash function that maps an element \( k \) to an integer in \([0, m - 1]\), where \( m \) is the size of the table. Let the \( i^{th} \) probe position for a value \( k \) be given by the function \( h(k, i) = (h(k) + c_1i + c_2i^2) \pmod{m} \), where \( c_2 \neq 0 \). If \( c_2 = 0 \), then \( h(k, i) \) degrades to a linear probe. For a given hash table, the values of \( c_1 \) and \( c_2 \) remain constant.

Examples:

- If \( h(k, i) = (h(k) + i + i^2) \pmod{m} \), then the probe sequence will be \( h(k), h(k) + 2, h(k) + 6, \ldots \)
- For \( m = 2^n \), a good choice for the constants are \( c_1 = c_2 = 1/2 \), as the values \( h(k, i) \) for \( i \) in \([0, m - 1]\) are all distinct. This leads to a probe sequence of \( h(k), h(k) + 1, h(k) + 3, h(k) + 6, \ldots \) where the values increase by \( 1, 2, 3, \ldots \)
- For prime \( m > 2 \), most choices of \( c_1 \) and \( c_2 \) will make \( h(k, i) \) distinct for \( i \) in \([0, (m - 1)/2]\). Such choices include \( c_1 = c_2 = 1/2 \), \( c_1 = c_2 = 1 \), and \( c_1 = 0 \), \( c_2 = 1 \). Because there are only about \( m/2 \) distinct probes for a given element, it is difficult to guarantee that insertions will succeed when the load factor is \( > 1/2 \).

Quadratic Probing Insertion

The problem, here, is to insert a key at an available key space in a given Hash Table using quadratic probing.\(^{[1]}\)

Algorithm to Insert key in Hash Table

1. Get the key \( k \)
2. Set counter \( j = 0 \)
3. Compute hash function \( h[k] = k \pmod{\text{SIZE}} \)
4. If hashtable[\( h[k] \)] is empty
   4.1 Insert key \( k \) at hashtable[\( h[k] \)]
   4.2 Stop
Else
   4.3 The key space at hashtable[\( h[k] \)] is occupied, so we need to find the next available key space
   4.4 Increment \( j \)
(4.5) Compute new hash function \( h[k] = (k + j^2) \mod \text{SIZE} \)

(4.6) Repeat Step 4 till \( j \) is more than \( \text{SIZE} \) of hash table

5. The hash table is full

6. Stop

C function for Key Insertion

```c
int quadratic_probing_insert(int *hashtable, int key, int *empty)
{
    /* hashtable[] is an integer hash table; empty[] is another array which indicates whether the key space is occupied;
       If an empty key space is found, the function returns the index of the bucket where the key is inserted, otherwise it returns (-1) if no empty key space is found */

    int j = 0, hk;
    hk = key % SIZE;
    while (j < SIZE)
    {
        if (empty[hk] == 1)
        {
            hashtable[hk] = key;
            full[hk] = 1;
            return (hk);
        }
        j++;
        hk = (key + j * j) % SIZE;
    }
    return (-1);
}
```
Example to Insert key in Hash Table

There are two possible cases to consider:

- **Key space at position \( h[k] \) is empty**: Insert the key at the position.
- **Key space at position \( h[k] \) is occupied**: Compute the next hash function \( h[k] \).

Consider a hash table initially containing some elements.

Suppose we want to insert a key 10 in the hash table.

\[
h[k] = 10 \mod 8 = 2
\]

Slot 2 being occupied the hash function will search for new available key space.

\[
h[k] = ( k + j \times j ) \mod \text{SIZE} \\
h[k] = ( 2 + 1 \times 1 ) \mod 8 = 3
\]

Slot 3 is also occupied, so the hash function will search for next available key space.

\[
h[k] = ( 2 + 2 \times 2 ) \mod 8 = 6
\]

Slot 6 is empty, so key will be inserted here.

**Quadratic Probing Search**

**Algorithm to Search Element in Hash Table**

1. Get the key \( k \) to be searched
2. Set counter \( j = 0 \)
3. Compute hash function \( h[k] = k \mod \text{SIZE} \)
4. If the key space at hashtable\[h[k]\] is occupied
   4.1) Compare the element at hashtable\[h[k]\] with the key \( k \).
   4.2) If they are equal
       4.2.1) The key is found at the bucket \( h[k] \)
       4.2.2) Stop
   Else
(4.3) The element might be placed at the next location given by the quadratic function
(4.4) Increment j
(4.5) Compute new hash function \(h[k] = (k + j \times j) \mod SIZE\)
(4.6) Repeat Step 4 till j is greater than SIZE of hash table

5. The key was not found in the hash table
6. Stop

C function for Key Searching

```c
int quadratic_probing_search(int *hashtable, int key, int *empty)
{
    /* If the key is found in the hash table, the function returns the
     index of the hashtable where the key is inserted, otherwise it
     returns (-1) if the key is not found */

    int j = 0, hk;
    hk = key  % SIZE;
    while (j < SIZE)
    {
        if ((empty[hk] == 0) && (hashtable[hk] == key))
            return (hk);
        j++;
        hk = (key + j * j) % SIZE;
    }
    return (-1);
}
```

Limitations

[2] For linear probing it is a bad idea to let the hash table get nearly full, because performance is degraded as the hash table gets filled. In the case of quadratic probing, the situation is even more drastic. There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions. If the hash table size is b (a prime greater than 3), it can be proven that the first \(b/2\) alternative locations including the initial location \(h(k)\) are all distinct and unique. Suppose, we assume two of the alternative locations to be given by \(h(k) + x^2 \mod b\) and \(h(k) + y^2 \mod b\), where \(0 \leq x, y \leq (b / 2)\). If these two locations point to the same key space, but \(x \neq y\). Then the following would have to be true,

\[
\begin{align*}
h(k) + x^2 &= h(k) + y^2 \mod b \\
x^2 &= y^2 \mod b \\
x^2 - y^2 &= 0 \mod b \\
(x - y)(x + y) &= 0 \mod b
\end{align*}
\]

As b (table size) is a prime greater than 3, either \((x - y)\) or \((x + y)\) has to be equal to zero. Since x and y are unique, \((x - y)\) cannot be zero. Also, since \(0 \leq x, y \leq (b / 2)\), \((x + y)\) cannot be zero.

Thus, by contradiction, it can be said that the first \((b / 2)\) alternative locations after \(h(k)\) are unique. So an empty key space can always be found as long as at most \((b / 2)\) locations are filled, i.e., the hash table is not more than half full.
Double hashing is a computer programming technique used in hash tables to resolve hash collisions, cases when two different values to be searched for produce the same hash key. It is a popular collision-resolution technique in open-addressed hash tables.

Like linear probing, it uses one hash value as a starting point and then repeatedly steps forward an interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is decided using a second, independent hash function (hence the name double hashing). Unlike linear probing and quadratic probing, the interval depends on the data, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering. In other words, given independent hash functions $h_1$ and $h_2$, the $j$th location in the bucket sequence for value $k$ in a hash table of size $m$ is:

$$h(k, j) = (h_1(k) + j \cdot h_2(k)) \mod m$$

### Disadvantages

Linear probing and, to a lesser extent, quadratic probing are able to take advantage of the data cache by accessing locations that are close together. Double hashing has larger intervals and is not able to achieve this advantage.

Like all other forms of open addressing, double hashing becomes linear as the hash table approaches maximum capacity. The only solution to this is to rehash to a larger size.

On top of that, it is possible for the secondary hash function to evaluate to zero. For example, if we choose $k=5$ with the following function:

$$h_2(k) = 5 - (k \mod 7)$$

The resulting sequence will always remain at the initial hash value. One possible solution is to change the secondary hash function to:

$$h_2(k) = (k \mod 7) + 1$$

This ensures that the secondary hash function will always be non zero.

### External links

- Hash Table Animation[^2]

### References

[^1]: http://www.siam.org/meetings/slenex05/papers/13heileman.pdf
Cuckoo hashing

Cuckoo hashing is a scheme in computer programming for resolving hash collisions of values of hash functions in a table. Cuckoo hashing was first described by Rasmus Pagh and Flemming Friche Rodler in 2001.[1] The name derives from the behavior of some species of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches.

Theory

The basic idea is to use two hash functions instead of only one. This provides two possible locations in the hash table for each key. In one of the commonly used variants of the algorithm, the hash table is split into two smaller tables of equal size, and each hash function provides an index into one of these two tables.

When a new key is inserted, a greedy algorithm is used: The new key is inserted in one of its two possible locations, "kicking out", that is, displacing, any key that might already reside in this location. This displaced key is then inserted in its alternative location, again kicking out any key that might reside there, until a vacant position is found, or the procedure enters an infinite loop. In the latter case, the hash table is rebuilt in-place[2] using new hash functions.

Lookup requires inspection of just two locations in the hash table, which takes constant time in the worst case (see Big O notation). This is in contrast to many other hash table algorithms, which may not have a constant worst-case bound on the time to do a lookup.

It can also be shown that insertions succeed in expected constant time,[1] even considering the possibility of having to rebuild the table, as long as the number of keys is kept below half of the capacity of the hash table, i.e., the load factor is below 50%. One method of proving this uses the theory of random graphs: one may form an undirected graph called the "Cuckoo Graph" that has a vertex for each hash table location, and an edge for each hashed value, with the endpoints of the edge being the two possible locations of the value. Then, the greedy insertion algorithm for adding a set of values to a cuckoo hash table succeeds if and only if the Cuckoo Graph for this set of values is a pseudoforest, a graph with at most one cycle in each of its connected components. This property is true with high probability for a random graph in which the number of edges is less than half the number of vertices.[3]
### Example

The following hash functions are given:

\[
h(k) = k \mod 11
\]

\[
h'(k) = \left\lfloor \frac{k}{11} \right\rfloor \mod 11
\]

<table>
<thead>
<tr>
<th>k</th>
<th>h(k)</th>
<th>h'(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>75</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>105</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1. table for h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
Cycle
If you now wish to insert the element 6, then you get into a cycle. In the last row of the table we find the same initial situation as at the beginning again.

\[ h(6) = 6 \mod 11 = 6 \]
\[ h'(6) = \left\lfloor \frac{6}{11} \right\rfloor \mod 11 = 0 \]

<table>
<thead>
<tr>
<th>considered key</th>
<th>table 1</th>
<th>table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>old value</td>
<td>new value</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>53</td>
<td>75</td>
<td>53</td>
</tr>
<tr>
<td>67</td>
<td>100</td>
<td>67</td>
</tr>
<tr>
<td>105</td>
<td>6</td>
<td>105</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>105</td>
<td>39</td>
</tr>
<tr>
<td>100</td>
<td>67</td>
<td>100</td>
</tr>
<tr>
<td>75</td>
<td>53</td>
<td>75</td>
</tr>
<tr>
<td>50</td>
<td>39</td>
<td>50</td>
</tr>
<tr>
<td>36</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>6</td>
</tr>
</tbody>
</table>
Generalizations and applications

Generalizations of cuckoo hashing that use more than 2 alternative hash functions can be expected to utilize a larger part of the capacity of the hash table efficiently while sacrificing some lookup and insertion speed. Using just three hash functions increases the load to 91%. Another generalization of cuckoo hashing consists in using more than one key per bucket. Using just 2 keys per bucket permits a load factor above 80%.

Other algorithms that use multiple hash functions include the Bloom filter. Cuckoo hashing can be used to implement a data structure equivalent to a Bloom filter. A simplified generalization of cuckoo hashing called skewed-associative cache is used in some CPU caches.

A study by Zukowski et al.[4] has shown that cuckoo hashing is much faster than chained hashing for small, cache-resident hash tables on modern processors. Kenneth Ross[5] has shown bucketized versions of cuckoo hashing (variants that use buckets that contain more than one key) to be faster than conventional methods also for large hash tables, when space utilization is high. The performance of the bucketized cuckoo hash table was investigated further by Askitis,[6] with its performance compared against alternative hashing schemes.


References

[2] Pagh and Rodler: "There is no need to allocate new tables for the rehashing: We may simply run through the tables to delete and perform the usual insertion procedure on all keys found not to be at their intended position in the table."

• A cool and practical alternative to traditional hash tables (http://www.ru.is/faculty/ulfar/CuckooHash.pdf), U. Erlingsson, M. Manasse, F. Mcsherry, 2006.
External links

- Cuckoo hash map written in C++ (http://sourceforge.net/projects/cuckoo-cpp/)
- Static cuckoo hashtable generator for C/C++ (http://www.theiling.de/projects/lookuptable.html)
- Cuckoo hashtable written in Java (http://Imonson.com/blog/?p=100)
- Generic Cuckoo hashmap in Java (http://github.com/joacima/Cuckoo-hash-map/blob/master/CuckooHashMap.java)

Hopscotch hashing

Hopscotch hashing is a scheme in computer programming for resolving hash collisions of values of hash functions in a table using open addressing. It is also well suited for implementing a concurrent hash table. Hopscotch hashing was introduced by Maurice Herlihy, Nir Shavit and Moran Tzafrir in 2008. The name is derived from the sequence of hops that characterize the table's insertion algorithm.

The algorithm uses a single array of \( n \) buckets. For each bucket, its neighborhood is a small collection of nearby consecutive buckets (i.e. one with close indexes to the original hashed bucket). The desired property of the neighborhood is that the cost of finding an item in the buckets of the neighborhood is close to the cost of finding it in the bucket itself (for example, by having buckets in the neighborhood fall within the same cache line). The size of the neighborhood must be sufficient to accommodate a logarithmic number of items in the worst case (i.e. it must accommodate \( \log(n) \) items), but only a constant number on average. If some bucket's neighborhood is filled, the table is resized.

In hopscotch hashing, as in cuckoo hashing, and unlike in linear probing, a given item will always be inserted-into and found-in the neighborhood of its hashed bucket. In other words, it will always be found either in its original hashed array entry, or in one of the next \( H-1 \) neighboring entries. \( H \) could, for example, be 32, the standard machine word size. The neighborhood is thus a "virtual" bucket that has fixed size and overlaps with the next \( H-1 \) buckets. To speed the search, each bucket (array entry) includes a "hop-information" word, an \( H \)-bit bitmap that indicates which of the next \( H-1 \) entries contain items that hashed to the current entry's virtual bucket. In this way, an item can be found quickly by looking at the word to see which entries belong to the bucket, and then scanning through the constant number of entries (most modern processors support special bit manipulation operations that make the lookup in the "hop-information" bitmap very fast).
Hopscotch hashing

Here is how to add item $x$ which was hashed to bucket $i$:

1. If the entry $i$ is empty, add $x$ to $i$ and return.
2. Starting at entry $i$, use a linear probe to find an empty entry at index $j$.
3. If the empty entry's index $j$ is within $H-1$ of entry $i$, place $x$ there and return. Otherwise, entry $j$ is too far from $i$.

   To create an empty entry closer to $i$, find an item $y$ whose hash value lies between $i$ and $j$, but within $H-1$ of $j$.
   Displacing $y$ to $j$ creates a new empty slot closer to $i$. Repeat until the empty entry is within $H-1$ of entry $i$, place $x$ there and return. If no such item $y$ exists, or if the bucket $i$ already contains $H$ items, resize and rehash the table.

The idea is that hopscotch hashing "moves the empty slot towards the desired bucket". This distinguishes it from linear probing which leaves the empty slot where it was found, possibly far away from the original bucket, or from cuckoo hashing that, in order to create a free bucket, moves an item out of one of the desired buckets in the target arrays, and only then tries to find the displaced item a new place.

To remove an item from the table, one simply removes it from the table entry. If the neighborhood buckets are cache aligned, then one could apply a reorganization operation in which items are moved into the now vacant location in order to improve alignment.

One advantage of hopscotch hashing is that it provides good performance at very high table load factors, even ones exceeding 0.9. Part of this efficiency is due to using a linear probe only to find an empty slot during insertion, not for every lookup as in the original linear probing hash table algorithm. Another advantage is that one can use any hash function, in particular simple ones that are close-to-universal.

References


Hash function

A hash function is any algorithm or subroutine that maps large data sets to smaller data sets, called keys. For example, a single integer can serve as an index to an array (cf. associative array). The values returned by a hash function are called hash values, hash codes, hash sums, checksums or simply hashes.

Hash functions are mostly used to accelerate table lookup or data comparison tasks such as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, and so on.

A hash function should be referentially transparent, i.e. if called twice on input that is "equal" (e.g. strings that consist of the same sequence of characters), it should give the same result. This is a contract in many programming languages that allow the user to override equality and hash functions for an object, that if two objects are equal their hash codes must be the same. This is important in order for it to be possible to find an element in a hash table quickly since two of the same element would both hash to the same slot.
Some hash functions may map two or more keys to the same hash value, causing a collision. Such hash functions try to map the keys to the hash values as evenly as possible because, as Hash Tables fill up, collisions become more frequent. Thus single digit hash values are frequently restricted to 80% of the size of the Table. Depending on the algorithm used, other properties may be required as well, such as double Hashing and Linear Probing. Although the idea was conceived in the 1950s,\[1\] the design of good hash functions is still a topic of active research.

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, randomization functions, error correcting codes, and cryptographic hash functions. Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimized differently. The HashKeeper database maintained by the American National Drug Intelligence Center, for instance, is more aptly described as a catalog of file fingerprints than of hash values.

### Applications

#### Hash tables

Hash functions are primarily used in hash tables, to quickly locate a data record (for example, a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to the hash. The index gives the place where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets.

In general, a hashing function may map several different keys to the same index. Therefore, each slot of a hash table is associated with (implicitly or explicitly) a set of records, rather than a single record. For this reason, each slot of a hash table is often called a bucket, and hash values are also called bucket indices.

Thus, the hash function only hints at the record’s location—it tells where one should start looking for it. Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries.

#### Caches

Hash functions are also used to build caches for large data sets stored in slow media. A cache is generally simpler than a hashed search table, since any collision can be resolved by discarding or writing back the older of the two colliding items.

#### Bloom filters

Hash functions are an essential ingredient of the Bloom filter, a compact data structure that provides an enclosing approximation to a set of them.

#### Finding duplicate records

When storing records in a large unsorted file, one may use a hash function to map each record to an index into a table \(T\), and collect in each bucket \(T[i]\) a list of the numbers of all records with the same hash value \(i\). Once the table is complete, any two duplicate records will end up in the same bucket. The duplicates can then be found by scanning every bucket \(T[i]\) which contains two or more members, fetching those records, and comparing them. With a table of appropriate size, this method is likely to be much faster than any alternative approach (such as sorting the file and comparing all consecutive pairs).
Finding similar records

Hash functions can also be used to locate table records whose key is similar, but not identical, to a given key; or pairs of records in a large file which have similar keys. For that purpose, one needs a hash function that maps similar keys to hash values that differ by at most \( m \), where \( m \) is a small integer (say, 1 or 2). If one builds a table \( T \) of all record numbers, using such a hash function, then similar records will end up in the same bucket, or in nearby buckets. Then one need only check the records in each bucket \( T[i] \) against those in buckets \( T[i+k] \) where \( k \) ranges between \(-m\) and \( m\).

This class includes the so-called acoustic fingerprint algorithms, that are used to locate similar-sounding entries in large collection of audio files. For this application, the hash function must be as insensitive as possible to data capture or transmission errors, and to "trivial" changes such as timing and volume changes, compression, etc.\(^2\)

Finding similar substrings

The same techniques can be used to find equal or similar stretches in a large collection of strings, such as a document repository or a genomic database. In this case, the input strings are broken into many small pieces, and a hash function is used to detect potentially equal pieces, as above.

The Rabin-Karp algorithm is a relatively fast string searching algorithm that works in \( O(n) \) time on average. It is based on the use of hashing to compare strings.

Geometric hashing

This principle is widely used in computer graphics, computational geometry and many other disciplines, to solve many proximity problems in the plane or in three-dimensional space, such as finding closest pairs in a set of points, similar shapes in a list of shapes, similar images in an image database, and so on. In these applications, the set of all inputs is some sort of metric space, and the hashing function can be interpreted as a partition of that space into a grid of cells. The table is often an array with two or more indices (called a grid file, grid index, bucket grid, and similar names), and the hash function returns an index tuple. This special case of hashing is known as geometric hashing or the grid method. Geometric hashing is also used in telecommunications (usually under the name vector quantization) to encode and compress multi-dimensional signals.

Properties

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. Note that different requirements apply to the other related concepts (cryptographic hash functions, checksums, etc.).

Low cost

The cost of computing a hash function must be small enough to make a hashing-based solution more efficient than alternative approaches. For instance, a self-balancing binary tree can locate an item in a sorted table of \( n \) items with \( O(\log n) \) key comparisons. Therefore, a hash table solution will be more efficient than a self-balancing binary tree if the number of items is large and the hash function produces few collisions and less efficient if the number of items is small and the hash function is complex.

Determinism

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the hashed data, in the mathematical sense of the term. This requirement excludes hash functions that depend on external variable parameters, such as pseudo-random number generators or the time of day. It also excludes functions that depend on the memory address of the object being hashed, because that address may change during execution (as may happen on systems that use certain methods of
garbage collection), although sometimes rehashing of the item is possible).

**Uniformity**

A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability. The reason for this last requirement is that the cost of hashing-based methods goes up sharply as the number of collisions—pairs of inputs that are mapped to the same hash value—increases. Basically, if some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

Note that this criterion only requires the value to be *uniformly distributed*, not *random* in any sense. A good randomizing function is (barring computational efficiency concerns) generally a good choice as a hash function, but the converse need not be true.

Hash tables often contain only a small subset of the valid inputs. For instance, a club membership list may contain only a hundred or so member names, out of the very large set of all possible names. In these cases, the uniformity criterion should hold for almost all typical subsets of entries that may be found in the table, not just for the global set of all possible entries.

In other words, if a typical set of \( m \) records is hashed to \( n \) table slots, the probability of a bucket receiving many more than \( m/n \) records should be vanishingly small. In particular, if \( m \) is less than \( n \), very few buckets should have more than one or two records. (In an ideal "perfect hash function", no bucket should have more than one record; but a small number of collisions is virtually inevitable, even if \( n \) is much larger than \( m \) -- see the birthday paradox).

When testing a hash function, the uniformity of the distribution of hash values can be evaluated by the chi-squared test.

**Variable range**

In many applications, the range of hash values may be different for each run of the program, or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data \( z \), and the number \( n \) of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to \( 2^{32} - 1 \)), divide the result by \( n \), and use the division's remainder. If \( n \) is itself a power of 2, this can be done by bit masking and bit shifting. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and \( n-1 \), for any \( n \) that may occur in the application. Depending on the function, the remainder may be uniform only for certain \( n \), e.g. odd or prime numbers.

It is possible to relax the restriction of the table size being a power of 2 and not having to perform any modulo, remainder or division operation -as these operation are considered computational costly in some contexts. For example, when \( n \) is significantly less than \( 2^b \) begin with a pseudo random number generator (PRNG) function \( P(key) \), uniform on the interval \([0, 2^b - 1]\). Consider the ratio \( q = 2^b / n \). Now the hash function can be seen as the value of \( P(key) / q \). Rearranging the calculation and replacing the \( 2^b \)-division by bit shifting right (\( >> \)) \( b \) times you end up with hash function \( n * P(key) >> b \).
Variable range with minimal movement (dynamic hash function)

When the hash function is used to store values in a hash table that outlives the run of the program, and the hash table needs to be expanded or shrunk, the hash table is referred to as a dynamic hash table.

A hash function that will relocate the minimum number of records when the table is resized is desirable. What is needed is a hash function $H(z,n)$ – where $z$ is the key being hashed and $n$ is the number of allowed hash values – such that $H(z,n+1) = H(z,n)$ with probability close to $n/(n+1)$.

Linear hashing and spiral storage are examples of dynamic hash functions that execute in constant time but relax the property of uniformity to achieve the minimal movement property.

Extendible hashing uses a dynamic hash function that requires space proportional to $n$ to compute the hash function, and it becomes a function of the previous keys that have been inserted.

Several algorithms that preserve the uniformity property but require time proportional to $n$ to compute the value of $H(z,n)$ have been invented.

Data normalization

In some applications, the input data may contain features that are irrelevant for comparison purposes. For example, when looking up a personal name, it may be desirable to ignore the distinction between upper and lower case letters. For such data, one must use a hash function that is compatible with the data equivalence criterion being used: that is, any two inputs that are considered equivalent must yield the same hash value. This can be accomplished by normalizing the input before hashing it, as by upper-casing all letters.

Continuity

A hash function that is used to search for similar (as opposed to equivalent) data must be as continuous as possible; two inputs that differ by a little should be mapped to equal or nearly equal hash values.

Note that continuity is usually considered a fatal flaw for check sums, cryptographic hash functions, and other related concepts. Continuity is desirable for hash functions only in some applications, such as hash tables that use linear search.

Hash function algorithms

For most types of hashing functions the choice of the function depends strongly on the nature of the input data, and their probability distribution in the intended application.

Trivial hash function

If the datum to be hashed is small enough, one can use the datum itself (reinterpreted as an integer in binary notation) as the hashed value. The cost of computing this "trivial" (identity) hash function is effectively zero. This hash function is perfect, as it maps each input to a distinct hash value.

The meaning of "small enough" depends on the size of the type that is used as the hashed value. For example, in Java, the hash code is a 32-bit integer. Thus the 32-bit integer `Integer` and 32-bit floating-point `Float` objects can simply use the value directly; whereas the 64-bit integer `Long` and 64-bit floating-point `Double` cannot use this method.

Other types of data can also use this perfect hashing scheme. For example, when mapping character strings between upper and lower case, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for "8", etc.). If each character is stored in 8 bits (as in ASCII or ISO Latin 1), the table has only $2^8 = 256$ entries; in the case of Unicode characters, the table would have $17 \times 2^{16} = 1114112$ entries.
The same technique can be used to map two-letter country codes like "us" or "za" to country names ($26^2=676$ table entries), 5-digit zip codes like 13083 to city names (100000 entries), etc. Invalid data values (such as the country code "xx" or the zip code 00000) may be left undefined in the table, or mapped to some appropriate "null" value.

**Perfect hashing**

A hash function that is injective—that is, maps each valid input to a different hash value—is said to be **perfect**. With such a function one can directly locate the desired entry in a hash table, without any additional searching.

**Minimal perfect hashing**

A perfect hash function for $n$ keys is said to be **minimal** if its range consists of $n$ consecutive integers, usually from 0 to $n-1$. Besides providing single-step lookup, a minimal perfect hash function also yields a compact hash table, without any vacant slots. Minimal perfect hash functions are much harder to find than perfect ones with a wider range.

**Hashing uniformly distributed data**

If the inputs are bounded-length strings (such as telephone numbers, car license plates, invoice numbers, etc.), and each input may independently occur with uniform probability, then a hash function need only map roughly the same number of inputs to each hash value. For instance, suppose that each input is an integer $z$ in the range 0 to $N-1$, and the output must be an integer $h$ in the range 0 to $n-1$, where $N$ is much larger than $n$. Then the hash function could be $h = z \mod n$ (the remainder of $z$ divided by $n$), or $h = (z \times n) \div N$ (the value $z$ scaled down by $n/N$ and truncated to an integer), or many other formulas.

Warning: $h = z \mod n$ was used in many of the original random number generators, but was found to have a number of issues. One of which is that as $n$ approaches $N$, this function becomes less and less uniform.
Hashing data with other distributions

These simple formulas will not do if the input values are not equally likely, or are not independent. For instance, most patrons of a supermarket will live in the same geographic area, so their telephone numbers are likely to begin with the same 3 to 4 digits. In that case, if \( n = 10000 \) or so, the division formula \((z \times n) \div N\), which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula \( z \mod n \), which is quite sensitive to the trailing digits, may still yield a fairly even distribution.

Hashing variable-length data

When the data values are long (or variable-length) character strings—such as personal names, web page addresses, or mail messages—their distribution is usually very uneven, with complicated dependencies. For example, text in any natural language has highly non-uniform distributions of characters, and character pairs, very characteristic of the language. For such data, it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way.

In cryptographic hash functions, a Merkle–Damgård construction is usually used. In general, the scheme for hashing such data is to break the input into a sequence of small units (bits, bytes, words, etc.) and combine all the units \( b[1], b[2], ..., b[m] \) sequentially, as follows

\[
S \leftarrow S0; \hspace{1cm} // \text{Initialize the state.} \\
\text{for } k \text{ in } 1, 2, ..., m \text{ do} \hspace{1cm} // \text{Scan the input data units:} \\
S \leftarrow F(S, b[k]); \hspace{1cm} // \text{Combine data unit } k \text{ into the state.} \\
\text{return } G(S, n) \hspace{1cm} // \text{Extract the hash value from the state.}
\]

This schema is also used in many text checksum and fingerprint algorithms. The state variable \( S \) may be a 32- or 64-bit unsigned integer; in that case, \( S0 \) can be 0, and \( G(S, n) \) can be just \( S \mod n \). The best choice of \( F \) is a complex issue and depends on the nature of the data. If the units \( b[k] \) are single bits, then \( F(S, b) \) could be, for instance

\[
\text{if } \text{highbit}(S) = 0 \hspace{1cm} \text{then} \\
\hspace{1cm} \text{return } 2 \times S + b \\
\text{else} \\
\hspace{1cm} \text{return } (2 \times S + b) \land P \hspace{1cm} \text{Here } \text{highbit}(S) \text{ denotes the most significant bit of } S; \text{ the } \land \text{ operator denotes unsigned integer multiplication with lost overflow; } \land \text{ is the bitwise exclusive or operation applied to words; and } P \text{ is a suitable fixed word.}[3]
\]

Special-purpose hash functions

In many cases, one can design a special-purpose (heuristic) hash function that yields many fewer collisions than a good general-purpose hash function. For example, suppose that the input data are file names such as FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, etc., with mostly sequential numbers. For such data, a function that extracts the numeric part \( k \) of the file name and returns \( k \mod n \) would be nearly optimal. Needless to say, a function that is exceptionally good for a specific kind of data may have dismal performance on data with different distribution.

Rolling hash

In some applications, such as substring search, one must compute a hash function \( h \) for every \( k \)-character substring of a given \( n \)-character string \( t \); where \( k \) is a fixed integer, and \( n \) is \( k \). The straightforward solution, which is to extract every such substring \( s \) of \( t \) and compute \( h(s) \) separately, requires a number of operations proportional to \( kn \).

However, with the proper choice of \( h \), one can use the technique of rolling hash to compute all those hashes with an effort proportional to \( k+n \).
Universal hashing

A universal hashing scheme is a randomized algorithm that selects a hashing function \( h \) among a family of such functions, in such a way that the probability of a collision of any two distinct keys is \( 1/n \), where \( n \) is the number of distinct hash values desired—indeed of its two keys. Universal hashing ensures (in a probabilistic sense) that the hash function application will behave as well as if it were using a random function, for any distribution of the input data. It will however have more collisions than perfect hashing, and may require more operations than a special-purpose hash function.

Hashing with checksum functions

One can adapt certain checksum or fingerprinting algorithms for use as hash functions. Some of those algorithms will map arbitrary long string data \( z \), with any typical real-world distribution—no matter how non-uniform and dependent—to a 32-bit or 64-bit string, from which one can extract a hash value in \( 0 \) through \( n-1 \).

This method may produce a sufficiently uniform distribution of hash values, as long as the hash range size \( n \) is small compared to the range of the checksum or fingerprint function. However, some checksums fare poorly in the avalanche test, which may be a concern in some applications. In particular, the popular CRC32 checksum provides only 16 bits (the higher half of the result) that are usable for hashing. Moreover, each bit of the input has a deterministic effect on each bit of the CRC32, that is one can tell without looking at the rest of the input, which bits of the output will flip if the input bit is flipped; so care must be taken to use all 32 bits when computing the hash from the checksum.\(^4\)

Hashing with cryptographic hash functions

Some cryptographic hash functions, such as SHA-1, have even stronger uniformity guarantees than checksums or fingerprints, and thus can provide very good general-purpose hashing functions.

In ordinary applications, this advantage may be too small to offset their much higher cost.\(^5\) However, this method can provide uniformly distributed hashes even when the keys are chosen by a malicious agent. This feature may help protect services against denial of service attacks.

Origins of the term

The term "hash" comes by way of analogy with its non-technical meaning, to "chop and mix". Indeed, typical hash functions, like the \( \text{mod} \) operation, "chop" the input domain into many sub-domains that get "mixed" into the output range to improve the uniformity of the key distribution.

Donald Knuth notes that Hans Peter Luhn of IBM appears to have been the first to use the concept, in a memo dated January 1953, and that Robert Morris used the term in a survey paper in CACM which elevated the term from technical jargon to formal terminology.\(^1\)

List of hash functions

- Bernstein hash\(^6\)
- Fowler-Noll-Vo hash function (32, 64, 128, 256, 512, or 1024 bits)
- Jenkins hash function (32 bits)
- Pearson hashing (8 bits)
- Zobrist hashing
References


External links

- General purpose hash function algorithms (C/C++/Pascal/Java/Python/Ruby) (http://www.partow.net/programming/hashfunctions/index.html)
- Hash Functions and Block Ciphers by Bob Jenkins (http://burtleburtle.net/bob/hash/index.html)
Perfect hash function

A **perfect hash function** for a set S is a hash function that maps distinct elements in S to a set of integers, with no collisions. A perfect hash function has many of the same applications as other hash functions, but with the advantage that no collision resolution has to be implemented.

Properties and uses

A perfect hash function for a specific set S that can be evaluated in constant time, and with values in a small range, can be found by a randomized algorithm in a number of operations that is proportional to the size of S. The minimal size of the description of a perfect hash function depends on the range of its function values: The smaller the range, the more space is required. Any perfect hash functions suitable for use with a hash table require at least a number of bits that is proportional to the size of S.

A perfect hash function with values in a limited range can be used for efficient lookup operations, by placing keys from S (or other associated values) in a table indexed by the output of the function. Using a perfect hash function is best in situations where there is a frequently queried large set, S, which is seldom updated. Efficient solutions to performing updates are known as dynamic perfect hashing, but these methods are relatively complicated to implement. A simple alternative to perfect hashing, which also allows dynamic updates, is cuckoo hashing.

Minimal perfect hash function

A **minimal perfect hash function** is a perfect hash function that maps n keys to n consecutive integers—usually \([0..n-1]\) or \([1..n]\). A more formal way of expressing this is: Let \(j\) and \(k\) be elements of some finite set \(K\). F is a minimal perfect hash function iff \(F(j) = F(k)\) implies \(j=k\) and there exists an integer \(a\) such that the range of F is \(a..a+|K|-1\). It has been proved that a general purpose minimal perfect hash scheme requires at least 1.44 bits/key.\(^{[1]}\) However the smallest currently use around 2.5 bits/key.

A minimal perfect hash function F is **order preserving** if keys are given in some order \(a_1\), \(a_2\), ..., and for any keys \(a_j\) and \(a_k\), \(j<k\) implies \(F(a_j)<F(a_k)\). Order-preserving minimal perfect hash functions require necessarily \(\Omega(n \log n)\) bits to be represented.

A minimal perfect hash function F is **monotone** if it preserves the lexicographical order of the keys. Monotone minimal perfect hash functions can be represented in very little space.

References


Further reading

Perfect hash function

(CIKM07), Lisbon, Portugal, November 2007.


External links

• Minimal Perfect Hashing (http://burtleburtle.net/bob/hash/perfect.html) by Bob Jenkins

• gperf (http://www.gnu.org/software/gperf/) is a Free software C and C++ perfect hash generator

• cmph (http://cmph.sourceforge.net/index.html) is Free Software implementing many perfect hashing methods

• Sux4J (http://sux4j.dsi.unimi.it/) is Free Software implementing perfect hashing, including monotone minimal perfect hashing in Java

• MPHSharp (http://www.dupuis.me/node/9) is Free Software implementing many perfect hashing methods in C#

Universal hashing

Using universal hashing (in a randomized algorithm or data structure) refers to selecting a hash function at random from a family of hash functions with a certain mathematical property (see definition below). This guarantees a low number of collisions in expectation, even if the data is chosen by an adversary. Many universal families are known (for hashing integers, vectors, strings), and their evaluation is often very efficient. Universal hashing has numerous uses in computer science, for example in implementations of hash tables, randomized algorithms, and cryptography.

Introduction

Assume we want to map keys from some universe $U$ into $m$ bins (labelled $[m] = \{0, \ldots, m - 1\}$). The algorithm will have to handle some data set $S \subseteq U$ of $|S| = n$ keys, which is not known in advance. Usually, the goal of hashing is to obtain a low number of collisions (keys from $S$ that land in the same bin). A deterministic hash function cannot offer any guarantee in an adversarial setting if the size of $U$ is greater than $m^2$, since the adversary may choose $S$ to be precisely the preimage of a bin. This means that all data keys land in the same bin, making hashing useless. Furthermore, a deterministic hash function does not allow for rehashing: sometimes the input data turns out to be bad for the hash function (e.g. there are too many collisions), so one would like to change the hash function.

The solution to these problems is to pick a function randomly from a family of hash functions. A family of functions $H = \{h : U \to [m]\}$ is called a universal family if,

$$\forall x, y \in U, x \neq y : \Pr_{h \in H} [h(x) = h(y)] \leq \frac{1}{m}.$$

In other words, any two keys of the universe collide with probability at most $1/m$ when the hash function $h$ is drawn randomly from $H$. This is exactly the probability of collision we would expect if the hash function assigned truly random hash codes to every key. Sometimes, the definition is relaxed to allow collision probability $O(1/m)$.

This concept was introduced by Carter and Wegman\(^1\) in 1977, and has found numerous applications in computer science (see, for example \(^2\)).
Many, but not all, universal families have the following stronger **uniform difference property**:

\[ \forall x, y \in U, x \neq y, \text{ when } h \text{ is drawn randomly from the family } H, \text{ the difference } h(x) - h(y) \text{ mod } m \text{ is uniformly distributed in } [m]. \]

Note that the definition of universality is only concerned with whether \( h(x) - h(y) = 0 \), which counts collisions. The uniform difference property is stronger. Indeed, given a universal family, one can produce a 2-independent hash function by adding a uniformly distributed random constant with values in \([m]\) to the hash functions. Since a shift by a constant is typically irrelevant in applications (e.g. hash tables), a careful distinction between universal and 2-independent hash families is often not made.\[^{[3]}\]

**Mathematical guarantees**

For any fixed set \( S \) of \( n \) keys, using a universal family guarantees the following properties.

1. For any fixed \( x \) in \( S \), the expected number of keys in the bin \( h(x) \) is \( n/m \). When implementing hash tables by chaining, this number is proportional to the expected running time of an operation involving the key \( x \) (for example a query, insertion or deletion).

2. The expected number of pairs of keys \( x, y \) in \( S \) with \( x \neq y \) that collide ( \( h(x) = h(y) \) ) is bounded above by \( n(n - 1)/2m \), which is of order \( O(n^2/m) \). When the number of bins, \( m \), is \( O(n) \), the expected number of collisions is \( O(n) \). When hashing into \( n^2 \) bins, there are no collisions at all with probability at least a half.

3. The expected number of keys in bins with at least \( t \) keys in them is bounded above by \( 2n/(t - 2(n/m) + 1) \).\[^{[4]}\] Thus, if the capacity of each bin is capped to three times the average size ( \( t = 3n/m \) ), the total number of keys in overflowing bins is at most \( O(m) \). This only holds with a hash family whose collision probability is bounded above by \( 1/m \). If a weaker definition is used, bounding it by \( O(1/m) \), this result is no longer true.\[^{[4]}\]

As the above guarantees hold for any fixed set \( S \), they hold if the data set is chosen by an adversary. However, the adversary has to make this choice before (or independent of) the algorithm's random choice of a hash function. If the adversary can observe the random choice of the algorithm, randomness serves no purpose, and the situation is the same as deterministic hashing.

The second and third guarantee are typically used in conjunction with rehashing. For instance, a randomized algorithm may be prepared to handle some \( O(n) \) number of collisions. If it observes too many collisions, it chooses another random \( h \) from the family and repeats. Universality guarantees that the number of repetitions is a geometric random variable.

**Constructions**

Since any computer data can be represented as one or more machine words, one generally needs hash functions for three types of domains: machine words ("integers"); fixed-length vectors of machine words; and variable-length vectors ("strings").

**Hashing integers**

This section refers to the case of hashing integers that fit in machines words; thus, operations like multiplication, addition, division, etc. are cheap machine-level instructions. Let the universe to be hashed be \( U = \{0, \ldots, u - 1\} \).

The original proposal of Carter and Wegman\[^{[1]}\] was to pick a prime \( p \geq u \) and define

\[ h_{a,b}(x) = ((ax + b) \text{ mod } p) \text{ mod } m \]
where $a, b$ are randomly chosen integers modulo $p$ with $a \neq 0$. Technically, adding $b$ is not needed for universality (but it does make the hash function 2-independent).

To see that $H = \{h_{a,b}\}$ is a universal family, note that $h(x) = h(y)$ only holds when

$$ax + b \equiv ay + b + i \cdot m \pmod{p}$$

for some integer $i$ between 0 and $p/m$. If $x \neq y$, their difference, $x - y$, is nonzero and has an inverse modulo $p$. Solving for $a$:

$$a \equiv i \cdot m \cdot (x - y)^{-1} \pmod{p}.$$  

There are $p - 1$ possible choices for $a$ (since $a = 0$ is excluded) and, varying $i$ in the allowed range, $\lfloor p/m \rfloor$ possible values for the right hand side. Thus the collision probability is

$$\frac{p/m}{(p - 1)}$$

which tends to $1/m$ for large $p$ as required. This analysis also shows that $b$ does not have to be randomised in order to have universality.

Another way to see $H$ is a universal family is via the notion of statistical distance. Write the difference $h(x) - h(y)$ as

$$h(x) - h(y) \equiv (a(x - y) \mod p) \pmod{m}.$$  

Since $x - y$ is nonzero and $a$ is uniformly distributed in $\{1, \ldots, p\}$, it follows that $a(x - y) \mod p$ is also uniformly distributed in $\{1, \ldots, p\}$. The distribution of $(h(x) - h(y)) \mod m$ is thus almost uniform, up to a difference in probability of $\pm 1/p$ between the samples. As a result, the statistical distance to a uniform family is $O(m/p)$, which becomes negligible when $p \gg m$.

**Avoiding modular arithmetic**

The state of the art for hashing integers is the **multiply-shift** scheme described by Dietzfelbinger et al. in 1997.[5] By avoiding modular arithmetic, this method is much easier to implement and also runs significantly faster in practice (usually by at least a factor of four[6]). The scheme assumes the number of bins is a power of two, $m = 2^M$. Let $w$ be the number of bits in a machine word. Then the hash functions are parametrised over odd positive integers $a < 2^w$ (that fit in a word of $w$ bits). To evaluate $h_a(x)$, multiply $x$ by $a$ modulo $2^w$ and then keep the high order $M$ bits as the hash code. In mathematical notation, this is

$$h_a(x) = (a \cdot x \mod 2^w) \div 2^{w-M}$$

and it can be implemented in C-like programming languages by

$$h_a(x) = (\text{unsigned}) (a \times x) >> (w-M)$$

This scheme does **not** satisfy the uniform difference property and is only $2/m$-almost-universal; for any $x \neq y$,

$$\Pr\{h_a(x) = h_a(y)\} \leq 2/m.$$  

To understand the behavior of the hash function, notice that, if $ax \mod 2^w$ and $ay \mod 2^w$ have the same highest-order $M$ bits, then $ax \mod 2^w$ has either all 1’s or all 0’s as its highest order $M$ bits (depending on whether $ax \mod 2^w$ or $ay \mod 2^w$ is larger). Assume that the least significant set bit of $x - y$ appears on position $w - c$. Since $a$ is a random odd integer and odd integers have inverses in the ring $\mathbb{Z}_{2^w}$, it follows that $a(x - y) \mod 2^w$ will be uniformly distributed among $w$-bit integers with the least significant set bit on position $w - c$. The probability that these bits are all 0’s or all 1’s is therefore at most $2^{2/M} = 2/m$. On the other hand, if $c < M$, then higher-order $M$ bits of $a(x - y) \mod 2^w$ contain both 0’s and 1’s, so it is certain that $h(x) \neq h(y)$. Finally, if $c = M$ then bit $w - M$ of $a(x - y) \mod 2^w$ is 1 and $h_a(x) = h_a(y)$ if and only if $x \equiv y \pmod{2^M}$, as $x = 2^{w-M} - 2^M + y = 3x \mod 2^w$. To obtain a truly ‘universal’ hash function, one can use the multiply-add-shift scheme

$$h_{a,b}(x) = ((ax + b) \mod 2^w) \div 2^{w-M}$$
Universal hashing

where \(a\) is a random odd positive integer with \(a < 2^w\) and \(b = i2^{w/2}\) where \(i\) is chosen at random from \(\{0, \ldots, 2^{w/2} - 1\}\). With these choices of \(a\) and \(b\), \(\Pr\{h_{a,b}(x) = h_{a,b}(y)\} \leq 1/m\) for all \(x \neq y \pmod{2^w}\).\(^7\)

### Hashing vectors

This section is concerned with hashing a fixed-length vector of machine words. Interpret the input as a vector \(\vec{x} = (x_0, \ldots, x_{k-1})\) of \(k\) machine words (integers of \(w\) bits each). If \(H\) is a universal family with the uniform difference property, the following family dating back to Carter and Wegman\(^1\) also has the uniform difference property (and hence is universal):

\[
h(\vec{x}) = \left( \sum_{i=0}^{k-1} h_i(x_i) \right) \pmod{m}, \quad \text{where each} \ h_i \in H \ \text{is chosen independently at random.}
\]

If \(m\) is a power of two, one may replace summation by exclusive or.\(^8\) In practice, if double-precision arithmetic is available, this is instantiated with the multiply-shift hash family of.\(^9\) Initialize the hash function with a vector \(\vec{a} = (a_0, \ldots, a_{k-1})\) of random odd integers on \(2^w\) bits each. Then if the number of bins is \(m = 2^M\) for \(M \leq w\):

\[
h_{a}(\vec{x}) = \left( \sum_{i=0}^{k-1} x_i \cdot a_i \pmod{2^{2w}} \right) \div 2^{2w-M}.
\]

It is possible to halve the number of multiplications, which roughly translates to a two-fold speed-up in practice.\(^8\) Initialize the hash function with a vector \(\vec{a} = (a_0, \ldots, a_{k-1})\) of random odd integers on \(2^w\) bits each. The following hash family is universal\(^10\):

\[
h_{a}(\vec{x}) = \left( \sum_{i=0}^{\lceil k/2 \rceil} (x_{2i} + a_{2i}) \cdot (x_{2i+1} + a_{2i+1}) \pmod{2^{2w}} \right) \div 2^{2w-M}.
\]

If double-precision operations are not available, one can interpret the input as a vector of half-words (\(w/2\)-bit integers). The algorithm will then use \(\lceil k/2 \rceil\) multiplications, where \(k\) was the number of half-words in the vector. Thus, the algorithm runs at a “rate” of one multiplication per word of input.

The same scheme can also be used for hashing integers, by interpreting their bits as vectors of bytes. In this variant, the vector technique is known as tabulation hashing and it provides a practical alternative to multiplication-based universal hashing schemes.\(^11\)

### Hashing strings

This refers to hashing a variable-sized vector of machine words. If the length of the string can be bounded by a small number, it is best to use the vector solution from above (conceptually padding the vector with zeros up to the upper bound). The space required is the maximal length of the string, but the time to evaluate \(h(s)\) is just the length of \(s\) (the zero-padding can be ignored when evaluating the hash function without affecting universality).\(^8\)

Now assume we want to hash \(\vec{x} = (x_0, \ldots, x_{k})\), where a good bound on \(\ell\) is not known a priori. A universal family proposed by.\(^9\) treats the string \(x\) as the coefficients of a polynomial modulo a large prime. If \(x_i \in [u]\), let \(p \geq \max\{u, m\}\) be a prime and define:

\[
h_{a}(\vec{x}) = h_{\text{int}} \left( \sum_{i=0}^{\ell} x_i \cdot a^i \mod p \right), \quad \text{where} \ a \in [p] \ \text{is uniformly random and} \ h_{\text{int}} \ \text{is chosen randomly from a universal family mapping integer domain} \ [p] \mapsto [m].
\]

Consider two strings \(\vec{x}, \vec{y}\) and let \(\ell\) be length of the longer one; for the analysis, the shorter string is conceptually padded with zeros up to length \(\ell\). A collision before applying \(h_{\text{int}}\) implies that \(a\) is a root of the polynomial with
coefficients $\bar{x} - \bar{y}$. This polynomial has at most $\ell$ roots modulo $p$, so the collision probability is at most $\ell/p$. The probability of a collision through the random $h_{\text{int}}$ brings the total collision probability to $\frac{1}{m} + \frac{\ell}{p}$. Thus, if the prime $p$ is sufficiently large compared to the length of strings hashed, the family is very close to universal (in statistical distance).

To mitigate the computational penalty of modular arithmetic, two tricks are used in practice:

1. One chooses the prime $p$ to be close to a power of two, such as a Mersenne prime. This allows arithmetic modulo $p$ to be implemented without division (using faster operations like addition and shifts). For instance, on modern architectures one can work with $p = 2^{61} - 1$, while $x_i$’s are 32-bit values.

2. One can apply vector hashing to blocks. For instance, one applies vector hashing to each 16-word block of the string, and applies string hashing to the results. Since the slower string hashing is applied on a substantially smaller vector, this will essentially be as fast as vector hashing.

References


Further reading

K-independent hashing

A family of hash functions is said to be \( k \)-independent or \( k \)-universal\(^{(1)} \) if selecting a hash function at random from the family guarantees that the hash codes of any designated \( k \) keys are independent random variables (see precise mathematical definitions below). Such families allow good average case performance in randomized algorithms or data structures, even if the input data is chosen by an adversary. The trade-offs between the degree of independence and the efficiency of evaluating the hash function are well studied, and many \( k \)-independent families have been proposed.

Introduction

The goal of hashing is usually to map keys from some large domain (universe) \( U \) into a smaller range, such as \( m \) bins (labelled \( [m] = \{0, \ldots, m - 1\} \)). In the analysis of randomized algorithms and data structures, it is often desirable for the hash codes of various keys to "behave randomly". For instance, if the hash code of each key were an independent random choice in \( [m] \), the number of keys per bin could be analyzed using the Chernoff bound. A deterministic hash function cannot offer any such guarantee in an adversarial setting, as the adversary may choose the keys to be the precisely the preimage of a bin. Furthermore, a deterministic hash function does not allow for rehashing: sometimes the input data turns out to be bad for the hash function (e.g. there are too many collisions), so one would like to change the hash function.

The solution to these problems is to pick a function randomly from a large family of hash functions. The randomness in choosing the hash function can be used to guarantee some desired random behavior of the hash codes of any keys of interest. The first definition along these lines was universal hashing, which guarantees a low collision probability for any two designated keys. The concept of \( k \)-independent hashing, introduced by Wegman and Carter in 1981\(^{(2)} \), strengthens the guarantees of random behavior to families of \( k \)-designated keys, and adds a guarantee on the uniform distribution of hash codes.

Mathematical Definitions

The strictest definition, introduced by Wegman and Carter\(^{(2)} \) under the name "strongly universal \( k \) hash family", is the following. A family of hash functions \( H = \{h : U \to [m]\} \) is \( k \)-independent iff for any \( k \) distinct keys \((x_1, \ldots, x_k) \in U^k\) and any \( k \) hash codes (not necessarily distinct) \((y_1, \ldots, y_k) \in [m]^k\), we have:

\[
\Pr_{h \in H} [h(x_1) = y_1 \land \cdots \land h(x_k) = y_k] = m^{-k}
\]

This definition is equivalent to the following two conditions:

1. for any fixed \( x \in U \), as \( h \) is drawn randomly from \( H \), \( h(x) \) is uniformly distributed in \([m]\).
2. for any fixed, distinct keys \( x_1, \ldots, x_k \in U \), as \( h \) is drawn randomly from \( H \), \( h(x_1), \ldots, h(x_k) \) are independent random variables.

Often it is inconvenient to achieve the perfect joint probability of \( m^{-k} \) due to rounding issues. Following\(^{(3)} \), one may define a \((\mu, k)\)-independent family to satisfy:

\[
\forall \text{ distinct } (x_1, \ldots, x_k) \in U^k \land \Pr_{h \in H} [h(x_1) = y_1 \land \cdots \land h(x_k) = y_k] \leq \mu / m^k
\]

Observe that, even if \( \mu \) is close to 1, \( h(x_i) \) are no longer independent random variables, which is often a problem in the analysis of randomized algorithms. Therefore, a more common alternative to dealing with rounding issues is to prove that the hash family is close in statistical distance to a \( k \)-independent family, which allows black-box use of the independence properties.


References


Further reading


Tabulation hashing

In computer science, tabulation hashing is a method for constructing universal families of hash functions by combining table lookup with exclusive or operations. It is simple and fast enough to be usable in practice, and has theoretical properties that (in contrast to some other universal hashing methods) make it usable with linear probing, cuckoo hashing, and the MinHash technique for estimating the size of set intersections. It was first proposed by Carter & Wegman (1979) and studied in more detail by Pătraşcu & Thorup (2011).

The method

Let $p$ denote the number of bits in a key to be hashed, and $q$ denote the number of bits desired in an output hash function. Let $r$ be a number smaller than $p$, and let $t$ be the smallest integer that is at least as large as $p/r$. For instance, if $r = 8$, then an $r$-bit number is a byte, and $t$ is the number of bytes per key.

The key idea of tabulation hashing is to view a key as a vector of $t$ $r$-bit numbers, use a lookup table filled with random values to compute a hash value for each of the $r$-bit numbers representing a given key, and combine these values with the bitwise binary exclusive or operation. The choice of $t$ and $r$ should be made in such a way that this table is not too large; e.g., so that it fits into the computer’s cache memory.

The initialization phase of the algorithm creates a two-dimensional array $T$ of dimensions $2^r$ by $t$, and fills the array with random numbers. Once the array $T$ is initialized, it can be used to compute the hash value $h(x)$ of any given key $x$. To do so, partition $x$ into $r$-bit values, where $x_0$ consists of the low order $r$ bits of $x$, $x_1$ consists of the next $r$ bits, etc. (E.g., again, with $r = 8$, $x_i$ is just the $i$th byte of $x$). Then, use these values as indices into $T$ and combine them with the exclusive or operation:

$$h(x) = T[x_0,0] \oplus T[x_1,1] \oplus T[x_2,2] \oplus ...$$

Universality

Carter & Wegman (1979) define a randomized scheme for generating hash functions to be universal if, for any two keys, the probability that they collide (that is, they are mapped to the same value as each other) is $1/m$, where $m$ is the number of values that the keys can take on. They defined a stronger property in the subsequent paper Wegman & Carter (1981): a randomized scheme for generating hash functions is $k$-independent if, for every $k$-tuple of keys, and each possible $k$-tuple of values, the probability that those keys are mapped to those values is $1/m^k$. 2-independent hashing schemes are automatically universal, and any universal hashing scheme can be converted into a 2-independent scheme by storing a random number $x$ in the initialization phase of the algorithm and adding $x$ to each hash value, so universality is essentially the same as 2-independence, but $k$-independence for larger values of $k$ is a
stronger property, held by fewer hashing algorithms.

As Pătraşcu & Thorup (2011) observe, tabulation hashing is 3-independent but not 4-independent. For any single key $x$, $T[x_0,0]$ is equally likely to take on any hash value, and the exclusive or of $T[x_0,0]$ with the remaining table values does not change this property. For any two keys $x$ and $y$, $x$ is equally likely to be mapped to any hash value as before, and there is at least one position $i$ where $x_i \neq y_i$; the table value $T[y_i,i]$ is used in the calculation of $h(y)$ but not in the calculation of $h(x)$, so even after the value of $h(x)$ has been determined, $h(y)$ is equally likely to be any valid hash value. Similarly, for any three keys $x$, $y$, and $z$, at least one of the three keys has a position $i$ where its value $z_i$ differs from the other two, so that even after the values of $h(x)$ and $h(z)$ are determined, $h(z)$ is equally likely to be any valid hash value.

However, this reasoning breaks down for four keys because there are sets of keys $w$, $x$, $y$, and $z$ where none of the four has a byte value that it does not share with at least one of the other keys. For instance, if the keys have two bytes each, and $w$, $x$, $y$, and $z$ are the four keys that have either zero or one as their byte values, then each byte value in each position is shared by exactly two of the four keys. For these four keys, the hash values computed by tabulation hashing will always satisfy the equation $h(w) \oplus h(x) \oplus h(y) \oplus h(z) = 0$, whereas for a 4-independent hashing scheme the same equation would only be satisfied with probability $1/m$. Therefore, tabulation hashing is not 4-independent.

Siegel (2004) uses the same idea of using exclusive or operations to combine random values from a table, with a more complicated algorithm based on expander graphs for transforming the key bits into table indices, to define hashing schemes that are $k$-independent for any constant or even logarithmic value of $k$. However, the number of table lookups needed to compute each hash value using Siegel's variation of tabulation hashing, while constant, is still too large to be practical, and the use of expanders in Siegel's technique also makes it not fully constructive.

One limitation of tabulation hashing is that it assumes that the input keys have a fixed number of bits. Lemire (2010) has studied variations of tabulation hashing that can be applied to variable-length strings, and shown that they can be universal (2-independent) but not 3-independent.

**Application to specific hashing techniques**

Because tabulation hashing is a universal hashing scheme, it can be used in any hashing-based algorithm in which universality is sufficient. For instance, in hash chaining, the expected time per operation is proportional to the sum of collision probabilities, which is the same for any universal scheme as it would be for truly random hash functions, and is constant whenever the load factor of the hash table is constant. Therefore, tabulation hashing can be used to compute hash functions for hash chaining with a theoretical guarantee of constant expected time per operation.[1]

However, universal hashing is not strong enough to guarantee the performance of some other hashing algorithms. For instance, for linear probing, 5-independent hash functions are strong enough to guarantee constant time operation, but there are 4-independent hash functions that fail.[2] Nevertheless, despite only being 3-independent, tabulation hashing provides the same constant-time guarantee for linear probing.[3]

Cuckoo hashing, another technique for implementing hash tables, guarantees constant time per lookup (regardless of the hash function). Insertions into a cuckoo hash table may fail, causing the entire table to be rebuilt, but such failures are sufficiently unlikely that the expected time per insertion (using either a truly random hash function or a hash function with logarithmic independence) is constant. With tabulation hashing, on the other hand, the best bound known on the failure probability is higher, high enough that insertions cannot be guaranteed to take constant expected time. Nevertheless, tabulation hashing is adequate to ensure the linear-expected-time construction of a cuckoo hash table for a static set of keys that does not change as the table is used.[3]
Notes

[2] For the sufficiency of 5-independent hashing for linear probing, see Pagh, Pagh & Ružić (2009). For examples of weaker hashing schemes that fail, see Pătraşcu & Thorup (2010).

References

• Lemire, Daniel (2010), The universality of iterated hashing over variable-length strings, arXiv:1008.1715.
A cryptographic hash function is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, the cryptographic hash value, such that an accidental or intentional change to the data will change the hash value. The data to be encoded is often called the "message," and the hash value is sometimes called the message digest or simply digest.

The ideal cryptographic hash function has four main or significant properties:

• it is easy (but not necessarily quick) to compute the hash value for any given message
• it is infeasible to generate a message that has a given hash
• it is infeasible to modify a message without changing the hash
• it is infeasible to find two different messages with the same hash

Cryptographic hash functions have many information security applications, notably in digital signatures, message authentication codes (MACs), and other forms of authentication. They can also be used as ordinary hash functions, to index data in hash tables, for fingerprinting, to detect duplicate data or uniquely identify files, and as checksums to detect accidental data corruption. Indeed, in information security contexts, cryptographic hash values are sometimes called (digital) fingerprints, checksums, or just hash values, even though all these terms stand for functions with rather different properties and purposes.

Properties

Most cryptographic hash functions are designed to take a string of any length as input and produce a fixed-length hash value.

A cryptographic hash function must be able to withstand all known types of cryptanalytic attack. As a minimum, it must have the following properties:

• Preimage resistance

  Given a hash \( h \) it should be difficult to find any message \( m \) such that \( h = \text{hash}(m) \). This concept is related to that of one-way function. Functions that lack this property are vulnerable to preimage attacks.

• Second-preimage resistance

  Given an input \( m_1 \) it should be difficult to find another input \( m_2 \) — where \( m_1 \neq m_2 \) — such that \( \text{hash}(m_1) = \text{hash}(m_2) \). This property is sometimes referred to as weak collision resistance, and functions that lack this property are vulnerable to second-preimage attacks.

• Collision resistance
It should be difficult to find two different messages \( m_1 \) and \( m_2 \) such that \( \text{hash}(m_1) = \text{hash}(m_2) \). Such a pair is called a cryptographic hash collision. This property is sometimes referred to as strong collision resistance. It requires a hash value at least twice as long as that required for preimage-resistance, otherwise collisions may be found by a birthday attack.

These properties imply that a malicious adversary cannot replace or modify the input data without changing its digest. Thus, if two strings have the same digest, one can be very confident that they are identical.

A function meeting these criteria may still have undesirable properties. Currently popular cryptographic hash functions are vulnerable to length-extension attacks: given \( h(m) \) and \( \text{len}(m) \) but not \( m \), by choosing a suitable \( m' \) an attacker can calculate \( h(m || m') \) where \( || \) denotes concatenation. This property can be used to break naive authentication schemes based on hash functions. The HMAC construction works around these problems.

Ideally, one may wish for even stronger conditions. It should be impossible for an adversary to find two messages with substantially similar digests; or to infer any useful information about the data, given only its digest. Therefore, a cryptographic hash function should behave as much as possible like a random function while still being deterministic and efficiently computable.

Checksum algorithms, such as CRC32 and other cyclic redundancy checks, are designed to meet much weaker requirements, and are generally unsuitable as cryptographic hash functions. For example, a CRC was used for message integrity in the WEP encryption standard, but an attack was readily discovered which exploited the linearity of the checksum.

**Degree of difficulty**

In cryptographic practice, "difficult" generally means "almost certainly beyond the reach of any adversary who must be prevented from breaking the system for as long as the security of the system is deemed important." The meaning of the term is therefore somewhat dependent on the application, since the effort that a malicious agent may put into the task is usually proportional to his expected gain. However, since the needed effort usually grows very quickly with the digest length, even a thousand-fold advantage in processing power can be neutralized by adding a few dozen bits to the latter.

In some theoretical analyses "difficult" has a specific mathematical meaning, such as not solvable in asymptotic polynomial time. Such interpretations of difficulty are important in the study of provably secure cryptographic hash functions but do not usually have a strong connection to practical security. For example, an exponential time algorithm can sometimes still be fast enough to make a feasible attack. Conversely, a polynomial time algorithm (e.g. one that requires \( n^{20} \) steps for \( n \)-digit keys) may be too slow for any practical use.

**Illustration**

An illustration of the potential use of a cryptographic hash is as follows: Alice poses a tough math problem to Bob, and claims she has solved it. Bob would like to try it himself, but would yet like to be sure that Alice is not bluffing. Therefore, Alice writes down her solution, appends a random nonce, computes its hash and tells Bob the hash value (whilst keeping the solution and nonce secret). This way, when Bob comes up with the solution himself a few days later, Alice can prove that she had the solution earlier by revealing the nonce to Bob. (This is an example of a simple commitment scheme; in actual practice, Alice and Bob will often be computer programs, and the secret would be something less easily spoofed than a claimed puzzle solution).
Applications

Verifying the integrity of files or messages

An important application of secure hashes is verification of message integrity. Determining whether any changes have been made to a message (or a file), for example, can be accomplished by comparing message digests calculated before, and after, transmission (or any other event).

For this reason, most digital signature algorithms only confirm the authenticity of a hashed digest of the message to be "signed." Verifying the authenticity of a hashed digest of the message is considered proof that the message itself is authentic.

A related application is password verification. Passwords are usually not stored in cleartext, for obvious reasons, but instead in digest form. To authenticate a user, the password presented by the user is hashed and compared with the stored hash.

File or data identifier

A message digest can also serve as a means of reliably identifying a file; several source code management systems, including Git, Mercurial and Monotone, use the sha1sum of various types of content (file content, directory trees, ancestry information, etc.) to uniquely identify them. Hashes are used to identify files on peer-to-peer filesharing networks. For example, in an ed2k link, an MD4-variant hash is combined with the file size, providing sufficient information for locating file sources, downloading the file and verifying its contents. Magnet links are another example. Such file hashes are often the top hash of a hash list or a hash tree which allows for additional benefits.

One of the main applications of a hash function is to allow the fast look-up of a data in a hash table. Being hash functions of a particular kind, cryptographic hash functions lend themselves well to this application too.

However, compared with standard hash functions, cryptographic hash functions tend to be much more expensive computationally. For this reason, they tend to be used in contexts where it is necessary for users to protect themselves against the possibility of forgery (the creation of data with the same digest as the expected data) by potentially malicious participants.

Pseudorandom generation and key derivation

Hash functions can also be used in the generation of pseudorandom bits, or to derive new keys or passwords from a single, secure key or password.

Hash functions based on block ciphers

There are several methods to use a block cipher to build a cryptographic hash function, specifically a one-way compression function.

The methods resemble the block cipher modes of operation usually used for encryption. All well-known hash functions, including MD4, MD5, SHA-1 and SHA-2 are built from block-cipher-like components designed for the purpose, with feedback to ensure that the resulting function is not bijective. SHA-3 finalists include functions with block-cipher-like components (e.g., Skein, BLAKE) and functions based on other designs (e.g., JH, Keccak).

A standard block cipher such as AES can be used in place of these custom block ciphers; that might be useful when an embedded system needs to implement both encryption and hashing with minimal code size or hardware area.

However, that approach can have costs in efficiency and security. The ciphers in hash functions are built for hashing: they use large keys and blocks, can efficiently change keys every block, and have been designed and vetted for resistance to related-key attacks. General-purpose ciphers tend to have different design goals. In particular, AES has key and block sizes that make it nontrivial to use to generate long hash values; AES encryption becomes less efficient when the key changes each block; and related-key attacks make it potentially less secure for use in a hash
A hash function must be able to process an arbitrary-length message into a fixed-length output. This can be achieved by breaking the input up into a series of equal-sized blocks, and operating on them in sequence using a one-way compression function. The compression function can either be specially designed for hashing or be built from a block cipher. A hash function built with the Merkle–Damgård construction is as resistant to collisions as is its compression function; any collision for the full hash function can be traced back to a collision in the compression function.

The last block processed should also be unambiguously length padded; this is crucial to the security of this construction. This construction is called the Merkle–Damgård construction. Most widely used hash functions, including SHA-1 and MD5, take this form.

The construction has certain inherent flaws, including length-extension and generate-and-paste attacks, and cannot be parallelized. As a result, many entrants in the current NIST hash function competition are built on different, sometimes novel, constructions.

**Use in building other cryptographic primitives**

Hash functions can be used to build other cryptographic primitives. For these other primitives to be cryptographically secure, care must be taken to build them correctly.

Message authentication codes (MACs) (also called keyed hash functions) are often built from hash functions. HMAC is such a MAC.

Just as block ciphers can be used to build hash functions, hash functions can be used to build block ciphers. Luby-Rackoff constructions using hash functions can be provably secure if the underlying hash function is secure.

Also, many hash functions (including SHA-1 and SHA-2) are built by using a special-purpose block cipher in a Davies-Meyer or other construction. That cipher can also be used in a conventional mode of operation, without the same security guarantees. See SHACAL, BEAR and LION.

Pseudorandom number generators (PRNGs) can be built using hash functions. This is done by combining a (secret) random seed with a counter and hashing it.

Some hash functions, such as Skein, Keccak, and RadioGatún output an arbitrarily long stream and can be used as a stream cipher, and stream ciphers can also be built from fixed-length digest hash functions. Often this is done by first building a cryptographically secure pseudorandom number generator and then using its stream of random bytes as keystream. SEAL is a stream cipher that uses SHA-1 to generate internal tables, which are then used in a keystream generator more or less unrelated to the hash algorithm. SEAL is not guaranteed to be as strong (or weak) as SHA-1.
Concatenation of cryptographic hash functions

Concatenating outputs from multiple hash functions provides collision resistance as good as the strongest of the algorithms included in the concatenated result. For example, older versions of TLS/SSL use concatenated MD5 and SHA-1 sums; that ensures that a method to find collisions in one of the functions doesn't allow forging traffic protected with both functions.

For Merkle-Damgård hash functions, the concatenated function is as collision-resistant as its strongest component,[2] but not more collision-resistant.[2] Joux[3] noted that 2-collisions lead to n-collisions: if it is feasible to find two messages with the same MD5 hash, it is effectively no more difficult to find as many messages as the attacker desires with identical MD5 hashes. Among the n messages with the same MD5 hash, there is likely to be a collision in SHA-1. The additional work needed to find the SHA-1 collision (beyond the exponential birthday search) is polynomial. This argument is summarized by Finney[4]. A more current paper and full proof of the security of such a combined construction gives a clearer and more complete explanation of the above.[5]

Cryptographic hash algorithms

There is a long list of cryptographic hash functions, although many have been found to be vulnerable and should not be used. Even if a hash function has never been broken, a successful attack against a weakened variant thereof may undermine the experts’ confidence and lead to its abandonment. For instance, in August 2004 weaknesses were found in a number of hash functions that were popular at the time, including SHA-0, RIPEMD, and MD5. This has called into question the long-term security of later algorithms which are derived from these hash functions — in particular, SHA-1 (a strengthened version of SHA-0), RIPEMD-128, and RIPEMD-160 (both strengthened versions of RIPEMD). Neither SHA-0 nor RIPEMD are widely used since they were replaced by their strengthened versions. As of 2009, the two most commonly used cryptographic hash functions are MD5 and SHA-1. However, MD5 has been broken; an attack against it was used to break SSL in 2008.[6]

The SHA-0 and SHA-1 hash functions were developed by the NSA. In February 2005, a successful attack on SHA-1 was reported, finding collisions in about 269 hashing operations, rather than the 280 expected for a 160-bit hash function. In August 2005, another successful attack on SHA-1 was reported, finding collisions in 263 operations. Theoretical weaknesses of SHA-1 exist as well,[7][8] suggesting that it may be practical to break within years. New applications can avoid these problems by using more advanced members of the SHA family, such as SHA-2, or using techniques such as randomized hashing[9][10] that do not require collision resistance.

However, to ensure the long-term robustness of applications that use hash functions, there is a competition to design a replacement for SHA-2, which will be given the name SHA-3 and become a FIPS standard around 2012.[11]

Some of the following algorithms are used often in cryptography; consult the article for each specific algorithm for more information on the status of each algorithm. Note that this list does not include candidates in the current NIST hash function competition. For additional hash functions see the box at the bottom of the page.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Output size (bits)</th>
<th>Internal state size</th>
<th>Block size</th>
<th>Length size</th>
<th>Word size</th>
<th>Collision attacks (complexity)</th>
<th>Preimage attacks (complexity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOST</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>32</td>
<td>Yes ($2^{105}[12]$)</td>
<td>Yes ($2^{192}[12]$)</td>
</tr>
<tr>
<td>HAVAL</td>
<td>256/224/192/160/128</td>
<td>256</td>
<td>1024</td>
<td>64</td>
<td>32</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MD2</td>
<td>128</td>
<td>384</td>
<td>128</td>
<td>No</td>
<td>32</td>
<td>Yes ($2^{63.3}[13]$)</td>
<td>Yes ($2^{73}[14]$)</td>
</tr>
<tr>
<td>MD5</td>
<td>128</td>
<td>128</td>
<td>512</td>
<td>64</td>
<td>32</td>
<td>Yes ($2^{20.6}[17]$)</td>
<td>Yes ($2^{13.4}[18]$)</td>
</tr>
<tr>
<td>PANAMA</td>
<td>256</td>
<td>8736</td>
<td>256</td>
<td>No</td>
<td>32</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>RadioGatún</td>
<td>up to 608/1216 (19 words)</td>
<td>58 words</td>
<td>3 words</td>
<td>No</td>
<td>1-64</td>
<td>With flaws ($2^{352}$ or $2^{704}[19]$)</td>
<td></td>
</tr>
<tr>
<td>RIPEMD</td>
<td>128</td>
<td>128</td>
<td>512</td>
<td>64</td>
<td>32</td>
<td>Yes ($2^{18}[13]$)</td>
<td></td>
</tr>
<tr>
<td>RIPEMD-128/256</td>
<td>128/256</td>
<td>128/256</td>
<td>512</td>
<td>64</td>
<td>32</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>RIPEMD-160/320</td>
<td>160/320</td>
<td>160/320</td>
<td>512</td>
<td>64</td>
<td>32</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SHA-0</td>
<td>160</td>
<td>160</td>
<td>512</td>
<td>64</td>
<td>32</td>
<td>Yes ($2^{13.6}[20]$)</td>
<td></td>
</tr>
<tr>
<td>SHA-1</td>
<td>160</td>
<td>160</td>
<td>512</td>
<td>64</td>
<td>32</td>
<td>Yes ($2^{51}[21]$)</td>
<td>No</td>
</tr>
<tr>
<td>SHA-256/224</td>
<td>256/224</td>
<td>256</td>
<td>512</td>
<td>64</td>
<td>32</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SHA-512/384</td>
<td>512/384</td>
<td>512</td>
<td>1024</td>
<td>128</td>
<td>64</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Tiger(2)-192/160/128</td>
<td>192/160/128</td>
<td>192</td>
<td>512</td>
<td>64</td>
<td>64</td>
<td>Yes ($2^{62.19}[22]$)</td>
<td>Yes ($2^{184.3}[16]$)</td>
</tr>
<tr>
<td>WHIRLPOOL</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>256</td>
<td>8</td>
<td>Yes ([23])</td>
<td></td>
</tr>
</tbody>
</table>

Note: The internal state here means the "internal hash sum" after each compression of a data block. Most hash algorithms also internally use some additional variables such as length of the data compressed so far since that is needed for the length padding in the end. See the Merkle-Damgård construction for details.

References

[1] Note that any two messages that collide the concatenated function also collide each component function, by the nature of concatenation. For example, if concat(sha1(message1), md5(message1)) == concat(sha1(message2), md5(message2)) then sha1(message1) == sha1(message2) and md5(message1) == md5(message2). The concatenated function could have other problems that the strongest hash lacks -- for example, it might leak information about the message when the strongest component does not, or it might be detectably nonrandom when the strongest component is not -- but it can't be less collision-resistant.

[2] More generally, if an attack can produce a collision in one hash function's internal state, attacking the combined construction is only as difficult as a birthday attack against the other function(s). For the detailed argument, see the Joux and Finney references that follow.


Further reading

- Christof Paar, Jan Pelzl, "Hash Functions" (http://wiki.crypto.rub.de/Buch/movies.php), Chapter 11 of "Understanding Cryptography, A Textbook for Students and Practitioners". (companion web site contains online cryptography course that covers hash functions), Springer, 2009.
Sets

Set (abstract data type)

In computer science, a set is an abstract data structure that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set. Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set.

Some set data structures are designed for static sets that do not change with time, and allow only query operations — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called dynamic or mutable sets, allow also the insertion and/or deletion of elements from the set.

A set can be implemented in many ways. For example, one can use a list, ignoring the order of the elements and taking care to avoid repeated values. Sets are often implemented using various flavors of trees, tries, or hash tables.

A set can be seen, and implemented, as a (partial) associative array, in which the value of each key-value pair has the unit type.

In type theory, sets are generally identified with their indicator function: accordingly, a set of values of type \( A \) may be denoted by \( 2^A \) or \( P(A) \). (Subtypes and subsets may be modeled by refinement types, and quotient sets may be replaced by setoids.) The characteristic function \( F \) of a set \( S \) is defined as:

\[
F(x) = \begin{cases} 
1, & \text{if } x \in S \\
0, & \text{if } x \not\in S 
\end{cases}
\]

In theory, many other abstract data structures can be viewed as set structures with additional operations and/or additional axioms imposed on the standard operations. For example, an abstract heap can be viewed as a set structure with a \( \min(S) \) operation that returns the element of smallest value.

Operations

Core set-theoretical operations

One may define the operations of the algebra of sets:

- \( \text{union}(S, T) \): returns the union of sets \( S \) and \( T \).
- \( \text{intersection}(S, T) \): returns the intersection of sets \( S \) and \( T \).
- \( \text{difference}(S, T) \): returns the difference of sets \( S \) and \( T \).
- \( \text{subset}(S, T) \): a predicate that tests whether the set \( S \) is a subset of set \( T \).
Static sets

Typical operations that may be provided by a static set structure $S$ are:

- $\text{is\_element\_of}(x, S)$: checks whether the value $x$ is in the set $S$.
- $\text{is\_empty}(S)$: checks whether the set $S$ is empty.
- $\text{size}(S)$ or $\text{cardinality}(S)$: returns the number of elements in $S$.
- $\text{enumerate}(S)$: yields the elements of $S$ in some arbitrary order.
- $\text{pick}(S)$: returns an arbitrary element of $S$.
- $\text{build}(x_1, x_2, \ldots, x_n)$: creates a set structure with values $x_1, x_2, \ldots, x_n$.

The $\text{enumerate}$ operation may return a list of all the elements, or an iterator, a procedure object that returns one more value of $S$ at each call.

Dynamic sets

Dynamic set structures typically add:

- $\text{create}()$: creates a new, initially empty set structure.
- $\text{create\_with\_capacity}(n)$: creates a new set structure, initially empty but capable of holding up to $n$ elements.
- $\text{create\_from}(\text{collection})$: creates a new set structure containing all the elements of the given collection.
- $\text{add}(S, x)$: adds the element $x$ to $S$, if it is not there already.
- $\text{remove}(S, x)$: removes the element $x$ from $S$, if it is there.
- $\text{capacity}(S)$: returns the maximum number of values that $S$ can hold.

Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted.

Additional operations

There are many other operations that can (in principle) be defined in terms of the above, such as:

- $\text{pop}(S)$: returns an arbitrary element of $S$, deleting it from $S$.
- $\text{find}(S, P)$: returns an element of $S$ that satisfies a given predicate $P$.
- $\text{clear}(S)$: delete all elements of $S$.
- $\text{equal}(S_1, S_2)$: checks whether the two given sets are equal (i.e. contain all and only the same elements).

Other operations can be defined for sets with elements of a special type:

- $\text{sum}(S)$: returns the sum of all elements of $S$ (for some definition of "sum").
- $\text{nearest}(S, x)$: returns the element of $S$ that is closest in value to $x$ (by some criterion).

Implementations

Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations. Some implementations are designed to improve the efficiency of very specialized operations, such as $\text{nearest}$ or $\text{union}$. Implementations described as "general use" typically strive to optimize the $\text{element\_of}$, $\text{add}$, and $\text{delete}$ operation.

Sets are commonly implemented in the same way as associative arrays, namely, a self-balancing binary search tree for sorted sets (which has $O(\log n)$ for most operations), or a hash table for unsorted sets (which has $O(1)$ average-case, but $O(n)$ worst-case, for most operations). A sorted linear hash table\(^1\) may be used to provide deterministically ordered sets. Sets may be viewed as associative arrays with the elements of the set as the keys and the values being undefined or null.
Other popular methods include arrays. In particular a subset of the integers 1..n can be implemented efficiently as an n-bit bit array, which also support very efficient union and intersection operations. A Bloom map implements a set probabilistically, using a very compact representation but risking a small chance of false positives on queries. The Boolean set operations can be implemented in terms of more elementary operations (pop, clear, and add), but specialized algorithms may yield lower asymptotic time bounds. If sets are implemented as sorted lists, for example, the naive algorithm for union(S, T) will take code proportional to the length m of S times the length n of T, whereas a variant of the list merging algorithm will do the job in time proportional to m+n. Moreover, there are specialized set data structures (such as the union-find data structure) that are optimized for one or more of these operations, at the expense of others.

**Language support**

One of the earliest languages to support sets was Pascal; many languages now include it, whether in the core language or in a standard library.

- Java offers the Set interface to support sets (with the HashSet class implementing it using a hash table), and the SortedSet sub-interface to support sorted sets (with the TreeSet class implementing it using a binary search tree).
- Apple’s Foundation framework (part of Cocoa) provides the Objective-C classes NSSet, NSmutableSet, NScountedSet, NSOrderedSet, and NSmutableOrderedSet. The CoreFoundation APIs provide the CFSet and CFMutableSet types for use in C.
- Python has built-in set and frozenset types since 2.4, and since Python 3.0 and 2.7, supports non-empty set literals using a curly-bracket syntax, e.g.: { x, y, z }.
- The .NET Framework provides the generic HashSet and SortedSet classes that implement the generic ISet interface.
- Ruby’s standard library includes a set module which contains Set and SortedSet classes that implement sets using hash tables, the latter allowing iteration in sorted order.
- OCaml’s standard library contains a Set module, which implements a functional set data structure using binary search trees.
- The GHC implementation of Haskell provides a Data.Set module, which implements a functional set data structure using binary search trees.
- The TCL Tcllib package provides a set module which implements a set data structure based upon TCL lists.

As noted in the previous section, in languages which do not directly support sets but do support associative arrays, sets can be emulated using associative arrays, by using the elements as keys, and using a dummy value as the values, which are ignored.

**In C++**

In C++, the Standard Template Library (STL) provides the set template class, which implements a sorted set using a binary search tree; SGI’s STL also provides the hash_set template class, which implements a set using a hash table.

In sets, the elements themselves are the keys, in contrast to sequenced containers, where elements are accessed using their (relative or absolute) position. Set elements must have a strict weak ordering.

Some of the member functions in C++ and their description is given in the table below:
### Set member functions

<table>
<thead>
<tr>
<th>Signature(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator begin();</td>
<td>Returns an iterator to the first element of the set.</td>
</tr>
<tr>
<td>iterator end();</td>
<td>Returns an iterator just before the end of the set.</td>
</tr>
<tr>
<td>bool empty() const;</td>
<td>Checks if the set container is empty (i.e. has a size of 0).</td>
</tr>
<tr>
<td>iterator find(const key_type &amp;x) const;</td>
<td>Searches the container for an element ( x ) and if found, returns an iterator to it, or else returns an iterator to set::end</td>
</tr>
<tr>
<td>void insert ( Input Iterator first, Input Iterator last );</td>
<td>Inserts an element into the set. The first version returns pair, with its member pair::first set to an iterator pointing to either the newly inserted element or to the element that already had its same value in the set. The pair::second element in the pair is set to true if a new element was inserted or false if an element with the same value existed. And the second version returns an iterator either pointing to newly inserted element or to element having same value in set.</td>
</tr>
<tr>
<td>pair&lt;iterator, bool&gt; insert ( const value_type&amp; a );</td>
<td></td>
</tr>
<tr>
<td>iterator insert ( position(iterator), const value_type&amp; a );</td>
<td></td>
</tr>
<tr>
<td>void clear();</td>
<td>Removes all elements in the set, making its size 0.</td>
</tr>
</tbody>
</table>

### Template parameters

Here value_type and iterator are member types defined in set containers. Firstly, value to be used to initialize inserted element. Then position of first element compared for the operation. It actually does not give the position where element is to be inserted but just an indication of possible insertion position in the container so as to make efficient insertion operation. Template type can be any type of input iterator. Iterators specify a range of elements and copies of these in range [first, last) are inserted in set.

### Multiset

A variation of the set is the **multiset** or **bag**, which is the same as a set data structure, but allows repeated ("equal") values. It is possible for objects in computer science to be considered "equal" under some equivalence relation but still distinct under another relation. Some types of multiset implementations will store distinct equal objects as separate items in the data structure; while others will collapse it down to one version (the first one encountered) and keep a positive integer count of the multiplicity of the element.

- C++'s Standard Template Library provides the **multiset** class for the sorted multiset, and SGI's STL provides the **hash_multiset** class, which implements a multiset using a hash table.
- For Java, third-party libraries provide multiset functionality:
  - Apache Commons Collections provides the **Bag** and **SortedBag** interfaces, with implementing classes like **HashBag** and **TreeBag**.
  - Google Collections provides the **Multiset** interface, with implementing classes like **HashMultiset** and **TreeMultiset**.
  - Apple provides the **NSCountedSet** class as part of Cocoa, and the **CFBag** and **CFMutableBag** types as part of CoreFoundation.
  - Python’s standard library includes **collections.Counter**, which is similar to a multiset.

Where a multiset data structure is not available, a workaround is to use a regular set, but override the equality predicate of its items to always return "not equal" on distinct objects (however, such will still not be able to store multiple occurrences of the same object) or use an associative array mapping the values to their integer multiplicities (this will not be able to distinguish between equal elements at all).
References

Bit array

A bit array (also known as a bitmap, a bitset, or a bitstring) is an array data structure that compactly stores
individual bits (boolean values). It implements a simple set data structure storing a subset of \( \{1,2,\ldots,n\} \) and is
effective at exploiting bit-level parallelism in hardware to perform operations quickly. A typical bit array stores
\( kw \) bits, where \( w \) is the number of bits in the unit of storage, such as a byte or word, and \( k \) is some nonnegative integer.
If \( w \) does not divide the number of bits to be stored, some space is wasted due to internal fragmentation.

Basic operations

Although most machines are not able to address individual bits in memory, nor have instructions to manipulate
single bits, each bit in a word can be singled out and manipulated using bitwise operations. In particular:

- OR can be used to set a bit to one: \( 11101010 \ OR \ 00000100 = 11101110 \)
- AND can be used to set a bit to zero: \( 11101010 \ AND \ 11111111 = 11101000 \)
- AND together with zero-testing can be used to determine if a bit is set:

\[
11101010 \ AND \ 00000000 = 0
\]
\[
11101010 \ AND \ 00000010 = 00000010 ≠ 0
\]

- XOR can be used to invert or toggle a bit:

\[
11101010 \ XOR \ 00000100 = 11101110
\]
\[
11101110 \ XOR \ 00000100 = 11101010
\]

To obtain the bit mask needed for these operations, we can use a bit shift operator to shift the number 1 to the left by
the appropriate number of places, as well as bitwise negation if necessary.

We can view a bit array as a subset of \( \{1,2,\ldots,n\} \), where a 1 bit indicates a number in the set and a 0 bit a number not
in the set. This set data structure uses about \( n/w \) words of space, where \( w \) is the number of bits in each machine word.
Whether the least significant bit or the most significant bit indicates the smallest-index number is largely irrelevant,
but the former tends to be preferred.
Given two bit arrays of the same size representing sets, we can compute their union, intersection, and set-theoretic difference using \( n/w \) simple bit operations each (\( 2n/w \) for difference), as well as the complement of either:

```plaintext
for i from 0 to n/w-1
    complement_a[i] := not a[i]
    union[i] := a[i] or b[i]
    intersection[i] := a[i] and b[i]
    difference[i] := a[i] and (not b[i])
```

If we wish to iterate through the bits of a bit array, we can do this efficiently using a doubly nested loop that loops through each word, one at a time. Only \( n/w \) memory accesses are required:

```plaintext
for i from 0 to n/w-1
    index := 0  // if needed
    word := a[i]
    for b from 0 to w-1
        value := word and 1 ≠ 0
        word := word shift right 1
        // do something with value
        index := index + 1  // if needed
```

Both of these code samples exhibit ideal locality of reference, and so get a large performance boost from a data cache. If a cache line is \( k \) words, only about \( n/wk \) cache misses will occur.

### More complex operations

#### Population / Hamming weight

If we wish to find the number of 1 bits in a bit array, sometimes called the population count or Hamming weight, there are efficient branch-free algorithms that can compute the number of bits in a word using a series of simple bit operations. We simply run such an algorithm on each word and keep a running total. Counting zeros is similar. See the Hamming weight article for examples of an efficient implementation.

#### Sorting

Similarly, sorting a bit array is trivial to do in \( O(n) \) time using counting sort — we count the number of ones \( k \), fill the last \( k/w \) words with ones, set only the low \( k \mod w \) bits of the next word, and set the rest to zero.

#### Inversion

Vertical flipping of a one-bit-per-pixel image, or some FFT algorithms, require to flip the bits of individual words (so \( b_{31} b_{30} \ldots b_0 \) becomes \( b_0 \ldots b_{30} b_{31} \)). When this operation is not available on the processor, it's still possible to proceed by successive passes, in this example on 32 bits:

- exchange two 16bit halfwords
- exchange bytes by pairs (0xddccbbaa -> 0xccddaabb)
- ...
- swap bits by pairs
- swap bits (b_{31} b_{30} ... b_1 b_0 -> b_{30} b_{31} ... b_0 b_1)

The last operation can be written \(((x&0x55555555)<<1) | (x&0xaaaaaaaa)>>1))\).
Find first one

The find first one operation identifies the one bit of the smallest index, that is the least significant bit having a one value. The find first zero operation similarly identifies the first zero bit. Each operation can be used instead of the other by complementing the input first.

Doing this operation quickly is useful in contexts such as priority queues. The application in this context is to identify the highest priority queue that is not empty. The find-first-one operation starting from the most significant bit is equivalent to computing the base 2 logarithm.

Many machines can quickly perform the operation on a single word using a single instruction. For example the x86 instruction bsr (bit scan reverse) finds the most significant one bit. The ffs (find first set) function in POSIX operating systems finds the least significant one. To expand such an instruction or function to longer arrays, one can find the first nonzero word and then run find first one on that word.

On machines that use two's complement arithmetic, which includes all conventional CPUs, find first one can be performed quickly by anding a word with its two's complement, that is, performing (w AND ¬w). This results in a word with only the least significant (rightmost) bit set of the bits that were set in w. For instance, if the original value were 6 (110), after this operation the result would be 2 (010). See Gosper's Hack for an example of this technique in use.

Compression

Large bit arrays tend to have long streams of zeroes or ones. This phenomenon wastes storage and processing time. Run-length encoding is commonly used to compress these long streams. However, by compressing bit arrays too aggressively we run the risk of losing the benefits due to bit-level parallelism (vectorization). Thus, instead of compressing bit arrays as streams of bits, we might compress them as streams bytes or words (see Bitmap index (compression)).

Examples:

- compressedbitset [2]: WAH Compressed BitSet for Java
- javaewah [3]: A compressed alternative to the Java BitSet class (using Enhanced WAH)
- CONCISE [4]: COmpressed 'N' Composable Integer Set, another bitmap compression scheme for Java
- EWAHBoolArray [5]: A compressed bitmap/bitset class in C++

Advantages and disadvantages

Bit arrays, despite their simplicity, have a number of marked advantages over other data structures for the same problems:

- They are extremely compact; few other data structures can store n independent pieces of data in n/w words.
- They allow small arrays of bits to be stored and manipulated in the register set for long periods of time with no memory accesses.
- Because of their ability to exploit bit-level parallelism, limit memory access, and maximally use the data cache, they often outperform many other data structures on practical data sets, even those that are more asymptotically efficient.

However, bit arrays aren't the solution to everything. In particular:

- Without compression, they are wasteful set data structures for sparse sets (those with few elements compared to their range) in both time and space. For such applications, compressed bit arrays, Judy arrays, tries, or even Bloom filters should be considered instead.
- Accessing individual elements can be expensive and difficult to express in some languages. If random access is more common than sequential and the array is relatively small, a byte array may be preferable on a machine with
byte addressing. A word array, however, is probably not justified due to the huge space overhead and additional cache misses it causes, unless the machine only has word addressing.

Applications

Because of their compactness, bit arrays have a number of applications in areas where space or efficiency is at a premium. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values.

Bit arrays are used for priority queues, where the bit at index \( k \) is set if and only if \( k \) is in the queue; this data structure is used, for example, by the Linux kernel, and benefits strongly from a find-first-zero operation in hardware.

Bit arrays can be used for the allocation of memory pages, inodes, disk sectors, etc. In such cases, the term bitmap may be used. However, this term is frequently used to refer to raster images, which may use multiple bits per pixel.

Another application of bit arrays is the Bloom filter, a probabilistic set data structure that can store large sets in a small space in exchange for a small probability of error. It is also possible to build probabilistic hash tables based on bit arrays that accept either false positives or false negatives.

Bit arrays and the operations on them are also important for constructing succinct data structures, which use close to the minimum possible space. In this context, operations like finding the \( n \)th 1 bit or counting the number of 1 bits up to a certain position become important.

Bit arrays are also a useful abstraction for examining streams of compressed data, which often contain elements that occupy portions of bytes or are not byte-aligned. For example, the compressed Huffman coding representation of a single 8-bit character can be anywhere from 1 to 255 bits long.

In information retrieval, bit arrays are a good representation for the posting lists of very frequent terms. If we compute the gaps between adjacent values in a list of strictly increasing integers and encode them using unary coding, the result is a bit array with a 1 bit in the \( n \)th position if and only if \( n \) is in the list. The implied probability of a gap of \( n \) is \( 1/2^n \). This is also the special case of Golomb coding where the parameter \( M \) is 1; this parameter is only normally selected when \( \log(2-p)/\log(1-p) \leq 1 \), or roughly the term occurs in at least 38% of documents.

Language support

The C programming language's bitfields, pseudo-objects found in structs with size equal to some number of bits, are in fact small bit arrays; they are limited in that they cannot span words. Although they give a convenient syntax, the bits are still accessed using bitwise operators on most machines, and they can only be defined statically (like C's static arrays, their sizes are fixed at compile-time). It is also a common idiom for C programmers to use words as small bit arrays and access bits of them using bit operators. A widely available header file included in the X11 system, xtrapbits.h, is "a portable way for systems to define bit field manipulation of arrays of bits.". A more explanatory description of aforementioned approach can be found in the comp.lang.c faq.

In C++, although individual bools typically occupy the same space as a byte or an integer, the STL type vector<bool> is a partial template specialization in which bits are packed as a space efficiency optimization. Since bytes (and not bits) are the smallest addressable unit in C++, the [] operator does not return a reference to an element, but instead returns a proxy reference. This might seem a minor point, but it means that vector<bool> is not a standard STL container, which is why the use of vector<bool> is generally discouraged. Another unique STL class, bitset, creates a vector of bits fixed at a particular size at compile-time, and in its interface and syntax more resembles the idiomatic use of words as bit sets by C programmers. It also has some additional power, such as the ability to efficiently count the number of bits that are set. The Boost C++ Libraries provide a dynamic_bitset class whose size is specified at run-time.
The D programming language provides bit arrays in both of its competing standard libraries. In Phobos, they are provided in `std.bitmanip`, and in Tango, they are provided in `tango.core.BitArray`. As in C++, the [] operator does not return a reference, since individual bits are not directly addressable on most hardware, but instead returns a `bool`.

In Java, the class `BitSet` creates a bit array that is then manipulated with functions named after bitwise operators familiar to C programmers. Unlike the `bitset` in C++, the Java `BitSet` does not have a "size" state (it has an effectively infinite size, initialized with 0 bits); a bit can be set or tested at any index. In addition, there is a class `EnumSet`, which represents a Set of values of an enumerated type internally as a bit vector, as a safer alternative to bitfields.

The .NET Framework supplies a `BitArray` collection class. It stores boolean values, supports random access and bitwise operators, can be iterated over, and its `Length` property can be changed to grow or truncate it.

Although Standard ML has no support for bit arrays, Standard ML of New Jersey has an extension, the `BitArray` structure, in its SML/NJ Library. It is not fixed in size and supports set operations and bit operations, including, unusually, shift operations.

Haskell likewise currently lacks standard support for bitwise operations, but both GHC and Hugs provide a `Data.Bits` module with assorted bitwise functions and operators, including shift and rotate operations and an "unboxed" array over boolean values may be used to model a Bit array, although this lacks support from the former module.

In Perl, strings can be used as expandable bit arrays. They can be manipulated using the usual bitwise operators (~ | & ^),[9] and individual bits can be tested and set using the `vec` function.[10]


References

External links
- bitarray module (http://pypi.python.org/pypi/bitarray) for Python
- `vector<bool>` Is Nonconforming, and Forces Optimization Choice (http://www.gotw.ca/publications/N1185.pdf)
Bloom filter

A Bloom filter, conceived by Burton Howard Bloom in 1970,[1] is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not; i.e. a query returns either "inside set (may be wrong)" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a counting filter). The more elements that are added to the set, the larger the probability of false positives.

Algorithm description

An empty Bloom filter is a bit array of $m$ bits, all set to 0. There must also be $k$ different hash functions defined, each of which maps or hashes some set element to one of the $m$ array positions with a uniform random distribution.

To add an element, feed it to each of the $k$ hash functions to get $k$ array positions. Set the bits at all these positions to 1.

To query for an element (test whether it is in the set), feed it to each of the $k$ hash functions to get $k$ array positions. If any of the bits at these positions are 0, the element is not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements.

The requirement of designing $k$ different independent hash functions can be prohibitive for large $k$. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields. Alternatively, one can pass $k$ different initial values (such as 0, 1, ..., $k$ - 1) to a hash function that takes an initial value; or add (or append) these values to the key. For larger $m$ and/or $k$, independence among the hash functions can be relaxed with negligible increase in false positive rate (Dillinger & Manolios (2004a), Kirsch & Mitzenmacher (2006)). Specifically, Dillinger & Manolios (2004b) show the effectiveness of deriving the $k$ indices using enhanced double hashing or triple hashing, variants of double hashing that are effectively simple random number generators seeded with the two or three hash values.

Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to $k$ bits, and although setting any one of those $k$ bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which are not permitted. In this approach re-adding a previously removed item is not possible, as one would have to remove it from the "removed" filter.

However, it is often the case that all the keys are available but are expensive to enumerate (for example, requiring many disk reads). When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.
Space and time advantages

While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers. A Bloom filter with 1% error and an optimal value of $k$, in contrast, requires only about 9.6 bits per element — regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. If a 1% false-positive rate seems too high, adding about 4.8 bits per element decreases it by ten times.

However, if the number of potential values is small and many of them can be in the set, the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element. Note also that hash tables gain a space and time advantage if they begin ignoring collisions and store only whether each bucket contains an entry; in this case, they have effectively become Bloom filters with $k = 1$.

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, $O(1)$, completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its $k$ lookups are independent and can be parallelized.

To understand its space efficiency, it is instructive to compare the general Bloom filter with its special case when $k = 1$. If $k = 1$, then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter ($k$ greater than 1) allows many more bits to be set while still maintaining a low false positive rate; if the parameters ($k$ and $m$) are chosen well, about half of the bits will be set, and these will be apparently random, minimizing redundancy and maximizing information content.
### Probability of false positives

Assume that a hash function selects each array position with equal probability. If \( m \) is the number of bits in the array, the probability that a certain bit is not set to one by a certain hash function during the insertion of an element is then

\[
1 - \frac{1}{m}.
\]

The probability that it is not set by any of the hash functions is

\[
\left(1 - \frac{1}{m}\right)^k.
\]

If we have inserted \( n \) elements, the probability that a certain bit is still 0 is

\[
\left(1 - \frac{1}{m}\right)^{kn};
\]

the probability that it is 1 is therefore

\[
1 - \left(1 - \frac{1}{m}\right)^{kn}.
\]

Now test membership of an element that is not in the set. Each of the \( k \) array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is often given as

\[
\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx (1 - e^{-kn/m})^k.
\]

This is not strictly correct as it assumes independence for the probabilities of each bit being set. However, assuming it is a close approximation we have that the probability of false positives decreases as \( m \) (the number of bits in the array) increases, and increases as \( n \) (the number of inserted elements) increases. For a given \( m \) and \( n \), the value of \( k \) (the number of hash functions) that minimizes the probability is

\[
\frac{m}{n} \ln 2 \approx 0.7 \frac{m}{n},
\]

which gives the false positive probability of

\[
2^{-k} \approx 0.6185^{m/n}.
\]

The required number of bits \( m \), given \( n \) (the number of inserted elements) and a desired false positive probability \( p \) (and assuming the optimal value of \( k \) is used) can be computed by substituting the optimal value of \( k \) in the
probability expression above:

\[ p = \left( 1 - e^{- (m/n \ln 2) n / m} \right)^{m/n \ln 2} \]

which can be simplified to:

\[ \ln p = -\frac{m}{n} (\ln 2)^2. \]

This results in:

\[ m = -\frac{n \ln p}{(\ln 2)^2}. \]

This means that in order to maintain a fixed false positive probability, the length of a Bloom filter must grow linearly with the number of elements being filtered. While the above formula is asymptotic (i.e. applicable as \( m, n \rightarrow \infty \)), the agreement with finite values of \( m, n \) is also quite good; the false positive probability for a finite bloom filter with \( m \) bits, \( n \) elements, and \( k \) hash functions is at most

\[ \left( 1 - e^{-k(n+0.5)/(m-1)} \right)^k. \]

So we can use the asymptotic formula if we pay a penalty for at most half an extra element and at most one fewer bit.

Goel & Gupta (2010).

**Interesting properties**

- Unlike sets based on hash tables, any Bloom filter can represent the entire universe of elements. In this case, all bits are 1. Another consequence of this property is that `add` never fails due to the data structure “filling up.” However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, so a negative value is never returned. At this point, the Bloom filter completely ceases to differentiate between differing inputs, and is functionally useless.

- Union and intersection of Bloom filters with the same size and set of hash functions can be implemented with bitwise OR and AND operations, respectively. The union operation on Bloom filters is lossless in the sense that the resulting Bloom filter is the same as the Bloom filter created from scratch using the union of the two sets. The intersect operation satisfies a weaker property: the false positive probability in the resulting Bloom filter is at most the false-positive probability in one of the constituent Bloom filters, but may be larger than the false positive probability in the Bloom filter created from scratch using the intersection of the two sets.

**Examples**

Google BigTable uses Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.\(^2\)

The Squid Web Proxy Cache uses Bloom filters for cache digests.\(^3\),\(^4\)

The Venti archival storage system uses Bloom filters to detect previously-stored data.\(^5\)

The SPIN model checker uses Bloom filters to track the reachable state space for large verification problems.\(^6\)

The Google Chrome web browser uses Bloom filters to speed up its Safe Browsing service.\(^7\)
Alternatives

Classic Bloom filters use $1.44 \log_2(1/\varepsilon)$ bits of space per inserted key, where $\varepsilon$ is the false positive rate of the Bloom filter. However, the space that is strictly necessary for any data structure playing the same role as a Bloom filter is only $\log_2(1/\varepsilon)$ per key (Pagh, Pagh & Rao 2005). Hence Bloom filters use 44% more space than a hypothetical equivalent optimal data structure. The number of hash functions used to achieve a given false positive rate $\varepsilon$ is proportional to $1/\varepsilon$ which is not optimal as it has been proved that an optimal data structure would need only a constant number of hash functions independent of the false positive rate.

Stern & Dill (1996) describe a probabilistic structure based on hash tables, hash compaction, which Dillinger & Manolios (2004b) identify as significantly more accurate than a Bloom filter when each is configured optimally. Dillinger and Manolios, however, point out that the reasonable accuracy of any given Bloom filter over a wide range of numbers of additions makes it attractive for probabilistic enumeration of state spaces of unknown size. Hash compaction is, therefore, attractive when the number of additions can be predicted accurately; however, despite being very fast in software, hash compaction is poorly-suited for hardware because of worst-case linear access time.

Putze, Sanders & Singler (2007) have studied some variants of Bloom filters that are either faster or use less space than classic Bloom filters. The basic idea of the fast variant is to locate the $k$ hash values associated with each key into one or two blocks having the same size as processor’s memory cache blocks (usually 64 bytes). This will presumably improve performance by reducing the number of potential memory cache misses. The proposed variants have however the drawback of using about 32% more space than classic Bloom filters.

The space efficient variant relies on using a single hash function that generates for each key a value in the range $[0, n/\varepsilon]$ where $\varepsilon$ is the requested false positive rate. The sequence of values is then sorted and compressed using Golomb coding (or some other compression technique) to occupy a space close to $n \log_2(1/\varepsilon)$ bits. To query the Bloom filter for a given key, it will suffice to check if its corresponding value is stored in the Bloom filter. Decompressing the whole Bloom filter for each query would make this variant totally unusable. To overcome this problem the sequence of values is divided into small blocks of equal size that are compressed separately. At query time only half a block will need to be decompressed on average. Because of decompression overhead, this variant may be slower than classic Bloom filters but this may be compensated by the fact that a single hash function need to be computed.

Another alternative to classic Bloom filter is the one based on space efficient variants of cuckoo hashing. In this case once the hash table is constructed, the keys stored in the hash table are replaced with short signatures of the keys. Those signatures are strings of bits computed using a hash function applied on the keys.

Extensions and applications

Counting filters

Counting filters provide a way to implement a delete operation on a Bloom filter without recreating the filter afresh. In a counting filter the array positions (buckets) are extended from being a single bit, to an $n$-bit counter. In fact, regular Bloom filters can be considered as counting filters with a bucket size of one bit. Counting filters were introduced by Fan et al. (1998).

The insert operation is extended to increment the value of the buckets and the lookup operation checks that each of the required buckets is non-zero. The delete operation, obviously, then consists of decrementing the value of each of the respective buckets.

Arithmetic overflow of the buckets is a problem and the buckets should be sufficiently large to make this case rare. If it does occur then the increment and decrement operations must leave the bucket set to the maximum possible value in order to retain the properties of a Bloom filter.
The size of counters is usually 3 or 4 bits. Hence counting Bloom filters use 3 to 4 times more space than static Bloom filters. In theory, an optimal data structure equivalent to a counting Bloom filter should not use more space than a static Bloom filter.

Another issue with counting filters is limited scalability. Because the counting Bloom filter table cannot be expanded, the maximal number of keys to be stored simultaneously in the filter must be known in advance. Once the designed capacity of the table is exceeded the false positive rate will grow rapidly as more keys are inserted.

Bonomi et al. (2006) introduced a data structure based on d-left hashing that is functionally equivalent but uses approximately half as much space as counting Bloom filters. The scalability issue does not occur in this data structure. Once the designed capacity is exceeded, the keys could be reinserted in a new hash table of double size.

The space efficient variant by Putze, Sanders & Singler (2007) could also be used to implement counting filters by supporting insertions and deletions.

**Data synchronization**

Bloom filters can be used for approximate data synchronization as in Byers et al. (2004). Counting Bloom filters can be used to approximate the number of differences between two sets and this approach is described in Agarwal & Trachtenberg (2006).

**Bloomier filters**

Chazelle et al. (2004) designed a generalization of Bloom filters that could associate a value with each element that had been inserted, implementing an associative array. Like Bloom filters, these structures achieve a small space overhead by accepting a small probability of false positives. In the case of "Bloomier filters", a *false positive* is defined as returning a result when the key is not in the map. The map will never return the wrong value for a key that is in the map.

The simplest Bloomier filter is near-optimal and fairly simple to describe. Suppose initially that the only possible values are 0 and 1. We create a pair of Bloom filters \( A_0 \) and \( B_0 \) which contain, respectively, all keys mapping to 0 and all keys mapping to 1. Then, to determine which value a given key maps to, we look it up in both filters. If it is in neither, then the key is not in the map. If the key is in \( A_0 \) but not \( B_0 \), then it does not map to 1, and has a high probability of mapping to 0. Conversely, if the key is in \( B_0 \) but not \( A_0 \), then it does not map to 0 and has a high probability of mapping to 1.

A problem arises, however, when both filters claim to contain the key. We never insert a key into both, so one or both of the filters is lying (producing a false positive), but we don't know which. To determine this, we have another, smaller pair of filters \( A_1 \) and \( B_1 \). \( A_1 \) contains keys that map to 0 and which are false positives in \( B_0 \); \( B_1 \) contains keys that map to 1 and which are false positives in \( A_0 \). But whenever \( A_0 \) and \( B_0 \) both produce positives, at most one of these cases must occur, and so we simply have to determine which if any of the two filters \( A_1 \) and \( B_1 \) contains the key, another instance of our original problem.

It may so happen again that both filters produce a positive; we apply the same idea recursively to solve this problem. Because each pair of filters only contains keys that are in the map and produced false positives on all previous filter pairs, the number of keys is extremely likely to quickly drop to a very small quantity that can be easily stored in an ordinary deterministic map, such as a pair of small arrays with linear search. Moreover, the average total search time is a constant, because almost all queries will be resolved by the first pair, almost all remaining queries by the second pair, and so on. The total space required is independent of \( n \), and is almost entirely occupied by the first filter pair.

Now that we have the structure and a search algorithm, we also need to know how to insert new key/value pairs. The program must not attempt to insert the same key with both values. If the value is 0, insert the key into \( A_0 \) and then test if the key is in \( B_0 \). If so, this is a false positive for \( B_0 \) and the key must also be inserted into \( A_1 \) recursively in the same manner. If we reach the last level, we simply insert it. When the value is 1, the operation is similar but with \( A \).
and $B$ reversed. Now that we can map a key to the value 0 or 1, how does this help us map to general values? This is simple. We create a single such Bloomier filter for each bit of the result. If the values are large, we can instead map keys to hash values that can be used to retrieve the actual values. The space required for a Bloomier filter with $n$-bit values is typically slightly more than the space for $2n$ Bloom filters.

A very simple way to implement Bloomier filters is by means of minimal perfect hashing. A minimal perfect hash function $h$ is first generated for the set of $n$ keys. Then an array is filled with $n$ pairs (signature, value) associated with each key at the positions given by function $h$ when applied on each key. The signature of a key is a string of $r$ bits computed by applying a hash function $g$ of range $2^r$ on the key. The value of $r$ is chosen such that $2^r \geq 1/\varepsilon$, where $\varepsilon$ is the requested false positive rate. To query for a given key, hash function $h$ is first applied on the key. This will give a position into the array from which we retrieve a pair (signature, value). Then we compute the signature of the key using function $g$. If the computed signature is the same as retrieved signature we return the retrieved value. The probability of false positive is $1/2^r$.

Another alternative to implement static bloomier and bloom filters based on matrix solving has been simultaneously proposed in Porat (2008), Dietzfelbinger & Pagh (2008) and Charles & Chellapilla (2008). The space usage of this method is optimal as it needs only $\log_2(\varepsilon)$ bits per key for a bloom filter. However, time to generate the bloom or bloomier filter can be very high. The generation time can be reduced to a reasonable value at the price of a small increase in space usage.

Dynamic Bloomier filters have been studied by Mortensen, Pagh & Pătraşcu (2005). They proved that any dynamic Bloomier filter needs at least around $\log(l)$ bits per key where $l$ is the length of the key. A simple dynamic version of Bloomier filters can be implemented using two dynamic data structures. Let the two data structures be noted $S_1$ and $S_2$. $S_1$ will store keys with their associated data while $S_2$ will only store signatures of keys with their associated data. Those signatures are simply hash values of keys in the range $[0, n/\varepsilon]$ where $n$ is the maximal number of keys to be stored in the Bloomier filter and $\varepsilon$ is the requested false positive rate. To insert a key in the Bloomier filter, its hash value is first computed. Then the algorithm checks if a key with the same hash value already exists in $S_2$. If this is not the case, the hash value is inserted in $S_2$ along with data associated with the key. If the same hash value already exists in $S_2$ then the key is inserted into $S_1$ along with its associated data. The deletion is symmetric: if the key already exists in $S_1$ it will be deleted from there, otherwise the hash value associated with the key is deleted from $S_2$. An issue with this algorithm is on how to store efficiently $S_1$ and $S_2$. For $S_1$ any hash algorithm can be used. To store $S_2$ Golomb coding could be applied to compress signatures to use a space close to $\log_2(1/\varepsilon)$ per key.

**Compact approximators**

Boldi & Vigna (2005) proposed a lattice-based generalization of Bloom filters. A compact approximator associates to each key an element of a lattice (the standard Bloom filters being the case of the Boolean two-element lattice). Instead of a bit array, they have an array of lattice elements. When adding a new association between a key and an element of the lattice, they maximize the current content of the $k$ array locations associated to the key with the lattice element. When reading the value associated to a key, they minimize the values found in the $k$ locations associated to the key. The resulting value approximates from above the original value.

**Stable Bloom filters**

Deng & Rafiei (2006) proposed Stable Bloom filters as a variant of Bloom filters for streaming data. The idea is that since there is no way to store the entire history of a stream (which can be infinite), Stable Bloom filters continuously evict stale information to make room for more recent elements. Since stale information is evicted, the Stable Bloom filter introduces false negatives, which do not appear in traditional bloom filters. The authors show that a tight upper bound of false positive rates is guaranteed, and the method is superior to standard bloom filters in terms of false
positive rates and time efficiency when a small space and an acceptable false positive rate are given.

**Scalable Bloom filters**

Almeida et al. (2007) proposed a variant of Bloom filters that can adapt dynamically to the number of elements stored, while assuring a minimum false positive probability. The technique is based on sequences of standard bloom filters with increasing capacity and tighter false positive probabilities, so as to ensure that a maximum false positive probability can be set beforehand, regardless of the number of elements to be inserted.

**Notes**

"Cache Digests are based on a technique first published by Pei Cao, called Summary Cache. The fundamental idea is to use a Bloom filter to represent the cache contents."

**References**

- Byers, John W.; Considine, Jeffrey; Mitzenmacher, Michael; Rost, Stanislav (2004), "Informed content delivery across adaptive overlay networks", *IEEE/ACM Transactions on Networking* 12 (5): 767, doi:10.1109/TNET.2004.836103
- Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson; Wallach, Deborah; Burrows, Mike; Chandra, Tushar; Fikes, Andrew et al. (2006), "Bigtable: A Distributed Storage System for Structured Data" ([http://labs.google.com/papers/bigtable.html](http://labs.google.com/papers/bigtable.html)), *Seventh Symposium on Operating System Design and Implementation*
- Chazelle, Bernard; Kilian, Joe; Rubinfeld, Ronitt; Tal, Ayal (2004), "The Bloomier filter: an efficient data structure for static support lookup tables" (http://www.ee.technion.ac.il/~ayellet/Ps/nelson.pdf), 
- Cohen, Saar; Matias, Yossi (2003), "Spectral Bloom Filters" (http://www.sigmod.org/sigmod03/eproceedings/papers/r09p02.pdf), 
  Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 
- Deng, Fan; Rafiei, Davood (2006), "Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters" (http://webdocs.cs.ualberta.ca/~drafiei/papers/DupDet06Sigmod.pdf), 
  Proceedings of the ACM SIGMOD Conference, pp. 25–36
- Dharmapurikar, Sarang; Song, Haoyu; Turner, Jonathan; Lockwood, John (2006), "Fast packet classification using Bloom filters" (http://www.arl.wustl.edu/~sarang/ancs6819-dharmapurikar.pdf), 
  Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, 
  pp. 61–70, doi:10.1145/1185347.1185356
- Dietzfelbinger, Martin; Pagh, Rasmus (2008), "Succinct Data Structures for Retrieval and Approximate Membership", 
  The Computing Research Repository (CoRR), arXiv:0803.3693
- Dillinger, Peter C.; Manolios, Panagiotis (2004a), "Fast and Accurate Bitstate Verification for SPIN" (http://www.ccs.neu.edu/home/pete/research/spin-3spin.html), 
  Proceedings of the 11th Internation Spin Workshop on Model Checking Software, 
  Springer-Verlag, Lecture Notes in Computer Science 2989
- Dillinger, Peter C.; Manolios, Panagiotis (2004b), "Bloom Filters in Probabilistic Verification" (http://www.ccs.neu.edu/home/pete/research/bloom-filters-verification.html), 
  Proceedings of the 5th Internation Conference on Formal Methods in Computer-Aided Design, 
  Springer-Verlag, Lecture Notes in Computer Science 3312
- Donnet, Benoit; Baynat, Bruno; Friedman, Timur (2006), "Retouched Bloom Filters: Allowing Networked Applications to Flexibly Trade Off False Positives Against False Negatives" (http://adetti.iscte.pt/events/CONEXT06/Conext06_Proceedings/papers/13.html), 
  CoNEXT 06 – 2nd Conference on Future Networking Technologies
- Eppstein, David; Goodrich, Michael T. (2007), "Space-efficient straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters", 
  Algorithms and Data Structures, 10th International Workshop, WADS 2007, 
  Springer-Verlag, Lecture Notes in Computer Science 4619, pp. 637–648, 
  arXiv:0704.3313
- Fan, Li; Cao, Pei; Almeida, Jussara; Broder, Andrei (2000), "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", 
  A preliminary version appeared at SIGCOMM '98.
- Goel, Ashish; Gupta, Pankaj (2010), "Small subset queries and bloom filters using ternary associative memories, with applications", 
- Kirsch, Adam; Mitzenmacher, Michael (2006), "Less Hashing, Same Performance: Building a Better Bloom Filter" (http://www.eecs.harvard.edu/~kirsch/pubs/bbfb/esa06.pdf), in Azar, Yossi; Erlebach, Thomas, 
  Algorithms – ESA 2006, 14th Annual European Symposium, 4168, 
  Springer-Verlag, Lecture Notes in Computer Science 4168, pp. 456–467, doi:10.1007/11841036
- Mortensen, Christian Worm; Pagh, Rasmus; Pătraşcu, Mihai (2005), "On dynamic range reporting in one dimension", 
  Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, 
- Pagh, Anna; Pagh, Rasmus; Rao, S. Srinivasa (2005), "An optimal Bloom filter replacement" (http://www.it-c.dk/people/pagh/papers/bloom.pdf), 
  Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 
  pp. 823–829
- Porat, Ely (2008), "An Optimal Bloom Filter Replacement Based on Matrix Solving", 
  The Computing Research Repository (CoRR), arXiv:0804.1845


External links

• Table of false-positive rates for different configurations (http://www.cs.wisc.edu/~cao/papers/summary-cache/node8.html) from a University of Wisconsin–Madison website

• Bloom Filters and Social Networks with Java applet demo (http://blogs.sun.com/bblfish/entry/my_bloomin_friends) from a Sun Microsystems website

• Interactive Processing demonstration (http://tr.ashcan.org/2008/12/bloomers.html) from ashcan.org


• "Using Bloom Filters" (http://www.perl.com/pub/2004/04/08/bloom_filters.html) Detailed Bloom Filter explanation using Perl

Implementations

• Implementation in C (http://en.literateprograms.org/Bloom_filter_(C)) from literateprograms.org

• Implementation in C++ and Object Pascal (http://www.partow.net/programming/hashfunctions/index.html) from partow.net

• Implementation in C# (http://codeplex.com/bloomfilter) from codeplex.com

• Implementation in Erlang (http://sites.google.com/site/scalablebloomfilters/) from sites.google.com

• Implementation in Haskell (http://hackage.haskell.org/cgi-bin/hackage-scripts/package/bloomfilter) from haskell.org

• Implementation in Java (http://wwwse.inf.tu-dresden.de/xsiena/bloom_filter) from tu-dresden.de

• Implementation in Javascript (http://la.ma.la/misc/js/bloomfilter/) from la.ma.la

• Implementation in Lisp (http://lennonodor.com/archives/000881.html) from lemonodor.com

• Implementation in Perl (http://search.cpan.org/dist/Bloom-Filter/) from cpan.org

• Implementation in PHP (http://code.google.com/p/php-bloom-filter/) from code.google.com

• Implementation in Python, Scalable Bloom Filter (http://pypi.python.org/pypi/pybloom/1.0.2) from pypi.python.org
MinHash

In computer science, MinHash (or the min-wise independent permutations locality sensitive hashing scheme) is a technique for quickly estimating how similar two sets are. The scheme was invented by Andrei Broder (1997), and initially used in the AltaVista search engine to detect duplicate web pages and eliminate them from search results. It has also been applied in large-scale clustering problems, such as clustering documents by the similarity of their sets of words.

Jaccard similarity and minimum hash values

The Jaccard similarity coefficient of two sets $A$ and $B$ is defined to be

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$  

It is a number between 0 and 1; it is 0 when the two sets are disjoint, 1 when they are equal, and strictly between 0 and 1 otherwise. It is a commonly used indicator of the similarity between two sets: two sets are more similar when their Jaccard index is closer to 1, and more dissimilar when their Jaccard index is closer to 0.

Let $h$ be a hash function that maps the members of $A$ and $B$ to distinct integers, and for any set $S$ define $h_{\text{min}}(S)$ to be the member $x$ of $S$ with the minimum value of $h(x)$. Then $h_{\text{min}}(A) = h_{\text{min}}(B)$ exactly when the minimum hash value of the union $A \cup B$ lies in the intersection $A \cap B$. Therefore,

$$\Pr[h_{\text{min}}(A) = h_{\text{min}}(B)] = J(A,B).$$

In other words, if $r$ is a random variable that is one when $h_{\text{min}}(A) = h_{\text{min}}(B)$ and zero otherwise, then $r$ is an unbiased estimator of $J(A,B)$, although it has too high a variance to be useful on its own. The idea of the MinHash scheme is to reduce the variance by averaging together several variables constructed in the same way.

Algorithm

Variant with many hash functions

The simplest version of the minhash scheme uses $k$ different hash functions, where $k$ is a fixed integer parameter, and represents each set $S$ by the $k$ values of $h_{\text{min}}(S)$ for these $k$ functions.

To estimate $J(A,B)$ using this version of the scheme, let $y$ be the number of hash functions for which $h_{\text{min}}(A) = h_{\text{min}}(B)$, and use $y/k$ as the estimate. This estimate is the average of $k$ different 0-1 random variables, each of which is one when $h_{\text{min}}(A) = h_{\text{min}}(B)$ and zero otherwise, and each of which is an unbiased estimator of $J(A,B)$. Therefore, their average is also an unbiased estimator, and by standard Chernoff bounds for sums of 0-1 random variables, its expected error is $O(1/\sqrt{k})$. Therefore, for any constant $\varepsilon > 0$ there is a constant $k = O(1/\varepsilon^2)$ such that the expected error of the estimate is at most $\varepsilon$. For example, 400 hashes would be required to estimate $J(A,B)$ with an expected error less than or equal to .05.
Variant with a single hash function

It may be computationally expensive to compute multiple hash functions, but a related version of MinHash scheme avoids this penalty by using only a single hash function and uses it to select multiple values from each set rather than selecting only a single minimum value per hash function. Let $h$ be a hash function, and let $k$ be a fixed integer. If $S$ is any set of $k$ or more values in the domain of $h$, define $h_{(k)}(S)$ to be the subset of the $k$ members of $S$ that have the smallest values of $h$. This subset $h_{(k)}(S)$ is used as a signature for the set $S$, and the similarity of any two sets is estimated by comparing their signatures.

Specifically, let $A$ and $B$ be any two sets. Then $X = h_{(k)}(h_{(k)}(A) \cup h_{(k)}(B)) = h_{(k)}(A \cup B)$ is a set of $k$ elements of $A \cup B$, and if $h$ is a random function then any subset of $k$ elements is equally likely to be chosen; that is, $X$ is a simple random sample of $A \cup B$. The subset $Y = X \cap h_{(k)}(A) \cap h_{(k)}(B)$ is the set of members of $X$ that belong to the intersection $A \cap B$. Therefore, $|Y|/k$ is an unbiased estimator of $J(A,B)$. The difference between this estimator and the estimator produced by multiple hash functions is that $Y$ always has exactly $k$ members, whereas the multiple hash functions may lead to a smaller number of sampled elements due to the possibility that two different hash functions may have the same minima. However, when $k$ is small relative to the sizes of the sets, this difference is negligible.

By standard Chernoff bounds for sampling without replacement, this estimator has expected error $O(1/\sqrt{k})$, matching the performance of the multiple-hash-function scheme.

Time analysis

The estimator $|Y|/k$ can be computed in time $O(k)$ from the two signatures of the given sets, in either variant of the scheme. Therefore, when $\varepsilon$ and $k$ are constants, the time to compute the estimated similarity from the signatures is also constant. The signature of each set can be computed in linear time, so when many pairwise similarities need to be estimated this method can lead to a substantial savings in running time compared to doing a full comparison of the members of each set.

Min-wise independent permutations

In order to implement the MinHash scheme as described above, one needs the hash function $h$ to define a random permutation on $n$ elements, where $n$ is the total number of distinct elements in the union of all of the sets to be compared. But because there are $n!$ different permutations, it would require $\Omega(n \log n)$ bits just to specify a truly random permutation, an infeasibly large number for even moderate values of $n$. Because of this fact, by analogy to the theory of universal hashing, there has been significant work on finding a family of permutations that is "min-wise independent", meaning that for any subset of the domain, any element is equally likely to be the minimum. It has been established that a min-wise independent family of permutations must include at least

$$\text{lcm}(1, 2, \ldots, n) \geq e^{n-o(n)}$$

different permutations, and therefore that it needs $\Omega(n)$ bits to specify a single permutation, still infeasibly large.$[2]$ Because of this impracticality, two variant notions of min-wise independence have been introduced: restricted min-wise independent permutations families, and approximate min-wise independent families. Restricted min-wise independence is the min-wise independence property restricted to certain sets of cardinality at most $k$.$[4]$ Approximate min-wise independence has at most a fixed probability $\varepsilon$ of varying from full independence.$[5]$
Applications

The original applications for MinHash involved clustering and eliminating near-duplicates among web documents, represented as sets of the words occurring in those documents. Similar techniques have also been used for clustering and near-duplicate elimination for other types of data, such as images: in the case of image data, an image can be represented as a set of smaller subimages cropped from it, or as sets of more complex image feature descriptions.

Schleimer, Wilkerson & Aiken (2003) used MinHash as part of a scheme for the detection of plagiarism in digital documents, by finding pieces of text that were copied from some large database of documents. Their scheme involves representing a document as the set of its substrings of a given length, partitioning the document into larger fixed-length windows, and using the substring with the minimum hash value as a representative value for each window. If a copied portion of text is longer than twice the window length, then its representative value will be sure to match one of the representatives stored in the database, and that window can be examined to determine how much of it was copied.

In data mining, Cohen et al. (2001) use MinHash as a tool for association rule learning. Given a database in which each entry has multiple attributes (viewed as a 0-1 matrix with a row per database entry and a column per attribute) they use MinHash-based approximations to the Jaccard index to identify candidate pairs of attributes that frequently co-occur, and then compute the exact value of the index for only those pairs to determine the ones whose frequencies of co-occurrence are below a given strict threshold.

Related topics

The MinHash scheme may be seen as an instance of locality sensitive hashing, a collection of techniques for using hash functions to map large sets of objects down to smaller hash values in such a way that, when two objects have a small distance to each other, their hash values are likely to be the same. In this instance, the signature of a set may be seen as its hash value. Other locality sensitive hashing techniques exist for Hamming distance between sets and cosine distance between vectors; locality sensitive hashing has important applications in nearest neighbor search algorithms.

References

Disjoint-set data structure

In computing, a **disjoint-set data structure** is a data structure that keeps track of a set of elements partitioned into a number of disjoint (nonoverlapping) subsets. A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure:

- **Find**: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- **Union**: Combine or merge two sets into a single set.

Because it supports these two operations, a disjoint-set data structure is sometimes called a **union-find data structure** or **merge-find set**. The other important operation, **MakeSet**, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved (see the Applications section).

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its **representative**, to represent the set as a whole. Then, **Find** returns the representative of the set that **x** belongs to, and **Union** takes two set representatives as its arguments.

**Disjoint-set linked lists**

A simple approach to creating a disjoint-set data structure is to create a linked list for each set. The element at the head of each list is chosen as its representative.

**MakeSet** creates a list of one element. **Union** appends the two lists, a constant-time operation. The drawback of this implementation is that **Find** requires $\Omega(n)$ or linear time to traverse the list backwards from a given element to the head of the list.

This can be avoided by including in each linked list node a pointer to the head of the list; then **Find** takes constant time, since this pointer refers directly to the set representative. However, **Union** now has to update each element of the list being appended to make it point to the head of the new combined list, requiring $\Omega(n)$ time.

When the length of each list is tracked, the required time can be improved by always appending the smaller list to the longer. Using this **weighted-union heuristic**, a sequence of $m$ **MakeSet**, **Union**, and **Find** operations on $n$ elements requires $O(m + n \log n)$ time.\(^1\) For asymptotically faster operations, a different data structure is needed.

**Analysis of the naïve approach**

We now explain the bound $O(n \log(n))$ above.

Suppose you have a collection of lists, each node of a list contains an object, the name of the list to which it belongs, and the number of elements in that list. Also assume that the sum of the number of elements in all lists is $n$ (i.e. there are $n$ elements overall). We wish to be able to merge any two of these lists, and update all of their nodes so that they still contain the name of the list to which they belong. The rule for merging the lists $A$ and $B$ is that if $A$ is larger than $B$ then merge the elements of $B$ into $A$ and update the elements that used to belong to $B$, and vice versa.

Choose an arbitrary element of list $L$, say $x$. We wish to count how many times in the worst case will $x$ need to have the name of the list to which it belongs updated. The element $x$ will only have its name updated when the list it belongs to is merged with another list of the same size or of greater size. Each time that happens, the size of the list to which $x$ belongs at least doubles. So finally, the question is "how many times can a number double before it is the size of $n$?" (then the list containing $x$ will contain all $n$ elements). The answer is exactly $\log_2(n)$. So for any given element of any given list in the structure described, it will need to be updated $\log_2(n)$ times in the worst case.
Disjoint-set data structure

A find operation can be done in $O(1)$ for this structure because each node contains the name of the list to which it belongs. A similar argument holds for merging the trees in the data structures discussed below, additionally it helps explain the time analysis of some operations in the binomial heap and Fibonacci heap data structures.

Disjoint-set forests

Disjoint-set forests are a data structure where each set is represented by a tree data structure, in which each node holds a reference to its parent node (see spaghetti stack). They were first described by Bernard A. Galler and Michael J. Fischer in 1964,[2] although their precise analysis took years.

In a disjoint-set forest, the representative of each set is the root of that set's tree. Find follows parent nodes until it reaches the root. Union combines two trees into one by attaching the root of one to the root of the other. One way of implementing these might be:

```plaintext
function MakeSet(x)
    x.parent := x
    x.rank   := 0

function Find(x)
    if x.parent == x
        return x
    else
        return Find(x.parent)

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    if xRoot == yRoot
        return
    if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
```

In this naive form, this approach is no better than the linked-list approach, because the tree it creates can be highly unbalanced; however, it can be enhanced in two ways.

The first way, called union by rank, is to always attach the smaller tree to the root of the larger tree, rather than vice versa. Since it is the depth of the tree that affects the running time, the tree with smaller depth gets added under the root of the deeper tree, which only increases the depth if the depths were equal. In the context of this algorithm, the term rank is used instead of depth since it stops being equal to the depth if path compression (described below) is also used. One-element trees are defined to have a rank of zero, and whenever two trees of the same rank $r$ are united, the rank of the result is $r+1$. Just applying this technique alone yields an amortized running-time of $O(\log n)$ per MakeSet, Union, or Find operation. Pseudocode for the improved MakeSet and Union:

```plaintext
function MakeSet(x)
    x.parent := x
    x.rank   := 0

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    if xRoot == yRoot
        return
    if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
        // x and y are not already in same set. Merge them.
        if xRoot.rank < yRoot.rank
```

// xRoot.rank + 1
```
xRoot.parent := yRoot
else if xRoot.rank > yRoot.rank
  yRoot.parent := xRoot
else
  yRoot.parent := xRoot
  xRoot.rank := xRoot.rank + 1

The second improvement, called path compression, is a way of flattening the structure of the tree whenever \textit{Find} is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as \textit{Find} recursively traverses up the tree, it changes each node’s parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly. Here is the improved \textit{Find}:

\begin{verbatim}
function Find(x)
  if x.parent == x
    return x
  else
    x.parent := Find(x.parent)
    return x.parent
\end{verbatim}

These two techniques complement each other; applied together, the amortized time per operation is only $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the function $f(n) = A(n, n)$, and $A$ is the extremely quickly-growing Ackermann function. Since $\alpha(n)$ is the inverse of this function, $\alpha(n)$ is less than 5 for all remotely practical values of $n$. Thus, the amortized running time per operation is effectively a small constant.

In fact, this is asymptotically optimal: Fredman and Saks showed in 1989 that \(\Omega(\alpha(n))\) words must be accessed by any disjoint-set data structure per operation on average.\[3\]

\section*{Applications}

Disjoint-set data structures model the partitioning of a set, for example to keep track of the connected components of an undirected graph. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle. The Union-Find algorithm is used in high-performance implementations of Unification.\[4\]

This data structure is used by the Boost Graph Library to implement its Incremental Connected Components \[5\] functionality. It is also used for implementing Kruskal’s algorithm to find the minimum spanning tree of a graph.

Note that the implementation as disjoint-set forests doesn’t allow deletion of edges—even without path compression or the rank heuristic.
History

While the ideas used in disjoint-set forests have long been familiar, Robert Tarjan was the first to prove the upper bound (and a restricted version of the lower bound) in terms of the inverse Ackermann function, in 1975. Until this time the best bound on the time per operation, proven by Hopcroft and Ullman, was $O(\log^* n)$, the iterated logarithm of $n$, another slowly-growing function (but not quite as slow as the inverse Ackermann function).

Tarjan and Van Leeuwen also developed one-pass $\text{Find}$ algorithms that are more efficient in practice while retaining the same worst-case complexity.

In 2007, Sylvain Conchon and Jean-Christophe Filliâtre developed a persistent version of the disjoint-set forest data structure, allowing previous versions of the structure to be efficiently retained, and formalized its correctness using the proof assistant Coq.

References


External links

- C++ implementation (http://www.boost.org/libs/disjoint_sets/disjoint_sets.html), part of the Boost C++ libraries
- *Wait-free Parallel Algorithms for the Union-Find Problem* (http://citeseer.ist.psu.edu/anderson94waitfree.html), a 1994 paper by Richard J. Anderson and Heather Woll describing a parallelized version of Union-Find that never needs to block
- Python implementation (http://code.activestate.com/recipes/215912-union-find-data-structure/)
Partition refinement

In the design of algorithms, **partition refinement** is a technique for representing a partition of a set as a data structure that allows the partition to be refined by splitting its sets into a larger number of smaller sets. In that sense it is dual to the union-find data structure, which also maintains a partition into disjoint sets but in which the operations merge pairs of sets together. More specifically, a partition refinement algorithm maintains a family of disjoint sets \( S_i \); at the start of the algorithm, this is just a single set containing all the elements in the data structure. At each step of the algorithm, a set \( X \) is presented to the algorithm, and each set \( S_i \) that contains members of \( X \) is replaced by two sets, the intersection \( S_i \cap X \) and the difference \( S_i \setminus X \). Partition refinement forms a key component of several efficient algorithms on graphs and finite automata.\(^1\)\(^2\)\(^3\)

Data structure

A partition refinement algorithm may be implemented by maintaining an object for each set that stores a collection of its elements, in a form such as a doubly linked list that allows for rapid deletion, and an object for each element that points to the set containing it. In addition, each set object should have an instance variable that may point to a second set into which it is being split.

To perform a refinement operation, loop through the elements of \( X \). For each element \( x \), find the set \( S_i \) containing \( x \), and check whether a second set for \( S_i \) has already been formed. If not, create the second set and add \( S_i \) to a list \( L \) of the sets that are split by the operation. Then, regardless of whether a new second set was formed, remove \( x \) from \( S_i \) and add it to the second set.

Finally, after all elements of \( X \) have been processed in this way, loop through \( L \), separating each current set \( S_i \) from the second set that has been split from it, and report both of these sets as newly formed sets from the refinement operation.

The time to perform the refinement operations in this way is \( O(|X|) \), independent of the number of elements or the total number of sets in the data structure.

Applications

Possibly the first application of partition refinement was in an algorithm by Hopcroft (1971) for DFA minimization. In this problem, one is given as input a deterministic finite automaton, and must find an equivalent automaton with as few states as possible. The algorithm maintains a partition of the states of the input automaton into subsets, with the property that any two states in different subsets must be mapped to different states of the output automaton; initially, there are two subsets, one containing all the accepting states and one containing the remaining states. At each step one of the subsets \( S_i \) and one of the input symbols \( x \) of the automaton are chosen, and the subsets of states are refined into states for which a transition labeled \( x \) would lead to \( S_i \), and states for which an \( x \)-transition would lead somewhere else. When a set \( S_i \) that has already been chosen is split by a refinement, only one of the two resulting sets (the smaller of the two) needs to be chosen again; in this way, each state participates in the sets \( X \) for \( O(s \log n) \) refinement steps and the overall algorithm takes time \( O(ns \log n) \), where \( n \) is the number of initial states and \( s \) is the size of the alphabet.\(^4\)

Partition refinement was applied by Sethi (1976) in an efficient implementation of the Coffman–Graham algorithm for parallel scheduling. Sethi showed that it could be used to construct a lexicographically ordered topological sort of a given directed acyclic graph in linear time; this lexicographic topological ordering is one of the key steps of the Coffman–Graham algorithm. In this application, the elements of the disjoint sets are vertices of the input graph and the sets \( X \) used to refine the partition are sets of neighbors of vertices. Since the total number of neighbors of all vertices is just the number of edges in the graph, the algorithm takes time linear in the number of edges, its input size.\(^5\)
Partition refinement also forms a key step in lexicographic breadth-first search, a graph search algorithm with applications in the recognition of chordal graphs and several other important classes of graphs. Again, the disjoint set elements are vertices and the set $X$ represent sets of neighbors, so the algorithm takes linear time.\[6\] \[7\]

**References**


Priority queues

Priority queue

A priority queue is an abstract data type in computer programming. It is exactly like a regular queue or stack data structure, but additionally, each element is associated with a "priority".

- **stack**: elements are pulled in last-in first-out-order (e.g. a stack of papers)
- **queue**: elements are pulled in first-in first-out-order (e.g. a line in a cafeteria)
- **priority queue**: elements are pulled highest-priority-first (e.g. cutting in line, or VIP service).

It is a common misconception that a priority queue is a heap. A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods.

A priority queue must at least support the following operations:

- **insert_with_priority**: add an element to the queue with an associated priority
- **pull_highest_priority_element**: remove the element from the queue that has the highest priority, and return it (also known as "pop_element(Off)", "get_maximum_element", or "get_front(most)_element"); some conventions consider lower priorities to be higher, so this may also be known as "get_minimum_element", and is often referred to as "get-min" in the literature; the literature also sometimes implement separate "peek_at_highest_priority_element" and "delete_element" functions, which can be combined to produce "pull_highest_priority_element")

More advanced implementations may support more complicated operations, such as **pull_lowest_priority_element**, inspecting the first few highest- or lowest-priority elements (peeking at the highest priority element can be made O(1) time in nearly all implementations), clearing the queue, clearing subsets of the queue, performing a batch insert, merging two or more queues into one, incrementing priority of any element, etc.

Similarity to queues

One can imagine a priority queue as a modified queue, but when one would get the next element off the queue, the highest-priority element is retrieved first.

Stacks and queues may be modeled as particular kinds of priority queues. In a stack, the priority of each inserted element is monotonically increasing; thus, the last element inserted is always the first retrieved. In a queue, the priority of each inserted element is monotonically decreasing; thus, the first element inserted is always the first retrieved.
Implementation

Naive implementations

There are a variety of simple, usually inefficient, ways to implement a priority queue. They provide an analogy to help one understand what a priority queue is:

- **Unsorted implementation**: This is perhaps the most naive implementation. Keep all the elements unsorted. Whenever the highest-priority element is requested, search through all elements for the one with the highest priority. (O(1) insertion time, O(n) pull time due to search)

- **Sorted list implementation**: Like a checkout line at the supermarket, but where important people get to "cut" in front of less important people. (if using a basic array, this takes O(n) insertion time, O(1) pullNext time (from the front), and on average O(n\*log(n)) time to initialize (if using quicksort))

These implementations are almost always terribly inefficient, but are meant to illustrate the concept of a priority queue.

Note that from a computational-complexity standpoint, priority queues are equivalent to sorting algorithms. See the next section for how efficient sorting algorithms can create efficient priority queues.

Usual implementation

To get better performance, priority queues typically use a heap as their backbone, giving O(log n) performance for inserts and removals, and O(n) to build initially. Alternatively, if a self-balancing binary search tree is used, insertion and removal also take O(log n) time, although building the tree from an existing sequence of elements takes O(n log n) time; this is a popular solution where one already has access to these data structures, such as through third-party or standard libraries.

Effect of different data structures

The designer of the priority queue should take into account what sort of access pattern the priority queue will be subject to, and what computational resources are most important to the designer. The designer can then use various specialized types of heaps:

There are a number of specialized heap data structures that either supply additional operations or outperform the above approaches. The binary heap uses O(log n) time for both operations, but allows peeking at the element of highest priority without removing it in constant time. Binomial heaps add several more operations, but require O(log n) time for peeking. Fibonacci heaps can insert elements, peek at the highest priority element, and increase an element’s priority in amortized constant time (deletions are still O(log n)).

While relying on a heap is a common way to implement priority queues, for integer data faster implementations exist (this can even apply to datatypes that have finite range, such as floats):

- When the set of keys is {1, 2, ..., C}, a van Emde Boas tree supports the minimum, maximum, insert, delete, search, extract-min, extract-max, predecessor and successor operations in \(O(\log \log C)\) time, but has a space cost for small queues of about \(O(2^n/m^2)\), where \(m\) is the number of bits in the priority value.[1]

- The Fusion tree algorithm by Fredman and Willard implements the minimum operation in O(1) time and insert and extract-min operations in \(O(\sqrt{\log n})\) time.[2]

For applications that do many "peek" operations for every "extract-min" operation, the time complexity for peek can be reduced to O(1) in all tree and heap implementations by caching the highest priority element after every insertion and removal. (For insertion this adds at most constant cost, since the newly inserted element need only be compared to the previously cached minimum element. For deletion, this at most adds an additional "peek" cost, which is nearly always cheaper than the deletion cost, so overall time complexity is not affected by this change).
Equivalence of priority queues and sorting algorithms

Using a priority queue to sort

The semantics of priority queues naturally suggest a sorting method: insert all the elements to be sorted into a priority queue, and sequentially remove them; they will come out in sorted order. This is actually the procedure used by several sorting algorithms, once the layer of abstraction provided by the priority queue is removed. This sorting method is equivalent to the following sorting algorithms:

- Heapsort if the priority queue is implemented with a heap.
- Smoothsort if the priority queue is implemented with a Leonardo heap.
- Selection sort if the priority queue is implemented with an unordered array.
- Insertion sort if the priority queue is implemented with an ordered array.
- Tree sort if the priority queue is implemented with a self-balancing binary search tree.

Using a sorting algorithm to make a priority queue

A sorting algorithm can also be used to implement a priority queue. Specifically, Thorup says:[3]

We present a general deterministic linear space reduction from priority queues to sorting implying that if we can sort up to \( n \) keys in \( S(n) \) time per key, then there is a priority queue supporting delete and insert in \( O(S(n)) \) time and find-min in constant time.

That is, if there is a sorting algorithm which can sort in \( O(S) \) time per key, where \( S \) is some function of \( n \) and word size \([4]\), then one can use the given procedure to create a priority queue where pulling the highest-priority element is \( O(1) \) time, and inserting new elements (and deleting elements) is \( O(S) \) time. For example if one has an \( O(n \ log(lg(n))) \) sort algorithm, one can easily create a priority queue with \( O(1) \) pulling and \( O(lg(lg(n))) \) insertion.

Libraries

A priority queue is often considered to be a "container data structure".

The Standard Template Library (STL), and the C++ 1998 standard, specifies priority_queue as one of the STL container adaptor class templates. It implements a max-priority-queue. Unlike actual STL containers, it does not allow iteration of its elements (it strictly adheres to its abstract data type definition). STL also has utility functions for manipulating another random-access container as a binary max-heap.

Python's heapq module implements a binary min-heap on top of a list.

Java's library contains a PriorityQueue class, which implements a min-priority-queue.

Go's library contains a container/heap module, which implements a min-heap on top of any compatible data structure.

The Standard PHP Library extension contains the class SplPriorityQueue.

Apple's Core Foundation framework contains a CFBinaryHeap structure, which implements a min-heap.

Applications

Bandwidth management

Priority queueing can be used to manage limited resources such as bandwidth on a transmission line from a network router. In the event of outgoing traffic queuing due to insufficient bandwidth, all other queues can be halted to send the traffic from the highest priority queue upon arrival. This ensures that the prioritized traffic (such as real-time traffic, e.g. an RTP stream of a VoIP connection) is forwarded with the least delay and the least likelihood of being rejected due to a queue reaching its maximum capacity. All other traffic can be handled when the highest priority
queue is empty. Another approach used is to send disproportionately more traffic from higher priority queues.

Many modern protocols for Local Area Networks also include the concept of Priority Queues at the Media Access Control (MAC) sub-layer to ensure that high-priority applications (such as VoIP or IPTV) experience lower latency than other applications which can be served with Best effort service. Examples include IEEE 802.11e (an amendment to IEEE 802.11 which provides Quality of Service) and ITU-T G.hn (a standard for high-speed Local area network using existing home wiring (power lines, phone lines and coaxial cables).

Usually a limitation (policer) is set to limit the bandwidth that traffic from the highest priority queue can take, in order to prevent high priority packets from choking off all other traffic. This limit is usually never reached due to high level control instances such as the Cisco Callmanager, which can be programmed to inhibit calls which would exceed the programmed bandwidth limit.

Discrete event simulation

Another use of a priority queue is to manage the events in a discrete event simulation. The events are added to the queue with their simulation time used as the priority. The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.

See also: Scheduling (computing), queueing theory

Dijkstra's algorithm

When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

A* and SMA* search algorithms

The A* search algorithm finds the shortest path between two vertices or nodes of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes; the one for which a lower bound on the total path length is smallest is given highest priority. If memory limitations make A* impractical, the SMA* algorithm can be used instead, with a double-ended priority queue to allow removal of low-priority items.

ROAM triangulation algorithm

The Real-time Optimally Adapting Meshes (ROAM) algorithm computes a dynamically changing triangulation of a terrain. It works by splitting triangles where more detail is needed and merging them where less detail is needed. The algorithm assigns each triangle in the terrain a priority, usually related to the error decrease if that triangle would be split. The algorithm uses two priority queues, one for triangles that can be split and another for triangles that can be merged. In each step the triangle from the split queue with the highest priority is split, or the triangle from the merge queue with the lowest priority is merged with its neighbours.
References


Further reading


External links

• C++ reference for std::priority_queue (http://en.cppreference.com/w/cpp/container/priority_queue)
• Descriptions (http://leekillough.com/heap/) by Lee Killough
• PQlib (http://bitbucket.org/trijezdci/pqlib/src/) - Open source Priority Queue library for C
• libpq/queue (http://github.com/vy/libpqhome) is a generic priority queue (heap) implementation (in C) used by the Apache HTTP Server project.
• Survey of known priority queue structures (http://www.theturingmachine.com/algorithms/heaps.html) by Stefan Xenos
• UC Berkeley - Computer Science 61B - Lecture 24: Priority Queues (http://video.google.com/videoplay?docid=3499489585174920878) (video) - introduction to priority queues using binary heap
• Double-Ended Priority Queues (http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c13/double.htm) by Sartaj Sahni
Heap (data structure)

In computer science, a **heap** is a specialized tree-based data structure that satisfies the *heap property*: if $B$ is a child node of $A$, then $\text{key}(A) \geq \text{key}(B)$. This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a *max-heap*. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a *min-heap*.) There is no restriction as to how many children each node has in a heap, although in practice each node has at most two. The heap is one maximally-efficient implementation of an abstract data type called a priority queue. Heaps are crucial in several efficient graph algorithms such as Dijkstra's algorithm, and in the sorting algorithm heapsort.

A *heap* data structure should not be confused with the *heap* which is a common name for dynamically allocated memory. The term was originally used only for the data structure. Some early popular languages such as LISP provided dynamic memory allocation using heap data structures, which gave the memory area its name\(^1\).

Heaps are usually implemented in an array, and do not require pointers between elements.

The operations commonly performed with a heap are:

- **create-heap**: create an empty heap
- **find-max or find-min**: find the maximum item of a max-heap or a minimum item of a min-heap, respectively
- **delete-max or delete-min**: removing the root node of a max- or min-heap, respectively
- **increase-key or decrease-key**: updating a key within a max- or min-heap, respectively
- **insert**: adding a new key to the heap
- **merge**: joining two heaps to form a valid new heap containing all the elements of both.

**Variants**

- 2-3 heap
- Beap
- Binary heap
- Binomial heap
- Brodal queue
- D-ary heap
- Fibonacci heap
- Leftist heap
- Pairing heap
- Skew heap
- Soft heap
Comparison of theoretic bounds for variants

The following time complexities\(^1\) are amortized (worst-time) time complexity for entries marked by an asterisk, and regular worst case time complexities for all other entries. O(f) gives asymptotic upper bound and Θ(f) is asymptotically tight bound (see Big O notation). Function names assume a min-heap.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary(^1)</th>
<th>Binomial(^1)</th>
<th>Fibonacci(^1)</th>
<th>Pairing(^2)</th>
<th>Brodal(^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>create-heap</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>?</td>
<td>O(1)</td>
</tr>
<tr>
<td>findMin</td>
<td>Θ(1)</td>
<td>O(log n)</td>
<td>Θ(1)</td>
<td>O(1)*</td>
<td>O(1)</td>
</tr>
<tr>
<td>deleteMin</td>
<td>Θ(log n)</td>
<td>Θ(log n)</td>
<td>O(log n)*</td>
<td>O(log n)*</td>
<td>O(log n)</td>
</tr>
<tr>
<td>insert</td>
<td>Θ(n)</td>
<td>O(log n)</td>
<td>Θ(1)</td>
<td>O(1)*</td>
<td>O(1)</td>
</tr>
<tr>
<td>decreaseKey</td>
<td>Θ(log n)</td>
<td>O(log n)</td>
<td>Θ(1)*</td>
<td>O(log n)*</td>
<td>O(1)</td>
</tr>
<tr>
<td>merge</td>
<td>Θ(n)</td>
<td>O(log n)**</td>
<td>Θ(1)</td>
<td>O(1)*</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

(*) Amortized time
(**) Where n is the size of the larger heap

Applications

The heap data structure has many applications.

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Selection algorithms: Finding the min, max, both the min and max, median, or even the k-th largest element can be done in linear time (often constant time) using heaps.\(^4\)
- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

Full and almost full binary heaps may be represented in a very space-efficient way using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc. Thus the children of the node at position \(n\) would be at positions \(2n\) and \(2n+1\) in a one-based array, or \(2n+1\) and \(2n+2\) in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by swapping elements which are out of order. As we can build a heap from an array without requiring extra memory (for the nodes, for example), heapsort can be used to sort an array in-place.

One more advantage of heaps over trees in some applications is that construction of heaps can be done in linear time using Tarjan's algorithm.

Implementations

- The C++ Standard Template Library provides the make_heap, push_heap and pop_heap algorithms for heaps (usually implemented as binary heaps), which operate on arbitrary random access iterators. It treats the iterators as a reference to an array, and uses the array-to-heap conversion. Container adaptor priority_queue also exists.
- The Java 2 platform (since version 1.5) provides the binary heap implementation with class java.util.PriorityQueue<E> in Java Collections Framework.
- Python has a heapq\(^5\) module that implements a priority queue using a binary heap.
- PHP has both maxheap (SplMaxHeap) and minheap (SplMinHeap) as of version 5.3 in the Standard PHP Library.
- Perl has implementations of binary, binomial, and Fibonacci heaps in the Heap\(^5\) distribution available on CPAN.
A **binary heap** is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:

- The **shape property**: the tree is a complete binary tree; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

- The **heap property**: each node is greater than or equal to each of its children according to a comparison predicate defined for the data structure.

Heaps with a mathematical "greater than or equal to" comparison function are called **max-heaps**; those with a mathematical "less than or equal to" comparison function are called **min-heaps**. Min-heaps are often used to implement priority queues.[1][2]

Since the ordering of siblings in a heap is not specified by the heap property, a single node's two children can be freely interchanged unless doing so violates the shape property (compare with treap).

The binary heap is a special case of the d-ary heap in which d = 2.

It is possible to modify the heap structure to allow extraction of both the smallest and largest element in $O(\log n)$ time.[3] To do this, the rows alternate between min heap and max heap. The algorithms are roughly the same, but, in each step, one must consider the alternating rows with alternating comparisons. The performance is roughly the same as a normal single direction heap. This idea can be generalised to a min-max-median heap.
Heap operations

Insert

To add an element to a heap we must perform an up-heap operation (also known as bubble-up, percolate-up, sift-up, trickle up, heapify-up, or cascade-up) in order to restore the heap property. We can do this in \( O(\log n) \) time, using a binary heap, by following this algorithm:

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

We do this at maximum once for each level in the tree—the height of the tree, which is \( O(\log n) \). However, since approximately 50% of the elements are leaves and 75% are in the bottom two levels, it is likely that the new element to be inserted will only move a few levels upwards to maintain the heap. Thus, binary heaps support insertion in average constant time, \( O(1) \).

Say we have a max-heap

![Max-Heap Diagram](image)

and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 is greater than 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:

![Max-Heap Diagram](image)

However the heap property is still violated since 15 is greater than 11, so we need to swap again:

![Max-Heap Diagram](image)

which is a valid max-heap. There is no need to check the children after this. Before we placed 15 on X, the heap was valid, meaning 11 is greater than 5. If 15 is greater than 11, and 11 is greater than 5, then 15 must be greater than 5, because of the transitive relation.

Remove

The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called down-heap (also known as bubble-down, percolate-down, sift-down, trickle down, heapify-down, or cascade-down).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

So, if we have the same max-heap as before, we remove the 11 and replace it with the 4.
Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements 4 and 8, is enough to restore the heap property and we need not swap elements further:

The downward-moving node is swapped with the larger of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position. This functionality is achieved by the Max-Heapify function as defined below in pseudocode for an array-backed heap $A$. Note that "A" is indexed starting at 1, not 0 as is common in many programming languages.

For the following algorithm to correctly re-heapify the array, the node at index $i$ and its two direct children must violate the heap property. If they do not, the algorithm will fall through with no change to the array.

**Max-Heapify** $(A, i)$:

```plaintext
left ← 2i
right ← 2i + 1
largest ← i
if left ≤ heap_length[A] and A[left] > A[i] then:
    largest ← left
if right ≤ heap_length[A] and A[right] > A[largest] then:
    largest ← right
if largest ≠ i then:
Max-Heapify(A, largest)
```

The down-heap operation (without the preceding swap) can also be used to modify the value of the root, even when an element is not being deleted.

**Building a heap**

A heap could be built by successive insertions. This approach requires $O(n \log n)$ time because each insertion takes $O(\log n)$ time and there are $n$ elements. However this is not the optimal method. The optimal method starts by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be represented by an array, see below). Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored. More specifically if all the subtrees starting at some height $h$ (measured from the bottom) have already been "heapified", the trees at height $h + 1$ can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap. This process takes $O(h)$ operations (swaps) per node. In this method most of the heapification takes place in the lower levels. The number of nodes at height $h$ is $\leq \left\lfloor \frac{n}{2^{h+1}} \right\rfloor$. Therefore, the cost of heapifying all subtrees is:
This uses the fact that the given infinite series \( h/2^h \) converges to 2.

The **Build-Max-Heap** function that follows, converts an array \( A \) which stores a complete binary tree with \( n \) nodes to a max-heap by repeatedly using **Max-Heapify** in a bottom up manner. It is based on the observation that the array elements indexed by \( \text{floor}(n/2) + 1, \text{floor}(n/2) + 2, \ldots, n \) are all leaves for the tree, thus each is a one-element heap. **Build-Max-Heap** runs **Max-Heapify** on each of the remaining tree nodes.

**Build-Max-Heap** \(^4\) (\( A \)):

\[
\text{heap_length}[A] \leftarrow \text{length}[A]
\]

\[
\text{for } i \leftarrow \text{floor}(\text{length}[A]/2) \text{ downto } 1 \text{ do}
\]

**Max-Heapify**(\( A, i \))

---

### Heap implementation

Heaps are commonly implemented with an array. Any binary tree can be stored in an array, but because a heap is always an almost complete binary tree, it can be stored compactly. No space is required for pointers; instead, the parent and children of each node can be found by arithmetic on array indices. These properties make this heap implementation a simple example of an implicit data structure or Ahnentafel list. Details depend on the root position, which in turn may depend on constraints of a programming language used for implementation, or programmer preference. Specifically, sometimes the root is placed at index 1, wasting space in order to simplify arithmetic.

Let \( n \) be the number of elements in the heap and \( i \) be an arbitrary valid index of the array storing the heap. If the tree root is at index 0, with valid indices 0 through \( n-1 \), then each element \( a[i] \) has

- children \( a[2i+1] \) and \( a[2i+2] \)
- parent \( a[\text{floor}((i-1)/2)] \)

Alternatively, if the tree root is at index 1, with valid indices 1 through \( n \), then each element \( a[i] \) has

- children \( a[2i] \) and \( a[2i+1] \)
- parent \( a[\text{floor}(i/2)] \).

This implementation is used in the heapsort algorithm, where it allows the space in the input array to be reused to store the heap (i.e. the algorithm is done in-place). The implementation is also useful for use as a Priority queue where use of a dynamic array allows insertion of an unbounded number of items.
The upheap/downheap operations can then be stated in terms of an array as follows: suppose that the heap property holds for the indices $b, b+1, \ldots, e$. The sift-down function extends the heap property to $b-1, b, b+1, \ldots, e$. Only index $i = b-1$ can violate the heap property. Let $j$ be the index of the largest child of $a[i]$ (for a max-heap, or the smallest child for a min-heap) within the range $b, \ldots, e$. (If no such index exists because $2i > e$ then the heap property holds for the newly extended range and nothing needs to be done.) By swapping the values of $a[i]$ and $a[j]$ the heap property for position $i$ is established. At this point, the only problem is that the heap property might not hold for index $j$. The sift-down function is applied tail-recursively to index $j$ until the heap property is established for all elements.

The sift-down function is fast. In each step it only needs two comparisons and one swap. The index value where it is working doubles in each iteration, so that at most $\log_2 e$ steps are required.

For big heaps and using virtual memory, storing elements in an array according to the above scheme is inefficient: (almost) every level is in a different page. B-heaps are binary heaps that keep subtrees in a single page, reducing the number of pages accessed by up to a factor of ten.[5]

The operation of merging two binary heaps takes $\Theta(n)$ for equal-sized heaps. The best you can do is (in case of array implementation) simply concatenating the two heap arrays and build a heap of the result.[6] When merging is a common task, a different heap implementation is recommended, such as binomial heaps, which can be merged in $O(\log n)$.

Additionally, a binary heap can be implemented with a traditional binary tree data structure, but there is an issue with finding the adjacent element on the last level on the binary heap when adding an element. This element can be determined algorithmically or by adding extra data to the nodes, called "threading" the tree—instead of merely storing references to the children, we store the inorder successor of the node as well.

**Derivation of children's index in an array implementation**

This derivation will show how for any given node $i$ (starts from zero), its children would be found at $2i + 1$ and $2i + 2$.

**Mathematical proof**

From the figure in "Heap Implementation" section, it can be seen that any node can store its children only after its right siblings and its left siblings' children have been stored. This fact will be used for derivation.

Total number of elements from root to any given level $l = 2^{l+1} - 1$, where $l$ starts at zero.

Suppose the node $i$ is at level $l$.

So, the total number of nodes from root to previous level would be $2^{(l-1)+1} - 1 = 2^l - 1$

Total number of nodes stored in the array till the index $i = i + 1$(Counting $i$ too)

So, total number of siblings on the left of $i$ is:

- Number of nodes including $i$—Number of nodes through the previous level—One node for $i$ itself
  
  $= (i + 1) - (2^l - 1) - 1$
  
  $= i + 1 - 2^l + 1 - 1$
  
  $= i - 2^l + 1$

Hence, total number of children of these siblings $= 2(i - 2^l + 1)$

Number of elements at any given level $l = 2^l$

So, total siblings to right of $i$ is:

- Total nodes in level $l$ — (Total siblings on left + 1)
  
  $= (2^l) - (i - 2^l + 2)$
  
  $= 2^l + 2^l - i - 2$
So, index of 1st child of node \( i \) would be:

\[
= 2^{i+1} - i - 2
\]

Intuitive proof

Although the mathematical approach proves this without doubt, but the simplicity of the resulting equation suggests that there should be a simpler way to arrive at this conclusion.

For this two facts should be noted.

- Children for node \( i \) will be found at the very first empty slot.
- Second is that, all nodes previous to node \( i \), right from the root, will have exact two children. This is necessary to maintain the shape of the heap.

Now since all nodes have two children (as per the second fact) so all memory slots taken by the children will be \( 2((i + 1) - 1) = 2i \). We add one since \( i \) starts at zero. Then we subtract one since node \( i \) doesn’t yet have any children.

This means all filled memory slots have been accounted for except one – the root node. Root is child to none. So finally, the count of all filled memory slots are \( 2i + 1 \).

So, by fact one and since our indexing starts at zero, \( 2i + 1 \) itself gives the index of the first child of \( i \).

Notes


External links

- Heap from Wolfram MathWorld (http://mathworld.wolfram.com/Heap.html)
- Using Binary Heaps in A* Pathfinding (http://www.policyalmanac.org/games/binaryHeaps.htm)
- Java Implementation of Binary Heap (http://sites.google.com/site/indy256/algo-en/binary_heap)
### d-ary heap

The **d-ary heap** or **d-heap** is a priority queue data structure, a generalization of the binary heap in which the nodes have \( d \) children instead of 2.\(^1\)[2][3] Thus, a binary heap is a 2-heap. According to Tarjan\(^2\) and Jensen et al.,\(^4\) d-ary heaps were invented by Donald B. Johnson in 1975.\(^1\)

This data structure allows decrease priority operations to be performed more quickly than binary heaps, at the expense of slower delete minimum operations. This tradeoff leads to better running times for algorithms such as Dijkstra's algorithm in which decrease priority operations are more common than delete min operations.\(^1\) [5] Additionally, d-ary heaps have better memory cache behavior than a binary heap, allowing them to run more quickly in practice despite having a theoretically larger worst-case running time.\(^6\)[7] Like binary heaps, d-ary heaps are an in-place data structure that uses no additional storage beyond that needed to store the array of items in the heap.\(^2\)[8]

### Data structure

The d-ary heap consists of an array of \( n \) items, each of which has a priority associated with it. These items may be viewed as the nodes in a complete d-ary tree, listed in breadth first traversal order: the item at position 0 of the array forms the root of the tree, the items at positions 1–\( d \) are its children, the next \( d^2 \) items are its grandchildren, etc.

Thus, the parent of the item at position \( i \) (for any \( i > 0 \)) is the item at position \( \lfloor (i - 1)/d \rfloor \) and its children are the items at positions \( di + 1 \) through \( di + d \). According to the heap property, in a min-heap, each item has a priority that is at least as large as its parent; in a max-heap, each item has a priority that is no larger than its parent.\(^2\)[3]

The minimum priority item in a min-heap (or the maximum priority item in a max-heap) may always be found at position 0 of the array. To remove this item from the priority queue, the last item \( x \) in the array is moved into its place, and the length of the array is decreased by one. Then, while item \( x \) and its children do not satisfy the heap property, item \( x \) is swapped with one of its children (the one with the smallest priority in a min-heap, or the one with the largest priority in a max-heap), moving it downward in the tree and later in the array, until eventually the heap property is satisfied. The same downward swapping procedure may be used to increase the priority of an item in a min-heap, or to decrease the priority of an item in a max-heap.\(^2\)[3]

To insert a new item into the heap, the item is appended to the end of the array, and then while the heap property is violated it is swapped with its parent, moving it upward in the tree and earlier in the array, until eventually the heap property is satisfied. The same upward-swapping procedure may be used to decrease the priority of an item in a min-heap, or to increase the priority of an item in a max-heap.\(^2\)[3]

To create a new heap from an array of \( n \) items, one may loop over the items in reverse order, starting from the item at position \( n - 1 \) and ending at the item at position 0, applying the downward-swapping procedure for each item.\(^2\)[3]

### Analysis

In a d-ary heap with \( n \) items in it, both the upward-swapping procedure and the downward-swapping procedure may perform as many as \( \log_d n = \log n / \log d \) swaps. In the upward-swapping procedure, each swap involves a single comparison of an item with its parent, and takes constant time. Therefore, the time to insert a new item into the heap, to decrease the priority of an item in a min-heap, or to increase the priority of an item in a max-heap, is \( O(\log n / \log d) \).

In the downward-swapping procedure, each swap involves \( d \) comparisons and takes \( O(d) \) time: it takes \( d - 1 \) comparisons to determine the minimum or maximum of the children and then one more comparison against the parent to determine whether a swap is needed. Therefore, the time to delete the root item, to increase the priority of an item in a min-heap, or to decrease the priority of an item in a max-heap, is \( O(d \log n / \log d) \).\(^2\)[3]

When creating a d-ary heap from a set of \( n \) items, most of the items are in positions that will eventually hold leaves of the d-ary tree, and no downward swapping is performed for those items. At most \( n/d + 1 \) items are non-leaves, and
may be swapped downwards at least once, at a cost of $O(d)$ time to find the child to swap them with. At most $n/d^2 + 1$ nodes may be swapped downward two times, incurring an additional $O(d)$ cost for the second swap beyond the cost already counted in the first term, etc. Therefore, the total amount of time to create a heap in this way is

$$\sum_{i=1}^{\log_d n} \left( \frac{n}{d^i} + 1 \right) O(d) = O(n).$$

The space usage of the $d$-ary heap, with insert and delete-min operations, is linear, as it uses no extra storage other than an array containing a list of the items in the heap. If changes to the priorities of existing items need to be supported, then one must also maintain pointers from the items to their positions in the heap, which again uses only linear storage.

**Applications**

Dijkstra’s algorithm for shortest paths in graphs and Prim’s algorithm for minimum spanning trees both use a min-heap in which there are $n$ delete-min operations and as many as $m$ decrease-priority operations, where $n$ is the number of vertices in the graph and $m$ is the number of edges. By using a $d$-ary heap with $d = m/n$, the total times for these two types of operations may be balanced against each other, leading to a total time of $O(m \log_{m/n} n)$ for the algorithm, an improvement over the $O(m \log n)$ running time of binary heap versions of these algorithms whenever the number of edges is significantly larger than the number of vertices. An alternative priority queue data structure, the Fibonacci heap, gives an even better theoretical running time of $O(m + n \log n)$, but in practice $d$-ary heaps are generally at least as fast, and often faster, than Fibonacci heaps for this application.

4-heaps may perform better than binary heaps in practice, even for delete-min operations. Additionally, a $d$-ary heap typically runs much faster than a binary heap for heap sizes that exceed the size of the computer’s cache memory: A binary heap typically requires more cache misses and virtual memory page faults than a $d$-ary heap, each one taking far more time than the extra work incurred by the additional comparisons a $d$-ary heap makes compared to a binary heap.

**References**


Binomial heap

In computer science, a **binomial heap** is a heap similar to a binary heap but also supports quickly merging two heaps. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation.

**Binomial tree**

A binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order \( k \) has a root node whose children are roots of binomial trees of orders \( k-1, k-2, ..., 2, 1, 0 \) (in this order).

A binomial tree of order \( k \) has \( 2^k \) nodes, height \( k \).

Because of its unique structure, a binomial tree of order \( k \) can be constructed from two trees of order \( k-1 \) trivially by attaching one of them as the leftmost child of root of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.

The name comes from the shape: a binomial tree of order \( n \) has \( \binom{n}{d} \) nodes at depth \( d \). (See Binomial coefficient.)
Binomial heap

Structure of a binomial heap

A binomial heap is implemented as a set of binomial trees that satisfy the binomial heap properties:

• Each binomial tree in a heap obeys the minimum-heap property: the key of a node is greater than or equal to the key of its parent.

• There can only be either one or zero binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

The second property implies that a binomial heap with \( n \) nodes consists of at most \( \log n + 1 \) binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes \( n \): each binomial tree corresponds to one digit in the binary representation of number \( n \). For example number 13 is 1101 in binary, \( 2^3 + 2^2 + 2^0 \), and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).

Example of a binomial heap containing 13 nodes with distinct keys.
The heap consists of three binomial trees with orders 0, 2, and 3.

Implementation

Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.

Merge

As mentioned above, the simplest and most important operation is the merging of two binomial trees of the same order within two binomial heaps. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree become a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

```plaintext
function mergeTree(p, q)
    if p.root.key <= q.root.key
        return p.addSubTree(q)
    else
        return q.addSubTree(p)
```
To merge two binomial trees of the same order, first compare the root key. Since 7>3, the black tree on the left (with root node 7) is attached to the grey tree on the right (with root node 3) as a subtree. The result is a tree of order 3.

The operation of **merging** two heaps is perhaps the most interesting and can be used as a subroutine in most other operations. The lists of roots of both heaps are traversed simultaneously, similarly as in the merge algorithm.

If only one of the heaps contains a tree of order \( j \), this tree is moved to the merged heap. If both heaps contain a tree of order \( j \), the two trees are merged to one tree of order \( j+1 \) so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order \( j+1 \) present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Each tree has order at most \( \log n \) and therefore the running time is \( O(\log n) \).

**function** merge(\( p \), \( q \))

```python
while not (p.end() and q.end())
    tree = mergeTree(p.currentTree(), q.currentTree())
    if not heap.currentTree().empty()
        tree = mergeTree(tree, heap.currentTree())
        heap.addTree(tree)
    else
        heap.addTree(tree)
    heap.next() p.next() q.next()
```
Insert

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes $O(\log n)$ time, however it has an amortized time of $O(1)$ (i.e. constant).

Find minimum

To find the minimum element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in $O(\log n)$ time, as there are just $O(\log n)$ trees and hence roots to examine.

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to $O(1)$. The pointer must be updated when performing any operation other than Find minimum. This can be done in $O(\log n)$ without raising the running time of any operation.

Delete minimum

To delete the minimum element from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees. Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each tree has at most $\log n$ children, creating this new heap is $O(\log n)$. Merging heaps is $O(\log n)$, so the entire delete minimum operation is $O(\log n)$.

```
function deleteMin(heap)
    min = heap.trees().first()
    for each current in heap.trees()
        if current.root < min then min = current
    for each tree in min.subTrees()
        tmp.addTree(tree)
    heap.removeTree(min)
    merge(heap, tmp)
```
**Decrease key**

After decreasing the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most \( \log n \), so this takes \( O(\log n) \) time.

**Delete**

To delete an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

**Performance**

All of the following operations work in \( O(\log n) \) time on a binomial heap with \( n \) elements:

- Insert a new element to the heap
- Find the element with minimum key
- Delete the element with minimum key from the heap
- Decrease key of a given element
- Delete given element from the heap
- Merge two given heaps to one heap

Finding the element with minimum key can also be done in \( O(1) \) by using an additional pointer to the minimum.

**Applications**

- Discrete event simulation
- Priority queues

**References**


**External links**

- Java applet simulation of binomial heap \(^2\)
- Python implementation of binomial heap \(^3\)
- Two C implementations of binomial heap \(^4\) (a generic one and one optimized for integer keys)
- Haskell implementation of binomial heap \(^5\)
- Common Lisp implementation of binomial heap \(^6\)
Fibonacci heap

In computer science, a Fibonacci heap is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap. Fibonacci heaps were developed by Michael L. Fredman and Robert E. Tarjan in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from Fibonacci numbers which are used in the running time analysis.

Find-minimum is $O(1)$ amortized time. Operations insert, decrease key, and merge (union) work in constant amortized time. Operations delete and delete minimum work in $O(\log n)$ amortized time. This means that starting from an empty data structure, any sequence of $a$ operations from the first group and $b$ operations from the second group would take $O(a + b \log n)$ time. In a binomial heap such a sequence of operations would take $O((a + b)\log (n))$ time. A Fibonacci heap is thus better than a binomial heap when $b$ is asymptotically smaller than $a$.

Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing Shortest paths.

Structure

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and operation decrease key sometimes cuts a node from its parent and forms a new tree.

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree $k$ is at least $F_{k+2}$, where $F_k$ is the $k$th Fibonacci number. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased in the operation delete minimum, where trees are linked together.
As a result of a relaxed structure, some operations can take a long time while others are done very quickly. In the amortized running time analysis we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

\[
\text{Potential} = t + 2m
\]

where \( t \) is the number of trees in the Fibonacci heap, and \( m \) is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked).

Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

**Implementation of operations**

To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost is constant. As mentioned above, **merge** is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time.

Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

Operation **extract minimum** (same as **delete minimum**) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was \( d \), it takes time \( O(d) \) to process all new roots and the potential increases by \( d-1 \). Therefore the amortized running time of this phase is \( O(d) = O(\log n) \).
However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to \( n \) roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots \( u \) and \( v \) have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length \( O(\log n) \) in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and the array is updated. The actual running time is \( O(\log n + m) \) where \( m \) is the number of roots at the beginning of the second phase. At the end we will have at most \( O(\log n) \) roots (because each has a different degree). Therefore the difference in the potential function from before this phase to after it is:

\[
O(\log n) - m,
\]

and the amortized running time is then at most \( O(\log n + m) + O(\log n) - m = O(\log n) \). Since we can scale up the units of potential stored at insertion in each node by the constant factor in the \( O(m) \) part of the actual cost for this phase.

In the third phase we check each of the remaining roots and find the minimum. This takes \( O(\log n) \) time and the potential does not change. The overall amortized running time of extract minimum is therefore \( O(\log n) \).

Operation **decrease key** will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. In the process we create some number, say \( k \), of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore the potential decreases by at least \( k - 2 \). The actual time to perform the cutting was \( O(k) \), therefore the amortized running time is constant.

Finally, operation **delete** can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap. Then we call extract minimum to remove it. The amortized running time of this operation is \( O(\log n) \).

**Proof of degree bounds**

The amortized performance of a Fibonacci heap depends on the degree (number of children) of any tree root being \( O(\log n) \), where \( n \) is the size of the heap. Here we show that the size of the (sub)tree rooted at any node \( x \) of degree \( d \) in the heap must have size at least \( F_{d+2} \), where \( F_k \) is the \( k \)th Fibonacci number. The degree bound follows from this and the fact (easily proved by induction) that \( F_{d+2} \geq \varphi^d \) for all integers \( d \geq 0 \), where \( \varphi = (1 + \sqrt{5})/2 \approx 1.62 \). (We then have \( n \geq F_{d+2} \geq \varphi^d \), and taking the log to base \( \varphi \) of both sides gives \( d \leq \log_{\varphi} n \) as required.)

Consider any node \( x \) somewhere in the heap (\( x \) need not be the root of one of the main trees). Define **size**(\( x \)) to be the size of the tree rooted at \( x \) (the number of descendants of \( x \), including \( x \) itself). We prove by induction on the height of \( x \) (the length of a longest simple path from \( x \) to a descendant leaf), that **size**(\( x \)) \( \geq F_{d+2} \) where \( d \) is the degree of \( x \).

**Base case:** If \( x \) has height 0, then \( d = 0 \), and **size**(\( x \)) = 1 = \( F_2 \).
Inductive case: Suppose $x$ has positive height and degree $d > 0$. Let $y_1, y_2, \ldots, y_d$ be the children of $x$, indexed in order of the times they were most recently made children of $x$ ($y_1$ being the earliest and $y_d$ the latest), and let $c_1, c_2, \ldots, c_d$ be their respective degrees. We claim that $c_i \geq i - 2$ for each $i$ with $2 \leq i \leq d$: Just before $y_i$ was made a child of $x$, $y_1, \ldots, y_{i-1}$ were already children of $x$, and so $x$ had degree at least $i - 1$ at that time. Since trees are combined only when the degrees of their roots are equal, it must have been that $y_i$ also had degree at least $i - 1$ at the time it became a child of $x$. From that time to the present, $y_i$ can only have lost at most one child (as guaranteed by the marking process), and so its current degree $c_i$ is at least $i - 2$. This proves the claim.

Since the heights of all the $y_i$ are strictly less than that of $x$, we can apply the inductive hypothesis to them to get $\text{size}(y_i) \geq F_{c_i+2} \geq F_{(i-2)+2} = F_i$. The nodes $x$ and $y_1$ each contribute at least 1 to $\text{size}(x)$, and so we have

$$\text{size}(x) \geq 2 + \sum_{i=2}^{d} \text{size}(y_i) \geq 2 + \sum_{i=2}^{d} F_i = 1 + \sum_{i=0}^{d} F_i.$$

A routine induction proves that $1 + \sum_{i=0}^{d} F_i = F_{d+2}$ for any $d \geq 0$, which gives the desired lower bound on $\text{size}(x)$.

Worst case

Although the total running time of a sequence of operations starting with an empty structure is bounded by the bounds given above, some (very few) operations in the sequence can take very long to complete (in particular delete and delete minimum have linear running time in the worst case). For this reason Fibonacci heaps and other amortized data structures may not be appropriate for real-time systems.

Summary of running times

<table>
<thead>
<tr>
<th>Common Operations</th>
<th>Effect</th>
<th>Unsorted Linked List</th>
<th>Self-balancing binary search tree</th>
<th>Binary heap</th>
<th>Binomial heap</th>
<th>Fibonacci heap</th>
<th>Brodal queue [2]</th>
<th>Pairing heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(data,key)</td>
<td>Adds data to the queue, tagged with key</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>findMin() -&gt; key,data</td>
<td>Returns key,data corresponding to min-value key</td>
<td>$O(n)$</td>
<td>$O(\log n)$ or $O(1)$ (***)</td>
<td>$O(1)$</td>
<td>$O(\log n)[3]$</td>
<td>$O(1)[1]$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>deleteMin()</td>
<td>Deletes data corresponding to min-value key</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)^*$</td>
<td></td>
</tr>
<tr>
<td>delete(node)</td>
<td>Deletes data corresponding to given key, given a pointer to the node being deleted</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)^*$</td>
<td></td>
</tr>
<tr>
<td>decreaseKey(node)</td>
<td>Decreases the key of a node, given a pointer to the node being modified</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)^*$</td>
<td>$O(1)^*$</td>
<td>Unknown but bounded: $\Omega(\log \log n), 2^{O(\sqrt{\log \log n})}$</td>
<td></td>
</tr>
</tbody>
</table>
Fibonacci heap

merge(heap1, heap2) -> heap3
Merges two heaps into a third

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
</table>
| O(1)          | Artifactual time
| O(m log(n+m)) | With trivial modification to store an additional pointer to the minimum element
| O(m+n)**      | Where m is the size of the larger heap
| O(log n)***    | Where n is the size of the larger heap

References


External links

- C implementation of Fibonacci heap (http://resnet.uoregon.edu/~gurney_j/jmpc/fib.html)
- MATLAB implementation of Fibonacci heap (http://www.mathworks.com/matlabcentral/fileexchange/30072-fibonacci-heap)
- De-recursived and memory efficient C implementation of Fibonacci heap (http://www.labri.fr/perso/pelegrin/code/#fibonacci) (free/libre software, CeCILL-B license (http://www.cecill.info/licences/Licence_CeCILL-B_V1-en.html))
- C++ template Fibonacci heap, with demonstration (http://ideone.com/9jYnv)
- Ruby implementation of the Fibonacci heap (with tests) (http://github.com/evansenter/f_heap)
**Pairing heap**

A pairing heap is a type of heap data structure with relatively simple implementation and excellent practical amortized performance. However, it has proven very difficult to determine the precise asymptotic running time of pairing heaps.

Pairing heaps are heap ordered multiway trees. Describing the various heap operations is relatively simple (in the following we assume a min-heap):

- **find-min**: simply return the top element of the heap.
- **merge**: compare the two root elements, the smaller remains the root of the result, the larger element and its subtree is appended as a child of this root.
- **insert**: create a new heap for the inserted element and merge it into the original heap.
- **decrease-key** (optional): remove the subtree rooted at the key to be decreased then merge it with the heap.
- **delete-min**: remove the root and merge its subtrees. Various strategies are employed.

The amortized time per delete-min is $O(\log n)$. The operations find-min, merge, and insert are $O(1)$ and decrease-key takes $O(\sqrt{\log \log n})$ amortized time. Fredman proved that the amortized time per decrease-key is at least $\Omega(\log \log n)$. Although this is worse than other priority queue algorithms such as Fibonacci heaps, which perform decrease-key in $O(1)$ amortized time, the performance in practice is excellent. Stasko and Vitter and Moret and Shapiro conducted experiments on pairing heaps and other heap data structures. They concluded that the pairing heap is as fast as, and often faster than, other efficient data structures like the binary heaps.

**Implementation**

A pairing heap is either an empty heap, or a pair consisting of a root element and a possibly empty list of pairing heaps. The heap ordering property requires that all the root elements of the subheaps in the list are not smaller than the root element of the heap. The following description assumes a purely functional heap that does not support the decrease-key operation.

```
type PairingHeap[Elem] = Empty Heap(elem: Elem, subheaps: List[PairingHeap[Elem])
```

**Operations**

**find-min**

The function find-min simply returns the root element of the heap:

```
function find-min(heap)
    if heap == Empty
        error
    else
        return heap.elem
```
**merge**

Merging with an empty heap returns the other heap, otherwise a new heap is returned that has the minimum of the two root elements as its root element and just adds the heap with the larger root to the list of subheaps:

```python
function merge(heap1, heap2)
    if heap1 == Empty
        return heap2
    elsif heap2 == Empty
        return heap1
    elseif heap1.elem < heap2.elem
        return Heap(heap1.elem, heap2 :: heap1.subheaps)
    else
        return Heap(heap2.elem, heap1 :: heap2.subheaps)
```

**insert**

The easiest way to insert an element into a heap is to merge the heap with a new heap containing just this element and an empty list of subheaps:

```python
function insert(elem, heap)
    return merge(Heap(elem, []), heap)
```

**delete-min**

The only non-trivial fundamental operation is the deletion of the minimum element from the heap. The standard strategy first merges the subheaps in pairs (this is the step that gave this datastructure its name) from left to right and then merges the resulting list of heaps from right to left:

```python
function delete-min(heap)
    if heap == Empty
        error
    elsif length(heap.subheaps) == 0
        return Empty
    elsif length(heap.subheaps) == 1
        return heap.subheaps[0]
    else
        return merge-pairs(heap.subheaps)
```

This uses the auxiliary function `merge-pairs`:

```python
function merge-pairs(l)
    if length(l) == 0
        return Empty
    elsif length(l) == 1
        return l[0]
    else
        return merge(merge(l[0], l[1]), merge-pairs(l[2.. ]))
```

That this does indeed implement the described two-pass left-to-right then right-to-left merging strategy can be seen from this reduction:
merge-pairs([H1, H2, H3, H4, H5, H6, H7])

=> merge(merge(H1, H2), merge-pairs([H3, H4, H5, H6, H7]))
    # merge H1 and H2 to H12, then the rest of the list
=> merge(H12, merge(merge(H3, H4), merge-pairs([H5, H6, H7])))
    # merge H3 and H4 to H34, then the rest of the list
=> merge(H12, merge(H34, merge(merge(H5, H6), merge-pairs([H7]))))
    # merge H5 and H6 to H56, then the rest of the list
=> merge(H12, merge(H34, merge(H56, H7)))
    # switch direction, merge the last two resulting heaps, giving H567
=> merge(H12, merge(H34, H567))
    # merge the last two resulting heaps, giving H34567
=> merge(H12, H34567)
    # finally, merge the first merged pair with the result of merging the rest
=> H1234567

References


External links

- pairing heaps (http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c13/pairing.htm), Sartaj Sahni
Double-ended priority queue

A double-ended priority queue (DEPQ)\(^1\) is an abstract data type similar to a priority queue except that it allows for efficient removal of both the maximum and minimum element. It is a data structure in which one can insert elements and then remove the elements with minimum or maximum priority. Every element in a DEPQ has a priority or value. In a DEPQ, it is possible to remove the elements in both ascending as well as descending order.\(^2\)

Operations

A double-ended priority queue features the follow operations:

**isEmpty()**
Checks if DEPQ is empty and returns true if empty.

**size()**
Returns the total number of elements preset in the DEPQ.

**getMin()**
Returns the element having least priority.

**getMax()**
Returns the element having highest priority.

**put(x)**
Inserts the element \(x\) in the DEPQ.

**removeMin()**
Removes an element with minimum priority and returns this element.

**removeMax()**
Removes an element with maximum priority and returns this element.

If an operation is to be performed on two elements having the same priority, then the element inserted first is chosen. Also, the priority of any element can be changed once it has been inserted in the DEPQ.\(^3\)
Double-ended priority queue

Implementation

Double-ended priority queues can be built from balanced binary search trees (where the minimum and maximum elements are the leftmost and rightmost leaves, respectively), or using specialized data structures like min-max heap and pairing heap.

Generic methods of arriving at double-ended priority queues from normal priority queues are:

Dual structure method

In this method two different priority queues for min and max are maintained. The same elements in both the PQs are shown with the help of correspondence pointers.

Total correspondence

Half the elements are in the min PQ and the other half in the max PQ. Each element in the min PQ has a one to one correspondence with an element in max PQ. If the number of elements in the DEPQ is odd, one of the elements is retained in a buffer. Priority of every element in the min PQ will be less than or equal to the corresponding element in the max PQ.
Leaf correspondence

In this method only the leaf elements of the min and max PQ form corresponding one to one pairs. It is not necessary for non-leaf elements to be in a one to one correspondence pair.\[^1\]

Interval heaps

Apart from the above mentioned correspondence methods, DEPQ’s can be obtained efficiently using interval heaps.\[^5\] An interval heap is like an embedded min-max heap in which each node contains two elements. It is a complete binary tree in which:\[^5\]

- The left element is less than or equal to the right element.
- Both the elements define a closed interval.
- Interval represented by any node except the root is a sub-interval of the parent node.
- Elements on the left hand side define a min heap.
- Elements on the right hand side define a max heap.

Depending on the number of elements, two cases are possible:\[^5\] -

1. **Even number of elements:** In this case, each node contains two elements say \(p\) and \(q\), with \(p \leq q\). Every node is then represented by the interval \([p, q]\).
2. **Odd number of elements:** In this case, each node except the last contains two elements represented by the interval \([p, q]\) whereas the last node will contain a single element and is represented by the interval \([p, p]\).

Inserting an element

Depending on the number of elements already present in the interval heap, following cases are possible:

- **Odd number of elements:** If the number of elements in the interval heap is odd, the new element is firstly inserted in the last node. Then, it is successively compared with the previous node elements and tested to satisfy the criteria essential for an interval heap as stated above. In case if the element does not satisfy any of the criteria, it is moved from the last node to the root until all the conditions are satisfied.\[^5\]
- **Even number of elements:** If the number of elements is even, then for the insertion of a new element an additional node is created. If the element falls to the left of the parent interval, it is considered to be in the min
heap and if the element falls to the right of the parent interval, it is considered in the max heap. Further, it is compared successively and moved from the last node to the root until all the conditions for interval heap are satisfied. If the element lies within the interval of the parent node itself, the process is stopped then and there itself and moving of elements does not take place.\footnote{5}

The time required for inserting an element depends on the number of movements required to meet all the conditions and is $O(\log n)$.

**Deleting an element**

- **Min element**: In an interval heap, the minimum element is the element on the left hand side of the root node. This element is removed and returned. To fill in the vacancy created on the left hand side of the root node, an element from the last node is removed and reinserted into the root node. This element is then compared successively with all the left hand elements of the descending nodes and the process stops when all the conditions for an interval heap are satisfied. In case if the left hand side element in the node becomes greater than the right side element at any stage, the two elements are swapped\footnote{5} and then further comparisons are done. Finally, the root node will again contain the minimum element on the left hand side.

- **Max element**: In an interval heap, the maximum element is the element on the right hand side of the root node. This element is removed and returned. To fill in the vacancy created on the right hand side of the root node, an element from the last node is removed and reinserted into the root node. Further comparisons are carried out on a similar basis as discussed above. Finally, the root node will again contain the max element on the right hand side.

Thus, with interval heaps, both the minimum and maximum elements can be removed efficiently traversing from root to leaf. Thus, a DEPQ can be obtained\footnote{5} from an interval heap where the elements of the interval heap are the priorities of elements in the DEPQ.

**Time Complexity**

**Interval Heaps**

When DEPQ's are implemented using Interval heaps consisting of $n$ elements, the time complexities for the various functions are formulated in the table below\footnote{1}

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>init()</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getmin()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getmax()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert(x)</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>removeMin()</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>removeMax()</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Double-ended priority queue

Pairing heaps

When DEPQ's are implemented using heaps or paring heaps consisting of \( n \) elements, the time complexities for the various functions are formulated in the table below\(^1\). For pairing heaps, it is an amortized complexity.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty( )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>getmin( )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>getmax( )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>insert(x)</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>removeMax( )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>removeMin( )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>

Applications

External sorting

One example application of the double-ended priority queue is external sorting. In an external sort, there are more elements than can be held in the computer's memory. The elements to be sorted are initially on a disk and the sorted sequence is to be left on the disk. The external quick sort is implemented using the DEPQ as follows:

1. Read in as many elements as will fit into an internal DEPQ. The elements in the DEPQ will eventually be the middle group (pivot) of elements.
2. Read in the remaining elements. If the next element is \( \leq \) the smallest element in the DEPQ, output this next element as part of the left group. If the next element is \( \geq \) the largest element in the DEPQ, output this next element as part of the right group. Otherwise, remove either the max or min element from the DEPQ (the choice may be made randomly or alternately); if the max element is removed, output it as part of the right group; otherwise, output the removed element as part of the left group; insert the newly input element into the DEPQ.
3. Output the elements in the DEPQ, in sorted order, as the middle group.
4. Sort the left and right groups recursively.

Other applications

An airport takeoff and landing system is a practical example of a Double-ended priority queue.

- An airplane which wants to take off or land is added to the queue
- Priority may be assigned to the airplane on the basis of parameters like fuel level (while landing) or long distance flights (during takeoff).
- The airplane with maximum priority is given the permission to land or take-off first.\(^6\)
See Also

- Queue (ADT)
- Priority queue
- Double-ended queue

References


Soft heap

In computer science, a soft heap is a variant on the simple heap data structure that has constant amortized time for 5 types of operations. This is achieved by carefully "corrupting" (increasing) the keys of at most a certain fixed percentage of values in the heap. The constant time operations are:

- **create**(S): Create a new soft heap
- **insert**(S, x): Insert an element into a soft heap
- **meld**(S, S'): Combine the contents of two soft heaps into one, destroying both
- **delete**(S, x): Delete an element from a soft heap
- **findmin**(S): Get the element with minimum key in the soft heap

It was designed by Bernard Chazelle in 2000. The term "corruption" in the structure is the result of what Chazelle called "carpooling" in a soft heap. Each node in the soft heap contains a linked-list of keys and one common key. The common key is an upper bound on the values of the keys in the linked-list. Once a key is added to the linked-list, it is considered corrupted because its value is never again relevant in any of the soft heap operations: only the common keys are compared. It is unpredictable which keys will be corrupted in this manner; it is only known that at most a fixed percentage will be corrupted. This is what makes soft heaps "soft"; you can't be sure whether or not any particular value you put into it will be corrupted. The purpose of these corruptions is effectively to lower the information entropy of the data, enabling the data structure to break through information-theoretic barriers regarding heaps.

Other heaps such as Fibonacci heaps achieve most of these bounds without any corruption, but cannot provide a constant-time bound on the critical delete operation. The percentage of values which are corrupted can be chosen freely, but the lower this is set, the more time insertions require (O(log 1/ε) for an error rate of ε).
Applications

Surprisingly, soft heaps are useful in the design of deterministic algorithms, despite their unpredictable nature. They were used to achieve the best complexity to date for finding a minimum spanning tree. They can also be used to easily build an optimal selection algorithm, as well as near-sorting algorithms, which are algorithms that place every element near its final position, a situation in which insertion sort is fast.

One of the simplest examples is the selection algorithm. Say we want to find the \( k \)th largest of a group of \( n \) numbers. First, we choose an error rate of \( 1/3 \); that is, at most 33% of the keys we insert will be corrupted. Now, we insert all \( n \) elements into the heap — at this point, at most \( n/3 \) keys are corrupted. Next, we delete the minimum element from the heap about \( n/3 \) times. Because this is decreasing the size of the heap, it cannot increase the number of corrupted elements. Thus there are still at most \( n/3 \) keys that are corrupted.

Now at least \( 2n/3 - n/3 = n/3 \) of the remaining keys are not corrupted, so each must be larger than every element we removed. Let \( L \) be the element that we have removed with the largest (actual) value, which is not necessarily the last element that we removed (because the last element we removed could have had its key corrupted, or increased, to a value larger than another element that we have already removed). \( L \) is larger than all the other \( n/3 \) elements that we removed and smaller than the remaining \( n/3 \) uncorrupted elements in the soft heap. Therefore, \( L \) divides the elements somewhere between \( 33%/66\% \) and \( 66%/33\% \). We then partition the set about \( L \) using the partition algorithm from quicksort and apply the same algorithm again to either the set of numbers less than \( L \) or the set of numbers greater than \( L \), neither of which can exceed \( 2n/3 \) elements. Since each insertion and deletion requires \( O(1) \) amortized time, the total deterministic time is \( T(n) = T(2n/3) + O(n) \). Using case 3 of the master theorem (with \( \epsilon = 1 \) and \( c = 2/3 \)), we know that \( T(n) = \Theta(n) \).

The final algorithm looks like this:

```plaintext
function softHeapSelect(a[1..n], k)
    if k = 1 then return minimum(a[1..n])
    create(S)
    for i from 1 to n
        insert(S, a[i])
    for i from 1 to n/3
        x := findmin(S)
        delete(S, x)
        xIndex := partition(a, x)  // Returns new index of pivot x
        if k < xIndex
            softHeapSelect(a[1..xIndex-1], k)
        else
            softHeapSelect(a[xIndex..n], k-xIndex+1)
```

References

References

Successors and neighbors

Binary search algorithm

<table>
<thead>
<tr>
<th>Class</th>
<th>Search algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>Array</td>
</tr>
<tr>
<td>Worst case performance</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Best case performance</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Average case performance</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Worst case space complexity</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

In computer science, a binary search or half-interval search algorithm finds the position of a specified value (the input "key") within a sorted array. At each stage, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index, or position, is returned. Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic divide and conquer search algorithm.

Overview

Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.

If the list to be searched contains more than a few items (a dozen, say) a binary search will require far fewer comparisons than a linear search, but it imposes the requirement that the list be sorted. Similarly, a hash search can be faster than a binary search but imposes still greater requirements. If the contents of the array are modified between searches, maintaining these requirements may take more time than the searches! And if it is known that some items will be searched for much more often than others, and it can be arranged that these items are at the start of the list, then a linear search may be the best.

Examples

Number guessing game

This rather simple game begins something like "I'm thinking of an integer between forty and sixty inclusive, and to your guesses I'll respond 'High', 'Low', or 'Yes!' as might be the case." Supposing that $N$ is the number of possible values (here, twenty-one as "inclusive" was stated), then at most $\lfloor \log_2 N \rfloor$ questions are required to determine the number, since each question halves the search space. Note that one less question (iteration) is required than for the general algorithm, since the number is already constrained to be within a particular range.
Even if the number to guess can be arbitrarily large, in which case there is no upper bound \( N \), The number can be found in at most \( 2\lfloor \log_2 k \rfloor \) steps (where \( k \) is the (unknown) selected number) by first finding an upper bound by repeated doubling. For example, if the number were 11, the following sequence of guesses could be used to find it: 1, 2, 4, 8, 16, 12, 10, 11

One could also extend the method to include negative numbers; for example the following guesses could be used to find \(-13\): 0, –1, –2, –4, –8, –16, –12, –14, –13.

**Word lists**

People typically use a mixture of the binary search and interpolative search algorithms when searching a telephone book, after the initial guess we exploit the fact that the entries are sorted and can rapidly find the required entry. For example when searching for Smith, if Rogers and Thomas have been found, one can flip to a page about halfway between the previous guesses. If this shows Samson, it can be concluded that Smith is somewhere between the Samson and Thomas pages so these can be divided.

**Applications to complexity theory**

Even if we do not know a fixed range the number \( k \) falls in, we can still determine its value by asking \( 2\lfloor \log_2 k \rfloor \) simple yes/no questions of the form "Is \( k \) greater than \( x \)?" for some number \( x \). As a simple consequence of this, if you can answer the question "Is this integer property \( k \) greater than a given value?" in some amount of time then you can find the value of that property in the same amount of time with an added factor of \( \log_2 k \). This is called a **reduction**, and it is because of this kind of reduction that most complexity theorists concentrate on decision problems, algorithms that produce a simple yes/no answer.

For example, suppose we could answer "Does this \( n \times n \) matrix have determinant larger than \( k \)?" in \( O(n^2) \) time. Then, by using binary search, we could find the (ceiling of the) determinant itself in \( O(n^2 \log d) \) time, where \( d \) is the determinant; notice that \( d \) is not the size of the input, but the size of the output.

**Performance**

With each test that fails to find a match at the probed position, the search is continued with one or other of the two sub-intervals, each at most half the size. More precisely, if the number of items, \( N \), is odd then both sub-intervals will contain \((N-1)/2\) elements, while if \( N \) is even then the two sub-intervals contain \( N/2 - 1 \) and \( N/2 \) elements.

If the original number of items is \( N \) then after the first iteration there will be at most \( N/2 \) items remaining, then at most \( N/4 \) items, at most \( N/8 \) items, and so on. In the worst case, when the value is not in the list, the algorithm must continue iterating until the span has been made empty; this will have taken at most \( \lfloor \log_2(N) \rfloor + 1 \) iterations, where the \( \lfloor \cdot \rfloor \) notation denotes the floor function that rounds its argument down to an integer. This worst case analysis is tight: for any \( N \) there exists a query that takes exactly \( \lfloor \log_2(N) \rfloor + 1 \) iterations. When compared to linear search, whose worst-case behaviour is \( N \) iterations, we see that binary search is substantially faster as \( N \) grows large. For example, to search a list of one million items takes as many as one million iterations with linear search, but never more than twenty iterations with binary search. However, a binary search can only be performed if the list is in sorted order.

**Average performance**

\( \log_2(N) - 1 \) is the expected number of probes in an average successful search, and the worst case is \( \log_2(N) \) just one more probe. If the list is empty, no probes at all are made. Thus binary search is a logarithmic algorithm and executes in \( O(\log N) \) time. In most cases it is considerably faster than a linear search. It can be implemented using iteration, or recursion. In some languages it is more elegantly expressed recursively; however, in some C-based languages tail recursion is not eliminated and the recursive version requires more stack space.
Binary search can interact poorly with the memory hierarchy (i.e. caching), because of its random-access nature. For in-memory searching, if the span to be searched is small, a linear search may have superior performance simply because it exhibits better locality of reference. For external searching, care must be taken or each of the first several probes will lead to a disk seek. A common method is to abandon binary searching for linear searching as soon as the size of the remaining span falls below a small value such as 8 or 16 or even more in recent computers. The exact value depends entirely on the machine running the algorithm.

Notice that for multiple searches with a fixed value for N, then (with the appropriate regard for integer division), the first iteration always selects the middle element at N/2, and the second always selects either N/4 or 3N/4, and so on. Thus if the array's key values are in some sort of slow storage (on a disc file, in virtual memory, not in the cpu's on-chip memory), keeping those three keys in a local array for a special preliminary search will avoid accessing widely separated memory. Escalating to seven or fifteen such values will allow further levels at not much cost in storage. On the other hand, if the searches are frequent and not separated by much other activity, the computer's various storage control features will more or less automatically promote frequently-accessed elements into faster storage.

When multiple binary searches are to be performed for the same key in related lists, fractional cascading can be used to speed up successive searches after the first one.

**Extensions**

There is no particular requirement that the array being searched has the bounds 1 to N. It is possible to search a specified range, elements first to last instead of 1 to N. All that is necessary is that the initialisation of the bounds be \( L := \text{first} - 1 \) and \( R := \text{last} + 1 \), then all proceeds as before.

The elements of the list are not necessarily all unique. If one searches for a value that occurs multiple times in the list, the index returned will be of the first-encountered equal element, and this will not necessarily be that of the first, last, or middle element of the run of equal-key elements but will depend on the positions of the values. Modifying the list even in seemingly unrelated ways such as adding elements elsewhere in the list may change the result. To find all equal elements an upward and downward linear search can be carried out from the initial result, stopping each search when the element is no longer equal. Thus, e.g. in a table of cities sorted by country, we can find all cities in a given country.

Several algorithms closely related to or extending binary search exist. For instance, noisy binary search solves the same class of projects as regular binary search, with the added complexity that any given test can return a false value at random. (Usually, the number of such erroneous results are bounded in some way, either in the form of an average error rate, or in the total number of errors allowed per element in the search space.) Optimal algorithms for several classes of noisy binary search problems have been known since the late seventies, and more recently, optimal algorithms for noisy binary search in quantum computers (where several elements can be tested at the same time) have been discovered.

**Variations**

There are many, and they are easily confused. Also, using a binary search within a sorting method is debatable.

**Exclusive or inclusive bounds**

The most significant differences are between the "exclusive" and "inclusive" forms of the bounds. In the "exclusive" bound form the span to be searched is \( (L + 1) \) to \( (R - 1) \), and this may seem clumsy when the span to be searched could be described in the "inclusive" form, as \( L \) to \( R \). Although the details differ the two forms are equivalent as can be seen by transforming one version into the other. The inclusive bound form may be attained by replacing all appearances of "L" by "(L – 1)" and "R" by "(R + 1)" then rearranging. Thus, the initialisation of \( L := 0 \) becomes \( L – \)
1) := 0 or L := 1, and R := N + 1 becomes (R + 1) := N + 1 or R := N. So far so good, but note now that the changes to L and R are no longer simply transferring the value of \( p \) to L or R as appropriate but now must be (R + 1) := p or R := p − 1, and (L − 1) := p or L := p + 1.

Thus, the gain of a simpler initialisation, done once, is lost by a more complex calculation, and which is done for every iteration. If that is not enough, the test for an empty span is more complex also, as compared to the simplicity of checking that the value of \( p \) is zero. Nevertheless, the inclusive bound form is found in many publications, such as Donald Knuth. *The Art of Computer Programming*, Volume 3: Sorting and Searching, Third Edition.

**Deferred detection of equality**

Because of the syntax difficulties discussed below, so that distinguishing the three states <, =, and > would have to be done with two comparisons, it is possible to use just one comparison and at the end when the span is reduced to zero, equality can be tested for. The solution distinguishes only < from >.=.

**Midpoint and width**

An entirely different variation involves abandoning the \( L \) and \( R \) pointers in favour of a current position \( p \) and a width \( w \) where at each iteration, \( p \) is adjusted by + or − \( w \) and \( w \) is halved. Professor Knuth remarks "It is possible to do this, but only if extreme care is paid to the details" — Section 6.2.1, page 414 of *The Art of Computer Programming*, Volume 3: Sorting and Searching, Third Edition, outlines an algorithm, with the further remark "Simpler approaches are doomed to failure!"

**Computer use**

**The algorithm**

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky… — Professor Donald Knuth

When Jon Bentley assigned it as a problem in a course for professional programmers, he found that an astounding ninety percent failed to code a binary search correctly after several hours of working on it[4], and another study shows that accurate code for it is only found in five out of twenty textbooks[5]. Furthermore, Bentley's own implementation of binary search, published in his 1986 book *Programming Pearls*, contains an error that remained undetected for over twenty years.[6]

**Numerical difficulties**

In a practical implementation, the variables used to represent the indices will be of finite size, hence only capable of representing a finite range of values. For example, 32-bit unsigned integers can only hold values from 0 to 4294967295. Most binary search algorithms use 32-bit signed integers, which can only hold values from -2147483648 to 2147483647. If the binary search algorithm is to operate on large arrays, this has two implications:

- The values first − 1 and last + 1 must both be representable within the finite bounds of the chosen integer type. Therefore, continuing the 32-bit unsigned example, the largest value that last may take is +4294967294, not +4294967295. A problem exists even for the "inclusive" form of the method, as if \( x > A(4294967295) \).Key, then on the final iteration the algorithm will attempt to store 4294967296 into L and fail. Equivalent issues apply to the lower limit, where first − 1 could become negative as when the first element of the array is at index zero.

- If the midpoint of the span is calculated as \( p := (L + R) / 2 \), then the value \( (L + R) \) will exceed the number range if last is greater than (for unsigned) 4294967295/2 or (for signed) 2147483647/2 and the search wanders toward the upper end of the search space. This can be avoided by performing the calculation as \( p := (R - L) / 2 + L \). When the problem arises for signed integers, a more efficient alternative is by performing the
calculation as \( p := (R + L) >>> 1 \), where \( >>> \) denotes the right logical shift operator. For example, this bug existed in Java SDK at `Arrays.binarySearch()` from 1.2 to 5.0 and fixed in 6.0.[7]

**Implementations**

**Iterative**

The following incorrect (see notes below) algorithm is slightly modified (to avoid overflow) from Niklaus Wirth’s in standard Pascal.[8]

```
min := 1;
max := N; {array size: var A : array [1..N] of integer}
repeat
  mid := (min+max) div 2;
  if x > A[mid] then
    min := mid + 1;
  else
    max := mid - 1;
  until (A[mid] = x) or (min > max);
```

*Note:* This code assumes 1-based array indexing. For languages that use 0-based indexing (most modern languages), \( min \) and \( max \) should be initialized to 0 and \( N-1 \).

*Note 2:* The code above does not return a result, nor indicates whether the element was found or not.

*Note 3:* The code above will not work correctly for empty arrays, because it attempts to access an element before checking to see if \( min > max \).

*Note 4:* After exiting the loop, the value of \( mid \) does not properly indicate whether the desired value was found in the array. One would need to perform one more comparison to determine if the value \( A[\text{mid}] \) is equal to \( x \).

This code uses inclusive bounds and a three-way test (for early loop termination in case of equality), but with two separate comparisons per iteration. It is not the most efficient solution.

**Recursive**

A simple, straightforward implementation is tail recursive; it recursively searches the subrange dictated by the comparison:

```
binary_search(Array[0..N-1], value, low, high):
  if (high < low):
    return -1 // not found
  mid = (low + high) / 2
  if (A[mid] > value):
    return binary_search(A, value, low, mid-1)
  else if (A[mid] < value):
    return binary_search(A, value, mid+1, high)
  else:
    return mid // found
```

It is invoked with initial \( low \) and \( high \) values of 0 and \( N-1 \).
Language support

Many standard libraries provide a way to do a binary search:

- C provides algorithm function bsearch in its standard library.
- C++'s STL provides algorithm functions binary_search, lower_bound and upper_bound.
- Java offers a set of overloaded binarySearch() static methods in the classes Arrays and Collections in the standard java.util package for performing binary searches on Java arrays and on Lists, respectively. They must be arrays of primitives, or the arrays or Lists must be of a type that implements the Comparable interface, or you must specify a custom Comparator object.
- Microsoft's .NET Framework 2.0 offers static generic versions of the binary search algorithm in its collection base classes. An example would be System.Array's method BinarySearch<T>(T[] array, T value).
- Python provides the bisect module.
- COBOL can perform binary search on internal tables using the SEARCH ALL statement.
- Perl can perform a generic binary search using the CPAN module Search::Binary.
- Go's standard library package contains functions Search, SearchInts, SearchFloat64s, and SearchStrings, which implement general binary search, as well as specific implementations for searching slices of integers, floating-point numbers, and strings, respectively.[11]
- For Objective-C, the Cocoa framework provides the NSArray -indexOfObject:inSortedRange:options:usingComparator: method in Mac OS X 10.6+. Apple's Core Foundation C framework also contains a CFArrayBSearchValues() function.

References

[10] CPAN: Search::Binary (http://search.cpan.org/~rant/Search-Binary-0.95/Binary.pm)

External links

- Google Research: Nearly All Binary Searches and Mergesorts are Broken (http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html)
- Binary search implemented in 12 languages (http://www.codecodex.com/wiki/Binary_search)
In computer science, a **binary search tree (BST)**, which may sometimes also be called an **ordered** or **sorted binary tree**, is a node-based binary tree data structure which has the following properties:\[1\]

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

### Operations

Operations on a binary search tree require comparisons between nodes. These comparisons are made with calls to a comparator, which is a subroutine that computes the total order (linear order) on any two values. This comparator can be explicitly or implicitly defined, depending on the language in which the BST is implemented.

### Searching

Searching a binary search tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method.

We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item
must not be present in the tree.

Here is the search algorithm in the Python programming language:

```python
# 'node' refers to the parent-node in this case
def search_binary_tree(node, key):
    if node is None:
        return None  # key not found
    if key < node.key:
        return search_binary_tree(node.leftChild, key)
    elif key > node.key:
        return search_binary_tree(node.rightChild, key)
    else:
        return node.value  # found key
```

... or equivalent Haskell:

```haskell
searchBinaryTree _ NullNode = Nothing
searchBinaryTree key (Node nodeKey nodeValue (leftChild, rightChild)) =
    case compare key nodeKey of
    LT -> searchBinaryTree key leftChild
    GT -> searchBinaryTree key rightChild
    EQ -> Just nodeValue
```

This operation requires $O(\log n)$ time in the average case, but needs $O(n)$ time in the worst case, when the unbalanced tree resembles a linked list (degenerate tree).

Assuming that BinarySearchTree is a class with a member function "search(int)" and a pointer to the root node, the algorithm is also easily implemented in terms of an iterative approach. The algorithm enters a loop, and decides whether to branch left or right depending on the value of the node at each parent node.

```c++
bool BinarySearchTree::search(int val)
{
    Node *next = this->root();

    while (next != NULL) {
        if (val == next->value()) {
            return true;
        } else if (val < next->value()) {
            next = next->left();
        } else {
            next = next->right();
        }
    }

    //not found
    return false;
}
```
Insertion

Insertion begins as a search would begin: if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in C++:

```cpp
/* Inserts the node pointed to by "newNode" into the subtree rooted at "treeNode" */
void InsertNode(Node* &treeNode, Node *newNode)
{
    if (treeNode == NULL)
        treeNode = newNode;
    else if (newNode->key < treeNode->key)
        InsertNode(treeNode->left, newNode);
    else
        InsertNode(treeNode->right, newNode);
}
```

The above "destructive" procedural variant modifies the tree in place. It uses only constant space, but the previous version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

```python
def binary_tree_insert(node, key, value):
    if node is None:
        return TreeNode(None, key, value, None)
    if key == node.key:
        return TreeNode(node.left, key, value, node.right)
    if key < node.key:
        return TreeNode(binary_tree_insert(node.left, key, value),
                         node.key, node.value, node.right)
    else:
        return TreeNode(node.left, node.key, node.value,
                         binary_tree_insert(node.right, key, value))
```

The part that is rebuilt uses $O(\log n)$ space in the average case and $O(n)$ in the worst case (see big-O notation).

In either version, this operation requires time proportional to the height of the tree in the worst case, which is $O(\log n)$ time in the average case over all trees, but $O(n)$ time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

Here is an iterative approach to inserting into a binary search tree in Java:

```java
private Node m_root;

public void insert(int data) {
```
if (m_root == null) {
    m_root = new TreeNode(data, null, null);
    return;
}
Node root = m_root;
while (root != null) {
    // Not the same value twice
    if (data == root.getData()) {
        return;
    } else if (data < root.getData()) {
        // insert left
        if (root.getLeft() == null) {
            root.setLeft(new TreeNode(data, null, null));
            return;
        } else {
            root = root.getLeft();
        }
    } else {
        // insert right
        if (root.getRight() == null) {
            root.setRight(new TreeNode(data, null, null));
            return;
        } else {
            root = root.getRight();
        }
    }
}

Below is a recursive approach to the insertion method.

private Node m_root;

public void insert(int data){
    if (m_root == null) {
        m_root = TreeNode(data, null, null);
    } else {
        internalInsert(m_root, data);
    }
}

private static void internalInsert(Node node, int data){
    // Not the same value twice
    if (data == node.mValue) {
        return;
    } else if (data < node.mValue) {
        if (node.getLeft() == null) {
            node.setLeft(new TreeNode(data, null, null));
        } else {
            internalInsert(node.getLeft(), data);
        }
    } else {
        if (node.getRight() == null) {
            node.setRight(new TreeNode(data, null, null));
        } else {
            internalInsert(node.getRight(), data);
        }
    }
}
Deletion

There are three possible cases to consider:

- **Deleting a leaf (node with no children):** Deleting a leaf is easy, as we can simply remove it from the tree.
- **Deleting a node with one child:** Remove the node and replace it with its child.
- **Deleting a node with two children:** Call the node to be deleted \( N \). Do not delete \( N \). Instead, choose either its in-order successor node or its in-order predecessor node, \( R \). Replace the value of \( N \) with the value of \( R \), then delete \( R \).

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so good implementations add inconsistency to this selection.

Running Time Analysis: Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

Here is the code in Python:

```python
def findMin(self):
    '''
    Finds the smallest element that is a child of *self*
    '''
    current_node = self
    while current_node.left_child:
        current_node = current_node.left_child
    return current_node
```
def replace_node_in_parent(self, new_value=None):
    
    # Removes the reference to *self* from *self.parent* and replaces it with *new_value*.
    
    if self.parent:
        if self == self.parent.left_child:
            self.parent.left_child = new_value
        else:
            self.parent.right_child = new_value
    if new_value:
        new_value.parent = self.parent

def binary_tree_delete(self, key):
    if key < self.key:
        self.left_child.binary_tree_delete(key)
    elif key > self.key:
        self.right_child.binary_tree_delete(key)
    else:
        # delete the key here
        if self.left_child and self.right_child:
            # get the smallest node that's bigger than *self*
            successor = self.right_child.findMin()
            self.key = successor.key
            # if *successor* has a child, replace it with that
            # at this point, it can only have a *right_child*
            # if it has no children, *right_child* will be "None"
            successor.replace_node_in_parent(successor.right_child)
        elif self.left_child or self.right_child:
            # if the node has only one child
            if self.left_child:
                self.replace_node_in_parent(self.left_child)
            else:
                self.replace_node_in_parent(self.right_child)
        else:
            # this node has no children
            self.replace_node_in_parent(None)

Source code in C++ (from http://www.algolist.net/Data_structures/Binary_search_tree). This URL also explains the operation nicely using diagrams.

bool BinarySearchTree::remove(int value) {
    if (root == NULL)
        return false;
    else {
        if (root->getValue() == value) {
            BSTNode auxRoot(0);
            auxRoot.setLeftChild(root);
            BSTNode* removedNode = root->remove(value, &auxRoot);
        }
    }
root = auxRoot.getLeft();
if (removedNode != NULL) {
    delete removedNode;
    return true;
} else
    return false;
} else {
    BSTNode* removedNode = root->remove(value, NULL);
    if (removedNode != NULL) {
        delete removedNode;
        return true;
    } else
        return false;
}

BSTNode* BSTNode::remove(int value, BSTNode *parent) {
if (value < this->value) {
    if (left != NULL)
        return left->remove(value, this);
    else
        return NULL;
} else if (value > this->value) {
    if (right != NULL)
        return right->remove(value, this);
    else
        return NULL;
} else {
    if (left != NULL && right != NULL) {
        this->value = right->minValue();
        return right->remove(this->value, this);
    } else if (parent->left == this) {
        parent->left = (left != NULL) ? left : right;
        return this;
    } else if (parent->right == this) {
        parent->right = (left != NULL) ? left : right;
        return this;
    }
}

int BSTNode::minValue() {
if (left == NULL)
    return value;
else
    return left->minValue();
Traversal

Once the binary search tree has been created, its elements can be retrieved in-order by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it’s recursively accessed. As with all binary trees, one may conduct a pre-order traversal or a post-order traversal, but neither are likely to be useful for binary search trees.

The code for in-order traversal in Python is given below. It will call **callback** for every node in the tree.

```python
def traverse_binary_tree(node, callback):
    if node is None:
        return
    traverse_binary_tree(node.leftChild, callback)
    callback(node.value)
    traverse_binary_tree(node.rightChild, callback)
```

Traversal requires Ω(n) time, since it must visit every node. This algorithm is also O(n), so it is asymptotically optimal.

Sort

A binary search tree can be used to implement a simple but efficient sorting algorithm. Similar to heapsort, we insert all the values we wish to sort into a new ordered data structure—in this case a binary search tree—and then traverse it in order, building our result:

```python
def build_binary_tree(values):
    tree = None
    for v in values:
        tree = binary_tree_insert(tree, v)
    return tree
def get_inorder_traversal(root):
    '''
    Returns a list containing all the values in the tree, starting at
    *root*.
    Traverses the tree in-order(leftChild, root, rightChild).
    '''
    result = []
    traverse_binary_tree(root, lambda element: result.append(element))
    return result
```

The worst-case time of **build_binary_tree** is $O(n^2)$—if you feed it a sorted list of values, it chains them into a linked list with no left subtrees. For example, **build_binary_tree([1, 2, 3, 4, 5])** yields the tree (1 (2 (3 (4 (5))))).

There are several schemes for overcoming this flaw with simple binary trees; the most common is the self-balancing binary search tree. If this same procedure is done using such a tree, the overall worst-case time is $O(n \log n)$, which is asymptotically optimal for a comparison sort. In practice, the poor cache performance and added overhead in time and space for a tree-based sort (particularly for node allocation) make it inferior to other asymptotically optimal sorts such as heapsort for static list sorting. On the other hand, it is one of the most efficient methods of **incremental**
sorting, adding items to a list over time while keeping the list sorted at all times.

Types
There are many types of binary search trees. AVL trees and red-black trees are both forms of self-balancing binary search trees. A splay tree is a binary search tree that automatically moves frequently accessed elements nearer to the root. In a treap ("tree heap"), each node also holds a (randomly chosen) priority and the parent node has higher priority than its children. Tango Trees are trees optimized for fast searches.

Two other titles describing binary search trees are that of a complete and degenerate tree. A complete tree is a tree with n levels, where for each level d <= n - 1, the number of existing nodes at level d is equal to $2^d$. This means all possible nodes exist at these levels. An additional requirement for a complete binary tree is that for the nth level, while every node does not have to exist, the nodes that do exist must fill from left to right.

A degenerate tree is a tree where for each parent node, there is only one associated child node. What this means is that in a performance measurement, the tree will essentially behave like a linked list data structure.

Performance comparisons
D. A. Heger (2004) presented a performance comparison of binary search trees. Treap was found to have the best average performance, while red-black tree was found to have the smallest amount of performance fluctuations.

Optimal binary search trees
If we don't plan on modifying a search tree, and we know exactly how often each item will be accessed, we can construct an optimal binary search tree, which is a search tree where the average cost of looking up an item (the expected search cost) is minimized.

Even if we only have estimates of the search costs, such a system can considerably speed up lookups on average. For example, if you have a BST of English words used in a spell checker, you might balance the tree based on word frequency in text corpora, placing words like "the" near the root and words like "agement" near the leaves. Such a tree might be compared with Huffman trees, which similarly seek to place frequently-used items near the root in order to produce a dense information encoding; however, Huffman trees only store data elements in leaves and these elements need not be ordered.

If we do not know the sequence in which the elements in the tree will be accessed in advance, we can use splay trees which are asymptotically as good as any static search tree we can construct for any particular sequence of lookup operations.

Alphabetic trees are Huffman trees with the additional constraint on order, or, equivalently, search trees with the modification that all elements are stored in the leaves. Faster algorithms exist for optimal alphabetic binary trees (OABTs).

Example:

```
procedure Optimum Search Tree(f, f', c):
    for j = 0 to n do
        c[j, j] = 0, F[j, j] = f'j
    for d = 1 to n do
        for i = 0 to (n - d) do
            j = i + d
            F[i, j] = F[i, j-1] + f' + f'j
            c[i, j] = MIN(i<k<=j){c[i, k-1] + c[k, j]} + F[i, j]
```
References


Further reading


External links

• Binary Search Tree C++ and Pascal (http://akyltist.ucoz.org/publ/algorithms/binary_search_tree_cpp_pas/3-1-0-4-)
• Binary Search Trees Animation (http://www.student.seas.gwu.edu/~idsv/idsv.html)
• Full source code to an efficient implementation in C++ (http://jdservr.homelinux.org/wiki/Binary_Search_Tree)
• Implementation of a Persistent Binary Search Tree in C (http://cgi.scs.carleton.ca/~dana/pbst)
• Iterative Implementation of Binary Search Trees in C# (http://www.goletas.com/solutions/collections/)
• An introduction to binary trees from Stanford (http://cslibrary.stanford.edu/110/)
• Dictionary of Algorithms and Data Structures - Binary Search Tree (http://www.nist.gov/dads/HTML/binarySearchTree.html)
• Binary Search Tree Example in Python (http://code.activestate.com/recipes/286239/)
• Interactive Data Structure Visualizations - Binary Tree Traversals (http://nova.umuc.edu/~jarc/idsv/lesson1.html)
• Literate implementations of binary search trees in various languages (http://en.literateprograms.org/Category:Binary_search_tree) on LiteratePrograms
• BST Tree Applet (http://people.ksp.sk/~kuko/bak/index.html) by Kubo Kovac
• Well-illustrated explanation of binary search tree. Implementations in Java and C++ (http://www.algolist.net/Data_structures/Binary_search_tree)
• Teaching Binary Search Tree through visualization (http://employees.oneonta.edu/zhangs/PowerPointPlatform/index.php)
Random binary tree

In computer science and probability theory, a random binary tree refers to a binary tree selected at random from some probability distribution on binary trees. Two different distributions are commonly used: binary trees formed by inserting nodes one at a time according to a random permutation, and binary trees chosen from a uniform discrete distribution in which all distinct trees are equally likely. It is also possible to form other distributions, for instance by repeated splitting. Adding and removing nodes directly in a random binary tree will in general disrupt its random structure, but the treap and related randomized binary search tree data structures use the principle of binary trees formed from a random permutation in order to maintain a balanced binary search tree dynamically as nodes are inserted and deleted.

For random trees that are not necessarily binary, see random tree.

Binary trees from random permutations

For any set of numbers (or, more generally, values from some total order), one may form a binary search tree in which each number is inserted in sequence as a leaf of the tree, without changing the structure of the previously inserted numbers. The position into which each number should be inserted is uniquely determined by a binary search in the tree formed by the previous numbers. For instance, if the three numbers (1,3,2) are inserted into a tree in that sequence, the number 1 will sit at the root of the tree, the number 3 will be placed as its right child, and the number 2 as the left child of the number 3. There are six different permutations of the numbers (1,2,3), but only five trees may be constructed from them. That is because the permutations (2,1,3) and (2,3,1) form the same tree.

Expected depth of a node

For any fixed choice of a value \( x \) in the given set of numbers, if one randomly permutes the numbers and forms a binary tree from them as described above, the expected value of the length of the path from the root of the tree to \( x \) is at most \( 2 \log x + O(1) \), where "log" denotes the natural logarithm function and the \( O \) introduces big O notation.

For, the expected number of ancestors of \( x \) is by linearity of expectation equal to the sum, over all other values \( y \) in the set, of the probability that \( y \) is an ancestor of \( x \). And a value \( y \) is an ancestor of \( x \) exactly when \( y \) is the first element to be inserted from the elements in the interval \([x,y]\). Thus, the values that are adjacent to \( x \) in the sorted sequence of values have probability 1/2 of being an ancestor of \( x \), the values one step away have probability 1/3, etc. Adding these probabilities for all positions in the sorted sequence gives twice a Harmonic number, leading to the bound above. A bound of this form holds also for the expected search length of a path to a fixed value \( x \) that is not part of the given set.\[1\]

The longest path

Although not as easy to analyze as the average path length, there has also been much research on determining the expectation (or high probability bounds) of the length of the longest path in a binary search tree generated from a random insertion order. It is now known that this length, for a tree with \( n \) nodes, is almost surely

\[
\frac{1}{\beta} \log n \approx 4.311 \log n,
\]

where \( \beta \) is the unique number in the range \( 0 < \beta < 1 \) satisfying the equation

\[
2\beta e^{1-\beta} = 1.\[2\]
Expected number of leaves

In the random permutation model, each of the numbers from the set of numbers used to form the tree, except for the smallest and largest of the numbers, has probability 1/3 of being a leaf in the tree, for it is a leaf when it inserted after its two neighbors, and any of the six permutations of these two neighbors and it are equally likely. By similar reasoning, the smallest and largest of the numbers have probability 1/2 of being a leaf. Therefore, the expected number of leaves is the sum of these probabilities, which for \( n \geq 2 \) is exactly \((n + 1)/3\).

Treaps and randomized binary search trees

In applications of binary search tree data structures, it is rare for the values in the tree to be inserted without deletion in a random order, limiting the direct applications of random binary trees. However, algorithm designers have devised data structures that allow insertions and deletions to be performed in a binary search tree, at each step maintaining as an invariant the property that the shape of the tree is a random variable with the same distribution as a random binary search tree.

If a given set of ordered numbers is assigned numeric priorities (distinct numbers unrelated to their values), these priorities may be used to construct a Cartesian tree for the numbers, a binary tree that has as its inorder traversal sequence the sorted sequence of the numbers and that is heap-ordered by priorities. Although more efficient construction algorithms are known, it is helpful to think of a Cartesian tree as being constructed by inserting the given numbers into a binary search tree in priority order. Thus, by choosing the priorities either to be a set of independent random real numbers in the unit interval, or by choosing them to be a random permutation of the numbers from 1 to \( n \) (where \( n \) is the number of nodes in the tree), and by maintaining the heap ordering property using tree rotations after any insertion or deletion of a node, it is possible to maintain a data structure that behaves like a random binary search tree. Such a data structure is known as a treap or a randomized binary search tree.[3]

Uniformly random binary trees

The number of binary trees with \( n \) nodes is a Catalan number: for \( n = 1, 2, 3, \ldots \) these numbers of trees are

\[
1, 2, 5, 14, 42, 132, 4862, 16796, \ldots \text{ (sequence A000108 in OEIS)}.
\]

Thus, if one of these trees is selected uniformly at random, its probability is the reciprocal of a Catalan number. Trees in this model have expected depth proportional to the square root of \( n \), rather than to the logarithm,[4] however, the Strahler number of a uniformly random binary tree, a more sensitive measure of the distance from a leaf in which a node has Strahler number \( i \) whenever it has either a child with that number or two children with number \( i - 1 \), is with high probability logarithmic.[5]

Due to their large heights, this model of equiprobable random trees is not generally used for binary search trees, but it has been applied to problems of modeling the parse trees of algebraic expressions in compiler design[6] (where the above-mentioned bound on Strahler number translates into the number of registers needed to evaluate an expression[7]) and for modeling evolutionary trees.[8] In some cases the analysis of random binary trees under the random permutation model can be automatically transferred to the uniform model.[9]

Random split trees

Devroye & Kruszewski (1996) generate random binary trees with \( n \) nodes by generating a real-valued random variable \( x \) in the unit interval \((0,1)\), assigning the first \( \lfloor xn \rfloor \) nodes (rounded down to an integer number of nodes) to the left subtree, the next node to the root, and the remaining nodes to the right subtree, and continuing recursively in each subtree. If \( x \) is chosen uniformly at random in the interval, the result is the same as the random binary search tree generated by a random permutation of the nodes, as any node is equally likely to be chosen as root; however, this formulation allows other distributions to be used instead. For instance, in the uniformly random binary tree model, once a root is fixed each of its two subtrees must also be uniformly random, so the uniformly random model
Random binary tree may also be generated by a different choice of distribution for \( x \). As Devroye and Kruszewski show, by choosing a beta distribution on \( x \) and by using an appropriate choice of shape to draw each of the branches, the mathematical trees generated by this process can be used to create realistic-looking botanical trees.

Notes

[5] Devroye & Kruszewski (1995). That it is at most logarithmic is trivial, because the Strahler number of every tree is bounded by the logarithm of the number of its nodes.

References

• Knuth, Donald M. (2005), "Draft of Section 7.2.1.6: Generating All Trees" (http://www-cs-faculty.stanford.edu/~knuth/fasc4a.ps.gz), The Art of Computer Programming, IV.
Tree rotation

A **tree rotation** is an operation on a binary tree that changes the structure without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down. It is used to change the shape of the tree, and in particular to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations.

There exists an inconsistency in different descriptions as to the definition of the **direction of rotations**. Some say that the direction of a rotation depends on the side which the tree nodes are shifted upon whilst others say that it depends on which child takes the root's place (opposite of the former). This article takes the approach of the side where the nodes get shifted to.

**Illustration**

The right rotation operation as shown in the image above is performed with Q as the root and hence is a right rotation on, or rooted at, Q. This operation results in a rotation of the tree in the clockwise direction. The inverse operation is the left rotation, which results in a movement in a counter-clockwise direction (the left rotation shown above is rooted at P). The key to understanding how a rotation functions is to understand its constraints. In particular the order of the leaves of the tree (when read left to right for example) cannot change (another way to think of it is that the order that the leaves would be visited in a depth first search must be the same after the operation as before ).

Another constraint is the main property of a binary search tree, namely that the right child is greater than the parent and the left child is lesser than the parent. Notice that the right child of a left child of the root of a sub-tree (for example node B in the diagram for the tree rooted at Q) can become the left child of the root, that itself becomes the right child of the "new" root in the rotated sub-tree, without violating either of those constraints. As you can see in the diagram, the order of the leaves doesn't change. The opposite operation also preserves the order and is the second kind of rotation.

Assuming this is a binary search tree, as stated above, the elements must be interpreted as variables that can be compared to each other. The alphabetic characters above are used as placeholders for these variables.
**Detailed Illustration**

When a subtree is rotated, the subtree side upon which it is rotated decreases its height by one node while the other subtree increases its height. This makes tree rotations useful for rebalancing a tree.

Using the terminology of **Root** for the parent node of the subtrees to rotate, **Pivot** for the node which will become the new parent node, **RS** for rotation side upon to rotate and **OS** for opposite side of rotation. In the above diagram for the root Q, the **RS** is C and the **OS** is P. The pseudo code for the rotation is:

\[
\begin{align*}
\text{Pivot} &= \text{Root}\text{.OS} \\
\text{Root}\text{.OS} &= \text{Pivot}\text{.RS} \\
\text{Pivot}\text{.RS} &= \text{Root} \\
\text{Root} &= \text{Pivot}
\end{align*}
\]

This is a constant time operation.

The programmer must also make sure that the root's parent points to the pivot after the rotation. Also, the programmer should note that this operation may result in a new root for the entire tree and take care to update pointers accordingly.

**Inorder Invariance**

The tree rotation renders the inorder traversal of the binary tree invariant. This implies the order of the elements are not affected when a rotation is performed in any part of the tree. Here are the inorder traversals of the trees shown above:

- **Left tree:** ((A, P, B), Q, C)
- **Right tree:** (A, P, (B, Q, C))

Computing one from the other is very simple. The following is example Python code that performs that computation:

```python
def right_rotation(treenode):
    left, Q, C = treenode
    A, P, B = left
    return (A, P, (B, Q, C))
```

Another way of looking at it is:

**Right Rotation of node Q:**

1. Let P be Q's left child.
2. Set P to be the new root.
3. Set Q's left child to be P's right child.
4. Set P's right child to be Q.
Left Rotation of node P:

Let Q be P's right child.
Set Q to be the new root.
Set P's right child to be Q's left child.
Set Q's left child to be P.

All other connections are left as-is.

There are also double rotations, which are combinations of left and right rotations. A double left rotation at X can be defined to be a right rotation at the right child of X followed by a left rotation at X; similarly, a double right rotation at X can be defined to be a left rotation at the left child of X followed by a right rotation at X.

Tree rotations are used in a number of tree data structures such as AVL trees, red-black trees, splay trees, and treaps. They require only constant time because they are local transformations: they only operate on 5 nodes, and need not examine the rest of the tree.

**Rotations for rebalancing**

A tree can be rebalanced using rotations. After a rotation, the side of the rotation increases its height by 1 whilst the side opposite the rotation decreases its height similarly. Therefore, one can strategically apply rotations to nodes whose left child and right child differ in height by more than 1. Self-balancing binary search trees apply this operation automatically. A type of tree which uses this rebalancing technique is the AVL tree.

**Rotation distance**

The rotation distance between any two binary trees with the same number of nodes is the minimum number of rotations needed to transform one into the other. With this distance, the set of \( n \)-node binary trees becomes a metric space: the distance is symmetric, positive when given two different trees, and satisfies the triangle inequality.

It is an open problem whether there exists a polynomial time algorithm for calculating rotation distance.

Daniel Sleator, Robert Tarjan and William Thurston showed that the rotation distance between any two \( n \)-node trees (for \( n \geq 11 \)) is at most \( 2n - 6 \), and that infinitely many pairs of trees are this far apart.\(^1\)

**References**


**External links**

- Java applets demonstrating tree rotations (http://www.cs.queensu.ca/home/jstewart/applets/bst/bst-rotation.html)
- The AVL Tree Rotations Tutorial (http://fortheloot.com/public/AVLTreeTutorial.rtf) (RTF) by John Hargrove
Self-balancing binary search tree

In computer science, a self-balancing (or height-balanced) binary search tree is any node based binary search tree that automatically keeps its height (number of levels below the root) small in the face of arbitrary item insertions and deletions.\[1\]

These structures provide efficient implementations for mutable ordered lists, and can be used for other abstract data structures such as associative arrays, priority queues and sets.

**Overview**

Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. A binary tree with height $h$ can contain at most $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ nodes. It follows that for a tree with $n$ nodes and height $h$:

\[
n \leq 2^{h+1} - 1
\]

And that implies:

\[
h \geq \lceil \log_2(n + 1) - 1 \rceil \geq \lceil \log_2 n \rceil .
\]

In other words, the minimum height of a tree with $n$ nodes is $\log_2(n)$, rounded down; that is, $\lceil \log_2 n \rceil$.\[1\]

However, the simplest algorithms for BST item insertion may yield a tree with height $h$ in rather common situations. For example, when the items are inserted in sorted key order, the tree degenerates into a linked list with $n$ nodes. The difference in performance between the two situations may be enormous: for $n = 1,000,000$, for example, the minimum height is $\ln 1,000,000$.\[2\]

If the data items are known ahead of time, the height can be kept small, in the average sense, by adding values in a random order, resulting in a random binary search tree. However, there are many situations (such as online algorithms) where this randomization is not viable.

Self-balancing binary trees solve this problem by performing transformations on the tree (such as tree rotations) at key times, in order to keep the height proportional to $\log_2(n)$. Although a certain overhead is involved, it may be justified in the long run by ensuring fast execution of later operations.

Maintaining the height always at its minimum value $\lceil \log_2(n) \rceil$ is not always viable; it can be proven that any insertion algorithm which did so would have an excessive overhead. Therefore, most self-balanced BST algorithms keep the height within a constant factor of this lower bound.
In the asymptotic ("Big-O") sense, a self-balancing BST structure containing \( n \) items allows the lookup, insertion, and removal of an item in \( O(\log n) \) worst-case time, and ordered enumeration of all items in \( O(n) \) time. For some implementations these are per-operation time bounds, while for others they are amortized bounds over a sequence of operations. These times are asymptotically optimal among all data structures that manipulate the key only through comparisons.

**Implementations**

Popular data structures implementing this type of tree include:

- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

**Applications**

Self-balancing binary search trees can be used in a natural way to construct and maintain ordered lists, such as priority queues. They can also be used for associative arrays; key-value pairs are simply inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables. One advantage of self-balancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items in *key order*, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key.

Self-balancing BSTs can be used to implement any algorithm that requires mutable ordered lists, to achieve optimal worst-case asymptotic performance. For example, if binary tree sort is implemented with a self-balanced BST, we have a very simple-to-describe yet asymptotically optimal \( O(n \log n) \) sorting algorithm. Similarly, many algorithms in computational geometry exploit variations on self-balancing BSTs to solve problems such as the line segment intersection problem and the point location problem efficiently. (For average-case performance, however, self-balanced BSTs may be less efficient than other solutions. Binary tree sort, in particular, is likely to be slower than mergesort or quicksort, because of the tree-balancing overhead as well as cache access patterns.)

Self-balancing BSTs are flexible data structures, in that it's easy to extend them to efficiently record additional information or perform new operations. For example, one can record the number of nodes in each subtree having a certain property, allowing one to count the number of nodes in a certain key range with that property in \( O(\log n) \) time. These extensions can be used, for example, to optimize database queries or other list-processing algorithms.

**References**


**External links**

- GNU libavl (http://adtinfo.org/), a LGPL-licensed library of binary tree implementations in C, with documentation
Treap

In computer science, the treap and the randomized binary search tree are two closely related forms of binary search tree data structures that maintain a dynamic set of ordered keys and allow binary searches among the keys. After any sequence of insertions and deletions of keys, the shape of the tree is a random variable with the same probability distribution as a random binary tree; in particular, with high probability its height is proportional to the logarithm of the number of keys, so that each search, insertion, or deletion operation takes logarithmic time to perform.

Treap

The treap was first described by Cecilia R. Aragon and Raimund Seidel in 1989[1][2]; its name is a portmanteau of tree and heap. It is a Cartesian tree[3] in which each key is given a (randomly chosen) numeric priority. As with any binary search tree, the inorder traversal order of the nodes is the same as the sorted order of the keys. The structure of the tree is determined by the requirement that it be heap-ordered: that is, the priority number for any non-leaf node must be greater than or equal to the priority of its children. Thus, as with Cartesian trees more generally, the root node is the maximum-priority node, and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

An equivalent way of describing the treap is that it could be formed by inserting the nodes highest-priority-first into a binary search tree without doing any rebalancing. Therefore, if the priorities are independent random numbers (from a distribution over a large enough space of possible priorities to ensure that two nodes are very unlikely to have the same priority) then the shape of a treap has the same probability distribution as the shape of a random binary search tree, a search tree formed by inserting the nodes without rebalancing in a randomly chosen insertion order. Because random binary search trees are known to have logarithmic height with high probability, the same is true for treaps.

Specifically, the treap supports the following operations:

- To search for a given key value, apply a standard binary search algorithm in a binary search tree, ignoring the priorities.
- To insert a new key \( x \) into the treap, generate a random priority \( y \) for \( x \). Binary search for \( x \) in the tree, and create a new node at the leaf position where the binary search determines a node for \( x \) should exist. Then, as long as \( x \) is not the root of the tree and has a larger priority number than its parent \( z \), perform a tree rotation that reverses the parent-child relation between \( x \) and \( z \).
- To delete a node \( x \) from the treap, if \( x \) is a leaf of the tree, simply remove it. If \( x \) has a single child \( z \), remove \( x \) from the tree and make \( z \) be the child of the parent of \( x \) (or make \( z \) the root of the tree if \( x \) had no parent). Finally, if \( x \) has two children, swap its position in the tree with the position of its immediate successor \( z \) in the sorted order, resulting in one of the previous cases. In this final case, the swap may violate the heap-ordering property for \( z \), so additional rotations may need to be performed to restore this property.
- To split a treap into two smaller treaps, those smaller than key \( x \), and those larger than key \( x \), insert \( x \) into the treap with maximum priority—larger than the priority of any node in the treap. After this insertion, \( x \) will be the root node of the treap, all values less than \( x \) will be found in the left subtree, and all values greater than \( x \) will be
found in the right subtreap. This costs as much as a single insertion into the treap.
• Merging two treaps (assumed to be the product of a former split), one can safely assume that the greatest value in
  the first treap is less than the smallest value in the second treap. Insert a value $x$, such that $x$ is larger than this
  max-value in the first treap, and smaller than the min-value in the second treap, and assign it the minimum
  priority. After insertion it will be a leaf node, and can easily be deleted. The result is one treap merged from the
  two original treaps. This is effectively “undoing” a split, and costs the same.

Aragon and Seidel also suggest assigning higher priorities to frequently accessed nodes, for instance by a process
that, on each access, chooses a random number and replaces the priority of the node with that number if it is higher
than the previous priority. This modification would cause the tree to lose its random shape; instead, frequently
accessed nodes would be more likely to be near the root of the tree, causing searches for them to be faster.

Blelloch and Reid-Miller\cite{4} describe an application of treaps to a problem of maintaining sets of items and
performing set union, set intersection, and set difference operations, using a treap to represent each set. Naor and
Nissim\cite{5} describe another application, for maintaining authorization certificates in public-key cryptosystems.

### Randomized binary search tree

The randomized binary search tree, introduced by Martínez and Roura subsequently to the work of Aragon and
Seidel on treaps\cite{6}, stores the same nodes with the same random distribution of tree shape, but maintains different
information within the nodes of the tree in order to maintain its randomized structure.

Rather than storing random priorities on each node, the randomized binary search tree stores at each node a small
integer, the number of its descendants (counting itself as one); these numbers may be maintained during tree rotation
operations at only a constant additional amount of time per rotation. When a key $x$ is to be inserted into a tree that
already has $n$ nodes, the insertion algorithm chooses with probability $1/(n + 1)$ to place $x$ as the new root of the tree,
and otherwise it calls the insertion procedure recursively to insert $x$ within the left or right subtree (depending on
whether its key is less than or greater than the root). The numbers of descendants are used by the algorithm to
calculate the necessary probabilities for the random choices at each step. Placing $x$ at the root of a subtree may be
performed either as in the treap by inserting it at a leaf and then rotating it upwards, or by an alternative algorithm
described by Martínez and Roura that splits the subtree into two pieces to be used as the left and right children of the
new node.

The deletion procedure for a randomized binary search tree uses the same information per node as the insertion
procedure, and like the insertion procedure it makes a sequence of $O(\log n)$ random decisions in order to join the two
subtrees descending from the left and right children of the deleted node into a single tree. If the left or right subtree
of the node to be deleted is empty, the join operation is trivial; otherwise, the left or right child of the deleted node is
selected as the new subtree root with probability proportional to its number of descendants, and the join proceeds
recursively.

### Comparison

The information stored per node in the randomized binary tree is simpler than in a treap (a small integer rather than a
high-precision random number), but it makes a greater number of calls to the random number generator ($O(\log n)$
calls per insertion or deletion rather than one call per insertion) and the insertion procedure is slightly more
complicated due to the need to update the numbers of descendants per node. A minor technical difference is that, in a
treap, there is a small probability of a collision (two keys getting the same priority), and in both cases there will be
statistical differences between a true random number generator and the pseudo-random number generator typically
used on digital computers. However, in any case the differences between the theoretical model of perfect random
choices used to design the algorithm and the capabilities of actual random number generators are vanishingly small.
Although the treap and the randomized binary search tree both have the same random distribution of tree shapes after each update, the history of modifications to the trees performed by these two data structures over a sequence of insertion and deletion operations may be different. For instance, in a treap, if the three numbers 1, 2, and 3 are inserted in the order 1, 3, 2, and then the number 2 is deleted, the remaining two nodes will have the same parent-child relationship that they did prior to the insertion of the middle number. In a randomized binary search tree, the tree after the deletion is equally likely to be either of the two possible trees on its two nodes, independently of what the tree looked like prior to the insertion of the middle number.

References

External links
• Collection of treap references and info (http://faculty.washington.edu/aragon/treaps.html) by Cecilia Aragon
• Treap Applet (http://people.ksp.sk/~kuko/bak/index.html) by Kubo Kovac
• Animated treap (http://www.ibr.cs.tu-bs.de/lehre/ss98/audi/applets/BST/Treap-Example.html)
• Randomized binary search trees (http://www.cs.uiuc.edu/class/sp09/cs473/notes/08-treaps.pdf). Lecture notes from a course by Jeff Erickson at UIUC. Despite the title, this is primarily about treaps and skip lists; randomized binary search trees are mentioned only briefly.
• A high performance key-value store based on treap (http://code.google.com/p/treapdb/) by Junyi Sun
• ActionScript3 implementation of a treap (http://code.google.com/p/as3-commons/source/browse/trunk/as3-commons-collections/src/main/actionscript/org/as3commons/collections/Treap.as)
In computer science, an **AVL tree** is a self-balancing binary search tree, and it was the first such data structure to be invented.[1] In an AVL tree, the heights of the two child subtrees of any node differ by at most one. Lookup, insertion, and deletion all take $O(\log n)$ time in both the average and worst cases, where $n$ is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The AVL tree is named after its two Soviet inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."[2]

The **balance factor** of a node is the height of its left subtree minus the height of its right subtree (sometimes opposite) and a node with balance factor 1, 0, or −1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

AVL trees are often compared with red-black trees because they support the same set of operations and because red-black trees also take $O(\log n)$ time for the basic operations. Because AVL trees are more rigidly balanced, they are faster than red-black trees for lookup intensive applications.[3] However, red-black trees are faster for insertion and removal.

**Operations**

Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the subtrees.

**Lookup**

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. Because of the height-balancing of the tree, a lookup takes $O(\log n)$ time. No special actions need to be taken, and the tree's structure is not modified by lookups. (This is in contrast to splay tree lookups, which do modify their tree's structure.)

If each node additionally records the size of its subtree (including itself and its descendants), then the nodes can be retrieved by index in $O(\log n)$ time as well.
Once a node has been found in a balanced tree, the *next or previous* nodes can be explored in amortized constant time. Some instances of exploring these "nearby" nodes require traversing up to $2\times \log(n)$ links (particularly when moving from the rightmost leaf of the root's left subtree to the leftmost leaf of the root's right subtree). However, exploring all $n$ nodes of the tree in this manner would use each link exactly twice: one traversal to enter the subtree rooted at that node, and another to leave that node's subtree after having explored it. And since by one possible definition of trees there are exactly $n-1$ links in any tree, the amortized cost is found to be $2\times(n-1)/n$, or approximately 2.

**Insertion**

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains $-1$, $0$, or $+1$ then no rotations are necessary. However, if the balance factor becomes $\pm 2$ then the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most one of the following cases needs to be resolved to restore the entire tree to the rules of AVL.

There are four cases which need to be considered, of which two are symmetric to the other two. Let $P$ be the root of the unbalanced subtree, with $R$ and $L$ denoting the right and left children of $P$ respectively.

**Right-Right case and Right-Left case:**
- If the balance factor of $P$ is $-2$ then the right subtree outweights the left subtree of the given node, and the balance factor of the right child ($R$) must be checked. The left rotation with $P$ as the root is necessary.
- If the balance factor of $R$ is $-1$, a *single left rotation* (with $P$ as the root) is needed (Right-Right case).
- If the balance factor of $R$ is $+1$, two different rotations are needed. The first rotation is a *right rotation* with $R$ as the root. The second is a *left rotation* with $P$ as the root (Right-Left case).

**Left-Left case and Left-Right case:**
- If the balance factor of $P$ is $+2$, then the left subtree outweights the right subtree of the given node, and the balance factor of the left child ($L$) must be checked. The right rotation with $P$ as the root is necessary.
- If the balance factor of $L$ is $+1$, a *single right rotation* (with $P$ as the root) is needed (Left-Left case).
- If the balance factor of $L$ is $-1$, two different rotations are needed. The first rotation is a *left rotation* with $L$ as the root. The second is a *right rotation* with $P$ as the root (Left-Right case).
Deletion

If the node is a leaf or has only one child, remove it. Otherwise, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as a replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

In addition to the balancing described above for insertions, if the balance factor for the tree is 2 and that of the left subtree is 0, a right rotation must be performed on P. The mirror of this case is also necessary.

The retracing can stop if the balance factor becomes −1 or +1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retrace continues. If the balance factor becomes −2 or +2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retrace towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is O(log n) for lookup, plus a maximum of O(log n) rotations on the way back to the root, so the operation can be completed in O(log n) time.

Comparison to other structures

Both AVL trees and red-black trees are self-balancing binary search trees, so they are very similar mathematically. The operations to balance the trees are different, but both occur in O(log n) time. The real difference between the two is the limiting height. For a tree of size $n$:

- An AVL tree's height is strictly less than $[4]$:
  \[
  \log_\phi(n+2) - 1 = \frac{\log_2(n + 2)}{\log_2(\phi)} - 1 = \log_\phi(2) \cdot \log_2(n+2) - 1 \approx 1.44 \log_2(n+2) - 1
  \]
  where $\phi$ is the golden ratio.

- A red-black tree's height is at most $2 \log_2(n + 1)$ $[5]$

AVL trees are more rigidly balanced than red-black trees, leading to slower insertion and removal but faster retrieval.
References


[5] Proof of asymptotic bounds

Further reading


External links

- Description from the Dictionary of Algorithms and Data Structures (http:www.nist.gov/dads/HTML/avltree.html)
- C++ Implementation (https:sourceforge.net/projects/standardavl/)
- Python Implementation (http:github.com/pgrafov/python-avl-tree/)
- Single C header file by Ian Piumarta (http:piumarta.com/software/tree/)
- AVL Tree Demonstration (http:www.strille.net/works/media_technology_projects/avl-tree_2001/)
- AVL Tree in examples (http:www.cs.ucf.edu/~reinhard/classes/cop3503/lectures/AVLTrees02.pdf)
- AVL tree applet – all the operations (http:webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html)
- Fast and efficient implementation of AVL Trees (http:github.com/fbuihuu/libtree)
A **red–black tree** is a type of self-balancing binary search tree, a data structure used in computer science, typically to implement associative arrays. The original structure was invented in 1972 by Rudolf Bayer[1] and named "symmetric binary B-tree," but acquired its modern name in a paper in 1978 by Leonidas J. Guibas and Robert Sedgewick.[2] It is complex, but has good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in \( O(\log n) \) time, where \( n \) is the total number of elements in the tree. Put very simply, a red–black tree is a binary search tree that inserts and deletes in such a way that the tree is always reasonably balanced.

### Terminology

A red–black tree is a special type of binary tree, used in computer science to organize pieces of comparable data, such as text fragments or numbers.

The leaf nodes of red–black trees do not contain data. These leaves need not be explicit in computer memory — a null child pointer can encode the fact that this child is a leaf — but it simplifies some algorithms for operating on red–black trees if the leaves really are explicit nodes. To save memory, sometimes a single sentinel node performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node.

Red–black trees, like all binary search trees, allow efficient in-order traversal in the fashion, Left–Root–Right, of their elements. The search-time results from the traversal from root to leaf, and therefore a balanced tree, having the least possible tree height, results in \( O(\log n) \) search time.
Properties

A red–black tree is a binary search tree where each node has a color attribute, the value of which is either red or black. In addition to the ordinary requirements imposed on binary search trees, the following requirements apply to red–black trees:

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted from other definitions. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
3. All leaves are the same color as the root.
4. Both children of every red node are black.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

These constraints enforce a critical property of red–black trees: that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red–black trees to be efficient in the worst-case, unlike ordinary binary search trees.

To see why this is guaranteed, it suffices to consider the effect of properties 4 and 5 together. For a red–black tree T, let B be the number of black nodes in property 5. Therefore the shortest possible path from the root of T to any leaf consists of B black nodes. Longer possible paths may be constructed by inserting red nodes. However, property 4 makes it impossible to insert more than one consecutive red node. Therefore the longest possible path consists of 2B nodes, alternating black and red.

The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black nodes, by property 5, this shows that no path is more than twice as long as any other path.

In many of the presentations of tree data structures, it is possible for a node to have only one child, and leaf nodes contain data. It is possible to present red–black trees in this paradigm, but it changes several of the properties and complicates the algorithms. For this reason, this article uses "null leaves", which contain no data and merely serve to indicate where the tree ends, as shown above. These nodes are often omitted in drawings, resulting in a tree that seems to contradict the above principles, but in fact does not. A consequence of this is that all internal (non-leaf) nodes have two children, although one or both of those children may be null leaves. Property 5 ensures that a red node must have either two black null leaves or two black non-leaves as children. For a black node with one null leaf child and one non-null-leaf child, properties 3, 4 and 5 ensure that the non-null-leaf child must be a red node with two black null leaves as children.

Some explain a red–black tree as a binary search tree whose edges, instead of nodes, are colored in red or black, but this does not make any difference. The color of a node in this article's terminology corresponds to the color of the edge connecting the node to its parent, except that the root node is always black (property 2) whereas the corresponding edge does not exist.
An analogy to B-trees of order 4

A red–black tree is similar in structure to a B-tree of order 4, where each node can contain between 1 to 3 values and (accordingly) between 2 to 4 child pointers. In such B-tree, each node will contain only one value matching the value in a black node of the red–black tree, with an optional value before and/or after it in the same node, both matching an equivalent red node of the red–black tree.

One way to see this equivalence is to "move up" the red nodes in a graphical representation of the red–black tree, so that they align horizontally with their parent black node, by creating together a horizontal cluster. In the B-tree, or in the modified graphical representation of the red–black tree, all leaf nodes are at the same depth.

The red–black tree is then structurally equivalent to a B-tree of order 4, with a minimum fill factor of 33% of values per cluster with a maximum capacity of 3 values.

This B-tree type is still more general than a red–black tree though, as it allows ambiguity in a red–black tree conversion—multiple red–black trees can be produced from an equivalent B-tree of order 4. If a B-tree cluster contains only 1 value, it is the minimum, black, and has two child pointers. If a cluster contains 3 values, then the central value will be black and each value stored on its sides will be red. If the cluster contains two values, however, either one can become the black node in the red–black tree (and the other one will be red).

So the order-4 B-tree does not maintain which of the values contained in each cluster is the root black tree for the whole cluster and the parent of the other values in the same cluster. Despite this, the operations on red–black trees are more economical in time because you don't have to maintain the vector of values. It may be costly if values are stored directly in each node rather than being stored by reference. B-tree nodes, however, are more economical in space because you don't need to store the color attribute for each node. Instead, you have to know which slot in the cluster vector is used. If values are stored by reference, e.g. objects, null references can be used and so the cluster can be represented by a vector containing 3 slots for value pointers plus 4 slots for child references in the tree. In that case, the B-tree can be more compact in memory, improving data locality.

The same analogy can be made with B-trees with larger orders that can be structurally equivalent to a colored binary tree: you just need more colors. Suppose that you add blue, then the blue–red–black tree defined like red–black trees but with the additional constraint that no two successive nodes in the hierarchy will be blue and all blue nodes will be children of a red node, then it becomes equivalent to a B-tree whose clusters will have at most 7 values in the following colors: blue, red, blue, black, blue, red, blue (For each cluster, there will be at most 1 black node, 2 red nodes, and 4 blue nodes).

For moderate volumes of values, insertions and deletions in a colored binary tree are faster compared to B-trees because colored trees don't attempt to maximize the fill factor of each horizontal cluster of nodes (only the minimum fill factor is guaranteed in colored binary trees, limiting the number of splits or junctions of clusters). B-trees will be faster for performing rotations (because rotations will frequently occur within the same cluster rather than with multiple separate nodes in a colored binary tree). However for storing large volumes, B-trees will be much faster as they will be more compact by grouping several children in the same cluster where they can be accessed locally.

All optimizations possible in B-trees to increase the average fill factors of clusters are possible in the equivalent multicolored binary tree. Notably, maximizing the average fill factor in a structurally equivalent B-tree is the same as reducing the total height of the multicolored tree, by increasing the number of non-black nodes. The worst case
occurs when all nodes in a colored binary tree are black, the best case occurs when only a third of them are black (and the other two thirds are red nodes).

Applications and related data structures

Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time. Not only does this make them valuable in time-sensitive applications such as real-time applications, but it makes them valuable building blocks in other data structures which provide worst-case guarantees; for example, many data structures used in computational geometry can be based on red–black trees, and the Completely Fair Scheduler used in current Linux kernels uses red–black trees.

The AVL tree is another structure supporting $O(\log n)$ search, insertion, and removal. It is more rigidly balanced than red–black trees, leading to slower insertion and removal but faster retrieval. This makes it attractive for data structures that may be built once and loaded without reconstruction, such as language dictionaries (or program dictionaries, such as the opcodes of an assembler or interpreter).

Red–black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets which can retain previous versions after mutations. The persistent version of red–black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

For every 2-4 tree, there are corresponding red–black trees with data elements in the same order. The insertion and deletion operations on 2-4 trees are also equivalent to color-flipping and rotations in red–black trees. This makes 2-4 trees an important tool for understanding the logic behind red–black trees, and this is why many introductory algorithm texts introduce 2-4 trees just before red–black trees, even though 2-4 trees are not often used in practice.

In 2008, Sedgewick introduced a simpler version of red–black trees called Left-Leaning Red–Black Trees by eliminating a previously unspecified degree of freedom in the implementation. The LLRB maintains an additional invariant that all red links must lean left except during inserts and deletes. Red–black trees can be made isometric to either 2-3 trees, or 2-4 trees, for any sequence of operations. The 2-4 tree isometry was described in 1978 by Sedgewick. With 2-4 trees, the isometry is resolved by a "color flip," corresponding to a split, in which the red color of two children nodes leaves the children and moves to the parent node. The tango tree, a type of tree optimized for fast searches, usually uses red–black trees as part of its data structure.

Operations

Read-only operations on a red–black tree require no modification from those used for binary search trees, because every red–black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red–black tree. Restoring the red–black properties requires a small number ($O(\log n)$ or amortized $O(1)$) of color changes (which are very quick in practice) and no more than three tree rotations (two for insertion). Although insert and delete operations are complicated, their times remain $O(\log n)$.

Insertion

Insertion begins by adding the node much as binary search tree insertion does and by coloring it red. Whereas in the binary search tree, we always add a leaf, in the red–black tree leaves contain no information, so instead we add a red interior node, with two black leaves, in place of an existing black leaf.

What happens next depends on the color of other nearby nodes. The term uncle node will be used to refer to the sibling of a node's parent, as in human family trees. Note that:

- Property 3 (All leaves are black) always holds.
- Property 4 (Both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.
• Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is threatened only by adding a black node, repainting a red node black (or vice versa), or a rotation.

*Note:* The label N will be used to denote the current node (colored red). At the beginning, this is the new node being inserted, but the entire procedure may also be applied recursively to other nodes (see case 3). P will denote N’s parent node, G will denote N’s grandparent, and U will denote N’s uncle. Note that in between some cases, the roles and labels of the nodes are exchanged, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions.

Each case will be demonstrated with example C code. The uncle and grandparent nodes can be found by these functions:

```c
struct node *grandparent(struct node *n)
{
    if ((n != NULL) && (n->parent != NULL))
        return n->parent->parent;
    else
        return NULL;
}

struct node *uncle(struct node *n)
{
    struct node *g = grandparent(n);
    if (g == NULL)
        return NULL; // No grandparent means no uncle
    if (n->parent == g->left)
        return g->right;
    else
        return g->left;
}
```

**Case 1:** The current node N is at the root of the tree. In this case, it is repainted black to satisfy Property 2 (The root is black). Since this adds one black node to every path at once, Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.

```c
void insert_case1(struct node *n)
{
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(n);
}
```

**Case 2:** The current node’s parent P is black, so Property 4 (Both children of every red node are black) is not invalidated. In this case, the tree is still valid. Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because the current node N has two black leaf children, but because N is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.
void insert_case2(struct node *n)
{
    if (n->parent->color == BLACK)
        return; /* Tree is still valid */
    else
        insert_case3(n);
}

Note: In the following cases it can be assumed that N has a grandparent node G, because its parent P is red, and if it were the root, it would be black. Thus, N also has an uncle node U, although it may be a leaf in cases 4 and 5.

Case 3: If both the parent P and the uncle U are red, then both of them can be repainted black and the grandparent G becomes red (to maintain Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes)). Now, the current red node N has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent G may now violate properties 2 (The root is black) or 4 (Both children of every red node are black) (property 4 possibly being violated since G may have a red parent). To fix this, the entire procedure is recursively performed on G from case 1. Note that this is a tail-recursive call, so it could be rewritten as a loop; since this is the only loop, and any rotations occur after this loop, this proves that a constant number of rotations occur.

void insert_case3(struct node *n)
{
    struct node *u = uncle(n), *g;

    if ((u != NULL) && (u->color == RED)) {
        n->parent->color = BLACK;
        u->color = BLACK;
        g = grandparent(n);
        g->color = RED;
        insert_case1(g);
    } else {
        insert_case4(n);
    }
}

Note: In the remaining cases, it is assumed that the parent node P is the left child of its parent. If it is the right child, left and right should be reversed throughout cases 4 and 5. The code samples take care of this.
**Case 4:** The parent $P$ is red but the uncle $U$ is black; also, the current node $N$ is the right child of $P$, and $P$ in turn is the left child of its parent $G$. In this case, a left rotation that switches the roles of the current node $N$ and its parent $P$ can be performed; then, the former parent node $P$ is dealt with using Case 5 (relabeling $N$ and $P$) because property 4 (Both children of every red node are black) is still violated. The rotation causes some paths (those in the sub-tree labelled "1") to pass through the node $N$ where they did not before. It also causes some paths (those in the sub-tree labelled "3") not to pass through the node $P$ where they did before. However, both of these nodes are red, so Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation. After this case has been completed, Property 4 (both children of every red node are black) is still violated, but now we can resolve this by continuing to Case 5.

```c
void insert_case4(struct node *n) {
    struct node *g = grandparent(n);

    if ((n == n->parent->right) && (n->parent == g->left)) {
        rotate_left(n->parent);
        n = n->left;
    } else if ((n == n->parent->left) && (n->parent == g->right)) {
        rotate_right(n->parent);
        n = n->right;
    }
    insert_case5(n);
}
```

**Case 5:** The parent $P$ is red but the uncle $U$ is black, the current node $N$ is the left child of $P$, and $P$ is the left child of its parent $G$. In this case, a right rotation on $G$ is performed; the result is a tree where the former parent $P$ is now the parent of both the current node $N$ and the former grandparent $G$. $G$ is known to be black, since its former child $P$ could not have been red otherwise (without violating Property 4). Then, the colors of $P$ and $G$ are switched, and the resulting tree satisfies Property 4 (Both children of every red node are black). Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) also remains satisfied, since all paths that went through any of these three nodes went through $G$ before, and now they all go through $P$. In each case, this is the only black node of the three.

```c
void insert_case5(struct node *n) {
    struct node *g = grandparent(n);

    n->parent->color = BLACK;
    g->color = RED;
}
```
Redblack tree

if ((n == n->parent->left) && (n->parent == g->left)) {
    rotate_right(g);
} else if ((n == n->parent->right) && (n->parent == g->right)) {
    rotate_left(g);
}

Note that inserting is actually in-place, since all the calls above use tail recursion.

Removal

In a regular binary search tree when deleting a node with two non-leaf children, we find either the maximum element in its left subtree (which is the in-order predecessor) or the minimum element in its right subtree (which is the in-order successor) and move its value into the node being deleted (as shown here). We then delete the node we copied the value from, which must have fewer than two non-leaf children. (Non-leaf children, rather than all children, are specified here because unlike normal binary search trees, red–black trees have leaf nodes anywhere they can have them, so that all nodes are either internal nodes with two children or leaf nodes with, by definition, zero children. In effect, internal nodes having two leaf children in a red–black tree are like the leaf nodes in a regular binary search tree.) Because merely copying a value does not violate any red–black properties, this reduces to the problem of deleting a node with at most one non-leaf child. Once we have solved that problem, the solution applies equally to the case where the node we originally want to delete has at most one non-leaf child as to the case just considered where it has two non-leaf children.

Therefore, for the remainder of this discussion we address the deletion of a node with at most one non-leaf child. We use the label M to denote the node to be deleted; C will denote a selected child of M, which we will also call "its child". If M does have a non-leaf child, call that its child, C; otherwise, choose either leaf as its child, C.

If M is a red node, we simply replace it with its child C, which must be black by definition. (This can only occur when M has two leaf children, because if the red node M had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would had been an invalid red–black tree by violation of Property 5.) All paths through the deleted node will simply pass through one less red node, and both the deleted node's parent and child must be black, so Property 3 ("All leaves are black") and Property 4 ("Both children of every red node are black") still hold.

Another simple case is when M is black and C is red. Simply removing a black node could break Properties 4 ("Both children of every red node are black") and 5 ("All paths from any given node to its leaf nodes contain the same number of black nodes"), but if we repaint C black, both of these properties are preserved.

The complex case is when both M and C are black. (This can only occur when deleting a black node which has two leaf children, because if the black node M had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would had been an invalid red–black tree by violation of Property 5.) We begin by replacing M with its child C. We will call (or label—that is, relabel) this child (in its new position) N, and its sibling (its new parent's other child) S. (S was previously the sibling of M.) In the diagrams below, we will also use P for N's new parent (M's old parent), S_L for S's left child, and S_R for S's right child (S cannot be a leaf because if N is black, which we presumed, then P's one subtree which includes N counts two black-height and thus P's other subtree which includes S must also count two black-height, which cannot be the case if S is a leaf node).

Note: In between some cases, we exchange the roles and labels of the nodes, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions. White represents an unknown color (either red or black).
We will find the sibling using this function:

```c
struct node *sibling(struct node *n)
{
    if (n == n->parent->left)
        return n->parent->right;
    else
        return n->parent->left;
}
```

*Note:* In order that the tree remains well-defined, we need that every null leaf remains a leaf after all transformations (that it will not have any children). If the node we are deleting has a non-leaf (non-null) child \( N \), it is easy to see that the property is satisfied. If, on the other hand, \( N \) would be a null leaf, it can be verified from the diagrams (or code) for all the cases that the property is satisfied as well.

We can perform the steps outlined above with the following code, where the function `replace_node` substitutes `child` into `n`'s place in the tree. For convenience, code in this section will assume that null leaves are represented by actual node objects rather than NULL (the code in the `Insertion` section works with either representation).

```c
void delete_one_child(struct node *n)
{
    /*
     * Precondition: n has at most one non-null child.
     */
    struct node *child = is_leaf(n->right) ? n->left : n->right;

    replace_node(n, child);
    if (n->color == BLACK) {
        if (child->color == RED)
            child->color = BLACK;
        else
            delete_case1(child);
    }
    free(n);
}
```

*Note:* If \( N \) is a null leaf and we do not want to represent null leaves as actual node objects, we can modify the algorithm by first calling `delete_case1()` on its parent (the node that we delete, \( n \) in the code above) and deleting it afterwards. We can do this because the parent is black, so it behaves in the same way as a null leaf (and is sometimes called a 'phantom' leaf). And we can safely delete it at the end as \( n \) will remain a leaf after all operations, as shown above.

If both \( N \) and its original parent are black, then deleting this original parent causes paths which proceed through \( N \) to have one fewer black node than paths that do not. As this violates Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes), the tree must be rebalanced. There are several cases to consider:

**Case 1:** \( N \) is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved.

```c
void delete_case1(struct node *n)
{
    if (n->parent != NULL)
```
Note: In cases 2, 5, and 6, we assume N is the left child of its parent P. If it is the right child, left and right should be reversed throughout these three cases. Again, the code examples take both cases into account.

Case 2: S is red. In this case we reverse the colors of P and S, and then rotate left at P, turning S into N's grandparent. Note that P has to be black as it had a red child. Although all paths still have the same number of black nodes, now N has a black sibling and a red parent, so we can proceed to step 4, 5, or 6. (Its new sibling is black because it was once the child of the red S.) In later cases, we will relabel N's new sibling as S.

```c
void delete_case2(struct node *n) {
    struct node *s = sibling(n);
    if (s->color == RED) {
        n->parent->color = RED;
        s->color = BLACK;
        if (n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent);
    }
    delete_case3(n);
}
```

Case 3: P, S, and S's children are black. In this case, we simply repaint S red. The result is that all paths passing through S, which are precisely those paths not passing through N, have one less black node. Because deleting N's original parent made all paths passing through N have one less black node, this evens things up. However, all paths through P now have one fewer black node than paths that do not pass through P, so Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is still violated. To correct this, we perform the rebalancing procedure on P, starting at case 1.

```c
void delete_case3(struct node *n) {
    struct node *s = sibling(n);
    if ((n->parent->color == BLACK) &&
```
Redblack tree

(s->color == BLACK) &&
(s->left->color == BLACK) &&
(s->right->color == BLACK)) {
  s->color = RED;
  delete_case1(n->parent);
} else
  delete_case4(n);

Case 4: S and S's children are black, but P is red. In this case, we simply exchange the colors of S and P. This does not affect the number of black nodes on paths going through S, but it does add one to the number of black nodes on paths going through N, making up for the deleted black node on those paths.

```c
void delete_case4(struct node *n)
{
  struct node *s = sibling(n);

  if ((n->parent->color == RED) &&
      (s->color == BLACK) &&
      (s->left->color == BLACK) &&
      (s->right->color == BLACK)) {
    s->color = RED;
    n->parent->color = BLACK;
  } else
    delete_case5(n);
}
```

Case 5: S is black, S's left child is red, S's right child is black, and N is the left child of its parent. In this case we rotate right at S, so that S's left child becomes S's parent and N's new sibling. We then exchange the colors of S and its new parent. All paths still have the same number of black nodes, but now N has a black sibling whose right child is red, so we fall into case 6. Neither N nor its parent are affected by this transformation. (Again, for case 6, we relabel N's new sibling as S.)

```c
void delete_case5(struct node *n)
{
```

```c
```
struct node *s = sibling(n);

if (s->color == BLACK) { /* this if statement is trivial, due to Case 2 (even though Case two changed the sibling to a sibling's child, the sibling's child can't be red, since no red parent can have a red child). */
/* the following statements just force the red to be on the left of the left of the parent, or right of the right, so case six will rotate correctly. */

if ((n == n->parent->left) &&
    (s->right->color == BLACK) &&
    (s->left->color == RED)) { /* this last test is trivial too due to cases 2-4. */
    s->color = RED;
    s->left->color = BLACK;
    rotate_right(s);
} else if ((n == n->parent->right) &&
           (s->left->color == BLACK) &&
           (s->right->color == RED)) { /* this last test is trivial too due to cases 2-4. */
    s->color = RED;
    s->right->color = BLACK;
    rotate_left(s);
}
}
delete_case6(n);

---

**Case 6:** S is black, S's right child is red, and N is the left child of its parent P. In this case we rotate left at P, so that S becomes the parent of P and S's right child. We then exchange the colors of P and S, and make S's right child black. The subtree still has the same color at its root, so Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) are not violated. However, N now has one additional black ancestor: either P has become black, or it was black and S was added as a black grandparent. Thus, the paths passing through N pass through one additional black node.
Meanwhile, if a path does not go through \(N\), then there are two possibilities:

- It goes through \(N\)'s new sibling. Then, it must go through \(S\) and \(P\), both formerly and currently, as they have only exchanged colors and places. Thus the path contains the same number of black nodes.
- It goes through \(N\)'s new uncle, \(S\)'s right child. Then, it formerly went through \(S\), \(S\)'s parent, and \(S\)'s right child (which was red), but now only goes through \(S\), which has assumed the color of its former parent, and \(S\)'s right child, which has changed from red to black (assuming \(S\)'s color: black). The net effect is that this path goes through the same number of black nodes.

Either way, the number of black nodes on these paths does not change. Thus, we have restored Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes). The white node in the diagram can be either red or black, but must refer to the same color both before and after the transformation.

```c
void delete_case6(struct node *n) {
    struct node *s = sibling(n);
    s->color = n->parent->color;
    n->parent->color = BLACK;

    if (n == n->parent->left) {
        s->right->color = BLACK;
        rotate_left(n->parent);
    } else {
        s->left->color = BLACK;
        rotate_right(n->parent);
    }
}
```

Again, the function calls all use tail recursion, so the algorithm is in-place. In the algorithm above, all cases are chained in order, except in delete case 3 where it can recurse to case 1 back to the parent node: this is the only case where an in-place implementation will effectively loop (after only one rotation in case 3).

Additionally, no tail recursion ever occurs on a child node, so the tail recursion loop can only move from a child back to its successive ancestors. No more than \(O(\log n)\) loops back to case 1 will occur (where \(n\) is the total number of nodes in the tree before deletion). If a rotation occurs in case 2 (which is the only possibility of rotation within the loop of cases 1–3), then the parent of the node \(N\) becomes red after the rotation and we will exit the loop. Therefore at most one rotation will occur within this loop. Since no more than two additional rotations will occur after exiting the loop, at most three rotations occur in total.

**Proof of asymptotic bounds**

A red black tree which contains \(n\) internal nodes has a height of \(O(\log(n))\).

Definitions:

- \(h(v)\) = height of subtree rooted at node \(v\)
- \(bh(v)\) = the number of black nodes (not counting \(v\) if it is black) from \(v\) to any leaf in the subtree (called the black-height).

**Lemma:** A subtree rooted at node \(v\) has at least \(2^{bh(v)} - 1\) internal nodes.

Proof of Lemma (by induction height):

**Basis:** \(h(v) = 0\)

If \(v\) has a height of zero then it must be null, therefore \(bh(v) = 0\). So:

\[
2^{bh(v)} - 1 = 2^0 - 1 = 1 - 1 = 0
\]
Inductive Step: \( v \) such that \( h(v) = k \), has at least \( 2^{bh(v)} - 1 \) internal nodes implies that \( v' \) such that \( h(v') = k+1 \) has at least \( 2^{bh(v')} - 1 \) internal nodes.

Since \( v' \) has \( h(v') > 0 \) it is an internal node. As such it has two children each of which have a black-height of either \( bh(v') \) or \( bh(v') - 1 \) (depending on whether the child is red or black, respectively). By the inductive hypothesis each child has at least \( \frac{1}{2} \) internal nodes, so \( v' \) has at least:

\[
2^{bh(v') - 1} - 1 + 2^{bh(v') - 1} - 1 + 1 = 2^{bh(v')} - 1
\]

internal nodes.

Using this lemma we can now show that the height of the tree is logarithmic. Since at least half of the nodes on any path from the root to a leaf are black (property 4 of a red black tree), the black-height of the root is at least \( h(root)/2 \).

By the lemma we get:

\[
2 \cdot \frac{h(root)}{2} - 1 \leq \log_2 (n + 1) \leq \frac{h(root)}{2} \leq 2 \log_2 (n + 1).
\]

Therefore the height of the root is \( O(\log(n)) \).

**Insertion complexity**

In the tree code there is only one loop where the node of the root of the red–black property that we wish to restore, \( x \), can be moved up the tree by one level at each iteration.

Since the original height of the tree is \( O(\log n) \), there are \( O(\log n) \) iterations. So overall the insert routine has \( O(\log n) \) complexity.

**Parallel algorithms**

Parallel algorithms for constructing red–black trees from sorted lists of items can run in constant time or \( O(\log\log n) \) time, depending on the computer model, if the number of processors available is proportional to the number of items. Fast search, insertion, and deletion parallel algorithms are also known.\(^5\)

**Notes**


**References**

- San Diego State University: CS 660: Red–Black tree notes (http://www.eli.sdsu.edu/courses/fall95/csx660/notes/RedBlackTree/RedBlack.html#RTFToC2), by Roger Whitney

External links

• In the C++ Standard Template Library, the containers std::set<value> and std::map<Key,Value> are typically based on red–black trees
• Tutorial and code for top-down Red–Black Trees (http://eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx)
• C code for Red–Black Trees (http://github.com/fbuihuu/libtree)
• Red–Black Tree C Code (http://www.mit.edu/~emin/source_code/red_black_tree/index.html)
• Lightweight Java implementation of Persistent Red–Black Trees (http://wiki.edinburghhacklab.com/PersistentRedBlackTreeSet)
• VBScript implementation of stack, queue, deque, and Red–Black Tree (http://www.ludvikjerabek.com/downloads.html)
• Red–Black Tree Demonstration (http://www.ece.uc.edu/~franco/C321/html/RedBlack/redblack.html)
• Red–Black Tree PHP5 Code (http://code.google.com/p/redblacktreephp/source/browse/#svn/trunk)
• In Java a freely available red black tree implementation is that of apache commons (http://commons.apache.org/collections/api-release/org/apache/commons/collections/bidimap/TreeBiMap.html)
• Java's TreeSet class internally stores its elements in a red black tree: http://java.sun.com/docs/books/tutorial/collections/interfaces/set.html
• Left Leaning Red Black Trees (http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf)
• Left Leaning Red Black Trees Slides (http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf)
• PPT slides demonstration of manipulating red black trees to facilitate teaching (http://employees.oneonta.edu/zhangs/PowerPointplatform/)
5 (http://www.boyet.com/Articles/RedBlack5.html), a C# Article series by Julian M. Bucknall.
• Binary Search Tree Insertion Visualization (https://www.youtube.com/watch?v=_VbTnLV8plU) on YouTube – Visualization of random and pre-sorted data insertions, in elementary binary search trees, and left-leaning red–black trees
Scapegoat tree

In computer science, a scapegoat tree is a self-balancing binary search tree, discovered by Arne Anderson and again by Igal Galperin and Ronald L. Rivest. It provides worst-case $O(\log n)$ lookup time, and $O(\log n)$ amortized insertion and deletion time.

Unlike most other self-balancing binary search trees that provide worst case $O(\log n)$ lookup time, scapegoat trees have no additional per-node memory overhead compared to a regular binary search tree: a node stores only a key and two pointers to the child nodes. This makes scapegoat trees easier to implement and, due to data structure alignment, can reduce node overhead by up to one-third.

Theory

A binary search tree is said to be weight balanced if half the nodes are on the left of the root, and half on the right. An $\alpha$-weight-balanced is therefore defined as meeting the following conditions:

\[
\begin{align*}
\text{size(left)} & \leq \alpha \times \text{size(node)} \\
\text{size(right)} & \leq \alpha \times \text{size(node)}
\end{align*}
\]

Where size can be defined recursively as:

```plaintext
function size(node)
    if node = nil 
        return 0 
    else 
        return size(node->left) + size(node->right) + 1 
    end
end
```

An $\alpha$ of 1 therefore would describe a linked list as balanced, whereas an $\alpha$ of 0.5 would only match almost complete binary trees.

A binary search tree that is $\alpha$-weight-balanced must also be $\alpha$-height-balanced, that is

\[
\text{height(tree)} \leq \log_{\frac{1}{\alpha}}(\text{NodeCount})
\]

Scapegoat trees are not guaranteed to keep $\alpha$-weight-balance at all times, but are always loosely $\alpha$-height-balance in that

\[
\text{height(scapegoat tree)} \leq \log_{\frac{1}{\alpha}}(\text{NodeCount}) + 1
\]

This makes scapegoat trees similar to red-black trees in that they both have restrictions on their height. They differ greatly though in their implementations of determining where the rotations (or in the case of scapegoat trees, rebalances) take place. Whereas red-black trees store additional 'color' information in each node to determine the location, scapegoat trees find a scapegoat which isn't $\alpha$-weight-balanced to perform the rebalance operation on. This is loosely similar to AVL trees, in that the actual rotations depend on 'balances' of nodes, but the means of determining the balance differs greatly. Since AVL trees check the balance value on every insertion/deletion, it is typically stored in each node; scapegoat trees are able to calculate it only as needed, which is only when a scapegoat needs to be found.

Unlike most other self-balancing search trees, scapegoat trees are entirely flexible as to their balancing. They support any $\alpha$ such that $0.5 < \alpha < 1$. A high $\alpha$ value results in fewer balances, making insertion quicker but lookups and deletions slower, and vice versa for a low $\alpha$. Therefore in practical applications, an $\alpha$ can be chosen depending on how frequently these actions should be performed.
Operations

Insertion

Insertion is implemented very similarly to an unbalanced binary search tree, however with a few significant changes. When finding the insertion point, the depth of the new node must also be recorded. This is implemented via a simple counter that gets incremented during each iteration of the lookup, effectively counting the number of edges between the root and the inserted node. If this node violates the $\alpha$-height-balance property (defined above), a rebalance is required.

To rebalance, an entire subtree rooted at a scapegoat undergoes a balancing operation. The scapegoat is defined as being an ancestor of the inserted node which isn’t $\alpha$-weight-balanced. There will always be at least one such ancestor. Rebalancing any of them will restore the $\alpha$-height-balanced property.

One way of finding a scapegoat, is to climb from the new node back up to the root and select the first node that isn’t $\alpha$-weight-balanced.

Climbing back up to the root requires $O(\log n)$ storage space, usually allocated on the stack, or parent pointers. This can actually be avoided by pointing each child at its parent as you go down, and repairing on the walk back up.

To determine whether a potential node is a viable scapegoat, we need to check its $\alpha$-weight-balanced property. To do this we can go back to the definition:

\[
\text{size(left)} \leq \alpha \times \text{size(node)} \\
\text{size(right)} \leq \alpha \times \text{size(node)}
\]

However a large optimisation can be made by realising that we already know two of the three sizes, leaving only the third having to be calculated.

Consider the following example to demonstrate this. Assuming that we're climbing back up to the root:

\[
\text{size(parent)} = \text{size(node)} + \text{size(sibling)} + 1
\]

But as:

\[
\text{size(inserted node)} = 1.
\]

The case is trivialized down to:

\[
\text{size}[x + 1] = \text{size}[x] + \text{size(sibling)} + 1
\]

Where $x = \text{this node}$, $x + 1 = \text{parent}$ and $\text{size(sibling)}$ is the only function call actually required.

Once the scapegoat is found, the subtree rooted at the scapegoat is completely rebuilt to be perfectly balanced.\[^3\] This can be done in $O(n)$ time by traversing the nodes of the subtree to find their values in sorted order and recursively choosing the median as the root of the subtree.

As rebalance operations take $O(n)$ time (dependent on the number of nodes of the subtree), insertion has a worst case performance of $O(n)$ time. However, because these worst-case scenarios are spread out, insertion takes $O(\log n)$ amortized time.
Sketch of proof for cost of insertion

Define the Imbalance of a node \( v \) to be the absolute value of the difference in size between its left node and right node minus 1, or 0, whichever is greater. In other words:

\[
I(v) = \max(|left(v) - right(v)| - 1, 0)
\]

Immediately after rebuilding a subtree rooted at \( v \), \( I(v) = 0 \).

**Lemma:** Immediately before rebuilding the subtree rooted at \( v \),

\[
I(v) = \Omega(|v|)
\]

( \( \Omega \) is Big O Notation.)

**Proof of lemma:**

Let \( v_0 \) be the root of a subtree immediately after rebuilding. \( h(v_0) = \log(|v_0| + 1) \). If there are \( \Omega(|v_0|) \) degenerate insertions (that is, where each inserted node increases the height by 1), then

\[
I(v) = \Omega(|v_0|).
\]

\( h(v) = h(v_0) + \Omega(|v_0|) \) and

\[
\log(|v|) < \log(|v_0| + 1) + 1.
\]

Since \( I(v) = \Omega(|v|) \) before rebuilding, there were \( \Omega(|v|) \) insertions into the subtree rooted at \( v \) that did not result in rebuilding. Each of these insertions can be performed in \( O(log |v|) \) time. The final insertion that causes rebuilding costs \( O(|v|) \). Using aggregate analysis it becomes clear that the amortized cost of an insertion is

\[
O(|v|)O(log n) + O(|v|) = O(log n)
\]

Deletion

Scapegoat trees are unusual in that deletion is easier than insertion. To enable deletion, scapegoat trees need to store an additional value with the tree data structure. This property, which we will call MaxNodeCount simply represents the highest achieved NodeCount. It is set to NodeCount whenever the entire tree is rebalanced, and after insertion is set to \( \max(\text{MaxNodeCount}, \text{NodeCount}) \).

To perform a deletion, we simply remove the node as you would in a simple binary search tree, but if

```plaintext
NodeCount <= MaxNodeCount / 2
```

then we rebalance the entire tree about the root, remembering to set MaxNodeCount to NodeCount.

This gives deletion its worst case performance of \( O(n) \) time, however it is amortized to \( O(\log n) \) average time.

Sketch of proof for cost of deletion

Suppose the scapegoat tree has \( n \) elements and has just been rebuilt (in other words, it is a complete binary tree). At most \( n/2 - 1 \) deletions can be performed before the tree must be rebuilt. Each of these deletions take \( O(\log n) \) time (the amount of time to search for the element and flag it as deleted). The \( n/2 \) deletion causes the tree to be rebuilt and takes \( O(\log n) + O(n) \) (or just \( O(n) \)) time. Using aggregate analysis it becomes clear that the amortized cost of a deletion is \( O(\log n) \):

\[
\sum_{i=1}^{n/2} O(\log n) + O(n) = \frac{n}{2} O(\log n) + O(n) = O(\log n)
\]
Lookup

Lookup is not modified from a standard binary search tree, and has a worst-case time of $O(\log n)$. This is in contrast to splay trees which have a worst-case time of $O(n)$. The reduced node memory overhead compared to other self-balancing binary search trees can further improve locality of reference and caching.

References


External links

• Scapegoat Tree Applet (http://people.ksp.sk/~kuko/bak/index.html) by Kubo Kovac
• Scapegoat Trees: Galperin and Rivest’s paper describing scapegoat trees (http://cg.scs.carleton.ca/~morin/teaching/5408/refts/gr93.pdf)
Splay tree

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in $O(\log n)$ amortized time. For many sequences of nonrandom operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985.[1]

All normal operations on a binary search tree are combined with one basic operation, called splaying. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

Advantages

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. The worst-case height - though unlikely - is $O(n)$, with the average being $O(\log n)$. Having frequently-used nodes near the root is an advantage for nearly all practical applications (also see Locality of reference), and is particularly useful for implementing caches and garbage collection algorithms.

Advantages include:

- Simple implementation—simpler than other self-balancing binary search trees, such as red-black trees or AVL trees.
- Comparable performance—average-case performance is as efficient as other trees.
- Small memory footprint—splay trees do not need to store any bookkeeping data.
- Possibility of creating a persistent data structure version of splay trees—which allows access to both the previous and new versions after an update. This can be useful in functional programming, and requires amortized $O(\log n)$ space per update.
- Working well with nodes containing identical keys—contrary to other types of self-balancing trees. Even with identical keys, performance remains amortized $O(\log n)$. All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the leftmost or rightmost node of a given key.
**Disadvantages**

Perhaps the most significant disadvantage of splay trees is that the height of a splay tree can be linear. For example, this will be the case after accessing all $n$ elements in non-decreasing order. Since the height of a tree corresponds to the worst-case access time, this means that the actual cost of an operation can be slow. However, the amortized access cost of this worst case is logarithmic, $O(\log n)$. Also, the expected access cost can be reduced to $O(\log n)$ by using a randomized variant\[^2\].

A splay tree can be worse than a static tree by at most a constant factor. Splay trees can change even when they are accessed in a 'read-only' manner (i.e. by find operations). This complicates the use of such splay trees in a multi-threaded environment. Specifically, extra management is needed if multiple threads are allowed to perform find operations concurrently.

**Operations**

**Splaying**

When a node $x$ is accessed, a splay operation is performed on $x$ to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves $x$ closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step depends on three factors:
- Whether $x$ is the left or right child of its parent node, $p$,
- whether $p$ is the root or not, and if not
- whether $p$ is the left or right child of its parent, $g$ (the grandparent of $x$).

It is important to remember to set $gg$ (the great-grandparent of $x$) to now point to $x$ after any splay operation. If $gg$ is null, then $x$ obviously is now the root and must be updated as such.

The three types of splay steps are:

**Zig Step:** This step is done when $p$ is the root. The tree is rotated on the edge between $x$ and $p$. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when $x$ has odd depth at the beginning of the operation.

![Zig Step Diagram]

**Zig-zig Step:** This step is done when $p$ is not the root and $x$ and $p$ are either both right children or are both left children. The picture below shows the case where $x$ and $p$ are both left children. The tree is rotated on the edge joining $p$ with its parent $g$, then rotated on the edge joining $x$ with $p$. Note that zig-zig steps are the only thing that differentiate splay trees from the rotate to root method introduced by Allen and Munro\[^3\] prior to the introduction of
splay trees.

Zig-zag Step: This step is done when $p$ is not the root and $x$ is a right child and $p$ is a left child or vice versa. The tree is rotated on the edge between $x$ and $p$, then rotated on the edge between $x$ and its new parent $g$.

Insertion
To insert a node $x$ into a splay tree:
1. First insert the node as with a normal binary search tree.
2. Then splay the newly inserted node $x$ to the top of the tree.

Deletion
To delete a node $x$, we use the same method as with a binary search tree: if $x$ has two children, we swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right sub tree (its in-order successor). Then we remove that node instead. In this way, deletion is reduced to the problem of removing a node with 0 or 1 children.

Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree. OR The node to be deleted is first splayed, i.e. brought to the root of the tree and then deleted. This leaves the tree with two sub trees. The maximum element of the left sub tree (: METHOD 1), or minimum of the right sub tree (: METHOD 2) is then splayed to the root. The right sub tree is made the right child of the resultant left sub tree (for METHOD 1). The root of left sub tree is the root of melded tree.
**Code in C language**

**Splay operation in BST**

Here x is the node on which the splay operation is performed and root is the root node of the tree.

```c
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *parent;
    struct node *left;
    struct node *right;
};

int data_print(struct node *x);

struct node *rightrotation(struct node *p, struct node *root);

struct node *leftrotation(struct node *p, struct node *root);

void splay(struct node *x, struct node *root);

struct node *insert(struct node *p, int value);

struct node *inorder(struct node *p);

struct node *delete(struct node *p, int value);

struct node *sucessor(struct node *x);

struct node *lookup(struct node *p, int value);

void splay(struct node *x, struct node *root)
{
    struct node *p, *g;

    /*check if node x is the root node*/
    if(x==root)
        return;

    /*Performs Zig step*/
    else if(x->parent==root)
    {
        if(x==x->parent->left)
            root=rightrotation(root,root);
        else
            root=leftrotation(root,root);
    }
    else
    {
        p=x->parent; /*now points to parent of x*/
        g=p->parent; /*now points to parent of x's parent*/
        /*Performs the Zig-zig step when x is left and x's parent is left*/
        if(x==p->left&&p==g->left)
            root=rightrotation(g,root);
    }
}
```
Splay tree

```c
struct node *rightrotation(struct node *p, struct node *root)
{
    struct node *x;
    x = p->left;
    p->left = x->right;
    if (x->right!=NULL) x->right->parent = p;
    x->right = p;
    if (p->parent!=NULL)
        if (p==p->parent->right) p->parent->right=x;
        else
            p->parent->left=x;
    x->parent = p->parent;
    p->parent = x;
    if (p==root)
        return x;
    else
        return root;
}
struct node *leftrotation(struct node *p, struct node *root)
{
    struct node *x;
```
x = p->right;
p->right = x->left;
if (x->left!=NULL) x->left->parent = p;
x->left = p;
if (p->parent!=NULL)
    if (p==p->parent->left) p->parent->left=x;
    else
        p->parent->right=x;
x->parent = p->parent;
p->parent = x;
if(p==root)
    return x;
else
    return root;
}

struct node *insert(struct node *p, int value)
{
    struct node *temp1,*temp2,*par,*x;
    if(p == NULL)
    {
        p=(struct node *)malloc(sizeof(struct node));
        if(p != NULL)
        {
            p->data = value;
            p->parent = NULL;
            p->left = NULL;
            p->right = NULL;
        }
    }
    else
    {
        printf("No memory is allocated\n");
        exit(0);
    }
    return(p);
}
else
{
    temp2 = p;
    while(temp2 != NULL)
    {
        temp1 = temp2;
        if(temp2->data > value)
            temp2 = temp2->left;
        else if(temp2->data < value)
            temp2 = temp2->right;
        else
            if(temp2->data == value)
```
return temp2;
}
if (temp1->data > value)
{
    par = temp1; // temp1 having the parent address, so that's it
    temp1->left = (struct node *)malloc(sizeof(struct node));
    temp1 = temp1->left;
    if (temp1 != NULL)
    {
        temp1->data = value;
        temp1->parent = par; // store the parent address.
        temp1->left = NULL;
        temp1->right = NULL;
    }
    else
    {
        printf("No memory is allocated\n");
        exit(0);
    }
}
else
{
    par = temp1; // temp1 having the parent node address.
    temp1->right = (struct node *)malloc(sizeof(struct node));
    temp1 = temp1->right;
    if (temp1 != NULL)
    {
        temp1->data = value;
        temp1->parent = par; // store the parent address
        temp1->left = NULL;
        temp1->right = NULL;
    }
    else
    {
        printf("No memory is allocated\n");
        exit(0);
    }
}
return (temp1);
```
```c
struct node *inorder(struct node *p)
{
    if (p != NULL)
    {
        inorder(p->left);
        printf("CURRENT %d
", p->data);
        printf("LEFT %d
", data_print(p->left));
        printf("PARENT %d
", data_print(p->parent));
        printf("RIGHT %d
", data_print(p->right));
        inorder(p->right);
    }
}

struct node *delete(struct node *p, int value)
{
    struct node *x,*y,*p1;
    struct node *root;
    struct node *s;
    root = p;
    x = lookup(p, value);
    if(x->data == value)
    {
        //if the deleted element is leaf
        if((x->left == NULL) && (x->right == NULL))
        {
            y = x->parent;
            if(x == (x->parent->right))
                y->right = NULL;
            else
                y->left = NULL;
            free(x);
        }
        //if deleted element having left child only
        else if((x->left != NULL) && (x->right == NULL))
        {
            if(x == (x->parent->left))
            {
                y = x->parent;
                x->left->parent = y;
                y->left = x->left;
                free(x);
            }
            else
            {
                y = x->parent;
                x->left->parent = y;
                y->right = x->left;
                free(x);
            }
        }
    }
    return root;
}
```
} //if deleted element having right child only
else if((x->left == NULL) && (x->right != NULL))
{
    if(x == (x->parent->left))
    {
        y = x->parent;
        x->right->parent = y;
        y->left = x->right;
        free(x);
    }
    else
    {
        y = x->parent;
        x->right->parent = y;
        y->right = x->right;
        free(x);
    }
}
} //if the deleted element having two child
else if((x->left != NULL) && (x->right != NULL))
{
    if(x == (x->parent->left))
    {
        s = sucessor(x);
        if(s != x->right)
        {
            y = s->parent;
            if(s->right != NULL)
            {
                s->right->parent = y;
                y->left = s->right;
            }
            else
            y->left = NULL;
            s->parent = x->parent;
            x->right->parent = s;
            x->left->parent = s;
            s->right = x->right;
            s->left = x->left;
            x->parent->left = s;
        }
    }
    else
    {
        y = s;
        s->parent = x->parent;
        x->left->parent = s;
        s->left = x->left;
    }
}
```
x->parent->left = s;
}
free(x);
}
else if(x == (x->parent->right))
{
    s = sucessor(x);
    if(s != x->right)
    {
        y = s->parent;
        if(s->right != NULL)
        {
            s->right->parent = y;
            y->left = s->right;
        }
        else
        y->left = NULL;
        s->parent = x->parent;
        x->right->parent = s;
        s->right = x->right;
        s->left = x->left;
        x->parent->right = s;
    }
    else
    {
        y = s;
        s->parent = x->parent;
        x->left->parent = s;
        s->left = x->left;
        x->parent->right = s;
    }
    free(x);
}
}
splay(y,root);
}
else
{
    splay(x,root);
}

struct node *sucessor(struct node *x)
{
    struct node *temp,*temp2;
    temp=temp2=x->right;
    while(temp != NULL)
```
{  
    temp2 = temp;  
    temp = temp->left;  
}  
return temp2; 
}  

//p is a root element of the tree  
struct node *lookup(struct node *p, int value)  
{  
    struct node *temp1,*temp2;  
    if(p != NULL)  
    {  
        temp1 = p;  
        while(temp1 != NULL)  
        {  
            temp2 = temp1;  
            if(temp1->data > value)  
                temp1 = temp1->left;  
            else if(temp1->data < value)  
                temp1 = temp1->right;  
            else  
                return temp1;  
        }  
        return temp2;  
    }  
    else  
    {  
        printf("NO element in the tree\n");  
        exit(0);  
    }  
}  
struct node *search(struct node *p, int value)  
{  
    struct node *x,*root;  
    root = p;  
    x = lookup(p,value);  
    if(x->data == value)  
    {  
        printf("Inside search if\n");  
        splay(x,root);  
    }  
    else  
    {  
        printf("Inside search else\n");  
        splay(x,root);  
    }  
}
main()
{
    struct node *root;    // the root element
    struct node *x;      // x is which element will come to root.
    int i;
    root = NULL;
    int choice = 0;
    int ele;
    while(1)
    {
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Display\n");
        printf("Enter your choice:");
        scanf("%d", &choice);
        if(choice==5)
            exit(0);
        switch(choice)
        {
            case 1:
                printf("Enter the element to be inserted:");
                scanf("%d", &ele);
                x = insert(root, ele);
                if(root != NULL)
                {
                    splay(x, root);
                }
                root = x;
                break;
            case 2:
                if(root == NULL)
                {
                    printf("Empty tree...");
                    continue;
                }
                printf("Enter the element to be delete:");
                scanf("%d", &ele);
                root = delete(root, ele);
                break;
            case 3:
                printf("Enter the element to be search\n");
                scanf("%d", &ele);
                x = lookup(root, ele);
                splay(x, root);
                root = x;
            case 4:
                printf("Display:");
                break;
        }
    }
}
Splay tree

```c
break;

case 4:
    printf("The elements are\n");
    inorder(root);
    break;

default:
    printf("Wrong choice\n");
    break;

int data_print(struct node *x)
{
    if ( x==NULL )
        return 0;
    else
        return x->data;
}
```

Analysis

A simple amortized analysis of static splay trees can be carried out using the potential method. Suppose that size(r) is the number of nodes in the subtree rooted at r (including r) and rank(r) = log_2(size(r)). Then the potential function P(t) for a splay tree t is the sum of the ranks of all the nodes in the tree. This will tend to be high for poorly-balanced trees, and low for well-balanced trees. We can bound the amortized cost of any zig-zig or zig-zag operation by:

\[
\text{amortized cost} = \text{cost} + P(t_f) - P(t_i) \leq 3(\text{rank}_f(x) - \text{rank}_i(x)),
\]

where \( x \) is the node being moved towards the root, and the subscripts "f" and "i" indicate after and before the operation, respectively. When summed over the entire splay operation, this telescopes to 3(rank(root)) which is \( O(\log n) \). Since there's at most one zig operation, this only adds a constant.

Performance theorems

There are several theorems and conjectures regarding the worst-case runtime for performing a sequence \( S \) of \( m \) accesses in a splay tree containing \( n \) elements.

Balance Theorem\[^1\]

The cost of performing the sequence \( S \) is \( O\left(n\left(1 + \log n\right) + m \log n\right) \). In other words, splay trees perform as well as static balanced binary search trees on sequences of at least \( n \) accesses.

Static Optimality Theorem\[^1\]

Let \( q_i \) be the number of times element \( i \) is accessed in \( S \). The cost of performing \( S \) is \( O\left(m + \sum_{i=1}^{n} q_i \log \frac{m}{q_i}\right) \). In other words, splay trees perform as well as optimum static binary search trees on sequences of at least \( n \) accesses.

Static Finger Theorem\[^1\]
Let \( i_j \) be the element accessed in the \( j^{th} \) access of \( S \) and let \( f \) be any fixed element (the finger). The cost of performing \( S \) is \( O(m + n \log n + \sum_{j=1}^{m} \log(|i_j - f| + 1)) \).

Working Set Theorem\(^{[1]}\)

Let \( t(j) \) be the number of distinct elements accessed between access \( j \) and the previous time element \( i_j \) was accessed. The cost of performing \( S \) is \( O(m + n \log n + \sum_{j=1}^{m} \log(t(j) + 1)) \).

Dynamic Finger Theorem\(^{[4]} \)\(^{[5]}\)

The cost of performing \( S \) is \( O(m + n + \sum_{j=1}^{m} \log(|i_{j+1} - i_j| + 1)) \).

Scanning Theorem\(^{[6]}\)

Also known as the **Sequential Access Theorem**. Accessing the \( n \) elements of a splay tree in symmetric order takes \( O(n) \) time, regardless of the initial structure of the splay tree. The tightest upper bound proven so far is \( 4.5n \)\(^{[7]}\).

**Dynamic optimality conjecture**

In addition to the proven performance guarantees for splay trees there is an unproven conjecture of great interest from the original Sleator and Tarjan paper. This conjecture is known as the **dynamic optimality conjecture** and it basically claims that splay trees perform as well as any other binary search tree algorithm up to a constant factor.

**Dynamic Optimality Conjecture:**\(^{[1]}\) Let \( A \) be any binary search tree algorithm that accesses an element \( x \) by traversing the path from the root to \( x \) at a cost of \( d(x) + 1 \), and that between accesses can make any rotations in the tree at a cost of 1 per rotation. Let \( A(S) \) be the cost for \( A \) to perform the sequence \( S \) of accesses. Then the cost for a splay tree to perform the same accesses is \( O(n + A(S)) \).

There are several corollaries of the dynamic optimality conjecture that remain unproven:

**Traversal Conjecture:**\(^{[1]}\) Let \( T_1 \) and \( T_2 \) be two splay trees containing the same elements. Let \( S \) be the sequence obtained by visiting the elements in \( T_2 \) in preorder (i.e. depth first search order). The total cost of performing the sequence \( S \) of accesses on \( T_1 \) is \( O(n) \).

**Deque Conjecture:**\(^{[8]} \)\(^{[6]} \)\(^{[9]}\) Let \( S \) be a sequence of \( m \) double-ended queue operations (push, pop, inject, eject). Then the cost of performing \( S \) on a splay tree is \( O(m + n) \).

**Split Conjecture:**\(^{[10]}\) Let \( S \) be any permutation of the elements of the splay tree. Then the cost of deleting the elements in the order \( S \) is \( O(n) \).

**References**


\(^{[2]}\) "Randomized Splay Trees: Theoretical and Experimental Results" (http://www2.informatik.hu-berlin.de/~albers/papers/ip02.pdf). Retrieved 31 May 2011.


Splay tree


External links
• NIST's Dictionary of Algorithms and Data Structures: Splay Tree (http://www.nist.gov/dads/HTML/splaytree.html)
• Implementations in C and Java (by Daniel Sleator) (http://www.link.cs.cmu.edu/link/ftp-site/splaying/)
• Pointers to splay tree visualizations (http://wiki.algoviz.org/AlgovizWiki/SplayTrees)
• Fast and efficient implementation of Splay trees (http://github.com/fbuihuu/libtree)
• Top-Down Splay Tree Java implementation (http://github.com/cpdomina/SplayTree)
• Zipper Trees (http://arxiv.org/abs/1003.0139)

Tango tree

A Tango tree is an online binary search tree that is $O(\log \log n)$-competitive proposed by Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu in 2004.

Overview

Tango trees were designed to surpass the usual $O(\log n)$ binary search tree cost of operations. They perform basic operations such as searches in $O(\log \log n)$ time. This optimization is achieved dynamically by adjusting the search tree structure after each search. They are similar in their dynamic behaviour to other types of structure like a Splay tree however the competitive ratio is dramatically improved.

The approach is similar to the Greedy BST algorithm that while searching for an element rearranges the nodes on the search path to minimize the cost of future searches.

For Tango Trees the approach is a classic divide and conquer approach combined with a bring to top approach.

The main divide and conquer idea behind this data structure is to extract from the original tree a number of virtual smaller subtrees all with a normal $O(\log number of subtree elements)$ cost of access. These subtrees are dynamically balanced to offer the usual $O(\log n)$ performance for data retrieval.

The bring to top approach is not done at the node level as much as at the subtree level which further improve competitiveness. Once the original tree has been adjusted to include a collection of these subtrees, it is possible to greatly improve the cost of access of these subtrees. Both the Tango tree and these subtrees are a type of Self-balancing binary search tree.

Tango tree achieves this outstanding competitive ratio by using a combination of augmentation of attributes in the data structure, a more elaborated algorithm and the use of other type of trees for some of its structure.
Example

Fig. 1 An example of a Tango Tree

Similar Data Structures

- Red Black tree, introduced by Bayer in 1972, having a $O(\log n)$ competitive ratio
- Splay tree, introduced by Sleator and Tarjan in 1985, having a $O(\log n)$ competitive ratio
- AVL tree, introduced by Adelson and Landis in 1962, having a $O(\log n)$ competitive ratio
- Multi-splay tree, introduced by Sleator and Wang in 2006, having a $O(\log \log n)$ competitive ratio

Advantages

Tango Trees offer unsurpassed competitive ratio retrieval for online data. Online data means that operations that are not known in advance before the data structure is created.

Outstanding search performance for a Tango tree relies on the fact that accessing nodes constantly updates the structure of the search trees. That way the searches are rerouted to searches in much shallower balanced trees.

Obviously, significantly faster access time constitutes an advantage for nearly all practical applications that offer searches as a use case. Dictionary searches like telephone directory search would be just one of the possible examples.

Disadvantages

The Tango tree focuses on data searches on static data structures, and does not support deletions or insertions, so it might not be appropriate in every situation.

The Tango tree uses augmentation, meaning storing more data in a node than in a node of a plain binary search tree. Tango trees use $O(\log \log n)$ bits. Although that is not a significant increase, it results in a bigger memory footprint.

It is a complex algorithm to implement like for instance splay tree, and it also makes use of rarely used operations of Red-Black Tree.

Tango trees change when they are accessed in a 'read-only' manner (i.e. by find operations). This complicates the use of Tango trees in a multi-threaded environment.

It is believed that Tango Tree would work in a practical situation where a very large data set with strong spatial and temporal coherence fits in the memory.
Terminology and Concepts

There are several types of trees besides the Red-Black trees (RB) used as a base for all Tree structures:

Reference Trees

Example of reference tree:

![Reference tree and preferred paths](image)

Tango Trees

See Fig 1 for an example of Tango tree

Auxiliary Trees

Example of auxiliary tree:

![Example of Auxiliary Tree formed from a Preferred Path](image)

As all trees are derived from RB trees so they are also [Binary Search Trees] with all their inherent behaviour.

Auxiliary trees can be considered sub-trees of the Tango Tree. Tango Trees are the actual employed trees while in production mode.

Reference Trees are used only initial set-up and for illustration of the concepts.

Any search in the Reference Tree creates a path from root to the searched node. We call that a Preferred Path and the Preferred Child attribute specific to the Reference Tree indicates if the preferred path of a node goes to the left or right child if any. A Preferred Path is determined by the longest path formed by preferred children. Any new search in the Reference Tree will carve new paths and modify the existing paths. Correspondingly, the preferred children change too.

Any switch from right to left or vice versa is called an Interleave. Interleaves changes are the basis for analysis of expected performance.
**Operations**

As we stated Tango Trees are static so they support only searches. That also means that there is a construction phase where the elements are inserted in the Tango Tree. That start-up cost and any search performance during the construction period is not considered part of the operational part of Tango trees therefore the performance is not \( O(\log \log n) \) competitive. The outstanding idea behind Tango Trees is to collect the nodes belonging to a Preferred Path as a balanced tree of height \( O(\log \log n) \) called auxiliary tree and then assemble them in a tree of trees where higher trees contain the mostly accessed preferred paths elements.

**Search**

To search for a node \( x \) in a Tango tree, we search for the element in the collection of Auxiliary Trees that make up the Tango Tree like in any ordinary binary search tree. Simultaneously we adjust the corresponding affected Auxiliary Trees in the Tango Tree. This will preserve the ideal structure that will give us this unprecedented search performance. We could achieve this ideal structure by updating the Reference Tree \( P \) after every search and recreating the Tango tree however this would be very expensive and nonperforming. The reasons why such a structure is \( O(\log \log n) \) competitive is explained in the Analysis. There is a direct way to update the structure and that is shown in the [Algorithm]

**Tango Tree Life Cycle**

The main phases are Construction and Operation

**Construction**

First create the reference tree and insert the desired data in it. Update the attributes of depth for each node. After this phase the data and the value of the depth for the nodes will remain unchanged. Let's call that field \( d \) for further reference and understand that it always refers to the Reference tree not to the Tango tree as that can cause confusions. While in principle the reference tree can be any balanced tree that is augmented with the depth of each node in the tree the TODO [Demaine et al. 2004] uses [red-black tree]. Secondly we will perform some warm-up searches with the goal to create a decent distribution of Preferred Paths in the Reference Tree. Remember there is no Tango tree and all this is done on line. This means that performance is not critical at this point.

After this begins the phase of collecting the preferred paths. Out of each Preferred Path we create a new Auxiliary Tree which is just an ordinary RedBlack Tree where the nodes inherit the value of field \( d \). That value will stay unchanged forever because it stays with the node even if the node is moved in the trees. There is no Tango Tree at this point. We add auxiliary trees such a way that the largest one is the top of a tree and the rest and 'hung' below it. This way we effectively create a forest where each tree is an Auxiliary tree. See Fig. 1 where the roots of the composing auxiliary tree are depicted by magenta nodes. It after this step that the Tango tree becomes operational. See [Construction Algorithms And Pseudo-code] for main algorithms And pseudo-code.
Operation
The operation phase is the main phase where we perform searches in the Tango tree. See [Operation Algorithms And Pseudo-code] for main algorithms And pseudo-code.

Data Augmentation

Reference Tree augmentation
 Besides the regular RB Tree fields we introduce two more fields:
• isPreferredChild representing the direction the last search traversing the node took. It could be either Boolean or a pointer to another node. In the figures it is represented by a red line between a node and its preferred child.
• d representing the depth of the node in the Reference Tree. (See value of d in Fig 3.)

Tango Tree Augmentation
 For each nodes in Tango or Auxiliary Tree we also introduce several new fields:
isRoot that will be false for all nodes except for the root. This is depicted in magenta for nodes where isRoot is true.
maxD that is the Maximum value of d for all the children of the node. This is depicted as MD in the figures. For some nodes this value is undefined and the field is not depicted in the figure.
minD that is the Minimum value of d for all the children of the node. This is depicted as mD in the figures. For some nodes this value is undefined and the field is not depicted in the figure.
isRoot, maxD and minD will change in time when nodes are moved around in the structure.

Algorithms
The main task in a Tango Tree is to maintain an ‘ideal’ structure mirroring changes that occur in the reference tree. As recreating the Tango tree form the reference tree would not be performing the algorithms will have only use and modify the Tango tree. That means that after the construction phase the reference tree could and should be destroyed. After this we would refer to it as virtual meaning ‘as if it existed’ . As described in [Demaine et al. 2004], the main goal is to maintain the ideal Tango structure that would mimic preferred paths changes on the virtual reference tree. So the purpose of the Tango algorithm is the constructing of a new state Ti of the Tango Tree based on the previous state Ti-1 or the Tango Tree and the new search of xi.

Tango Search
During the search for xi, one or many auxiliary trees could to be traversed. During this Tango walk phase of the Tango search every time when we are crossing from an auxiliary tree in a new auxiliary tree we perform exactly one cut operation on the tree just traversed.
The result of the Cut is then Joined with the newly entered auxiliary tree set of data and repeats for each new auxiliary tree encountered creating a snowball effect. Note that in this analogy, the snowball casts off some nodes and collects other nodes in its trajectory towards the final auxiliary tree that contains the searched element. Before performing a cut, the new auxiliary tree is queried to obtain a cut depth which is used in processing the previous auxiliary tree. Starting from a Reference tree in Fig. 7 obtained after performing searches on 17, 7, 1, 18, 23, 31, 27, 13, 9, 20, we see the corresponding Tango Tree as in Fig. 8. On this tree during a walk towards element xi =23 we would have traversed the top auxiliary tree 20 and entered auxiliary tree 22. Note that preferred paths are marked in red and auxiliary tree roots in magenta. Remember that all auxiliary trees are actually RB trees so the red shadows on some nodes represent the red nodes in the RB trees.
In case we would have searched for element 7 during the Tango Walk we would have crossed the top auxiliary tree rooted at 20, the auxiliary tree rooted at 10, auxiliary tree rooted at 2 and auxiliary tree rooted at 6. That means that we would have to perform the Tango Cut-And-Join several times. For each entrance to a new auxiliary tree, the tree that was just crossed is processed by a Tango cut algorithm. In order to perform the Tango cut we need to determine the cut range. A specific d value is the input data in the algorithm to determine the cut range.

**Determining d**

This specific d value is determined every time when we cross the boundary to a new auxiliary tree and it is determined from the soon-to-be-traversed auxiliary tree. We know when we cross the boundary by looking at the isRoot attribute of each node en route to xi. We ignore the isRoot attribute of the root of Tango tree. To simplify the code we don’t even set it on Tango root and that is why top nodes are not colored in magenta in any of the figures.

The value of d is determined by subtracting 1 from the minimum between the root of the new auxiliary tree minD value and its current d value.

So in search for 23 we reach auxiliary tree rooted at 22 we calculate minimum between its minD value and its d value and we subtract 1 and we get this special value of d =2. Please observe by looking in the reference tree in fig. 7 that, that is exactly the depth value where the new search will change the value of a preferred child and create a new interleave. See node 20 on the reference tree in fig 7.

There are some particular cases when either minD or d are out of range or unknown. In the implementation that is represented by a -1, a value which denotes + or -infinity if the value denotes the right side of the range or left side of the range. In all the figures the nodes where the value of minD or maxD is -1 do not show the corresponding value(s) for brevity reasons.

The value of -1 is screened out during the determination of a minimum so it does not mask legitimate d values. This is not the case for maxD.
Determining the cutting range

Considering the auxiliary tree A in Fig 9, we need to determine nodes l and r. According to [Demaine et al. 2004] the node l is the node with the minimum key value (1) and the depth greater than d (2) and can be found in O(log k) time where k is the number of nodes in A.

We observe by looking at the virtual reference tree that all nodes to be cut are actually under the interleave created by the search at the node corresponding to the auxiliary tree the tango search is about to enter. In Fig. 9 we show the reference tree at state ti after a search for 9 and in Fig. 10 we show the reference tree at state ti+1 after a search for 15. The interleave appears at node 12 and we want to ‘cut’ all the nodes in the original path that are below 12. We observe their keys are all in a range of values and their d values are higher than the cutting d which is the depth of the interleave node (d=2).

Of course we need to find the nodes to be cut not in the reference tree but in the corresponding auxiliary tree. We use the value of depth obtained during the Tango search as input to calculate the cut range.

We need to determine the minimum key of a node l that has the depth greater than d. This node can be found in O(log k) time and is the left boundary of our interval. In the same way, a node r with the maximum key value and the depth greater than d can also be found in O(log k), where k is the number of nodes in A.

We already observed that the keys of the nodes to cut are in a key range so instead to take all the nodes in the corresponding auxiliary tree and check if their d is greater than the input d we can just find the left side of the range and the right side of the range.
Finding l

We first find l which is the leftmost element to be cut (using getL). We determine it by walking to the leftmost child whose sub-tree has either its d value or maximum depth greater than d.

No nodes meeting this criteria result in l being -infinity (or NIL in the implementation). This means that the cut interval extends all the way to the left so during the cut or the join less split and concatenate operations have to be performed.

Finding r

For finding r we walk right to the rightmost node whose d value is greater than d. No nodes meeting this criteria result in r being +infinity (or NIL in the implementation). This means that the cut interval extends all the way to the left so during the cut or the join less split and concatenate operations have to be performed.

Algorithms 7 and 8 describe the implementation of getL and getR.

Finding l' and r'

Following the determination of l and r, the predecessor l' of the node l and the successor r' of r are found. These new nodes are the first nodes that 'survive' the cut. A NIL in l will also result in a NIL l' and a NIL in r will result in a NIL in r'.

Both nodes being NIL take a new meaning signifying that all the nodes will survive the cut so practically we can skip the Tango cut and takes the set of nodes directly to Tango join.

During the Tango Search when we finally reached the searched node the tango cut algorithm is run once more however its input provided by Tango search is changed. The input value of d is now the d of the searched node and the tree to be cut is the auxiliary tree that contains the searched node.

During the Tango Search after encountering any new auxiliary tree (NAT) the Tango cut is normally applied on the tree above (except for final cut as described above) and results in a new structure of auxiliary trees. Let's call this result of the cut A.

The Join operation will join a set of nodes with the nodes in the NAT. There is an exception that applies when we reached the searched node, in which case we join A with the auxiliary tree rooted at the preceding marked node of the searched node.

The Tango cut algorithm consists of a sequence of tree split, mark and concatenate algorithms. The Tango join algorithm consists of a different sequence of tree split, un-mark and concatenate algorithms. Marking a node is just setting the isRoot attribute to true and un-marking setting the same attribute to false.

RB Split and RB Concatenate

Red-black trees support search, split and concatenate in O(log n) time. The split and concatenate operations are the following:

Split: A red-black tree is split by rearranging the tree so that x is at the root, the left sub-tree of x is a red-black tree on the nodes with keys less than x, and the right sub-tree of x is a red-black tree on the nodes with keys greater than x.

Concatenate: A red-black tree is concatenated by rearranging x's sub-tree to form a red-black tree on x and the nodes in its sub-tree. As condition for this operation all the nodes in one of the trees have to have lower values than the nodes in the other tree.

Note that the behavior for split and concatenate in [Demaine et al. 2004] differs slightly from the standard functionality of these operations as the signature of the operations differ in terms of number and type of input and output parameters.
The two operations describe above apply only to an auxiliary tree and do not cross into other auxiliary trees. We use the isRoot information for each node to avoid wandering in other trees.

**Tango Cut algorithm**

The main purpose of the cut is to separate a tree in two sets of nodes. Nodes need to be pushed to the top because the search path in the virtual reference tree traversed them and nodes that become less important (cut nodes) and are pushed downwards to the next auxiliary tree.

We already determined the values of \(l'\) and \(r'\) and we want to cut the input auxiliary tree (A in Fig. 11) in the range \(l'\) to \(r'\). We need to split A in three trees and then cut the middle tree and re-assemble the remaining parts. We start with a split at \(l'\) that creates tree B and tree C. Tree B will be free of nodes to cut but tree C will contain nodes to cut and nodes to keep. We do a second split at \(r'\) and obtain trees D and E. D contains all nodes to be cut and E contains all the nodes to keep. We then mark the root of D as isRoot therefore logically pushing it to a lower level. We then concatenate at \(r'\). Note: this is not really a standard RB concatenate but rather a merge between a node and a tree. As a result we obtained C. The last operation is to concatenate C tree, B tree and node \(l'\) obtaining the nodes we want to keep in the new tree A. Note: this is not really a standard RB concatenate but rather a merge between a node and a tree and then a standard two tree concatenation.

The resulting new node A is actually composed of two auxiliary trees: the top one that contains nodes we want to favor and the lower ’hung’ D which contains nodes that get pushed downwards via this operation. The nodes being pushed downwards are exactly the nodes that were on the old preferred path corresponding to the auxiliary tree being processed but not in the new preferred path. The nodes being pushed upwards (now in A) are exactly the nodes that were became part of the new preferred path due to the performed search. Fig. 11 shows this flow of operations.

![Fig. 11 Tango cut consisting of split, mark and concatenate operations on an auxiliary tree](image)

The following special cases may occur:

- \(l'\) and \(r'\) are NIL so cut is performed.
- \(r'\) is NIL we just do a split at \(l'\) and C becomes the result while we hang B under C via a mark operation.
- \(l'\) is NIL we just do a split at \(r'\) and E becomes the result and we hang the left resulting tree under E.

The result will be then joined with the content of the next auxiliary tree via the Tango Join algorithm.

**Tango Join algorithm**

During the Tango Search after encountering any new auxiliary tree (NAT) the Tango Cut being applied on the tree above the NAT results in a new structure of auxiliary trees. We can think of that as a Tango sub tree. It will normally contain at least two connected auxiliary trees. The top tree A containing the nodes we want to keep close to the surface of the Tango Tree (so we can achieve the 'bring to top' approach) and 'hang' to it is auxiliary tree D which was pushed downwards. In case of search for 23 all of the nodes from previous auxiliary tree should be kept close to surface and the set of nodes destined to be moved in D is empty so we have no D. Regardless of the situation the result of the Tango cut will contain at least one node (the Tango root) in Tree A. Let's call this result of the cut A.
The Join will join a set of nodes with the nodes in the NAT. That is done via two splits, an un-mark and two concatenates.

Fig. 12 shows the high level sequence where A is coming from the previous cut tree and B is the NAT. We observe that NAT is actually hung under the tree that was just cut therefore the values of its keys are all in a range of two adjacent keys (a key and its successor) in the tree that was just cut. That is normal for any BST. If the NAT is hanging as a left tree the parent node marks the right side of the range while its predecessor (in the tree that was just cut universe) marks the left side of the range. So in order to join the two trees we just have to wedge B under to the left of its parent in A. Similarly for the case where B happens to hang to the right of its parent where we wedge the content of B to the right of its parent. In order to find the wedge insertion point we can just search in the A for the root of NAT. Of course that value is not in NAT but it will find a close value and by taking its predecessor or successor (depending on the search algorithm and if the close value was before or after the value) we find the two nodes between where B should be wedged. Let’s call these values lPrime and rPrime. Next is to split A first at lPrime and then at rPrime therefore creating three trees, a First part (FP), middle part (MP) and last part (LP). While these operations where done in the A universe they also need to carry all the other auxiliary trees as in the Tango universe. In the Tango (forest) universe we discover that MD is actually is B however is severed logically from rPrime because its root is marked as isRoot and it appears like a hung auxiliary tree. Since we want that wedged, we “un-mark” it) by resetting its isRoot attribute and making it logically part of rPrime. Now we have the final structure in place but we still need to concatenate it first at rPrime and then at lPrime to absorb all the nodes under the same Joined resulting tree.

The difficulty in doing this is the fact that the standard concatenates do not take nodes but just trees. A two tree and a node operation could be constructed and then repeated to obtain a Tango concatenate; however, it is hard to preserve the RB integrity and is dependant on the order of operations so the resulting structure is different even if it contains the same node. That is an issue because we can not control the exact reproduction of the ideal structure as if generated from the reference tree. It can be done in such a way to contain all nodes and preserve the correct RB tree structure however the geometry is dictated by the RB concatenates and is not necessarily the ideal geometry mirroring perfectly the reference tree.

![Fig. 12 Tango cut consisting of split, mark and concatenate operations on an auxiliary tree](image)

So for example let’s say search for 23. We obtain A as the result on the first Tango cut on the top auxiliaty tree. See Fig. 13 where 22 is the root of NAT.

![Fig. 13 Tango tree before a search for 23](image)
We use the value of NAT (22) to search in the tree above and we obtain 20 and 24 as the lPrime and rPrime nodes. We split at lPrime (20) and we obtain FP as in Fig. 14 and LP as in Fig. 15.

![Fig. 14 First sub-tree of Tango tree (FP) after executing the first split of Tango join operation](image)

We then split for the second time at rPrime (24) to get the last tree LT as in Fig. 15.

![Fig. 15 Last sub-tree of Tango tree (FP) after executing the second split of the Tango join operation](image)

Next we unmark B which is rooted at 22 and we obtain the result in Fig 16. As you can see 22 now is part of the top of the structure. That makes sense if you look at Fig. 10 representing the ‘virtual’ reference tree. To reach 23 which is our target we would have had to go through 22.

![Fig. 16 Sub-tree rooted at rPrime after executing the second split of the Tango join operation](image)

We then concatenate with rPrime and obtain the result in presented in Fig. 17: Second concatenation takes place and it this particular example will not result in rearranging of the nodes so Fig. 17 is the final result of the Join operation.
Construction Algorithms And Pseudo-code

Construct the reference tree and perform warm-up searches.

**Function:** constructReferenceTree  
**Input:** None  
**Output:** ReferenceTree p

```java
ReferenceTree p = new ReferenceTree();
insertDataInReferenceTree();
p.setAllDepths();
p.warmUpSearches();
```

```java
ArrayList<PreferredPath> paths = p.collectAndSortPreferredPaths();
assert paths.size() > 0;
```

```java
PreferredPath top = p.collectTopNodePath(p.root);
TangoTree tangoTree = new TangoTree(top);
tangoTree.updateMinMaxD();
```

```java
while (takeNext PreferredPath path in paths) do
    if (path.top = p.root) then
        continue; // skip the top path as it was already added
    else
        RBTree auxTree = new RBTree(path);
        auxTree.updateMinMaxD();
        auxTree.root.isRoot = true;
        tangoTree.hang(auxTree);
```

```java
return p;
```

Construct an Auxiliary tree out of a Preferred Path. Used in the construction phase.

**Function:** constructAuxiliaryTree  
**Input:** Preferred Path path  
**Output:** AuxiliaryTree this

```java
RBTree(PreferredPath path)
```

```java
this()
    RefTreeNode refNode = path.top
    while (Next PreferredPath path in paths exists) do
        RBNode n = new RBNode(refNode.value, RedBlackColor.RED, null, null)
        n.d = refNode.d
        this.insert(n)
        refNode = refNode.getPreferredChild()
```

Construct a Tango tree from the Reference tree.

**Function:** constructTangoTree  
**Input:** ReferenceTree p  
**Output:** TangoTree tangoTree

```java
ArrayList<PreferredPath> paths = p.collectAndSortPreferredPaths();
assert paths.size() > 0;
PreferredPath top = p.collectTopNodePath(p.root);
```

```java
TangoTree tangoTree = new TangoTree(top);
tangoTree.updateMinMaxD();
```

```java
while (takeNext PreferredPath path in paths) do
    if (path.top = p.root) then
        continue; // skip the top path as it was already added
    else
        RBTree auxTree = new RBTree(path);
        auxTree.updateMinMaxD();
        auxTree.root.isRoot = true;
        tangoTree.hang(auxTree);
```

```java
return p;
```
Tango tree

```java
TangoTree tangoTree = new TangoTree(top)
tangoTree.updateMinMaxD()

while (Next PreferredPath path in paths exists) do
    if (path.top = p.root) then
        continue; // skip the top path as it was already added
    else
        RBTree auxTree = new RBTree(path)
        auxTree.updateMinMaxD()
        auxTree.root.isRoot = true

return tangoTree
```

Set the depths of all nodded in the tree. Used only once for setting the depths of all nodes in the reference tree.

**Function:** setAllDepths

**Input:** None

```
setAllDepthsHelper(root)
```

**Algorithm 14.** Set the depths of all nodded in the tree.

**Function:** setAllDepthsHelper

**Input:** RefTreeNode n

```
if (n == NILL) then
    return
n.setD(getDepth(n))
if (n.right != null) then
    setAllDepthsHelper(((RefTreeNode) n.right))
if (n.left != null) then
    setAllDepthsHelper((RefTreeNode) (n.left))
```

---

**Operation Algorithms And Pseudo-code**

There is just one operation: search that calls a number of algorithms to rearrange the data structure.

**Tango search pseudocode.** Used by the main operation on a Tango tree.

**Function:** tangoSearch

**Input:** int vKey

```
Node currentRoot = root
Node currentNode = root
Boolean found = false

while (true) do
    if (currentNode == NILL) then
        found = false
        break
    if (currentNode.isRoot && currentNode != root) then
        cutAndJoin(minIgnoreMinusOne(currentNode.minD, currentNode.d) - 1, currentRoot, currentNode)
        currentRoot = currentNode
    if (currentNode.value == vKey) then
        found = true
    if (currentNode != currentRoot) then
        cutAndJoin(currentNode.d, currentRoot, currentRoot)
```

break
if (currentNode.value < vKey) then
    currentNode = (RBNode) currentNode.right
else
    currentNode = (RBNode) currentNode.left
if (found) then
    return currentNode
else
    return NILL

Tango Cut and Join used by Tango Search
Function: cutAntJoin
Input: int d, Node currentRoot, Node newlyFoundRoot
RBNode n = currentRoot
RBNode l = NILL // l is the last node that gets cut
RBNode lPrime = NILL // l’ is the first node that escapes from cutting
RBNode r = NILL
RBNode rPrime = NILL
if (currentRoot.maxD <= d) // no nodes to be cut besides maybe the root
    if (currentRoot.d > d)
        l = currentRoot
        r = currentRoot
        lPrime = getPredecessor(l, currentRoot)
        rPrime = getSuccessor(r, currentRoot)
else // there are nodes to be cut underneath
    l = getL(d, n, currentRoot)
    // determine lPrime as predecessor of l or NILL if it is the last
    if (l != NILL)
        lPrime = getPredecessor(l, currentRoot)
    else
        lPrime = NILL // - infinity maybe redundant

    // end calculating l and l prime
    // find r the right side node of the cutting range based on value
    n = currentRoot
    r = getR(d, n, currentRoot)
    if (r != NILL) // the root is not to be cut
        rPrime = getSuccessor(r, currentRoot)
checkLandR(d, l, r, currentRoot)
RBTree aTree = NILL
if (lPrime == NILL && rPrime == NILL) // nothing to cut therefore so
    aTree = new RBTree()
aTree.root = currentRoot
else
RBTreePair aAndDtreePair = new RBTreePair()
aTree = tangoCut(lPrime, rPrime, currentRoot)

RBTree afterCutAndJoin = tangoJoin(aTree,
newlyFoundRootOrCurrentIfWeFound)

Tango Cut used by Tango cut And Join to separate nodes that need to be
pushed to top from the rest of nodes.

Function: tangoCut
Input: RBNode lPrime, RBNode rPrime, RBNode aRoot
Output: RBTree

saveShadowAuxTrees(aRoot)
if (lPrime == null || rPrime == null) {// just one splitAnd CONCATenate

return simplifiedTangoCut(lPrime, rPrime, aRoot)
}

RBTree a = new RBTree()
a.root = aRoot

RBTreePair firstPartAndLastPart = new RBTreePair()
split(lPrime, a, firstPartAndLastPart)
RBTree b = firstPartAndLastPart.firstPart
RBTree c = firstPartAndLastPart.lastPart
firstPartAndLastPart.firstPart.verifyProperties()
firstPartAndLastPart.lastPart.verifyProperties()
firstPartAndLastPart = new RBTreePair()
split(rPrime, c, firstPartAndLastPart)// problem
firstPartAndLastPart.firstPart.verifyProperties()
firstPartAndLastPart.lastPart.verifyProperties()
RBTree d = firstPartAndLastPart.firstPart
RBTree e = firstPartAndLastPart.lastPart
// disconnect d
rPrime.left = NILL
d.root.parent = NILL

Tango Join used by Tango Cut and Join to join the result of Tango cut
to auxiliary trees.

Function: tangoJoin
Input: RBTree a, RBNode newlyFoundRoot
Output: RBTree finalJoinResult

RBTree bPrevOp = new RBTree()
RBTree d = new RBTree()
order(bPrevOp, d, a, newlyFoundRoot)
RBNodePair lAndR = bPrevOp.searchBoundedLPrimeAndRPrime(d.root.value)
if (lPrime == null || rPrime == null) // just one split and one
concatenate

return simplifiedTangoJoin(lPrime, rPrime, bPrevOp, d.root)

RBNode lPrime = lAndR.lPrime
RBNode rPrime = lAndR.rPrime

RBTreePair firstPartAndLastPart = new RBTreePair()
split(lPrime, bPrevOp, firstPartAndLastPart)
RBTree b = firstPartAndLastPart.firstPart
RBTree c = firstPartAndLastPart.lastPart
firstPartAndLastPart.firstPart.verifyProperties()
firstPartAndLastPart.lastPart.verifyProperties()
firstPartAndLastPart = new RBtreesPair()
split(rPrime, c, firstPartAndLastPart) //
firstPartAndLastPart.firstPart.verifyProperties()
firstPartAndLastPart.lastPart.verifyProperties()
RBTree e = firstPartAndLastPart.lastPart
// reconnect d which is normally newlyFoundRoot
d.root.isRoot = false // un-mark, a difference from tangoCut
// concatenate part
rPrime.parent = NILL // avoid side effects
RBTree res1 = concatenate(rPrime, d, e)
lPrime.parent = NILL // avoid side effects
RBTree res2 = concatenate(lPrime, b, res1)
return res2

Check if a node n is in an auxiliary tree defined by currentRoot. Used to verify wandering.
Function: isInThisTree
Input: RBNode n, RBNode currentRoot
Output: Boolean v
if (n.isRoot and n != currentRoot) then
  return false
else
  return true

Find node l as left of range used by Tango Cut, different from the
[Demaine et al. 2004] paper.
Function: getL
Input: int d, Node n, Node currentRoot
Output: Node l
Node l = n
if (left[n] != NIL) and (not(isRoot(n) or n == currentRoot) and
((left(n).maxD > d) or (left(n).d > d)) then
  l=getL(d, left(n), currentRoot)
else
  if (n.d > d)
    l = n
  else
    l=getL(d, right(n), currentRoot)
return l

Find node r as the right limit of the range used by Tango Cut.
Function: getR
Input: int d, Node n, Node currentRoot
Output: Node r

Node r = n
if (right[n] != NIL) and (not(isRoot(n) or n == currentRoot) and ((right(n).maxD > d) or (right(n).d > d)) then
  r=getR(d, left(n), currentRoot)
else
  if (n.d > d)
    r = n
  else
    r=getR(d, right(n), currentRoot)
return r

Return Sibling. Within enclosing auxiliary tree boundary
Function: siblingBound
Input:  RBNode n, RBNode boundingRoot
Output: Node p
if (n == left[parent[n]] && isInThisTreeAcceptsNull(left[parent[n]], boundingRoot)) then
  if (isInThisTreeAcceptsNull(right[parent[n]], boundingRoot)) then
    return right[parent[n]]
  else
    return NIL
else
  if (isInThisTreeAcceptsNull(left[parent[n]], boundingRoot)) then
    return left[parent[n]]
  else
    return NIL

Return Uncle. Within enclosing auxiliary tree boundary
Function: uncleBound
Input:  RBNode n, RBNode boundingRoot
Output: Node p
if (isInThisTreeAcceptsNull(parent[n], boundingRoot)) then
  return siblingBound(boundingRoot)
else
  return NIL

Update Min Max D values in red black tree augmented node. Used to update Tango Tree node attributes.
Function: updateMinMaxD
Input:  RBNode n
int minCandidate
if (n.left != NIL) then
  updateMinMaxD(n.left)
if (n.right != NIL) {
  updateMinMaxD(n.right)
if (n.left != NILL) then
    int maxFromLeft = max(n.left.d, n.left.maxD)
    n.maxD = maxFromLeft > n.maxD ? maxFromLeft : n.maxD
    if (n.left.minD != -1) {
        minCandidate = min(n.left.d, n.left.minD)
    } else
        minCandidate = n.left.d
    if (n.minD != -1) then
        n.minD = min(n.minD, minCandidate)
    else
        n.minD = minCandidate
if (n.right != NILL) then
    int maxFromRight = max(n.right.d, n.right.maxD)
    n.maxD = maxFromRight > n.maxD ? maxFromRight : n.maxD
    if (n.right.minD != -1) then
        minCandidate = min(n.right.d, n.right.minD)
    else
        minCandidate = n.right.d
    if (n.minD != -1) then
        n.minD = min(n.minD, minCandidate)
    else
        n.minD = minCandidate

Search Bounded lPrime and rPrime used by Tango Join.
Function: searchBoundedLPrimeAndRPrime
Input:   RBTree rbTree, int value
Output:  NodePair p
RBN ode n = root
RBN ode prevPrevN = NILL
RBN ode prevN = NILL
RBN odePair lAndR
while (n != NILL && isInThisTree(n, rbTree.root)) do
    int compResult = value.compareTo(n.value)
    if (key(n) == value) then
        lAndR = new RBN odePair(n, n)
        return lAndR
    else
        if (key(n) < value) then
            prevPrevN = prevN
            prevN = n
            if (isInThisTree(n.left, rbTree.root)) then
                n = n.left
            else
                n = NILL
        else
            prevPrevN = prevN
            prevN = n
if (isInThisTree(n.right, rbTree.root) then
    n = n.right
else
    n = NIL
lAndR = new RBNODEPair(prevPrevN, prevN)

return lAndR

The minimum in a binary search tree is always located if the left side path is traversed down to the leaf in $O(\log n)$ time:

Minimum Value Tree pseudocode. Used by successor.

Function: min_val_tree
Input: Node x
Output: Node x
while left(x) != NIL do
    x = left(x)
return x

Maximum Value Tree pseudocode used by predecessor.

Function: max_val_tree
Input: Node x
Output: Node x
while right(x) != NIL do
    x = right(x)
return x

The next two algorithms describe how to compute the predecessor and successor of a node. The predecessor of a node $x$ is a node with the greatest value smaller the key[$x$].

Predecessor computing pseudocode used to find lPrime.

Function: predecessor
Input: RBNODE x
Output: RBNODE y
RBNODE y = null
if (n.left != null && isInThisTree((RBNODE) (n.left)), root))
    return getMaximum((RBNODE) (n.left), root)
y = (RBNODE) (n.parent)
if (y == currentRoot) // don't let it escape above its own root
    return NIL
--------------------------------------------------------------------->
while (y != NIL && y != currentRoot && n == ((RBNODE) (y.left))) do
    n = y
    if (isInThisTree(((RBNODE) (y.parent)), root))
        y = (RBNODE) (y.parent)
    else
        y = null
return (RBNODE) y
Successor computing algorithm used to find rPrime.

Function: successor
Input: Node x
Output: Node y

RBNode y = NILL
if (n.right != NILL && isInThisTree({(RBNode) (n.right)}, currentRoot))
    return getMinimum({(RBNode) (n.right)}, currentRoot)

y = (RBNode) (n.parent)
if (y == currentRoot) // don't let it escape above its own root
    return NILL

while (y != NILL && isInThisTree(y, currentRoot) && n == {(RBNode) (y.right)})
    n = y
    y = (RBNode) (y.parent)
return (RBNode) y

Traverse Tree pseudocode used by Tango Search.

Function: traverse_tree
Input: Node x
Output: None

if x != NIL and InTheAuxiliaryTree(x) then
    traverse_tree (left(x))
    traverse_tree (right(x))

Search Tree algorithm used to find lPrime and rPrime for Tango Join.

Function: search_tree
Input: Node x, value
Output: Node x

if x = NIL or k = key[x] then
    return x
if k < key[x] then
    return search_tree(left[x]; value)
else
    return search_tree(right[x]; value)

Find minimum by ignoring specific values used for the calculation of d.

Function: minIgnoreMinusOne
Input: int minD, int d
Output: int d
if (minD == -1) then
    return d
if (d == -1) then
    return minD
return min(minD, d)


Analysis

Here are some elements necessary to understand why the Tango Tree achieve such an amazing performance and become $O(\log \log n)$ competitive.

Wilber’s 1st Lower Bound [Wil89]

Fix an arbitrary static lower bound tree $P$ with no relation to the actual BST $T$, but over the same keys. In the application that we consider, $P$ is a perfect binary tree. For each node $y$ in $P$, we label each access $X_1 \text{ L}$ if key $X_1$ is in $y$’s left subtree in $P$, $R$ if key $X_1$ is in $y$’s right subtree in $P$, or leave it blank if otherwise. For each $y$, we count the number of interleaves (the number of alterations) between accesses to the left and right subtrees: $\text{interleave}(y) = \# \text{ of alternations } L \text{ ? } R$.

Wilber’s 1st Lower Bound [Wil89] states that the total number of interleaves is a lower bound for all BST data structures serving the access sequence $X$. The lower bound tree $P$ must remain static. Proof.

We define the transition point of $y$ in $P$ to be the highest node $z$ in the BST $T$ such that the root-to-$z$ path in $T$ includes a node from the left and right subtrees if $y$ in $P$. Observe that the transition point is well defined, and does not change until we touch $z$. In addition, the transition point is unique for every node $y$.

Lemma 1

The running time of an access $x_i$ is $O((k + 1)(1 + \log \log n))$, where $k$ is the number of nodes whose preferred child changes during access $x_i$.

Lemma 2

The number of nodes whose preferred child changes from left to right or from right to left during an access $x_i$ is equal to the interleave bound $IB_i(X)$ of access $x_i$.

Theorem 1

The running time of the Tango BST on an sequence $X$ of $m$ accesses over the universe $1, 2, ..., n$, is $O((OPT(X) + n)(1 + \log \log n))$ where $OPT(X)$ is the cost of the offline optimal BST servicing $X$.

Corollary 1.1

When $m = n$, the running time of the Tango BST is $O(OPT(X)(1 + \log \log n))$
Bibliography


• ERIK D. DEMAINE, DION HARMON, JOHN IACONO, AND MIHAI PATRASCU, 2004, Dynamic optimality-almost, FOCS 2004


• DANIEL DOMINIC SLEATOR, Open Source top-down splay tree implementation, http://www.link.cs.cmu.edu/splay/
External links

- DANIEL DOMINIC SLEATOR, Open Source top-down splay tree implementation [3]

References


Skip list

A skip list is a data structure for storing a sorted list of items using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. These auxiliary lists allow item lookup with efficiency comparable to balanced binary search trees (that is, with number of probes proportional to log \( n \) instead of \( n \)).

![Skip list diagram](image)

Each link of the sparser lists skips over many items of the full list in one step, hence the structure's name. These forward links may be added in a randomized way with a geometric / negative binomial distribution. Insert, search and delete operations are performed in logarithmic expected time. The links may also be added in a non-probabilistic way so as to guarantee amortized (rather than merely expected) logarithmic cost.

Description

A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer \( i \) appears in layer \( i+1 \) with some fixed probability \( p \) (two commonly-used values for \( p \) are 1/2 or 1/4). On average, each element appears in \( 1/(1-p) \) lists, and the tallest element (usually a special head element at the front of the skip list) in \( \log_{1/p} n \) lists.

A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, or the search reaches the end of the linked list, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is at most \( 1/p \), which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. Therefore, the total expected cost of a search is \( \left(\log_{1/p} n\right)/p \), which is \( \mathcal{O}(\log n) \) when \( p \) is a constant. By choosing different values of \( p \), it is possible to trade search costs against storage costs.
**Implementation details**

The elements used for a skip list can contain more than one pointer since they can participate in more than one list.

Insertions and deletions are implemented much like the corresponding linked-list operations, except that "tall" elements must be inserted into or deleted from more than one linked list.

$O(n)$ operations, which force us to visit every node in ascending order (such as printing the entire list), provide the opportunity to perform a behind-the-scenes derandomization of the level structure of the skip-list in an optimal way, bringing the skip list to $O(\log n)$ search time. (Choose the level of the $i$'th finite node to be $1$ plus the number of times we can repeatedly divide $i$ by $2$ before it becomes odd. Also, $i=0$ for the negative infinity header as we have the usual special case of choosing the highest possible level for negative and/or positive infinite nodes.) However this also allows someone to know where all of the higher-than-level 1 nodes are and delete them.

Alternatively, we could make the level structure quasi-random in the following way:

```plaintext
make all nodes level 1
j ← 1
while the number of nodes at level j > 1 do
    for each i'th node at level j do
        if i is odd
            if i is not the last node at level j
                randomly choose whether to promote it to level j+1
            else
                do not promote
        end if
        else if i is even and node i-1 was not promoted
            promote it to level j+1
        end if
    repeat
    j ← j + 1
repeat
```

Like the derandomized version, quasi-randomization is only done when there is some other reason to be running a $O(n)$ operation (which visits every node).

The advantage of this quasi-randomness is that it doesn't give away nearly as much level-structure related information to an adversarial user as the de-randomized one. This is desirable because an adversarial user who is able to tell which nodes are not at the lowest level can pessimize performance by simply deleting higher-level nodes.

The search performance is still guaranteed to be logarithmic.

It would be tempting to make the following "optimization": In the part which says "Next, for each i'th...", forget about doing a coin-flip for each even-odd pair. Just flip a coin once to decide whether to promote only the even ones or only the odd ones. Instead of $\Theta(n \log n)$ coin flips, there would only be $\Theta(\log n)$ of them. Unfortunately, this gives the adversarial user a 50/50 chance of being correct upon guessing that all of the even numbered nodes (among the ones at level 1 or higher) are higher than level one. This is despite the property that he has a very low probability of guessing that a particular node is at level $N$ for some integer $N$.

The following proves these two claims concerning the advantages of quasi-randomness over the totally derandomized version. First, to prove that the search time is guaranteed to be logarithmic. Suppose a node $n$ is searched for, where $n$ is the position of the found node among the nodes of level 1 or higher. If $n$ is even, then there is a 50/50 chance that it is higher than level 1. However, if it is not higher than level 1 then node $n-1$ is guaranteed to be higher than level 1. If $n$ is odd, then there is a 50/50 chance that it is higher than level 1. Suppose that it is not; there is a 50/50 chance that node $n-1$ is higher than level 1. Suppose that this is not either; we are guaranteed that
node n-2 is higher than level 1. The analysis can then be repeated for nodes of level 2 or higher, level 3 or higher, etc. always keeping in mind that n is the position of the node among the ones of level k or higher for integer k. So the search time is constant in the best case (if the found node is the highest possible level) and 2 times the worst case for the search time for the totally derandomized skip-list (because we have to keep moving left twice rather than keep moving left once).

Next, an examination of the probability of an adversarial user's guess of a node being level k or higher being correct. First, the adversarial user has a 50/50 chance of correctly guessing that a particular node is level 2 or higher. This event is independent of whether or not the user correctly guesses at some other node being level 2 or higher. If the user knows the positions of two consecutive nodes of level 2 or higher, and knows that the one on the left is in an odd numbered position among the nodes of level 2 or higher, the user has a 50/50 chance of correctly guessing which one is of level 3 or higher. So, the user's probability of being correct, when guessing that a node is level 3 or higher, is 1/4. Inductively continuing this analysis, we see that the user's probability of guessing that a particular node is level k or higher is 1/(2^(k-1)).

The above analyses only work when the number of nodes is a power of two. However, because of the third rule which says, "Finally, if i is odd and also the last node at level 1 then do not promote." (where we substitute the appropriate level number for 1) it becomes a sequence of exact-power-of-two-sized skiplists, concatenated onto each other, for which the analysis does work. In fact, the exact powers of two correspond to the binary representation for the number of nodes in the whole list.

A skip list, upon which we have not recently performed either of the above mentioned Θ(n) operations, does not provide the same absolute worst-case performance guarantees as more traditional balanced tree data structures, because it is always possible (though with very low probability) that the coin-flips used to build the skip list will produce a badly balanced structure. However, they work well in practice, and the randomized balancing scheme has been argued to be easier to implement than the deterministic balancing schemes used in balanced binary search trees. Skip lists are also useful in parallel computing, where insertions can be done in different parts of the skip list in parallel without any global rebalancing of the data structure. Such parallelism can be especially advantageous for resource discovery in an ad-hoc Wireless network because a randomized skip list can be made robust to the loss of any single node.[2]

There has been some evidence that skip lists have worse real-world performance and space requirements than B trees due to memory locality and other issues.[3]

**Indexable skiplist**

As described above, a skiplist is capable of fast Θ(log n) insertion and removal of values from a sorted sequence, but it has only slow Θ(n) lookups of values at a given position in the sequence (i.e. return the 500th value); however, with a minor modification the speed of random access indexed lookups can be improved to Θ(log n).

For every link, also store the width of the link. The width is defined as the number of bottom layer links being traversed by each of the higher layer "express lane" links.

For example, here are the widths of the links in the example at the top of the page:

```
1                        10
o----> o-------------------o-------------------o-------------------o TOPE LEVEL
  1   3   2   5
o----> o-------------------o o-------------------o-------------------o LEVEL 3
  1   2   1   2   5
o----> o-------------------o o----> o-------------------o-------------------o LEVEL 2
  1   1   1   1   1   1   1   1   1   1
o----> o----> o----> o----> o----> o----> o----> o----> o----> o----> o----> o BOTTOM LEVEL
```
Notice that the width of a higher level link is the sum of the component links below it (i.e., the width 10 link spans the links of widths 3, 2 and 5 immediately below it). Consequently, the sum of all widths is the same on every level ($10 + 1 = 1 + 3 + 2 + 5 = 1 + 2 + 1 + 2 + 5$).

To index the skiplist and find the i-th value, traverse the skiplist while counting down the widths of each traversed link. Descend a level whenever the upcoming width would be too large.

For example, to find the node in the fifth position (Node 5), traverse a link of width 1 at the top level. Now four more steps are needed but the next width on this level is ten which is too large, so drop one level. Traverse one link of width 3. Since another step of width 2 would be too far, drop down to the bottom level. Now traverse the final link of width 1 to reach the target running total of 5 (1+3+1).

```plaintext
function lookupByPositionIndex(i)
    node ← head
    i ← i + 1                           # don't count the head as a step
    for level from top to bottom do
        while i ≥ node.width[level] do  # if next step is not too far
            i ← i - node.width[level]    # subtract the current width
            node ← node.next[level]      # traverse forward at the current level
        repeat
        repeat
        return node.value
    end function
```

This method of implementing indexing is detailed in Section 3.4 Linear List Operations in "A skiplist cookbook" by William Pugh [4].

Also, see Running Median using an Indexable Skiplist [5] for a complete implementation written in Python and for an example of it being used to solve a computationally intensive statistics problem. And see Regaining Lost Knowledge [6] for the history of that solution.

**History**

Skip lists were first described in 1990 by William Pugh. He details how they work in *Skip lists: a probabilistic alternative to balanced trees* in Communications of the ACM, June 1990, 33(6) 668-676. See also citations [7] and downloadable documents [8].

To quote the author:

*Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.*
Usages

List of applications and frameworks that use skip lists:

- QMap[9] template class of Qt that provides a dictionary.
- Redis[10] is an ANSI-C open-source persistent key/value store for Posix systems, uses skip lists in its implementation of ordered sets.
- Running Median using an Indexable Skiplist[5] is a Python implementation of a skiplist augmented by link widths to make the structure indexable (giving fast access to the nth item). The indexable skiplist is used to efficiently solve the running median problem (recomputing medians and quartiles as values are added and removed from a large sliding window).
- ConcurrentSkipListSet[12] and ConcurrentSkipListMap[13] in the Java 1.6 API.

Skip lists are also used in distributed applications (where the nodes represent physical computers, and pointers represent network connections) and for implementing highly scalable concurrent priority queues with less lock contention,[14] or even without locking,[15][16][17] as well lockless concurrent dictionaries.[18] There are also several US patents for using skip lists to implement (lockless) priority queues and concurrent dictionaries.

References

[14] Skiplist-based concurrent priority queues (http://dx.doi.org/10.1109/IPDPS.2000.845994)
External links


- "Skip list" entry (http://nist.gov/dads/HTML/skiplist.html) in the Dictionary of Algorithms and Data Structures
- SkipDB, a BerkeleyDB-style database implemented using skip lists. (http://dekte.com/projects/opensource/SkipDB/)
- Skip Lists lecture (MIT OpenCourseWare: Introduction to Algorithm) (http://videolectures.net/mit6046jf05_demaine_lec12/)

Demo applets

- Skip List Applet (http://people.ksp.sk/~kuko/bak/index.html) by Kubo Kovac
- Thomas Wenger's demo applet on skiplists (http://iamwww.unibe.ch/~wenger/DA/SkipList/)

Implementations

- A generic Skip List in C++ (http://codingplayground.blogspot.com/2009/01/generic-skip-list-skiplist.html) by Antonio Gulli
- Algorithm::SkipList, implementation in Perl on CPAN (http://search.cpan.org/~rrwo/Algorithm-SkipList-1.02/)
- John Shipman's implementation in Python (http://infohost.nmt.edu/tcc/help/lang/python/examples/pyskip/)
- A Lua port of John Shipman's Python version (http://love2d.org/wiki/Skip_list)
In computer science, a **B-tree** is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. (Comer 1979, p. 123) Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and filesystems.

### Overview

In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

Each internal node of a B-tree will contain a number of keys. Usually, the number of keys is chosen to vary between $d$ and $2d$. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has $2d$ keys, then adding a key to that node can be accomplished by splitting the $2d$ key node into two $d$ key nodes and adding the key to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbor each have $d$ keys, then a key may be deleted from the internal node by combining with its neighbor. Deleting the key would make the internal node have $d - 1$ keys; joining the neighbor would add $d$ keys plus one more key brought down from the neighbor's parent. The result is an entirely full node of $2d$ keys.

The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-tree is sometimes described with the parameters $(d + 1) - (2d + 1)$ or simply with the highest
branching order. \((2d + 1)\).

A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node further away from the root.

B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes, because then the cost of accessing the node may be amortized over multiple operations within the node. This usually occurs when the nodes are in secondary storage such as disk drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases and the number of expensive node accesses is reduced. In addition, rebalancing the tree occurs less often. The maximum number of child nodes depends on the information that must be stored for each child node and the size of a full disk block or an analogous size in secondary storage. While 2-3 B-trees are easier to explain, practical B-trees using secondary storage want a large number of child nodes to improve performance.

**Variants**

The term B-tree may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B⁺-tree and the B⁻-tree.

- In the B⁺-tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access. (Comer 1979, p. 129)
- The B⁻-tree balances more neighboring internal nodes to keep the internal nodes more densely packed. (Comer 1979, p. 129) This variant requires non-root nodes to be at least 2/3 full instead of 1/2. (Knuth 1973, p. 478) To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it. When both nodes are full, then the two nodes are split into three.
- Counted B-trees store, with each pointer within the tree, the number of nodes in the subtree below that pointer.\(^1\) This allows rapid searches for the Nth record in key order, or counting the number of records between any two records, and various other related operations.

**Etymology unknown**

Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972), but they did not explain what, if anything, the B stands for. Douglas Comer explains:

> The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees. (Comer 1979, p. 123 footnote 1)

Donald Knuth speculates on the etymology of B-trees in his May, 1980 lecture on the topic "CS144C classroom lecture about disk storage and B-trees", suggesting the "B" may have originated from Boeing or from Bayer's name.\(^2\)
The database problem

Time to search a sorted file

Usually, sorting and searching algorithms have been characterized by the number of comparison operations that must be performed using order notation. A binary search of a sorted table with $N$ records, for example, can be done in $O(\log_2 N)$ comparisons. If the table had 1,000,000 records, then a specific record could be located with about 20 comparisons: $\log_2 1,000,000 = 19.931\ldots$.

Large databases have historically been kept on disk drives. The time to read a record on a disk drive can dominate the time needed to compare keys once the record is available. The time to read a record from a disk drive involves a seek time and a rotational delay. The seek time may be 0 to 20 or more milliseconds, and the rotational delay averages about half the rotation period. For a 7200 RPM drive, the rotation period is 8.33 milliseconds. For a drive such as the Seagate ST3500320NS, the track-to-track seek time is 0.8 milliseconds and the average reading seek time is 8.5 milliseconds.\[^3\] For simplicity, assume reading from disk takes about 10 milliseconds.

Naively, then, the time to locate one record out of a million would take 20 disk reads times 10 milliseconds per disk read, which is 0.2 seconds.

The time won't be that bad because individual records are grouped together in a disk block. A disk block might be 16 kilobytes. If each record is 160 bytes, then 100 records could be stored in each block. The disk read time above was actually for an entire block. Once the disk head is in position, one or more disk blocks can be read with little delay. With 100 records per block, the last 6 or so comparisons don't need to do any disk reads—the comparisons are all within the last disk block read.

To speed the search further, the first 13 to 14 comparisons (which each required a disk access) must be sped up.

An index speeds the search

A significant improvement can be made with an index. In the example above, initial disk reads narrowed the search range by a factor of two. That can be improved substantially by creating an auxiliary index that contains the first record in each disk block (sometimes called a sparse index). This auxiliary index would be 1% of the size of the original database, but it can be searched more quickly. Finding an entry in the auxiliary index would tell us which block to search in the main database; after searching the auxiliary index, we would have to search only that one block of the main database—at a cost of one more disk read. The index would hold 10,000 entries, so it would take at most 14 comparisons. Like the main database, the last 6 or so comparisons in the aux index would be on the same disk block. The index could be searched in about 8 disk reads, and the desired record could be accessed in 9 disk reads.

The trick of creating an auxiliary index can be repeated to make an auxiliary index to the auxiliary index. That would make an aux-aux index that would need only 100 entries and would fit in one disk block.

Instead of reading 14 disk blocks to find the desired record, we only need to read 3 blocks. Reading and searching the first (and only) block of the aux-aux index identifies the relevant block in aux-index. Reading and searching that aux-index block identifies the relevant block in the main database. Instead of 150 milliseconds, we need only 30 milliseconds to get the record.

The auxiliary indices have turned the search problem from a binary search requiring roughly $\log_2 N$ disk reads to one requiring only $\log_b N$ disk reads where $b$ is the blocking factor (the number of entries per block): $b = 100$ entries per block; $\log_b 1,000,000 = 3 \text{(reads)}$.

In practice, if the main database is being frequently searched, the aux-aux index and much of the aux index may reside in a disk cache, so they would not incur a disk read.
**Insertions and deletions cause trouble**

If the database does not change, then compiling the index is simple to do, and the index need never be changed. If there are changes, then managing the database and its index becomes more complicated.

Deleting records from a database doesn't cause much trouble. The index can stay the same, and the record can just be marked as deleted. The database stays in sorted order. If there are a lot of deletions, then the searching and storage become less efficient.

Insertions are a disaster in a sorted sequential file because room for the inserted record must be made. Inserting a record before the first record in the file requires shifting all of the records down one. Such an operation is just too expensive to be practical.

A trick is to leave some space lying around to be used for insertions. Instead of densely storing all the records in a block, the block can have some free space to allow for subsequent insertions. Those records would be marked as if they were "deleted" records.

Now, both insertions and deletions are fast as long as space is available on a block. If an insertion won't fit on the block, then some free space on some nearby block must be found and the auxiliary indices adjusted. The hope is enough space is nearby that a lot of blocks do not need to be reorganized. Alternatively, some out-of-sequence disk blocks may be used.

**The B-tree uses all those ideas**

The B-tree uses all of the above ideas:

- It keeps records in sorted order for sequential traversing
- It uses a hierarchical index to minimize the number of disk reads
- It uses partially-full blocks to speed insertions and deletions
- The index is elegantly adjusted with a recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least ½ full. A B-tree can handle an arbitrary number of insertions and deletions.

**Technical description**

**Terminology**

The terminology used for B-trees is inconsistent in the literature:

Unfortunately, the literature on B-trees is not uniform in its use of terms relating to B-Trees. (Folk & Zoellick 1992, p. 362)

Bayer & McCreight (1972), Comer (1979), and others define the **order** of B-tree as the minimum number of keys in a non-root node. Folk & Zoellick (1992) points out that terminology is ambiguous because the maximum number of keys is not clear. An order 3 B-tree might hold a maximum of 6 keys or a maximum of 7 keys. Knuth (1993b) avoids the problem by defining the **order** to be maximum number of children (which is one more than the maximum number of keys).

The term **leaf** is also inconsistent. Bayer & McCreight (1972) considered the leaf level to be the lowest level of keys, but Knuth (1993b) considered the leaf level to be one level below the lowest keys. (Folk & Zoellick 1992, p. 363)

There are many possible implementation choices. In some designs, the leaves may hold the entire data record; in other designs, the leaves may only hold pointers to the data record. Those choices are not fundamental to the idea of a B-tree. [4]

There are also unfortunate choices like using the variable $k$ to represent the number of children when $k$ could be confused with the number of keys.
For simplicity, most authors assume there are a fixed number of keys that fit in a node. The basic assumption is the key size is fixed and the node size is fixed. In practice, variable length keys may be employed. (Folk & Zoellick 1992, p. 379)

**Definition**

According to Knuth's definition, a B-tree of order \( m \) (the maximum number of children for each node) is a tree which satisfies the following properties:

1. Every node has at most \( m \) children.
2. Every node (except root) has at least \( m/2 \) children.
3. The root has at least two children if it is not a leaf node.
4. All leaves appear in the same level, and carry information.
5. A non-leaf node with \( k \) children contains \( k-1 \) keys.

Each internal node’s elements act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 separation values or elements: \( a_1 \) and \( a_2 \). All values in the leftmost subtree will be less than \( a_1 \), all values in the middle subtree will be between \( a_1 \) and \( a_2 \), and all values in the rightmost subtree will be greater than \( a_2 \).

**Internal nodes**

Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a maximum of \( U \) children and a minimum of \( L \) children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between \( L-1 \) and \( U-1 \)). \( U \) must be either \( 2L \) or \( 2L-1 \); therefore each internal node is at least half full. The relationship between \( U \) and \( L \) implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there’s room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

**The root node**

The root node’s number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than \( L-1 \) elements in the entire tree, the root will be the only node in the tree, with no children at all.

**Leaf nodes**

Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers. A B-tree of depth \( n+1 \) can hold about \( U \) times as many items as a B-tree of depth \( n \), but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

Some balanced trees store values only at leaf nodes, and use different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree, and may use the same structure for all nodes. However, since leaf nodes never have children, the B-trees benefit from improved performance if they use a specialized structure.
**Best case and worst case heights**

The best case height of a B-Tree is:

\[ \log_m n. \]

The worst case height of a B-Tree is:

\[ \log_{m/2} n \]

where \( m \) is the maximum number of children a node can have.

**Algorithms**

**Warning**: the discussion below uses "element", "value", "key", "separator", and "separation value" to mean essentially the same thing. The terms are not clearly defined. There are some subtle issues at the root and leaves.

**Search**

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value.

Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.
**Insertion**

All insertions start at a leaf node. To insert a new element:

1. Search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:
   1. If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
   2. Otherwise, the node is full, evenly split it into two nodes so:
      1. A single median is chosen from among the leaf's elements and the new element.
      2. Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
      3. The separation value is inserted in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is $U-1$. When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number $U-1$ of elements into two legal nodes. If this number is odd, then $U=2L$ and one of the new nodes contains $(U-2)/2 = L-1$ elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If $U-1$ is even, then $U=2L-1$, so there are $2L-2$ elements in the node. Half of this number is $L-1$, which is the minimum number of elements allowed per node.

An improved algorithm supports a single pass down the tree from the root to the node where the insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage. However, to use this improved algorithm, we must be able to send one element to the parent and split the remaining $U-2$ elements into two legal nodes, without adding a new element. This requires $U = 2L$ rather than $U = 2L-1$, which accounts for why some textbooks impose this requirement in defining B-trees.

**Deletion**

There are two popular strategies for deletion from a B-Tree:

1. Locate and delete the item, then restructure the tree to regain its invariants, OR
2. Do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring.

The algorithm below uses the former strategy.

There are two special cases to consider when deleting an element:

1. The element in an internal node may be a separator for its child nodes
2. Deleting an element may put its node under the minimum number of elements and children.
The procedures for these cases are in order below.

**Deletion from a leaf node**
1. Search for the value to delete
2. If the value's in a leaf node, simply delete it from the node
3. If underflow happens, check siblings, and either transfer a key or fuse the siblings together
4. If deletion happened from right child, retrieve the max value of left child if it has no underflow
5. In vice-versa situation, retrieve the min element from right

**Deletion from an internal node**
Each element in an internal node acts as a separation value for two subtrees, and when such an element is deleted, two cases arise.

In the first case, both of the two child nodes to the left and right of the deleted element have the minimum number of elements, namely $L-1$. They can then be joined into a single node with $2L-2$ elements, a number which does not exceed $U-1$ and so is a legal node. Unless it is known that this particular B-tree does not contain duplicate data, we must then also (recursively) delete the element in question from the new node.

In the second case, one of the two child nodes contains more than the minimum number of elements. Then a new separator for those subtrees must be found. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is the smallest element which is still greater than the separator. Both of those elements are in leaf nodes, and either can be the new separator for the two subtrees.

1. If the value is in an internal node, choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator
2. This has deleted an element from a leaf node, and so is now equivalent to the previous case

**Rebalancing after deletion**
If deleting an element from a leaf node has brought it under the minimum size, some elements must be redistributed to bring all nodes up to the minimum. In some cases the rearrangement will move the deficiency to the parent, and the redistribution must be applied iteratively up the tree, perhaps even to the root. Since the minimum element count doesn't apply to the root, making the root be the only deficient node is not a problem. The algorithm to rebalance the tree is as follows:

1. If the right sibling has more than the minimum number of elements
   1. Add the separator to the end of the deficient node
   2. Replace the separator in the parent with the first element of the right sibling
   3. Append the first child of the right sibling as the last child of the deficient node
2. Otherwise, if the left sibling has more than the minimum number of elements
   1. Add the separator to the start of the deficient node
   2. Replace the separator in the parent with the last element of the left sibling
   3. Insert the last child of the left sibling as the first child of the deficient node
3. If both immediate siblings have only the minimum number of elements
   1. Create a new node with all the elements from the deficient node, all the elements from one of its siblings, and the separator in the parent between the two combined sibling nodes
   2. Remove the separator from the parent, and replace the two children it separated with the combined node
   3. If that brings the number of elements in the parent under the minimum, repeat these steps with that deficient node, unless it is the root, since the root is permitted to be deficient
The only other case to account for is when the root has no elements and one child. In this case it is sufficient to replace it with its only child.

**Initial construction**

In applications, it is frequently useful to build a B-tree to represent a large existing collection of data and then update it incrementally using standard B-tree operations. In this case, the most efficient way to construct the initial B-tree is not to insert every element in the initial collection successively, but instead to construct the initial set of leaf nodes directly from the input, then build the internal nodes from these. This approach to B-tree construction is called bulkloading. Initially, every leaf but the last one has one extra element, which will be used to build the internal nodes.

For example, if the leaf nodes have maximum size 4 and the initial collection is the integers 1 through 24, we would initially construct 4 leaf nodes containing 5 values each and 1 which contains 4 values:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

We build the next level up from the leaves by taking the last element from each leaf node except the last one. Again, each node except the last will contain one extra value. In the example, suppose the internal nodes contain at most 2 values (3 child pointers). Then the next level up of internal nodes would be:

```
5 10 15 20
```

```
1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19 21 22 23 24
```

This process is continued until we reach a level with only one node and it is not overfilled. In the example only the root level remains:

```
15
```

```
5 10 20
```

```
1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19 21 22 23 24
```
In filesystems

In addition to its use in databases, the B-tree is also used in filesystems to allow quick random access to an arbitrary block in a particular file. The basic problem is turning the file block \( i \) address into a disk block (or perhaps to a cylinder-head-sector) address.

Some operating systems require the user to allocate the maximum size of the file when the file is created. The file can then be allocated as contiguous disk blocks. Converting to a disk block: the operating system just adds the file block address to the starting disk block of the file. The scheme is simple, but the file cannot exceed its created size.

Other operating systems allow a file to grow. The resulting disk blocks may not be contiguous, so mapping logical blocks to physical blocks is more involved.

MS-DOS, for example, used a simple File Allocation Table (FAT). The FAT has an entry for each disk block, and that entry identifies whether its block is used by a file and if so, which block (if any) is the next disk block of the same file. So, the allocation of each file is represented as a linked list in the table. In order to find the disk address of file block \( i \), the operating system (or disk utility) must sequentially follow the file's linked list in the FAT. Worse, to find a free disk block, it must sequentially scan the FAT. For MS-DOS, that was not a huge penalty because the disks and files were small and the FAT had few entries and relatively short file chains. In the FAT12 filesystem (used on floppy disks and early hard disks), there were no more than 4,080 entries, and the FAT would usually be resident in memory. As disks got bigger, the FAT architecture began to confront penalties. On a large disk using FAT, it may be necessary to perform disk reads to learn the disk location of a file block to be read or written.

TOPS-20 (and possibly TENEX) used a 0 to 2 level tree that has similarities to a B-Tree. A disk block was 512 36-bit words. If the file fit in a 512 (\( 2^9 \)) word block, then the file directory would point to that physical disk block. If the file fit in \( 2^{18} \) words, then the directory would point to an aux index; the 512 words of that index would either be NULL (the block isn't allocated) or point to the physical address of the block. If the file fit in \( 2^{27} \) words, then the directory would point to a block holding an aux-aux index; each entry would either be NULL or point to an aux index. Consequently, the physical disk block for a \( 2^{27} \) word file could be located in two disk reads and read on the third.

Apple's filesystem HFS+, Microsoft's NTFS and some Linux filesystems, such as btrfs and Ext4, use B-trees. B*-trees are used in the HFS and Reiser4 file systems.

Variations

Access concurrency

Lehman and Yao showed that all read locks could be avoided (and thus concurrent access greatly improved) by linking the tree blocks at each level together with a "next" pointer. This results in a tree structure where both insertion and search operations descend from the root to the leaf. Write locks are only required as a tree block is modified. This maximizes access concurrency by multiple users, an important consideration for databases and/or other B-Tree based ISAM storage methods. The cost associated with this improvement is that empty pages cannot be removed from the btree during normal operations. (However, see for various strategies to implement node merging, and source code at.)

United States Patent 5283894, granted In 1994, appears to show a way to use a 'Meta Access Method' to allow concurrent B+Tree access and modification without locks. The technique accesses the tree 'upwards' for both searches and updates by means of additional in-memory indexes that point at the blocks in each level in the block cache. No reorganization for deletes is needed and there are no 'next' pointers in each block as in Lehman and Yao.
Notes

[4] Bayer & McCreight (1972) avoided the issue by saying an index element is a (physically adjacent) pair of \((x, a)\) where \(x\) is the key, and \(a\) is some associated information. The associated information might be a pointer to a record or records in a random access, but what it was didn't really matter. Bayer & McCreight (1972) states, "For this paper the associated information is of no further interest."
[5] For FAT, what is called a "disk block" here is what the FAT documentation calls a "cluster", which is fixed-size group of one or more contiguous whole physical disk sectors. For the purposes of this discussion, a cluster has no significant difference from a physical sector.
[6] Two of these were reserved for special purposes, so only 4078 could actually represent disk blocks (clusters).

References

• Folk, Michael J.; Zoellick, Bill (1992), File Structures (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4

Original papers


External links

• B-Tree animation applet (http://slady.net/java/bt/view.php) by slady
• B-tree and UB-tree on Scholarpedia (http://www.scholarpedia.org/article/B-tree_and UB-tree) Curator: Dr Rudolf Bayer
• B-Trees: Balanced Tree Data Structures (http://www.blueerwhite.org/btree)
• B-Tree Tutorial (http://cis.stvincent.edu/html/tutorials/swd/btree/btree.html)
• The InfinityDB BTree implementation (http://www.boilerbay.com/infinitydb/TheDesignOfTheInfinityDatabaseEngine.htm)
• Cache Oblivious B(+)-trees (http://supertech.csail.mit.edu/CacheObliviousBTree.html)
• Dictionary of Algorithms and Data Structures entry for B*-tree (http://www.nist.gov/dads/HTML/bstartree.html)
In computer science, a B+ tree or B plus tree is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a key. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment (usually called a "block" or "node"). In a B+ tree, in contrast to a B-tree, all records are stored at the leaf level of the tree; only keys are stored in interior nodes.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context—in particular, file systems. This is primarily because unlike binary search trees, B+ trees have very high fanout (typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.


**Overview**

The order, or branching factor $b$ of a B+ tree measures the capacity of nodes (i.e. the number of children nodes) for internal nodes in the tree. The actual number of children for a node, referred to here as $m$, is constrained for internal nodes so that $\lceil b/2 \rceil \leq m \leq b$. The root is an exception: it is allowed to have as few as two children. For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children, but are constrained so that the number of keys must be at least $\lceil b/2 \rceil$ and at most $b - 1$. In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node. (The root is also the single leaf, in this case.) This node is permitted to have as little as one key if necessary, and at most $b$.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Children Type</th>
<th>Min Children</th>
<th>Max Children</th>
<th>Example $b = 7$</th>
<th>Example $b = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Node (when it is the only node in the tree)</td>
<td>Keys</td>
<td>1</td>
<td>$b$</td>
<td>1 - 7</td>
<td>1 - 100</td>
</tr>
<tr>
<td>Root Node</td>
<td>Internal Nodes or Leaf Nodes</td>
<td>2</td>
<td>$b$</td>
<td>2 - 7</td>
<td>2 - 100</td>
</tr>
<tr>
<td>Internal Node</td>
<td>Internal Nodes or Leaf Nodes</td>
<td>$\lceil b/2 \rceil$</td>
<td>$b$</td>
<td>4 - 7</td>
<td>50 - 100</td>
</tr>
<tr>
<td>Leaf Node</td>
<td>Keys</td>
<td>$\lceil b/2 \rceil$</td>
<td>$b - 1$</td>
<td>3 - 6</td>
<td>50 - 99</td>
</tr>
</tbody>
</table>
Algorithms

Search
The algorithm to perform a search for a record r follows pointers to the correct child of each node until a leaf is reached. Then, the leaf is scanned until the correct record is found (or until failure).

```
function search(record r)
    u := root
    while (u is not a leaf) do
        choose the correct pointer in the node
        move to the first node following the pointer
        u := current node
    scan u for r
```

This pseudocode assumes that no repetition is allowed.

Insertion
Perform a search to determine what bucket the new record should go into.

• If the bucket is not full, add the record.
• Otherwise, split the bucket.
  • Allocate new leaf and move half the bucket's elements to the new bucket.
  • Insert the new leaf's smallest key and address into the parent.
  • If the parent is full, split it too.
    • Add the middle key to the parent node.
  • Repeat until a parent is found that need not split.
• If the root splits, create a new root which has one key and two pointers.

[Code required]

Deletion
• Start at root, find leaf L where entry belongs.
• Remove the entry.
  • If L is at least half-full, done!
  • If L has fewer entries than it should,
    • Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    • If re-distribution fails, merge L and sibling.
• If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
• Merge could propagate to root, decreasing height.
Characteristics

For a $b$-order B+ tree with $h$ levels of index:

- The maximum number of records stored is $n_{\text{max}} = b^h - b^{h-1}$
- The minimum number of records stored is $n_{\text{min}} = 2 \left\lceil \frac{b}{2} \right\rceil^{h-1}$
- The minimum number of keys is $n_{k\text{min}} = 2 \left\lceil \frac{b}{2} \right\rceil^h - 1$
- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations
- Finding a record requires $O(\log_b n)$ operations
- Removing a (previously located) record requires $O(\log_b n)$ operations
- Performing a range query with $k$ elements occurring within the range requires $O(\log_b n + k)$ operations
- Performing a pagination query with page size $s$ and page number $p$ requires $O(p \times s)$ operations

Implementation

The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. This illustrates one of the significant advantages of a B+-tree over a B-tree; in a B-tree, since not all keys are present in the leaves, such an ordered linked list cannot be constructed. A B+-tree is thus particularly useful as a database system index, where the data typically resides on disk, as it allows the B+-tree to actually provide an efficient structure for housing the data itself (this is described in [9] as index structure "Alternative 1").

If a storage system has a block size of $B$ bytes, and the keys to be stored have a size of $k$, arguably the most efficient B+ tree is one where $b = (B/k) - 1$. Although theoretically the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable.

If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

B+ trees can also be used for data stored in RAM. In this case a reasonable choice for block size would be the size of processor's cache line. However, some studies have proved that a block size a few times larger than the processor's cache line can deliver better performance if cache prefetching is used.

Space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers. For string keys, space can be saved by using the following technique: Normally the $i$th entry of an internal block contains the first key of block $i+1$. Instead of storing the full key, we could store the shortest prefix of the first key of block $i+1$ that is strictly greater (in lexicographic order) than last key of block $i$. There is also a simple way to compress pointers: if we suppose that some consecutive blocks $i, i+1...i+k$ are stored contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

All the above compression techniques have some drawbacks. First, a full block must be decompressed to extract a single element. One technique to overcome this problem is to divide each block into sub-blocks and compress them separately. In this case searching or inserting an element will only need to decompress or compress a sub-block instead of a full block. Another drawback of compression techniques is that the number of stored elements may vary
considerably from a block to another depending on how well the elements are compressed inside each block.

**History**

The B tree was first described in the paper *Organization and Maintenance of Large Ordered Indices. Acta Informatica* 1: 173–189 (1972) by Rudolf Bayer and Edward M. McCreight. There is no single paper introducing the B+ tree concept. Instead, the notion of maintaining all data in leaf nodes is repeatedly brought up as an interesting variant. An early survey of B trees also covering B+ trees is Douglas Comer: "The Ubiquitous B-Tree*[^10]*", ACM Computing Surveys 11(2): 121–137 (1979). Comer notes that the B+ tree was used in IBM's VSAM data access software and he refers to an IBM published article from 1973.

**References**


**External links**

- B+ tree in Python, used to implement a list (http://pypi.python.org/pypi/blist)
- Dr. Monge's B+ Tree index notes (http://www.eecs.csulb.edu/~monge/classes/share/B+TreeIndexes.html)
- Effect of node size on the performance of cache conscious B+-trees (http://www.eecs.umich.edu/~jignesh/quickstep/publ/ci.pdf)
- Fractal Prefetching B+-trees (http://www.pittsburgh.intel-research.net/people/gibbons/papers/fpbptrees.pdf)
- Towards pB+-trees in the field: implementations Choices and performance (http://gemo.futurs.inria.fr/events/EXPDB2006/PAPERS/Jonsson.pdf)
- Cache-Conscious Index Structures for Main-Memory Databases (https://oa.doria.fi/bitstream/handle/10024/2906/cachecon.pdf?sequence=1)
- Cache Oblivious B(+-)trees (http://supertech.csail.mit.edu/cacheObliviousBTree.html)
- The Power of B-Trees: CouchDB B+ Tree Implementation (http://books.couchdb.org/relax/appendix/btrees)
Implementations

- Interactive B+ Tree Implementation in C (http://www.amittai.com/prose/bplustree.html)
- Memory based B+ tree implementation as C++ template library (http://idlebox.net/2007/stx-btree/)
- Stream based B+ tree implementation as C++ template library (http://gitorious.org/bp-tree/main)
- Open Source C++ B+ Tree Implementation (http://www.scalingweb.com/bplus_tree.php)
- Open Source Javascript B+ Tree Implementation (http://blog.conquex.com/?p=84)
- Perl implementation of B+ trees (http://search.cpan.org/~hanenkamp/Tree-BPTree-1.07)
- Java/C#/Python implementations of B+ trees (http://bplusdotnet.sourceforge.net)
- File based B+Tree in C# with threading and MVCC support (http://csharptest.net/?page_id=563)
Integer and string searching

Trie

In computer science, a **trie**, or **prefix tree**, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key it is associated with. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.

The term trie comes from retrieval. Following the etymology, the inventor, Edward Fredkin, pronounces it English pronunciation: /ˈtriː/ "tree". However, it is pronounced English pronunciation: /ˈtraɪ/ "try" by other authors.

In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the trie works.)

Though it is most common, tries need not be keyed by character strings. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a **bitwise trie** is keyed on the individual bits making up a short, fixed size of bits such as an integer number or pointer to memory.
Advantages relative to other search algorithms

A series of graphs showing how different algorithms scale with number of items

Unlike most other algorithms, tries have the peculiar feature that the time to insert, or to delete or to find is almost identical because the code paths followed for each are almost identical. As a result, for situations where code is inserting, deleting and finding in equal measure tries can handily beat binary search trees or even hash tables, as well as being better for the CPU's instruction and branch caches.

The following are the main advantages of tries over binary search trees (BSTs):

- Looking up keys is faster. Looking up a key of length \( m \) takes worst case \( O(m) \) time. A BST performs \( O(\log n) \) comparisons of keys, where \( n \) is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys if the tree is balanced. Hence in the worst case, a BST takes \( O(m \log n) \) time. Moreover, in the worst case \( \log n \) will approach \( m \). Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines.
- Tries are more space efficient when they contain a large number of short keys, because nodes are shared between keys with common initial subsequences.
- Tries facilitate longest-prefix matching, helping to find the key sharing the longest possible prefix of characters all unique.
- The number of internal nodes from root to leaf equals the length of the key. Balancing the tree is therefore no concern.

The following are the main advantages of tries over hash tables:

- Tries support ordered iteration, whereas iteration over a hash table will result in a pseudorandom order given by the hash function (also, the order of hash collisions is implementation defined), which is usually meaningless.
- Tries facilitate longest-prefix matching, but hashing does not, as a consequence of the above. Performing such a "closest fit" find can, depending on implementation, be as quick as an exact find.
- Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their index when it becomes full - a very expensive operation. Tries therefore have much better bounded worst case time costs, which is important for latency sensitive programs.
- By avoiding the hash function, tries are generally faster than hash tables for small keys like integers and pointers.
Applications

As replacement of other data structures

As mentioned, a trie has a number of advantages over binary search trees. A trie can also be used to replace a hash table, over which it has the following advantages:

- Looking up data in a trie is faster in the worst case, O(m) time, compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is O(N) time, but far more typically is O(1), with O(m) time spent evaluating the hash.
- There are no collisions of different keys in a trie.
- Buckets in a trie which are analogous to hash table buckets that store key collisions are necessary only if a single key is associated with more than one value.
- There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
- A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

- Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random-access time is high compared to main memory.
- Some keys, such as floating point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless a bitwise trie can handle standard IEEE single and double format floating point numbers.

Dictionary representation

A common application of a trie is storing a dictionary, such as one found on a mobile telephone. Such applications take advantage of a trie’s ability to quickly search for, insert, and delete entries; however, if storing dictionary words is all that is required (i.e. storage of information auxiliary to each word is not required), a minimal acyclic deterministic finite automaton would use less space than a trie. This is because an acyclic deterministic finite automaton can compress identical branches from the trie which correspond to the same suffixes (or parts) of different words being stored.

Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking and hyphenation software.

Algorithms

We can describe trie lookup (and membership) easily. Given a recursive trie type, storing an optional value at each node, and a list of children tries, indexed by the next character, (here, represented as a Haskell data type):

```haskell
data Trie a =
    Trie { value :: Maybe a,
           children :: [(Char, Trie a)] }
```

We can lookup a value in the trie as follows:

```haskell
find :: String -> Trie a -> Maybe a
find [] t = value t
find (k:ks) t = case lookup k (children t) of
    Nothing -> Nothing
    Just t' -> find ks t'
```
In an imperative style, and assuming an appropriate data type in place, we can describe the same algorithm in Python (here, specifically for testing membership). Note that `children` is map of a node's children; and we say that a "terminal" node is one which contains a valid word.

```python
def find(node, key):
    for char in key:
        if char not in node.children:
            return None
        else:
            node = node.children[char]
    return node.value
```

A simple Ruby version:

```ruby
class Trie
    def initialize()
        @trie = Hash.new()
    end

    def build(str)
        node = @trie
        str.each_char { |ch|
            cur = ch
            prev_node = node
            node = node[cur]
            if node == nil
                prev_node[cur] = Hash.new()
                node = prev_node[cur]
            end
        }
    end

    def find(str)
        node = @trie
        str.each_char { |ch|
            cur = ch
            node = node[cur]
            if node == nil
                return false
            end
        }
        return true
    end
end
```
Sorting
Lexicographic sorting of a set of keys can be accomplished with a simple trie-based algorithm as follows:

- Insert all keys in a trie.
- Output all keys in the trie by means of pre-order traversal, which results in output that is in lexicographically increasing order. Pre-order traversal is a kind of depth-first traversal. In-order traversal is another kind of depth-first traversal that is more appropriate for outputting the values that are in a binary search tree rather than a trie.

This algorithm is a form of radix sort.

A trie forms the fundamental data structure of Burstsort, currently (2007) the fastest known, memory/cache-based, string sorting algorithm.[6]

A parallel algorithm for sorting N keys based on tries is O(1) if there are N processors and the lengths of the keys have a constant upper bound. There is the potential that the keys might collide by having common prefixes or by being identical to one another, reducing or eliminating the speed advantage of having multiple processors operating in parallel.

Full text search
A special kind of trie, called a suffix tree, can be used to index all suffixes in a text in order to carry out fast full text searches.

Bitwise tries
Bitwise tries are much the same as a normal character based trie except that individual bits are used to traverse what effectively becomes a form of binary tree. Generally, implementations use a special CPU instruction to very quickly find the first set bit in a fixed length key (e.g. GCC’s __builtin_clz() intrinsic). This value is then used to index a 32 or 64 entry table which points to the first item in the bitwise trie with that number of leading zero bits. The search then proceeds by testing each subsequent bit in the key and choosing child[0] or child[1] appropriately until the item is found.

Although this process might sound slow, it is very cache-local and highly parallelizable due to the lack of register dependencies and therefore in fact performs excellently on modern out-of-order execution CPUs. A red-black tree for example performs much better on paper, but is highly cache-unfriendly and causes multiple pipeline and TLB stalls on modern CPUs which makes that algorithm bound by memory latency rather than CPU speed. In comparison, a bitwise trie rarely accesses memory and when it does it does so only to read, thus avoiding SMP cache coherency overhead, and hence is becoming increasingly the algorithm of choice for code which does a lot of insertions and deletions such as memory allocators (e.g. recent versions of the famous Doug Lea’s allocator (dlmalloc) and its descendents).

A reference implementation of bitwise tries in C and C++ useful for further study can be found at http://www.nedprod.com/programs/portable/nedtries/.
Compressing tries

When the trie is mostly static, i.e. all insertions or deletions of keys from a prefilled trie are disabled and only lookups are needed, and when the trie nodes are not keyed by node specific data (or if the node's data is common) it is possible to compress the trie representation by merging the common branches.[7] This application is typically used for compressing lookup tables when the total set of stored keys is very sparse within their representation space.

For example it may be used to represent sparse bitsets (i.e. subsets of a much fixed enumerable larger set) using a trie keyed by the bit element position within the full set, with the key created from the string of bits needed to encode the integral position of each element. The trie will then have a very degenerate form with many missing branches, and compression becomes possible by storing the leaf nodes (set segments with fixed length) and combining them after detecting the repetition of common patterns or by filling the unused gaps.

Such compression is also typically used in the implementation of the various fast lookup tables needed to retrieve Unicode character properties (for example to represent case mapping tables, or lookup tables containing the combination of base and combining characters needed to support Unicode normalization). For such application, the representation is similar to transforming a very large unidimensional sparse table into a multidimensional matrix, and then using the coordinates in the hyper-matrix as the string key of an uncompressed trie. The compression will then consist of detecting and merging the common columns within the hyper-matrix to compress the last dimension in the key; each dimension of the hypermatrix stores the start position within a storage vector of the next dimension for each coordinate value, and the resulting vector is itself compressible when it is also sparse, so each dimension (associated to a layer level in the trie) is compressed separately.

Some implementations do support such data compression within dynamic sparse tries and allow insertions and deletions in compressed tries, but generally this has a significant cost when compressed segments need to be split or merged, and some tradeoff has to be made between the smallest size of the compressed trie and the speed of updates, by limiting the range of global lookups for comparing the common branches in the sparse trie.

The result of such compression may look similar to trying to transform the trie into a directed acyclic graph (DAG), because the reverse transform from a DAG to a trie is obvious and always possible, however it is constrained by the form of the key chosen to index the nodes.

Another compression approach is to "unravel" the data structure into a single byte array.[8] This approach eliminates the need for node pointers which reduces the memory requirements substantially and makes memory mapping possible which allows the virtual memory manager to load the data into memory very efficiently.

Another compression approach is to "pack" the trie.[2] Liang describes a space-efficient implementation of a sparse packed trie applied to hyphenation, in which the descendants of each node may be interleaved in memory.

External links

- NIST's Dictionary of Algorithms and Data Structures: Trie[9]
- Trie implementation and visualisation in flash[10]
- Using Tries[12] Topcoder tutorial
- An Implementation of Double-Array Trie[13]
- de la Briandais Tree[14]
- Discussing a trie implementation in Lisp[15]
- ServerKit "parse trees" implement a form of Trie in C[16]
- A simple implementation of Trie in Python[17]
- A Trie implemented in Ruby[18]
- A reference implementation of bitwise tries in C and C++[19]
- A reference implementation in Java[20]
• A quick tutorial on TRIE in Java and C++ [21]
• A compact C implementation of Judy Tries [22]
• Data::Trie [23] and Tree::Trie [24] Perl implementations.

References

[23] http://search.cpan.org/~hammond/data-trie-0.01/Trie.pm

Radix tree

In computer science, a radix tree (also patricia trie or radix trie) is a space-optimized trie data structure where each node with only one child is merged with its child. The result is that every internal node has at least two children. Unlike in regular tries, edges can be labeled with sequences of characters as well as single characters. This makes them much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.

It supports the following main operations, all of which are $O(k)$, where $k$ is the maximum length of all strings in the set:

- **Lookup**: Determines if a string is in the set. This operation is identical to tries except that some edges consume multiple characters.
- **Insert**: Add a string to the tree. We search the tree until we can make no further progress. At this point we either add a new outgoing edge labeled with all remaining characters in the input string, or if there is already an outgoing edge sharing a prefix with the remaining input string, we split it into two edges (the first labeled with the common prefix) and proceed. This splitting step ensures that no node has more children than there are possible string characters.
- **Delete**: Delete a string from the tree. First, we delete the corresponding leaf. Then, if its parent only has one child remaining, we delete the parent and merge the two incident edges.
- **Find predecessor**: Locates the largest string less than a given string, by lexicographic order.
- **Find successor**: Locates the smallest string greater than a given string, by lexicographic order.

A common extension of radix trees uses two colors of nodes, 'black' and 'white'. To check if a given string is stored in the tree, the search starts from the top and follows the edges of the input string until no further progress can be made. If the search-string is consumed and the final node is a black node, the search has failed; if it is white, the search has succeeded. This enables us to add a large range of strings with a common prefix to the tree, using white nodes, then remove a small set of "exceptions" in a space-efficient manner by inserting them using black nodes.

Applications

As mentioned, radix trees are useful for constructing associative arrays with keys that can be expressed as strings. They find particular application in the area of IP routing, where the ability to contain large ranges of values with a few exceptions is particularly suited to the hierarchical organization of IP addresses.[1] They are also used for inverted indexes of text documents in information retrieval.

History

Donald R. Morrison first described what he called "Patricia trees" in 1968,[2] the name comes from the acronym PATRICIA, which stands for "Practical Algorithm To Retrieve Information Coded In Alphanumeric". Gernot Gwehenberger independently invented and described the data structure at about the same time.[3]
Comparison to other data structures

(In the following comparisons, it is assumed that the keys are of length $k$ and the data structure contains $n$ elements.)

Unlike balanced trees, radix trees permit lookup, insertion, and deletion in $O(k)$ time rather than $O(\log n)$. This doesn't seem like an advantage, since normally $k \geq \log n$, but in a balanced tree every comparison is a string comparison requiring $O(k)$ worst-case time, many of which are slow in practice due to long common prefixes. In a trie, all comparisons require constant time, but it takes $m$ comparisons to look up a string of length $m$. Radix trees can perform these operations with fewer comparisons and require many fewer nodes.

Radix trees also share the disadvantages of tries, however: as they can only be applied to strings of elements or elements with an efficiently reversible mapping (injection) to strings, they lack the full generality of balanced search trees, which apply to any data type with a total ordering. A reversible mapping to strings can be used to produce the required total ordering for balanced search trees, but not the other way around. This can also be problematic if a data type only provides a comparison operation, but not a (de)serialization operation.

Hash tables are commonly said to have expected $O(1)$ insertion and deletion times, but this is only true when considering computation of the hash of the key to be a constant time operation. When hashing the key is taken into account, hash tables have expected $O(k)$ insertion and deletion times, but may take longer in the worst-case depending on how collisions are handled. Radix trees have worst-case $O(k)$ insertion and deletion. The successor/predecessor operations of radix trees are also not implemented by hash tables.

Variants

The HAT-trie is a radix tree based cache-conscious data structure that offers efficient string storage and retrieval, and ordered iterations. Performance, with respect to both time and space, is comparable to the cache-conscious hashtable.\[4\] [5]

References


External links

- Patricia Tree (http://www.nist.gov/dads/HTML/patriciatree.html), NIST Dictionary of Algorithms and Data Structures
- Crit-bit trees (http://cr.yp.to/critbit.html), by Daniel J. Bernstein
- Radix Tree API in the Linux Kernel (http://lwn.net/Articles/175432/), by Jonathan Corbet
- Kart (key alteration radix tree) (http://code.dogmap.org/kart/), by Paul Jarc
Implementations

- C++ Standard library has a trie implementation (http://gcc.gnu.org/onlinedocs/libstdc++/ext/lookup.html)
- Java implementation of Radix Tree (http://badgenow.com/p/radixtree/), by Tahseen Ur Rehman
- C# implementation of a Radix Tree (http://paratechnical.blogspot.com/2011/03/radix-tree-implementation-in-c.html)
- Practical Algorithm Template Library (http://code.google.com/p/patrl/), a C++ library on PATRICIA tries (VC++ >=2003, GCC G++ 3.x), by Roman S. Klyujkov
- Patricia Trie C++ template class implementation (http://www.codeproject.com/KB/string/PatriciaTrieTemplateClass.aspx), by Radu Gruian
- Patricia Trie implementation in Java (http://code.google.com/p/patricia-trie/), by Roger Kapsi and Sam Berlin
- Crit-bit trees (http://github.com/agl/critbit) forked from C code by Daniel J. Bernstein
- Patricia Trie implementation in C (http://cprops.sourceforge.net/gen/docs/trie_8c-source.html), in libcprops (http://cprops.sourceforge.net)
- Patricia Trees : efficient sets and maps over integers in (http://www.lri.fr/~filliatr/ftp/ocaml/ds) OCaml, by Jean-Christophe Filliâtre

Directed acyclic word graph

In computer science, a directed acyclic word graph (sometimes abbreviated as DAWG) is a data structure that represents a set of strings, and allows for a query operation that tests whether a given string belongs to the set in time proportional to its length. In these respects, a DAWG is very similar to a trie, but it is much more space efficient.

A DAWG is represented as a directed acyclic graph with a single source vertex (a vertex with no incoming edges), in which each edge of the graph is labeled by a letter, symbol, or special end-of-string marker, and in which each vertex has at most one outgoing edge for each possible letter or symbol. The strings represented by the DAWG are formed by the symbols on paths in the DAWG from the source vertex to any sink vertex (a vertex with no outgoing edges). A DAWG can also be interpreted as an acyclic finite automaton that accepts the words that are stored in the DAWG.

Thus, a trie (a rooted tree with the same properties of having edges labeled by symbols and strings formed by root-to-leaf paths) is a special kind of DAWG. However, by allowing the same vertices to be reached by multiple paths, a DAWG may use significantly fewer vertices than a trie. Consider, for example, the four English words "tap", "taps", "top", and "tops". A trie for those four words would have 11 vertices, one for each of the strings formed as a
Directed acyclic word graph

prefix of one of these words, or for one of the words followed by the end-of-string marker. However, a DAWG can represent these same four words using only six vertices $v_i$ for $0 \leq i \leq 5$, and the following edges: an edge from $v_0$ to $v_1$ labeled "t", two edges from $v_1$ to $v_2$ labeled "a" and "o", an edge from $v_2$ to $v_3$ labeled "p", an edge $v_3$ to $v_4$ labeled "s", and edges from $v_3$ and $v_4$ to $v_5$ labeled with the end-of-string marker.

The primary difference between DAWG and trie is the elimination of suffix redundancy in storing strings. The trie eliminates prefix redundancy since all common prefixes are shared between strings, such as between doctors and doctorate the doctor prefix is shared. In a DAWG common suffixes are also shared, such as between desertion and destruction both the prefix des- and suffix -tion are shared. For dictionary sets of common English words, this translates into major memory usage reduction.

Because the terminal nodes of a DAWG can be reached by multiple paths, a DAWG cannot directly store auxiliary information relating to each path, e.g. a word's frequency in the English language. However, if at each node we store a count of the number of unique paths through the structure from that point, we can use it to retrieve the index of a word, or a word given its index.¹ The auxiliary information can then be stored in an array.

References

- Appel, Andrew; Jacobsen, Guy (1988), possibly first mention of the data structure (http://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/scrabble.pdf), "The World's Fastest Scrabble Program" (PDF), Communications of the ACM.

External links

- National Institute of Standards and Technology (http://www.nist.gov/dads/HTML/directedAcyclicWordGraph.html)
- DAWG implementation in C# by Samuel Allen (http://dotnetperls.com/directed-acyclic-word-graph)
- Optimal DAWG Creation Step By Step Treatment (http://www.pathcom.com/~vadco/dawg.html)
Suffix tree

In computer science, a **suffix tree** (also called PAT tree or, in an earlier form, position tree) is a data structure that presents the suffixes of a given string in a way that allows for a particularly fast implementation of many important string operations.

The suffix tree for a string $S$ is a tree whose edges are labeled with strings, such that each suffix of $S$ corresponds to exactly one path from the tree's root to a leaf. It is thus a radix tree (more specifically, a Patricia tree) for the suffixes of $S$.

Constructing such a tree for the string $S$ takes time and space linear in the length of $S$. Once constructed, several operations can be performed quickly, for instance locating a substring in $S$, locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern etc. Suffix trees also provided one of the first linear-time solutions for the longest common substring problem. These speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.

**History**

The concept was first introduced as a position tree by Weiner in 1973,[1] which Donald Knuth subsequently characterized as "Algorithm of the Year 1973". The construction was greatly simplified by McCreight in 1976,[2] and also by Ukkonen in 1995.[3][4] Ukkonen provided the first online-construction of suffix trees, now known as Ukkonen's algorithm, with running time that matched the then fastest algorithms. These algorithms are all linear-time for constant-size alphabet, and have worst-case running time of $O(n \log n)$ in general.

In 1997, Martin Farach[5] gave the first suffix tree construction algorithm that is optimal for all alphabets. In particular, this is the first linear-time algorithm for strings drawn from an alphabet of integers in a polynomial range. This latter algorithm has become the basis for new algorithms for constructing both suffix trees and suffix arrays, for example, in external memory, compressed, succinct, etc.
Definition

The suffix tree for the string \( S \) of length \( n \) is defined as a tree such that ([6] page 90):

- the paths from the root to the leaves have a one-to-one relationship with the suffixes of \( S \),
- edges spell non-empty strings,
- and all internal nodes (except perhaps the root) have at least two children.

Since such a tree does not exist for all strings, \( S \) is padded with a terminal symbol not seen in the string (usually denoted \( \$ \)). This ensures that no suffix is a prefix of another, and that there will be \( n \) leaf nodes, one for each of the \( n \) suffixes of \( S \). Since all internal non-root nodes are branching, there can be at most \( n - 1 \) such nodes, and \( n + (n - 1) + 1 = 2n \) nodes in total (\( n \) leaves, \( n - 1 \) internal nodes, 1 root).

Suffix links are a key feature for older linear-time construction algorithms, although most newer algorithms, which are based on Farach's algorithm, dispense with suffix links. In a complete suffix tree, all internal non-root nodes have a suffix link to another internal node. If the path from the root to a node spells the string \( \chi \alpha \), where \( \chi \) is a single character and \( \alpha \) is a string (possibly empty), it has a suffix link to the internal node representing \( \alpha \). See for example the suffix link from the node for \( \text{ANA} \) to the node for \( \text{NA} \) in the figure above. Suffix links are also used in some algorithms running on the tree.

Generalised suffix tree

Generalised suffix tree is a suffix tree made for a set of words instead only for a single word. It represents all suffixes from this set of words. Each word must be terminated by a different differentiation symbol or word.

Functionality

A suffix tree for a string \( S \) of length \( n \) can be built in \( \Theta(n) \) time, if the letters come from an alphabet of integers in a polynomial range (in particular, this is true for constant-sized alphabets). ([5] For larger alphabets, the running time is dominated by first sorting the letters to bring them into a range of size \( O(n) \); in general, this takes \( O(n \log n) \) time. The costs below are given under the assumption that the alphabet is constant.

Assume that a suffix tree has been built for the string \( S \) of length \( n \), or that a generalised suffix tree has been built for the set of strings \( D = \{ S_1, S_2, \ldots, S_K \} \) of total length \( n = |n_1| + |n_2| + \cdots + |n_K| \). You can:

- Search for strings:
  - Check if a string \( P \) of length \( m \) is a substring in \( O(m) \) time ([6] page 92).
  - Find the first occurrence of the patterns \( P_1, \ldots, P_q \) of total length \( m \) as substrings in \( O(m) \) time.
  - Find all \( z \) occurrences of the patterns \( P_1, \ldots, P_q \) of total length \( m \) as substrings in \( O(m + z) \) time ([6] page 123).
  - Search for a regular expression \( P \) in time expected sublinear in \( n \) ([7]).
  - Find for each suffix of a pattern \( P \), the length of the longest match between a prefix of \( P[i \ldots m] \) and a substring in \( D \) in \( \Theta(m) \) time ([6] page 132). This is termed the matching statistics for \( P \).

- Find properties of the strings:
  - Find the longest common substrings of the string \( S_i \) and \( S_j \) in \( \Theta(n_i + n_j) \) time ([6] page 125).
  - Find all maximal pairs, maximal repeats or supermaximal repeats in \( \Theta(n + z) \) time ([6] page 144).
  - Find the Lempel–Ziv decomposition in \( \Theta(n) \) time ([6] page 166).
  - Find the longest repeated substrings in \( \Theta(n) \) time.
  - Find the most frequently occurring substrings of a minimum length in \( \Theta(n) \) time.
  - Find the shortest strings from \( \Sigma \) that do not occur in \( D \), in \( O(n + z) \) time, if there are \( z \) such strings.
  - Find the shortest substrings occurring only once in \( \Theta(n) \) time.
  - Find, for each \( i \), the shortest substrings of \( S_i \) not occurring elsewhere in \( D \) in \( \Theta(n) \) time.
The suffix tree can be prepared for constant time lowest common ancestor retrieval between nodes in $\Theta(n)$ time (chapter 8). You can then also:

- Find the longest common prefix between the suffixes $S_i[p..n_i]$ and $S_j[q..n_j]$ in $\Theta(1)$ time (page 196).
- Search for a pattern $P$ of length $m$ with at most $k$ mismatches in $O(kn + z)$ time, where $z$ is the number of hits (page 200).
- Find all $z$ maximal palindromes in $\Theta(n)$ time (page 198), or $\Theta(gn)$ time if gaps of length $g$ are allowed, or $\Theta(kn)$ if $k$ mismatches are allowed (page 201).
- Find all $z$ tandem repeats in $O(n \log n + z)$, and $k$-mismatch tandem repeats in $O(kn \log(n/k) + z)$ (page 204).
- Find the longest substrings common to at least $k$ strings in $D$ for $k = 2, \ldots, K$ in $\Theta(n)$ time (page 205).

**Applications**

Suffix trees can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology and other application areas. Primary applications include:

- String search, in $O(m)$ complexity, where $m$ is the length of the sub-string (but with initial $O(n)$ time required to build the suffix tree for the string)
- Finding the longest repeated substring
- Finding the longest common substring
- Finding the longest palindrome in a string

Suffix trees are often used in bioinformatics applications, searching for patterns in DNA or protein sequences (which can be viewed as long strings of characters). The ability to search efficiently with mismatches might be considered their greatest strength. Suffix trees are also used in data compression; they can be used to find repeated data, and can be used for the sorting stage of the Burrows–Wheeler transform. Variants of the LZW compression schemes use suffix trees (LZSS). A suffix tree is also used in suffix tree clustering, a data clustering algorithm used in some search engines (first introduced in).

**Implementation**

If each node and edge can be represented in $\Theta(1)$ space, the entire tree can be represented in $\Theta(n)$ space. The total length of all the strings on all of the edges in the tree is $O(n^2)$, but each edge can be stored as the position and length of a substring of $S$, giving a total space usage of $\Theta(n)$ computer words. The worst-case space usage of a suffix tree is seen with a fibonacci word, giving the full $2^n$ nodes.

An important choice when making a suffix tree implementation is the parent-child relationships between nodes. The most common is using linked lists called sibling lists. Each node has a pointer to its first child, and to the next node in the child list it is a part of. Hash maps, sorted/unsorted arrays (with array doubling), and balanced search trees may also be used, giving different running time properties. We are interested in:

- The cost of finding the child on a given character.
- The cost of inserting a child.
- The cost of enlisting all children of a node (divided by the number of children in the table below).

Let $\sigma$ be the size of the alphabet. Then you have the following costs:
Suffix tree

<table>
<thead>
<tr>
<th></th>
<th>Lookup</th>
<th>Insertion</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sibling lists / unsorted arrays</td>
<td>$O(\sigma)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Hash maps</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$O(\sigma)$</td>
</tr>
<tr>
<td>Balanced search tree</td>
<td>$O(\log \sigma)$</td>
<td>$O(\log \sigma)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted arrays</td>
<td>$O(\log \sigma)$</td>
<td>$O(\sigma)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Hash maps + sibling lists</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Note that the insertion cost is amortised, and that the costs for hashing are given perfect hashing.

The large amount of information in each edge and node makes the suffix tree very expensive, consuming about ten to twenty times the memory size of the source text in good implementations. The suffix array reduces this requirement to a factor of four, and researchers have continued to find smaller indexing structures.

External construction

Suffix trees quickly outgrow the main memory on standard machines for sequence collections in the order of gigabytes. As such, their construction calls for external memory approaches.

There are theoretical results for constructing suffix trees in external memory. The algorithm by Farach et al. \cite{Farach1997} is theoretically optimal, with an I/O complexity equal to that of sorting. However, as discussed for example in \cite{Gusfield1997}, the overall intricacy of this algorithm has prevented, so far, its practical implementation.

On the other hand, there have been practical works for constructing disk-based suffix trees which scale to (few) GB/hours. The state of the art methods are TDD, \cite{TDD1999}\cite{Trellis1999}, DiGeST, \cite{Digest2000} and B²ST. \cite{B2ST2002}

TDD and TRELLIS scale up to the entire human genome – approximately 3GB – resulting in a disk-based suffix tree of a size in the tens of gigabytes. \cite{TDD1999}\cite{Trellis1999} However, these methods cannot handle efficiently collections of sequences exceeding 3GB. \cite{Digest2000} DiGeST performs significantly better and is able to handle collections of sequences in the order of 6GB in about 6 hours. \cite{Digest2000} The source code and documentation for the latter is available from \cite{DigestSource2000}. All these methods can efficiently build suffix trees for the case when the tree does not fit in main memory, but the input does. The most recent method, B²ST, \cite{B2ST2002} scales to handle inputs that do not fit in main memory. ERA \cite{ERA2002} is a recent parallel suffix tree construction method that is significantly faster. ERA can index the entire human genome in 19 minutes on an 8-core desktop computer with 16GB RAM. On a simple Linux cluster with 16 nodes (4GB RAM per node), ERA can index the entire human genome in less than 9 minutes.

References


\cite{Allison2008} Allison, L. "Suffix Trees" (http://www.allisons.org/ll/AlgDS/Tree/Suffix/). Retrieved 2008-10-14.
Suffix tree


External links

- Suffix Trees (http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c16/suffix.htm) by Sartaj Sahni
- Suffix Trees (http://www.allisons.org/Il/AlgDS/Tree/Suffix/) by Lloyd Allison
- suffix_tree (http://mil.cas.technion.ac.il/~yona/suffix_tree/) ANSI C implementation of a Suffix Tree
- libstree (http://www.cl.cam.ac.uk/~cpk25/libstree/), a generic suffix tree library written in C
- Tree::Suffix (http://search.cpan.org/dist/Tree-Suffix/), a Perl binding to libstree
- Strmat (http://www.cs.ucdavis.edu/~gusfield/strmat.html) a faster generic suffix tree library written in C (uses arrays instead of linked lists)
- SuffixTree (http://hkn.eecs.berkeley.edu/~dyoo/python/suffix_trees/) a Python binding to Strmat
- Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice (http://www.balkenhol.net/papers/t1043.pdf.gz), application of suffix trees in the BWT
- Theory and Practice of Succinct Data Structures (http://www.cs.helsinki.fi/group/suds/), C++ implementation of a compressed suffix tree
- Practical Algorithm Template Library (http://code.google.com/p/patl/), a C++ library with suffix tree implementation on PATRICIA trie, by Roman S. Klyujkov
- A Java implementation (http://en.literateprograms.org/Suffix_tree_(Java))
**Suffix array**

In computer science, a **suffix array** is an array of integers giving the starting positions of suffixes of a string in lexicographical order.

**Details**

Consider the string

<table>
<thead>
<tr>
<th>index</th>
<th>sorted suffix</th>
<th>lcp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>$</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>a$</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>abra$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>abracadabra$</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>acadabra$</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>adabra$</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>bra$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>bracadabra$</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>cadabra$</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>dabra$</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>ra$</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>racadabra$</td>
<td>2</td>
</tr>
</tbody>
</table>

of length 12, that ends with a sentinel letter $, appearing only once, and less (in lexicographical order) than any other letter in the string.

It has twelve suffixes: "abracadabra$", "bracadabra$", "racadabra$", and so on down to "a$" and "$" that can be sorted into lexicographical order to obtain:

If the original string is available, each suffix can be completely specified by the index of its first character. The suffix array is the array of the indices of suffixes sorted in lexicographical order. For the string "abracadabra$", using one-based indexing, the suffix array is \{12,11,8,1,4,6,9,2,5,7,10,3\}, because the suffix "$" begins at position 12, "a$" begins at position 11, "abra$" begins at position 8, and so forth.

The longest common prefix is also shown above as lcp. This value, stored alongside the list of prefix indices, indicates how many characters a particular suffix has in common with the suffix directly above it, starting at the beginning of both suffixes. The lcp is useful in making some string operations more efficient. For example, it can be used to avoid comparing characters that are already known to be the same when searching through the list of suffixes. The fact that the minimum lcp value belonging to a consecutive set of sorted suffixes gives the longest common prefix among all of those suffixes can also be useful.
### Algorithms

The easiest way to construct a suffix array is to use an efficient comparison sort algorithm. This requires \(O(n \log n)\) suffix comparisons, but a suffix comparison requires \(O(n)\) time, so the overall runtime of this approach is \(O(n^2 \log n)\). More sophisticated algorithms improve this to \(O(n \log n)\) by exploiting the results of partial sorts to avoid redundant comparisons. Several \(\Theta(n)\) algorithms (of Pang Ko and Srinivas Aluru, Juha Kärkkäinen and Peter Sanders, etc.) have also been developed which provide faster construction and have space usage of \(O(n)\) with low constants. These latter are derived from the suffix tree construction algorithm of Farach. Recent work by Salson et al. proposes an algorithm for updating the suffix array of a text that has been edited instead of rebuilding a new suffix array from scratch. Even if the theoretical worst-case time complexity is \(O(n \log n)\), it appears to perform well in practice: experimental results from the authors showed that their implementation of dynamic suffix arrays is generally more efficient than rebuilding when considering the insertion of a reasonable number of letters in the original text (Léonard, Mouchard and Salson).

### Applications

The suffix array of a string can be used as an index to quickly locate every occurrence of a substring within the string. Finding every occurrence of the substring is equivalent to finding every suffix that begins with the substring. Thanks to the lexicographical ordering, these suffixes will be grouped together in the suffix array, and can be found efficiently with a binary search. If implemented straightforwardly, this binary search takes \(O(m \log n)\) time, where \(m\) is the length of the substring \(W\). The following pseudo-code from Manber and Myers shows how to find \(W\) (or the suffix lexicographically immediately before \(W\) if \(W\) is not present) in a suffix array with indices stored in \(pos\), starting with 1 as the first index.

```plaintext
if W <= suffixAt(pos[1]) then
    ans = 1
else if W > suffixAt(pos[n]) then
    ans = n
else
    L = 1, R = n
    while R - L > 1 do
        M = (L + R)/2
        if W <= suffixAt(pos[M]) then
            R = M
        else
            L = M
    ans = R
```

To avoid redoing comparisons, extra data structures giving information about the longest common prefixes (LCPs) of suffixes are constructed, giving \(O(m + \log n)\) search time.

Suffix sorting algorithms can be used to perform the Burrows–Wheeler transform (BWT). Technically the BWT requires sorting cyclic permutations of a string, not suffixes. We can fix this by appending to the string a special end-of-string character which sorts lexicographically before every other character. Sorting cyclic permutations is then equivalent to sorting suffixes.
Suffix arrays are used to look up substrings in Example-Based Machine Translation, demanding much less storage than a full phrase table as used in Statistical machine translation.

History
Suffix arrays were originally developed by Gene Myers and Udi Manber to reduce memory consumption compared to a suffix tree. This began the trend towards compressed suffix arrays and BWT-based compressed full-text indices.

References

External links
- Various algorithms for constructing Suffix Arrays in Java, with performance tests [3]
- Suffix sorting module for BWT in C code [4]
- Suffix Array Implementation in Ruby [5]
- Suffix array library and tools [6]
- Project containing Suffix Array Implementation in Java [7]
- Project containing various Suffix Array c/c++ Implementations with a unified interface [8]
- A fast, lightweight, and robust C API library to construct the suffix array [9]

References
### van Emde Boas tree

<table>
<thead>
<tr>
<th>van Emde Boas tree</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Non-binary tree</td>
</tr>
<tr>
<td>Invented</td>
<td>1977</td>
</tr>
<tr>
<td>Invented by</td>
<td>Peter van Emde Boas</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Asymptotic complexity in big O notation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$O(M)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O(\log \log M)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(\log \log M)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(\log \log M)$</td>
</tr>
</tbody>
</table>

A van Emde Boas tree (or van Emde Boas priority queue), also known as a vEB tree, is a tree data structure which implements an associative array with $m$-bit integer keys. It performs all operations in $O(\log m)$ time. Notice that $m$ is the size of the keys — therefore $O(\log m)$ is $O(\log \log n)$ in a tree where every key below $n$ is set, exponentially better than a full self-balancing binary search tree. They also have good space efficiency when they contain a large number of elements, as discussed below. They were invented by a team led by Peter van Emde Boas in 1977.\(^1\)

### Supported operations

The operations supported by a vEB tree are those of an ordered associative array, which includes the usual associative array operations along with two more order operations, FindNext and FindPrevious:\(^2\)

- **Insert**: insert a key/value pair with an $m$-bit key
- **Delete**: remove the key/value pair with a given key
- **Lookup**: find the value associated with a given key
- **FindNext**: find the key/value pair with the smallest key at least a given $k$
- **FindPrevious**: find the key/value pair with the largest key at most a given $k$

### How it works

For the sake of simplicity, let $\log_2 m = k$ for some integer $k$. Define $M=2^m$. A vEB tree $T$ over the universe $\{0,...,M-1\}$ has a root node that stores an array $T.children$ of length $M^{1/2}$. $T.children[i]$ is a pointer to a vEB tree that is responsible for the values $\{iM^{1/2},..., (i+1)M^{1/2}-1\}$. Additionally, $T$ stores two values $T.min$ and $T.max$ as well as an auxiliary vEB tree $T.aux$.  

![An example Van Emde Boas tree with dimension 5 and the root's aux structure after 1, 2, 3, 5, 8 and 10 have been inserted.](image)
Data is stored in a vEB tree as follows: The smallest value currently in the tree is stored in \( T.\text{min} \) and largest value is stored in \( T.\text{max} \). These two values are not stored anywhere else in the vEB tree. If \( T \) is empty then we use the convention that \( T.\text{max}=1 \) and \( T.\text{min}=M \). Any other value \( x \) is stored in the subtree \( T.\text{children}[i] \) where 
\[
i = \left\lfloor \frac{x}{M^{1/2}} \right\rfloor.
\]
The auxiliary tree \( T.\text{aux} \) keeps track of which children are non-empty, so \( T.\text{aux} \) contains the value \( j \) if and only if \( T.\text{children}[j] \) is non-empty.

**FindNext**

The operation \( \text{FindNext}(T, x) \) that searches for the successor of an element \( x \) in a vEB tree proceeds as follows: If \( x \leq T.\text{min} \) then the search is complete, and the answer is \( T.\text{min} \). If \( x > T.\text{max} \) then the next element does not exist, return \( M \). Otherwise, let \( i = \frac{x}{M^{1/2}} \). If \( x \leq T.\text{children}[i].\text{max} \) then the value being searched for is contained in \( T.\text{children}[i] \) so the search proceeds recursively in \( T.\text{children}[i] \). Otherwise, We search for the value \( i \) in \( T.\text{aux} \). This gives us the index \( j \) of the first subtree that contains an element larger than \( x \). The algorithm then returns \( T.\text{children}[j].\text{min} \). The element found on the children level needs to be composed with the high bits to form a complete next element.

```plaintext
FindNext(T, x)
    if (x <= T.min)
        return T.min
    if (x > T.max)            // no next element
        return M
    i = floor(x/sqrt(M))
    lo = x % sqrt(M)
    hi = x - lo
    if (lo <= T.children[i].max)
        return hi + FindNext(T.children[i], lo)
    return hi + T.children[FindNext(T.aux, i+1)].min
```

Note that, in any case, the algorithm performs \( O(1) \) work and then possibly recurses on a subtree over a universe of size \( M^{1/2} \) (an \( m/2 \) bit universe). This gives a recurrence for the running time of \( T(m)=T(m/2) + O(1) \), which resolves to \( O(\log m) = O(\log \log M) \).

**Insert**

The call \( \text{Insert}(T, x) \) that inserts a value \( x \) into a vEB tree \( T \) operates as follows:

If \( T \) is empty then we set \( T.\text{min} = T.\text{max} = x \) and we are done.

Otherwise, if \( x < T.\text{min} \) then we insert \( T.\text{min} \) into the subtree \( i \) responsible for \( T.\text{min} \) and then set \( T.\text{min} = x \). If \( T.\text{children}[i] \) was previously empty, then we also insert \( i \) into \( T.\text{aux} \).

Otherwise, if \( x > T.\text{max} \) then we insert \( T.\text{max} \) into the subtree \( i \) responsible for \( T.\text{max} \) and then set \( T.\text{max} = x \). If \( T.\text{children}[i] \) was previously empty, then we also insert \( i \) into \( T.\text{aux} \).

Otherwise, \( T.\text{min} < x < T.\text{max} \) so we insert \( x \) into the subtree \( i \) responsible for \( x \). If \( T.\text{children}[i] \) was previously empty, then we also insert \( i \) into \( T.\text{aux} \).

In code:

```plaintext
Insert(T, x)
    if (T.min > T.max)    // T is empty
        T.min = T.max = x;
        return
    if (T.min = T.max)
        if (x < T.min)
van Emde Boas tree

The key to the efficiency of this procedure is that inserting an element into an empty vEB tree takes $O(1)$ time. So, even though the algorithm sometimes makes two recursive calls, this only occurs when the first recursive call was into an empty subtree. This gives the same running time recurrence of $T(m) = T(m/2) + O(1)$ as before.

**Delete**

Deletion from vEB trees is the trickiest of the operations. The call $\text{Delete}(T, x)$ that deletes a value $x$ from a vEB tree $T$ operates as follows:

If $T\.min = T\.max = x$ then $x$ is the only element stored in the tree and we set $T\.min = M$ and $T\.max = -1$ to indicate that the tree is empty.

Otherwise, if $x = T\.min$ then we need to find the second-smallest value $y$ in the vEB tree, delete it from its current location, and set $T\.min = y$. The second-smallest value $y$ is either $T\.max$ or $T\.children[T\.aux\.min].min$, so it can be found in $O(1)$ time. In the latter case we delete $y$ from the subtree that contains it.

Similarly, if $x = T\.max$ then we need to find the second-largest value $y$ in the vEB tree, delete it from its current location, and set $T\.max = y$. The second-largest value $y$ is either $T\.min$ or $T\.children[T\.aux\.max].max$, so it can be found in $O(1)$ time. In the latter case, we delete $y$ from the subtree that contains it.

In case where $x$ is not $T\.min$ or $T\.max$, and $T$ has no other elements, we know $x$ is not in $T$ and return without further operations.

Otherwise, we have the typical case where $x \neq T\.min$ and $x \neq T\.max$. In this case we delete $x$ from the subtree $T\.children[i]$ that contains $x$.

In any of the above cases, if we delete the last element $x$ or $y$ from any subtree $T\.children[i]$ then we also delete $i$ from $T\.aux$.

In code:

```c
Delete(T, x)
    if (T\.min == T\.max == x)
        T\.min = M
        T\.max = -1
        return
    if (x == T\.min)
        if (T\.aux is empty)
            T\.min = T\.max
            return
        else
            x = T\.children[T\.aux\.min].min
```
Again, the efficiency of this procedure hinges on the fact that deleting from a vEB tree that contains only one element takes only constant time. In particular, the last line of code only executes if \( x \) was the only element in \( T.children[i] \) prior to the deletion.

**Discussion**

The assumption that \( \log m \) is an integer is unnecessary. The operations \( x/sqrt(m) \) and \( x%sqrt(m) \) can be replaced by taking only higher-order ceil(m/2) and the lower-order floor(m/2) bits of \( x \), respectively. On any existing machine, this is more efficient than division or remainder computations.

The implementation described above uses pointers and occupies a total space of \( O(M) = O(2^m) \). This can be seen as follows. The recurrence is \( S(M) = O(\sqrt{M}) + (\sqrt{M} + 1) \cdot S(\sqrt{M}) \). Resolving that would lead to \( S(M) \in (1 + \sqrt{M})^{\log \log M} + \log \log M \cdot O(\sqrt{M}) \). One can, fortunately, also show that \( S(M) = M - 2^\beta \) by induction.\[^3\]

In practical implementations, especially on machines with shift-by-k and find first zero instructions, performance can further be improved by switching to a bit array once \( m \) equal to the word size (or a small multiple thereof) is reached. Since all operations on a single word are constant time, this does not affect the asymptotic performance, but it does avoid the majority of the pointer storage and several pointer dereferences, achieving a significant practical savings in time and space with this trick.

An obvious optimization of vEB trees is to discard empty subtrees. This makes vEB trees quite compact when they contain many elements, because no subtrees are created until something needs to be added to them. Initially, each element added creates about \( \log(m) \) new trees containing about \( m/2 \) pointers all together. As the tree grows, more and more subtrees are reused, especially the larger ones. In a full tree of \( 2^m \) elements, only \( O(2^m) \) space is used. Moreover, unlike a binary search tree, most of this space is being used to store data: even for billions of elements, the pointers in a full vEB tree number in the thousands.

However, for small trees the overhead associated with vEB trees is enormous: on the order of \( 2^{m/2} \). This is one reason why they are not popular in practice. One way of addressing this limitation is to use only a fixed number of bits per level, which results in a trie. Other structures, including y-fast tries and x-fast tries have been proposed that have comparable update and query times but use only \( O(n) \) or \( O(n \log M) \) space where \( n \) is the number of elements stored in the data structure.
van Emde Boas tree

References


Fusion tree

A fusion tree is a type of tree data structure in computer science. It implements an associative array with integer keys up to a fixed size; by exploiting the constant-time machine word multiplication operation available on many real processors, it is able to achieve all operations in

\[ O\left(\frac{\log n}{\log \log n}\right) \]

time (see Big O notation), which is slightly faster asymptotically than a self-balancing binary search tree.

References
• MIT CS 6.897: Advanced Data Structures: Lecture 4, Fusion Trees [1], Prof. Erik Demaine (Spring 2003)
• MIT CS 6.851: Advanced Data Structures: Lecture 13, Fusion Tree nodes [2], Prof. Erik Demaine (Spring 2007)

References