

Contents

Articles

Big O notation	1
Binary tree	12
Binary search tree	20
B-tree	32
AVL tree	44
Red–black tree	48
Hash function	63
Priority queue	71
Heap (data structure)	76
Binary heap	79
Leftist tree	85
Topological sorting	88
Breadth-first search	92
Dijkstra's algorithm	95
Prim's algorithm	101
Kruskal's algorithm	105
Bellman–Ford algorithm	108
Depth-first search	112
Biconnected graph	117
Huffman coding	118
Floyd–Warshall algorithm	128
Sorting algorithm	132
Quicksort	141
Boyer–Moore string search algorithm	152

References

Article Sources and Contributors	156
Image Sources, Licenses and Contributors	160

Article Licenses

License	162
---------	-----

Big O notation

In mathematics, **big O notation** is used to describe the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. It is a member of a larger family of notations that is called **Landau notation**, **Bachmann–Landau notation** (after Edmund Landau and Paul Bachmann), or **asymptotic notation**. In computer science, big O notation is used to classify algorithms by how they respond (*e.g.*, in their processing time or working space requirements) to changes in input size.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. A description of a function in terms of big O notation

usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols o , Ω , ω , and Θ , to describe other kinds of bounds on asymptotic growth rates.

Big O notation is also used in many other fields to provide similar estimates.

Formal definition

Let $f(x)$ and $g(x)$ be two functions defined on some subset of the real numbers. One writes

$$f(x) = O(g(x)) \text{ as } x \rightarrow \infty$$

if and only if there is a positive constant M such that for all sufficiently large values of x , $f(x)$ is at most M multiplied by $g(x)$ in absolute value. That is, $f(x) = O(g(x))$ if and only if there exists a positive real number M and a real number x_0 such that

$$|f(x)| \leq M|g(x)| \text{ for all } x > x_0.$$

In many contexts, the assumption that we are interested in the growth rate as the variable x goes to infinity is left unstated, and one writes more simply that $f(x) = O(g(x))$. The notation can also be used to describe the behavior of f near some real number a (often, $a = 0$): we say

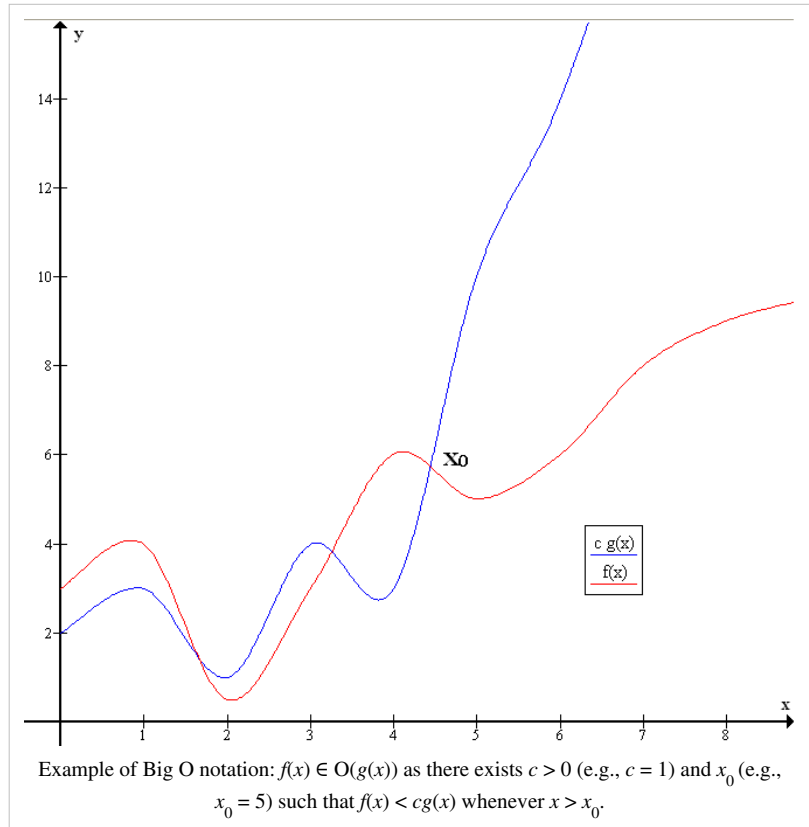
$$f(x) = O(g(x)) \text{ as } x \rightarrow a$$

if and only if there exist positive numbers δ and M such that

$$|f(x)| \leq M|g(x)| \text{ for } |x - a| < \delta.$$

If $g(x)$ is non-zero for values of x sufficiently close to a , both of these definitions can be unified using the limit superior:

$$f(x) = O(g(x)) \text{ as } x \rightarrow a$$



if and only if

$$\limsup_{x \rightarrow a} \left| \frac{f(x)}{g(x)} \right| < \infty.$$

Example

In typical usage, the formal definition of O notation is not used directly; rather, the O notation for a function $f(x)$ is derived by the following simplification rules:

- If $f(x)$ is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.
- If $f(x)$ is a product of several factors, any constants (terms in the product that do not depend on x) are omitted.

For example, let $f(x) = 6x^4 - 2x^3 + 5$, and suppose we wish to simplify this function, using O notation, to describe its growth rate as x approaches infinity. This function is the sum of three terms: $6x^4$, $-2x^3$, and 5 . Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of x , namely $6x^4$. Now one may apply the second rule: $6x^4$ is a product of 6 and x^4 in which the first factor does not depend on x . Omitting this factor results in the simplified form x^4 . Thus, we say that $f(x)$ is a big-oh of (x^4) or mathematically we can write $f(x) = O(x^4)$. One may confirm this calculation using the formal definition: let $f(x) = 6x^4 - 2x^3 + 5$ and $g(x) = x^4$. Applying the formal definition from above, the statement that $f(x) = O(x^4)$ is equivalent to its expansion,

$$|f(x)| \leq M|g(x)|$$

for some suitable choice of x_0 and M and for all $x > x_0$. To prove this, let $x_0 = 1$ and $M = 13$. Then, for all $x > x_0$:

$$\begin{aligned} |6x^4 - 2x^3 + 5| &\leq 6x^4 + |2x^3| + 5 \\ &\leq 6x^4 + 2x^4 + 5x^4 \\ &\leq 13x^4 \\ &\leq 13|x^4| \end{aligned}$$

so

$$|6x^4 - 2x^3 + 5| \leq 13|x^4|.$$

Usage

Big O notation has two main areas of application. In mathematics, it is commonly used to describe how closely a finite series approximates a given function, especially in the case of a truncated Taylor series or asymptotic expansion. In computer science, it is useful in the analysis of algorithms. In both applications, the function $g(x)$ appearing within the $O(\dots)$ is typically chosen to be as simple as possible, omitting constant factors and lower order terms. There are two formally close, but noticeably different, usages of this notation: infinite asymptotics and infinitesimal asymptotics. This distinction is only in application and not in principle, however—the formal definition for the "big O" is the same for both cases, only with different limits for the function argument.

Infinite asymptotics

Big O notation is useful when analyzing algorithms for efficiency. For example, the time (or the number of steps) it takes to complete a problem of size n might be found to be $T(n) = 4n^2 - 2n + 2$. As n grows large, the n^2 term will come to dominate, so that all other terms can be neglected — for instance when $n = 500$, the term $4n^2$ is 1000 times as large as the $2n$ term. Ignoring the latter would have negligible effect on the expression's value for most purposes. Further, the coefficients become irrelevant if we compare to any other order of expression, such as an expression containing a term n^3 or n^4 . Even if $T(n) = 1,000,000n^2$, if $U(n) = n^3$, the latter will always exceed the former once n grows larger than 1,000,000 ($T(1,000,000) = 1,000,000^3 = U(1,000,000)$). Additionally, the number of steps depends on the details of the machine model on which the algorithm runs, but different types of machines typically vary by only a constant factor in the number of steps needed to execute an algorithm. So the big O notation captures what

remains: we write either

$$T(n) = O(n^2)$$

or

$$T(n) \in O(n^2)$$

and say that the algorithm has *order of* n^2 time complexity. Note that "=" is not meant to express "is equal to" in its normal mathematical sense, but rather a more colloquial "is", so the second expression is technically accurate (see the "Equals sign" discussion below) while the first is a common abuse of notation.^[1]

Infinitesimal asymptotics

Big O can also be used to describe the error term in an approximation to a mathematical function. The most significant terms are written explicitly, and then the least-significant terms are summarized in a single big O term. For example,

$$e^x = 1 + x + \frac{x^2}{2} + O(x^3) \quad \text{as } x \rightarrow 0$$

expresses the fact that the error, the difference $e^x - (1 + x + x^2/2)$, is smaller in absolute value than some constant times $|x^3|$ when x is close enough to 0.

Properties

If a function $f(n)$ can be written as a finite sum of other functions, then the fastest growing one determines the order of $f(n)$. For example

$$f(n) = 9 \log n + 5(\log n)^3 + 3n^2 + 2n^3 = O(n^3).$$

In particular, if a function may be bounded by a polynomial in n , then as n tends to *infinity*, one may disregard *lower-order* terms of the polynomial. $O(n^c)$ and $O(n^n)$ are very different. If c is greater than one, then the latter grows much faster. A function that grows faster than n^c for any c is called *superpolynomial*. One that grows more slowly than any exponential function of the form c^n is called *subexponential*. An algorithm can require time that is both superpolynomial and subexponential; examples of this include the fastest known algorithms for integer factorization. $O(\log n)$ is exactly the same as $O(\log(n^c))$. The logarithms differ only by a constant factor (since $\log(n^c) = c \log n$) and thus the big O notation ignores that. Similarly, logs with different constant bases are equivalent. Exponentials with different bases, on the other hand, are not of the same order. For example, 2^n and 3^n are not of the same order. Changing units may or may not affect the order of the resulting algorithm. Changing units is equivalent to multiplying the appropriate variable by a constant wherever it appears. For example, if an algorithm runs in the order of n^2 , replacing n by cn means the algorithm runs in the order of $c^2 n^2$, and the big O notation ignores the constant c^2 . This can be written as $c^2 n^2 \in O(n^2)$. If, however, an algorithm runs in the order of 2^n , replacing n with cn gives $2^{cn} = (2^c)^n$. This is not equivalent to 2^n in general. Changing of variable may affect the order of the resulting algorithm. For example, if an algorithm's running time is $O(n)$ when measured in terms of the number n of *digits* of an input number x , then its running time is $O(\log x)$ when measured as a function of the input number x itself, because $n = \Theta(\log x)$.

Product

$$f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

$$f \cdot O(g) \subset O(fg)$$

Sum

$$f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(|g_1| + |g_2|)$$

This implies $f_1 \in O(g)$ and $f_2 \in O(g) \Rightarrow f_1 + f_2 \in O(g)$, which means that $O(g)$ is a convex cone.

If f and g are positive functions, $f + O(g) \in O(f + g)$

Multiplication by a constant

Let k be a constant. Then:

$$O(kg) = O(g) \text{ if } k \text{ is nonzero.}$$

$$f \in O(g) \Rightarrow kf \in O(g).$$

Multiple variables

Big O (and little o , and Ω ...) can also be used with multiple variables. To define Big O formally for multiple variables, suppose $f(\vec{x})$ and $g(\vec{x})$ are two functions defined on some subset of \mathbb{R}^n . We say

$$f(\vec{x}) \text{ is } O(g(\vec{x})) \text{ as } \vec{x} \rightarrow \infty$$

if and only if

$$\exists C \exists M > 0 \text{ such that } |f(\vec{x})| \leq C|g(\vec{x})| \text{ for all } \vec{x} \text{ with } x_i > M \text{ for all } i.$$

For example, the statement

$$f(n, m) = n^2 + m^3 + O(n + m) \text{ as } n, m \rightarrow \infty$$

asserts that there exist constants C and M such that

$$\forall n, m > M: |g(n, m)| \leq C(n + m),$$

where $g(n, m)$ is defined by

$$f(n, m) = n^2 + m^3 + g(n, m).$$

Note that this definition allows all of the coordinates of \vec{x} to increase to infinity. In particular, the statement

$$f(n, m) = O(n^m) \text{ as } n, m \rightarrow \infty$$

(i.e., $\exists C \exists M \forall n \forall m \dots$) is quite different from

$$\forall m: f(n, m) = O(n^m) \text{ as } n \rightarrow \infty$$

(i.e., $\forall m \exists C \exists M \forall n \dots$).

Matters of notation

Equals sign

The statement " $f(x)$ is $O(g(x))$ " as defined above is usually written as $f(x) = O(g(x))$. Some consider this to be an abuse of notation, since the use of the equals sign could be misleading as it suggests a symmetry that this statement does not have. As de Bruijn says, $O(x) = O(x^2)$ is true but $O(x^2) = O(x)$ is not.^[2] Knuth describes such statements as "one-way equalities", since if the sides could be reversed, "we could deduce ridiculous things like $n = n^2$ from the identities $n = O(n^2)$ and $n^2 = O(n^2)$."^[3] For these reasons, it would be more precise to use set notation and write $f(x) \in O(g(x))$, thinking of $O(g(x))$ as the class of all functions $h(x)$ such that $|h(x)| \leq C|g(x)|$ for some constant C .^[3]

However, the use of the equals sign is customary. Knuth pointed out that "mathematicians customarily use the = sign as they use the word 'is' in English: Aristotle is a man, but a man isn't necessarily Aristotle."^[4]

Other arithmetic operators

Big O notation can also be used in conjunction with other arithmetic operators in more complicated equations. For example, $h(x) + O(f(x))$ denotes the collection of functions having the growth of $h(x)$ plus a part whose growth is limited to that of $f(x)$. Thus,

$$g(x) = h(x) + O(f(x))$$

expresses the same as

$$g(x) - h(x) \in O(f(x)) .$$

Example

Suppose an algorithm is being developed to operate on a set of n elements. Its developers are interested in finding a function $T(n)$ that will express how long the algorithm will take to run (in some arbitrary measurement of time) in terms of the number of elements in the input set. The algorithm works by first calling a subroutine to sort the elements in the set and then perform its own operations. The sort has a known time complexity of $O(n^2)$, and after the subroutine runs the algorithm must take an additional $55n^3 + 2n + 10$ time before it terminates. Thus the overall time complexity of the algorithm can be expressed as

$$T(n) = O(n^2) + 55n^3 + 2n + 10.$$

This can perhaps be most easily read by replacing $O(n^2)$ with "some function that grows asymptotically no faster than n^2 ". Again, this usage disregards some of the formal meaning of the "=" and "+" symbols, but it does allow one to use the big O notation as a kind of convenient placeholder.

Declaration of variables

Another feature of the notation, although less exceptional, is that function arguments may need to be inferred from the context when several variables are involved. The following two right-hand side big O notations have dramatically different meanings:

$$f(m) = O(m^n) ,$$

$$g(n) = O(m^n) .$$

The first case states that $f(m)$ exhibits polynomial growth, while the second, assuming $m > 1$, states that $g(n)$ exhibits exponential growth. To avoid confusion, some authors use the notation

$$g(x) \in O(f(x)) .$$

rather than the less explicit

$$g \in O(f) ,$$

Multiple usages

In more complicated usage, $O(\dots)$ can appear in different places in an equation, even several times on each side. For example, the following are true for $n \rightarrow \infty$

$$\begin{aligned}(n+1)^2 &= n^2 + O(n) \\ (n + O(n^{1/2}))(n + O(\log n))^2 &= n^3 + O(n^{5/2}) \\ n^{O(1)} &= O(e^n).\end{aligned}$$

The meaning of such statements is as follows: for *any* functions which satisfy each $O(\dots)$ on the left side, there are *some* functions satisfying each $O(\dots)$ on the right side, such that substituting all these functions into the equation makes the two sides equal. For example, the third equation above means: "For any function $f(n) = O(1)$, there is some function $g(n) = O(e^n)$ such that $n^{f(n)} = g(n)$." In terms of the "set notation" above, the meaning is that the class of functions represented by the left side is a subset of the class of functions represented by the right side. In this use the "=" is a formal symbol that unlike the usual use of "=" is not a symmetric relation. Thus for example $n^{O(1)} = O(e^n)$ does not imply the false statement $O(e^n) = n^{O(1)}$.

Orders of common functions

Further information: Time complexity#Table of common time complexities

Here is a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case, c is a constant and n increases without bound. The slower-growing functions are generally listed first.

Notation	Name	Example
$O(1)$	constant	Determining if a number is even or odd; using a constant-size lookup table
$O(\log \log n)$	double logarithmic	Finding an item using interpolation search in a sorted array of uniformly distributed values.
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap.
$O(n^c)$, $0 < c < 1$	fractional power	Searching in a kd-tree
$O(n)$	linear	Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; Adding two n -bit integers by ripple carry.
$O(n \log^* n)$	$n \log$ -star n	Performing triangulation of a simple polygon using Seidel's algorithm. (Note $\log^*(n) = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{if } n > 1 \end{cases}$)
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear	Performing a Fast Fourier transform; heapsort, quicksort (best and average case), or merge sort
$O(n^2)$	quadratic	Multiplying two n -digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), Shell sort, quicksort (worst case), selection sort or insertion sort
$O(n^c)$, $c > 1$	polynomial or algebraic	Tree-adjointing grammar parsing; maximum matching for bipartite graphs
$L_n[\alpha, c]$, $0 < \alpha < 1 = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n)$, $c > 1$	exponential	Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the traveling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with expansion by minors.

The statement $f(n) = O(n!)$ is sometimes weakened to $f(n) = O(n^n)$ to derive simpler formulas for asymptotic complexity. For any $k > 0$ and $c > 0$, $O(n^c(\log n)^k)$ is a subset of $O(n^{c+\varepsilon})$ for any $\varepsilon > 0$, so may be considered as a polynomial with some bigger order.

Related asymptotic notations

Big O is the most commonly used asymptotic notation for comparing functions, although in many cases Big O may be replaced with Big Theta Θ for asymptotically tighter bounds. Here, we define some related notations in terms of Big O , progressing up to the family of Bachmann–Landau notations to which Big O notation belongs.

Little-o notation

The relation $f(x) \in o(g(x))$ is read as " $f(x)$ is little-o of $g(x)$ ". Intuitively, it means that $g(x)$ grows much faster than $f(x)$, or similarly, the growth of $f(x)$ is nothing compared to that of $g(x)$. It assumes that f and g are both functions of one variable. Formally, $f(n) = o(g(n))$ as $n \rightarrow \infty$ means that for every positive constant ε there exists a constant N such that

$$|f(n)| \leq \varepsilon |g(n)| \quad \text{for all } n \geq N. \quad [3]$$

Note the difference between the earlier formal definition for the big- O notation, and the present definition of little- o : while the former has to be true for *at least one* constant M the latter must hold for *every* positive constant ε , however small.^[1] In this way little- o notation makes a stronger statement than the corresponding big- O notation: every function that is little- o of g is also big- O of g , but not every function that is big- O of g is also little- o of g (for instance g itself is not, unless it is identically zero near ∞).

If $g(x)$ is nonzero, or at least becomes nonzero beyond a certain point, the relation $f(x) = o(g(x))$ is equivalent to

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0.$$

For example,

- $2x \in o(x^2)$
- $2x^2 \notin o(x^2)$
- $1/x \in o(1)$

Little- o notation is common in mathematics but rarer in computer science. In computer science the variable (and function value) is most often a natural number. In mathematics, the variable and function values are often real numbers. The following properties can be useful:

- $o(f) + o(f) \subseteq o(f)$
- $o(f)o(g) \subseteq o(fg)$
- $o(o(f)) \subseteq o(f)$
- $o(f) \subset O(f)$ (and thus the above properties apply with most combinations of o and O).

As with big O notation, the statement " $f(x)$ is $o(g(x))$ " is usually written as $f(x) = o(g(x))$, which is a slight abuse of notation.

Family of Bachmann–Landau notations

Notation	Name	Intuition	Informal definition: for sufficiently large n ...	Formal Definition	Notes
$f(n) \in O(g(n))$	Big Omicron; Big O; Big Oh	f is bounded above by g (up to constant factor) asymptotically	$ f(n) \leq g(n) \cdot k$ for some k	$\exists k > 0 \exists n_0 \forall n > n_0 f(n) \leq g(n) \cdot k $ or $\exists k > 0 \exists n_0 \forall n > n_0 f(n) \leq g(n) \cdot k$	
$f(n) \in \Omega(g(n))$	Big Omega	f is bounded below by g (up to constant factor) asymptotically	$f(n) \geq g(n) \cdot k$ for some positive k	$\exists k > 0 \exists n_0 \forall n > n_0 g(n) \cdot k \leq f(n)$	Since the beginning of the 20th century, papers in number theory have been increasingly and widely using this notation in the weaker sense that $f = o(g)$ is false.
$f(n) \in \Theta(g(n))$	Big Theta	f is bounded both above and below by g asymptotically	$g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$ for some positive k_1, k_2	$\exists k_1 > 0 \exists k_2 > 0 \exists n_0 \forall n > n_0 g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$	
$f(n) \in o(g(n))$	Small Omicron; Small O; Small Oh	f is dominated by g asymptotically	$ f(n) \leq g(n) \cdot \varepsilon$ for every ε	$\forall \varepsilon > 0 \exists n_0 \forall n > n_0 f(n) \leq g(n) \cdot \varepsilon $	
$f(n) \in \omega(g(n))$	Small Omega	f dominates g asymptotically	$f(n) \geq g(n) \cdot k$ for every k	$\forall k > 0 \exists n_0 \forall n > n_0 g(n) \cdot k < f(n)$	
$f(n) \sim g(n)$	On the order of	f is equal to g asymptotically	$f(n)/g(n) \rightarrow 1$	$\forall \varepsilon > 0 \exists n_0 \forall n > n_0 \left \frac{f(n)}{g(n)} - 1 \right < \varepsilon$	

Bachmann–Landau notation was designed around several mnemonics, as shown in the As $n \rightarrow \infty$, eventually... column above and in the bullets below. To conceptually access these mnemonics, "omicron" can be read "o-micron" and "omega" can be read "o-mega". Also, the lower-case versus capitalization of the Greek letters in Bachmann–Landau notation is mnemonic.

- The *o-micron* mnemonic: The o-micron reading of $f(n) \in O(g(n))$ and of $f(n) \in o(g(n))$ can be thought of as "O-smaller than" and "o-smaller than", respectively. This *micro*/smaller mnemonic refers to: for sufficiently large input parameter(s), f grows at a rate that may henceforth be **less** than cg regarding $g \in O(f)$ or $g \in o(f)$.
- The *o-mega* mnemonic: The o-mega reading of $f(n) \in \Omega(g(n))$ and of $f(n) \in \omega(g(n))$ can be thought of as "O-larger than". This *mega*/larger mnemonic refers to: for sufficiently large input parameter(s), f grows at a rate that may henceforth be **greater** than cg regarding $g \in \Omega(f)$ or $g \in \omega(f)$.

- The **upper-case mnemonic**: This mnemonic reminds us when to use the upper-case Greek letters in $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$: for sufficiently large input parameter(s), f grows at a rate that may henceforth be **equal** to \mathcal{Cg} regarding $g \in O(f)$.
- The **lower-case mnemonic**: This mnemonic reminds us when to use the lower-case Greek letters in $f(n) \in o(g(n))$ and $f(n) \in \omega(g(n))$: for sufficiently large input parameter(s), f grows at a rate that is henceforth **inequal** to \mathcal{Cg} regarding $g \in O(f)$.

Aside from Big O notation, the Big Theta Θ and Big Omega Ω notations are the two most often used in computer science; the Small Omega ω notation is rarely used in computer science.

Use in computer science

Informally, especially in computer science, the Big O notation often is permitted to be somewhat abused to describe an asymptotic tight bound where using Big Theta Θ notation might be more factually appropriate in a given context. For example, when considering a function $T(n) = 73n^3 + 22n^2 + 58$, all of the following are generally acceptable, but tightnesses of bound (i.e., numbers 2 and 3 below) are usually strongly preferred over laxness of bound (i.e., number 1 below).

1. $T(n) = O(n^{100})$, which is identical to $T(n) \in O(n^{100})$
2. $T(n) = O(n^3)$, which is identical to $T(n) \in O(n^3)$
3. $T(n) = \Theta(n^3)$, which is identical to $T(n) \in \Theta(n^3)$.

The equivalent English statements are respectively:

1. $T(n)$ grows asymptotically no faster than n^{100}
2. $T(n)$ grows asymptotically no faster than n^3
3. $T(n)$ grows asymptotically as fast as n^3 .

So while all three statements are true, progressively more information is contained in each. In some fields, however, the Big O notation (number 2 in the lists above) would be used more commonly than the Big Theta notation (bullets number 3 in the lists above) because functions that grow more slowly are more desirable. For example, if $T(n)$ represents the running time of a newly developed algorithm for input size n , the inventors and users of the algorithm might be more inclined to put an upper asymptotic bound on how long it will take to run without making an explicit statement about the lower asymptotic bound.

Extensions to the Bachmann–Landau notations

Another notation sometimes used in computer science is \tilde{O} (read *soft-O*): $f(n) = \tilde{O}(g(n))$ is shorthand for $f(n) = O(g(n) \log^k g(n))$ for some k . Essentially, it is Big O notation, ignoring logarithmic factors because the growth-rate effects of some other super-logarithmic function indicate a growth-rate explosion for large-sized input parameters that is more important to predicting bad run-time performance than the finer-point effects contributed by the logarithmic-growth factor(s). This notation is often used to obviate the "nitpicking" within growth-rates that are stated as too tightly bounded for the matters at hand (since $\log^k n$ is always $o(n^\varepsilon)$ for any constant k and any $\varepsilon > 0$). The L notation, defined as

$$L_n[\alpha, c] = O\left(e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}\right),$$

is convenient for functions that are between polynomial and exponential.

Generalizations and related usages

The generalization to functions taking values in any normed vector space is straightforward (replacing absolute values by norms), where f and g need not take their values in the same space. A generalization to functions g taking values in any topological group is also possible. The "limiting process" $x \rightarrow x_o$ can also be generalized by introducing an arbitrary filter base, i.e. to directed nets f and g . The o notation can be used to define derivatives and differentiability in quite general spaces, and also (asymptotical) equivalence of functions,

$$f \sim g \iff (f - g) \in o(g)$$

which is an equivalence relation and a more restrictive notion than the relationship " f is $\Theta(g)$ " from above. (It reduces to $\lim f/g = 1$ if f and g are positive real valued functions.) For example, $2x$ is $\Theta(x)$, but $2x - x$ is not $o(x)$.

Graph theory

Big O notation is used to describe the running time of graph algorithms. A graph \mathbf{G} is an ordered pair (\mathbf{V}, \mathbf{E}) where \mathbf{V} is the set of vertices and \mathbf{E} is the set of edges. For expressing computational complexity, the relevant parameters are usually not the actual sets, but rather the number of elements in each set: the number of vertices $V = |\mathbf{V}|$ and the number of edges $E = |\mathbf{E}|$. The operator $\|\cdot\|$ measures the cardinality (i.e., the number of elements) of the set. Inside asymptotic notation, it is common to use the symbols V and E , when one means $|\mathbf{V}|$ and $|\mathbf{E}|$. Another common convention uses n and m to refer to $|\mathbf{V}|$ and $|\mathbf{E}|$ respectively; it avoids the confusing the sets with their cardinalities.

History (Bachmann–Landau, Hardy, and Vinogradov notations)

The symbol O was first introduced by number theorist Paul Bachmann in 1894, in the second volume of his book *Analytische Zahlentheorie* ("analytic number theory"), the first volume of which (not yet containing big O notation) was published in 1892.^[5] The number theorist Edmund Landau adopted it, and was thus inspired to introduce in 1909 the notation o ^[6]; hence both are now called Landau symbols. The former was popularized in computer science by Donald Knuth, who re-introduced the related Omega and Theta notations.^[7] Knuth also noted that the Omega notation had been introduced by Hardy and Littlewood^[8] under a different meaning " $\neq o$ " (i.e. "is not an o of"), and proposed the above definition. Hardy and Littlewood's original definition (which was also used in one paper by Landau^[9]) is still used in number theory (where Knuth's definition is never used). In fact, Landau introduced in 1924, in the paper just mentioned, the symbols Ω_R ("rechts") and Ω_L ("links"), precursors for the modern symbols Ω_+ ("is not smaller than a small o of") and Ω_- ("is not larger than a o of"). Thus the Omega symbols (with their original meanings) are sometimes also referred to as "Landau symbols". Also, Landau never used the Big Theta and small omega symbols.

Hardy's symbols were (in terms of the modern O notation)

$$f \preceq g \iff f \in O(g) \quad \text{and} \quad f \prec g \iff f \in o(g);$$

(Hardy however never defined or used the notation $\prec\prec$, nor \ll , as it has been sometimes reported). It should also be noted that Hardy introduces the symbols \preceq and \prec (as well as some other symbols) in his 1910 tract "Orders of Infinity", and makes use of it only in three papers (1910–1913). In the remaining papers (nearly 400!) and books he constantly uses the Landau symbols O and o .

Hardy's notation is not used anymore. On the other hand, in 1947, the Russian number theorist Ivan Matveyevich Vinogradov introduced his notation \ll , which has been increasingly used in number theory instead of the O notation. We have

$$f \ll g \iff f \in O(g),$$

and frequently both notations are used in the same paper.

The big-O, standing for "order of", was originally a capital omicron; today the identical-looking Latin capital letter O is used, but never the digit zero.

References

- [1] Thomas H. Cormen et al., 2001, *Introduction to Algorithms*, Second Edition (<http://highered.mcgraw-hill.com/sites/0070131511/>)
- [2] N. G. de Bruijn (1958). *Asymptotic Methods in Analysis* (http://books.google.com/?id=_tnwmvHmVwMC&pg=PA5&vq=The+trouble+is). Amsterdam: North-Holland. pp. 5–7. ISBN 978-0-486-64221-5. .
- [3] Ronald Graham, Donald Knuth, and Oren Patashnik (1994). 0-201-55802-5 *Concrete Mathematics* (<http://books.google.com/?id=pntQAAAAAAAJ&dq=editions:ISBN>) (2 ed.). Reading, Massachusetts: Addison–Wesley. p. 446. ISBN 978-0-201-55802-9. 0-201-55802-5.
- [4] Donald Knuth (June/July 1998). "Teach Calculus with Big O" (<http://www.ams.org/notices/199806/commentary.pdf>). *Notices of the American Mathematical Society* **45** (6): 687. . (Unabridged version (<http://www-cs-staff.stanford.edu/~knuth/ocalc.tex>))
- [5] Nicholas J. Higham, *Handbook of writing for the mathematical sciences*, SIAM. ISBN 0-89871-420-6, p. 25
- [6] Edmund Landau. *Handbuch der Verteilung der Primzahlen*, Leipzig 1909, p.883.
- [7] Donald Knuth. *Big Omicron and big Omega and big Theta* (<http://doi.acm.org/10.1145/1008328.1008329>), ACM SIGACT News, Volume 8, Issue 2, 1976.
- [8] G. H. Hardy and J. E. Littlewood, *Some problems of Diophantine approximation*, Acta Mathematica 37 (1914), p. 225
- [9] E. Landau, *Nachr. Gesell. Wiss. Gött. Math-phys. Kl.* 1924, 137–150.

Further reading

- Paul Bachmann. *Die Analytische Zahlentheorie. Zahlentheorie*. pt. 2 Leipzig: B. G. Teubner, 1894.
- Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. 2 vols. Leipzig: B. G. Teubner, 1909.
- G. H. Hardy. *Orders of Infinity: The 'Infinitärcalcul' of Paul du Bois-Reymond*, 1910.
- Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison–Wesley, 1997. ISBN 0-201-89683-4. Section 1.2.11: Asymptotic Representations, pp. 107–123.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw–Hill, 2001. ISBN 0-262-03293-7. Section 3.1: Asymptotic notation, pp. 41–50.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Pages 226–228 of section 7.1: Measuring complexity.
- Jeremy Avigad, Kevin Donnelly. *Formalizing O notation in Isabelle/HOL* (<http://www.andrew.cmu.edu/~avigad/Papers/bigo.pdf>)
- Paul E. Black, "big-O notation" (<http://www.nist.gov/dads/HTML/bigOnotation.html>), in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 11 March 2005. Retrieved December 16, 2006.
- Paul E. Black, "little-o notation" (<http://www.nist.gov/dads/HTML/littleOnotation.html>), in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. Retrieved December 16, 2006.
- Paul E. Black, " Ω " (<http://www.nist.gov/dads/HTML/omegaCapital.html>), in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. Retrieved December 16, 2006.
- Paul E. Black, " ω " (<http://www.nist.gov/dads/HTML/omega.html>), in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 29 November 2004. Retrieved December 16, 2006.
- Paul E. Black, " Θ " (<http://www.nist.gov/dads/HTML/theta.html>), in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. Retrieved December 16, 2006.

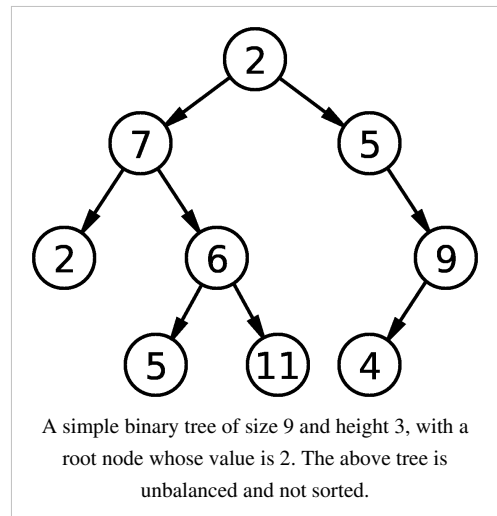
External links

- Introduction to Asymptotic Notations (<http://www.soe.ucsc.edu/classes/cmps102/Spring04/TantaloAsymp.pdf>)
- Landau Symbols (<http://mathworld.wolfram.com/LandauSymbols.html>)
- O-Notation Visualizer: Interactive Graphs of Common O-Notations (<https://students.ics.uci.edu/~zmohiudd/ONotationVisualizer.html>)
- Big-O Notation – What is it good for (http://www.perlmonks.org/?node_id=573138)

Binary tree

In computer science, a **binary tree** is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right". Nodes with children are parent nodes, and child nodes may contain references to their parents. Outside the tree, there is often a reference to the "root" node (the ancestor of all nodes), if it exists. Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left or right child. A tree which does not have any node other than root node is called a null tree. In a binary tree a degree of every node is maximum two. A tree with n nodes has exactly $n-1$ branches or degree.

Binary trees are used to implement binary search trees and binary heaps.

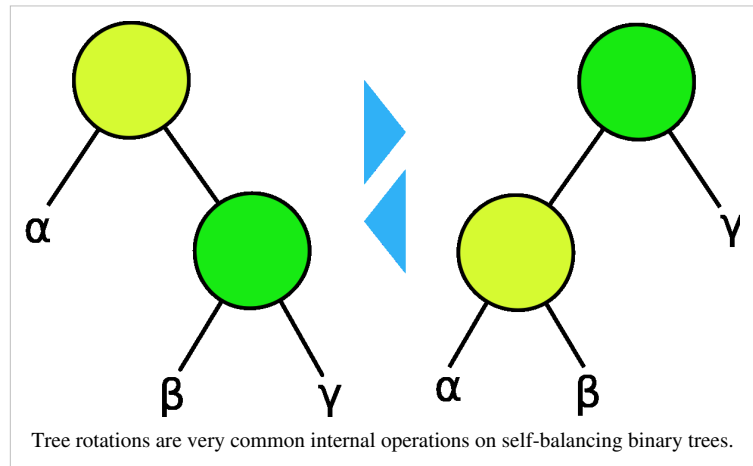


Definitions for rooted trees

- A **directed edge** refers to the link from the parent to the child (the arrows in the picture of the tree).
- The root node of a tree is the node with no parents. There is at most one root node in a rooted tree.
- A leaf node has no children.
- The **depth** of a node n is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a **level** of the tree. The root node is at depth zero.
- The **depth** (or *height*) of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a depth of zero.
- **Siblings** are nodes that share the same parent node.
- A node p is an **ancestor** of a node q if it exists on the path from the root to node q . The node q is then termed as a **descendant** of p .
- The **size** of a node is the number of descendants it has including itself.
- **In-degree** of a node is the number of edges arriving at that node.
- **Out-degree** of a node is the number of edges leaving that node.
- The root is the only node in the tree with In-degree = 0.
- All the leaf nodes have Out-degree = 0.

Types of binary trees

- A **rooted binary tree** is a tree with a root node in which every node has at most two children.
- A **full binary tree** (sometimes **proper binary tree** or **2-tree** or **strictly binary tree**) is a tree in which every node other than the leaves has two children. Or, perhaps more clearly, every node in a binary tree has exactly 0 or 2 children. Sometimes a full tree is ambiguously defined as a perfect tree.
- A **perfect binary tree** is a *full binary tree* in which all *leaves* are at the same *depth* or same *level*, and in which every parent has two children.^[1] (This is ambiguously also called a *complete binary tree*.)
- A **complete binary tree** is a binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible.^[2]
- An **infinite complete binary tree** is a tree with a countably infinite number of levels, in which every node has two children, so that there are 2^d nodes at level d . The set of all nodes is countably infinite, but the set of all infinite paths from the root is uncountable: it has the cardinality of the continuum. These paths corresponding by an order preserving bijection to the points of the Cantor set, or (through the example of the Stern–Brocot tree) to the set of positive irrational numbers.
- A **balanced binary tree** is commonly defined as a binary tree in which the depth of the two subtrees of every node differ by 1 or less,^[3] although in general it is a binary tree where no leaf is much farther away from the root than any other leaf. (Different balancing schemes allow different definitions of "much farther".^[4]) Binary trees that are balanced according to this definition have a predictable depth (how many nodes are traversed from the root to a leaf, root counting as node 0 and subsequent as 1, 2, ..., depth). This depth is equal to the integer part of $\log_2(n)$ where n is the number of nodes on the balanced tree. Example 1: balanced tree with 1 node, $\log_2(1) = 0$ (depth = 0). Example 2: balanced tree with 3 nodes, $\log_2(3) = 1.59$ (depth=1). Example 3: balanced tree with 5 nodes, $\log_2(5) = 2.32$ (depth of tree is 2 nodes).
- A **degenerate tree** is a tree where for each parent node, there is only one associated child node. This means that in a performance measurement, the tree will behave like a linked list data structure.



Note that this terminology often varies in the literature, especially with respect to the meaning of "complete" and "full".

Properties of binary trees

- The number n of nodes in a perfect binary tree can be found using this formula: $n = 2^{h+1} - 1$ where h is the depth of the tree.
- The number n of nodes in a binary tree of height h is at least $n = h + 1$ and at most $n = 2^{h+1} - 1$ where h is the depth of the tree.
- The number L of leaf nodes in a perfect binary tree can be found using this formula: $L = 2^h$ where h is the depth of the tree.
- The number n of nodes in a perfect binary tree can also be found using this formula: $n = 2L - 1$ where L is the number of leaf nodes in the tree.
- The number of null links (absent children of nodes) in a complete binary tree of n nodes is $(n + 1)$.

- The number $n - L$ of internal nodes (non-leaf nodes) in a Complete Binary Tree of n nodes is $\lfloor n/2 \rfloor$.
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.^[5]

Proof:

Let n = the total number of nodes

B = number of branches

n_0 , n_1 , n_2 represent the number of nodes with no children, a single child, and two children respectively.

$B = n - 1$ (since all nodes except the root node come from a single branch)

$$B = n_1 + 2*n_2$$

$$n = n_1 + 2*n_2 + 1$$

$$n = n_0 + n_1 + n_2$$

$$n_1 + 2*n_2 + 1 = n_0 + n_1 + n_2 \implies n_0 = n_2 + 1$$

Common operations

There are a variety of different operations that can be performed on binary trees. Some are mutator operations, while others simply return useful information about the tree.

Insertion

Nodes can be inserted into binary trees in between two other nodes or added after an external node. In binary trees, a node that is inserted is specified as to which child it is.

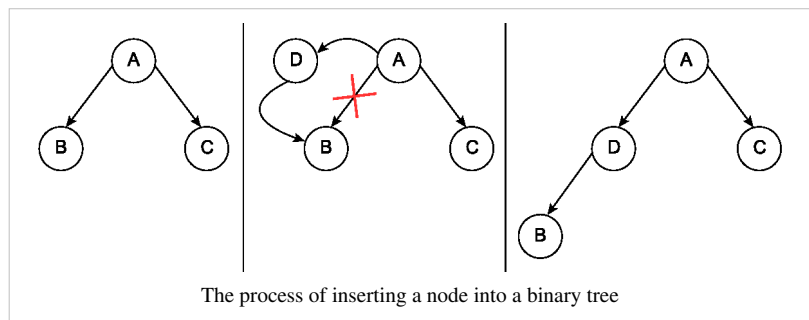
External nodes

Say that the external node being added on to is node A. To add a new node after node A, A assigns the new node as one of its children and the new node assigns node A as its parent.

Internal nodes

Insertion on internal nodes is slightly more complex than on external nodes. Say that the internal node is node A and that node B is the child of A. (If the insertion is to insert a right child, then B is the right child of A, and similarly with a left child insertion.) A assigns its child to the new node and the new node assigns its parent to A.

Then the new node assigns its child to B and B assigns its parent as the new node.

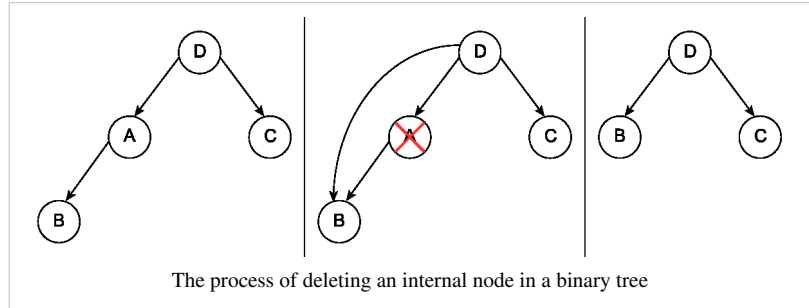


Deletion

Deletion is the process whereby a node is removed from the tree. Only certain nodes in a binary tree can be removed unambiguously.^[6]

Node with zero or one children

Say that the node to delete is node A. If a node has no children (external node), deletion is accomplished by setting the child of A's parent to null. If it has one child, set the parent of A's child to A's parent and set the child of A's parent to A's child.



Node with two children

In a binary tree, a node with two children cannot be deleted unambiguously.^[6] However, in certain binary trees these nodes *can* be deleted, including binary search trees.

Iteration

Often, one wishes to visit each of the nodes in a tree and examine the value there, a process called iteration or enumeration. There are several common orders in which the nodes can be visited, and each has useful properties that are exploited in algorithms based on binary trees:

- In-Order: Left child, Root, Right child.
- Pre-Order: Root, Left child, Right child
- Post-Order: Left Child, Right child, Root

Pre-order, in-order, and post-order traversal

Pre-order, in-order, and post-order traversal visit each node in a tree by recursively visiting each node in the left and right subtrees of the root.

Depth-first order

In depth-first order, we always attempt to visit the node farthest from the root that we can, but with the caveat that it must be a child of a node we have already visited. Unlike a depth-first search on graphs, there is no need to remember all the nodes we have visited, because a tree cannot contain cycles. Pre-order is a special case of this. See depth-first search for more information.

Breadth-first order

Contrasting with depth-first order is breadth-first order, which always attempts to visit the node closest to the root that it has not already visited. See breadth-first search for more information. Also called a *level-order traversal*.

Type theory

In type theory, a binary tree with nodes of type A is defined inductively as $T_A = \mu\alpha. 1 + A \times \alpha \times \alpha$.

Definition in graph theory

For each binary tree data structure, there is equivalent rooted binary tree in graph theory.

Graph theorists use the following definition: A binary tree is a connected acyclic graph such that the degree of each vertex is no more than three. It can be shown that in any binary tree of two or more nodes, there are exactly two more nodes of degree one than there are of degree three, but there can be any number of nodes of degree two. A **rooted binary tree** is such a graph that has one of its vertices of degree no more than two singled out as the root.

With the root thus chosen, each vertex will have a uniquely defined parent, and up to two children; however, so far there is insufficient information to distinguish a left or right child. If we drop the connectedness requirement, allowing multiple connected components in the graph, we call such a structure a forest.

Another way of defining binary trees is a recursive definition on directed graphs. A binary tree is either:

- A single vertex.
- A graph formed by taking two binary trees, adding a vertex, and adding an edge directed from the new vertex to the root of each binary tree.

This also does not establish the order of children, but does fix a specific root node.

Combinatorics

In combinatorics one considers the problem of counting the number of full binary trees of a given size. Here the trees have no values attached to their nodes (this would just multiply the number of possible trees by an easily determined factor), and trees are distinguished only by their structure; however the left and right child of any node are distinguished (if they are different trees, then interchanging them will produce a tree distinct from the original one). The size of the tree is taken to be the number n of internal nodes (those with two children); the other nodes are leaf nodes and there are $n + 1$ of them. The number of such binary trees of size n is equal to the number of ways of fully parenthesizing a string of $n + 1$ symbols (representing leaves) separated by n binary operators (representing internal nodes), so as to determine the argument subexpressions of each operator. For instance for $n = 3$ one has to parenthesize a string like $X * X * X * X$, which is possible in five ways:

$$((X*X)*X)*X, \quad (X*(X*X))*X, \quad (X*X)*(X*X), \quad X*((X*X)*X), \quad X*(X*(X*X)).$$

The correspondence to binary trees should be obvious, and the addition of redundant parentheses (around an already parenthesized expression or around the full expression) is disallowed (or at least not counted as producing a new possibility).

There is a unique binary tree of size 0 (consisting of a single leaf), and any other binary tree is characterized by the pair of its left and right children; if these have sizes i and j respectively, the full tree has size $i + j + 1$. Therefore the number C_n of binary trees of size n has the following recursive description $C_0 = 1$, and $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$ for any positive integer n . It follows that C_n is the Catalan number of index n .

The above parenthesized strings should not be confused with the set of words of length $2n$ in the Dyck language, which consist only of parentheses in such a way that they are properly balanced. The number of such strings satisfies the same recursive description (each Dyck word of length $2n$ is determined by the Dyck subword enclosed by the initial '(' and its matching ')' together with the Dyck subword remaining after that closing parenthesis, whose lengths

$2i$ and $2j$ satisfy $i + j + 1 = n$); this number is therefore also the Catalan number C_n . So there are also five Dyck words of length 10:

$()()(), ()(()), (())(), (())(), (((()))).$

These Dyck words do not correspond in an obvious way to binary trees. A bijective correspondence can nevertheless be defined as follows: enclose the Dyck word in an extra pair of parentheses, so that the result can be interpreted as a Lisp list expression (with the empty list $()$ as only occurring atom); then the dotted-pair expression for that proper list is a fully parenthesized expression (with NIL as symbol and '.' as operator) describing the corresponding binary tree (which is in fact the internal representation of the proper list).

The ability to represent binary trees as strings of symbols and parentheses implies that binary trees can represent the elements of a free magma on a singleton set.

Methods for storing binary trees

Binary trees can be constructed from programming language primitives in several ways.

Nodes and references

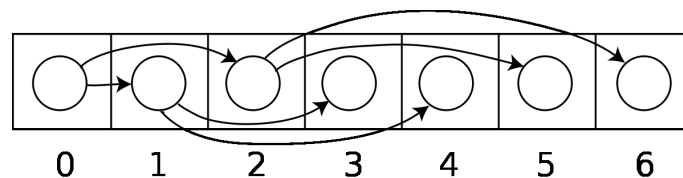
In a language with records and references, binary trees are typically constructed by having a tree node structure which contains some data and references to its left child and its right child. Sometimes it also contains a reference to its unique parent. If a node has fewer than two children, some of the child pointers may be set to a special null value, or to a special sentinel node.

In languages with tagged unions such as ML, a tree node is often a tagged union of two types of nodes, one of which is a 3-tuple of data, left child, and right child, and the other of which is a "leaf" node, which contains no data and functions much like the null value in a language with pointers.

Arrays

Binary trees can also be stored in breadth-first order as an implicit data structure in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index i , its children are found at indices $2i + 1$ (for the left child) and $2i + 2$ (for the right), while its parent (if any) is found at index $\left\lfloor \frac{i - 1}{2} \right\rfloor$ (assuming the root has index zero). This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it is expensive to grow and wastes space proportional to $2^h - n$ for a tree of depth h with n nodes.

This method of storage is often used for binary heaps. No space is wasted because nodes are added in breadth-first order.



Encodings

Succinct encodings

A succinct data structure is one which takes the absolute minimum possible space, as established by information theoretical lower bounds. The number of different binary trees on n nodes is C_n , the n th Catalan number (assuming we view trees with identical *structure* as identical). For large n , this is about 4^n ; thus we need at least about $\log_2 4^n = 2n$ bits to encode it. A succinct binary tree therefore would occupy only 2 bits per node.

One simple representation which meets this bound is to visit the nodes of the tree in preorder, outputting "1" for an internal node and "0" for a leaf. [7] If the tree contains data, we can simply simultaneously store it in a consecutive array in preorder. This function accomplishes this:

```
function EncodeSuccinct(node n, bitstring structure, array data) {
  if n = nil then
    append 0 to structure;
  else
    append 1 to structure;
    append n.data to data;
    EncodeSuccinct(n.left, structure, data);
    EncodeSuccinct(n.right, structure, data);
}
```

The string *structure* has only $2n + 1$ bits in the end, where n is the number of (internal) nodes; we don't even have to store its length. To show that no information is lost, we can convert the output back to the original tree like this:

```
function DecodeSuccinct(bitstring structure, array data) {
  remove first bit of structure and put it in b
  if b = 1 then
    create a new node n
    remove first element of data and put it in n.data
    n.left = DecodeSuccinct(structure, data)
    n.right = DecodeSuccinct(structure, data)
    return n
  else
    return nil
}
```

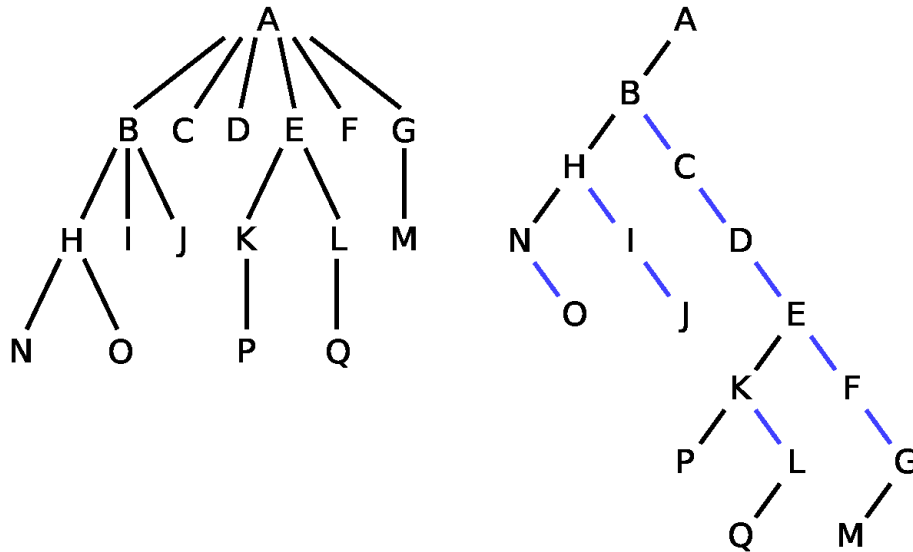
More sophisticated succinct representations allow not only compact storage of trees but even useful operations on those trees directly while they're still in their succinct form.

Encoding general trees as binary trees

There is a one-to-one mapping between general ordered trees and binary trees, which in particular is used by Lisp to represent general ordered trees as binary trees. To convert a general ordered tree to binary tree, we only need to represent the general tree in left child-sibling way. The result of this representation will be automatically binary tree, if viewed from a different perspective. Each node N in the ordered tree corresponds to a node N' in the binary tree; the *left* child of N' is the node corresponding to the first child of N , and the *right* child of N' is the node corresponding to N 's next sibling --- that is, the next node in order among the children of the parent of N . This binary tree representation of a general order tree is sometimes also referred to as a left child-right sibling binary tree (LCRS tree), or a doubly chained tree, or a Filial-Heir chain.

One way of thinking about this is that each node's children are in a linked list, chained together with their *right* fields, and the node only has a pointer to the beginning or head of this list, through its *left* field.

For example, in the tree on the left, A has the 6 children {B,C,D,E,F,G}. It can be converted into the binary tree on the right.



The binary tree can be thought of as the original tree tilted sideways, with the black left edges representing *first child* and the blue right edges representing *next sibling*. The leaves of the tree on the left would be written in Lisp as:

```
((N O) I J) C D ((P) (Q)) F (M))
```

which would be implemented in memory as the binary tree on the right, without any letters on those nodes that have a left child.

Notes

- [1] "perfect binary tree" (<http://www.nist.gov/dads/HTML/perfectBinaryTree.html>). NIST. .
- [2] "complete binary tree" (<http://www.nist.gov/dads/HTML/completeBinaryTree.html>). NIST. .
- [3] Aaron M. Tenenbaum, et. al Data Structures Using C, Prentice Hall, 1990 ISBN 0-13-199746-7
- [4] Paul E. Black (ed.), entry for *data structure* in *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology. 15 December 2004. Online version (<http://xw2k.nist.gov/dads/HTML/balancedtree.html>) Accessed 2010-12-19.
- [5] Mehta, Dinesh; Sartaj Sahni (2004). *Handbook of Data Structures and Applications*. Chapman and Hall. ISBN 1-58488-435-5.
- [6] Dung X. Nguyen (2003). "Binary Tree Structure" (<http://www.clear.rice.edu/comp212/03-spring/lectures/22/>). rice.edu. . Retrieved December 28, 2010.
- [7] http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L12/lecture12.pdf

References

- Donald Knuth. *The art of computer programming vol 1. Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.3, especially subsections 2.3.1–2.3.2 (pp. 318–348).
- Kenneth A Berman, Jerome L Paul. *Algorithms: Parallel, Sequential and Distributed*. Course Technology, 2005. ISBN 0-534-42057-5. Chapter 4. (pp. 113–166).

External links

- flash actionscript 3 opensource implementation of binary tree (<http://www.dpd.nl/opensource>) — opensource library
- GameDev.net's article about binary trees (<http://www.gamedev.net/reference/programming/features/trees2/>)
- Binary Tree Proof by Induction (<http://www.brpreiss.com/books/opus4/html/page355.html>)

- Balanced binary search tree on array How to create bottom-up an Ahnentafel list, or a balanced binary search tree on array (<http://piergiu.wordpress.com/2010/02/21/balanced-binary-search-tree-on-array/>)

Binary search tree

Binary search tree		
Type	Tree	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

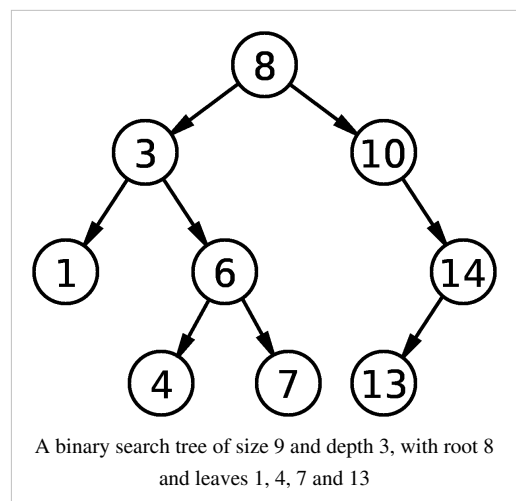
In computer science, a **binary search tree (BST)**, which may sometimes also be called an **ordered** or **sorted binary tree**, is a node-based binary tree data structure which has the following properties:^[1]

- 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.
- There must be no duplicate nodes.

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 keys. 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 key can be a recursive or iterative process.

We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful. If the key 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 key is found or the remaining subtree is null. If the searched key 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 pseudocode (iterative version, finds a BST node):

```
algorithm Find(key, root):
    current-node := root
    while current-node is not Nil do
        if current-node.key = key then
            return current-node
        else if key < current-node.key then
            current-node := current-node.left
        else
            current-node := current-node.right
```

The following recursive version is equivalent:

```
algorithm Find-recursive(key, node): // call initially with node = root
    if node.key = key then
        node
    else if key < node.key then
        Find-recursive(key, node.left)
    else
        Find-recursive(key, node.right)
```

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).

Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in C++:

```
/* 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);
}

```

or, alternatively, in Java:

```

public void InsertNode(Node n, double key) {
    if (key < n.key) {
        if (n.left == null) {
            n.left = new Node(key);
        }

        else {
            InsertNode(n.left, key);
        }
    }

    else if (key > n.key) {
        if (n.right == null) {
            n.right = new Node(key);
        }
        else {
            InsertNode(n.right, key);
        }
    }
}
}

```

The above *destructive* procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior 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:

```

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 $\Theta(\log n)$ 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 key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key 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 key.

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:

```
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) {
        // Choose not add 'data' if already present (an implementation
decision)
        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 = new TreeNode(data, null, null);
    } else {
        internalInsert(m_root, data);
    }
}

private static void internalInsert(Node node, int data) {
    // Choose not add 'data' if already present (an implementation
    // decision)
    if (data == node.getKey()) {
        return;
    } else if (data < node.getKey()) {
        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);
        }
    }
}

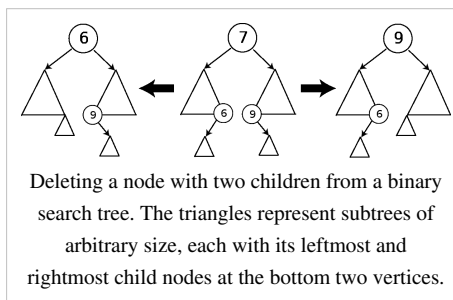
```

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:

```
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: # if both children are
present
            # 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)

```

Here is the code in C++.

```

template <typename T>
bool BST<T>::Delete(const T & itemToDelete)
{
    bool r_ans = Delete(root, itemToDelete);
    return r_ans;
}

template <typename T>
bool BST<T>::Delete(Node<T>* & ptr, const T& key)           //helper delete
function
{
    if (ptr==nullptr)
    {
        return false;           // item not in BST
    }

    if (key < ptr->data)
    {
        Delete(ptr->LeftChild, key);
    }
    else if (key > ptr->data)
    {
        Delete(ptr->RightChild, key);
    }
    else
    {
        Node<T> *temp;

        if (ptr->LeftChild==nullptr)
        {
            temp = ptr->RightChild;
            delete ptr;
            ptr = temp;
        }
        else if (ptr->RightChild==nullptr)
        {
            temp = ptr->LeftChild;
            delete ptr;
            ptr = temp;
        }
    }
}

```

```

        else //2 children
        {
            temp = ptr->RightChild;

            while(temp->LeftChild!=nullptr)
            {
                temp = temp->LeftChild;
            }
            ptr->data = temp->data;
            Delete(temp,temp->data);
        }
    }
}

```

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.

```

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 $O(n)$ time, since it must visit every node. This algorithm is also $O(n)$, so it is asymptotically optimal.

An in-order traversal algorithm for C is given below.

```

void in_order_traversal(struct Node *n, void (*cb)(void*))
{
    struct Node *cur, *pre;

    if(!n)
        return;

    cur = n;

    while(cur) {
        if(!cur->left) {
            cb(cur->val);
            cur = cur->right;
        } else {
            pre = cur->left;

```

```

        while(pre->right && pre->right != cur)
            pre = pre->right;

        if (!pre->right) {
            pre->right = cur;
            cur = cur->left;
        } else {
            pre->right = NULL;
            cb(cur->val);
            cur = cur->right;
        }
    }
}

```

An alternate recursion-free algorithm for in-order traversal using a stack and `goto` statements is provided below. The stack contains nodes whose right subtrees have yet to be explored. If a node has an unexplored left subtree (a condition tested at the `try_left` label), then the node is pushed (marking its right subtree for future exploration) and the algorithm descends to the left subtree. The purpose of the `loop_top` label is to avoid moving to the left subtree when popping to a node (as popping to a node indicates that its left subtree has already been explored.)

```

void in_order_traversal(struct Node *n, void (*cb)(void*))
{
    struct Node *cur;
    struct Stack *stack;

    if (!n)
        return;

    stack = stack_create();
    cur = n;

try_left:
    /* check for the left subtree */
    if (cur->left) {
        stack_push(stack, cur);
        cur = cur->left;
        goto try_left;
    }

loop_top:
    /* call callback */
    cb(cur->val);

    /* check for the right subtree */
    if (cur->right) {
        cur = cur->right;
        goto loop_top;
    }
}

```

```

        goto try_left;
    }

    cur = stack_pop(stack);
    if (cur)
        goto loop_top;

    stack_destroy(stack);
}

```

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:

```

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 $d \leq n - 1$, 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 n th 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)^[2] 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 variations.

Optimal binary search trees

If we do not 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 *agerasia* near the leaves.

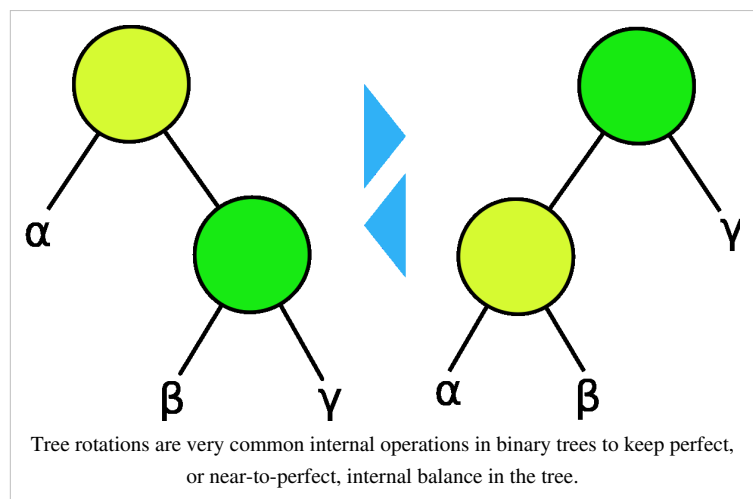
Such a tree might be compared with Huffman trees, which similarly seek to place frequently used items near the root in order to produce a dense information encoding; however, Huffman trees only store data elements in leaves and these elements need not be ordered.

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

- [1] Gilberg, R.; Forouzan, B. (2001), "8", *Data Structures: A Pseudocode Approach With C++*, Pacific Grove, CA: Brooks/Cole, p. 339, ISBN 0-534-95216-X
- [2] Heger, Dominique A. (2004), "A Disquisition on The Performance Behavior of Binary Search Tree Data Structures" (<http://www.cepis.org/upgrade/files/full-2004-V.pdf>), *European Journal for the Informatics Professional* **5** (5): 67–75,

Further reading

- Paul E. Black, Binary Search Tree (<http://www.nist.gov/dads/HTML/binarySearchTree.html>) at the NIST Dictionary of Algorithms and Data Structures.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "12: Binary search trees, 15.5: Optimal binary search trees". *Introduction to Algorithms* (2nd ed.). MIT Press & McGraw-Hill. pp. 253–272, 356–363. ISBN 0-262-03293-7.
- Jarc, Duane J. (3 December 2005). "Binary Tree Traversals" (<http://nova.umuc.edu/~jarc/idsv/lesson1.html>). *Interactive Data Structure Visualizations*. University of Maryland.
- Knuth, Donald (1997). "6.2.2: Binary Tree Searching". *The Art of Computer Programming*. **3: "Sorting and Searching"** (3rd ed.). Addison-Wesley. pp. 426–458. ISBN 0-201-89685-0.
- Long, Sean. "Binary Search Tree" (<http://employees.oneonta.edu/zhangs/PowerPointPlatform/resources/samples/binarysearchtree.ppt>) (PPT). *Data Structures and Algorithms Visualization - A PowerPoint Slides Based Approach*. SUNY Oneonta.
- Parlante, Nick (2001). "Binary Trees" (<http://cslibrary.stanford.edu/110/BinaryTrees.html>). *CS Education Library*. Stanford University.

External links

- Literate implementations of binary search trees in various languages (http://en.literateprograms.org/Category:Binary_search_tree) on LiteratePrograms
- Goleta, Maksim (27 November 2007). "Goletas.Collections" (<http://goletas.com/csharp-collections/>). *goletas.com*. Includes an iterative C# implementation of AVL trees.
- Jansens, Dana. "Persistent Binary Search Trees" (<http://cg.scs.carleton.ca/~dana/pbst>). Computational Geometry Lab, School of Computer Science, Carleton University. C implementation using GLib.
- Kovac, Kubo. "Binary Search Trees" (<http://people.ksp.sk/~kuko/bak/>) (Java applet). Korešpondenčný seminár z programovania.
- Madru, Justin (18 August 2009). "Binary Search Tree" (http://jdserver.homelinux.org/wiki/Binary_Search_Tree). *JDServer*. C++ implementation.
- Tarreau, Willy (2011). "Elastic Binary Trees (ebtree)" (<http://1wt.eu/articles/ebtree/>). *1wt.eu*.
- Binary Search Tree Example in Python (<http://code.activestate.com/recipes/286239/>)
- "References to Pointers (C++)" ([http://msdn.microsoft.com/en-us/library/1sf8shae\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/1sf8shae(v=vs.80).aspx)). *MSDN*. Microsoft. 2005. Gives an example binary tree implementation.
- Igushev, Eduard. "Binary Search Tree C++ implementation" (<http://igushev.com/implementations/binary-search-tree-cpp/>).
- Stromberg, Daniel. "Python Search Tree Empirical Performance Comparison" (<http://stromberg.dnsalias.org/~strombrg/python-tree-and-heap-comparison/>).

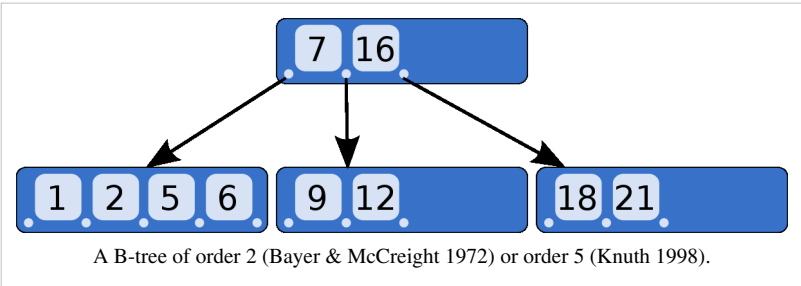
B-tree

B-tree		
Type	Tree	
Invented	1972	
Invented by	Rudolf Bayer, Edward M. McCreight	
Time complexity in big O notation		
	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

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 are 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 be 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 1998, p. 488) 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: $\log_2 1,000,000 = 19.931\dots$

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; $\log_b 1,000,000 = 3$ 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 that enough space is nearby such 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 ideas described above. In particular, a B-tree:

- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with an elegant recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.

Technical description

Terminology

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 1998, p. 483) 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 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 $\lceil m/2 \rceil$ children.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k-1$ keys.
5. All leaves appear in the same level, and carry information.

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

Let h be the height of the classic B-tree. Let $n > 0$ be the number of entries in the tree.^[5] Let m be the maximum number of children a node can have. Each node can have at most $m-1$ keys.

It can be shown (by induction for example) that a B-tree of height h with all its keys completely filled has $n = m^h - 1$ keys. Hence, the best case height of a B-tree is:

$$\lceil \log_m(n+1) \rceil.$$

Let d be the minimum number of children an internal (non-root) node can have. For an ordinary B-tree, $d = \lceil m/2 \rceil$.

The worst case height of a B-tree is:

$$\lfloor \log_d((n+1)/2) + 1 \rfloor$$

Comer (1979, p. 127) and Cormen et al. (year, pp. 383–384) give a slightly different expression for the worst case height (perhaps because the root node is considered to have height 0).

$$h \leq \lfloor \log_d\left(\frac{n+1}{2}\right) \rfloor.$$

Algorithms

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.

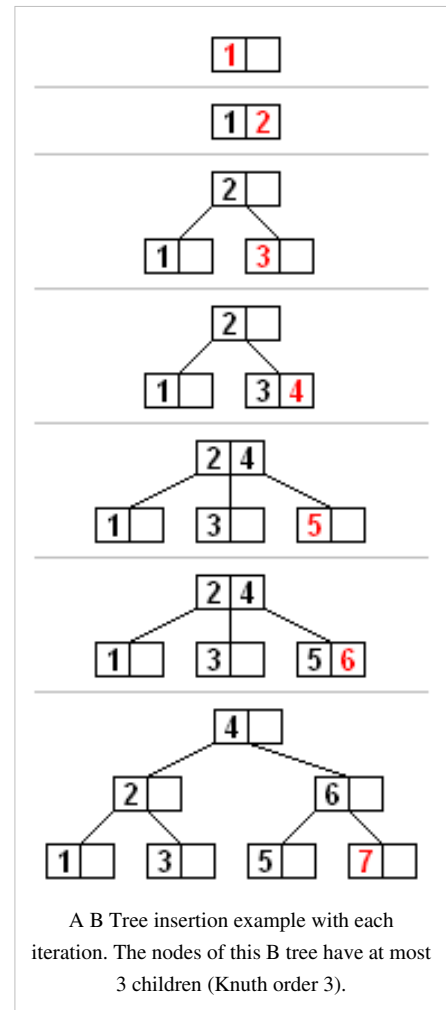
Insertion

All insertions start at a leaf node. To insert a new element, 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 (Mond & Raz 1985) 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 is 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 is in a leaf node, simply delete it from the node.
3. If underflow happens, rebalance the tree as described in section "Rebalancing after deletion" below.

Deletion from an internal node

Each element in an internal node acts as a separation value for two subtrees, therefore we need to find a replacement for separation. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is still greater than the separator. Both of those elements are in leaf nodes, and either one can be the new separator for the two subtrees. Algorithmically described below:

1. 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,^[6] 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^[7] 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,^[8] AIX (jfs2) 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^[9] 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 ^[10] for various strategies to implement node merging, and source code at.^[11])

United States Patent 5283894, granted In 1994, appears to show a way to use a 'Meta Access Method' ^[12] 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

- [1] Counted B-Trees (<http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html>), retrieved 2010-01-25
- [2] Knuth's video lectures from Stanford (<http://scpd.stanford.edu/knuth/index.jsp>)
- [3] Seagate Technology LLC, Product Manual: Barracuda ES.2 Serial ATA, Rev. F., publication 100468393, 2008 ([http://www.seagate.com/staticfiles/support/disc/manuals/NL35 Series &BC ES Series/Barracuda ES.2 Series/100468393f.pdf](http://www.seagate.com/staticfiles/support/disc/manuals/NL35%20Series%20&BC%20ES%20Series/Barracuda%20ES.2%20Series/100468393f.pdf)), page 6
- [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] If n is zero, then no root node is needed, so the height of an empty tree is not well defined.
- [6] 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.
- [7] Two of these were reserved for special purposes, so only 4078 could actually represent disk blocks (clusters).
- [8] Mark Russinovich. "Inside Win2K NTFS, Part 1" (<http://msdn2.microsoft.com/en-us/library/ms995846.aspx>). Microsoft Developer Network. Archived (<http://web.archive.org/web/20080413181940/http://msdn2.microsoft.com/en-us/library/ms995846.aspx>) from the original on 13 April 2008. . Retrieved 2008-04-18.
- [9] "Efficient locking for concurrent operations on B-trees" (<http://portal.acm.org/citation.cfm?id=319663&dl=GUIDE&coll=GUIDE&CFID=61777986&CFTOKEN=74351190>). Portal.acm.org. doi:10.1145/319628.319663. . Retrieved 2012-06-28.
- [10] <http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA232287&Location=U2&doc=GetTRDoc.pdf>
- [11] "Downloads - high-concurrency-btree - High Concurrency B-Tree code in C - Google Project Hosting" (<http://code.google.com/p/high-concurrency-btree/downloads/list>). Code.google.com. . Retrieved 2012-06-28.
- [12] Lockless Concurrent B+Tree (<http://www.freepatentsonline.com/5283894.html>)

References

- Bayer, R.; McCreight, E. (1972), "Organization and Maintenance of Large Ordered Indexes" (http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/Bayer_hist.pdf), *Acta Informatica* **1** (3): 173–189
- Comer, Douglas (June 1979), "The Ubiquitous B-Tree", *Computing Surveys* **11** (2): 123–137, doi:10.1145/356770.356776, ISSN 0360-0300.
- Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2001), *Introduction to Algorithms* (Second ed.), MIT Press and McGraw-Hill, pp. 434–454, ISBN 0-262-03293-7. Chapter 18: B-Trees.
- Folk, Michael J.; Zoellick, Bill (1992), *File Structures* (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4
- Knuth, Donald (1998), *Sorting and Searching*, The Art of Computer Programming, **Volume 3** (Second ed.), Addison-Wesley, ISBN 0-201-89685-0. Section 6.2.4: Multiway Trees, pp. 481–491. Also, pp. 476–477 of section 6.2.3 (Balanced Trees) discusses 2-3 trees.
- Mond, Yehudit; Raz, Yoav (1985), "Concurrency Control in B+-Trees Databases Using Preparatory Operations" (<http://www.informatik.uni-trier.de/~ley/db/conf/vldb/MondR85.html>), *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases*: 331–334.

Original papers

- Bayer, Rudolf; McCreight, E. (July 1970), *Organization and Maintenance of Large Ordered Indices*, **Mathematical and Information Sciences Report No. 20**, Boeing Scientific Research Laboratories.
- Bayer, Rudolf (1971), "Binary B-Trees for Virtual Memory", Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California. November 11–12, 1971.

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.bluerwhite.org/btree>)
- NIST's Dictionary of Algorithms and Data Structures: B-tree (<http://www.nist.gov/dads/HTML/btree.html>)

- 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>)
 - Open Data Structures - Section 14.2 - B-Trees (http://opendatastructures.org/versions/edition-0.1e/ods-java/14_2_B_Trees.html)
-

AVL tree

AVL tree		
Type	Tree	
Invented	1962	
Invented by	G. M. Adelson-Velskii and E. M. Landis	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

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; if at any time they differ by more than one, rebalancing is done to restore this property. 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]

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] Similar to red-black trees, AVL trees are in general not weight-balanced;^[4] that is, sibling nodes can have hugely differing numbers of descendants.

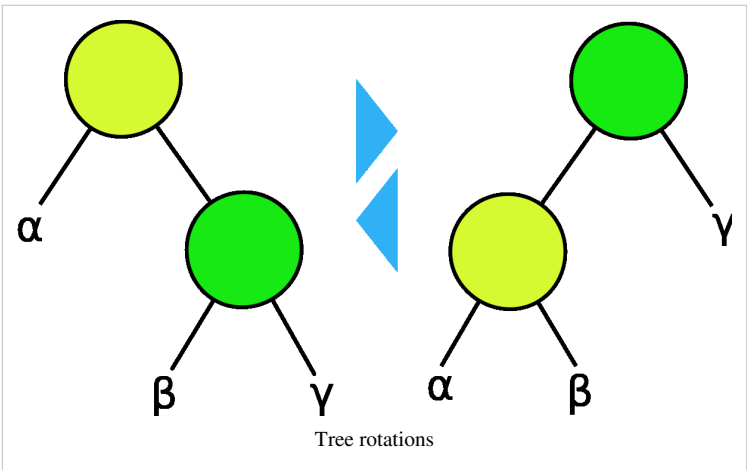
Operations

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.

Searching

Lookup in an AVL tree is performed exactly like in any 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 there are $n-1$ 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. The balance factor is calculated as follows: $\text{balanceFactor} = \text{height}(\text{left-subtree}) - \text{height}(\text{right-subtree})$. For each node checked, if the balance factor remains -1 , 0 , or $+1$ then no rotations are necessary. However, if balance factor becomes less than -1 or greater than $+1$, 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.

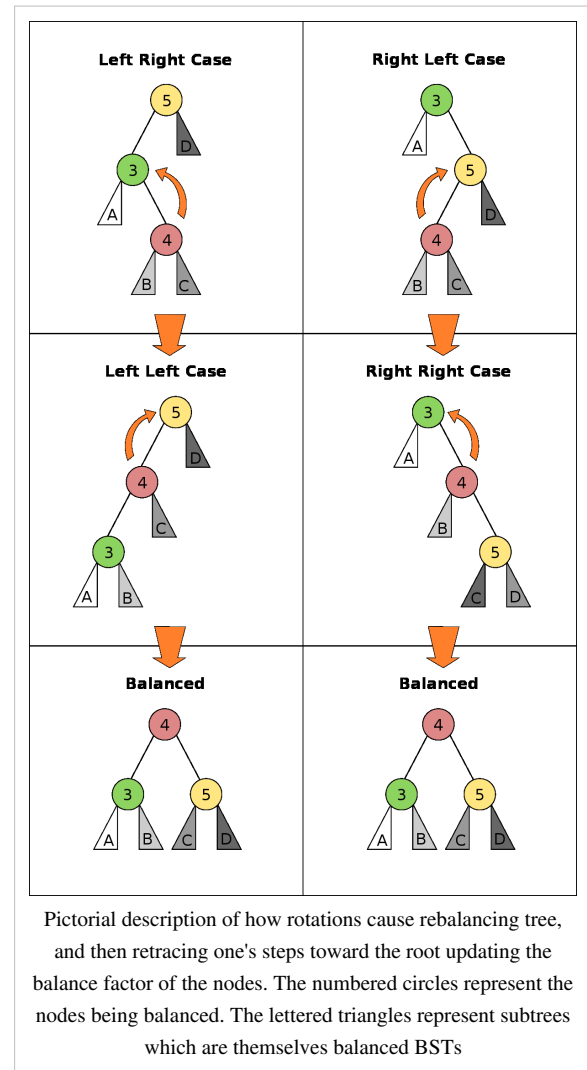
There are four cases which need to be considered, of which two are symmetric to the other two. Let P be the root of the unbalanced subtree, with R and L denoting the right and left children of P respectively.

Right-Right case and Right-Left case:

- If the balance factor of P is -2 then the right subtree outweighs 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:

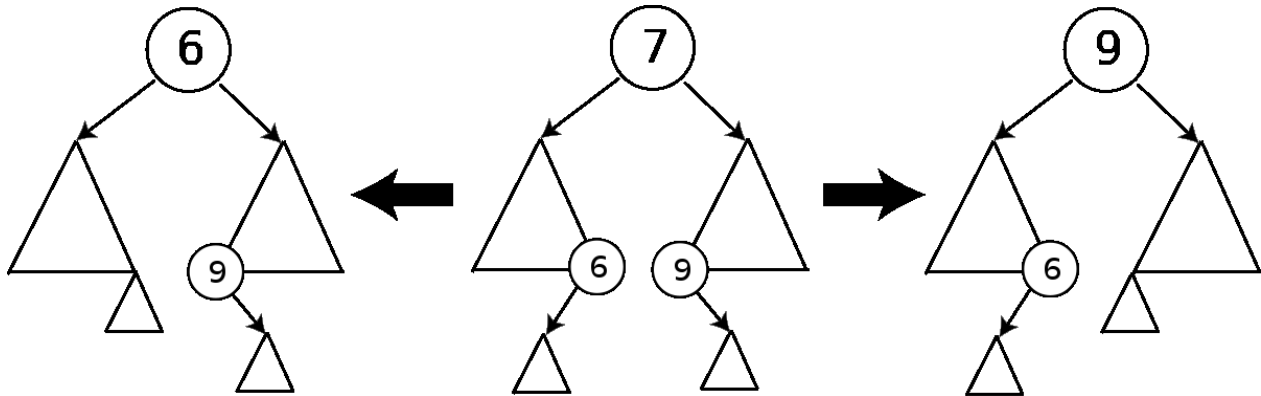
- If the balance factor of P is 2 , then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must be checked. The right rotation with P as the root is necessary.
 - If the balance factor of L is $+1$, a **single right rotation** (with P as the 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 sub tree (in order predecessor) or the smallest in its right sub tree (in order successor), and remove that node. The node that was found as a replacement has at most one sub tree. 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 retracing needs to continue. 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 retracing 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:^{[5][6]}

$$\log_{\varphi}(\sqrt{5}(n+2))-2 = \frac{\log_2(\sqrt{5}(n+2))}{\log_2(\varphi)}-2 = \log_{\varphi}(2) \cdot \log_2(\sqrt{5}(n+2))-2 \approx 1.44 \log_2(n+2)-0.328$$

where φ is the golden ratio.

- A red-black tree's height is at most $2 \log_2(n+1)$ ^[7]

AVL trees are more rigidly balanced than red-black trees, leading to slower insertion and removal but faster retrieval.

References

- [1] Robert Sedgewick, *Algorithms*, Addison-Wesley, 1983, ISBN 0-201-06672-6, page 199, chapter 15: Balanced Trees.
- [2] Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences* **146**: 263–266. **(Russian)** English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259–1263, 1962.
- [3] Pfaff, Ben (June 2004). "Performance Analysis of BSTs in System Software" (<http://www.stanford.edu/~blp/papers/libavl.pdf>) (PDF). Stanford University. .
- [4] AVL trees are not weight-balanced? (<http://cs.stackexchange.com/questions/421/avl-trees-are-not-weight-balanced>)
- [5] Burkhard, Walt (Spring 2012). "AVL Dictionary Data Type Implementation" (<http://ieng6.ucsd.edu/~cs100s/public/Notes/CS100s12.pdf>). *Advanced Data Structures*. La Jolla: A.S. Soft Reserves (<http://softreserves.ucsd.edu/>), UC San Diego. p. 103. .
- [6] Knuth, Donald E. (2000). *Sorting and searching* (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. pp. 460. ISBN 0-201-89685-0.
- [7] Proof of asymptotic bounds

Further reading

- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 458–475 of section 6.2.3: Balanced Trees.

External links

- xdg library (<https://github.com/vilkov/libxdg/wiki>) by Dmitriy Vilkov: Serializable straight C-implementation could easily be taken from this library under GNU-LGPL and AFL v2.0 licenses.
- Description from the Dictionary of Algorithms and Data Structures (<http://www.nist.gov/dads/HTML/avltree.html>)
- Python Implementation (<http://github.com/pgrafv/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 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>)
- PHP Implementation (<https://github.com/mondrake/Rbppavl>)
- C++ implementation which can be used as an array (<http://www.codeproject.com/Articles/12347/AVL-Binary-Tree-for-C>)
- Self balancing AVL tree with Concat and Split operations (<http://code.google.com/p/self-balancing-avl-tree/>)

Red–black tree

Red–black tree		
Type	Tree	
Invented	1972	
Invented by	Rudolf Bayer	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

A **red–black tree** is a type of self-balancing binary search tree, a data structure used in computer science, typically used to implement associative arrays.

Since it is a balanced tree, it guarantees searching, insertion, and deletion in $O(\log n)$ time, where n is the total number of elements in the tree.^[1]

History

The original structure was invented in 1972 by Rudolf Bayer^[2] and named "symmetric binary B-tree," but acquired its modern name in a paper in 1978 by Leonidas J. Guibas and Robert Sedgwick.^[3]

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.

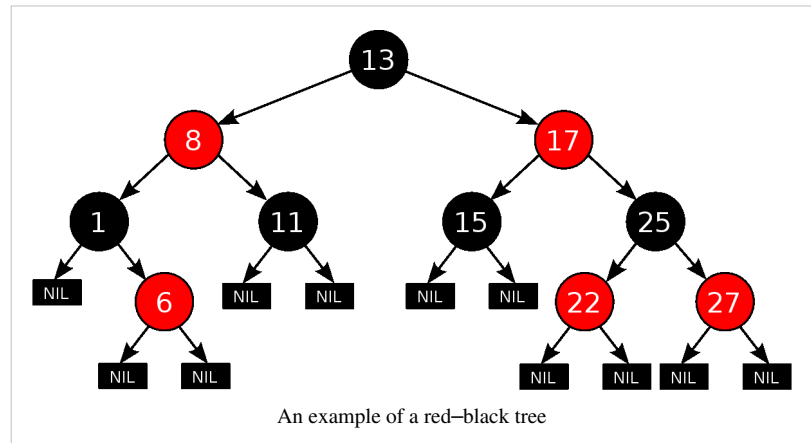
The leaf nodes of red–black trees do not contain data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red–black trees if the leaves really are explicit nodes. To save memory, sometimes 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 (that is: in the order 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

In addition to the requirements imposed on a binary search trees, with red–black trees:

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted. 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 (NIL) are black. (All leaves are 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 height-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. Red-black trees are in general not weight-balanced^[4], that is sibling nodes can have hugely differing numbers of descendants.

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.

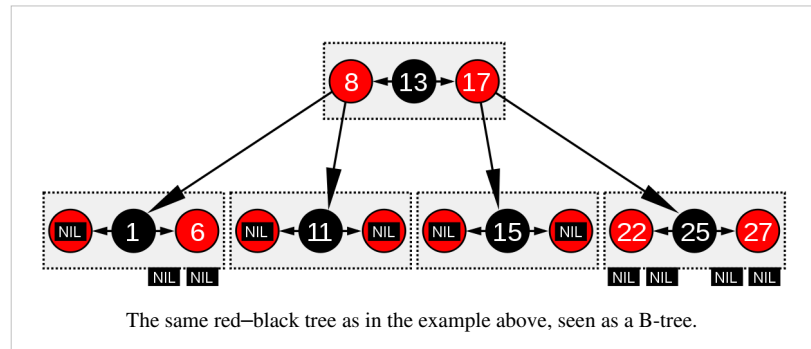
The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black nodes, by property 5, this shows that no path is more than twice as long as any other path.

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.

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, Sedgwick introduced a simpler version of the red–black tree called the left-leaning red-black tree^[5] 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,^[6] or 2-4 trees,^[5] for any sequence of operations. The 2-4 tree isometry was described in 1978 by Sedgwick. 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 as any 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:

```
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.

```
void insert_case1(struct node *n)
{
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(n);
}
```

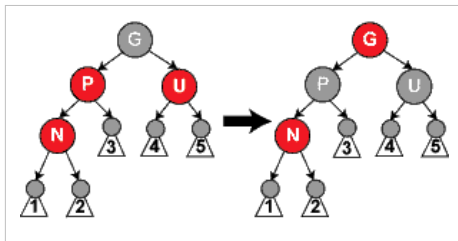
Case 2: The current node's parent **P** is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid. property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because the current 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.

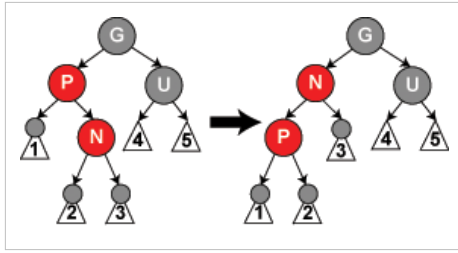
```

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);
    }
}

```

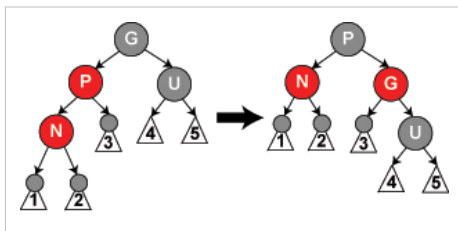
Note: In the remaining cases, it is assumed that the parent node **P** is the left child of its parent. If it is the right child, *left* and *right* should be reversed throughout cases 4 and 5. The code samples take care of this.



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.

```
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.

```
void insert_case5(struct node *n)
{
    struct node *g = grandparent(n);

    n->parent->color = BLACK;
    g->color = RED;
}
```

```

    if (n == n->parent->left)
        rotate_right(g);
    else
        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 property 4. (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 violate 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 have 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 **M** and **C** were black, then **P**'s one subtree which included **M** counted 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:


```

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).

```

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:

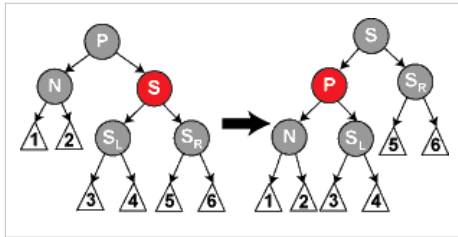
Case 1: **N** is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved.

```

void delete_case1(struct node *n)
{
    if (n->parent != NULL)
        delete_case2(n);
}

```

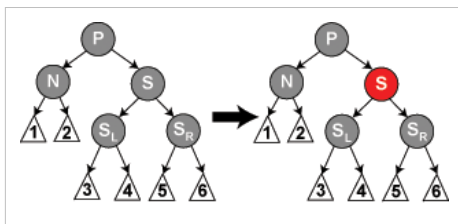
Note: In cases 2, 5, and 6, we assume **N** is the left child of its parent **P**. If it is the right child, *left* and *right* should be reversed throughout these three cases. Again, the code examples take both cases into account.



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**.

```
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.

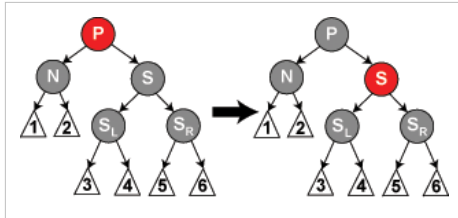
```
void delete_case3(struct node *n)
{
    struct node *s = sibling(n);

    if ((n->parent->color == BLACK) &&
        (s->color == BLACK) &&
        (s->left->color == BLACK) &&
        (s->right->color == BLACK)) {
```

```

        s->color = RED;
        delete_case1(n->parent);
    } else
        delete_case4(n);
}

```



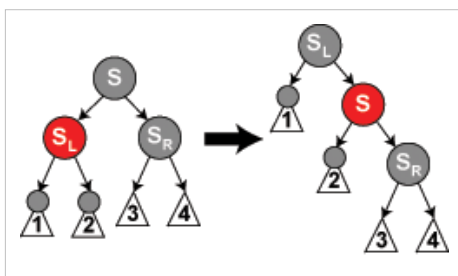
Case 4: **S** and **S**'s children are black, but **P** is red. In this case, we simply exchange the colors of **S** and **P**. This does not affect the number of black nodes on paths going through **S**, but it does add one to the number of black nodes on paths going through **N**, making up for the deleted black node on those paths.

```

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**.)

```

void delete_case5(struct node *n)
{
    struct node *s = sibling(n);

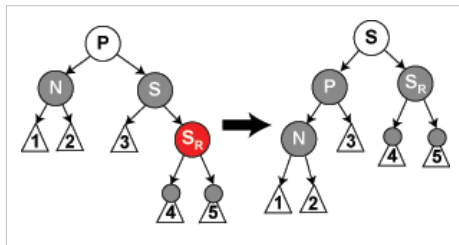
    if (s->color == BLACK) { /* this if statement is trivial,

```

```

due to case 2 (even though case 2 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.

```

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 $2^{bh(v')-1} - 1$ 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 $h(\text{root})/2$. By the lemma we get:

$$n \geq 2^{\frac{h(\text{root})}{2}} - 1 \leftrightarrow \log_2(n+1) \geq \frac{h(\text{root})}{2} \leftrightarrow h(\text{root}) \leq 2 \log_2(n+1).$$

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.^[7]

Notes

- [1] John Morris. "Red-Black Trees" (http://www.cs.auckland.ac.nz/~jmor159/PLDS210/red_black.html). .>
- [2] Rudolf Bayer (1972). "Symmetric binary B-Trees: Data structure and maintenance algorithms" ([http://www.springerlink.com/content/](http://www.springerlink.com/content/qh51m2014673513j/)[qh51m2014673513j/](http://www.springerlink.com/content/qh51m2014673513j/)). *Acta Informatica* **1** (4): 290–306. doi:10.1007/BF00289509. .
- [3] Leonidas J. Guibas and Robert Sedgwick (1978). "A Dichromatic Framework for Balanced Trees" (<http://doi.ieeecomputersociety.org/10.1109/SFCS.1978.3>). *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. pp. 8–21. doi:10.1109/SFCS.1978.3. .
- [4] Not all Red-Black trees are balanced? (<http://cs.stackexchange.com/questions/342/not-all-red-black-trees-are-balanced>)
- [5] <http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>
- [6] <http://www.cs.princeton.edu/courses/archive/fall08/cos226/lectures/10BalancedTrees-2x2.pdf>
- [7] H. Park and K. Park (2001). "Parallel algorithms for red–black trees" (<http://www.sciencedirect.com/science/article/pii/S0304397500002875>). *Theoretical computer science* (Elsevier) **262** (1–2): 415–435. doi:10.1016/S0304-3975(00)00287-5. .

References

- Mathworld: Red–Black Tree (<http://mathworld.wolfram.com/Red-BlackTree.html>)
- San Diego State University: CS 660: Red–Black tree notes (<http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html#RTFToC2>), by Roger Whitney
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Chapter 13: Red–Black Trees, pp. 273–301.
- Pfaff, Ben (June 2004). "Performance Analysis of BSTs in System Software" (<http://www.stanford.edu/~blp/papers/libavl.pdf>) (PDF). Stanford University.
- Okasaki, Chris. "Red–Black Trees in a Functional Setting" (<http://www.eecs.usma.edu/webs/people/okasaki/jfp99.ps>) (PS).

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_rbtrees.aspx)
- C code for Red–Black Trees (<http://github.com/fbuihuu/libtree>)
- Red–Black Tree in GNU libavl C library by Ben Pfaff (http://www.stanford.edu/~blp/avl/libavl.html/Red_002dBlack-Trees.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/TreeBidiMap.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>)
- Left-Leaning Red–Black Tree in ANS-Forth by Hugh Aguilar (<http://www.forth.org/novice.html>) See ASSOCIATION.4TH for the LLRB tree.
- An implementation of left-leaning red-black trees in C# (<http://blogs.msdn.com/b/delay/archive/2009/06/02/maintaining-balance-a-versatile-red-black-tree-implementation-for-net-via-silverlight-wpf-charting.aspx>)
- PPT slides demonstration of manipulating red black trees to facilitate teaching (<http://employees.oneonta.edu/zhangs/PowerPointplatform/>)
- OCW MIT Lecture by Prof. Erik Demaine on Red Black Trees (<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-10-red-black-trees-rotations-insertions-deletions/>) -
- 1 (<http://www.boyet.com/Articles/RedBlack1.html>) 2 (<http://www.boyet.com/Articles/RedBlack2.html>) 3 (<http://www.boyet.com/Articles/RedBlack3.html>) 4 (<http://www.boyet.com/Articles/RedBlack4.html>) 5 (<http://www.boyet.com/Articles/RedBlack5.html>), a C# Article series by Julian M. Bucknall.
- Open Data Structures - Chapter 9 - Red-Black Trees (http://opendatastructures.org/versions/edition-0.1e/ods-java/9_Red_Black_Trees.html)
- 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
- Red Black Tree API in the Linux kernel (<http://lwn.net/Articles/184495/>)

Hash function

A **hash function** is any algorithm or subroutine that maps large data sets of variable length, called *keys*, to smaller data sets of a fixed length. For example, a person's name, having a variable length, could be hashed to a single integer. The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, **checksums** or simply **hashes**.

Descriptions

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 (stable), i.e., if called twice on input that is "equal" (for example, 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: if two objects are equal, their hash codes must be the same. This is crucial to finding an element in a hash table quickly, because 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 collisions become more frequent as hash tables fill up. 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.

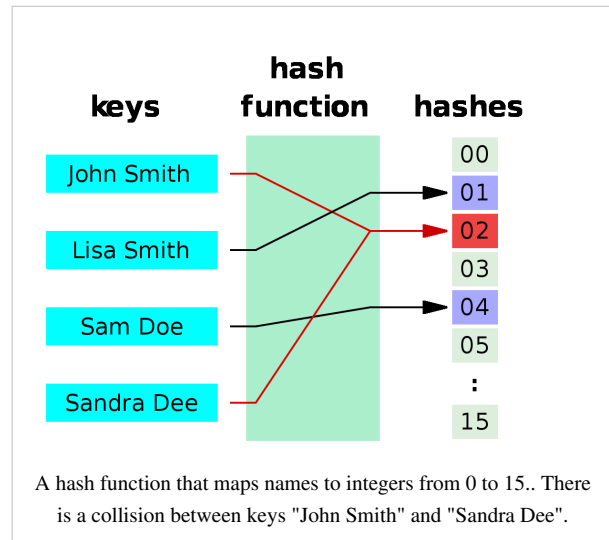
Hash functions are related to (and often confused with) checksums, check digits, fingerprints, randomization functions, error correcting codes, and cryptographic hash functions. Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimized differently. 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.

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. This is also used in file comparison.

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.^[2]

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.^[3]

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.).

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 data to be hashed, 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 value of n that may occur in the application. Depending on the function, the remainder may be uniform only for certain values of n , e.g. odd or prime numbers.

We can allow the table size n to not be a power of 2 and still not have to perform any remainder or division operation, as these computations are sometimes costly. For example, let n be significantly less than 2^b . Consider a pseudo random number generator (PRNG) function $P(\text{key})$ that is uniform on the interval $[0, 2^b - 1]$. A hash function uniform on the interval $[0, n-1]$ is $n P(\text{key})/2^b$. We can replace the division by a (possibly faster) right bit

shift: $nP(\text{key}) \gg b$.

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 checksums, 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.

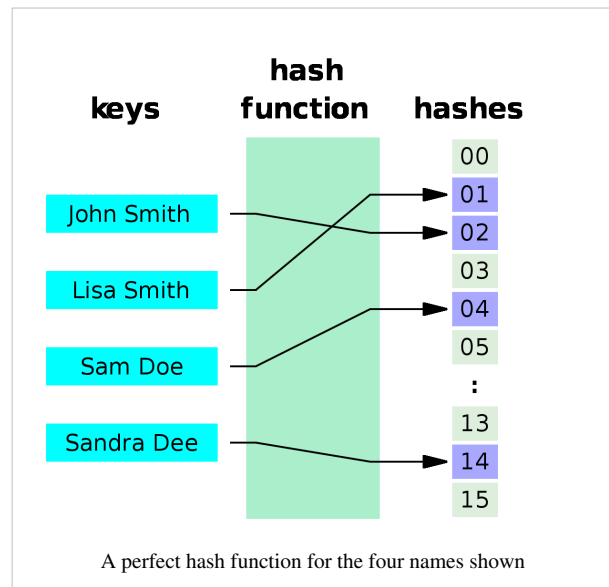
Other types of data can also use this perfect hashing scheme. For example, when mapping character strings between upper and lower case, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for "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.



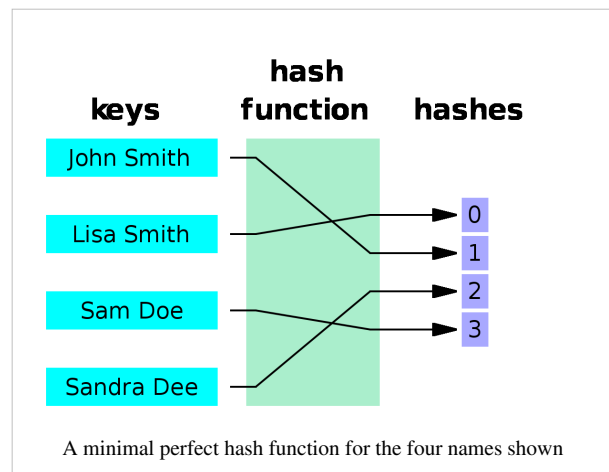
Minimal perfect hashing

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 $h = z \bmod n$ (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: $h = z \bmod n$ 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 m is 10000 or so, the division formula $(z \times m) \div M$, which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula $z \bmod M$, 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 ← S0;                                // Initialize the state.
for k in 1, 2, ..., m do                // Scan the input data units:
    S ← F(S, b[k]);                    // Combine data unit k into the state.

return G(S, n) // 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, S_0 can be 0, and $G(S,n)$ can be just $S \bmod n$. 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

```
if highbit(S) = 0 then
    return 2 * S + b
else
```

```
return (2 * S + b) ^ P
```

Here $highbit(S)$ denotes the most significant bit of S ; the '*' operator denotes unsigned integer multiplication with lost overflow; '^' is the bitwise exclusive or operation applied to words; and P is a suitable fixed word.^[4]

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 $k \bmod n$ 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 $k \cdot n$. 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—independently of the 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.^[5]

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.^[6] 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.

Hashing By Nonlinear Table Lookup

Tables of random numbers (such as 256 random 32 bit integers) can provide high-quality non-linear functions to be used as hash functions or other purposes such as cryptography. The key to be hashed would be split in 8-bit (one byte) parts and each part will be used as an index for the non-linear table. The table values will be added by arithmetic or XOR addition to the hash output value. Because the table is just 1024 bytes in size, it will fit into the cache of modern microprocessors and allow for very fast execution of the hashing algorithm. As the table value is on average much longer than 8 bit, one bit of input will affect nearly all output bits. This is different to multiplicative hash functions where higher-value input bits do not affect lower-value output bits.

This algorithm has proven to be very fast and of high quality for hashing purposes (especially hashing of integer number keys).

Efficient Hashing Of Strings

Modern microprocessors will allow for much faster processing, if 8-bit character Strings are not hashed by processing one character at a time, but by interpreting the string as an array of 32 bit or 64 bit integers and hashing/accumulating these "wide word" integer values by means of arithmetic operations (e.g. multiplication by constant and bit-shifting). The remaining characters of the string which are smaller than the word length of the CPU must be handled differently (e.g. being processed one character at a time).

This approach has proven to speed up hash code generation by a factor of five or more on modern microprocessors of a word size of 64 bit.

A far better approach for converting strings to a numeric value that avoids the problem with some strings having great similarity ("Aaaaaaaaaa" and "Aaaaaaaaaab") is to use a Cyclic redundancy check (CRC) of the string to compute a 32- or 64-bit value. While it is possible that two different strings will have the same CRC, the likelihood is very small and only requires that one check the actual string found to determine whether one has an exact match. The CRC approach works for strings of any length. CRCs will differ radically for strings such as "Aaaaaaaaaa" and "Aaaaaaaaaab".

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 **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^[7]
- 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

- [1] Knuth, Donald (1973). *The Art of Computer Programming, volume 3, Sorting and Searching*. pp. 506–542.
- [2] Bloom, Burton (July 1970). "Space/time trade-offs in hash coding with allowable errors" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.2080&rep=rep1&type=pdf>). *Communications of the ACM* **Volume 13** (Issue 7). doi:10.1145/362686.362692. .
- [3] "Robust Audio Hashing for Content Identification by Jaap Haitsma, Ton Kalker and Job Oostveen" (<http://citeseer.ist.psu.edu/rd/11787382,504088,1,0,25,Download/http://citeseer.ist.psu.edu/cache/papers/cs/25861/http:zSzzSzwwww.extra.research.philips.comzSznatlabzSdownloadzSaudiofpzSzcmbi01audiohashv1.0.pdf/haitsma01robust.pdf>)
- [4] Broder, A. Z. (1993). "Some applications of Rabin's fingerprinting method". *Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag. pp. 143–152.
- [5] Bret Mulvey, *Evaluation of CRC32 for Hash Tables* (<http://home.comcast.net/~bretm/hash/8.html>), in *Hash Functions* (<http://home.comcast.net/~bretm/hash/>). Accessed April 10, 2009.
- [6] Bret Mulvey, *Evaluation of SHA-1 for Hash Tables* (<http://home.comcast.net/~bretm/hash/9.html>), in *Hash Functions* (<http://home.comcast.net/~bretm/hash/>). Accessed April 10, 2009.
- [7] "Hash Functions" (<http://www.cse.yorku.ca/~oz/hash.html>). *cse.yorku.ca*. September 22, 2003. . Retrieved November 1, 2012. "the djb2 algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c."

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>)
- The Goulburn Hashing Function (<http://www.webcitation.org/query?url=http://www.geocities.com/drone115b/Goulburn06.pdf&date=2009-10-25+21:06:51>) (PDF) by Mayur Patel
- MIT's Introduction to Algorithms: Hashing 1 (<http://video.google.com/videoplay?docid=-727485696209877198&q=source:014117792397255896270&hl=en>) MIT OCW lecture Video
- MIT's Introduction to Algorithms: Hashing 2 (<http://video.google.com/videoplay?docid=2307261494964091254&q=source:014117792397255896270&hl=en>) MIT OCW lecture Video
- Hash Fuction Construction for Textual and Geometrical Data Retrieval (http://herakles.zcu.cz/~skala/PUBL/PUBL_2010/2010_WSEAS-Corfu_Hash-final.pdf) Latest Trends on Computers, Vol.2, pp. 483–489, CSCC conference, Corfu, 2010

Priority queue

In computer science, a **priority queue** is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

- *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)

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. For instance, one can keep all the elements in an unsorted list. Whenever the highest-priority element is requested, search through all elements for the one with the highest priority. (In big O notation: $O(1)$ insertion time, $O(n)$ pull time due to search.)

Usual implementation

To improve 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, when a self-balancing binary search tree is used, insertion and removal also take $O(\log n)$ time, although building trees from existing sequences of elements takes $O(n \log n)$ time; this is typical where one might already have access to these data structures, such as with third-party or standard libraries.

Note that from a computational-complexity standpoint, priority queues are congruent to sorting algorithms. See the next section for how efficient sorting algorithms can create efficient priority queues.

There are several specialized heap data structures that either supply additional operations or outperform these approaches. The binary heap uses $O(\log n)$ time for both operations, but also allow queries of the element of highest priority without removing it in constant time. Binomial heaps add several more operations, but require $O(\log n)$ time for requests. Fibonacci heaps can insert elements, query the highest priority element, and increase an element's priority in amortized constant time^[1] though deletions are still $O(\log n)$). Brodal queues can do this in worst-case constant time.

While relying on a heap is a common way to implement priority queues, for integer data, faster implementations exist. This can even apply to data-types that have a finite range, such as floats:

- When the set of keys is $\{1, 2, \dots, C\}$, a van Emde Boas tree would support 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 $O(2^{m/2})$, where m is the number of bits in the priority value.^[2]
- 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.^[3]

For applications that do many "peek" operations for every "extract-min" operation, the time complexity for peek actions 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 a constant cost, since the newly inserted element is compared only to the previously cached minimum element. For deletion, this at most adds an additional "peek" cost, which is typically cheaper than the deletion cost, so overall time complexity is not significantly impacted.

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^[4]:

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,^[5] 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 $O(n \lg \lg n)$ sort algorithm, one can 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. The Boost (C++ libraries) also have an implementation in the library heap.

Python's `heapq`^[6] 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`^[7] module, which implements a min-heap on top of any compatible data structure.

The Standard PHP Library extension contains the class `SplPriorityQueue`^[8].

Apple's Core Foundation framework contains a `CFBinaryHeap`^[9] structure, which implements a min-heap.

Applications

Bandwidth management

Priority queuing 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, although one also needs the ability to alter the priority of a particular vertex in the priority queue efficiently.

Huffman coding

Huffman coding requires one to repeatedly obtain the two lowest-frequency trees. A priority queue makes this efficient.

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

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 20: Fibonacci Heaps, pp.476–497. Third edition p518.
- [2] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84. IEEE Computer Society, 1975.
- [3] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 48(3):533–551, 1994
- [4] Mikkel Thorup. 2007. Equivalence between priority queues and sorting. J. ACM 54, 6, Article 28 (December 2007). DOI=10.1145/1314690.1314692 (<http://doi.acm.org/10.1145/1314690.1314692>)
- [5] <http://courses.csail.mit.edu/6.851/spring07/scribe/lec17.pdf>
- [6] <http://docs.python.org/library/heapq.html>
- [7] <http://golang.org/pkg/container/heap/>
- [8] <http://us2.php.net/manual/en/class.splpriorityqueue.php>
- [9] <http://developer.apple.com/library/mac/#documentation/CoreFoundation/Reference/CFBinaryHeapRef/Reference/reference.html>

Further reading

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 6.5: Priority queues, pp. 138–142.

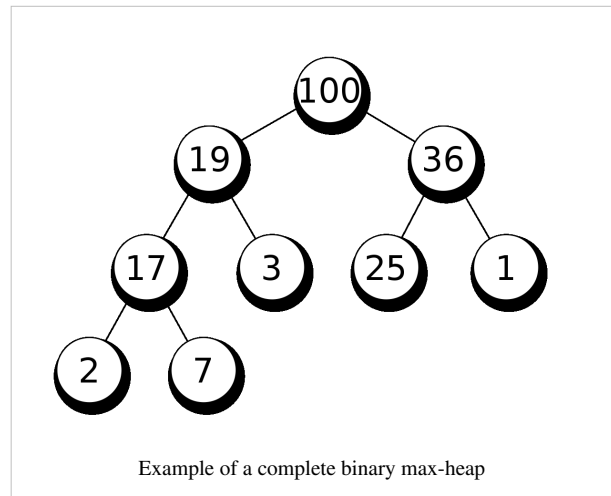
External links

- C++ reference for `std::priority_queue` (http://en.cppreference.com/w/cpp/container/priority_queue)
- Descriptions (<http://leekillough.com/heaps/>) by Lee Killough
- PQlib (<http://bitbucket.org/trijezdci/pqlib/src/>) - Open source Priority Queue library for C
- libpqueue (<http://github.com/vy/libpqueue>) 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/dsaa/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 A is a parent node of B then $\text{key}(A)$ is ordered with respect to $\text{key}(B)$ with the same ordering applying across the heap. Either the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node (this kind of heap is called *max heap*) or the keys of parent nodes are less than or equal to those of the children (*min heap*).

Note that, as shown in the graphic, there is no implied ordering between siblings or cousins and no implied sequence for an in-order traversal (as there would be in, e.g., a binary search tree). The heap relation mentioned above applies only between nodes and their immediate parents.



The maximum number of children each node can have depends on the type of heap, but in many types it is 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.

Implementation and operations

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
- (*a variant*) *create-heap*: create a heap out of given array of elements
- *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.

Different types of heaps implement the operations in different ways, but notably, insertion is often done by adding the new element at the end of the heap in the first available free space. This will tend to violate the heap property, and so the elements are then reordered until the heap property has been reestablished. Construction of a binary (or d-ary) heap out of given array of elements may be preformed faster than a sequence of consecutive insertions into originally empty heap using the classic Floyd's algorithm, with the worst-case number of comparisons equal to $2N - 2s_2(N) - e_2(N)$ (for a binary heap), where $s_2(N)$ is the sum of all digits of the binary representation of N and $e_2(N)$ is the exponent of 2 in the prime factorization of N .^[1]

Variants

- 2-3 heap
- Beap
- Binary heap
- Binomial heap
- Brodal queue
- D-ary heap
- Fibonacci heap
- Leftist heap
- Pairing heap
- Skew heap
- Soft heap
- Weak heap
- Leaf heap
- Radix heap
- Randomized meldable heap

Comparison of theoretic bounds for variants

The following time complexities^[2] 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 $\Theta(f)$ is asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	Binary ^[2]	Binomial ^[2]	Fibonacci ^[2]	Pairing ^[3]	Brodal*** ^[4]	RP ^[5]
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)$	$O(\log n)^*$
insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$O(\log n)^*$	$\Theta(1)$	$\Theta(1)^*$
merge	$\Theta(n)$	$O(\log n)^{**}$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$	$\Theta(1)$

(*)Amortized time

(**)Where n is the size of the larger heap

(***)Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with n elements can be constructed bottom-up in $O(n)$.^[6]

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.^[7]
- 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.

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. However, there is no standard support for the decrease/increase-key operation. See also `gheap`^[8] - STL-like generalized heap implementation in C++ with D-heap and B-heap support.
- The Java 2 platform (since version 1.5) provides the binary heap implementation with class `java.util.PriorityQueue<E>`^[9] in Java Collections Framework.
- Python has a `heapq`^[6] 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`^[10] distribution available on CPAN.
- The Go library contains a `heap`^[7] package with heap algorithms that operate on an arbitrary type that satisfied a given interface.
- Apple's Core Foundation library contains a `CFBinaryHeap`^[11] structure.

References

- [1] Suchenek, Marek A. (2012), "Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program", *Fundamenta Informaticae* (IOS Press) **120** (1): 75-92, doi:10.3233/FI-2012-751.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (1990): Introduction to algorithms. MIT Press / McGraw-Hill.
- [3] Iacono, John (2000), "Improved upper bounds for pairing heaps", *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science, **1851**, Springer-Verlag, pp. 63–77, doi:10.1007/3-540-44985-X_5
- [4] <http://www.cs.au.dk/~gerth/papers/soda96.pdf>
- [5] Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (2009). "Rank-pairing heaps" (<http://www.cs.princeton.edu/~sssix/papers/rp-heaps.pdf>). *SIAM J. Computing*: 1463–1485. .
- [6] Goodrich, Michael T.; Tamassia, Roberto (2004). "7.3.6. Bottom-Up Heap Construction". *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341.
- [7] Frederickson, Greg N. (1993), "An Optimal Algorithm for Selection in a Min-Heap" (http://ftp.cs.purdue.edu/research/technical_reports/1991/TR_91-027.pdf), *Information and Computation*, **104**, Academic Press, pp. 197–214, doi:10.1006/inco.1993.1030,
- [8] <https://github.com/valyala/gheap>
- [9] <http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html>
- [10] <http://search.cpan.org/perldoc?Heap>
- [11] <https://developer.apple.com/library/mac/#documentation/CoreFoundation/Reference/CFBinaryHeapRef/Reference/reference.html>

External links

- Heap (<http://mathworld.wolfram.com/Heap.html>) at Wolfram MathWorld

Binary heap

Binary Heap		
Type	Tree	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	N/A Operation	N/A Operation
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

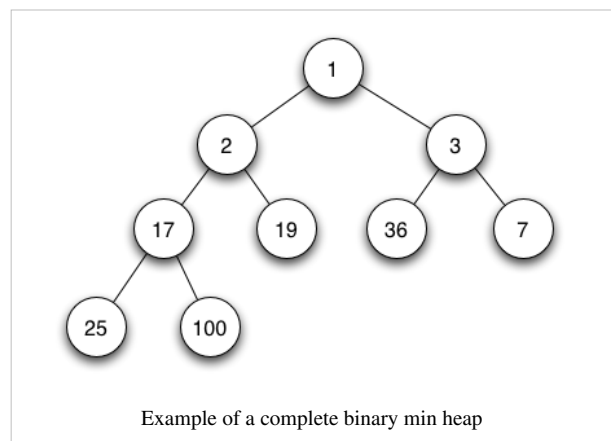
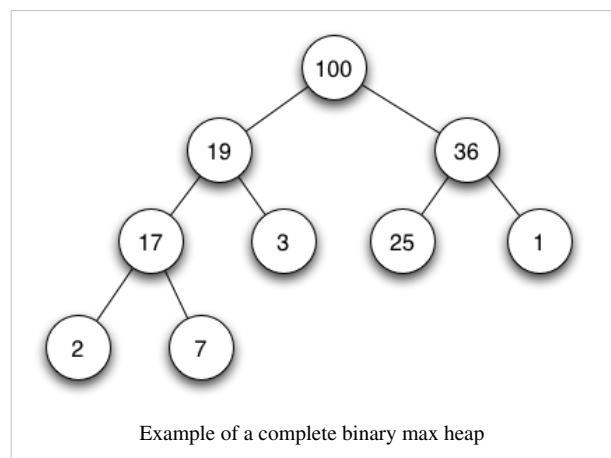
A **binary heap** is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:

- 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 $d = 2$.



Heap operations

Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take $O(\log n)$ time.

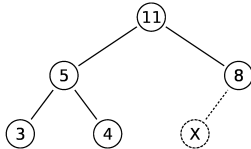
Insert

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*), by following this algorithm:

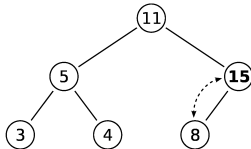
1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a time complexity of $O(\log n)$.

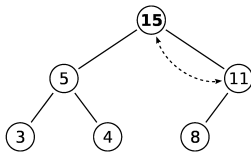
As an example, say we have a max-heap



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:



However the heap property is still violated since 15 is greater than 11, so we need to swap again:



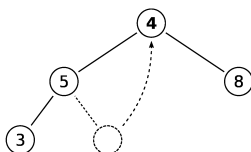
which is a valid max-heap. There is no need to check the children after this. Before we placed 15 on X, the heap was valid, meaning 11 is greater than 5. If 15 is greater than 11, and 11 is greater than 5, then 15 must be greater than 5, because of the transitive relation.

Delete

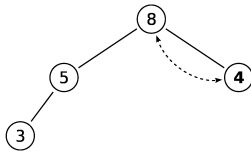
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, *cascade-down* and *extract-min/max*).

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^[3] (A, i):

$left \leftarrow 2i$

$right \leftarrow 2i + 1$

$largest \leftarrow i$

if $left \leq \text{heap_length}[A]$ **and** $A[left] > A[largest]$ **then**:

$largest \leftarrow left$

if $right \leq \text{heap_length}[A]$ **and** $A[right] > A[largest]$ **then**:

$largest \leftarrow right$

if $largest \neq i$ **then**:

swap $A[i] \leftrightarrow A[largest]$

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.

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or $O(\log n)$.

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. Since the height of the heap is $\lceil \lg(n) \rceil$, the number of nodes at

height h is $\leq \left\lceil 2^{(\lg n - h) - 1} \right\rceil = \left\lceil \frac{2^{\lg n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$. Therefore, the cost of heapifying all subtrees is:

$$\begin{aligned}
 \sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) &= O \left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right) \\
 &\leq O \left(n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} \right) \\
 &= O(n)
 \end{aligned}$$

This uses the fact that the given infinite series $h / 2^h$ converges to 2.

The exact value of the above (the worst-case number of comparisons during the heap construction) is known to be equal to:

$$2n - 2s_2(n) - e_2(n),^{[4]}$$

where $s_2(n)$ is the sum of all digits of the binary representation of n and $e_2(n)$ is the exponent of 2 in the prime factorization of n .

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 $\text{floor}(n/2) + 1, \text{floor}(n/2) + 2, \dots, n$ 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^[3](A):

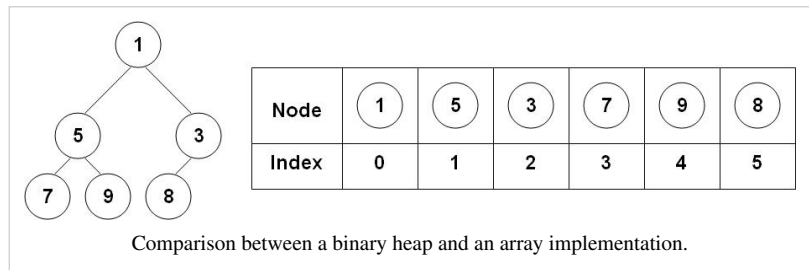
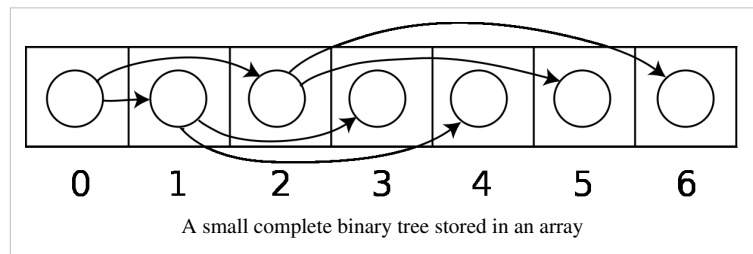
$\text{heap_length}[A] \leftarrow \text{length}[A]$

for $i \leftarrow \text{floor}(\text{length}[A]/2)$ **downto** 1 **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, sacrificing 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 $b, b+1, \dots, e$. The sift-down function extends the heap property to $b-1, b, b+1, \dots, e$. 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, \dots, 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 $a[i]$ 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] A heap on n elements can be merged with a heap on k elements using $O(\log n \log k)$ key comparisons, or, in case of a pointer-based implementation, in $O(\log n \log k)$ time.^[7] An algorithm for splitting a heap on n elements into two heaps on k and $n-k$ elements, respectively, based on a new view of heaps as an ordered collections of subheaps was presented in^[8]. The algorithm requires $O(\log n * \log n)$ comparisons. The view also presents a new and conceptually simple algorithm for merging heaps. 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.

It is possible to modify the heap structure to allow extraction of both the smallest and largest element in $O(\log n)$ time.^[9] 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.

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

$$\begin{aligned}
 &= \text{Number of nodes including } i - \text{Number of nodes through the previous level} - \text{One node for } i \text{ itself} \\
 &= (i + 1) - (2^l - 1) - 1 \\
 &= i + 1 - 2^l + 1 - 1 \\
 &= i - 2^l + 1
 \end{aligned}$$

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:-

$$\begin{aligned} &= \text{Total nodes in level } l - (\text{Total siblings on left} + 1) \\ &= (2^l) - (i - 2^l + 2) \\ &= 2^l + 2^l - i - 2 \\ &= 2^{l+1} - i - 2 \end{aligned}$$

So, index of 1st child of node i would be:-

$$\begin{aligned} &= i + \text{Total siblings on right} + 2 * \text{Total siblings on left} + 1 \\ &= i + (2^{l+1} - i - 2) + 2(i - 2^l + 1) + 1 \\ &= i + 2^{l+1} - i - 2 + 2i - 2^{l+1} + 2 + 1 \\ &= i - i + 2i + 2^{l+1} - 2^{l+1} - 2 + 2 + 1 \\ &= 2i + 1 [\text{Proved}] \end{aligned}$$

Intuitive proof

Although the mathematical approach proves this without doubt, 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 up to the root, will have exactly 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 .

References

- [1] "heapq – Heap queue algorithm" (<http://docs.python.org/library/heapq.html>). *Python Standard Library*. .
- [2] "Class PriorityQueue" (<http://download.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html>). *Java™ Platform Standard Ed. 6*. .
- [3] Cormen, T. H. & al. (2001), *Introduction to Algorithms* (2nd ed.), Cambridge, Massachusetts: The MIT Press, ISBN 0-07-013151-1
- [4] Suchenek, Marek A. (2012), "Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program", *Fundamenta Informaticae* (IOS Press) **120** (1): 75-92, doi:10.3233/FI-2012-751.
- [5] Poul-Henning Kamp. "You're Doing It Wrong" (<http://queue.acm.org/detail.cfm?id=1814327>). ACM Queue. June 11, 2010.
- [6] Chris L. Kuszmaul. "binary heap" (<http://nist.gov/dads/HTML/binaryheap.html>). Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 16 November 2009.
- [7] J.-R. Sack and T. Strothotte "An Algorithm for Merging Heaps" (<http://www.springerlink.com/content/k24440h5076w013q/>), Acta Informatica 22, 171-186 (1985).
- [8] . J.-R. Sack and T. Strothotte "A characterization of heaps and its applications" (<http://www.sciencedirect.com/science/article/pii/089054019090026E>) Information and Computation Volume 86, Issue 1, May 1990, Pages 69–86.
- [9] Atkinson, M.D., J.-R. Sack, N. Santoro, and T. Strothotte (1 October 1986). "Min-max heaps and generalized priority queues." (<http://cg.scs.carleton.ca/~morin/teaching/5408/refs/minmax.pdf>). Programming techniques and Data structures. Comm. ACM, 29(10): 996–1000. .

External links

- Binary Heap Applet (<http://people.ksp.sk/~kuko/bak/index.html>) by Kubo Kovac
- 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)
- C++ implementation of generalized heap with Binary Heap support (<https://github.com/valyala/gheap>)
- Open Data Structures - Section 10.1 - BinaryHeap: An Implicit Binary Tree (http://opendatastructures.org/versions/edition-0.1e/ods-java/10_1_BinaryHeap_Implicit_Bi.html)
- Ighushev, Eduard. "Binary Heap C++ implementation" (<http://igushev.com/implementations/binary-heap-cpp/>).

Leftist tree

In computer science, a **leftist tree** or **leftist heap** is a priority queue implemented with a variant of a binary heap. Every node has an *s-value* which is the distance to the nearest leaf. In contrast to a *binary heap*, a leftist tree attempts to be very unbalanced. In addition to the heap property, leftist trees are maintained so the right descendant of each node has the lower s-value.

The leftist tree was invented by Clark Allan Crane. The name comes from the fact that the left subtree is usually taller than the right subtree.

When inserting a new node into a tree, a new one-node tree is created and merged into the existing tree. To delete a minimum item, we remove the root and the left and right sub-trees are then merged. Both these operations take $O(\log n)$ time. For insertions, this is slower than binary heaps which support insertion in amortized constant time, $O(1)$ and $O(\log n)$ worst-case.

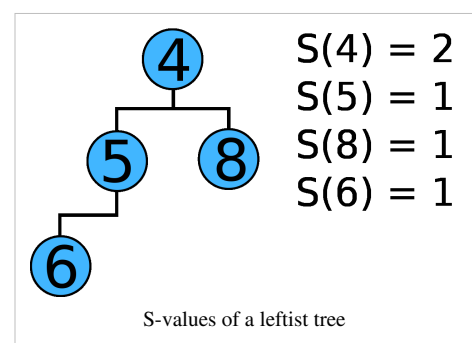
Leftist trees are advantageous because of their ability to merge quickly, compared to binary heaps which take $\Theta(n)$. In almost all cases, skew heaps have better performance.

Bias

The usual leftist tree is a *height-biased* leftist tree. However, other biases can exist, such as in the *weight-biased* leftist tree.

S-value

The s-value of a node is the distance from that node to the nearest leaf of the extended binary representation of the tree ^[1]. The extended representation (not shown) fills out the tree so that each node has 2 children (adding a total of 5 leaves here). The minimum distance to these leaves are marked in the diagram. Thus s-value of 4 is 2, since the closest leaf is that of 8 --if 8 were extended. The s-value of 5 is 1 since its extended representation would have one leaf itself.



Merging height biased leftist trees

Merging two nodes together depends on whether the tree is a min or max height biased leftist tree. For a min height biased leftist tree, set the higher valued node as its right child of the lower valued node. If the lower valued node already has a right child, then merge the higher valued node with the sub-tree rooted by the right child of the lower valued node.

After merging, the s-value of the lower valued node must be updated (see above section, s-value). Now check if the lower valued node has a left child. If it does not, then move the right child to the left. If it does have a left child, then the child with the highest s-value should go on the left.

Java code for merging a min height biased leftist tree

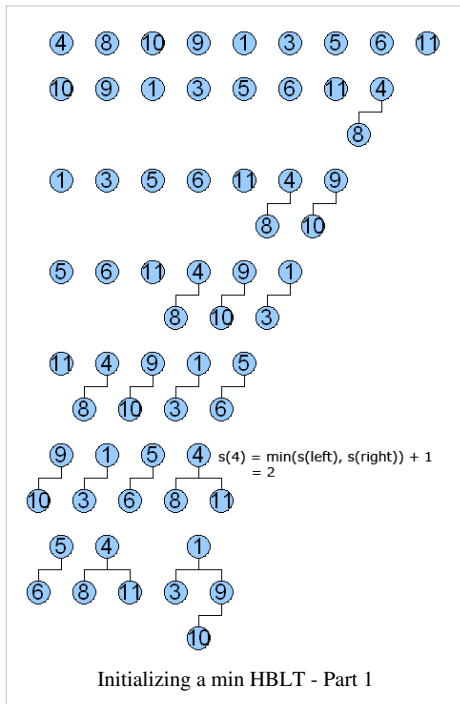
```
public Node merge(Node x, Node y) {
    if (x == null)
        return y;
    if (y == null)
        return x;

    // if this was a max height biased leftist tree, then the
    // next line would be: if(x.element < y.element)
    if (x.element.compareTo(y.element) > 0) {
        // x.element > y.element
        Node temp = x;
        x = y;
        y = temp;
    }

    x.rightChild = merge(x.rightChild, y);

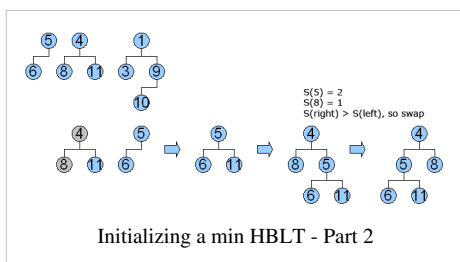
    if (x.leftChild == null) {
        // left child doesn't exist, so move right child to the left side
        x.leftChild = x.rightChild;
        x.rightChild = null;
        x.s = 1;
    } else {
        // left child does exist, so compare s-values
        if (x.leftChild.s < x.rightChild.s) {
            Node temp = x.leftChild;
            x.leftChild = x.rightChild;
            x.rightChild = temp;
        }
        // since we know the right child has the lower s-value, we can just
        // add one to its s-value
        x.s = x.rightChild.s + 1;
    }
    return x;
}
```

Initializing a height biased leftist tree

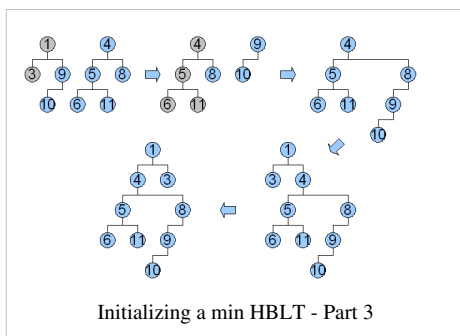


Initializing a height biased leftist tree is primarily done in one of two ways. The first is to merge each node one at a time into one HBLT. This process is inefficient and takes $O(n \log n)$ time. The other approach is to use a queue to store each node and resulting tree. The first two items in the queue are removed, merged, and placed back into the queue. This can initialize a HBLT in $O(n)$ time. This approach is detailed in the three diagrams supplied. A min height biased leftist tree is shown.

To initialize a min HBLT, place each element to be added to the tree into a queue. In the example (see Part 1 to the left), the set of numbers [4, 8, 10, 9, 1, 3, 5, 6, 11] are initialized. Each line of the diagram represents another cycle of the algorithm, depicting the contents of the queue. The first five steps are easy to follow. Notice that the freshly created HBLT is added to the end of the queue. In the fifth step, the first occurrence of an s-value greater than 1 occurs. The sixth step shows two trees merged with each other, with predictable results.



In part 2 a slightly more complex merge happens. The tree with the lower value (tree x) has a right child, so merge must be called again on the subtree rooted by tree x's right child and the other tree. After the merge with the subtree, the resulting tree is put back into tree x. The s-value of the right child ($s=2$) is now greater than the s-value of the left child ($s=1$), so they must be swapped. The s-value of the root node 4 is also now 2.



Part 3 is the most complex. Here, we recursively call merge twice (each time with the right child's subtree that is not grayed out). This uses the same process described for part 2.

External links

- Leftist Trees ^[2], Sartaj Sahni

References

- <http://mathworld.wolfram.com/ExtendedBinaryTree.html>
- <http://www.cise.ufl.edu/~sahni/cop5536/slides/lec114.pdf>

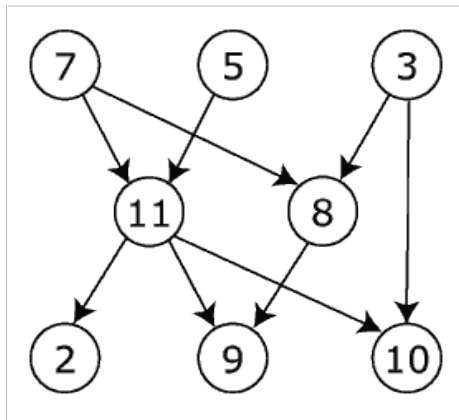
Topological sorting

In computer science, a **topological sort** (sometimes abbreviated **topsort** or **toposort**) or **topological ordering** of a directed graph is a linear ordering of its vertices such that, for every edge uv , u comes before v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Examples

The canonical application of topological sorting (topological order) is in scheduling a sequence of jobs or tasks based on their dependencies; topological sorting algorithms were first studied in the early 1960s in the context of the PERT technique for scheduling in project management (Jarnagin 1960). The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started (for example, when washing clothes, the washing machine must finish before we put the clothes to dry). Then, a topological sort gives an order in which to perform the jobs.

In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, serialization, and resolving symbol dependencies in linkers.



The graph shown to the left has many valid topological sorts, including:

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10

Algorithms

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges ($O(|V| + |E|)$).

One of these algorithms, first described by Kahn (1962), works by choosing vertices in the same order as the eventual topological sort. First, find a list of "start nodes" which have no incoming edges and insert them into a set S ; at least one such node must exist in an acyclic graph. Then:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    insert n into L
```

```

for each node  $m$  with an edge  $e$  from  $n$  to  $m$  do
    remove edge  $e$  from the graph
    if  $m$  has no other incoming edges then
        insert  $m$  into  $S$ 
if graph has edges then
    return error (graph has at least one cycle)
else
    return  $L$  (a topologically sorted order)

```

If the graph is a DAG, a solution will be contained in the list L (the solution is not necessarily unique). Otherwise, the graph must have at least one cycle and therefore a topological sorting is impossible.

Note that, reflecting the non-uniqueness of the resulting sort, the structure S can be simply a set or a queue or a stack. Depending on the order that nodes n are removed from set S , a different solution is created. A variation of Kahn's algorithm that breaks ties lexicographically forms a key component of the Coffman–Graham algorithm for parallel scheduling and layered graph drawing.

An alternative algorithm for topological sorting is based on depth-first search. For this algorithm, edges point in the opposite direction as the previous algorithm (and the opposite direction to that shown in the diagram in the Examples section above). There is an edge from x to y if job x depends on job y (in other words, if job y must be completed before job x can be started). The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort:

```

 $L \leftarrow$  Empty list that will contain the sorted nodes
 $S \leftarrow$  Set of all nodes with no incoming edges
for each node  $n$  in  $S$  do
    visit( $n$ )
function visit(node  $n$ )
    if  $n$  has not been visited yet then
        mark  $n$  as visited
        for each node  $m$  with an edge from  $n$  to  $m$  do
            visit( $m$ )
        add  $n$  to  $L$ 

```

Note that each node n gets added to the output list L only after considering all other nodes on which n depends (all descendant nodes of n in the graph). Specifically, when the algorithm adds node n , we are guaranteed that all nodes on which n depends are already in the output list L : they were added to L either by the preceding recursive call to `visit()`, or by an earlier call to `visit()`. Since each edge and node is visited once, the algorithm runs in linear time. Note that the simple pseudocode above cannot detect the error case where the input graph contains cycles. The algorithm can be refined to detect cycles by watching for nodes which are visited more than once during any nested sequence of recursive calls to `visit()` (e.g., by passing a list down as an extra argument to `visit()`, indicating which nodes have already been visited in the current call stack). This depth-first-search-based algorithm is the one described by Cormen et al. (2001); it seems to have been first described in print by Tarjan (1976).

Uniqueness

If a topological sort has the property that all pairs of consecutive vertices in the sorted order are connected by edges, then these edges form a directed Hamiltonian path in the DAG. If a Hamiltonian path exists, the topological sort order is unique; no other order respects the edges of the path. Conversely, if a topological sort does not form a Hamiltonian path, the DAG will have two or more valid topological orderings, for in this case it is always possible to form a second valid ordering by swapping two consecutive vertices that are not connected by an edge to each other. Therefore, it is possible to test in polynomial time whether a unique ordering exists, and whether a Hamiltonian path exists, despite the NP-hardness of the Hamiltonian path problem for more general directed graphs (Vernet & Markenzon 1997).

Relation to partial orders

Topological orderings are also closely related to the concept of a linear extension of a partial order in mathematics.

A partially ordered set is just a set of objects together with a definition of the " \leq " inequality relation, satisfying the axioms of reflexivity ($x = x$), antisymmetry (if $x \leq y$ and $y \leq x$ then $x = y$) and transitivity (if $x \leq y$ and $y \leq z$, then $x \leq z$). A total order is a partial order in which, for every two objects x and y in the set, either $x \leq y$ or $y \leq x$. Total orders are familiar in computer science as the comparison operators needed to perform comparison sorting algorithms. For finite sets, total orders may be identified with linear sequences of objects, where the " \leq " relation is true whenever the first object precedes the second object in the order; a comparison sorting algorithm may be used to convert a total order into a sequence in this way. A linear extension of a partial order is a total order that is compatible with it, in the sense that, if $x \leq y$ in the partial order, then $x \leq y$ in the total order as well.

One can define a partial ordering from any DAG by letting the set of objects be the vertices of the DAG, and defining $x \leq y$ to be true, for any two vertices x and y , whenever there exists a directed path from x to y ; that is, whenever y is reachable from x . With these definitions, a topological ordering of the DAG is the same thing as a linear extension of this partial order. Conversely, any partial ordering may be defined as the reachability relation in a DAG. One way of doing this is to define a DAG that has a vertex for every object in the partially ordered set, and an edge xy for every pair of objects for which $x \leq y$. An alternative way of doing this is to use the transitive reduction of the partial ordering; in general, this produces DAGs with fewer edges, but the reachability relation in these DAGs is still the same partial order. By using these constructions, one can use topological ordering algorithms to find linear extensions of partial orders.

References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "Section 22.4: Topological sort", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 549–552, ISBN 0-262-03293-7.
- Jarnagin, M. P. (1960), *Automatic machine methods of testing PERT networks for consistency*, Technical Memorandum No. K-24/60, Dahlgren, Virginia: U. S. Naval Weapons Laboratory.
- Kahn, Arthur B. (1962), "Topological sorting of large networks", *Communications of the ACM* **5** (11): 558–562, doi:10.1145/368996.369025.
- Tarjan, Robert E. (1976), "Edge-disjoint spanning trees and depth-first search", *Acta Informatica* **6** (2): 171–185, doi:10.1007/BF00268499.
- Vernet, Oswaldo; Markenzon, Lilian (1997), "Hamiltonian problems for reducible flowgraphs", *Proc. 17th International Conference of the Chilean Computer Science Society (SCCC '97)*, pp. 264–267, doi:10.1109/SCCC.1997.637099.

External links

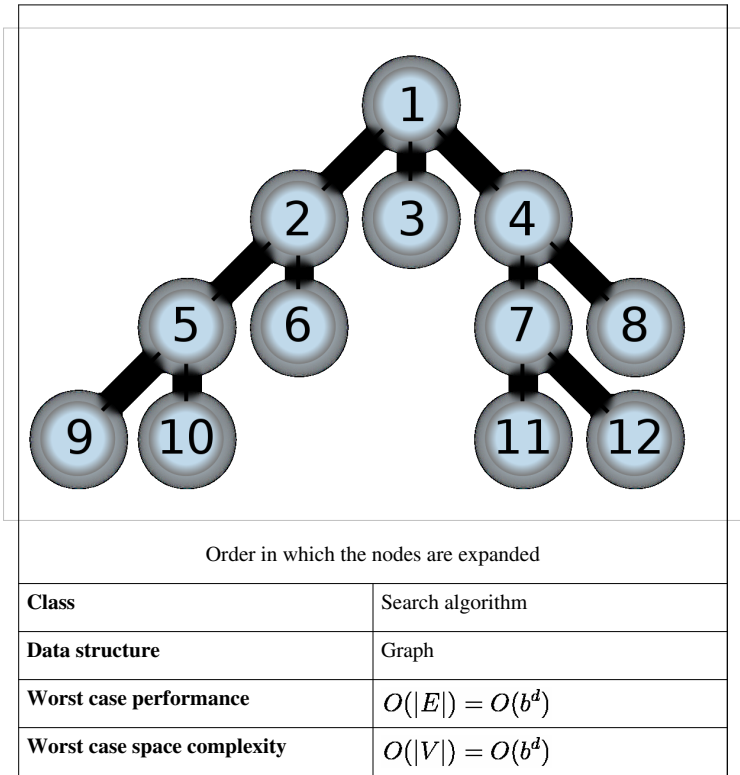
- NIST Dictionary of Algorithms and Data Structures: topological sort ^[1]
- Weisstein, Eric W., "TopologicalSort ^[2]" from MathWorld.

References

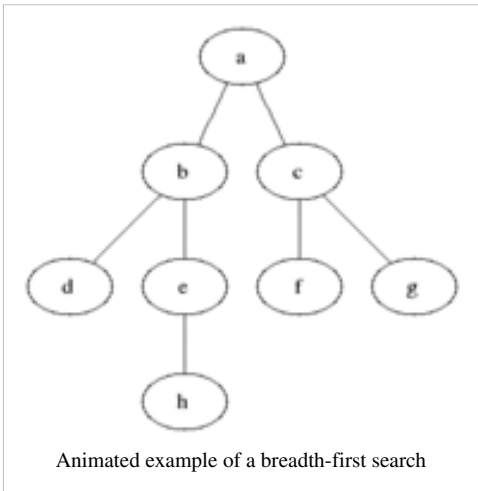
- [1] <http://www.nist.gov/dads/HTML/topologicalSort.html>
[2] <http://mathworld.wolfram.com/TopologicalSort.html>
-

Breadth-first search

Breadth-first search



In graph theory, **breadth-first search (BFS)** is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. Compare it with the depth-first search.



Algorithm

The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node and examine it
 - If the element sought is found in this node, quit the search and return a result.
 - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.

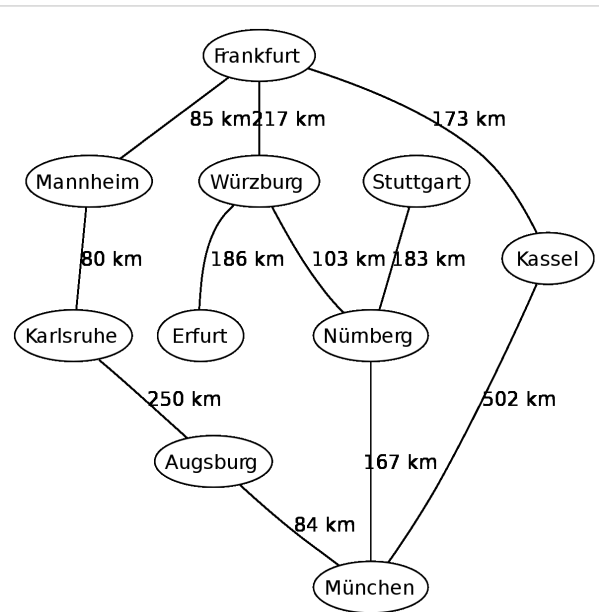
Pseudocode

Input: A graph G and a root v of G

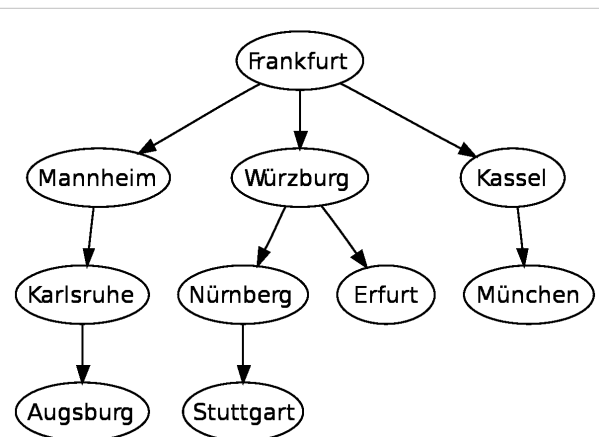
```

1  procedure BFS( $G, v$ ) :
2      create a queue  $Q$ 
3      enqueue  $v$  onto  $Q$ 
4      mark  $v$ 
5      while  $Q$  is not empty:
6           $t \leftarrow Q.dequeue()$ 
7          if  $t$  is what we are looking for:
8              return  $t$ 
9          for all edges  $e$  in  $G.adjacentEdges(t)$  do
12              $u \leftarrow G.adjacentVertex(t, e)$ 
13             if  $u$  is not marked:
14                 mark  $u$ 
15                 enqueue  $u$  onto  $Q$ 

```



An example map of Germany with some connections between cities



The breadth-first tree obtained when running BFS on the given map and starting in Frankfurt

Features

Space complexity

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$ where $|V|$ is the cardinality of the set of vertices.

Time complexity

The time complexity can be expressed as $O(|E|)$ since every vertex and every edge will be explored in the worst case. Note: $O(|E|)$ may vary between $O(|V|)$ and $O(|V|^2)$, depending on how sparse the input graph is (assuming that the graph is connected).

Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

- Finding all nodes within one connected component
- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes u and v (with path length measured by number of edges)
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

Finding connected components

The set of nodes reached by a BFS (breadth-first search) form the connected component containing the starting node.

Testing bipartiteness

BFS can be used to test bipartiteness, by starting the search at any vertex and giving alternating labels to the vertices visited during the search. That is, give label 0 to the starting vertex, 1 to all its neighbours, 0 to those neighbours' neighbours, and so on. If at any step a vertex has (visited) neighbours with the same label as itself, then the graph is not bipartite. If the search ends without such a situation occurring, then the graph is bipartite.

References

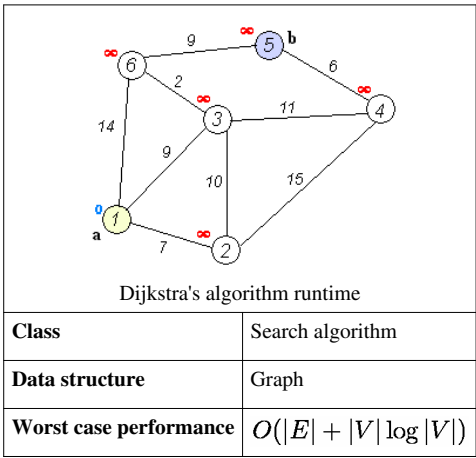
- Knuth, Donald E. (1997), *The Art Of Computer Programming Vol 1. 3rd ed.* (<http://www-cs-faculty.stanford.edu/~knuth/taocp.html>), Boston: Addison-Wesley, ISBN 0-201-89683-4

External links

- Breadth-First Explanation and Example (<http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch14.html#QQ1-46-92>)

Dijkstra's algorithm

Dijkstra's algorithm



Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959,^{[1][2]} is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

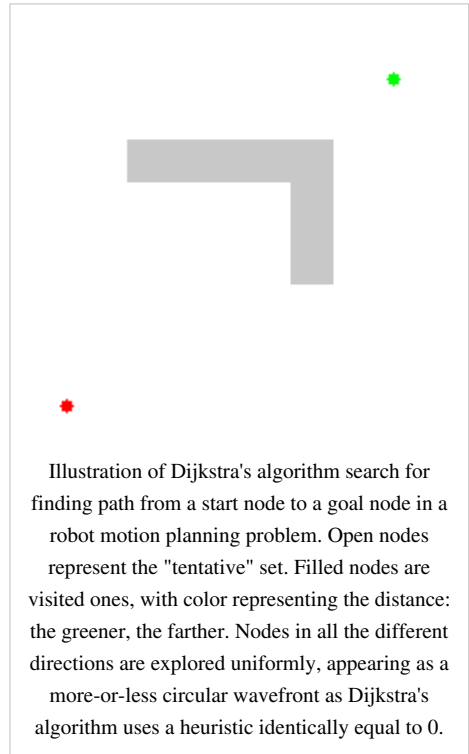
For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Dijkstra's original algorithm does not use a min-priority queue and runs in $O(|V|^2)$. The idea of this algorithm is also given in (Leyzorek et al. 1957). The implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V| \log |V|)$ is due to (Fredman & Tarjan 1984). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded nonnegative weights.

Algorithm

Let the node at which we are starting be called the **initial node**. Let the **distance of node Y** be the distance from the **initial node** to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes except the initial node.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. For example, if the current node A is marked with a tentative distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be $6+2=8$. If this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the *unvisited set*.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again; its distance recorded now is final and minimal.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal), then stop. The algorithm has finished.
6. Set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3.



Description

Note: For ease of understanding, this discussion uses the terms **intersection**, **road** and **map** — however, formally these terms are **vertex**, **edge** and **graph**, respectively.

Suppose you want to find the shortest path between two intersections on a city map, a starting point and a destination. The order is conceptually simple: to start, mark the distance to every intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that that intersection has not yet been *visited*; some variants of this method simply leave the intersection unlabeled. Now, at each iteration, select a *current* intersection. For the first iteration the current intersection will be the starting point and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first) the current intersection will be the closest unvisited intersection to the starting point—this will be easy to find.

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring intersection, mark the current intersection as *visited* and select the unvisited intersection with lowest distance (from the starting point) -- or lowest label—as the current intersection. Nodes marked as visited are labeled with the

shortest path from the starting point to it and will not be revisited or returned to.

Continue this process of updating the neighboring intersections with the shortest distances, then marking the current intersection as visited and moving onto the closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection) you have determined the shortest path to it, from the starting point, and can trace your way back, following the arrows in reverse.

Of note is the fact that this algorithm makes no attempt to direct "exploration" towards the destination as one might expect. Rather, the sole consideration in determining the next "current" intersection is its distance from the starting point. In some sense, this algorithm "expands outward" from the starting point, iteratively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path, however it may also reveal one of the algorithm's weaknesses: its relative slowness in some topologies.

Pseudocode

In the following algorithm, the code $u := \text{vertex in } Q \text{ with smallest } \text{dist}[]$, searches for the vertex u in the vertex set Q that has the least $\text{dist}[u]$ value. That vertex is removed from the set Q and returned to the user. $\text{dist_between}(u, v)$ calculates the length between the two neighbor-nodes u and v . The variable alt on lines 20 & 22 is the length of the path from the root node to the neighbor node v if it were to go through u . If this path is shorter than the current shortest path recorded for v , that current path is replaced with this alt path. The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```

1  function Dijkstra(Graph, source):
2      for each vertex  $v$  in Graph:                                // Initializations
3           $\text{dist}[v] := \text{infinity}$  ;                                // Unknown distance function from
4                                                                // source to  $v$ 
5           $\text{previous}[v] := \text{undefined}$  ;                            // Previous node in optimal path
6      end for                                                    // from source
7
8       $\text{dist}[\text{source}] := 0$  ;                                       // Distance from source to source
9       $Q := \text{the set of all nodes in Graph}$  ;                      // All nodes in the graph are
10                                                                // unoptimized - thus are in  $Q$ 
11      while  $Q$  is not empty:                                       // The main loop
12           $u := \text{vertex in } Q \text{ with smallest distance in } \text{dist}[]$  ; // Start node in first case
13          remove  $u$  from  $Q$  ;
14          if  $\text{dist}[u] = \text{infinity}$ :
15              break ;                                             // all remaining vertices are
16          end if                                                  // inaccessible from source
17
18          for each neighbor  $v$  of  $u$ :                                // where  $v$  has not yet been
19                                                                // removed from  $Q$ .
20               $\text{alt} := \text{dist}[u] + \text{dist\_between}(u, v)$  ;
21              if  $\text{alt} < \text{dist}[v]$ :                                    // Relax ( $u, v, a$ )
22                   $\text{dist}[v] := \text{alt}$  ;
23                   $\text{previous}[v] := u$  ;
24                  decrease-key  $v$  in  $Q$ ;                            // Reorder  $v$  in the Queue
25              end if

```

```

26         end for
27     end while
28 return dist;

```

If we are only interested in a shortest path between vertices `source` and `target`, we can terminate the search at line 13 if `u = target`. Now we can read the shortest path from `source` to `target` by reverse iteration:

```

1  S := empty sequence
2  u := target
3  while previous[u] is defined:           // Construct the shortest path with a stack S
4      insert u at the beginning of S      // Push the vertex into the stack
5      u := previous[u]                   // Traverse from target to source
6  end while ;

```

Now sequence `S` is the list of vertices constituting one of the shortest paths from `source` to `target`, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between `source` and `target` (there might be several different ones of the same length). Then instead of storing only a single node in each entry of `previous[]` we would store all nodes satisfying the relaxation condition. For example, if both `r` and `source` connect to `target` and both of them lie on different shortest paths through `target` (because the edge cost is the same in both cases), then we would add both `r` and `source` to `previous[target]`. When the algorithm completes, `previous[]` data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these shortest paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search.

Running time

An upper bound of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of $|E|$ and $|V|$ using big-O notation.

For any implementation of vertex set Q the running time is in $O(|E| \cdot dk_Q + |V| \cdot em_Q)$, where dk_Q and em_Q are times needed to perform decrease key and extract minimum operations in set Q , respectively.

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q . In this case, the running time is $O(|E| + |V|^2) = O(|V|^2)$.

For sparse graphs, that is, graphs with far fewer than $O(|V|^2)$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. With a binary heap, the algorithm requires $\Theta((|E| + |V|) \log |V|)$ time (which is dominated by $\Theta(|E| \log |V|)$, assuming the graph is connected). To avoid $O(|V|)$ look-up in decrease-key step on a vanilla binary heap, it is necessary to maintain a supplementary index mapping each vertex to the heap's index (and keep it up to date as priority queue Q changes), making it take only $O(\log |V|)$ time instead. The Fibonacci heap improves this to $O(|E| + |V| \log |V|)$. Note that for directed acyclic graphs, it is possible to find shortest paths from a given starting vertex in linear time, by processing the vertices in a topological order, and calculating the path length for each vertex to be the minimum length obtained via any of its incoming edges.^[3]

Related problems and algorithms

The functionality of Dijkstra's original algorithm can be extended with a variety of modifications. For example, sometimes it is desirable to present solutions which are less than mathematically optimal. To obtain a ranked list of less-than-optimal solutions, the optimal solution is first calculated. A single edge appearing in the optimal solution is removed from the graph, and the optimum solution to this new graph is calculated. Each edge of the original solution is suppressed in turn and a new shortest-path calculated. The secondary solutions are then ranked and presented after the first optimal solution.

Dijkstra's algorithm is usually the working principle behind link-state routing protocols, OSPF and IS-IS being the most common ones.

Unlike Dijkstra's algorithm, the Bellman-Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex s . (The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed.)

The A* algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored, if additional information is available that provides a lower bound on the "distance" to the target. This approach can be viewed from the perspective of linear programming: there is a natural linear program for computing shortest paths, and solutions to its dual linear program are feasible if and only if they form a consistent heuristic (speaking roughly, since the sign conventions differ from place to place in the literature). This feasible dual / consistent heuristic defines a nonnegative reduced cost and A* is essentially running Dijkstra's algorithm with these reduced costs. If the dual satisfies the weaker condition of admissibility, then A* is instead more akin to the Bellman-Ford algorithm.

The process that underlies Dijkstra's algorithm is similar to the greedy process used in Prim's algorithm. Prim's purpose is to find a minimum spanning tree that connects all nodes in the graph; Dijkstra is concerned with only two nodes. Prim's does not evaluate the total weight of the path from the starting node, only the individual path.

Dynamic programming perspective

From a dynamic programming point of view, Dijkstra's algorithm is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the **Reaching** method.^{[4][5][6]}

In fact, Dijkstra's explanation of the logic behind the algorithm,^[7] namely

Problem 2. Find the path of minimum total length between two given nodes P and Q .

We use the fact that, if R is a node on the minimal path from P to Q , knowledge of the latter implies the knowledge of the minimal path from P to R .

is a paraphrasing of Bellman's famous Principle of Optimality in the context of the shortest path problem.

Notes

- [1] Dijkstra, Edsger; Thomas J. Misa, Editor (2010-08). "An Interview with Edsger W. Dijkstra". *Communications of the ACM* **53** (8): 41–47. doi:10.1145/1787234.1787249. "What is the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path which I designed in about 20 minutes. One morning I was shopping with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path."
- [2] Dijkstra 1959
- [3] http://www.boost.org/doc/libs/1_44_0/libs/graph/doc/dag_shortest_paths.html
- [4] Sniedovich, M. (2006). "Dijkstra's algorithm revisited: the dynamic programming connexion" (<http://matwbn.icm.edu.pl/ksiazki/cc/cc35/cc3536.pdf>) (PDF). *Journal of Control and Cybernetics* **35** (3): 599–620. . Online version of the paper with interactive computational modules. (http://www.ifors.ms.unimelb.edu.au/tutorial/dijkstra_new/index.html)
- [5] Denardo, E.V. (2003). *Dynamic Programming: Models and Applications*. Mineola, NY: Dover Publications. ISBN 978-0-486-42810-9.
- [6] Sniedovich, M. (2010). *Dynamic Programming: Foundations and Principles*. Francis & Taylor. ISBN 978-0-8247-4099-3.
- [7] Dijkstra 1959, p. 270

References

- Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" (<http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf>). *Numerische Mathematik* **1**: 269–271. doi:10.1007/BF01386390.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw-Hill. pp. 595–601. ISBN 0-262-03293-7.
- Fredman, Michael Lawrence; Tarjan, Robert E. (1984). "Fibonacci heaps and their uses in improved network optimization algorithms" (<http://www.computer.org/portal/web/csdl/doi/10.1109/SFCS.1984.715934>). *25th Annual Symposium on Foundations of Computer Science* (IEEE): 338–346. doi:10.1109/SFCS.1984.715934.
- Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (<http://portal.acm.org/citation.cfm?id=28874>). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874.
- Zhan, F. Benjamin; Noon, Charles E. (February 1998). "Shortest Path Algorithms: An Evaluation Using Real Road Networks". *Transportation Science* **32** (1): 65–73. doi:10.1287/trsc.32.1.65.
- Leyzorek, M.; Gray, R. S.; Johnson, A. A.; Ladew, W. C.; Meaker, Jr., S. R.; Petry, R. M.; Seitz, R. N. (1957). *Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — A Study of Model Techniques for Communication Systems*. Cleveland, Ohio: Case Institute of Technology.

External links

- Oral history interview with Edsger W. Dijkstra (<http://purl.umn.edu/107247>), Charles Babbage Institute University of Minnesota, Minneapolis.
- Dijkstra's Algorithm in C# (<http://www.codeproject.com/KB/recipes/ShortestPathCalculation.aspx>)
- Fast Priority Queue Implementation of Dijkstra's Algorithm in C# (<http://www.codeproject.com/KB/recipes/FastHeapDijkstra.aspx>)
- Applet by Carla Laffra of Pace University (<http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html>)
- Animation of Dijkstra's algorithm (<http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html>)
- Visualization of Dijkstra's Algorithm (http://students.ceid.upatras.gr/~papagel/english/java_docs/minDijk.htm)
- Shortest Path Problem: Dijkstra's Algorithm (<http://www-b2.is.tokushima-u.ac.jp/~ikedasuuri/dijkstra/Dijkstra.shtml>)
- Dijkstra's Algorithm Applet (<http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm>)
- Open Source Java Graph package with implementation of Dijkstra's Algorithm (<http://code.google.com/p/annas/>)
- Haskell implementation of Dijkstra's Algorithm (<http://bonsaicode.wordpress.com/2011/01/04/programming-praxis-dijkstra-s-algorithm/>) on Bonsai code
- Java Implementation of Dijkstra's Algorithm (http://algowiki.net/wiki/index.php?title=Dijkstra's_algorithm) on AlgoWiki
- QuickGraph, Graph Data Structures and Algorithms for .NET (<http://quickgraph.codeplex.com/>)
- Implementation in Boost C++ library (http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/dijkstra_shortest_paths.html)
- Implementation in T-SQL (<http://hansolav.net/sql/graphs.html>)
- A Java library for path finding with Dijkstra's Algorithm and example applet (<http://www.stackframe.com/software/PathFinder>)

- A MATLAB program for Dijkstra's algorithm (<http://www.mathworks.com/matlabcentral/fileexchange/20025-advanced-dijkstras-minimum-path-algorithm>)

Prim's algorithm

In computer science, **Prim's algorithm** is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is also sometimes called the **DJP algorithm**, the **Jarník algorithm**, or the **Prim–Jarník algorithm**.

Other algorithms for this problem include Kruskal's algorithm and Borůvka's algorithm. However, these other algorithms can also find minimum spanning forests of disconnected graphs, while Prim's algorithm requires the graph to be connected.

Description

Informal

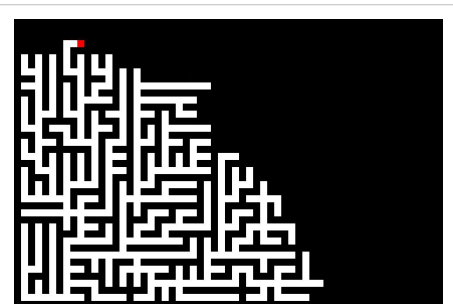
- create a tree containing a single vertex, chosen arbitrarily from the graph
- create a set containing all the edges in the graph
- loop until every edge in the set connects two vertices in the tree
 - remove from the set an edge with minimum weight that connects a vertex in the tree with a vertex not in the tree
 - add that edge to the tree

Technical

An empty graph cannot have a spanning tree, so we begin by assuming that the graph is non-empty.

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

- Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
- Initialize: $V_{\text{new}} = \{x\}$, where x is an arbitrary node (starting point) from V , $E_{\text{new}} = \{\}$
- Repeat until $V_{\text{new}} = V$:
 - Choose an edge $\{u, v\}$ with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
 - Add v to V_{new} , and $\{u, v\}$ to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree



Prim's algorithm has many applications, such as in the generation of this maze, which applies Prim's algorithm to a randomly weighted grid graph.

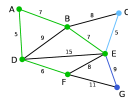
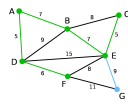
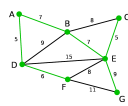
Time complexity

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(V^2)$
binary heap and adjacency list	$O((V + E) \log V) = O(E \log V)$
Fibonacci heap and adjacency list	$O(E + V \log V)$

A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can be shown to run in time $O(E \log V)$ where E is the number of edges and V is the number of vertices. Using a more sophisticated Fibonacci heap, this can be brought down to $O(E + V \log V)$, which is asymptotically faster when the graph is dense enough that E is $\omega(V)$.

Example run

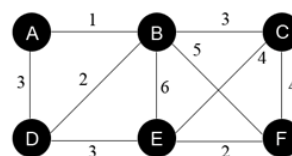
Image	U	possible edges	$V \setminus U$	Description
	{}		{A,B,C,D,E,F,G}	This is our original weighted graph. The numbers near the edges indicate their weight.
	{D}	{D,A} = 5 V {D,B} = 9 {D,E} = 15 {D,F} = 6	{A,B,C,E,F,G}	Vertex D has been arbitrarily chosen as a starting point. Vertices A , B , E and F are connected to D through a single edge. A is the vertex nearest to D and will be chosen as the second vertex along with the edge AD .
	{A,D}	{D,B} = 9 {D,E} = 15 {D,F} = 6 V {A,B} = 7	{B,C,E,F,G}	The next vertex chosen is the vertex nearest to <i>either</i> D or A . B is 9 away from D and 7 away from A , E is 15, and F is 6. F is the smallest distance away, so we highlight the vertex F and the edge DF .
	{A,D,F}	{D,B} = 9 {D,E} = 15 {A,B} = 7 V {F,E} = 8 {F,G} = 11	{B,C,E,G}	The algorithm carries on as above. Vertex B , which is 7 away from A , is highlighted.
	{A,B,D,F}	{B,C} = 8 {B,E} = 7 V {D,B} = 9 cycle {D,E} = 15 {F,E} = 8 {F,G} = 11	{C,E,G}	In this case, we can choose between C , E , and G . C is 8 away from B , E is 7 away from B , and G is 11 away from F . E is nearest, so we highlight the vertex E and the edge BE .

	{A,B,D,E,F}	$\{B,C\} = 8$ $\{D,B\} = 9$ cycle $\{D,E\} = 15$ cycle $\{E,C\} = 5$ V $\{E,G\} = 9$ $\{F,E\} = 8$ cycle $\{F,G\} = 11$	{C,G}	Here, the only vertices available are C and G . C is 5 away from E , and G is 9 away from E . C is chosen, so it is highlighted along with the edge EC .
	{A,B,C,D,E,F}	$\{B,C\} = 8$ $\{D,B\} = 9$ cycle $\{D,E\} = 15$ cycle $\{E,G\} = 9$ V $\{F,E\} = 8$ cycle $\{F,G\} = 11$	{G}	Vertex G is the only remaining vertex. It is 11 away from F , and 9 away from E . E is nearer, so we highlight G and the edge EG .
	{A,B,C,D,E,F,G}	$\{B,C\} = 8$ $\{D,B\} = 9$ cycle $\{D,E\} = 15$ cycle $\{F,E\} = 8$ cycle $\{F,G\} = 11$ cycle	{ }	Now all the vertices have been selected and the minimum spanning tree is shown in green. In this case, it has weight 39.

Proof of correctness

Let P be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in a subgraph to a vertex outside the subgraph. Since P is connected, there will always be a path to every vertex. The output Y of Prim's algorithm is a tree, because the edge and vertex added to tree Y are connected. Let Y_l be a minimum spanning tree of graph P . If $Y_l = Y$ then Y is a minimum spanning tree. Otherwise, let e be the first edge added during the construction of tree Y that is not in tree Y_l , and V be the set of vertices connected by the edges added before edge e . Then one endpoint of edge e is in set V and the other is not. Since tree Y_l is a spanning tree of graph P , there is a path in tree Y_l joining the two endpoints. As one travels along the path, one must encounter an edge f joining a vertex in set V to one that is not in set V . Now, at the iteration when edge e was added to tree Y , edge f could also have been added and it would be added instead of edge e if its weight was less than e (we know we encountered the opportunity to take " f " before " e " because " f " is connected to V , and we visited every vertex of V before the vertex to which we connected " e " [" e " is connected to the last vertex we visited in V]). Since edge f was not added, we conclude that

SET: { }



This animation shows how Prim's algorithm runs in a graph. (Click the image to see the animation)

$$w(f) \geq w(e).$$

Let tree Y_2 be the graph obtained by removing edge f from and adding edge e to tree Y_1 . It is easy to show that tree Y_2 is connected, has the same number of edges as tree Y_1 , and the total weights of its edges is not larger than that of tree Y_1 , therefore it is also a minimum spanning tree of graph P and it contains edge e and all the edges added before it during the construction of set V . Repeat the steps above and we will eventually obtain a minimum spanning tree of graph P that is identical to tree Y . This shows Y is a minimum spanning tree.

References

- V. Jarník: *O jistém problému minimálním* [About a certain minimal problem], *Práce Moravské Přírodovědecké Společnosti*, 6, 1930, pp. 57–63. (in Czech)
- R. C. Prim: *Shortest connection networks and some generalizations*. In: *Bell System Technical Journal*, 36 (1957), pp. 1389–1401
- D. Cheriton and R. E. Tarjan: *Finding minimum spanning trees*. In: *SIAM Journal on Computing*, 5 (Dec. 1976), pp. 724–741
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Third Edition. MIT Press, 2009. ISBN 0-262-03384-4. Section 23.2: The algorithms of Kruskal and Prim, pp. 631–638.

External links

- Animated example of Prim's algorithm (<http://students.ceid.upatras.gr/~papagel/project/prim.htm>)
- Prim's Algorithm (Java Applet) (<http://www.mincel.com/java/prim.html>)
- Minimum spanning tree demonstration Python program by Ronald L. Rivest (<http://people.csail.mit.edu/rivest/programs.html>)
- Open Source Java Graph package with implementation of Prim's Algorithm (<http://code.google.com/p/annas/>)
- Open Source C# class library with implementation of Prim's Algorithm (<http://code.google.com/p/ngenerics/>)

Kruskal's algorithm

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

This algorithm first appeared in *Proceedings of the American Mathematical Society*, pp. 48–50 in 1956, and was written by Joseph Kruskal.

Other algorithms for this problem include Prim's algorithm, Reverse-Delete algorithm, and Borůvka's algorithm.

Description

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty and F is not yet spanning
 - remove an edge with minimum weight from S
 - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
 - otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Complexity

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

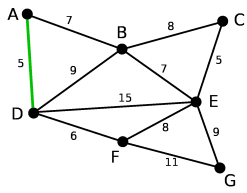
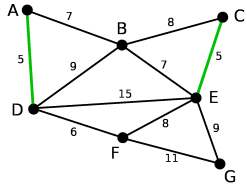
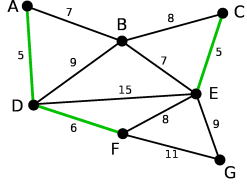
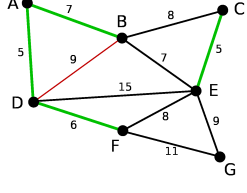
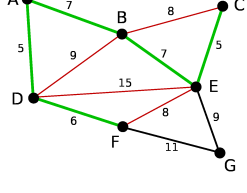
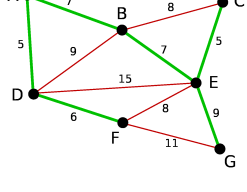
- E is at most V^2 and $\log V^2 = 2 \log V$ is $O(\log V)$.
- If we ignore isolated vertices, which will each be their own component of the minimum spanning forest, $V \leq E+1$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from S " to operate in constant time. Next, we use a disjoint-set data structure (Union&Find) to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

Provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the algorithm can use more sophisticated disjoint-set data structure to run in $O(E \alpha(V))$ time, where α is the extremely slowly growing inverse of the single-valued Ackermann function.

Example

Download the example data. ^[1]

Image	Description
	AD and CE are the shortest edges, with length 5, and AD has been arbitrarily chosen, so it is highlighted.
	CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.
	The next edge, DF with length 6, is highlighted using much the same method.
	The next-shortest edges are AB and BE , both with length 7. AB is chosen arbitrarily, and is highlighted. The edge BD has been highlighted in red, because there already exists a path (in green) between B and D , so it would form a cycle (ABD) if it were chosen.
	The process continues to highlight the next-smallest edge, BE with length 7. Many more edges are highlighted in red at this stage: BC because it would form the loop BCE , DE because it would form the loop DEBA , and FE because it would form FEBAD .
	Finally, the process finishes with the edge EG of length 9, and the minimum spanning tree is found.

Proof of correctness

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed spanning tree is of minimal weight.

Spanning Tree

Let P be a connected, weighted graph and let Y be the subgraph of P produced by the algorithm. Y cannot have a cycle, since the last edge added to that cycle would have been within one subtree and not between two different trees. Y cannot be disconnected, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of P .

Minimality

We show that the following proposition P is true by induction: If F is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains F .

- Clearly P is true at the beginning, when F is empty: any minimum spanning tree will do, and there exists one because a weighted connected graph always has a minimum spanning tree.
- Now assume P is true for some non-final edge set F and let T be a minimum spanning tree that contains F . If the next chosen edge e is also in T , then P is true for $F + e$. Otherwise, $T + e$ has a cycle C and there is another edge f that is in C but not F . (If there were no such edge f , then e could not have been added to F , since doing so would have created the cycle C .) Then $T - f + e$ is a tree, and it has the same weight as T , since T has minimum weight and the weight of f cannot be less than the weight of e , otherwise the algorithm would have chosen f instead of e . So $T - f + e$ is a minimum spanning tree containing $F + e$ and again P holds.
- Therefore, by the principle of induction, P holds when F has become a spanning tree, which is only possible if F is a minimum spanning tree itself.

References

- Joseph. B. Kruskal: *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem* ^[2]. In: *Proceedings of the American Mathematical Society*, Vol 7, No. 1 (Feb, 1956), pp. 48–50
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 23.2: The algorithms of Kruskal and Prim, pp. 567–574.
- Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*, Fourth Edition. John Wiley & Sons, Inc., 2006. ISBN 0-471-73884-0. Section 13.7.1: Kruskal's Algorithm, pp. 632.

External links

- Download the example minimum spanning tree data. ^[1]
- Animation of Kruskal's algorithm (Requires Java plugin) ^[3]
- C# Implementation ^[4]

References

- [1] <http://www.carlschroedl.com/blog/comp/kruskals-minimum-spanning-tree-algorithm/2012/05/14/>
- [2] [http://links.jstor.org/sici?sici=0002-9939\(195602\)7%3A1%3C48%3AOTSSO%3E2.0.CO%3B2-M](http://links.jstor.org/sici?sici=0002-9939(195602)7%3A1%3C48%3AOTSSO%3E2.0.CO%3B2-M)
- [3] <http://students.ceid.upatras.gr/~papagel/project/kruskal.htm>
- [4] http://www.codeproject.com/KB/recipes/Kruskal_Algorithm.aspx

Bellman–Ford algorithm

Bellman–Ford algorithm

Class	Single-source shortest path problem (for weighted directed graphs)
Data structure	Graph
Worst case performance	$O(V E)$
Worst case space complexity	$O(V)$

The **Bellman–Ford algorithm** computes single-source shortest paths in a weighted digraph.^[1] For graphs with only non-negative edge weights, the faster Dijkstra's algorithm also solves the problem. Thus, Bellman–Ford is used primarily for graphs with negative edge weights. The algorithm is named after its developers, Richard Bellman and Lester Ford, Jr.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.^[2] However, if a graph contains a "negative cycle", i.e., a cycle whose edges sum to a negative value, then walks of arbitrarily low weight can be constructed by repeatedly following the cycle, so there may not be a *shortest* path. In such a case, the Bellman-Ford algorithm can detect negative cycles and report their existence, but it cannot produce a correct "shortest path" answer if a negative cycle is reachable from the source.^{[3][1]}

Algorithm

Bellman–Ford is based on dynamic programming approach. In its basic structure it is similar to Dijkstra's Algorithm, but instead of greedily selecting the minimum-weight node not yet processed to relax, it simply relaxes *all* the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. The repetitions allow minimum distances to propagate accurately throughout the graph, since, in the absence of negative cycles, the shortest path can visit each node at most only once. Unlike the greedy approach, which depends on certain structural assumptions derived from positive weights, this straightforward approach extends to the general case.

Bellman–Ford runs in $O(|V||E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively.

```

procedure BellmanFord(list vertices, list edges, vertex source)
    // This implementation takes in a graph, represented as lists of vertices
    // and edges, and modifies the vertices so that their distance and
    // predecessor attributes store the shortest paths.

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then v.distance := 0
        else v.distance := infinity
        v.predecessor := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge uv in edges: // uv is the edge from u to v
            u := uv.source
            v := uv.destination
            if u.distance + uv.weight < v.distance:
                v.distance := u.distance + uv.weight

```

```

        v.predecessor := u

// Step 3: check for negative-weight cycles
for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if u.distance + uv.weight < v.distance:
        error "Graph contains a negative-weight cycle"

```

Proof of correctness

The correctness of the algorithm can be shown by induction. The precise statement shown by induction is:

Lemma. After i repetitions of *for cycle*:

- If $\text{Distance}(u)$ is not infinity, it is equal to the length of some path from s to u ;
- If there is a path from s to u with at most i edges, then $\text{Distance}(u)$ is at most the length of the shortest path from s to u with at most i edges.

Proof. For the base case of induction, consider $i=0$ and the moment before *for cycle* is executed for the first time. Then, for the source vertex, $\text{source.distance} = 0$, which is correct. For other vertices u , $u.\text{distance} = \text{infinity}$, which is also correct because there is no path from *source* to u with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex's distance is updated by $v.\text{distance} := u.\text{distance} + uv.\text{weight}$. By inductive assumption, $u.\text{distance}$ is the length of some path from *source* to u . Then $u.\text{distance} + uv.\text{weight}$ is the length of the path from *source* to v that follows the path from *source* to u and then goes to v .

For the second part, consider the shortest path from *source* to u with at most i edges. Let v be the last vertex before u on this path. Then, the part of the path from *source* to v is the shortest path from *source* to v with at most $i-1$ edges. By inductive assumption, $v.\text{distance}$ after $i-1$ cycles is at most the length of this path. Therefore, $uv.\text{weight} + v.\text{distance}$ is at most the length of the path from s to u . In the i^{th} cycle, $u.\text{distance}$ gets compared with $uv.\text{weight} + v.\text{distance}$, and is set equal to it if $uv.\text{weight} + v.\text{distance}$ was smaller. Therefore, after i cycles, $u.\text{distance}$ is at most the length of the shortest path from *source* to u that uses at most i edges.

If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so at step 3 no further improvements can be made. Conversely, suppose no improvement can be made. Then for any cycle with vertices $v[0], \dots, v[k-1]$,

$$v[i].\text{distance} \leq v[(i-1) \bmod k].\text{distance} + v[(i-1) \bmod k]v[i].\text{weight}$$

Summing around the cycle, the $v[i].\text{distance}$ terms and the $v[(i-1) \bmod k]$ distance terms cancel, leaving

$$0 \leq \sum \text{from } 1 \text{ to } k \text{ of } v[(i-1) \bmod k]v[i].\text{weight}$$

I.e., every cycle has nonnegative weight.

Finding negative cycles

When the algorithm is used to find shortest paths, the existence of negative cycles is a problem, preventing the algorithm from finding a correct answer. However, since it terminates upon finding a negative cycle, the Bellman-Ford algorithm can be used for applications in which this is the target to be sought - for example in cycle-cancelling techniques in network flow analysis.^[1]

Applications in routing

A distributed variant of the Bellman–Ford algorithm is used in distance-vector routing protocols, for example the Routing Information Protocol (RIP). The algorithm is distributed because it involves a number of nodes (routers) within an Autonomous system, a collection of IP networks typically owned by an ISP. It consists of the following steps:

1. Each node calculates the distances between itself and all other nodes within the AS and stores this information as a table.
2. Each node sends its table to all neighboring nodes.
3. When a node receives distance tables from its neighbors, it calculates the shortest routes to all other nodes and updates its own table to reflect any changes.

The main disadvantages of the Bellman–Ford algorithm in this setting are as follows:

- It does not scale well.
- Changes in network topology are not reflected quickly since updates are spread node-by-node.
- Count to infinity (if link or node failures render a node unreachable from some set of other nodes, those nodes may spend forever gradually increasing their estimates of the distance to it, and in the meantime there may be routing loops).

Yen's improvement

Yen (1970) described an improvement to the Bellman–Ford algorithm for a graph without negative-weight cycles. Yen's improvement first assigns some arbitrary linear order on all vertices and then partitions the set of all edges into one of two subsets. The first subset, E_f , contains all edges (v_i, v_j) such that $i < j$; the second, E_b , contains edges (v_i, v_j) such that $i > j$. Each vertex is visited in the order $v_1, v_2, \dots, v_{|V|}$, relaxing each outgoing edge from that vertex in E_f . Each vertex is then visited in the order $v_{|V|}, v_{|V|-1}, \dots, v_1$, relaxing each outgoing edge from that vertex in E_b . Yen's improvement effectively halves the number of "passes" required for the single-source-shortest-path solution.

Notes

- [1] Bang-Jensen, Jørgen; Gregory Gutin. "Section 2.3.4: The Bellman-Ford-Moore algorithm" (<http://www.cs.rhul.ac.uk/books/dbook/>). *Digraphs: Theory, Algorithms and Applications* (First ed.). ISBN 978-1-84800-997-4. .
- [2] Sedgewick, Robert. "Section 21.7: Negative Edge Weights" (<http://safari.oreilly.com/0201361213/ch21lev1sec7>). *Algorithms in Java* (Third ed.). ISBN 0-201-36121-3.
- [3] Kleinberg, Jon; Tardos, Éva (2006), *Algorithm Design*, New York: Pearson Education, Inc..

References

- Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* **16**: 87–90. MR0102435..
- Ford, L. R., Jr.; Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press..
- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. *Introduction to Algorithms*. MIT Press and McGraw-Hill., Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 24.1: The Bellman–Ford algorithm, pp. 588–592. Problem 24-1, pp. 614–615.

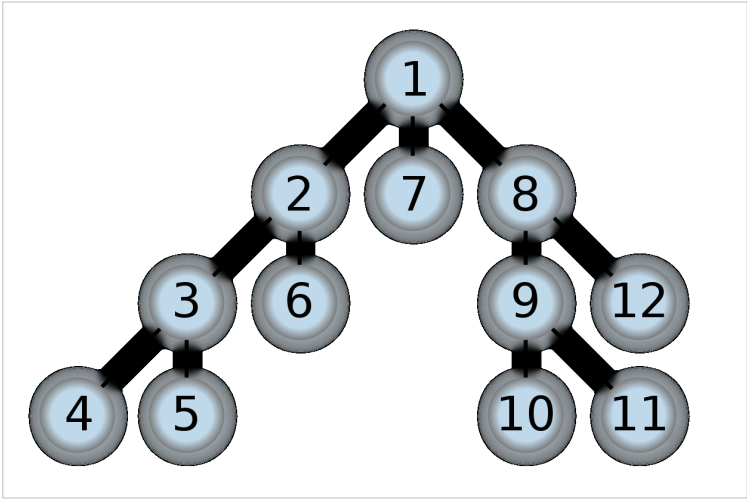
- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. *Introduction to Algorithms*. MIT Press and McGraw-Hill., Third Edition. MIT Press, 2009. ISBN 978-0-262-53305-8. Section 24.1: The Bellman–Ford algorithm, pp. 651–655.
- Heineman, George T.; Pollice, Gary; Selkow, Stanley (2008). "Chapter 6: Graph Algorithms". *Algorithms in a Nutshell*. O'Reilly Media. pp. 160–164. ISBN 978-0-596-51624-6.
- Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks". *Quarterly of Applied Mathematics* **27**: 526–530. MR0253822..

External links

- C code example (<http://snippets.dzone.com/posts/show/4828>)
 - Open Source Java Graph package with Bellman-Ford Algorithms (<http://code.google.com/p/annas/>)
-

Depth-first search

Depth-first search



Order in which the nodes are visited

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(E)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor b searched to depth d
Worst case space complexity	$O(V)$ if entire graph is traversed without repetition, $O(\text{longest path length searched})$ for implicit graphs without elimination of duplicate nodes

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux^[1] as a strategy for solving mazes.^{[2][3]}

Formal definition

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

Properties

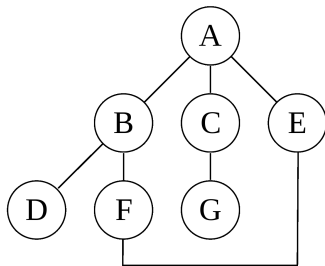
The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time $O(|E|)$, linear in the size of the graph. In these applications it also uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the time and space bounds are the same as for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

For applications of DFS to search problems in artificial intelligence, however, the graph to be searched is often either too large to visit in its entirety or even infinite, and DFS may suffer from non-termination when the length of a path in the search tree is infinite. Therefore, the search is only performed to a limited depth, and due to limited memory availability one typically does not use data structures that keep track of the set of all previously visited vertices. In this case, the time is still linear in the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, much smaller than the space needed for searching to the same depth using breadth-first search. For such applications, DFS also lends itself much better to heuristic methods of choosing a likely-looking branch. When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits; in the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

DFS may be also used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree.

Example

For the following graph:



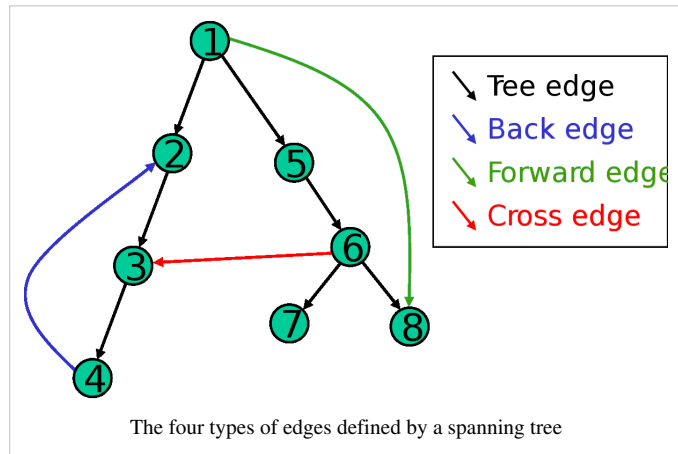
a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening is one technique to avoid this infinite loop and would reach all nodes.

Output of a depth-first search

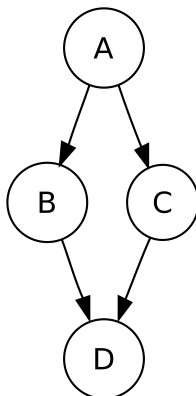
A convenient description of a depth first search of a graph is in terms of a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: **forward edges**, which point from a node of the tree to one of its descendants, **back edges**, which point from a node to one of its ancestors, and **cross edges**, which do neither. Sometimes **tree edges**, edges which belong to the spanning tree itself, are classified separately from forward edges. If the original graph is undirected then all of its edges are tree edges or back edges.



Vertex orderings

It is also possible to use the depth-first search to linearly order the vertices of the original graph (or tree). There are three common ways of doing this:

- A **preordering** is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation.
- A **postordering** is a list of the vertices in the order that they were *last* visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.
- A **reverse postordering** is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. Reverse postordering is not the same as preordering. For example, when searching the directed graph



beginning at node A, one visits the nodes in sequence, to produce lists either A B D B A C A, or A C D C A B A (depending upon whether the algorithm chooses to visit B or C first). Note that repeat visits in the form of backtracking to a node, to check if it has still unvisited neighbours, are included here (even if it is found to have none). Thus the possible preorderings are A B D C and A C D B (order by node's leftmost occurrence in above list), while the possible reverse postorderings are A C B D and A B C D (order by node's rightmost occurrence in above list). Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flow. The graph above might represent the flow of control in a code fragment like

```

if (A) then {
    B
} else {
    C
}
D

```

and it is natural to consider this code in the order A B C D or A C B D, but not natural to use the order A B D C or A C D B.

Pseudocode

Input: A graph G and a vertex v of G

Output: A labeling of the edges in the connected component of v as discovery edges and back edges

```

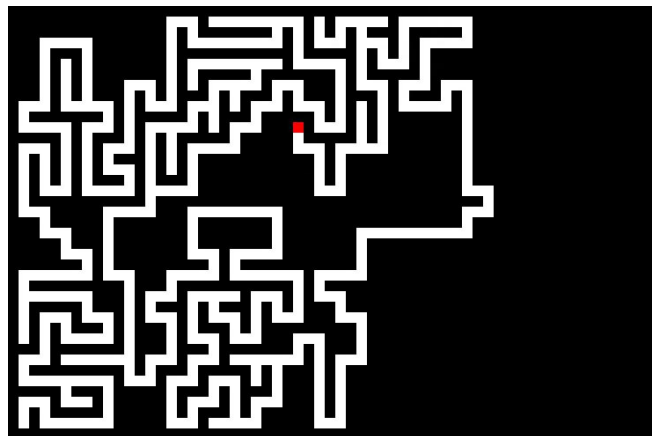
1  procedure DFS( $G, v$ ) :
2      label  $v$  as explored
3      for all edges  $e$  in  $G.\text{adjacentEdges}(v)$  do
4          if edge  $e$  is unexplored then
5               $w \leftarrow G.\text{adjacentVertex}(v, e)$ 
6              if vertex  $w$  is unexplored then
7                  label  $e$  as a discovery edge
8                  recursively call DFS( $G, w$ )
9              else
10                 label  $e$  as a back edge

```

Applications

Algorithms that use depth-first search as a building block include:

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding 3-(edge or vertex)-connected components.
- Finding the bridges of a graph.
- Finding strongly connected components.
- Planarity testing^{[4][5]}
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- Maze generation may use a randomized depth-first search.
- Finding biconnectivity in graphs.



Randomized algorithm similar to depth-first search used in generating a maze.

Notes

- [1] Charles Pierre Trémaux (1859–1882) École Polytechnique of Paris (X:1876), French engineer of the telegraph in Public conference, December 2, 2010 – by professor Jean Pelletier-Thibert in Académie de Macon (Burgundy – France) – (Abstract published in the Annals academic, March 2011 – ISSN: 0980-6032)
- [2] Even, Shimon (2011), *Graph Algorithms* (<http://books.google.com/books?id=m3QTSMYm5rkC&pg=PA46>) (2nd ed.), Cambridge University Press, pp. 46–48, ISBN 978-0-521-73653-4, .
- [3] Sedgewick, Robert (2002), *Algorithms in C++: Graph Algorithms* (3rd ed.), Pearson Education, ISBN 978-0-201-36118-6.
- [4] Hopcroft, John; Tarjan, Robert E. (1974), "Efficient planarity testing", *Journal of the Association for Computing Machinery* **21** (4): 549–568, doi:10.1145/321850.321852.
- [5] de Fraysseix, H.; Ossona de Mendez, P.; Rosenstiehl, P. (2006), "Trémaux Trees and Planarity", *International Journal of Foundations of Computer Science* **17** (5): 1017–1030, doi:10.1142/S0129054106004248.

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.
- Knuth, Donald E. (1997), *The Art Of Computer Programming Vol 1. 3rd ed* (<http://www-cs-faculty.stanford.edu/~knuth/taocp.html>), Boston: Addison-Wesley, ISBN 0-201-89683-4, OCLC 155842391
- Goodrich, Michael T. (2001), *Algorithm Design: Foundations, Analysis, and Internet Examples*, Wiley, ISBN 0-471-38365-1

External links

- Depth-First Explanation and Example (<http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch14.html>)
- C++ Boost Graph Library: Depth-First Search (http://www.boost.org/libs/graph/doc/depth_first_search.html)
- Depth-First Search Animation (for a directed graph) (<http://www.cs.duke.edu/cs680/jawaa/DFSanim.html>)
- Depth First and Breadth First Search: Explanation and Code (http://www.kirupa.com/developer/actionscript/depth_breadth_search.htm)
- QuickGraph ([http://quickgraph.codeplex.com/Wiki/View.aspx?title=Depth First Search Example](http://quickgraph.codeplex.com/Wiki/View.aspx?title=Depth+First+Search+Example)), depth first search example for .Net
- Depth-first search algorithm illustrated explanation (Java and C++ implementations) (http://www.algolist.net/Algorithms/Graph_algorithms/Undirected/Depth-first_search)
- YAGSBPL – A template-based C++ library for graph search and planning (<http://code.google.com/p/yagsbpl/>)

Biconnected graph

In graph theory, a **biconnected graph** is a connected and "nonseparable" graph, meaning that if any vertex were to be removed, the graph will remain connected. Therefore a biconnected graph has no articulation vertices.

The property of being 2-connected is equivalent to biconnectivity, with the caveat that the complete graph of two vertices is sometimes regarded as biconnected but not 2-connected.

This property is especially useful in maintaining a graph with a two-fold redundancy, to prevent disconnection upon the removal of a single edge (or connection).

The use of **biconnected** graphs is very important in the field of networking (see Network flow), because of this property of redundancy.

Definition

A **biconnected** undirected graph is a connected graph that is not broken into disconnected pieces by deleting any single vertex (and its incident edges).

A **biconnected** directed graph is one such that for any two vertices v and w there are two directed paths from v to w which have no vertices in common other than v and w .

Nonseparable (or 2-connected) graphs (or blocks) with n nodes (sequence A002218 in OEIS)

Vertices	Number of Possibilities
1	0
2	1
3	1
4	3
5	10
6	56
7	468
8	7123
9	194066
10	9743542
11	900969091
12	153620333545
13	48432939150704
14	28361824488394169
15	30995890806033380784
16	63501635429109597504951
17	244852079292073376010411280
18	1783160594069429925952824734641
19	24603887051350945867492816663958981

References

- Eric W. Weisstein. "Biconnected Graph." From MathWorld--A Wolfram Web Resource. <http://mathworld.wolfram.com/BiconnectedGraph.html>
- Paul E. Black, "biconnected graph", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. (accessed TODAY) Available from: <http://www.nist.gov/dads/HTML/biconnectedGraph.html>

External links

- The tree of the biconnected components Java implementation^[1] in the jBPT library (see BCTree class).

References

[1] <http://code.google.com/p/jbpt/>

Huffman coding

Huffman tree generated from the exact frequencies of the text "this is an example of a huffman tree". The frequencies and codes of each character are below. Encoding the sentence with this code requires 135 bits, as opposed to 288 bits if 36 characters of 8 bits were used. (This assumes that the code tree structure is known to the decoder and thus does not need to be counted as part of the transmitted information.)

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110

l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

In computer science and information theory, **Huffman coding** is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol. It was developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol) that expresses the most common source symbols using shorter strings of bits than are used for less common source symbols. Huffman was able to design the most efficient compression method *of this type*: no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code. The running time of Huffman's method is fairly efficient, it takes $O(n \log n)$ operations to construct it. A method was later found to design a Huffman code in linear time if input probabilities (also known as *weights*) are sorted.^[1]

For a set of symbols with a uniform probability distribution and a number of members which is a power of two, Huffman coding is equivalent to simple binary block encoding, e.g., ASCII coding. Huffman coding is such a widespread method for creating prefix codes that the term "Huffman code" is widely used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

Although Huffman's original algorithm is optimal for a symbol-by-symbol coding (i.e. a stream of unrelated symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown, not identically distributed, or not independent (e.g., "cat" is more common than "cta"). Other methods such as arithmetic coding and LZW coding often have better compression capability: both of these methods can combine an arbitrary number of symbols for more efficient coding, and generally adapt to the actual input statistics, the latter of which is useful when input probabilities are not precisely known or vary significantly within the stream. However, the limitations of Huffman coding should not be overstated; it can be used adaptively, accommodating unknown, changing, or context-dependent probabilities. In the case of known independent and identically distributed random variables, combining symbols reduces inefficiency in a way that approaches optimality as the number of symbols combined increases.

History

In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exam. The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient.^[2]

In doing so, the student outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code. Huffman avoided the major flaw of the suboptimal Shannon-Fano coding by building the tree from the bottom up instead of from the top down.

Problem definition

Informal description

Given

A set of symbols and their weights (usually proportional to probabilities).

Find

A prefix-free binary code (a set of codewords) with minimum expected codeword length (equivalently, a tree with minimum weighted path length from the root).

Formalized description

Input.

Alphabet $A = \{a_1, a_2, \dots, a_n\}$, which is the symbol alphabet of size n .

Set $W = \{w_1, w_2, \dots, w_n\}$, which is the set of the (positive) symbol weights (usually proportional to probabilities), i.e. $w_i = \text{weight}(a_i)$, $1 \leq i \leq n$.

Output.

Code $C(A, W) = \{c_1, c_2, \dots, c_n\}$, which is the set of (binary) codewords, where c_i is the codeword for a_i , $1 \leq i \leq n$.

Goal.

Let $L(C) = \sum_{i=1}^n w_i \times \text{length}(c_i)$ be the weighted path length of code C . Condition: $L(C) \leq L(T)$ for any code $T(A, W)$.

Samples

Input (A, W)	Symbol (a_i)	a	b	c	d	e	Sum
	Weights (w_i)	0.10	0.15	0.30	0.16	0.29	= 1
Output C	Codewords (c_i)	010	011	11	00	10	$L(C) = 2.25$
	Codeword length (in bits) (l_i)	3	3	2	2	2	
	Weighted path length ($l_i w_i$)	0.30	0.45	0.60	0.32	0.58	

Optimality	Probability budget (2^{-l_i})	1/8	1/8	1/4	1/4	1/4	= 1.00
	Information content (in bits) ($-\log_2 w_i$)	3.32	2.74	1.74	2.64	1.79	
	Entropy ($-w_i \log_2 w_i$)	0.332	0.411	0.521	0.423	0.518	$H(A) = 2.205$

For any code that is *biunique*, meaning that the code is *uniquely decodable*, the sum of the probability budgets across all symbols is always less than or equal to one. In this example, the sum is strictly equal to one; as a result, the code is termed a *complete* code. If this is not the case, you can always derive an equivalent code by adding extra symbols (with associated null probabilities), to make the code complete while keeping it *biunique*.

As defined by Shannon (1948), the information content h (in bits) of each symbol a_i with non-null probability is

$$h(a_i) = \log_2 \frac{1}{w_i}.$$

The entropy H (in bits) is the weighted sum, across all symbols a_i with non-zero probability w_i , of the information content of each symbol:

$$H(A) = \sum_{w_i > 0} w_i h(a_i) = \sum_{w_i > 0} w_i \log_2 \frac{1}{w_i} = - \sum_{w_i > 0} w_i \log_2 w_i.$$

(Note: A symbol with zero probability has zero contribution to the entropy, since $\lim_{w \rightarrow 0^+} w \log_2 w = 0$. So for simplicity, symbols with zero probability can be left out of the formula above.)

As a consequence of Shannon's source coding theorem, the entropy is a measure of the smallest codeword length that is theoretically possible for the given alphabet with associated weights. In this example, the weighted average codeword length is 2.25 bits per symbol, only slightly larger than the calculated entropy of 2.205 bits per symbol. So not only is this code optimal in the sense that no other feasible code performs better, but it is very close to the theoretical limit established by Shannon.

Note that, in general, a Huffman code need not be unique, but it is always one of the codes minimizing $L(C)$.

Basic technique

Compression

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n . A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the **symbol** itself, the **weight** (frequency of appearance) of the symbol and optionally, a link to a **parent** node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol **weight**, links to **two child nodes** and the optional link to a **parent** node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n - 1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node (thus not considering them anymore), and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree.

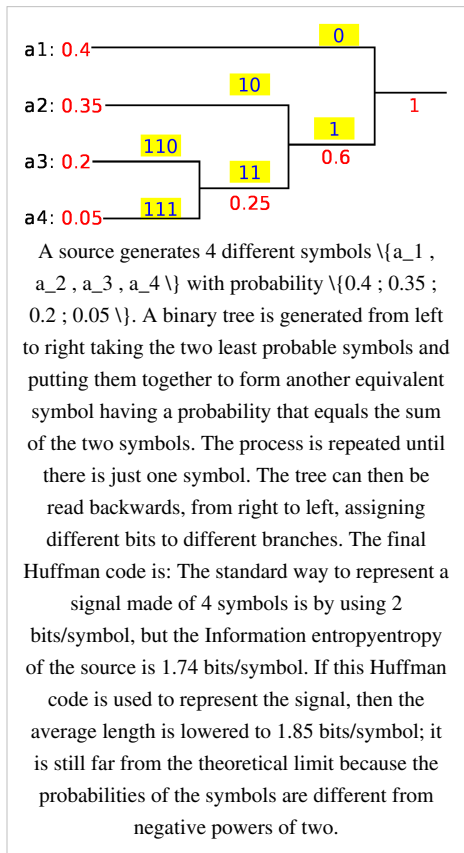
The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
 1. Remove the two nodes of highest priority (lowest probability) from the queue
 2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
 3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require $O(\log n)$ time per insertion, and a tree with n leaves has $2n-1$ nodes, this algorithm operates in $O(n \log n)$ time, where n is the number of symbols.

If the symbols are sorted by probability, there is a linear-time ($O(n)$) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
 1. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
 2. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.



3. Enqueue the new node into the rear of the second queue.
4. The remaining node is the root node; the tree has now been generated.

Although this algorithm may appear "faster" complexity-wise than the previous algorithm using a priority queue, this is not actually the case because the symbols need to be sorted by probability before-hand, a process that takes $O(n \log n)$ time in itself.

In many cases, time complexity is not very important in the choice of algorithm here, since n here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when n grows to be very large.

It is generally beneficial to minimize the variance of codeword length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

Here's an example using the French subject string "j'aime aller sur le bord de l'eau les jeudis ou les jours impairs":

File :	b	p	`	m	j	o	d	a	i	r	u	l	s	e	
	1	1	2	2	3	3	3	4	4	5	5	6	6	8	12

Decompression

Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise, the information to reconstruct the tree must be sent a priori. A naive approach might be to prepend the frequency count of each

character to the compression stream. Unfortunately, the overhead in such a case could amount to several kilobytes, so this method has little practical use. If the data is compressed using canonical encoding, the compression model can be precisely reconstructed with just $B2^B$ bits of information (where B is the number of bits per symbol). Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes (assuming an 8-bit alphabet). Many other techniques are possible as well. In any case, since the compressed data can include unused "trailing bits" the decompressor must be able to determine when to stop producing output. This can be accomplished by either transmitting the length of the decompressed data along with the compression model or by defining a special code symbol to signify the end of input (the latter method can adversely affect code length optimality, however).

Main properties

The probabilities used can be generic ones for the application domain that are based on average experience, or they can be the actual frequencies found in the text being compressed. (This variation requires that a frequency table or other hint as to the encoding must be stored with the compressed text; implementations employ various tricks to store tables efficiently.)

Huffman coding is optimal when the probability of each input symbol is a negative power of two. Prefix codes tend to have inefficiency on small alphabets, where probabilities often fall between these optimal points. "Blocking", or expanding the alphabet size by grouping multiple symbols into "words" of fixed or variable-length before Huffman coding helps both to reduce that inefficiency and to take advantage of statistical dependencies between input symbols within the group (as in the case of natural language text). The worst case for Huffman coding can happen when the probability of a symbol exceeds $2^{-1} = 0.5$, making the upper limit of inefficiency unbounded. These situations often respond well to a form of blocking called run-length encoding; for the simple case of Bernoulli processes, Golomb coding is a provably optimal run-length code.

Arithmetic coding produces some gains over Huffman coding, although arithmetic coding has higher computational complexity. Also, arithmetic coding was historically a subject of some concern over patent issues. However, as of mid-2010, various well-known effective techniques for arithmetic coding have passed into the public domain as the early patents have expired.

Variations

Many variations of Huffman coding exist, some of which use a Huffman-like algorithm, and others of which find optimal prefix codes (while, for example, putting different restrictions on the output). Note that, in the latter case, the method need not be Huffman-like, and, indeed, need not even be polynomial time. An exhaustive list of papers on Huffman coding and its variations is given by "Code and Parse Trees for Lossless Source Encoding"[3].

n -ary Huffman coding

The **n -ary Huffman** algorithm uses the $\{0, 1, \dots, n-1\}$ alphabet to encode message and build an n -ary tree. This approach was considered by Huffman in his original paper. The same algorithm applies as for binary (n equals 2) codes, except that the n least probable symbols are taken together, instead of just the 2 least probable. Note that for n greater than 2, not all sets of source words can properly form an n -ary tree for Huffman coding. In this case, additional 0-probability place holders must be added. This is because the tree must form an n to 1 contractor; for binary coding, this is a 2 to 1 contractor, and any sized set can form such a contractor. If the number of source words is congruent to 1 modulo $n-1$, then the set of source words will form a proper Huffman tree.

Adaptive Huffman coding

A variation called **adaptive Huffman coding** involves calculating the probabilities dynamically based on recent actual frequencies in the sequence of source symbols, and changing the coding tree structure to match the updated probability estimates. It is very rare in practice, because the cost of updating the tree makes it slower than optimized adaptive arithmetic coding, that is more flexible and has a better compression.

Huffman template algorithm

Most often, the weights used in implementations of Huffman coding represent numeric probabilities, but the algorithm given above does not require this; it requires only that the weights form a totally ordered commutative monoid, meaning a way to order weights and to add them. The **Huffman template algorithm** enables one to use any kind of weights (costs, frequencies, pairs of weights, non-numerical weights) and one of many combining methods (not just addition). Such algorithms can solve other minimization problems, such as minimizing $\max_i [w_i + \text{length}(c_i)]$, a problem first applied to circuit design [4].

Length-limited Huffman coding/minimum variance Huffman coding

Length-limited Huffman coding is a variant where the goal is still to achieve a minimum weighted path length, but there is an additional restriction that the length of each codeword must be less than a given constant. The package-merge algorithm solves this problem with a simple greedy approach very similar to that used by Huffman's algorithm. Its time complexity is $O(nL)$, where L is the maximum length of a codeword. No algorithm is known to solve this problem in linear or linearithmic time, unlike the presorted and unsorted conventional Huffman problems, respectively.

Huffman coding with unequal letter costs

In the standard Huffman coding problem, it is assumed that each symbol in the set that the code words are constructed from has an equal cost to transmit: a code word whose length is N digits will always have a cost of N , no matter how many of those digits are 0s, how many are 1s, etc. When working under this assumption, minimizing the total cost of the message and minimizing the total number of digits are the same thing.

Huffman coding with unequal letter costs is the generalization in which this assumption is no longer assumed true: the letters of the encoding alphabet may have non-uniform lengths, due to characteristics of the transmission medium. An example is the encoding alphabet of Morse code, where a 'dash' takes longer to send than a 'dot', and therefore the cost of a dash in transmission time is higher. The goal is still to minimize the weighted average codeword length, but it is no longer sufficient just to minimize the number of symbols used by the message. No algorithm is known to solve this in the same manner or with the same efficiency as conventional Huffman coding.

Optimal alphabetic binary trees (Hu-Tucker coding)

In the standard Huffman coding problem, it is assumed that any codeword can correspond to any input symbol. In the alphabetic version, the alphabetic order of inputs and outputs must be identical. Thus, for example, $A = \{a, b, c\}$ could not be assigned code $H(A, C) = \{00, 1, 01\}$, but instead should be assigned either $H(A, C) = \{00, 01, 1\}$ or $H(A, C) = \{0, 10, 11\}$. This is also known as the **Hu-Tucker** problem, after the authors of the paper presenting the first linearithmic solution to this optimal binary alphabetic problem^[5], which has some similarities to Huffman algorithm, but is not a variation of this algorithm. These optimal alphabetic binary trees are often used as binary search trees.

The canonical Huffman code

If weights corresponding to the alphabetically ordered inputs are in numerical order, the Huffman code has the same lengths as the optimal alphabetic code, which can be found from calculating these lengths, rendering Hu-Tucker coding unnecessary. The code resulting from numerically (re-)ordered input is sometimes called the *canonical Huffman code* and is often the code used in practice, due to ease of encoding/decoding. The technique for finding this code is sometimes called **Huffman-Shannon-Fano coding**, since it is optimal like Huffman coding, but alphabetic in weight probability, like Shannon-Fano coding. The Huffman-Shannon-Fano code corresponding to the example is $\{000, 001, 01, 10, 11\}$, which, having the same codeword lengths as the original solution, is also optimal.

Applications

Arithmetic coding can be viewed as a generalization of Huffman coding, in the sense that they produce the same output when every symbol has a probability of the form $1/2^k$; in particular it tends to offer significantly better compression for small alphabet sizes. Huffman coding nevertheless remains in wide use because of its simplicity and high speed. Intuitively, arithmetic coding can offer better compression than Huffman coding because its "code words" can have effectively non-integer bit lengths, whereas code words in Huffman coding can only have an integer number of bits. Therefore, there is an inefficiency in Huffman coding where a code word of length k only optimally matches a symbol of probability $1/2^k$ and other probabilities are not represented as optimally; whereas the code word length in arithmetic coding can be made to exactly match the true probability of the symbol.

Huffman coding today is often used as a "back-end" to some other compression methods. DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by Huffman coding (or variable-length prefix-free codes with a similar structure, although perhaps not necessarily designed by using Huffman's algorithm).

Notes

- [1] Jan van Leeuwen, On the construction of Huffman trees, ICALP 1976, 382-410
- [2] see Ken Huffman (1991)
- [3] <http://scholar.google.com/scholar?hl=en&lr=&cluster=6556734736002074338>
- [4] <http://citeseer.ist.psu.edu/context/665634/0>
- [5] T.C. Hu and A.C. Tucker, *Optimal computer search trees and variable length alphabetical codes*, Journal of SIAM on Applied Mathematics, vol. 21, no. 4, December 1971, pp. 514-532.

References

- For Java Implementation see: GitHub:GlanK (<https://github.com/GlanK/Huffman-Compression>)
- D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, pp 1098–1102. Huffman's original article.
- Ken Huffman. Profile: David A. Huffman (<http://www.huffmancoding.com/david/scientific.html>), Scientific American, September 1991, pp. 54–58
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 16.3, pp. 385–392.

External links

- Huffman Encoding process animation (<http://demo.tinyray.com/huffman>)
- Huffman Encoding & Decoding Animation (<http://www.cs.pitt.edu/~kirk/cs1501/animations/Huffman.html>)
- n-ary Huffman Template Algorithm (http://alexvn.freeseervers.com/s1/huffman_template_algorithm.html)
- Huffman Tree visual graph generator (<http://huffman.ooz.ie/>)
- Sloane A098950 (<http://www.research.att.com/projects/OEIS?Anum=A098950>) Minimizing k-ordered sequences of maximum height Huffman tree
- Mordecai J. Golin, Claire Kenyon, Neal E. Young " Huffman coding with unequal letter costs (http://www.cs.ust.hk/faculty/golin/pubs/LOP_PTAS_STOC.pdf)" (PDF), STOC 2002 (<http://www.informatik.uni-trier.de/~ley/db/conf/stoc/stoc2002.html>): 785-791
- Huffman Coding: A CS2 Assignment (<http://www.cs.duke.edu/cs2d/poop/huff/info/>) a good introduction to Huffman coding
- A quick tutorial on generating a Huffman tree (http://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html)
- Pointers to Huffman coding visualizations (<http://web-cat.cs.vt.edu/AlgovizWiki/HuffmanCodingTrees>)
- Huffman in C (<http://huffman.sourceforge.net>)
- Description of an implementation in Python ([http://en.literateprograms.org/Huffman_coding_\(Python\)](http://en.literateprograms.org/Huffman_coding_(Python)))
- Explanation of Huffman coding with examples in several languages (http://rosettacode.org/wiki/Huffman_codes)
- Interactive Huffman Tree Construction (<http://www.hightechdreams.com/weaver.php?topic=huffmancoding>)
- A C program doing basic Huffman coding on binary and text files (<http://github.com/elijahbal/huffman-coding/>)
- Efficient implementation of Huffman codes for blocks of binary sequences (<http://www.reznik.org/software.html#ABC>)
- Introductory description to text compression using Huffman Coding (<http://jara.de/goodbits/2011/08/14/introduction-to-data-compression-huffman-coding/>)

Floyd–Warshall algorithm

Floyd–Warshall algorithm

Class	All-pairs shortest path problem (for weighted graphs)
Data structure	Graph
Worst case performance	$O(V ^3)$
Best case performance	$\Omega(V ^3)$
Worst case space complexity	$\Theta(V ^2)$

In computer science, the **Floyd–Warshall algorithm** (also known as **Floyd's algorithm**, **Roy–Warshall algorithm**, **Roy–Floyd algorithm**, or the **WFI algorithm**) is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights) and also for finding transitive closure of a relation R . A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices, though it does not return details of the paths themselves. The algorithm is an example of dynamic programming. It was published in its currently recognized form by Robert Floyd in 1962. However, it is essentially the same as algorithms previously published by Bernard Roy in 1959 and also by Stephen Warshall in 1962 for finding the transitive closure of a graph.^[1] The modern formulation of Warshall's algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962.

Algorithm

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with only $\Theta(|V|^3)$ comparisons in a graph. This is remarkable considering that there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Consider a graph G with vertices V , each numbered 1 through N . Further consider a function $\text{shortestPath}(i, j, k)$ that returns the shortest possible path from i to j using vertices only from the set $\{1, 2, \dots, k\}$ as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to $k + 1$.

For each of these pairs of vertices, the true shortest path could be either (1) a path that only uses vertices in the set $\{1, \dots, k\}$ or (2) a path that goes from i to $k + 1$ and then from $k + 1$ to j . We know that the best path from i to j that only uses vertices 1 through k is defined by $\text{shortestPath}(i, j, k)$, and it is clear that if there were a better path from i to $k + 1$ to j , then the length of this path would be the concatenation of the shortest path from i to $k + 1$ (using vertices in $\{1, \dots, k\}$) and the shortest path from $k + 1$ to j (also using vertices in $\{1, \dots, k\}$).

If $w(i, j)$ is the weight of the edge between vertices i and j , we can define $\text{shortestPath}(i, j, k)$ in terms of the following recursive formula: the base case is

$$\text{shortestPath}(i, j, 0) = w(i, j)$$

and the recursive case is

$$\text{shortestPath}(i, j, k) = \min(\text{shortestPath}(i, j, k-1), \text{shortestPath}(i, k, k-1) + \text{shortestPath}(k, j, k-1))$$

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing $\text{shortestPath}(i, j, k)$ for all (i, j) pairs for $k = 1$, then $k = 2$, etc. This process continues until $k = n$, and we have found the shortest path for all (i, j) pairs using any intermediate vertices.

Behavior with negative cycles

A negative cycle is a cycle whose edges sum to a negative value. Between any pair of vertices which form part of a negative cycle, the shortest path is not well-defined because the path can be arbitrarily negative. For numerically meaningful output, the Floyd–Warshall algorithm assumes that there are no negative cycles. Nevertheless, if there are negative cycles, the Floyd–Warshall algorithm can be used to detect them. The intuition is as follows:

- The Floyd–Warshall algorithm iteratively revises path lengths between all pairs of vertices (i, j) , including where $i = j$;
- Initially, the length of the path (i, i) is zero;
- A path $\{(i, k), (k, i)\}$ can only improve upon this if it has length less than zero, i.e. denotes a negative cycle;
- Thus, after the algorithm, (i, i) will be negative if there exists a negative-length path from i back to i .

Hence, to detect negative cycles using the Floyd–Warshall algorithm, one can inspect the diagonal of the path matrix, and the presence of a negative number indicates that the graph contains at least one negative cycle.^[2] Obviously, in an undirected graph a negative edge creates a negative cycle (i.e., a closed walk) involving its incident vertices.

Path reconstruction

The Floyd–Warshall algorithm typically only provides the lengths of the paths between all pairs of vertices. With simple modifications, it is possible to create a method to reconstruct the actual path between any two endpoint vertices. While one may be inclined to store the actual path from each vertex to each other vertex, this is not necessary, and in fact, is very costly in terms of memory. For each vertex, one need only store the information about the highest index intermediate vertex one must pass through if one wishes to arrive at any given vertex. Therefore, information to reconstruct all paths can be stored in a single $N \times N$ matrix *next* where $\text{next}[i][j]$ represents the highest index vertex one must travel through if one intends to take the shortest path from i to j . Implementing such a scheme is trivial; when a new shortest path is found between two vertices, the matrix containing the paths is updated. The *next* matrix is updated along with the *path* matrix, so at completion both tables are complete and accurate, and any entries which are infinite in the *path* table will be null in the *next* table. The path from i to j is the path from i to $\text{next}[i][j]$, followed by the path from $\text{next}[i][j]$ to j . These two shorter paths are determined recursively. This modified algorithm runs with the same time and space complexity as the unmodified algorithm.

```

1  procedure FloydWarshallWithPathReconstruction ()
2      for k := 1 to n
3          for i := 1 to n
4              for j := 1 to n
5                  if path[i][k] + path[k][j] < path[i][j] then {
6                      path[i][j] := path[i][k] + path[k][j];
7                      next[i][j] := k; }
8
9  function Path (i, j)
10     if path[i][j] equals infinity then
11         return "no path";
12     int intermediate := next[i][j];
13     if intermediate equals 'null' then
14         return " "; /* there is an edge from i to j, with no vertices between */
15     else
16         return Path(i, intermediate) + intermediate + Path(intermediate, j);

```

Analysis

To find all n^2 of $\text{shortestPath}(i,j,k)$ (for all i and j) from those of $\text{shortestPath}(i,j,k-1)$ requires $2n^2$ operations. Since we begin with $\text{shortestPath}(i,j,0) = \text{edgeCost}(i,j)$ and compute the sequence of n matrices $\text{shortestPath}(i,j,1)$, $\text{shortestPath}(i,j,2)$, ..., $\text{shortestPath}(i,j,n)$, the total number of operations used is $n \cdot 2n^2 = 2n^3$. Therefore, the complexity of the algorithm is $\Theta(n^3)$.

Applications and generalizations

The Floyd–Warshall algorithm can be used to solve the following problems, among others:

- Shortest paths in directed graphs (Floyd's algorithm).
- Transitive closure of directed graphs (Warshall's algorithm). In Warshall's original formulation of the algorithm, the graph is unweighted and represented by a Boolean adjacency matrix. Then the addition operation is replaced by logical conjunction (AND) and the minimum operation by logical disjunction (OR).
- Finding a regular expression denoting the regular language accepted by a finite automaton (Kleene's algorithm)
- Inversion of real matrices (Gauss–Jordan algorithm) ^[3]
- Optimal routing. In this application one is interested in finding the path with the maximum flow between two vertices. This means that, rather than taking minima as in the pseudocode above, one instead takes maxima. The edge weights represent fixed constraints on flow. Path weights represent bottlenecks; so the addition operation above is replaced by the minimum operation.
- Testing whether an undirected graph is bipartite.
- Fast computation of Pathfinder networks.
- Widest paths/Maximum bandwidth paths

Implementations

Implementations are available for many programming languages.

- For C++, in the `boost::graph` ^[4] library
- For C#, at `QuickGraph` ^[5]
- For Java, in the Apache commons `graph` ^[6] library, or at `Algowiki` ^[7]
- For MATLAB, in the `Matlab_bgl` ^[8] package
- For Perl, in the `Graph` ^[9] module
- For PHP, on page ^[10] and PL/pgSQL, on page ^[11] at Microshell
- For Python, in the `NetworkX` library
- For R, in package `e1017` ^[12]
- For Ruby, in script ^[13]

References

- [1] Weisstein, Eric. "Floyd-Warshall Algorithm" (<http://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html>). *Wolfram MathWorld*. . Retrieved 13 November 2009.
- [2] "Lecture 12: Shortest paths (continued)" (<http://www.ieor.berkeley.edu/~ieor266/Lecture12.pdf>) (PDF). *Network Flows and Graphs*. Department of Industrial Engineering and Operations Research, University of California, Berkeley. 7 October 2008. .
- [3] Penaloza, Rafael. "Algebraic Structures for Transitive Closure" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.7650>). .
- [4] <http://www.boost.org/libs/graph/doc/>
- [5] <http://www.codeplex.com/quickgraph>
- [6] <http://svn.apache.org/repos/asf/commons/dormant/graph2/branches/jakarta/src/java/org/apache/commons/graph/impl/AllPaths.java>
- [7] http://algowiki.net/wiki/index.php?title=Floyd-Warshall%27s_algorithm
- [8] <http://www.mathworks.com/matlabcentral/fileexchange/10922>
- [9] <http://search.cpan.org/search?query=Graph&mode=all>

- [10] <http://www.microshell.com/programming/computing-degrees-of-separation-in-social-networking/2/>
- [11] <http://www.microshell.com/programming/floyd-warshal-algorithm-in-postgresql-plpgsql/3/>
- [12] <http://cran.r-project.org/web/packages/e1071/index.html>
- [13] <https://github.com/chollier/ruby-floyd>

- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L. (1990). *Introduction to Algorithms* (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8.
 - Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565;
 - Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.
- Floyd, Robert W. (June 1962). "Algorithm 97: Shortest Path". *Communications of the ACM* **5** (6): 345. doi:10.1145/367766.368168.
- Ingerman, Peter Z. (November 1962). *Algorithm 141: Path Matrix*. **5**. p. 556. doi:10.1145/368996.369016.
- Kleene, S. C. (1956). "Representation of events in nerve nets and finite automata". In C. E. Shannon and J. McCarthy. *Automata Studies*. Princeton University Press. pp. 3–42.
- Warshall, Stephen (January 1962). "A theorem on Boolean matrices". *Journal of the ACM* **9** (1): 11–12. doi:10.1145/321105.321107.
- Kenneth H. Rosen (2003). *Discrete Mathematics and Its Applications, 5th Edition*. Addison Wesley. ISBN 0-07-119881-4 (ISE).
- Roy, Bernard (1959). "Transitivité et connexité.". *C. R. Acad. Sci. Paris* **249**: 216–218.

External links

- Interactive animation of the Floyd–Warshall algorithm (http://www.pms.informatik.uni-muenchen.de/lehre/compgeometry/Gosper/shortest_path/shortest_path.html#visualization)
 - The Floyd–Warshall algorithm in C#, as part of QuickGraph (<http://quickgraph.codeplex.com/>)
 - Visualization of Floyd's algorithm (http://students.ceid.upatras.gr/~papagel/english/java_docs/allmin.htm)
-

Sorting algorithm

In computer science, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation (reordering) of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.^[1] Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2006). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

Classification

Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the list (n). For typical sorting algorithms good behavior is $O(n \log n)$ and bad behavior is $O(n^2)$. (See Big O notation.) Ideal behavior for a sort is $O(n)$, but this is not possible in the average case. Comparison-based sorting algorithms, which evaluate the elements of the list via an abstract key comparison operation, need at least $O(n \log n)$ comparisons for most inputs.
- Computational complexity of swaps (for "in place" algorithms).
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place". Strictly, an in place sort needs only $O(1)$ memory beyond the items being sorted; sometimes $O(\log(n))$ additional memory is considered "in place".
- Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, *etc.* Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
- Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Stability

Stable sorting algorithms maintain the relative order of records with equal keys. (A key is that portion of the record which is the basis for the sort; it may or may not include all of the record.) If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list. When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of

numbers are to be sorted by their first component:

```
(4, 2) (3, 7) (3, 1) (5, 6)
```

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

```
(3, 7) (3, 1) (4, 2) (5, 6) (order maintained)
(3, 1) (3, 7) (4, 2) (5, 6) (order changed)
```

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional computational cost.

Sorting based on a primary, secondary, tertiary, etc. sort key can be done by any sorting method, taking all sort keys into account in comparisons (in other words, using a single composite sort key). If a sorting method is stable, it is also possible to sort multiple times, each time with one sort key. In that case the keys need to be applied in order of increasing priority.

Example: sorting pairs of numbers as above by second, then first component:

```
(4, 2) (3, 7) (3, 1) (5, 6) (original)
(3, 1) (4, 2) (5, 6) (3, 7) (after sorting by second component)
(3, 1) (3, 7) (4, 2) (5, 6) (after sorting by first component)
```

On the other hand:

```
(3, 7) (3, 1) (4, 2) (5, 6) (after sorting by first component)
(3, 1) (4, 2) (5, 6) (3, 7) (after sorting by second component,
                             order by first component is disrupted).
```

Comparison of algorithms

In this table, n is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. These are all comparison sorts. The run time and the memory of algorithms could be measured using various notations like theta, omega, Big-O, small-o, etc. The memory and the run times below are applicable for all the 5 notations.

Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends	Partitioning	Quicksort is usually done in place with $O(\log(n))$ stack space. Most implementations are unstable, as stable in-place partitioning is more complex. Naïve variants use an $O(n)$ space array to store the partition.
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends; worst case is n	Yes	Merging	Highly parallelizable (up to $O(\log(n))$) for processing large amounts of data.
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes	Merging	Implemented in Standard Template Library (STL); ^[2] can be implemented as a stable sort based on stable in-place merging. ^[3]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, where d is the number of inversions
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space, for example using lists. ^[4] Used to sort this table in Safari or other Webkit web browser. ^[5]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	n comparisons when the data is already sorted or reverse sorted.
Shell sort	n	$n(\log n)^2$ or $n^{3/2}$	Depends on gap sequence; best known is $n(\log n)^2$	1	No	Insertion	
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes	Insertion	When using a self-balancing binary search tree
Cycle sort	—	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes
Library sort	—	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	—	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences within $O(n \log n)$
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive sort - n comparisons when the data is already sorted, and 0 swaps.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	—	$n \log n$	$n \log n$			Selection	
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging	
Comb sort	n	$n \log n$	n^2	1	No	Exchanging	Small code size
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Bogosort	n	$n \cdot n!$	$n \cdot n! \rightarrow \infty$	1	No	Luck	Randomly permute the array and check if sorted.
[6]	—	$n \log n$	$n \log n$	1	Yes		

The following table describes integer sorting algorithms and other sorting algorithms that are not comparison sorts. As such, they are not limited by a $\Omega(n \log n)$ lower bound. Complexities below are in terms of n , the number of items to be sorted, k , the size of each key, and d , the digit size used by the implementation. Many of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that $n \ll 2^k$, where \ll means "much less than."

Non-comparison sorts

Name	Best	Average	Worst	Memory	Stable	$n \ll 2^k$	Notes
Pigeonhole sort	—	$n + 2^k$	$n + 2^k$	2^k	Yes	Yes	
Bucket sort (uniform keys)	—	$n + k$	$n^2 \cdot k$	$n \cdot k$	Yes	No	Assumes uniform distribution of elements from the domain in the array. ^[7]
Bucket sort (integer keys)	—	$n + r$	$n + r$	$n + r$	Yes	Yes	r is the range of numbers to be sorted. If $r = \mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$ ^[8]
Counting sort	—	$n + r$	$n + r$	$n + r$	Yes	Yes	r is the range of numbers to be sorted. If $r = \mathcal{O}(n)$ then Avg RT = $\mathcal{O}(n)$ ^[7]
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	n	Yes	No	[7][8]
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + \frac{k}{d} \cdot 2^d$	Yes	No	Stable version uses an external array of size n to hold all of the bins
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$\frac{k}{d} \cdot 2^d$	No	No	In-Place. k/d recursion levels, 2^d for count array
Spreadsor	—	$n \cdot \frac{k}{d}$	$n \cdot \left(\frac{k}{s} + d\right)$	$\frac{k}{d} \cdot 2^d$	No	No	Asymptotics are based on the assumption that $n \ll 2^k$, but the algorithm does not require this.

The following table describes some sorting algorithms that are impractical for real-life use due to extremely poor performance or a requirement for specialized hardware.

Name	Best	Average	Worst	Memory	Stable	Comparison	Other notes
Bead sort	—	N/A	N/A	—	N/A	No	Requires specialized hardware
Simple pancake sort	—	n	n	$\log n$	No	Yes	Count is number of flips.
Spaghetti (Poll) sort	n	n	n	n^2	Yes	Polling	This A linear-time, analog algorithm for sorting a sequence of items, requiring $\mathcal{O}(n)$ stack space, and the sort is stable. This requires n parallel processors. Spaghetti sort#Analysis
Sorting networks	—	$\log n$	$\log n$	$n \cdot \log(n)$	Yes	No	Requires a custom circuit of size $\mathcal{O}(n \cdot \log(n))$

Additionally, theoretical computer scientists have detailed other sorting algorithms that provide better than $\mathcal{O}(n \log n)$ time complexity with additional constraints, including:

- Han's algorithm, a deterministic algorithm for sorting keys from a domain of finite size, taking $\mathcal{O}(n \log \log n)$ time and $\mathcal{O}(n)$ space.^[9]
- Thorup's algorithm, a randomized algorithm for sorting keys from a domain of finite size, taking $\mathcal{O}(n \log \log n)$ time and $\mathcal{O}(n)$ space.^[10]
- An integer sorting algorithm taking $\mathcal{O}\left(n \sqrt{\log \log n}\right)$ expected time and $\mathcal{O}(n)$ space.^[11]

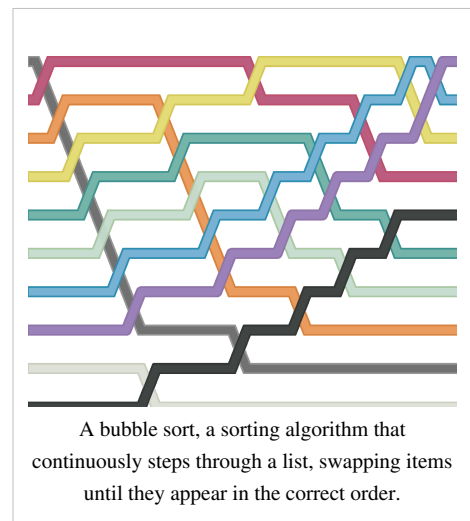
Algorithms not yet compared above include:

- Odd-even sort
- Flashsort
- Burtsort
- Postman sort
- Stooge sort
- Samplesort
- Bitonic sorter

Summaries of popular sorting algorithms

Bubble sort

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average and worst case performance is $O(n^2)$, so it is rarely used to sort large, unordered, data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any number of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only $2n$ time.



Selection sort

Selection sort is an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

Insertion sort

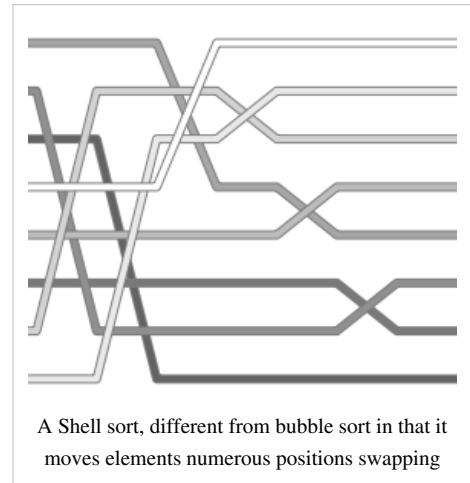
Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort (see below) is a variant of insertion sort that is more efficient for larger lists.

Shell sort

Shell sort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

Comb sort

Comb sort is a relatively simple sorting algorithm originally designed by Włodzimierz Dobosiewicz in 1980. Later it was rediscovered and popularized by Stephen Lacey and Richard Box with a *Byte Magazine* article published in April 1991. Comb sort improves on bubble sort, and rivals algorithms like Quicksort. The basic idea is to eliminate *turtles*, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. (*Rabbits*, large values around the beginning of the list, do not pose a problem in bubble sort)



Merge sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is $O(n \log n)$. Merge sort has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in the programming languages Perl,^[12] Python (as timsort^[13]), and Java (also uses timsort as of JDK7^[14]), among others. Merge sort has been used in Java at least since 2000 in JDK1.3.^{[15][16]}

Heapsort

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes $O(\log n)$ time, instead of $O(n)$ for a linear scan as in simple selection sort. This allows Heapsort to run in $O(n \log n)$ time, and this is also the worst case complexity.

Quicksort

Quicksort is a divide and conquer algorithm which relies on a *partition* operation: to partition an array an element called a *pivot* is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest $O(\log n)$ space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower $O(n^2)$ performance, if at each step the median is chosen as the pivot then the algorithm works in

$O(n \log n)$. Finding the median however, is an $O(n)$ operation on unsorted lists and therefore exacts its own penalty with sorting.

Counting sort

Counting sort is applicable when each input is known to belong to a particular set, S , of possibilities. The algorithm runs in $O(|S| + n)$ time and $O(|S|)$ memory where n is the length of the input. It works by creating an integer array of size $|S|$ and using the i th bin to count the occurrences of the i th member of S in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm cannot often be used because S needs to be reasonably small for it to be efficient, but the algorithm is extremely fast and demonstrates great asymptotic behavior as n increases. It also can be modified to provide stable behavior.

Bucket sort

Bucket sort is a divide and conquer sorting algorithm that generalizes Counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. A variation of this method called the single buffered count sort is faster than quicksort.

Due to the fact that bucket sort must use a limited number of buckets it is best suited to be used on data sets of a limited scope. Bucket sort would be unsuitable for data that have a lot of variation, such as social security numbers.

Radix sort

Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in $O(n \cdot k)$ time. Radix sort can process digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit, and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the counting sort algorithm to be used internally by the radix sort. Hybrid sorting approach, such as using insertion sort for small bins improves performance of radix sort significantly.

Distribution sort

Distribution sort refers to any sorting algorithm where data are distributed from their input to multiple intermediate structures which are then gathered and placed on the output. For example, both bucket sort and flashsort are distribution based sorting algorithms.

Timsort

Timsort finds runs in the data, creates runs with insertion sort if necessary, and then uses merge sort to create the final sorted list. It has the same complexity ($O(n \log n)$) in the average and worst cases, but with pre-sorted data it goes down to $O(n)$.

Memory usage patterns and index sorting

When the size of the array to be sorted approaches or exceeds the available primary memory, so that (much slower) disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can dominate the performance characteristics of an algorithm. Thus, the number of passes and the localization of comparisons can be more important than the raw number of comparisons, since comparisons of nearby elements to one another happen at system bus speed (or, with caching, even at CPU speed), which, compared to disk speed, is virtually instantaneous.

For example, the popular recursive quicksort algorithm provides quite reasonable performance with adequate RAM, but due to the recursive way that it copies portions of the array it becomes much less practical when the array does not fit in RAM, because it may cause a number of slow copy or move operations to and from disk. In that scenario, another algorithm may be preferable even if it requires more total comparisons.

One way to work around this problem, which works well when complex records (such as in a relational database) are being sorted by a relatively small key field, is to create an index into the array and then sort the index, rather than the entire array. (A sorted version of the entire array can then be produced with one pass, reading from the index, but often even that is unnecessary, as having the sorted index is adequate.) Because the index is much smaller than the entire array, it may fit easily in memory where the entire array would not, effectively eliminating the disk-swapping problem. This procedure is sometimes called "tag sort".^[17]

Another technique for overcoming the memory-size problem is to combine two algorithms in a way that takes advantages of the strength of each to improve overall performance. For instance, the array might be subdivided into chunks of a size that will fit easily in RAM (say, a few thousand elements), the chunks sorted using an efficient algorithm (such as quicksort or heapsort), and the results merged as per mergesort. This is less efficient than just doing mergesort in the first place, but it requires less physical RAM (to be practical) than a full quicksort on the whole array.

Techniques can also be combined. For sorting very large sets of data that vastly exceed system memory, even the index may need to be sorted using an algorithm or combination of algorithms designed to perform reasonably with virtual memory, i.e., to reduce the amount of swapping required.

Some other sorting algorithms also i.e. New friends sort algorithm, Relative split and concatenate sort etc

Inefficient/humorous sorts

Some algorithms are extremely slow compared to those discussed above, such as the Bogosort $O(n \cdot n!)$ and the Stooge sort $O(n^{2.7})$.

References

- [1] Demuth, H. Electronic Data Sorting. PhD thesis, Stanford University, 1956.
- [2] http://www.sgi.com/tech/stl/stable_sort.html
- [3] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8381>
- [4] http://www.algolist.net/Algorithms/Sorting/Selection_sort
- [5] <http://svn.webkit.org/repository/webkit/trunk/Source/JavaScriptCore/runtime/ArrayPrototype.cpp>
- [6] <http://www.springerlink.com/content/d7348168624070v7/>
- [7] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7.
- [8] Goodrich, Michael T.; Tamassia, Roberto (2002). "4.5 Bucket-Sort and Radix-Sort". *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons. pp. 241–243.
- [9] Y. Han. *Deterministic sorting in $\mathcal{O}(n \log \log n)$ time and linear space*. Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, Montreal, Quebec, Canada, 2002, p.602-608.

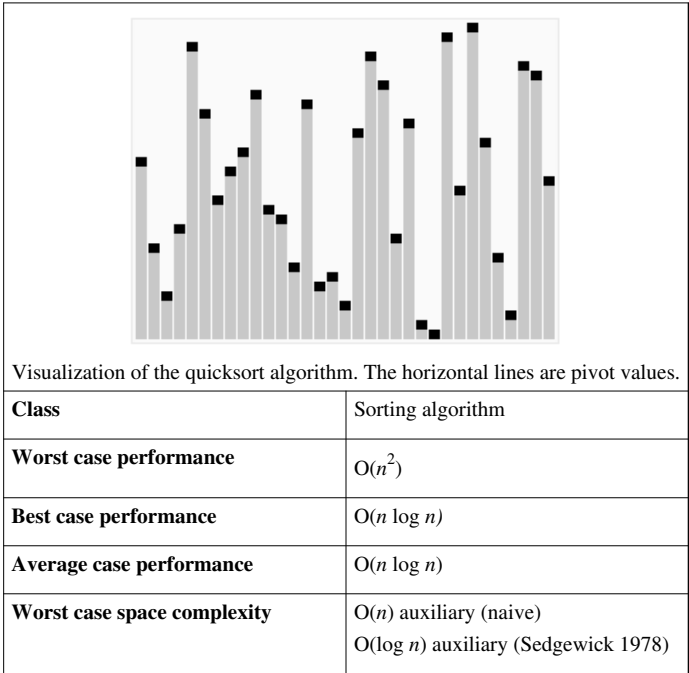
- [10] M. Thorup. *Randomized Sorting in $\mathcal{O}(n \log \log n)$ Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations*. Journal of Algorithms, Volume 42, Number 2, February 2002, pp. 205-230(26)
- [11] Han, Y. and Thorup, M. 2002. Integer Sorting in $\mathcal{O}\left(n\sqrt{\log \log n}\right)$ Expected Time and Linear Space. In *Proceedings of the 43rd Symposium on Foundations of Computer Science* (November 16–19, 2002). FOCS. IEEE Computer Society, Washington, DC, 135-144.
- [12] Perl sort documentation (<http://perldoc.perl.org/functions/sort.html>)
- [13] Tim Peters's original description of timsort (<http://svn.python.org/projects/python/trunk/Objects/listsort.txt>)
- [14] <http://hg.openjdk.java.net/jdk7/tl/jdk/rev/bfd7abda8f79>
- [15] Merge sort in Java 1.3 ([http://java.sun.com/j2se/1.3/docs/api/java/util/Arrays.html#sort\(java.lang.Object\[\]\)](http://java.sun.com/j2se/1.3/docs/api/java/util/Arrays.html#sort(java.lang.Object[]))), Sun.
- [16] Java 1.3 live since 2000
- [17] Definition of "tag sort" according to PC Magazine (http://www.pcmag.com/encyclopedia_term/0,2542,t=tag+sort&i=52532,00.asp)
- D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*.

External links

- Sorting Algorithm Animations (<http://www.sorting-algorithms.com/>) - Graphical illustration of how different algorithms handle different kinds of data sets.
- Sequential and parallel sorting algorithms (<http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm>) - Explanations and analyses of many sorting algorithms.
- Dictionary of Algorithms, Data Structures, and Problems (<http://www.nist.gov/dads/>) - Dictionary of algorithms, techniques, common functions, and problems.
- Slightly Skeptical View on Sorting Algorithms (<http://www.softpanorama.org/Algorithms/sorting.shtml>)
Discusses several classic algorithms and promotes alternatives to the quicksort algorithm.

Quicksort

Quicksort



Quicksort is a sorting algorithm developed by Tony Hoare that, on average, makes $O(n \log n)$ comparisons to sort n items. It is also known as **partition-exchange sort**. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare. Quicksort is often faster in practice than other $O(n \log n)$ algorithms.^[1] Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only $O(\log n)$ additional space.^[2]

Quicksort is a comparison sort and, in efficient implementations, is not a stable sort.

History

The quicksort algorithm was developed in 1960 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University. At that time, Hoare worked in a project on machine translation for the National Physical Laboratory. He developed the algorithm in order to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.^[3]

Algorithm

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Simple version

In simple pseudocode, the algorithm might be expressed as this:

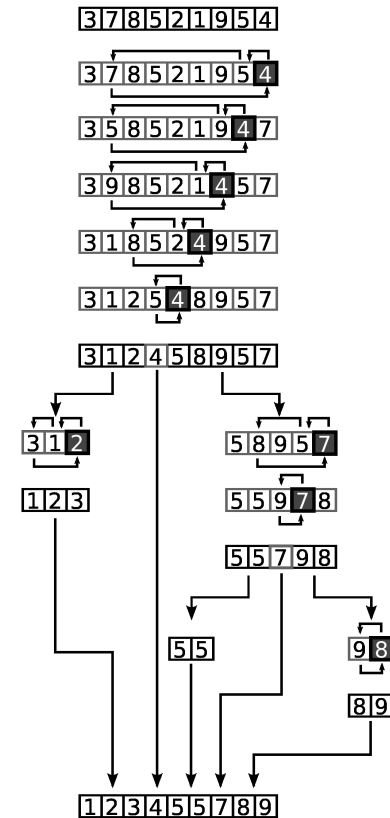
```
function quicksort('array')
  if length('array') ≤ 1
    return 'array' // an array of zero or one elements is
already sorted
  select and remove a pivot value 'pivot' from 'array'
  create empty lists 'less' and 'greater'
  for each 'x' in 'array'
    if 'x' ≤ 'pivot' then append 'x' to 'less'
    else append 'x' to 'greater'
  return concatenate(quicksort('less'), 'pivot',
quicksort('greater')) // two recursive calls
```

Notice that we only examine elements by comparing them to other elements. This makes quicksort a comparison sort. This version is also a stable sort (assuming that the "for each" method retrieves elements in original order, and the pivot selected is the last among those of equal value).

The correctness of the partition algorithm is based on the following two arguments:

- At each iteration, all the elements processed so far are in the desired position: before the pivot if less than the pivot's value, after the pivot if greater than the pivot's value (loop invariant).
- Each iteration leaves one fewer element to be processed (loop variant).

The correctness of the overall algorithm can be proven via induction: for zero or one element, the algorithm leaves the data unchanged; for a larger data set it produces the concatenation of two parts, elements less than the pivot and elements greater than it, themselves sorted by the recursive hypothesis.



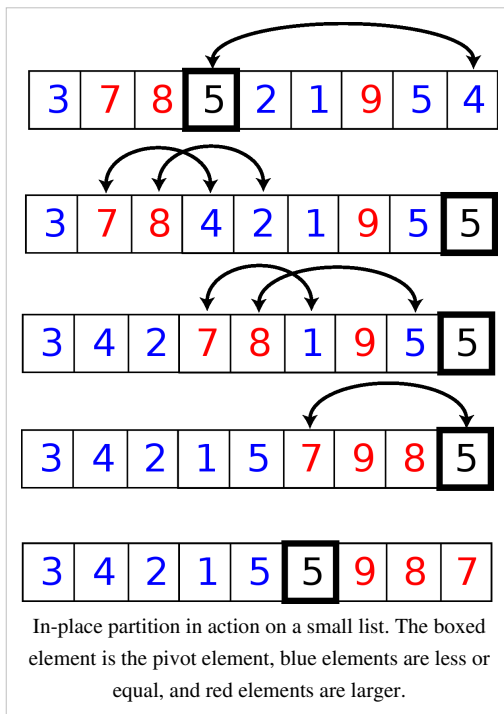
Full example of quicksort on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition. However, always choosing the last element in the partition as the pivot in this way results in poor performance ($O(n^2)$) on *already sorted* lists, or lists of identical elements. Since sub-lists of sorted / identical elements crop up a lot towards the end of a sorting procedure on a large set, versions of the quicksort algorithm which choose the pivot as the middle element run much more quickly than the algorithm described in this diagram on large sets of numbers.

6 5 3 1 8 7 2 4

An example of quicksort.

In-place version

The disadvantage of the simple version above is that it requires $O(n)$ extra storage space, which is as bad as merge sort. The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complex version which uses an in-place partition algorithm and can achieve the complete sort using $O(\log n)$ space (not counting the input) on average (for the call stack). We start with a partition function:



```
// left is the index of the leftmost element of the array
// right is the index of the rightmost element of the array
(inclusive)
// number of elements in subarray = right-left+1
function partition(array, 'left', 'right', 'pivotIndex')
    'pivotValue' := array['pivotIndex']
    swap array['pivotIndex'] and array['right'] // Move pivot to end
    'storeIndex' := 'left'
    for 'i' from 'left' to 'right' - 1 // left ≤ i < right
        if array['i'] < 'pivotValue'
            swap array['i'] and array['storeIndex']
            'storeIndex' := 'storeIndex' + 1
    swap array['storeIndex'] and array['right'] // Move pivot to its
final place
return 'storeIndex'
```

This is the in-place partition algorithm. It partitions the portion of the array between indexes *left* and *right*, inclusively, by moving all elements less than `array[pivotIndex]` before the pivot, and the equal or greater elements after it. In the process it also finds the final position for the pivot element, which it returns. It temporarily moves the pivot element to the end of the subarray, so that it doesn't get in the way. Because it only uses exchanges, the final list has the same elements as the original list. Notice that an element may be exchanged multiple times before reaching its final place. Also, in case of pivot duplicates in the input array, they can be spread across the right subarray, in any order. This doesn't represent a partitioning failure, as further sorting will reposition and finally "glue" them together.

This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the `storeIndex`. However, this form is probably the easiest to understand.

Once we have this, writing quicksort itself is easy:

```
function quicksort(array, 'left', 'right')

    // If the list has 2 or more items
    if 'left' < 'right'

        // See "Choice of pivot" section below for possible choices
        choose any 'pivotIndex' such that 'left' ≤ 'pivotIndex' ≤
'right'

        // Get lists of bigger and smaller items and final position
of pivot
        'pivotNewIndex' := partition(array, 'left', 'right',
'pivotIndex')

        // Recursively sort elements smaller than the pivot
        quicksort(array, 'left', 'pivotNewIndex' - 1)

        // Recursively sort elements at least as big as the pivot
        quicksort(array, 'pivotNewIndex' + 1, 'right')
```

Each recursive call to this *quicksort* function reduces the size of the array being sorted by at least one element, since in each invocation the element at *pivotNewIndex* is placed in its final position. Therefore, this algorithm is guaranteed to terminate after at most n recursive calls. However, since *partition* reorders elements within a partition, this version of quicksort is not a stable sort.

Implementation issues

Choice of pivot

In very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element. Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case. The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot (as recommended by R. Sedgwick).^{[4][5]}

Selecting a pivot element is also complicated by the existence of integer overflow. If the boundary indices of the subarray being sorted are sufficiently large, the naïve expression for the middle index, $(left + right)/2$, will cause overflow and provide an invalid pivot index. This can be overcome by using, for example, $left + (right-left)/2$ to index the middle element, at the cost of more complex arithmetic. Similar issues arise in some other methods of selecting the pivot element.

Optimizations

Two other important optimizations, also suggested by R. Sedgwick, as commonly acknowledged, and widely used in practice are:^{[6][7][8]}

- To make sure at most $O(\log N)$ space is used, recurse first into the smaller half of the array, and use a tail call to recurse into the other.
- Use insertion sort, which has a smaller constant factor and is thus faster on small arrays, for invocations on such small arrays (i.e. where the length is less than a threshold t determined experimentally). This can be implemented by leaving such arrays unsorted and running a single insertion sort pass at the end, because insertion sort handles nearly sorted arrays efficiently. A separate insertion sort of each small segment as they are identified adds the overhead of starting and stopping many small sorts, but avoids wasting effort comparing keys across the many segment boundaries, which keys will be in order due to the workings of the quicksort process. It also improves the cache use.

Parallelization

Like merge sort, quicksort can also be parallelized due to its divide-and-conquer nature. Individual in-place partition operations are difficult to parallelize, but once divided, different sections of the list can be sorted in parallel. The following is a straightforward approach: If we have P processors, we can divide a list of n elements into P sublists in $O(n)$ average time, then sort each of these in $O\left(\frac{n}{P} \log \frac{n}{P}\right)$ average time. Ignoring the $O(n)$ preprocessing and merge times, this is linear speedup. If the split is blind, ignoring the values, the merge naïvely costs $O(n)$. If the split partitions based on a succession of pivots, it is tricky to parallelize and naïvely costs $O(n)$. Given $O(\log n)$ or more processors, only $O(n)$ time is required overall, whereas an approach with linear speedup would achieve $O(\log n)$ time for overall.

One advantage of this simple parallel quicksort over other parallel sort algorithms is that no synchronization is required, but the disadvantage is that sorting is still $O(n)$ and only a sublinear speedup of $O(\log n)$ is achieved. A new thread is started as soon as a sublist is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds.^[9] For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW PRAM with n processors by performing partitioning implicitly.^[10]

Formal analysis

Average-case analysis using discrete probability

Quicksort takes $O(n \log n)$ time on average, when the input is a random permutation. Why? For a start, it is not hard to see that the partition operation takes $O(n)$ time.

In the most unbalanced case, each time we perform a partition we divide the list into two sublists of size 0 and $n - 1$ (for example, if all elements of the array are equal). This means each recursive call processes a list of size one less than the previous list. Consequently, we can make $n - 1$ nested calls before we reach a list of size 1. This means that the call tree is a linear chain of $n - 1$ nested calls. The i th call does $O(n - i)$ work to do the partition, and $\sum_{i=0}^{n-1} (n - i) = O(n^2)$, so in that case Quicksort takes $O(n^2)$ time. That is the worst case: given knowledge of which comparisons are performed by the sort, there are adaptive algorithms that are effective at generating worst-case input for quicksort on-the-fly, regardless of the pivot selection strategy.^[11]

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\log n / \log 2$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\log n / \log 2$. But no two calls at

the same level of the call tree process the same part of the original list; thus, each level of calls needs only $O(n)$ time all together (each call has some constant overhead, but since there are only $O(n)$ calls at each level, this is subsumed in the $O(n)$ factor). The result is that the algorithm uses only $O(n \log n)$ time.

In fact, it's not necessary to be perfectly balanced; even if each pivot splits the elements with 75% on one side and 25% on the other side (or any other fixed fraction), the call depth is still limited to $\log n / \log(4/3)$, so the total running time is still $O(n \log n)$.

So what happens on average? If the pivot has rank somewhere in the middle 50 percent, that is, between the 25th percentile and the 75th percentile, then it splits the elements with at least 25% and at most 75% on each side. If we could consistently choose a pivot from the two middle 50 percent, we would only have to split the list at most $\log n / \log(4/3)$ times before reaching lists of size 1, yielding an $O(n \log n)$ algorithm.

When the input is a random permutation, the pivot has a random rank, and so it is not guaranteed to be in the middle 50 percent. However, when we start from a random permutation, in each recursive call the pivot has a random rank in its list, and so it is in the middle 50 percent about half the time. That is good enough. Imagine that you flip a coin: heads means that the rank of the pivot is in the middle 50 percent, tail means that it isn't. Imagine that you are flipping a coin over and over until you get k heads. Although this could take a long time, on average only $2k$ flips are required, and the chance that you won't get k heads after $100k$ flips is highly improbable (this can be made rigorous using Chernoff bounds). By the same argument, Quicksort's recursion will terminate on average at a call depth of only $2(\log n / \log(4/3))$. But if its average call depth is $O(\log n)$, and each level of the call tree processes at most n elements, the total amount of work done on average is the product, $O(n \log n)$. Note that the algorithm does not have to verify that the pivot is in the middle half—if we hit it any constant fraction of the times, that is enough for the desired complexity.

Average-case analysis using recurrences

An alternative approach is to set up a recurrence relation for the $T(n)$ factor, the time needed to sort a list of size n . In the most unbalanced case, a single Quicksort call involves $O(n)$ work plus two recursive calls on lists of size 0 and $n - 1$, so the recurrence relation is

$$T(n) = O(n) + T(0) + T(n - 1) = O(n) + T(n - 1).$$

This is the same relation as for insertion sort and selection sort, and it solves to worst case $T(n) = O(n^2)$.

In the most balanced case, a single quicksort call involves $O(n)$ work plus two recursive calls on lists of size $n/2$, so the recurrence relation is

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right).$$

The master theorem tells us that $T(n) = O(n \log n)$.

The outline of a formal proof of the $O(n \log n)$ expected time complexity follows. Assume that there are no duplicates as duplicates could be handled with linear time pre- and post-processing, or considered cases easier than the analyzed. When the input is a random permutation, the rank of the pivot is uniform random from 0 to $n-1$. Then the resulting parts of the partition have sizes i and $n-i-1$, and i is uniform random from 0 to $n-1$. So, averaging over all possible splits and noting that the number of comparisons for the partition is $n - 1$, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$C(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (C(i) + C(n - i - 1))$$

Solving the recurrence gives $C(n) = 2n \ln n = 1.39n \log_2 n$.

This means that, on average, quicksort performs only about 39% worse than in its best case. In this sense it is closer to the best case than the worst case. Also note that a comparison sort cannot use less than $\log_2(n!)$ comparisons on

average to sort n items (as explained in the article Comparison sort) and in case of large n , Stirling's approximation yields $\log_2(n!) \approx n(\log_2 n - \log_2 e)$, so quicksort is not much worse than an ideal comparison sort. This fast average runtime is quicksort's practical dominance over other sorting algorithms.

Analysis of Randomized quicksort

Using the same analysis, one can show that Randomized quicksort has the desirable property that, for any input, it requires only $O(n \log n)$ expected time (averaged over all choices of pivots). However, there exists a combinatorial proof, more elegant than both the analysis using discrete probability and the analysis using recurrences.

To each execution of Quicksort corresponds the following binary search tree (BST): the initial pivot is the root node; the pivot of the left half is the root of the left subtree, the pivot of the right half is the root of the right subtree, and so on. The number of comparisons of the execution of Quicksort equals the number of comparisons during the construction of the BST by a sequence of insertions. So, the average number of comparisons for randomized Quicksort equals the average cost of constructing a BST when the values inserted (x_1, x_2, \dots, x_n) form a random permutation.

Consider a BST created by insertion of a sequence (x_1, x_2, \dots, x_n) of values forming a random permutation. Let C denote the cost of creation of the BST. We have: $C = \sum_i \sum_{j < i} (\text{whether during the insertion of } x_i \text{ there was a comparison to } x_j)$.

By linearity of expectation, the expected value $E(C)$ of C is $E(C) = \sum_i \sum_{j < i} \Pr(\text{during the insertion of } x_i \text{ there was a comparison to } x_j)$.

Fix i and $j < i$. The values x_1, x_2, \dots, x_j , once sorted, define $j+1$ intervals. The core structural observation is that x_i is compared to x_j in the algorithm if and only if x_i falls inside one of the two intervals adjacent to x_j .

Observe that since (x_1, x_2, \dots, x_n) is a random permutation, $(x_1, x_2, \dots, x_j, x_i)$ is also a random permutation, so the probability that x_i is adjacent to x_j is exactly $2/(j+1)$.

We end with a short calculation: $E(C) = \sum_i \sum_{j < i} 2/(j+1) = O(\sum_i \log i) = O(n \log n)$.

Space complexity

The space used by quicksort depends on the version used.

The in-place version of quicksort has a space complexity of $O(\log n)$, even in the worst case, when it is carefully implemented using the following strategies:

- in-place partitioning is used. This unstable partition requires $O(1)$ space.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(\log n)$ space. Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack. This idea, as discussed above, was described by R. Sedgwick, and keeps the stack depth bounded by $O(\log n)$.^{[4][5]}

Quicksort with in-place and unstable partitioning uses only constant additional space before making any recursive call. Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most $O(\log n)$ nested recursive calls, it uses $O(\log n)$ space. However, without Sedgwick's trick to limit the recursive calls, in the worst case quicksort could make $O(n)$ nested recursive calls and need $O(n)$ auxiliary space.

From a bit complexity viewpoint, variables such as *left* and *right* do not use constant space; it takes $O(\log n)$ bits to index into a list of n items. Because there are such variables in every stack frame, quicksort using Sedgwick's trick requires $O((\log n)^2)$ bits of space. This space requirement isn't too terrible, though, since if the list contained distinct elements, it would need at least $O(n \log n)$ bits of space.

Another, less common, not-in-place, version of quicksort uses $O(n)$ space for working storage and can implement a stable sort. The working storage allows the input array to be easily partitioned in a stable manner and then copied back to the input array for successive recursive calls. Sedgwick's optimization is still appropriate.

Selection-based pivoting

A selection algorithm chooses the k th smallest of a list of numbers; this is an easier problem in general than sorting. One simple but effective selection algorithm works nearly in the same manner as quicksort, except that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist which contains the desired element. This small change lowers the average complexity to linear or $O(n)$ time, and makes it an in-place algorithm. A variation on this algorithm brings the worst-case time down to $O(n)$ (see selection algorithm for more information).

Conversely, once we know a worst-case $O(n)$ selection algorithm is available, we can use it to find the ideal pivot (the median) at every step of quicksort, producing a variant with worst-case $O(n \log n)$ running time. In practical implementations, however, this variant is considerably slower on average.

Variants

There are four well known variants of quicksort:

- **Balanced quicksort:** choose a pivot likely to represent the middle of the values to be sorted, and then follow the regular quicksort algorithm.
- **External quicksort:** The same as regular quicksort except the pivot is replaced by a buffer. First, read the $M/2$ first and last elements into the buffer and sort them. Read the next element from the beginning or end to balance writing. If the next element is less than the least of the buffer, write it to available space at the beginning. If greater than the greatest, write it to the end. Otherwise write the greatest or least of the buffer, and put the next element in the buffer. Keep the maximum lower and minimum upper keys written to avoid resorting middle elements that are in order. When done, write the buffer. Recursively sort the smaller partition, and loop to sort the remaining partition. This is a kind of three-way quicksort in which the middle partition (buffer) represents a sorted subarray of elements that are *approximately* equal to the pivot.
- **Three-way radix quicksort** (developed by Sedgwick and also known as **multikey quicksort**): is a combination of radix sort and quicksort. Pick an element from the array (the pivot) and consider the first character (key) of the string (multikey). Partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character. Recursively sort the "less than" and "greater than" partitions on the same character. Recursively sort the "equal to" partition by the next character (key). Given we sort using bytes or words of length W bits, the best case is $O(KN)$ and the worst case $O(2^K N)$ or at least $O(N^2)$ as for standard quicksort, given for unique keys $N < 2^K$, and K is a hidden constant in all standard comparison sort algorithms including quicksort. This is a kind of three-way quicksort in which the middle partition represents a (trivially) sorted subarray of elements that are *exactly* equal to the pivot.
- **Quick radix sort** (also developed by Powers as a $o(K)$ parallel PRAM algorithm). This is again a combination of radix sort and quicksort but the quicksort left/right partition decision is made on successive bits of the key, and is thus $O(KN)$ for N K -bit keys. Note that all comparison sort algorithms effectively assume an ideal K of $O(\log N)$ as if k is smaller we can sort in $O(N)$ using a hash table or integer sorting, and if $K \gg \log N$ but elements are unique within $O(\log N)$ bits, the remaining bits will not be looked at by either quicksort or quick radix sort, and otherwise all comparison sorting algorithms will also have the same overhead of looking through $O(K)$ relatively useless bits but quick radix sort will avoid the worst case $O(N^2)$ behaviours of standard quicksort and quick radix sort, and will be faster even in the best case of those comparison algorithms under these conditions of $\text{uniqueprefix}(K) \gg \log N$. See Powers^[12] for further discussion of the hidden overheads in comparison, radix and parallel sorting.

Comparison with other sorting algorithms

Quicksort is a space-optimized version of the binary tree sort. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls. The algorithms make exactly the same comparisons, but in a different order. An often desirable property of a sorting algorithm is stability - that is the order of elements that compare equal is not changed, allowing controlling order of multikey tables (e.g. directory or folder listings) in a natural way. This property is hard to maintain for in situ (or in place) quicksort (that uses only constant additional space for pointers and buffers, and $\log N$ additional space for the management of explicit or implicit recursion). For variant quicksorts involving extra memory due to representations using pointers (e.g. lists or trees) or files (effectively lists), it is trivial to maintain stability. The more complex, or disk-bound, data structures tend to increase time cost, in general making increasing use of virtual memory or disk.

The most direct competitor of quicksort is heapsort. Heapsort's worst-case running time is always $O(n \log n)$. But, heapsort is assumed to be on average somewhat slower than standard in-place quicksort. This is still debated and in research, with some publications indicating the opposite.^{[13][14]} Introsort is a variant of quicksort that switches to heapsort when a bad case is detected to avoid quicksort's worst-case running time. If it is known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it.

Quicksort also competes with mergesort, another recursive sort algorithm but with the benefit of worst-case $O(n \log n)$ running time. Mergesort is a stable sort, unlike standard in-place quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Like mergesort, quicksort can be implemented as an in-place stable sort,^[15] but this is seldom done. Although quicksort can be written to operate on linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that, when operating on arrays, efficient implementations require $O(n)$ auxiliary space, whereas the variant of quicksort with in-place partitioning and tail recursion uses only $O(\log n)$ space. (Note that when operating on linked lists, mergesort only requires a small, constant amount of auxiliary storage.)

Bucket sort with two buckets is very similar to quicksort; the pivot in this case is effectively the value in the middle of the value range, which does well on average for uniformly distributed inputs.

Notes

- [1] S. S. Skiena, *The Algorithm Design Manual*, Second Edition, Springer, 2008, p. 129
- [2] "Data structures and algorithm: Quicksort" (<http://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsor1a.html>). Auckland University. .
- [3] "An Interview with C.A.R. Hoare" (<http://cacm.acm.org/magazines/2009/3/21782-an-interview-with-car-hoare/fulltext>). Communications of the ACM, March 2009 ("premium content"). .
- [4] R. Sedgewick, *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*, 3rd Edition, Addison-Wesley
- [5] R. Sedgewick, Implementing quicksort programs, *Comm. ACM*, 21(10):847-857, 1978.
- [6] qsort.c in GNU libc: (<http://www.cs.columbia.edu/~hgs/teaching/isp/hw/qsor.c>), (<http://repo.or.cz/w/glibc.git/blob/HEAD:/stdlib/qsor.c>)
- [7] http://home.tiscalinet.ch/t_wolf/tw/ada95/sorting/index.html
- [8] <http://www.ugrad.cs.ubc.ca/~cs260/chnotes/ch6/Ch6CovCompiled.html>
- [9] R. Miller, L. Boxer, *Algorithms Sequential & Parallel, A Unified Approach*, Prentice Hall, NJ, 2006
- [10] David M. W. Powers, Parallelized Quicksort and Radixsort with Optimal Speedup (<http://citeseer.ist.psu.edu/327487.html>), *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.
- [11] M. D. McIlroy. A Killer Adversary for Quicksort. *Software Practice and Experience*: vol.29, no.4, 341-344. 1999. At Citeseer (<http://citeseer.ist.psu.edu/212772.html>)
- [12] David M. W. Powers, Parallel Unification: Practical Complexity (<http://david.wardpowers.info/Research/AI/papers/199501-ACAW-PUPC.pdf>), Australasian Computer Architecture Workshop, Flinders University, January 1995
- [13] Hsieh, Paul (2004). "Sorting revisited." (<http://www.azillionmonkeys.com/qed/sort.html>). www.azillionmonkeys.com. . Retrieved 26 April 2010.
- [14] MacKay, David (1 December 2005). "Heapsort, Quicksort, and Entropy" (<http://users.aims.ac.za/~mackay/sorting/sorting.html>). users.aims.ac.za/~mackay. . Retrieved 26 April 2010.

- [15] A Java implementation of in-place stable quicksort (<http://h2database.googlecode.com/svn/trunk/h2/src/tools/org/h2/dev/sort/InPlaceStableQuicksort.java>)

References

- R. Sedgwick, Implementing quicksort programs, *Comm. ACM*, 21(10):847-857, 1978. Implementing Quicksort Programs (<http://delivery.acm.org/10.1145/360000/359631/p847-sedgwick.pdf?key1=359631&key2=9191985921&coll=DL&dl=ACM&CFID=6618157&CFTOKEN=73435998>)
- Brian C. Dean, "A Simple Expected Running Time Analysis for Randomized 'Divide and Conquer' Algorithms." *Discrete Applied Mathematics* 154(1): 1-5. 2006.
- Hoare, C. A. R. "Partition: Algorithm 63," "Quicksort: Algorithm 64," and "Find: Algorithm 65." *Comm. ACM* 4(7), 321-322, 1961
- Hoare, C. A. R. "Quicksort." (<http://dx.doi.org/10.1093/comjnl/5.1.10>) *Computer Journal* 5 (1): 10-15. (1962). (Reprinted in Hoare and Jones: *Essays in computing science* (<http://portal.acm.org/citation.cfm?id=SERIES11430.63445>), 1989.)
- David Musser. Introspective Sorting and Selection Algorithms, *Software Practice and Experience* vol 27, number 8, pages 983-993, 1997
- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 113–122 of section 5.2.2: Sorting by Exchanging.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 7: Quicksort, pp. 145–164.
- A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting." *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997. pp. 370–379.
- Faron Moller. Analysis of Quicksort (http://www.cs.swan.ac.uk/~csfm/Courses/CS_332/quicksort.pdf). CS 332: Designing Algorithms. Department of Computer Science, Swansea University.
- Conrado Martínez and Salvador Roura, *Optimal sampling strategies in quicksort and quickselect*. *SIAM J. Computing* 31(3):683-705, 2001.
- Jon L. Bentley and M. Douglas McIlroy, Engineering a Sort Function (<http://citeseer.ist.psu.edu/bentley93engineering.html>), *Software—Practice and Experience*, Vol. 23(11), 1249–1265, 1993

External links

- Animated Sorting Algorithms: Quick Sort (<http://www.sorting-algorithms.com/quick-sort>) – graphical demonstration and discussion of quick sort
- Animated Sorting Algorithms: 3-Way Partition Quick Sort (<http://www.sorting-algorithms.com/quick-sort-3-way>) – graphical demonstration and discussion of 3-way partition quick sort
- Interactive Tutorial for Quicksort (<http://pages.stern.nyu.edu/~panos/java/Quicksort/index.html>)
- Quicksort applet (<http://www.yorku.ca/syichen/research/sorting/index.html>) with "level-order" recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.2 - Quicksort (http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION00141200000000000000)
- Multidimensional quicksort in Java (<http://fiehnlab.ucdavis.edu/staff/wohlgemuth/java/quicksort>)
- Literate implementations of Quicksort in various languages (<http://en.literateprograms.org/Category:Quicksort>) on LiteratePrograms
- A colored graphical Java applet (<http://coderaptors.com/?QuickSort>) which allows experimentation with initial state and shows statistics

Boyer–Moore string search algorithm

In computer science, the **Boyer–Moore string search algorithm** is an efficient string searching algorithm that is the standard benchmark for practical string search literature.^[1] It was developed by Robert S. Boyer and J Strother Moore in 1977.^[2] The algorithm preprocesses the string being searched for (the pattern), but not the string being searched in (the text). It is thus well-suited for applications in which the text does not persist across multiple searches. The Boyer-Moore algorithm uses information gathered during the preprocess step to skip sections of the text, resulting in a lower constant factor than many other string algorithms. In general, the algorithm runs faster as the pattern length increases.

Definitions

```

A N P A N M A N -
P A N - - - - -
- P A N - - - -
- - P A N - - -
- - - P A N - -
- - - - P A N -
- - - - - P A N -

```

Alignments of pattern **PAN** to text **ANPANMAN**, from **k=3** to **k=8**. A match occurs at **k=5**.

- $S[i]$ refers to the character at index i of string S , counting from 1.
- $S[i..j]$ refers to the substring of string S starting at index i and ending at j , inclusive.
- A prefix of S is a substring $S[1..i]$ for some i in range $[1, n]$, where n is the length of S .
- A suffix of S is a substring $S[i..n]$ for some i in range $[1, n]$, where n is the length of S .
- The string to be searched for is called the **pattern**.
- The string being searched in is called the **text**.
- The pattern is referred to with symbol P .
- The text is referred to with symbol T .
- The length of P is n .
- The length of T is m .
- An **alignment** of P to T is an index k in T such that the last character of P is aligned with index k of T .
- A **match** or **occurrence** of P occurs at an alignment if P is equivalent to $T[(k-n+1)..k]$.

Description

The Boyer-Moore algorithm searches for occurrences of P in T by performing explicit character comparisons at different alignments. Instead of a brute-force search of all alignments (of which there are $m - n + 1$), Boyer-Moore uses information gained by preprocessing P to skip as many alignments as possible.

The algorithm begins at alignment $k = n$, so the start of P is aligned with the start of T . Characters in P and T are then compared starting at index n in P and k in T , moving downward: the strings are matched from the end and toward the beginning of P . The comparisons continue until either a mismatch occurs or the beginning of P is reached (which means there is a match), after which the alignment is shifted to the right according to the maximum value permitted by a number of rules. The comparisons are performed again at the new alignment, and the process repeats until the alignment is shifted past the end of T .

The shift rules are implemented as constant-time table lookups, using tables generated during the preprocessing of P .

Shift Rules

The Bad Character Rule

Description

```

- - - - X - - K - - -
A N P A N M A N A M -
- N N A A M A N - - -
- - - N N A A M A N -

```

Demonstration of bad character rule with pattern **NNAAMAN**.

The bad-character rule considers the character in T at which the comparison process failed (assuming such a failure occurred). The next occurrence of that character to the left in P is found, and a shift which brings that occurrence in line with the mismatched occurrence in T is proposed. If the mismatched character does not occur to the left in P , a shift is proposed that moves the entirety of P past the point of mismatch.

Preprocessing

Methods vary on the exact form the table for the bad character rule should take, but a simple constant-time lookup solution is as follows: create a 2D table which is indexed first by the index of the character c in the alphabet and second by the index i in the pattern. This lookup will return the occurrence of c in P with the next-highest index $j < i$ or -1 if there is no such occurrence. The proposed shift will then be $i - j$, with $O(1)$ lookup time and $O(kn)$ space, assuming a finite alphabet of length k .

The Good Suffix Rule

Description

```

- - - - X - - K - - - - -
M A N P A N A M A N A P -
A N A M P N A M - - - - -
- - - - A N A M P N A M -

```

Demonstration of good suffix rule with pattern **ANAMPNAM**.

The good suffix rule is markedly more complex in both concept and implementation than the bad character rule. It is the reason comparisons begin at the end of the pattern rather than the start, and is formally stated thus:^[3]

Suppose for a given alignment of P and T , a substring t of T matches a suffix of P , but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and the character to the left of t' in P differs from the character to the left of t in P . Shift P to the right so that substring t' in P is below substring t in T . If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T . If no such shift is possible, then shift P by n places to the right. If an occurrence of P is found, then shift P by the least amount so that a *proper* prefix of the shifted P matches a suffix of the occurrence of P in T . If no such shift is possible, then shift P by n places, that is, shift P past T .

Preprocessing

The good suffix rule requires two tables: one for use in the general case, and another for use when either the general case returns no meaningful result or a match occurs. These tables will be designated L and H respectively. Their definitions are as follows:^[3]

For each i , $L[i]$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L[i]]$ and such that the character preceding that suffix is not equal to $P[i-1]$. $L[i]$ is defined to be zero if there is no position satisfying the condition.

Let $H[i]$ denote the length of the largest suffix of $P[i..n]$ that is also a prefix of P , if one exists. If none exists, let $H[i]$ be zero.

Both of these tables are constructible in $O(n)$ time and use $O(n)$ space. The alignment shift for index i in P is given by $n - L[i]$ or $n - H[i]$. H should only be used if $L[i]$ is zero or a match has been found.

The Galil Rule

A simple but important optimization of Boyer-Moore was put forth by Galil in 1979.^[4] As opposed to shifting, the Galil rule deals with speeding up the actual comparisons done at each alignment by skipping sections that are known to match. Suppose that at an alignment k_1 , P is compared with T down to character c of T . Then if P is shifted to k_2 such that its left end is between c and k_1 , in the next comparison phase a prefix of P must match the substring $T[(k_2 - n)..k_1]$. Thus if the comparisons get down to position k_1 of T , an occurrence of P can be recorded without explicitly comparing past k_1 . In addition to increasing the efficiency of Boyer-Moore, the Galil rule is required for proving linear-time execution in the worst case.

Performance

The Boyer-Moore algorithm as presented in the original paper has worst-case running time of $O(n+m)$ only if the pattern does *not* appear in the text. This was first proved by Knuth, Morris, and Pratt in 1977,^[5] followed by Guibas and Odlyzko in 1980^[6] with an upper bound of $5m$ comparisons in the worst case. Cole gave a proof with an upper bound of $3m$ comparisons in the worst case in 1991.^[7]

When the pattern *does* occur in the text, running time of the original algorithm is $O(nm)$ in the worst case. This is easy to see when both pattern and text consist solely of the same repeated character. However, inclusion of the Galil rule results in linear runtime across all cases.^{[4][7]}

Variants

The Boyer-Moore-Horspool algorithm is a simplification of the Boyer-Moore algorithm using only the bad character rule.

The Apostolico-Giancarlo algorithm speeds up the process of checking whether a match has occurred at the given alignment by skipping explicit character comparisons. This uses information gleaned during the pre-processing of the pattern in conjunction with suffix match lengths recorded at each match attempt. Storing suffix match lengths requires an additional table equal in size to the text being searched.

References

- [1] Hume and Sunday (1991) [*Fast String Searching*] SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(11), 1221–1248 (NOVEMBER 1991)
- [2] Boyer, Robert S.; Moore, J Strother (October 1977). "A Fast String Searching Algorithm." (<http://dl.acm.org/citation.cfm?doid=359842.359859>). *Comm. ACM* (New York, NY, USA: Association for Computing Machinery) **20** (10): 762–772. doi:10.1145/359842.359859. ISSN 0001-0782. .
- [3] Gusfield, Dan (1999) [1997], "Chapter 2 - Exact Matching: Classical Comparison-Based Methods", *Algorithms on Strings, Trees, and Sequences* (1 ed.), Cambridge University Press, pp. 19–21, ISBN 0521585198
- [4] Galil, Z. (September 1979). "On improving the worst case running time of the Boyer-Moore string matching algorithm" (<http://dl.acm.org/citation.cfm?id=359146.359148>). *Comm. ACM* (New York, NY, USA: Association for Computing Machinery) **22** (9): 505–508. doi:10.1145/359146.359148. ISSN 0001-0782. .
- [5] Knuth, Donald; Morris, James H.; Pratt, Vaughan (1977). "Fast pattern matching in strings" (<http://citeseer.ist.psu.edu/context/23820/0>). *SIAM Journal on Computing* **6** (2): 323–350. doi:10.1137/0206024. .
- [6] Guibas, Odlyzko; Odlyzko, Andrew (1977). "A new proof of the linearity of the Boyer-Moore string searching algorithm" (<http://dl.acm.org/citation.cfm?id=1382431.1382552>). *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA: IEEE Computer Society): 189–195. doi:10.1109/SFCS.1977.3. .
- [7] Cole, Richard (September 1991). "Tight bounds on the complexity of the Boyer-Moore string matching algorithm" (<http://dl.acm.org/citation.cfm?id=127830>). *Proceedings of the 2nd annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA: Society for Industrial and Applied Mathematics): 224–233. ISBN 0-89791-376-0. .

External links

- Original paper on the Boyer-Moore algorithm (<http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf>)
- An example of the Boyer-Moore algorithm (<http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html>) from the homepage of J Strother Moore, co-inventor of the algorithm
- Richard Cole's 1991 paper proving runtime linearity (<http://www.cs.nyu.edu/cs/faculty/cole/papers/CHPZ95.ps>)

Article Sources and Contributors

Big O notation *Source:* http://en.wikipedia.org/w/index.php?oldid=523942604 *Contributors:* 132.204.25.xxx, 4v4l0n42, A-Ge0, A. Pichler, ABCD, Abdull, Adashiel, Addps4cat, Aelvin, Ahmad Faridi, Ahoerstemeier, Alan smithee, Alex Selby, Algoman101, Alksentsr, AllanBz, Altenmann, AnOddName, Andre Engels, Andreas Kaufmann, Ankit Maity, Anonymous Dissident, Anthony Appleyard, Apanag, Arjayay, Arno Matthias, Arunmoezhi, Arvindn, Ascánder, AvicAWB, AxelBoldt, B4hand, BMB, Bagsc, Barak Sh, Baronjonas, Ben pcc, BenFrantzDale, Bergstra, Bird of paradox, Bkell, Bomazi, Booyabazooka, Borgx, Brad7777, Breno, Brion VIBBER, Btyner, Bubba73, Buster79, C45207, CBKATopsails, CRGreathouse, Calculuslover, Charlow3, Charles Matthews, CharlesGillingham, ChazBeckett, ChrisForno, Colfulus, Compotoatoj, Connelly, Conversion script, Cookie4869, CosineKitty, Curb Chain, Curps, Cybercobra, CyborgTosser, Czar, D4g0thur, DFS454, Dachshund, Dadudadu, Damian Yerrick, Danakil, Danny, Dark Charles, David Eppstein, Davidwt, Dcljr, Dcoetzee, Deeparnab, Den fjåtræde ankan, Derlay, Dhuss, Diberri, Diego diaz espinosa, Dionyziz, DniQ, Donfbreed, Doradus, Dr. Universe, Draco flavus, Drpaule, Duagloth, Dysprosia, EconoPhysicist, Efnar, El C, Elephant in a tornado, Eleveneleven, Elias, EmilJ, Enochlau, Epachamo, Eric119, Ernie shoemaker, Eus Kevin, FauxFaux, Fayenatic london, Fede Reghe, Felix Wiemann, Fennec, FiachraByrne, Fibonacci, FilipeS, Flouran, Foxjwill, Fredrik, Fvw, Gadig, Gene Ward Smith, GeordieMcBain, Giftlite, Gilliam, Gjd001, Glassmage, Glrx, Gracenotes, Graham87, Gremagor, Gutza, H.ehsaan, Haham hanuka, Hans Adler, Hdante, Head, Headbomb, HenningThieleman, Henrygb, Hermel, Hlg, Ichernev, Intr, InverseHypercube, Isis, Ixf64, JHMM13, JIP, Jacobolus, James.S, Jaredwf, Javit, Jeronimo, Jim1138, Jleede, JoeKearney, JoergenB, JohnWStockwell, Jolsfa123, Jonathanzung, Josephjevan, JoshuaZ, Jowan2005, Jpkotta, Jthillik, Justin W Smith, Jwh335, Kan8eDie, Katsushi, KneeLess, Koendelaere, Koertefa, Koffiyahoo, Kri, LC, LOL, Lambiam, Lamro, Leithp, Leonard G., LeonardoGregianin, Leycec, Linas, Ling.Nut, Lugnad, Luqui, MFH, MIT Trekkie, Macrakis, Mad Jaqk, Maksim-e, Marc van Leeuwen, MarkOlah, MathMartin, Matiasholte, Mattbuck, McKay, Mcstrother, Melcombe, Michael Harding, Michael Rogers, Michael Slone, Miguel, Mike Schwartz, Mindmatrix, Mitchoyoshitaka, Miy, Mobius, Modeha, Mpagano, Mrpsilon, Mstuumel, Mxn, Nbarth, NehpestTheFirst, Neile, Nejko, NeoUrfahrner, Netheril96, Ngorade, Nils Grimsmo, NovaDog, O Pavlos, Oleg Alexandrov, Oliphaunt, Opelio, Optikos, Ot2, PGWG, PL290, Patrick, Patrick Lucas, Paul August, PaulTanenbaum, Paxcoder, Pcuff, Pete4512, PhilKnight, Philip Trueman, Plutor, Poor Yorick, Prumpf, Quendus, Qwfp, R'n'B, R.e.b., R3m0t, Raknarf44, Rebroad, Reinderien, RexNL, Rfl, Riceplaytexas, Rjwilmsi, RobertBorgersen, RobinK, Rockingravn, Roventhot, Rocyte, Rschwieb, Ruud Koot, Sabalka, Sameer0s, Sapphorain, SchifftyThree, Sciurine, ScotsmanRS, Shalom Yehiel, Shizhao, Shoesssss, Shreevatsa, Simetrical, Simon Fenney, Skaraoke, Sligocki, Smjg, Sophus Bie, Spitzak, Stefan.karpinski, Stephen Compall, Stevenj, Stevertigo, Stimpj, Sydbarett74, Syncategoremata, Szepi, TNARasslin, Taemyr, TakuyaMurata, Tardis, Tarotcards, Taw, The Anome, TheBiggestFootballFan, TheSeven, Thenub314, Tide rolls, Timwi, Tony Fox, Tosha, Tritium6, Tyco.skinner, Ultimuss, Universallss, User A1, Vanisheduser12a67, Vecter, Vedant, VictorAnyakin, WchevPart, Whosyourjudas, Whouk, Wikibuki, Writer on wiki, Wtmitchell, Yarin Kaul, ZAB, Zack, Zeitgeist2.718, Zero sharp, ZeroOne, ZiggyMo, Zowch, Zundark, Zvika, Île flottante, 666 anonymous edits

Binary tree *Source:* http://en.wikipedia.org/w/index.php?oldid=524333729 *Contributors:* 15turnsm, 7, ABCD, Aarsalankhalid, Abu adam, Adam majewski, Ahoerstemeier, Ahyl1, Airplaneman, Alex.vatchenko, Alienus, Alléfant, Altenmann, Andreas Kaufmann, AndrewKepert, AnotherPerson 2, Antaeus Feldspar, Aroundthewayboy, Ateeq.s, B4hand, Behranga, Beland, Belovedeagle, Bender2k14, Bhadani, BigDunc, Bipladv IIC, Bkell, BlckKnght, Bluebusy, Bobo192, Bojan1989, Bonadea, Brentsmith101, Brianga, Brucelee, Caltas, Calvin zcx, Card Zero, Cbraga, Cdiggins, Charles Matthews, Chicodroid, Chris the speller, Chris857, Ck lostsword, Classicaelcan, Cncplyr, Coelacan, Conversion script, Cybercobra, Cyhawk, Czar.pino, DMacks, Darangho, David Eppstein, David Shay, David-Sarah Hopwood, Dawynn, Dcoetzee, Dguido, Djcollom, Dkasak, Dominus, Dontdoit, Doriftu, Doug s, Dr. Sunglasses, Drano, DuaneBailey, Duoduoduo, Dysprosia, Ekeeb, Encognito, Ferkelparade, Frankrod44, Frozendice, FvdP, Garyzx, Gdevanla, Giftlite, Gilliam, Gimmetrow, Gsmodi, Happyuk, Hazmat2, Heirpixel, ISTD351, IanS1967, Ilana, Itchy Archibald, JabberWok, Jafet, Jerome Charles Potts, Jerryobject, JimsMaher, John Quincy Adding Machine, Jonfore, Josell2, Josephskeller, Jprg1966, Judstman, Jmjm1964, Mrwojo, Kamath.nakul, Kamirao, KaragouniS, Kbolino, Kgashok, Kgautam28, Kilidiplomus, Kuru, L.C, LandruBek, Liao, Liftarn, LightningDragon, Linas, LithiumBreather, Loisel, LokiClock, Lone boatman, Loolo, Loopwhile1, Lotje, MONGO, Mahahahaneapneap, Malleus Fatuorum, Marc van Leeuwen, Mark Renier, Martinp23, Materialsscientist, MathijsM, Matir, Maurice Carbonaro, Mbdeir, Mboverload, Mcl, Mdnahas, Metricopolus, Mhayes46, Michael Angelkovich, Michael Hardy, Michael Slone, Microbizz, Mike Christie, Minesweeper, Mjm1964, Mrwojo, Neomagic100, Nippoo, Noldoaran, Nonexistent, Oblivious, Oli Filth, Ontariolot, Opelio, Orphic, Otterdam, ParticleMan, Petr, Pkg, Philip Trueman, Pit, Pohl, Pp007, Ppellei, RG2, RadioFan, Rahulgatm, Rege, Reinderien, Rhanekom, Rich Farmbrough, Rohitgoyal100, Rotteduqnc, Roybristow, Rspeer, Rzelnik, Rönin, SGBailey, Sapeur, Shentino, Shinjixman, Shmomuffin, Shounjun, Silver hr, Simeon, Smallpond, SmartGuy Old, Someone else, SpaceFlight89, Spaceflight, Sveysr, Sss41, Stickee, Sun Creator, Taemyr, TakuyaMurata, Tarquin, Tarrahatiks, Tdsmith, The Thing That Should Not Be, Thrappier, Vegpuff, Waggars, Widr, WillNess, Wtarreau, Wælgæst wæfre, XJAmRastafire, Xevior, Xnn, Ynhockey, Yuubinbako, Zero sharp, Zetawoof, Zipdisc, Zvn, 459 anonymous edits

Binary search tree *Source:* http://en.wikipedia.org/w/index.php?oldid=524420225 *Contributors:* 2620:0:1000:1B01:F08A:D18F:B5D4:3D36, 4get, Abednigo, Abu adam, Adamuu, AgentSnorch, Ahyl1, AlanSherlock, Alansohn, Alexsh, Allan McInnes, Andreas Kaufmann, Anoopjohnson, Avermapub, Awu, BAXelrod, BPositive, Banaticus, Beetstra, Bernard François, Bkell, Booyabazooka, Bryan Derksen, Burakov, Butros, Calbaer, Capricorn42, Casted, Chery, Cochito, Conversion script, Cybercobra, D6, Damian Yerrick, Danadocus, Dcoetzee, DevastatorIIC, Dicklyon, Dimchord, Djcollom, Doctordiehard, Doradus, Dysprosia, Dzikaosna, Ecb29, Enochlau, Evil Monkey, Ezhiki, Farazbhinder, Frankrod44, Fredrik, Func, GRHooked, Gaius Cornelius, Garoth, Giftlite, Glenn, Googl, Gorffy, GregorB, Grunt, Hadal, Ham Pastrami, Hathawayc, Havardk, Hu12, Hyad, IgushevEdward, Ilana, Ivan Kuckir, Ixf64, JForget, James pic, Jdm64, Jdurham6, Jerryobject, Jin, Jms49, Jogers, Josell2, Karl-Henner, Kate, Kewlito, Konnetikut, Konstantin Pest, Kragen, Kulp, Kurapix, LOL, Lanov, Liao, LilHelpa, LittleDan, Loren.wilton, Madhan virgo, Matekm, MatrixFrog, Maximamaxim, Maximus Rex, Mb1000, McLareN212, MegaHasher, Metalmax, Mgius, Michael Hardy, Michael Slone, Mikeputnam, Mindmatrix, Minesweeper, MladenWiki, Moe Epsilon, MrOllie, MrSomeone, Mrwojo, Mweber, Nakarumaka, Nerdgerl, Neurodivergent, Nils schmidt hamburg, Nixdorf, Nneoneo, Nomen4Omen, Nux, Ohnoitsjamie, Oleg Alexandrov, Oli Filth, Oliphaunt, One half 3544, Oni Lukos, Onomou, Ontariolot, Oskar Sigvardsson, P0nc, Phil Boswell, PhilipMW, Phishman3579, Pion, Postdlf, Qiq, Qleem, Quuxplusone, Quertyus, RJK1984, Rdebar, Regnaror, Rhanekom, Richardj311, Rolpa, Roysthink, Rudo.Thomas, Ruud Koot, S3000, SPTWitter, Salrizvy, SchumacherTechnologies, Shen, Shmomuffin, Sketch-TheFox, Skier Dude, Smallman12q, Solsan88, Spadgos, Spiff, Sss41, SteveAyre, Swapsy, Tajo, Taw, Tbvdm, The Parting Glass, TheMandarin, Theone256, Thesevenseas, Theta4, Timwi, Tobias Bergemann, Tomt22, TrainUnderwater, Trevor Andersen, VKokielov, Vdm, Vectorpaladin13, Vocaro, Vromascanu, Wavelength, WikHead, WikiWizard, WillNess, Wmayner, Wtarreau, Wtmitchell, X1024, Xevior, Yaderbh, Your Lord and Master, ZeroOne, ماني, 343 anonymous edits

B-tree *Source:* http://en.wikipedia.org/w/index.php?oldid=520740848 *Contributors:* 128.139.197.xxx, ABCD, AaronSw, Abrech, Aednichols, Ahyl1, Alansohn, Alaric, Alfalfahotshots, AlastairMcMillan, Altenmann, Altas, AlyM, Anakin101, Anders Kaseorg, Andreas Kaufmann, Andytwig, AnnaFrance, Antimatter15, Appoose, Aubrey Jaffer, Avono, BAXelrod, Battamer, Beeson, Betzaar, Bezenek, Billinghurst, Bkell, Bladefistx2, Bor4kip, Bovineone, Bryan Derksen, Btwied, CanadianLinuxUser, Carbuncle, Cbraga, Chadwick, Charles Matthews, Chmod007, Chris the speller, Ciphergoth, Ck lostsword, ContivityGoddess, Conversion script, Cp3149, Ctxpcp, Curps, Cutter, Cybercobra, Daev, DanielKlein24, Dcoetzee, Decrease789, Dlae, Dmn, Don4of4, Dpotter, Dravecky, Dysprosia, EEMIV, Ed g2s, Eddvella, Edward, Ethan, Fabriciodansjossilla, FatalError, Fgdadsfgdsagfd, Flying Bishop, Fragglet, Fredrik, FreplySpang, Fresheneesz, FvdP, Gdr, Giftlite, Glrx, GoodPeriodGal, Ham Pastrami, H2o1ian, Hariva, Hbent, Headbomb, I do not exist, Inquisitus, Iohannes Animosus, JCLately, JWSchmidt, Jacosi, Jeff Wheeler, Jirka6, Jdawson7, Joahannes, Joe07734, John ch fr, John lindgren, John of Reading, Jorge Stolfi, JoshuSasori, Jy00912345, Kate, Ketil, Kinema, Kinu, Knutux, Kovianyo, Kpias, Kukolar, Lamdk, Lee J Haywood, Leibniz, Levin, Lfstevens, Loadmaster, Luna Santin, MIT Trekkie, MachineRebel, Makkuro, Malbrain, Matttoothman, Merit 07, Mfwtiten, Mhss, Michael Angelkovich, Michael Hardy, Mikeblas, Minesweeper, Mnogo, MoAJgnome, MorgothX, Mrmaz, Mrwojo, NGPriest, Nayuki, Neile, Nishantjr, Noodlez84, Norm mit, Oldsharp, Oli Filth, P199, PKT, Patmorin, Paushali, Peter bertok, Pgan002, Phishman3579, Postrach, Priyank bolia, PrologFan, Psyphen, Ptheoch, Pyschobbens, Quadrescence, Qutezuce, R. S. Shaw, RMcPhillip, Redrose64, Rich Farmbrough, Rp, Rpajares, Ruud Koot, Sandeep.a.v, Sandman@Ilpg.org, SickTwist, Simon04, SirSeal, Slady, Slike, Spiff, Ssbohio, Stephan Leclercq, Stevemidgle, Strake, Ta bu shi da yu, Talldcan, Teles, The Fifth Horseman, Tjdw, Tobias Bergemann, TomYHChan, Trusilver, Tuolunne0, Twimoki, Uday, Uw.Antony, Verbal, Viroglyph, Wantnot, WinampLlama, Wipe, Wkailey, WolfKeeper, Wout.mertens, Wsloand, Wtanaka, Wtmitchell, Yakushima, Zearin, 408 anonymous edits

AVL tree *Source:* http://en.wikipedia.org/w/index.php?oldid=518121453 *Contributors:* 2001:B30:1002:C03:959C:93A8:21B0:8826, Adamd1008, Adamuu, Aent, Agrawalogesh, Akerbos, Alex Kapranoff, AlexGreat, Altenmann, Anant sogani, Andreas Kaufmann, Andrew Weintraub, Apanag, Astral, Autotf6, Avicennasis, Axe-Lander, BenBac, Benoit fraikin, Binnacle, Bkil, Blackllotus, Bluebusy, Byrial, Casterovx, Caviere, ChrisMP1, Codingrecipes, Compusense, Conversion script, CostinulAT, Cybercobra, Cyhawk, Daewoolama, Damian Yerrick, Darangho, David.Mestel, Dawynn, Dcamp314, Dcoetzee, Denisarona, Dicklyon, DmitriyVilkov, Docboat, Doradus, Drilnoth, Dtrebbien, Dysprosia, Eleuther, Enviroboy, Epachamo, Euchiasmus, Evil Monkey, Flyingspuds, Fredrik, FvdP, G0gogesc300, Gaius Cornelius, Geek84, Geoff55, Gnowor, Greenrd, Greg Tyler, Gruu, Gulliveig, Gurch, Gökhan, II MusLim HyBrId II, Iamnitin, Ichimonji10, Infrogmation, Intr, InverseHypercube, J.delanoy, Jac16888, Jeepey, Jeff02, Jennavecia, Jim1138, Jirka6, Jll, Joeyadams, Josell2, KGasso, Kain2396, Kdau, Kenyon, Kingpin13, Kjolb, Ksulli10, Kukolar, Kuru, Kushalbiswas777, LOL, Lankiveil, Larry V, Leszek Jaficzuk, Leuko, M, MarkHeily, Materialsscientist, MattyIX, Mauritsmaartendejong, Mckaysalisbury, Mellerho, Merovingian, Michael Hardy, Michael M Clarke, Michael miceli, Mike Rosoff, Mikm, Minesweeper, Mjkoo, MladenWiki, Mnogo, Moberg, Mohammad ahad, Momet, Mr.Berna, Msanchez1978, Mtanti, Mzruya, NanlinWiki, Neile, Nguyễn Hữu Đức, Nixdorf, Nnemo, Noldoaran, Northamerica1000, Nysin, Obradovic Goran, Oleg Alexandrov, Oliversisson, Ommiy-Pangaeus, Orimosenzon, Paul D. Anderson, Pavel Vozenilek, Pedrito, Pgan002, Phishman3579, Pnorcks, Poor Yorick, RJFJR, Ravithurana, Resper, Rockslave, Ruud Koot, ST47, Safety Cap, Sebutecure, Seyen, Shlomif, Shmomuffin, Smalljim, Srivesh, Ste4k, Tamfang, Tide rolls, Tobias Bergemann, Toby Douglass, Tphyahoo, Tsemii, Tsoft, UnwashedMeme, Uw.Antony, Vektor330, Vlad.c.manea, West.andrew.g, Xevior, Yksykyks, Zian, 387 anonymous edits

Red-black tree *Source:* http://en.wikipedia.org/w/index.php?oldid=522042542 *Contributors:* 203.37.81.xxx, 206.176.2.xxx, 2607:F140:400:1028:D41E:C074:460:983, Abu adam, Adamuu, Ahoerstemeier, Ahyl1, Akerbos, AlanU, Altenmann, Andreas Kaufmann, Aplusbi, Awakenrz, Banej, Belfry, Belovedeagle, BenFrantzDale, Bezenek, Binnacle, Bioskope, Blackllotus, Blow, Bonadea, Bovineone, Brona, C. A. Russell, Cababunga, Card Zero, Caviare, Cburnett, Cjcollier, Connelly, Consed, Conversion script, Cybercobra, Cyp, David Eppstein, Dcoetzee, Deepakabhyankar, Dfeuer, Dkhalion, Drak, Dreamyshade, Drebs, Dysprosia, Dlugosz, Enochlau, Eprb123, ErikHaugen, Fawcett5, Fcp2007, Fragglet, Fredrik, FvdP, Ghakko, Ghewiglit, Giftlite, Gimbold13, Giraffedata, Glrx, Gnathan87, Graham87, Grandphuba, H2g2bob, H2o2lian, Hariva, Hawke666, Headbomb, Hermel, Hgkamath, Hnn79, Hugh Aguilair, Humpback, Idogbert, Iav, JMBucknall, Jamesfisher, Jaxl, Jengelh, Jerome.Abel, JingguoYao, Jleede, Jodawi, Johnuniq, Joriki, Joshhibschman, Jtsiomb, Jzcool, Karakak, Karl-Henner, Karlhendrikse, Kenyon, Khalil Sawant, Kmels, Kragen, Kukolar, Kyle Hardgrave, Laurier12, Leonard G., LiDaobing, Linuxrocks123, Loading, Lukax, Lunakeet, Madhurtanwani, MatrixFrog, Maxis ftw, Mgrand, Michael

Ertzeid, Fresheneesz, Gatoatigrado, Giftlite, Hairy Dude, Hariva, Herry12, Hike395, Igoldste, IgushevEdward, Inquam, Isnow, Jamelan, Jaredwf, Jirka6, Joe Schmedley, Justin W Smith, K3rb, Karlhendrikse, Kingsindian, Kiril Simeonovskii, Krivokon.dmitry, Krp, LC, LOL, Lambiam, Lim Wei Quan, MER-C, Magioladitis, Matusz, Maurobio, McIntosh Natura, Michael Hardy, Mihirpmehta, Mindbleach, Mitchellduffy, Miym, Moink, N4nojohn, Obradovic Goran, Opium, Pit, Platypus222, Poor Yorick, Prasanth.moothedath, Purpy Purple, R6144, Rami R, Rbrewer42, Rmhughes, Romanm, Ruud Koot, SchuminWeb, Seano1, Sethleb, Shen, SiobhanHansa, Squizz, SuperSack56, Swfung8, Tameralkuly, Teimu.tm, Treyshonuff, Turketwh, UrsusArctos, Whosyourjudas, Wikimol, Wikizoli, Wtmitchell, Yc16, Ysangkok, ZeroOne, 188 anonymous edits

Kruskal's algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=522696412> *Contributors:* 4v4l0n42, Abc45624, Abrech, Abu adam, Adamsandberg, Ahyl1, Alcides, Aliok ao, Altenmann, Andre Engels, Ap, Arthena, AySz88, BioTronic, Bkell, Boothy443, CarpeCerevisi, Chipchap, Chmod007, CommonsDelinker, Cronholm144, David Eppstein, Dcoetzee, Deineka, Denisarona, Dixtosa, DoriSmith, Dysprosia, Ech29, Eda eng, Giftlite, GregorB, HJ Mitchell, Hammadhaleem, Happyuk, Headbomb, Hhbs, Hike395, IgushevEdward, Isnow, Jamelan, Jaredwf, Jax Omen, Jeri Aulurtve, Jgarrett, JohnBlackburne, Jonsafari, Kenyon, Kruskal, Kvikram, LC, Levin, Lone boatman, M412k, MRFraga, Maciek.nowakowski, Marekpetrik, MarkSweep, MathMartin, Mbarrenecheajr, Mgreenbe, Michael Angelkovich, Michael Blone, Michael.jaeger, Mikae, MindAfterMath, MiqayelMinasyan, Miserlou, Mitchellduffy, Mmtux, MrOllie, Msh210, Mysid, NawlinWiki, Oli Filth, Omargamil, Oskar Sigvardsson, Panarchy, Pinethicket, Poor Yorick, Pranay Varma, Qwertyus, R3m0t, Sarcelles, Shellreef, Shen, Shreyasjoshis, Simeon, SiobhanHansa, Spangineer, THEN WHO WAS PHONE?, The Anome, Tokataro, Treyshonuff, Trunks175, Wakimakirolls, Wapcaplet, Wavelength, YahoKa, Yc16, Zero0000, 168 anonymous edits

Bellman–Ford algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=524345420> *Contributors:* Aaron Rotenberg, Abednigo, Aednichols, Aene, Agthor, AlexCovarrubias, Altenmann, Anubas, Andris, Arlekean, B3virq3b, Backslash Forwardslash, BenFrantzDale, Bjozen, Bkell, BlankVerse, Brona, CBM, Charles Matthews, CiaPan, CIPHERgott, David Eppstein, Dcoetzee, Docu, Drdevil44, Ech29, Enochlaur, Epr123, Ferengi, FrankTobia, Fredrik, Fvw, Gadfium, Giftlite, GregorB, Guahuala, Happyuk, Headbomb, Heineman, Helix84, Iceblock, Istanton, Itai, J.delanoy, Jamelan, Jaredwf, Jellyworld, Jftuga, Josteinaj, Justin W Smith, Konstable, LOL, Lavv17, Lone boatman, Mario777Zelda, Mathmike, Mazin07, Mclid, Michael Hardy, Miym, Monday, N Shar, Naroz, Nihiltes, Nils Grimsdal, Nullzero, Orbst, P b1999, PanLevan, Pion, Pjrm, Poor Yorick, Posix4e, Psjks, Quuxplusone, Razvilmsi, RobinK, Rseper, Ruud Koot, SQL, Salix alba, Sam Hoocevar, Shuroo, Sigkill, Solon.KR, SpyMagician, Stdazi, Stderr.dk, Stern, Str82no1, Tomo, ToneDaBass, Tsunanet, Tvidas, Waldir, Wmahan, Writer130, Zholdas, 165 anonymous edits

Depth-first search *Source:* <http://en.wikipedia.org/w/index.php?oldid=524448906> *Contributors:* 2001:6B0:1:1041:223:6CFF:FE7F:5E70, A5b, ABCD, Andre Engels, Antiusar, Apcsoola, Apanag, Batkins, Bbi5291, Beetstra, BenRG, Braddjwinter, Bryan Derksen, Bubba73, CesarB, Citrus538, Clacker, Craig Barkhouse, Curtmack, DHN, David Eppstein, Davidyeotb, Dcoetzee, Dreske, Duncan Hope, Dysprosia, EditTor, Edlee, ErikvanB, Fkodama, Fraggleg81, Frecklefoot, Frizzil, G8moon, Giftlite, Gschoyru, Gurch, Gustavb, Hansamurai, Hemanshu, Hermel, Itai, Ixf64, Jaredwf, Jef41341, Jeltz, Jerf, Jinlin, Jonon, Justin Mauger, Justin W Smith, Kate, Kdau, Kesla, Koertefta, Kromped, LittleDan, MarkSweep, Marked, Martynas Patasius, MaterialsScientist, Mctpyt, Mentifisto, Michael Hardy, Minknight, Mild Bill Hiccup, Miles, Mipadi, Mjrm, Moosebumps, Nuroonstolz, Nuno Tavares, Nvrnmnd, Pmussur, Poor Yorick, Pukeye, Purpy Purple, Qcrank, Qwertyus, Qx2020, Regnaron, Rich Farmbrough, Rick Norwood, Rror, RxS, SPTWriter, Shuroo, Smallman12q, Snowwolf, Srrrgei, Staszek Lem, Stefan, Steveraport, Stimp, Subh83, Svick, Tauwasser, Taxipom, Thegeneralguy, Thesilverbail, Thumperward, TimBentley, Vroo, Vsh3r, Waldir, Wavelength, Yonestar, Yuide, Z10x, 214 anonymous edits

Biconnected graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=522172910> *Contributors:* David Eppstein, Giftlite, Kmote, Maxal, Mboverload, McKay, Mikmorg, Open2universe, Tayste, 3 anonymous edits

Huffman coding *Source:* <http://en.wikipedia.org/w/index.php?oldid=522385524> *Contributors:* 129.128.164.xxx, AaronSw, Ablewisuk, Ahoerstemeier, AlanUS, Alejo2083, Alex Vinokur, AllyUnion, Andre Engels, Andreas Kaufmann, AnkitChachada, AnnaFrance, Antaeus Feldspar, Apantomimehorse, Arslanteginghazi, B4hand, Batjohan, Betacommand, Bnearns, Bobkart, Bobo192, Boffy b, Brianski, CBM, Cal 1234, Calbaer, CanisRufus, Caroliano, Charles Matthews, Chojin79, Classicaecon, Conversion script, CountingPine, Cyp, Damian Yerrick, DanielCD, David Peacham, Dbenbenn, Dcirovic, Dcoetzee, Digwuren, Dmcq, DmitriyV, Don4of4, Drilnoth, Dspradaw, ERcheck, Eppeliteloop, Electron9, Elwikipedista, Ep118, Excalabyte, Fabiogramos, FalseAlarm, Ferengi, First Harmonic, Foolip, Fredrik, Furrykef, GENtLe, Gaius Cornelius, Giftlite, Gnomz007, Graham87, GregorB, Grunt, Gwickwire, Hadal, Haham hanuka, Haseo9999, Hayk, Helixblue, Hermel, Hezink8, HoCkEy PUCK, Hodja Nasreddin, Hupfis, Ihope127, Ike9898, Inkling, Iridescent, Iseeaboar, Itmozart, JaGa, Japo, Jaredwf, Jarek Duda, JIinton, Joanna040290, Joshi1983, Jsaintro, Jshadias, Kapil.xerox, Kistaro Windrider, Kijjava, KnightRider, Kojiroh, Kungfuadam, Kurykh, Kvng, Lee J Haywood, Liberatus, M412k, MiTalum, Maghnus, Majorly, Mentifisto, Meteficha, Michael Hardy, Michael miceli, Mindvirus, Mjib, Mofochickam, MoreNet, MrOllie, Mntanti, Mulligatawny, N5lin, Nanobear, Navy.nathusiast, Nick1nldram, Nishantjr, Noldoaran, Ogigiri, Oli Filth, Omegatron, Ooz dot ie, Oravec, Otus, OverlordQ, OwenBlacker, PatheticCopyEditor, Paul August, Paul Niquette, Pebkac, Phantomsteve, Piano non troppo, Picasso angelo, PickUpAxis, Pitel, Pizzadeliveryboy, Red Baron, Requestion, Rjstott, RobertG, Robth, Russavia, Ruud Koot, S2000magician, SamuelRiv, Seb, Sebastiangarth, Shadowjams, Shanes, Smallman12q, Smithers888, Spoon!, Stephen B Streater, Susvolans, TakuyaMurata, Tandrasz, Thavron, The Anome, TheEgyptian, Thue, Thumperward, Tide rolls, Timwi, Tiny green, Tinyraysite, Tobias Bergemann, Tom harrison, Trifon Triantafyllidis, VTBassMatt, Vecrumba, Vegasprof, Verdy p, Vina, Vog, WODUP, Wavelength, Wikizoli, Willy411432, Yuvrajd, ZeroOne, Zigger, Ziusudra, Zundark, 338, זיכרונות anonymous edits

Floyd–Warshall algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=524328862> *Contributors:* 16@r, 2001:41B8:83F:1004:0:0:0:27A6, Aenima23, AlanUS, AlexandreZ, Altenmann, Anharrington, Barfooz, Beardsbloke, Buaagg, C. Siebert, Cachedio, Closedmouth, Quimper, Daveagp, David Eppstein, Dcoetzee, Dfrankow, Dittymathew, Donhalcon, DrAndrewColes, Fata11955, Fintler, Frankrodd4, Gaius Cornelius, Giftlite, Greenleaf, Greenmatter, GregorB, Gutworth, Harrigan, Hjlfreyer, Integr, J. Finkelstein, JLAtondre, Jaredwf, Jellyworld, Jerryobject, Joy, Julian Mendez, Justin W Smith, Kanitani, Kenyon, Kesla, KitMarlow, Kletos, Leycec, LiDaobing, Luv2run, Maco1421, Magioladitis, MarkSweep, Md.aftabuddin, Michael Hardy, Minghong, Minority Report, Mqchen, Mwk soul, Nanobear, Netvor, Nicapicella, Nishantjr, Nneonneo, Obradovic Goran, Oliver, Opium, Pandemias, Phil Boswell, Pilotguy, Pjrm, Polymerbringer, Poor Yorick, Pxtreme75, Quuxplusone, Rababerski, Raknarf44, RobinK, Roman Munich, Ropez, Roseperrone, Ruud Koot, Sakanarm, SchreiberBike, Shadowjams, Shyamal, Simoneau, Smurfix, Soumya92, Specs112, Sr3d, SteveJothan, Strainu, Svick, Taejo, Teles, Treyshonuff, Volkan YAZICI, W3bbo, Wickethewok, Xyzyz n, 220 anonymous edits

Sorting algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=523840082> *Contributors:* -OOPSIE-, 124Nick, 132.204.27.xxx, A3 nm, A5b, AManWithNoPlan, Aaron Rotenberg, Abhishekupadhyaya, Accelerometer, Adair2324, AdamProcter, Advance512, Aeons, Aeonx, Aeriform, Agteller, Agor, Aguyude, Ahoerstemeier, Ahshabazz, Ahyl1, Alain Amiouini, Alansohn, AlexPlank, Alksub, AllyUnion, Altenmann, Alvestrand, Amirmalekzadeh, Anadverb, Andre Engels, Andy M. Wang, Ang3lboy2001, Angela, Arpi0292, Artonio, Arvind, Astronouth7303, AxelBoldt, BACbKA, Bachrach44, Balabiot, Baltar, Gaius, Barton2, Bbi5291, Beland, Ben Standeven, BenFrantzDale, BenKovitz, Bender2k14, Bento00, Bidabadi, Bkell, Bobo192, Boleyn, Booyabazooka, Bradyyoung01, Brendan179, Bryan Derksen, BrightShadow, Bubba73, BurtAlert, C. A. Russell, C7protal, CJLL Wright, Caesura, Calculuslover, Calixte, CambridgeBayWeather, Carey Evans, Ccn, Charles Matthews, Chenopodiaceous, Chinju, Chris the speller, Ciaccona, Circular17, ClockworkSoul, Codeman38, Cole Kitchen, Compfreak7, Conversion script, Cpl Syx, Crashmatrix, Crumpuppet, Cubero031, Cwolsfsheep, Cyan, Cybercobra, Cymbalta, Cyrius, DHN, DIY, DaVinci, Daiyuda, Damian Yerrick, Danakil, Daniel Quinlan, DarkFalls, Darkwind, DarrylNester, Darin Panda, David Eppstein, Dcirovic, Dcoetzee, Deanonwikk, Debacorkl, Decrypt3, Deepakjoy, Deskana, DevastatorILC, Dgse87, Diannaa, Dihard, Domingos, Doradus, Duck1123, Duvavici1, Dybdahl, Dysprosia, EdC, Eddideigel, Efansoftware, Eliz81, Energy Dome, Etokopcedute, Fagstena, Falcon8765, Fastily, Fawcett5, Firsfron, Foobarnix, Foot, Fragglet, Fred Bauder, Fredrik, Frencheigh, Fresheneesz, Fuzzy, GanKeyu, GateKeeper, Gavia immer, Gdr, Giftlite, Glrx, Grafen, Graham87, Graue, GregorB, H3nry, HJ Mitchell, Hadal, Hagerman, Hairhorn, Hamaad.s, Hannes Hirtzel, Hashar, Hede2000, Hgranqvist, Hirtzel, Hobart, HolyCookie, Hpa, IMale, Indefual, InverseHypercube, Iridescent, Itsameen-bc103112, J.delanoy, JBakaka, JLAtondre, JRSpriggs, JTN, Jacht, Jaguaraki, Jamesday, Japo, Jay Litman, Jbonneau, Jeffq, Jeffrey Mall, Jeronimo, Jesin, Jirka6, Jj137, Jll, Jmw02824, Jokes Free4Me, JonGinny, Jonadab, Jonas Köller, Josh Kehn, Joshk, Jthemphill, Justin W Smith, Jwoodger, Kalraritz, Kevinsystrom, Kievite, Kingjames iv, KlappCK, Knutux, Kragen, KyuubiSeal, LC, Ldoron, Lee J Haywood, LiHilpa, Lowercase Sigma, Luna Santin, Lzap, Makeemlighter, Malcolm Farmer, Mandarax, Mark Renier, MarkisLandis, MartinHarper, Marvon7Newby, MarvonNewby, Mas.morozov, MaterialsScientist, MattGiuca, Matthew0028, Mav, Maximus Rex, Mbernard707, Mdd4696, Mdr, Medich1985, Methecooldude, Michael Greiner, Michael Hardy, Michaelbluejay, Mike Rosoff, Mindmatrix, Mountain, Mr Elmo, Mrck@charter.net, Mrjeff, Musiphil, Myanw, NTF, Nanshu, Nayuki, Nevakee11, NewEnglandYankee, NickT988, Nicolaum, Nikai, Nish0009, Nixdorf, Nknight, Nomen4Omen, OfekRon, Olathe, Olivier, Omegatron, Ondra.pelech, OoS, Oskar Sigvardsson, Oguz Ergin, Pablo.cl, Pajz, Pamulapati, Panarchy, Panu-Kristian Poiksalo, PatPeter, Patrick, Paul Murray, Phassan, Pcap, Pcc3@ij.net, Pelister, Perstar, Pete142, Petri Krohn, Pfalstad, Philomathoholic, PierreBoudes, Piet Delpoit, Populus, Pparent, PsyberS, Pyfan, Quaeler, RHaworth, RJFJR, RapidR, Rasinj, Raul654, RaulMetumtam, RazoriCE, Rcbarnes, Reyk, Riana, Roadrunner, Robert L, Robin S. RobinK, Rodspade, Roman V. Odaisky, Rsathish, Rursus, Ruud Koot, Ryguasu, Scalene, Schnozzinkobenstein, Shadowjams, Shanes, Shredwheat, SimonP, SiobhanHansa, Sir Nicholas de Mimsy-Porpington, Slashme, Sligocki, Smartech, Snickel11, Sophus Bie, Soultaco, SouthernNights, Spoon!, Ssd, StanfordProgrammer, Staplesauce, Starwiz, Stephen Howe, Stephenb, StewieK, Suanshsinghal, Summentier, Sven nestle2, Swamp Ig, Swift, T4bits, TakuyaMurata, Tamfang, Taw, Tawker, Teles, Templatetypedef, The Anome, The Thing That Should Not Be, TheKMan, TheRingess, Thefourlinestar, Thinking of England, ThomasTomMueller, Thumperward, Timwi, Tobias Bergemann, Tortoise74, TowerDragon, Travuun, Traxter, Twikir, Tyler McHenry, UTSRelativity, Udiock, Ulfben, UpstateNYer, User A1, VTBassMatt, Valenciano, Veganfanatic, VeryVerily, Veryangrypenguin, Verycuriousboy, Vrenator, Wantnot, Wazimuko, Wei.cs, Wfunction, Wiki.ryansmith, WillNess, Wimt, Worch, Writenonsand, Ww, Yansa, Yuval madar, Zawersh, Zipcodeman, Ztothefifth, Zundark, 880 anonymous edits

Quicksort *Source:* <http://en.wikipedia.org/w/index.php?oldid=523487763> *Contributors:* 0, 1exec1, 4v4l0n42, A Meteorite, Aaditya 7, Aaronbrick, Abednigo, Abu adam, Adicarlo, AdmN, Aerion, Ahyl1, Alain Amiouini, AlanBarrett, Alcauchy, Alexander256, Allan McInnes, Altenmann, Andrei Stroe, Arcturus4669, Aroben, Arto B, Arvindn, AxelBoldt, BarretB, Bartosz, Bastetswarrior, Bayard, Beetstra, Benbread, Bengski68, Berlinguyinica, Bertrc, Biker Biker, Bkell, Black Falcon, Blaisorblade, Bluear, Bluemoose, Booyabazooka, Boulevardier, Bputman, BrokenSegue, Btwied, Bubba73, C. Iorenz, C7protal, Calwikk, Carl7j, CarlosHoyos, Cdiggins, Centrx, CesarB, Chameleon, Chinju, Chrischan, Chrisdene, Chrisk02, CiaPan, CmcFarland, CobaltBlue, ColdShine, Composingleigh, Compotatoj, Comrade009, Connelly, Croc hunter, Crowst, Cumthsc, Cybercobra, Cyde, Czarkoff, Daekharel, Dan100, DanBishop, Danakil, Dancraggs, Daniel5Ko, Darrel francis, Darren Strash, David Bunde, Dbfirs, Dcoetzee, Decrypt3, Diego UFCG, Dila, Dinomite, Diomidis Spinellis, Discospinster, Dissident, Dmccarty, Dmwpowers, Docolloinni, DomQ, Domokato, Donhalcon, Doradus, Dr.RMills, Dysprosia, ESkog, Equor, Elciel0917, Eleassar, Elharo, Empaisley, Entropy, Eric119, Etrigg, Evil saline, Faridani, Fastilysock, Fennec, Feradz, Ferkelparade, Francis Tyers, Frankrod44, Frantisek.jandos, Fredrik, Fresheneesz, Fried-peach, Func, Furru, Fx2, Fxer, Gdo01, Giftlite, Glane23, Glrx, Gpollock, Graham87, GregorB, Gschoyru, HJ Mitchell, Haker4o, HalHal, Hamza1886, Hannes Hirtzel, Hdante, Helios2k6, Hfastedge, Indeed123, Intangir, Integre8e, Ipeiritis, J Adrian, Jamesday, Jao, Jason

Davies, Jazmatician, Jeff3000, Jevy1234, Jfmantis, Jnoring, John Comeau, John lindgren, John of Reading, Jokes Free4Me, Josh Kehn, Jpbowen, Juliand, Jusjih, JustAHappyCamper, Jóna Þórunn, Kanie, Kapildalwani, Kbk, Kbolino, Kingturtle, KittyKAY4, Knutux, Kragen, Kuszi, LOL, Larosek, Lars Trebing, Lhuang3, Liftarn, LilHelpa, LittleDan, LodeRunner, Lord Emsworth, Lordmetroid, MC10, MH, Magister Mathematicae, Mark91, Mathiastck, McKay, Mecej4, Meekohi, Mh26, Michael Shields, Mihai Damian, Mike Rosoft, Mikeblas, Modster, MrOllie, Mswen, Murray Langton, N Vale, NTF, Narendrak, NawlinWiki, Nearffxx, Necklace, Neile, Neohaven, NevilleDNZ, NickW557, NickyMcLean, Nik 0 0, Ninjatummen, Nixdorf, Nmnogueira, Nneonneo, Noisylo65, Norm mit, Notheruser, Nothing1212, NovellGuy, Nwerneck, Obscuranym, Ocolon, Ohnoitsjamie, Oli Filth, Orderud, Oskar Sigvardsson, OverlordQ, Oğuz Ergin, PGSONIC, Pakaran, Pako, Panzi, Patmorin, PatrickFisher, Perpetuo2, Pete142, Pgan002, Phe, Philip Trueman, Pifactorial, Pmcjones, Populus, Postrach, Pushingbits, Quantumelixir, Quuxplusone, R3m0t, Rami R, Randywombat, Raul654, Rdargent, RedWolf, Rhanekom, Richss, Roger Hui, RolandH, Romanm, Rony fhebian, Rrufai, Rsanchezaez, Rspeer, Rursus, Ruud Koot, SKATEMANKING, SPTWriter, Sabb0ur, Sam Hocevar, SamuelRiv, Sandare, Scebert, Scott Paeth, Seaphoto, Senfo, Sf222, Shanes, Shashmik11, Shellreef, Simetrical, SiobhanHansa, Sir Edward V, Sir Nicholas de Mimsy-Porpington, Sjakkalle, Skapur, Sligocki, Snowolf, Solde9, Soliloquial, Spearhead, Spiff, Stdazi, StephenDow, Stevietheman, Sverdrup, Svick, Swfung8, Swift, Sychen, TakuyaMurata, Tanadeau, Tavianator, The Anome, TheCois, Theclawen, Themanian, ThomasTomMueller, Tide rolls, Tim32, Timneu22, Timwi, Tobias Bergemann, Tompagenet, Too Old, Udirock, Ungzd, UrsaFoot, User A1, Versus22, Vikreykja, Weedwhacker128, Wfaxon, Wik, Wikid77, Wisgary, Worch, Ww, Xezbeth, Yandman, Ylloh, Yulin11, Zarrandreas, Zeno Gantner, ZeroOne, Znuipi, Михајло Анђелковић, 784 anonymous edits

Boyer–Moore string search algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=521328685> *Contributors:* Abednigo, Adfellin, Alex.mccarthy, Ancheta Wis, Andrew Helwer, Antaeus Feldspar, Art1x com, Aunndroid, BD2412, Balabiot, Beland, Biker Biker, Billava, Billyoneal, Bisquit, Blueyoshi321, Booyabazooka, Borgx, ChrisGualtieri, Cneubauer, Cwalgampaya, Czaner, Damian Yerrick, DaveWF, Dcoetzee, Dekart, DocWatson42, Dpakoha, Duplicity, Edsarian, Ellassint, Evgeny Lykhin, Eyal0, Fbriere, Fib, Freaky Dug, Fredrik, Furrykef, Infinito, J12f, Jashmenn, Jemfinch, Jinghaoxu, JoeMarfice, JustAHappyCamper, Jy2wong, Kayvee, Klutzy, Kostmo, Kucyla, Lauren Lester, Lisamh, Lumpynifkin, M.O.X, Mathias126, Maximus Rex, Mboverload, Mi1ror, Mikeblas, Moink, Mr flea, Murray Langton, Neelpulse, Nickjay, Nneonneo, Ott2, PedR, Phe, PhilKnight, Plindenbaum, Pne, Quuxplusone, RJFJR, Radagast83, Rich Farmbrough, Ruud Koot, Ryan Reich, SeekerOfThePath, Smallman12q, Snowgene, SummerWithMorons, Szabolcs Nagy, Thegeneralguy, Tide rolls, Tim Starling, Tim.head, Tobias Bergemann, Triddle, TripleF, Watcher, Wikibob, Wthrower, Ww, Xillion, YUL89YYZ, Zearin, 201 anonymous edits

Image Sources, Licenses and Contributors

File:Big-O notation.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Big-O-notation.png> *License:* GNU General Public License *Contributors:* Fede Reghe, Razorbliss

Image:binary tree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_tree.svg *License:* Public Domain *Contributors:* User:Dcoetzee

File:BinaryTreeRotations.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:BinaryTreeRotations.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Josell7

File:Insertion of binary tree node.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Insertion_of_binary_tree_node.svg *License:* Creative Commons Zero *Contributors:* Hazmat2

File:Deletion of internal binary tree node.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Deletion_of_internal_binary_tree_node.svg *License:* Creative Commons Zero *Contributors:* Hazmat2

Image:Binary tree in array.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_tree_in_array.svg *License:* Public Domain *Contributors:* User:Dcoetzee

Image:N-ary to binary.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:N-ary_to_binary.svg *License:* Public Domain *Contributors:* CyHawk

File:Binary search tree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_search_tree.svg *License:* Public Domain *Contributors:* User:Booyabazooka, User:Dcoetzee

File:binary search tree delete.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_search_tree_delete.svg *License:* Public Domain *Contributors:* User:Dcoetzee

File:B-tree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:B-tree.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* CyHawk

Image:B tree insertion example.png *Source:* http://en.wikipedia.org/w/index.php?title=File:B_tree_insertion_example.png *License:* Public Domain *Contributors:* User:Maxtremus

File: AVL Tree Rebalancing.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:AVL_Tree_Rebalancing.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* CyHawk

Image:binary search tree delete.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_search_tree_delete.svg *License:* Public Domain *Contributors:* User:Dcoetzee

Image:Red-black tree example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_example.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* en:User:Cburnett

Image:Red-black tree example (B-tree analogy).svg *Source:* [http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_example_\(B-tree_analogy\).svg](http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_example_(B-tree_analogy).svg) *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* fr:Utilisateur:Verdy_p

Image:Red-black tree insert case 3.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_insert_case_3.png *License:* Public Domain *Contributors:* Users Cintrom, Deco, Deelkar on en.wikipedia

Image:Red-black tree insert case 4.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_insert_case_4.png *License:* Public Domain *Contributors:* User Deco on en.wikipedia

Image:Red-black tree insert case 5.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_insert_case_5.png *License:* Public Domain *Contributors:* User Deco on en.wikipedia

Image:Red-black tree delete case 2.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_delete_case_2.png *License:* Public Domain *Contributors:* Users Deelkar, Deco on en.wikipedia

Image:Red-black tree delete case 3.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_delete_case_3.png *License:* Public Domain *Contributors:* User Deco on en.wikipedia

Image:Red-black tree delete case 4.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_delete_case_4.png *License:* Public Domain *Contributors:* User Deco on en.wikipedia

Image:Red-black tree delete case 5.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_delete_case_5.png *License:* Public Domain *Contributors:* User Deco on en.wikipedia

Image:Red-black tree delete case 6.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_delete_case_6.png *License:* Public Domain *Contributors:* User Deco on en.wikipedia

Image:Hash table 4 1 1 0 0 1 0 LL.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_4_1_1_0_0_1_0_LL.svg *License:* Public Domain *Contributors:* Jorge Stolfi

Image:Hash table 4 1 1 0 0 0 0 LL.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_4_1_1_0_0_0_0_LL.svg *License:* Public Domain *Contributors:* Jorge Stolfi

Image:Hash table 4 1 0 0 0 0 0 LL.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_4_1_0_0_0_0_0_LL.svg *License:* Public Domain *Contributors:* Jorge Stolfi

Image:Max-Heap.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Max-Heap.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Ermishin

File:Max-heap.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Max-heap.png> *License:* Public Domain *Contributors:* Created by Onar Vikingstad 2005.

File:Min-heap.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Min-heap.png> *License:* Public Domain *Contributors:* Original uploader was Vikingstad at en.wikipedia

File:Heap add step1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_add_step1.svg *License:* Public Domain *Contributors:* Ilmari Karonen

File:Heap add step2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_add_step2.svg *License:* Public Domain *Contributors:* Ilmari Karonen

File:Heap add step3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_add_step3.svg *License:* Public Domain *Contributors:* Ilmari Karonen

File:Heap remove step1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_remove_step1.svg *License:* Public Domain *Contributors:* Ilmari Karonen

File:Heap remove step2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_remove_step2.svg *License:* Public Domain *Contributors:* Ilmari Karonen

File:Binary tree in array.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_tree_in_array.svg *License:* Public Domain *Contributors:* User:Dcoetzee

File:Binary Heap with Array Implementation.JPG *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_Heap_with_Array_Implementation.JPG *License:* Creative Commons Zero *Contributors:* Bobmath

image:Leftist-trees-S-value.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Leftist-trees-S-value.svg> *License:* Public Domain *Contributors:* Computergeeksjw (talk)

Image:Min-height-biased-leftist-tree-initialization-part1.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Min-height-biased-leftist-tree-initialization-part1.png> *License:* Public Domain *Contributors:* Buss, Qef

Image:Min-height-biased-leftist-tree-initialization-part2.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Min-height-biased-leftist-tree-initialization-part2.png> *License:* Public Domain *Contributors:* Buss, Qef

Image:Min-height-biased-leftist-tree-initialization-part3.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Min-height-biased-leftist-tree-initialization-part3.png> *License:* Public Domain *Contributors:* Buss, Qef

Image:Directed acyclic graph.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Directed_acyclic_graph.png *License:* Public Domain *Contributors:* Anarkman, Dcoetzee, Ddxc, EugeneZelenko, Joey-das-WBF

Image:Breadth-first-tree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Breadth-first-tree.svg> *License:* Creative Commons Attribution 3.0 *Contributors:* Alexander Drichel

Image:Animated BFS.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Animated_BFS.gif *License:* GNU Free Documentation License *Contributors:* Blake Matheny. Original uploader was Bmatheny at en.wikipedia

Image:MapGermanyGraph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:MapGermanyGraph.svg> *License:* Public Domain *Contributors:* AndreasPraefcke, Mapmarks, MistWiz, Regnaron

Image:GermanyBFS.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:GermanyBFS.svg> *License:* Public Domain *Contributors:* Regnaron

Image:Dijkstra Animation.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Dijkstra_Animation.gif *License:* Public Domain *Contributors:* Ibmuua

Image:Dijkstras progress animation.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Dijkstras_progress_animation.gif *License:* Creative Commons Attribution 3.0 *Contributors:* Subh83

File:MAZE 30x20 Prim.ogv *Source:* http://en.wikipedia.org/w/index.php?title=File:MAZE_30x20_Prim.ogv *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Purpy Puppel

Image:Prim Algorithm 0.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_0.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel

Image:Prim Algorithm 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_1.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_2.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_3.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_4.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_5.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 6.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_6.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel, Stefan Birkner

Image:Prim Algorithm 7.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Prim_Algorithm_7.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Alexander Drichel, Stefan Birkner

File:Prim-algorithm-animation-2.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Prim-algorithm-animation-2.gif> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* fungszevai

Image:Kruskal Algorithm 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_1.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_2.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_3.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_4.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_5.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Kruskal Algorithm 6.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kruskal_Algorithm_6.svg *License:* Public Domain *Contributors:* Maksim, Yuval Madar

Image:Depth-first-tree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Depth-first-tree.svg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Alexander Drichel

Image:graph.traversal.example.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Graph.traversal.example.svg> *License:* GNU Free Documentation License *Contributors:* Miles

Image:Tree edges.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_edges.svg *License:* Public Domain *Contributors:* User:Stimpy

Image:If-then-else-control-flow-graph.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:If-then-else-control-flow-graph.svg> *License:* Public Domain *Contributors:* BenRG

File:MAZE 30x20 DFS.ogv *Source:* http://en.wikipedia.org/w/index.php?title=File:MAZE_30x20_DFS.ogv *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Purpy Purple

Image:Huffman tree 2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Huffman_tree_2.svg *License:* Public Domain *Contributors:* Meteficha

Image:Huffman coding example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Huffman_coding_example.svg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* Alessio Damato

Image:Huffman huff demo.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Huffman_huff_demo.gif *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Eeppeliloop

File:Bubblesort-edited-color.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Bubblesort-edited-color.svg> *License:* Creative Commons Zero *Contributors:* User:Pmdumuid

File:Shellsort-edited.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Shellsort-edited.png> *License:* Public domain *Contributors:* by crashmatrix (talk)

File:Sorting quicksort anim.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Sorting_quicksort_anim.gif *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* Wikipedia:en>User:RolandH

Image:Quicksort-diagram.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Quicksort-diagram.svg> *License:* Public Domain *Contributors:* Znupi

File:Quicksort-example.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Quicksort-example.gif> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Matt Chan

Image:Partition example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Partition_example.svg *License:* Public Domain *Contributors:* User:Dcoetzee

License

Creative Commons Attribution-Share Alike 3.0 Unported
[//creativecommons.org/licenses/by-sa/3.0/](https://creativecommons.org/licenses/by-sa/3.0/)
