

Data structures

Contents

Articles

Introduction	1
Data structure	1
Linked data structure	3
Succinct data structure	5
Implicit data structure	5
Compressed data structure	6
Search data structure	7
Static and dynamic data structures	9
Persistent data structure	9
Concurrent data structure	10
Abstract data types	13
Abstract data type	13
List	20
Stack	23
Queue	34
Deque	42
Priority queue	44
Map	48
Bidirectional map	53
Multimap	53
Set	54
Tree	57
Arrays	60
Array data structure	60
Row-major order	66
Dope vector	68
Iliffe vector	68
Dynamic array	69
Hashed array tree	72
Gap buffer	73
Circular buffer	74

Sparse array	83
Bit field	84
Bit array	87
Bitboard	92
Parallel array	97
Lookup table	98
Lists	105
Linked list	105
XOR linked list	121
Unrolled linked list	123
VList	125
Skip list	127
Jump list	132
Self-organizing list	133
Binary trees	134
Binary tree	134
Binary search tree	140
Self-balancing binary search tree	149
Tree rotation	151
Weight-balanced tree	154
Threaded binary tree	155
AVL tree	157
Red-black tree	161
AA tree	175
Scapegoat tree	182
Splay tree	186
T-tree	192
Rope	195
Top Trees	197
Tango Trees	201
van Emde Boas tree	201
Cartesian tree	205
Treap	210
B-trees	213
B-tree	213

B+ tree	223
B*-tree	226
B sharp tree	227
Dancing tree	227
2-3 tree	228
2-3-4 tree	229
Queaps	231
Fusion tree	235
Bx-tree	235
Heaps	239
Heap	239
Binary heap	241
Binomial heap	245
Fibonacci heap	251
2-3 heap	255
Pairing heap	255
Beap	258
Leftist tree	259
Skew heap	262
Soft heap	266
Ternary heap	268
D-ary heap	270
Tries	271
Trie	271
Radix tree	275
Suffix tree	278
Suffix array	282
Compressed suffix array	286
FM-index	287
Generalised suffix tree	287
B-trie	289
Judy array	289
Directed acyclic word graph	290
Multiway trees	292
Ternary search tree	292

And-or tree	293
(a,b)-tree	295
Link/cut tree	296
SPQR tree	296
Spaghetti stack	299
Disjoint-set data structure	300
Space-partitioning trees	304
Space partitioning	304
Binary space partitioning	305
Segment tree	310
Interval tree	313
Range tree	318
Bin	319
kd-tree	321
Implicit kd-tree	328
min/max kd-tree	331
Adaptive k-d tree	332
Quadtree	332
Octree	335
Linear octrees	337
Z-order	337
UB-tree	341
R-tree	341
R+ tree	344
R* tree	345
Hilbert R-tree	346
X-tree	353
Metric tree	354
VP-tree	354
BK-tree	355
Hashes	356
Hash table	356
Hash function	368
Open addressing	377
Lazy deletion	380
Linear probing	381

Quadratic probing	382
Double hashing	383
Cuckoo hashing	384
Coalesced hashing	386
Perfect hash function	389
Universal hashing	390
Linear hashing	392
Extendible hashing	394
2-choice hashing	408
Pearson hashing	408
Fowler–Noll–Vo hash function	409
Bitstate hashing	411
Bloom filter	411
Locality preserving hashing	422
Morton number	422
Zobrist hashing	423
Rolling hash	424
Hash list	425
Hash tree	426
Prefix hash tree	429
Hash trie	429
Hash array mapped trie	430
Distributed hash table	430
Consistent hashing	435
Stable hashing	436
Koordinate	437
Graphs	440
Graph	440
Adjacency list	442
Adjacency matrix	444
And-inverter graph	446
Binary decision diagram	448
Binary moment diagram	452
Zero suppressed decision diagram	454
Propositional directed acyclic graph	455
Graph-structured stack	456
Scene graph	457

Appendix	462
Big O notation	462
Amortized analysis	472
Locality of reference	473
Standard Template Library	477

References

Article Sources and Contributors	484
Image Sources, Licenses and Contributors	492

Article Licenses

License	496
---------	-----

Introduction

Data structure

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

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

Data structures are used in almost every program or software system. Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and internet indexing services. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Basic principles

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

Abstract data structures

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations.

This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

Language support

Assembly languages and some low-level languages, such as BCPL, generally lack support for data structures. Many high-level programming languages, on the other hand, have special syntax or other built-in support for certain data structures, such as vectors (one-dimensional arrays) in the C programming language, multi-dimensional arrays in Pascal, linked lists in Common Lisp, and hash tables in Perl and in Python. Many languages also provide basic facilities such as references and the definition record data types, that programmers can use to build arbitrarily complex structures.

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

Modern languages also generally support modular programming, the separation between the interface of a library module and its implementation. Some provide opaque data types that allow clients to hide implementation details.

Object-oriented programming languages, such as C++, .NET Framework and Java, use classes for this purpose.

With the advent of multi-core processors, many known data structures have concurrent versions that allow multiple computing threads to access the data structure simultaneously.

Classification of data structure

Primitive / Non-primitive: Primitive data structures are basic data structure and are directly operated upon machine instructions, e.g., integer, character. Non-primitive data structures are derived from primitive data structures, e.g., structure, union, array.

Homogeneous / Heterogeneous: In homogeneous data structures all elements are of the same type, e.g., array. In heterogeneous data structures elements are of different types, e.g. structure.

Static / Dynamic: In static data structures memory is allocated at the time of compilation. In dynamic data structures memory is allocated at run-time, through functions such as malloc, calloc, etc.

Linear / Non-linear: Linear data structures maintain a linear relationship between their elements, e.g., array. Non-linear data structures do not maintain any linear relationship between their elements, e.g., in a tree.

See also

- Concurrent data structure
- Linked data structure
- List of data structures
- Comparison of data structures
- Data model
- Data modeling
- Dynamization
- Persistent data structure

References

- [1] 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://www.itl.nist.gov/div897/sqg/dads/HTML/datastructur.html>) Accessed 2009-05-21.
 - [2] Entry *data structure* in the Encyclopædia Britannica (2009) Online entry (<http://www.britannica.com/EBchecked/topic/152190/data-structure>) accessed on 2009-05-21.
- Niklaus Wirth, *Algorithms and Data Structures*
 - Donald Knuth, *The Art of Computer Programming*, vol. 1. Addison-Wesley.
 - Dinesh Metha and Sartaj Sahni *Handbook of Data Structures and Applications*, Chapman and Hall/CRC Press, 2007.

External links

- Descriptions (<http://nist.gov/dads/>) from the Dictionary of Algorithms and Data Structures
- <http://www.cse.unr.edu/~bebis/CS308/>
- Data structures course with animations (http://www.cs.auckland.ac.nz/software/AlgAnim/ds_ToC.html)
- Data structure tutorials with animations (<http://courses.cs.vt.edu/~csonline/DataStructures/Lessons/index.html>)
- An Examination of Data Structures ([http://msdn.microsoft.com/en-us/library/aa289148\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa289148(VS.71).aspx))
- CollectionSpy (<http://www.collectionspy.com>) — A profiler and viewer for Java's standard data structure classes (see Java Collections Framework)

Linked data structure

In computer science, a **linked data structure** is a data structure which consists of a set of data records (*nodes*) linked together and organized by references (*links* or *pointers*).

In linked data structures, the links are usually treated as special data types that can only be dereferenced or compared for equality. Linked data structures are thus contrasted with arrays and other data structures that require performing arithmetic operations on pointers. This distinction holds even when the nodes are actually implemented as elements of a single array, and the references are actually array indices: as long as no arithmetic is done on those indices, the data structure is essentially a linked one.

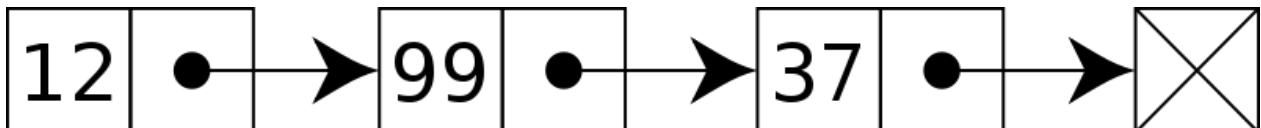
Linked data structures include linked lists, search trees, expression trees, and many other widely used data structures. They are also key building blocks for many efficient algorithms, such as topological sort^[1] and set union-find^[2].

Common Types of Linked Data Structures

Linked Lists

Basic Properties

- Objects, called nodes, are linked in a linear sequence
- A reference to the first node of the list is always kept. This is called the 'head' or 'front'.^[3]



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

Example in Java

This is an example of the node class used to store integers in a Java implementation of a linked list.

```
public class IntNode {
    public int value;
    public IntNode link;
    public IntNode(int v) { value = v; }
}
```

Search Trees

Basic Properties

- Objects, called nodes, are stored in an ordered set
- In-order traversal provides an ascending readout of the data in the tree
- Sub trees of the tree are in themselves, trees.

Advantages and disadvantages

Against Arrays

Compared to arrays, linked data structures allow more flexibility in organizing the data and in allocating space for it. With arrays, we must choose a size for our array once and for all, this can be a potential waste of memory. A linked data structure is built dynamically and never needs to be bigger than the programmer requires. It also requires no guessing in terms of how much space you must allocate when using a linked data structure. This is a feature that is key in saving wasted memory.

The nodes of a linked data structure can also be moved individually to different locations without affecting the logical connections between them, unlike arrays. With due care, a process can add or delete nodes to one part of a data structure even while other processes are working on other parts.

On the other hand, access to any particular node in a linked data structure requires following a chain of references that stored in it. If the structure has n nodes, and each node contains at most b links, there will be some nodes that cannot be reached in less than $\log_b n$ steps. For many structures, some nodes may require worst case up to $n-1$ steps. In contrast, many array data structures allow access to any element with a constant number of operations, independent of the number of entries.

General Disadvantages

Linked data structures also may also incur in substantial memory allocation overhead (if nodes are allocated individually) and frustrate memory paging and processor caching algorithms (since they generally have poor locality of reference). In some cases, linked data structures may also use more memory (for the link fields) than competing array structures. This is because linked data structures are not contiguous. Instances of data can be found all over in memory, unlike arrays.

In some theoretical models of computation that enforce the constraints of linked structures, such as the pointer machine, many problems require more steps than in the unconstrained random access machine model.

See also

- List of data structures

References

- [1] Donald Knuth, The Art of Computer Programming
- [2] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, Volume 7, Issue 5 (May 1964), pages 301-303. The paper originating disjoint-set forests. ACM Digital Library (<http://portal.acm.org/citation.cfm?doid=364099.364331>)
- [3] <http://www.cs.toronto.edu/~hojjat/148s07/lectures/week5/07linked.pdf>

Succinct data structure

In computer science, a **succinct data structure** for a given data type is a representation of the underlying combinatorial object that uses an amount of space "close" to the information theoretic lower bound, together with efficient algorithms for navigation, search, insertion and deletion operations. The concept was introduced by Jacobson [1], to encode bit vectors, (unlabeled) trees, and planar graphs.

A natural example is the representation of a binary tree: an arbitrary binary tree on n nodes can be represented in $2n + o(n)$ bits while supporting a variety of operations on any node, which includes finding its parent, its left and right child, and returning the size of its subtree, each in constant time. The number of different binary trees on n nodes is $\binom{2^n}{n} / (n+1)$. 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.

A related notion is that of a compressed data structure, in which the size of the data structure depends upon the particular data being represented. When the data are compressible, as is often the case in practice for natural language text, the compressed data structure can occupy substantially less space than the information-theoretic lower bound.

References

[1] Jacobson, Guy, *Space-Efficient Static Trees and Graphs*, doi:10.1109/SFCS.1989.63533, ISBN 0-8186-1982-1

Implicit data structure

In computer science, an **implicit data structure** is a data structure that uses very little memory besides the actual data elements. It is called "implicit" because most of the structure of the elements is expressed implicitly by their order. Another term used interchangeably is **space efficient**. Definitions of "very little" is vague and can mean from $O(1)$ to $O(\log n)$ extra space. Implicit data structures are frequently also succinct data structures.

Although one may argue that disk space is no longer a problem and we should not concern ourselves with improving space utilization, the issue that implicit data structures are designed to improve is main memory utilization. Hard disks, or any other means of large data capacity, I/O devices, are orders of magnitudes slower than main memory. Hence, the higher percentage of a task can fit in buffers in main memory the less dependence is on slow I/O devices. Hence, if a larger chunk of an implicit data structure fits in main memory the operations performed on it can be faster even if the asymptotic running time is not as good as its space-oblivious counterpart.

Examples

Examples of implicit data structures include

- Heap
- Beap

Further readings

- See publications of Hervé Brönnimann [1], J. Ian Munro [2], Greg Frederickson [3]

References

[1] <http://photon.poly.edu/~hbr/>

[2] <http://www.cs.uwaterloo.ca/~imunro/>

[3] <http://www.cs.purdue.edu/people/gnf>

Compressed data structure

The term **compressed data structure** arises in the computer science subfields of algorithms, data structures, and theoretical computer science. It refers to a data structure whose operations are roughly as fast as those of a conventional data structure for the problem, but whose size can be substantially smaller. The size of the compressed data structure is typically highly dependent upon the entropy of the data being represented.

Important examples of compressed data structures include the compressed suffix array^[1] ^[2] and the FM-index,^[3] both of which can represent an arbitrary text of characters T for pattern matching. Given any input pattern P , they support the operation of finding if and where P appears in T . The search time is proportional to the sum of the length of pattern P , a very slow-growing function of the length of the text T , and the number of reported matches. The space they occupy is roughly equal to the size of the text T in entropy-compressed form, such as that obtained by Prediction by Partial Matching or gzip. Moreover, both data structures are self-indexing, in that they can reconstruct the text T in a random access manner, and thus the underlying text T can be discarded. In other words, they simultaneously provide a compressed and quickly searchable representation of the text T . They represent a substantial space improvement over the conventional suffix tree and suffix array, which occupy many times more space than the size of T . They also support searching for arbitrary patterns, as opposed to the inverted index, which can support only word-based searches. In addition, inverted indexes do not have the self-indexing feature.

An important related notion is that of a succinct data structure, which uses space roughly equal to the information-theoretic minimum, which is a worst-case notion of the space needed to represent the data. In contrast, the size of a compressed data structure depends upon the particular data being represented. When the data are compressible, as is often the case in practice for natural language text, the compressed data structure can occupy substantially less space than the information-theoretic minimum.

References

- [1] R. Grossi and J. S. Vitter, Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching], *Proceedings of the 32nd ACM Symposium on Theory of Computing*, May 2000, 397-406. Journal version in *SIAM Journal on Computing*, 35(2), 2005, 378-407.
- [2] R. Grossi, A. Gupta, and J. S. Vitter, High-Order Entropy-Compressed Text Indexes, *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms*, January 2003, 841-850.
- [3] P. Ferragina and G. Manzini, Opportunistic Data Structures with Applications, *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, November 2000, 390-398. Journal version in Indexing Compressed Text, *Journal of the ACM*, 52(4), 2005, 552-581.

Search data structure

In computer science, a **search data structure** is any data structure that allows the efficient retrieval of specific items from a set of items, such as a specific record from a database.

The simplest, most general, and least efficient search structure is merely an unordered sequential list of all the items. Locating the desired item in such a list, by the linear search method, inevitably requires a number of operations proportional to the number n of items, in the worst case as well as in the average case. Useful search data structures allow faster retrieval; however, they are limited to queries of some specific kind. Moreover, since the cost of building such structures is at least proportional to n , they only pay off if several queries are to be performed on the same database (or on a database that changes little between queries).

Static search structures are designed for answering many queries on a fixed database; **dynamic** structures also allow insertion, deletion, or modification of items between successive queries. In the dynamic case, one must also consider the cost of fixing the search structure to account for the changes in the database.

Classification

The simplest kind of query is to locate a record that has a specific field (the *key*) equal to a specified value v . Other common kinds of query are "find the item with smallest (or largest) key value", "find the item with largest key value not exceeding v ", "find all items with key values between specified bounds v_{\min} and v_{\max} ".

In certain databases the key values may be points in some multi-dimensional space. For example, the key may be a geographic position (latitude and longitude) on the Earth. In that case, common kinds of queries are *find the record with a key closest to a given point v*, or *"find all items whose key lies at a given distance from v"*, or *"find all items within a specified region R of the space"*.

A common special case of the latter are simultaneous range queries on two or more simple keys, such as "find all employee records with salary between 50,000 and 100,000 and hired between 1995 and 2007".

Single ordered keys

- Array if the key values span a moderately compact interval.
- Priority-sorted list; see linear search
- Key-sorted array; see binary search
- Self-balancing binary search tree
- Hash table

Finding the smallest element

- Heap

Asymptotic worst-case analysis

In this table, the asymptotic notation $O(f(n))$ means "not exceeding some fixed multiple of $f(n)$ " in the worst case".

	Insert	Delete	Search	Find maximum	Space usage
Unsorted array	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Value-indexed array	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
Unsorted linked list	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Self-balancing binary tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Heap	$O(\log n)$	$O(\log n)^{\dagger}$	$O(n)$	$O(1)$	$O(n)$
Hash table	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

† The deletion cost is $O(\log n)$ for the minimum or maximum, $O(n)$ for an arbitrary element.

This table is only an approximate summary; for each data structure there are special situations and variants that may lead to different costs. Also two or more data structures can be combined to obtain lower costs.

See also

- List of data structures

Static and dynamic data structures

In computer science, a **static data structure** is a data structure created for an input data set which is not supposed to change within the scope of the problem. When a single element is to be added or deleted, the update of a static data structure incurs significant costs, often comparable with the construction of the data structure from scratch. In real applications, **dynamic data structures** are used, which allow for efficient updates when data elements are inserted or deleted.

There exist methods, called dynamization, to convert static data structures into dynamic ones.

Static data means the data that occurs one time during its life time. Once it is created it cannot be deleted or modified. Static data is the root of dynamic data and linked with dynamic data as index of them.

Persistent data structure

In computing, a **persistent data structure** is a data structure which always preserves the previous version of itself when it is modified; such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. A persistent data structure is *not* a data structure committed to persistent storage, such as a disk; this is a different and unrelated sense of the word "persistent."

A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. The data structure is **fully persistent** if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called **confluently persistent**. Structures that are not persistent are called **ephemeral**.

These types of data structures are particularly common in logical and functional programming, and in a purely functional program all data is immutable, so all data structures are automatically fully persistent. Persistent data structures can also be created using in-place updating of data and these may, in general, use less time or storage space than their purely functional counterparts.

While persistence can be achieved by simple copying, this is inefficient in time and space, because most operations make only small changes to a data structure. A better method is to exploit the similarity between the new and old versions to share structure between them, such as using the same subtree in a number of tree structures. However, because it rapidly becomes infeasible to determine how many previous versions share which parts of the structure, and because it is often desirable to discard old versions, this necessitates an environment with garbage collection.

Perhaps the simplest persistent data structure is the singly-linked list or *cons*-based list, a simple list of objects formed by each carrying a reference to the next in the list. This is persistent because we can take a *tail* of the list, meaning the last k items for some k , and add new nodes on to the front of it. The tail will not be duplicated, instead becoming shared between both the old list and the new list. So long as the contents of the tail are immutable, this sharing will be invisible to the program.

Many common reference-based data structures, such as red-black trees, and queues, can easily be adapted to create a persistent version.

See also

- Persistence (computer science)
- Navigational Database

References

- Making Data Structures Persistent ^[1]
 - Persistent Data Structures (survey) ^[2]
 - Fully persistent arrays for efficient incremental updates and voluminous reads ^[3]
 - Real-Time Deques, Multihead Turing Machines, and Purely Functional Programming ^[4]
-

References

- [1] <http://www.cs.cmu.edu/~sleator/papers/Persistence.htm>
- [2] <http://citeseer.ist.psu.edu/674112.html>
- [3] <http://citeseer.ist.psu.edu/301077.html>
- [4] <http://citeseer.ist.psu.edu/117501.html>

Concurrent data structure

In computer science, a **concurrent data structure** is a particular way of storing and organizing data for access by multiple computing threads (or processes) on a computer.

Historically, such data structures were used on uniprocessor machines with operating systems that supported multiple computing threads (or processes). The term concurrency captured the multiplexing/interleaving of the threads' operations on the data by the operating system, even though the processors never issued two operations that accessed the data simultaneously.

Today, as multiprocessor computer architectures that provide parallelism become the dominant computing platform (through the proliferation of multi-core processors), the term has come to stand mainly for data structures that can be accessed by multiple threads which may actually access the data simultaneously because they run on different processors that communicate with one another. The concurrent data structure (sometimes also called a shared data structure) is usually considered to reside in an abstract storage environment called shared memory, though this memory may be physically implemented as either a "tightly coupled" or a distributed collection of storage modules.

Basic principles

Concurrent data structures, intended for use in parallel or distributed computing environments, differ from "sequential" data structures, intended for use on a uniprocessor machine, in several ways ^[1]. Most notably, in a sequential environment one specifies the data structure's properties and checks that they are implemented correctly, by providing **safety properties**. In a concurrent environment, the specification must also describe **liveness properties** which an implementation must provide. Safety properties usually state that something bad never happens, while liveness properties state that something good keeps happening. These properties can be expressed, for example, using Linear Temporal Logic.

The type of liveness requirements tend to define the data structure. The method calls can be **blocking** or **non-blocking** (see non-blocking synchronization). Data structures are not restricted to one type or the other, and can allow combinations where some method calls are blocking and others are non-blocking (examples can be found in the Java concurrency software library).

The safety properties of concurrent data structures must capture their behavior given the many possible interleavings of methods called by different threads. It is quite intuitive to specify how abstract data structures behave in a sequential setting in which there are no interleavings. Therefore, many mainstream approaches for arguing the safety properties of a concurrent data structure (such as serializability, linearizability, sequential consistency, and quiescent consistency^[1]) specify the structures properties sequentially, and map its concurrent executions to a collection of sequential ones.

In order to guarantee the safety and liveness properties, concurrent data structures must typically (though not always) allow threads to reach consensus as to the results of their simultaneous data access and modification requests. To support such agreement, concurrent data structures are implemented using special primitive synchronization operations (see synchronization primitives) available on modern multiprocessor machines that allow multiple threads to reach consensus. This consensus can be reached achieved in a blocking manner by using locks, or without locks, in which case it is non-blocking. There is a wide body of theory on the design of concurrent data structures (see bibliographical references).

Design and Implementation

Concurrent data structures are significantly more difficult to design and to verify as being correct than their sequential counterparts.

The primary source of this additional difficulty is concurrency, exacerbated by the fact that threads must be thought of as being completely asynchronous: they are subject to operating system preemption, page faults, interrupts, and so on.

On today's machines, the layout of processors and memory, the layout of data in memory, the communication load on the various elements of the multiprocessor architecture all influence performance. Furthermore, there is a tension between correctness and performance: algorithmic enhancements that seek to improve performance often make it more difficult to design and verify a correct data structure implementation.

A key measure for performance is scalability, captured by the speedup of the implementation. Speedup is a measure of how effectively the application is utilizing the machine it is running on. On a machine with P processors, the **speedup** is the ratio of the structures execution time on a single processor to its execution time on T processors. Ideally, we want linear speedup: we would like to achieve a speedup of P when using P processors. Data structures whose speedup grows with P are called **scalable**. The extent to which one can scale the performance of a concurrent data structure is captured by a formula known as Amdahl's law and more refined versions of it such as Gustafson's law.

A key issue with the performance of concurrent data structures is the level of **memory contention**: the overhead in traffic to and from memory as a result of multiple threads concurrently attempting to access the same locations in memory. This issue is most acute with blocking implementations in which locks control access to memory. In order to acquire a lock, a thread must repeatedly attempt to modify that location. On a cache-coherent multiprocessor (one in which processors have local caches that are updated by hardware in order to keep them consistent with the latest values stored) this results in long waiting times for each attempt to modify the location, and is exacerbated by the additional memory traffic associated with unsuccessful attempts to acquire the lock.

References

- [1] Mark Moir and Nir Shavit (2007). "Concurrent Data Structures". in Dinesh Metha and Sartaj Sahni. *'Handbook of Data Structures and Applications'* (1st ed.). Chapman and Hall/CRC Press. pp. 47-14 — 47-30.
- Nancy Lynch "Distributed Computing"
- Hagit Attiya and Jennifer Welch "Distributed Computing: Fundamentals, Simulations And Advanced Topics, 2nd Ed"
- Doug Lea, "Concurrent Programming in Java: Design Principles and Patterns"
- Maurice Herlihy and Nir Shavit, "The Art of Multiprocessor Programming"
- Mattson, Sanders, and Massingil "Patterns for Parallel Programming"

External links

- JSR 166: Concurrency Utilities (<http://www.jcp.org/en/jsr/detail?id=166>) intro page (<http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>)

Abstract data types

Abstract data type

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

For example, an abstract stack data structure could be defined by two operations: push, that inserts some data item into the structure, and pop, that extracts an item from it; with the constraint that each pop always returns the most recently pushed item that has not been popped yet. When analyzing the efficiency of algorithms that use stacks, one may also specify that both operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

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

The notion of abstract data types is related to the concept of data abstraction, important in object-oriented programming and design by contract methodologies for software development .

Defining an abstract data type

There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Imperative abstract data type definitions

In the "imperative" view, which is closer to the philosophy of imperative programming languages, an abstract data structure is conceived as an entity that is *mutable* — meaning that it may be in different *states* at different times. Some operations may change the state of the ADT; therefore, the order in which operations are evaluated is important, and the same operation on the same entities may have different effects if executed at different times — just like the instructions of a computer, or the commands and procedures of an imperative language. To underscore this view, it is customary to say that the operations are *executed* or *applied*, rather than *evaluated*. The imperative style is often used when describing abstract algorithms.^[2]

Abstract variable

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

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

with the constraint that

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

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

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

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

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

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

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

Instance creation

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

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

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

Preconditions, postconditions, and invariants

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

Example: abstract stack (imperative)

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

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

with the constraint that

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

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

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

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

$$\{ U \leftarrow y; V \leftarrow z; W \leftarrow x \}$$

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

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

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

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

Single-instance style

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

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

Functional ADT definitions

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

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

Example: abstract stack (functional)

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

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

with the following axioms:

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

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

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

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

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

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

As in some other branches of mathematics, it is customary to assume also that the stack states are only those whose existence can be proved from the axioms in a finite number of steps. In the stack ADT example above, this rule means that every stack is a *finite* sequence of values, that becomes the empty stack (Λ) after a finite number of pops. By themselves, the axioms above do not exclude the existence of infinite stacks (that can be popped forever, each time yielding a different state) or circular stacks (that return to the same state after a finite number of pops). In particular, they do not exclude states s such that $\text{pop}(s) = s$ or $\text{push}(s,x) = s$ for some x . However, since one cannot obtain such stack states with the given operations, they are assumed "not to exist".

Typical operations

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

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

In imperative-style ADT definitions, one often finds also

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

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

Examples

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

- Container
- Deque
- List
- Map
- Multimap
- Multiset
- Priority queue
- Queue
- Set
- Stack
- String
- Tree

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

Implementation

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

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

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

When implementing an ADT, each instance (in imperative-style definitions) or each state (in functional-style definitions) is usually represented by a handle of some sort.^[3]

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

Example: implementation of the stack ADT

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

Imperative-style interface

An imperative-style interface might be:

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

stack_T stack_create(void);                  /* Create new stack
instance, initially empty. */
void stack_push(stack_T s, stack_Item e);   /* Add an item at the top of
the stack. */
stack_Item stack_pop(stack_T s);            /* Remove the top item from
the stack and return it. */
int stack_empty(stack_T ts);               /* Check whether stack is
empty. */
```

This implementation could be used in the following manner:

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

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

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

Functional-style interface

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

```
typedef struct stack_Rep stack_Rep;           /* Type: stack state
representation (an opaque record). */
typedef stack_Rep *stack_T;                  /* Type: handle to a stack
state (an opaque pointer). */
typedef void *stack_Item;                   /* Type: item (arbitrary
address). */

stack_T stack_empty(void);                    /* Returns the empty stack
state. */
stack_T stack_push(stack_T s, stack_Item x); /* Adds x at the top of s,
returns the resulting state. */
stack_Item stack_top(stack_T s);            /* Returns the item
currently at the top of s. */
stack_T stack_pop(stack_T s);              /* Remove the top item
from s, returns the resulting state. */
```

ADT libraries

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

Built-in abstract data types

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

See also

- initial algebra
- Concept (generic programming)
- Design by contract
- Formal methods
- Functional specification
- Liskov substitution principle
- Object-oriented programming
- Type system
- Type theory
- Algebraic data type
- Generalized algebraic data type

External links

- Abstract data type [4] in NIST Dictionary of Algorithms and Data Structures

References

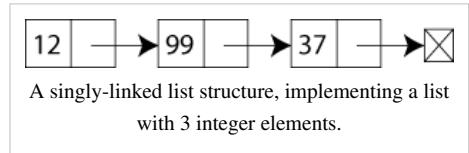
- [1] Barbara Liskov, Programming with Abstract Data Types, in Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, pp. 50--59, 1974, Santa Monica, California
- [2] Donald Knuth, The Art of Computer Programming
- [3] Robert Sedgewick (1998). *Algorithms in C*. Addison/Wesley. ISBN 0-201-31452-5., definition 4.4.
- [4] <http://www.nist.gov/dads/HTML/abstractDataType.html>

List

In computer science, a **list** or **sequence** is an abstract data structure that implements an ordered collection of values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a finite sequence, that is, a tuple. Each instance of a value in the list is usually called an **item**, **entry**, or **element** of the list; if the same value occurs multiple times, each occurrence is considered a distinct item.

The name **list** is also used for several concrete data structures that can be used to implement abstract lists, especially linked lists.

The so-called **static** list structures allow only inspection and enumeration of the values. A **mutable** or **dynamic** list may allow items to be inserted, replaced, or deleted during the list's existence.



Many programming languages provide support for **list data types**, and have special syntax and semantics for lists and list operations. Often a list can be constructed by writing the items in sequence, separated by commas, semicolons, or spaces, within a pair of delimiters such as parentheses '()', brackets, '[]', braces '{}', or angle brackets '<>'. Some languages may allow list types to be indexed or sliced like array types. In object-oriented programming languages, lists are usually provided as instances of subclasses of a generic "list" class. List data types are often implemented using arrays or linked lists of some sort, but other data structures may be more appropriate for some applications. In some contexts, such as in Lisp programming, the term list may refer specifically to a linked list rather than an array.

In type theory and functional programming, abstract lists are usually defined inductively by four operations: *nil* that yields the empty list, *cons*, which adds an item at the beginning of a list, *first*, that returns the first element of a list, and *rest* that returns a list minus its first element. Formally, Peano's natural numbers can be defined as abstract lists with elements of unit type.

Operations

Implementation of the list data structure may provide some of the following operations:

- a constructor for creating an empty list;
- an operation for testing whether or not a list is empty;
- an operation for prepending an entity to a list
- an operation for appending an entity to a list
- an operation for determining the first component (or the "head") of a list
- an operation for referring to the list consisting of all the components of a list except for its first (this is called the "tail" of the list.)

Characteristics

Lists have the following properties:

- The **size of lists**. It indicates how many elements there are in the list.
- **Equality** of lists:
 - In mathematics, sometimes equality of lists is defined simply in terms of object identity: two lists are equal if and only if they are the same object.
 - In modern programming languages, equality of lists is normally defined in terms of structural equality of the corresponding entries, except that if the lists are typed, then the list types may also be relevant.
- Lists may be **typed**. This implies that the entries in a list must have types that are compatible with the list's type. It is common that lists are typed when they are implemented using arrays.
- Each element in the list has an **index**. The first element commonly has index 0 or 1 (or some other predefined integer). Subsequent elements have indices that are 1 higher than the previous element. The last element has index <initial index> + <size> – 1.
 - It is possible to retrieve the element at a particular index.
 - It is possible to traverse the list in the order of increasing index.
 - It is possible to change the element at a particular index to a different value, without affecting any other elements.
 - It is possible to insert an element at a particular index. The indices of previous elements at that and higher indices are increased by 1.
 - It is possible to remove an element at a particular index. The indices of previous elements at that and higher indices are decreased by 1.

Implementations

Lists are typically implemented either as linked lists (either singly or doubly-linked) or as arrays, usually variable length or dynamic arrays.

The standard way of implementing lists, originating with the programming language Lisp, is to have each element of the list contain both its value and a pointer indicating the location of the next element in the list. This results in either a linked list or a tree, depending on whether the list has nested sublists. Some older Lisp implementations (such as the Lisp implementation of the Symbolics 3600) also supported "compressed lists" (using CDR coding) which had a special internal representation (invisible to the user). Lists can be manipulated using iteration or recursion. The former is often preferred in imperative programming languages, while the latter is the norm in functional languages.

Programming language support

Some languages do not offer a list data structure, but offer the use of associative arrays or some kind of table to emulate lists. For example, Lua provides tables. Although Lua stores lists that have numerical indices as arrays internally, they still appear as hash tables.

In Lisp, lists are the fundamental data type and can represent both program code and data. In most dialects, the list of the first three prime numbers could be written as (list 2 3 5). In several dialects of Lisp, including Scheme, a list is a collection of pairs, consisting of a value and a pointer to the next pair (or null value), making a singly-linked list.

Applications

As the name implies, lists can be used to store a list of records. The items in a list can be sorted for the purpose of fast search (binary search).

Because in computing, lists are easier to realize than sets, a finite set in mathematical sense can be realized as a list with additional restrictions, that is, duplicate elements are disallowed and such that order is irrelevant. If the list is sorted, it speeds up determining if a given item is already in the set but in order to ensure the order, it requires more time to add new entry to the list. In efficient implementations, however, sets are implemented using self-balancing binary search trees or hash tables, rather than a list.

Abstract definition

The abstract list type L with elements of some type E (a monomorphic list) is defined by the following functions:

$$\begin{aligned} \text{nil: } () &\rightarrow L \\ \text{cons: } E \times L &\rightarrow L \\ \text{first: } L &\rightarrow E \\ \text{rest: } L &\rightarrow L \end{aligned}$$

with the axioms

$$\begin{aligned} \text{first}(\text{cons}(e, l)) &= e \\ \text{rest}(\text{cons}(e, l)) &= l \end{aligned}$$

for any element e and any list l . It is implicit that

$$\begin{aligned} \text{cons}(e, l) &\neq l \\ \text{cons}(e, l) &\neq e \\ \text{cons}(e_1, l_1) &= \text{cons}(e_2, l_2) \text{ if } e_1 = e_2 \text{ and } l_1 = l_2 \end{aligned}$$

Note that $\text{first}(\text{nil}())$ and $\text{last}(\text{nil}())$ are not defined.

These axioms are equivalent to those of the abstract stack data type.

In type theory, the above definition is more simply regarded as an inductive type defined in terms of constructors: nil and cons . In algebraic terms, this can be represented as the transformation $1 + E \times L \rightarrow L$. first and rest are then obtained by pattern matching on the cons constructor and separately handling the nil case.

The list monad

The list type forms a monad with the following functions (using E^* rather than L to represent monomorphic lists with elements of type E):

$$\begin{aligned} \text{return: } A \rightarrow A^* &= a \mapsto \text{cons } a \text{ nil} \\ \text{bind: } A^* \rightarrow (A \rightarrow B^*) \rightarrow B^* &= l \mapsto f \mapsto \begin{cases} \text{nil} & \text{if } l = \text{nil} \\ \text{append } (f a) (\text{bind } l' f) & \text{if } l = \text{cons } a l' \end{cases} \end{aligned}$$

where append is defined as:

$$\text{append: } A^* \rightarrow A^* \rightarrow A^* = l_1 \mapsto l_2 \mapsto \begin{cases} l_2 & \text{if } l_1 = \text{nil} \\ \text{cons } a (\text{append } l'_1 l_2) & \text{if } l_1 = \text{cons } a l'_1 \end{cases}$$

Alternately, the monad may be defined in terms of operations return , fmap and join , with:

$$\text{fmap: } (A \rightarrow B) \rightarrow (A^* \rightarrow B^*) = f \mapsto l \mapsto \begin{cases} \text{nil} & \text{if } l = \text{nil} \\ \text{cons } (f a) (\text{fmap } f l') & \text{if } l = \text{cons } a l' \end{cases}$$

$$\text{join}: A^{**} \rightarrow A^* = l \mapsto \begin{cases} \text{nil} & \text{if } l = \text{nil} \\ \text{append } a (\text{join } l') & \text{if } l = \text{cons } a \ l' \end{cases}$$

Note that *fmap*, *join*, *append* and *bind* are well-defined, since they're applied to progressively deeper arguments at each recursive call.

The list type is an additive monad, with *nil* as the monadic zero and *append* as monadic sum.

Lists form a monoid under the *append* operation. The identity element of the monoid is the empty list, *nil*. In fact, this is the free monoid over the set of list elements.

See also

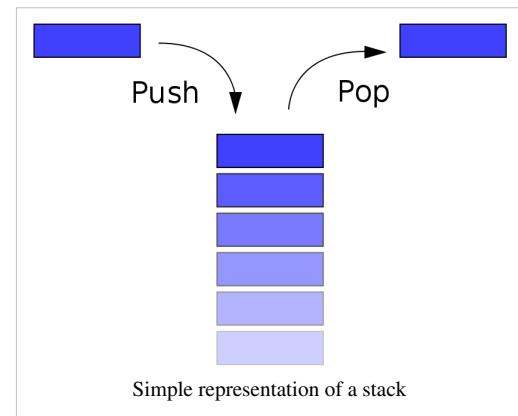
- Set (computer science)

Stack

In computer science, a **stack** is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: *push* and *pop*. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.

A stack is a *restricted data structure*, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order.

Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.



History

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

Inessential operations

In modern computer languages, the stack is usually implemented with more operations than just "push" and "pop". Some implementations have a function which returns the current length of the stack. Another typical helper operation *top*^[1] (also known as *peek*) can return the current top element of the stack without removing it.

Software stacks

Implementation

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

Array

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

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

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

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

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

```
int pop(STACK *ps)
{
    if (ps->size == 0) {
        fputs("Error: stack underflow\n", stderr);
        abort();
    } else
        return ps->items[--ps->size];
}
```

Linked list

The **linked-list** implementation is equally simple and straightforward. In fact, a stack linked-list is much simpler than most linked-list implementations: it requires that we implement a linked-list where only the head node or element can be removed, or popped, and a node can only be inserted by becoming the new head node.

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

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

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

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

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

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

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

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

```
}
```

Stacks and programming languages

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

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

Related data structures

The functionality of a queue and a stack can be combined in a deque data structure. Briefly put, a queue is a First In First Out (**FIFO**) data structure.

Hardware stacks

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

Basic architecture of a stack

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

The two operations applicable to all stacks are:

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

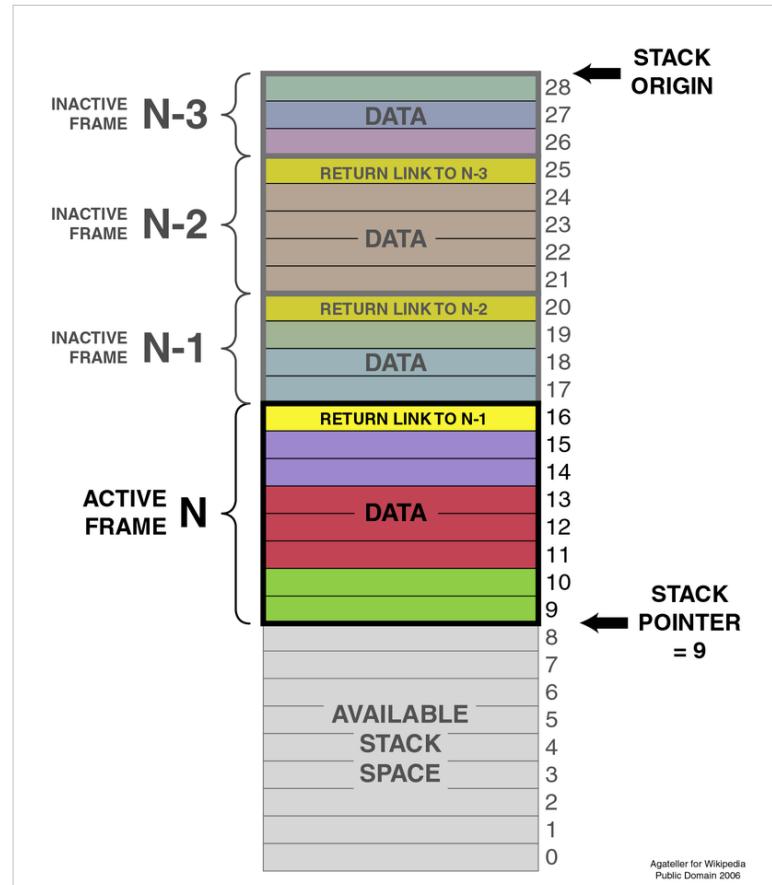
There are many variations on the basic principle of stack operations. Every stack has a fixed location in memory at which it begins. As data items are added to the stack, the stack pointer is displaced to indicate the current extent of the stack, which expands away from the origin (either up or down,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

For example, a stack might start at a memory location of one thousand, and expand towards lower addresses, in which case new data items are stored at locations ranging below 1000, and the stack pointer is decremented each time a new item is added. When an item is removed from the stack, the stack pointer is incremented.

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

Some environments that rely heavily on stacks may provide additional operations, for example:

- *Dup(licate)*: the top item is popped, and then pushed again (twice), so that an additional copy of the former top item is now on top, with the original below it.
- *Peek*: the topmost item is inspected (or returned), but the stack pointer is not changed, and the stack size does not change (meaning that the item remains on the stack). This is also called **top** operation in many articles.
- *Swap* or *exchange*: the two topmost items on the stack exchange places.



Agateller for Wikipedia
Public Domain 2006

A typical stack, storing local data and call information for nested procedure calls (not necessarily nested procedures!). This stack grows downward from its origin. The stack pointer points to the current topmost datum on the stack. A push operation decrements the pointer and copies the data to the stack; a pop operation copies data from the stack and then increments the pointer. Each procedure called in the program stores procedure return information (in yellow) and local data (in other colors) by pushing them onto the stack. This type of stack implementation is extremely common, but it is vulnerable to buffer overflow attacks (see the text).

- *Rotate*: the n topmost items are moved on the stack in a rotating fashion. For example, if $n=3$, items 1, 2, and 3 on the stack are moved to positions 2, 3, and 1 on the stack, respectively. Many variants of this operation are possible, with the most common being called *left rotate* and *right rotate*.

Stacks are either visualized growing from the bottom up (like real-world stacks), or, with the top of the stack in a fixed position (see image [note in the image, the top (28) is the stack 'bottom', since the stack 'top' is where items are pushed or popped from]), a coin holder, a Pez dispenser, or growing from left to right, so that "topmost" becomes "rightmost". This visualization may be independent of the actual structure of the stack in memory. This means that a *right rotate* will move the first element to the third position, the second to the first and the third to the second. Here are two equivalent visualizations of this process:

apple		banana
banana	====right rotate==>	cucumber
cucumber		apple
cucumber		apple
banana	====left rotate==>	cucumber
apple		banana

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

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

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

Hardware support

Stack in main memory

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

Stack in registers or dedicated memory

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

exemplified by modern x87 implementations.

Sun SPARC, AMD Am29000, and Intel i960 are all examples of architectures using register windows within a register-stack as another strategy to avoid the use of slow main memory for function arguments and return values.

There are also a number of small microprocessors that implements a stack directly in hardware and some microcontrollers have a fixed-depth stack that is not directly accessible. Examples are the PIC microcontrollers, the Computer Cowboys MuP21, the Harris RTX line, and the Novix NC4016. Many stack-based microprocessors were used to implement the programming language Forth at the microcode level. Stacks were also used as a basis of a number of mainframes and mini computers. Such machines were called stack machines, the most famous being the Burroughs B5000.

Applications

Stacks are ubiquitous in the computing world.

Expression evaluation and syntax parsing

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

Example (general)

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

```
1 2 4 * + 3 +
```

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

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

Like the following way (the *Stack* is displayed after *Operation* has taken place):

Input	Operation	Stack (after op)
1	Push operand	1
2	Push operand	2, 1
4	Push operand	4, 2, 1
*	Multiply	8, 1
+	Add	9
3	Push operand	3, 9
+	Add	12

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

Example in C

```
#include<stdio.h>

int main()
```

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

Example (Pascal)

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

```
{
programmer : clx321
file   : stack.pas
unit   : Pstack.tpu
}

program TestStack;
{this program use ADT of Stack, i will assume that the unit of ADT of
Stack has already existed}

uses
  PStack;    {ADT of STACK}

{dictionary}
const
  mark = '.';

var
  data : stack;
  f : text;
  cc : char;
```

```
ccInt, cc1, cc2 : integer;

{functions}
IsOperand (cc : char) : boolean;      {JUST Prototype}
  {return TRUE if cc is operand}
ChrToInt (cc : char) : integer;        {JUST Prototype}
  {change char to integer}
Operator (cc1, cc2 : integer) : integer; {JUST Prototype}
  {operate two operands}

{algorithms}
begin
  assign (f, cc);
  reset (f);
  read (f, cc);  {first elmt}
  if (cc = mark) then
    begin
      writeln ('empty archives !');
    end
  else
    begin
      repeat
        if (IsOperand (cc)) then
          begin
            ccInt := ChrToInt (cc);
            push (ccInt, data);
          end
        else
          begin
            pop (cc1, data);
            pop (cc2, data);
            push (data, Operator (cc2, cc1));
          end;
        read (f, cc);  {next elmt}
      until (cc = mark);
    end;
  close (f);
end.
```

Runtime memory management

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

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

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

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

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

Security

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

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

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

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

See also

- Queue
- Double-ended queue
- Call stack
- Stack-based memory allocation
- Stack machine

Further reading

- Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.
- 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 10.1: Stacks and queues, pp. 200–204.

External links

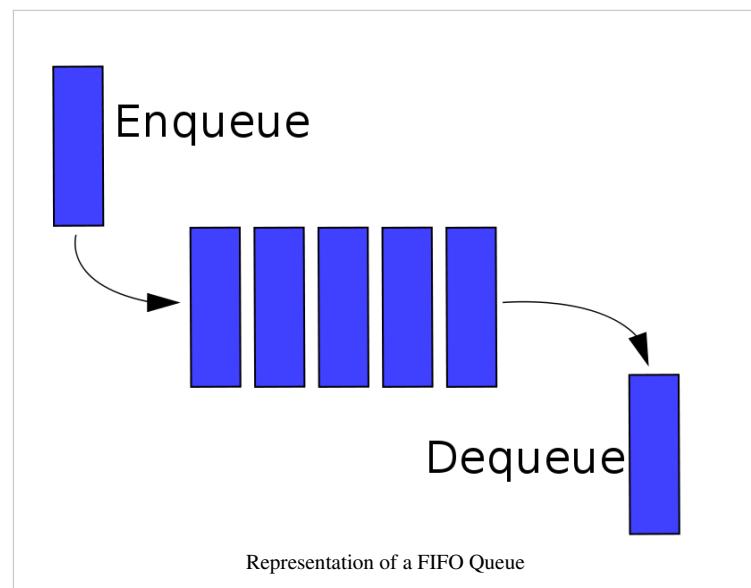
- Stack Machines - the new wave ^[2]
- Bounding stack depth ^[3]
- Libsafe - Protecting Critical Elements of Stacks ^[4]
- Stack Size Analysis for Interrupt-driven Programs ^[5] (322 KB)
- Pointers to stack visualizations ^[6]
-  This article incorporates public domain material from the NIST document "Bounded stack" ^[7] by Paul E. Black (Dictionary of Algorithms and Data Structures).

References

- [1] Horowitz, Ellis: "Fundamentals of Data Structures in Pascal", page 67. Computer Science Press, 1984
- [2] http://www.ece.cmu.edu/~koopman/stack_computers/index.html
- [3] <http://www.cs.utah.edu/~regehr/stacktool>
- [4] <http://research.avayalabs.com/project/libsafe/>
- [5] <http://www.cs.ucla.edu/~palsberg/paper/sas03.pdf>
- [6] <http://web-cat.cs.vt.edu/AlgovizWiki/Stacks>
- [7] <http://www.nist.gov/dads/HTML/boundedstack.html>

Queue

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



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

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

Operations

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

`bool empty()`

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

`T& front()`

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

`void pop()`

Removes the item at the front of a non-empty queue.

`void push(const T& foo)`

Inserts the argument foo at the back of the queue.

`size_type size()`

Returns the total number of elements in the queue.

Example C Program

```
#include <stdio.h>

int main() {
    int a[100], i, j;

    printf("To DQueue Enter -1\n");

    for(i=0;;) {
        printf("NQueue ");
        scanf("%d", &a[i]);

        if(a[i]==0)
            break;

        if(a[i]==-1) {
            a[i]=0;

            if(i==0) {
                printf("Wrong\n");
                continue;
            }
            printf("DQueue = %d\n", a[0]);
        }

        for(j=0; j<i; j++)
            a[j]=a[j+1];
        i--;
    }
    else
        i++;
}

for(j=0; j<i; j++)
    printf("%d ", a[j]);

return 0;
}
```

Example Pascal Program

Based on an example in R.S Algorithms .

```
program queue;
type
link=^node;
node=record
next:link;
```

```
key:integer;
end;

var

U:integer;
tail,Head:link;

procedure queinit; // Initializes the queue
begin

  New(tail);
  tail^.next:=nil;
  head:=tail;
end;

procedure put(u: integer); // Puts number u into the queue
var
  t: link;
begin
  New(t);
  tail^.key := u;
  tail^.next := t;
  tail := t;
end;

function pop:integer; //Pops one number at a time
var
  s:link;
begin
  pop:=head^.key;
  s:=head;
  head:=head^.next;
  dispose(s);
end;

function empty:boolean;//checks if the queue is empty
begin
  empty:=head=tail;
end;

begin
  queinit;
  u := 1;
  put(u); // Put 1 in the queue

  u := 2;
```

```
put(u); // Put 2 in the queue  
  
u := 3;  
put(u); // Put 3 in the queue  
end.
```

Example C# Class

```
using System;  
  
namespace QueueExample  
{  
    public class Queue  
    {  
        private object[] _Queue;  
        private int _HeadPointer;  
        private int _TailPointer;  
        private int _Size;  
        private int _Count;  
  
        #region Constructor  
        public Queue(int Size)  
        {  
            this._Size = Size;  
            this._Queue = new object[Size];  
            this._HeadPointer = 0;  
            this._TailPointer = 0;  
            this._Count = 0;  
        }  
        #endregion  
  
        #region Methods  
        public virtual void Enqueue(object item)  
        {  
            lock (this)  
            {  
                if (this.IsFull())  
                {  
                    throw new Exception("Queue is full !");  
                }  
                else  
                {  
                    this._Queue[this._TailPointer] = item;  
                    this._Count++;  
                }  
  
                // Bump tail pointer to next available location  
                this._TailPointer = this._TailPointer + 1;  
            }  
        }  
    }  
}
```

```
        if (_TailPointer >= _Queue.Length)
        {
            _TailPointer = 0;
        }
    }

    public virtual object Dequeue()
    {
        lock (this)
        {
            object item = null;

            if (_Queue.IsEmpty())
            {
                throw new Exception("Queue is empty !");
            }
            else
            {
                item = _Queue[_HeadPointer];
                _Count--;

                // Bump head pointer, wrap to beginning if it
                reaches the end
                _HeadPointer = _HeadPointer + 1;
                if (_HeadPointer >= _Queue.Length)
                {
                    _HeadPointer = 0;
                }
            }

            return item;
        }
    }

    public virtual bool IsEmpty()
    {
        return (_Count == 0);
    }

    public virtual bool IsFull()
    {
        return (_Count == _Size);
    }

    public virtual int Count()
    {
```

```

        return this._Count;
    }
    #endregion
}
}

```

Example Java Code

```

public void insert(Element x) {
    Node New;
    New = new Node(x, null);
    if (NodeHead == null) {
        NodeHead = New;
    } else {
        NodeEnd.Next = New;
    }
    NodeEnd = New;
}

public Element head() throws IllegalArgumentException {
    if (NodeHead == null) {
        throw new IllegalArgumentException();
    } else {
        return NodeHead.Info;
    }
}

public Queue() {
    // Return Empty Queue
    NodeHead = null;
    NodeEnd = null;
}

```

Example Javascript Code

```

function Queue() {
    this.data = [];

    this.isEmpty = function() {
        return (this.data.length == 0);
    };

    this.enqueue = function(obj) {
        this.data.push(obj);
    }

    this.dequeue = function() {
        var ret = this.data[0];
        this.data.splice(0, 1);
        return ret;
    }
}

```

```
        this.data.splice(0,1);
        return ret;
    }

    this.peek = function() {
        return this.data[0];
    }

    this.clear = function() {
        this.data = [];
    }
}
```

Representing a Queue

The defining attribute of a queue data structure is the fact that allows access to only the front and back of the structure. Furthermore, elements can only be removed from the front and can only be added to the back. In this way, an appropriate metaphor often used to represent queues is the idea of a **checkout line** (*Ford/Topp p. 385*). Other examples of queues are people traveling up an escalator, machine parts on an assembly line, or cars in line at a petrol station. The recurring theme is clear: queues are essentially the same as a queue you would get in a shop waiting to pay.

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

Queue implementation

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

A practical implementation of a queue, e.g. with pointers, of course does have some capacity limit, that depends on the concrete situation it is used in. For a data structure the executing computer will eventually run out of memory, thus limiting the queue size. Queue **overflow** results from trying to add an element onto a full queue and queue **underflow** happens when trying to remove an element from an empty queue.

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

See also

- Deque
- Priority queue
- Queueing theory
- Stack - the "opposite" of a queue: LIFO (Last In First Out)
- Circular buffer

References

- Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.
- 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 10.1: Stacks and queues, pp. 200–204.
- William Ford, William Topp. *Data Structures with C++ and STL*, Second Edition. Prentice Hall, 2002. ISBN 0-13-085850-1. Chapter 8: Queues and Priority Queues, pp. 386–390.
- Adam Drozdek. *Data Structures and Algorithms in C++*, Third Edition. Thomson Course Technology, 2005. ISBN 0-534-49182-0. Chapter 4: Stacks and Queues, pp. 137–169.

External links

- STL Quick Reference ^[1]
- qDecoder's C++ circular queue implementation ^[2] — opensource library which supports FIFO and STACK for the user objects
- CollectionSpy ^[3] — A profiler for Java's Collections Framework with explicit support for plain and blocking queues.

 This article incorporates public domain material from the NIST document "Bounded queue" ^[4] by Paul E. Black (Dictionary of Algorithms and Data Structures).

References

- [1] <http://www.halpernwrightsoftware.com/stdc++-scratch/quickref.html#containers14>
- [2] <http://www.qdecoder.org/goto/qQueue.html>
- [3] <http://www.collectionspy.com>
- [4] <http://www.nist.gov/dads/HTML/boundedqueue.html>

Deque

In computer science, a **double-ended queue** (often abbreviated to **deque**, pronounced *deck*) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail).^[1] It is also often called a **head-tail linked list**.

Naming conventions

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

Distinctions and sub-types

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

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

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

Operations

The following operations are possible on a deque:

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

Implementations

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

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

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

Language support

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

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

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

Python 2.4 introduced the `collections` module with support for deque objects.

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

GHC's `Data.Sequence`^[2] module implements an efficient, functional deque structure in Haskell.

Complexity

- In a doubly-linked list implementation, the time complexity of all deque operations is $O(1)$. Additionally, the time complexity of insertion or deletion in the middle, given an iterator, is $O(1)$; however, the time complexity of random access by index is $O(n)$.
- In a growing array, the amortized time complexity of all deque operations is $O(1)$. Additionally, the time complexity of random access by index is $O(1)$; but the time complexity of insertion or deletion in the middle is $O(n)$.

Applications

One example where a deque can be used is the *A-Steal* job scheduling algorithm.^[3] This algorithm implements task scheduling for several processors. A separate deque with threads to be executed is maintained for each processor. To execute the next thread, the processor gets the first element from the deque (using the "remove first element" deque operation). If the current thread forks, it is put back to the front of the deque ("insert element at front") and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. its deque is empty), it can "steal" a thread from another processor: it gets the last element from the deque of another processor ("remove last element") and executes it.

See also

- Queue
- Priority queue

External links

- SGI STL Documentation: deque<T, Alloc>^[4]
- Code Project: An In-Depth Study of the STL Deque Container^[5]
- Diagram of a typical STL deque implementation^[6]
- deque implementation in flash actionscript 3 open source library^[7]

References

- [1] Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.
- [2] <http://www.haskell.org/ghc/docs/latest/html/libraries/containers/Data-Sequence.html>
- [3] Eitan Frachtenberg, Uwe Schwiegelshohn (2007). *Job Scheduling Strategies for Parallel Processing: 12th International Workshop, JSSPP 2006*. Springer. ISBN 3540710345. See p.22.
- [4] <http://www.sgi.com/tech/stl/Deque.html>
- [5] http://www.codeproject.com/KB/stl/vector_vs_deque.aspx
- [6] <http://pages.cpsc.ucalgary.ca/~kremer/STL/1024x768/deque.html>
- [7] <http://www.dpdk.nlopensource>

Priority queue

A **priority queue** is an abstract data type in computer programming that supports the following three operations:

- insertWithPriority: add an element to the queue with an associated priority
- getNext: remove the element from the queue that has the *highest priority*, and return it (also known as "PopElement(Off)", or "GetMinimum")
- peekAtNext (optional): look at the element with *highest priority* without removing it

For an analogy, see the Implementation section below.

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 one 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

Simple 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:

- *Sorted list implementation:* Like a checkout line at the supermarket, but where important people get to "cut" in front of less important people. ($O(\log(n))$) insertion time (can binary search for insertion position) if implemented using arrays, $O(n)$ insertion time if implemented using linked lists; $O(1)$ get-next time)
- *Unsorted list implementation:* Keep a list of elements as the queue. To add an element, append it to the end. To get the next element, search through all elements for the one with the highest priority. ($O(1)$ insertion time, $O(n)$ get-next due to search)

These implementations are usually inefficient, though can be good depending on the workload (for example, if one does very few GetNext operations, the unsorted list approach may work well). In practice, priority queues blend these two approaches, so any operation takes roughly $O(\log(n))$ time or less.

Usual implementation

To get better performance, priority queues typically use a heap as their backbone, giving $O(\log n)$ performance for inserts and removals. Alternatively, if a self-balancing binary search tree is used, all three operations take $O(\log n)$ time; this is a popular solution where one already has access to these data structures, such as through third-party or standard libraries.

Effect of different data structures

The designer of the priority queue should take into account what sort of access pattern the priority queue will be subject to, and what computational resources are most important to the designer. The designer can then use various specialized types of heaps:

There are a number of specialized heap data structures that either supply additional operations or outperform the above approaches. The binary heap uses $O(\log n)$ time for both operations, but allows peeking at the element of highest priority without removing it in constant time. Binomial heaps add several more operations, but require $O(\log n)$ time for peeking. Fibonacci heaps can insert elements, peek at the maximum priority element, and increase an element's priority in amortized constant time (deletions are still $O(\log n)$).

While relying on a heap is a common way to implement priority queues, for integer data faster implementations exist (this can even apply to datatypes that have finite range, such as floats):

- When the set of keys is $\{1, 2, \dots, C\}$, a van Emde Boas tree supports the *minimum*, *maximum*, *insert*, *delete*, *search*, *extract-min*, *extract-max*, *predecessor* and *successor* operations in $O(\log \log C)$ time, but has a space cost for small queues of about $O(2^{m/2})$, where m is the number of bits in the priority value.^[1]
- The Fusion tree algorithm by Fredman and Willard implements the *minimum* operation in $O(1)$ time and *insert* and *extract-min* operations in $O(\sqrt{\log n})$ time.^[2]

For applications that do many "peek" operations for every "extract-min" operation, the time complexity for peek can be reduced to $O(1)$ in all tree and heap implementations by caching the highest priority element after every insertion and removal. (For insertion this adds at most constant cost, since the newly inserted element need only be compared to the previously cached minimum element. For deletion, this at most adds an additional "peek" cost, which is nearly always cheaper than the deletion cost, so overall time complexity is not affected by this change).

Container libraries

The Standard Template Library (STL), and the C++ 1998 standard, specifies priority_queue^[3] as one of the STL container adaptor class templates. It implements a max-priority-queue. Unlike actual STL containers, it does not allow iteration of its elements (it strictly adheres to its abstract data type definition). STL also has utility functions for manipulating another random-access container as a binary max-heap.

Python's heapq^[4] module implements a binary min-heap on top of a list.

Java's library contains a PriorityQueue class, which implements a min-priority-queue.

The Standard PHP Library extension contains the class SplPriorityQueue^[5]

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

A* and SMA* search algorithms

The A* search algorithm finds the shortest path between two vertices 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.

Relationship to sorting algorithms

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.
- Selection sort if the priority queue is implemented with an ordered array.
- Insertion sort if the priority queue is implemented with an unordered array.

A sorting algorithm can also be used to implement a priority queue.

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

- Descriptions ^[6] by Lee Killough
- PQlib ^[7] - Open source Priority Queue library for C
- libpqueue ^[8] is a generic priority queue (heap) implementation (in C) used by the Apache HTTP Server project.
- Survey of known priority queue structures ^[9] by Stefan Xenos
- UC Berkeley - Computer Science 61B - Lecture 24: Priority Queues ^[10] (video) - introduction to priority queues using binary heap
- Double-Ended Priority Queues ^[11] by Sartaj Sahni

References

- [1] 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.
- [2] 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
- [3] http://www.cplusplus.com/reference/stl/priority_queue/
- [4] <http://www.python.org/doc/lib/module-heappq.html>
- [5] <http://us2.php.net/manual/en/class.splpriorityqueue.php>
- [6] <http://leekillough.com/heaps/>
- [7] <http://bitbucket.org/trijezdc/pqlib/src/>
- [8] <http://github.com/vy/libpqueue>
- [9] <http://www.theturingmachine.com/algorithms/heaps.html>
- [10] <http://video.google.com/videoplay?docid=3499489585174920878>
- [11] <http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c13/double.htm>

Map

An **associative array** (also **associative container**, **map**, **mapping**, **dictionary**, **finite map**, and in query-processing an **index** or **index file**) is an abstract data type composed of a collection of unique keys and a collection of values, where each key is associated with one value (or set of values). The operation of finding the value associated with a key is called a *lookup* or indexing, and this is the most important operation supported by an associative array. The relationship between a key and its value is sometimes called a mapping or binding. For example, if the value associated with the key "bob" is 7, we say that our array *maps* "bob" to 7. Associative arrays are very closely related to the mathematical concept of a function with a finite domain. As a consequence, a common and important use of associative arrays is in memoization.

From the perspective of a computer programmer, an associative array can be viewed as a generalization of an array. While a regular array maps an integer key (index) to a value of arbitrary data type, an associative array's keys can also be arbitrarily typed. In some programming languages, such as Python, the keys of an associative array do not even need to be of the same type.

Content-addressable memory (CAM) systems use a special type of computer memory to improve the performance of lookups in associative arrays and are used in specialized applications. Several supercomputers from the 1970s implemented CAM directly in hardware, and were known as *associative computers*.

Operations

The operations that are usually defined for an associative array are:

- **Add:** Bind a new key to a new value
- **Reassign:** Bind an old key to a new value
- **Remove:** Unbind a key from a value and remove the key from the key set
- **Lookup:** Find the value (if any) that is bound to a key

Basic abstract arrays

An abstract array structure has two main operations, *fetch* and *store*. In the functional specification style, where the entities are array states rather than the array themselves, they are

- $\text{fetch}(A, I)$: obtains the data stored in the element of the array whose state is A and whose indices I .
- $\text{store}(A, I, V)$: the array state that results by setting the value of that element to V in array state A .

These operations are required to satisfy the axioms^[1]

- $\text{fetch}(\text{store}(A, I, V), I) = V$
- $\text{fetch}(\text{store}(A, I, V), J) = \text{fetch}(A, J)$ if $I \neq J$

for any array state A , any value V , and any tuples I, J for which the operations are defined.

The first axiom means that each element behaves like an ordinary variable. The second axiom means that elements with distinct indices behave as disjoint variables, so that storing a value in one element does not affect the value of any other element.

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

Variations of abstract arrays

A **resizable**, **growable**, or **dynamic array** is an array ADT that allows the range of indices to be increased without changing the values of its current elements. For one-dimensional arrays, this facility is often provided as an $\text{append}(A,x)$ operation that increases the size of the array A by one and then sets the value of the last element to x . Resizable arrays are one popular way of implementing lists.

Examples

One can think of a telephone book as an example of an associative array, where names are the keys and phone numbers are the values. Using the usual array-like notation, we might write

```
phonebook['ada'] = '01-1234-56'
phonebook['charles'] = '02-4321-56'
```

and so on. These entries can be thought of as two records in a database table:

Name (Index)	Phone # (Value)
ada	01-1234-56
charles	02-4321-56

To retrieve the element from the associative array, we use a similar notation i.e.

```
x = phonebook['ada']
y = phonebook['charles']
```

or alternatively:

```
x = fetch( phonebook , ada );
y = fetch( phonebook , charles );
```

where for both examples, $x = 01-1234-56$ and $y = 02-4321-56$ after execution.

Another example would be a dictionary where words are the keys and definitions are the values.

```
dictionary['toad'] = 'four legged amphibian'
dictionary['cow'] = 'female four-legged domesticated mammalian ruminant'
```

Since a database equivalent is that of a table containing precisely two fields - key and value - we can use an associative array to store any information which can be held in this form.

```
capital['england'] = 'london'
capital['france'] = 'paris'

bossof['subeditor'] = 'editor'
bossof['reporter'] = 'subeditor'

salary['editor'] = 50000
salary['reporter'] = 30000
```

Data structures for representing

Associative arrays are usually used when lookup is the most frequent operation. For this reason, implementations are usually designed to allow speedy lookup, at the expense of slower insertion and a larger storage footprint than other data structures (such as association lists).

Efficient representations

There are two main efficient data structures used to represent associative arrays, the hash table and the self-balancing binary search tree (such as a red-black tree or an AVL tree). Skip lists are also an alternative, though relatively new and not as widely used. B-trees (and variants) can also be used, and are commonly used when the associative array is too large to reside entirely in memory, for instance in a simple database. Relative advantages and disadvantages include:

- **Asymptotic operation performance:** Hash tables have faster average lookup and insertion time, $O(1)$, compared to a balanced binary search tree's $\Theta(\log n)$, while balanced trees have faster worst-case lookup and insertion time, $O(\log n)$ as compared to $\Theta(n)$. Skip lists have $O(n)$ worst-case and $O(\log n)$ average-case operation times, but with less insertion and deletion overhead in practice than balanced binary trees.
- **Ordering preservation:** Balanced binary trees and skip lists preserve ordering — allowing one to efficiently iterate over the keys in order or to efficiently locate an association whose key is nearest to a given value. Hash tables do not preserve ordering and therefore cannot perform these operations as efficiently (they require the data to be sorted in a separate step).
- **Range queries:** Balanced binary trees can be easily adapted to efficiently assign a single value to a large ordered range of keys, or to count the number of keys in an ordered range. (With n elements in the array and performing the operation on a contiguous range of m keys, a balanced binary tree will take $O(\log(n)+m)$ time while a hash table would need $\Theta(n)$ time as it needs to search the entire table.)
- **Allocation behavior:** Hash tables with open addressing store all data in a large contiguous block of memory that is reallocated infrequently, whereas tree allocations perform many small, frequent allocations. As a result hash tables may be difficult to allocate in a fragmented heap, and conversely trees may cause heap fragmentation. Trees are also more vulnerable to inefficiencies in allocators.
- **Compactness:** Hash tables can have more compact storage for small value types, especially when the values are bits.
- **Persistence:** There are simple persistent versions of balanced binary trees, which are especially prominent in functional languages.
- **Supporting new key types:** Building a hash table requires a reasonable hash function for the key type, which can be difficult to write well, while balanced binary trees and skip lists only require a total ordering on the keys.

Sometimes simple implementations of one data structure or the other have disadvantages that can be overcome by better design. For example:

- Hash tables that use untrusted input as keys may be vulnerable to denial-of-service attacks where an untrusted user supplies data intended to generate large numbers of collisions. This may be overcome by choosing hash functions at random from a universal family, or by hashing untrusted input with a cryptographic hash function before insertion.
- Simple balanced trees waste space on pointers and allocation metadata; these problems can be mitigated by storing multiple elements in each node and by using memory pools.

Association lists

A simple but generally inefficient type of associative array is an *association list*, [2] [3] often called an *alist* for short, which simply stores a linked list of key-value pairs. Each lookup does a linear search through the list looking for a key match. Such a data structure is commonly used in Lisp/Scheme. [4]

Advantages of association lists include:

- It need only be known how to test keys for *equality* — which is minimal for maps supporting the four basic operations — while the above alternatives require a *linear order* comparison or a *hash function*.
- For small associative arrays, common in some applications, association lists can take less time and space than other data structures.
- Insertions are done in constant time by adding the new association to the head of the list.

Specialized representations

If the keys have a specific type, one can often use specialized data structures to gain performance. For example, integer-keyed maps can be implemented using radix trees or Judy arrays, and are useful space-saving replacements for sparse arrays. Because this type of data structure can perform longest-prefix matching, they're particularly useful in applications where a single value is assigned to most of a large range of keys with a common prefix except for a few exceptions, such as in routing tables.

String-keyed maps can avoid extra comparisons during lookups by using tries.

Multimap

A variation of the map (associative array) is the multimap, which is the same as map data structures, but allows a key to be mapped to more than one value. Formally, a multimap can be thought of as a regular associative array that maps unique keys to nonempty sets of values, although actual implementation may vary. C++'s Standard Template Library provides the "multimap" container for the sorted multimap, SGI's STL provides the "hash_multimap" container, which implements a multimap using a hash table, and some varieties of LPC have built-in multimap support.

Language support

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

Built-in syntactic support for associative arrays was introduced by SNOBOL4, under the name "table". MUMPS made multi-dimensional associative arrays, optionally persistent, its key data structure. SETL supported them as one possible implementation of sets and maps. Most modern scripting languages, starting with AWK and including Perl, Tcl, JavaScript, Python, Ruby, and Lua, support associative arrays as a primary container type.

In many more languages, they are available as library functions without special syntax.

Associative arrays have a variety of names. In Smalltalk, Objective-C, .NET, Python, and REALbasic they are called *dictionaries*; in Perl and Ruby they are called *hashes*; in C++ and Java they are called *maps* (see map (C++) and Map) and in Common Lisp and Windows PowerShell they are called *hashtables* (since both typically use this implementation). In PHP all arrays can be associative, except that the keys are limited to integers and strings. In JavaScript, all objects behave as associative arrays. In Visual FoxPro they are called Collections.

In the scripting language Lua, associative arrays, called *tables*, are used as the primitive building block for all data structures, even arrays. In MUMPS, the associative arrays are typically stored as B-trees.

See also

- Bidirectional map
- Lookup table

External links

- TreeDictionary<K,V> Implementation in C# ^[5]
- NIST's Dictionary of Algorithms and Data Structures: Associative Array ^[6]
- NIST's Dictionary of Algorithms and Data Structures: Association List ^[7]
- SGI STL page on Associative Containers ^[8]
- SGI STL page on std::map ^[8]
- SGI STL page on std::hash_map ^[9]

References

- [1] Lukham, Suzuki (1979), "Verification of array, record, and pointer operations in Pascal". *ACM Transactions on Programming Languages and Systems* **1**(2), 226–244.
- [2] "dictionary" (<http://www.nist.gov/dads/HTML/dictionary.html>). National Institute of Standards and Technology. .
- [3] "Data Structures" (<http://books.google.com/books?id=nh0okI1a1sQC&pg=PA299>). *Real World Haskell*. O'Reilly. 2008. p. 299. ISBN 9780596514983. .
- [4] "When should I use a hash table instead of an association list?" (<http://www.faqs.org/faqs/lisp-faq/part2/section-2.html>). lisp-faq/part2. 1996-02-20. .
- [5] <http://goletas.com/solutions/collections/>
- [6] <http://www.nist.gov/dads/HTML/assocarray.html>
- [7] <http://www.nist.gov/dads/HTML/dictionary.html>
- [8] <http://www.sgi.com/tech/stl/Map.html>
- [9] http://www.sgi.com/tech/stl/hash_map.html

Bidirectional map

In computer science, a **bidirectional map** is an associative data structure in which both types can be used as key.

External links

- <http://commons.apache.org/collections/api-release/org/apache/commons/collections/BidiMap.html>
- http://cablemodem.fibertel.com.ar/mcape/oss/projects/mc_projects/boost_projects/boost_bimap.html#
- <http://www.codeproject.com/KB/stl/bimap.aspx>

Multimap

A **mymap** (sometimes also **multihash**) is a generalization of a map or associative array abstract data type in which more than one value may be associated with and returned for a given key. Both map and mymap are particular cases of containers (see for example C++ Standard Template Library containers). Often the mymap is implemented as a map with lists or sets as the map values.

Examples

- In a student enrollment system, where students may be enrolled in multiple classes simultaneously, there might be an association for each enrollment of a student in a course, where the key is the student ID and the value is the course ID. If a student is enrolled in three courses, there will be three associations containing the same key.
- The index of a book may report any number of references for a given index term, and thus may be coded as a mymap from index terms to any number of reference locations.

Language support

C++'s Standard Template Library provides the mymap container for the sorted mymap using a self-balancing binary search tree,^[1] and SGI's STL extension provides the hash_mymap container, which implements a mymap using a hash table.^[2]

Apache Commons Collections provides a MultiMap interface for Java.^[3] It also provides a MultiValueMap implementing class that makes a MultiMap out of a Map object and a type of Collection.^[4]

Google Collections also provides an interface Multimap and implementations.^[5]

See also

- Abstract data type for the type of concept in general
- Associative array for the more fundamental abstract data type
- Multiset for the case where same item can appear several times

References

- [1] "mymap<Key, Data, Compare, Alloc>" (<http://www.sgi.com/tech/stl/Mymap.html>). *Standard Template Library Programmer's Guide*. Silicon Graphics International. .
- [2] "hash_multiset<Key, HashFcn, EqualKey, Alloc>" (http://www.sgi.com/tech/stl/hash_multiset.html). *Standard Template Library Programmer's Guide*. Silicon Graphics International. .
- [3] "Interface MultiMap" (<http://commons.apache.org/collections/api-release/org/apache/commons/collections/MultiMap.html>). *Commons Collections 3.2.1 API, Apache Commons. .*

-
- [4] "Class MultiValueMap" (<http://commons.apache.org/collections/api-release/org/apache/commons/collections/map/MultiValueMap.html>). *Commons Collections 3.2.1 API, Apache Commons.* .
 - [5] "Interface Multimap<K,V>" (<http://google-collections.googlecode.com/svn/trunk/javadoc/com/google/common/collect/Multimap.html>). *Google Collections Library 1.0.* .

Set

In computer science, a **set** is an abstract data structure that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set.

Some set data structures are designed for **static sets** that do not change with time, and allow only query operations — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and/or deletion of elements from the set.

A set can be implemented in many ways. For example, one can use a list, ignoring the order of the elements and taking care to avoid repeated values. Sets are often implemented using various flavors of trees, tries, hash tables, and more.

A set can be seen, and implemented, as a (partial) associative array, in which the value of each key-value pair has the unit type.

In type theory, sets are generally identified with their indicator function: accordingly, a set of values of type A may be denoted by 2^A or $\mathcal{P}(A)$. (Subtypes and subsets may be modeled by refinement types, and quotient sets may be replaced by setoids.)

Operations

Typical operations that may be provided by a static set structure S are

- $\text{element_of}(x,S)$: checks whether the value x is in the set S .
- $\text{empty}(S)$: checks whether the set S is empty.
- $\text{size}(S)$: returns the number of elements in S .
- $\text{enumerate}(S)$: yields the elements of S in some arbitrary order.
- $\text{pick}(S)$: returns an arbitrary element of S .
- $\text{build}(x_1, x_2, \dots, x_n)$: creates a set structure with values x_1, x_2, \dots, x_n .

The enumerate operation may return a list of all the elements, or an iterator, a procedure object that returns one more value of S at each call.

Dynamic set structures typically add:

- $\text{create}(n)$: creates a new set structure, initially empty but capable of holding up to n elements.
- $\text{add}(S,x)$: adds the element x to S , if it is not there already.
- $\text{delete}(S,x)$: removes the element x from S , if it is there.
- $\text{capacity}(S)$: returns the maximum number of values that S can hold.

Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted.

There are many other operations that can (in principle) be defined in terms of the above, such as:

- $\text{pop}(S)$: returns an arbitrary element of S , deleting it from S .
- $\text{find}(S, P)$: returns an element of S that satisfies a given predicate P .
- $\text{clear}(S)$: delete all elements of S .

In particular, one may define the Boolean operations of set theory:

- $\text{union}(S,T)$: returns the union of sets S and T .

- $\text{intersection}(S, T)$: returns the intersection of sets S and T .
- $\text{difference}(S, T)$: returns the difference of sets S and T .
- $\text{subset}(S, T)$: a predicate that tests whether the set S is a subset of set T .

Other operations can be defined for sets with elements of a special type:

- $\text{sum}(S)$: returns the sum of all elements of S (for some definition of "sum").
- $\text{nearest}(S, x)$: returns the element of S that is closest in value to x (by some criterion).

In theory, many other abstract data structures can be viewed as set structures with additional operations and/or additional axioms imposed on the standard operations. For example, an abstract heap can be viewed as a set structure with a $\text{min}(S)$ operation that returns the element of smallest value.

Implementations

Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations. Some implementations are designed to improve the efficiency of very specialized operations, such as nearest or union. Implementations described as "general use" typically strive to optimize the element_of, add, and delete operation.

Sets are commonly implemented in the same way as associative arrays, namely, a self-balancing binary search tree for sorted sets (which has $O(\log n)$ for most operations), or a hash table for unsorted sets (which has $O(1)$ average-case, but $O(n)$ worst-case, for most operations). A sorted linear hash table^[1] may be used to provide deterministically ordered sets.

Other popular methods include arrays. In particular a subset of the integers $1..n$ can be implemented efficiently as an n -bit bit array, which also support very efficient union and intersection operations. A Bloom map implements a set probabilistically, using a very compact representation but risking a small chance of false positives on queries.

The Boolean set operations can be implemented in terms of more elementary operations (pop, clear, and add), but specialized algorithms may yield lower asymptotic time bounds. If sets are implemented as sorted lists, for example, the naive algorithm for $\text{union}(S, T)$ will take code proportional to the length m of S times the length n of T ; whereas a variant of the list merging algorithm will do the job in time proportional to $m+n$. Moreover, there are specialized set data structures (such as the union-find data structure) that are optimized for one or more of these operations, at the expense of others.

Language support

One of the earliest languages to support sets was Pascal; many languages now include it, whether in the core language or in a standard library.

- Java offers the Set interface to support sets (with the HashSet class implementing it using a hash table), and the SortedSet sub-interface to support sorted sets (with the TreeSet class implementing it using a binary search tree).
- In C++, STL provides the set template class, which implements a sorted set using a binary search tree; and SGI's STL provides the hash_set class, which implements a set using a hash table.
- Apple's Foundation framework (part of Cocoa) provides the Objective-C classes NSSet^[2], NSMutableSet^[3], and NSCountedSet^[4]. The CoreFoundation APIs provide the CFSet^[5] and CFMutableSet^[6] types for use in C.
- Python has a built-in set type, and since Python 3.0, supports non-empty set literals using the curly-bracket syntax, e.g.: { x, y, z }.
- The .NET Framework implements a set in the generic HashSet^[7] class.
- Ruby's library contains a Set class, which implements a set using a hash table.
- OCaml's standard library contains a Set module, which implements a functional set data structure using binary search trees.

- The GHC implementation of Haskell provides a Data.Set module, which implements a functional set data structure using binary search trees.

As noted in the previous section, in languages which do not directly support sets but do support associative arrays, sets can be emulated using associative arrays, by using the elements as keys, and using a dummy value as the values, which are ignored.

Multiset

A variation of the set is the **multiset** or **bag**, which is the same as a set data structure, but allows repeated values. Formally, a multiset can be thought of as an associative array that maps unique elements to positive integers, indicating the multiplicity of the element, although actual implementation may vary.

- C++'s Standard Template Library provides the "multiset" class for the sorted multiset, and SGI's STL provides the "hash_multiset" class, which implements a multiset using a hash table.
- For Java, third-party libraries provide multiset functionality:
 - Apache Commons Collections provides the Bag^[8] and SortedBag interfaces, with implementing classes like HashBag and TreeBag.
 - Google Collections provides the Multiset^[9] interface, with implementing classes like HashMultiset and TreeMultiset.
- Apple provides the NSCountedSet^[4] class as part of Cocoa, and the CFBag^[10] and CFMutableBag^[11] types as part of CoreFoundation.

See also

- Bloom filter
- Disjoint set
- Standard Template Library - contains a C++ set class

References

- [1] Wang, Thomas (1997), *Sorted Linear Hash Table* (<http://www.concentric.net/~Ttwang/tech/sorthash.htm>),
- [2] http://developer.apple.com/documentation/Cocoa/Reference/Foundation/Classes/NSSet_Class/
- [3] http://developer.apple.com/documentation/Cocoa/Reference/Foundation/Classes/NSMutableSet_Class/
- [4] http://developer.apple.com/documentation/Cocoa/Reference/Foundation/Classes/NSCountedSet_Class/
- [5] <http://developer.apple.com/documentation/CoreFoundation/Reference/CFSetRef/>
- [6] <http://developer.apple.com/documentation/CoreFoundation/Reference/CFMutableSetRef/>
- [7] <http://msdn.microsoft.com/en-us/library/bb359438.aspx>
- [8] <http://commons.apache.org/collections/api-release/org/apache/commons/collections/Bag.html>
- [9] <http://google-collections.googlecode.com/svn/trunk/javadoc/com/google/common/collect/Multiset.html>
- [10] <http://developer.apple.com/documentation/CoreFoundation/Reference/CFBagRef/>
- [11] <http://developer.apple.com/documentation/CoreFoundation/Reference/CFMutableBagRef/>

Tree

In computer science, a **tree** is a widely-used data structure that emulates a hierarchical tree structure with a set of linked nodes.

Mathematically, it is not a tree, but an arborescence: an acyclic connected graph where each node has zero or more *children* nodes and at most one *parent* node. Furthermore, the children of each node have a specific order.

Terminology

A node is a structure which may contain a value, a condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more **child nodes**, which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's **parent node** (or *ancestor node*, or superior). A node has at most one parent.

Nodes that do not have any children are called **leaf nodes**. They are also referred to as terminal nodes.

A free tree is a tree that is not rooted.

The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The **depth** of a node is the length of the path to its root (i.e., its *root path*). This is commonly needed in the manipulation of the various self balancing trees, AVL Trees in particular. Conventionally, the value -1 corresponds to a subtree with no nodes, whereas zero corresponds to a subtree with one node.

The topmost node in a tree is called the **root node**. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following **edges** or **links**. (In the formal definition, each such path is also unique). In diagrams, it is typically drawn at the top. In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node.

An **internal node** or **inner node** is any node of a tree that has child nodes and is thus not a leaf node.

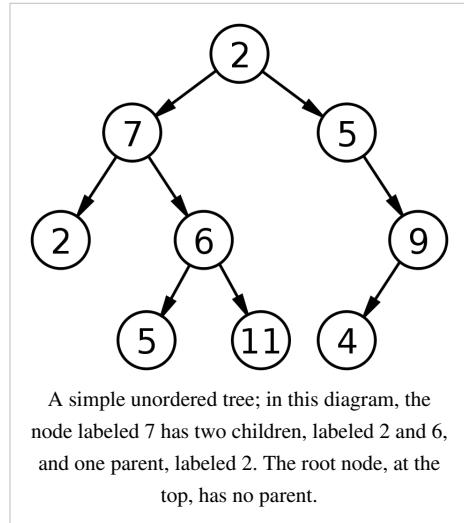
A **subtree** of a tree T is a tree consisting of a node in T and all of its descendants in T . (This is different from the formal definition of subtree used in graph theory.^[1]) The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a **proper subtree** (in analogy to the term proper subset).

Tree representations

There are many different ways to represent trees; common representations represent the nodes as records allocated on the heap (not to be confused with the heap data structure) with pointers to their children, their parents, or both, or as items in an array, with relationships between them determined by their positions in the array (e.g., binary heap).

Trees and graphs

The tree data structure can be generalized to represent directed graphs by allowing cycles to form in the parent-child relationship. Instead of parents and children, we speak of the *sources* and *targets* of the edges of the directed graph. However this is not a common implementation strategy; see graph (data structure).



Relationship with trees in graph theory

In graph theory, a tree is a connected acyclic graph; unless stated otherwise, trees and graphs are undirected. There is no one-to-one correspondence between such trees and trees as data structure. We can take an arbitrary undirected tree, arbitrarily pick one of its vertices as the *root*, make all its edges directed by making them point away from the root node - producing an *arborescence* - and assign an order to all the nodes. The result corresponds to a tree data structure. Picking a different root or different ordering produces a different one.

Traversal methods

Stepping through the items of a tree, by means of the connections between parents and children, is called **walking the tree**, and the action is a **walk** of the tree. Often, an operation might be performed when a pointer arrives at a particular node. A walk in which each parent node is traversed before its children is called a **pre-order walk**; a walk in which the children are traversed before their respective parents are traversed is called a **post-order walk**.

Common operations

- Enumerating all the items
- Enumerating a section of a tree
- Searching for an item
- Adding a new item at a certain position on the tree
- Deleting an item
- Removing a whole section of a tree (called **pruning**)
- Adding a whole section to a tree (called **grafting**)
- Finding the root for any node

Common uses

- Manipulate hierarchical data
- Make information easy to search (see tree traversal)
- Manipulate sorted lists of data
- As a workflow for compositing digital images for visual effects
- Router algorithms

See also

- Tree (graph theory)
- Tree (set theory)
- Tree structure
- Hierarchy (mathematics)

Other trees

- DSW algorithm
- Enfilade
- Left child-right sibling binary tree
- Matrix tree

References

- [1] E.W. Weisstein. ""Subtree."" [\(http://mathworld.wolfram.com/Subtree.html\)](http://mathworld.wolfram.com/Subtree.html). .
- Eric W. Weisstein "Subtree." From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/Subtree.html>
 - Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Section 2.3: Trees, pp. 308–423.
 - 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 10.4: Representing rooted trees, pp. 214–217. Chapters 12–14 (Binary Search Trees, Red-Black Trees, Augmenting Data Structures), pp. 253–320.

External links

- Description (<http://www.nist.gov/dads/HTML/tree.html>) from the Dictionary of Algorithms and Data Structures
- STL-like C++ tree class (<http://tree.phi-sci.com>)
- Description of tree data structures from ideainfo.8m.com (<http://ideainfo.8m.com>)
- flash actionscript 3 opensource implementation of tree and binary tree (<http://www.dpdk.nl/opensource>) — opensource library

Arrays

Array data structure

In computer science, an **array data structure** or simply **array** is a data structure consisting of a collection of *elements* (values or variables), each identified by one or more integer *indices*, stored so that the address of each element can be computed from its index tuple by a simple mathematical formula.^[1] ^[2] For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, ... 2036 (this memory allocation can vary because some computers use other than 4 bytes to store integer type variables); so that the element with index i has address $2000 + 4 \times i$.^[3]

Array structures are the computer analog of the mathematical concepts of vector, matrix, and tensor. Indeed, an array with one or two indices is often called a **vector** or **matrix** structure, respectively. Arrays are often used to implement tables, especially lookup tables; so the word **table** is sometimes used as synonym of array.

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

Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually fixed while the array is in use.

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

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

History

Array structures were used in the first digital computers, when programming was still done in machine language, for data tables, vector and matrix computations, and many other purposes. Von Neumann wrote the first array sorting program (merge sort) in 1945, when the first stored-program computer was still being built.^[4] p. 159 Array indexing was originally done by self-modifying code, and later using index registers and indirect addressing. Some mainframes designed in the 1960s, such as the Burroughs B5000 and its successors, had special instructions for array indexing that included index bounds checking..

Assembly languages generally have no special support for arrays, other than what the machine itself provides. The earliest high-level programming languages, including FORTRAN (1957), COBOL (1960), and ALGOL 60 (1960), had support for multi-dimensional arrays.

Applications

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

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

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

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

Addressing formulas

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

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

One-dimensional arrays

The one dimensional arrays are also known as Single dimension array and is a type of Linear Array. In the one dimension array the data type is followed by the variable name which is further followed by the single subscript i.e. the array can be represented in the row or column wise. It contains a single subscript and that is why it is known as one dimensional array because one subscript can either represent a row or a column.

As an example consider auto int new[10];

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

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

If the valid element indices begin at 0, the constant B is simply the address of the first element of the array. For this reason, the C programming language specifies that array indices always begin at 0; and many programmers will call that element "zeroth" rather than "first".

However, one can choose the index of the first element by an appropriate choice of the base address B . For example, if the array has five elements, indexed 1 through 5, and the base address B is replaced by $B - 30c$, then the indices of those same elements will be 31 to 35. If the numbering does not start at 0, the constant B may not be the address of any element.

Multidimensional arrays

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

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

$$B + c_1 \cdot i_1 + c_2 \cdot i_2 + \dots + c_k \cdot i_k$$

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

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

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

Dope vectors

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

Compact layouts

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

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

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

In the row-major order layout (adopted by C for statically declared arrays), the elements of each row are stored in consecutive positions:

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

In Column-major order (traditionally used by Fortran), the elements of each column are consecutive in memory:

1	4	7	2	5	8	3	6	9
---	---	---	---	---	---	---	---	---

For arrays with three or more indices, "row major order" puts in consecutive positions any two elements whose index tuples differ only by one in the *last* index. "Column major order" is analogous with respect to the *first* index.

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

product $A \cdot B$ of two matrices, it would be best to have A stored in row-major order, and B in column-major order.

Array resizing

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

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

Non-linear formulas

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

Efficiency

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

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

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

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

Efficiency comparison with other data structures

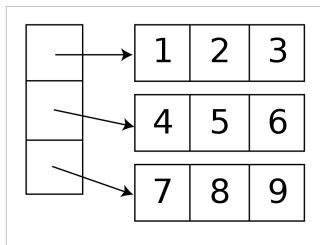
	Array	Dynamic array	Linked list	Balanced tree
Indexing	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Insertion/deletion at end	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
Insertion/deletion in middle	N/A	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
Wasted space (average)	0	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

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

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

Balanced trees require $O(\log n)$ time for indexed access, but also permit inserting or deleting elements in $O(\log n)$ time,^[5] whereas growable arrays require linear ($\Theta(n)$) time to insert or delete elements at an arbitrary position.

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



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

Meaning of dimension

In computer science, the "dimension" of an array is that its domain, namely the number of indices needed to select an element; whereas in mathematics it usually refers to the dimension of the set of all matrices, that is, the number of elements in the array. Thus, an array with 5 rows and 4 columns (hence 20 elements) is said to be "two-dimensional" in computing contexts, but "20-dimensional" in mathematics.

See also

Applications

- Register file
- Branch table
- Lookup table
- Bitmap
- Bit array
- Raster graphics
- String (computer science)

- Suffix array
- Double-ended queue (deque)
- Heap
- Queue
- Stack
- Heap
- Hash table

Some array-based algorithms

- Counting sort
- Binary search algorithm

Technical topics

- Array slicing
- Content-addressable memory
- Index register
- Parallel array
- Offset (computer science)
- Random access
- Row-major order
- Stride of an array

Slight variants

- Dynamic array
- Variable-length array

Alternative structures for abstract tables

- Fusion tree
- Hashed array tree
- Hash array mapped trie
- Iliffe vector, used to implement multidimensional arrays
- Judy array, a kind of hash table
- Linked list
- Sparse array
- VList, a list-array hybrid

Other

- Data structure

External links

- NIST's Dictionary of Algorithms and Data Structures: Array ^[6]

References

- [1] Paul E. Black, "array", in *Dictionary of Algorithms and Data Structures*, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 26 August 2008 (accessed 10 September 2008). (<http://www.nist.gov/dads/HTML/array.html>)
- [2] Donald Knuth, *The Art of Computer Programming*, vol. 1. Addison-Wesley
- [3] David R. Richardson (2002), *The Book on Data Structures*. iUniverse, 112 pages. ISBN 0595240399, 9780595240395.
- [4] Donald Knuth, *The Art of Computer Programming*, vol. 3. Addison-Wesley
- [5] Counted B-Tree (<http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html>)
- [6] <http://www.nist.gov/dads/HTML/array.html>

Row-major order

In computing, **row-major order** and **column-major order** describe methods for storing multidimensional arrays in linear memory. Following standard matrix notation, rows are identified by the first index of a two-dimensional array and columns by the second index. Array layout is critical for correctly passing arrays between programs written in different languages. It is also important for performance when traversing an array because accessing array elements that are contiguous in memory is usually faster than accessing elements which are not, due to caching.

Row-major order is used in C; column-major order is used in Fortran and Matlab.

Row-major order

In row-major storage, a multidimensional array in linear memory is accessed such that rows are stored one after the other. It is the approach used by the C programming language as well as many other languages, with the notable exceptions of Fortran and MATLAB.

When using row-major order, the difference between addresses of array cells in increasing rows is larger than addresses of cells in increasing columns. For example, consider this 2×3 array:

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

Declaring this array in C as

```
int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

would find the array laid-out in linear memory as:

```
1 2 3 4 5 6
```

The difference in offset from one column to the next is 1 and from one row to the next is 3. The linear offset from the beginning of the array to any given element $A[\text{row}][\text{column}]$ can then be computed as:

$$\text{offset} = \text{row} * \text{NUMCOLUMNS} + \text{column}$$

Where NUMCOLUMNS is the number of columns in the array.

The above formula only works when using the C convention of labeling the first element 0. In other words, row 1, column 2 in matrix A, would be represented as $A[0][1]$

Note that this technique generalizes, so a $2 \times 2 \times 2$ array looks like:

```
int A[2][2][2] = {{{1,2}, {3,4}}, {{5,6}, {7,8}}};
```

and the array would be laid-out in linear memory as:

```
1 2 3 4 5 6 7 8
```

Column-major order

Column-major order is a similar method of flattening arrays onto linear memory, but the columns are listed in sequence. The programming languages Fortran, MATLAB,^[1] Octave and R^[2] use column-major ordering. The array

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

if stored in linear memory with column-major order would look like the following:

```
1 4 2 5 3 6
```

With columns listed first. The memory offset could then be computed as:

$$\text{offset} = \text{row} + \text{column} * \text{NUMROWS}$$

where NUMROWS represents the number of rows in the array—in this case, 2.

Treating a row-major array as a column-major array is the same as transposing it. Because performing a transpose requires data movement, and is quite difficult to do in-place for non-square matrices, such transpositions are rarely performed explicitly. For example, software libraries for linear algebra, such as the BLAS, typically provide options to specify that certain matrices are to be interpreted in transposed order to avoid the necessity of data movement.

Generalization to higher dimensions

It is possible to generalize both of these concepts to arrays with greater than two dimensions. For higher-dimensional arrays, the ordering determines which dimensions of the array are more consecutive in memory. Any of the dimensions could be consecutive, just as a two-dimensional array could be listed column-first or row-first. The difference in offset between listings of that dimension would then be determined by a product of other dimensions. It is uncommon, however, to have any variation except ordering dimensions first to last or last to first. These two variations correspond to row-major and column-major, respectively.

More explicitly, consider a d -dimensional array with dimensions N_k ($k=1\dots d$). A given element of this array is specified by a tuple (n_1, n_2, \dots, n_d) of d (zero-based) indices $n_k \in [0, N_k - 1]$. In row-major order, the memory-offset of this element is given by:

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\dots + N_2 n_1) \dots)) = \sum_{k=1}^d \left(\prod_{\ell=k+1}^d N_\ell \right) n_k$$

In column-major order, the memory-offset of this element is given by:

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\dots + N_{d-1} n_d) \dots)) = \sum_{k=1}^d \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

Note that the difference between row-major and column-major order is simply that the order of the dimensions is reversed. Equivalently, in row-major order the rightmost indices vary faster as one steps through consecutive memory locations, while in column-major order the leftmost indices vary faster.

See also

- Matrix representation
- Vectorization (mathematics), the equivalent of turning a matrix into the corresponding column-major vector.

References

- Donald E. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, third edition, section 2.2.6 (Addison-Wesley: New York, 1997).
- [1] Matlab documentation, mxCalcSingleSubscript function (<http://www.mathworks.com/access/helpdesk/help/techdoc/apiref/mxcalcsingle subscript.html>) (retrieved from Mathworks.com, March 2010).
- [2] *An Introduction to R*, Section 5.1: Arrays (<http://cran.r-project.org/doc/manuals/R-intro.html#Arrays>) (retrieved March 2010).

Dope vector

In computer programming, a **dope vector** is a data structure used to hold information about a data object^[1], e.g. an array, especially its memory layout.

A dope vector typically contains information about the type of array element, rank of an array, the extents of an array, and the stride of an array as well as a pointer to the block in memory containing the array elements.

It is often used in compilers to pass entire arrays between procedures in a high level language like Fortran.

- [1] Pratt T. and M. Zelkowitz, Programming Languages: Design and Implementation (Third Edition), Prentice Hall, Upper Saddle River, NJ, (1996) pp 114

Iliffe vector

In computer programming, an **Iliffe vector** (also known as a display) is a data structure used to implement multi-dimensional arrays. Named after John K. Iliffe, an Iliffe vector for an n dimensional array (where $n > 2$) consists of a vector (or 1 dimensional array) of pointers to an $n-1$ dimensional array. They are often used to avoid the need for expensive multiplication operations when performing address calculation on an array element. They can also be used to implement triangular arrays, or other kinds of irregularly shaped arrays.

Their disadvantages include the need for multiple chained pointer indirections to access an element, and the extra work required to determine the next row in an n -dimensional array to allow an optimising compiler to prefetch it. Both of these are a source of delays on systems where the CPU is significantly faster than main memory.

The Iliffe vector for a 2-dimensional array is simply a vector of pointers to vectors of data, i.e., the Iliffe vector represents the columns of an array where each column element is a pointer to a row vector.

Multidimensional arrays in languages such as Java and Atlas Autocode are implemented as Iliffe vectors.

Iliffe vectors are contrasted with dope vectors in languages such as Fortran, which contain the stride factors and offset values for the subscripts in each dimension.

References

- John K. Iliffe (1961). "The Use of The Genie System in Numerical Calculations". *Annual Review in Automatic Programming* 2: 1–28. doi:10.1016/S0066-4138(61)80002-5., see page 25

Dynamic array

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

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

Bounded-size dynamic arrays and capacity

The simplest dynamic array is constructed by allocating a fixed-size array and then dividing it into two parts: the first stores the elements of the dynamic array and the second is reserved, or unused. We can then add or remove elements at the end of the dynamic array in constant time by using the reserved space, until this space is completely consumed. The number of elements used by the dynamic array contents is its *logical size* or *size*, while the size of the underlying array is called the dynamic array's *capacity*, which is the maximum possible logical size.

In applications where the logical size is bounded, this data structure suffices. Resizing the underlying array is an expensive operation, typically involving copying the entire contents of the array.

Geometric expansion and amortized cost

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

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

As n elements are inserted, the capacities form a geometric progression. Expanding the array by any constant proportion ensures that inserting n elements takes $O(n)$ time overall, meaning that each insertion takes amortized constant time. The value of this proportion a leads to a time-space tradeoff: the average time per insertion operation is about $a/(a-1)$, while the number of wasted cells is bounded above by $(a-1)n$. The choice of a depends on the library or application: $a=3/2$ ^[1] and $a=2$ are commonly-used.

Many dynamic arrays also deallocate some of the underlying storage if its size drops below a certain threshold, such as 30% of the capacity.

Dynamic arrays are a common example when teaching amortized analysis.

Performance

	Linked list	Array	Dynamic array
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Insertion/deletion at end	$\Theta(1)$	N/A	$\Theta(1)$
Insertion/deletion in middle	$\Theta(1)$	N/A	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$

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

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

Dynamic arrays benefit from many of the advantages of arrays, including good locality of reference and data cache utilization, compactness (low memory use), and random access. They usually have only a small fixed additional overhead for storing information about the size and capacity. This makes dynamic arrays an attractive tool for building cache-friendly data structures.

Compared to linked lists, dynamic arrays have faster indexing (constant time versus linear time) and typically faster iteration due to improved locality of reference; however, dynamic arrays require linear time to insert or delete at an arbitrary location, since all following elements must be moved, while linked lists can do this in constant time. This disadvantage is mitigated by the gap buffer and *tiered vector* variants discussed under *Variants* below. Also, in a highly-fragmented memory region, it may be expensive or impossible to find contiguous space for a large dynamic array, whereas linked lists do not require the whole data structure to be stored contiguously.

Variants

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

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

Hashed Array Tree (HAT) is a dynamic array algorithm invented by Sitarski in 1996.^[3] Hashed Array Tree wastes order $n^{1/2}$ amount of storage space, where n is the number of elements in the array. The algorithm has $O(1)$ amortized performance when appending a series of objects to the end of a Hashed Array Tree.

In a 1999 paper^[4], Brodnik et al. describe a tiered dynamic array data structure, which wastes only $n^{1/2}$ space for n elements at any point in time, and they prove a lower bound showing that any dynamic array must waste this much space if the operations are to remain amortized constant time. Additionally, they present a variant where growing and shrinking the buffer has not only amortized but worst-case constant time.

Bagwell (2002)^[5] presented the VList algorithm, which can be adapted to implement a dynamic array.

Language support

C++'s std::vector is an implementation of dynamic arrays, as are the ArrayList^[6] classes supplied with the Java API and the .NET Framework. The generic List<> class supplied with version 2.0 of the .NET Framework is also implemented with dynamic arrays. Delphi and D implement dynamic arrays at the language's core. Many scripting languages such as Perl and PHP offer dynamic arrays as a built-in primitive data type.

See also

- Dynamic memory allocation

References

- [1] See, for example, the source code of java.util.ArrayList class from OpenJDK 6 (<http://hg.openjdk.java.net/jdk6/jdk6/file/e0e25ac28560/src/share/classes/java/util/ArrayList.java>).
- [2] Goodrich, Michael T.; Kloss II, John G. (1999), "Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences" (<http://citeseer.ist.psu.edu/519744.html>), *Workshop on Algorithms and Data Structures* **1663**: pp. 205–216, doi:10.1007/3-540-48447-7_21,
- [3] Sitarski, Edward (September 1996), *Algorithm Alley* (<http://www.ddj.com/architect/184409965?pgno=5>), "HATs: Hashed array trees", *Dr. Dobb's Journal* **21** (11),
- [4] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (Technical Report CS-99-09), *Resizable Arrays in Optimal Time and Space* (<http://www.cs.uwaterloo.ca/research/tr/1999/09/CS-99-09.pdf>), Department of Computer Science, University of Waterloo,
- [5] Bagwell, Phil (2002), *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays* (<http://citeseer.ist.psu.edu/bagwell02fast.html>), EPFL,
- [6] Javadoc on ArrayList
 - 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 17.4: Dynamic tables, pp.416–425.

External links

- NIST Dictionary of Algorithms and Data Structures: Dynamic array (<http://www.nist.gov/dads/HTML/dynamicarray.html>)
- VPOOL (<http://www.bsdua.org/libbsdua.html#vpool>) - C language implementation of dynamic array.
- CollectionSpy (<http://www.collectionspy.com>) — A Java profiler with explicit support for debugging ArrayList- and Vector-related issues.

Hashed array tree

In computer science, a **hashed array tree** (HAT) is a dynamic array algorithm invented by Edward Sitarski in 1996.

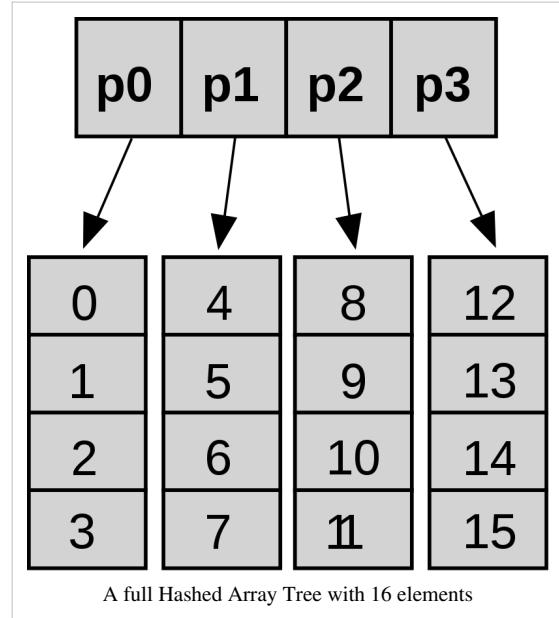
^[1] Whereas simple dynamic array data structures based on geometric expansion waste linear ($\Omega(n)$) space, where n is the number of elements in the array, hashed array trees waste only order $n^{1/2}$ storage space. It can perform indexing in constant ($O(1)$) time, but indexing is not as quick in practice as for simple dynamic arrays. The algorithm has $O(1)$ amortized performance when appending a series of objects to the end of a hashed array tree. Contrary to its name, it does not use hash functions.

Definitions

As defined by Sitarski, a hashed array tree has a top-level directory containing a power of two number of leaf arrays. All leaf arrays are the same size as the top-level directory. This structure superficially resembles a hash table with array-based collision chains, which is the basis for the name *hashed array tree*. A full hashed array tree can hold m^2 elements, where m is the size of the top-level directory.^[1]

Expansions and size reductions

When a hashed array tree is full, its directory and leaves must be restructured to twice its prior size to accommodate additional append operations. There are multiple alternatives for reducing size: when a Hashed Array Tree is one eighth full, it can be restructured to a smaller, half-full hashed array tree; another option is only freeing unused leaf arrays.



Related data structures

Brodnik et al. ^[2] presented a dynamic array algorithm with a similar space wastage profile to hashed array trees. Brodnik's implementation retains previously allocated leaf arrays, with a more complicated address calculation function as compared to hashed array trees.

See also

- VList

References

- [1] Sitarski, Edward (September 1996), *Algorithm Alley* (<http://www.ddj.com/architect/184409965?pgno=5>), "HATs: Hashed array trees", *Dr. Dobb's Journal* **21** (11),
- [2] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (Technical Report CS-99-09), *Resizable Arrays in Optimal Time and Space* (<http://www.cs.uwaterloo.ca/research/tr/1999/09/CS-99-09.pdf>), Department of Computer Science, University of Waterloo,

Gap buffer

In computer science, a **gap buffer** is a dynamic array that allows efficient insertion and deletion operations clustered near the same location. Gap buffers are especially common in text editors, where most changes to the text occur at or near the current location of the cursor. The text is stored in a large buffer in two contiguous segments, with a gap between them for inserting new text. Moving the cursor involves copying text from one side of the gap to the other (sometimes copying is delayed until the next operation that changes the text). Insertion adds new text at the end of the first segment. Deletion increases the size of the gap.

The advantage of using a gap buffer over more sophisticated data structures (such as linked lists) is that the text is represented simply as two literal strings, which take very little extra space and which can be searched and displayed very quickly.

The disadvantage is that operations at different locations in the text and ones that fill the gap (requiring a new gap to be created) require re-copying most of the text, which is especially inefficient for large files. The use of gap buffers is based on the assumption that such recopying occurs rarely enough that its cost can be amortized over the more common cheap operations.

A gap buffer is used in most Emacs editors.

Example

Below are some examples of operations with buffer gaps. The gap is represented pictorially by the empty space between the square brackets. This representation is a bit misleading: in a typical implementation, the endpoints of the gap are tracked using pointers or array indices, and the contents of the gap are ignored; this allows, for example, deletions to be done by adjusting a pointer without changing the text in the buffer. It is a common programming practice to use a semi-open interval for the gap pointers, i.e. the start-of-gap points to the invalid character following the last character in the first buffer, and the end-of-gap points to the first valid character in the second buffer (or equivalently, the pointers are considered to point "between" characters).

Initial state:

```
This is the way [      ]out.
```

User inserts some new text:

```
This is the way the world started [      ]out.
```

User moves the cursor before "started"; system moves "started" from the first buffer to the second buffer.

```
This is the way the world [      ]started out.
```

User adds text filling the gap; system creates new gap:

```
This is the way the world as we know it [      ]started  
out.
```

See also

- Dynamic array, the special case of a gap buffer where the gap is always at the end
- Zipper (data structure), conceptually a generalization of the gap buffer.
- Linked list
- Circular buffer
- Rope (computer science)

External references

- Overview and implementation in .NET/C# ^[1]
- Brief overview and sample C++ code ^[2]
- Implementation of a cyclic sorted gap buffer in .NET/C# ^[3]
- Use of gap buffer in early editor. ^[4] (First written somewhere between 1969 and 1971)

References

[1] <http://www.codeproject.com/KB/recipes/GenericGapBuffer.aspx>

[2] <http://www.lazyhacker.com/gapbuffer/gapbuffer.htm>

[3] <http://www.codeproject.com/KB/recipes/SplitArrayDictionary.aspx>

[4] <http://history.dcs.ed.ac.uk/archive/apps/ecce/hmd/e915.imp.html>

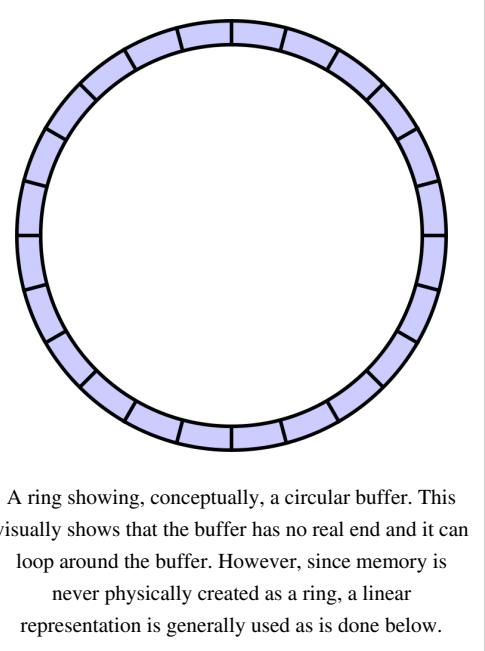
Circular buffer

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

Uses

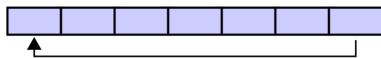
An example that could possibly use an overwriting circular buffer is with multimedia. If the buffer is used as the bounded buffer in the producer-consumer problem then it is probably desired for the producer (e.g., an audio generator) to overwrite old data if the consumer (e.g., the sound card) is unable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

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

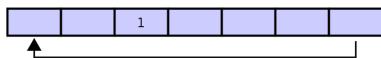


How it works

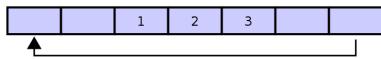
A circular buffer first starts empty and of some predefined length. For example, this is a 7-element buffer:



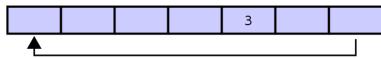
Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a circular buffer):



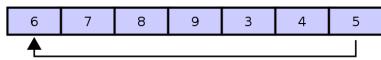
Then assume that two more elements are added — 2 & 3 — which get appended after the 1:



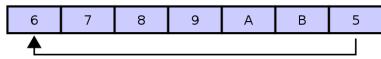
If two elements are then removed from the buffer then they come from the end. The two elements removed, in this case, are 1 & 2 leaving the buffer with just a 3:



If the buffer has 7 elements then it is completely full:

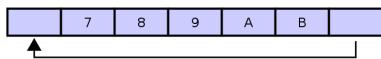


A consequence of the circular buffer is that when it is full and a subsequent write is performed, then it starts overwriting the oldest data. In this case, two more elements — A & B — are added and they *overwrite* the 3 & 4:



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

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



Circular buffer mechanics

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

Start / End Pointers

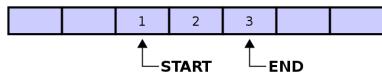
Generally, a circular buffer requires three pointers:

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

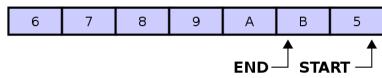
Alternatively, a fixed-length buffer with two integers to keep track of indices can be used in languages that do not have pointers.

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

This image shows a partially-full buffer:



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

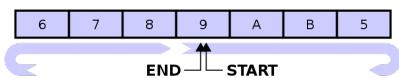


What to note about the second one is that after each element is overwritten then the start pointer is incremented as well.

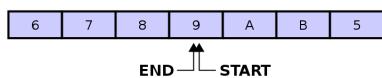
Difficulties

Full / Empty Buffer Distinction

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



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



To solve this problem there are a number of solutions:

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

Always Keep One Slot Open

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

The advantages are:

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

The disadvantages are:

- You can never use the entire buffer.
- If you cannot read over the buffer border, you get a lot of situations where you can only read one element at once.

An example implementation in C:

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

#define BUFFER_SIZE 25

void buffer_char(char c);
char unbuffer_char(void);
```

```
//a buffer with BUFFER_SIZE slots
char circular_buffer[BUFFER_SIZE];

//integers to index circular_buffer
int start, end;

int main(int argc, char *argv[])
{
    char sentence[] = {"The quick brown dog jumped over the lazy
fox."};
    int i;

    //add sentence into the buffer
    for (i = 0; i < strlen(sentence); i++) {
        buffer_char(sentence[i]);
    }

    //read the contents of the buffer
    while(start != end) {
        printf("%c", unbuffer_char());
    }

    printf("\n");
    return 0;
}

void buffer_char(char c)
{
    //Use modulo as a trick to wrap around the end of the buffer
    back to the beginning
    if ((end + 1) % BUFFER_SIZE != start) {
        circular_buffer[end] = c;
        end = (end + 1) % BUFFER_SIZE;
    }
    //otherwise, the buffer is full; don't do anything
}

char unbuffer_char(void)
{
    char temp = circular_buffer[start];
    start = (start + 1) % BUFFER_SIZE;
    return (temp);
}
```

Use a Fill Count

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

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

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

The advantages are:

- Simple.
- Needs only one additional variable.

The disadvantage is:

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

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

The advantages are:

- Simple.
- No additional variables.

The disadvantage is:

- Additional overhead when generating the write pointer.

Read / Write Counts

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

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

The advantage is:

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

The disadvantage is:

- You need two additional variables.

Absolute indices

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

The advantage is:

- No extra variables are needed.

The disadvantages are:

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

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

Multiple Read Pointers

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

Chunked Buffer

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

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

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

Optimization

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

Exemplary POSIX Implementation

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

#define report_exceptional_condition() abort()

struct ring_buffer
{
    void *address;

    unsigned long count_bytes;
```

```
unsigned long write_offset_bytes;
unsigned long read_offset_bytes;
};

//Warning order should be at least 12 for Linux

void
ring_buffer_create (struct ring_buffer *buffer, unsigned long order)
{
    char path[] = "/dev/shm/ring-buffer-XXXXXX";
    int file_descriptor;
    void *address;
    int status;

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

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

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

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

    buffer->address = mmap (NULL, buffer->count_bytes << 1, PROT_NONE,
                           MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);

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

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

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

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

    if (address != buffer->address + buffer->count_bytes)
```

```
    report_exceptional_condition ();

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

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

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

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

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

void *
ring_buffer_read_address (struct ring_buffer *buffer)
{
    return buffer->address + buffer->read_offset_bytes;
}

void
ring_buffer_read_advance (struct ring_buffer *buffer,
                           unsigned long count_bytes)
{
    buffer->read_offset_bytes += count_bytes;

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

```
    }

}

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

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

void
ring_buffer_clear (struct ring_buffer *buffer)
{
    buffer->write_offset_bytes = 0;
    buffer->read_offset_bytes = 0;
}
```

External links

- <http://c2.com/cgi/wiki?CircularBuffer>
- The Bip Buffer - The Circular Buffer with a Twist ^[1]
- Boost: Templated Circular Buffer Container ^[2]
- Circular Buffer - A LabVIEW Implementation ^[3]
- <http://www.dspguide.com/ch28/2.htm>

References

- [1] <http://www.codeproject.com/KB/IP/bipbuffer.aspx>
[2] http://www.boost.org/doc/libs/1_39_0/libs/circular_buffer/doc/circular_buffer.html
[3] <http://zone.ni.com/devzone/cda/tut/p/id/7188>

Sparse array

In computer science, a **sparse array** is an array in which most of the elements have the same value (known as the default value—usually 0 or null).

A naive implementation of an array may allocate space for the entire array, but in the case where there are few non-default values, this implementation is inefficient. Typically the algorithm used instead of an ordinary array is determined by other known features (or statistical features) of the array, for instance if the sparsity is known in advance, or if the elements are arranged according to some function (e.g. occur in blocks).

As an example, a spreadsheet containing 100×100 mostly empty cells might be more efficiently stored as a linked list rather than an array containing ten thousand array elements.

A heap memory allocator inside a program might choose to store regions of blank space inside a linked list rather than storing all of the allocated regions in, say a bit array.

See also

- Sparse matrix

External links

- Boost sparse vector class ^[1]
- Rodolphe Buda, "Two Dimensional Aggregation Procedure: An Alternative to the Matrix Algebraic Algorithm", *Computational Economics*, 31(4), May, pp.397–408, 2008. ^[2]

References

[1] http://boost.org/libs/numeric/ublas/doc/vector_sparse.htm

[2] <http://portal.acm.org/citation.cfm?id=1363086&jmp=cit&coll=GUIDE&dl=GUIDE>

Bit field

A **bit field** is a common idiom used in computer programming to compactly store a value as a short series of bits. A bit field is most commonly used to represent integral types of known, fixed bit-width. Perhaps the most well known usage of bit-fields is to represent single bit flags, with each flag stored in a separate bit.

A bit field is distinguished from a bit array in that the latter is used to store a large set of bits indexed by integers and is often wider than any integral type supported by the language. Bit fields, on the other hand, typically fit within a machine word, and the denotation of bits is independent of their numerical index.

Examples

Example implementation in C:

```
#define PREFERENCE_LIKES_ICE_CREAM (1 << 0) /* 0x01 */
#define PREFERENCE_PLAYS_GOLF      (1 << 1) /* 0x02 */
#define PREFERENCE_WATCHES_TV       (1 << 2) /* 0x04 */
#define PREFERENCE_READS_BOOKS     (1 << 3) /* 0x08 */

unsigned char preference;

void set_preference(unsigned char flag) {
    preference |= flag;
}

void reset_preference(unsigned char flag) {
    preference &= ~flag;
}

unsigned char get_preference(unsigned char flag) {
    return (preference & flag) != 0;
}
```

Instead of using hardcoded numerical representations for the powers of two (0x08), the use of the bit shift operator ($1 \ll 3$) with an incrementing shift operand is recommended for easier readability.

Kernighan and Ritchie's book, *The C Programming Language*, describes a method for defining and accessing fields directly. Using this method, bitwise operators are not needed as bit members can be accessed the same as struct members. An example using a struct follows:

```
struct preferences {
    unsigned int likes_ice_cream : 1;
    unsigned int plays_golf : 1;
    unsigned int watches_tv : 1;
    unsigned int reads_books : 1;
};

struct preferences fred;

fred.likes_ice_cream = 1;
```

```

fred.plays_golf = 1;
fred.watches_tv = 1;
fred.reads_books = 0;

if (fred.likes_ice_cream == 1)
    /* ... */

```

However, bit members in structs have potential practical drawbacks. First, the ordering of bits in memory is cpu dependent and memory padding rules can vary between compilers. In addition, less well optimized compilers sometimes generate poor quality code for reading and writing bit members and there are potentially thread safety issues relating to bit fields because most machines cannot manipulate arbitrary sets of bits in memory, but must instead load and store whole words;^[1] e.g. the following would not be thread-safe, in spite of the use of a mutex for each member:

```

struct foo {
    int flag : 1;
    int counter : 15;
};

struct foo my_foo;

/* ... */

/* in thread 1 */

pthread_mutex_lock(&my_mutex_for_flag);
my_foo.flag = !my_foo.flag;
pthread_mutex_unlock(&my_mutex_for_flag);

/* in thread 2 */

pthread_mutex_lock(&my_mutex_for_counter);
++my_foo.counter;
pthread_mutex_unlock(&my_mutex_for_counter);

```

The root of the problem is that on most machines it is impossible to load and store flag and counter separately, when both are stored in the same word. In order for this to be thread-safe you should use a single mutex to protect both flag and counter, instead of two.

Homogeneity and mathematical structure

In the examples above, the individual power-of-two values are declared as macros (ending up being the machine word). Since bitfields are by essence destined to be combined with the bitwise OR operator, such code

```

enum preference {
    preference_likes_ice_cream = 1<<0,
    preference_plays_golf = 1<<1,
    preference_likes_watching_tv = 1<<2,
    preference_reads_books = 1<<3,
};

```

fails the type-safety principle when combining `preference_plays_golf|preference_likes_ice_cream`, that does not belong to the enumeration.

Quantities defined as the combination of bits are actually elements of the elementary abelian group $(\mathbb{Z}/2\mathbb{Z})^n$; and the relation defined as $a \leq b$ when $a \& b = a$ only creates a partial order ($1011b$ is greater than $1010b$ but $1011b$ and $1101b$ are not comparable).

This remark is of interest when designing variable importance debug channels (from `informative' to `fatal'); the regular integer comparison cannot be used to filter out part of the messages.

Nevertheless, a bit field can be safely and elegantly implemented using a bit array where the **bit indices** for each flag are values of an enumerated type (like the `EnumSet` class in Java); this avoids the dangers of direct bitwise manipulations.

See also

- Bitboard, used in chess and similar games.
- Bit array
- Flag word

External links

- Explanation from a book ^[2]
- Description from another wiki ^[3]
- Use case in a C++ guide ^[4]
- bit library ^[5]

References

- [1] Threads Cannot Be Implemented As a Library (<http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf>)
- [2] http://publications.gbdirect.co.uk/c_book/chapter6/bitfields.html
- [3] <http://c2.com/cgi/wiki?BitField>
- [4] <http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=131>
- [5] <http://libbit.sourceforge.net/>

Bit array

A **bit array** (also known as a **bitmap**, a **bitset**, or a **bitstring**) is an array data structure that compactly stores individual bits (boolean values). It implements a simple set data structure storing a subset of $\{1, 2, \dots, n\}$ and is effective at exploiting bit-level parallelism in hardware to perform operations quickly. A typical bit array stores kw bits, where w is the number of bits in the unit of storage, such as a byte or word, and k is some nonnegative integer. If w does not divide the number of bits to be stored, some space is wasted due to internal fragmentation.

Basic operations

Although most machines are not able to address individual bits in memory, nor have instructions to manipulate single bits, each bit in a word can be singled out and manipulated using bitwise operations. In particular:

- OR can be used to set a bit to one: $11101010 \text{ OR } 00000100 = 11101110$
- AND can be used to set a bit to zero: $11101010 \text{ AND } 11111101 = 11101000$
- AND together with zero-testing can be used to determine if a bit is set:

$$11101010 \text{ AND } 00000001 = 00000000 = 0$$

$$11101010 \text{ AND } 00000010 = 00000010 \neq 0$$

- XOR can be used to invert or toggle a bit:

$$11101010 \text{ XOR } 00000100 = 11101110$$

$$11101110 \text{ XOR } 00000100 = 11101010$$

To obtain the bit mask needed for these operations, we can use a bit shift operator to shift the number 1 to the left by the appropriate number of places, as well as bitwise negation if necessary.

We can view a bit array as a subset of $\{1, 2, \dots, n\}$, where a 1 bit indicates a number in the set and a 0 bit a number not in the set. This set data structure uses about n/w words of space, where w is the number of bits in each machine word. Whether the least significant bit or the most significant bit indicates the smallest-index number is largely irrelevant, but the former tends to be preferred.

Given two bit arrays of the same size representing sets, we can compute their union, intersection, and set-theoretic difference using n/w simple bit operations each ($2n/w$ for difference), as well as the complement of either:

```
for i from 0 to n/w-1
    complement_a[i] := not a[i]
    union[i]        := a[i] or b[i]
    intersection[i] := a[i] and b[i]
    difference[i]   := a[i] and (not b[i])
```

If we wish to iterate through the bits of a bit array, we can do this efficiently using a doubly-nested loop that loops through each word, one at a time. Only n/w memory accesses are required:

```
for i from 0 to n/w-1
    index := 0      // if needed
    word := a[i]
    for b from 0 to w-1
        value := word and 1 ≠ 0
        word := word shift right 1
        // do something with value
        index := index + 1 // if needed
```

Both of these code samples exhibit ideal locality of reference, and so get a large performance boost from a data cache. If a cache line is k words, only about n/wk cache misses will occur.

More complex operations

Population / Hamming weight

If we wish to find the number of 1 bits in a bit array, sometimes called the *population function*, or Hamming weight, there are efficient branch-free algorithms that can compute the number of bits in a word using a series of simple bit operations. We simply run such an algorithm on each word and keep a running total. Counting zeros is similar. See the Hamming weight article for examples of an efficient implementation.

Sorting

Similarly, sorting a bit array is trivial to do in $O(n)$ time using counting sort — we count the number of ones k , fill the last k/w words with ones, set only the low $k \bmod w$ bits of the next word, and set the rest to zero.

Inversion

Vertical flipping of a one-bit-per-pixel image, or some FFT algorithms, require to flip the bits of individual words (so $b_{31} b_{30} \dots b_0$ becomes $b_0 \dots b_{30} b_{31}$). When this operation is not available on the processor, it's still possible to proceed by successive passes, in this example on 32 bits:

```
exchange two 16bit halfwords
exchange bytes by pairs (0xddccbaa -> 0xccddabb)
...
swap bits by pairs
swap bits (b_{31} b_{30} ... b_1 b_0 -> b_{30} b_{31} ... b_0 b_1)
```

The last operation can be written $((x \& 0x55555555) << 1) | (x \& 0xaaaaaaaa) >> 1)$.

Find first one

Bit arrays are useful in some contexts as priority queues. The goal in such a context is to identify the one bit of smallest index, that is the least significant bit has the highest priority. Some machines (including normal x86 PCs) have a *find first one* or *find first zero* operation that does this on a single word. With this, the operation is obvious: find the first nonzero word and run *find first one* on it, or *find first zero* on its complement. On machines that do not feature this operation, the operation can be reproduced using sequences of bit operations.

On machines that use two's complement arithmetic, which includes all conventional CPUs, the *find first one* function can be performed quickly by anding a word with its two's complement, that is performing $(w \text{ AND } -w)$ results in a word with only the rightmost bit set of the bits that were set before the operation. For instance, if the original value were 6 (110), after this operation the result would be 2 (010).

Find-first-zero operation, starting from the leftmost bit, is equivalent to computing the base 2 logarithm. This helps computing rapidly such quantities on machines that comprise it in their basic instruction set (ex: clz in MIPS).

Compression

Large bit arrays tend to have long streams of zeroes or ones. This phenomenon wastes storage and processing time. Run-length encoding is commonly used to compress these long streams. However, by compressing bit arrays too aggressively we run the risk of losing the benefits due to bit-level parallelism (vectorization). Thus, instead of compressing bit arrays as streams of bits, we might compress them as streams bytes or words (see Bitmap index (compression)).

Examples:

- compressedbitset^[1]: WAH Compressed BitSet for Java
- javaewah^[2]: A compressed alternative to the Java BitSet class (using Enhanced WAH)
- CONCISE^[3]: COmpressed 'N' Composable Integer Set, another bitmap compression scheme for Java

Advantages and disadvantages

Bit arrays, despite their simplicity, have a number of marked advantages over other data structures for the same problems:

- They are extremely compact; few other data structures can store n independent pieces of data in n/w words.
- They allow small arrays of bits to be stored and manipulated in the register set for long periods of time with no memory accesses.
- Because of their ability to exploit bit-level parallelism, limit memory access, and maximally use the data cache, they often outperform many other data structures on practical data sets, even those that are more asymptotically efficient.

However, bit arrays aren't the solution to everything. In particular:

- Without compression, they are wasteful set data structures for sparse sets (those with few elements compared to their range) in both time and space. For such applications, compressed bit arrays, Judy arrays, tries, or even Bloom filters should be considered instead.
- Accessing individual elements can be expensive and difficult to express in some languages. If random access is more common than sequential and the array is relatively small, a byte array may be preferable on a machine with byte addressing. A word array, however, is probably not justified due to the huge space overhead and additional cache misses it causes, unless the machine only has word addressing.

Applications

Because of their compactness, bit arrays have a number of applications in areas where space or efficiency is at a premium. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values.

We mentioned above that bit arrays are used for priority queues, where the bit at index k is set if and only if k is in the queue; this data structure is used, for example, by the Linux kernel, and benefits strongly from a find-first-zero operation in hardware.

Bit arrays can be used for the allocation of memory pages, inodes, disk sectors, etc. In such cases, the term *bitmap* may be used. However, this term is frequently used to refer to raster images, which may use multiple bits per pixel.

Another application of bit arrays is the Bloom filter, a probabilistic set data structure that can store large sets in a small space in exchange for a small probability of error. It is also possible to build probabilistic hash tables based on bit arrays that accept either false positives or false negatives.

Bit arrays and the operations on them are also important for constructing succinct data structures, which use close to the minimum possible space. In this context, operations like finding the n th 1 bit or counting the number of 1 bits up to a certain position become important.

Bit arrays are also a useful abstraction for examining streams of compressed data, which often contain elements that occupy portions of bytes or are not byte-aligned. For example, the compressed Huffman coding representation of a single 8-bit character can be anywhere from 1 to 255 bits long.

In information retrieval, bit arrays are a good representation for the posting lists of very frequent terms. If we compute the gaps between adjacent values in a list of strictly increasing integers and encode them using unary coding, the result is a bit array with a 1 bit in the n th position if and only if n is in the list. The implied probability of

a gap of n is $1/2^n$. This is also the special case of Golomb coding where the parameter M is 1; this parameter is only normally selected when $-\log(2-p)/\log(1-p) \leq 1$, or roughly the term occurs in at least 38% of documents.

Language support

The C programming language's *bitfields*, pseudo-objects found in structs with size equal to some number of bits, are in fact small bit arrays; they are limited in that they cannot span words. Although they give a convenient syntax, the bits are still accessed using bitwise operators on most machines, and they can only be defined statically (like C's static arrays, their sizes are fixed at compile-time). It is also a common idiom for C programmers to use words as small bit arrays and access bits of them using bit operators. A widely available header file included in the X11 system, xtrapbits.h, is "a portable way for systems to define bit field manipulation of arrays of bits.".

In C++, although individual bools typically occupy the same space as a byte or an integer, the STL type `vector<bool>` is a partial specialization in which bits are packed as a space efficiency optimization. Since bytes (and not bits) are the smallest addressable unit in C++, the [] operator does *not* return a reference to an element, but instead returns a proxy reference. This might seem a minor point, but it means that `vector<bool>` is *not* a standard STL container, which is why the use of `vector<bool>` is generally discouraged. Another unique STL class, `bitset`, creates a vector of bits fixed at a particular size at compile-time, and in its interface and syntax more resembles the idiomatic use of words as bit sets by C programmers. It also has some additional power, such as the ability to efficiently count the number of bits that are set. The Boost C++ Libraries provide a `dynamic_bitset` class whose size is specified at run-time.

The D programming language provides bit arrays in both of its competing standard libraries. In Phobos, they are provided in `std.bitmanip`, and in Tango, they are provided in `tango.core.BitArray`. As in C++, the [] operator does not return a reference, since individual bits are not directly addressable on most hardware, but instead returns a bool.

In Java, the class `BitSet` creates a bit array that is then manipulated with functions named after bitwise operators familiar to C programmers. Unlike the `bitset` in C++, the Java `BitSet` expands dynamically if a bit is set at an index beyond the current size of the bit vector. In addition, there is a class `EnumSet`, which represents a Set of values of an enumerated type internally as a bit vector, as a safer alternative to `bitfields`.

The .NET Framework supplies a `BitArray` collection class. It stores boolean values, supports random access and bitwise operators, can be iterated over, and its `Length` property can be changed to grow or truncate it.

Although Standard ML has no support for bit arrays, Standard ML of New Jersey has an extension, the `BitArray` structure, in its SML/NJ Library. It is not fixed in size and supports set operations and bit operations, including, unusually, shift operations.

Haskell likewise currently lacks standard support for bitwise operations, but both GHC and Hugs provide a `Data.Bits` module with assorted bitwise functions and operators, including shift and rotate operations.

In Perl, strings can be used as expandable bit arrays. They can be manipulated using the usual bitwise operators (`~` | `&` `^`)^[4], and individual bits can be test and set using the `vec`^[5] function.

See also

- Bit field
- Bitboard Chess and similar games.
- Bitmap index
- Binary numeral system
- Bitstream

External links

- mathematical bases ^[6] by Pr. D.E.Knuth
- bitarray module ^[7] for Python
- boost::dynamic_bitset ^[8]
- std::bitset ^[9]
- vector<bool> Is Nonconforming, and Forces Optimization Choice ^[10]
- vector<bool>: More Problems, Better Solutions ^[11]

References

- [1] <http://code.google.com/p/compressedbitset/>
- [2] <http://code.google.com/p/javaewah/>
- [3] <http://ricerca.mat.uniroma3.it/users/colanton/concise.html>
- [4] <http://perldoc.perl.org/perlop.html#Bitwise-String-Operators>
- [5] <http://perldoc.perl.org/functions/vec.html>
- [6] <http://www-cs-faculty.stanford.edu/~knuth/fasc1a.ps.gz>
- [7] <http://pypi.python.org/pypi/bitarray>
- [8] http://www.boost.org/libs/dynamic_bitset/dynamic_bitset.html
- [9] <http://www.sgi.com/tech/stl/bitset.html>
- [10] <http://www.gotw.ca/publications/N1185.pdf>
- [11] <http://www.gotw.ca/publications/N1211.pdf>

Bitboard

A **bitboard** is a data structure commonly used in computer systems that play board games.

Definition

A bitboard, often used for boardgames such as chess, checkers and othello, is a specialization of the bitset data structure, where each bit represents a game position or state, designed for optimization of speed and/or memory or disk use in mass calculations. Bits in the same bitboard relate to each other in the rules of the game often forming a game position when taken together. Other bitboards are commonly used as masks to transform or answer queries about positions. The "game" may be any game-like system where information is tightly packed in a structured form with "rules" affecting how the individual units or pieces relate.

Short description

Bitboards are used in many of the world's best chess playing programs. They help the programs analyze chess positions with few CPU instructions and hold a massive number of positions in memory efficiently.

Bitboards are interesting because they allow the computer to answer some questions about game state with one logical operation. For example, if a chess program wants to know if the white player has any pawns in the center of the board (center four squares) it can just compare a bitboard for the player's pawns with one for the center of the board using a logical AND operation. If there are no center pawns then the result will be zero.

Query results can also be represented using bitboards. For example, the query "What are the squares between X and Y?" can be represented as a bitboard. These query results are generally pre-calculated, so that a program can simply retrieve a query result with one memory load.

However, as a result of the massive compression and encoding, bitboard programs are not easy for software developers to either write or debug.

History

The bitboard method for holding a board game appears to have been invented in the mid 1950's, by Arthur Samuel and was used in his checkers program. The method was published in 1959 as "Some Studies in Machine Learning Using the Game of Checkers" in the IBM Journal of Research and Development.

For the more complicated game of chess, it appears the method was independently rediscovered later by the Kaissa team in the Soviet Union in the late 1960s, although not publicly documented, and again by the authors of the U.S. Northwestern University program "Chess" in the early 1970s, and documented in 1977 in "Chess Skill in Man and Machine".

Description for all games or applications

A bitboard or bit field is a format that stuffs a whole group of related boolean variables into the same integer, typically representing positions on a board game. Each bit is a position, and when the bit is positive, a property of that position is true. In chess, for example, there would be a bitboard for black knights. There would be 64-bits where each bit represents a chess square. Another bitboard might be a constant representing the center four squares of the board. By comparing the two numbers with a bitwise logical AND instruction, we get a third bitboard which represents the black knights on the center four squares, if any. This format is generally more CPU and memory friendly than others.

General technical advantages and disadvantages

Processor use

Pros

The advantage of the bitboard representation is that it takes advantage of the essential logical bitwise operations available on nearly all CPUs that complete in one cycle and are full pipelined and cached etc. Nearly all CPUs have AND, OR, NOR, and XOR. Many CPUs have additional bit instructions, such as finding the "first" bit, that make bitboard operations even more efficient. If they do not have instructions well known algorithms can perform some "magic" transformations that do these quickly.

Furthermore, modern CPUs have instruction pipelines that queue instructions for execution. A processor with multiple execution units can perform more than one instruction per cycle if more than one instruction is available in the pipeline. Branching (the use of conditionals like if) makes it harder for the processor to fill its pipeline(s) because the CPU can't tell what it needs to do in advance. Too much branching makes the pipeline less effective and potentially reduces the number of instructions the processor can execute per cycle. Many bitboard operations require fewer conditionals and therefore increase pipelining and make effective use of multiple execution units on many CPUs.

CPUs have a bit width which they are designed toward and can carry out bitwise operations in one cycle in this width. So, on a 64-bit or more CPU, 64-bit operations can occur in one instruction. There may be support for higher or lower width instructions. Many 32-bit CPUs may have some 64-bit instructions and those may take more than one cycle or otherwise be handicapped compared to their 32-bit instructions.

If the bitboard is larger than the width of the instruction set, then a performance hit will be the result. So a program using 64-bit bitboards would run faster on a real 64-bit processor than on a 32-bit processor.

Cons

Some queries are going to take longer than they would with perhaps arrays, but bitboards are generally used in conjunction with array boards in chess programs.

Memory use

Pros

Bitboards are extremely compact. Since only a very small amount of memory is required to represent a position or a mask, more positions can find their way into registers, full speed cache, Level 2 cache, etc. In this way, compactness translates into better performance (on most machines). Also on some machines this might mean that more positions can be stored in main memory before going to disk.

Cons

For some games writing a suitable bitboard engine requires a fair amount of source code that will be longer than the straight forward implementation. For limited devices (like cell phones) with a limited number of registers or processor instruction cache, this can cause a problem. For full sized computers it may cause cache misses between level one and level two cache. This is a potential problem--not a major drawback. Most machines will have enough instruction cache so that this isn't an issue.

Source code

Bitboard source code is very dense and sometimes hard to read. It must be documented very well.

Chess bitboards

Standard

The first bit usually represents A1 (the lower left square), and the 64th bit represents H8 (the diagonally opposite square).

There are twelve types of pieces, and each type gets its own bitboard. Black pawns get a board, white pawns, etc. Together these twelve boards can represent a position. Some trivial information also needs to be tracked elsewhere; the programmer may use boolean variables for whether each side is in check, can castle, etc.

Constants are likely available, such as WHITE_SQUARES, BLACK_SQUARES, FILE_A, RANK_4 etc. More interesting ones might include CENTER, CORNERS, CASTLE_SQUARES, etc.

Examples of variables would be WHITE_ATTACKING, ATTACKED_BY_PAWN, WHITE_PASSED_PAWN, etc.

Rotated

"Rotated" bitboards are usually used in programs that use bitboards. Rotated bitboards make certain operations more efficient. While engines are simply referred to as "rotated bitboard engines," this is a misnomer as rotated boards are used in *addition* to normal boards making these hybrid standard/rotated bitboard engines.

These bitboards rotate the bitboard positions by 90 degrees, 45 degrees, and/or 315 degrees. A typical bitboard will have one byte per rank of the chess board. With this bitboard it's easy to determine rook attacks across a rank, using a table indexed by the occupied square and the occupied positions in the rank (because rook attacks stop at the first occupied square). By rotating the bitboard 90 degrees, rook attacks across a file can be examined the same way. Adding bitboards rotated 45 degrees and 315 degrees produces bitboards in which the diagonals are easy to examine. The queen can be examined by combining rook and bishop attacks. Rotated bitboards appear to have been developed separately and (essentially) simultaneously by the developers of the DarkThought and Crafty programs. The Rotated bitboard is hard to understand if one doesn't have a firm grasp of normal bitboards and why they work. Rotated bitboards should be viewed as a clever but advanced optimization.

Magics

Magic move bitboard generation is a new and fast alternative to rotated move bitboard generators. These are also more versatile than rotated move bitboard generators because the generator can be used independently from any position. The basic idea is that you can use a multiply, right-shift hashing function to index a move database, which can be as small as 1.5K. A speedup is gained because no rotated bitboards need to be updated, and because the lookups are more cache-friendly.

Other bitboards

Many other games besides chess benefit from bitboards.

- In Connect Four, they allow for very efficient testing for four consecutive discs, by just two shift+and operations per direction.
- In the Conway's Game of Life, they are a possible alternative to arrays.
- Othello/Reversi (see the Reversi article).

See also

- Bit array
- Bit field
- Bit manipulation
- Bit twiddler
- Bitwise operation
- Board representation (chess)
- Boolean algebra (logic)
- Instruction set
- Instruction pipeline
- Opcode
- Bytecode

External links

Checkers

- Checkers Bitboard Tutorial ^[1] by Jonathan Kreuzer

Chess

Articles

- Programming area of the Beowulf project ^[2]
- Heinz, Ernst A. How DarkThought plays chess. *ICCA Journal*, Vol. 20(3), pp. 166-176, Sept. 1997 ^[3]
- Laramee, Francois-Dominic. Chess Programming Part 2: Data Structures. ^[4]
- Verhelst, Paul. Chess Board Representations ^[5]
- Hyatt, Robert. Chess program board representations ^[6]
- Hyatt, Robert. Rotated bitmaps, a new twist on an old idea ^[7]
- Frayn, Colin. How to implement bitboards in a chess engine (chess programming theory) ^[8]
- Pepicelli, Glen. *Bitfields, Bitboards, and Beyond* ^[9] -(Example of bitboards in the Java Language and a discussion of why this optimization works with the Java Virtual Machine (www.OnJava.com publisher: O'Reilly 2005))

Code examples

- [10] The author of the Frenzee engine had posted some source examples.

Implementations

Open source

- Beowulf^[11] Unix, Linux, Windows. Rotated bitboards.
- Crafty See the Crafty article. Written in straight C. Rotated bitboards in the old versions, now uses magic bitboards. Strong.
- GNU Chess See the GNU Chess Article.
- Gray Matter^[12] C++, rotated bitboards.
- KnightCap GPL. ELO of 2300.
- Pepito^[13] C. Bitboard, by Carlos del Cacho. Windows and Linux binaries as well as source available.
- Simontacchi^[14] Rotated bitboards.

Closed source

- DarkThought Home Page^[15]

Othello

- A complete discussion^[16] of Othello (Reversi) engines with some source code including an Othello bitboard in C and assembly.

References

- [1] <http://www.3dkingdoms.com/checkers/bitboards.htm>
- [2] <http://www.frayn.net/beowulf/theory.html>
- [3] <http://supertech.lcs.mit.edu/~heinz/dt/node2.html>
- [4] <http://www.gamedev.net/reference/programming/features/chess2/page3.asp>
- [5] <http://chess.verhelst.org/1997/03/10/representations/>
- [6] <http://www.cis.uab.edu/info/faculty/hyatt/boardrep.html>
- [7] <http://www.cis.uab.edu/info/faculty/hyatt/bitmaps.html>
- [8] <http://www.frayn.net/beowulf/theory.html#bitboards>
- [9] <http://www.onjava.com/pub/a/onjava/2005/02/02/bitsets.html>
- [10] <http://www.geocities.com/ruleren/sources.html>
- [11] <http://www.frayn.net/beowulf/index.html>
- [12] <http://code.google.com/p/gray-matter/>
- [13] <http://www.quarkchess.de/pepito/>
- [14] <http://simontacchi.sourceforge.net/>
- [15] <http://supertech.lcs.mit.edu/~heinz/dt/>
- [16] <http://www.radagast.se/othello/>

Parallel array

In computing, a **parallel array** is a data structure for representing arrays of records. It keeps a separate, homogeneous array for each field of the record, each having the same number of elements. Then, objects located at the same index in each array are implicitly the fields of a single record. Pointers from one object to another are replaced by array indices. This contrasts with the normal approach of storing all fields of each record together in memory. For example, one might declare an array of 100 names, each a string, and 100 ages, each an integer, associating each name with the age that has the same index.

An example in C using parallel arrays:

```
int ages[] = {0,           17,          2,           52,           25};
char *names[] = {"None",    "Mike",      "Billy",     "Tom",
                 "Stan"};
int parent[] = {0 /*None*/, 3 /*Tom*/, 1 /*Mike*/, 0 /*None*/, 3
/*Tom*/};

for(i = 1; i <= 4; i++) {
    printf("Name: %s, Age: %d, Parent: %s \n",
           names[i], ages[i], names[parent[i]]);
}
```

Or, in Python:

```
firstName = ['Joe', 'Bob', 'Frank', 'Hans']
lastName = ['Smith', 'Seger', 'Sinatra', 'Schultze']
heightInCM = [169, 158, 201, 199]

for i in xrange(len(firstName)):
    print "Name: %s %s" % (firstName[i], lastName[i])
    print "Height in CM: %s" % heightInCM[i]
```

Parallel arrays have a number of practical advantages over the normal approach:

- They can be used in languages which support only arrays of primitive types and not of records (or perhaps don't support records at all).
- Parallel arrays are simple to understand and use, and are often used where declaring a record is more trouble than it's worth.
- They can save a substantial amount of space in some cases by avoiding alignment issues. For example, one of the fields of the record can be a single bit, and its array would only need to reserve one bit for each record, whereas in the normal approach many more bits would "pad" the field so that it consumes an entire byte or a word.
- If the number of items is small, array indices can occupy significantly less space than full pointers, particularly on architectures with large words.
- Sequentially examining a single field of each record in the array is very fast on modern machines, since this amounts to a linear traversal of a single array, exhibiting ideal locality of reference and cache behavior.

However, parallel arrays also have several strong disadvantages, which serves to explain why they are not generally preferred:

- They have significantly worse locality of reference when visiting the records sequentially and examining multiple fields of each record, which is the norm.
- They obscure the relationship between fields of a single record.

- They have little direct language support (the language and its syntax typically express no relationship between the arrays in the parallel array.)
- They are expensive to grow or shrink, since each of several arrays must be reallocated.

The bad locality of reference is the worst issue. However, a compromise can be made in some cases: if a structure can be divided into groups of fields that are generally accessed together, an array can be constructed for each group, and its elements are records containing only these subsets of the larger structure's fields. This is a valuable way of speeding up access to very large structures with many members, while keeping the portions of the structure tied together. An alternative to tying them together using array indexes is to use references to tie the portions together, but this can be less efficient in time and space. Some compiler optimizations, particularly for vector processors, are able to perform this transformation automatically when arrays of structures are created in the program.

See also

- An example in the linked list article
- Column-oriented DBMS

References

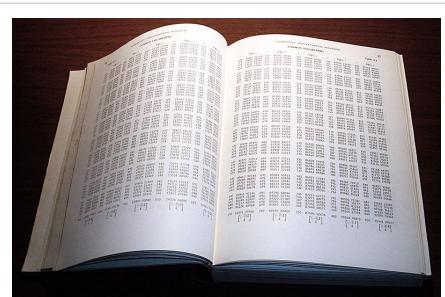
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Page 209 of section 10.3: Implementing pointers and objects.

Lookup table

In computer science, a **lookup table** is a data structure, usually an array or associative array, often used to replace a runtime computation with a simpler array indexing operation. The savings in terms of processing time can be significant^[1], since retrieving a value from memory is often faster than undergoing an 'expensive' computation or input/output operation. The tables may be precalculated and stored in static program storage or calculated (or "pre-fetched") as part of a programs initialization phase (memoization). Lookup tables are also used extensively to validate input values by matching against a list of valid (or invalid) items in an array and, in some programming languages, may include pointer functions (or offsets to labels) to process the matching input.

History

Before the advent of computers, printed lookup tables of values were used by people to speed up hand calculations of complex functions, such as in trigonometry, logarithms, and statistical density functions^[2]. School children are often taught to memorize "times tables" to avoid calculations of the most commonly used numbers (up to 9×9 or 12×12). Even as early as 493 A.D., Victorius of Aquitaine wrote a 98-column multiplication table which gave (in Roman numerals) the product of every number from 2 to 50 times and the rows were "a list of numbers starting with one thousand, descending by hundreds to one hundred, then descending by tens to ten, then by ones to one, and then the fractions down to 1/144"^[3]



Part of a 20th century table of common logarithms in the reference book Abramowitz and Stegun.

Early in the history of computers, input/output operations were particularly slow - even in comparison to processor speeds of the time. It made sense to reduce expensive read operations by a form of manual caching by creating either static lookup tables (embedded in the program) or dynamic prefetched arrays to contain only the most commonly occurring data items. Despite the introduction of systemwide caching that now automates this process, application level lookup tables can still improve performance for data items that rarely, if ever, change.

Examples

Simple lookup in an array, an associative array or a linked list (unsorted list)

This is known as a linear search or Brute-force search, each element being checked for equality in turn and the associated value, if any, used as a result of the search. This is often the slowest search method unless frequently occurring values occur early in the list. For a one dimensional array or linked list, the lookup is usually to determine whether or not there is a match with an 'input' data value.

Linked lists vs. arrays

Linked lists have several advantages over arrays:

- Insertion or deletion of an element at a specific point of a list is a constant time operation. (While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", an algorithm that iterates over the elements may have to skip a large number of vacant slots).
- arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while an array will eventually fill up, and then have to be resized — an expensive operation, that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed, may have to be resized in order to avoid wasting too much space.

On the other hand:

- arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to quickly look up an element by its index , such as heapsort. See also trivial hash function below.
- Sequential access on arrays is also faster than on linked lists on many machines, because they have greater locality of reference and thus benefit more from processor caching.
- linked lists require extra storage needed for references, that often makes them impractical for lists of small data items such as characters or boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

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

Binary search in an array or an associative array (sorted list)

This is known as a binary chop search, sometimes referred to as a "Divide and conquer algorithm", each element being found by determining which half of the table a match may be found in and repeating until either success or failure. Only possible if the list is sorted but gives good performance even if the list is lengthy.

Trivial hash function

For a trivial hash function lookup, the unsigned raw data value is used *directly* as an index to a one dimensional table to extract a result. For small ranges, this can be amongst the fastest lookup, even exceeding binary search speed with zero branches and executing in constant time.

Counting '1' bits in a series of bytes

One discrete problem that is expensive to solve on many computers, is that of counting the number of bits which are set to 1 in a (binary) number, sometimes called the *population function*. For example, the decimal number "37" is "00100101" in binary, so it contains three bits that are set to binary "1".

A simple example of C code, designed to count the 1 bits in a *int*, might look like this:

```
int count_ones(unsigned int x) {
    int result = 0;
    while (x != 0)
        result++, x = x & (x-1);
    return result;
}
```

This apparently simple algorithm can take potentially hundreds of cycles even on a modern architecture, because it makes many branches in the loop - and branching is slow. This can be ameliorated using loop unrolling and some other more compiler optimizations. There is however a simple and much faster algorithmic solution - using a trivial hash function table lookup.

Simply construct a static table, *bits_set*, with 256 entries giving the number of one bits set in each possible byte value (e.g. 0x00 = 0, 0x01 = 1, 0x02 = 1, and so on). Then use this table to find the number of ones in each byte of the integer using a trivial hash function lookup on each byte in turn, and sum them. This requires no branches, and just four indexed memory accesses, considerably faster than the earlier code.

```
/* (this code assumes that 'int' is 32-bits wide) */
int count_ones(unsigned int x) {
    return bits_set[ x           & 255] + bits_set[(x >> 8) & 255]
        + bits_set[(x >> 16) & 255] + bits_set[(x >> 24) & 255];
}
```

The above source can be improved easily, (avoiding OR'ing, and shifting) by 'recasting' 'x' as a 4 byte unsigned char array and, preferably, coded in-line as a single statement instead of being a function. Note that even this simple algorithm can be too slow now, because the original code might run faster from the cache of modern processors, and (large) lookup tables do not fit well in caches and can cause a slower access to memory (in addition, in the above example, it requires computing addresses within a table, to perform the four lookups needed).

L.U.T.s in Image processing

In data analysis applications, such as image processing, a lookup table (LUT) is used to transform the input data into a more desirable output format. For example, a grayscale picture of the planet Saturn will be transformed into a color image to emphasize the differences in its rings.

A classic example of reducing run-time computations using lookup tables is to obtain the result of a trigonometry calculation, such as the sine of a value. Calculating trigonometric functions can substantially slow a computing application. The same application can finish much sooner when it first precalculates the sine of a number of values, for example for each whole number of degrees (The table can be defined as static variables at compile time, reducing repeated run time costs). When the program requires the sine of a value, it can use the lookup table to retrieve the closest sine value from a memory address, and may also take the step of interpolating to the sine of the desired value, instead of calculating by mathematical formula. Lookup tables are thus used by mathematics co-processors in computer systems. An error in a lookup table was responsible for Intel's infamous floating-point divide bug.

Functions of a single variable (such as sine and cosine) may be implemented by a simple array. Functions involving two or more variables require multidimensional array indexing techniques. The latter case may thus employ a two-dimensional array of **power[x][y]** to replace a function to calculate x^y for a limited range of x and y values. Functions that have more than one result may be implemented with lookup tables that are arrays of structures.

As mentioned, there are intermediate solutions that use tables in combination with a small amount of computation, often using interpolation. Pre-calculation combined with interpolation can produce higher accuracy for values that fall between two precomputed values. This technique requires slightly more time to be performed but can greatly enhance accuracy in applications that require the higher accuracy. Depending on the values being precomputed, pre-computation with interpolation can also be used to shrink the lookup table size while maintaining accuracy.

In image processing, lookup tables are often called **LUTs** and give an output value for each of a range of index values. One common LUT, called the *colormap* or *palette*, is used to determine the colors and intensity values with which a particular image will be displayed. Windowing in computed tomography refers to a related concept.

While often effective, employing a lookup table may nevertheless result in a severe penalty if the computation that the LUT replaces is relatively simple. Memory retrieval time and the complexity of memory requirements can increase application operation time and system complexity relative to what would be required by straight formula computation. The possibility of polluting the cache may also become a problem. Table accesses for large tables will almost certainly cause a cache miss. This phenomenon is increasingly becoming an issue as processors outpace memory. A similar issue appears in rematerialization, a compiler optimization. In some environments, such as the Java programming language, table lookups can be even more expensive due to mandatory bounds-checking involving an additional comparison and branch for each lookup.

There are two fundamental limitations on when it is possible to construct a lookup table for a required operation. One is the amount of memory that is available: one cannot construct a lookup table larger than the space available for the table, although it is possible to construct disk-based lookup tables at the expense of lookup time. The other is the time required to compute the table values in the first instance; although this usually needs to be done only once, if it takes a prohibitively long time, it may make the use of a lookup table an inappropriate solution. As previously stated however, tables can be statically defined in many cases.

Computing sines

Most computers, which only perform basic arithmetic operations, cannot directly calculate the sine of a given value. Instead, they use the CORDIC algorithm or a complex formula such as the following Taylor series to compute the value of sine to a high degree of precision:

$$\sin(x) \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} \text{ (for } x \text{ close to 0)}$$

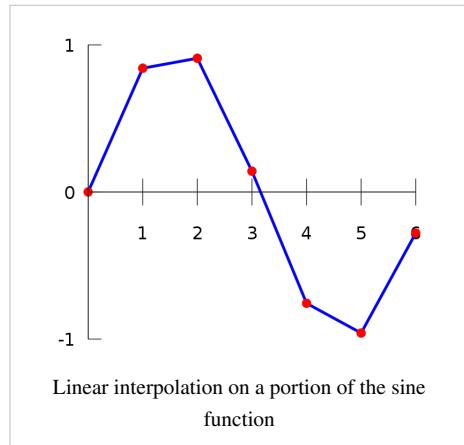
However, this can be expensive to compute, especially on slow processors, and there are many applications, particularly in traditional computer graphics, that need to compute many thousands of sine values every second. A common solution is to initially compute the sine of many evenly distributed values, and then to find the sine of x we choose the sine of the value closest to x . This will be close to the correct value because sine is a continuous function with a bounded rate of change. For example:

```
real array sine_table[-1000..1000]
for x from -1000 to 1000
    sine_table[x] := sine(pi * x / 1000)

function lookup_sine(x)
    return sine_table[round(1000 * x / pi)]
```

Unfortunately, the table requires quite a bit of space: if IEEE double-precision floating-point numbers are used, over 16,000 bytes would be required. We can use fewer samples, but then our precision will significantly worsen. One good solution is linear interpolation, which draws a line between the two points in the table on either side of the value and locates the answer on that line. This is still quick to compute, and much more accurate for smooth functions such as the sine function. Here is our example using linear interpolation:

```
function lookup_sine(x)
    x1 := floor(x*1000/pi)
    y1 := sine_table[x1]
    y2 := sine_table[x1+1]
    return y1 + (y2-y1)*(x*1000/pi-x1)
```



When using interpolation, the size of the lookup table can be reduced by using *non-uniform sampling*, which means that where the function is close to straight, we use few sample points, while where it changes value quickly we use more sample points to keep the approximation close to the real curve. For more information, see interpolation.

Other usage of lookup tables

Caches

Storage caches (including disk caches for files, or processor caches for either for code or data) work also like a lookup table. The table is built with very fast memory instead of being stored on slower external memory, and maintains two pieces of data for a subrange of bits composing an external memory (or disk) address (notably the lowest bits of any possible external address):

- one piece (the tag) contains the value of the remaining bits of the address; if these bits match with those from the memory address to read or write, then the other piece contains the cached value for this address.
- the other piece maintains the data associated to that address.

A single (fast) lookup is performed to read the tag in the lookup table at the index specified by the lowest bits of the desired external storage address, and to determine if the memory address is hit by the cache. When a hit is found, no access to external memory is needed (except for write operations, where the cached value may need to be updated asynchronously to the slower memory after some time, or if the position in the cache must be replaced to cache another address).

Hardware LUTs

In digital logic, an n -bit lookup table can be implemented with a multiplexer whose select lines are the inputs of the LUT and whose outputs are constants. An n -bit LUT can encode any n -input Boolean function by modeling such functions as truth tables. This is an efficient way of encoding Boolean logic functions, and LUTs with 4-6 bits of input are in fact the key component of modern FPGAs.

See also

- Branch table
- Memoization
- Memory bound function
- Palette and Colour Look-Up Table - for the usage in computer graphics
- 3D LUT – usage in film

External links

- Fast table lookup using input character as index for branch table ^[4]
- Art of Assembly: Calculation via Table Lookups ^[5]
- Color Presentation of Astronomical Images ^[6]
- "Bit Twiddling Hacks" (includes lookup tables) ^[7] By Sean Eron Anderson of Stanford university
- Memoization in C++ ^[8] by Paul McNamee, Johns Hopkins University showing savings
- "The Quest for an Accelerated Population Count" ^[9] by Henry S. Warren, Jr.

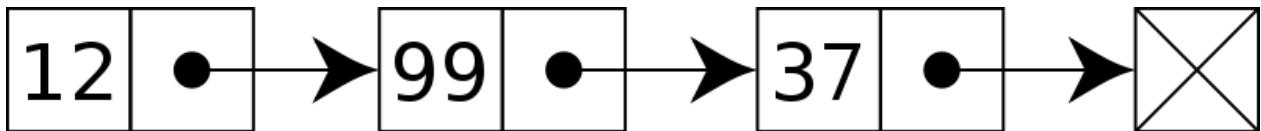
References

- [1] <http://apl.jhu.edu/~paulmac/c++-memoization.html>
- [2] <http://www.amazon.com/dp/0198508417> "The History of Mathematical Tables: From Sumer to Spreadsheets"
- [3] Maher & Makowski 2001, p.383
- [4] http://en.wikibooks.org/wiki/360_Assembly/Branch_Instructions
- [5] <http://webster.cs.ucr.edu/AoA/Windows/HTML/TableLookups.html>
- [6] <http://www.allthesky.com/articles/imagecolor.html>
- [7] <http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable>
- [8] <http://apl.jhu.edu/~paulmac/c++-memoization.html>
- [9] <http://books.google.co.uk/books?id=gJrmszNHQV4C&lpg=PT169&ots=rKUXyBR7ug&dq=beautiful%20code%20%22population%20count%22&pg=PT169#v=onepage&q=beautiful%20code%20%22population%20count%22&f=false>

Lists

Linked list

In computer science, a **linked list** is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a *link*) to the next record in the sequence.



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

Linked lists are among the simplest and most common data structures; they provide an easy implementation for several important abstract data structures, including stacks, queues, hash tables, symbolic expressions, and skip lists.

The principal benefit of a linked list over a conventional array is that the order of the linked items may be different from the order that the data items are stored in memory or on disk. For that reason, linked lists allow insertion and removal of nodes at any point in the list, with a constant number of operations.

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

Linked lists can be implemented in most languages. Languages such as Lisp and Scheme have the data structure built in, along with operations to access the linked list. Procedural languages, such as C, or object-oriented languages, such as C++ and Java, typically rely on mutable references to create linked lists.

History

Linked lists were developed in 1955-56 by Allen Newell, Cliff Shaw and Herbert Simon at RAND Corporation as the primary data structure for their Information Processing Language. IPL was used by the authors to develop several early artificial intelligence programs, including the Logic Theory Machine, the General Problem Solver, and a computer chess program. Reports on their work appeared in *IRE Transactions on Information Theory* in 1956, and several conference proceedings from 1957-1959, including *Proceedings of the Western Joint Computer Conference* in 1957 and 1958, and *Information Processing* (Proceedings of the first UNESCO International Conference on Information Processing) in 1959. The now-classic diagram consisting of blocks representing list nodes with arrows pointing to successive list nodes appears in "Programming the Logic Theory Machine" by Newell and Shaw in *Proc. WJCC*, February 1957. Newell and Simon were recognized with the ACM Turing Award in 1975 for having "made basic contributions to artificial intelligence, the psychology of human cognition, and list processing".

The problem of machine translation for natural language processing led Victor Yngve at Massachusetts Institute of Technology (MIT) to use linked lists as data structures in his COMIT programming language for computer research in the field of linguistics. A report on this language entitled "A programming language for mechanical translation" appeared in *Mechanical Translation* in 1958.

LISP, standing for *list processor*, was created by John McCarthy in 1958 while he was at MIT and in 1960 he published its design in a paper in the Communications of the ACM, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". One of LISP's major data structures is the linked list.

By the early 1960s, the utility of both linked lists and languages which use these structures as their primary data representation was well established. Bert Green of the MIT Lincoln Laboratory published a review article entitled "Computer languages for symbol manipulation" in *IRE Transactions on Human Factors in Electronics* in March 1961 which summarized the advantages of the linked list approach. A later review article, "A Comparison of list-processing computer languages" by Bobrow and Raphael, appeared in *Communications of the ACM* in April 1964.

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

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

Basic concepts and nomenclature

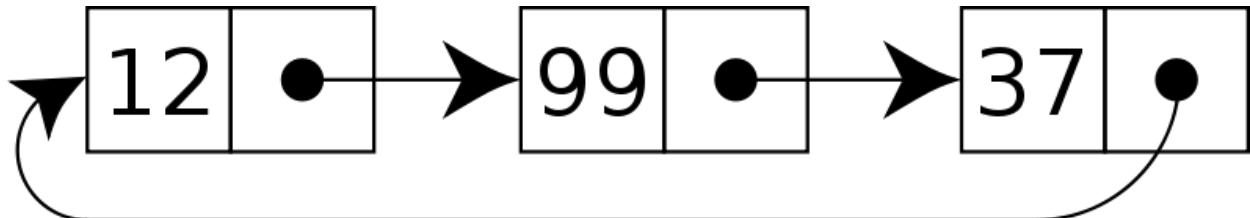
Each record of a linked list is often called an **element** or **node**.

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

The **head** of a list is its first node, and the **tail** is the list minus that node (or a pointer thereto). In Lisp and some derived languages, the tail may be called the **CDR** (pronounced *could-R*) of the list, while the payload of the head node may be called the **CAR**.

Linear and circular lists

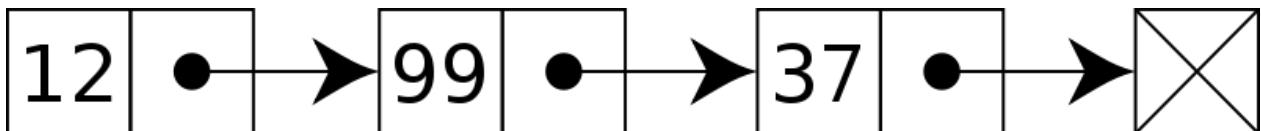
In the last node of a list, the link field often contains a **null** reference, a special value that is interpreted by programs as meaning "there is no such node". A less common convention is to make it point to the first node of the list; in that case the list is said to be **circular** or **circularly linked**; otherwise it is said to be **open** or **linear**.



A circular linked list

Singly-, doubly-, and multiply-linked lists

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



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

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



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

The technique known as XOR-linking allows a doubly-linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

In a **multiply-linked list**, each node contains two or more link fields, each field being used to connect the same set of data records in a different order (e.g., by name, by department, by date of birth, etc.). (While doubly-linked lists can be seen as special cases of multiply-linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.)

Sentinel nodes

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

Empty lists

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

Hash linking

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

List handles

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

Combining alternatives

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

Tradeoffs

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

Linked lists vs. dynamic arrays

	Linked list	Array	Dynamic array
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Insertion/deletion at end	$\Theta(1)$	N/A	$\Theta(1)$
Insertion/deletion in middle	$\Theta(1)$	N/A	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$

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

Linked lists have several advantages over dynamic arrays. Insertion of an element at a specific point of a list is a constant-time operation, whereas insertion in a dynamic array may require moving half of the elements, or more. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", this causes fragmentation that impedes the performance of iteration.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while a dynamic array will eventually fill up its underlying array data structure and have to reallocate — an expensive operation, one that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may have to be resized in order to avoid wasting too much space.

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

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

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

A good example that highlights the pros and cons of using dynamic arrays vs. linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle n times. Once you

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

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

Singly-linked linear lists vs. other lists

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

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

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

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

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

Doubly-linked vs. singly-linked

Double-linked lists require more space per node (unless one uses xor-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly-linked list, one must have the *previous* node's address. Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

Circularly-linked vs. linearly-linked

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

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

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

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

Using sentinel nodes

Sentinel node may simplify certain list operations, by ensuring that the next and/or previous nodes exist for every element, and that even empty lists have at least one node. One may also use a sentinel node at the end of the list, with an appropriate data field, to eliminate some end-of-list tests. For example, when scanning the list looking for a node with a given value x , setting the sentinel's data field to x makes it unnecessary to test for end-of-list inside the loop. Another example is the merging two sorted lists: if their sentinels have data fields set to $+\infty$, the choice of the next output node does not need special handling for empty lists.

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

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

The same trick can be used to simplify the handling of a doubly-linked linear list, by turning it into a circular doubly-linked list with a single sentinel node. However, in this case, the handle should be a single pointer to the dummy node itself.^[1]

Linked list operations

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

Linearly-linked lists

Singly-linked lists

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

```

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

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

```

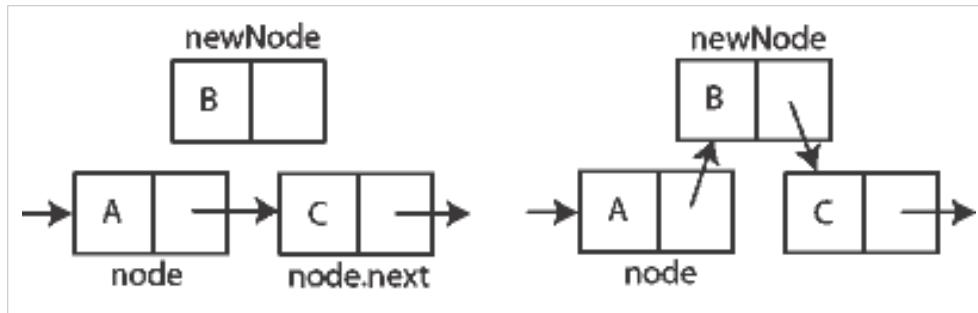
Traversal of a singly-linked list is simple, beginning at the first node and following each *next* link until we come to the end:

```

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

```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done; instead, you have to locate it while keeping track of the previous node.



```

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

```

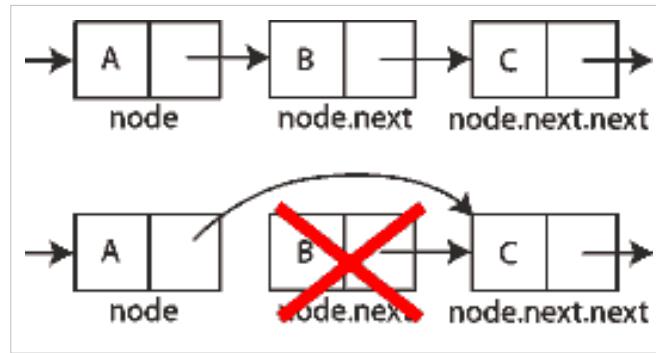
Inserting at the beginning of the list requires a separate function. This requires updating *firstNode*.

```

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

```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.



```
function removeAfter(node node) { // remove node past this one
    obsoleteNode := node.next
    node.next := node.next.next
    destroy obsoleteNode
}
```

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

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

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

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

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

Circularly-linked list

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

Circularly-linked lists can be either singly or doubly linked.

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

Algorithms

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

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

Notice that the test "**while** node **≠** someNode" must be at the end of the loop. If it were replaced by the test "" at the beginning of the loop, the procedure would fail whenever the list had only one node.

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

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

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

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

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

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

Linked lists using arrays of nodes

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

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

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

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

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

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

Index	Next	Prev	Name	Balance
0	1	4	Jones, John	123.45
1	-1	0	Smith, Joseph	234.56
2 (listHead)	4	-1	Adams, Adam	0.00
3			Ignore, Ignatius	999.99
4	0	2	Another, Anita	876.54
5				
6				
7				

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

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

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

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

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

This approach has one main disadvantage, however: it creates and manages a private memory space for its nodes. This leads to the following issues:

- It increases complexity of the implementation.
- Growing a large array when it is full may be difficult or impossible, whereas finding space for a new linked list node in a large, general memory pool may be easier.

- Adding elements to a dynamic array will occasionally (when it is full) unexpectedly take linear ($O(n)$) instead of constant time (although it's still an amortized constant).
- Using a general memory pool leaves more memory for other data if the list is smaller than expected or if many nodes are freed.

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

Language support

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

In languages that support Abstract data types or templates, linked list ADTs or templates are available for building linked lists. In other languages, linked lists are typically built using references together with records. Here is a complete example in C:

```
#include <stdio.h>      /* for printf */
#include <stdlib.h>     /* for malloc */

typedef struct node {
    int data;
    struct node *next; /* pointer to next element in list */
} LLIST;

LLIST *list_add(LLIST **p, int i);
void list_remove(LLIST **p);
LLIST **list_search(LLIST **n, int i);
void list_print(LLIST *n);

LLIST *list_add(LLIST **p, int i)
{
    if (p == NULL)
        return NULL;

    LLIST *n = malloc(sizeof(LLIST));
    if (n == NULL)
        return NULL;

    n->next = *p; /* the previous element (*p) now becomes the "next"
element */
    *p = n;         /* add new empty element to the front (head) of the
list */
    n->data = i;

    return *p;
}
```

```
void list_remove(LLIST **p) /* remove head */
{
    if (*p != NULL && *p->next != NULL)
    {
        LLIST *n = *p;
        *p = (*p)->next;
        free(n);
    }
}

LLIST **list_search(LLIST **n, int i)
{
    if (*n == NULL)
        return NULL;

    while (*n != NULL)
    {
        if ((*n)->data == i)
        {
            return n;
        }
        n = &(*n)->next;
    }
    return NULL;
}

void list_print(LLIST *n)
{
    if (n == NULL)
    {
        printf("list is empty\n");
    }
    while (n != NULL)
    {
        printf("print %p %p %d\n", n, n->next, n->data);
        n = n->next;
    }
}

int main(void)
{
    LLIST *n = NULL;

    list_add(&n, 0); /* list: 0 */
    list_add(&n, 1); /* list: 1 0 */
    list_add(&n, 2); /* list: 2 1 0 */
    list_add(&n, 3); /* list: 3 2 1 0 */
```

```
list_add(&n, 4); /* list: 4 3 2 1 0 */
list_print(n);
list_remove(&n);           /* remove first (4) */
list_remove(&n->next);    /* remove new second (2) */
list_remove(list_search(&n, 1)); /* remove cell containing 1
(first) */
list_remove(&n->next);    /* remove second to last node (0)
*/
list_remove(&n);           /* remove last (3) */
list_print(n);

return 0;
}
```

Internal and external storage

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

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

In general, if a set of data structures needs to be included in multiple linked lists, external storage is the best approach. If a set of data structures need to be included in only one linked list, then internal storage is slightly better, unless a generic linked list package using external storage is available. Likewise, if different sets of data that can be stored in the same data structure are to be included in a single linked list, then internal storage would be fine.

Another approach that can be used with some languages involves having different data structures, but all have the initial fields, including the *next* (and *prev* if double linked list) references in the same location. After defining separate structures for each type of data, a generic structure can be defined that contains the minimum amount of data shared by all the other structures and contained at the top (beginning) of the structures. Then generic routines can be created that use the minimal structure to perform linked list type operations, but separate routines can then handle the specific data. This approach is often used in message parsing routines, where several types of messages are received, but all start with the same set of fields, usually including a field for message type. The generic routines are used to add new messages to a queue when they are received, and remove them from the queue in order to process the message. The message type field is then used to call the correct routine to process the specific type of message.

Example of internal and external storage

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

```
record member { // member of a family
    member next
    string firstName
    integer age
}

record family { // the family itself
    family next
    string lastName
    string address
    member members // head of list of members of this family
}
```

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

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

Using external storage, we would create the following structures:

```
record node { // generic link structure
    node next
    pointer data // generic pointer for data at node
}

record member { // structure for family member
    string firstName
    integer age
}

record family { // structure for family
    string lastName
    string address
    node members // head of list of members of this family
}
```

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

```
famNode := Families // start at head of families list
while famNode ≠ null { // loop through list of families
    aFamily := (family)famNode.data // extract family from node
```

```
print information about family
memNode := aFamily.members // get list of family members
while memNode != null { // loop through list of members
    aMember := (member)memNode.data // extract member from node
    print information about member
    memNode := memNode.next
}
famNode := famNode.next
}
```

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

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

Speeding up search

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

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

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

Related data structures

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

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

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

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

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

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

References

- "Definition of a linked list" [2]. National Institute of Standards and Technology. 2004-08-16. Retrieved 2004-12-14.
- Antonakos, James L.; Mansfield, Kenneth C., Jr. (1999). *Practical Data Structures Using C/C++*. Prentice-Hall. pp. 165–190. ISBN 0-13-280843-9.
- Collins, William J. (2005) [2002]. *Data Structures and the Java Collections Framework*. New York: McGraw Hill. pp. 239–303. ISBN 0-07-282379-8.
- Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein (2003). *Introduction to Algorithms*. MIT Press. pp. 205–213 & 501–505. ISBN 0-262-03293-7.
- Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein (2001). "10.2: Linked lists". *Introduction to Algorithms* (2nd ed.). MIT Press. pp. 204–209. ISBN 0-262-03293-7.
- Green, Bert F. Jr. (1961). "Computer Languages for Symbol Manipulation". *IRE Transactions on Human Factors in Electronics* (2): 3–8.
- McCarthy, John (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" [3]. *Communications of the ACM*.
- Knuth, Donald (1997). "2.2.3-2.2.5". *Fundamental Algorithms* (3rd ed.). Addison-Wesley. pp. 254–298. ISBN 0-201-89683-4.
- Newell, Allen; Shaw, F. C. (1957). "Programming the Logic Theory Machine". *Proceedings of the Western Joint Computer Conference*: 230–240.
- Parlante, Nick (2001). "Linked list basics" [4]. Stanford University. Retrieved 2009-09-21.
- Sedgewick, Robert (1998). *Algorithms in C*. Addison Wesley. pp. 90–109. ISBN 0-201-31452-5.
- Shaffer, Clifford A. (1998). *A Practical Introduction to Data Structures and Algorithm Analysis*. New Jersey: Prentice Hall. pp. 77–102. ISBN 0-13-660911-2.
- Wilkes, Maurice Vincent (1964). "An Experiment with a Self-compiling Compiler for a Simple List-Processing Language". *Annual Review in Automatic Programming* (Pergamon Press) 4 (1).
- Wilkes, Maurice Vincent (1964). "Lists and Why They are Useful". *Proceeds of the ACM National Conference, Philadelphia 1964* (ACM) (P-64): F1-1.
- Shanmugasundaram, Kulesh (2005-04-04). "Linux Kernel Linked List Explained" [5]. Retrieved 2009-09-21.

External links

- Description [2] from the Dictionary of Algorithms and Data Structures
- Some linked list materials are available from the Stanford University Computer Science department:
 - Introduction to Linked Lists [6]
 - Linked List Problems [7]
- SimCList, an open source C library for linked lists [8]
- SGLib, an collection of algorithms for operating on linked lists (and other containers) [9]
- shared singly linked list implementations [10]
- Patent for the idea of having nodes which are in several linked lists simultaneously [11] (note that this technique was widely used for many decades before the patent was granted)
- Non-patented data structure for having nodes which are in several linked lists simultaneously [12]

References

- [1] Ford, William and Topp, William *Data Structures with C++ using STL Second Edition* (2002). Prentice-Hall. ISBN 0-13-085850-1, pp. 466-467
- [2] <http://nist.gov/dads/HTML/linkedList.html>
- [3] <http://www-formal.stanford.edu/jmc/recursiv.html>
- [4] <http://cslibrary.stanford.edu/103/LinkedListBasics.pdf>
- [5] <http://isis.poly.edu/kulesh/stuff/src/klist/>
- [6] <http://cslibrary.stanford.edu/103/>
- [7] <http://cslibrary.stanford.edu/105/>
- [8] <http://mij.oltrelinux.com-devel/simclist/>
- [9] <http://sglib.sourceforge.net/>
- [10] <http://www.cs.chalmers.se/~noble/manual/sllist.html>
- [11] <http://www.google.com/patents?vid=USPAT7028023>
- [12] <http://jroller.com/page/bobfoster>

XOR linked list

XOR linked lists are a data structure used in computer programming. They take advantage of the bitwise exclusive disjunction (XOR) operation, here denoted by \oplus , to decrease storage requirements for doubly-linked lists. An ordinary doubly-linked list stores addresses of the previous and next list items in each list node, requiring two address fields: ... A B C D E ... \rightarrow next \rightarrow next \rightarrow next \rightarrow $<\!\!<$ prev $<\!\!<$ prev $<\!\!<$ prev $<\!\!<$ An XOR linked list compresses the same information into *one* address field by storing the bitwise XOR of the address for *previous* and the address for *next* in one field: ... A B C D E ... \leftrightarrow A \oplus C \leftrightarrow B \oplus D \leftrightarrow C \oplus E \leftrightarrow When you traverse the list from left to right: supposing you are at C, you can take the address of the previous item, B, and XOR it with the value in the link field (B \oplus D). You will then have the address for D and you can continue traversing the list. The same pattern applies in the other direction.

To start traversing the list in either direction from some point, you need the address of two consecutive items, not just one. If the addresses of the two consecutive items are reversed, you will end up traversing the list in the opposite direction.

This form of linked list may be inadvisable:

- General-purpose debugging tools cannot follow the XOR chain, making debugging more difficult;^[1]
- The price for the decrease in memory usage is an increase in code complexity, making maintenance more expensive;
- Most garbage collection schemes do not work with data structures that do not contain literal pointers;
- XOR of pointers is not defined in some contexts (e.g., the C language), although many languages provide some kind of type conversion between pointers and integers;
- The pointers will be unreadable if one isn't traversing the list — for example, if the pointer to a list item was contained in another data structure;
- While traversing the list you need to remember the address of the previously accessed node in order to calculate the next node's address.

Computer systems have increasingly cheap and plentiful memory, and storage overhead is not generally an overriding issue outside specialized embedded systems. Where it is still desirable to reduce the overhead of a linked list, unrolling provides a more practical approach (as well as other advantages, such as increasing cache performance and speeding random access).

Features

- Given only one list item, one cannot immediately obtain the addresses of the other elements of the list.
- Two XOR operations suffice to do the traversal from one item to the next, the same instructions sufficing in both cases. Consider a list with items {...B C D...} and with R1 and R2 being registers containing, respectively, the address of the current (say C) list item and a work register containing the XOR of the current address with the previous address (say C⊕D). Cast as System/360 instructions:

X R2,Link R2 <- C⊕D ⊕ B⊕D (i.e. B⊕C, "Link" being the link field in the current record, containing B⊕D) XR
R1,R2 R1 <- C ⊕ B⊕C (i.e. B, voilà: the next record)

- End of list is signified by imagining a list item at address zero placed adjacent to an end point, as in {0 A B C...}. The link field at A would be 0⊕B. An additional instruction is needed in the above sequence after the two XOR operations to detect a zero result in developing the address of the current item,
- A list end point can be made reflective by making the link pointer be zero. A zero pointer is a *mirror*. (The XOR of the left and right neighbor addresses, being the same, is zero.)

Why does it work?

The key is the first operation, and the properties of XOR:

- X⊕X=0
- X⊕0=X
- X⊕Y=Y⊕X
- (X⊕Y)⊕Z=X⊕(Y⊕Z)

The R2 register always contains the XOR of the address of current item C with the address of the predecessor item P: C⊕P. The Link fields in the records contain the XOR of the left and right successor addresses, say L⊕R. XOR of R2 (C⊕P) with the current link field (L⊕R) yields C⊕P⊕L⊕R.

- If the predecessor was L, the P(=L) and L *cancel out* leaving C⊕R.
- If the predecessor had been R, the P(=R) and R cancel, leaving C⊕L.

In each case, the result is the XOR of the current address with the next address. XOR of this with the current address in R1 leaves the next address. R2 is left with the requisite XOR pair of the (now) current address and the predecessor.

Variations

The underlying principle of the XOR linked list can be applied to any reversible binary operation. Replacing XOR by addition or subtraction gives slightly different, but largely equivalent, formulations:

Addition linked list

... A B C D E ... <-> A+C <-> B+D <-> C+E <-> This kind of list has exactly the same properties as the XOR linked list, except that a zero link field is not a "mirror". The address of the next node in the list is given by subtracting the previous node's address from the current node's link field.

Subtraction linked list

... A B C D E ... <-> C-A <-> D-B <-> E-C <-> This kind of list differs from the "traditional" XOR linked list in that the instruction sequences needed to traverse the list forwards is different from the sequence needed to traverse the list in reverse. The address of the next node, going forwards, is given by *adding* the link field to the previous node's address; the address of the preceding node is given by *subtracting* the link field from the next node's address.

The subtraction linked list is also special in that the entire list can be relocated in memory without needing any patching of pointer values, since adding a constant offset to each address in the list will not require any changes to the values stored in the link fields. (See also [Serialization](#).) This is an advantage over both XOR linked lists and traditional linked lists. The subtraction linked list also does not require casting C pointers to integers. The subtraction of two C pointers yields an integer. Note that it is still not portable to all platforms since subtraction of pointers is only guaranteed to be valid if the two pointers are elements in the same array or allocated block.

See also

- XOR swap algorithm

References

[1] <http://www.iecc.com/gclist/GC-faq.html#GC,%20C,%20and%20C++>

Unrolled linked list

In computer programming, an **unrolled linked list** is a variation on the linked list which stores multiple elements in each node. It can drastically increase cache performance, while decreasing the memory overhead associated with storing list metadata such as references. It is related to the B-tree.

Overview

A typical unrolled linked list node looks like this:

```
record node {
    node next      // reference to next node in list
    int numElements // number of elements in this node, up to
maxElements
    array elements // an array of numElements elements, with space
allocated for maxElements elements
}
```

Each node holds up to a certain maximum number of elements, typically just large enough so that the node fills a single cache line or a small multiple thereof. A position in the list is indicated by both a reference to the node and a position in the elements array. It's also possible to include a *previous* pointer for an unrolled doubly-linked linked list.

To insert a new element, we simply find the node the element should be in and insert the element into the elements array, incrementing numElements. If the array is already full, we first insert a new node either preceding or following the current one and move half of the elements in the current node into it.

To remove an element, similarly, we simply find the node it is in and delete it from the elements array, decrementing numElements. If numElements falls below $\text{maxElements} \div 2$ then we pull elements from adjacent nodes to fill it back up to this level. If both adjacent nodes are too low, we combine it with one adjacent node and then move some

values into the other. This is necessary to avoid wasting space.

Performance

One of the primary benefits of unrolled linked lists is decreased storage requirements. All nodes (except at most one) are at least half-full. If many random inserts and deletes are done, the average node will be about three-quarters full, and if inserts and deletes are only done at the beginning and end, almost all nodes will be full. Assume that:

- m = maxElements, the maximum number of elements in each elements array;
- v = the overhead per node for references and element counts;
- s = the size of a single element.

Then, the space used for n elements varies between $(v + sm)\lceil n/m \rceil$, roughly $(v/m + s)n$, and twice this value. For comparison, ordinary linked lists require $(v + s)n$ space, although v may be smaller, and arrays, one of the most compact data structures, require sn space. Unrolled linked lists effectively spread the overhead v over a number of elements of the list. Thus, we see the most significant space gain when overhead is large, maxElements is large, or elements are small.

If the elements are particularly small, such as bits, the overhead can be as much as 64 times larger than the data on many machines. Moreover, many popular memory allocators will keep a small amount of metadata for each node allocated, increasing the effective overhead v . Both these make unrolled linked lists more attractive.

Because unrolled linked list nodes each store a count next to the *next* field, retrieving the k th element of an unrolled linked list (indexing) can be done in $n/m + 1$ cache misses, up to a factor of m better than ordinary linked lists. Additionally, if the size of each element is small compared to the cache line size, the list can be iterated over in order with fewer cache misses than ordinary linked lists. In either case, operation time still increases linearly with the size of the list.

See also

- CDR coding, another technique for decreasing overhead and improving cache locality in linked lists similar to unrolled linked lists.
- the skip list, a similar variation on the linked list, also offers fast traversal but hurts the advantages of linked lists (quick insert/deletion) less than an unrolled linked list
- the B-tree and T-tree, data structures that are similar to unrolled linked lists in the sense that each of them could be viewed as an "unrolled binary tree"
- XOR linked list, a doubly linked list that uses one XORed pointer per node instead of two ordinary pointers.

VList

In computer science, the **VList** is a persistent data structure designed by Phil Bagwell in 2002 that combines the fast indexing of arrays with the easy extension of cons-based (or singly-linked) linked lists.^[1]

Like arrays, VLists have constant-time lookup on average and are highly compact, requiring only $O(\log n)$ storage for pointers, allowing them to take advantage of locality of reference. Like singly-linked or cons-based lists, they are persistent, and elements can be added to or removed from the front in constant time. Length can also be found in $O(\log n)$ time.

The primary operations of a VList are:

- Locate the k th element ($O(1)$ average, $O(\log n)$ worst-case)
- Add an element to the front of the VList ($O(1)$ average, with an occasional allocation)
- Obtain a new array beginning at the second element of an old array ($O(1)$)
- Compute the length of the list ($O(\log n)$)

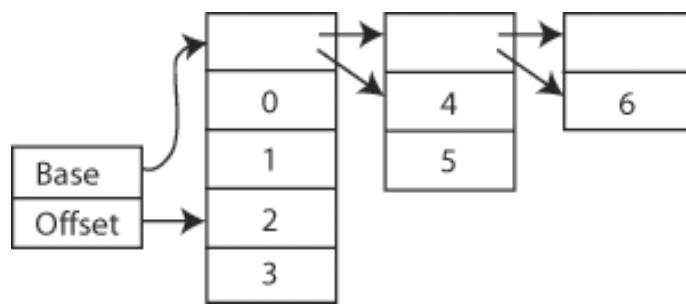
The primary advantages VLists have over arrays are that they are threadsafe (when locking is used; see the discussion page) and that different updated versions of the VList automatically share structure. Because VLists are immutable, they are most useful in functional programming languages, where their efficiency allows a purely functional implementation of data structures traditionally thought to require mutable arrays, such as hash tables.

However, VLists also have a number of disadvantages over their competitors:

- While immutability is a benefit, it is also a drawback, making it inefficient to modify elements in the middle of the array.
- Access near the end of the list can be as expensive as $O(\log n)$; it is only constant on average over all elements. This is still, however, much better than performing the same operation on cons-based lists.
- Wasted space in the first block is proportional to n . This is similar to linked lists, but there are data structures with less overhead. When used as a fully persistent data structure, the overhead may be considerably higher and this data structure may not be appropriate.

Structure

The underlying structure of a VList can be seen as a singly-linked list of arrays whose sizes decrease geometrically; in its simplest form, the first contains the first half of the elements in the list, the next the first half of the remainder, and so on. Each of these blocks stores some information such as its size and a pointer to the next.



An array-list. The reference shown refers to the VList (2,3,4,5,6).

The average constant-time indexing operation comes directly from this structure; given a random valid index, we simply observe the size of the blocks and follow pointers until we reach the one it should be in. The chance is 1/2 that it falls in the first block and we need not follow any pointers; the chance is 1/4 we have to follow only one, and so on, so that the expected number of pointers we have to follow is:

$$\sum_{i=1}^{\lceil \log_2 n \rceil} \frac{i-1}{2^i} < \sum_{i=1}^{\infty} \frac{i-1}{2^i} = 1.$$

Any particular reference to a VList is actually a $\langle base, offset \rangle$ pair indicating the position of its first element in the data structure described above. The *base* part indicates which of the arrays its first element falls in, while the *offset* part indicates its index in that array. This makes it easy to "remove" an element from the front of the list; we simply increase the offset, or increase the base and set the offset to zero if the offset goes out of range. If a particular reference is the last to leave a block, the block will be garbage-collected if such facilities are available, or otherwise must be freed explicitly.

Because the lists are constructed incrementally, the first array in the array list may not contain twice as many values as the next one, although the rest do; this does not significantly impact indexing performance. We nevertheless allocate this much space for the first array, so that if we add more elements to the front of the list in the future we can simply add them to this list and update the size. If the array fills up, we create a new array, twice as large again as this one, and link it to the old first array.

The trickier case, however, is adding a new item to the front of a list, call it A, which starts somewhere in the middle of the array-list data structure. This is the operation that allows VLists to be persistent. To accomplish this, we create a new array, and we link it to the array containing the first element of A. The new array must also store the offset of the first element of A in that array. Then, we can proceed to add any number of items we like to our new array, and any references into this new array will point to VLists which share a tail of values with the old array. Note that with this operation it is possible to create VLists which degenerate into simple linked lists, thus obliterating the performance claims made at the beginning of this article.

Variants

VList may be modified to support the implementation of a growable array. In the application of a growable array, immutability is no longer required. Instead of growing at the beginning of the list, the ordering interpretation is reversed to allow growing at the end of the array.

See also

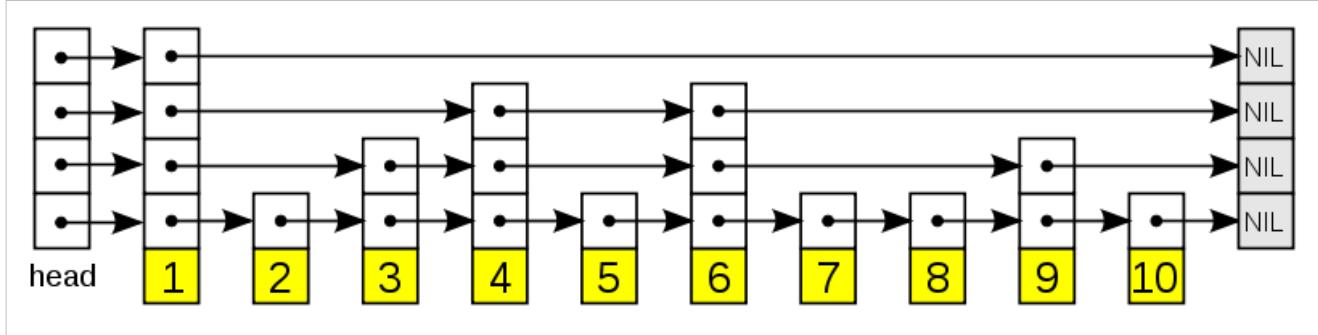
- Purely Functional Data Structures
- C++ implementation of VLists ^[2]
- An alternate link to PDF version of the paper ^[3]

References

- [1] Bagwell, Phil (2002), *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays* (<http://citeseer.ist.psu.edu/bagwell02fast.html>), EPFL,
 - [2] <http://www.oolt.org/doc/vlist.html>
 - [3] <http://infoscience.epfl.ch/record/64410/files/techlists.pdf>
-

Skip list

A **skip list** is a data structure for storing a sorted list of items, using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. These auxiliary lists allow item lookup with efficiency comparable to balanced binary search trees (that is, with number of probes proportional to $\log n$ instead of n).



Each link of the sparser lists skips over many items of the full list in one step, hence the structure's name. These forward links may be added in a randomized way with a geometric / negative binomial distribution^[1]. Insert, search and delete operations are performed in logarithmic expected time. The links may also be added in a non-probabilistic way so as to guarantee amortized (rather than merely expected) logarithmic cost.

Description

A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer $i+1$ with some fixed probability p (two commonly-used values for p are $1/2$ or $1/4$). On average, each element appears in $1/(1-p)$ lists, and the tallest element (usually a special head element at the front of the skip list) in $\log_{1/p} n$ lists.

A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is at most $1/p$, which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. Therefore, the total *expected* cost of a search is $(\log_{1/p} n)/p$, which is $\mathcal{O}(\log n)$ when p is a constant. By choosing different values of p , it is possible to trade search costs against storage costs.

Implementation details

The elements used for a skip list can contain more than one pointer since they can participate in more than one list.

Insertions and deletions are implemented much like the corresponding linked-list operations, except that "tall" elements must be inserted into or deleted from more than one linked list.

$\Theta(n)$ operations, which force us to visit every node in ascending order (such as printing the entire list), provide the opportunity to perform a behind-the-scenes derandomization of the level structure of the skip-list in an optimal way, bringing the skip list to $\mathcal{O}(\log n)$ search time. (Choose the level of the i 'th finite node to be 1 plus the number of times we can repeatedly divide i by 2 before it becomes odd. Also, $i=0$ for the negative infinity header as we have the usual special case of choosing the highest possible level for negative and/or positive infinite nodes.) However this also allows someone to know where all of the higher-than-level 1 nodes are and delete them.

Alternatively, we could make the level structure quasi-random in the following way:

```

make all nodes level 1
j ← 1
while the number of nodes at level j > 1 do
    for each i'th node at level j do
        if i is odd
            if i is not the last node at level j
                randomly choose whether to promote it to level j+1
            else
                do not promote
            end if
        else if i is even and node i-1 was not promoted
            promote it to level j+1
        end if
    repeat
    j ← j + 1
repeat

```

Like the derandomized version, quasi-randomization is only done when there is some other reason to be running a $\Theta(n)$ operation (which visits every node).

The advantage of this quasi-randomness is that it doesn't give away nearly as much level-structure related information to an adversarial user as the de-randomized one. This is desirable because an adversarial user who is able to tell which nodes are not at the lowest level can pessimize performance by simply deleting higher-level nodes. The search performance is still guaranteed to be logarithmic.

It would be tempting to make the following "optimization": In the part which says "Next, for each i'th...", forget about doing a coin-flip for each even-odd pair. Just flip a coin once to decide whether to promote only the even ones or only the odd ones. Instead of $\Theta(n \lg n)$ coin flips, there would only be $\Theta(\lg n)$ of them. Unfortunately, this gives the adversarial user a 50/50 chance of being correct upon guessing that all of the even numbered nodes (among the ones at level 1 or higher) are higher than level one. This is despite the property that he has a very low probability of guessing that a particular node is at level N for some integer N.

The following proves these two claims concerning the advantages of quasi-randomness over the totally derandomized version. First, to prove that the search time is guaranteed to be logarithmic. Suppose a node n is searched for, where n is the position of the found node among the nodes of level 1 or higher. If n is even, then there is a 50/50 chance that it is higher than level 1. However, if it is not higher than level 1 then node n-1 is guaranteed to be higher than level 1. If n is odd, then there is a 50/50 chance that it is higher than level 1. Suppose that it is not; there is a 50/50 chance that node n-1 is higher than level 1. Suppose that this is not either; we are guaranteed that node n-2 is higher than level 1. The analysis can then be repeated for nodes of level 2 or higher, level 3 or higher, etc. always keeping in mind that n is the position of the node among the ones of level k or higher for integer k. So the search time is constant in the best case (if the found node is the highest possible level) and 2 times the worst case for the search time for the totally derandomized skip-list (because we have to keep moving left twice rather than keep moving left once).

Next, an examination of the probability of an adversarial user's guess of a node being level k or higher being correct. First, the adversarial user has a 50/50 chance of correctly guessing that a particular node is level 2 or higher. This event is independent of whether or not the user correctly guesses at some other node being level 2 or higher. If the user knows the positions of two consecutive nodes of level 2 or higher, and knows that the one on the left is in an odd numbered position among the nodes of level 2 or higher, the user has a 50/50 chance of correctly guessing which one is of level 3 or higher. So, the user's probability of being correct, when guessing that a node is level 3 or higher, is 1/4. Inductively continuing this analysis, we see that the user's probability of guessing that a particular node is

level k or higher is $1/(2^{k-1})$.

The above analyses only work when the number of nodes is a power of two. However, because of the third rule which says, "Finally, if i is odd and also the last node at level 1 then do not promote." (where we substitute the appropriate level number for 1) it becomes a sequence of exact-power-of-two-sized skiplists, concatenated onto each other, for which the analysis does work. In fact, the exact powers of two correspond to the binary representation for the number of nodes in the whole list.

A skip list, upon which we have not recently performed either of the above mentioned $\Theta(n)$ operations, does not provide the same absolute worst-case performance guarantees as more traditional balanced tree data structures, because it is always possible (though with very low probability) that the coin-flips used to build the skip list will produce a badly balanced structure. However, they work well in practice, and the randomized balancing scheme has been argued to be easier to implement than the deterministic balancing schemes used in balanced binary search trees. Skip lists are also useful in parallel computing, where insertions can be done in different parts of the skip list in parallel without any global rebalancing of the data structure. Such parallelism can be especially advantageous for resource discovery in an ad-hoc Wireless network because a randomized skip list can be made robust to the loss of any single node^[2].

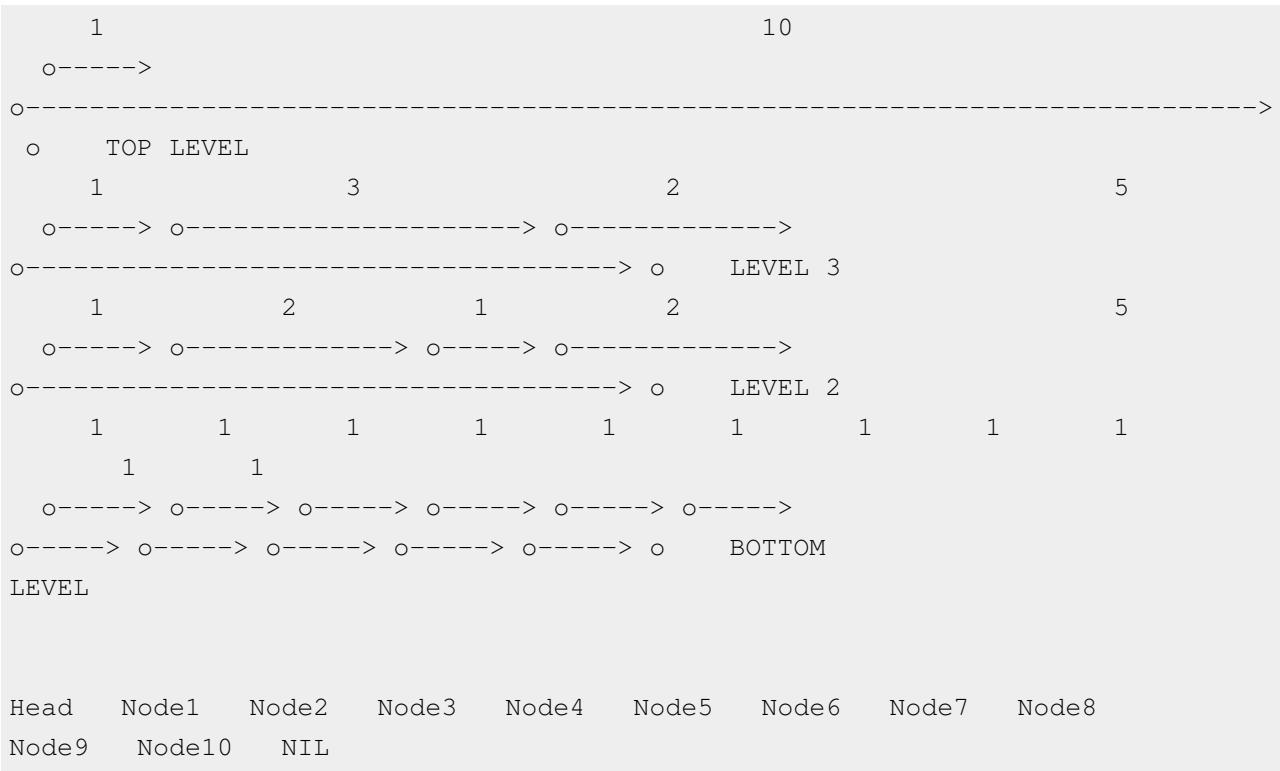
There has been some evidence that skip lists have worse real-world performance and space requirements than B trees due to memory locality and other issues.^[3]

Indexable skiplist

As described above, a skiplist is capable of fast $\Theta(\log n)$ insertion and removal of values from a sorted sequence, but it has only slow $\Theta(n)$ lookups of values at a given position in the sequence (i.e. return the 500th value); however, with a minor modification the speed of random access indexed lookups can be improved to $\Theta(\log n)$.

For every link, also store the width of the link. The width is defined as the number of bottom layer links being traversed by each of the higher layer "express lane" links.

For example, here are the widths of the links in the example at the top of the page:



Notice that the width of a higher level link is the sum of the component links below it (i.e. the width 10 link spans the links of widths 3, 2 and 5 immediately below it). Consequently, the sum of all widths is the same on every level ($10 + 1 = 1 + 3 + 2 + 5 = 1 + 2 + 1 + 2 + 5$).

To index the skiplist and find the i^{th} value, traverse the skiplist while counting down the widths of each traversed link. Descend a level whenever the upcoming width would be too large.

For example, to find the node in the fifth position (Node 5), traverse a link of width 1 at the top level. Now four more steps are needed but the next width on this level is ten which is too large, so drop one level. Traverse one link of width 3. Since another step of width 2 would be too far, drop down to the bottom level. Now traverse the final link of width 1 to reach the target running total of 5 (1+3+1).

```
function lookupByPositionIndex(i)
    node ← head
    i ← i + 1                                # don't count the head as a
step
    for level from top to bottom do
        while i ≥ node.width[level] do    # if next step is not too far
            i ← i - node.width[level]    # subtract the current width
            node ← node.next[level]      # traverse forward at the
current level
        repeat
    repeat
    return node.value
end function
```

This method of implementing indexing is detailed in Section 3.4 Linear List Operations in "A skip list cookbook" by William Pugh [4].

Also, see Running Median using an Indexable Skiplist [5] for a complete implementation written in Python and for an example of it being used to solve a computationally intensive statistics problem. And see Regaining Lost Knowledge [6] for the history of that solution.

History

Skip lists were invented in 1990 by William Pugh. He details how they work in *Skip lists: a probabilistic alternative to balanced trees* in Communications of the ACM, June 1990, 33(6) 668-676. See also citations [7] and downloadable documents [8].

To quote the inventor:

Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.

Usages

List of applications and frameworks that use skip lists:

- QMap^[9] template class of Qt that provides a dictionary.
- skipdb^[10] is an open-source database format using ordered key/value pairs.
- Running Median using an Indexable Skiplist^[5] is a Python implementation of a skiplist augmented by link widths to make the structure indexable (giving fast access to the nth item). The indexable skiplist is used to efficiently solve the running median problem (recomputing medians and quartiles as values are added and removed from a large sliding window).

See also

- Deterministic skip list

External links

- Skip Lists: A Probabilistic Alternative to Balanced Trees^[11] - William Pugh's original paper
- "Skip list" entry^[12] in the Dictionary of Algorithms and Data Structures
- Lecture 12: Skip Lists^[13] of Introduction to Algorithms 2005 on MIT OpenCourseWare
- Skip Lists: A Linked List with Self-Balancing BST-Like Properties^[14] on MSDN in C# 2.0
- SkipDB, a BerkeleyDB-style database implemented using skip lists.^[15]

Demo applets

- Skip List Applet^[16] by Kubo Kovac
- Thomas Wenger's demo applet on skiplists^[17]

Implementations

- A generic Skip List in C++^[18] by Antonio Gulli
- Algorithm::SkipList, implementation in Perl on CPAN^[19]
- John Shipman's didactic zero-defect implementation in Python^[20]
- ConcurrentSkipListSet documentation for Java 6^[21] (and sourcecode^[22])

References

- [1] Pugh, William (June 1990). "Skip lists: a probabilistic alternative to balanced trees". *Communications of the ACM* **33** (6): 668–676. doi:10.1145/78973.78977.
- [2] Shah, Gauri Ph.D.; James Aspnes (December 2003) (PDF). *Distributed Data Structures for Peer-to-Peer Systems* (<http://www.cs.yale.edu/~homes/shah/pubs/thesis.pdf>). . Retrieved 2008-09-23.
- [3] http://resnet.uoregon.edu/~gurney_j/jmpc/skiplist.html
- [4] <http://cg.scs.carleton.ca/~morin/teaching/5408/refs/p90b.pdf>
- [5] <http://code.activestate.com/recipes/576930/>
- [6] <http://rhettinger.wordpress.com/2010/02/06/lost-knowledge/>
- [7] <http://citeseer.ist.psu.edu/pugh90skip.html>
- [8] <ftp://ftp.cs.umd.edu/pub/skipLists/>
- [9] <http://doc.trolltech.com/4.5/qmap.html#details>
- [10] <http://www.dekorte.com/projects/opensource/skipdb/>
- [11] <ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf>
- [12] <http://nist.gov/dads/HTML/skiplist.html>
- [13] <http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/VideoLectures/detail/embed12.htm>
- [14] [http://msdn.microsoft.com/en-us/library/ms379573\(VS.80\).aspx#datastructures20_4_topic4](http://msdn.microsoft.com/en-us/library/ms379573(VS.80).aspx#datastructures20_4_topic4)
- [15] <http://dekorte.com/projects/opensource/SkipDB/>
- [16] <http://people.ksp.sk/~kuko/bak/index.html>
- [17] <http://iamwww.unibe.ch/~wenger/DA/SkipList/>
- [18] <http://codingplayground.blogspot.com/2009/01/generic-skip-list-skiplist.html>

- [19] <http://search.cpan.org/~rrwo/Algorithm-SkipList-1.02/>
- [20] <http://infohost.nmt.edu/tcc/help/lang/python/examples/pyskip/>
- [21] <http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListSet.html>
- [22] <http://www.docjar.com/html/api/java/util/concurrent/ConcurrentSkipListSet.java.html>

Jump list

In computer science, a **jump list** is a data structure which resembles an ordered doubly linked list. Instead of only *next* and *previous* links, several nodes contain links to nodes farther away, with the distance increasing geometrically. This allows the dictionary operations *search*, *insert* and *delete* to be executed in $O(\sqrt[3]{n})$ time.

See also

- Dictionary
- Linked list
- Skip list

References

-  This article incorporates public domain material from the NIST document "jump list" ^[1] by Paul E. Black (Dictionary of Algorithms and Data Structures).
- Arne Andersson and Thomas Ottmann, *New Tight Bounds on Uniquely Represented Dictionaries* ^[2], SIAM Journal of Computing, 24(5):1091-1103, 1995.

References

- [1] <http://www.nist.gov/dads/HTML/jumpList.html>
- [2] <http://user.it.uu.se/~arnea/ps/andetigh.ps>

Self-organizing list

A **self-organizing list** is a list that reorders its elements based on some self-organizing heuristic to improve average access time.

Some say, the "Self Organizing List" is a poor man's Hash table (see Gwydion Dylan Library Reference Guide ^[1]). By using a probabilistic strategy, it yields nearly constant time in the best case for insert/delete operations, although the worst case remains linear.

References

- NIST DADS entry ^[2]

there are four ways in which the list can be self organized: 1. ordering 2. transpose 3. move to front 4. count

References

[1] <http://www.opendylan.org/gdref/gdlibs/libs-collection-extensions-organized-list.html>

[2] <http://www.nist.gov/dads/HTML/selfOrganizingList.html>

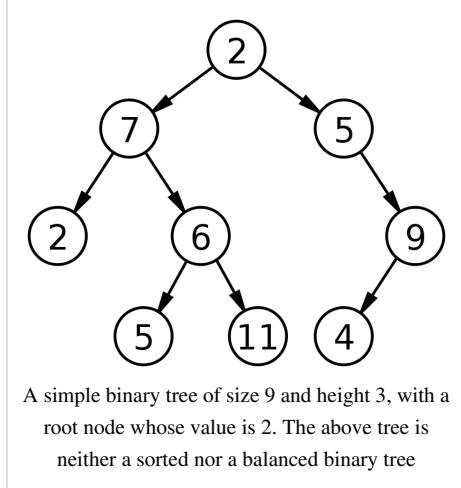
Binary trees

Binary tree

In computer science, a **binary tree** is a tree data structure in which each node has at most two children. Typically the first node is known as the parent and the child nodes are called *left* and *right*. In type theory, a binary tree with nodes of type A is defined inductively as $T_A = \mu\alpha. 1 + A \times \alpha \times \alpha$. Binary trees are commonly 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 (or one^[1]).
- The **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 a node (the root) has a height of zero (or one^[2]).
- **Siblings** are nodes that share the same parent node.
- If a path exists from node p to node q, where node p is closer to the root node than q, then p is an **ancestor** of q and q is 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.
- Root is the only node in the tree with In-degree = 0.



Types of binary trees

- A **rooted binary tree** is a **rooted** tree 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.
- A **perfect binary tree** is a *full binary tree* in which all *leaves* are at the same *depth* or same *level*.^[3] (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.^[4]
- An **infinite complete binary tree** is a tree with \aleph_0 levels, where for each level d the number of existing nodes at level d is equal to 2^d . The cardinal number of the set of all nodes is \aleph_0 . The cardinal number of the set of all paths is 2^{\aleph_0} or equivalently, assuming the axiom of choice, \aleph_1 . (See Continuum hypothesis.)
- A **balanced binary tree** is where the depth of all the *leaves* differs by at most 1. Balanced trees 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 balanced tree with 5 nodes, $\log_2(5) = 2.32$ (depth of tree is 2 nodes).

- A **rooted complete binary tree** can be identified with a free magma.
- 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.

A rooted tree has a top node as root.

Properties of binary trees

- The number of nodes n in a perfect binary tree can be found using this formula: $n = 2^{h+1} - 1$ where h is the height of the tree.
- The number of nodes n in a complete binary tree is minimum: $n = 2^h$ and maximum: $n = 2^{h+1} - 1$ where h is the height of the tree.
- The number of nodes n 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 leaf nodes n in a perfect binary tree can be found using this formula: $n = 2^h$ where h is the height of the tree.
- The number of NULL links in a Complete Binary Tree of n -node is $(n+1)$.
- The number of leaf node in a Complete Binary Tree of n -node is $UpperBound(n/2)$.
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.^[5]
- Note that this terminology often varies in the literature, especially with respect to the meaning "complete" and "full".

Definition 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

The groupings of pairs of nodes in a tree can be represented as pairs of letters, surrounded by parenthesis. Thus, $(a b)$ denotes the binary tree whose left subtree is a and whose right subtree is b . Strings of balanced pairs of parenthesis may therefore be used to denote binary trees in general. The set of all possible strings consisting entirely of balanced parentheses is known as the Dyck language.

Given n nodes, the total number of ways in which these nodes can be arranged into a binary tree is given by the Catalan number C_n . For example, $C_2 = 2$ declares that $(a 0)$ and $(0 a)$ are the only binary trees possible that have two nodes, and $C_3 = 5$ declares that $((a 0) 0)$, $(0 a) 0)$, $(0 (a 0))$, $(0 (0 a))$, and $(a b)$ are the only five binary trees possible that have 3 nodes. Here 0 represents a subtree that is not present.

The ability to represent binary trees as strings of symbols and parentheses implies that binary trees can represent the elements of a magma. Conversely, the set of all possible binary trees, together with the natural operation of attaching trees to one-another, forms a magma, the free magma.

Given a string representing a binary tree, the operators to obtain the left and right subtrees are sometimes referred to as car and cdr.

Methods for storing binary trees

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

Nodes and references

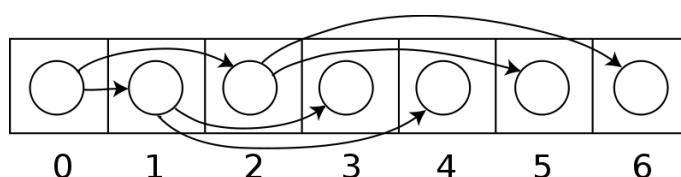
In a language with records and references, binary trees are typically constructed by having a tree node structure which contains some data and references to its left child and its right child. Sometimes it also contains a reference to its unique parent. If a node has fewer than two children, some of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

Ahnentafel list

Binary trees can also be stored 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 height h with n nodes.



A binary tree can also be represented in the form of array as well as adjacency linked list. In the case of array, each node(root, left, right) is simply placed in the index and there is no connection mentioned about the relationship between parents and children. But In linked list representation we can find the relationship between parent and children. In array representation the nodes are accessed by calculating the index. This method is used in languages like FORTRAN which doesn't have dynamic memory allocation. We can't insert a new node into array implemented

binary tree with ease, but it is easily possible in binary tree implemented with linked list.

Methods of iterating over binary trees

Often, one wishes to visit each of the nodes in a tree and examine the value there. 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.

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. If the root node is visited before its subtrees, this is pre-order; if after, post-order; if between, in-order. In-order traversal is useful in binary search trees, where this traversal visits the nodes in increasing order.

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*.

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. [6] 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
}

```

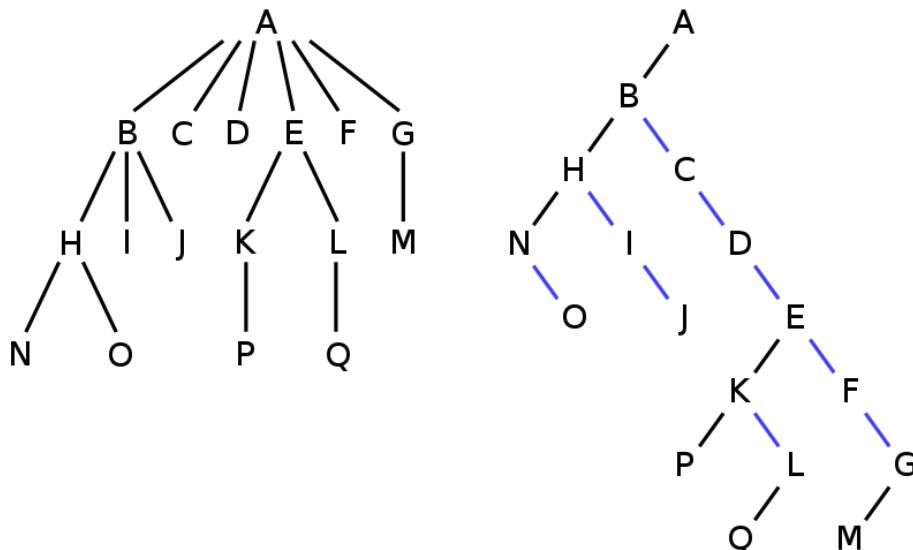
More sophisticated succinct representations allow not only compact storage of trees but even useful operations on those trees directly while they're still in their succinct form.

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 First-Child/Next-Sibling binary 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.

See also

- 2-3 tree
- 2-3-4 tree
- AA tree
- B-tree
- Binary space partitioning
- Binary tree of preimages in Julia set (IIM/J)
- Kraft's inequality
- Random binary tree
- Recursion (computer science)
- Strahler number
- Threaded binary tree

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 ^[7] — opensource library
- [7] — GameDev.net's article about binary trees
- [8] — Binary Tree Proof by Induction
- Balance binary search tree on array ^[9] How to create bottom-up an Ahnentafel list, or a balanced binary search tree on array
- Binary tree ^[10] orcik Blog

References

- [1] <http://www.laynetworks.com/cs04.htm>
- [2] <http://www.laynetworks.com/cs04.htm>
- [3] "perfect binary tree" (<http://www.nist.gov/dads/HTML/perfectBinaryTree.html>). NIST. .
- [4] "complete binary tree" (<http://www.nist.gov/dads/HTML/completeBinaryTree.html>). NIST. .
- [5] Mehta, Dinesh; Sartaj Sahni (2004). *Handbook of Data Structures and Applications*. Chapman and Hall. ISBN 1584884355.
- [6] http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L12/lecture12.pdf
- [7] <http://www.gamedev.net/reference/programming/features/trees2/>
- [8] <http://www.brpreiss.com/books/opus4/html/page355.html>
- [9] <http://piergiu.wordpress.com/2010/02/21/balanced-binary-search-tree-on-array/>
- [10] <http://orcik.net/binary-tree/>

Binary search tree

In computer science, a **binary search tree (BST)** 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 or equal than the node's key.
- Both the left and right subtrees must also be binary search trees.

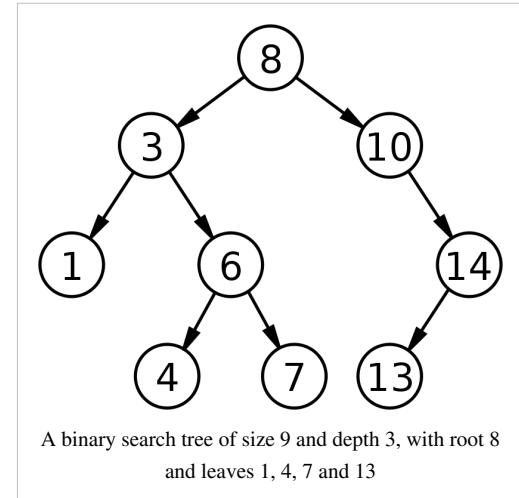
From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.

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 tree require comparisons between nodes. These comparisons are made with calls to a comparator, which is a subroutine that computes the total order (linear order) on any two values. This comparator can be explicitly or implicitly defined, depending on the language in which the BST is implemented.

Searching

Searching a binary tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method.

We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

Here is the search algorithm in the Python programming language:

```
# 'node' refers to the parent-node in this case
def search_binary_tree(node, key):
    if node is None:
        return None # key not found
    if key < node.key:
        return search_binary_tree(node.leftChild, key)
    elif key > node.key:
        return search_binary_tree(node.rightChild, key)
    else: # key is equal to node key
```

```
return node.value # found key
```

... or equivalent Haskell:

```
searchBinaryTree _ NullNode = Nothing
searchBinaryTree key (Node nodeKey nodeValue (leftChild, rightChild)) =
  case compare key nodeKey of
    LT -> searchBinaryTree key leftChild
    GT -> searchBinaryTree key rightChild
    EQ -> Just nodeValue
```

This operation requires $O(\log n)$ time in the average case, but needs $O(n)$ time in the worst-case, when the unbalanced tree resembles a linked list (degenerate tree).

Assuming that `BinarySearchTree` is a class with a member function "search(int)" and a pointer to the root node, the algorithm is also easily implemented in terms of an iterative approach. The algorithm enters a loop, and decides whether to branch left or right depending on the value of the node at each parent node.

```
bool BinarySearchTree::search(int val)
{
    Node* next = this->root();

    while (next != 0)
    {
        if (val == next->value())
        {
            return true;
        }
        else if (val < next->value())
        {
            next = next->left();
        }
        else if (val > next->value())
        {
            next = next->right();
        }
    }

    //not found
    return false;
}
```

Here is the search algorithm in the Java programming language:

```
public boolean search(TreeNode node, int data) {
    if (node == null) {
        return false;
    }
    if (node.getData() == data) {
        return true;
    } else if (data < node.getData()) {
```

```

        // data must be in left subtree
        return search(node.getLeft(), data);
    } else {
        // data must be in right subtree
        return search(node.getRight(), data);
    }
}

```

Insertion

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

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

```

/* Inserts the node pointed to by "newNode" into the subtree rooted at
"treeNode" */
void InsertNode(Node* &treeNode, Node *newNode)
{
    if (treeNode == NULL)
        treeNode = newNode;
    else if (newNode->key < treeNode->key)
        InsertNode(treeNode->left, newNode);
    else
        InsertNode(treeNode->right, newNode);
}

```

The above "destructive" procedural variant modifies the tree in place. It uses only constant space, but the previous version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

```

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 $\Omega(n)$ 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 $\Omega(n)$ time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

Here is an iterative approach to inserting into a binary search tree the in Java Programming Language:

```
public void insert(int data) {
    if (root == null) {
        root = new TreeNode(data, null, null);
    } else {
        TreeNode current = root;
        while (current != null) {
            if (data < current.getData()) {
                // insert left
                if (current.getLeft() == null) {
                    current.setLeft(new TreeNode(data, null,
null));
                    return;
                } else {
                    current = current.getLeft();
                }
            } else {
                // insert right
                if (current.getRight() == null) {
                    current.setRight(new TreeNode(data, null,
null));
                    return;
                } else {
                    current = current.getRight();
                }
            }
        }
    }
}
```

Below is a recursive approach to the insertion method. As pointers are not available in Java we must return a new node pointer to the caller as indicated by the final line in the method.

```
public TreeNode insert(TreeNode node, int data) {
    if (node == null) {
        node = new TreeNode(data, null, null);
    } else {
        if (data < node.getData()) {
            // insert left
            node.left = insert(node.getLeft(), data);
        } else {
            if (data > node.getData()) {
                // insert right
                node.right = insert(node.getRight(), data);
            }
        }
    }
    return node;
}
```

```

    } else {
        // insert right
        node.right = insert(node.getRight(), data);
    }
}

return node;
}

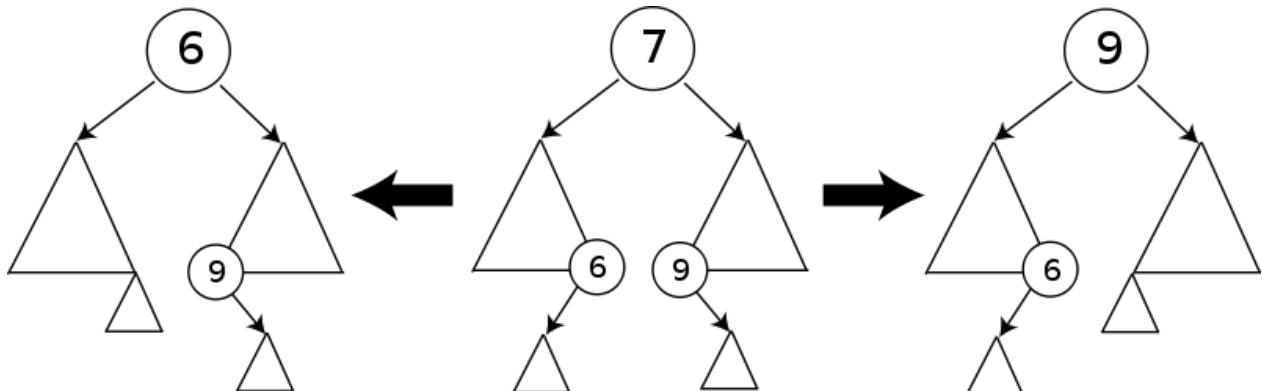
```

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:** Delete it 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. (Note: R itself has up to one child.)

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 == 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)

```

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

Sort

A binary search tree can be used to implement a simple but efficient sorting algorithm. Similar to heapsort, we insert all the values we wish to sort into a new ordered data structure—in this case a binary search tree—and then traverse it in order, building our result:

```
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 $\Theta(n^2)$ —if you feed it a sorted list of values, it chains them into a linked list with no left subtrees. For example, `traverse_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 priority and the parent node has higher priority than its children.

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 fluctuations.

Optimal binary search trees

If we don't plan on modifying a search tree, and we know exactly how often each item will be accessed, we can construct an **optimal binary search tree**, which is a search tree where the average cost of looking up an item (the *expected search cost*) is minimized.

Even if we only have estimates of the search costs, such a system can considerably speed up lookups on average. For example, if you have a BST of English words used in a spell checker, you might balance the tree based on word frequency in text corpora, placing words like "the" near the root and words like "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]
```

See also

- Binary search
- Binary tree
- Self-balancing binary search tree
- Randomized binary search tree
- B-tree
- Data structure
- Trie
- Ternary search tree
- Elastic binary tree
- Hash table
- Skip list

Further reading

- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 6.2.2: Binary Tree Searching, pp.426–458.
- 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 12: Binary search trees, pp.253–272. Section 15.5: Optimal binary search trees, pp.356–363.

External links

- Full source code to an efficient implementation in C++ ^[3]
- Implementation of Binary Search Trees in C ^[4]
- Implementation of a Persistent Binary Search Tree in C ^[5]
- Implementation of Binary Search Trees in Java ^[6]
- Iterative Implementation of Binary Search Trees in C# ^[7]
- An introduction to binary trees from Stanford ^[8]
- Dictionary of Algorithms and Data Structures - Binary Search Tree ^[9]
- Binary Search Tree Example in Python ^[10]
- Java Model illustrating the behaviour of binary search trees(In JAVA Applet) ^[11]
- Interactive Data Structure Visualizations - Binary Tree Traversals ^[12]
- Literate implementations of binary search trees in various languages ^[13] on LiteratePrograms
- BST Tree Applet ^[16] by Kubo Kovac
- Well-illustrated explanation of binary search tree. Implementations in Java and C++ ^[14]

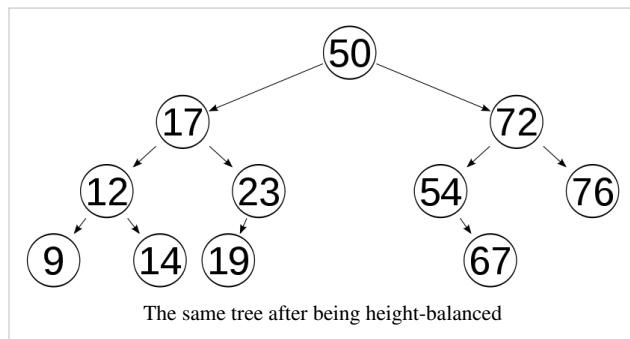
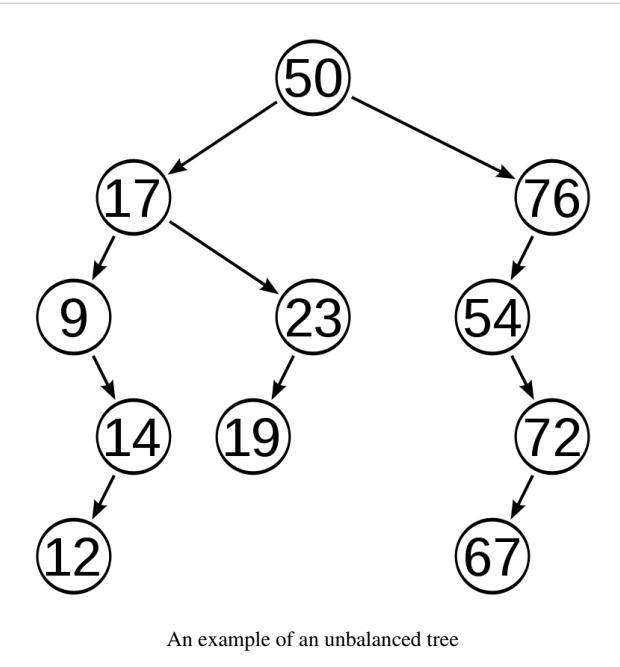
References

- [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.upgrade-cepis.org/issues/2004/5/up5-5Mosaic.pdf>), *European Journal for the Informatics Professional* 5 (5),
- [3] http://jdserver.homelinux.org/wiki/Binary_Search_Tree
- [4] http://wiki.portugal-a-programar.org/c:snippet:binary_search_tree
- [5] <http://cg.scs.carleton.ca/~dana/pbst>
- [6] <http://www.24bytes.com/Binary-Search-Tree.html>
- [7] <http://www.goletas.com/solutions/collections/>
- [8] <http://cslibrary.stanford.edu/110/>
- [9] <http://www.nist.gov/dads/HTML/binarySearchTree.html>
- [10] <http://aspn.activestate.com/ASPNet/Cookbook/Python/Recipe/286239>
- [11] <http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm>
- [12] <http://nova.umuc.edu/~jarc/idsv/lesson1.html>
- [13] http://en.literateprograms.org/Category:Binary_search_tree
- [14] http://www.algolist.net/Data_structures/Binary_search_tree

Self-balancing binary search tree

In computer science, a **self-balancing** (or **height-balanced**) **binary search tree** is any binary search tree data structure that automatically keeps its height (number of levels below the root) small in the face of arbitrary item insertions and deletions.^[1]

These structures provide efficient implementations for mutable ordered lists, and can be used for other abstract data structures such as associative arrays, priority queues and sets.



Overview

Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. Since a binary tree with height h contains at most $2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$ nodes, it follows that the minimum height of a tree with n nodes is $\log_2(n)$, rounded down; that is, $\lfloor \log_2 n \rfloor$.^[1]

However, the simplest algorithms for BST item insertion may yield a tree with height n in rather common situations. For example, when the items are inserted in sorted key order, the tree degenerates into a linked list with n nodes. The difference in performance between the two situations may be enormous: for $n = 1,000,000$, for example, the minimum height is $\lfloor \log_2(1,000,000) \rfloor = 19$.

If the data items are known ahead of time, the height can be kept small, in the average sense, by adding values in a random order, resulting in a random binary search tree. However, there are many situations (such as online algorithms) where this randomization is not viable.

Self-balancing binary trees solve this problem by performing transformations on the tree (such as tree rotations) at key times, in order to keep the height proportional to $\log_2(n)$. Although a certain overhead is involved, it may be justified in the long run by ensuring fast execution of later operations.

Maintaining the height always at its minimum value $\lfloor \log_2(n) \rfloor$ is not always viable; it can be proven that any insertion algorithm which did so would have an excessive overhead. Therefore, most self-balanced BST algorithms keep the height within a constant factor of this lower bound.

In the asymptotic ("Big-O") sense, a self-balancing BST structure containing n items allows the lookup, insertion, and removal of an item in $O(\log n)$ worst-case time, and ordered enumeration of all items in $O(n)$ time. For some implementations these are per-operation time bounds, while for others they are amortized bounds over a sequence of operations. These times are asymptotically optimal among all data structures that manipulate the key only through comparisons.

Implementations

Popular data structures implementing this type of tree include:

- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

Applications

Self-balancing binary search trees can be used in a natural way to construct and maintain ordered lists, such as priority queues. They can also be used for associative arrays; key-value pairs are simply inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables. One advantage of self-balancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items *in key order*, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key.

Self-balancing BSTs can be used to implement any algorithm that requires mutable ordered lists, to achieve optimal worst-case asymptotic performance. For example, if binary tree sort is implemented with a self-balanced BST, we have a very simple-to-describe yet asymptotically optimal $O(n \log n)$ sorting algorithm. Similarly, many algorithms in computational geometry exploit variations on self-balancing BSTs to solve problems such as the line segment intersection problem and the point location problem efficiently. (For average-case performance, however, self-balanced BSTs may be less efficient than other solutions. Binary tree sort, in particular, is likely to be slower than mergesort or quicksort, because of the tree-balancing overhead as well as cache access patterns.)

Self-balancing BSTs are flexible data structures, in that it's easy to extend them to efficiently record additional information or perform new operations. For example, one can record the number of nodes in each subtree having a certain property, allowing one to count the number of nodes in a certain key range with that property in $O(\log n)$ time. These extensions can be used, for example, to optimize database queries or other list-processing algorithms.

See also

- DSW algorithm
- Fusion tree
- Skip list
- Sorting

External links

- Dictionary of Algorithms and Data Structures: Height-balanced binary search tree ^[2]
- GNU libavl ^[3], a LGPL-licensed library of binary tree implementations in C, with documentation

References

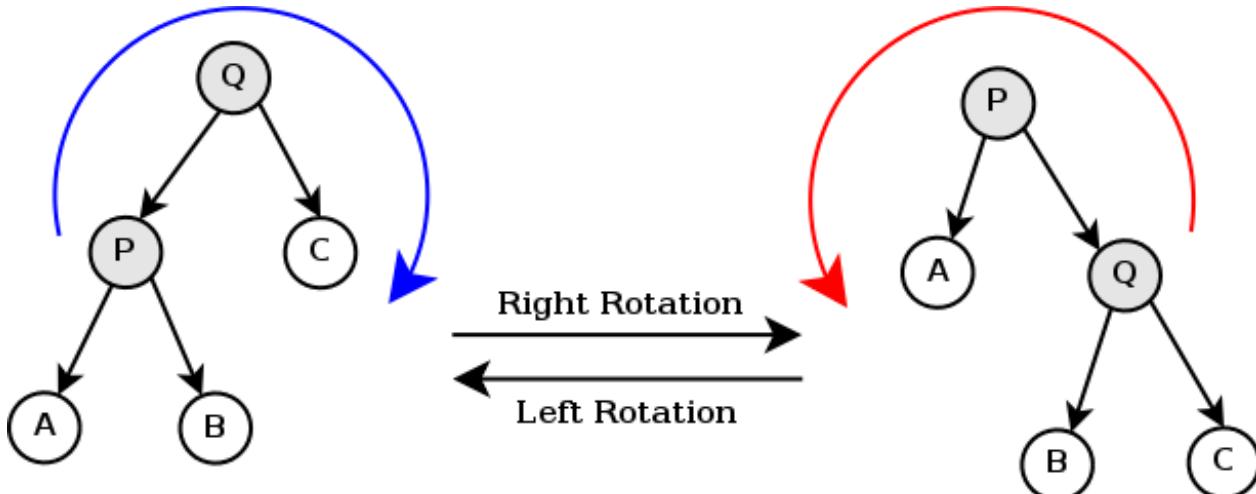
- [1] Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 6.2.3: Balanced Trees, pp.458–481.
- [2] <http://www.nist.gov/dads/HTML/heightBalancedTree.html>
- [3] <http://adtinfo.org/>

Tree rotation

A **tree rotation** is an operation on a binary tree that changes the structure without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down. They are used to change the shape of the tree, and in particular to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations.

This article takes the approach of the side where the nodes get shifted is the direction of the rotation.

Illustration



The right rotation operation as shown in the image above is performed with Q as the root and hence is a right rotation on, or rooted at, Q . This operation results in a rotation of the tree in the clockwise direction. The symmetric operation is the left rotation which results in a movement in an counter-clockwise direction (the left rotation shown above is rooted at P).

Assuming this is a binary search tree, as stated above, the elements must be interpreted as variables and not as alphabetic characters.

Detailed Illustration

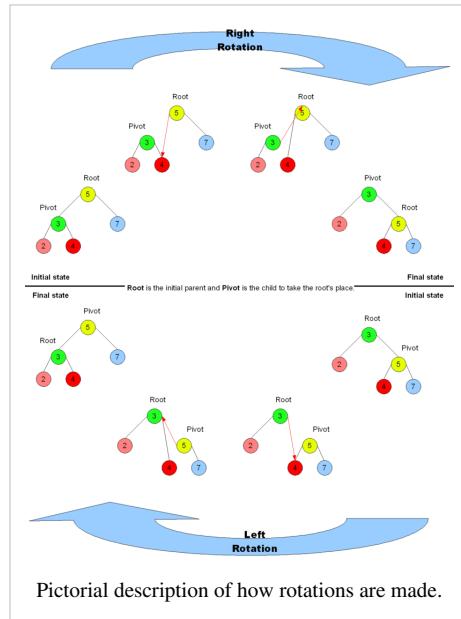
When a subtree is rotated, the subtree side upon which it is rotated decreases its height by one node while the other subtree increases its height. This makes it useful for rebalancing a tree.

Using the terminology of **Root** for the parent node of the subtrees to rotate, **Pivot** for the node which will become the new parent node, **RS** for rotation side upon to rotate and **OS** for opposite side of rotation. In the above diagram for the root Q, the **RS** is C and the **OS** is P. The pseudo code for the rotation is:

```
Pivot = Root.OS
Root.OS = Pivot.RS
Pivot.RS = Root
Root = Pivot
```

This is a constant time operation.

The programmer must also make sure that the root's parent points to the pivot after the rotation. Also, the programmer should note that this operation may result in a new root for the entire tree and take care to update pointers accordingly.



Inorder Invariance

The tree rotation renders the inorder traversal of the binary tree invariant. This implies the order of the elements are not affected when a rotation is performed in any part of the tree. Here are the inorder traversals of the trees shown above:

Left tree: ((A, P, B), Q, C)	Right tree: (A, P, (B, Q, C))
------------------------------	-------------------------------

Computing one from the other is very simple. The following is example Python code that performs that computation:

```
def right_rotation(treenode):
    left, Q, C = treenode
    A, P, B = left
    return (A, P, (B, Q, C))
```

Another way of looking at it is:

Left Rotation of node P:

```
Let Q be P's right child.
Set Q to be the new root.
Set P's right child to be Q's left child.
Set Q's left child to be P.
```

Right Rotation of node Q:

```
Let P be Q's left child.
Set P to be the new root.
Set Q's left child to be P's right child.
Set P's right child to be Q.
```

All other connections are left as-is.

There are also *double rotations*, which are combinations of left and right rotations. A *double left* rotation at X can be defined to be a right rotation at the right child of X followed by a left rotation at X; similarly, a *double right* rotation at X can be defined to be a left rotation at the left child of X followed by a right rotation at X.

Tree rotations are used in a number of tree data structures such as AVL trees, red-black trees, splay trees, and treaps. They require only constant time because they are *local* transformations: they only operate on 5 nodes, and need not examine the rest of the tree.

Rotations for rebalancing

A tree can be rebalanced using rotations. After a rotation, the side of the rotation increases its height by 1 whilst the side opposite the rotation decreases its height similarly. Therefore, one can strategically apply rotations to nodes whose left child and right child differ in height by more than 1. Self-balancing binary search trees apply this operation automatically. A type of tree which uses this rebalancing technique is the AVL tree.



Pictorial description of how rotations cause rebalancing in an AVL tree.

Rotation distance

The **rotation distance** between any two binary trees with the same number of nodes is the minimum number of rotations needed to transform one into the other. With this distance, the set of n -node binary trees becomes a metric space: the distance is symmetric, positive when given two different trees, and satisfies the triangle inequality.

It is an open problem whether there exists a polynomial time algorithm for calculating rotation distance. However, Daniel Sleator, Robert Tarjan and William Thurston showed that the rotation distance between any two n -node trees (for $n \geq 11$) is at most $2n - 6$, and that infinitely many pairs of trees are this far apart.^[1]

External links

- Java applets demonstrating tree rotations^[2]
- The AVL Tree Rotations Tutorial^[3] (RTF) by John Hargrove

See also

- AVL tree, red-black tree, and splay tree, kinds of binary search tree data structures that use rotations to maintain balance.
- Associativity of a binary operation means that performing a tree rotation on it does not change the final result.
- Tamari lattice, a partially ordered set in which the elements can be defined as binary trees and the ordering between elements is defined by tree rotation.

References

- [1] Sleator, Daniel D.; Tarjan, Robert E.; Thurston, William P. (1988), "Rotation distance, triangulations, and hyperbolic geometry", *Journal of the American Mathematical Society* 1 (3): 647–681, doi:10.2307/1990951, MR928904.
- [2] <http://www.cs.queensu.ca/home/jstewart/applets/bst/bst-rotation.html>
- [3] <http://fortheloot.com/public/AVLTreeTutorial.rtf>

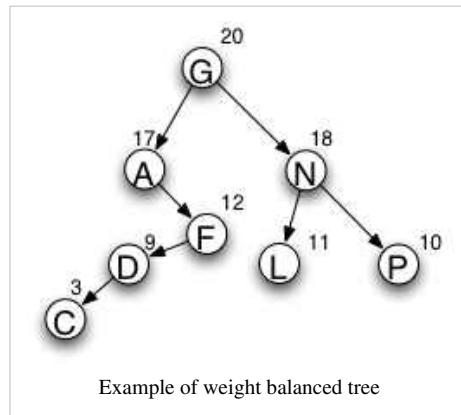
Weight-balanced tree

A **weight-balanced binary tree** is a binary tree which is balanced based on knowledge of the probabilities of searching for each individual node. Within each subtree, the node with the highest weight appears at the root. This can result in more efficient searching performance.

Construction of such a tree is similar to that of a Treap, but node weights are chosen randomly in the latter.

The diagram

In the diagram to the right, the letters represent node *values* and the numbers represent node *weights*. Values are used to order the tree, as in a general binary search tree. The weight may be thought of as a probability or activity count associated with the node. In the diagram, the root is G because its weight is the greatest in the tree. The left subtree begins with A because, out of all nodes with values that come before G, A has the highest weight. Similarly, N is the highest-weighted node that comes after G.



Timing analysis

A weight balanced tree gives close to optimal values for the expected length of successful search calculations. From the above example we get

$$\text{ELOSS} = \text{depth}(\text{node A}) * \text{probability}(\text{node A}) + \text{depth}(\text{node C}) * \text{probability}(\text{node C}) + \dots$$

$$\text{ELOSS} = 2 * 0.17 + 5 * 0.03 + 4 * 0.09 + 3 * 0.12 + 1 * 0.20 + 3 * 0.11 + 3 * 0.10 + 2 * 0.18$$

$$\text{ELOSS} = 2.4$$

This is the expected number of nodes that will be examined before finding the desired node.

See also

- Binary tree
- AVL tree
- B-tree
- Binary space partitioning
- Red-black tree
- Treap

References

- Jean-Paul Tremblay and Grant A. Cheston. *Data Structures and Software Development in an object-oriented domain*, Eiffel Edition. Prentice Hall, 2001. ISBN 0-13-787946-6.

Threaded binary tree

A **threaded binary tree** may be defined as follows:

"A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node."

(Van Wyk, Christopher J. Data Structures and C Programs, Addison-Wesley, 1988, p. 175. ISBN 978-0-201-16116-8.)

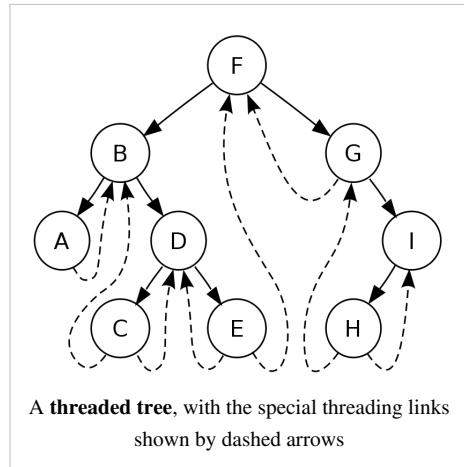
A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal.

It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable (for finding the parent pointer via DFS).

This is possible, because if a node (k) has a right child (m) then m's left pointer must be either a child, or a thread back to k. In the case of a left child, that left child must also have a left child or a thread back to k, and so we can follow m's left children until we find a thread, pointing back to k. The situation is similar for when m is the left child of k

In Python:

```
def parent(node):
    if node is node.tree.root:
        return None
    else:
        x = node
        y = node
        while True:
            if is_thread(y.right):
                p = y.right
                if p is None or p.left is not node:
                    p = x
                    while not is_thread(p.left):
                        p = p.left
                    p = p.left
            return p
        elif is_thread(x.left):
            p = x.left
            if p is None or p.right is not node:
                p = y
```



A **threaded tree**, with the special threading links shown by dashed arrows

```
        while not is_thread(p.right):
            p = p.right
            p = p.right
        return p
    x = x.left
    y = y.right
```

External links

- Tutorial on threaded binary trees ^[1]
- GNU libavl 2.0.2, Section on threaded binary search trees ^[2]

References

- [1] http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_bst1.aspx#thread
- [2] <http://www.stanford.edu/~blp/avl/libavl.html/Threaded-Binary-Search-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 is 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; therefore, it is also said to be height-balanced. 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 inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."^[2]

The **balance factor** of a node is the height of its left subtree minus the height of its right subtree, and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

AVL trees are often compared with red-black trees because they support the same set of operations and because red-black trees also take $O(\log n)$ time for the basic operations. AVL trees perform better than red-black trees for lookup-intensive applications.^[3] The AVL tree balancing algorithm appears in many computer science curricula.

Operations

The basic operations of an AVL tree generally 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.

Insertion

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains -1 , 0 , or 1 then no rotations are necessary. However, if the balance factor becomes 2 or -2 then the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most two tree rotations are needed 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. Let R be the right child of P . Let L be the left child of P .

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. If the balance factor of R is ≤ 0 , a **left rotation** is needed with P as the root. If the balance factor of R is $+1$, a **double left rotation** is needed. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

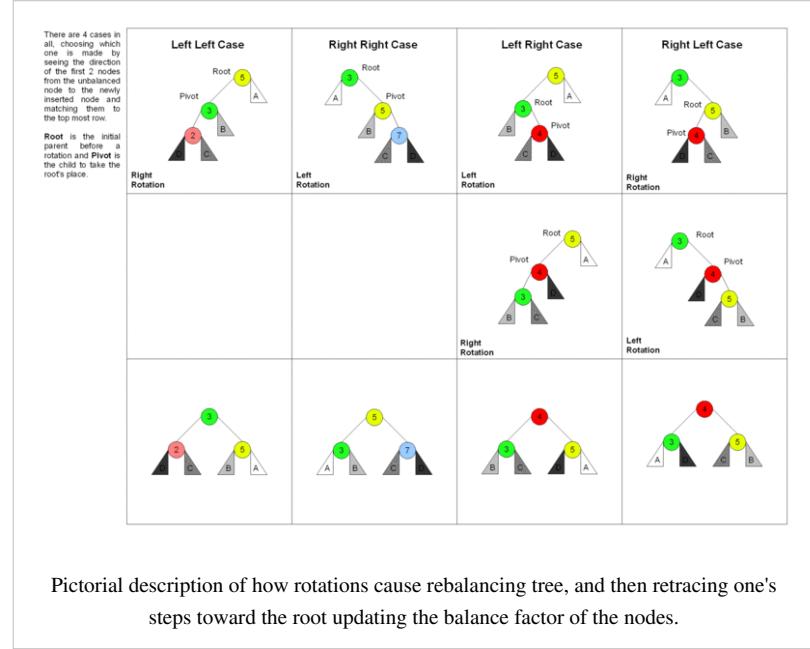
Left-Left case and Left-Right case: If the balance factor of P is $+2$, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must be checked. If the balance factor of L is ≥ 0 , a **right rotation** is needed with P as the root. If the balance factor of L is -1 , a **double right rotation** is needed. The first rotation is a left rotation with L as the root. The second is a right rotation with P as the root.

Algorithms for all the above four case can be found here ^[4].

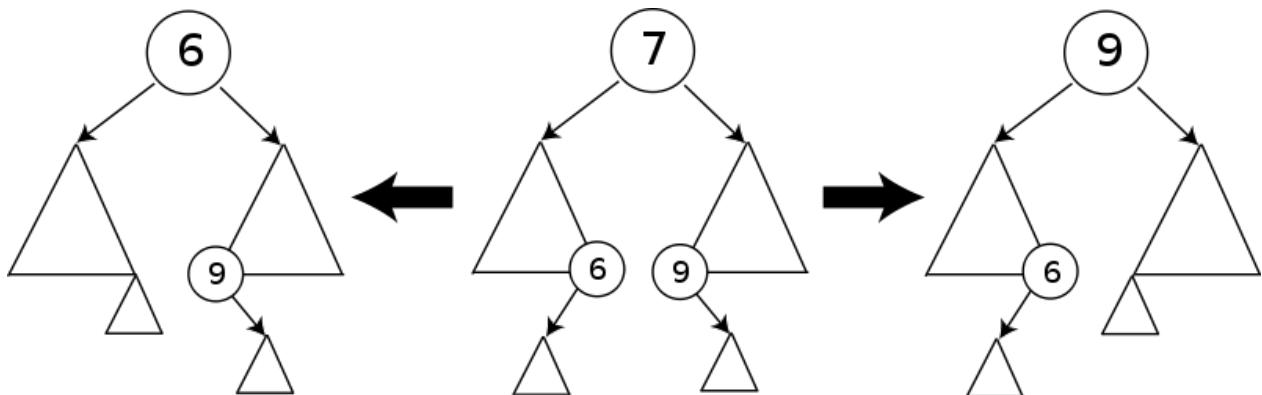
Deletion

If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



Pictorial description of how rotations cause rebalancing tree, and then retracing one's steps toward the root updating the balance factor of the nodes.



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.

Lookup

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. Because of the height-balancing of the tree, a lookup takes $O(\log n)$ time. No special provisions 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. A few cases require traversing up to $2 * \log(n)$ links. However exploring all n nodes in the tree in this manner would use each link exactly twice, and there are $n - 1$ links, so the amortized cost is $2 * (n - 1) / n$, approximately 2.

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 constant time. The real difference between the two is the limiting height. For a tree of size n :

- An AVL tree's height is limited to $1.44 \cdot \log_2(n)$
- A red-black tree's height is limited to $2 \cdot \log_2(n)$

The AVL tree is more rigidly balanced than Red-Black trees, leading to slower insertion and removal but faster retrieval.

See also

- Trees
- Tree rotation
- Splay tree
- Red-black tree
- Matrix tree
- B-tree
- T-tree
- List of data structures

References

- 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.
- [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] <http://www.cse.ohio-state.edu/~sgomori/570/avlrotations.html>

External links

- Description from the Dictionary of Algorithms and Data Structures (<http://www.nist.gov/dads/HTML/avltree.html>)
- C++ Implementation (<https://sourceforge.net/projects/standardavl/>)
- 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/)

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 computing science, typically used to implement associative arrays. The original structure was invented in 1972 by Rudolf Bayer^[1] who called them "symmetric binary B-trees", but acquired its modern name in a paper in 1978 by Leonidas J. Guibas and Robert Sedgewick.^[2] It is complex, but has good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in $O(\log n)$ time, where n is total number of elements in the tree. Put very simply, a red-black tree is a binary search tree which inserts and removes intelligently, to ensure the tree is reasonably balanced.

Terminology

A red-black tree is a special type of binary tree, used in computer science to organize pieces of comparable data, such as text fragments or numbers.

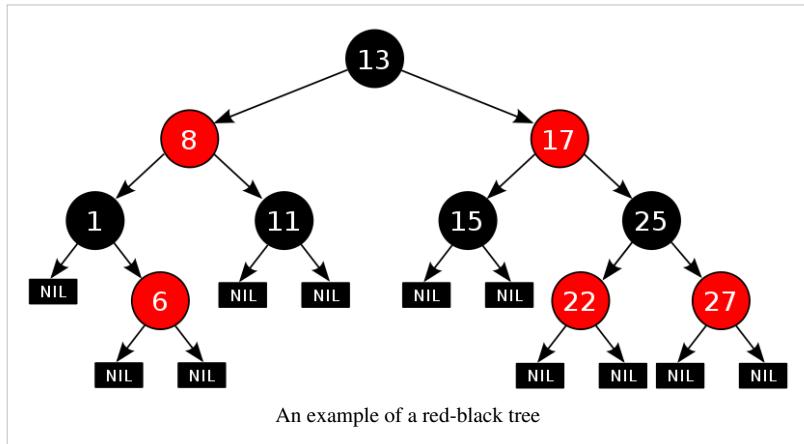
The leaf nodes of red-black trees do not contain data. These leaves need not be explicit in computer memory — a null child pointer can encode the fact that this child is a leaf — but it simplifies some algorithms for operating on red-black trees if the leaves really are explicit nodes. To save memory, sometimes a single sentinel node performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node.

Red-black trees, like all binary search trees, allow efficient in-order traversal in the fashion, Left-Root-Right, of their elements. The search-time results from the traversal from root to leaf, and therefore a balanced tree, having the least possible tree height, results in $O(\log n)$ search time.

Properties

A red-black tree is a binary search tree where each node has a *color* attribute, the value of which is either *red* or *black*. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements apply to red-black trees:

1. A node is either red or black.
2. The root is black. (This rule is used in some definitions and not others. Since the root can always be changed from red to black but not necessarily vice-versa this rule has little effect on analysis.)
3. All leaves are black.
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 longest path from the root to any leaf is no more than twice as long as the shortest path from the root to any other leaf in that tree. The result is that the tree is roughly balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red-black trees to be efficient in the worst-case, unlike ordinary binary search trees.

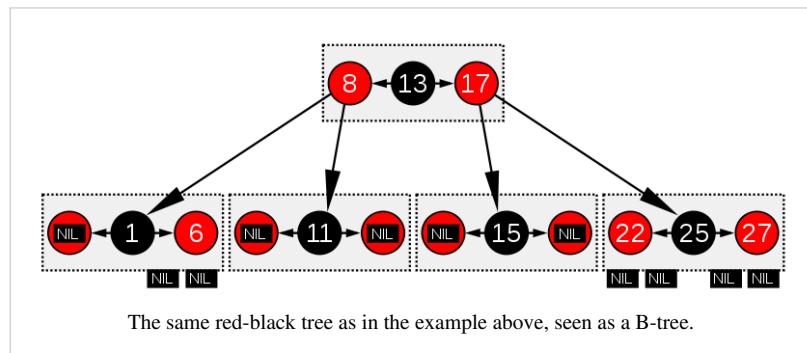
To see why these properties guarantee this, it suffices to note that no path can have two red nodes in a row, due to property 4. The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black 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 which seems to contradict the above principles, but which 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.

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.



The same red-black tree as in the example above, seen as a B-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 order codes of an assembler or interpreter).

Red-black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets which can retain previous versions after mutations. The persistent version of red-black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

For every 2-4 tree, there are corresponding red-black trees with data elements in the same order. The insertion and deletion operations on 2-4 trees are also equivalent to color-flipping and rotations in red-black trees. This makes 2-4 trees an important tool for understanding the logic behind red-black trees, and this is why many introductory algorithm texts introduce 2-4 trees just before red-black trees, even though 2-4 trees are not often used in practice.

In 2008, Sedgewick introduced a simpler version of red-black trees called Left-Leaning Red-Black Trees [3] 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 [4], or 2-4 trees [3], for any sequence of operations. The 2-4 tree isometry was described in 1978 by Sedgewick. With 2-4 trees, the isometry is resolved by a "color flip," corresponding to a split, in which the red color of two children nodes leaves the children and moves to the parent node.

Operations

Read-only operations on a red-black tree require no modification from those used for binary search trees, because every red-black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red-black tree. Restoring the red-black properties requires a small number ($O(\log n)$ or amortized $O(1)$) of color changes (which are very quick in practice) and no more than three tree rotations (two for insertion). Although insert and delete operations are complicated, their times remain $O(\log n)$.

Insertion

Insertion begins by adding the node much as binary search tree insertion does and by coloring it red. Whereas in the binary search tree, we always add a leaf, in the red-black tree leaves contain no information, so instead we add a red interior node, with two black leaves, in place of an existing black leaf.

What happens next depends on the color of other nearby nodes. The term *uncle node* will be used to refer to the sibling of a node's parent, as in human family trees. Note that:

- Property 3 (All leaves are black) always holds.
- Property 4 (Both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.

- Property 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) is threatened only by adding a black node, repainting a red node black, or a rotation.

Note: The label **N** will be used to denote the node being inserted (colored red); **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 these 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 new 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 new 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 new 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.

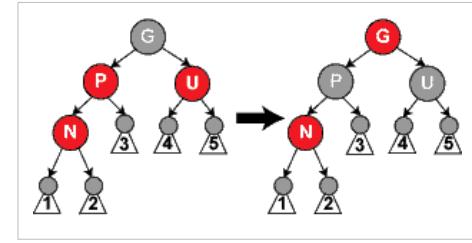
```

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 nodes 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 new 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, this 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.



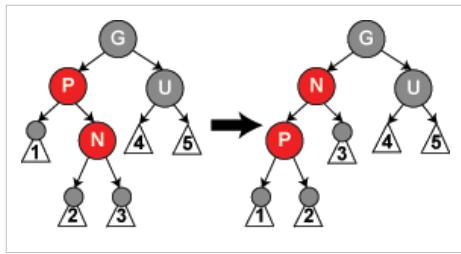
```

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.

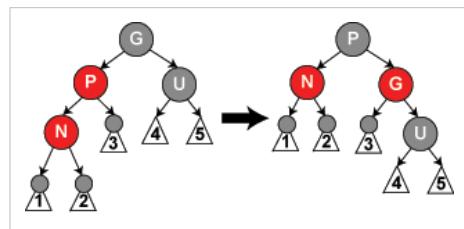


Case 4: The parent **P** is red but the uncle **U** is black; also, the new node **N** is the right child of **P**, and **P** in turn is the left child of its parent **G**. In this case, a left rotation that switches the roles of the new node **N** and its parent **P** can be performed; then, the former 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 new node where they did not before, but both 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.

```
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 new 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 the parent of **P** is performed; the result is a tree where the former parent **P** is now the parent of both the new node **N** and the former grandparent **G**. **G** is known to be black, since its former child **P** could not have been red otherwise. 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) && (n->parent == g->left)) {
        rotate_right(g);
    } else { /* (n == n->parent->right) and (n->parent == g->right) */
        rotate_left(g);
    }
}
```

```
}
```

Note that inserting is actually in-place, since all the calls above use tail recursion.

Removal

In a normal binary search tree, when deleting a node with two non-leaf children, we find either the maximum element in its left subtree or the minimum element in its right subtree, 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 less than two non-leaf children. 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. It does not matter whether this node is the node we originally wanted to delete or the node we copied the value from.

For the remainder of this discussion we can assume we are deleting a node with at most one non-leaf child, which we will call its child (if it has only leaf children, let one of them be its child). If we are deleting a red node, we can simply replace it with its child, which must be black (Actually a red node can have either two non leaf black children or two leaf children which are black per definition, thus in this case the red node is replaced by a leaf because it was required the node to be deleted has at most one non leaf child). 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 properties 3 (All leaves, including *nulls*, are black) and 4 (Both children of every red node are black) still hold.

Another simple case is when the deleted node is black and its child 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 its child black, both of these properties are preserved.

The complex case is when both the node to be deleted and its child are black (Actually we are going to delete a black node which has two leaf children, because if the black node had one non leaf child on the 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 had been an invalid red-black tree). We begin by replacing the node to be deleted with its child. We will call (or *label*) this child (in its new position) **N**, and its sibling (its new parent's other child) **S**. In the diagrams below, we will also use **P** for **N**'s new parent, **S_L** for **S**'s left child, and **S_R** for **S**'s right child (**S** cannot be a leaf because if **N** is black, what we presumed, than **P**'s one subtree which includes **N** counts two black-height and thus **P**'s other subtree which includes **S** must also count two black-height what 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 these 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);
}
```

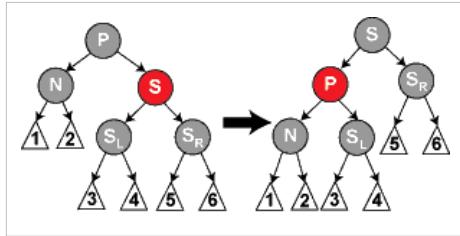
Note: If N is a null leaf and we do not want to represent null leaves as actual node objects, we can modify the algorithm by first calling `delete_case1()` on its parent (the node that we delete, n in the code above) and deleting it afterwards. We can do this because the parent is black, so it behaves 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);
}
```

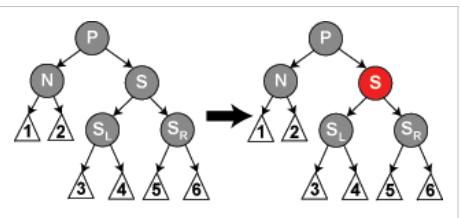
Note: In cases 2, 5, and 6, we assume N is the left child of its parent P. If it is the right child, *left* and *right* should be reversed throughout these three cases. Again, the code examples take both cases into account.



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);
}
```



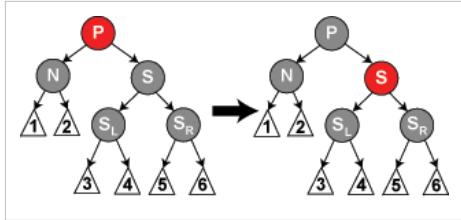
Case 3: **P**, **S**, and **S**'s children are black. In this case, we simply repaint **S** red. The result is that all paths passing through **S**, which are precisely those paths *not* passing through **N**, have one less black node. Because deleting **N**'s original parent made all paths passing through **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
```

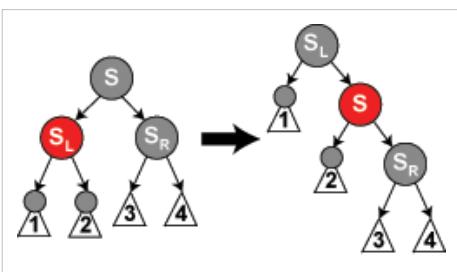
```
        delete_case4(n);
}
```



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 two changed the sibling to a sibling's
child,
the sibling's child can't be red, since no red parent can have a red
```

```

child). */

// the following statements just force the red to be on the left of the
left of the parent,
// or right of the right, so case six will rotate correctly.

if ((n == n->parent->left) &&
      (s->right->color == BLACK) &&
      (s->left->color == RED)) { // this last test is trivial
too due to cases 2-4.

      s->color = RED;
      s->left->color = BLACK;
      rotate_right(s);

} else if ((n == n->parent->right) &&
            (s->left->color == BLACK) &&
            (s->right->color == RED)) { // this last test is
trivial too due to cases 2-4.

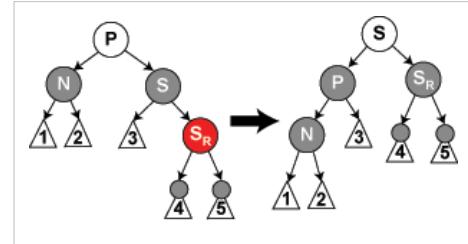
      s->color = RED;
      s->right->color = BLACK;
      rotate_left(s);
}

}

delete_case6(n);
}

```

Case 6: S is black, S's right child is red, and N is the left child of its parent P. In this case we rotate left at P, so that S becomes the parent of P and S's right child. We then exchange the colors of P and S, and make S's right child black. The subtree still has the same color at its root, so Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) are not violated. However, N now has one additional black ancestor: either P has become black, or it was black and S was added as a black grandparent. Thus, the paths passing through N pass through one additional black node.



Meanwhile, if a path does not go through N, then there are two possibilities:

- It goes through N's new sibling. Then, it must go through S and P, both formerly and currently, as they have only exchanged colors and places. Thus the path contains the same number of black nodes.
- It goes through N's new uncle, S's right child. Then, it formerly went through S, S's parent, and S's right child (which was red), but now only goes through S, which has assumed the color of its former parent, and S's right child, which has changed from red to black (assuming S's color: black). The net effect is that this path goes through the same number of black nodes.

Either way, the number of black nodes on these paths does not change. Thus, we have restored Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes). The white node in the diagram can be either red or black, but must refer to the same color both before and after the transformation.

```

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 $\text{bh}(v) = 0$. 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 has a black-height of either $bh(v')$ or $bh(v')-1$ (depending on whether the child is red or black). 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 $O(\log(n))$.

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.

See also

- Trees
- Tree rotation
- Splay tree
- AVL tree
- Matrix tree
- B-tree
- T-tree
- List of data structures

References

- Mathworld: Red-Black Tree ^[5]
- San Diego State University: CS 660: Red-Black tree notes ^[6], 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" ^[7] (PDF). Stanford University.
- Okasaki, Chris. "Red-Black Trees in a Functional Setting" ^[8] (PS).

External links

- In the C++ Standard Template Library, the containers `std::set<Value>` and `std::map<Key,Value>` are often based on red-black trees
- Efficient implementation of Red-Black Trees ^[9]
- Red-Black Tree in GNU libavl C library by Ben Pfaff ^[10]
- Red-Black Tree C Code ^[11]
- Red-Black Tree PHP5 Code ^[12]
- In Java a freely available red black tree implementation is that of apache commons ^[13]
- Java's TreeSet class internally stores its elements in a red black tree [14]

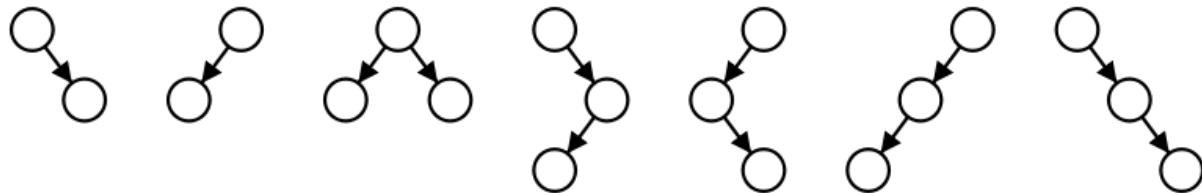
References

- [1] Rudolf Bayer (1972). "Symmetric binary B-Trees: Data structure and maintenance algorithms" (<http://www.springerlink.com/content/qh51m2014673513j/>). *Acta Informatica* 1 (4): 290–306. doi:10.1007/BF00289509. .
- [2] Leonidas J. Guibas and Robert Sedgewick (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.
- [3] <http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>
- [4] <http://www.cs.princeton.edu/courses/archive/fall08/cos226/lectures/10BalancedTrees-2x2.pdf>
- [5] <http://mathworld.wolfram.com/Red-BlackTree.html>
- [6] <http://www.elis.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html#RTFToC2>
- [7] <http://www.stanford.edu/~blp/papers/libavl.pdf>
- [8] <http://www.eecs.usma.edu/webs/people/okasaki/jfp99.ps>
- [9] http://eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx
- [10] http://www.stanford.edu/~blp/avl/libavl.html/Red_002dBlack-Trees.html
- [11] http://web.mit.edu/~emin/www/source_code/red_black_tree/index.html
- [12] <http://code.google.com/p/redblacktreeph/source/browse/#svn/trunk>
- [13] <http://commons.apache.org/collections/api-release/org/apache/commons/collections/bidimap/TreeBidiMap.html>
- [14] <http://java.sun.com/docs/books/tutorial/collections/interfaces/set.html>

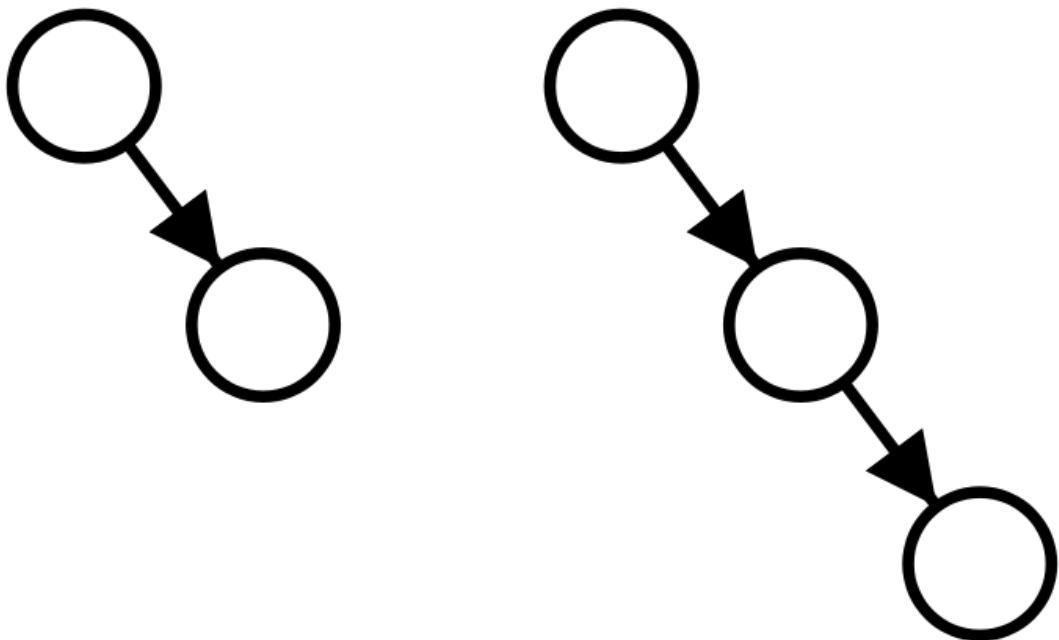
AA tree

An **AA tree** in computer science is a form of balanced tree used for storing and retrieving ordered data efficiently. AA trees are named for Arne Andersson, their inventor.

AA trees are a variation of the red-black tree, which in turn is an enhancement to the binary search tree. Unlike red-black trees, red nodes on an AA tree can only be added as a right subchild. In other words, no red node can be a left sub-child. This results in the simulation of a 2-3 tree instead of a 2-3-4 tree, which greatly simplifies the maintenance operations. The maintenance algorithms for a red-black tree need to consider seven different shapes to properly balance the tree:



An AA tree on the other hand only needs to consider two shapes due to the strict requirement that only right links can be red:



Balancing Rotations

Typically, AA trees are implemented with levels instead of colors, unlike red-black trees. Each node has a level field, and the following invariants must remain true for the tree to be valid:

1. The level of a leaf node is one.
2. The level of a left child is strictly less than that of its parent.
3. The level of a right child is less than or equal to that of its parent.
4. The level of a right grandchild is strictly less than that of its grandparent.
5. Every node of level greater than one must have two children.

Only two operations are needed for maintaining balance in an AA tree. These operations are called skew and split. Skew is a right rotation when an insertion or deletion creates a left horizontal link, which can be thought of as a left red link in the red-black tree context. Split is a conditional left rotation when an insertion or deletion creates two horizontal right links, which once again corresponds to two consecutive red links in red-black trees.

```
function skew is
    input: T, a node representing an AA tree that needs to be
    rebalanced.
    output: Another node representing the rebalanced AA tree.

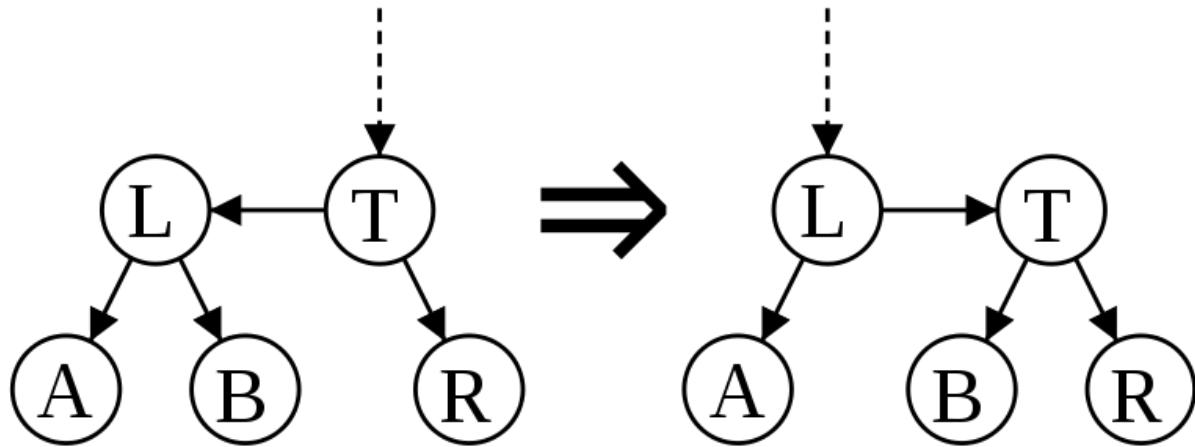
    if nil(T) then
        return Nil
    else if level(left(T)) == level(T) then
        Swap the pointers of horizontal left links.
        L = left(T)
        left(T) := right(L)
        right(L) := T
```

```

    return L
else
    return T
end if
end function

```

Skew:



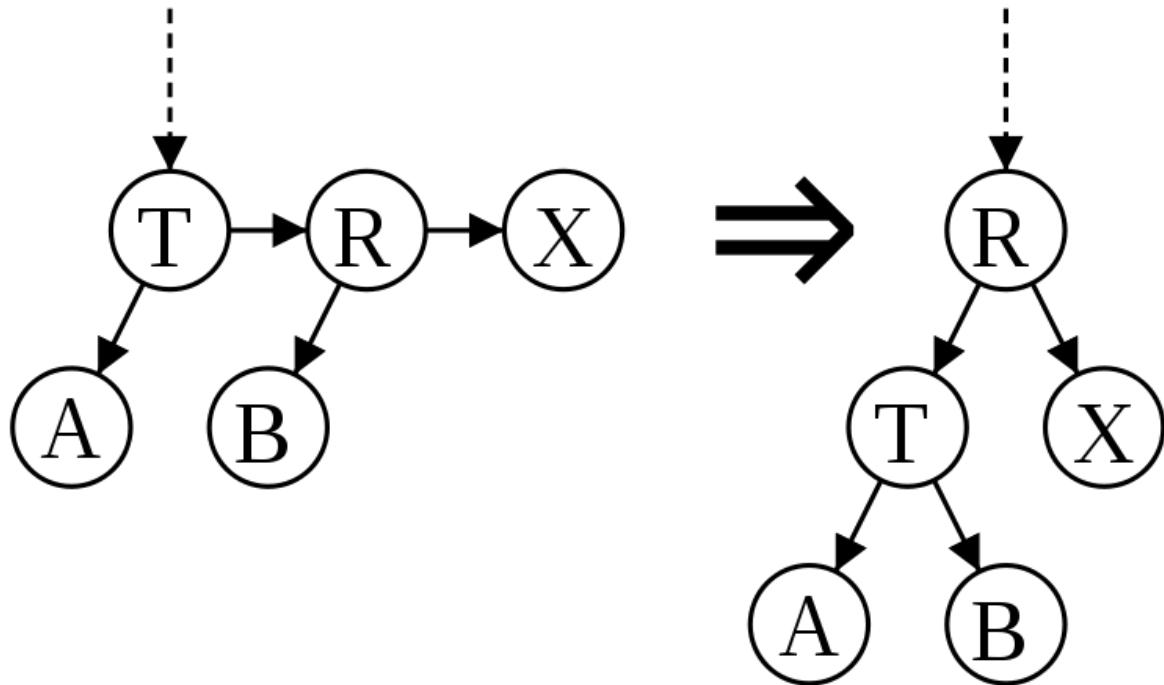
```

function split is
    input: T, a node representing an AA tree that needs to be
    rebalanced.
    output: Another node representing the rebalanced AA tree.

    if nil(T) then
        return Nil
    else if level(T) == level(right(right(T))) then
        We have two horizontal right links. Take the middle node,
        elevate it, and return it.
        R = right(T)
        right(T) := left(R)
        left(R) := T
        level(R) := level(R) + 1
        return R
    else
        return T
    end if
end function

```

Split:



Insertion

Insertion begins with the normal binary tree search and insertion procedure. Then, as the call stack unwinds, it's easy to check the validity of the tree and perform any rotations as necessary. If a horizontal left link arises, a skew will be performed, and if two horizontal right links arise, a split will be performed, possibly incrementing the level of the new root node of the current subtree. Note in the code as given above the increment of level(T). This makes it necessary to continue checking the validity of the tree as the modifications bubble up from the leaves.

```

function insert is
  input: X, the value to be inserted, and T, the root of the tree to
  insert it into.
  output: A balanced version T including X.
  
```

Do the normal binary tree insertion procedure. Set the result of the

recursive call to the correct child in case a new node was created or the root of the subtree changes.

```

if nil(T) then
  Create a new leaf node with X.
  return node(X, 1, Nil, Nil)
else if X < value(T) then
  left(T) := insert(X, left(T))
else if X > value(T) then
  right(T) := insert(X, right(T))
end if
  
```

Note that the case of X == value(T) is unspecified. As given, an

```

insert
    will have no effect. The implementor may desire different
behavior.

    Perform skew and then split. The conditionals that determine
whether or

    not a rotation will occur or not are inside of the procedures, as
given
    above.

    T := skew(T)
    T := split(T)

    return T
end function

```

Deletion

As in most balanced binary trees, the deletion of an internal node can be turned into the deletion of a leaf node by swapping the internal node with either its closest predecessor or successor, depending on which are in the tree or the implementor's whims. Retrieving a predecessor is simply a matter of following one left link and then all of the remaining right links. Similarly, the successor can be found by going right once and left until a null pointer is found. Because of the AA property of all nodes of level greater than one having two children, the successor or predecessor node will be in level 1, making their removal trivial.

To re-balance a tree, there are a few approaches. The one described by Andersson in his original paper ^[1] is the simplest, and it is described here, although actual implementations may opt for a more optimized approach. After a removal, the first step to maintaining tree validity is to lower the level of any nodes whose children are two levels below them, or who are missing children. Then, the entire level must be skewed and split. This approach was favored, because when laid down conceptually, it has three easily understood separate steps:

1. Decrease the level, if appropriate.
2. Skew the level.
3. Split the level.

However, we have to skew and split the entire level this time instead of just a node, complicating our code.

```

function delete is
    input: X, the value to delete, and T, the root of the tree from
which it should be deleted.
    output: T, balanced, without the value X.

    if X > value(T) then
        right(T) := delete(X, right(T))
    else if X < value(T) then
        left(T) := delete(X, left(T))
    else
        If we're a leaf, easy, otherwise reduce to leaf case.
        if leaf(T) then
            return Nil
        else if nil(left(T)) then
            L := successor(T)

```

```

        right(T) := delete(L, right(T))
        value(T) := L
else
    L := predecessor(T)
    left(T) := delete(L, left(T))
    value(T) := L
end if
end if

Rebalance the tree. Decrease the level of all nodes in this level

if
    necessary, and then skew and split all nodes in the new level.
    T := decrease_level(T)
    T := skew(T)
    right(T) := skew(right(T))
    right(right(T)) := skew(right(right(T)))
    T := split(T)
    right(T) := split(right(T))
    return T
end function

```

```

function decrease_level is
    input: T, a tree for which we want to remove links that skip
levels.
    output: T with its level decreased.

    should_be = min(level(left(T)), level(right(T))) + 1
    if should_be < level(T) then
        level(T) := should_be
        if should_be < level(right(T)) then
            level(right(T)) := should_be
        end if
    end if
    return T
end function

```

A good example of deletion by this algorithm is present in the Andersson paper^[1].

Performance

The performance of an AA tree is equivalent to the performance of a red-black tree. While an AA tree makes more rotations than a red-black tree, the simpler algorithms tend to be faster, and all of this balances out to result in similar performance. A red-black tree is more consistent in its performance than an AA tree, but an AA tree tends to be flatter, which results in slightly faster search times.

See also

- Red-black tree
- B-tree
- AVL tree
- Matrix tree

References

- A. Andersson. Balanced search trees made simple ^[1]
- A. Andersson. A note on searching in a binary search tree ^[2]

External links

- AA-Tree Applet ^[16] by Kubo Kovac
- AAT ^[3] - Open source AA tree library for C
- AA Visual 2007 1.5 - OpenSource Delphi program for educating AA tree structures ^[4]
- Thorough tutorial with lots of code ^[5]
- Practical implementation ^[6]
- Object Oriented implementation with tests ^[7]
- Comparison of AA trees, red-black trees, treaps, skip lists, and radix trees ^[8]
- An example C implementation ^[9]

References

- [1] <http://user.it.uu.se/~arnea/abs/simp.html>
- [2] <http://user.it.uu.se/~arnea/abs/searchproc.html>
- [3] <http://bitbucket.org/trijezdci/bstlib/src/>
- [4] <http://www.softpedia.com/get/Others/Home-Education/AA-Visual-2007.shtml>
- [5] http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_andersson.aspx
- [6] http://www.eternallyconfuzzled.com/libs/jsw_atree.zip
- [7] http://www.cs.fiu.edu/~weiss/dsaa_c++3/code/
- [8] <http://www.upgrade-cepis.org/issues/2004/5/up5-5Mosaic.pdf>
- [9] <http://www.rational.co.za/aatree.c>

Scapegoat tree

In computer science, a **scapegoat tree** is a self-balancing binary search tree, invented by Igal Galperin, Jakob Tepč and Ronald L. Rivest. It provides worst-case $O(\log n)$ lookup time, and $O(\log n)$ amortized insertion and deletion time.

Unlike other self-balancing binary search trees that provide worst case $O(\log n)$ lookup time, scapegoat trees have no additional per-node overhead compared to a regular binary search tree.^[1] This makes scapegoat trees easier to implement and, due to data structure alignment, can reduce node overhead by up to one-third.

Theory

A binary search tree is said to be weight balanced if half the nodes are on the left of the root, and half on the right. An α -weight-balanced is therefore defined as meeting the following conditions:

```
size(left) <= α * size(node)
size(right) <= α * size(node)
```

Where size can be defined recursively as:

```
function size(node)
  if node = nil
    return 0
  else
    return size(node->left) + size(node->right) + 1
end
```

An α of 1 therefore would describe a linked list as balanced, whereas an α of 0.5 would only match almost complete binary trees.

A binary search tree that is α -weight-balanced must also be **α -height-balanced**, that is

```
height(tree) <= log1/α(NodeCount)
```

Scapegoat trees are not guaranteed to keep α -weight-balance at all times, but are always loosely α -height-balance in that

```
height(scapegoat tree) <= log1/α(NodeCount) + 1
```

This makes scapegoat trees similar to red-black trees in that they both have restrictions on their height. They differ greatly though in their implementations of determining where the rotations (or in the case of scapegoat trees, rebalances) take place. Whereas red-black trees store additional 'color' information in each node to determine the location, scapegoat trees find a **scapegoat** which isn't α -weight-balanced to perform the rebalance operation on. This is loosely similar to AVL trees, in that the actual rotations depend on 'balances' of nodes, but the means of determining the balance differs greatly. Since AVL trees check the balance value on every insertion/deletion, it is typically stored in each node; scapegoat trees are able to calculate it only as needed, which is only when a scapegoat needs to be found.

Unlike most other self-balancing search trees, scapegoat trees are entirely flexible as to their balancing. They support any α such that $0.5 \leq \alpha < 1$. A high α value results in fewer balances, making insertion quicker but lookups and deletions slower, and vice versa for a low α . Therefore in practical applications, an α can be chosen depending on how frequently these actions should be performed.

Operations

Insertion

Insertion is implemented very similarly to an unbalanced binary search tree, however with a few significant changes.

When finding the insertion point, the depth of the new node must also be recorded. This is implemented via a simple counter that gets incremented during each iteration of the lookup, effectively counting the number of edges between the root and the inserted node. If this node violates the α -height-balance property (defined above), a rebalance is required.

To rebalance, an entire subtree rooted at a **scapegoat** undergoes a balancing operation. The scapegoat is defined as being an ancestor of the inserted node which isn't α -weight-balanced. There will always be at least one such ancestor. Rebalancing any of them will restore the α -height-balanced property.

One way of finding a scapegoat, is to climb from the new node back up to the root and select the first node that isn't α -weight-balanced.

Climbing back up to the root requires $O(\log n)$ storage space, usually allocated on the stack, or parent pointers. This can actually be avoided by pointing each child at its parent as you go down, and repairing on the walk back up.

To determine whether a potential node is a viable scapegoat, we need to check its α -weight-balanced property. To do this we can go back to the definition:

```
size(left) <= α * size(node)  
size(right) <= α * size(node)
```

However a large optimisation can be made by realising that we already know two of the three sizes, leaving only the third having to be calculated.

Consider the following example to demonstrate this. Assuming that we're climbing back up to the root:

```
size(parent) = size(node) + size(brother) + 1
```

But as:

```
size(inserted node) = 1.
```

The case is trivialized down to:

```
size[x+1] = size[x] + size(brother) + 1
```

Where x = this node, $x + 1$ = parent and $\text{size}(\text{brother})$ is the only function call actually required.

Once the scapegoat is found, a standard binary search tree rebalance operation is performed.

As rebalance operations take $O(n)$ time dependent on the number of nodes of the subtree, insertion has a worst case performance of $O(n)$ time, however amortized has $O(\log n)$ average time.

Sketch of proof for cost of insertion

Define the Imbalance of a node v to be the absolute value of the difference in size between its left node and right node minus 1, or 0, whichever is greater. In other words:

$$I(v) = \max(|\text{left}(v)| - |\text{right}(v)| - 1, 0)$$

Immediately after rebuilding a subtree rooted at v , $I(v) = 0$.

Lemma: Immediately before rebuilding the subtree rooted at v , $I(v) = \Omega(|v|)$

Proof of lemma:

Let v_0 be the root of a subtree immediately after rebuilding. $h(v_0) = \log(|v_0| + 1)$. If there are $\Omega(|v_0|)$ degenerate insertions (that is, where each inserted node increases the height by 1), then $I(v) = \Omega(|v_0|)$, $h(v) = h(v_0) + \Omega(|v_0|)$ and $\log(|v|) \leq \log(|v_0| + 1) + 1$.

Since $I(v) = \Omega(|v|)$ before rebuilding, there were $\Omega(|v|)$ insertions into the subtree rooted at v that did not result in rebuilding. Each of these insertions can be performed in $O(\log n)$ time. The final insertion that causes rebuilding costs $O(|v|)$. Using aggregate analysis it becomes clear that the amortized cost of an insertion is $O(\log n)$:

$$\frac{\Omega(|v|)O(\log n) + O(|v|)}{\Omega(|v|)} = O(\log n)$$

The deletion operation

Scapegoat trees are unusual in that deletion is easier than insertion. To enable deletion, scapegoat trees need to store an additional value with the tree data structure. This property, which we will call MaxNodeCount simply represents the highest achieved NodeCount. It is set to NodeCount whenever the entire tree is rebalanced, and after insertion is set to $\max(\text{MaxNodeCount}, \text{NodeCount})$.

To perform a deletion, we simply remove the node as you would in a simple binary search tree, but if

```
NodeCount <= MaxNodeCount / 2
```

then we rebalance the entire tree about the root, remembering to set MaxNodeCount to NodeCount.

This gives deletion its worst case performance of $O(n)$ time, however it is amortized to $O(\log n)$ average time.

Sketch of proof for cost of deletion

Suppose the scapegoat tree has n elements and has just been rebuilt (in other words, it is a complete binary tree). At most $n/2 - 1$ deletions can be performed before the tree must be rebuilt. Each of these deletions take $O(\log n)$ time (the amount of time to search for the element and flag it as deleted). The $n/2$ deletion causes the tree to be rebuilt and takes $O(\log n) + O(n)$ (or just $O(n)$) time. Using aggregate analysis it becomes clear that the amortized cost of a deletion is $O(\log n)$:

$$\frac{\sum_{1}^{\frac{n}{2}} O(\log n) + O(n)}{\frac{n}{2}} = \frac{\frac{n}{2}O(\log n) + O(n)}{\frac{n}{2}} = O(\log n)$$

Lookup

Lookup is not modified from a standard binary search tree, and has a worst-case time of $O(\log n)$. This is in contrast to splay trees which have a worst-case time of $O(n)$. The reduced node overhead compared to other self-balancing binary search trees can further improve locality of reference and caching.

Further reading

- Andersson, Arne (1999), "General balanced trees", *Journal of Algorithms* **30**: pp. 1–28,
doi:10.1006/jagm.1998.0967

See also

- Splay tree
- Matrix tree

External links

- Scapegoat Tree Applet ^[16] by Kubo Kovac
- Scapegoat Trees: the original publication describing scapegoat trees ^[2]
- On Consulting a Set of Experts and Searching (full version paper) ^[3]

References

- [1] Galperin, Igal; Rivest, Ronald L. (1993), "Scapegoat trees" (<http://portal.acm.org/citation.cfm?id=313676>), *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*: pp. 165–174,
- [2] <http://cg.scs.carleton.ca/~morin/teaching/5408/refs/gr93.pdf>
- [3] <http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-700.pdf>

Splay tree

A **splay tree** is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in $O(\log(n))$ amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985.^[1]

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

Advantages

Good performance for a splay tree depends on the fact that it is self-balancing, and indeed self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. This is an advantage for nearly all practical applications, and is particularly useful for implementing caches and garbage collection algorithms.

Advantages include:

- Simple implementation—simpler than other self-balancing binary search trees, such as red-black trees or AVL trees.
- Comparable performance -- average-case performance is as efficient as other trees.
- Small memory footprint—splay trees do not need to store any bookkeeping data.
- Possibility of creating a persistent data structure version of splay trees—which allows access to both the previous and new versions after an update. This can be useful in functional programming, and requires amortized $O(\log n)$ space per update.
- Working well with nodes containing identical keys—contrary to other types of self-balancing trees. Even with identical keys, performance remains amortized $O(\log n)$. All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the left most or right most node of a given key.

Disadvantages

Disadvantages:

- There could exist trees which perform "slightly" faster ($a \log(\log(N))$) factor for a given distribution of input queries.
- Poor performance on uniform access (with workaround) -- a splay tree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree for uniform access.

One worst-case issue with the basic splay tree algorithm is that of sequentially accessing all the elements of the tree in the sorted order. This leaves the tree completely unbalanced (this takes n accesses, each an $O(\log n)$ operation). Reaccessing the first item triggers an operation that takes $O(n)$ operations to rebalance the tree before returning the first item. This is a significant delay for that final operation, although the amortized performance over the entire sequence is actually $O(\log n)$. However, recent research shows that randomly rebalancing the tree can avoid this unbalancing effect and give similar performance to the other self-balancing algorithms.

Operations

Splaying

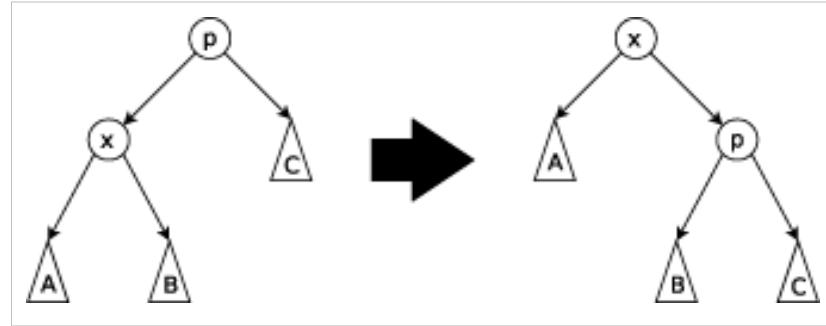
When a node x is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step depends on three factors:

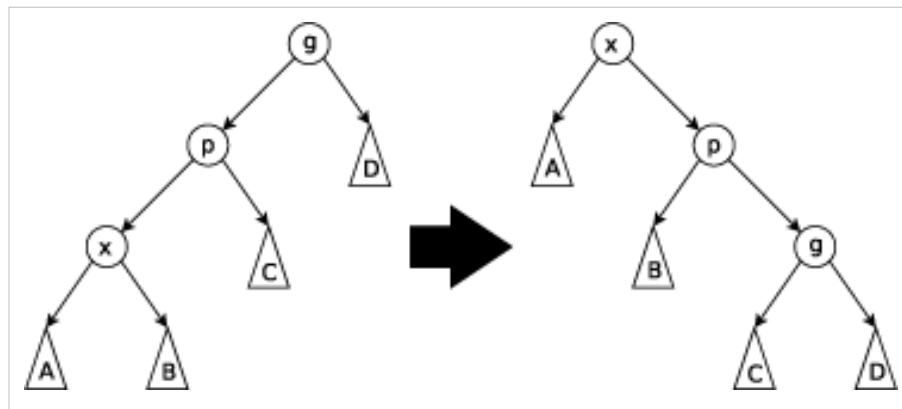
- Whether x is the left or right child of its parent node, p ,
- whether p is the root or not, and if not
- whether p is the left or right child of its parent, g (the *grandparent* of x).

The three types of splay steps are:

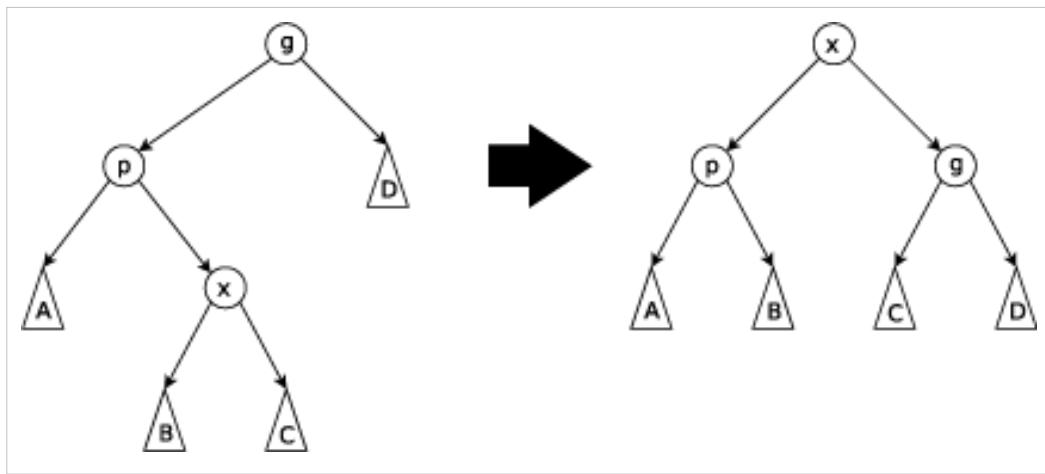
Zig Step: This step is done when p is the root. The tree is rotated on the edge between x and p . Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when x has odd depth at the beginning of the operation.



Zig-zig Step: This step is done when p is not the root and x and p are either both right children or are both left children. The picture below shows the case where x and p are both left children. The tree is rotated on the edge joining p with its parent g , then rotated on the edge joining x with p . Note that zig-zig steps are the only thing that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro^[2] prior to the introduction of splay trees.



Zig-zag Step: This step is done when p is not the root and x is a right child and p is a left child or vice versa. The tree is rotated on the edge between x and p , then rotated on the edge between x and its new parent g .



Insertion

To insert a node x into a splay tree, we first insert it as with a normal binary search tree. Then we splay the new node x to the top of the tree.

Deletion

To delete a node x , we use the same method as with a binary search tree. If x has two children, we replace its value with either the rightmost node of its left subtree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor). Then we remove that node instead. In this way, deletion is reduced to the problem of removing a node with 0 or 1 children. After deletion, we splay the parent of the removed node to the top of the tree.

Code in "C" language

A complete program of splay tree written in C is uploaded at this location ^[3]

Splay operation in BST

Here x is the node on which the splay operation is performed and root is the root node of the tree.

```
void splay (struct node *x, struct node *root)
{
    node *p,*g;
    /*check if node x is the root node*/
    if(x==root);
    /*Performs Zig step*/
    else if(x->parent==root)
    {
        if(x==(x->parent)->left)
            rightrotation(root);
        else
            leftrotation(root);
    }
    else
    {
        p=x->parent; /*now points to parent of x*/
        g=p->parent; /*now points to parent of x's parent*/
        if(g==root)
            if(x==(g->left))
                if(x==(p->left))
                    rightrightrotation(root);
                else
                    leftrightrotation(root);
            else
                rightleftrightrotation(root);
        else
            if(x==(g->left))
                if(x==(p->left))
                    rightleftrotation(root);
                else
                    leftrightleftrotation(root);
            else
                rightleftrightleftrotation(root);
    }
}
```

```

/*Performs the Zig-zig step when x is left and x's parent is left*/
if(x==p->left&&p==g->left)
{
    rightrotation(g);
    rightrotation(p);
}
/*Performs the Zig-zig step when x is right and x's parent is right*/
else if(x==p->right&&p==g->right)
{
    leftrotation(g);
    leftrotation(p);
}
/*Performs the Zig-zag step when x's is right and x's parent is left*/
else if(x==p->right&&p==g->left)
{
    leftrotation(p);
    rightrotation(g);
}
/*Performs the Zig-zag step when x's is left and x's parent is right*/
else if(x==p->left&&p==g->right)
{
    rightrotation(p);
    leftrotation(g);
}
splay(x, root);
}
}

```

Analysis

A simple amortized analysis of static splay trees can be carried out using the potential method. Suppose that $\text{size}(r)$ is the number of nodes in the subtree rooted at r (including r) and $\text{rank}(r) = \log_2(\text{size}(r))$. Then the potential function $P(t)$ for a splay tree t is the sum of the ranks of all the nodes in the tree. This will tend to be high for poorly-balanced trees, and low for well-balanced trees. We can bound the amortized cost of any zig-zig or zig-zag operation by:

$$\text{amortized cost} = \text{cost} + P(t_f) - P(t_i) \leq 3(\text{rank}_f(x) - \text{rank}_i(x)),$$

where x is the node being moved towards the root, and the subscripts "f" and "i" indicate after and before the operation, respectively. When summed over the entire splay operation, this telescopes to $3(\text{rank}(\text{root}))$ which is $O(\log n)$. Since there's at most one zig operation, this only adds a constant.

Performance theorems

There are several theorems and conjectures regarding the worst-case runtime for performing a sequence S of m accesses in a splay tree containing n elements.

Balance Theorem^[1]

The cost of performing the sequence S is $O(m(1 + \log n) + n \log n)$. In other words, splay trees perform as well as static balanced binary search trees on sequences of at least n accesses.

Static Optimality Theorem^[1]

Let q_i be the number of times element i is accessed in S . The cost of performing S is $O\left(m + \sum_{i=1}^n q_i \log \frac{m}{q_i}\right)$. In other words, splay trees perform as well as optimum static binary search trees on sequences of at least n accesses.

Static Finger Theorem^[1]

Let i_j be the element accessed in the j^{th} access of S and let f be any fixed element (the finger). The cost of performing S is $O\left(m + n \log n + \sum_{j=1}^m \log(|i_j - f| + 1)\right)$.

Working Set Theorem^[1]

Let $t(j)$ be the number of distinct elements accessed between access j and the previous time element i_j was accessed. The cost of performing S is $O\left(m + n \log n + \sum_{j=1}^m \log(t(j) + 1)\right)$.

Dynamic Finger Theorem^{[4] [5]}

The cost of performing S is $O\left(m + n + \sum_{j=1}^m \log(|i_{j+1} - i_j| + 1)\right)$.

Scanning Theorem^[6]

Also known as the **Sequential Access Theorem**. Accessing the n elements of a splay tree in symmetric order takes $O(n)$ time, regardless of the initial structure of the splay tree. The tightest upper bound proven so far is $4.5n$.^[7]

Dynamic optimality conjecture

In addition to the proven performance guarantees for splay trees there is an unproven conjecture of great interest from the original Sleator and Tarjan paper. This conjecture is known as the *dynamic optimality conjecture* and it basically claims that splay trees perform as well as any other binary search tree algorithm up to a constant factor.

Dynamic Optimality Conjecture:^[1] Let A be any binary search tree algorithm that accesses an element x by traversing the path from the root to x at a cost of $d(x) + 1$, and that between accesses can make any rotations in the tree at a cost of 1 per rotation. Let $A(S)$ be the cost for A to perform the sequence S of accesses. Then the cost for a splay tree to perform the same accesses is $O(n + A(S))$.

There are several corollaries of the dynamic optimality conjecture that remain unproven:

Traversal Conjecture:^[1] Let T_1 and T_2 be two splay trees containing the same elements. Let S be the sequence obtained by visiting the elements in T_2 in preorder (*i.e.* depth first search order). The total cost of performing the sequence S of accesses on T_1 is $O(n)$.

Deque Conjecture:^{[8] [6] [9]} Let S be a sequence of m double-ended queue operations (push, pop, inject, eject). Then the cost of performing S on a splay tree is $O(m + n)$.

Split Conjecture:^[10] Let S be any permutation of the elements of the splay tree. Then the cost of deleting the elements in the order S is $O(n)$.

See also

- Knuth, Donald. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Page 478 of section 6.2.3.
- finger tree
- Link/cut tree
- Scapegoat tree
- Zipper (data structure)
- Matrix tree

External links

Algorithm

- Sleator and Tarjan, *Self-adjusting Binary Search Trees*^[11] (the original publication)
- NIST's *Dictionary of Algorithms and Data Structures: Splay Tree*^[12]

Implementations

- Implementations in C and Java by Sleator (one of the original inventors)^[13]
- FreeBSD's single header file implementation^[14]
- Splay^[3] - Open source Splay tree library for C
- Splay porting in C++^[15] by Antonio Gulli

Visualizations

- New York University: Dept of Computer Science: Algorithm Visualization: Splay Trees^[16]
- Pointers to splay tree visualizations^[17]
- Splay Tree Applet^[18]
- AVL, Splay and Red/Black Applet^[11]
- Splay Tree pictures^[19]

References

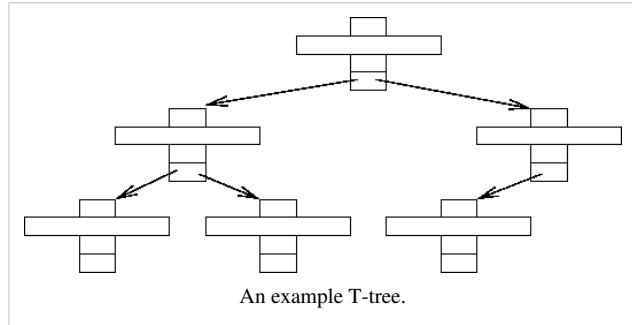
- [1] Sleator, Daniel D.; Tarjan, Robert E. (1985), "Self-Adjusting Binary Search Trees" (<http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf>), *Journal of the ACM (Association for Computing Machinery)* **32** (3): 652–686, doi:10.1145/3828.3835,
- [2] Allen, Brian; and Munro, Ian (1978), "Self-organizing search trees", *Journal of the ACM* **25** (4): 526–535
- [3] <http://sites.google.com/site/sahawamybirthplace/splay.cpp?attredirects=0>
- [4] Cole, Richard; Mishra, Bud; Schmidt, Jeanette; and Siegel, Alan (2000), "On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences", *SIAM (Society for Industrial and Applied Mathematics) Journal on Computing* **30**: 1–43
- [5] Cole, Richard (2000), "On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof", *SIAM Journal on Computing* **30**: 44–85
- [6] Tarjan, Robert E. (1985), "Sequential access in splay trees takes linear time", *Combinatorica* **5**: 367–378
- [7] Elmasry, Amr (2004), "On the sequential access theorem and Deque conjecture for splay trees", *Theoretical Computer Science* **314** (3): 459–466
- [8] Pettie, Seth (2008), "Splay Trees, Davenport-Schinzel Sequences, and the Deque Conjecture", *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*: 1115–1124
- [9] Sundar, Rajamani (1992), "On the Deque conjecture for the splay algorithm", *Combinatorica* **12** (1): 95–124
- [10] Lucas, Joan M. (1991), "On the Competitiveness of Splay Trees: Relations to the Union-Find Problem", *Online Algorithms, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) Series in Discrete Mathematics and Theoretical Computer Science Vol. 7*: 95–124
- [11] <http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf>

- [12] <http://www.nist.gov/dads/HTML/splaytree.html>
- [13] <http://www.link.cs.cmu.edu/link/ftp-site/splaying/>
- [14] <http://fxr.watson.org/fxr/source//sys/tree.h>
- [15] http://codingplayground.blogspot.com/2009_04_01_archive.html
- [16] <http://www.cs.nyu.edu/algvis/java/SplayTree.html>
- [17] <http://web-cat.cs.vt.edu/AlgovizWiki/SplayTrees>
- [18] <http://www.ibr.cs.tu-bs.de/lehre/ss98/audii/applets/BST/SplayTree-Example.html>
- [19] <http://www.eli.sdsu.edu/courses/fall95/cs660/notes/splay/Splay.html>

T-tree

In computer science a **T-tree** is a type of binary tree data structure that is used by main-memory databases, such as Datablitz, *eXtremeDB*, MySQL Cluster, Oracle TimesTen and Kairos^[1]MobileLite^[2].

A T-tree is a balanced index tree data structure optimized for cases where both the index and the actual data are fully kept in memory, just as a B-tree is an index structure optimized for storage on block oriented secondary storage devices like hard disks. T-trees seek to gain the performance benefits of in-memory tree structures such as AVL trees while avoiding the large storage space overhead which is common to them.



T-trees do not keep copies of the indexed data fields within the index tree nodes themselves. Instead, they take advantage of the fact that the actual data is always in main memory together with the index so that they just contain pointers to the actual data fields.

The "T" in T-tree refers to the shape of the node data structures in the original paper that first described this type of index.^[3]

Performance

Although T-trees seem to be widely used for main-memory databases, recent research indicates that they actually do not perform better than B-trees on modern hardware:

Rao, Jun; Kenneth A. Ross (1999). "Cache conscious indexing for decision-support in main memory"^[4]. *Proceedings of the 25th International Conference on Very Large Databases (VLDB 1999)*. Morgan Kaufmann. pp. 78–89.

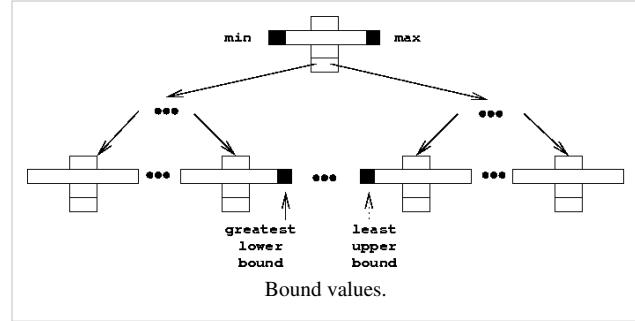
Kim, Kyungwha; Junho Shim, and Ig-hoon Lee (2007). "Cache conscious trees: How do they perform on contemporary commodity microprocessors?". *Proceedings of the 5th International Conference on Computational Science and Its Applications (ICCSA 2007)*. Springer. pp. 189–200. doi:10.1007/978-3-540-74472-6_15.

The main reason seems to be that the traditional assumption of memory references having uniform cost is no longer valid given the current speed gap between cache access and main memory access.

Node structures

A T-tree node usually consists of pointers to the parent node, the left and right child node, an ordered array of data pointers and some extra control data. Nodes with two subtrees are called *internal nodes*, nodes without subtrees are called *leaf nodes* and nodes with only one subtree are named *half-leaf nodes*. A node is called the *bounding node* for a value if the value is between the node's current minimum and maximum value, inclusively.

For each internal node leaf or half leaf nodes exist that contain the predecessor of its smallest data value (called the *greatest lower bound*) and one that contains the successor of its largest data value (called the *least upper bound*). Leaf and half-leaf nodes can contain any number of data elements from one to the maximum size of the data array. Internal nodes keep their occupancy between predefined minimum and maximum numbers of elements



Algorithms

Search

- Search starts at the root node
- If the current node is the bounding node for the search value then search its data array. Search fails if the value is not found in the data array.
- If the search value is less than the minimum value of the current node then continue search in its left subtree. Search fails if there is no left subtree.
- If the search value is greater than the maximum value of the current node then continue search in its right subtree. Search fails if there is no right subtree.

Insertion

- Search for a bounding node for the new value. If such a node exists then
 - check whether there is still space in its data array, if so then insert the new value and finish
 - if no space is available then remove the minimum value from the node's data array and insert the new value. Now proceed to the node holding the greatest lower bound for the node that the new value was inserted to. If the removed minimum value still fits in there then add it as the new maximum value of the node, else create a new right subnode for this node.
- If no bounding node was found then insert the value into the last node searched if it still fits into it. In this case the new value will either become the new minimum or maximum value. If the value doesn't fit anymore then create a new left or right subtree.

If a new node was added then the tree might need to be rebalanced, as described below.

Deletion

- Search for bounding node of the value to be deleted. If no bounding node is found then finish.
- If the bounding node does not contain the value then finish.
- delete the value from the node's data array

Now we have to distinguish by node type:

- Internal node:

If the node's data array now has less than the minimum number of elements then move the greatest lower bound value of this node to its data value. Proceed with one of the following two steps for the half leaf or leaf node the value was removed from.

- Leaf node:

If this was the only element in the data array then delete the node. Rebalance the tree if needed.

- Half leaf node:

If the node's data array can be merged with its leaf's data array without overflow then do so and remove the leaf node. Rebalance the tree if needed.

Rotation and balancing

A T-tree is implemented on top of an underlying self-balancing binary search tree. Specifically, Lehman and Carey's article describes a T-tree balanced like an AVL tree: it becomes out of balance when a node's child trees differ in height by at least two levels. This can happen after an insertion or deletion of a node. After an insertion or deletion, the tree is scanned from the leaf to the root. If an imbalance is found, one tree rotation or pair of rotations is performed, which is guaranteed to balance the whole tree.

When the rotation results in an internal node having fewer than the minimum number of items, items from the node's new child(ren) are moved into the internal node.

See also

- Tree (graph theory)
- Tree (set theory)
- Tree structure
- Exponential tree

Other trees

- B-tree (2-3 tree, B+ tree, B*-tree, UB-tree)
- DSW algorithm
- Dancing tree
- Fusion tree
- kd-tree
- Octree
- Quadtree
- R-tree
- Radix tree
- Skip list
- Matrix tree
- T-tree
- T-pyramid
- Top Trees

External links

- Oracle TimesTen FAQ entry on index mentioning T-Trees [5]
- Oracle Whitepaper: Oracle TimesTen Products and Technologies [6]
- DataBlitz presentation mentioning T-Trees [7]
- An Open-source T*-tree Library [8]

References

- [1] <http://www.kairosdbms.com>
- [2] <http://www.emware.co.kr>
- [3] Tobin J. Lehman and Michael J. Carey, A Study of Index Structures for Main Memory Database Management Systems. VLDB 1986 (<http://www.vldb.org/conf/1986/P294.PDF>)
- [4] <http://www.vldb.org/dblp/db/conf/vldb/RaoR99.html>
- [5] http://www.oracle.com/technology/products/timesten/htdocs/faq/technical_faq.html##6
- [6] http://www.oracle.com/technology/products/timesten/pdf/wp/timesten_tech_wp_dec_2005.pdf
- [7] <http://www.dependability.org/wg10.4/timedepend/08-Rasto.pdf>
- [8] <http://code.google.com/p/ttree/>

Rope

This article is about the data structure. For other uses, see Rope (disambiguation).

In computer programming, a **rope** is a heavyweight string, involving the use of a concatenation tree representation. The concept was introduced in a paper called "Ropes: an Alternative to Strings".^[1]

Description

A rope is essentially a binary tree whose leaves are arrays of characters. A node in the tree has a left child and a right child - the left child is the first part of the string, while the right child is the final part of the string. Concatenation of two ropes simply involves the creation of a new tree node with both ropes as children.

The main advantages of ropes as compared to storing strings as character arrays is that they enable much faster concatenation than ordinary strings, and don't require a large contiguous memory space to store a large string. The main disadvantages are greater overall space usage and slower indexing, both of which become more severe as the tree structure becomes larger and deeper. However, many practical applications of indexing involve only iteration over the string, which remains fast as long as the leaf nodes are large enough to benefit from cache effects.

Comparison to array-based strings

This table compares the *algorithmic* characteristics of string and rope implementations, not their "raw speed". Array-based strings may have smaller overhead, so (for example) concatenation and substring operations may sometimes be faster on small strings than small ropes, but will usually be faster on large ropes than large strings.

Operation	Rope performance	String performance
concatenation	constant	linear
substring	logarithmic	linear
indexing	logarithmic	constant
iteration	linear	linear

See also

- The Cedar programming environment, which used ropes "almost since its inception"^[1]

External links

- SGI's implementation of ropes for C++ ^[2]
- libstdc++ support for ropes ^[3]
- Ropes for Java ^[4]
- Ropes ^[5] for Ocaml

References

- [1] Boehm, Hans-J; Atkinson, Russ; and Plass, Michael (December 1995). "Ropes: an Alternative to Strings" (<http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol25/issue12/spe986.pdf>) (PDF). *Software—Practice & Experience* (New York, NY, USA: John Wiley & Sons, Inc.) **25** (12): 1315–1330. doi:10.1002/spe.4380251203. .
- [2] <http://www.sgi.com/tech/stl/Rope.html>
- [3] <http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.3/a00223.html>
- [4] <http://ahmadsoft.org/ropes/>
- [5] <http://rope.forge.ocamlcore.org/doc/Rope.html>

Top Trees

The **Top Tree** is a binary tree based data structure for unrooted dynamic trees which is used mainly for carrying out various path related operations, it allows simple Divide and conquer algorithms. It has since been augmented to maintain dynamically various properties of a Tree such as Diameter, Center and Median.

A Top Tree \mathfrak{R} is defined for an *underlying tree* T and a pair of vertices ∂T called as External Boundary Vertices

Glossary

Boundary Node

See Boundary Vertex

Boundary Vertex

A vertex in a connected subtree is a *Boundary Vertex* if it is connected to a vertex outside the subtree by an edge.

External Boundary Vertices

Up to a pair of vertices in the Top Tree \mathfrak{R} can be called as External Boundary Vertices, they can be thought of as Boundary Vertices of the cluster which represents the entire Top Tree.

Cluster

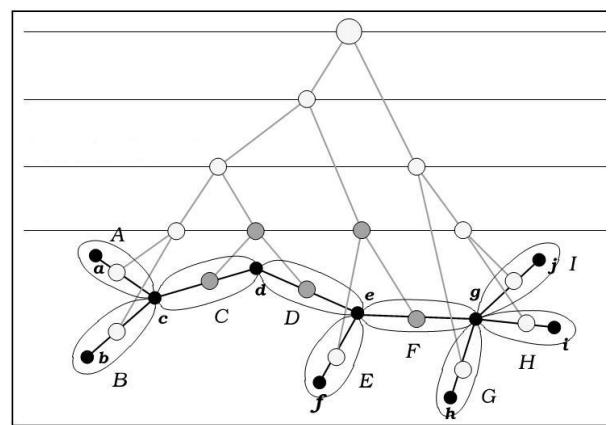
A *cluster* is a connected subtree with at most two Boundary Vertices. The set of Boundary Vertices of a given cluster C is denoted as ∂C . With each cluster C the user may associate some meta information $Info(C)$, and give methods to maintain it under the various internal operations.

Path Cluster

If $\pi(C)$ contains at least one edge then C is called a *Path Cluster*.

Point Cluster

See Leaf Cluster



An image depicting a Top tree built on an underlying tree (black nodes). A tree divided into edge clusters and the complete top-tree for it. Filled nodes in the top-tree are path-clusters, while small circle nodes are leaf-clusters. The big circle node is the root. Capital letters denote clusters, non-capital letters are nodes.

Leaf Cluster

If $\pi(\mathcal{C})$ does not contain any edge i.e \mathcal{C} has only one Boundary Vertex then \mathcal{C} is called a *Leaf Cluster*.

Edge Cluster

A Cluster containing a single edge is called an *Edge Cluster*.

Leaf Edge Cluster

A Leaf in the original Cluster is represented by a Cluster with just a single Boundary Vertex and is called a *Leaf Edge Cluster*.

Path Edge Cluster

Edge Clusters with two Boundary Nodes are called *Path Edge Cluster*.

Internal Node

A node in $\mathcal{C} \setminus \partial\mathcal{C}$ is called an *Internal Node* of \mathcal{C} .

Cluster Path

The path between the Boundary Vertices of \mathcal{C} is called the *cluster path* of \mathcal{C} and it is denoted by $\pi(\mathcal{C})$.

Mergeable Clusters

Two Clusters \mathcal{A} and \mathcal{B} are *Mergeable* if $\mathcal{A} \cap \mathcal{B}$ is a singleton set (they have exactly one node in common) and $\mathcal{A} \cup \mathcal{B}$ is a Cluster.

Introduction

Top Trees are used for maintaining a Dynamic forest (set of trees) under link and cut operations.

The basic idea is to maintain a balanced Binary tree \mathfrak{R} of logarithmic height in the number of nodes in the original tree \mathcal{T} (i.e in $\mathcal{O}(\log n)$ time) ; the **Top Tree** essentially represents the recursive subdivision of the original tree \mathcal{T} into clusters.

In general the tree \mathcal{T} may have weight on its edges.

There is a one to one correspondence with the edges of the original tree \mathcal{T} and the leaf nodes of the Top Tree \mathfrak{R} and each internal node of \mathfrak{R} represents a cluster that is formed due to the union of the clusters that are its children.

The Top Tree data structure can be initialized in $\mathcal{O}(n)$ time. \mathcal{T}

Therefore the Top Tree \mathfrak{R} over $(\mathcal{T}, \partial\mathcal{T})$ is a binary tree such that

- The nodes of \mathfrak{R} are clusters of $(\mathcal{T}, \partial\mathcal{T})$;
- The leaves of \mathfrak{R} are the edges of \mathcal{T} ;
- Sibling clusters are neighbours in the sense that they intersect in a single vertex, and then their parent cluster is their union.
- Root of \mathfrak{R} if the tree \mathcal{T} itself, with a set of at most two External Boundary Vertices.

A tree with a single node has an empty top tree, and one with just an edge is just a single node.

These trees are freely augmentable allowing the user a wide variety of flexibility and productivity without going into the details of the internal workings of the data structure, something which is also referred to as the *Black Box*.

Dynamic Operations

The following two are the user allowable Forest Updates.

- **Link(v, w):** Where v and w are nodes in different trees \mathcal{T}_1 and \mathcal{T}_2 . It returns a single top tree representing $\mathfrak{R}_v \cap \mathfrak{R}_w \cap (v, w)$
- **Cut(v, w):** Removes the Edge (v, w) from a tree \mathcal{T} with Top Tree \mathfrak{R} , thereby turning it into two trees \mathcal{T}_v and \mathcal{T}_w and returning two Top Trees \mathfrak{R}_v and \mathfrak{R}_w .

Expose(v, w): Is called as a subroutine for implementing most of the path related queries on a Top Tree. It makes v and w the External Boundary Vertices of the Top Tree and returns the new Root cluster.

Internal Operations

The Forest updates are all carried out by a sequence of at most $\mathcal{O}(\log n)$ Internal Operations, the sequence of which is computed in further $\mathcal{O}(\log n)$ time.

- **Merge (\mathcal{A}, \mathcal{B}):** Here \mathcal{A} and \mathcal{B} are *Mergeable Clusters*, it returns \mathcal{C} as the parent cluster of \mathcal{A} and \mathcal{B} and with boundary vertices as the boundary vertices of $\mathcal{A} \cup \mathcal{B}$. Updates to $Info(\mathcal{C})$ are carried out accordingly.
- **Split (\mathcal{C}):** Here \mathcal{C} is $\mathcal{A} \cup \mathcal{B}$. This deletes the cluster \mathcal{C} from \mathfrak{R} methods are then called to update $Info(\mathcal{A})$ and $Info(\mathcal{B})$.

The next two functions are analogous to the above two and are used for base clusters.

- **Create (v, w):** Creates a cluster \mathcal{C} for the edge (v, w) . Sets $\partial C = \partial(v, w)$. Methods are then called to compute $Info(\mathcal{C})$.
- **Eradicate (\mathcal{C}):** \mathcal{C} is the edge cluster (v, w) . It deletes the cluster \mathcal{C} from the top tree. The $Info(\mathcal{C})$ is stored by calling a user defined function, as it may also happen that during a tree update, a leaf cluster may change to a path cluster and the converse.

Interesting Results and Applications

A number of interesting applications have been derived for these Top Trees some of them include

- ([SLEATOR AND TARJAN 1983]). We can maintain a dynamic collection of weighted trees in $\mathcal{O}(\log n)$ time per link and cut, supporting queries about the maximum edge weight between any two vertices in $O(\log n)$ time.
 - Proof outline: It involves maintaining at each node the maximum weight (max_wt) on its cluster path, if it is a point cluster then max_wt(\mathcal{C}) is initialised as $-\infty$. When a cluster is a union of two clusters then it is the maximum value of the two merged clusters. If we have to find the max wt between v and w then we do $\mathcal{C} = \text{Expose}(v, w)$, and report max_wt(\mathcal{C}).
- ([SLEATOR AND TARJAN 1983]). In the scenario of the above application we can also add a common weight x to all edges on a given path $v \dots w$ in $\mathcal{O}(\log n)$ time.
 - Proof outline: We introduce a weight called extra(\mathcal{C}) to be added to all the edges in $\pi(\mathcal{C})$. Which is maintained appropriately ; split(\mathcal{C}) requires that, for each path child \mathcal{A} of \mathcal{C} , we set max_wt(\mathcal{A}) := max_wt(\mathcal{A}) + extra(\mathcal{C}) and extra(\mathcal{A}) := extra(\mathcal{A}) + extra(\mathcal{C}). For $\mathcal{C} := \text{join}(\mathcal{A}, \mathcal{B})$, we set max_wt(\mathcal{C}) := max {max_wt(\mathcal{A}), max_wt(\mathcal{B})} and extra(\mathcal{C}) := 0. Finally, to find the maximum weight on the path $v \dots w$, we set $\mathcal{C} := \text{Expose}(v, w)$ and return max_wt(\mathcal{C}).
- ([GOLDBERG ET AL. 1991]). We can ask for the maximum weight in the underlying tree containing a given vertex v in $\mathcal{O}(\log n)$ time.
 - Proof outline: This requires maintaining additional information about the maximum weight non cluster path edge in a cluster under the Merge and Split operations.
- The distance between two vertices v and w can be found in $\mathcal{O}(\log n)$ time as length(Expose (v, w)).

- Proof outline: We will maintain the length $\text{length}(\mathcal{C})$ of the cluster path. The length is maintained as the maximum weight except that, if \mathcal{C} is created by a join(Merge), $\text{length}(\mathcal{C})$ is the sum of lengths stored with its path children.
- Queries regarding diameter of a tree and its subsequent maintenance takes $\mathcal{O}(\log n)$ time.
- The Center and Median can be maintained under Link(Merge) and Cut(Split) operations in $\mathcal{O}(\log n)$ time.

Implementation

Top Trees have been implemented in a variety of ways, some of them include implementation using a *Multilevel Partition* (Top-trees and dynamic graph algorithms Jacob Holm and Kristian de Lichtenberg. Technical Report), and even by using Sleator-Tarjan s-t trees, Fredericksons Topology Trees (Alstrup et al. Maintaining Information in Fully Dynamic Trees with Top Trees).

Using Multilevel Partitioning

Any partitioning of clusters of a tree \mathcal{T} can be represented by a Cluster Partition Tree CPT (\mathcal{T}), by replacing each cluster in the tree \mathcal{T} by an edge. If we use a strategy P for partitioning \mathcal{T} then the CPT would be $\text{CPT}_P \mathcal{T}$. This is done recursively till only one edge remains.

We would notice that all the nodes of the corresponding Top Tree \mathfrak{R} are uniquely mapped into the edges of this multilevel partition. There may be some edges in the multilevel partition that do not correspond to any node in the Top tree, these are the edges which represent only a single child in the level below it, i.e a simple cluster. Only the edges that correspond to composite clusters correspond to nodes in the Top Tree \mathfrak{R} .

A Partitioning Strategy is important while we partition the Tree \mathcal{T} into clusters. Only a careful strategy ensures that we end up in an $\mathcal{O}(\log n)$ height Multilevel Partition (and therefore the Top Tree).

- The number of edges in subsequent levels should decrease by a constant factor.
- If a lower level is changed by an update then we should be able to update the one immediately above it using at most a constant number of insertions and deletions.

The above partitioning strategy ensures the maintenance of the Top Tree in $\mathcal{O}(\log n)$ time.

References

- Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup, *Maintaining information in fully dynamic trees with top trees*, ACM Transactions on Algorithms (TALG), Vol. 1 (2005), 243–264, doi:10.1145/1103963.1103966 [1]
- Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Section 2.3: Trees, pp.308–423.
- 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 10.4: Representing rooted trees, pp.214–217. Chapters 12–14 (Binary Search Trees, Red-Black Trees, Augmenting Data Structures), pp.253–320.

External links

- Maintaining Information in Fully Dynamic Trees with Top Trees. Alstrup et al [2]
- Self Adjusting Top Trees. Tarjan and Werneck [3]
- Self-Adjusting Top Trees. Tarjan and Werneck, Proc. 16th SoDA, 2005 [4]

References

- [1] <http://dx.doi.org/10.1145/1103963.1103966>
[2] <http://arxiv.org/abs/cs.DS/0310065>
[3] <http://www.cs.princeton.edu/~rwerneck/docs/TW05.htm>
[4] <http://portal.acm.org/citation.cfm?id=1070547&dl=&coll=&CFID=15151515&CFTOKEN=6184618>

Tango Trees

Tango Trees is an online binary search tree developed by Erik Demaine, Dion Harmon, John Iacono, and Mihai Patrascu that is $O(\lg \lg(n))$ -competitive.

External links

- "Dynamic Optimality-Almost" [1]

References

- [1] http://theory.csail.mit.edu/~edemaine/papers/Tango_SICOMP/paper.pdf

van Emde Boas tree

A **van Emde Boas tree** (or **van Emde Boas priority queue**), also known as a **vEB tree**, is a tree data structure which implements an associative array with m -bit integer keys. It performs all operations in $O(\log m)$ time. Notice that m is the *size* of the keys — therefore $O(\log m)$ is $O(\log \log n)$ in a full tree, exponentially better than a self-balancing binary search tree. They also have good space efficiency when they contain a large number of elements, as discussed below. They were invented by a team led by Peter van Emde Boas in 1977.^[1]

Supported operations

The operations supported by a vEB tree are those of an *ordered associative array*, which includes the usual associative array operations along with two more *order* operations, *FindNext* and *FindPrevious*:^[2]

- *Insert*: insert a key/value pair with an m -bit key
- *Delete*: remove the key/value pair with a given key
- *Lookup*: find the value associated with a given key
- *FindNext*: find the key/value pair with the smallest key at least a given k
- *FindPrev*: find the key/value pair with the largest key at most a given k

How it works

For the sake of simplicity, let $\log_2 m = k$ for some integer k . Define $M=2^m$. A vEB tree T over the universe $\{0, \dots, M-1\}$ has a root node that stores an array $T.children$ of length $M^{1/2}$. $T.children[i]$ is a pointer to a vEB tree that is responsible for the values $\{iM^{1/2}, \dots, (i+1)M^{1/2}-1\}$. Additionally, T stores two values $T.min$ and $T.max$ as well as an auxiliary vEB tree $T.aux$.

Data is stored in a vEB tree as follows: The smallest value currently in the tree is stored in $T.min$ and largest value is stored in $T.max$. These two values are not stored anywhere else in the vEB tree. If T is empty then we use the convention that $T.max=-1$ and $T.min=M$. Any other value x is stored in the subtree $T.children[i]$ where $i = \lfloor x/M^{1/2} \rfloor$. The auxiliary tree $T.aux$ keeps track of which children are non-empty, so $T.aux$ contains the value j if and only if $T.children[j]$ is non-empty.

FindNext

The operation $FindNext(T, x)$ that searches for the successor of an element x in a vEB tree proceeds as follows: If $x \leq T.min$ then the search is complete, and the answer is $T.min$. Otherwise, let $i=x/M^{1/2}$. If $x \leq T.children[i].max$ then the value being searched for is contained in $T.children[i]$ so the search proceeds recursively in $T.children[i]$. Otherwise, We search for the value i in $T.aux$. This gives us the index j of the first subtree that contains an element larger than x . The algorithm then returns $T.children[j].min$.

```
FindNext (T, x)
  if (x <= T.min)
    return T.min
  i = floor(x/sqrt(M))
  if (x <= T.children[i].max)
    return FindNext (T.children[i], x % sqrt(M))
  return T.children[FindNext (T.aux, i+1)].min
```

Note that, in any case, the algorithm performs $O(1)$ work and then possibly recurses on a subtree over a universe of size $M^{1/2}$ (an $m/2$ bit universe). This gives a recurrence for the running time of $T(m)=T(m/2) + O(1)$, which resolves to $O(\log m) = O(\log \log M)$.

Insert

The call $Insert(T, x)$ that inserts a value x into a vEB tree T operates as follows:

If T is empty then we set $T.min = T.max = x$ and we are done.

Otherwise, if $x < T.min$ then we insert $T.min$ into the subtree i responsible for $T.min$ and then set $T.min = x$. If $T.children[i]$ was previously empty, then we also insert i into $T.aux$

Otherwise, if $x > T.max$ then we insert $T.max$ into the subtree i responsible for $T.max$ and then set $T.max = x$. If $T.children[i]$ was previously empty, then we also insert i into $T.aux$

Otherwise, $T.min < x < T.max$ so we insert x into the subtree i responsible for x . If $T.children[i]$ was previously empty, then we also insert i into $T.aux$.

In code:

```
Insert (T, x)
  if (T.min > T.max)      // T is empty
    T.min = T.max = x;
    return
  if (x < T.min)
    swap (x, T.min)
```

```

if (x > T.max)
    swap(x, T.max)
i = x/sqrt(M)
Insert(T.children[i], x % sqrt(M))
if (T.children[i].min == T.children[i].max)
    Insert(T.aux, i)

```

The key to the efficiency of this procedure is that inserting an element into an empty vEB tree takes $O(1)$ time. So, even though the algorithm sometimes makes two recursive calls, this only occurs when the first recursive call was into an empty subtree. This gives the same running time recurrence of $T(m)=T(m/2) + O(1)$ as before.

Delete

Deletion from vEB trees is the trickiest of the operations. The call $\text{Delete}(T, x)$ that deletes a value x from a vEB tree T operates as follows:

If $T.\min = T.\max = x$ then x is the only element stored in the tree and we set $T.\min = M$ and $T.\max = -1$ to indicate that the tree is empty.

Otherwise, if $x = T.\min$ then we need to find the second-smallest value y in the vEB tree, delete it from its current location, and set $T.\min=y$. The second-smallest value y is either $T.\max$ or $T.\children[T.\aux.\min].\min$, so it can be found in 'O'(1) time. In the latter case we delete y from the subtree that contains it.

Similarly, if $x = T.\max$ then we need to find the second-largest value y in the vEB tree, delete it from its current location, and set $T.\max=y$. The second-largest value y is either $T.\min$ or $T.\children[T.\aux.\max].\max$, so it can be found in 'O'(1) time. In the latter case, we delete y from the subtree that contains it.

Otherwise, we have the typical case where $x \neq T.\min$ and $x \neq T.\max$. In this case we delete x from the subtree $T.\children[i]$ that contains x .

In any of the above cases, if we delete the last element x or y from any subtree $T.\children[i]$ then we also delete i from $T.\aux$

In code:

```

Delete(T, x)
if (T.min == T.max == x)
    T.min = M
    T.max = -1
    return
if (x == T.min)
    if (T.aux is empty)
        T.min = T.max
        return
    else
        x = T.children[T.aux.min].min
        T.min = x
if (x == T.max)
    if (T.aux is empty)
        T.max = T.min
        return
    else
        x = T.children[T.aux.max].max
        T.max = x

```

```

i = floor(x/sqrt(M))
Delete(T.children[i], x%sqrt(M))
if (T.children[i] is empty)
    Delete(T.aux, i)

```

Again, the efficiency of this procedure hinges on the fact that deleting from a vEB tree that contains only one element takes only constant time. In particular, the last line of code only executes if x was the only element in $T.children[i]$ prior to the deletion.

Discussion

The assumption that $\log m$ is an integer is, of course, unnecessary. The operations $x/sqrt(m)$ and $x \%sqrt(m)$ can be replaced by taking only higher-order $ceil(m/2)$ and the lower-order $floor(m/2)$ bits of x , respectively. On any existing machine, this is more efficient than division or remainder computations.

The implementation described above uses pointers and occupies a total space of $O(M)=O(2^m)$, as can be seen by verifying that the solution to the recurrence $S(M) = O(1) + (M^{1/2}+1)*S(M^{1/2})$ is $O(M)$.

In practical implementations, especially on machines with *shift-by-k* and *find first zero* instructions, performance can further be improved by switching to a bit array once m equal to the word size (or a small multiple thereof) is reached. Since all operations on a single word are constant time, this does not affect the asymptotic performance, but it does avoid the majority of the pointer storage and several pointer dereferences, achieving a significant practical savings in time and space with this trick.

An obvious optimization of vEB trees is to discard empty subtrees. This makes vEB trees quite compact when they contain many elements, because no subtrees are created until something needs to be added to them. Initially, each element added creates about $\log(m)$ new trees containing about $m/2$ pointers all together. As the tree grows, more and more subtrees are reused, especially the larger ones. In a full tree of 2^m elements, only $O(2^m)$ space is used. Moreover, unlike a binary search tree, most of this space is being used to store data: even for billions of elements, the pointers in a full vEB tree number in the thousands.

However, for small trees the overhead associated with vEB trees is enormous: on the order of $2^{m/2}$. This is one reason why they are not popular in practice. One way of addressing this limitation is to use only a fixed number of bits per level, which results in a trie. Other structures, including Y-fast trees and X-fast trees have been proposed that have comparable update and query times but use only $O(n)$ space where n is the number of elements stored in the data structure.

References

- [1] Peter van Emde Boas, R. Kaas, and E. Zijlstra: *Design and Implementation of an Efficient Priority Queue* (*Mathematical Systems Theory* 10: 99-127, 1977)
- [2] Gudmund Skovbjerg Frandsen: *Dynamic algorithms: Course notes on van Emde Boas trees (PDF)* (<http://www.daimi.au.dk/~gudmund/dynamicF04/vEB.pdf>) (University of Aarhus, Department of Computer Science)
- Erik Demaine, Shantnu Sen, and Jeff Lindy. Massachusetts Institute of Technology. 6.897: Advanced Data Structures (Spring 2003). Lecture 1 notes: Fixed-universe successor problem, van Emde Boas (http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L1/lecture1.pdf). Lecture 2 notes: More van Emde Boas, ... (http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L2/lecture2.pdf).

Cartesian tree

In computer science, a **Cartesian tree** is a binary tree derived from a sequence of numbers; it can be uniquely defined from the properties that it is heap-ordered and that a symmetric (in-order) traversal of the tree returns the original sequence. Introduced by Vuillemin (1980) in the context of geometric range searching data structures, Cartesian trees have also been used in the definition of the treap and randomized binary search tree data structures for binary search problems. The Cartesian tree for a sequence may be constructed in linear time using a stack-based algorithm for finding all nearest smaller values in a sequence.

Definition

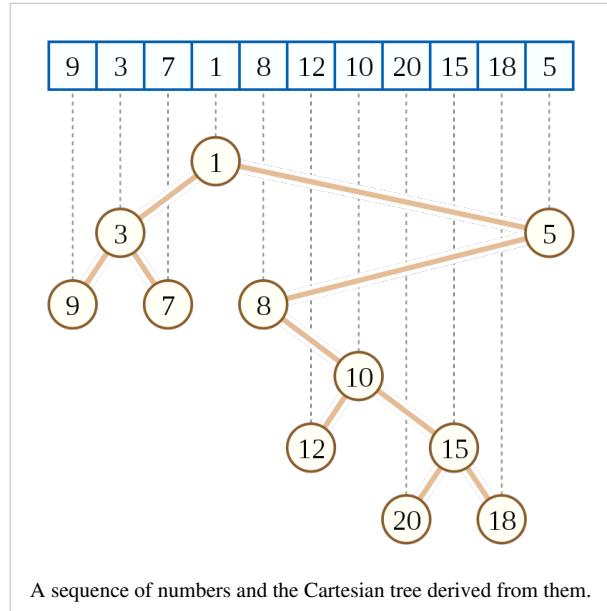
The Cartesian tree for a sequence of distinct numbers can be uniquely defined by the following properties:

1. The Cartesian tree for a sequence has one node for each number in the sequence. Each node is associated with a single sequence value.
2. A symmetric (in-order) traversal of the tree results in the original sequence. That is, the left subtree consists of the values earlier than the root in the sequence order, while the right subtree consists of the values later than the root, and a similar ordering constraint holds at each lower node of the tree.
3. The tree has the heap property: the parent of any non-root node has a smaller value than the node itself.^[1]

Based on the heap property, the root of the tree must be the smallest number in the sequence. From this, the tree itself may also be defined recursively: the root is the minimum value of the sequence, and the left and right subtrees are the Cartesian trees for the subsequences to the left and right of the root value. Therefore, the three properties above uniquely define the Cartesian tree.

If a sequence of numbers contains repetitions, the Cartesian tree may be defined by determining a consistent tie-breaking rule (for instance, determining that the first of two equal elements is treated as the smaller of the two) before applying the above rules.

An example of a Cartesian tree is shown in the figure above.



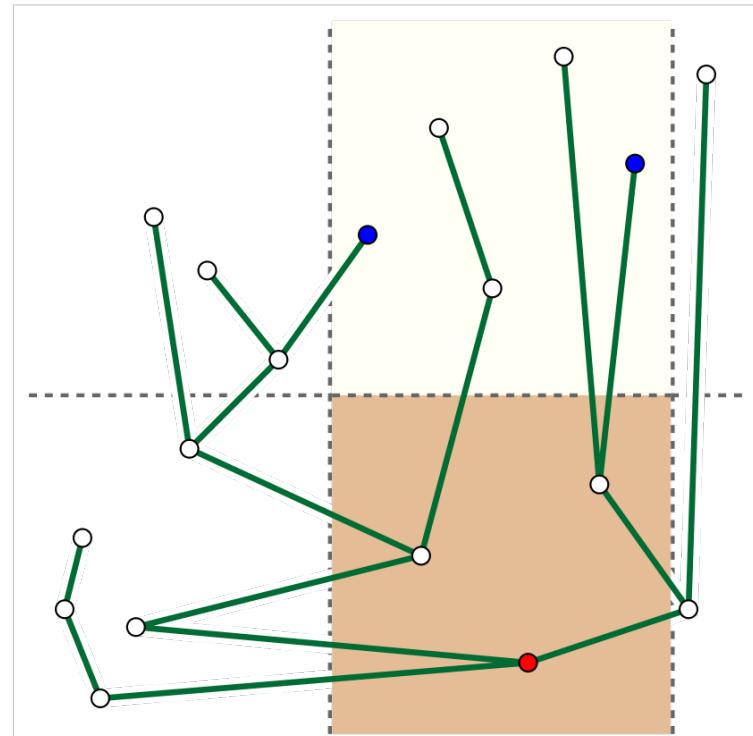
A sequence of numbers and the Cartesian tree derived from them.

Range searching and lowest common ancestors

Cartesian trees may be used as part of an efficient data structure for range minimum queries, a range searching problem involving queries that ask for the minimum value in a contiguous subsequence of the original sequence.^[2] In a Cartesian tree, this minimum value may be found at the lowest common ancestor of the leftmost and rightmost values in the subsequence. For instance, in the subsequence (12,10,20,15) of the sequence shown in the first illustration, the minimum value of the subsequence (10) forms the lowest common ancestor of the leftmost and rightmost values (12 and 15). Because lowest common ancestors may be found in constant time per query, using a data structure that takes linear space to store and that may be constructed in linear time,^[3] the same bounds hold for the range minimization problem.

Bender & Farach-Colton (2000) reversed this relationship between the two data structure problems by showing that lowest common ancestors in an input tree could be solved efficiently applying a non-tree-based technique for range minimization. Their data structure uses an Euler tour technique to transform the input tree into a sequence and then finds range minima in the resulting sequence. The sequence resulting from this transformation has a special form (adjacent numbers, representing heights of adjacent nodes in the tree, differ by ± 1) which they take advantage of in their data structure; to solve the range minimization problem for sequences that do not have this special form, they use Cartesian trees to transform the range minimization problem into a lowest common ancestor problem, and then apply the Euler tour technique to transform the problem again into one of range minimization for sequences with this special form.

The same range minimization problem may also be given an alternative interpretation in terms of two dimensional range searching. A collection of finitely many points in the Cartesian plane may be used to form a Cartesian tree, by sorting the points by their x -coordinates and using the y -coordinates in this order as the sequence of values from which this tree is formed. If S is the subset of the input points within some vertical slab defined by the inequalities $L \leq x \leq R$, p is the leftmost point in S (the one with minimum x -coordinate), and q is the rightmost point in S (the one with maximum y -coordinate) then the lowest common ancestor of p and q in the Cartesian tree is the bottommost point in the slab. A three-sided range query, in which the task is to list all points within a region bounded by the three inequalities $L \leq x \leq R$ and $y \leq T$, may be answered by finding this bottommost point b , comparing its y -coordinate to T , and (if the point lies within the three-sided region) continuing recursively in the two slabs bounded between p and b and between b and q . In this way, after the leftmost and rightmost points in the slab are identified, all points within the three-sided region may be listed in constant time per point.^[4]



Two-dimensional range-searching using a Cartesian tree: the bottom point (red in the figure) within a three-sided region with two vertical sides and one horizontal side (if the region is nonempty) may be found as the nearest common ancestor of the leftmost and rightmost points (the blue points in the figure) within the slab defined by the vertical region boundaries. The remaining points in the three-sided region may be found by splitting it by a vertical line through the bottom point and recursing.

Treaps

Main article: Treap

Because a Cartesian tree is a binary tree, it is natural to use it as a binary search tree for an ordered sequence of values. However, defining a Cartesian tree based on the same values that form the search keys of a binary search tree does not work well: the Cartesian tree of a sorted sequence is just a path, rooted at its leftmost endpoint, and binary searching in this tree degenerates to sequential search in the path. However, it is possible to generate more-balanced search trees by generating *priority* values for each search key that are different than the key itself, sorting the inputs by their key values, and using the corresponding sequence of priorities to generate a Cartesian tree. This construction may equivalently be viewed in the geometric framework described above, in which the x -coordinates of a set of points are the search keys and the y -coordinates are the priorities.

This idea was applied by Seidel & Aragon (1996), who suggested the use of random numbers as priorities. The data structure resulting from this random choice is called a treap, due to its combination of binary search tree and binary heap features. An insertion into a treap may be performed by inserting the new key as a leaf of an existing tree, choosing a priority for it, and then performing tree rotation operations along a path from the node to the root of the tree to repair any violations of the heap property caused by this insertion; a deletion may similarly be performed by a constant amount of change to the tree followed by a sequence of rotations along a single path in the tree.

If the priorities of each key are chosen randomly and independently once whenever the key is inserted into the tree, the resulting Cartesian tree will have the same properties as a random binary search tree, a tree computed by inserting the keys in a randomly chosen permutation starting from an empty tree, with each insertion leaving the previous tree structure unchanged and inserting the new node as a leaf of the tree. Random binary search trees had been studied for much longer, and are known to behave well as search trees (they have logarithmic depth with high probability); the same good behavior carries over to treaps. It is also possible, as suggested by Aragon and Seidel, to reprioritize frequently-accessed nodes, causing them to move towards the root of the treap and speeding up future accesses for the same keys.

Efficient construction

A Cartesian tree may be constructed in linear time from its input sequence. One method is to simply process the sequence values in left-to-right order, maintaining the Cartesian tree of the nodes processed so far, in a structure that allows both upwards and downwards traversal of the tree. To process each new value x , start at the node representing the value prior to x in the sequence and follow the path from this node to the root of the tree until finding a value y smaller than x . This node y is the parent of x , and the previous right child of y becomes the new left child of x . The total time for this procedure is linear, because the time spent searching for the parent y of each new node x can be charged against the number of nodes that are removed from the rightmost path in the tree.^[4]

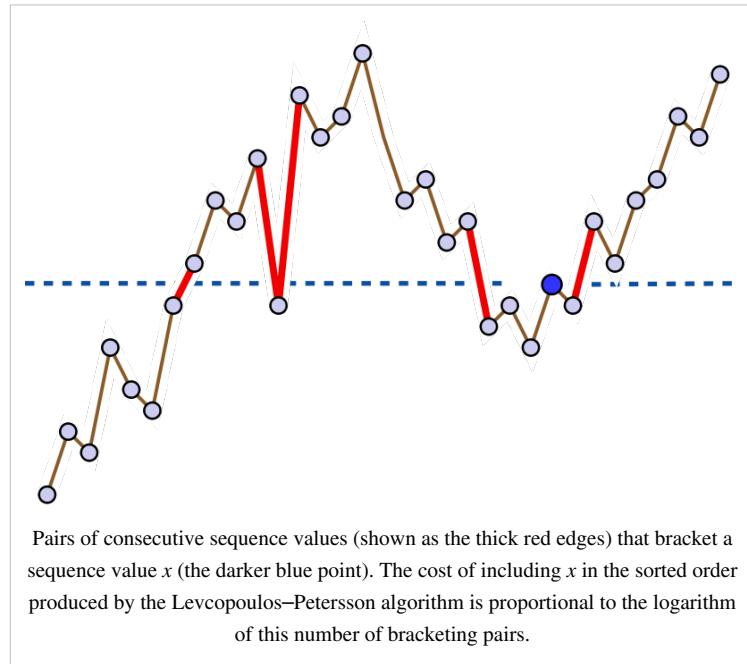
An alternative linear-time construction algorithm is based on the all nearest smaller values problem. In the input sequence, one may define the *left neighbor* of a value x to be the value that occurs prior to x , is smaller than x , and is closer in position to x than any other smaller value. The *right neighbor* is defined symmetrically. The sequence of left neighbors may be found by an algorithm that maintains a stack containing a subsequence of the input. For each new sequence value x , the stack is popped until it is empty or its top element is smaller than x , and then x is pushed onto the stack. The left neighbor of x is the top element at the time x is pushed. The right neighbors may be found by applying the same stack algorithm to the reverse of the sequence. The parent of x in the Cartesian tree is either the left neighbor of x or the right neighbor of x , whichever exists and has a larger value. The left and right neighbors may also be constructed efficiently by parallel algorithms, so this formulation may be used to develop efficient parallel algorithms for Cartesian tree construction.^[5]

Application in sorting

Levcopoulos & Petersson (1989) describe a sorting algorithm based on Cartesian trees. They describe the algorithm as based on a tree with the maximum at the root, but it may be modified straightforwardly to support a Cartesian tree with the convention that the minimum value is at the root. For consistency, it is this modified version of the algorithm that is described below.

The Levcopoulos–Petersson algorithm can be viewed as a version of selection sort or heap sort that maintains a priority queue of candidate minima, and that at each step finds and removes the minimum value in this queue, moving this value to the end of an output sequence. In their algorithm, the priority queue consists only of elements whose parent in the Cartesian tree has already been found and removed. Thus, the algorithm consists of the following steps:

1. Construct a Cartesian tree for the input sequence
2. Initialize a priority queue, initially containing only the tree root
3. While the priority queue is non-empty:
 - Find and remove the minimum value x in the priority queue
 - Add x to the output sequence
 - Add the Cartesian tree children of x to the priority queue



As Levcopoulos and Petersson show, for input sequences that are already nearly sorted, the size of the priority queue will remain small, allowing this method to take advantage of the nearly-sorted input and run more quickly. Specifically, the worst-case running time of this algorithm is $O(n \log k)$, where k is the average, over all values x in the sequence, of the number of consecutive pairs of sequence values that bracket x . They also prove a lower bound stating that, for any n and $k = \omega(1)$, any comparison-based sorting algorithm must use $\Omega(n \log k)$ comparisons for some inputs.

History

Cartesian trees were introduced and named by Vuillemin (1980). The name is derived from the Cartesian coordinate system for the plane: in Vuillemin's version of this structure, as in the two-dimensional range searching application discussed above, a Cartesian tree for a point set has the sorted order of the points by their x -coordinates as its symmetric traversal order, and it has the heap property according to the y -coordinates of the points. Gabow, Bentley & Tarjan (1984) and subsequent authors followed the definition here in which a Cartesian tree is defined from a sequence; this change generalizes the geometric setting of Vuillemin to allow sequences other than the sorted order of x -coordinates, and allows the Cartesian tree to be applied to non-geometric problems as well.

References

- Bender, Michael A.; Farach-Colton, Martin (2000), "The LCA problem revisited" ^[6], *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, Springer-Verlag, Lecture Notes in Computer Science 1776, pp. 88–94.
- Berkman, Omer; Schieber, Baruch; Vishkin, Uzi (1993), "Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values", *Journal of Algorithms* **14** (3): 344–370, doi:10.1006/jagm.1993.101.
- Fischer, Johannes; Heun, Volker (2006), "Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE", *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, **4009**, Springer-Verlag, pp. 36–48, doi:10.1007/11780441_5
- Fischer, Johannes; Heun, Volker (2007), "A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array.", *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, Lecture Notes in Computer Science, **4614**, Springer-Verlag, pp. 459–470, doi:10.1007/978-3-540-74450-4_41
- Gabow, Harold N.; Bentley, Jon Louis; Tarjan, Robert E. (1984), "Scaling and related techniques for geometry problems", *STOC '84: Proc. 16th ACM Symp. Theory of Computing*, New York, NY, USA: ACM, pp. 135–143, doi:10.1145/800057.808675, ISBN 0-89791-133-4.
- Harel, Dov; Tarjan, Robert E. (1984), "Fast algorithms for finding nearest common ancestors", *SIAM J. Comput.* **13** (2): 338–355, doi:10.1137/0213024.
- Levcopoulos, Christos; Petersson, Ola (1989), "Heapsort - Adapted for Presorted Files", *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, **382**, London, UK: Springer-Verlag, pp. 499–509, doi:10.1007/3-540-51542-9_41.
- Seidel, Raimund; Aragon, Cecilia R. (1996), "Randomized Search Trees" ^[7], *Algorithmica* **16** (4/5): 464–497, doi:10.1007/s004539900061.
- Schieber, Baruch; Vishkin, Uzi (1988), "On finding lowest common ancestors: simplification and parallelization", *SIAM J. Comput.* **17**: 1253–1262, doi:10.1137/0217079.
- Vuillemin, Jean (1980), "A unifying look at data structures", *Commun. ACM* (New York, NY, USA: ACM) **23** (4): 229–239, doi:10.1145/358841.358852.

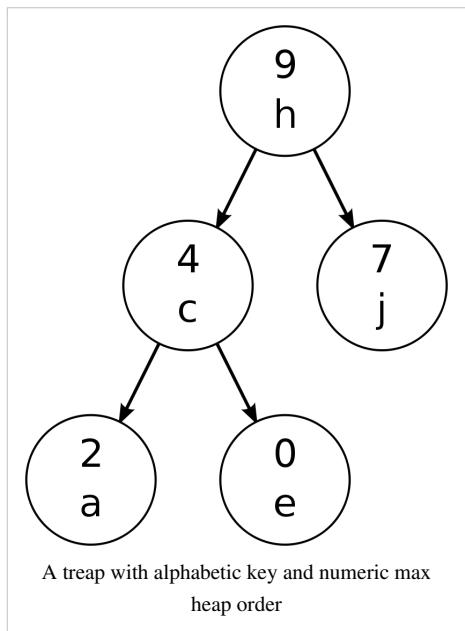
References

- [1] In some references, the ordering is reversed, so the parent of any node always has a larger value and the root node holds the maximum value.
[2] Gabow, Bentley & Tarjan (1984); Bender & Farach-Colton (2000).
[3] Harel & Tarjan (1984); Schieber & Vishkin (1988).
[4] Gabow, Bentley & Tarjan (1984).
[5] Berkman, Schieber & Vishkin (1993).
[6] <http://www.cs.sunysb.edu/~bender/pub/lca.ps>
[7] <http://citeseer.ist.psu.edu/seidel96randomized.html>

Treap

In computer science, the **treap** and the **randomized binary search tree** are two closely-related forms of binary search tree data structures that maintain a dynamic set of ordered keys and allow binary searches among the keys. After any sequence of insertions and deletions of keys, the shape of the tree is a random variable with the same probability distribution as a random binary tree; in particular, with high probability its height is proportional to the logarithm of the number of keys, so that each search, insertion, or deletion operation takes logarithmic time to perform.

Treap



The treap was first described by Cecilia R. Aragon and Raimund Seidel in 1989;^[1] ^[2] its name is a portmanteau of tree and heap. It is a Cartesian tree^[3] in which each key is given a (randomly chosen) numeric priority. As with any binary search tree, the inorder traversal order of the nodes is the same as the sorted order of the keys. The structure of the tree is determined by the requirement that it be heap-ordered: that is, the priority number for any non-root node must be greater than or equal to the priority of its parent. Thus, as with Cartesian trees more generally, the root node is the minimum-priority node, and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

An equivalent way of describing the treap is that it could be formed by inserting the nodes in priority order into a binary search tree without doing any rebalancing. Therefore, if the priorities are independent random numbers (from a distribution over a large enough space of

possible priorities to ensure that two nodes are very unlikely to have the same priority) then the shape of a treap has the same probability distribution as the shape of a random binary search tree, a search tree formed by inserting the nodes without rebalancing in a randomly chosen insertion order. Because random binary search trees are known to have logarithmic height with high probability, the same is true for treaps.

Specifically, the treap supports the following operations:

- To search for a given key value, apply a standard binary search algorithm in a binary search tree, ignoring the priorities.
- To insert a new key x into the treap, generate a random priority y for x . Binary search for x in the tree, and create a new node at the leaf position where the binary search determines a node for x should exist. Then, as long as x is not the root of the tree and has a smaller priority number than its parent z , perform a tree rotation that reverses the parent-child relation between x and z .
- To delete a node x from the heap, if x is a leaf of the tree, simply remove it. If x has a single child z , remove x from the tree and make z be the child of the parent of x (or make z the root of the tree if x had no parent). Finally, if x has two children, swap its position in the tree with the position of its immediate successor z in the sorted order, resulting in one of the previous two cases. In this final case, the swap may violate the heap-ordering property for z , so additional rotations may need to be performed to restore this property.
- To split a treap into two smaller treaps, those smaller than key x , and those larger than key x , insert x into the treap with minimum priority. After this insertion, x will be the root node of the treap, all values less than x will be found in the left subtrep, and all values greater than x will be found in the right subtrep. This costs as much as a

single insertion into the treap.

- Merging two treaps (assumed to be the product of a former split), one can safely assume that the greatest value in the first treap is less than the smallest value in the second treap. Insert a value x , such that x is larger than this max-value in the first treap, and smaller than the min-value in the second treap, and assign it the maximum priority. After insertion it will be a leaf node, and can easily be deleted. The result is one treap merged from the two original treaps. This is effectively "undoing" a split, and costs the same.

Aragon and Seidel also suggest assigning smaller priorities to frequently accessed nodes, for instance by a process that, on each access, chooses a random number and replaces the priority of the node with that number if it is smaller than the previous priority. This modification would cause the tree to lose its random shape; instead, frequently accessed nodes would be more likely to be near the root of the tree, causing searches for them to be faster.

Blelloch and Reid-Miller^[4] describe an application of treaps to a problem of maintaining sets of items and performing set union, set intersection, and set difference operations, using a treap to represent each set. Naor and Nissim^[5] describe another application, for maintaining authorization certificates in public-key cryptosystems.

Randomized binary search tree

The randomized binary search tree, introduced by Martínez and Roura subsequently to the work of Aragon and Seidel on treaps,^[6] stores the same nodes with the same random distribution of tree shape, but maintains different information within the nodes of the tree in order to maintain its randomized structure.

Rather than storing random priorities on each node, the randomized binary search tree stores at each node a small integer, the number of its descendants (counting itself as one); these numbers may be maintained during tree rotation operations at only a constant additional amount of time per rotation. When a key x is to be inserted into a tree that already has n nodes, the insertion algorithm chooses with probability $1/(n + 1)$ to place x as the new root of the tree, and otherwise it calls the insertion procedure recursively to insert x within the left or right subtree (depending on whether its key is less than or greater than the root). The numbers of descendants are used by the algorithm to calculate the necessary probabilities for the random choices at each step. Placing x at the root of a subtree may be performed either as in the treap by inserting it at a leaf and then rotating it upwards, or by an alternative algorithm described by Martínez and Roura that splits the subtree into two pieces to be used as the left and right children of the new node.

The deletion procedure for a randomized binary search tree uses the same information per node as the insertion procedure, and like the insertion procedure it makes a sequence of $O(\log n)$ random decisions in order to join the two subtrees descending from the left and right children of the deleted node into a single tree. If the left or right subtree of the node to be deleted is empty, the join operation is trivial; otherwise, the left or right child of the deleted node is selected as the new subtree root with probability proportional to its number of descendants, and the join proceeds recursively.

Comparison

The information stored per node in the randomized binary tree is simpler than in a treap (a small integer rather than a high-precision random number), but it makes a greater number of calls to the random number generator ($O(\log n)$ calls per insertion or deletion rather than one call per insertion) and the insertion procedure is slightly more complicated due to the need to update the numbers of descendants per node. A minor technical difference is that, in a treap, there is a small probability of a collision (two keys getting the same priority) while in the randomized binary search tree the random choices require fractional probabilities that are not the dyadic rational numbers that can be generated on computers; however, in both cases the differences between the theoretical model of perfect random choices used to design the algorithm and the capabilities of actual random number generators are vanishingly small.

Although the treap and the randomized binary search tree both have the same random distribution of tree shapes after each update, the history of modifications to the trees performed by these two data structures over a sequence of insertion and deletion operations may be different. For instance, in a treap, if the three numbers 1, 2, and 3 are inserted in the order 1, 3, 2, and then the number 2 is deleted, the remaining two nodes will have the same parent-child relationship that they did prior to the insertion of the middle number. In a randomized binary search tree, the tree after the deletion is equally likely to be either of the two possible trees on its two nodes, independently of what the tree looked like prior to the insertion of the middle number.

External links

- Collection of treap references and info ^[7] by Cecilia Aragon
- Treap Applet ^[16] by Kubo Kovac
- Animated treap ^[8]
- Randomized binary search trees ^[9]. Lecture notes from a course by Jeff Erickson at UIUC. Despite the title, this is primarily about treaps and skip lists; randomized binary search trees are mentioned only briefly.
- VB6 implementation of treaps ^[10]. Visual basic 6 implementation of treaps as a COM object.

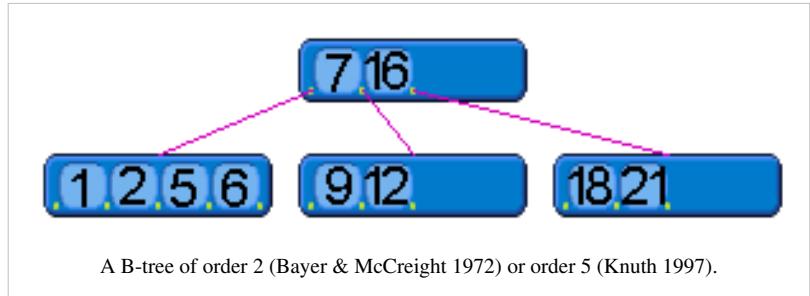
References

- [1] Aragon, Cecilia R.; Seidel, Raimund (1989), "Randomized Search Trees", *Proc. 30th Symp. Foundations of Computer Science (FOCS 1989)*, Washington, D.C.: IEEE Computer Society Press, pp. 540–545, doi:10.1109/SFCS.1989.63531, ISBN 0-8186-1982-1
- [2] Seidel, Raimund; Aragon, Cecilia R. (1996), "Randomized Search Trees" (<http://citeseer.ist.psu.edu/seidel96randomized.html>), *Algorithmica* **16** (4/5): pp. 464–497, doi:10.1007/s004539900061, .
- [3] Vuillemin, Jean (1980), "A unifying look at data structures", *Commun. ACM* (New York, NY, USA: ACM) **23** (4): 229–239, doi:10.1145/358841.358852.
- [4] Blelloch, Guy E.; Reid-Miller, Margaret, (1998), "Fast set operations using treaps", *Proc. 10th ACM Symp. Parallel Algorithms and Architectures (SPAA 1998)*, New York, NY, USA: ACM, pp. 16–26, doi:10.1145/277651.277660, ISBN 0-89791-989-0.
- [5] Naor, M.; Nissim, K. (April 2000), "Certificate revocation and certificate update" (<http://eprints.kfupm.edu.sa/29443/1/29443.pdf>), *IEEE Journal on Selected Areas in Communications* **18** (4): 561–570, doi:10.1109/49.839932, .
- [6] Martínez, Conrado; Roura, Salvador (1998), "Randomized binary search trees" (<http://citeseer.ist.psu.edu/article/martinez97randomized.html>), *Journal of the ACM* **45** (2): 288–323, doi:10.1145/274787.274812, .
- [7] <http://acs.lbl.gov/~aragon/treaps.html>
- [8] <http://www.ibr.cs.tu-bs.de/lehre/ss98/audii/applets/BST/Treap-Example.html>
- [9] <http://www.cs.uiuc.edu/class/sp09/cs473/notes/08-treaps.pdf>
- [10] <http://www.fernando-rodriguez.com/a-high-performance-alternative-to-dictionary>

B-trees

B-tree

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 amortized time. The B-tree is a generalization of a binary search tree in that more than two paths diverge from a single node. (Comer, p. 123)



Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and filesystems.

Overview

In B-trees internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

Each internal node of a B-tree will contain a number of keys. Usually, the number of keys is chosen to vary between d and $2d$. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has $2d$ keys, then adding a key to that node can be accomplished by splitting the $2d$ key node into two d key nodes and adding the key to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbor each have d keys, then a key may be deleted from the internal node by combining with its neighbor. Deleting the key would make the internal node have $d - 1$ keys; joining the neighbor would add d keys plus one more key brought down from the neighbor's parent. The result is an entirely full node of $2d$ keys.

The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-tree is sometimes described with the parameters $(d + 1) — (2d + 1)$ or simply with the highest branching order, $(2d + 1)$.

A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node further away from the root.

B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes. 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, p. 129)
- The B^* -tree balances more neighboring internal nodes to keep the internal nodes more densely packed. (Comer, p. 129) For example, a non-root node of a B-tree must be only half full, but a non-root node of a B^* -tree must be two-thirds full.
- 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)

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^[2]. 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 second.

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.

In practice, if the main database is being frequently searched, the aux-aux index and much of the aux index may reside in a disk cache, so they would not incur a disk read.

Insertions and deletions cause trouble

If the database does not change, then compiling the index is simple to do, and the index need never be changed. If there are changes, then managing the database and its index becomes more complicated.

Deleting records from a database doesn't cause much trouble. The index can stay the same, and the record can just be marked as deleted. The database stays in sorted order. If there are a lot of deletions, then the searching and storage become less efficient.

Insertions are a disaster in a sorted sequential file because room for the inserted record must be made. Inserting a record before the first record in the file requires shifting all of the records down one. Such an operation is just too expensive to be practical.

A trick is to leave some space lying around to be used for insertions. Instead of densely storing all the records in a block, the block can have some free space to allow for subsequent insertions. Those records would be marked as if they were "deleted" records.

Now, both insertions and deletions are fast as long as space is available on a block. If an insertion won't fit on the block, then some free space on some nearby block must be found and the auxiliary indices adjusted. The hope is enough space is nearby that a lot of blocks do not need to be reorganized. Alternatively, some out-of-sequence disk blocks may be used.

The B-tree uses all those ideas

The B-tree uses all the above ideas. It keeps the records in sorted order so they may be sequentially traversed. It uses a hierarchical index to minimize the number of disk reads. The index is elegantly adjusted with a recursive algorithm. The B-tree uses partially full blocks to speed insertions and deletions. In addition, a B-tree minimizes waste by making sure the interior nodes are at least 1/2 full. A B-tree can handle an arbitrary number of insertions and deletions.

Technical description

Terminology

The terminology used for B-trees is inconsistent in the literature:

Unfortunately, the literature on B-trees is not uniform in its use of terms relating to B-Trees. (Folk & Zoellick 1992, p. 362)

Bayer & McCreight (1972), Comer (1979), and others define the **order** of B-tree as the minimum number of keys in a non-root node. Folk & Zoellick (1992) points out that terminology is ambiguous because the maximum number of keys is not clear. An order 3 B-tree might hold a maximum of 6 keys or a maximum of 7 keys. Knuth (1993b) avoids the problem by defining the **order** to be maximum number of children (which is one more than the maximum number of keys).

The term **leaf** is also inconsistent. Bayer & McCreight (1972) considered the leaf level to be the lowest level of keys, but Knuth (1993b) considered the leaf level to be one level below the lowest keys. (Folk & Zoellick 1992, p. 363) There are many possible implementation choices. In some designs, the leaves may hold the entire data record; in other designs, the leaves may only hold pointers to the data record. Those choices are not fundamental to the idea of a B-tree. 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."

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

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

Each internal node's elements act as separation values which divide its subtrees. For example, if an internal node has three child nodes (or subtrees) then it must have two 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 in a B-tree – nodes which are not leaf nodes – are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of U children and – other than the root – a **minimum** of L children. For all internal nodes other than the root, the number of elements is one less than the number of child pointers; the number of elements is between $L-1$ and $U-1$. The number U must be either $2L$ or $2L-1$; thus each internal node is at least half full. This 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 is 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.

Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers.

The root node still has the upper limit on the number of children, 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, and it will have no children at all.

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 the leaf nodes, and so have 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, a specialized structure for leaf nodes in B-trees will improve performance.

Best case and worst case heights

The best case height of a B-Tree is:

$$\log_m n.$$

The worst case height of a B-Tree is:

$$\log_{m/2} n$$

where m is the maximum number of children a node can have.

Algorithms

Warning: the discussion below uses "element", "value", "key", "separator", and "separation value" to mean essentially the same thing. The terms are not clearly defined. There are some subtle issues at the root and leaves.

Search

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value.

Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.

Insertion

All insertions start at a leaf node. To insert a new element

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, so evenly split it into two nodes.
 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. Insert the separation value in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is $U-1$. When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number $U-1$ of elements into two legal nodes. If this number is odd, then $U=2L$ and one of the new nodes contains $(U-2)/2 = L-1$ elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If $U-1$ is even, then $U=2L-1$, so there are $2L-2$ elements in the node. Half of this number is $L-1$, which is the minimum number of elements allowed per node.

An improved algorithm supports a single pass down the tree from the root to the node where the insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage. However, to use this improved algorithm, we must be able to send one element to the parent and split the remaining $U-2$ elements into two legal nodes, without adding a new element. This requires $U = 2L$ rather than $U = 2L-1$, which accounts for why some textbooks impose this requirement in defining B-trees.

Deletion

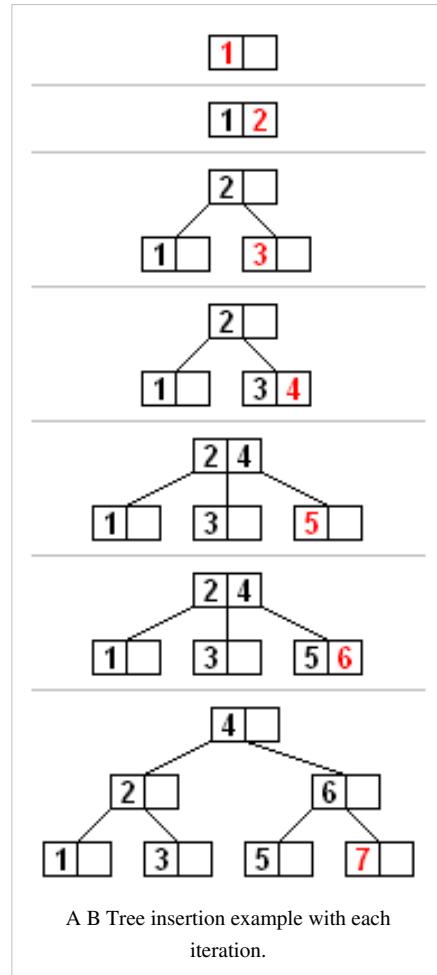
There are two popular strategies for deletion from a B-Tree.

- locate and delete the item, then restructure the tree to regain its invariants
- or
- do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

The algorithm below uses the former strategy.

There are two special cases to consider when deleting an element:

1. the element in an internal node may be a separator for its child nodes



2. deleting an element may put its node under the minimum number of elements and children.

Each of these cases will be dealt with in order.

Deletion from a leaf node

- Search for the value to delete.
- If the value is in a leaf node, it can simply be deleted from the node,
- If underflow happens, check siblings to either transfer a key or fuse the siblings together.
- if deletion happened from right child retrieve the max value of left child if there is no underflow in left child
- in vice-versa situation retrieve the min element from right

Deletion from an internal node

Each element in an internal node acts as a separation value for two subtrees, and when such an element is deleted, two cases arise. In the first case, both of the two child nodes to the left and right of the deleted element have the minimum number of elements, namely $L-1$. They can then be joined into a single node with $2L-2$ elements, a number which does not exceed $U-1$ and so is a legal node. Unless it is known that this particular B-tree does not contain duplicate data, we must then also (recursively) delete the element in question from the new node.

In the second case, one of the two child nodes contains more than the minimum number of elements. Then a new separator for those subtrees must be found. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is the smallest element which is still greater than the separator. Both of those elements are in leaf nodes, and either can be the new separator for the two subtrees.

- If the value is in an internal node, choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
- 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:

- If the right sibling has more than the minimum number of elements
 - Add the separator to the end of the deficient node.
 - Replace the separator in the parent with the first element of the right sibling.
 - Append the first child of the right sibling as the last child of the deficient node
- Otherwise, if the left sibling has more than the minimum number of elements.
 - Add the separator to the start of the deficient node.
 - Replace the separator in the parent with the last element of the left sibling.
 - Insert the last child of the left sibling as the first child of the deficient node
- If both immediate siblings have only the minimum number of elements
 - 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.
 - Remove the separator from the parent, and replace the two children it separated with the combined node.
 - 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 may 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
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

B-trees in filesystems

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 logical block i address into a physical 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 physical block: the operating system just adds the logical block address to the starting physical 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 physical disk block, and that entry identifies the next physical disk block of a file. The result is the disk blocks of a file are in a linked list. In order to find the physical address of block i , the operating system must sequentially search the FAT. For MS/DOS, that was not a huge penalty because the disks were small and the FAT had few entries. In the FAT12 filesystem, there were only 4,096 entries, and the FAT would usually be resident. As disks got bigger, the FAT architecture confronts penalties. It may be necessary to perform disk reads to learn the physical address of a block the user wants to read.

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 a 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+ and Microsoft's NTFS^[3] use B-trees.

Notes

Access concurrency

Lehman and Yao^[4] 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 ^[5] for various strategies to implement node merging, and source code at ^[6].)

See also

- B+ Tree
- Binary Search Tree
- Matrix tree

References

- [1] Counted B-Trees (<http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html>), retrieved 2010-01-25
- [2] 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), page 6
- [3] Mark Russinovich. "Inside Win2K NTFS, Part 1" (<http://msdn2.microsoft.com/en-us/library/ms995846.aspx>). Microsoft Developer Network. . Retrieved 2008-04-18.
- [4] <http://portal.acm.org/citation.cfm?id=319663&dl=GUIDE&coll=GUIDE&CFID=61777986&CFTOKEN=74351190>
- [5] <http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA232287&Location=U2&doc=GetTRDoc.pdf>
- [6] <http://code.google.com/p/high-concurrency-btree/downloads/list>
- 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" (<https://eprints.kfupm.edu.sa/71585/1/71585.pdf>), *Computing Surveys* **11** (2): 123–137. (slow download)
- 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 (1997), *Sorting and Searching*, The Art of Computer Programming, **Volume 3** (Third 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.

Original papers

- Bayer, Rudolf; McCreight, E. (1970), *Organization and Maintenance of Large Ordered Indices*, **Mathematical and Information Sciences Report No. 20**, Boeing Scientific Research Laboratories. July, 1970.
- 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

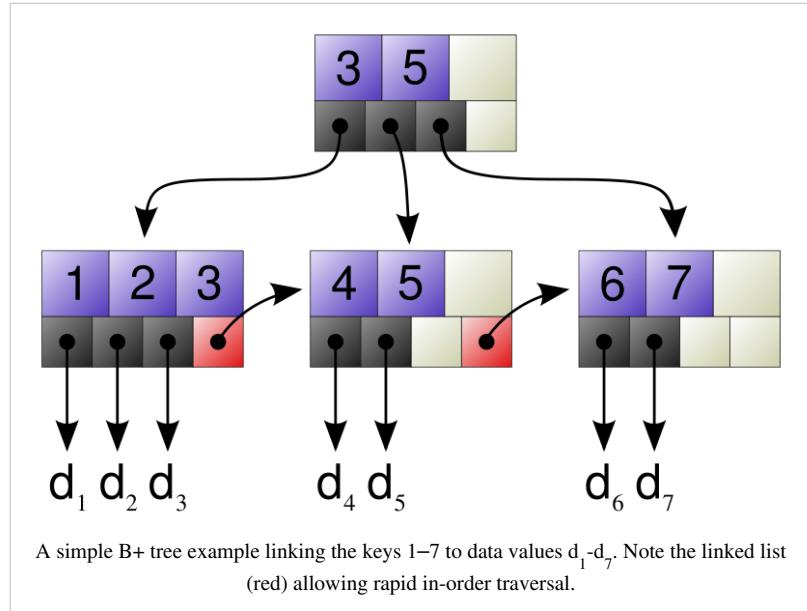
- B-Tree animation applet (<http://slady.net/java/bt/view.php>) by slady
- B-tree and UB-tree on Scholarpedia (http://www.scholarpedia.org/article/B-tree_and_UB-tree) Curator: Dr Rudolf Bayer
- B-Trees: Balanced Tree Data Structures (<http://www.bluerwhite.org/btree>)
- NIST's Dictionary of Algorithms and Data Structures: B-tree (<http://www.nist.gov/dads/HTML/btree.html>)
- C++ source code for a balanced tree (B-tree) (Windows required for test timings) (<http://touc.org/btree.html>)
- WB disk-based B-tree C-library/Java-library/C#-library (<http://people.csail.mit.edu/jaffer/WB>)
- B-Trees in Perl (<http://perl.plover.com/BTree/>)

B+ tree

In computer science, a **B+ tree** (BplusTree) is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a *key*. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment (usually called a "block" or "node"). In a B+ tree, in contrast to a B-tree, all records are stored at the leaf level of the tree; only keys are stored in interior nodes.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

NTFS, ReiserFS, NSS, XFS, and JFS filesystems all use this type of tree for metadata indexing. Relational database management systems such as IBM DB2^[1], Informix^[1], Microsoft SQL Server^[1], Oracle 8^[1], Sybase ASE^[1], PostgreSQL^[2], Firebird, MySQL^[3] and SQLite^[4] support this type of tree for table indices. Key-value database management systems such as Tokyo Cabinet^[5] and Tokyo Tyrant support this type of tree for data access. InfinityDB^[6] is a concurrent BTree.



Details

The order, or branching factor b of a B+ tree measures the capacity of nodes (i.e. the number of children nodes) for internal nodes in the tree. The actual number of children for a node, referred to here as m , is constrained for internal nodes so that $\lceil b/2 \rceil \leq m \leq b$. The root is an exception: it is allowed to have as few as two children. For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children (by definition), but are constrained so that the number of keys must be at least $\lceil (b - 1)/2 \rceil$ and at most $b - 1$. In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node. (The root is also the single leaf, in this case.) This node is permitted to have as little as one key if necessary.

Search

The algorithm to perform a search for a record r follows pointers to the correct child of each node until a leaf is reached. Then, the leaf is scanned until the correct record is found (or until failure).

```
function search(record r)
  u := root
  while (u is not a leaf) do
    choose the correct pointer in the node
    move to the first node following the pointer
```

```

u := current node
scan u for r

```

This pseudocode assumes that no repetition is allowed.

Insertion

- do a search to determine what bucket the new record should go in
- if the bucket is not full, add the record.
- otherwise, split the bucket.
- allocate new leaf and move half the bucket's elements to the new bucket
- insert the new leaf's smallest key and address into the parent.
- if the parent is full, split it also
- now add the middle key to the parent node
- repeat until a parent is found that need not split
- if the root splits, create a new root which has one key and two pointers.

Characteristics

For a b -order B+ tree with h levels of index:

- The maximum number of records stored is $n_{max} = b^h - b^{h-1}$
- The minimum number of keys is $n_{kmin} = \left(\frac{b}{2}\right)^{h-1}$
- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations in the worst case
- Finding a record requires $O(\log_b n)$ operations in the worst case
- Removing a (previously located) record requires $O(\log_b n)$ operations in the worst case
- Performing a range query with k elements occurring within the range requires $O(\log_b n + k)$ operations in the worst case.

Implementation

The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree.

If a storage system has a block size of B bytes, and the keys to be stored have a size of k , arguably the most efficient B+ tree is one where $b = (B/k) - 1$. Although theoretically the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable.

If nodes of the B+ tree are organised as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

B+ trees can also be used for data stored in RAM. In this case a reasonable choice for block size would be the size of processor's cache line. However some studies have proved that a block size few times larger than processor's cache line can deliver better performance if cache prefetching is used.

Space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers. For string keys, space can be saved by using the following technique: Normally the i th entry of an internal block contains the first key of block $i+1$. Instead of storing the full key, we could store the shortest prefix of the first key of block $i+1$ that is strictly greater (in lexicographic order) than last key of block i . There is also a simple way to compress pointers: if we suppose that some consecutive blocks $i, i+1\dots i+k$ are stored contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

All the above compression techniques have some drawbacks. First, a full block must be decompressed to extract a single element. One technique to overcome this problem is to divide each block into sub-blocks and compress them separately. In this case searching or inserting an element will only need to decompress or compress a sub-block instead of a full block. Another drawback of compression techniques is that the number of stored elements may vary considerably from a block to another depending on how well the elements are compressed inside each block.

History

The B tree was first described in the paper *Organization and Maintenance of Large Ordered Indices*. *Acta Informatica 1*: 173–189 (1972) by Rudolf Bayer and Edward M. McCreight. There is no single paper introducing the B+ tree concept. Instead, the notion of maintaining all data in leaf nodes is repeatedly brought up as an interesting variant. An early survey of B trees also covering B+ trees is Douglas Comer: "The Ubiquitous B-Tree" [7], ACM Computing Surveys 11(2): 121–137 (1979). Comer notes that the B+ tree was used in IBM's VSAM data access software and he refers to an IBM published article from 1973.

See also

- Binary Search Tree
- B# Tree
- B-tree
- Bitmap index
- Divide and conquer algorithm

External links

- B+ tree in Python, used to implement a list [8]
- B+ tree implementation as C++ template library [9]
- Dr. Monge's B+ Tree index notes [10]
- Interactive B+ Tree Implementation in C [11]
- Open Source Javascript B+ Tree Implementation [12]
- Perl implementation of B+ trees [13]
- java/C#/python implementations of B+ trees [14]
- Evaluating the performance of CSB+-trees on Multithreaded Architectures [15]
- Effect of node size on the performance of cache conscious B+-trees [16]
- Fractal Prefetching B+-trees [17]
- Towards pB+-trees in the field: implementations Choices and performance [18]
- Cache-Conscious Index Structures for Main-Memory Databases [19]

References

- [1] Ramakrishnan Raghu, Gehrke Johannes - Database Management Systems, McGraw-Hill Higher Education (2000), 2nd edition (en) page 267
- [2] PostgreSQL documentation (<http://www.postgresql.org/docs/8.1/static/gist-extensibility.html>)
- [3] Colin Charles Agenda - Morning sessions at MySQL MiniConf (<http://www.bytebot.net/blog/archives/2008/01/28/morning-sessions-at-mysql-miniconf>)
- [4] SQLite Version 3 Overview (<http://sqlite.org/version3.html>)
- [5] Tokyo Cabinet reference (<http://1978th.net/tokyocabinet/>)
- [6] The Design Of The InfinityDB Database Engine (<http://boilerbay.com/infinitydb/TheDesignOfTheInfinityDatabaseEngine.htm>)
- [7] <http://doi.acm.org/10.1145/356770.356776>
- [8] <http://pypi.python.org/pypi/blist>
- [9] <http://idlebox.net/2007/stx-btree/>
- [10] <http://www.cecs.csulb.edu/%7emonge/classes/share/B+TreeIndexes.html>
- [11] <http://www.amittai.com/prose/bplustree.html>
- [12] <http://blog.conquex.com/?p=84>
- [13] <http://search.cpan.org/~hanenkamp/Tree-BPTree-1.07>
- [14] <http://bplusdotnet.sourceforge.net>
- [15] http://www2.enel.ucalgary.ca/~whassane/papers/CSB+_ccece_2007.pdf
- [16] <http://www.eecs.umich.edu/~jignesh/quickstep/publ/cci.pdf>
- [17] <http://www.pittsburgh.intel-research.net/people/gibbons/papers/fpbptrees.pdf>
- [18] <http://gemo.futurs.inria.fr/events/EXPDB2006/PAPERS/Jonsson.pdf>
- [19] <https://oa.doria.fi/bitstream/handle/10024/2906/cachecon.pdf?sequence=1>

B*-tree

A **B*-tree** is a tree data structure, a variety of B-tree used in the HFS and Reiser4 file systems, which requires non-root nodes to be at least 2/3 full instead of 1/2. To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with the node next to it. When both are full, then the two of them are split into three. It also requires the 'leftmost' key never to be used.

The term is not in general use today as the implementation was never looked on positively by the computer science community-at-large; most people use "B-tree" generically to refer to all the variations and refinements of the basic data structure.

Some modern uses for B*-trees are in floorplanning problems for ICs.^[1]

Variants

A B*-tree should not be confused with a B+ tree, which is one where the leaf nodes of the tree are chained together in the form of a linked list. That is efficient for searching at the cost of a more expensive insertion.

There is also a B**-tree defined by an academic professor listed in the IEEE 0-8186-4212-2 1993.^[2]

External links

- Dictionary of Algorithms and Data Structures entry for B*-tree^[3]

References

- [1] [1] Y. Chang, Y. Chang, G. Wu, and S. Wu, "B*-Trees: a new representation for non-slicing floorplans," Proceedings of the 37th conference on Design automation, ACM New York, NY, USA, 2000, p. 458–463.
- [2] Anestis A. Toptsis (1993-05-27). "B**-tree: a data organization method for high storage utilization" (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=315364). *Computing and Information, 1993. Proceedings ICCI '93..* Sudbury, Ontario, Canada. pp. 277–281. ISBN 0-8186-4212-2. . Retrieved 2007-02-17.
- [3] <http://www.nist.gov/dads/HTML/bstartree.html>

B sharp tree

A **B# tree** is a data structure similar to a B+ tree with rotations allowed among brothers only (immediate siblings).

Insertion Procedure:

Find the node where the new key is to be inserted. If that node is full we must try to perform a rotation with one of our brothers. If all of our brothers are at capacity we must split the node as we do in a B+ Tree.

Dancing tree

In computer science, a **dancing tree** is a tree data structure, which is similar to B+ tree. Invented by Hans Reiser, for use by the Reiser4 file system. As opposed to self-balancing binary search trees that attempt to keep their nodes balanced at all times, dancing trees only balance their nodes when flushing data to a disk (either because of memory constraints or because a transaction has completed).^[1]

The idea behind this is to speed up file system operations by delaying optimization of the tree and only writing to disk when necessary, as writing to disk is thousands of times slower than writing to memory. Also, because this optimization is done less often than with other tree data structures, the optimization can be more extensive.

In some sense, this can be considered to be a self-balancing binary search tree that is optimized for storage on a slow medium, in that the on-disc form will always be balanced but will get no mid-transaction writes; doing so eases the difficulty (at the time) of adding and removing nodes, and instead performs these (slow) rebalancing operations at the same time as the (much slower) write to the storage medium.

However, a (negative) side-effect of this behavior is witnessed in cases of unexpected shutdown, incomplete data writes, and other occurrences that may prevent the final (balanced) transaction from completing. In general, dancing trees will pose a greater difficulty for data recovery from incomplete transactions than a normal tree; though this can be addressed by either adding extra transaction logs or developing an algorithm to locate data on disk not previously present, then going through with the optimizations once more before continuing with any other pending operations/transactions.

External links

- *Software Engineering Based Reiser4 Design Principles* ^[2]
- Description of the Reiser4 internal tree ^[3]

References

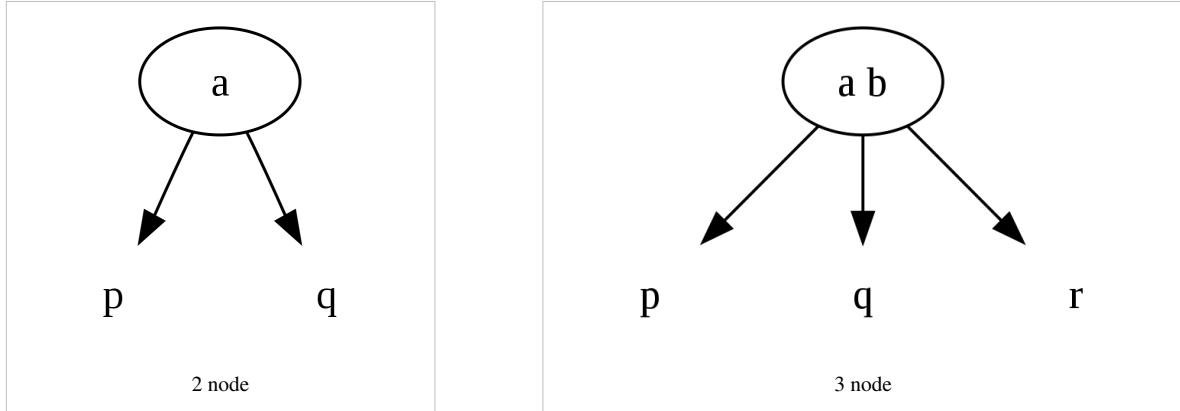
[1] Hans Reiser. "Reiser4 release notes - Dancing Tree" (http://web.archive.org/web/20071024001500/http://www.namesys.com/v4/v4.html#dancing_tree). Archive.org, as Namesys.com is no longer accessible. . Retrieved 2009-07-22.

[2] http://www.namesys.com/v4/v4.html#dancing_tree

[3] <http://nikitadanilov.blogspot.com/2006/03/reiser4-1-internal-tree.html>

2-3 tree

A **2-3 tree** in computer science is a type of data structure, a B-tree where every node with children (internal node) has either two children and one data element (2-nodes) or three children and two data elements (3-nodes). Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements.



2-3 trees are an isometry of AA trees, meaning that they are equivalent data structures. In other words, for every 2-3 tree, there exists at least one AA tree with data elements in the same order. 2-3 trees are balanced, meaning that each right, center, and left subtree contains the same or close to the same amount of data.

Properties

- Every non-leaf is a 2-node or a 3-node. A 2-node contains one data item and has two children. A 3-node contains two data items and has 3 children.
- All leaves are at the same level (the bottom level)
- All data is kept in sorted order
- Every non-leaf node will contain 1 or 2 fields.

Non-leaf nodes

These contain one or two fields which indicate the range of values in its subtrees. If a node has two children, it will have one field; if the node has three children, it will have two fields. Each non-leaf node will contain a value in field 1 which is greater than the largest item in its left sub-tree, but less than or equal to the smallest item in its right sub-tree (or center sub-tree, if it has three children). If that node has three children, field 2 contains a value which is greater than the largest value in the center sub-tree, but less than or equal to the smallest item in its right sub-tree. The purpose of these values is to direct a search function to the correct sub-tree and eventually to the correct data node.

External links

- 2-3 Trees Complete Description ^[1]
- 2-3 Tree Java Applet ^[2]
- 2-3 Tree In-depth description ^[3]
- 2-3 Tree in F# ^[4]

References

- [1] http://www.cs.ucr.edu/cs14/cs14_06win/slides/2-3_trees_covered.pdf
 [2] <http://www.cosc.canterbury.ac.nz/mukundan/dsal/TwoThreeTree.html>
 [3] <http://www.aihorizon.com/essays/basiccs/trees/twothree.htm>
 [4] <http://v2matveev.blogspot.com/2010/03/data-structures-2-3-tree.html>

2-3-4 tree

A **2-3-4 tree** (also called a **2-4 tree**), in computer science, is a self-balancing data structure that is commonly used to implement dictionaries. 2-3-4 trees are B-trees of order 4; like B-trees in general, they can search, insert and delete in $O(\log n)$ time. One property of a 2-3-4 tree is that all external nodes are at the same depth.

2-3-4 trees are an isometry of red-black trees, meaning that they are equivalent data structures. In other words, for every 2-3-4 tree, there exists at least one red-black tree with data elements in the same order. Moreover, insertion and deletion operations on 2-3-4 trees that cause node expansions, splits and merges are equivalent to the color-flipping and rotations in red-black trees. Introductions to red-black trees usually introduce 2-3-4 trees first, because they are conceptually simpler. 2-3-4 trees, however, can be difficult to implement in most programming languages because of the large number of special cases involved in operations on the tree. Red-black trees are simpler to implement, so tend to be used instead.

Magic of the **Lower Left Leaf**. If you are a bit careful when doing fusions, the Lower Left Leaf is always the same node, from begin to end, when working with the tree. So the minimum element can be found in constant time $O(1)$. By using in-order retrieval from that point of p elements the p lowest elements can be found in $O(p \log(n))$ time.

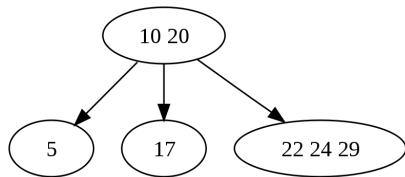
Insertion

To insert a value, we start at the root of the 2-3-4 tree:

1. If the current node is a 4-node:
 - Split the remaining 3-node up into a pair of 2-nodes (the now missing middle value is handled in the next step).
 - If this is the root node (which thus has no parent):
 - the middle value becomes the new root 2-node and the tree height increases by 1. Ascend into the root.
 - Otherwise, push the middle value up into the parent node. Ascend into the parent node.
2. Find the child whose interval contains the value to be inserted.
3. If the child is empty, insert the value into current node and finish.
 - Otherwise, descend into the child and repeat from step 1.^[1] ^[2]

Example

To insert the value "25" into this 2-3-4 tree:



- Begin at the root (10, 20) and descend towards the rightmost child (22, 24, 29). (Its interval $(20, \infty)$ contains 25.)
- Node (22, 24, 29) is a 4-node, so its middle element 24 is pushed up into the parent node.
- The remaining 3-node (22, 29) is split into a pair of 2-nodes (22) and (29). Ascend back into the new parent (10, 20, 24).
- Descend towards the rightmost child (29). (Its interval $(24, \infty)$ contains 25.)
- Node (29) has no rightmost child. (The child for interval $(29, \infty)$ is empty.) Stop here and insert value 25 into this node.

Deletion

Deletion is the more complex operation and involves many special cases.

First the element to be deleted needs to be found. The element must be in a node at the bottom of the tree; otherwise, it must be swapped with another element which precedes it in in-order traversal (which must be in a bottom node) and that element removed instead.

If the element is to be removed from a 2-node, then a node with no elements would result. This is called **underflow**. To solve underflow, an element is pulled from the parent node into the node where the element is being removed, and the vacancy created in the parent node is replaced with an element from a sibling node. (Sibling nodes are those which share the same parent node.) This is called **transfer**.

If the siblings are 2-nodes themselves, underflow still occurs, because now the sibling has no elements. To solve this, two sibling nodes are **fused** together (after pulling element from the parent node).

If the parent is a 2-node, underflow will occur on the parent node. This is solved by using the methods above. This may cause different parent node to sustain underflow as deletions and replacements are being made, referred to as **underflow cascading**.

Deletion in a 2-3-4 tree is $O(\log n)$, while transfer and fusion constant time, $O(1)$.^[1] ^[3]

See also

- B tree
- 2-3 tree
- Red-black tree

External links

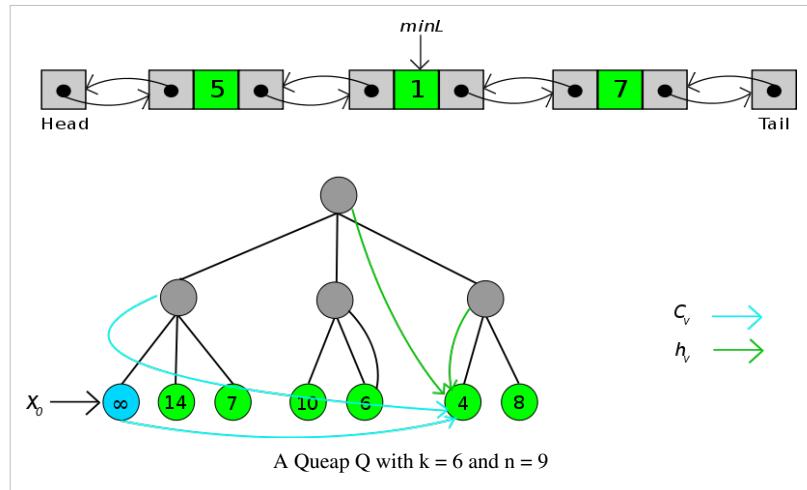
- Animation of a 2-3-4 Tree ^[4]
- Java Applet showing a 2-3-4 Tree ^[5]

References

- [1] Ford, William; Topp, William (2002), *Data Structures with C++ Using STL* (2nd ed.), New Jersey: Prentice Hall, pp. 683, ISBN 0-13-085850-1
- [2] Goodrich, Michael T; Tamassia, Roberto; Mount, David M (2002), *Data Structures and Algorithms in C++*, Wiley, ISBN 0-471-20208-8
- [3] Grama, Ananth (2004). "(2,4) Trees" (<http://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13a.pdf>). *CS251: Data Structures Lecture Notes*. Department of Computer Science, Purdue University. . Retrieved 2008-04-10.
- [4] <http://www.cse.ohio-state.edu/~bondhugu/acads/234-tree/index.shtml>
- [5] <http://www.cs.unm.edu/~rlpm/499/tft.html>

Queaps

In computer science, a **queap** is a priority queue data structure that points to the smallest stored item. Queap is composed of a doubly linked list and a 2-4 tree data structure. The data structure satisfies the queueish property, a complement of the working set property, which makes the search operations of some element x to run in $O(lgq(x))$ amortized time where $q(x)$ is the number of items that has been in the priority queue longer than x .



The doubly linked list keeps a list of k inserted elements. When a deletion operation occurs, the k items are added to the 2-4 tree. The item is then deleted from the tree. Each structure points to the minimum one. The 2-4 tree has been modified for this structure in order to access the smallest element in constant time. The queap was invented by John Iacono and Stefan Langerman [1]

Description

Queap is a priority queue that inserts elements in $O(1)$ amortized time, and removes the minimum element in $O(log(k+2))$ if there are k items in the heap longer than the element to be extracted. The queap has a property called the queueish property: the time to search for element x is $O(lgq(x))$ where $q(x)$ is equal to $n - 1 - w(x)$ and $w(x)$ is the number of distinct items that has been accessed by operations such as searching, inserting, or deleting. $q(x)$ is defined as how many elements have not been accessed since x 's last access. Indeed, the queueish property is the complement of the splay tree working set property: the time to search for element x is $O(lgw(x))$.

Queap can be represented by two data structures: a doubly linked list and a modified version of 2-4 tree. The doubly linked list, L , is used for a series of insert and locate-min operations. The queap keeps a pointer to the minimum element stored in the list. To add element x to list L , the element x is added to the end of the list and a bit variable in element x is set to one. This operation is done to determine if the element is either in the list or in a 2-4 tree.

A 2-4 tree is used when a delete operation occurs. If the item x is already in tree T , the item is removed using the 2-4 tree delete operation. Otherwise, the item x is in list L (done by checking if the bit variable is set). All the elements stored in list L are then added to the 2-4 tree, setting the bit variable of each element to zero. x is then removed from T .

Queap uses only the 2-4 tree structure properties, not a search tree. The modified 2-4 tree structure is as follows. Suppose list L has the following set of elements: $x_1, x_2, x_3, \dots, x_k$. When the deletion operation is invoked, the

set of elements stored in L is then added to the leaves of the 2-4 tree in that order, proceeded by a dummy leaf containing an infinite key. Each internal node of T has a pointer h_v , which points to the smallest item in subtree v . Each internal node on path P from the root to x_0 has a pointer c_v , which points to the smallest key in $T - T_v - \{r\}$. The h_v pointers of internal nodes on path P are ignored. The queap has a pointer to c_{x_0} , which points to the smallest element in T . An application of queap includes a unique set of high priority events and extraction of the highest priority event for processing.

Operations

Let $minL$ be a pointer that points to the minimum element in the doubly linked list L , c_{x_0} be the minimum element stored in the 2-4 tree, T , k be the number of elements stored in T , and n be the total number of elements stored in queap Q . The operations are as follows:

New(Q): Initializes a new empty queap.

Initialize an empty doubly linked list L and 2-4 tree T . Set k and n to zero.

Insert(Q, x): Add the element x to queap Q .

Insert the element x in list L . Set the bit in element x to one to demonstrate that the element is in the list L . Update the $minL$ pointer if x is the smallest element in the list. Increment n by 1.

Minimum(Q): Retrieve a pointer to the smallest element from queap Q .

If $key(minL) < key(c_{x_0})$, return $minL$. Otherwise return c_{x_0} .

Delete(Q, x): Remove element x from queap Q .

If the bit of the element x is set to one, the element is stored in list L . Add all the elements from L to T , setting the bit of each element to zero. Each element is added to the parent of the right most child of T using the insert operation of the 2-4 tree. L becomes empty. Update h_v pointers for all the nodes v whose children are new/modifed, and repeat the process with the next parent until the parent is equal to the root. Walk from the root to node x_0 , and update the c_v values. Set k equal to n .

If the bit of the element x is set to zero, x is a leaf of T . Delete x using the 2-4 tree delete operation. Starting from node x , walk in T to node x_0 , updating h_v and c_v pointers. Decrement n and k by 1.

DeleteMin(Q): Delete and return the smallest element from queap Q .

Invoke the **Minimum(Q)** operation. The operation returns min . Invoke the **Delete(Q, min)** operation. Return min .

CleanUp(Q): Delete all the elements in list L and tree T .

Starting from the first element in list L , traverse the list, deleting each node.

Starting from the root of the tree T , traverse the tree using the post-order traversal algorithm, deleting each node in the tree.

Analysis

The running time is analyzed using the Amortized Analysis tool. The potential function for queap Q will be $\phi(Q) = c|L|$ where $Q = (T, L)$.

Insert(Q, x): The cost of the operation is $O(1)$. The size of list L grows by one, the potential increases by some constant c .

Minimum(Q): The operation does not alter the data structure so the amortized cost is equal to its actual cost, $O(1)$.

Delete(Q, x): There are two cases:

Case 1: If x is in tree T , then the amortized cost is not modified. The delete operation is $O(1)$ amortized 2-4 tree. Since x was removed from the tree, h_v and c_v pointers may need updating. At most, there will be $O(\lg q(x))$ updates.

Case 2: If x is in list L , then all the elements from L are inserted in T . This has a cost of $a|L|$ of some constant a , amortized over the 2-4 tree. After inserting and updating the h_v and c_v pointers, the total time spent is bounded by $2a|L|$.

The second operation is to delete x from T , and to walk on the path from x to x_0 , correcting h_v and c_v values. The time is spent at most $2a|L| + O(lgq(x))$. If $c > 2a$, then the amortized cost will be $O(lgq(x))$.

Delete(Q, x): is the addition of the amortized cost of **Minimum(Q)** and **Delete(Q, x)**, which is $O(lgq(x))$.

Code Example

A small java implementation of a queap:

```
public class Queap
{
    public int n, k;
    public List<Element> l; //Element is a generic data type
    public QueapTree t;     //a 2-4 tree, modified for Queap purpose

    public Element minL;

    private Queap() {
        n = 0;
        k = 0;
        l = new LinkedList<Element>();
        t = new QueapTree();
    }

    public static Queap New() {
        return new Queap();
    }

    public static void Insert(Queap Q, Element x) {
        if (Q.n == 0)
            Q.minL = x;
        Q.l.add(x);
        x.inList = true;
        if (x.compareTo(Q.minL) < 0)
            Q.minL = x;
    }

    public static Element Minimum(Queap Q) {
        //t is a 2-4 tree and x0, cv are tree nodes.
        if (Q.minL.compareTo(Q.t.x0.cv.key) < 0)
            return Q.minL;

        return Q.t.x0.cv.key;
    }

    public static void Delete(Queap Q, QueapNode x) {
```

```
        Q.t.deleteLeaf(x);
        --Q.n;
        --Q.k;
    }

    public static void Delete(Queap Q, Element x) {
        QueapNode n;
        if (x.inList) {
            //set inList of all the elements in the list to
false
            n = Q.t.insertList(Q.l, x);
            Q.k = Q.n;
            Delete(Q, n);
        }
        else if ((n = Q.t.x0.cv).key == x)
            Delete(Q, n);
    }

    public static Element DeleteMin(Queap Q) {
        Element min = Minimum(Q);
        Delete(Q, min);
        return min;
    }
}
```

See Also

- Queue (data structure)
- Priority queue
- Splay tree
- 2-4 tree
- Doubly Linked List
- Amortized analysis

References

- [1] * Iacono, John && Langerman, Stefan: *Queaps*, Springer New York, Algorithmica 42(1): 49-56 (2005)

Fusion tree

A **fusion tree** is a tree data structure that implements an associative array with integer keys up to a fixed size; by exploiting the constant-time machine word multiplication operation available on many real processors, it is able to achieve all operations in

$$O\left(\frac{\log n}{\log \log n}\right)$$

time (see *Big O notation*), which is slightly faster asymptotically than a self-balancing binary search tree.

References

- MIT CS 6.897: Advanced Data Structures: Lecture 4, Fusion Trees^[1], Prof. Erik Demaine

References

[1] http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L4/lecture4.pdf

Bx-tree

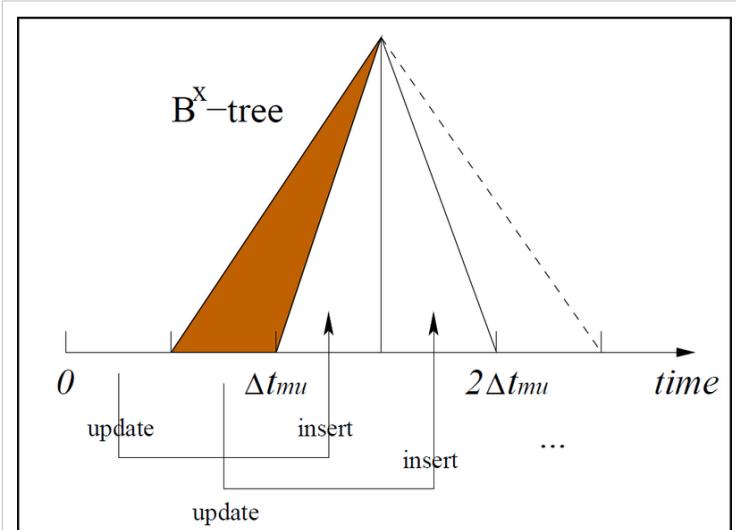
In computer science, the **B^X tree** is a query and update efficient B+ tree-based index structure for moving objects.

Index structure

The base structure of the B^X-tree is a B+ tree in which the internal nodes serve as a directory, each containing a pointer to its right sibling. In the earlier version of the B^X-tree,^[1] the leaf nodes contained the moving-object locations being indexed and corresponding index time. In the optimized version,^[2] each leaf node entry contains the id, velocity, single-dimensional mapping value and the latest update time of the object. The fanout is increased by not storing the locations of moving objects, as these can be derived from the mapping values.

Utilize the B+ tree for moving objects

As for many other moving objects indexes, a 2-dimensional moving object is modeled as a linear function as $O = ((x, y), (vx, vy), t)$, where (x, y) and (vx, vy) are location and velocity of the object at a given time instance t , i.e., the time of last update. The B+ tree is a structure for indexing single dimensional data. In order to adopt the B+ tree as a moving object index, the B^x -tree uses a linearization technique which helps to integrate objects' location at time t into single dimensional value. Specifically, objects are first partitioned according to their update time. For objects within the same partition, the B^x -tree stores their locations at a given time which are estimated by linear interpolation. By doing so, the B^x -tree keeps a consistent view of all objects within the same partition without storing the update time of an objects.



An example of the B^x -tree with the number of index partitions equal to 2 within one maximum update interval tmu . In this example, there are maximum 3 partitions existing at the same time. After linearization, object locations inserted at time 0 are indexed in partition 0 with label timestamp 0.5 tmu , object locations updated during time 0 to 0.5 tmu are indexed in partition 1 with label timestamp tmu , and so on (as indicated by arrows). As time elapses, repeatedly the first range expires (shaded area), and a new range is appended (dashed line).

Secondly, the space is partitioned by a grid and the location of an object is linearized within the partitions according to a space-filling curve, e.g., the Peano or Hilbert curves.

Finally, with the combination of the partition number (time information) and the linear order (location information), an object is indexed in B^x -tree with a one dimensional index key B^x value:

$$B^x\text{value}(O, t) = [\text{indexpartition}]_2 + [xrep]_2$$

Here index-partition is an index partition determined by the update time and xrep is the space-filling curve value of the object position at the indexed time, $[X]_2$ denotes the binary value of x , and “+” means concatenation.

Given an object $O((7, 2), (-0.1, 0.05), 10)$, $tmu = 120$, the B^x value for O can be computed as follows.

1. O is indexed in partition 0 as mentioned. Therefore, indexpartition = $(00)_2$.
2. O 's position at the label timestamp of partition 0 is $(1, 5)$.
3. Using Z-curve with order = 3, the Z-value of O , i.e., xrep is $(010011)_2$.
4. Concatenating indexpartition and xrep, B^x value $(00010011)_2 = 19$.

Insertion, Update and Deletion

Given a new object, its index key is computed and then the object is inserted into the B^x -tree as in the B+ tree. An update consists of a deletion followed by an insertion. An auxiliary structure is employed to keep the latest key of each index so that an object can be deleted by searching for the key. The indexing key is computed before affecting the tree. In this way, the B^x -tree directly inherits the good properties of the B+ tree, and achieves efficient update performance.

Queries

Range query

A range query retrieves all objects whose location falls within the rectangular range $q = ([qx_1, qy_1]; [qx_2; qy_2])$ at time t_q not prior to the current time.

The B^x -tree uses query-window enlargement technique to answer queries. Since the B^x -tree stores an object's location as of sometime after its update time, the enlargement involves two cases: a location must either be brought back to an earlier time or forward to a later time. The main idea is to enlarge the query window so that it encloses all objects whose positions are not within query window at its label timestamp but will enter the query window at the query timestamp.

After the enlargement, the partitions of the B^x -tree need to be traversed to find objects falling in the enlarged query window. In each partition, the use of a space-filling curve means that a range query in the native, two-dimensional space becomes a set of range queries in the transformed, one-dimensional space.^[1]

To avoid excessively large query region after expansion in skewed datasets, an optimization of the query algorithm exists,^[3] which improves the query efficiency by avoiding unnecessary query enlargement.

K nearest neighbor query

K nearest neighbor query is computed by iteratively performing range queries with an incrementally enlarged search region until k answers are obtained. Another possibility is to employ similar queryig ideas in The iDistance Technique.

Other queries

The range query and K Nearest Neighbor query algorithms can be easily extended to support interval queries, continuous queries, etc.^[2]

Adapting relational database engines to accommodate moving objects

Since the B^x -tree is an index built on top of a B+ tree index, all operations in the B^x -tree, including the insertion, deletion and search, are the same as those in the B+ tree. There is no need to change the implementations of these operations. The only difference is to implement the procedure of deriving the indexing key as a stored procedure in an existing DBMS. Therefore the B^x -tree can be easily integrated into existing DBMS without touching the kernel.

SpADE^[4] is moving object management system built on top of a popular relational database system MySQL, which uses the B^x -tree for indexing the objects. In the implementation, moving object data is transformed and stored directly on MySQL, and queries are transformed into standard SQL statements which are efficiently processed in the relational engine. Most importantly, all these are achieved neatly and independently without infiltrating into the MySQL core.

Performance tuning

Potential problem with data skew

The B^x tree uses a grid for space partitioning while mapping two-dimensional location into one-dimensional key. This may introduce performance degradation to both query and update operations while dealing with skewed data. If grid cell is oversize, many objects are contained in a cell. Since objects in a cell are indistinguishable to the index, there will be some overflow nodes in the underlying B+ tree. The existing of overflow pages not only destroys the balancing of the tree but also increases the update cost. As for the queries, for the given query region, large cell incurs more false positives and increases the processing time. On the other hand, if the space is partitioned with finer grid, i.e. smaller cells, each cell contains few objects. There is hardly overflow pages so that the update cost is minimized. Fewer false positives are retrieved in a query. However, more cells are needed to be searched. The increase in the number of cells searched also increases the workload of a query.

Index tuning

The ST^2B -tree^[5] introduces a self-tuning framework for tuning the performance of the B^x -tree while dealing with data skew in space and data change with time. In order to deal with data skew in space, the ST^2B -tree splits the entire space into regions of different object density using a set of reference points. Each region uses an individual grid whose cell size is determined by the object density inside of it.

The B^x -tree have multiple partitions regarding different time intervals. As time elapsed, each partition grows and shrinks alternately. The ST^2B -tree utilizes this feature to tune the index online in order to adjust the space partitioning to make itself accommodate to the data changes with time. In particular, as a partition shrinks to empty and starts growing, it chooses a new set of reference points and new grid for each reference point according to the latest data density. The tuning is based on the latest statistics collected during a given period of time, so that the way of space partitioning is supposed to fit the latest data distribution best. By this means, the ST^2B -tree is expected to minimize the effect caused by data skew in space and data changes with time.

See also

- B+ tree
- Hilbert curve
- Z-order (curve)

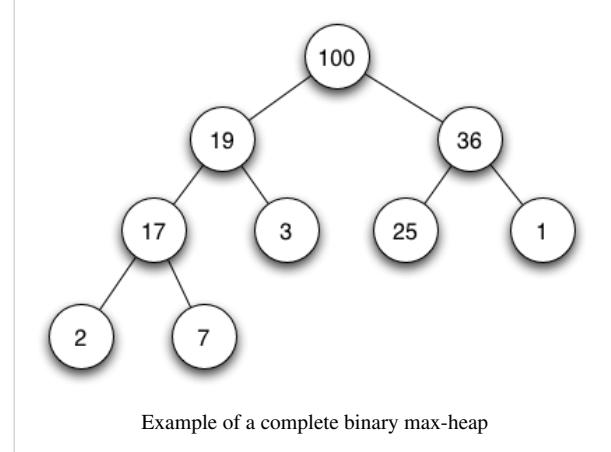
References

- [1] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and Update Efficient B+tree based Indexing of Moving Objects (<http://www.vldb.org/conf/2004/RS20P3.PDF>). In Proceedings of 30th International Conference on Very Large Data Bases (VLDB), pages 768-779, 2004.
- [2] Dan Lin. Indexing and Querying Moving Objects Databases (http://web.mst.edu/~lindan/publication/thesis_lindan.pdf), PhD thesis, National University of Singapore, 2006.
- [3] Jensen, C.S., D. Tiesyte, N. Tradišauskas, Robust B+-Tree-Based Indexing of Moving Objects, in Proceedings of the Seventh International Conference on Mobile Data Management (http://www.cs.aau.dk/~csj/Papers/Files/2006_JensenMDM.pdf), Nara, Japan, 9 pages, May 9–12, 2006.
- [4] SpADE (<http://www.comp.nus.edu.sg/~spade>): A SPatio-temporal Autonomic Database Engine for location-aware services.
- [5] Su Chen, Beng Chin Ooi, Kan-Lee. Tan, and Mario A. Nascimento, ST2B-tree: A Self-Tunable Spatio-Temporal B+-tree for Moving Objects (<http://www.comp.nus.edu.sg/~chensu/sigmod08.pdf>). In Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD), page 29-42, 2008.

Heaps

Heap

In computer science, a **heap** is a specialized tree-based data structure that satisfies the *heap property*: if B is a child node of A , then $\text{key}(A) \geq \text{key}(B)$. This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a *max-heap*. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a *min-heap*.) The several variants of heaps are the prototypical most efficient implementations of the abstract data type priority queues. Priority queues are useful in many applications. In particular, heaps are crucial in several efficient graph algorithms.



The operations commonly performed with a heap are

- *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.

Heaps are used in the sorting algorithm heapsort.

Variants

- 2-3 heap
- Beap
- Binary heap
- Binomial heap
- D-ary heap
- Fibonacci heap
- Leftist heap
- Pairing heap
- Skew heap
- Soft heap
- Ternary heap
- Treap

Comparison of theoretic bounds for variants

The following time complexities^[1] are worst-case time for binary and binomial heaps and amortized time complexity for Fibonacci heap. $O(f)$ gives asymptotic upper bound and $\Theta(f)$ is asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	Binary	Binomial	Fibonacci
findMin	$\Theta(1)$	$\Theta(\log n)$ or $\Theta(1)$	$\Theta(1)$
deleteMin	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
decreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
merge	$\Theta(n)$	$O(\log n)$	$\Theta(1)$

For pairing heaps, the *insert* and *merge* operations have $O(1)$ amortized complexity.^[2] *decreaseKey* does not have $O(1)$ amortized complexity.^[3] ^[4]

Heap applications

The heap data structure has many applications.

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Selection algorithms: Finding the min, max, both the min and max, median, or even the k -th largest element can be done in linear time using heaps.
- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by an order of polynomial. Examples of such problems are Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

Full and almost full binary heaps may be represented in a very space-efficient way using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc. Thus the children of the node at position n would be at positions $2n$ and $2n+1$ in a one-based array, or $2n+1$ and $2n+2$ in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by swapping elements which are out of order. As we can build a heap from an array without requiring extra memory (for the nodes, for example), heapsort can be used to sort an array in-place.

One more advantage of heaps over trees in some applications is that construction of heaps can be done in linear time using Tarjan's algorithm.

Heap implementations

- The C++ Standard Template Library provides the `make_heap`, `push_heap` and `pop_heap` algorithms for 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 detailed above.
- The Java 2 platform (since version 1.5) provides the binary heap implementation with class `java.util.PriorityQueue<E>` in Java Collections Framework.
- Python has a `heapq` module that implements a priority queue using a binary heap.
- PHP has both `maxheap` (`SplMaxHeap`) and `minheap` (`SplMinHeap`) as of version 5.3 in the Standard PHP Library.
- Perl has implementations of binary, binomial, and Fibonacci heaps in the `Heap` ^[5] distribution available on CPAN

See also

- Sorting algorithm
- Fibonacci heap

References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (1990): Introduction to algorithms. MIT Press / McGraw-Hill.
- [2] 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.
- [3] "CiteSeerX — Pairing Heaps are Sub-optimal" (<http://citeseer.ist.psu.edu/112443.html>). Citeseer.ist.psu.edu. . Retrieved 2009-09-23.
- [4] "On the efficiency of pairing heaps and related data structures" (<http://portal.acm.org/citation.cfm?id=320214>). Portal.acm.org. 2003-07-31. doi:10.1007/BF01840439. . Retrieved 2009-09-23.
- [5] <http://search.cpan.org/perldoc?Heap>

Binary heap

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:

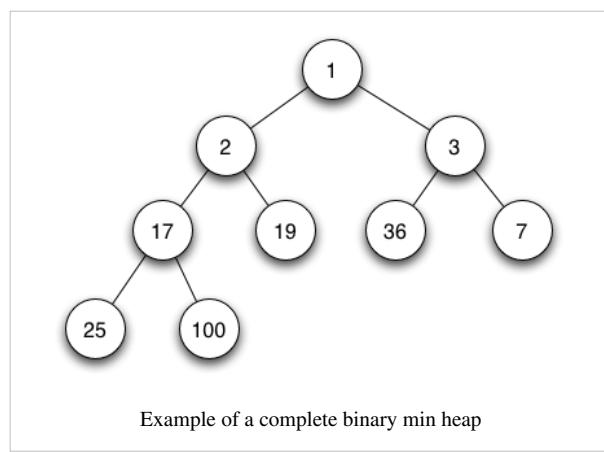
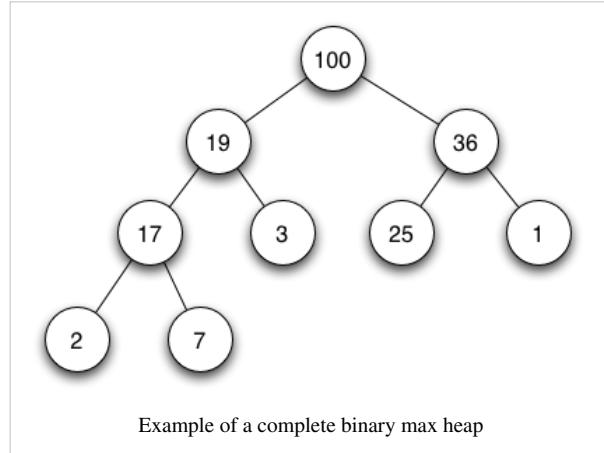
- The *shape property*: the tree is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- The *heap property*: each node is greater than or equal to each of its children according to some comparison predicate which is fixed for the entire data structure.

"Greater than or equal to" means according to whatever comparison function is chosen to sort the heap, not necessarily "greater than or equal to" in the mathematical sense (since the quantities are not always numerical). Heaps where the comparison function is mathematical "greater than or equal to" are called max-heaps; those where the comparison function is mathematical "less than" are called "min-heaps". Conventionally, min-heaps are used, since they are readily applicable for use in priority queues.

Note that the ordering of siblings in a heap is not specified by the heap property, so the two children of a parent can be freely interchanged, as long as this does not violate the shape and heap properties (compare with treap).

The binary heap is a special case of the d-ary heap in which $d = 2$.

It is possible to modify the heap structure to allow extraction of both the smallest and largest element in $O(\log n)$ time.^[1] To do this the rows alternate between min heap and max heap. The algorithms are roughly the same, but in each step must consider the alternating rows with alternating comparisons. The performance is roughly the same as a normal single direction heap. This idea can be generalised to a min-max-median heap.



Heap operations

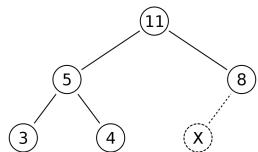
Adding to the heap

If we have a heap, and we add an element, we can perform an operation known as *up-heap*, *bubble-up*, *percolate-up*, *sift-up*, or *heapify-up* in order to restore the heap property. We can do this in $O(\log n)$ time, using a binary heap, by following this algorithm:

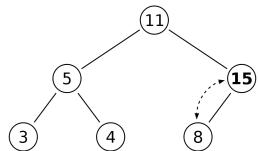
1. Add the element on the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

We do this at maximum for each level in the tree—the height of the tree, which is $O(\log n)$. However, since approximately 50% of the elements are leaves and 75% are in the bottom two levels, it is likely that the new element to be inserted will only move a few levels upwards to maintain the heap. Thus, binary heaps support insertion in *average* constant time, $O(1)$.

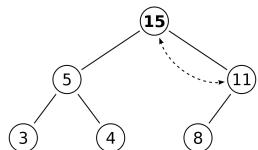
Say we have a max-heap



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:



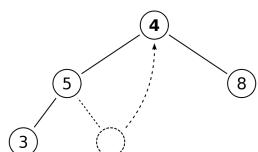
However the heap property is still violated since 15 is greater than 11, so we need to swap again:



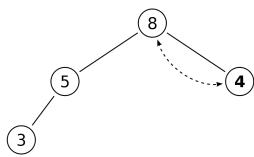
which is a valid max-heap.

Deleting the root from the heap

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—starts by replacing it with the last element on the last level. 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. The operation that restores the property is called *down-heap*, *bubble-down*, *percolate-down*, *sift-down*, or *heapify-down*. 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 .

Max-Heapify^[2] (A, i):

```

 $left \leftarrow 2i$ 
 $right \leftarrow 2i + 1$ 
if  $left \leq \text{heap-length}[A]$  and  $A[left] > A[i]$  then:
     $largest \leftarrow left$ 
else:
     $largest \leftarrow i$ 
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( $A, largest$ )

```

Note that the down-heap operation (without the preceding swap) can be used in general to modify the value of the root, even when an element is not being deleted.

Building a heap

A heap could be built by successive insertions. This approach requires $O(n \lg n)$ time because each insertion takes $O(\lg n)$ time and there are n elements (' $\lg()$ ' denotes a binary logarithm here.) However this is not the optimal method. The optimal method starts by arbitrarily putting the elements on a binary tree (which could be represented by an array, see below). Then starting from the lowest level and moving upwards until the heap property is restored by shifting the root of the subtree downward as in the deletion algorithm. 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 children when building a max-heap, or minimum children when building a min-heap). This process takes $O(h)$ operations (swaps). In this method most of the heapification takes place in the lower levels. The number of nodes at height h is $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$. Therefore, the cost

of heapifying all subtrees is:

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

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

The **Build-Max-Heap** function that follows, converts an array A which stores a complete binary tree with n nodes to a max-heap by repeatedly using **Max-Heapify** in a bottom up manner. It is based on the observation that the array elements indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ are all leaves for the tree, thus each is an one-element heap.

Build-Max-Heap runs **Max-Heapify** on each of the remaining tree nodes.

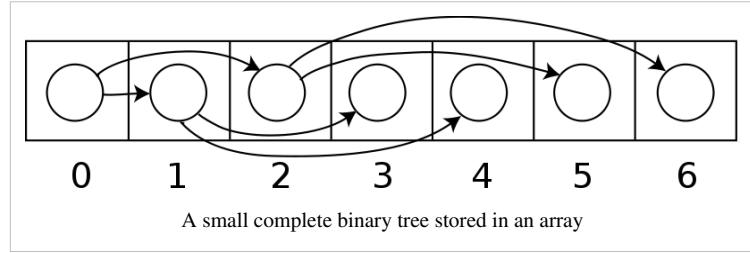
Build-Max-Heap^[2](A):

```
heap-length[A] ← length[A]
for i ← floor(length[A]/2) downto 1 do
    Max-Heapify(A, i)
```

Heap implementation

It is perfectly acceptable to use a traditional binary tree data structure to implement a binary heap. There is an issue with finding the adjacent element on the last level on the binary heap when adding an element which can be resolved algorithmically or by adding extra data to the nodes, called "threading" the tree—that is, instead of merely storing references to the children, we store the inorder successor of the node as well.

However, a more common approach, and an approach aligned with the theory behind heaps, is to store the heap in 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 simple arithmetic on array indices. Details depend on the root position (which in turn may depend on constraints of a programming language used for implementation). If the tree root item has index 0 (n tree elements are $a[0] \dots a[n-1]$), then for each index i , element $a[i]$ has children $a[2i+1]$ and $a[2i+2]$, and the parent $a[\text{floor}((i-1)/2)]$, as shown in the figure. If the root is $a[1]$ (tree elements are $a[1] \dots a[n]$), then for each index i , element $a[i]$ has children $a[2i]$ and $a[2i+1]$, and the parent $a[\text{floor}(i/2)]$. This is a simple example of an implicit data structure or Ahnentafel list.



This approach is particularly useful 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 in-place). However it requires allocating the array before filling it, which makes this method not that useful in priority queues implementation, where the number of tasks (heap elements) is not necessarily known in advance.

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.

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.^[3] When merging is a common task, a different heap implementation is recommended, such as binomial heaps, which can be merged in $O(\log n)$.

See also

- Heap
- Heapsort

External links

- Binary Heap Applet ^[16] by Kubo Kovac
- Heap from Wolfram MathWorld ^[4]
- Using Binary Heaps in A* Pathfinding ^[5]
- Java Implementation of Binary Heap ^[6]

References

- [1] Atkinson, M.D., J.-R. Sack, N. Santoro, and T. Strothotte (1 October 1986). "Min-max heaps and generalized priority queues." (<http://cgs.scs.carleton.ca/~morin/teaching/5408/refs/minmax.pdf>). Programming techniques and Data structures. Comm. ACM, 29(10): 996–1000. .
- [2] Cormen, T. H. & al. (2001), *Introduction to Algorithms* (2nd ed.), Cambridge, Massachusetts: The MIT Press, ISBN 0070131511
- [3] <http://nist.gov/dads/HTML/binaryheap.html>
- [4] <http://mathworld.wolfram.com/Heap.html>
- [5] <http://www.policyalmanac.org/games/binaryHeaps.htm>
- [6] http://sites.google.com/site/indy256/algo-en/binary_heap

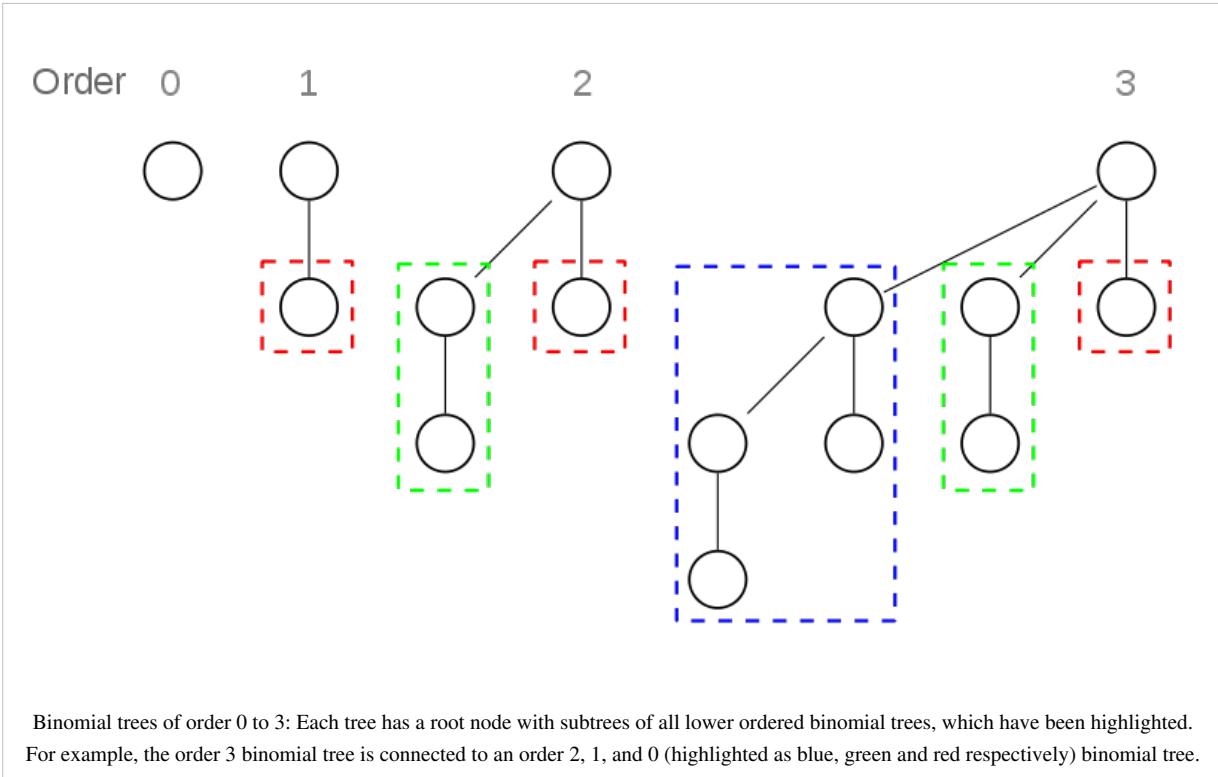
Binomial heap

In computer science, a **binomial heap** is a heap similar to a binary heap but also supports quickly merging two heaps. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation.

Binomial tree

A binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order k has a root node whose children are roots of binomial trees of orders $k-1, k-2, \dots, 2, 1, 0$ (in this order).



A binomial tree of order k has 2^k nodes, height k .

Because of its unique structure, a binomial tree of order k can be constructed from two trees of order $k-1$ trivially by attaching one of them as the leftmost child of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.

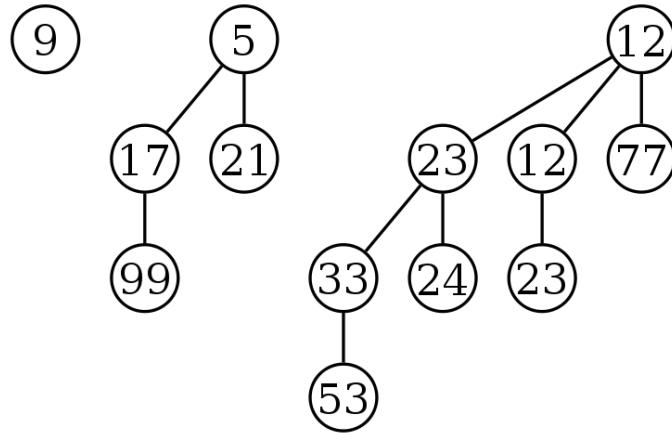
Structure of a binomial heap

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can only be either *one* or *zero* binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

The second property implies that a binomial heap with n elements consists of at most $\log n + 1$ binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of elements n : each binomial tree corresponds to digit one in the binary representation of number n . For example number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, and thus a binomial heap with 13 elements will consist of three binomial trees of orders 3, 2, and 0 (see figure below).



Example of a binomial heap containing 13 elements with distinct keys.

The heap consists of three binomial trees with orders 0, 2, and 3.

Implementation

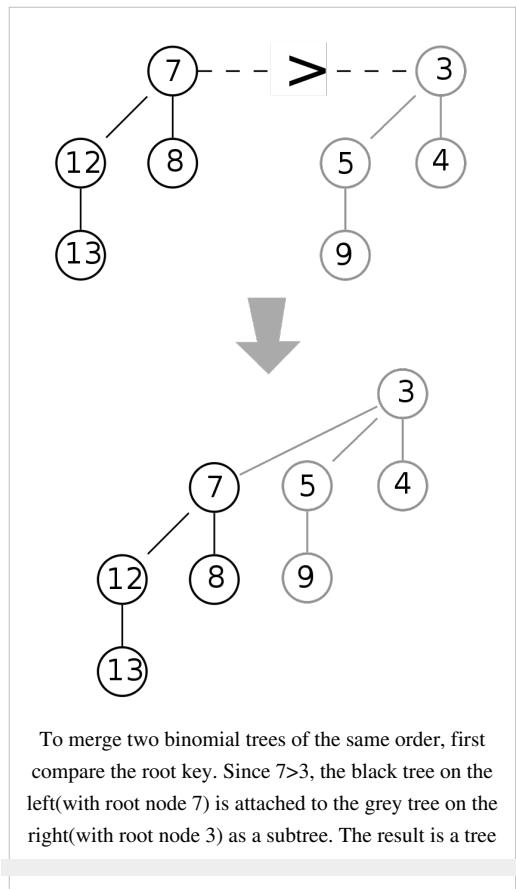
Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.

Merge

As mentioned above, the simplest and most important operation is the merging of two binomial trees of the same order within two binomial heaps. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree become a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

```

function mergeTree(p, q)
  if p.root <= q.root
    return p.addSubTree(q)
  else
    return q.addSubTree(p)
  
```



The operation of **merging** two heaps is perhaps the most interesting and can be used as a subroutine in most other operations. The lists of roots of both heaps are traversed simultaneously, similarly as in the merge algorithm.

If only one of the heaps contains a tree of order j , this tree is moved to the merged heap. If both heaps contain a tree of order j , the two trees are merged to one tree of order $j+1$ so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order $j+1$ present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Each tree has order at most $\log n$ and therefore the running time is $O(\log n)$.

```
function merge (p, q)
    while not ( p.end() and q.end() )
        tree = mergeTree(p.currentTree(), q.currentTree())
        if not heap.currentTree().empty()
            tree = mergeTree(tree, heap.currentTree())
            heap.addTree(tree)
        else
            heap.addTree(tree)
        heap.next() p.next() q.next()
```

Insert

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes $O(\log n)$ time, however it has an *amortized* time of $O(1)$ (i.e. constant).

Find minimum

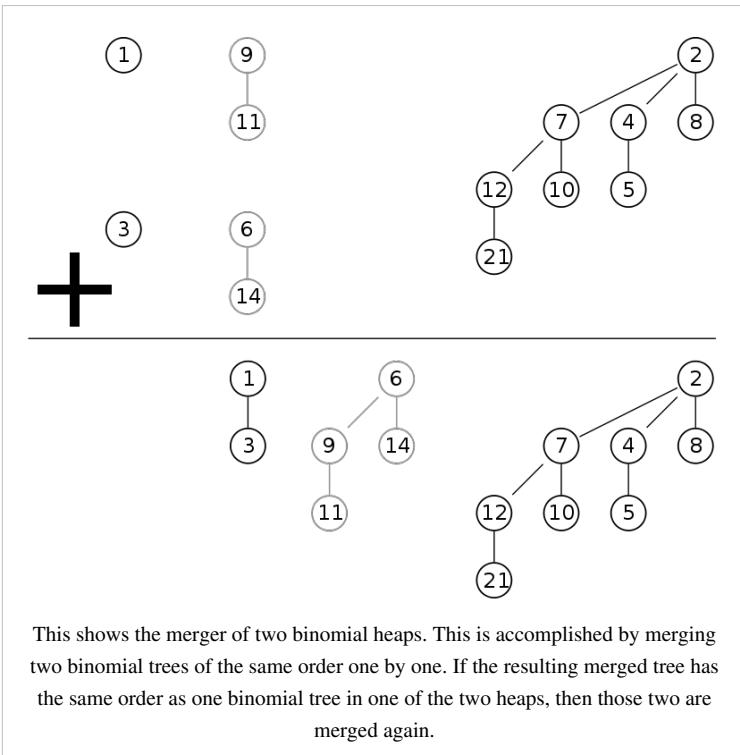
To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in $O(\log n)$ time, as there are just $O(\log n)$ trees and hence roots to examine.

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to $O(1)$. The pointer must be updated when performing any operation other than Find minimum. This can be done in $O(\log n)$ without raising the running time of any operation.

Delete minimum

To **delete the minimum element** from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees. Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap.

```
function deleteMin(heap)
    min = heap.trees().first()
    for each current in heap.trees()
        if current.root < min then min = current
    for each tree in min.subTrees()
        tmp.addTree(tree)
    heap.removeTree(min)
    merge(heap, tmp)
```



This shows the merger of two binomial heaps. This is accomplished by merging two binomial trees of the same order one by one. If the resulting merged tree has the same order as one binomial tree in one of the two heaps, then those two are merged again.

Decrease key

After **decreasing** the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most $\log n$, so this takes $O(\log n)$ time.

Delete

To **delete** an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

Performance

All of the following operations work in $O(\log n)$ time on a binomial heap with n elements:

- Insert a new element to the heap
- Find the element with minimum key
- Delete the element with minimum key from the heap
- Decrease key of a given element
- Delete given element from the heap
- Merge two given heaps to one heap

Finding the element with minimum key can also be done in $O(1)$ by using an additional pointer to the minimum.

See also

- Fibonacci heap
- Soft heap

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. Chapter 19: Binomial Heaps, pp.455–475.
- Vuillemin, J. (1978). A data structure for manipulating priority queues.^[1] *Communications of the ACM* **21**, 309–314.

External links

- Java applet simulation of binomial heap^[2]
- Python implementation of binomial heap^[3]
- Two C implementations of binomial heap^[4] (a generic one and one optimized for integer keys)
- Java implementation of binomial heap^[5]

References

- [1] <http://portal.acm.org/citation.cfm?id=359478>
- [2] <http://www.cs.yorku.ca/~aaw/Sotirios/BinomialHeap.html>
- [3] <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/511508>
- [4] http://www.cs.unc.edu/~bbb/#binomial_heaps
- [5] <http://niketanpansare.com/MinBinHeap.aspx>

Fibonacci heap

In computer science, a **Fibonacci heap** is a heap data structure consisting of a forest of trees. It has a better amortized running time than a binomial heap. Fibonacci heaps were developed by Michael L. Fredman and Robert E. Tarjan in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from Fibonacci numbers which are used in the running time analysis.

Operations insert, find minimum, decrease key, and merge (union) work in constant amortized time. Operations delete and delete minimum work in $O(\log n)$ amortized time. This means that starting from an empty data structure, any sequence of a operations from the first group and b operations from the second group would take $O(a + b \log n)$ time. In a binomial heap such a sequence of operations would take $O((a + b)\log(n))$ time. A Fibonacci heap is thus better than a binomial heap when b is asymptotically smaller than a .

Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing shortest paths in a graph, and Prim's algorithm for computing a minimum spanning tree of a graph.

Structure

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree or a single tree of depth n . This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree.

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2} , where F_k is the k th Fibonacci number. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased in the operation *delete minimum*, where trees are linked together.

As a result of a relaxed structure, some operations can take a long time while others are done very quickly. In the amortized running time analysis we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

$$\text{Potential} = t + 2m$$

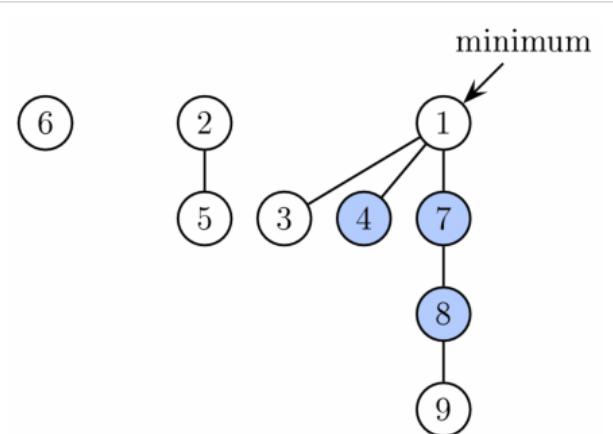


Figure 1. Example of a Fibonacci heap. It has three trees of degrees 0, 1 and 3. Three vertices are marked (shown in blue). Therefore the potential of the heap is 9.

where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked).

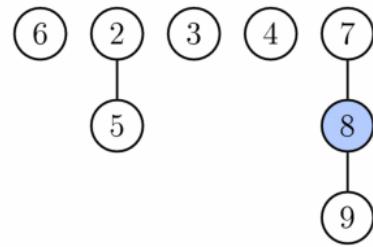
Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

Implementation of operations

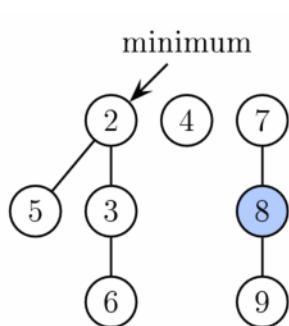
To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost is constant. As mentioned above, **merge** is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time. Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

Operation **extract minimum** (same as *delete minimum*) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d , it takes time $O(d)$ to process all new roots and the potential increases by $d-1$. Therefore the amortized running time of this phase is $O(d) = O(\log n)$.



Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

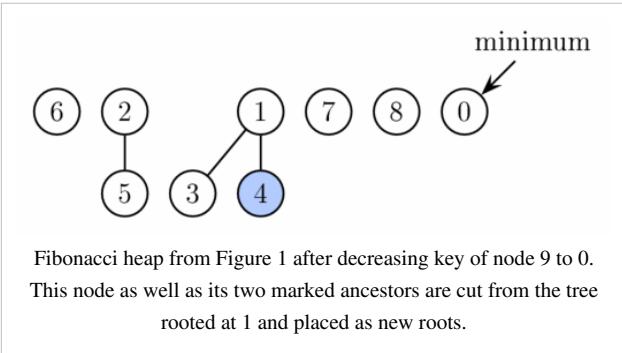


Fibonacci heap from Figure 1 after extract minimum is completed.
First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to n roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots u and v have the same degree, we make one of them a child of the other so that the one with smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length $O(\log n)$ in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and the array is updated. The actual running time is $O(\log n + m)$ where m is the number of roots at the beginning of the second phase. At the end we will have at most $O(\log n)$ roots (because each has a different degree). Therefore the potential decreases by at least $m \cdot O(\log n)$ and the amortized running time is $O(\log n)$.

In the third phase we check each of the remaining roots and find the minimum. This takes $O(\log n)$ time and the potential does not change. The overall amortized running time of extract minimum is therefore $O(\log n)$.

Operation **decrease key** will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. In the process we create some number, say k , of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore the potential decreases by at least $k - 2$. The actual time to perform the cutting was $O(k)$, therefore the amortized running time is constant.



Finally, operation **delete** can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap. Then we call extract minimum to remove it. The amortized running time of this operation is $O(\log n)$.

Proof of degree bounds

The amortized performance of a Fibonacci heap depends on the degree (number of children) of any tree root being $O(\log n)$, where n is the size of the heap. Here we show that the size of the (sub)tree rooted at any node x of degree d in the heap must have size at least F_{d+2} , where F_k is the k th Fibonacci number. The degree bound follows from this and the fact (easily proved by induction) that $F_{d+2} \geq \varphi^d$ for all integers $d \geq 0$, where $\varphi = (1 + \sqrt{5})/2 \doteq 1.62$. (We then have $n \geq F_{d+2} \geq \varphi^d$, and taking the log to base φ of both sides gives $d \leq \log_\varphi n$ as required.)

Consider any node x somewhere in the heap (x need not be the root of one of the main trees). Define **size**(x) to be the size of the tree rooted at x (the number of descendants of x , including x itself). We prove by induction on the height of x (the length of a longest simple path from x to a descendant leaf), that **size**(x) $\geq F_{d+2}$, where d is the degree of x .

Base case: If x has height 0, then $d = 0$, and **size**(x) = 1 = F_2 .

Inductive case: Suppose x has positive height and degree $d > 0$. Let y_1, y_2, \dots, y_d be the children of x , indexed in order of the times they were most recently made children of x (y_1 being the earliest and y_d the latest), and let c_1, c_2, \dots, c_d be their respective degrees. We **claim** that $c_i \geq i-2$ for each i with $2 \leq i \leq d$: Just before y_i was made a child x , y_1, \dots, y_{i-1} were already children of x , and so x had degree at least $i-1$ at that time. Since trees are combined only when the degrees of their roots are equal, it must have been that y_i also had degree at least $i-1$ at the time it became a child of x . From that time to the present, y_i can only have lost at most one child (as guaranteed by the marking process), and so its current degree c_i is at least $i-2$. This proves the **claim**.

Since the heights of all the y_i are strictly less than that of x , we can apply the inductive hypothesis to them to get **size**(y_i) $\geq F_{c_i+2} \geq F_{(i-2)+2} = F_i$. The nodes x and y_1 each contribute at least 1 to **size**(x), and so we have

$$\text{size}(x) \geq 2 + \sum_{i=2}^d \text{size}(y_i) \geq 2 + \sum_{i=2}^d F_i = 1 + \sum_{i=0}^d F_i.$$

A routine induction proves that $1 + \sum_{i=0}^d F_i = F_{d+2}$ for any $d \geq 0$, which gives the desired lower bound on **size**(x).

Worst case

Although the total running time of a sequence of operations starting with an empty structure is bounded by the bounds given above, some (very few) operations in the sequence can take very long to complete (in particular delete and delete minimum have linear running time in the worst case). For this reason Fibonacci heaps and other amortized data structures may not be appropriate for real-time systems.

Summary of running times

	Linked List	Balanced binary Tree	(Min-)Heap	Fibonacci Heap	Brodal Queue [1]
insert	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
accessmin	$O(n)$	$O(\log n)$ or $O(1)$ (**)	$O(1)$	$O(1)$	$O(1)$
deletemin	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$	$O(\log n)$
decreasekey	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)^*$	$O(1)$
delete	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$	$O(\log n)$
merge	$O(1)$	$O(m \log(n+m))$	$O(m \log(n+m))$	$O(1)$	$O(1)$

(*)Amortized time

(**)With trivial modification to the conventional Binary tree

References

- [1] (<http://citeseeerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.8133>)
- Fredman M. L. & Tarjan R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. (<http://doi.acm.org/10.1145/28869.28874>) *Journal of the ACM* 34(3), 596-615.
- 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.
- Brodal, G. S. 1996. Worst-case efficient priority queues. (<http://portal.acm.org/citation.cfm?id=313883>) In Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (Atlanta, Georgia, United States, January 28 - 30, 1996). Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, 52-58.

External links

- Java applet simulation of a Fibonacci heap (<http://www.cs.yorku.ca/~aaw/Jason/FibonacciHeapAnimation.html>)
- C implementation of Fibonacci heap (http://resnet.uoregon.edu/~gurney_j/jmpc/fib.html)
- Pseudocode of the Fibonacci heap algorithm (<http://www.cs.princeton.edu/~wayne/cs423/fibonacci/FibonacciHeapAlgorithm.html>)

2-3 heap

In computer science, a **2-3 heap** is a data structure, a variation on the heap, designed by Tadao Takaoka in 1999. The structure is similar to the Fibonacci heap, and borrows from the 2-3 tree.

Time costs for some common heap operations:

- *delete-min* takes $O(\log(n))$ amortized time
- *decrease-key* takes constant amortized time
- *insertion* takes constant amortized time.

References

- Tadao Takaoka. *Theory of 2-3 Heaps*^[1], Cocoon (1999).

References

[1] <http://www.cosc.canterbury.ac.nz/~tad/2-3heaps.pdf>

Pairing heap

Pairing heaps are a type of heap data structure with relatively simple implementation and excellent practical amortized performance. However, it has proven very difficult to determine the precise asymptotic running time of pairing heaps.

Pairing heaps are heap ordered multiway trees. Describing the various heap operations is relatively simple (in the following we assume a min-heap):

- *find-min*: simply return the top element of the heap.
- *merge*: compare the two root elements, the smaller remains the root of the result, the larger element and its subtree is appended as a child of this root.
- *insert*: create a new heap for the inserted element and *merge* into the original heap.
- *decrease-key* (optional): remove the subtree rooted at the key to be decreased then *merge* it with the heap.
- *delete-min*: remove the root and *merge* its subtrees. Various strategies are employed.

The amortized time per *delete-min* is $O(\log n)$.^[1] The operations *find-min*, *merge*, and *insert* take $O(1)$ amortized time^[2] and *decrease-key* takes $2^{O(\sqrt{\log \log n})}$ amortized time.^[3] Fredman proved that the amortized time per *decrease-key* is at least $\Omega(\log \log n)$.^[4] That is, they are less efficient than Fibonacci heaps, which perform *decrease-key* in $O(1)$ amortized time.

Stasko and Vitter^[5] and Moret and Shapiro^[6] conducted experiments on pairing heaps and other heap data structures. They concluded that the pairing heap is as fast as, and often faster than, other efficient data structures like the binary heaps.

Implementation

A pairing heap is either an empty heap, or a pair consisting of a root element and a possibly empty list of pairing heaps. The heap ordering property requires that all the root elements of the subheaps in the list are not smaller than the root element of the heap. The following description assumes a purely functional heap that does not support the *decrease-key* operation.

```
type PairingHeap[Elem] = Empty Heap(elem: Elem, subheaps:  
List[PairingHeap[Elem]])
```

Operations

find-min

The function *find-min* simply returns the root element of the heap:

```
function find-min(heap)  
  if heap == Empty  
    error  
  else  
    return heap.elem
```

merge

Merging with an empty heap returns the other heap, otherwise a new heap is returned that has the minimum of the two root elements as its root element and just adds the heap with the larger root to the list of subheaps:

```
function merge(heap1, heap2)  
  if heap1 == Empty  
    return heap2  
  elsif heap2 == Empty  
    return heap1  
  elseif heap1.elem < heap2.elem  
    return Heap(heap1.elem, heap2 :: heap1.subheaps)  
  else  
    return Heap(heap2.elem, heap1 :: heap2.subheaps)
```

insert

The easiest way to insert an element into a heap is to merge the heap with a new heap containing just this element and an empty list of subheaps:

```
function insert(elem, heap)  
  return merge(Heap(elem, []), heap)
```

delete-min

The only non-trivial fundamental operation is the deletion of the minimum element from the heap. The standard strategy first merges the subheaps in pairs (this is the step that gave this datastructure its name) from left to right and then merges the resulting list of heaps from right to left:

```
function delete-min(heap)  
  if heap == Empty  
    error
```

```

elsif length(heap.subheaps) == 0
    return Empty
elsif length(heap.subheaps) == 1
    return heap.subheaps[0]
else
    return merge-pairs(heap.subheaps)

```

This uses the auxilliary function *merge-pairs*:

```

function merge-pairs(l)
    if length(l) == 0
        return Empty
    elsif length(l) == 1
        return l[0]
    else
        return merge(merge(l[0], l[1]), merge-pairs(l[2.. ]))

```

That this does indeed implement the described two-pass left-to-right then right-to-left merging strategy can be seen from this reduction:

```

merge-pairs([H1, H2, H3, H4, H5, H6, H7])
=> merge(merge(H1, H2), merge-pairs([H3, H4, H5, H6, H7]))
    # merge H1 and H2 to H12, then the rest of the list
=> merge(H12, merge(merge(H3, H4), merge-pairs([H5, H6, H7])))
    # merge H3 and H4 to H34, then the rest of the list
=> merge(H12, merge(H34, merge(merge(H5, H6), merge-pairs([H7]))))
    # merge H5 and H6 to H56, then the rest of the list
=> merge(H12, merge(H34, merge(H56, H7)))
    # switch direction, merge the last two resulting heaps, giving
H567
=> merge(H12, merge(H34, H567))
    # merge the last two resulting heaps, giving H34567
=> merge(H12, H34567)
    # finally, merge the first merged pair with the result of merging
the rest
=> H1234567

```

External links

- Sartaj Sahni's pairing heaps page ^[7]

References

- [1] Fredman, Michael L.; Sedgewick, Robert; Sleator, Daniel D.; Tarjan, Robert E. (1986), "The pairing heap: a new form of self-adjusting heap", *Algorithmica* **1** (1): 111–129, doi:10.1007/BF01840439.
- [2] 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.
- [3] Pettie, Seth (2005), "Towards a final analysis of pairing heaps", *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science*, pp. 174–183, doi:10.1109/SFCS.2005.75.
- [4] Fredman, Michael L. (1999), "On the efficiency of pairing heaps and related data structures", *Journal of the ACM* **46** (4): 473–501, doi:10.1145/320211.320214.

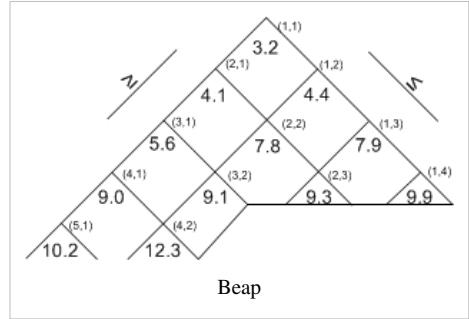
- [5] Stasko, John T.; Vitter, Jeffrey S. (1987), "Pairing heaps: experiments and analysis", *Communications of the ACM* **30** (3): 234–249, doi:10.1145/214748.214759.
- [6] Moret, Bernard M. E.; Shapiro, Henry D. (1991), "An empirical analysis of algorithms for constructing a minimum spanning tree", *Proc. 2nd Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, **519**, Springer-Verlag, pp. 400–411, doi:10.1007/BFb0028279.
- [7] <http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c13/pairing.htm>

Beap

Beap, short for **bi-parental heap**, introduced by Ian Munro and Hendra Suwanda. In this data structure a node usually has two parents (unless it is the first or last on a level) and two children (unless it is on the last level). What separates the beap from Williams' heap is that beap allows sublinear search.

Performance

The height of the structure is approximately \sqrt{n} . Also, assuming the last level is full, the number of elements on that level is also \sqrt{n} . In fact, because of these properties all basic operations (insert, remove, find) run in $O(\sqrt{n})$ time on average. Find operations in the heap can be $O(n)$ in the worst case. Removal and insertion of new elements involves propagation of elements up or down (much like in a heap) in order to restore the beap invariant. An additional perk is that beap provides constant time access to the smallest element and $O(\sqrt{n})$ time for the maximum element.



Advantages

The main advantage of the beap is that it can be fully implemented in-place - only data nodes need to be present (no pointers or other extra information is required). However, this structure should be used with care since it is not $O(\log n)$ nor does it perform much better than a vector for small values of n.

References

J. Ian Munro and Hendra Suwanda. "Implicit data structures for fast search and update". *Journal of Computer and System Sciences*, 21(2):236250, 1980.

J.W.J Williams in Algorithms 232, "Heapsort", *Comm. ACM* 7 (June 1964), 347-348

Leftist tree

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.

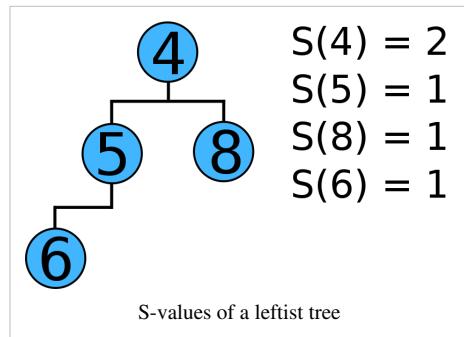
Leftist trees are advantageous because of their ability to merge quickly, compared to binary heaps which take $\Theta(n)$. In almost all cases, skew heaps have better performance.

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]. In the diagram, the extended representation (not shown) fills in the tree to make it a complete binary tree (adding five leaves), 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

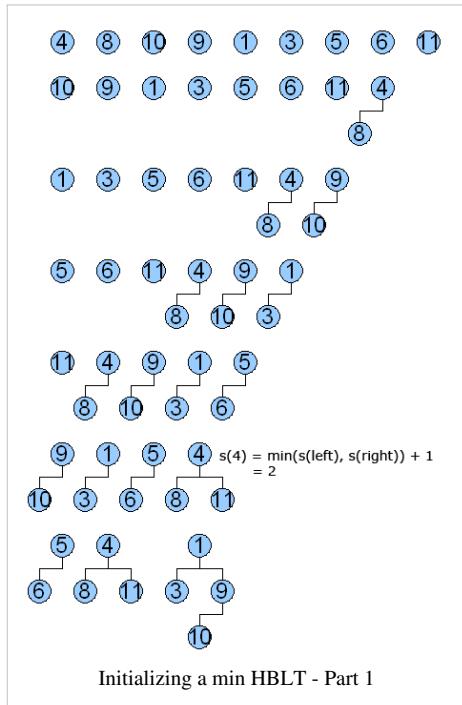
```
public Node merge(Node x, Node y) {
    if(x == null)
        return y;
    if(y == null)
        return x;

    // 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);

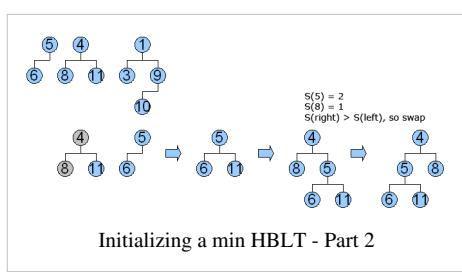
    if(x.leftChild == null) {
        // left child doesn't exist, so move right child to the left side
        x.leftChild = x.rightChild;
        x.rightChild = null;
        x.s = 1;
    } else {
        // left child does exist, so compare s-values
        if(x.leftChild.s < x.rightChild.s) {
            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

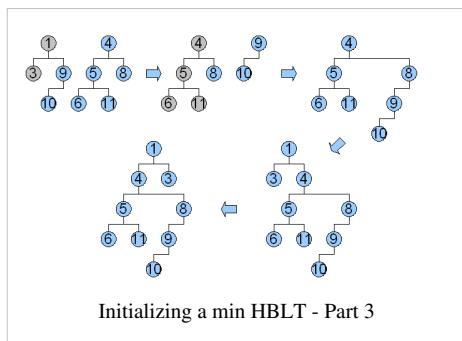


Initializing a height biased leftist tree is primarily done in one of two ways. The first is to merge each node one at a time into one HBLT. This process is inefficient and takes $O(n \log n)$ time. The other approach is to use a queue to store each node and resulting tree. The first two items in the 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] at Dr. Sartaj Sahni's website (Department Chair in CISE at University of Florida)
- Java implementation of leftist tree ^[3]

References

- [1] <http://mathworld.wolfram.com/ExtendedBinaryTree.html>
- [2] <http://www.cise.ufl.edu/~sahni/cop5536/slides/lec114.pdf>
- [3] <http://niketanpansare.com/MinLeftistTree.aspx>

Skew heap

A **skew heap** is a heap data structure implemented as a binary tree. Skew heaps are advantageous because of their ability to merge more quickly than binary heaps. In contrast with binary heaps, there are no structural constraints, so there is no guarantee that the height of the tree is logarithmic. Only two conditions must be satisfied:

- The general heap order must be enforced
- Every operation (add, remove_min, merge) on two skew heaps must be done using a special *skew heap merge*.

A skew heap is a self-adjusting form of a leftist heap which attempts to maintain balance by unconditionally swapping all nodes in the merge path when merging two heaps. (The merge operation is also used when adding and removing values.)

With no structural constraints, it may seem that a skew heap would be horribly inefficient. However, amortized complexity analysis can be used to demonstrate that all operations on a skew heap can be done in $O(\log n)$.[1]

Definition

Skew heaps may be described with the following recursive definition:

- A heap with only one element is a skew heap.
- The result of *skew merging* two skew heaps sh_1 and sh_2 is also a skew heap.

Operations

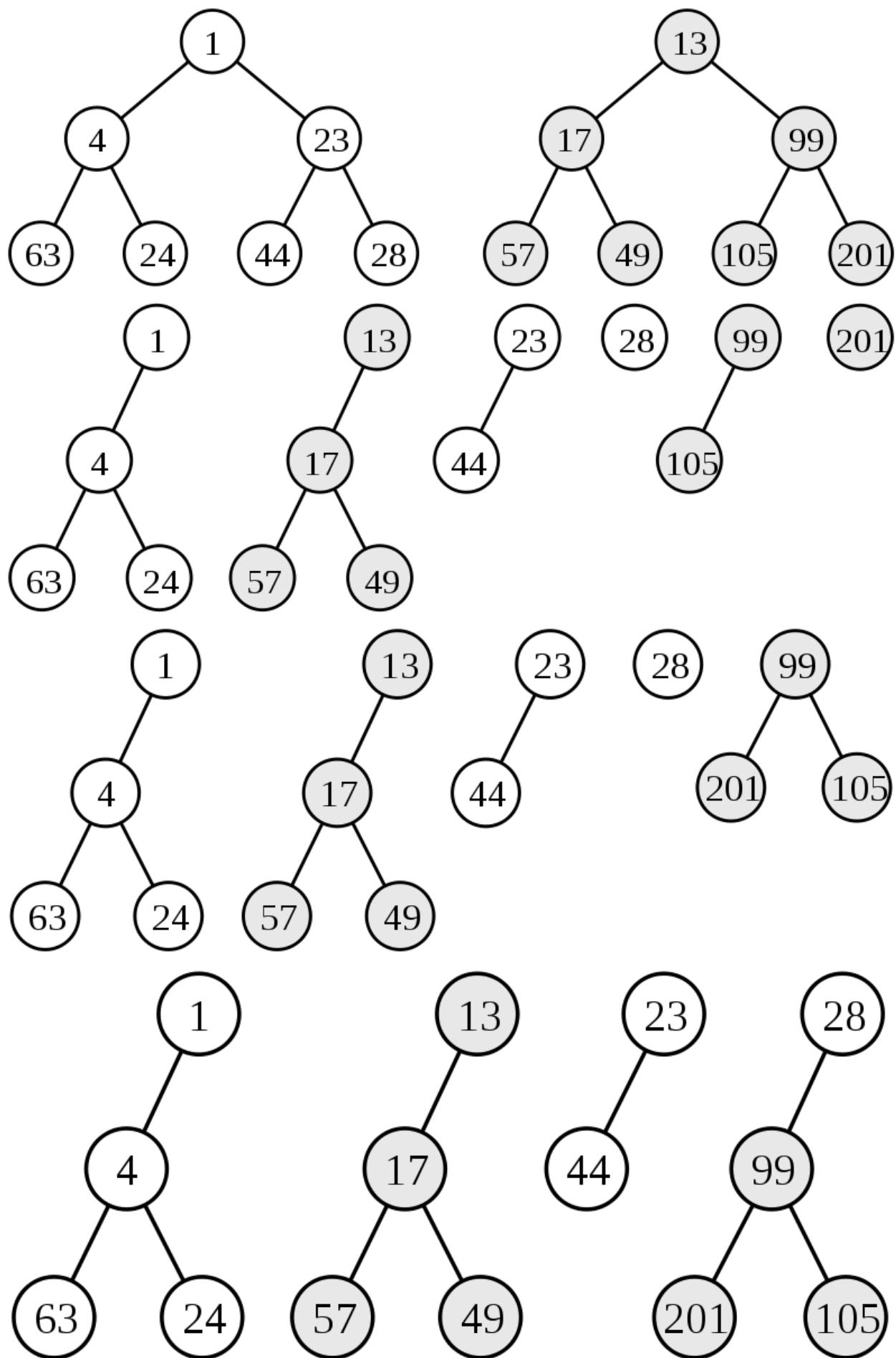
Merging Two Heaps

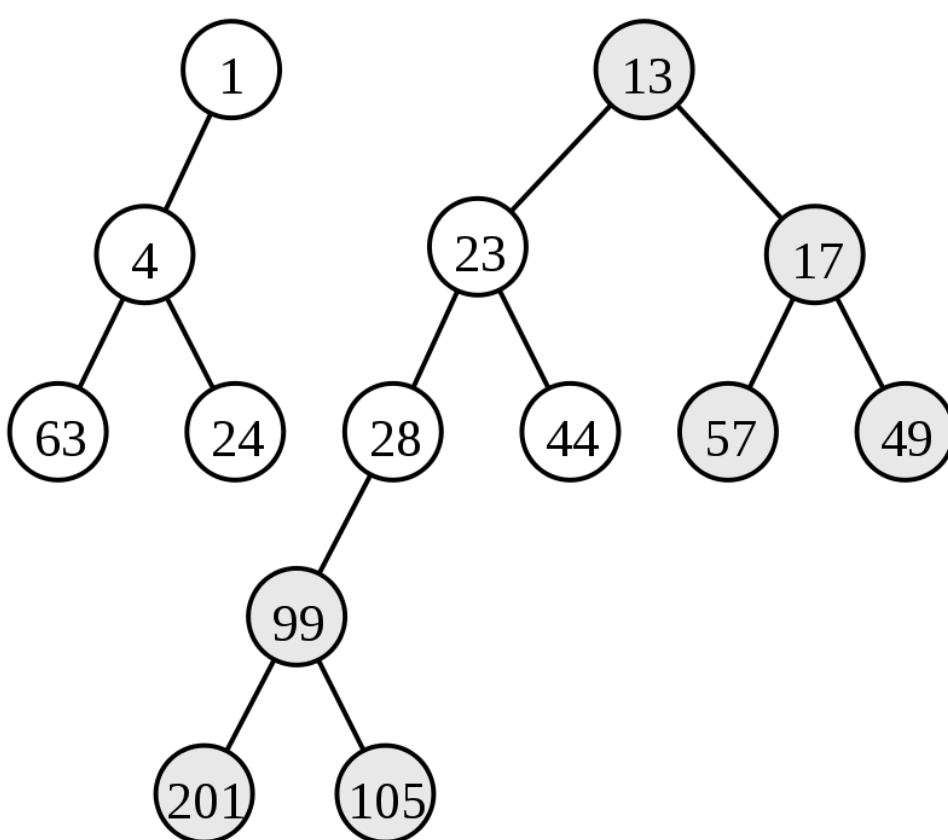
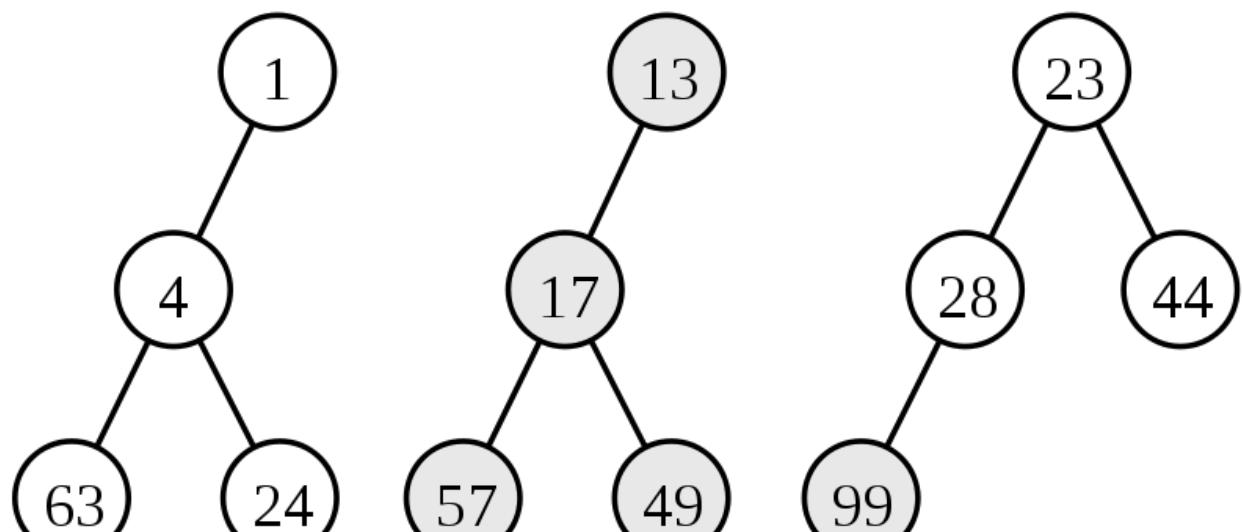
When two skew heaps are to be merged together, we can use the same process as the merge of two leftist heaps:

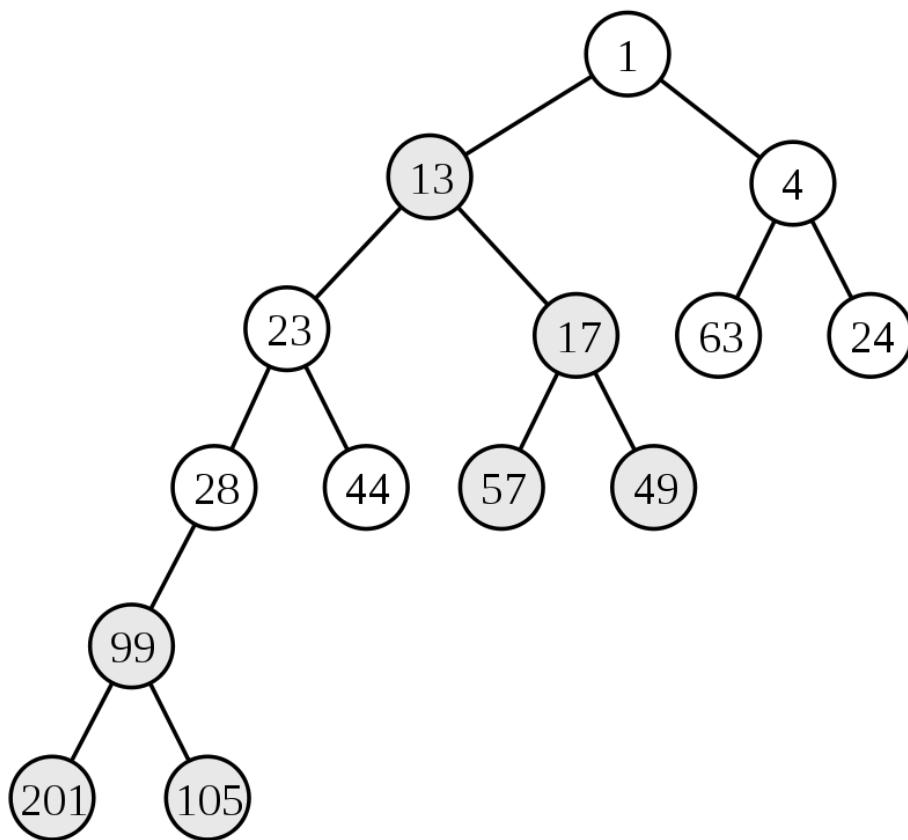
- Compare roots of two heaps
- Recursively merge the heap that has the larger root with the right subtree of the other heap.
- Make the resulting merge the right subtree of the heap that has smaller root.
- Swap the children of the new heap

Alternatively, there is a non-recursive approach which tends to be a little clearer, but does require some sorting at the outset.

- Split each heap into subtrees by cutting every rightmost path. (From the root node, sever the right node and make the right child its own subtree.) This will result in a set of trees in which the root either only has a left child or no children at all.
- Sort the subtrees in ascending order based on the value of the root node of each subtree.
- While there are still multiple subtrees, iteratively recombine the last two (from right to left).
 - If the root of the second-to-last subtree has a left child, swap it to be the right child.
 - Link the root of the last subtree as the left child of the second-to-last subtree.







Adding Values

Adding a value to a skew heap is like merging a tree with one node together with the original tree. The root node is linked as the left child of the new value, which itself becomes the new root.

Removing Values

Removing the first value in a heap can be accomplished by removing the root and (a) merging the child subtrees if the root has two children, (b) replacing the root with the non-nil child, or (c) ending with an empty heap.

Implementation

In many functional languages, skew heaps become extremely simple to implement. Here is a complete sample implementation in Haskell.

```

data SkewHeap a = Empty | SkewNode a (SkewHeap a) (SkewHeap a)

singleton :: a -> SkewHeap a
singleton x = SkewNode x Empty Empty

union :: Ord a => SkewHeap a -> SkewHeap a -> SkewHeap a
t1 `union` t2 = case (t1, t2) of
  (Empty, _)           -> t2
  (_, Empty)          -> t1
  (SkewNode x1 l1 r1, SkewNode x2 l2 r2)
    | x1 <= x2      -> SkewNode x1 (t2 `union` r1) l1
    | otherwise       -> SkewNode x2 (t1 `union` r2) l2
  
```

```

insert :: Ord a => a -> SkewHeap a -> SkewHeap a
insert x heap = singleton x `union` heap

extractMin :: Ord a => SkewHeap a -> Maybe (a, SkewHeap a)
extractMin Empty = Nothing
extractMin (SkewNode x l r) = Just (x, l `union` r)

```

References

[1] <http://www.cse.yorku.ca/~andy/courses/4101/lecture-notes/LN5.pdf>

Original papers:

- D. Sleator and R. Tarjan, "Self-Adjusting Heaps," SIAM Journal on Computing, 15(1), 1986, pp. 52-69. — (<http://www.cs.cmu.edu/~sleator/papers/Adjusting-Heaps.htm>)

Analysis and Proofs:

- CSE 4101 lecture notes, York University (<http://www.cse.yorku.ca/~andy/courses/4101/lecture-notes/LN5.pdf>)

External links

- Animations comparing leftist heaps and skew heaps, York University (<http://www.cse.yorku.ca/~aaw/Pourhashemi/>)
 - Java applet for simulating heaps, Kansas State University (<http://people.cis.ksu.edu/~rhowell/viewer/heapviewer.html>)
-

Soft heap

In computer science, the **soft heap**, designed by Bernard Chazelle in 2000, is a variant on the simple heap data structure. By carefully "corrupting" (increasing) the keys of at most a certain fixed percentage of values in the heap, it is able to achieve amortized constant-time bounds for all five of its operations:

- **create(S):** Create a new soft heap
- **insert(S, x):** Insert an element into a soft heap
- **meld(S, S'):** Combine the contents of two soft heaps into one, destroying both
- **delete(S, x):** Delete an element from a soft heap
- **findmin(S):** Get the element with minimum key in the soft heap

The term "corruption" here is the result of what Chazelle called "carpooling" in a soft heap. Each node in the soft heap contains a linked-list of keys and one common key. The common key is an upper bound on the values of the keys in the linked-list. Once a key is added to the linked-list, it is considered corrupted because its value is never again relevant in any of the soft heap operations: only the common keys are compared. It is unpredictable which keys will be corrupted in this manner; it is only known that at most a fixed percentage will be corrupted. This is what makes soft heaps "soft"; you can't be sure whether or not any particular value you put into it will be corrupted. The purpose of these corruptions is effectively to lower the information entropy of the data, enabling the data structure to break through information-theoretic barriers regarding heaps.

Other heaps such as fibonacci heaps achieve most of these bounds without any corruption, but cannot provide a constant-time bound on the critical *delete* operation. The percentage of values which are corrupted can be chosen freely, but the lower this is set, the more time insertions require ($O(\log 1/\epsilon)$ for an error rate of ϵ).

Applications

Surprisingly, soft heaps are useful in the design of deterministic algorithms, despite their unpredictable nature. They were the key in creating the best-known algorithm for finding a minimum spanning tree to date. They can also be used to easily build an optimal selection algorithm, as well as *near-sorting* algorithms, which are algorithms that place every element near its final position, a situation in which insertion sort is fast.

One of the simplest examples is the selection algorithm. Say we want to find the k th largest of a group of n numbers. First, we choose an error rate of $1/3$; that is, at most 33% of the keys we insert will be corrupted. Now, we insert all n elements into the heap — at this point, at most $n/3$ keys are corrupted. Next, we delete the minimum element from the heap about $n/3$ times. Because this is decreasing the size of the heap, it cannot increase the number of corrupted elements. Thus there are still at most $n/3$ keys that are corrupted.

Now at least $2n/3 - n/3 = n/3$ of the remaining keys are not corrupted, so each must be larger than every element we removed. Let L be the element that we have removed with the largest (actual) value, which is not necessarily the last element that we removed (because the last element we removed could have had its key corrupted, or increased, to a value larger than another element that we have already removed). L is larger than all the other $n/3$ elements that we removed and smaller than the remaining $n/3$ uncorrupted elements in the soft heap. Therefore, L divides the elements somewhere between $33\%/66\%$ and $66\%/33\%$. We then partition the set about L using the *partition* algorithm from quicksort and apply the same algorithm again to either the set of numbers less than L or the set of numbers greater than L , neither of which can exceed $2n/3$ elements. Since each insertion and deletion requires $O(1)$ amortized time, the total deterministic time is $T(n) = T(2n/3) + O(n)$. Using case 3 of the master theorem (with $\varepsilon=1$ and $c=2/3$), we know what that $T(n) = \Theta(n)$.

The final algorithm looks like this:

```
function softHeapSelect(a[1..n], k)
    if k = 1 then return minimum(a[1..n])
    create(S)
    for i from 1 to n
        insert(S, a[i])
    for i from 1 to n/3
        x := findmin(S)
        delete(S, x)
    xIndex := partition(a, x) // Returns new index of pivot x
    if k < xIndex
        softHeapSelect(a[1..xIndex-1], k)
    else
        softHeapSelect(a[xIndex..n], k-xIndex+1)
```

References

- Chazelle, B. 2000. The soft heap: an approximate priority queue with optimal error rate. ^[1] *J. ACM* 47, 6 (Nov. 2000), 1012-1027.
- Kaplan, H. and Zwick, U. 2009. A simpler implementation and analysis of Chazelle's soft heaps. ^[2] In *Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms* (New York, New York, January 04 - 06, 2009). Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, 477-485.

References

- [1] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.9705>
- [2] http://www.siam.org/proceedings/soda/2009/SODA09_053_kaplanh.pdf

Ternary heap

A ternary heap is a data structure in computer science. It is part of the heap family. It inherits from ternary tree and the heap. Often there are two kinds of ternary heaps: the **min ternary heap** and the **max ternary heap**.

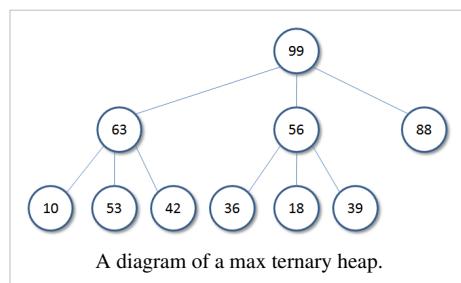
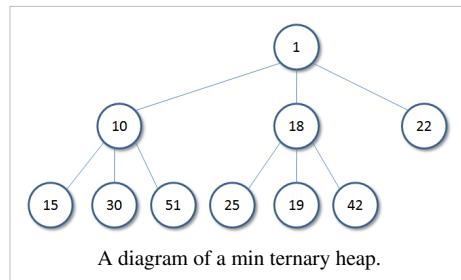
- **Min Ternary Heap**

- As illustrated, a min ternary heap has the smallest element as its root. Parent node has to be smaller than or equal to its children.

- **Max Ternary Heap**

- A max ternary heap has the biggest element as its root. Each node has to be greater than or equal to its children.

In both cases, a ternary heap has to satisfy the heap property, which is every level of tree has to be filled, or if the last level is not filled, the leaves are distributed from left to right. These rules have to be strictly followed when implementing the code to build a ternary heap.



Heapify

The process of building a heap is called **Heapify**. A way to implement this data structure is to use a collection, e.g.: Vectors in Java.

Each element in the collection is a node of the ternary heap, where the index is in the order of from top to bottom, from left to right.

The example max ternary heap in a Vector will be like this:

```
[99, 63, 56, 88, 10, 53, 42, 36, 18, 39]
```

For each node, assuming that its index is i, and that indexing starts with node 0

- **parent's index:** $\text{floor}((i - 1)/3)$
- **left child's:** $3 * i + 1$
- **middle child's:** $3 * i + 2$
- **right child's:** $3 * i + 3$

The following C-like pseudocode shows how to heapify a vector to a **max** ternary heap.^[1]

```

Max_Heapify(A, i) {
    largest = -1;
    l = left(i);
    m = middle(i);
    r = right(i);

    if (l < size(A) && (A[l] > A[i])) {
        if (m < size(A) && (A[m] > A[l])) {
            if (r < size(A) && (A[r] > A[m])) {
                swap(A[l], A[r]);
                Max_Heapify(A, r);
            } else {
                swap(A[l], A[m]);
                Max_Heapify(A, m);
            }
        } else {
            swap(A[l], A[m]);
            Max_Heapify(A, m);
        }
    }
}
  
```

```
    largest = l;
} else {
    largest = i;
}

if (m < size(A) && (A[m] > A[largest])) {
    largest = m;
}

if (r < size(A) && (A[r] > A[largest])) {
    largest = r;
}

if (largest != i) {
    swap(A[i], A[largest]);
    Max_Heapify(A, largest);
}
}
```

See also

- Binary Heap
- Ternary Tree

[1] Cormen, Thomas; Charles E. Leiserson. *Introduction to Algorithms*. pp. 128–130.

D-ary heap

The **d-ary heap** or **d-heap** is a generalization of the binary heap data structure whose non-leaf nodes have d children, instead of 2. Thus, a binary heap is a 2-heap.

According to Jensen et al.^[1], d-ary heaps were invented by Johnson in 1975^[2].

Because the d-heap is shallower than a binary heap, it has faster running times, $O(\log_d N)$, for *insert* operations. However, for d values greater than two, the *delete-min* operation is more expensive, because although the tree is shallower, the minimum of d children must be found, which takes $d - 1$ comparisons using a standard algorithm. This raises the time for this operation to $O(d \log_d N)$. If d is a constant, both running times are, of course, $O(\log N)$.

Although array implementations of d-ary heaps can be built, the multiplications and divisions needed to traverse the nodes of the heap can no longer be handled by simple bit shift operations unless d is a power of 2. This can seriously increase the running time. d -heaps are interesting in theory, because there are many algorithms where the number of insertions is much greater than the number of deleteMins (and thus a theoretical speedup is possible). They are also of interest when the priority queue is too large to fit entirely in main memory. In this case, a d -heap can be advantageous in much the same way as B-Trees. Finally, there is evidence suggesting that 4-heaps may outperform binary heaps in practice.

The most glaring weakness of the heap implementation, aside from the inability to perform finds, is that combining two heaps into one is a hard operation. This extra operation is known as a merge. There are quite a few ways of implementing heaps so that the running time of a merge is $O(\log N)$.

References

- [1] Jensen; Katajainen; Vitale (2004) (PDF). *An extended truth about heaps* (<http://www.cphstl.dk/Report/In-place-multiway-heaps/experimental-study.pdf>). .
- [2] Johnson, D.B. (1975). "Priority queues with update and finding minimum spanning trees". *Information Processing Letters* **4**: 53–57. doi:10.1016/0020-0190(75)90001-0.

Tries

Trie

In computer science, a **trie**, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.

The term trie comes from "retrieval." Following the etymology, the inventor, Edward Fredkin, pronounces it English pronunciation: /'tri:/ "tree".^[1] ^[2] However, it is pronounced English pronunciation: /'traɪ/ "try" by other authors.^[3] ^[2]

In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

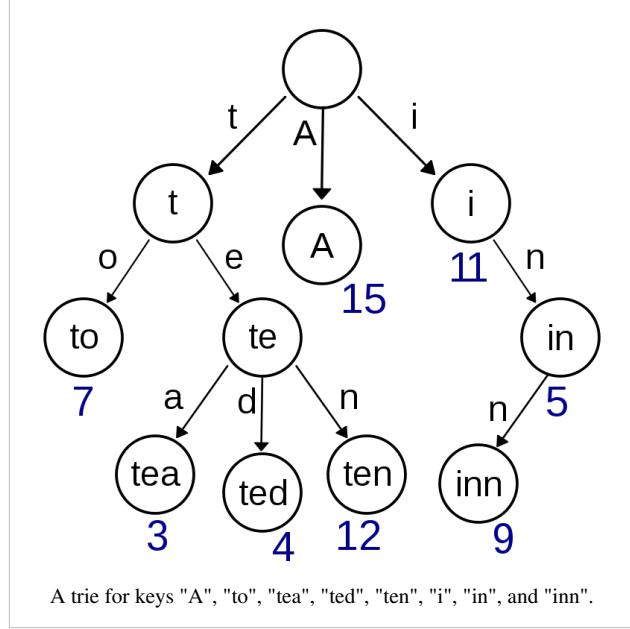
It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the trie works.)

Though it is most common, tries need not be keyed by character strings. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes.

Advantages relative to binary search tree

The following are the main advantages of tries over binary search trees (BSTs):

- Looking up keys is faster. Looking up a key of length m takes worst case $O(m)$ time. A BST performs $O(\log(n))$ comparisons of keys, where n is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys if the tree is balanced. Hence in the worst case, a BST takes $O(m \log n)$ time. Moreover, in the worst case $\log(n)$ will approach m . Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines.
- Tries can require less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys with common initial subsequences.
- Tries facilitate longest-prefix matching, helping to find the key sharing the longest possible prefix of characters all unique.



Applications

As replacement of other data structures

As mentioned, a trie has a number of advantages over binary search trees.^[4] A trie can also be used to replace a hash table, over which it has the following advantages:

- Looking up data in a trie is faster in the worst case, $O(m)$ time, compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is $O(N)$ time, but far more typically is $O(1)$, with $O(m)$ time spent evaluating the hash.
- There are no collisions of different keys in a trie.
- Buckets in a trie which are analogous to hash table buckets that store key collisions are only necessary if a single key is associated with more than one value.
- There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
- A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

- Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random access time is high compared to main memory.^[5]
- It is not easy to represent all keys as strings, such as floating point numbers - a straightforward encoding using the bitstring of their encoding leads to long chains and prefixes that are not particularly meaningful.

Dictionary representation

A common application of a trie is storing a dictionary, such as one found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries; however, if storing dictionary words is all that is required (i.e. storage of information auxiliary to each word is not required), a minimal acyclic deterministic finite automaton would use less space than a trie.

Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking and hyphenation^[2] software.

Algorithms

We can describe trie lookup (and membership) easily. Given a recursive trie type, storing an optional value at each node, and a list of children tries, indexed by the next character, (here, represented as a Haskell data type):

```
data Trie a = Trie { value :: Maybe a , children :: [(Char,Trie a)] }
```

We can lookup a value in the trie as follows:

```
find :: String -> Trie a -> Maybe a
find [] t = value t
find (k:ks) t = case lookup k (children t) of Nothing -> Nothing
Just t' -> find ks t'
```

In an imperative style, and assuming an appropriate data type in place, we can describe the same algorithm in Python (here, specifically for testing membership). Note that children is map of a node's children; and we say that a "terminal" node is one which contains a valid word.

```
def is_key_in_trie(node, key):
    if not key:                                     # base case: key is an empty
        string                                         # we have a match if the node
    return node.is_terminal()                         # is terminal
```

```

c = key[0]                                # first character in key
if c not in node.children:                 # no child node?
    return False                           # key is not empty, so there
is no match
child = node.children[c]                   # get child node
tail = key[1:]                            # the rest of the key after
first character
return is_key_in_trie(child, tail)         # recurse

```

Sorting

Lexicographic sorting of a set of keys can be accomplished with a simple trie-based algorithm as follows:

- Insert all keys in a trie.
- Output all keys in the trie by means of pre-order traversal, which results in output that is in lexicographically increasing order. Pre-order traversal is a kind of depth-first traversal. In-order traversal is another kind of depth-first traversal that is more appropriate for outputting the values that are in a binary search tree rather than a trie.

This algorithm is a form of radix sort.

A trie forms the fundamental data structure of Burstsrt; currently (2007) the fastest known, memory/cache-based, string sorting algorithm.^[6]

A parallel algorithm for sorting N keys based on tries is O(1) if there are N processors and the length of the keys have a constant upper bound. There is the potential that the keys might collide by having common prefixes or by being identical to one another, reducing or eliminating the speed advantage of having multiple processors operating in parallel.

Full text search

A special kind of trie, called a suffix tree, can be used to index all suffixes in a text in order to carry out fast full text searches.

Compressing tries

When the trie is mostly static, i.e. all insertions or deletions of keys from a prefilled trie are disabled and only lookups are needed, and when the trie nodes are not keyed by node specific data (or if the node's data is common) it is possible to compress the trie representation by merging the common branches^[7]. This application is typically used for compressing lookup tables when the total set of stored keys is very sparse within their representation space.

For example it may be used to represent sparse bitsets (i.e. subsets of a much fixed enumerable larger set) using a trie keyed by the bit element position within the full set, with the key created from the string of bits needed to encode the integral position of each element. The trie will then have a very degenerate form with many missing branches, and compression becomes possible by storing the leaf nodes (set segments with fixed length) and combining them after detecting the repetition of common patterns or by filling the unused gaps.

Such compression is also typically used, in the implementation of the various fast lookup tables needed to retrieve Unicode character properties (for example to represent case mapping tables, or lookup tables containing the combination of base and combining characters needed to support Unicode normalization). For such application, the representation is similar to transforming a very large unidimensional sparse table into a multidimensional matrix, and then using the coordinates in the hyper-matrix as the string key of an uncompressed trie. The compression will then consist into detecting and merging the common columns within the hyper-matrix to compress the last dimension in the key; each dimension of the hypermatrix stores the start position within a storage vector of the next dimension for

each coordinate value, and the resulting vector is itself compressible when it is also sparse, so each dimension (associated to a layer level in the trie) is compressed separately.

Some implementations do support such data compression within dynamic sparse tries and allow insertions and deletions in compressed tries, but generally this has a significant cost when compressed segments need to be split or merged, and some tradeoff has to be made between the smallest size of the compressed trie and the speed of updates, by limiting the range of global lookups for comparing the common branches in the sparse trie.

The result of such compression may look similar to trying to transform the trie into a directed acyclic graph (DAG), because the reverse transform from a DAG to a trie is obvious and always possible, however it is constrained by the form of the key chosen to index the nodes.

Another compression approach is to "unravel" the data structure into a single byte array [8]. This approach eliminates the need for node pointers which reduces the memory requirements substantially and makes memory mapping possible which allows the virtual memory manager to load the data into memory very efficiently.

Another compression approach is to "pack" the trie [2]. Liang describes a space-efficient implementation of a sparse packed trie applied to hyphenation, in which the descendants of each node may be interleaved in memory.

See also

- Radix tree
- Deterministic finite automata
- Prefix Hash Tree
- Directed acyclic word graph (aka DAWG)
- Judy array
- Burstsrt
- Ternary search tries
- Search algorithm
- Luleå algorithm
- Acyclic deterministic finite automata
- Extendable hashing
- Huffman coding
- Hash trie
- Hash array mapped trie

External links

- NIST's Dictionary of Algorithms and Data Structures: Trie [9]
- Tries [10] by Lloyd Allison
- Using Tries [11] Topcoder tutorial
- An Implementation of Double-Array Trie [12]
- de la Briandais Tree [13]
- Discussing a trie implementation in Lisp [14]
- ServerKit "parse trees" implement a form of Trie in C [15]

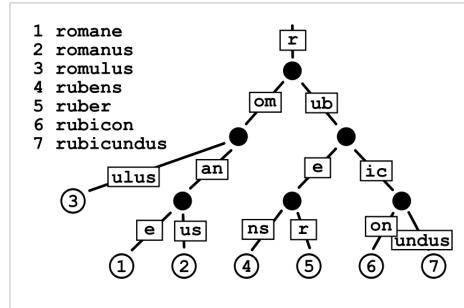
References

- [1] Paul E. Black. *Dictionary of Algorithms and Data Structures*, NIST. <http://www.nist.gov/dads/HTML/trie.html>
- [2] Liang, Frank (June 1983), *Word hy-phen-a-tion by com-pu-ter* (<http://www.tug.org/docs/liang/liang-thesis.pdf>), Frank M. Liang, Ph.D. thesis, Stanford University., , retrieved 2010-03-28
- [3] Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 6.3: Digital Searching, page 492.
- [4] Jon Bentley and Robert Sedgewick (1998). "Ternary Search Trees" (<http://www.ddj.com/windows/184410528>). *Dr. Dobbs Journal*. .
- [5] Edward Fredkin (1960). "Trie Memory". *Communications of the ACM* 3 (9): 490. doi:10.1145/367390.367400.
- [6] "Cache-Efficient String Sorting Using Copying" (<http://www.cs.mu.oz.au/~rsinha/papers/SinhaRingZobel-2006.pdf>) (PDF). . Retrieved 2008-11-15.
- [7] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, Richard E. Watson (2000). "Incremental Construction of Minimal Acyclic Finite-State Automata" (<http://www.mitpressjournals.org/doi/abs/10.1162/089120100561601>). *Computational Linguistics*. Association for Computational Linguistics. pp. Vol. 26, No. 1, Pages 3–16. doi:10.1162/089120100561601. Archived from the original (<http://www.pg.gda.pl/~jandac/daciuk98.ps.gz>) on 2006-03-13. . Retrieved 2009-05-28. "This paper presents a method for direct building of minimal acyclic finite states automaton which recognizes a given finite list of words in lexicographical order. Our approach is to construct a minimal automaton in a single phase by adding new strings one by one and minimizing the resulting automaton on-the-fly"

- [8] Ulrich Germann, Eric Joanic, Samuel Larkin (2009). "Tightly packed tries: how to fit large models into memory, and make them load fast, too" (<http://portal.acm.org/citation.cfm?id=1621952>). *ACL Workshops: Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*. Association for Computational Linguistics. pp. Pages 31–39.. "We present Tightly Packed Tries (TPTs), a compact implementation of read-only, compressed trie structures with fast on-demand paging and short load times. We demonstrate the benefits of TPTs for storing n-gram back-off language models and phrase tables for statistical machine translation. Encoded as TPTs, these databases require less space than flat text file representations of the same data compressed with the gzip utility. At the same time, they can be mapped into memory quickly and be searched directly in time linear in the length of the key, without the need to decompress the entire file. The overhead for local decompression during search is marginal."
- [9] <http://www.nist.gov/dads/HTML/trie.html>
- [10] <http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Trie/>
- [11] <http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=usingTries>
- [12] <http://linux.thai.net/~thep/datrie/datrie.html>
- [13] <http://tom.biodome.org/brianda.html>
- [14] http://groups.google.com/group/comp.lang.lisp/browse_thread/thread/01e485291d150938/9aacb626fa26c516
- [15] <http://serverkit.org/apiref-wip/node59.html>
- de la Brianda, R. (1959). "File Searching Using Variable Length Keys". *Proceedings of the Western Joint Computer Conference*: 295–298.

Radix tree

A **radix tree**, **Patricia trie/tree**, or **crit bit tree** is a specialized set data structure based on the trie that is used to store a set of strings. In contrast with a regular trie, the edges of a Patricia trie are labelled with sequences of characters rather than with single characters. These can be strings of characters, bit strings such as integers or IP addresses, or generally arbitrary sequences of objects in lexicographical order. Sometimes the names *radix tree* and *crit bit tree* are only applied to trees storing integers and *Patricia trie* is retained for more general inputs, but the structure works the same way in all cases.



Overview

The radix tree is easiest to understand as a space-optimized trie where each node with only one child is merged with its child. The result is that every internal node has at least two children. Unlike in regular tries, edges can be labeled with sequences of characters as well as single characters. This makes them much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.

It supports the following main operations, all of which are $O(k)$, where k is the maximum length of all strings in the set:

- **Lookup:** Determines if a string is in the set. This operation is identical to tries except that some edges consume multiple characters.
- **Insert:** Add a string to the tree. We search the tree until we can make no further progress. At this point we either add a new outgoing edge labeled with all remaining characters in the input string, or if there is already an outgoing edge sharing a prefix with the remaining input string, we split it into two edges (the first labeled with the common prefix) and proceed. This splitting step ensures that no node has more than two children.
- **Delete:** Delete a string from the tree. First, we delete the corresponding leaf. Then, if its parent only has one child remaining, we delete the parent and merge the two incident edges.
- **Find predecessor:** Locates the largest string less than a given string, by lexicographic order.
- **Find successor:** Locates the smallest string greater than a given string, by lexicographic order.

A common extension of radix trees uses two colors of nodes, 'black' and 'white'. To check if a given string is stored in the tree, the search starts from the top and follows the edges of the input string until no further progress can be made. If the search-string is consumed and the final node is a black node, the search has failed; if it is white, the search has succeeded. This enables us to add a large range of strings with a common prefix to the tree, using white nodes, then remove a small set of "exceptions" in a space-efficient manner by *inserting* them using black nodes.

Applications

As mentioned, radix trees are useful for constructing associative arrays with keys that can be expressed as strings. They find particular application in the area of IP routing, where the ability to contain large ranges of values with a few exceptions is particularly suited to the hierarchical organization of IP addresses.^[1] They are also used for inverted indexes of text documents in information retrieval.

History

Donald R. Morrison first described what he called "Patricia tries" in 1968;^[2] the name comes from the acronym **PATRICIA**, which stands for "*Practical Algorithm To Retrieve Information Coded In Alphanumeric*". Gernot Gwehenberger independently invented and described the data structure at about the same time.^[3]

Comparison to other data structures

(In the following comparisons, it is assumed that the keys are of length k and the data structure contains n elements.) Unlike balanced trees, radix trees permit lookup, insertion, and deletion in $O(k)$ time rather than $O(\log n)$. This doesn't seem like an advantage, since normally $k \geq \log n$, but in a balanced tree every comparison is a string comparison requiring $O(k)$ worst-case time, many of which are slow in practice due to long common prefixes. In a trie, all comparisons require constant time, but it takes m comparisons to look up a string of length m . Radix trees can perform these operations with fewer comparisons and require many fewer nodes.

Radix trees also share the disadvantages of tries, however: as they can only be applied to strings of elements or elements with an efficiently reversible mapping (injection) to strings, they lack the full generality of balanced search trees, which apply to any data type with a total ordering. A reversible mapping to strings can be used to produce the required total ordering for balanced search trees, but not the other way around. This can also be problematic if a data type only provides a comparison operation, but not a (de)serialization operation.

Hash tables are commonly said to have expected $O(1)$ insertion and deletion times, but this is only true when considering computation of the hash of the key to be a constant time operation. When hashing the key is taken into account, hash tables have expected $O(k)$ insertion and deletion times, but will take longer in the worst-case depending on how collisions are handled. Radix trees have worst-case $O(k)$ insertion and deletion. The successor/predecessor operations of radix trees are also not implemented by hash tables.

Variants

The **HAT-trie** is a radix tree based cache-conscious data structure that offers efficient string storage and retrieval, and ordered iterations. Performance, with respect to both time and space, is comparable to the cache-conscious hashtable.^[4]

External links

- Algorithms and Data Structures Research & Reference Material: PATRICIA^[5], by Lloyd Allison, Monash University
- Patricia Tree^[6], NIST Dictionary of Algorithms and Data Structures
- Crit-bit trees^[7], by Daniel J. Bernstein
- Radix Tree API in the Linux Kernel^[8], by Jonathan Corbet
- Kart (key alteration radix tree)^[9], by Paul Jarc

Implementations

- A heavily commented dictionary implementation with a binary Radix Tree^[10], by Herbert Glarner (in Linoleum, a cross platform assembler)
- Java implementation of Radix Tree^[11], by Tahseen Ur Rehman
- Practical Algorithm Template Library^[12], a C++ library on PATRICIA tries (VC++ >=2003, GCC G++ 3.x), by Roman S. Klyujkov
- Patricia Trie C++ template class implementation^[13], by Radu Gruian
- Haskell standard library implementation "based on big-endian patricia trees"^[14]. With source code^[15], and a tarball^[16].
- Patricia Trie implementation in Java^[17], by Roger Kapsi and Sam Berlin
- Crit-bit trees^[18] forked from C code by Daniel J. Bernstein
- Patricia Trie implementation in C^[19], in libcprops^[20]

References

- [1] Knizhnik, Konstantin. "Patricia Tries: A Better Index For Prefix Searches" (<http://www.ddj.com/architect/208800854>), *Dr. Dobb's Journal*, June, 2008.
- [2] Morrison, Donald R. Practical Algorithm to Retrieve Information Coded in Alphanumeric (<http://portal.acm.org/citation.cfm?id=321481>)
- [3] G. Gwehenberger, Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen. (<http://cr.yp.to/bib/1968/gwehenberger.html>) Elektronische Rechenanlagen 10 (1968), pp. 223–226
- [4] Askitis, Nikolas; Sinha, Ranjan (2007), "HAT-trie: A Cache-conscious Trie-based Data Structure for Strings" (<http://portal.acm.org/citation.cfm?id=1273749.1273761&coll=GUIDE&dl=>), *Proceedings of the 30th Australasian Conference on Computer science* **62**: pp. 97–105, ISBN 1-920-68243-0,
- [5] <http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/PATRICIA/>
- [6] <http://www.nist.gov/dads/HTML/patriciatree.html>
- [7] <http://cr.yp.to/critbit.html>
- [8] <http://lwn.net/Articles/175432/>
- [9] <http://code.dogmap.org/kart/>
- [10] <http://herbert.gandraxa.com/herbert/dic.asp>
- [11] <http://code.google.com/p/radixtree/>
- [12] <http://code.google.com/p/patl/>
- [13] <http://www.codeproject.com/KB/string/PatriciaTrieTemplateClass.aspx>
- [14] <http://www.haskell.org/ghc/dist/current/docs/libraries/containers/Data-IntMap.html>
- [15] <http://www.haskell.org/ghc/dist/current/docs/libraries/src/Data-IntMap.html>
- [16] <http://hackage.haskell.org/package/containers>
- [17] <http://code.google.com/p/patricia-trie/>
- [18] <http://github.com/agl/critbit>
- [19] http://cprops.sourceforge.net/gen/docs/trie_8c-source.html

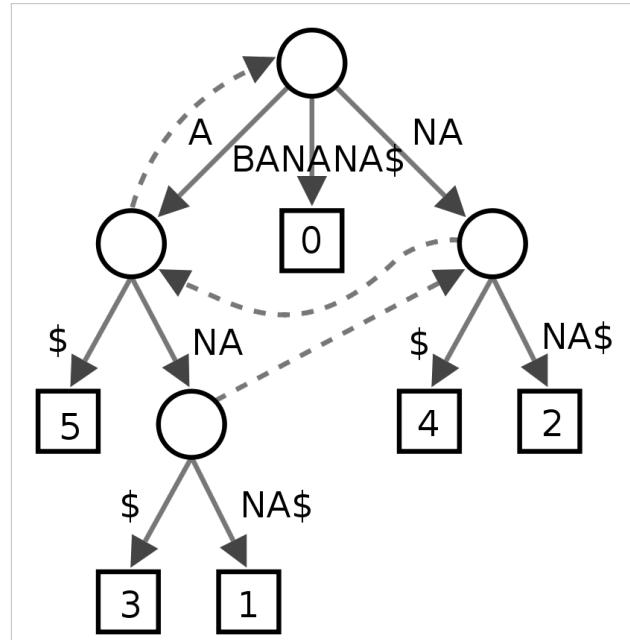
[20] <http://cprops.sourceforge.net>

Suffix tree

In computer science, a **suffix tree** (also called **PAT tree** or, in an earlier form, **position tree**) is a data structure that presents the suffixes of a given string in a way that allows for a particularly fast implementation of many important string operations.

The suffix tree for a string S is a tree whose edges are labeled with strings, such that each suffix of S corresponds to exactly one path from the tree's root to a leaf. It is thus a radix tree (more specifically, a Patricia trie) for the suffixes of S .

Constructing such a tree for the string S takes time and space linear in the length of S . Once constructed, several operations can be performed quickly, for instance locating a substring in S , locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern etc. Suffix trees also provided one of the first linear-time solutions for the longest common substring problem. These speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.



Suffix tree for the string BANANA. Each substring is terminated with special character \$. The six paths from the root to a leaf (shown as boxes) correspond to the six suffixes A\$, NA\$, ANA\$, NANA\$, ANANA\$ and BANANA\$. The numbers in the boxes give the start position of the corresponding suffix. Suffix links drawn dashed.

History

The concept was first introduced as a *position tree* by Weiner in 1973^[1] in a paper which Donald Knuth subsequently characterized as "Algorithm of the Year 1973". The construction was greatly simplified by McCreight in 1976^[2], and also by Ukkonen in 1995^[3]^[4]. Ukkonen provided the first linear-time online-construction of suffix trees, now known as Ukkonen's algorithm.

Definition

The suffix tree for the string S of length n is defined as a tree such that (^[5] page 90):

- the paths from the root to the leaves have a one-to-one relationship with the suffixes of S ,
- edges spell non-empty strings,
- and all internal nodes (except perhaps the root) have at least two children.

Since such a tree does not exist for all strings, S is padded with a terminal symbol not seen in the string (usually denoted \$). This ensures that no suffix is a prefix of another, and that there will be n leaf nodes, one for each of the n suffixes of S . Since all internal non-root nodes are branching, there can be at most $n - 1$ such nodes, and $n + (n - 1) + 1 = 2n$ nodes in total (n leaves, n-1 internal nodes, 1 root).

Suffix links are a key feature for linear-time construction of the tree. In a complete suffix tree, all internal non-root nodes have a suffix link to another internal node. If the path from the root to a node spells the string χ^α , where χ

is a single character and α is a string (possibly empty), it has a suffix link to the internal node representing α . See for example the suffix link from the node for ANA to the node for NA in the figure above. Suffix links are also used in some algorithms running on the tree.

Functionality

A suffix tree for a string S of length n can be built in $\Theta(n)$ time, if the alphabet is constant or integer^[6].

Otherwise, the construction time depends on the implementation. The costs below are given under the assumption that the alphabet is constant. If it is not, the cost depends on the implementation (see below).

Assume that a suffix tree has been built for the string S of length n , or that a generalised suffix tree has been built for the set of strings $D = \{S_1, S_2, \dots, S_K\}$ of total length $n = |n_1| + |n_2| + \dots + |n_K|$. You can:

- Search for strings:
 - Check if a string P of length m is a substring in $O(m)$ time^[5] page 92).
 - Find the first occurrence of the patterns P_1, \dots, P_q of total length m as substrings in $O(m)$ time.
 - Find all z occurrences of the patterns P_1, \dots, P_q of total length m as substrings in $O(m + z)$ time^[5] page 123).
 - Search for a regular expression P in time expected sublinear in n ^[7].
 - Find for each suffix of a pattern P , the length of the longest match between a prefix of $P[i..m]$ and a substring in D in $\Theta(m)$ time^[5] page 132). This is termed the **matching statistics** for P .
- Find properties of the strings:
 - Find the longest common substrings of the string S_i and S_j in $\Theta(n_i + n_j)$ time^[5] page 125).
 - Find all maximal pairs, maximal repeats or supermaximal repeats in $\Theta(n + z)$ time^[5] page 144).
 - Find the Lempel-Ziv decomposition in $\Theta(n)$ time^[5] page 166).
 - Find the longest repeated substrings in $\Theta(n)$ time.
 - Find the most frequently occurring substrings of a minimum length in $\Theta(n)$ time.
 - Find the shortest strings from Σ that do not occur in D , in $O(n + z)$ time, if there are z such strings.
 - Find the shortest substrings occurring only once in $\Theta(n)$ time.
 - Find, for each i , the shortest substrings of S_i not occurring elsewhere in D in $\Theta(n)$ time.

The suffix tree can be prepared for constant time lowest common ancestor retrieval between nodes in $\Theta(n)$ time^[5] chapter 8). You can then also:

- Find the longest common prefix between the suffixes $S_i[p..n_i]$ and $S_j[q..n_j]$ in $\Theta(1)$ ^[5] page 196).
- Search for a pattern P of length m with at most k mismatches in $O(kn + z)$ time, where z is the number of hits^[5] page 200).
- Find all z maximal palindromes in $\Theta(n)$ ^[5] page 198), or $\Theta(gn)$ time if gaps of length g are allowed, or $\Theta(kn)$ if k mismatches are allowed^[5] page 201).
- Find all z tandem repeats in $O(n \log n + z)$, and k-mismatch tandem repeats in $O(kn \log(n/k) + z)$ ^[5] page 204).
- Find the longest substrings common to at least k strings in D for $k = 2..K$ in $\Theta(n)$ time^[5] page 205).

Applications

Suffix trees can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology and other application areas.^[8] Primary applications include:^[8]

- String search, in $O(m)$ complexity, where m is the length of the sub-string (but with initial $O(n)$ time required to build the suffix tree for the string)
- Finding the longest repeated substring
- Finding the longest common substring
- Finding the longest palindrome in a string

Suffix trees are often used in bioinformatics applications, searching for patterns in DNA or protein sequences (which can be viewed as long strings of characters). The ability to search efficiently with mismatches might be considered their greatest strength. Suffix trees are also used in data compression; they can be used to find repeated data, and can be used for the sorting stage of the Burrows-Wheeler transform. Variants of the LZW compression schemes use suffix trees (LZSS). A suffix tree is also used in suffix tree clustering, a data clustering algorithm used in some search engines (first introduced in^[9]).

Implementation

If each node and edge can be represented in $\Theta(1)$ space, the entire tree can be represented in $\Theta(n)$ space. The total length of all the strings on all of the edges in the tree is $O(n^2)$, but each edge can be stored as the position and length of a substring of S , giving a total space usage of $\Theta(n)$ computer words. The worst-case space usage of a suffix tree is seen with a fibonacci string, giving the full $2n$ nodes.

An important choice when making a suffix tree implementation is the parent-child relationships between nodes. The most common is using linked lists called **sibling lists**. Each node has pointer to its first child, and to the next node in the child list it is a part of. Hash maps, sorted/unordered arrays (with array doubling), and balanced search trees may also be used, giving different running time properties. We are interested in:

- The cost of finding the child on a given character.
- The cost of inserting a child.
- The cost of enlisting all children of a node (divided by the number of children in the table below).

Let σ be the size of the alphabet. Then you have the following costs:

	Lookup	Insertion	Traversal
Sibling lists / unsorted arrays	$O(\sigma)$	$\Theta(1)$	$\Theta(1)$
Hash maps	$\Theta(1)$	$\Theta(1)$	$O(\sigma)$
Balanced search tree	$O(\log \sigma)$	$O(\log \sigma)$	$O(1)$
Sorted arrays	$O(\log \sigma)$	$O(\sigma)$	$O(1)$
Hash maps + sibling lists	$O(1)$	$O(1)$	$O(1)$

Note that the insertion cost is amortised, and that the costs for hashing are given perfect hashing.

The large amount of information in each edge and node makes the suffix tree very expensive, consuming about ten to twenty times the memory size of the source text in good implementations. The suffix array reduces this requirement to a factor of four, and researchers have continued to find smaller indexing structures.

External Construction

Suffix trees quickly outgrow the main memory on standard machines for sequence collections in the order of gigabytes. As such, their construction calls for external memory approaches.

There are theoretical results for constructing suffix trees in external memory. The algorithm by Farach et al. [10] is theoretically optimal, with an I/Os complexity equal to that of sorting. However, as discussed for example in [11], the overall intricacy of this algorithm has prevented, so far, its practical implementation.

On the other hand, there have been practical works for constructing disk-based suffix trees which scale to (few) GB/hours. The state of the art methods are TDD [12], TRELLIS [13], DiGeST [14], and B²ST [15]. TDD and TRELLIS scale up to the entire human genome - approximately 3GB - resulting in a disk-based suffix tree of a size in the tens of gigabytes [12], [13]. However, these methods cannot handle efficiently collections of sequences exceeding 3GB [14]. DiGeST performs significantly better and is able to handle collections of sequences in the order of 6GB in about 6 hours [14]. The source code and documentation for the latter is available from [16]. All these methods can efficiently build suffix trees for the case when the tree does not fit in main memory, but the input does. The most recent method, B2ST [15], scales to handle inputs that do not fit in main memory.

See also

- Suffix array

External links

- Suffix Trees [17] by Dr. Sartaj Sahni (CISE Department Chair at University of Florida)
- Suffix Trees [18] by Lloyd Allison
- NIST's Dictionary of Algorithms and Data Structures: Suffix Tree [19]
- suffix_tree [20] ANSI C implementation of a Suffix Tree
- libstree [21], a generic suffix tree library written in C
- Tree::Suffix [22], a Perl binding to libstree
- Strmat [23] a faster generic suffix tree library written in C (uses arrays instead of linked lists)
- SuffixTree [24] a Python binding to Strmat
- Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice [25], application of suffix trees in the BWT
- Theory and Practice of Succinct Data Structures [26], C++ implementation of a compressed suffix tree]
- Practical Algorithm Template Library [12], a C++ library with suffix tree implementation on PATRICIA trie, by Roman S. Klyujkov

References

- [1] P. Weiner (1973). "Linear pattern matching algorithm". *14th Annual IEEE Symposium on Switching and Automata Theory*. pp. 1–11.
- [2] Edward M. McCreight (1976). "A Space-Economical Suffix Tree Construction Algorithm" (<http://doi.acm.org/10.1145/321941.321946>). *Journal of the ACM* **23** (2): 262–272. doi:10.1145/321941.321946..
- [3] E. Ukkonen (1995). "On-line construction of suffix trees" (<http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf>). *Algorithmica* **14** (3): 249–260. doi:10.1007/BF01206331..
- [4] R. Giegerich and S. Kurtz (1997). "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction" (<http://www.zbh.uni-hamburg.de/staff/kurtz/papers/GieKur1997.pdf>). *Algorithmica* **19** (3): 331–353. doi:10.1007/PL00009177..
- [5] Gusfield, Dan (1999) [1997]. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. USA: Cambridge University Press. ISBN 0-521-58519-8.
- [6] Martin Farach (1997). "Optimal suffix tree construction with large alphabets" (<ftp://dimacs.rutgers.edu/pub/dimacs/TechnicalReports/TechReports/1996/96-48.ps.gz>). *Foundations of Computer Science, 38th Annual Symposium on*. pp. 137–143. .
- [7] Ricardo A. Baeza-Yates and Gaston H. Gonnet (1996). "Fast text searching for regular expressions or automaton searching on tries". *Journal of the ACM (ACM Press)* **43** (6): 915–936. doi:10.1145/235809.235810.
- [8] Allison, L.. "Suffix Trees" (<http://www.allisons.org/ll/AlgDS/Tree/Suffix/>). . Retrieved 2008-10-14.

- [9] Oren Zamir and Oren Etzioni (1998). "Web document clustering: a feasibility demonstration". *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. pp. 46--54.
- [10] Martin Farach-Colton, Paolo Ferragina, S. Muthukrishnan (2000). "On the sorting-complexity of suffix tree construction.". *J. ACM* 47(6) **47** (6): 987–1011.
- [11] Smyth, William (2003). *Computing Patterns in Strings*. Addison-Wesley.
- [12] Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel (2003). "Practical Suffix Tree Construction". *VLDB '03: Proceedings of the 30th International Conference on Very Large Data Bases*. Morgan Kaufmann. pp. 36–47.
- [13] Benjarath Phoophakdee and Mohammed J. Zaki (2007). "Genome-scale disk-based suffix tree indexing". *SIGMOD '07: Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM. pp. 833–844.
- [14] Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton (2008). "A new method for indexing genomes using on-disk suffix trees". *CIKM '08: Proceedings of the 17th ACM Conference on Information and Knowledge Management*. ACM. pp. 649–658.
- [15] Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton (2009). "Suffix trees for very large genomic sequences". *CIKM '09: Proceedings of the 18th ACM Conference on Information and Knowledge Management*. ACM.
- [16] "The disk-based suffix tree for pattern search in sequenced genomes" (http://webhome.cs.uvic.ca/~mgbarsky/DIGEST_SEARCH). . Retrieved 10152009.
- [17] <http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c16/suffix.htm>
- [18] <http://www.allisons.org/ll/AlgDS/Tree/Suffix/>
- [19] <http://www.nist.gov/dads/HTML/suffixtree.html>
- [20] http://mila.cs.technion.ac.il/~yona/suffix_tree/
- [21] <http://www.cl.cam.ac.uk/~cpk25/libstree/>
- [22] <http://search.cpan.org/dist/Tree-Suffix/>
- [23] <http://www.cs.ucdavis.edu/~gusfield/strmat.html>
- [24] http://hkn.eecs.berkeley.edu/~dyoo/python/suffix_trees/
- [25] <http://www.balkenhol.net/papers/t1043.pdf.gz>
- [26] <http://www.cs.helsinki.fi/group/suds/>

Suffix array

In computer science, a **suffix array** is an array of integers giving the starting positions of suffixes of a string in lexicographical order.

Details

Consider the string

1	2	3	4	5	6	7	8	9	10	11	12
a	b	r	a	c	a	d	a	b	r	a	\$

of length 12, that ends with a sentinel letter \$, appearing only once and less than any other letter in the string.

It has twelve suffixes: "abracadabra\$", "bracadabra\$", "racadabra\$", and so on down to "a\$" and "\$" that can be sorted into lexicographical order to obtain:

index	sorted suffix	lcp
12	\$	0
11	a\$	0
8	abra\$	1
1	abracadabra\$	4
4	acadabra\$	1
6	adabra\$	1
9	bra\$	0
2	bracadabra\$	3
5	cadabra\$	0
7	dabra\$	0
10	ra\$	0
3	racadabra\$	2

If the original string is available, each suffix can be completely specified by the index of its first character. The suffix array is the array of the indices of suffixes sorted in lexicographical order. For the string "abracadabra\$", using one-based indexing, the suffix array is {12,11,8,1,4,6,9,2,5,7,10,3}, because the suffix "\$" begins at position 12, "a\$" begins at position 11, "abra\$" begins at position 8, and so forth.

The longest common prefix is also shown above as lcp. This value, stored alongside the list of prefix indices, indicates how many characters a particular suffix has in common with the suffix directly above it, starting at the beginning of both suffixes. The lcp is useful in making some string operations more efficient. For example, it can be used to avoid comparing characters that are already known to be the same when searching through the list of suffixes. The fact that the minimum lcp value belonging to a consecutive set of sorted prefixes gives the longest common prefix among all of those suffixes can also be useful.

Algorithms

The easiest way to construct a suffix array is to use an efficient comparison sort algorithm. This requires $O(n \log n)$ suffix comparisons, but a suffix comparison requires $O(n)$ time, so the overall runtime of this approach is $O(n^2 \log n)$. More sophisticated algorithms improve this to $O(n \log n)$ by exploiting the results of partial sorts to avoid redundant comparisons. Several $\Theta(n)$ algorithms (of Pang Ko and Srinivas Aluru, Juha Kärkkäinen and Peter Sanders, etc) have also been developed which provide faster construction and have space usage of $O(n)$ with low constants.

Recent work by Salson *et al.* proposes an algorithm for updating the suffix array of a text that has been edited instead of rebuilding a new suffix array from scratch. Even if the theoretical worst-case time complexity is $O(n \log n)$, it appears to perform well in practice: experimental results from the authors showed that their implementation of dynamic suffix arrays is generally more efficient than rebuilding when comparing the insertion of 500 characters into their dynamic array with a complete rebuild of a static suffix array containing the updated text.

Applications

The suffix array of a string can be used as an index to quickly locate every occurrence of a substring within the string. Finding every occurrence of the substring is equivalent to finding every suffix that begins with the substring. Thanks to the lexicographical ordering, these suffixes will be grouped together in the suffix array, and can be found efficiently with a binary search. If implemented straightforwardly, this binary search takes $O(m \log n)$ time, where m is the length of the substring W . The following pseudo-code from Manber and Myers shows how to find W (or the suffix lexicographically immediately before W if W is not present) in a suffix array with indices stored in p and longest common prefixes stored in lcp .

```

if W <= suffixAt(pos[0]) then
    ans = 0
else if W > suffixAt(pos[m-1]) then
    ans = m
else
{
    L = 0, R = m-1
    while R-L > 1 do
    {
        M = (L + R)/2
        if W <= suffixAt(pos[M]) then
            R = M
        else
            L = M
    }
    ans = R
}

```

To avoid redoing comparisons, extra data structures giving information about the longest common prefixes (LCPs) of suffixes are constructed, giving $O(m + \log n)$ search time.

Suffix sorting algorithms can be used to perform the Burrows–Wheeler transform (BWT). Technically the BWT requires sorting cyclic permutations of a string, not suffixes. We can fix this by appending to the string a special end-of-string character which sorts lexicographically before every other character. Sorting cyclic permutations is then equivalent to sorting suffixes.

Suffix arrays are used to look up substrings in Example-Based Machine Translation, demanding much less storage than a full phrase table as used in Statistical machine translation.

History

Suffix arrays were originally developed by Gene Myers and Udi Manber to reduce memory consumption compared to a suffix tree. This began the trend towards compressed suffix arrays and BWT-based compressed full-text indices.

References

- Udi Manber and Gene Myers (1991). "Suffix arrays: a new method for on-line string searches". *SIAM Journal on Computing*, Volume 22, Issue 5 (October 1993), pp. 935–948.
- Pang Ko and Srinivas Aluru (2003). "Space efficient linear time construction of suffix arrays." In *Combinatorial Pattern Matching (CPM 03)*. LNCS 2676, Springer, 2003, pp 203–210.

- Juha Kärkkäinen and Peter Sanders (2003). "Simple linear work suffix array construction." In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*. LNCS 2719, Springer, 2003, pp. 943–955.
- Klaus-Bernd Schürmann and Jens Stoye (2005). "An incomplex algorithm for fast suffix array construction". In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the 2nd Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO 2005)*, 2005, pp. 77–85.
- Mikaël Salson, Martine Léonard, Thierry Lecroq and Laurent Mouchard (2009) "Dynamic Extended Suffix Arrays", *Journal of Discrete Algorithms* 2009 [1].

External links

- Various algorithms for constructing Suffix Arrays in Java, with performance tests [2]
- Suffix sorting module for BWT in C code [3]
- Suffix Array Implementation in Ruby [4]
- Suffix array library and tools [5]
- Project containing Suffix Array Implementation in Java [6]
- Project containing various Suffix Array c/c++ Implementations with a unified interface [7]

References

- [1] <http://dx.doi.org/10.1016/j.jda.2009.02.007>
- [2] <http://www.jsuffixarrays.org>
- [3] <http://code.google.com/p/compression-code/downloads/list>
- [4] <http://www.codeodor.com/index.cfm/2007/12/24/The-Suffix-Array/1845>
- [5] <http://sary.sourceforge.net/index.html.en>
- [6] <http://duch.mimuw.edu.pl/~abuczyns/kolokacje/index-en.html>
- [7] <http://pizzachili.dcc.uchile.cl/>

Compressed suffix array

The Compressed Suffix Array^[1] ^[2] is a compressed data structure for pattern matching. Given a text T of n characters from an alphabet Σ , the compressed suffix array support searching for arbitrary patterns in T . For an input pattern P of m characters, the search time is equal to n times the higher-order entropy of the text T , plus some extra bits to store the empirical statistical model plus $o(n)$.

The original instantiation of the compressed suffix array^[1] solved a long-standing open problem by showing that fast pattern matching was possible using only a linear-space data structure, namely, one proportional to the size of the text T , which takes $O(n \log_{|\Sigma|} n)$ bits. The conventional suffix array and suffix tree use $\Omega(n \log n)$ bits, which is substantially larger. The basis for the data structure is a recursive decomposition using the "neighbor function," which allows a suffix array to be represented by one of half its length. The construction is repeated multiple times until the resulting suffix array uses a linear number of bits. Following work showed that the actual storage space was related to the zeroth-order entropy and that the index supports self-indexing.^[3] The space bound was further improved using adaptive coding with longer contexts to achieve the ultimate goal of higher-order entropy.^[2] The space usage is extremely competitive in practice with other state-of-the-art compressors,^[4] and it also supports fast pattern matching.

The memory accesses made by compressed suffix arrays and other compressed data structures for pattern matching are typically not localized, and thus these data structures have been notoriously hard to design efficiently for use in external memory. Recent progress using geometric duality takes advantage of the block access provided by disks to speed up the I/O time significantly^[5]

References

- [1] R. Grossi and J. S. Vitter, Compressed Suffix Arrays and Suffix Trees, with Applications to Text Indexing and String Matching], *SIAM Journal on Computing*, 35(2), 2005, 378-407. An earlier version appeared in *Proceedings of the 32nd ACM Symposium on Theory of Computing*, May 2000, 397-406.
- [2] R. Grossi, A. Gupta, and J. S. Vitter, High-Order Entropy-Compressed Text Indexes, *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms*, January 2003, 841-850.
- [3] K. Sadakane, Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Arrays, *Proceedings of the International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science, vol. 1969, Springer, December 2000, 410-421.
- [4] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, Indexing Equals Compression: Experiments on Suffix Arrays and Trees, *ACM Transactions on Algorithms*, 2(4), 2006, 611-639.
- [5] W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter, On Entropy-Compressed Text Indexing in External Memory, *Proceedings of the Conference on String Processing and Information Retrieval*, August 2009.

FM-index

The **FM-index** is type of Substring index based on the Burrows-Wheeler transform, with some similarities to the Suffix Array.

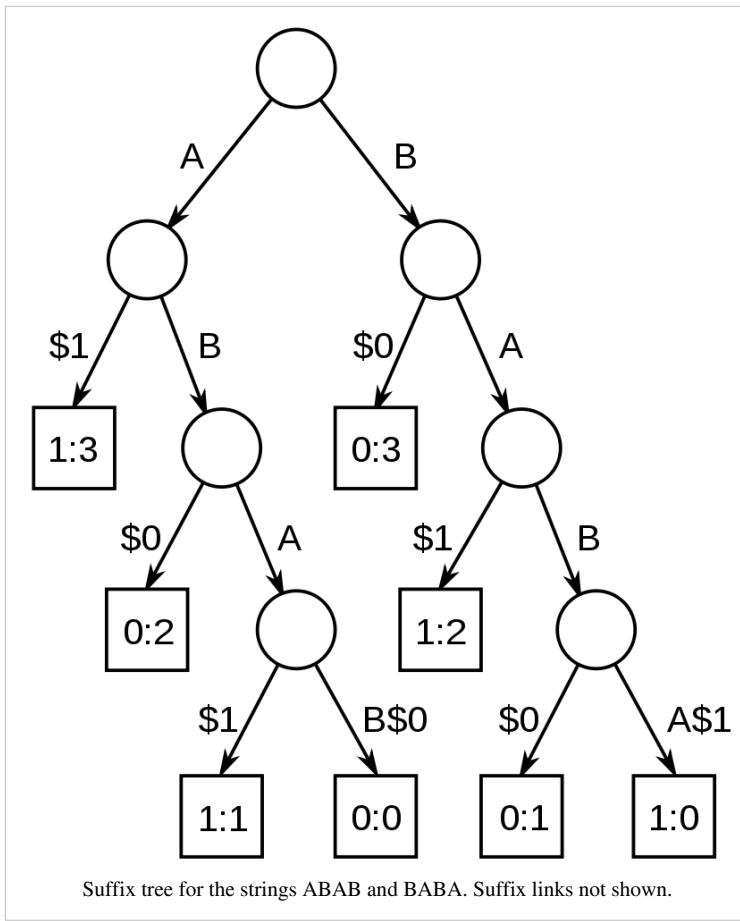
It is named after the creators of the algorithm, Paolo Ferragina and Giovanni Manzini,^[1] who describe it as an opportunistic data structure as it allows compression of the input text while still permitting fast substring queries.

It allows both the query time and storage space requirements to be sublinear with respect to the size of the input data.

References

- [1] Opportunistic Data Structures with Applications. Paolo Ferragina and Giovanni Manzini. (2000) p.390 Proceedings of the 41st Annual Symposium on Foundations of Computer Science

Generalised suffix tree



A **generalised suffix tree** is a suffix tree for a set of strings. Given the set of strings $D = S_1, S_2, \dots, S_d$ of total length n , it is a Patricia trie containing all n suffixes of the strings. It is mostly used in bioinformatics.^{BRCR}

Functionality

It can be built in $\Theta(n)$ time and space, and can be used to find all z occurrences of a string P of length m in $O(m + z)$ time, which is asymptotically optimal (assuming the size of the alphabet is constant, see ^{Gus97} page 119).

When constructing such a tree, each string should be padded with a unique out-of-alphabet marker symbol (or string) to ensure no suffix is a substring of another, guaranteeing each suffix is represented by a unique leaf node.

Algorithms for constructing a GST include Ukkonen's algorithm and McCreight's algorithm.

Example

A suffix tree for the strings ABAB and BABA is shown in a figure above. They are padded with the unique terminator strings \$0 and \$1. The numbers in the leaf nodes are string number and starting position. Notice how a left to right traversal of the leaf nodes corresponds to the sorted order of the suffixes. The terminators might be strings or unique single symbols. Edges on \$ from the root are left out in this example.

Alternatives

An alternative to building a generalised suffix tree is to concatenate the strings, and build a regular suffix tree or suffix array for the resulting string. When hits are evaluated after a search, global positions are mapped into documents and local positions with some algorithm and/or data structure, such as a binary search in the starting/ending positions of the documents.

References

- Lucas Chi Kwong Hui (1992). "Color Set Size Problem with Applications to String Matching" ^[1]. *Combinatorial Pattern Matching, Lecture Notes in Computer Science*, 644.. pp. 230–243.
- Paul Bieganski, John Riedl, John Carlis, and Ernest F. Retzel (1994). "Generalized Suffix Trees for Biological Sequence Data" ^[2]. *Biotechnology Computing, Proceedings of the Twenty-Seventh Hawaii International Conference on..* pp. 35–44.
- Gusfield, Dan (1999) [1997]. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. USA: Cambridge University Press. ISBN 0-521-58519-8.

External links

- Online GST demo ^[3]: a web demo for generating a generalised suffix tree.

References

- [1] <http://www.springerlink.com/content/y565487707522555/>
- [2] http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=323593
- [3] <http://www.cosmion.net/jeroen/software/gst/>

B-trie

The **B-trie** is a trie-based data structure that can store and retrieve variable-length strings efficiently on disk.^[1]

The B-trie was compared against several high-performance variants of B-tree that were designed for string keys. It was shown to offer superior performance, particularly under skew access (i.e., many repeated searches). It is currently a leading choice for maintaining a string dictionary on disk, along with other disk-based tasks, such as maintaining an index to a string database or for accumulating the vocabulary of a large text collection.

References

- [1] Askitis, Nikolas; Zobel, Justin (2008), "B-tries for Disk-based String Management" (<http://www.springerlink.com/content/x7545u2g85675u17/>), *VLDB Journal*: pp. 1–26, ISBN 1066-8888 (Print) 0949-877X (Online),

Judy array

In computer science and software engineering, a **Judy array** is a complex but very fast associative array data structure for storing and looking up values using integer or string keys. Unlike normal arrays, Judy arrays may be sparse; that is, they may have large ranges of unassigned indices.

Judy arrays are designed to keep the number of processor cache-line fills as low as possible, and the algorithm is internally complex in an attempt to satisfy this goal as often as possible. Due to these cache optimizations, Judy arrays are fast, sometimes even faster than a hash table. And because Judy arrays are a type of trie, they consume much less memory than hash tables.

Roughly speaking, it is similar to a highly-optimised 256-ary trie data structure.^[1]

Judy arrays are also believed to be less vulnerable to algorithmic complexity attacks.^[2]

The Judy array was invented by Doug Baskins and named after his sister.

External links

- Main Judy arrays site^[3]
- Programming with Judy: C LanguageJudy^[4]
- How Judy arrays work and why they are so fast^[5]
- A complete technical description of Judy arrays^[6]
- An independent performance comparison of Judy to Hash Tables^[7]

References

- [1] Alan Silverstein, " Judy IV Shop Manual (http://judy.sourceforge.net/application/shop_interm.pdf)", 2002
[2] Denial of Service via Algorithmic Complexity Attacks (<http://www.cs.rice.edu/~scrosby/hash/>)
[3] <http://judy.sourceforge.net/>
[4] <http://docs.hp.com/en/B6841-90001/index.html>
[5] <http://judy.sourceforge.net/downloads/10minutes.htm>
[6] http://judy.sourceforge.net/application/shop_interm.pdf
[7] <http://www.nothings.org/computer/judy/>

Directed acyclic word graph

In computer science, a **directed acyclic word graph** (sometimes abbreviated as a **DAWG**) is a data structure that represents a set of strings, and allows for a query operation that tests whether a given string belongs to the set in time proportional to its length. In these respects, a DAWG is very similar to a trie, but it is much more space efficient.

A DAWG is represented as a directed acyclic graph with a single source vertex (a vertex with no incoming edges), in which each edge of the graph is labeled by a letter, symbol, or special end-of-string marker, and in which each vertex has at most one outgoing edge for each possible letter or symbol. The strings represented by the DAWG are formed by the symbols on paths in the DAWG from the source vertex to any sink vertex (a vertex with no outgoing edges).

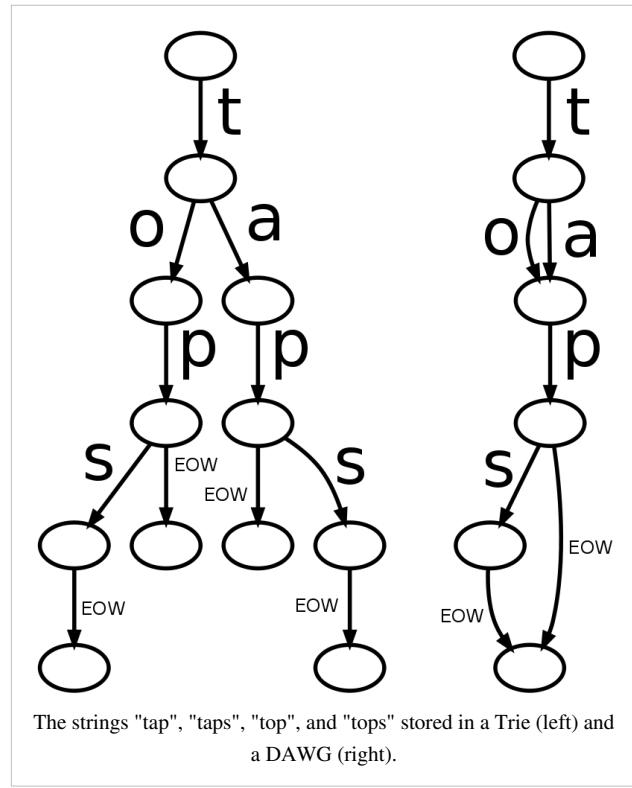
Thus, a trie (a rooted tree with the same properties of having edges labeled by symbols and strings formed by root-to-leaf paths) is a special kind of DAWG.

However, by allowing the same vertices to be reached by multiple paths, a DAWG may use significantly fewer vertices than a trie. Consider, for example, the four English words "tap", "taps", "top", and "tops". A trie for those four words would have 11 vertices, one for each of the strings formed as a prefix of one of these words, or for one of the words followed by the end-of-string marker. However, a DAWG can represent these same four words using only six vertices v_i for $0 \leq i \leq 5$, and the following edges: an edge from v_0 to v_1 labeled "t", two edges from v_1 to v_2 labeled "a" and "o", an edge from v_2 to v_3 labeled "p", an edge v_3 to v_4 labeled "s", and edges from v_3 and v_4 to v_5 labeled with the end-of-string marker.

The primary difference between DAWG and trie is the elimination of suffix redundancy in storing strings. The trie eliminates prefix redundancy since all common prefixes are shared between strings, such as between *doctors* and *doctorate* the *doctor* prefix is shared. In a DAWG common suffixes are also shared, such as between *desertion* and *desertification* both the prefix *deserti-* and suffix *-tion* are shared. For dictionary sets of common English words, this translates into major memory usage reduction.

References

- Appel, Andrew; Jacobsen, Guy (1988), *possibly first mention of the data structure*^[1], "The World's Fastest Scrabble Program", *Communications of the ACM*: 572–578.
- Crochemore, Maxime; Vérité, Renaud (1997), "Direct construction of compact directed acyclic word graphs", *Combinatorial Pattern Matching*, Lecture Notes in Computer Science, Springer-Verlag, pp. 116–129, doi:10.1007/3-540-63220-4_55.
- Inenaga, S.; Hoshino, H.; Shinohara, A.; Takeda, M.; Arikawa, S. (2001), "On-line construction of symmetric compact directed acyclic word graphs"^[2], *Proc. 8th Int. Symp. String Processing and Information Retrieval, 2001. SPIRE 2001.*, pp. 96–110.
- Jansen, Cees J. A.; Boekee, Dick E. (1990), "On the significance of the directed acyclic word graph in cryptology", *Advances in Cryptology — AUSCRYPT '90*, Lecture Notes in Computer Science, **453**,



The strings "tap", "taps", "top", and "tops" stored in a Trie (left) and a DAWG (right).

Springer-Verlag, pp. 318–326, doi:10.1007/BFb0030372.

External links

- National Institute of Standards and Technology ^[3]
- <http://www.wutka.com/dawg.html> ^[4]
- Optimal DAWG Creation Step By Step Treatment ^[5] By JohnPaul Adamovsky (Aerospace Engineer)

References

- [1] <http://portal.acm.org/citation.cfm?id=42420>
- [2] http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=989743
- [3] <http://www.nist.gov/dads/HTML/directedAcyclicWordGraph.html>
- [4] <http://www.wutka.com/dawg.html>
- [5] <http://www.pathcom.com/~vadco/dawg.html>

Multiway trees

Ternary search tree

In computer science, a **ternary search tree** is a ternary (three-way) tree data structure of strings which combines the speed of a prefix search tree, or trie, with the space efficiency of a binary search tree.^[1]

Binary trees and tries represent different approaches to looking up strings. In a binary tree, each node represents one string in the data set. Thus, the number of nodes is minimized. Also, each node contains links to only two other nodes. Thus, the size of each node is minimal. Binary search trees are very compact. However, to find a string within the tree, that entire string must be compared with the entire string contained at each node searched. This is slow.

A trie is optimized for speed at the expense of size. Descending the trie only involves looking at each character once, eliminating string comparisons. In the most basic implementation, each node has as many children as letters in the string's alphabet. The trie is navigated by looking up the string's characters in sequence and descending through the nodes. The process is fast, but the number and size of the nodes is large.

The ternary search tree replaces each node of the trie with a modified binary search tree. For sparse tries, this binary tree will be smaller than a trie node. Each binary tree implements a single-character lookup. It has the typical left and right children which are checked if the lookup character is greater or less than the node's character, respectively. A third child is used if the lookup character is found on that particular node. Unlike the other children, it links to the root of the binary search tree for the next character in the string.

Therefore, string lookup in a ternary search tree consists of a series of binary searches for individual characters.

Ternary search trees are usually rejected in favor of hash tables. Also, there are other ways to construct a trie. For example, a radix tree trie in which each node stores a bit string, and has two children for the possible next bits, may be more compact than a ternary search tree without being slow.

See also

- Ternary search
- Search algorithm
- Binary search
- Interpolation search
- Linear search

References

- [1] Bentley, Jon; Sedgewick, Bob (April 1), "Ternary Search Trees" (<http://www.ddj.com/184410528>), *Dr. Dobb's Journal*,
 • *Algorithms in C/C++/Java*, third ed. (Parts 1-4), Robert Sedgewick, Addison Wesley.

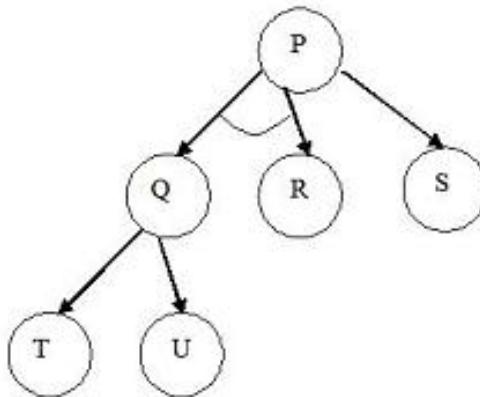
External links

- Ternary Search Trees (<http://www.cs.princeton.edu/~rs/strings/>)
- Ternary Search Trees (Dr. Dobbs) (<http://www.ddj.com/documents/s=921/ddj9804a/>)
- Tree::Ternary (Perl module) (<http://search.cpan.org/~mrogaski/Tree-Ternary-0.03/Ternary.pm>)
- Ternary Search Tree code (<http://dasnar.sdf-eu.org/res/ctst-README.html>)
- STL-compliant Ternary Search Tree in C++ (<http://abc.se/~re/code/tst/>)
- Ternary Search Tree in C++ (<http://ternary.sourceforge.net>)

And-or tree

An **and-or tree** is a graphical representation of the reduction of problems (or goals) to conjunctions and disjunctions of subproblems (or subgoals).

Example



The and-or tree:

represents the search space for solving the problem P, using the goal-reduction methods:

P if Q and R

P if S

Q if T

Q if U

Definitions

Given an initial problem P_0 and set of problem solving methods of the form:

$P \text{ if } P_1 \text{ and } \dots \text{ and } P_n$

the associated and-or tree is a set of labelled nodes such that:

1. The root of the tree is a node labelled by P_0 .
1. For every node N labelled by a problem or sub-problem P and for every method of the form $P \text{ if } P_1 \text{ and } \dots \text{ and } P_n$, there exists a set of children nodes N_1, \dots, N_n of the node N , such that each node N_i is labelled by P_i . The nodes are conjoined by an arc, to distinguish them from children of N that might be associated with other methods.

A node N , labelled by a problem P , is a success node if there is a method of the form $P \text{ if nothing}$ (i.e., P is a "fact"). The node is a failure node if there is no method for solving P .

If all of the children of a node N , conjoined by the same arc, are success nodes, then the node N is also a success node. Otherwise the node is a failure node.

Search strategies

An and-or tree specifies only the search space for solving a problem. Different search strategies for searching the space are possible. These include searching the tree depth-first, breadth-first, or best-first using some measure of desirability of solutions. The search strategy can be sequential, searching or generating one node at a time, or parallel, searching or generating several nodes in parallel.

Relationship with logic programming

The methods used for generating and-or trees are propositional logic programs (without variables). In the case of logic programs containing variables, the solutions of conjoint sub-problems must be compatible. Subject to this complication, sequential and parallel search strategies for and-or trees provide a computational model for executing logic programs.

Relationship with two-player games

And-or trees can also be used to represent the search spaces for two-person games. The root node of such a tree represents the problem of one of the players winning the game, starting from the initial state of the game. Given a node N , labelled by the problem P of the player winning the game from a particular state of play, there exists a single set of conjoint children nodes, corresponding to all of the opponents responding moves. For each of these children nodes, there exists a set of non-conjoint children nodes, corresponding to all of the player's defending moves.

For solving game trees with proof-number search family of algorithms, game trees are to be mapped to And/Or trees. MAX-nodes (i.e. maximizing player to move) are represented as OR nodes, MIN-nodes map to AND nodes. The mapping is possible, when the search is done with only a binary goal, which usually is "player to move wins the game".

Bibliography

- Luger, G.F. and Stubblefield, W.A. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving* (2nd Edition), The Benjamin/Cummings Publishing Company, Inc., 1993.
- Nilsson, N.J. *Artificial Intelligence, A New Synthesis*, Morgan Kaufmann Publishers, Inc., 1998.

(a,b)-tree

In computer science, an **(a,b) tree** is a specific kind of search tree.

An (a,b) tree has all of its leaves at the same depth, and all internal nodes have between a and b children, where a and b are integers such that $2 \leq a \leq (b + 1)/2$. The root may have as few as zero children.

Definition

Let $a, b \in N$ such that $a \leq b$. Then tree a tree T is an **(a,b) tree** when:

- Every inner node except the root has at least a and maximally b child nodes.
- Root has maximally b child nodes.
- All paths from the root to the leaves are of the same length.

Inner node representation

Every inner node v has the following representation:

- Let ρ_v be the number of child nodes of node v .
- Let $S_v[1 \dots \rho_v]$ be pointers to child nodes.
- Let $H_v[1 \dots \rho_v - 1]$ be an array of keys such that $H_v[i]$ equals the largest key in the subtree pointed to by $S_v[i]$.

See also

- B-tree

References

-  This article incorporates public domain material from the NIST document "(a,b)-tree" [1] by Paul E. Black (Dictionary of Algorithms and Data Structures).

References

[1] <http://www.nist.gov/dads/HTML/abtree.html>

Link/cut tree

A **link/cut tree** is a type of data structure capable of merging (link operation) and splitting (cut operation) data sets in $O(\log(n))$ amortized time, as well as finding which tree an element belongs to in $O(\log(n))$ amortized time. In the original publication, Sleator and Tarjan referred to link/cut trees as "dynamic trees."

See also

- Splay tree

External links

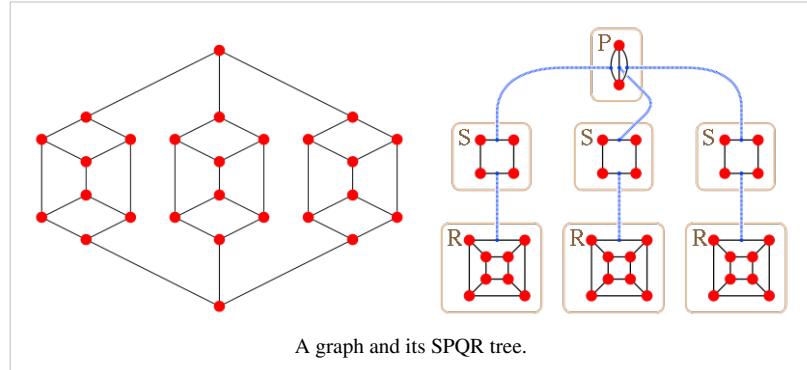
- "A Data Structure for Dynamic Trees", Sleator and Tarjan (the original publication) ^[1]
- "Self-adjusting Binary Search Trees", Sleator and Tarjan ^[11]
- "Finding Minimum-Cost Circulations by Canceling Negative Cycles", Goldberg and Tarjan (application to min-cost circulation) ^[2]

References

- [1] <http://www.cs.cmu.edu/~sleator/papers/dynamic-trees.pdf>
[2] <http://portal.acm.org/citation.cfm?id=76368>

SPQR tree

In graph theory, a branch of mathematics, the **triconnected components** of a biconnected graph are a system of smaller graphs that describe all of the the 2-vertex cuts in the graph. An **SPQR tree** is a tree data structure used in computer science, and more specifically graph algorithms, to represent the triconnected components of a graph. The SPQR tree of a graph may be constructed in linear time^[1] and has several applications in dynamic graph algorithms and graph drawing.



The basic structures underlying the SPQR tree, the triconnected components of a graph, and the connection between this decomposition and the planar embeddings of a planar graph, were first investigated by Saunders Mac Lane (1937); these structures were used in efficient algorithms by several other researchers^[2] prior to their formalization as the SPQR tree by Di Battista and Tamassia (1989, 1990, 1996).

Structure

An SPQR tree takes the form of an unrooted tree in which for each node x there is associated an undirected graph or multigraph G_x . The node, and the graph associated with it, may have one of four types, given the initials SPQR:

- In an S node, the associated graph is a cycle graph with three or more vertices and edges. This case is analogous to series composition in series-parallel graphs; the S stands for "series".^[3]
- In a P node, the associated graph is a dipole graph, a multigraph with two vertices and three or more edges, the planar dual to a cycle graph. This case is analogous to parallel composition in series-parallel graphs; the P stands for "parallel".^[3]
- In a Q node, the associated graph has a single edge. This trivial case is necessary to handle the graph that has only one edge, but does not appear in the SPQR trees of more complex graphs.
- In an R node, the associated graph is a 3-connected graph that is not a cycle or dipole. The R stands for "rigid": in the application of SPQR trees in planar graph embedding, the associated graph of an R node has a unique planar embedding.^[3]

Each edge xy between two nodes of the SPQR tree is associated with two directed *virtual edges*, one of which is an edge in G_x and the other of which is an edge in G_y . Each edge in a graph G_x may be a virtual edge for at most one SPQR tree edge.

An SPQR tree T represents a 2-connected graph G_T , formed as follows. Whenever SPQR tree edge xy associates the virtual edge ab of G_x with the virtual edge cd of G_y , form a single larger graph by merging a and c into a single supervertex, merging b and d into another single supervertex, and deleting the two virtual edges. That is, the larger graph is the 2-clique-sum of G_x and G_y . Performing this gluing step on each edge of the SPQR tree produces the graph G_T ; the order of performing the gluing steps does not affect the result. Each vertex in one of the graphs G_x may be associated in this way with a unique vertex in G_T the supervertex into which it was merged.

Typically, it is not allowed within an SPQR tree for two S nodes to be adjacent, nor for two P nodes to be adjacent, because if such an adjacency occurred the two nodes could be merged together into a single larger node. With this assumption, the SPQR tree is uniquely determined from its graph. When a graph G is represented by an SPQR tree with no adjacent P nodes and no adjacent S nodes, then the graphs G_x associated with the nodes of the SPQR tree are known as the triconnected components of G .

Finding 2-vertex cuts

With the SPQR tree of a graph G (without Q nodes) it is straightforward to find every pair of vertices u and v in G such that removing u and v from G leaves a disconnected graph, and the connected components of the remaining graphs:

- The two vertices u and v may be the two endpoints of a virtual edge in the graph associated with an R node, in which case the two components are represented by the two subtrees of the SPQR tree formed by removing the corresponding SPQR tree edge.
- The two vertices u and v may be the two vertices in the graph associated with a P node that has two or more virtual edges. In this case the components formed by the removal of u and v are represented by subtrees of the SPQR tree, one for each virtual edge in the node.
- The two vertices u and v may be two vertices in the graph associated with an S node such that either u and v are not adjacent, or the edge uv is virtual. If the edge is virtual, then the pair (u,v) also belongs to a node of type P and R and the components are as described above. If the two vertices are not adjacent then the two components are represented by two paths of the cycle graph associated with the S node and with the SPQR tree nodes attached to those two paths.

Embeddings of planar graphs

If a planar graph is 3-connected, it has a unique planar embedding up to the choice of which face is the outer face and of orientation of the embedding: the faces of the embedding are exactly the nonseparating cycles of the graph. However, for a planar graph (with labeled vertices and edges) that is 2-connected but not 3-connected, there may be greater freedom in finding a planar embedding. Specifically, whenever two nodes in the SPQR tree of the graph are connected by a pair of virtual edges, it is possible to flip the orientation of one of the nodes relative to the other one. Additionally, in a P node of the SPQR tree, the different parts of the graph connected to virtual edges of the P node may be arbitrarily permuted. All planar representations may be described in this way.

References

- Bienstock, Daniel; Monma, Clyde L. (1988), "On the complexity of covering vertices by faces in a planar graph", *SIAM Journal on Computing* **17** (1): 53–76, doi:10.1137/0217004.
- Di Battista, Giuseppe; Tamassia, Roberto (1989), "Incremental planarity testing", *Proc. 30th Annual Symposium on Foundations of Computer Science*, pp. 436–441, doi:10.1109/SFCS.1989.63515.
- Di Battista, Giuseppe; Tamassia, Roberto (1990), "On-line graph algorithms with SPQR-trees", *Proc. 17th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, **443**, Springer-Verlag, pp. 598–611, doi:10.1007/BFb0032061.
- Di Battista, Giuseppe; Tamassia, Roberto (1996), "On-line planarity testing" [4], *SIAM Journal on Computing* **25** (5): 956–997, doi:10.1137/S0097539794280736.
- Gutwenger, Carsten; Mutzel, Petra (2001), "A linear time implementation of SPQR-trees", *Proc. 8th International Symposium on Graph Drawing (GD 2000)*, Lecture Notes in Computer Science, **1984**, Springer-Verlag, pp. 77–90, doi:10.1007/3-540-44541-2_8.
- Hopcroft, John; Tarjan, Robert (1973), "Dividing a graph into triconnected components", *SIAM Journal on Computing* **2** (3): 135–158, doi:10.1137/0202012.
- Mac Lane, Saunders (1937), "A structural characterization of planar combinatorial graphs", *Duke Mathematical Journal* **3** (3): 460–472, doi:10.1215/S0012-7094-37-00336-3.

External links

- SQPR tree implementation [5] in the Open Graph Drawing Framework.

References

- [1] Hopcroft & Tarjan (1973); Gutwenger & Mutzel (2001).
- [2] E.g., Hopcroft & Tarjan (1973) and Bienstock & Monma (1988), both of which are cited as precedents by Di Battista and Tamassia.
- [3] Di Battista & Tamassia (1989).
- [4] <http://cs.brown.edu/research/pubs/pdfs/1996/DiBattista-1996-OPT.pdf>
- [5] http://www.ogdf.net/doc-ogdf/classogdf_1_1_s_p_q_r_tree.html

Spaghetti stack

A **spaghetti stack** (also called a **cactus stack** or **saguaro stack**) in computer science is an N-ary tree data structure in which child nodes have pointers to the parent nodes (but not vice-versa). When a list of nodes is traversed from a leaf node to the root node by chasing these parent pointers, the structure looks like a linked list stack.^[1] It can be analogized to a linked list having one and only parent pointer called "next" or "link", and ignoring that each parent may have other children (which are not accessible anyway since there are no downward pointers).

Spaghetti stack structures arise in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use.

For example, a compiler for a language such as C creates a spaghetti stack as it opens and closes symbol tables representing block scopes. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level "parent" symbol table and so on. Thus when the compiler is later performing translations over the abstract syntax tree, for any given expression, it can fetch the symbol table representing that expression's environment and can resolve references to identifiers. If the expression refers to a variable X, it is first sought after in the leaf symbol table representing the inner-most lexical scope, then in the parent and so on.

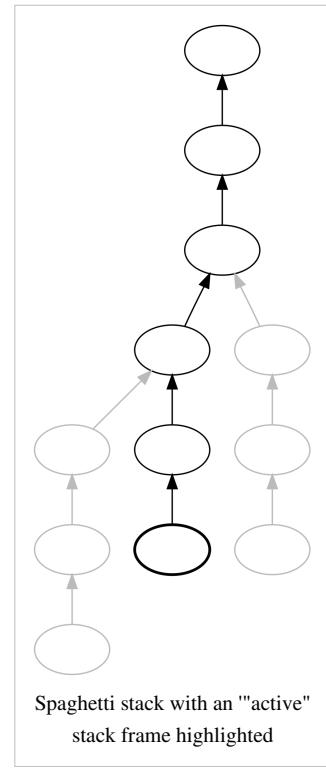
A similar data structure appears in disjoint-set forests, a type of disjoint-set data structure.

Use in programming language runtimes

The term *spaghetti stack* is closely associated with implementations of programming languages that support continuations. Spaghetti stacks are used to implement the actual run-time stack containing variable bindings and other environmental features. When continuations must be supported, a function's local variables cannot be destroyed when that function returns: a saved continuation may later re-enter into that function, and will expect not only the variables there to be intact, but it will also expect the entire stack to be there, so it can return again! To resolve this problem, stack frames can be dynamically allocated in a spaghetti stack structure, and simply left behind to be garbage collected when no continuations refer to them any longer. This type of structure also solves both the upward and downward funarg problems, so first-class lexical closures are readily implemented in that substrate also.

Examples of languages that use spaghetti stacks are:

- Languages having first-class continuations such as Scheme, Standard ML
- Languages where the execution stack can be inspected and modified at runtime such as Smalltalk
- The Felix programming language



Spaghetti stack with an "active" stack frame highlighted

See also

- Persistent data structure
- Segmented stack

References

- [1] Machinery, Sponsored (1988). *Proceedings of the 1988 Acm Conference on Lisp and Functional Programming*. New York: ACM Press.
ISBN 9780897912730.

Disjoint-set data structure

Given a set of elements, it is often useful to break them up or partition them into a number of separate, nonoverlapping sets. A **disjoint-set data structure** is a data structure that keeps track of such a partitioning. A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure:

- *Find*: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- *Union*: Combine or merge two sets into a single set.

Because it supports these two operations, a disjoint-set data structure is sometimes called a *union-find data structure* or *merge-find set*. The other important operation, *MakeSet*, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved (see the *Applications* section).

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find(x)* returns the representative of the set that *x* belongs to, and *Union* takes two set representatives as its arguments.

Disjoint-set linked lists

A simple approach to creating a disjoint-set data structure is to create a linked list for each set. The element at the head of each list is chosen as its representative.

MakeSet creates a list of one element. *Union* appends the two lists, a constant-time operation. The drawback of this implementation is that *Find* requires $\Omega(n)$ or linear time.

This can be avoided by including in each linked list node a pointer to the head of the list; then *Find* takes constant time. However, *Union* now has to update each element of the list being appended to make it point to the head of the new combined list, requiring $\Omega(n)$ time.

When the length of each list is tracked, the required time can be improved by always appending the smaller list to the longer. Using this *weighted-union heuristic*, a sequence of m *MakeSet*, *Union*, and *Find* operations on n elements requires $O(m + n \log n)$ time.^[1] For asymptotically faster operations, a different data structure is needed.

Disjoint-set forests

Disjoint-set forests are a data structure where each set is represented by a tree data structure, in which each node holds a reference to its parent node (see spaghetti stack). They were first described by Bernard A. Galler and Michael J. Fischer in 1964,^[2] although their precise analysis took years.

In a disjoint-set forest, the representative of each set is the root of that set's tree. *Find* follows parent nodes until it reaches the root. *Union* combines two trees into one by attaching the root of one to the root of the other. One way of implementing these might be:

```
function MakeSet (x)
    x.parent := x

function Find(x)
    if x.parent == x
        return x
    else
        return Find(x.parent)

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

In this naive form, this approach is no better than the linked-list approach, because the tree it creates can be highly unbalanced; however, it can be enhanced in two ways.

The first way, called *union by rank*, is to always attach the smaller tree to the root of the larger tree, rather than vice versa. Since it is the depth of the tree that affects the running time, the tree with smaller depth gets added under the root of the deeper tree, which only increases the depth if the depths were equal. In the context of this algorithm, the term *rank* is used instead of *depth* since it stops being equal to the depth if path compression (described below) is also used. One-element trees are defined to have a rank of zero, and whenever two trees of the same rank r are united, the rank of the result is $r+1$. Just applying this technique alone yields an amortized running-time of $O(\log n)$ per *MakeSet*, *Union*, or *Find* operation. Pseudocode for the improved *MakeSet* and *Union*:

```
function MakeSet (x)
    x.parent := x
    x.rank    := 0

function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    if xRoot.rank > yRoot.rank
        yRoot.parent := xRoot
    else if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
    else if xRoot != yRoot // Unless x and y are already in same set,
merge them
        yRoot.parent := xRoot
        xRoot.rank := xRoot.rank + 1
```

The second improvement, called *path compression*, is a way of flattening the structure of the tree whenever *Find* is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root

node; they all share the same representative. To effect this, as *Find* recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly. Here is the improved *Find*:

```
function Find(x)
    if x.parent == x
        return x
    else
        x.parent := Find(x.parent)
        return x.parent
```

These two techniques complement each other; applied together, the amortized time per operation is only $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the function $f(n) = A(n, n)$, and A is the extremely quickly-growing Ackermann function. Since $\alpha(n)$ is the inverse of this function, $\alpha(n)$ is less than 5 for all remotely practical values of n . Thus, the amortized running time per operation is effectively a small constant. In fact, this is asymptotically optimal: Fredman and Saks showed in 1989 that $\Omega(\alpha(n))$ words must be accessed by any disjoint-set data structure per operation on average.^[3]

Applications

Disjoint-set data structures model the partitioning of a set, for example to keep track of the connected components of an undirected graph. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle.

This data structure is used by the Boost Graph Library to implement its Incremental Connected Components^[4] functionality. It is also used for implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

Note that the implementation as disjoint-set forests doesn't allow deletion of edges—even without path compression or the rank heuristic—this is not as easy, although more complex schemes have been designed^[5] that can deal with this type of incremental update.

History

While the ideas used in disjoint-set forests have long been familiar, Robert Tarjan was the first to prove the upper bound (and a restricted version of the lower bound) in terms of the inverse Ackermann function, in 1975.^[6] Until this time the best bound on the time per operation, proven by Hopcroft and Ullman,^[7] was $O(\log^* n)$, the iterated logarithm of n , another slowly-growing function (but not quite as slow as the inverse Ackermann function).

Tarjan and van Leeuwen also developed one-pass *Find* algorithms that are more efficient in practice while retaining the same worst-case complexity.^[8]

In 2007, Sylvain Conchon and Jean-Christophe Filliâtre developed a persistent version of the disjoint-set forest data structure, allowing previous versions of the structure to be efficiently retained, and formalized its correctness using the proof assistant Coq.^[9]

External links

- C++ and C# implementations^[10], by Emil Stefanov
- C++ implementation^[11], part of the Boost C++ libraries
- Java implementation^[12], by Mark Allen Weiss
- Java applet: A Graphical Union-Find Implementation^[13], by Rory L. P. McGuire
- *Wait-free Parallel Algorithms for the Union-Find Problem*^[14], a 1994 paper by Richard J. Anderson and Heather Woll describing a parallelized version of Union-Find that never needs to block
- Python implementation^[15]

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 21: Data structures for Disjoint Sets, pp. 498–524.
- [2] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, Volume 7, Issue 5 (May 1964), pp. 301–303. The paper originating disjoint-set forests. ACM Digital Library (<http://portal.acm.org/citation.cfm?doid=364099.364331>)
- [3] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pp. 345–354. May 1989. "Theorem 5: Any CPROBE($\log n$) implementation of the set union problem requires $\Omega(m \alpha(m, n))$ time to execute m Find's and $n-1$ Union's, beginning with n singleton sets."
- [4] http://www.boost.org/libs/graph/doc/incremental_components.html
- [5] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems, *ACM Computing Surveys*, Volume 23, Issue 3 (September 1991), pp. 319–344. ACM Digital Library (<http://portal.acm.org/citation.cfm?id=116878>)
- [6] Tarjan, Robert Endre (1975). "Efficiency of a Good But Not Linear Set Union Algorithm" (<http://portal.acm.org/citation.cfm?id=321884>). *Journal of the ACM* **22** (2): 215–225. doi:10.1145/321879.321884. .
- [7] Hopcroft, J. E.; Ullman, J. D. (1973). "Set Merging Algorithms" (<http://dx.doi.org/10.1137/0202024>). *SIAM Journal on Computing* **2** (4): 294–303. doi:10.1137/0202024. .
- [8] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.
- [9] Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In ACM SIGPLAN Workshop on ML, Freiburg, Germany, October 2007.
- [10] <http://www.emilstefanov.net/Projects/DisjointSets.aspx>
- [11] http://www.boost.org/libs/disjoint_sets/disjoint_sets.html
- [12] <http://www.cs.berkeley.edu/~jrs/61bs00/hw/hw9/set/DisjointSets.java>
- [13] <http://www.cs.unm.edu/~rlpm/499/uf.html>
- [14] <http://citeseer.ist.psu.edu/anderson94waitfree.html>
- [15] <http://code.activestate.com/recipes/215912-union-find-data-structure/>

Space-partitioning trees

Space partitioning

In mathematics, **space partitioning** is the process of dividing a space (usually a Euclidean space) into two or more disjoint subsets (see also partition of a set). In other words, space partitioning divides a space into non-overlapping regions. Any point in the space can then be identified to lie in exactly one of the regions.

Space-partitioning systems are often hierarchical, meaning that a space (or a region of space) is divided into several regions, and then the same space-partitioning system is recursively applied to each of the regions thus created. The regions can be organized into a tree, called a **space-partitioning tree**.

Most space-partitioning systems use planes (or, in higher dimensions, hyperplanes) to divide space: points on one side of the plane form one region, and points on the other side form another. Points exactly on the plane are usually arbitrarily assigned to one or the other side. Recursively partitioning space using planes in this way produces a BSP tree, one of the most common forms of space partitioning.

Space partitioning is particularly important in computer graphics, where it is frequently used to organize the objects in a virtual scene. Storing objects in a space-partitioning data structure makes it easy and fast to perform certain kinds of geometry queries — for example, determining whether two objects are close to each other in collision detection, or determining whether a ray intersects an object in ray tracing (Ray Tracing - Auxiliary Data Structures [1]).

In integrated circuit design, an important step is design rule check. This step ensures that the completed design is manufacturable. The check involves rules that specify widths and spacings and other geometry patterns. A modern design can have billions of polygons that represent wires and transistors. Efficient checking relies heavily on geometry query. For example, a rule may specify that any polygon must be at least n nanometers from any other polygon. This is converted into a geometry query by enlarging a polygon by n at all sides and query to find all intersecting polygons.

Common space partitioning systems include:

- BSP trees
- Quadtrees
- Octrees
- kd-trees
- Bins
- R-trees
- Bounding volume hierarchies
- SEADSS

Spatial Partitioning Examples:

References

[1] <http://undergraduate.csse.uwa.edu.au/units/CITS4241/Handouts/Lecture14.html>

Binary space partitioning

Binary space partitioning (BSP) is a method for recursively subdividing a space into convex sets by hyperplanes. This subdivision gives rise to a representation of the scene by means of a tree data structure known as a **BSP tree**.

Originally, this approach was proposed in 3D computer graphics to increase the rendering efficiency. Some other applications include performing geometrical operations with shapes (constructive solid geometry) in CAD, collision detection in robotics and 3D computer games, and other computer applications that involve handling of complex spatial scenes.

Overview

In computer graphics it is desirable that the drawing of a scene be both correct and quick. A simple way to draw a scene is the painter's algorithm: draw it from back to front painting the background over with each closer object. However, that approach is quite limited since time is wasted drawing objects that will be overdrawn later, and not all objects will be drawn correctly.

Z-buffering can ensure that scenes are drawn correctly and eliminate the ordering step of the painter's algorithm, but it is expensive in terms of memory use. BSP trees will split up objects so that the painter's algorithm will draw them correctly without need of a Z-buffer and eliminate the need to sort the objects; as a simple tree traversal will yield them in the correct order. It also serves as base for other algorithms, such as visibility lists, which seek to reduce overdraw.

The downside is the requirement for a time consuming pre-processing of the scene, which makes it difficult and inefficient to directly implement moving objects into a BSP tree. This is often overcome by using the BSP tree together with a Z-buffer, and using the Z-buffer to correctly merge movable objects such as doors and monsters onto the background scene.

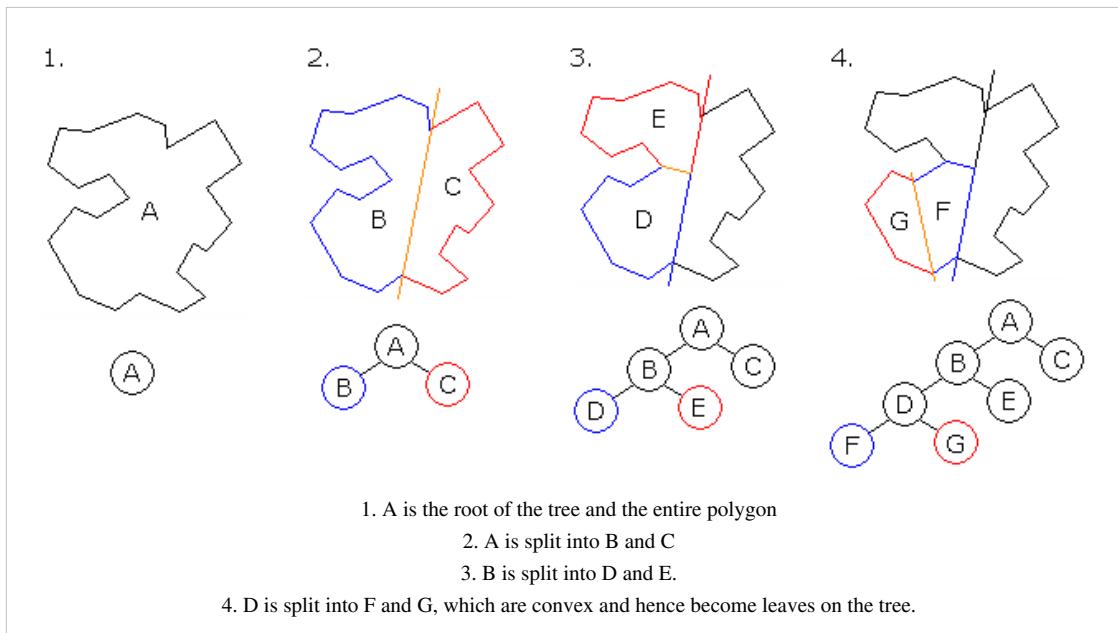
BSP trees are often used by 3D computer games, particularly first-person shooters and those with indoor environments. Probably the earliest game to use a BSP data structure was *Doom* (see *Doom* engine for an in-depth look at *Doom*'s BSP implementation). Other uses include ray tracing and collision detection.

Generation

Binary space partitioning is a generic process of recursively dividing a scene into two until the partitioning satisfies one or more requirements. The specific method of division varies depending on its final purpose. For instance, in a BSP tree used for collision detection, the original object would be partitioned until each part becomes simple enough to be individually tested, and in rendering it is desirable that each part be convex so that the painter's algorithm can be used.

The final number of objects will inevitably increase since lines or faces that cross the partitioning plane must be split into two, and it is also desirable that the final tree remains reasonably balanced. Therefore the algorithm for correctly and efficiently creating a good BSP tree is the most difficult part of an implementation. In 3D space, planes are used to partition and split an object's faces; in 2D space lines split an object's segments.

The following picture illustrates the process of partitioning an irregular polygon into a series of convex ones. Notice how each step produces polygons with fewer segments until arriving at G and F, which are convex and require no further partitioning. In this particular case, the partitioning line was picked between existing vertices of the polygon and intersected none of its segments. If the partitioning line intersects a segment, or face in a 3D model, the offending segment(s) or face(s) have to be split into two at the line/plane because each resulting partition must be a full, independent object.



Since the usefulness of a BSP tree depends upon how well it was generated, a good algorithm is essential. Most algorithms will test many possibilities for each partition until finding a good compromise and might also keep backtracking information in memory so that if a branch of the tree is found to be unsatisfactory other alternative partitions may be tried. Therefore producing a tree usually requires long computations.

BSP trees were also used to represent natural images. Construction methods of BSP trees of images were first introduced as efficient representations, where only a few hundred nodes can represent an image that normally require hundreds-of-thousands of pixels. Fast algorithms were also developed to construct BSP trees of images using computer vision and signal processing algorithms. These algorithms in conjunction with advanced entropy coding and signal approximation approaches were used to develop image compression methods.

Rendering a scene with visibility information from the BSP tree

BSP trees are used to improve rendering performance in calculating visible triangles for the painter's algorithm for instance. The tree can be traversed in linear time from an arbitrary viewpoint.

Since a painter's algorithm works by drawing polygons farthest from the eye first, the following code recurses to the bottom of the tree and draws the polygons. As the recursion unwinds, polygons closer to the eye are drawn over far polygons. Because the BSP tree already splits polygons into trivial pieces, the hardest part of the painter's algorithm is already solved - code for back to front tree traversal. [1]

```
traverse_tree(bsp_tree* tree, point eye)
{
    location = tree->find_location(eye);

    if(tree->empty())
        return;

    if(location > 0)          // if eye in front of location
    {
        traverse_tree(tree->back, eye);
        display(tree->polygon_list);
```

```

        traverse_tree(tree->front, eye);
    }
    else if(location < 0) // eye behind location
    {
        traverse_tree(tree->front, eye);
        display(tree->polygon_list);
        traverse_tree(tree->back, eye);
    }
    else // eye coincidental with partition hyperplane
    {
        traverse_tree(tree->front, eye);
        traverse_tree(tree->back, eye);
    }
}

```

Other space partitioning structures

BSP trees divide a region of space into two subregions at each node. They are related to quadtrees and octrees, which divide each region into four or eight subregions, respectively.

Relationship Table

Name	p	s
Binary Space Partition	1	2
Quadtree	2	4
Octree	3	8

where p is the number of dividing planes used, and s is the number of subregions formed.

BSP trees can be used in spaces with any number of dimensions, but quadtrees and octrees are most useful in subdividing 2- and 3-dimensional spaces, respectively. Another kind of tree that behaves somewhat like a quadtree or octree, but is useful in any number of dimensions, is the *kd-tree*.

Timeline

- 1969 Schumacker et al published a report that described how carefully positioned planes in a virtual environment could be used to accelerate polygon ordering. The technique made use of depth coherence, which states that a polygon on the far side of the plane cannot, in any way, obstruct a closer polygon. This was used in flight simulators made by GE as well as Evans and Sutherland. However, creation of the polygonal data organization was performed manually by scene designer.
- 1980 Fuchs et al. [FUCH80] extended Schumacker's idea to the representation of 3D objects in a virtual environment by using planes that lie coincident with polygons to recursively partition the 3D space. This provided a fully automated and algorithmic generation of a hierarchical polygonal data structure known as a Binary Space Partitioning Tree (BSP Tree). The process took place as an off-line preprocessing step that was performed once per environment/object. At run-time, the view-dependent visibility ordering was generated by traversing the tree.
- 1981 Naylor's Ph.D thesis containing a full development of both BSP trees and a graph-theoretic approach using strongly connected components for pre-computing visibility, as well as the connection between the two methods. BSP trees as a dimension independent spatial search structure was emphasized, with applications to visible surface determination. The thesis also included the first empirical data demonstrating that the size of the tree and

the number of new polygons was reasonable (using a model of the Space Shuttle).

- 1983 Fuchs et al. describe a micro-code implementation of the BSP tree algorithm on an Ikonas frame buffer system. This was the first demonstration of real-time visible surface determination using BSP trees.
- 1987 Thibault and Naylor described how arbitrary polyhedra may be represented using a BSP tree as opposed to the traditional b-rep (boundary representation). This provided a solid representation vs. a surface based-representation. Set operations on polyhedra were described using a tool, enabling Constructive Solid Geometry (CSG) in real-time. This was the fore runner of BSP level design using brushes, introduced in the Quake editor and picked up in the Unreal Editor.
- 1990 Naylor, Amanatides, and Thibault provide an algorithm for merging two bsp trees to form a new bsp tree from the two original trees. This provides many benefits including: combining moving objects represented by BSP trees with a static environment (also represented by a BSP tree), very efficient CSG operations on polyhedra, exact collisions detection in $O(\log n * \log n)$, and proper ordering of transparent surfaces contained in two interpenetrating objects (has been used for an x-ray vision effect).
- 1990 Teller and Séquin proposed the offline generation of potentially visible sets to accelerate visible surface determination in orthogonal 2D environments.
- 1991 Gordon and Chen [CHEN91] described an efficient method of performing front-to-back rendering from a BSP tree, rather than the traditional back-to-front approach. They utilised a special data structure to record, efficiently, parts of the screen that have been drawn, and those yet to be rendered. This algorithm, together with the description of BSP Trees in the standard computer graphics textbook of the day (Foley, Van Dam, Feiner and Hughes) was used by John Carmack in the making of *Doom*.
- 1992 Teller's PhD thesis described the efficient generation of potentially visible sets as a pre-processing step to acceleration real-time visible surface determination in arbitrary 3D polygonal environments. This was used in Quake and contributed significantly to that game's performance.
- 1993 Naylor answers the question of what characterizes a good bsp tree. He used expected case models (rather than worst case analysis) to mathematically measure the expected cost of searching a tree and used this measure to build good BSP trees. Intuitively, the tree represents an object in a multi-resolution fashion (more exactly, as a tree of approximations). Parallels with Huffman codes and probabilistic binary search trees are drawn.
- 1993 Hayder Radha's PhD thesis described (natural) image representation methods using BSP trees. This includes the development of an optimal BSP-tree construction framework for any arbitrary input image. This framework is based on a new image transform, known as the Least-Square-Error (LSE) Partitioning Line (LPE) transform. H. Radha' thesis also developed an optimal rate-distortion (RD) image compression framework and image manipulation approaches using BSP trees.

References

- [FUCH80] H. Fuchs, Z. M. Kedem and B. F. Naylor. "On Visible Surface Generation by A Priori Tree Structures." ACM Computer Graphics, pp 124–133. July 1980.
- [THIBAULT87] W. Thibault and B. Naylor, "Set Operations on Polyhedra Using Binary Space Partitioning Trees", Computer Graphics (Siggraph '87), 21(4), 1987.
- [NAYLOR90] B. Naylor, J. Amanatides, and W. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations", Computer Graphics (Siggraph '90), 24(3), 1990.
- [NAYLOR93] B. Naylor, "Constructing Good Partitioning Trees", Graphics Interface (annual Canadian CG conference) May, 1993.
- [CHEN91] S. Chen and D. Gordon. "Front-to-Back Display of BSP Trees." IEEE Computer Graphics & Algorithms, pp 79–85. September 1991.

- [RADHA91] H. Radha, R. Leonardi, M. Vetterli, and B. Naylor "Binary Space Partitioning Tree Representation of Images," Journal of Visual Communications and Image Processing 1991, vol. 2(3).
- [RADHA93] H. Radha, "Efficient Image Representation using Binary Space Partitioning Trees.", Ph.D. Thesis, Columbia University, 1993.
- [RADHA96] H. Radha, M. Vetterli, and R. Leonardi, "Image Compression Using Binary Space Partitioning Trees," IEEE Transactions on Image Processing, vol. 5, No.12, December 1996, pp. 1610-1624. and then there was GOD
- [WINTER99] AN INVESTIGATION INTO REAL-TIME 3D POLYGON RENDERING USING BSP TREES. Andrew Steven Winter. April 1999. available online
- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf (2000). *Computational Geometry* (2nd revised edition ed.). Springer-Verlag. ISBN 3-540-65620-0. Section 12: Binary Space Partitions: pp.251–265. Describes a randomized Painter's Algorithm.
- Christer Ericson: *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Verlag Morgan Kaufmann, S. 349-382, Jahr 2005, ISBN 1-55860-732-3

[1] Binary Space Partition Trees in 3d worlds (<http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/document.html>)

External links

- BSP trees presentation (<http://www.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html>)
- Another BSP trees presentation (http://www.cc.gatech.edu/classes/AY2004/cs4451a_fall/bsp.pdf) (PDF)
- A Java applet which demonstrates the process of tree generation (<http://symbolcraft.com/graphics/bsp/>)
- A Master Thesis about BSP generating (<http://www.gamedev.net/reference/programming/features/bsptree/bsp.pdf>)
- BSP Trees: Theory and Implementation (<http://www.devmaster.net/articles/bsp-trees/>)
- BSP in 3D space (<http://www.euclideanspace.com/threed/solidmodel/spatialdecomposition/bsp/index.htm>)

Segment tree

In computer science, a **segment tree** is a tree data structure for storing intervals, or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, its content cannot be modified once the structure is built. A similar data structure is the interval tree.

A segment tree for a set I of n intervals uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time. Segment trees support searching for all the intervals that contain a query point in $O(\log n + k)$, k being the number of retrieved intervals or segments^[1]. In the common word-level RAM model, the query time is not optimal, since $O(\sqrt{\log n} / \log \log n + k)$ for a query can be achieved by a predecessor search combined with the data structure in^[1]. If the range of all interval end points is in $\{1, \dots, O(n)\}$, a simple data structure with preprocessing time $O(n)$ and query time $O(1+k)$ exists^[1].

Applications of the segment tree are in the areas of computational geometry, and geographic information systems. The segment tree can be generalized to higher dimension spaces as well.

Structure description

This section describes the structure of a segment tree in a one-dimensional space.

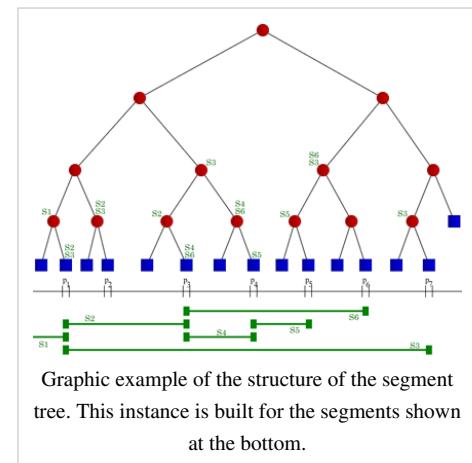
Let S be a set of intervals, or segments. Let p_1, p_2, \dots, p_m be the list of distinct interval endpoints, sorted from left to right. Consider the partitioning of the real line induced by those points. The regions of this partitioning are called *elementary intervals*. Thus, the elementary intervals are, from left to right:

$$(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \dots, (p_{m-1}, p_m), [p_m, p_m], (p_m, +\infty)$$

That is, the list of elementary intervals consists of open intervals between two consecutive endpoints p_i and p_{i+1} , alternated with closed intervals consisting of a single endpoint. Single points are treated themselves as intervals because the answer to a query is not necessarily the same at the interior of an elementary interval and its endpoints^[2].

Given a set I of intervals, or segments, a segment tree T for I is structured as follows:

- T is a binary tree.
- Its leafs correspond to the elementary intervals induced by the endpoints in I , in an ordered way: the leftmost leaf corresponds to the leftmost interval, and so on. The elementary intervals corresponding to a leaf v is denoted $\text{Int}(v)$.
- The internal nodes of T corresponds to intervals that are the union of elementary intervals: the interval $\text{Int}(N)$ corresponding to node N is the union of the intervals corresponding to the leafs of the tree rooted at N . That implies that $\text{Int}(N)$ is the union of the intervals of its two children.
- Each node or leaf v in T stores the interval $\text{Int}(v)$ and a set of intervals, in some data structure. This canonical subset of node v contains the intervals $[x, x']$ from I such that $[x, x']$ contains $\text{Int}(v)$ and does not contain $\text{Int}(\text{parent}(v))$. That is, each segment in I stores the segments that span through its interval, but does not span through the interval of any parent^[3].



Storage requirements

This section analyzes the storage cost of a segment tree in a one-dimensional space.

A segment tree T on a set I of n intervals uses $O(n\log n)$ storage.

Proof:

Lemma: Any interval $[x, x']$ of I is stored in the canonical set for at most two nodes at the same depth.

Proof: Let v_1, v_2, v_3 be the three nodes at the same depth, numbered from left to right; and let w be the parent node of v_2 . Suppose $[x, x']$ is stored at v_1 and v_3 . This means that $[x, x']$ spans the whole interval from the left endpoint of $\text{Int}(v_1)$ to the right endpoint of $\text{Int}(v_3)$. Because v_2 lies between v_1 and v_3 , $\text{Int}(w)$ must be contained in $[x, x']$. Hence, $[x, x']$ will not be stored at v_2 .

The set I has at most $4n + 1$ elementary intervals. Because T is a binary balanced tree with at most $4n + 1$ leaves, its height is $O(\log n)$. Since any interval is stored at most twice at a given depth of the tree, that the total amount of storage is $O(n\log n)$ ^[4].

Construction

This section describes the construction of a segment tree in a one-dimensional space.

A segment tree from the set of segments I , can be built as follows. First, the endpoints of the intervals in I are sorted. The elementary intervals are obtained from that. Then, a balanced binary tree is built on the elementary intervals, and for each node v it is determined the interval $\text{Int}(v)$ it represents. It remains to compute the canonical subsets for the nodes. To achieve this, the intervals in I are inserted one by one into the segment tree. An interval $X = [x, x']$ can be inserted in a subtree rooted at T , using the following procedure^[5]:

- If $\text{Int}(T)$ is contained in X then store X at T , and finish.
- Else:
 - If X intersects the canonical subset of the left child of T , then insert X in that child, recursively.
 - If X intersects the canonical subset of the right child of T , then insert X in that child, recursively.

The complete construction operation takes $O(n\log n)$ time, being n the amount of segments in I .

Proof

Sorting the endpoints takes $O(n\log n)$. Building a balanced binary tree from the sorted endpoints, takes linear time on n .

The insertion of an interval $X = [x, x']$ into the tree, costs $O(\log n)$.

Proof: Visiting every node takes constant time (assuming that canonical subsets are stored in a simple data structure like a linked list). When we visit node v , we either store X at v , or $\text{Int}(v)$ contains an endpoint of X . As proved above, an interval is stored at most twice at each level of the tree. There is also at most one node at every level whose corresponding interval contains x , and one node whose interval contains x' . So, at most four nodes per level are visited. Since there are $O(\log n)$ levels, the total cost of the insertion is $O(\log n)$ ^[1].

Query

This section describes the query operation of a segment tree in a one-dimensional space.

A query for a segment tree, receives a point q_x , and retrieves a list of all the segments stored which contain the point q_x .

Formally stated; given a node (subtree) v and a query point q_x , the query can be done using the following algorithm:

- Report all the intervals in $I(v)$.
- If v is not a leaf:
 - If q_x is in $\text{Int}(\text{left child of } v)$ then
 - Perform a query in the left child of v .
 - Else
 - Perform a query in the right child of v .

In a segment tree that contains n intervals, those containing a given query point can be reported in $O(\log n + k)$ time, where k is the number of reported intervals.

Proof: The query algorithm visits one node per level of the tree, so $O(\log n)$ nodes in total. In the other hand, at a node v , the segments in I are reported in $O(1 + k_v)$ time, where k_v is the number of intervals at node v , reported. The sum of all the k_v for all nodes v visited, is k , the number of reported segments.^[4].

Generalization for higher dimensions

The segment tree can be generalized to higher dimension spaces, in the form of multi-level segment trees. In higher dimension versions, the segment tree stores a collection of axis-parallel (hyper-)rectangles, and can retrieve the rectangles that contain a given query point. The structure uses $O(n \log^{d-1} n)$ storage, and answers queries in $O(\log^d n)$.

The use of fractional cascading lowers the query time bound by a logarithmic factor. The use of the interval tree on the deepest level of associated structures lowers the storage bound with a logarithmic factor.^[6]

Notes

The query that asks for all the intervals containing a given point, is often referred as *stabbing query*^[1].

The segment tree is less efficient than the interval tree for range queries in one dimension, due to its higher storage requirement: $O(n \log n)$ against the $O(n)$ of the interval tree. The importance of the segment tree is that the segments within each node's canonical subset can be stored in any arbitrary manner^[7].

Another plus of the segment tree, is that it can easily be adapted to counting queries; that is, report the number of segments containing a given point, instead of reporting the segments themselves. Instead of storing the intervals in the canonical subsets, it can simply be stored an integer representing their amount. Such a segment tree uses linear storage, and requires an $O(\log n)$ query time.^[8]

A version for higher dimensions of the interval tree and the priority search tree does not exist, that is, there is no clear extension of these structures that solves the analogous problem in higher dimensions. But the structures can be used as associated structure of segment trees.^[6]

History

The segment tree was discovered by J. L. Bentley in 1977; in "Solutions to Klee's rectangle problems" [7].

References

- [1] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, p. 227)
- [2] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, p. 224)
- [3] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, pp. 225–226)
- [4] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, p. 226)
- [5] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, pp. 226–227)
- [6] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, p. 230)
- [7] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, p. 229)
- [8] (de Berg, van Kreveld, Overmars, Schwarzkopf 2000, pp. 229–230)

-
- de Berg, Mark; van Kreveld, Marc; Overmars, Mark; Schwarzkopf, Otfried (2000), *Computational Geometry: algorithms and applications* (2nd ed.), Springer-Verlag Berlin Heidelberg New York, ISBN 3-540-65620-0
 - [1] Jens M. Schmidt. *Interval Stabbing Problems in Small Integer Ranges*. ISAAC'09, 2009, pp. 163-172.

Interval tree

In computer science, an **interval tree** is an ordered tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. It is often used for windowing queries, for example, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene.

The trivial solution is to visit each interval and test whether it intersects the given point or interval, which requires $\Theta(n)$ time, where n is the number of intervals in the collection. Since a query may return all intervals, for example if the query is a large interval intersecting all intervals in the collection, this is asymptotically optimal; however, we can do better by considering output-sensitive algorithms, where the runtime is expressed in terms of m , the number of intervals produced by the query. Interval trees are dynamic, i.e., they allow to insert or delete intervals. They obtain a query time of $O(\log n)$ while the preprocessing time to construct the data structure is $O(n \log n)$ (but the space consumption is $O(n)$). Data structures with better query times and preprocessing time exists for the static setting: If the unit-cost word RAM is the model of computation, $O(\sqrt{\log n / \log \log n} + m)$ can be obtained for each query. If the range of interval end points is in $\{1, \dots, O(n)\}$, a preprocessing time of $O(n)$ with a query time of $O(1+m)$ is possible [Schmidt].

Naive approach

In a simple case, the intervals do not overlap and they can be inserted into a simple binary tree and queried in $O(\log n)$ time. However, with arbitrarily overlapping intervals, there is no way to compare two intervals for insertion into the tree since orderings sorted by the beginning points or the ending points may be different. A naive approach might be to build two parallel trees, one ordered by the beginning point, and one ordered by the ending point of each interval. This allows discarding half of each tree in $O(\log n)$ time, but the results must be merged, requiring $O(n)$ time. This gives us queries in $O(n + \log n) = O(n)$, which is no better than brute-force.

Interval trees solve this problem. This article describes two alternative designs for an interval tree, dubbed the *centered interval tree* and the *augmented tree*.

Centered interval tree

Queries require $O(\log n + m)$ time, with n being the total number of intervals and m being the number of reported results. Construction requires $O(n \log n)$ time, and storage requires $O(n)$ space.

Construction

Given a set of n intervals on the number line, we want to construct a data structure so that we can efficiently retrieve all intervals overlapping another interval or point.

We start by taking the entire range of all the intervals and dividing it in half at x_{center} (in practice, x_{center} should be picked to keep the tree relatively balanced). This gives three sets of intervals, those completely to the left of x_{center} which we'll call S_{left} , those completely to the right of x_{center} which we'll call S_{right} , and those overlapping x_{center} which we'll call S_{center} .

The intervals in S_{left} and S_{right} are recursively divided in the same manner until there are no intervals left.

The intervals in S_{center} that overlap the center point are stored in a separate data structure linked to the node in the interval tree. This data structure consists of two lists, one containing all the intervals sorted by their beginning points, and another containing all the intervals sorted by their ending points.

The result is a binary tree with each node storing:

- A center point
- A pointer to another node containing all intervals completely to the left of the center point
- A pointer to another node containing all intervals completely to the right of the center point
- All intervals overlapping the center point sorted by their beginning point
- All intervals overlapping the center point sorted by their ending point

Intersecting

Given the data structure constructed above, we receive queries consisting of ranges or points, and return all the ranges in the original set overlapping this input.

With an Interval

First, we can reduce the case where an interval R is given as input to the simpler case where a single point is given as input. We first find all ranges with beginning or end points inside the input interval R using a separately constructed tree. In the one-dimensional case, we can use a simple tree containing all the beginning and ending points in the interval set, each with a pointer to its corresponding interval.

A binary search in $O(\log n)$ time for the beginning and end of R reveals the minimum and maximum points to consider. Each point within this range references an interval that overlaps our range and is added to the result list. Care must be taken to avoid duplicates, since an interval might begin and end within R . This can be done using a binary flag on each interval to mark whether or not it has been added to the result set.

The only intervals not yet considered are those overlapping R that do not have a point inside R , in other words, intervals that enclose it. To find these, we pick any point inside R and use the algorithm below to find all intervals intersecting that point (again, being careful to remove duplicates).

With a Point

The task is to find all intervals in the tree that overlap a given point x . The tree is walked with a similar recursive algorithm as would be used to traverse a traditional binary tree, but with extra affordance for the intervals overlapping the "center" point at each node.

For each tree node, x is compared to x_{center} , the midpoint used in node construction above. If x is less than x_{center} , the leftmost set of intervals, S_{left} , is considered. If x is greater than x_{center} , the rightmost set of intervals, S_{right} , is considered.

As each node is processed as we traverse the tree from the root to a leaf, the ranges in its S_{center} are processed. If x is less than x_{center} , we know that all intervals in S_{center} end after x , or they could not also overlap x_{center} . Therefore, we need only find those intervals in S_{center} that begin before x . We can consult the lists of S_{center} that have already been constructed. Since we only care about the interval beginnings in this scenario, we can consult the list sorted by beginnings. Suppose we find the closest number no greater than x in this list. All ranges from the beginning of the list to that found point overlap x because they begin before x and end after x (as we know because they overlap x_{center} which is larger than x). Thus, we can simply start enumerating intervals in the list until the endpoint value exceeds x .

Likewise, if x is greater than x_{center} , we know that all intervals in S_{center} must begin before x , so we find those intervals that end after x using the list sorted by interval endings.

If x exactly matches x_{center} , all intervals in S_{center} can be added to the results without further processing and tree traversal can be stopped.

Higher Dimensions

The interval tree data structure can be generalized to a higher dimension N with identical query and construction time and $O(n \log n)$ space.

First, a range tree in N dimensions is constructed that allows efficient retrieval of all intervals with beginning and end points inside the query region R . Once the corresponding ranges are found, the only thing that is left are those ranges that enclose the region in some dimension. To find these overlaps, N interval trees are created, and one axis intersecting R is queried for each. For example, in two dimensions, the bottom of the square R (or any other horizontal line intersecting R) would be queried against the interval tree constructed for the horizontal axis. Likewise, the left (or any other vertical line intersecting R) would be queried against the interval tree constructed on the vertical axis.

Each interval tree also needs an addition for higher dimensions. At each node we traverse in the tree, x is compared with S_{center} to find overlaps. Instead of two sorted lists of points as was used in the one-dimensional case, a range tree is constructed. This allows efficient retrieval of all points in S_{center} that overlap region R .

Deletion

If after deleting an interval from the tree, the node containing that interval contains no more intervals, that node may be deleted from the tree. This is more complex than a normal binary tree deletion operation.

An interval may overlap the center point of several nodes in the tree. Since each node stores the intervals that overlap it, with all intervals completely to the left of its center point in the left subtree, similarly for the right subtree, it follows that each interval is stored in the node closest to the root from the set of nodes whose center point it overlaps.

Normal deletion operations in a binary tree (for the case where the node being deleted has two children) involve promoting a node further from the root to the position of the node being deleted (usually the leftmost child of the right subtree, or the rightmost child of the left subtree). As a result of this promotion, some nodes that were above the promoted node will become descendants of it; it is necessary to search these nodes for intervals that also overlap the promoted node, and move those intervals into the promoted node. As a consequence, this may result in new

empty nodes, which must be deleted, following the same algorithm again.

Balancing

The same issues that affect deletion also affect rotation operations; rotation must preserve the invariant that intervals are stored as close to the root as possible.

Augmented tree

Another way to represent intervals is described in CLRS, Section 14.3: Interval trees, pp. 311–317.

Both insertion and deletion require $O(\log n)$ time, with n being the total number of intervals.

Use a simple ordered tree, for example a binary search tree or self-balancing binary search tree, where the tree is ordered by the 'low' values of the intervals, and an extra annotation is added to every node recording the maximum high value of both its subtrees. It is simple to maintain this attribute in only $O(h)$ steps during each addition or removal of a node, where h is the height of the node added or removed in the tree, by updating all ancestors of the node from the bottom up. Additionally, the tree rotations used during insertion and deletion may require updating the high value of the affected nodes.

Now, it's known that two intervals A and B overlap only when both $A.\text{low} \leq B.\text{high}$ and $A.\text{high} \geq B.\text{low}$. When searching the trees for nodes overlapping with a given interval, you can immediately skip:

- all nodes to the right of nodes whose low value is past the end of the given interval.
- all nodes that have their maximum 'high' value below the start of the given interval.

A total order can be defined on the intervals by ordering them first by their 'low' value and finally by their 'high' value. This ordering can be used to prevent duplicate intervals from being inserted into the tree in $O(\log n)$ time, versus the $O(k + \log n)$ time required to find duplicates if k intervals overlap a new interval.

Java Example: Adding a new interval to the tree

The key of each node is the interval itself and the value of each node is the end point of the interval:

```
public void add(Interval i) {
    put(i, i.getEnd());
}
```

Java Example: Searching a point or an interval in the tree

To search for an interval, you walk the tree, omitting those branches which can't contain what you're looking for. The simple case is looking for a point:

```
// Search for all intervals which contain "p", starting with the
// node "n" and adding matching intervals to the list "result"
public void search(IntervalNode n, Point p, List<Interval> result) {
    // Don't search nodes that don't exist
    if (n == null)
        return;

    // If p is to the right of the rightmost point of any interval
    // in this node and all children, there won't be any matches.
    if (p.compareTo(n.getValue()) > 0)
        return;
```

```

// Search left children
if (n.getLeft() != null)
    search(cast(n.getLeft()), p, result);

// Check this node
if (n.getKey().contains(p))
    result.add(n.getKey());

// If p is to the left of the start of this interval,
// then it can't be in any child to the right.
if (p.compareTo(n.getKey().getStart()) < 0)
    return;

// Otherwise, search right children
if (n.getRight() != null)
    search(cast(n.getRight()), p, result);
}

```

The code to search for an interval is exactly the same except for the check in the middle:

```

// Check this node
if (n.getKey().overlapsWith(i))
    result.add (n.getKey());

```

`overlapsWith()` is defined as:

```

public boolean overlapsWith(Interval other) {
    return start.compareTo(other.getEnd()) <= 0 &&
           end.compareTo(other.getStart()) >= 0;
}

```

Higher dimension

This can be extended to higher dimensions by cycling through the dimensions at each level of the tree. For example, for two dimensions, the odd levels of the tree might contain ranges for the x coordinate, while the even levels contain ranges for the y coordinate. However, it is not quite obvious how the rotation logic will have to be extended for such cases to keep the tree balanced.

A much simpler solution is to use nested interval trees. First, create a tree using the ranges for the y coordinate. Now, for each node in the tree, add another interval tree on the the x ranges, for all elements whose y range intersect that node's y range.

The advantage of this solution is that it can be extended to an arbitrary amount of dimensions using the same code base.

At first, the cost for the additional trees might seem prohibitive but that is usually not the case. As with the solution above, you need one node per x coordinate, so this cost is the same in both solutions. The only difference is that you need an additional tree structure per vertical interval. This structure is typically very small (a pointer to the root node plus maybe the number of nodes and the height of the tree).

References

- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*, Second Revised Edition. Springer-Verlag 2000. Section 10.1: Interval Trees, pp. 212–217.
- 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 (referred to by the abbreviation CLRS for the names of the authors)
- Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985
- Jens M. Schmidt. *Interval Stabbing Problems in Small Integer Ranges*. ISAAC'09, 2009, pp. 163-172.

External links

- CGAL : Computational Geometry Algorithms Library in C++^[1] contains a robust implementation of Range Trees

References

[1] <http://www.cgal.org/>

Range tree

In computer science, a **range tree** is an ordered tree data structure to hold a list of points. It allows all points within a given range to be efficiently retrieved, and is typically used in two or higher dimensions.

It is similar to a kd-tree except with faster query times of $O(\log^d n + k)$ but worse storage of $O(n \log^{(d-1)} n)$, with d being the dimension of the space, n being the number of points in the tree, and k being the number of points retrieved for a given query.

Range trees may be contrasted with interval trees: instead of storing points and allowing points in a given range to be retrieved efficiently, an interval tree stores intervals and allows the intervals containing a given point to be retrieved efficiently.

External links

- CGAL : Computational Geometry Algorithms Library in C++^[1] contains a robust implementation of Range Trees
- A C# Implementation of a Range Tree^[1]

References

- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*, Second Revised Edition. Springer-Verlag 2000. Section 5.3: Range Trees, pp.105-110.
- David M. Mount. *Lecture Notes: CMSC 754 Computational Geometry*^[2]. Lecture 23: Orthogonal Range Trees, pp. 102-104.

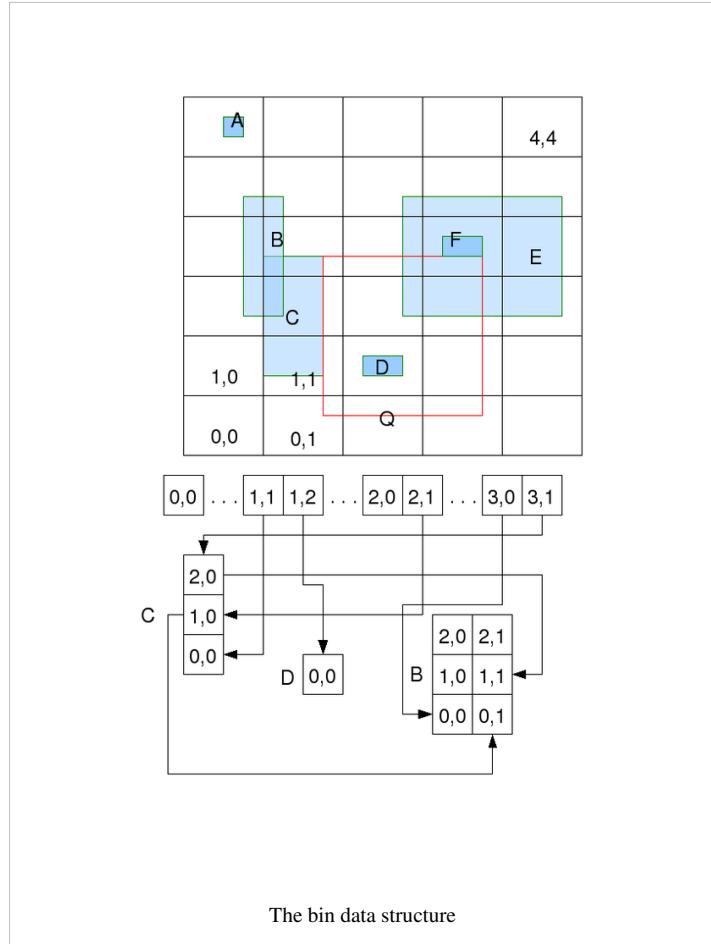
References

- [1] <http://www.emilstefanov.net/Projects/RangeSearchTree.aspx>
- [2] <http://www.cs.umd.edu/class/fall2005/cmsc754/Lects/754lects.pdf>

Bin

In computational geometry, the **bin** data structure allows efficient region queries, i.e., if there are some axis-aligned rectangles on a 2D plane, answer the question *Given a query rectangle, return all rectangles intersecting it.* kd-tree is another data structure that can answer this question efficiently. In the example in the figure, A, B, C, D, E, and F are existing rectangles, the query with the rectangle Q should return C, D, E and F, if we define all rectangles as closed intervals.

The data structure partitions a region of the 2D plane into uniform-sized *bins*. The bounding box of the bins encloses all *candidate* rectangles to be queried. All the bins are arranged in a 2D array. All the candidates are represented also as 2D arrays. The size of a candidate's array is the number of bins it intersects. For example, in the figure, candidate B has 6 elements arranged in a 3 row by 2 column array because it intersects 6 bins in such an arrangement. Each bin contains the head of a singly-linked list. If a candidate intersects a bin, it is chained to the bin's linked list. Each element in a candidate's array is a link node in the corresponding bin's linked list.



The bin data structure

Operations

Query

From the query rectangle Q , we can find out which bin its lower-left corner intersects efficiently by simply subtracting the bin's bounding box's lower-left corner from the lower-left corner of Q and dividing the result by the width and height of a bin respectively. We then iterate the bins Q intersects and examine all the candidates in the linked-lists of these bins. For each candidate we check if it does indeed intersect Q . If so and it is not previously reported, then we report it. We can use the convention that we only report a candidate the first time we find it. This can be done easily by clipping the candidate against the query rectangle and comparing its lower-left corner against the current location. If it is a match then we report, otherwise we skip.

Insertion and deletion

Insertion is linear to the number of bins a candidate intersects because inserting a candidate into 1 bin is constant time. Deletion is more expensive because we need to search the singly-linked list of each bin the candidate intersects.

In a multithread environment, insert, delete and query are mutually exclusive. However, instead of locking the whole data structure, a sub-range of bins may be locked. Detailed performance analysis should be done to justify the overhead.

Efficiency and tuning

The analysis is similar to a hash table. The worst-case scenario is that all candidates are concentrated in one bin. Then query is $O(n)$, delete is $O(n)$, and insert is $O(1)$, where n is the number of candidates. If the candidates are evenly spaced so that each bin has a constant number of candidates, The query is $O(k)$ where k is the number of bins the query rectangle intersects. Insert and delete are $O(m)$ where m is the number of bins the inserting candidate intersects. In practice delete is much slower than insert.

Like a hash table, bin's efficiency depends a lot on the distribution of both location and size of candidates and queries. In general, the smaller the query rectangle, the more efficient the query. The bin's size should be such that it contains as few candidates as possible but large enough so that candidates do not span too many bins. If a candidate spans many bins, a query has to skip this candidate over and over again after it is reported at the first bin of intersection. For example, in the figure, E is visited 4 times in the query of Q and so has to be skipped 3 times.

To further speed up the query, divisions can be replaced by right shifts. This requires the number of bins along an axis direction to be an exponent of 2.

Compared to other range query data structures

Against kd-tree, the bin structure allows efficient insertion and deletion without the complexity of rebalancing. This can be very useful in algorithms that need to incrementally add shapes to the search data structure.

See also

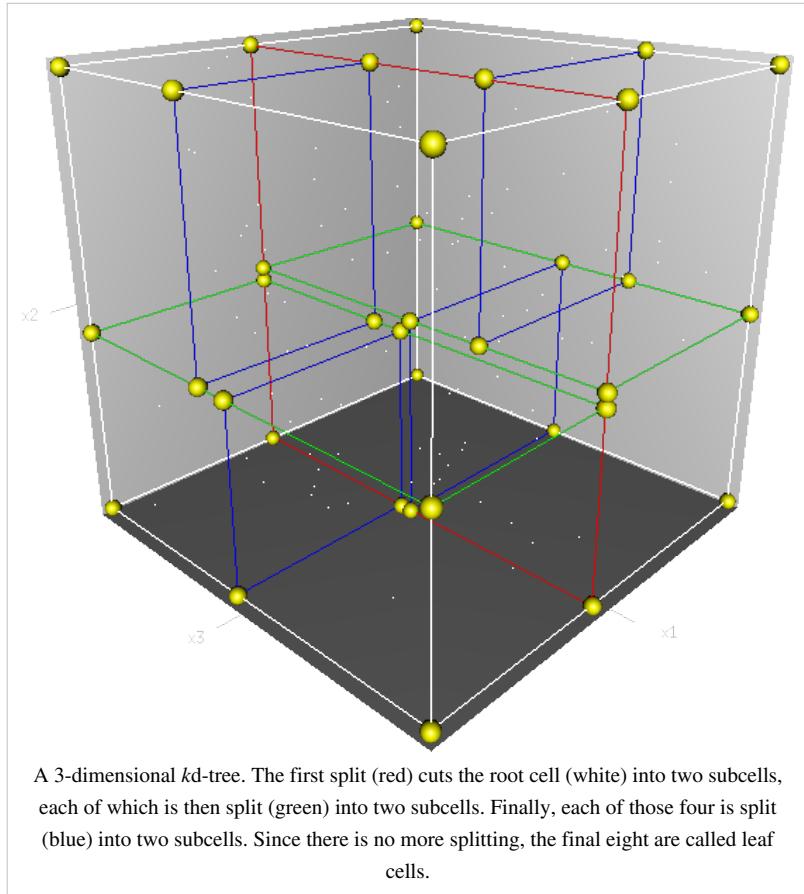
- kd-tree is another efficient range query data structure.
- Space partitioning

kd-tree

In computer science, a **kd-tree** (short for *k-dimensional tree*) is a space-partitioning data structure for organizing points in a *k*-dimensional space. kd-trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). kd-trees are a special case of BSP trees.

Informal Description

The kd-tree is a binary tree in which every node is a *k*-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as subspaces. Points to the left of this hyperplane represent the left sub-tree of that node and points right of the hyperplane are represented by the right sub-tree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the *k*-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right sub tree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis.



A 3-dimensional kd-tree. The first split (red) cuts the root cell (white) into two subcells, each of which is then split (green) into two subcells. Finally, each of those four is split (blue) into two subcells. Since there is no more splitting, the final eight are called leaf cells.

Operations on kd-trees

Construction

Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct kd-trees. The canonical method of kd-tree construction has the following constraints:

- As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, the root would have an *x*-aligned plane, the root's children would both have *y*-aligned planes, the root's grandchildren would all have *z*-aligned planes, the next level would have an *x*-aligned plane, and so on.)
- Points are inserted by selecting the median of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of points into the algorithm up-front.)

This method leads to a balanced kd-tree, in which each leaf node is about the same distance from the root. However, balanced trees are not necessarily optimal for all applications.

Note also that it is not *required* to select the median point. In that case, the result is simply that there is no guarantee that the tree will be balanced. A simple heuristic to avoid coding a complex linear-time median-finding algorithm nor using an $O(n \log n)$ sort is to use sort to find the median of a *fixed* number of *randomly* selected points to serve as the cut line. Practically this technique often results in nicely balanced trees.

Given a list of n points, the following algorithm will construct a balanced kd-tree containing those points.

```
function kdTree (list of points pointList, int depth)
{
    if pointList is empty
        return nil;
    else
    {
        // Select axis based on depth so that axis cycles through all
        valid values
        var int axis := depth mod k;

        // Sort point list and choose median as pivot element
        select median by axis from pointList;

        // Create node and construct subtrees
        var tree_node node;
        node.location := median;
        node.leftChild := kdTree(points in pointList before median,
depth+1);
        node.rightChild := kdTree(points in pointList after median,
depth+1);
        return node;
    }
}
```

It is common that points "after" the median include only ones that are greater than or equal to the median. Another approach is to define a "superkey" function that compares the points in other dimensions. Lastly, it may be acceptable to let points equal to the median lie on either side.

This algorithm implemented in the Python programming language is as follows:

```
class Node:pass

def kdTree(pointList, depth=0):
    if not pointList:
        return

        # Select axis based on depth so that axis cycles through all valid
values
    k = len(pointList[0]) # assumes all points have the same dimension
    axis = depth % k

    # Sort point list and choose median as pivot element
    pointList.sort(key=lambda point: point[axis])
```

```

median = len(pointList)/2 # choose median

# Create node and construct subtrees
node = Node()
node.location = pointList[median]
node.leftChild = kdTree(pointList[0:median], depth+1)
node.rightChild = kdTree(pointList[median+1:], depth+1)
return node

```

Example usage would be:

```

pointList = [(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)]
tree = kdTree(pointList)

```

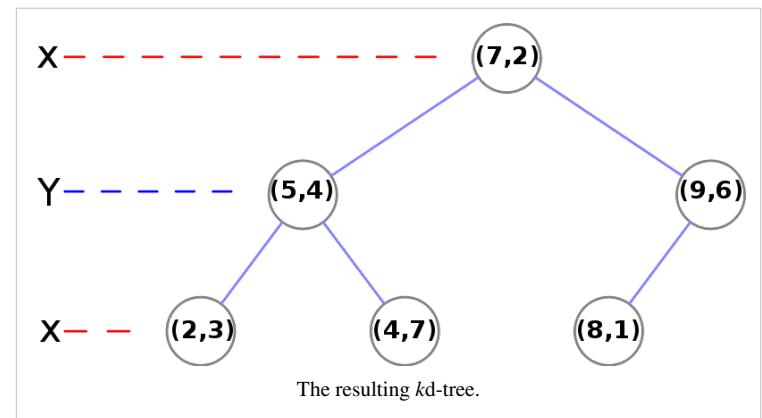
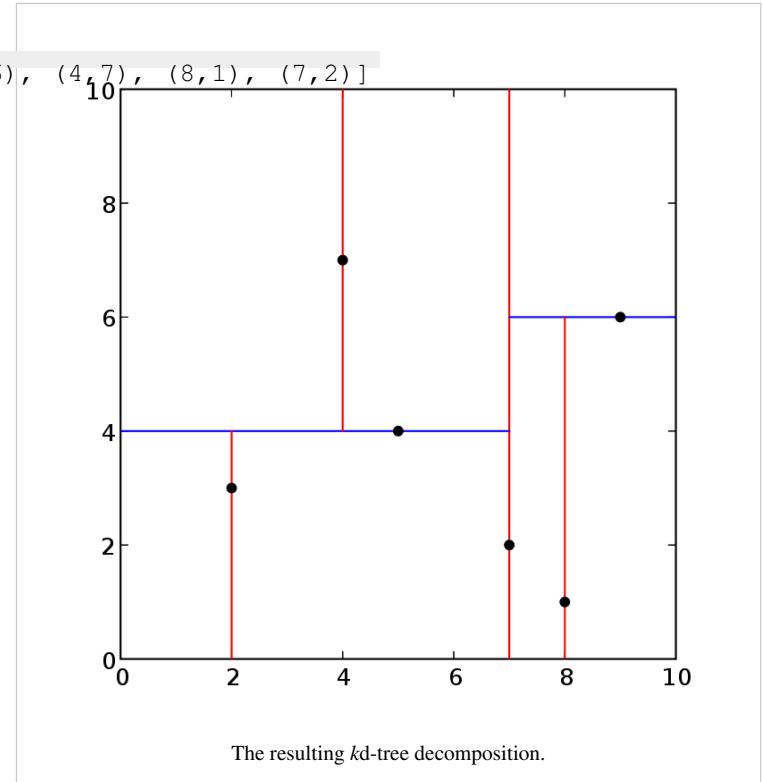
The tree generated is shown on the right.

This algorithm creates the invariant that for any node, all the nodes in the left subtree are on one side of a splitting plane, and all the nodes in the right subtree are on the other side. Points that lie on the splitting plane may appear on either side. The splitting plane of a node goes through the point associated with that node (referred to in the code as *node.location*).

Adding elements

One adds a new point to a *kd*-tree in the same way as one adds an element to any other search tree. First, traverse the tree, starting from the root and moving to either the left or the right child depending on whether the point to be inserted is on the "left" or "right" side of the splitting plane. Once you get to the node under which the child should be located, add the new point as either the left or right child of the leaf node, again depending on which side of the node's splitting plane contains the new node.

Adding points in this manner can cause the tree to become unbalanced, leading to decreased tree performance. The rate of tree



performance degradation is dependent upon the spatial distribution of tree points being added, and the number of points added in relation to the tree size. If a tree becomes too unbalanced, it may need to be re-balanced to restore the performance of queries that rely on the tree balancing, such as nearest neighbour searching.

Removing elements

To remove a point from an existing *kd*-tree, without breaking the invariant, the easiest way is to form the set of all nodes and leaves from the children of the target node, and recreate that part of the tree. This differs from regular search trees in that no child can be selected for a "promotion", since the splitting plane for lower-level nodes is not along the required axis for the current tree level.

Balancing

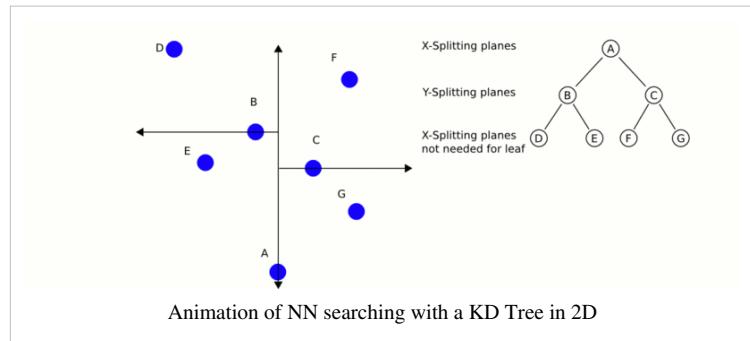
Balancing a *kd*-tree requires care. Because *kd*-trees are sorted in multiple dimensions, the tree rotation technique cannot be used to balance them — this may break the invariant.

Nearest neighbor search

The nearest neighbor (NN) algorithm aims to find the point in the tree which is nearest to a given input point. This search can be done efficiently by using the tree properties to quickly eliminate large portions of the search space. Searching for a nearest neighbor in a *kd*-tree proceeds as follows:

1. Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes right or left depending on whether the point is greater or less than the current node in the split dimension).
2. Once the algorithm reaches a leaf node, it saves that node point as the "current best"
3. The algorithm unwinds the recursion of the tree, performing the following steps at each node:
 1. If the current node is closer than the current best, then it becomes the current best.
 2. The algorithm checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In concept, this is done by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. Since the hyperplanes are all axis-aligned this is implemented as a simple comparison to see whether the difference between the splitting coordinate of the search point and current node is less than the distance (overall coordinates) from the search point to the current best.
 1. If the hypersphere crosses the plane, there could be nearer points on the other side of the plane, so the algorithm must move down the other branch of the tree from the current node looking for closer points, following the same recursive process as the entire search.
 2. If the hypersphere doesn't intersect the splitting plane, then the algorithm continues walking up the tree, and the entire branch on the other side of that node is eliminated.

Generally the algorithm uses squared distances for comparison to avoid computing square roots. Additionally, it can save computation by holding the squared current best distance in a variable for comparison.



Sample LUA - NN Search

```

function kdsearchnn( here, point, best )

    if here == nil then
        return best
    end

    if best == nil then
        best = here
    end

    -- consider the current node --
    disthere = distance(here,point)
    distbest = distance(best,point)
    if disthere < distbest then
        best = here
    end

    -- search the near branch --
    child = child_near(here,point)
    best = kdsearchnn( child, point, best )

    -- search the away branch - maybe --
    distbest = distance(best,point)
    distaxis = distance_axis(here,point)
    if distaxis < distbest then
        child = child_away(here,point)
        best = kdsearchnn( child, point, best )
    end

    return best
end

```

Finding the nearest point is an $O(\log N)$ operation in the case of randomly distributed points if N . Analyses of binary search trees has found that the worst case search time for an k -dimensional KD tree containing N nodes is given by the following equation^[1].

$$t_{worst} = O(k \cdot N^{1-\frac{1}{k}})$$

These poor running times only apply when N is on the order of the number of dimensions. In very high dimensional spaces, the curse of dimensionality causes the algorithm to need to visit many more branches than in lower dimensional spaces. In particular, when the number of points is only slightly higher than the number of dimensions, the algorithm is only slightly better than a linear search of all of the points.

The algorithm can be extended in several ways by simple modifications. It can provide the k -Nearest Neighbors to a point by maintaining k current bests instead of just one. Branches are only eliminated when they can't have points closer than any of the k current bests.

It can also be converted to an approximation algorithm to run faster. For example, approximate nearest neighbour searching can be achieved by simply setting an upper bound on the number points to examine in the tree, or by

interrupting the search process based upon a real time clock (which may be more appropriate in hardware implementations). Nearest neighbour for points that are in the tree already can be achieved by not updating the refinement for nodes that give zero distance as the result, this has the downside of discarding points that are not unique, but are co-located with the original search point.

Approximate nearest neighbor is useful in real time applications such as robotics due to the significant speed increase gained by not searching for the best point exhaustively. One of its implementations is Best Bin First.

High-Dimensional Data

kd-trees are not suitable for efficiently finding the nearest neighbour in high dimensional spaces. As a general rule, if the dimensionality is k , then number of points in the data, N , should be $N \gg 2^k$. Otherwise, when kd-trees are used with high-dimensional data, most of the points in the tree will be evaluated and the efficiency is no better than exhaustive search.^[2], and approximate nearest-neighbour methods are used instead.

Complexity

- Building a static kd-tree from n points takes $O(n \log^2 n)$ time if an $O(n \log n)$ sort is used to compute the median at each level. The complexity is $O(n \log n)$ if a linear median-finding algorithm such as the one described in Cormen *et al.*^[3] is used.
- Inserting a new point into a balanced kd-tree takes $O(\log n)$ time.
- Removing a point from a balanced kd-tree takes $O(\log n)$ time.
- Querying an axis-parallel range in a balanced kd-tree takes $O(n^{1-1/k} + m)$ time, where m is the number of the reported points, and k the dimension of the kd-tree.

Variations

Instead of points

Instead of points, a kd-tree can also contain rectangles or hyperrectangles^[4]. A 2D rectangle is considered a 4D object ($x_{\text{low}}, x_{\text{high}}, y_{\text{low}}, y_{\text{high}}$). Thus range search becomes the problem of returning all rectangles intersecting the search rectangle. The tree is constructed the usual way with all the rectangles at the leaves. In an orthogonal range search, the *opposite* coordinate is used when comparing against the median. For example, if the current level is split along x_{high} , we check the x_{low} coordinate of the search rectangle. If the median is less than the x_{low} coordinate of the search rectangle, then no rectangle in the left branch can ever intersect with the search rectangle and so can be pruned. Otherwise both branches should be traversed. See also interval tree, which is a 1-dimensional special case.

Points only in leaves

It is also possible to define a kd-tree with points stored solely in leaves^[5]. This form of kd-tree allows a variety of split mechanics other than the standard median split. The midpoint splitting rule^[6] selects on the middle of the longest axis of the space being searched, regardless of the distribution of points. This guarantees that the aspect ratio will be at most 2:1, but the depth is dependent on the distribution of points. A variation, called sliding-midpoint, only splits on the middle if there are points on both sides of the split. Otherwise, it splits on point nearest to the middle. Maneewongvatana and Mount show that this offers "good enough" performance on common data sets. Using sliding-midpoint, an approximate nearest neighbor query can be answered in $O\left(\frac{1}{\epsilon^d} \log n\right)$. Approximate range counting can be answered in $O\left(\log n + \left(\frac{1}{\epsilon}\right)^d\right)$ with this method.

See also

- implicit kd-tree
- min/max kd-tree
- Quadtree
- Octree
- Bounding Interval Hierarchy
- Nearest neighbor search
- Klee's measure problem

External links

- libkdTree++^[7], an open-source STL-like implementation of kd-trees in C++.
- A tutorial on KD Trees^[8]
- A C++ implementation of kd-trees for 3D point clouds^[9], part of the Mobile Robot Programming Toolkit (MRPT)
- kdTree^[10] A simple C library for working with KD-Trees

References

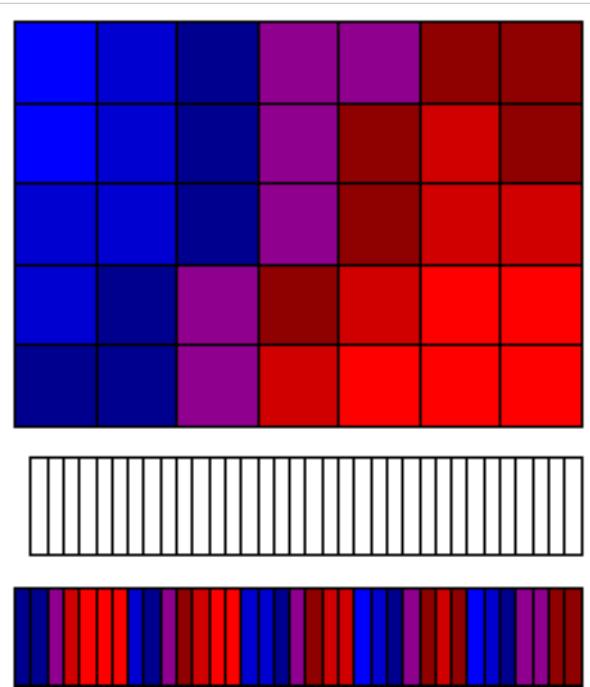
- [1] Lee, D. T.; Wong, C. K. (1977), "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees", *Acta Informatica* 9 (1): 23–29, doi:10.1007/BF00263763
- [2] Jacob E. Goodman, Joseph O'Rourke and Piotr Indyk (Ed.) (2004). "Chapter 39 : Nearest neighbors in high-dimensional spaces". *Handbook of Discrete and Computational Geometry* (2nd ed.). CRC Press.
- [3] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. *Introduction to Algorithms*. MIT Press and McGraw-Hill. Chapter 10.
- [4] J. L. Bentley. Multidimensional binary search trees used for associative searching (<http://doi.acm.org/10.1145/361002.361007>). Communications of the ACM, 18(9):509-517, 1975.
- [5] de Berg, Mark et al. Computational Geometry: Algorithms and Applications, 3rd Edition, pages 99-101. Springer, 2008.
- [6] S. Maneewongvatana and D. M. Mount. It's okay to be skinny, if your friends are fat (<http://www.cs.umd.edu/~mount/Papers/cgc99-smpack.pdf>). 4th Annual CGC Workshop on Computational Geometry, 1999.
- [7] <http://libkdTree.alioth.debian.org>
- [8] <http://www.autonlab.org/autonweb/14665/version/2/part/5/data/moore-tutorial.pdf?branch=main&language=en>
- [9] http://babel.isa.uma.es/mrpt/index.php/The_hierarchical_model_of_metric_maps#Point_maps
- [10] <http://code.google.com/p/kdTree/>

Implicit kd-tree

An **implicit kd-tree** is a *kd-tree* defined implicitly above a rectilinear grid. Its split planes' positions and orientations are not given explicitly but implicitly by some recursive splitting-function defined on the hyperrectangles belonging to the tree's nodes. Each inner node's split plane is positioned on a grid plane of the underlying grid, partitioning the node's grid into two subgrids.

Nomenclature and References

The terms "min/max *kd-tree*" and "implicit *kd-tree*" are sometimes mixed up. This is because the first publication using the term "implicit *kd-tree*"^[1] did actually use explicit min/max *kd-trees* but referred to them as "implicit *kd-trees*" to indicate that they may be used to ray trace implicitly given iso surfaces. Nevertheless this publication used also slim *kd-trees* which are a subset of the implicit *kd-trees* with the restriction that they can only be built over integer hyperrectangles with sidelengths that are powers of two. Implicit *kd-trees* as defined here have shortly after been introduced^[2]. A nice overview to implicit *kd-trees* can be found in^[3]. As it is possible to assign attributes to implicit *kd-tree* nodes, one may refer to an implicit *kd-tree* which has min/max values assigned to its nodes as an "implicit min/max *kd-tree*".



Construction and storage of a 2D implicit max *kd-tree* using the grid median splitting-function. Each cell of the rectilinear grid has one scalar value from low (bright blue) to high (bright red) assigned to it. The grid's memory footprint is indicated in the lower line. The implicit max *kd-tree*'s predefined memory footprint needs one scalar value less than that. The storing of the node's max values is indicated in the upper line.

Construction

Implicit *kd-trees* are in general not constructed explicitly. When accessing a node, its split plane orientation and position are evaluated using the specific splitting-function defining the tree. Different splitting-functions may result in different trees for the same underlying grid.

Splitting-functions

Splitting-functions may be adapted to special purposes. Underneath two specifications of special splitting-function classes.

- **Non-degenerated splitting-functions** do not allow the creation of degenerated nodes (nodes whose corresponding integer hyperrectangle's volume is equal zero). Their corresponding implicit *kd-trees* are full binary trees, which have for n leaf nodes $n - 1$ inner nodes. Their corresponding implicit *kd-trees* are **non-degenerated implicit kd-trees**.
- **complete splitting-functions** are non-degenerated splitting-functions whose corresponding implicit *kd-tree*'s leaf nodes are single grid cells such that they have one inner node less than the amount of gridcells given in the grid. The corresponding implicit *kd-trees* are **complete implicit kd-trees**.

A complete splitting function is for example the **grid median splitting-function**. It creates fairly balanced implicit *kd*-trees by using *k*-dimensional integer hyperrectangles $hyprec[2][k]$ belonging to each node of the implicit *kd*-tree. The hyperrectangles define which gridcells of the rectilinear grid belong to their corresponding node. If the volume of this hyperrectangle equals one, the corresponding node is a single grid cell and is therefore not further subdivided and marked as leaf node. Otherwise the hyperrectangle's longest extend is chosen as orientation o . The corresponding split plane p is positioned onto the grid plane that is closest to the hyperrectangle's grid median along that orientation.

Split plane orientation o :

```
o = min{argmax(i = 1 ... k: (hyprec[1][i] - hyprec[0][i]))}
```

Split plane position p :

```
p = roundDown((hyprec[0][o] + hyprec[1][o]) / 2)
```

Assigning attributes to implicit *kd*-tree-nodes

An obvious advantage of implicit *kd*-trees is that their split plane's orientations and positions need not to be stored explicitly.

But some applications require besides the split plane's orientations and positions further attributes at the inner tree nodes. These attributes may be for example single bits or single scalar values, defining if the subgrids belonging to the nodes are of interest or not. For complete implicit *kd*-trees it is possible to pre-allocate a correctly sized array of attributes and to assign each inner node of the tree to a unique element in that allocated array.

The amount of gridcells in the grid is equal the volume of the integer hyperrectangle belonging to the grid. As a complete implicit *kd*-tree has one inner node less than grid cells, it is known in advance how many attributes need to be stored. The relation "*Volume of integer hyperrectangle to inner nodes*" defines together with the complete splitting-function a recursive formula assigning to each split plane a unique element in the allocated array. The corresponding algorithm is given in C-pseudo code underneath.

```
// Assigning attributes to inner nodes of a complete implicit kd-tree

// create an integer help hyperrectangle hyprec (its volume vol(hyprec)
// is equal the amount of leaves)
int hyprec[2][k] = ;
// allocate once the array of attributes for the entire implicit
// kd-tree
attr *a = new attr[volume(hyprec) - 1];

attr implicitKdTreeAttributes(int hyprec[2][k], attr *a)
{
    if(vol(hyprec) > 1) // the current node is an inner node
    {
        // evaluate the split plane's orientation o and its position p
        // using the underlying complete split-function
        int o, p;
        completeSplittingFunction(hyprec, &o, &p);
        // evaluate the children's integer hyperrectangles hyprec_l and
        hyprec_r
        int hyprec_l[2][k], hyprec_r[2][k];
        hyprec_l      = hyprec;
```

```

hyprec_l[1][o] = p;
hyprec_r       = hyprec;
hyprec_r[0][o] = p;
// evaluate the children's memory location a_l and a_r
attr* a_l = a + 1;
attr* a_r = a + vol(hyprec_l);
// evaluate recursively the children's attributes c_l and c_r
attr c_l = implicitKdTreeAttributes(hyprec_l, a_l);
attr c_r = implicitKdTreeAttributes(hyprec_r, a_r);
// merge the children's attributes to the current attribute c
attr c = merge(c_l, c_r);
// store the current attribute and return it
a[0] = c;
return c;
}

// The current node is a leaf node. Return the attribute belonging to
// the corresponding gridcell
return attribute(hyprec);
}

```

It is worth mentioning that this algorithm works for all rectilinear grids. The corresponding integer hyperrectangle does not necessarily have to have sidelengths that are powers of two.

Applications

Implicit max-*kd* trees are used for ray casting isosurfaces/MIP (maximum intensity projection). The attribute assigned to each inner node is the maximal scalar value given in the subgrid belonging to the node. Nodes are not traversed if their scalar values are smaller than the searched iso-value/current maximum intensity along the ray. The low storage requirements of the implicit max *kd*-tree and the favorable visualization complexity of ray casting allow to ray cast (and even change the isosurface for) very large scalar fields at interactive framerates on commodity PCs.

Complexity

Given an implicit *kd*-tree spanned over an *k*-dimensional grid with *n* gridcells.

- Assigning attributes to the nodes of the tree takes $O(kn)$ time.
- Storing attributes to the nodes takes $O(n)$ memory.
- Ray casting iso-surfaces/MIP an underlying scalar field using the corresponding implicit max *kd*-tree takes roughly $O(\lg(n))$ time.

See also

- *kd-tree*
- min/max *kd-tree*

References

- [1] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek and Hans-Peter Seidel "Faster Isosurface Ray Tracing using Implicit KD-Trees" IEEE Transactions on Visualization and Computer Graphics (2005)
- [2] Matthias Groß, Carsten Lojewski, Martin Bertram and Hans Hagen "Fast Implicit *kd*-Trees: Accelerated Isosurface Ray Tracing and Maximum Intensity Projection for Large Scalar Fields" CGIM07: Proceedings of Computer Graphics and Imaging (2007) 67-74
- [3] Matthias Groß (PhD, 2009) Towards Scientific Applications for Interactive Ray Casting (<http://kluedo.ub.uni-kl.de/volltexte/2009/2361/>)

min/max kd-tree

A **min/max kd-tree** is a *kd-tree* with two scalar values - a minimum and a maximum - assigned to its nodes. The minimum/maximum of an inner node is equal the minimum/maximum of its children's minima/maxima.

Construction

Min/max *kd-trees* may be constructed recursively. Starting with the root node, the splitting plane orientation and position is evaluated. Then the children's splitting planes and min/max values are evaluated recursively. The min/max value of the current node is simply the minimum/maximum of its children's minima/maxima.

Properties

The min/max *kd-tree* has - besides the properties of an *kd-tree* - the special property that an inner node's min/max values coincide each with a min/max value of either one child. This allows to discard the storage of min/max values at the leaf nodes by storing two bits at inner nodes, assigning min/max values to the children: Each inner node's min/max values will be known in advance, where the root node's min/max values are stored separately. Each inner node has besides two min/max values also two bits given, defining to which child those min/max values are assigned (0: to the left child 1: to the right child). The non-assigned min/max values of the children are the from the current node already known min/max values. The two bits may also be stored in the least significant bits of the min/max values which have therefore to be approximated by fractioning them down/up.

The resulting memory reduction is not minor, as the leaf nodes of full binary *kd-trees* are one half of the tree's nodes.

Applications

Min/max *kd-trees* are used for ray casting isosurfaces/MIP (maximum intensity projection). Isosurface ray casting only traverses nodes for which the chosen isovalue lies in between the min/max value of the current node. Nodes that do not fulfill that requirement do not contain an isosurface to the given isovalue and are therefore skipped (empty space skipping). For MIP are nodes not traversed if their maximum is smaller than the current maximum intensity along the ray. The favorable visualization complexity of ray casting allows to ray cast (and even change the isosurface for) very large scalar fields at interactive framerates on commodity PCs. Especial implicit max *kd-trees* are an optimal choice for visualizing scalar fields defined on rectilinear grids (see also [1], [2], [3]).

See also

- *kd-tree*
- implicit *kd-tree*

References

- [1] Matthias Groß, Carsten Lojewski, Martin Bertram and Hans Hagen "Fast Implicit KD-Trees: Accelerated Isosurface Ray Tracing and Maximum Intensity Projection for Large Scalar Fields" CGIM07: Proceedings of Computer Graphics and Imaging (2007) 67-74
- [2] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek and Hans-Peter Seidel "Faster Isosurface Ray Tracing using Implicit KD-Trees" IEEE Transactions on Visualization and Computer Graphics (2005)
- [3] Matthias Groß (PhD, 2009) Towards Scientific Applications for Interactive Ray Casting (<http://kluedo.ub.uni-kl.de/volltexte/2009/2361/>)

Adaptive k-d tree

An **adaptive k-d tree** is a tree for multidimensional points where successive levels may be split along different dimensions.

 This article incorporates public domain material from the NIST document "Adaptive k-d tree" [1] by Paul E. Black (Dictionary of Algorithms and Data Structures).

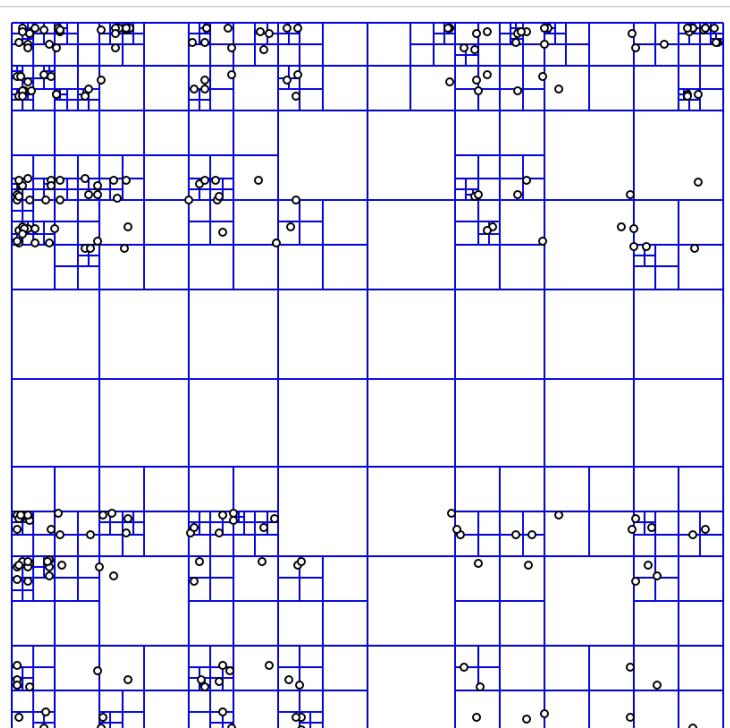
References

- [1] <http://www.nist.gov/dads/HTML/adaptkdtree.html>

Quadtree

A **quadtree** is a tree data structure in which each internal node has up to four children. Quadtrees are most often used to partition a two dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a *Q-tree*. All forms of Quadtrees share some common features:

- They decompose space into adaptable cells
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits
- The tree directory follows the spatial decomposition of the Quadtree



Types

Quadtrees may be classified according to the type of data they represent, including areas, points, lines and curves. Quadtrees may also be classified by whether the shape of the tree is independent of the order data is processed. Some common types of quadtrees are:

The region quadtree

The region quadtree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on with each leaf node containing data corresponding to a specific subregion. Each node in the tree either has exactly four children, or has no children (a leaf node). The region quadtree is not strictly a 'tree' - as the positions of subdivisions are independent of the data. They are more precisely called 'tries'.

A region quadtree with a depth of n may be used to represent an image consisting of $2^n \times 2^n$ pixels, where each pixel value is 0 or 1. The root node represents the entire image region. If the pixels in any region are not entirely 0s or 1s, it is subdivided. In this application, each leaf node represents a block of pixels that are all 0s or all 1s.

A region quadtree may also be used as a variable resolution representation of a data field. For example, the temperatures in an area may be stored as a quadtree, with each leaf node storing the average temperature over the subregion it represents.

If a region quadtree is used to represent a set of point data (such as the latitude and longitude of a set of cities), regions are subdivided until each leaf contains at most a single point.

Point quadtree

The point quadtree is an adaptation of a binary tree used to represent two dimensional point data. It shares the features of all quadtrees but is a true tree as the center of a subdivision is always on a point. The tree shape depends on the order data is processed. It is often very efficient in comparing two dimensional ordered data points, usually operating in $O(\log n)$ time.

Node structure for a point quadtree

A node of a point quadtree is similar to a node of a binary tree, with the major difference being that it has four pointers (one for each quadrant) instead of two ("left" and "right") as in an ordinary binary tree. Also a key is usually decomposed into two parts, referring to x and y coordinates. Therefore a node contains following information:

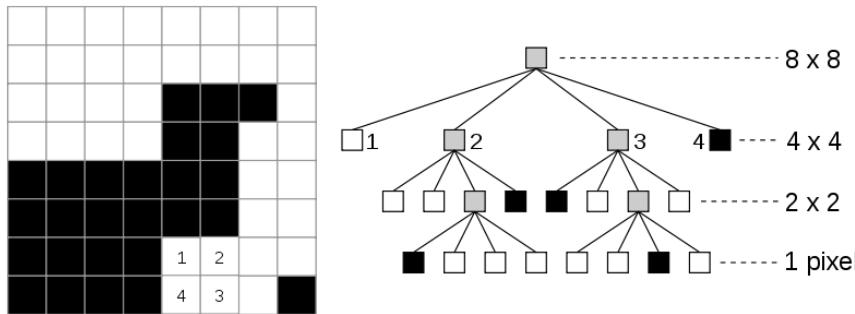
- 4 Pointers: quad['NW'], quad['NE'], quad['SW'], and quad['SE']
- point; which in turn contains:
 - key; usually expressed as x, y coordinates
 - value; for example a name

Edge quadtree

Edge quadtree are specifically used to store lines rather than points. Curves are approximated by subdividing cells to a very fine resolution. This can result in extremely unbalanced trees which may defeat the purpose of indexing.

Some common uses of quadtrees

- Image representation



- Spatial indexing
- Efficient collision detection in two dimensions
- View frustum culling of terrain data
- Storing sparse data, such as a formatting information for a spreadsheet or for some matrix calculations
- Solution of multidimensional fields (computational fluid dynamics, electromagnetism)
- Conway's Game of Life simulation program.^[1]

Quadtrees are the two-dimensional analog of octrees.

Caveats

If geometric subdividing fails to reduce the item count for each quadrant, (e.g., for overlapping data,) QuadTree subpartitioning fails, and the capacity must be breached for the algorithm to continue. For example, if the maximum capacity for a quadrant is 8, and there are 9 points at (0, 0), subpartitioning would produce three empty quadrants, and one containing the original 9 points, and so on. Because the tree must allow more than 8 points in such a quadrant, QuadTrees can approach O(N) complexity for data sets with arbitrary geometry (e.g., maps or graphs).

References

General references

1. Raphael Finkel and J.L. Bentley (1974). "Quad Trees: A Data Structure for Retrieval on Composite Keys". *Acta Informatica* 4 (1): 1–9. doi:10.1007/BF00288933.
2. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf (2000). *Computational Geometry* (2nd revised ed.). Springer-Verlag. ISBN 3-540-65620-0. Chapter 14: Quadtrees: pp.291–306.

See also

- Octree
- Binary space partitioning
- Kd-tree
- R-tree
- UB-tree
- Spatial index

- Spatial database

External links

- A discussion of the Quadtree and an application [2]
- Considerable discussion and demonstrations of Spatial Indexing [3]
- Example C# code for a quad tree [4]

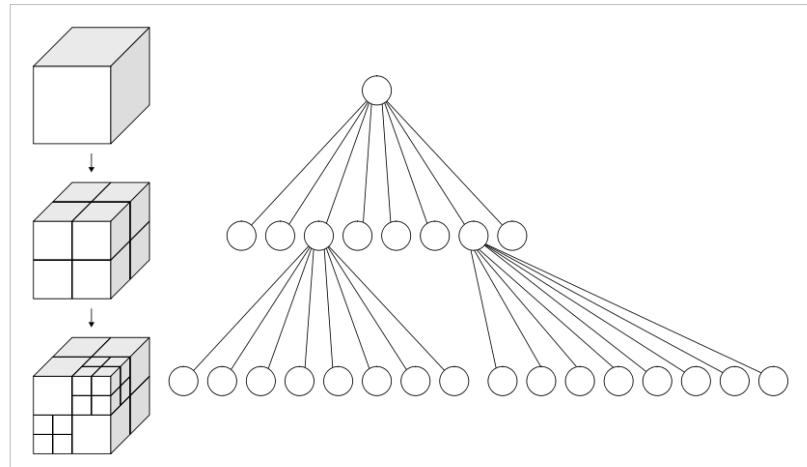
References

- [1] Tomas G. Rokicki (2006-04-01). "An Algorithm for Compressing Space and Time" (<http://www.ddj.com/hpc-high-performance-computing/184406478>). . Retrieved 2009-05-20.
- [2] <http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html>
- [3] <http://homepages.ge.ucl.ac.uk/~mhaklay/java.htm>
- [4] <http://digitseven.com/QuadTree.aspx>

Octree

An **octree** is a tree data structure in which each internal node has up to eight children. Octrees are most often used to partition a three dimensional space by recursively subdividing it into eight octants. Octrees are the three-dimensional analog of quadtrees. The name is formed from *oct* + *tree*, and normally written "*octree*", not "*octtree*".

Each node in an octree subdivides the space it represents into eight octants. In a point region (PR) octree, the node stores an explicit 3-dimensional point, which is the "center" of the subdivision for that node; the point defines one of the corners for each of the eight children. In an MX octree, the subdivision point is implicitly the center of the space the node represents. The root node of a PR octree can represent infinite space; the root node of an MX octree must represent a finite bounded space so that the implicit centers are well-defined. Octrees are never considered kD-trees, as kD-trees split along a dimension and octrees split around a point. kD-trees are also always binary, which is not true of octrees.



Left: Recursive subdivision of a cube into octants. Right: The corresponding octree.

- Spatial indexing
- Efficient collision detection in three dimensions
- View frustum culling
- Fast Multipole Method
- Unstructured grid
- Finite element analysis

Application to color quantization

The octree color quantization algorithm, invented by Gervautz and Purgathofer in 1988, encodes image color data as an octree up to nine levels deep. Octrees are used because $2^3 = 8$ and there are three color components in the RGB system. The node index to branch out from at the top level is determined by a formula that uses the most significant bits of the red, green, and blue color components, e.g. $4r + 2g + b$. The next lower level uses the next bit significance, and so on. Less significant bits are sometimes ignored to reduce the tree size.

The algorithm is highly memory efficient because the tree's size can be limited. The bottom level of the octree consists of leaf nodes that accrue color data not represented in the tree; these nodes initially contain single bits. If much more than the desired number of palette colors are entered into the octree, its size can be continually reduced by seeking out a bottom-level node and averaging its bit data up into a leaf node, pruning part of the tree. Once sampling is complete, exploring all routes in the tree down to the leaf nodes, taking note of the bits along the way, will yield approximately the required number of colors.

See also

- Quadtree
- kd-tree
- Loose octrees
- Bounding Interval Hierarchy
- Klee's measure problem
- Linear octrees
- Balanced octrees
- Cube 2, a 3D game engine in which geometry is almost entirely based on octrees
- OGRE, has an Octree Scene Manager Implementation
- Irrlicht Engine, supports octree scene nodes
- Dendro^[1], a parallel multigrid library for finite element computations using octrees

External links

- Octree Quantization in Microsoft Systems Journal^[2]
- Color Quantization using Octrees in Dr. Dobb's^[3]
- Color Quantization using Octrees in Dr. Dobb's Source Code^[4]
- Octree Color Quantization Overview^[5]
- Parallel implementation of octtree generation algorithm, P. Sojan Lal, A Unnikrishnan, K Poulose Jacob, ICIP 1997, IEEE Digital Library^[6]
- Generation of Octrees from Raster Scan with Reduced Information Loss, P. Sojan Lal, A Unnikrishnan, K Poulose Jacob, IASTED International conference VIIP 2001^[7] [8]
- C++ implementation (GPL license)^[9]
- Parallel Octrees for Finite Element Applications^[10]
- Cube 2: Sauerbraten - a game written in the octree-heavy Cube 2 engine^[11]
- Ogre - A 3d Object-oriented Graphics Rendering Engine with a Octree Scene Manager Implementation (LGPL license)^[12]
- Dendro: parallel multigrid for octree meshes (MPI/C++ implementation)^[1]

References

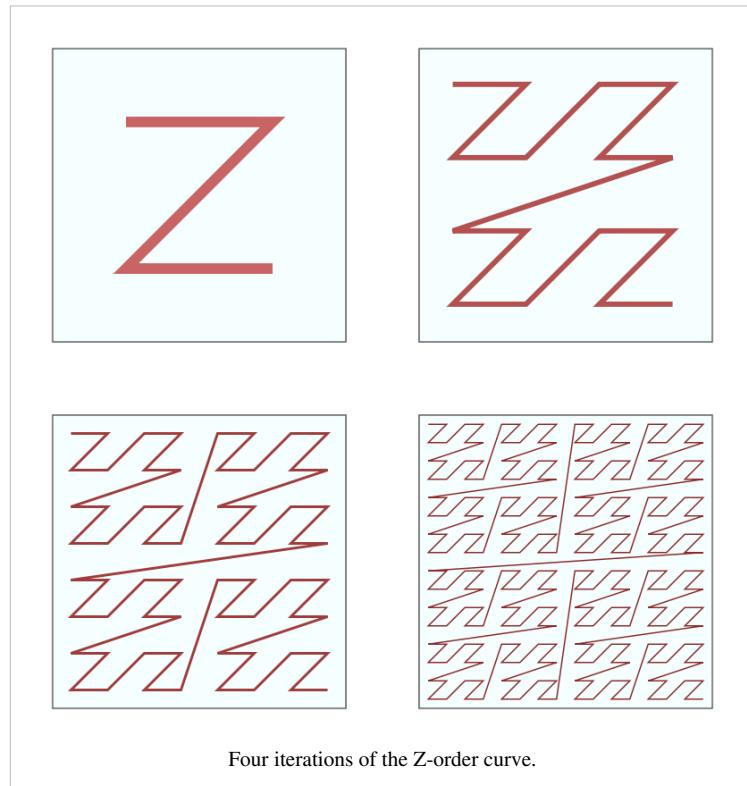
- [1] <http://www.cc.gatech.edu/csela/dendro>
- [2] <http://www.microsoft.com/msj/archive/S3F1.aspx>
- [3] <http://www.ddj.com/184409805>
- [4] <ftp://66.77.27.238/sourcecode/ddj/1996/9601.zip>
- [5] http://web.cs.wpi.edu/~matt/courses/cs563/talks/color_quant/CQoctree.html
- [6] http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=727419
- [7] <http://dblp.uni-trier.de/db/conf/viip/viip2001.html#LalUJ01>
- [8] http://www.actapress.com/catalogue2009/proc_series13.html#viip2001
- [9] <http://nomis80.org/code/octree.html>
- [10] <http://sc07.supercomputing.org/schedule/pdf/pap117.pdf>
- [11] <http://www.sauerbraten.org/>
- [12] <http://www.ogre3d.org>

Linear octrees

An octree is said to be **complete** if every internal node has exactly 8 child nodes. If the maximum permissible depth of an octree is fixed a priori, then it is sufficient to store the complete list of leaf nodes of the octree. Such a representation is referred to a **Linear octree**, since a linear array is sufficient for this representation instead of the tree data structure. All the nodes of the octree can be generated from the list of its leaf nodes. Space filling curves are often used to represent linear octrees.

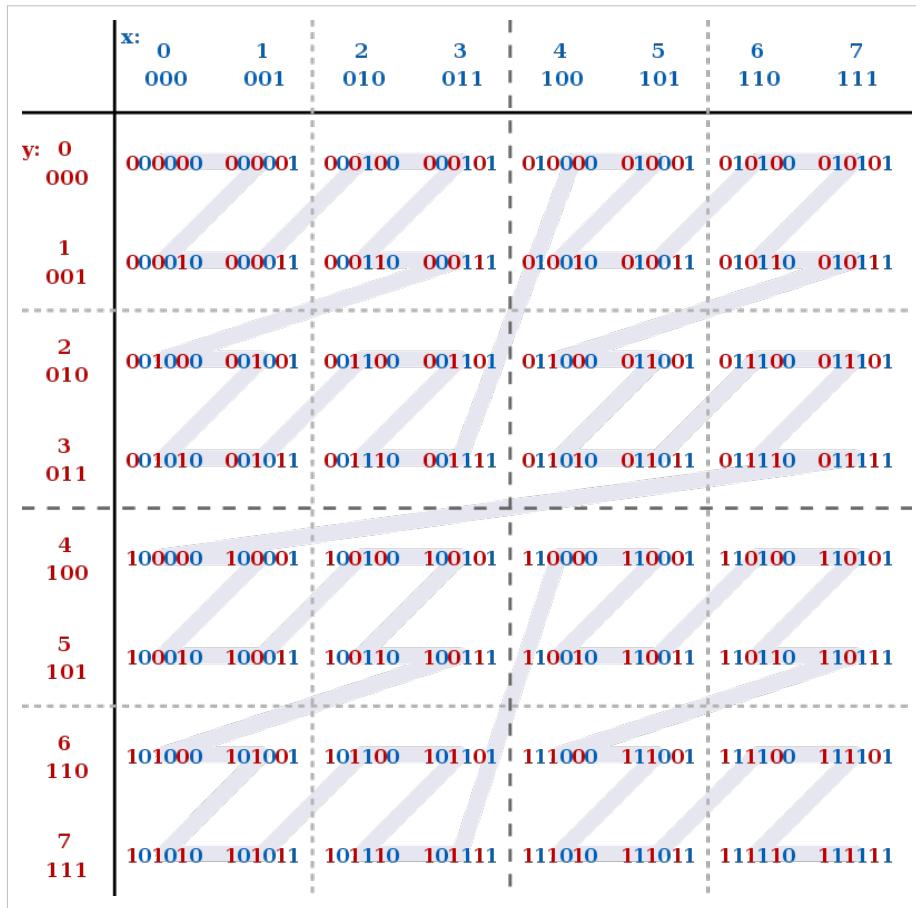
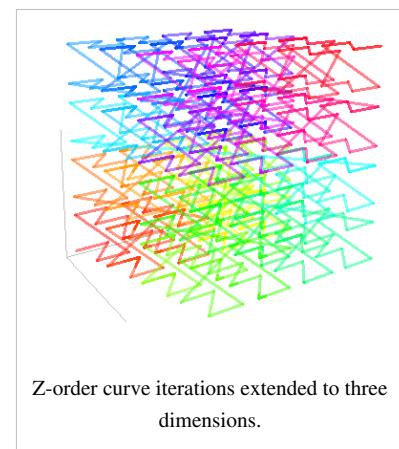
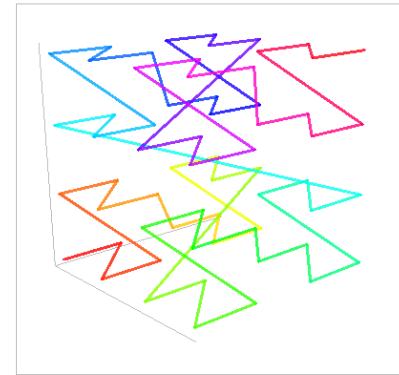
Z-order

Z-order, Morton-order or Morton code first proposed in 1966 by G. M. Morton,^[1] is a space-filling curve which is often used in computer science: Due to its good locality-preserving behaviour it is used in data structures for mapping multidimensional data to one dimension. The z-value of a point in multidimensions is simply calculated by interleaving the binary representations of its coordinate values. Once the data are sorted into this ordering, any one-dimensional data structure can be used such as binary search trees, B-trees, skip lists or (with low significant bits truncated) hash tables. The resulting ordering can equivalently be described as the order one would get from a depth-first traversal of a quadtree; because of its close connection with quadtrees, the Z-ordering can be used to efficiently construct quadtrees and related higher dimensional data structures.^[2]



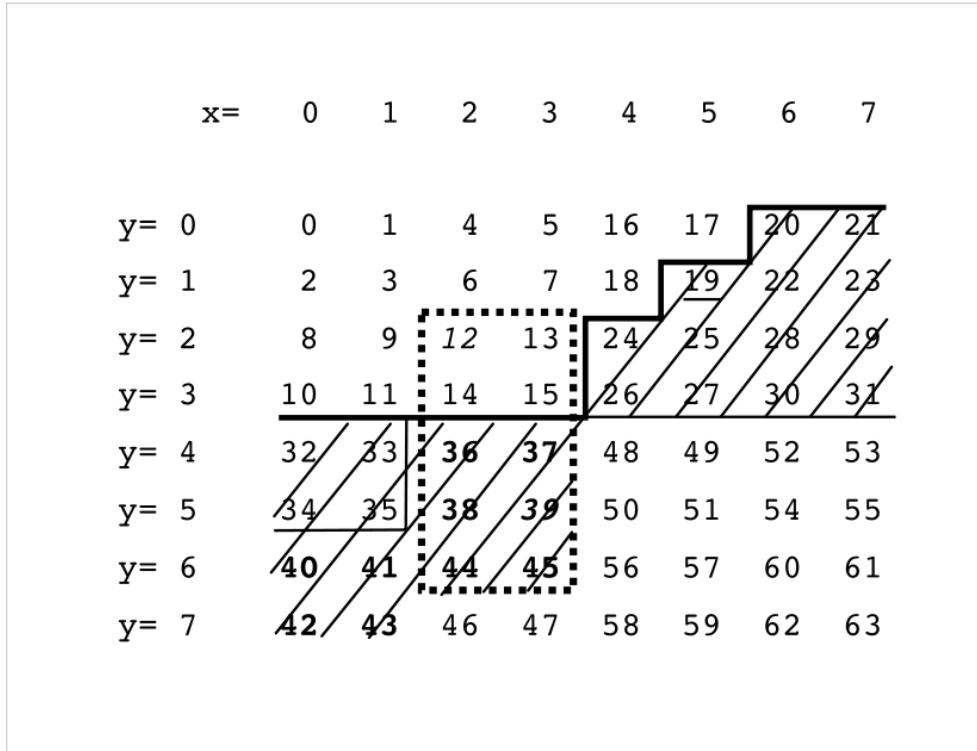
Coordinate values

The figure below shows the Z-values for the two dimensional case with integer coordinates $0 \leq x \leq 7$, $0 \leq y \leq 7$ (shown both in decimal and binary). Interleaving the binary coordinate values yields binary z -values as shown. Connecting the z -values in their numerical order produces the recursively Z-shaped curve.



Use with one-dimensional data structures for range searching

Although well locality preserving, for efficient range searches an algorithm is necessary for calculating, from a point encountered in the data structure, the next Z-value which is in the multidimensional search range:



In this example, the range being queried ($x=2..3$, $y=2..6$) is indicated by the dotted rectangle. Its highest Z-value (MAX) is 45. In this example, the value F=19 is encountered when searching a data structure in increasing Z-value direction, so we would have to search in the interval between F and MAX (hatched area). To speed up the search, one would calculate the next Z-value which is in the search range, called BIGMIN (36 in the example) and only search in the interval between BIGMIN and MAX (bold values), thus skipping most of the hatched area. Searching in decreasing direction is analogous with LITMAX which is the highest Z-value in the query range lower than F. The BIGMIN problem has first been stated and its solution shown in Tropf and Herzog^[3]. This solution is also used in UB-trees ("GetNextZ-address"). As the approach does not depend on the one dimensional data structure chosen, there is still free choice of structuring the data, so well known methods such as balanced trees can be used to cope with dynamic data (in contrast for example to R-trees where special considerations are necessary). Similarly, this independence makes it easier to incorporate the method into existing databases.

Applying the method hierarchically (according to the data structure at hand), optionally in both increasing and decreasing direction, yields highly efficient multidimensional range search which is important in both commercial and technical applications, e.g. as a procedure underlying nearest neighbour searches. Z-order is one of the few multidimensional access methods that has found its way into commercial database systems (Oracle database 1995^[4], Transbase 2000^[5]).

As long ago as 1966, G.M.Morton proposed Z-order for file sequencing of a static two dimensional geographical database. Areal data units are contained in one or a few quadratic frames represented by their sizes and lower right corner Z-values, the sizes complying with the Z-order hierarchy at the corner position. With high probability, changing to an adjacent frame is done with one or a few relatively small scanning steps.

Related structures

As an alternative, the Hilbert curve has been suggested as it has a better order-preserving behaviour, but here the calculations are much more complicated, leading to significant processor overhead. BIGMIN source code for both Z-curve and Hilbert-curve were described in a patent by H. Tropf.^[6]

For a recent overview on multidimensional data processing, including e.g. nearest neighbour searches, see Hanan Samet's textbook.^[7]

Applications in linear algebra

The Strassen algorithm for matrix multiplication is based on splitting the matrices in four blocks, and then recursively each of these blocks in four smaller blocks, until the blocks are single elements (or more practically: until reaching matrices so small that the trivial algorithm is faster). Arranging the matrix elements in Z-order then improves locality, and has the additional advantage (compared to row- or column-major ordering) that the subroutine for multiplying two blocks does not need to know the total size of the matrix, but only the size of the blocks and their location in memory.

See also

- UB-tree
- Hilbert curve
- Hilbert R-tree
- Spatial index
- locality preserving hashing
- Matrix representation

References

- [1] Morton, G. M. (1966), *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, Technical Report, Ottawa, Canada: IBM Ltd..
- [2] Bern, M.; Eppstein, D.; Teng, S.-H. (1999), "Parallel construction of quadtrees and quality triangulations", *Int. J. Comp. Geom. & Appl.* **9** (6): 517–532, doi:10.1142/S0218195999000303.
- [3] Tropf, H.; Herzog, H. (1981), "Multidimensional Range Search in Dynamically Balanced Trees" (<http://www.vision-tools.com/h-tropf/multidimensionalrangequery.pdf>), *Angewandte Informatik* **2**: 71–77, .
- [4] Gaede, Volker; Guenther, Oliver (1998), "Multidimensional access methods" (<http://www-static.cc.gatech.edu/computing/Database/readinggroup/articles/p170-gaede.pdf>), *ACM Computing Surveys* **30** (2): 170–231, doi:10.1145/280277.280279, .
- [5] Ramsak, Frank; Markl, Volker; Fenk, Robert; Zirkel, Martin; Elhardt, Klaus; Bayer, Rudolf (2000), "Integrating the UB-tree into a Database System Kernel" (<http://www.mistral.in.tum.de/results/publications/RMF+00.pdf>), *Int. Conf. on Very Large Databases (VLDB)*, pp. 263–272, .
- [6] Tropf, H., "Database system and method for organizing data elements according to a Hilbert curve", US 7321890 (<http://v3.espacenet.com/textdoc?DB=EPODOC&IDX=US7321890>), issued January 22, 2008.
- [7] Samet, H. (2006), *Foundations of Multidimensional and Metric Data Structures*, San Francisco: Morgan-Kaufmann.

UB-tree

The **UB-tree** as proposed by Rudolf Bayer and Volker Markl is a balanced tree for storing and efficiently retrieving multidimensional data. It is basically a B+ tree (information only in the leaves) with records stored according to Z-order, also called Morton order. Z-order is simply calculated by bitwise interleaving the keys.

Insertion, deletion, and point query are done as with ordinary B+ trees. To perform range searches in multidimensional point data, however, an algorithm must be provided for calculating, from a point encountered in the data base, the next Z-value which is in the multidimensional search range.

The original algorithm to solve this key problem was exponential with the dimensionality and thus not feasible^[1] ("GetNextZ-address"). A solution to this "crucial part of the UB-tree range query" linear with the z-address bit length has been described later.^[2] This method has already been described in an older paper^[3] where using Z-order with search trees has first been proposed.

References

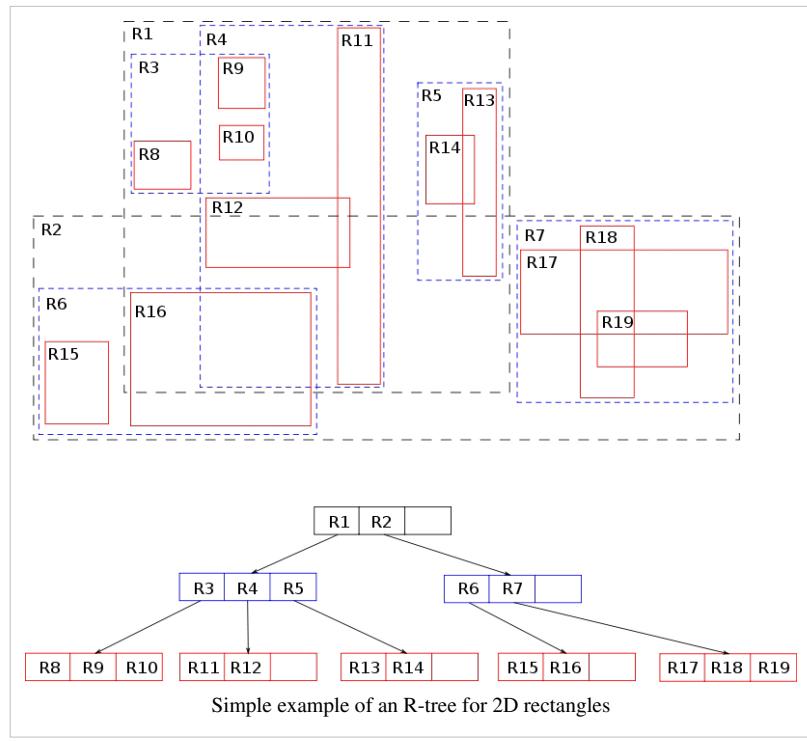
- [1] Markl, V. (1999). *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique* (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.6487>)..
- [2] Ramsak, Frank; Markl, Volker; Fenk, Robert; Zirkel, Martin; Elhardt, Klaus; Bayer, Rudolf (September 10–14, 2000). "Integrating the UB-tree into a Database System Kernel" (<http://www.vldb.org/dblp/db/conf/vldb/RamsakMFZEB00.html>). 26th International Conference on Very Large Data Bases. pp. 263–272. .
- [3] Tropf, H.; Herzog, H. "Multidimensional Range Search in Dynamically Balanced Trees" (<http://www.vision-tools.com/h-tropf/multidimensionalrangequery.pdf>) (PDF). *Angewandte Informatik (Applied Informatics)* (2/1981): 71–77. ISSN 0013-5704. .

R-tree

R-trees are tree data structures that are similar to B-trees, but are used for spatial access methods, i.e., for indexing multi-dimensional information; for example, the (X, Y) coordinates of geographical data. A common real-world usage for an R-tree might be: "Find all museums within 2 kilometres (1.2 mi) of my current location".

The data structure splits space with hierarchically nested, and possibly overlapping, minimum bounding rectangles (MBRs, otherwise known as bounding boxes, i.e. "rectangle", what the "R" in R-tree stands for).

Each node of an R-tree has a variable number of entries (up to some pre-defined maximum).



Each entry

within a non-leaf node stores two pieces of data: a way of identifying a child node, and the bounding box of all entries within this child node.

The insertion and deletion algorithms use the bounding boxes from the nodes to ensure that "nearby" elements are placed in the same leaf node (in particular, a new element will go into the leaf node that requires the least enlargement in its bounding box). Each entry within a leaf node stores two pieces of information; a way of identifying the actual data element (which, alternatively, may be placed directly in the node), and the bounding box of the data element.

Similarly, the searching algorithms (e.g., intersection, containment, nearest) use the bounding boxes to decide whether or not to search inside a child node. In this way, most of the nodes in the tree are never "touched" during a search. Like B-trees, this makes R-trees suitable for databases, where nodes can be paged to memory when needed.

Different algorithms can be used to split nodes when they become too full, resulting in the *quadratic* and *linear* R-tree sub-types.

R-trees do not historically guarantee good worst-case performance, but generally perform well with real-world data. However, a new algorithm was published in 2004 that defines the Priority R-Tree, which claims to be as efficient as the currently most efficient methods and is at the same time worst-case optimal.

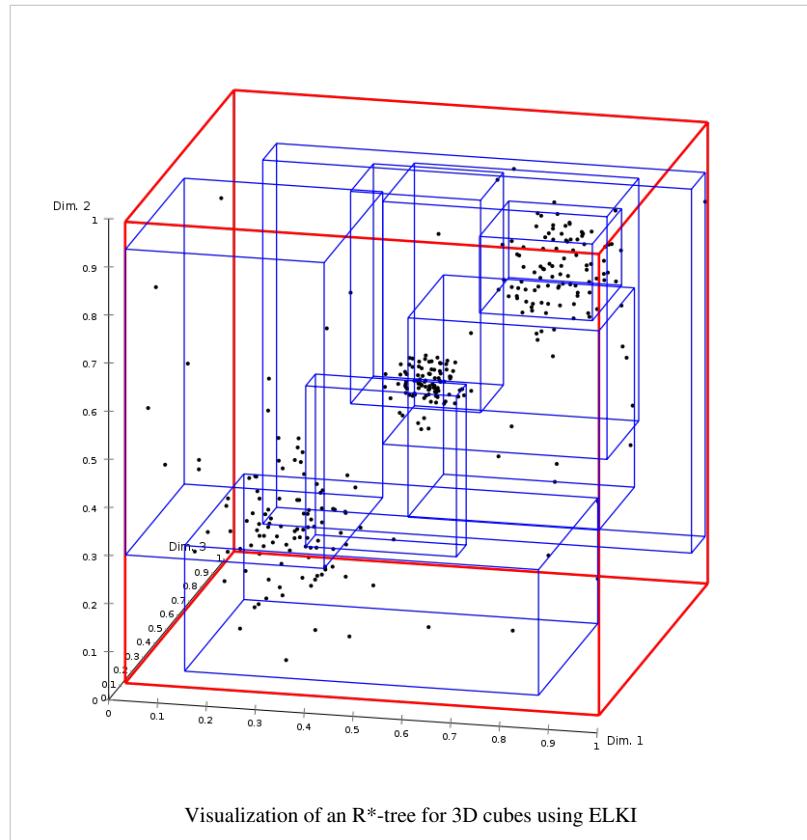
Variants

- R* tree
- R+ tree
- Hilbert R-tree
- Priority R-Tree (PR-Tree) - The PR-tree performs similarly to the best known R-tree variants on real-life and relatively evenly distributed data, but outperforms them significantly on more extreme data.^[1]

Algorithm

Search

The input is a search rectangle (Query box). Searching is quite similar to searching in a B+tree. The search starts from the root node of the tree. Every internal node contains a set of rectangles and pointers to the corresponding child node and every leaf node contains the rectangles of spatial objects (the pointer to some spatial object can be there). For every rectangle in a node, it has to be decided if it overlaps the search rectangle or not. If yes, the corresponding child node has to be searched also. Searching is done like this in a recursive manner until all overlapping nodes have been traversed. When a leaf node is reached, the contained bounding boxes (rectangles) are



Visualization of an R*-tree for 3D cubes using ELKI

tested against the search rectangle and their objects (if there are any) are put into the result set if they lie within the search rectangle.

Insertion

To insert an object, the tree is traversed recursively from the root node. All rectangles in the current internal node are examined. The constraint of least coverage is employed to insert an object, i.e., the box that needs least enlargement to enclose the new object is selected. In the case where there is more than one rectangle that meets this criterion, the one with the smallest area is chosen. Inserting continues recursively in the chosen node. Once a leaf node is reached, a straightforward insertion is made if the leaf node is not full. If the leaf node is full, it must be split before the insertion is made. A few splitting algorithms have been proposed for good R-tree performance.

Bulk-loading

- Sort-Tile-Recursive (STR)^[2]
- Packed Hilbert R-Tree - Uses the Hilbert value of the center of a rectangle to sort the leaf nodes and recursively builds the tree.
- Nearest-X - Rectangles are sorted on the x-coordinate and nodes are created.

See also

- Segment tree
- Interval tree - A degenerate R-Tree for 1 dimension (usually time).
- Bounding volume hierarchy
- Spatial index
- GiST

References

- [1] Lars Arge, Mark de Berg, Herman J. Haverkort, Ke Yi: The Priority RTree: A Practically Efficient and WorstCase Optimal RTree (<http://www.win.tue.nl/~mdberg/Papers/prtree.pdf>)
- [2] Scott T. Leutenegger, Jeffrey M. Edgington and Mario A. Lopez: STR: A Simple and Efficient Algorithm for R-Tree Packing (<http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=4D62F569DDC2B520D1658983F40AC9DC?doi=10.1.1.106.4996&rep=rep1&type=pdf>)
- Antonin Guttman: *R-Trees: A Dynamic Index Structure for Spatial Searching*, Proc. 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–57. ISBN 0-89791-128-8
- Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, Yannis Theodoridis: *R-Trees: Theory and Applications*, Springer, 2005. ISBN 1-85233-977-2
- N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331 (<http://dbs.mathematik.uni-marburg.de/publications/myPapers/1990/BKSS90.pdf>)

External links

- R-tree portal (<http://www.rtreeportal.org/>)
- R-Trees: A Dynamic Index Structure for Spatial Searching (<http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf>)
- R-Tree implementations: C & C++ (<http://superliminal.com/sources/sources.htm#C&C++Code>), Java applet (<http://gis.umb.no/gis/applets/rtree2/jdk1.1/>), Common Lisp (<http://www.cliki.net/spatial-trees>), Python (<http://pypi.python.org/pypi/Rtree/>), Javascript (<http://stackulator.com/rtree/>).

R+ tree

In computer science, an **R+ tree** is a tree data structure, a variant of the R tree, used for indexing spatial information.

Difference between R+ trees and R trees

R+ trees are a compromise between R-trees; and kd-trees; they avoid overlapping of internal nodes by inserting an object into multiple leaves if necessary.

R+ trees differ from R trees in that:

- Nodes are not guaranteed to be at least half filled
- The entries of any internal node do not overlap
- An object ID may be stored in more than one leaf node

Advantages

- Because nodes are not overlapped with each other, point query performance benefits since all spatial regions are covered by at most one node.
- A single path is followed and fewer nodes are visited than with the R-tree

Disadvantages

- Since rectangles are duplicated, an R+ tree can be larger than an R tree built on same data set.
- Construction and maintenance of R+ trees is more complex than the construction and maintenance of R trees and other variants of the R tree.

See also

- R tree
- R* tree
- Hilbert R-tree
- kd-tree

References

T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects^[1]. In VLDB, 1987.

References

[1] <http://citeseer.ist.psu.edu/sellis87rtree.html>

R* tree

R*-trees are a variant of R-trees used for indexing spatial information. R*-trees support point and spatial data at the same time with a slightly higher cost than other R-trees. It was proposed by Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger in 1990.

Difference between R*-trees and R-trees

Minimization of both coverage and overlap is crucial to the performance of R-trees. The R*-tree attempts to reduce both, using a combination of a revised node split algorithm and the concept of forced reinsertion at node overflow. This is based on the observation that R-tree structures are highly susceptible to the order in which their entries are inserted, so an insertion-built (rather than bulk-loaded) structure is likely to be sub-optimal. Deletion and reinsertion of entries allows them to "find" a place in the tree that may be more appropriate than their original location.

When a node overflows, a portion of its entries are removed from the node and reinserted into the tree. (In order to avoid an indefinite cascade of reinsertions caused by subsequent node overflow, the reinsertion routine may be called only once in each level of the tree when inserting any one new entry.) This has the effect of producing more well-clustered groups of entries in nodes, reducing node coverage. Furthermore, actual node splits are often postponed, causing average node occupancy to rise.

Performance

- Likely significant improvement over other R tree variants, but there is overhead due to the reinsertion method.
- Efficiently supports point and spatial data at the same time

Algorithm

The R*-tree uses the same algorithm as the R-tree for query and delete operations. The primary difference is the insert algorithm, specifically how it chooses which branch to insert the new node into and the methodology for splitting a node that is full.

See also

- R-tree
- Hilbert R-tree
- R+ tree
- X-tree

References

- Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331 ^[1]

External links

- R-tree Demo [2]
- The TPIE Library contains a C++ R* tree implementation [3]
- A header-only C++ R* Tree Implementation [4]
- Java and C++ implementation are in the Spatial Index Library [5]

References

- [1] <http://dbs.mathematik.uni-marburg.de/publications/myPapers/1990/BKSS90.pdf>
- [2] <http://donar.umiacs.umd.edu/quadtree/points/rtrees.html>
- [3] <http://www.cs.duke.edu/TPIE/>
- [4] <http://www.virtualroadside.com/blog/index.php/2008/10/04/r-tree-implementation-for-cpp/>
- [5] <http://research.att.com/~marioh/spatialindex/>

Hilbert R-tree

Hilbert R-tree, an R-tree variant, is an index for multidimensional objects like lines, regions, 3-D objects, or high dimensional feature-based parametric objects. It can be thought of as an extension to B+-tree for multidimensional objects.

The performance of R-trees depends on the quality of the algorithm that clusters the data rectangles on a node. Hilbert R-trees use space-filling curves, and specifically the Hilbert curve, to impose a linear ordering on the data rectangles.

There are two types of Hilbert R-tree, one for static database and one for dynamic databases. In both cases, space filling curves and specifically the Hilbert curve are used to achieve better ordering of multidimensional objects in the node. This ordering has to be ‘good’, in the sense that it should group ‘similar’ data rectangles together, to minimize the area and perimeter of the resulting minimum bounding rectangles (MBRs). Packed Hilbert R-trees are suitable for static databases in which updates are very rare or in which there are no updates at all.

The dynamic Hilbert R-tree is suitable for dynamic databases where insertions, deletions, or updates may occur in real time. Moreover, dynamic Hilbert R-trees employ flexible deferred splitting mechanism to increase the space utilization. Every node has a well defined set of sibling nodes. By adjusting the split policy the Hilbert R-tree can achieve a degree of space utilization as high as is desired. This is done by proposing an ordering on the R-tree nodes. The Hilbert R-tree sorts rectangles according to the Hilbert value of the center of the rectangles (i.e., MBR). (The Hilbert value of a point is the length of the Hilbert curve from the origin to the point.) Given the ordering, every node has a well-defined set of sibling nodes; thus, deferred splitting can be used. By adjusting the split policy, the Hilbert R-tree can achieve as high utilization as desired. To the contrary, other R-tree variants have no control over the space utilization.

The basic idea

Although the following example is for a static environment, it explains the intuitive principles for good R-tree design. These principles are valid for both static and dynamic databases. Roussopoulos and Leifker proposed a method for building a packed R-tree that achieves almost 100% space utilization. The idea is to sort the data on the x or y coordinate of one of the corners of the rectangles. Sorting on any of the four coordinates gives similar results. In this discussion points or rectangles are sorted on the x coordinate of the lower left corner of the rectangle. In the discussion below the Roussopoulos and Leifker’s method is referred to as the lowx packed R-tree. The sorted list of rectangles is scanned; successive rectangles are assigned to the same R-tree leaf node until that node is full; a new leaf node is then created and the scanning of the sorted list continues. Thus, the nodes of the resulting R-tree will be

fully packed, with the possible exception of the last node at each level. Thus, the utilization is $\approx 100\%$. Higher levels of the tree are created in a similar way.

Figure 1 highlights the problem of the lowx packed R-tree. Figure 1[Right] shows the leaf nodes of the R-tree that the lowx packing method will create for the points of Figure 1 [Left]. The fact that the resulting father nodes cover little area explains why the lowx packed R-tree achieves excellent performance for point queries. However, the fact that the fathers have large perimeters, explains the degradation of performance for region queries. This is consistent with the analytical formulas for R-tree performance^[1]. Intuitively, the packing algorithm should ideally assign nearby points to the same leaf node. Ignorance of the y coordinate by the lowx packed R-tree tends to violate this empirical rule.

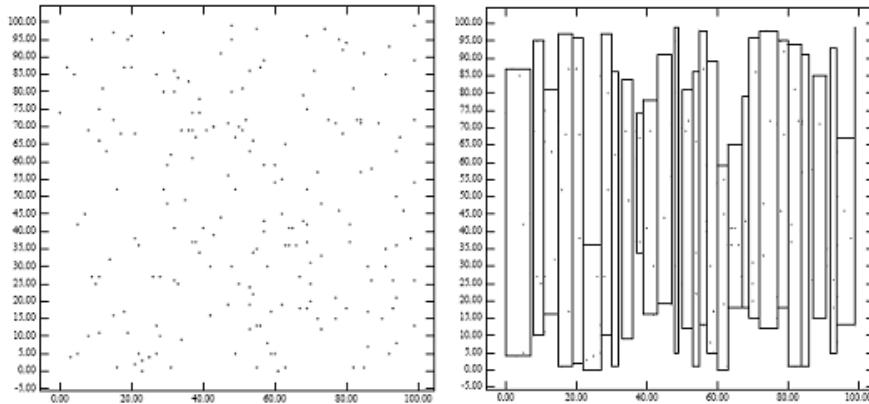


Figure 1: [Left] 200 points uniformly distributed; [Right] MBR of nodes generated by the '*lowx packed R-tree*' algorithm

This section describes two variants of the Hilbert R-trees. The first index is suitable for the static database in which updates are very rare or in which there are no updates at all. The nodes of the resulting R-tree will be fully packed, with the possible exception of the last node at each level. Thus, the space utilization is $\approx 100\%$; this structure is called a packed Hilbert R-tree. The second index, called a Dynamic Hilbert R-tree, supports insertions and deletions, and is suitable for a dynamic environment.

Packed Hilbert R-trees

The following provides a brief introduction to the Hilbert curve. The basic Hilbert curve on a 2×2 grid, denoted by H_1 is shown in Figure 2. To derive a curve of order i , each vertex of the basic curve is replaced by the curve of order $i - 1$, which may be appropriately rotated and/or reflected. Figure 2 also shows the Hilbert curves of order two and three. When the order of the curve tends to infinity, like other space filling curves, the resulting curve is a fractal, with a fractal dimension of two^{[2], [1]}. The Hilbert curve can be generalized for higher dimensionalities. Algorithms for drawing the two-dimensional curve of a given order can be found in^[3] and^[2]. An algorithm for higher dimensionalities is given in^[4].

The path of a space filling curve imposes a linear ordering on the grid points; this path may be calculated by starting at one end of the curve and following the path to the other end. The actual coordinate values of each point can be calculated. However, for the Hilbert curve this is much harder than for example the Z-order curve. Figure 2 shows one such ordering for a 4×4 grid (see curve H_2). For example, the point $(0,0)$ on the H_2 curve has a Hilbert value of 0, while the point $(1,1)$ has a Hilbert value of 2.

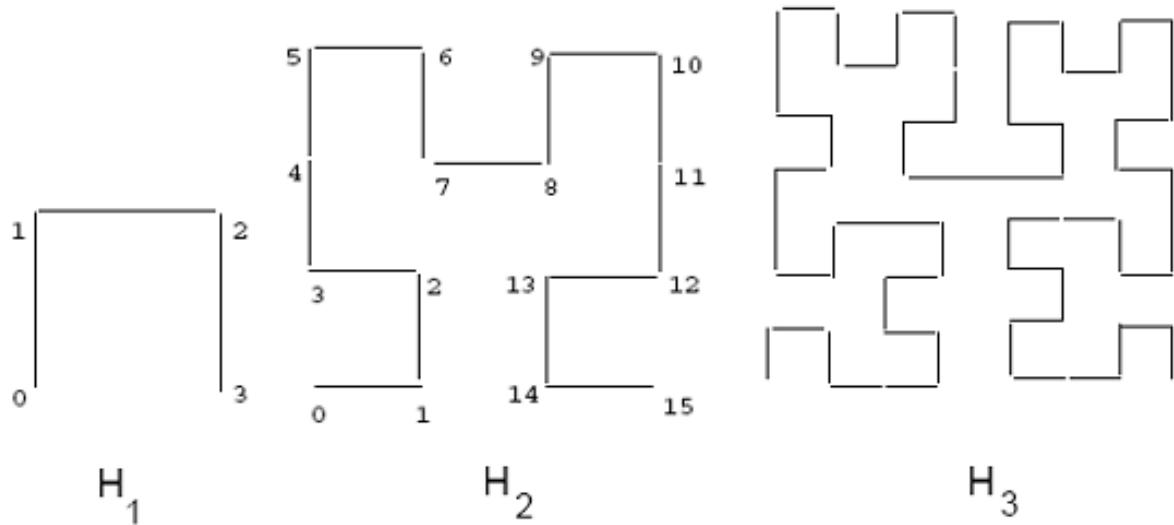


Figure 2: Hilbert curves of order 1, 2, and 3

The Hilbert curve imposes a linear ordering on the data rectangles and then traverses the sorted list, assigning each set of C rectangles to a node in the R-tree. The final result is that the set of data rectangles on the same node will be close to each other in the linear ordering, and most likely in the native space; thus, the resulting R-tree nodes will have smaller areas. Figure 2 illustrates the intuitive reasons why our Hilbert-based methods will result in good performance. The data is composed of points (the same points as given in Figure 1). By grouping the points according to their Hilbert values, the MBRs of the resulting R-tree nodes tend to be small square-like rectangles. This indicates that the nodes will likely have small area and small perimeters. Small area values result in good performance for point queries; small area and small perimeter values lead to good performance for larger queries.

Algorithm Hilbert-Pack

(packs rectangles into an R-tree)

Step 1. Calculate the Hilbert value for each data rectangle

Step 2. Sort data rectangles on ascending Hilbert values

Step 3. /* Create leaf nodes (level l=0) */

- While (there are more rectangles)
 - generate a new R-tree node
 - assign the next C rectangles to this node
- Step 4. /* Create nodes at higher level (l + 1) */
- While (there are > 1 nodes at level l)
 - sort nodes at level l ≥ 0 on ascending
 - creation time
 - repeat Step 3

The assumption here is that the data are static or the frequency of modification is low. This is a simple heuristic for constructing an R-tree with 100% space utilization which at the same time will have as good response time as possible.

Dynamic Hilbert R-trees

The performance of R-trees depends on the quality of the algorithm that clusters the data rectangles on a node. Hilbert R-trees use space-filling curves, and specifically the Hilbert curve, to impose a linear ordering on the data rectangles. The Hilbert value of a rectangle is defined as the Hilbert value of its center.

Tree structure

The Hilbert R-tree has the following structure. A leaf node contains at most C_1 entries each of the form $(R, \text{obj_id})$ where C_1 is the capacity of the leaf, R is the MBR of the real object $(x_{\text{low}}, x_{\text{high}}, y_{\text{low}}, y_{\text{high}})$ and obj_id is a pointer to the object description record. The main difference between the Hilbert R-tree and the R*-tree [5] is that non-leaf nodes also contain information about the LHV's (Largest Hilbert Value). Thus, a non-leaf node in the Hilbert R-tree contains at most C_n entries of the form $(R, \text{ptr}, \text{LHV})$ where C_n is the capacity of a non-leaf node, R is the MBR that encloses all the children of that node, ptr is a pointer to the child node, and LHV is the largest Hilbert value among the data rectangles enclosed by R . Notice that since the non-leaf node picks one of the Hilbert values of the children to be the value of its own LHV, there is not extra cost for calculating the Hilbert values of the MBR of non-leaf nodes. Figure 3 illustrates some rectangles organized in a Hilbert R-tree. The Hilbert values of the centers are the numbers near the 'x' symbols (shown only for the parent node 'II'). The LHV's are in brackets. Figure 4 shows how the tree of Figure 3 is stored on the disk; the contents of the parent node 'II' are shown in more detail. Every data rectangle in node 'I' has a Hilbert value $v \leq 33$; similarly every rectangle in node 'II' has a Hilbert value greater than 33 and ≤ 107 , etc.

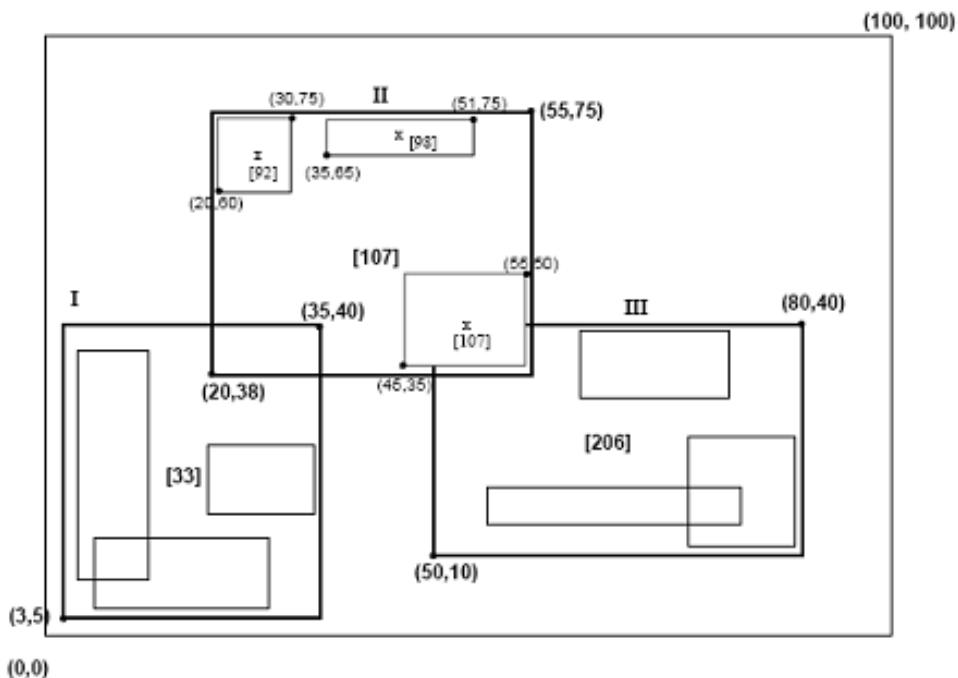


Figure 3: Data rectangles organized in a Hilbert R-tree (Hilbert values and LHV's are in Brackets)

A plain R-tree splits a node on overflow, creating two nodes from the original one. This policy is called a 1-to-2 splitting policy. It is possible also to defer the split, waiting until two nodes split into three. Note that this is similar to the B*-tree split policy. This method is referred to as the 2-to-3 splitting policy. In general, this can be extended to s -to- $(s+1)$ splitting policy; where s is the order of the splitting policy. To implement the order- s splitting policy, the overflowing node tries to push some of its entries to one of its $s - 1$ siblings; if all of them are full, then s -to- $(s+1)$ split need to be done. The $s - 1$ siblings are called the cooperating siblings. Next, the algorithms for searching, insertion, and overflow handling are described in details.

Searching

The searching algorithm is similar to the one used in other R-tree variants. Starting from the root, it descends the tree and examines all nodes that intersect the query rectangle. At the leaf level, it reports all entries that intersect the query window w as qualified data items.

Algorithm Search(node Root, rect w):

S1. Search nonleaf nodes:

Invoke Search for every entry whose MBR intersects the query window w.

S2. Search leaf nodes:

Report all entries that intersect the query window w as candidates.

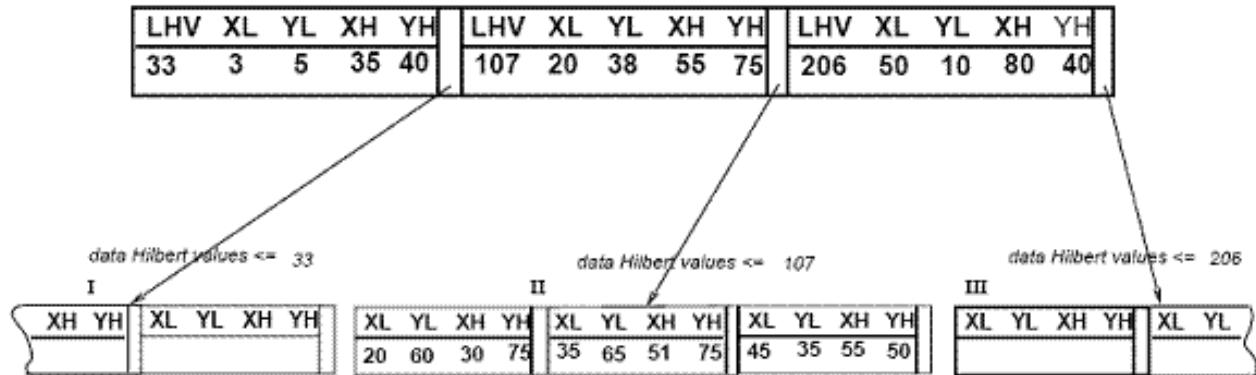


Figure 4: The file structure for the Hilbert R-tree

Insertion

To insert a new rectangle r in the Hilbert R-tree, the Hilbert value h of the center of the new rectangle is used as a key. At each level the node with the minimum LHV of all its siblings is chosen. When a leaf node is reached, the rectangle r is inserted in its correct order according to h. After a new rectangle is inserted in a leaf node N, **AdjustTree** is called to fix the MBR and LHV values in the upper-level nodes.

Algorithm Insert(node Root, rect r): /* Inserts a new rectangle r in the Hilbert R-tree. h is the Hilbert value of the rectangle*/

I1. Find the appropriate leaf node:

Invoke ChooseLeaf(r, h) to select a leaf node L in which to place r.

I2. Insert r in a leaf node L:

If L has an empty slot, insert r in L in the appropriate place according to the Hilbert order and return.
If L is full, invoke HandleOverflow(L,r), which will return new leaf if split was inevitable,

I3. Propagate changes upward:

Form a set S that contains L, its cooperating siblings and the new leaf (if any)
Invoke AdjustTree(S).

I4. Grow tree taller:

If node split propagation caused the root to split, create

a new root whose children are the two resulting nodes.

Algorithm ChooseLeaf(rect r, int h):

/* Returns the leaf node in which to place a new rectangle r. */

C1. Initialize:

Set N to be the root node.

C2. Leaf check:

If N is a leaf_ return N.

C3. Choose subtree:

If N is a non-leaf node, choose the entry (R, ptr, LHV)

with the minimum LHV value greater than h.

C4. Descend until a leaf is reached:

Set N to the node pointed by ptr and repeat from C2.

Algorithm AdjustTree(set S):

/* S is a set of nodes that contains the node being updated, its cooperating siblings (if overflow has occurred) and the newly created node NN (if split has occurred). The routine ascends from the leaf level towards the root, adjusting MBR and LHV of nodes that cover the nodes in S. It propagates splits (if any) */

A1. If root level is reached, stop.
A2. Propagate node split upward:

Let Np be the parent node of N.

If N has been split, let NN be the new node.

Insert NN in Np in the correct order according to its Hilbert value if there is room. Otherwise, invoke HandleOverflow(Np , NN).

If Np is split, let PP be the new node.

A3. Adjust the MBR's and LHV's in the parent level:

Let P be the set of parent nodes for the nodes in S.

Adjust the corresponding MBR's and LHV's of the nodes in P appropriately.

A4. Move up to next level:

Let S become the set of parent nodes P, with

NN = PP, if Np was split.

repeat from A1.

Deletion

In the Hilbert R-tree there is no need to re-insert orphaned nodes whenever a father node underflows. Instead, keys can be borrowed from the siblings or the underflowing node is merged with its siblings. This is possible because the nodes have a clear ordering (according to Largest Hilbert Value, LHV); in contrast, in R-trees there is no such concept concerning sibling nodes. Notice that deletion operations require s cooperating siblings, while insertion operations require s - 1 siblings.

Algorithm Delete(r):

D1. Find the host leaf:

Perform an exact match search to find the leaf node L

that contains r.

D2. Delete r :

 Remove r from node L.

D3. If L underflows

 borrow some entries from s cooperating siblings.

 if all the siblings are ready to underflow.

 merge s + 1 to s nodes,

 adjust the resulting nodes.

D4. Adjust MBR and LHV in parent levels.

 form a set S that contains L and its cooperating

 siblings (if underflow has occurred).

 invoke **AdjustTree(S)**.

Overflow handling

The overflow handling algorithm in the Hilbert R-tree treats the overflowing nodes either by moving some of the entries to one of the s - 1 cooperating siblings or by splitting s nodes into s + 1 nodes.

Algorithm HandleOverflow(node N, rect r):

/* return the new node if a split occurred. */

H1. Let ε be a set that contains all the entries from N

 and its s - 1 cooperating siblings.

H2. Add r to ε.

H3. If at least one of the s - 1 cooperating siblings is not full,

 distribute ε evenly among the s nodes according to Hilbert values.

H4. If all the s cooperating siblings are full,

 create a new node NN and

 distribute ε evenly among the s + 1 nodes according

 to Hilbert values

 return NN.

See also

- R tree
- R* tree
- R+ tree

Notes and references

- [1] I. Kamel and C. Faloutsos, On Packing R-trees, Second International ACM Conference on Information and Knowledge Management (CIKM), pages 490–499, Washington D.C., 1993.
- [2] H. Jagadish. Linear clustering of objects with multiple attributes. In Proc. of ACM SIGMOD Conf., pages 332–342, Atlantic City, NJ, May 1990.
- [3] J. Griffiths. An algorithm for displaying a class of space-filling curves, Software-Practice and Experience 16(5), 403–411, May 1986.
- [4] T. Bially. Space-filling curves. Their generation and their application to bandwidth reduction. IEEE Trans. on Information Theory. IT15(6), 658–664, November 1969.
- [5] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In Proc. of ACM SIGMOD, pages 322–331, Atlantic City, NJ, May 1990.

- I. Kamel and C. Faloutsos. Parallel R-Trees. In Proc. of ACM SIGMOD Conf., pages 195–204 San Diego, CA, June 1992. Also available as Tech. Report UMIACS TR 92-1, CS-TR-2820.
- I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In Proc. of VLDB Conf., pages 500–509, Santiago, Chile, September 1994. Also available as Tech_Report UMIACS TR 93-12.1 CS-TR-3032.1.
- N. Koudas, C. Faloutsos and I. Kamel. Declustering Spatial Databases on a Multi-Computer Architecture, International Conference on Extending Database Technology (EDBT), pages 592–614, 1996.
- N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using Packed R-trees. In Proc. of ACM SIGMOD, pages 17–31, Austin, TX, May 1985.
- M. Schroeder. Fractals, Chaos, Power Laws: Minutes From an Infinite Paradise. W.H. Freeman and Company, NY, 1991.
- T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: a dynamic index for multi-dimensional objects. In Proc. 13th International Conference on VLDB, pages 507–518, England, September 1987.

X-tree

In computer science, an **X-tree** is an index tree structure based on the R-tree used for storing data in many dimensions. It differs from R-trees, R+-trees and R*-trees because it emphasizes prevention of overlap in the bounding boxes. In cases where nodes cannot be split without preventing overlap, the node split will be deferred, resulting in **super-nodes**. In extreme cases, the tree will linearize, which defends against worst-case behaviors observed in some other data structures.

External links

- The X-tree: An Index Structure for High-Dimensional Data ^[1]
- A X-tree reference implementation in C++, written by Hans-Peter Kriegel ^[2]

References

- [1] <http://www.vldb.org/conf/1996/P028.PDF>
[2] <http://www.rtreeportal.org/code/xtree.tar.gz>

Metric tree

A **metric tree** is any tree data structure specialized to index data in metric spaces. Metric trees exploit properties of metric spaces such as the triangle inequality to make accesses to the data more efficient. Examples include vp-trees, cover trees, m-trees^[1] and bk trees.

External links

- An M-Tree Applet^[2]

References

- [1] <http://www-db.deis.unibo.it/Mtree/>
[2] <http://www.cmarschner.net/mmtree.html>

VP-tree

A **vantage point tree**, or **vp-tree** is a BSP tree that segregates data in a metric space by choosing a position in the space (the "vantage point") and dividing the data points into two partitions: those that are nearer to the vantage point than a threshold, and those that are not. By repeatedly applying this procedure to partition the data into smaller and smaller sets, a tree data structure is created where neighbors in the tree are likely to be neighbors in the space.^[1]

This iterative partitioning process is similar to that of a kd-tree, but uses circular (or spherical, hyperspherical, etc) rather than rectilinear partitions. In 2D Euclidean space, this can be visualized as a series of circles segregating the data.

The vp-tree is particularly useful in dividing data in a non-standard metric space into a BSP tree.

References

- [1] Yianilos, Peter N. (1993). "Data structures and algorithms for nearest neighbor search in general metric spaces" (<http://pnylab.com/pny/papers/vptree/vptree/>). *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 311–321. pny93. . Retrieved 2008-08-22.

BK-tree

A **BK-tree** is a metric tree suggested by Burkhard and Keller^{BK73} specifically adapted to discrete metric spaces. For simplicity, let us consider **integer** discrete metric $\varrho(x, y)$. Then, BK-tree is defined in the following way. An arbitrary element a is selected as root node. Root node may have zero or more subtrees. The k -th subtree is recursively built of all elements b such that $\varrho(a, b) = k$. BK-trees can be used for approximate string matching in a dictionary^{BN98}.

References

- W. Burkhard and R. Keller. Some approaches to best-match file searching, CACM, 1973^[1]
- R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed queries trees. In M. Crochemore and D. Gusfield, editors, 5th Combinatorial Pattern Matching, LNCS 807, pages 198-212, Asilomar, CA, June 1994.
- Ricardo Baeza-Yates and Gonzalo Navarro. Fast Approximate String Matching in a Dictionary. Proc. SPIRE'98 [2]

External links

- A BK-tree implementation in Common Lisp^[3] with cool test results and performance graphs.
- BK-tree implementation written in Python^[4] and its a port to Haskell^[5].
- A good explanation of BK-Trees and their relationship to metric spaces [6]

References

- [1] <http://doi.acm.org/10.1145/362003.362025>
- [2] <http://citeseer.ist.psu.edu/1593.html>
- [3] <http://cliki.net/bk-tree>
- [4] <http://hupp.org/adam/weblog/?p=103>
- [5] <http://hupp.org/adam/hg/bktree/file/tip/BKTree.hs>
- [6] <http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-Trees>

Hashes

Hash table

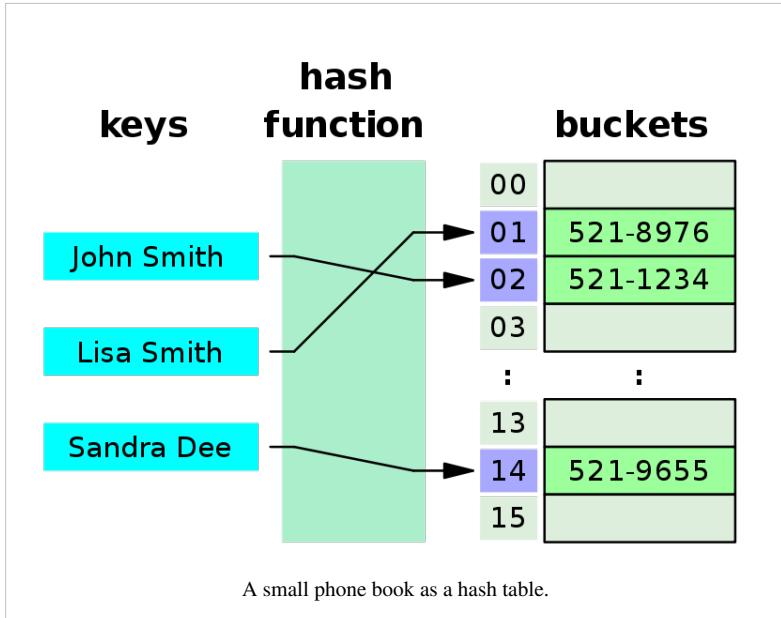
In computer science, a **hash table** or **hash map** is a data structure that uses a hash function to efficiently map certain identifiers or keys (e.g., person names) to associated values (e.g., their telephone numbers). The hash function is used to transform the key into the index (the *hash*) of an array element (the *slot* or *bucket*) where the corresponding value is to be sought.

Ideally the hash function should map each possible key to a different slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after creation). Most hash table designs assume that *hash collisions*—pairs of different keys with the same hash values—are normal occurrences and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average (indeed, amortized^[1]) cost per operation.^{[2] [3]}

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

Hash tables should not be confused with the hash lists and hash trees used in cryptography and data transmission.



Hash function

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

```
index = f(key, arrayLength)
```

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

Choosing a good hash function

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

A basic requirement is that the function should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions, and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests, e.g., a chi-squared test.^[4]

The distribution needs to be uniform only for table sizes s that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of s , the hash function needs to be uniform only when s is a power of two. On the other hand, some hashing algorithms provide uniform hashes only when s is a prime number.^[5]

For open addressing schemes, the hash function should also avoid *clustering*, the mapping of two or more keys to consecutive slots. Such clustering may cause the lookup cost to skyrocket, even if the load factor is low and collisions are infrequent. The popular multiplicative hash^[2] is claimed to have particularly poor clustering behavior.^[5]

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

Some authors claim that good hash functions should have the avalanche effect; that is, a single-bit change in the input key should affect, on average, half the bits in the output. Some popular hash functions do not have this property.^[4]

Perfect hash function

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

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

Collision resolution

Collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2500 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday paradox there is a 95% chance of at least two of the keys being hashed to the same slot.

Therefore, most hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

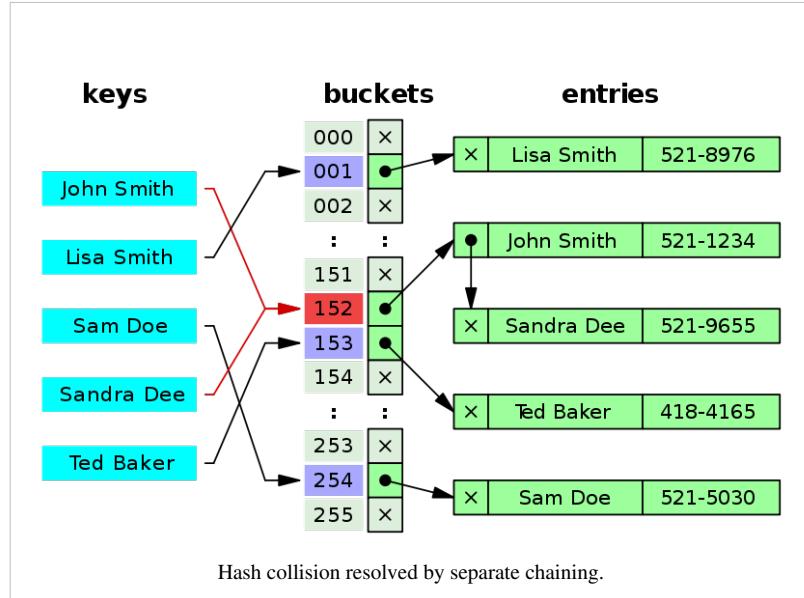
Load factor

The performance of most collision resolution methods does not depend directly on the number n of stored entries, but depends strongly on the table's *load factor*, the ratio n/s between n and the size s of its bucket array. With a good hash function, the average lookup cost is nearly constant as the load factor increases from 0 up to 0.7 or so. Beyond that point, the probability of collisions and the cost of handling them increases.

On the other hand, as the load factor approaches zero, the size of the hash table increases with little improvement in the search cost, and memory is wasted.

Separate chaining

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



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

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

Chained hash tables remain effective even when the number of entries n is much higher than the number of slots. Their performance degrades more gracefully (linearly) with the load factor. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.

For separate-chaining, the worst-case scenario is when all entries were inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries; so the worst-case cost is proportional to the number n of entries in the table.

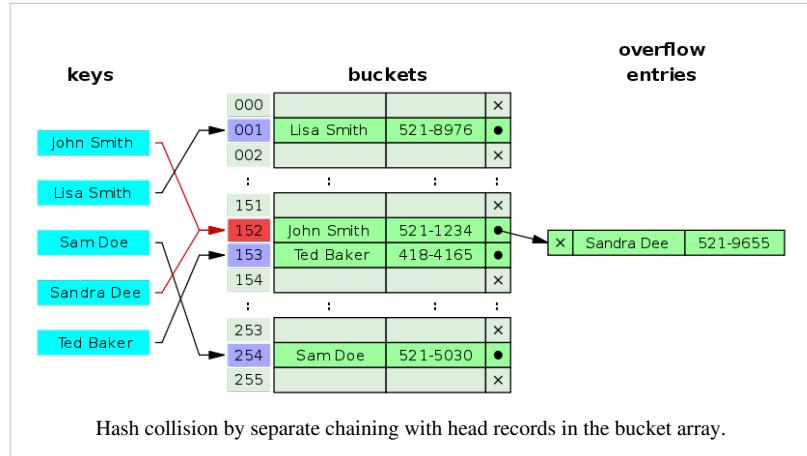
The bucket chains are often implemented as ordered lists, sorted by the key field; this choice approximately halves the average cost of unsuccessful lookups, compared to an unordered list. However, if some keys are much more likely to come up than others, an unordered list with move-to-front heuristic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

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

Separate chaining with list heads

Some chaining implementations store the first record of each chain in the slot array itself.^[3] The purpose is to increase cache efficiency of hash table access. To save memory space, such hash tables often have about as many slots as stored entries, meaning that many slots have two or more entries.

Separate chaining with other structures



Hash collision by separate chaining with head records in the bucket array.

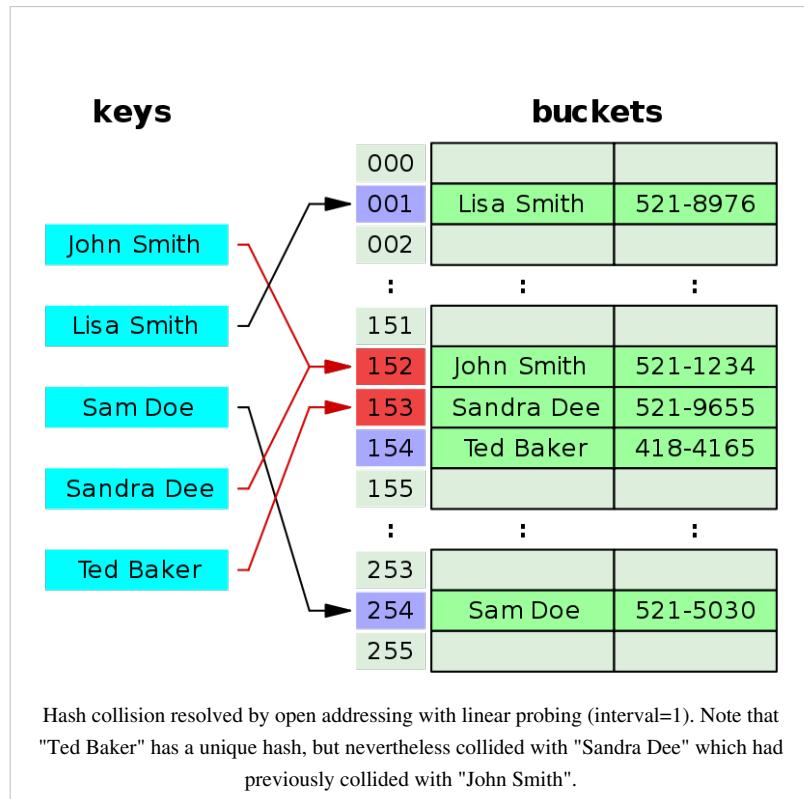
Instead of a list, one can use any other data structure that supports the required operations. By using a self-balancing tree, for example, the theoretical worst-case time of a hash table can be brought down to $O(\log n)$ rather than $O(n)$. However, this approach is only worth the trouble and extra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

The variant called array hashing uses a dynamic array to store all the entries that hash to the same bucket.^{[6] [7]} Each inserted entry gets appended to the end of the dynamic array that is assigned to the hashed slot. This variation makes more effective use of CPU caching, since the bucket entries are stored in sequential memory positions. It also dispenses with the next pointers that are required by linked lists, which saves space when the entries are small, such as pointers or single-word integers.

An elaboration on this approach is the so-called dynamic perfect hashing^[8], where a bucket that contains k entries is organized as a perfect hash table with k^2 slots. While it uses more memory (n^2 slots for n entries, in the worst case), this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion.

Open addressing

In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.^[9] The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called **closed hashing**; it should not be confused with "open hashing" or "closed addressing" which usually mean separate chaining.)



Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by another hash function

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance seriously degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order. Even experienced programmers may find such clustering hard to avoid.

Open addressing only saves memory if the entries are small (less than 4 times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.

Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (which is usually the only one). It also has better locality of reference, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups.

Hash tables with open addressing are also easier to serialize, because they don't use pointers.

On the other hand, normal open addressing is a poor choice for large elements, since these elements fill entire cache lines (negating the cache advantage), and a large amount of space is wasted on large empty table slots. If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records that can be stored within the table (internal storage) and fit in a cache line. They are particularly suitable for elements of one word or less. In cases where the tables are expected to have high load factors, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

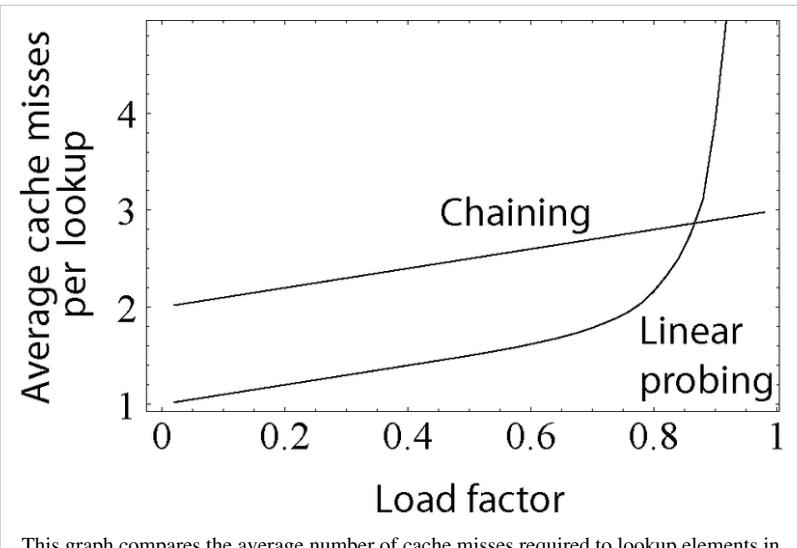
Ultimately, used sensibly, any kind of hash table algorithm is usually fast *enough*; and the percentage of a calculation spent in hash table code is low. Memory usage is rarely considered excessive. Therefore, in most cases the differences between these algorithms is marginal, and other considerations typically come into play.

Coalesced hashing

A hybrid of chaining and open addressing, coalesced hashing links together chains of nodes within the table itself.^[9] Like open addressing, it achieves space usage and (somewhat diminished) cache advantages over chaining. Like chaining, it does not exhibit clustering effects; in fact, the table can be efficiently filled to a high density. Unlike chaining, it cannot have more elements than table slots.

Robin Hood hashing

One interesting variation on double-hashing collision resolution is that of Robin Hood hashing^[10]. The idea is that a key already inserted may be displaced by a new key if its probe count is larger than the key at the current position. The net effect of this is that it reduces worst case search times in the table. This is similar to Knuth's ordered hash tables except the criteria for bumping a key does not depend on a direct relationship between the keys. Since both the worst case and the variance on the number of probes is reduced dramatically an interesting variation is to probe the table starting at the expected successful probe value and then expand from that position in both directions^[11]. External Robin Hashing is an extension of this algorithm where the table is stored in an external file and each table position corresponds to a fixed sized page or bucket with B records^[12].



This graph compares the average number of cache misses required to lookup elements in tables with chaining and linear probing. As the table passes the 80%-full mark, linear probing's performance drastically degrades.

Cuckoo hashing

Another alternative open-addressing solution is cuckoo hashing, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions.

Hopscotch hashing

Another alternative open-addressing solution is hopscotch hashing,^[13] which combines the approaches of cuckoo hashing and linear probing, yet seems in general to avoid their limitations. In particular it works well even when the load factor grows beyond 0.9. The algorithm is well suited for implementing a resizable concurrent hash table.

The hopscotch hashing algorithm works by defining a neighborhood of buckets near the original hashed bucket, where a given entry is always found. Thus, search is limited to the number of entries in this neighborhood, which is logarithmic in the worst case, constant on average, and with proper alignment of the neighborhood typically requires one cache miss. When inserting an entry, one first attempts to add it to a bucket in the neighborhood. However, if all buckets in this neighborhood are occupied, the algorithm traverses buckets in sequence until an open slot (an unoccupied bucket) is found (as in linear probing). At that point, since the empty bucket is outside the neighborhood, items are repeatedly displaced in a sequence of hops (in a manner reminiscent of cuckoo hashing, but with the difference that in this case the empty slot is being moved into the neighborhood, instead of items being moved out with the hope of eventually finding an empty slot). Each hop brings the open slot closer to the original neighborhood, without invalidating the neighborhood property of any of the buckets along the way. In the end the open slot has been moved into the neighborhood, and the entry being inserted can be added to it.

Dynamic resizing

To keep the load factor under a certain limit e.g., under 3/4, many table implementations expand the table when items are inserted. In Java's `HashMap`^[14] class, for example, the default load factor threshold for table expansion is 0.75. Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than one will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays.

Resizing is accompanied by a full or incremental table **rehash** whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table - followed by a **rehash** - when items are deleted. From the point of space-time tradeoffs, this operation is similar to the deallocation in dynamic arrays.

Resizing by copying all entries

A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold r_{\max} . Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold r_{\min} , all entries are moved to a new smaller table.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries n and of the number m of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If m elements are inserted into that table, the total number of extra re-insertions that occur in all dynamic resizings of the table is at most $m-1$. In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

Incremental resizing

Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually:

- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move r elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least $(r + 1)/r$ during resizing.

Other solutions

Linear hashing^[15] is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible look-up functions.

Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hashes, where resizing is prohibitively costly.

Performance analysis

In the simplest model, the hash function is completely unspecified and the table does not resize. For the best possible choice of hash function, a table of size n with open addressing has no collisions and holds up to n elements, with a single comparison for successful lookup, and a table of size n with chaining and k keys has the minimum $\max(0, k-n)$ collisions and $O(1 + k/n)$ comparisons for lookup. For the worst choice of hash function, every insertion causes a collision, and hash tables degenerate to linear search, with $\Omega(k)$ amortized comparisons per insertion and up to k comparisons for a successful lookup.

Adding rehashing to this model is straightforward. As in a dynamic array, geometric resizing by a factor of b implies that only k/b^i keys are inserted i or more times, so that the total number of insertions is bounded above by $bk/(b-1)$, which is $O(k)$. By using rehashing to maintain $k < n$, tables using both chaining and open addressing can have unlimited elements and perform successful lookup in a single comparison for the best choice of hash function.

In more realistic models the hash function is a random variable over a probability distribution of hash functions, and performance is computed on average over the choice of hash function. When this distribution is uniform, the assumption is called "simple uniform hashing" and it can be shown that hashing with chaining requires $\Theta(1 + k/n)$ comparisons on average for an unsuccessful lookup, and hashing with open addressing requires $\Theta(1/(1 - k/n))$.^[16] Both these bounds are constant if we maintain $k/n < c$ using table resizing, where c is a fixed constant less than 1.

Features

Advantages

The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more). Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

If the set of key-value pairs is fixed and known ahead of time (so insertions and deletions are not allowed), one may reduce the average lookup cost by a careful choice of the hash function, bucket table size, and internal data structures. In particular, one may be able to devise a hash function that is collision-free, or even perfect (see below). In this case the keys need not be stored in the table.

Drawbacks

Hash tables can be more difficult to implement than self-balancing binary search trees. Choosing an effective hash function for a specific application is more an art than a science. In open-addressed hash tables it is fairly easy to create a poor hash function.

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree. Thus hash tables are not effective when the number of entries is very small. (However, in some cases the high cost of computing the hash function can be mitigated by saving the hash value together with the key.)

For certain string processing applications, such as spell-checking, hash tables may be less efficient than tries, finite automata, or Judy arrays. Also, if each key is represented by a small enough number of bits, then, instead of a hash table, one may use the key directly as the index into an array of values. Note that there are no collisions in this case.

The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. Therefore, there is no efficient way to efficiently locate an entry whose key is *nearest* to a given key. Listing all n entries in some specific order generally requires a separate sorting step, whose cost is proportional to $\log(n)$ per entry. In comparison, ordered search trees have lookup and insertion cost proportional to $\log(n)$, but allow finding the nearest key at about the same cost, and *ordered* enumeration of all entries at constant cost per entry.

If the keys are not stored (because the hash function is collision-free), there may be no easy way to enumerate the keys that are present in the table at any given moment.

Although the *average* cost per operation is constant and fairly small, the cost of a single operation may be quite high. In particular, if the hash table uses dynamic resizing, an insertion or deletion operation may occasionally take time proportional to the number of entries. This may be a serious drawback in real-time or interactive applications.

Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays. Compact data structures such as arrays, searched with linear search, may be faster if the table is relatively small and keys are integers or other short strings. According to Moore's Law, cache sizes are growing exponentially and so what is considered "small" may be increasing. The optimal performance point varies from system to system.

Hash tables become quite inefficient when there are many collisions. While extremely uneven hash distributions are extremely unlikely to arise by chance, a malicious adversary with knowledge of the hash function may be able to supply information to a hash which creates worst-case behavior by causing excessive collisions, resulting in very poor performance (i.e., a denial of service attack). In critical applications, either universal hashing can be used or a data structure with better worst-case guarantees may be preferable.^[17]

Uses

Associative arrays

Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects), especially in interpreted programming languages like AWK, Perl, and PHP.

Database indexing

Hash tables may also be used for disk-based persistent data structures and database indices (such as dbm) although balanced trees are more popular in these applications.

Caches

Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually the one that is currently stored in the table.

Sets

Besides recovering the entry which has a given key, many hash table implementations can also tell whether such an entry exists or not.

Those structures can therefore be used to implement a set data structure, which merely records whether a given key belongs to a specified set of keys. In this case, the structure can be simplified by eliminating all parts which have to do with the entry values. Hashing can be used to implement both static and dynamic sets.

Object representation

Several dynamic languages, such as Python, JavaScript, and Ruby, use hash tables to implement objects. In this representation, the keys are the names of the members and methods of the object, and the values are pointers to the corresponding member or method.

Unique data representation

Hash tables can be used by some programs to avoid creating multiple character strings with the same contents. For that purpose, all strings in use by the program are stored in a single hash table, which is checked whenever a new string has to be created. This technique was introduced in Lisp interpreters under the name hash consing, and can be used with many other kinds of data (expression trees in a symbolic algebra system, records in a database, files in a file system, binary decision diagrams, etc.)

Implementations

In programming languages

Many programming languages provide hash table functionality, either as built-in associative arrays or as standard library modules. In C++0x, for example, the `hash_map` and `unordered_map` classes provide hash tables for keys and values of arbitrary type.

In PHP 5, the Zend 2 engine uses one of the hash functions from Daniel J. Bernstein to generate the hash values used in managing the mappings of data pointers stored in a HashTable. In the PHP source code, it is labelled as "DJBX33A" (Daniel J. Bernstein, Times 33 with Addition).

Python's built-in hash table implementation, in the form of the dict type, is highly optimized as dictionaries are used internally to implement namespaces.

Independent packages

- Google Sparse Hash^[18] The Google SparseHash project contains several C++ hash-map implementations in use at Google, with different performance characteristics, including an implementation that optimizes for space and one that optimizes for speed. The memory-optimized one is extremely memory-efficient with only 2 bits/entry of overhead.
- SunriseDD^[19] An open source C library for hash table storage of arbitrary data objects with lock-free lookups, built-in reference counting and guaranteed order iteration. The library can participate in external reference counting systems or use its own built-in reference counting. It comes with a variety of hash functions and allows the use of runtime supplied hash functions via callback mechanism. Source code is well documented.
- uthash^[20] This is an easy-to-use hash table for C structures.
- A number of language runtimes and/or standard libraries use hash tables to implement their support for associative arrays.
- Software written to minimize memory usage can conserve memory by keeping all allocated strings in a hash table. If an already existing string is found a pointer to that string is returned; otherwise, a new string is allocated and added to the hash table. (This is the normal technique used in Lisp for the names of variables and functions; see the documentation for the intern and intern-soft functions if you are using that language.) The data compression achieved in this manner is usually around 40%.

History

The idea of hashing arose independently in different places. In January 1953, H. P. Luhn wrote an internal IBM memorandum that used hashing with chaining.^[21] G. N. Amdahl, E. M. Boehme, N. Rochester, and A. L. Samuel implemented a program using hashing at about the same time. Open addressing with linear probing (relatively prime stepping) is credited to Amdahl, but Ershov (in Russia) had the same idea.^[21]

See also

- Rabin-Karp string search algorithm
- Stable hashing
- Consistent hashing
- Extendible hashing
- Lazy deletion
- Pearson hashing

Related data structures

There are several data structures that use hash functions but cannot be considered special cases of hash tables:

- Bloom filter, a structure that implements an enclosing approximation of a set, allowing insertions but not deletions.
- Distributed hash table, a resilient dynamic table spread over several nodes of a network.

Further reading

- "9: Maps and Dictionaries". *Data Structures and Algorithms in Java* (4th ed.). Wiley. pp. 369–418.
ISBN 0-471-73884-0.

External links

- A Hash Function for Hash Table Lookup ^[22] by Bob Jenkins.
- Hash functions ^[23] by Paul Hsieh
- Libhashish ^[24] is one of the most feature rich hash libraries (built-in hash functions, several collision strategies, extensive analysis functionality, ...)
- NIST entry on hash tables ^[25]
- Open addressing hash table removal algorithm from ICI programming language, *ici_set_unassign* in *set.c* ^[26] (and other occurrences, with permission).
- The Perl Wikibook - Perl Hash Variables
- A basic explanation of how the hash table works by Reliable Software ^[27]
- Lecture on Hash Tables ^[28]
- Hash-tables in C ^[29] – two simple and clear examples of hash tables implementation in C with linear probing and chaining
- C Hash Table ^[30]
- Implementation of HashTable in C ^[31]
- MIT's Introduction to Algorithms: Hashing 1 ^[32] MIT OCW lecture Video
- MIT's Introduction to Algorithms: Hashing 2 ^[33] MIT OCW lecture Video
- CollectionSpy ^[3] – A profiler for Java's Collections Framework with explicit support for debugging hashtable-related issues.

References

- [1] Charles E. Leiserson, *Amortized Algorithms, Table Doubling, Potential Method* (http://videolectures.net/mit6046jf05_leiserson_lec13/)
Lecture 13, course MIT 6.046J/18.410J Introduction to Algorithms - Fall 2005
- [2] Donald Knuth (1998). *The Art of Computer Programming' 3: Sorting and Searching* (2nd ed.). Addison-Wesley. pp. 513–558.
ISBN 0-201-89685-0.
- [3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms* (second ed.). MIT Press and McGraw-Hill. pp. 221–252. ISBN 978-0-262-53196-2.
- [4] Bret Mulvey, Hash Functions (<http://bretm.home.comcast.net/~bretm/hash/>). Accessed April 11, 2009
- [5] Thomas Wang (1997), Prime Double Hash Table (<http://www.concentric.net/~Ttwang/tech/primehash.htm>). Accessed April 11, 2009
- [6] Askitis, Nikolas; Zobel, Justin (2005). *Cache-conscious Collision Resolution in String Hash Tables* (<http://www.springerlink.com/content/b61721172558qf03/>). **3772**. 91–102. doi:10.1007/11575832_11. ISBN 1721172558..
- [7] Askitis, Nikolas (2009). *Fast and Compact Hash Tables for Integer Keys* (<http://www.crpit.com/VolumeIndexU.html#Vol91>). **91**. 113–122. ISBN 978-1-920682-72-9. .
- [8] Erik Demaine, Jeff Lind. 6.897: Advanced Data Structures. MIT Computer Science and Artificial Intelligence Laboratory. Spring 2003. http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L2/lecture2.pdf
- [9] Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe J. (1990). *Data Structures Using C*. Prentice Hall. pp. 456–461, pp. 472.
ISBN 0-13-199746-7.
- [10] Celis, Pedro (1986). Robin Hood hashing. Technical Report Computer Science Department, University of Waterloo CS-86-14.

- [11] Viola, Alfredo (October 2005). "Exact distribution of individual displacements in linear probing hashing". *Transactions on Algorithms (TALG)* (ACM) **1** (2,): 214–242. doi:10.1145/1103963.1103965. ISSN:1549-6325.
- [12] Celis, Pedro (March, 1988). External Robin Hood Hashing. Technical Report Computer Science Department, Indiana University TR246.
- [13] Herlihy, Maurice and Shavit, Nir and Tzafrir, Moran (2008). "Hopscotch Hashing". *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*. Arcachon, France: Springer-Verlag. pp. 350–364.
- [14] <http://java.sun.com/javase/6/docs/api/java/util/HashMap.html>
- [15] Litwin, Witold (1980). "Linear hashing: A new tool for file and table addressing". *Proc. 6th Conference on Very Large Databases*. pp. 212–223.
- [16] Doug Dunham. CS 4521 Lecture Notes (<http://www.duluth.umn.edu/~ddunham/cs4521s09/notes/ch11.txt>). University of Minnesota Duluth. Theorems 11.2, 11.6. Last modified 21 April 2009.
- [17] Crosby and Wallach's *Denial of Service via Algorithmic Complexity Attacks* (http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf).
- [18] <http://code.google.com/p/google-sparsehash/>
- [19] <http://www.sunrisetel.net/software/devtools/sunrise-data-dictionary.shtml>
- [20] <http://uthash.sourceforge.net/>
- [21] Mehta, Dinesh P.; Sahni, Sartaj. *Handbook of Datastructures and Applications*. pp. 9–15. ISBN 1584884355.
- [22] <http://www.burtleburtle.net/bob/hash/doobs.html>
- [23] <http://www.azillionmonkeys.com/qed/hash.html>
- [24] <http://libhashish.sourceforge.net/>
- [25] <http://www.nist.gov/dads/HTML/hashtab.html>
- [26] <http://ici.cvs.sourceforge.net/ici/ici/set.c?view=markup>
- [27] <http://www.relisoft.com/book/lang/pointer/8hash.html>
- [28] <http://compgeom.cs.uiuc.edu/~jeffe/teaching/373/notes/06-hashing.pdf>
- [29] <http://task3.cc/308/hash-maps-with-linear-probing-and-separate-chaining/>
- [30] <http://www.cl.cam.ac.uk/~cwc22/hashtable/>
- [31] http://wiki.portugal-a-programar.org/c:snippet:hash_table_c
- [32] <http://video.google.com/videoplay?docid=-727485696209877198&q=source%3A014117792397255896270&hl=en>
- [33] <http://video.google.com/videoplay?docid=2307261494964091254&q=source%3A014117792397255896270&hl=en>

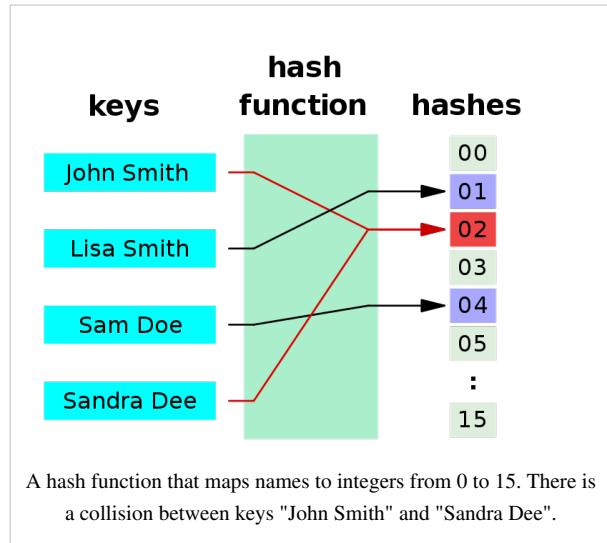
Hash function

A **hash function** is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array. The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, or simply **hashes**.

Hash functions are mostly used to speed up 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 may map two or more keys to the same hash value. In many applications, it is desirable to minimize the occurrence of such collisions, which means that the hash function must map the keys to the hash values as evenly as possible. Depending on the application, other properties may be required as well. 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 optimised differently. The HashKeeper database



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

maintained by the National Drug Intelligence Center, for instance, is more aptly described as a catalog of file fingerprints than of hash values.

Applications

Hash tables

Hash functions are primarily used in hash tables, to quickly locate a data record (for example, a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to the hash. The index gives the place where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets.

In general, a hashing function may map several different keys to the same index. Therefore, each slot of a hash table is associated with (implicitly or explicitly) a set of records, rather than a single record. For this reason, each slot of a hash table is often called a *bucket*, and hash values are also called *bucket indices*.

Thus, the hash function only hints at the record's location—it tells where one should start looking for it. Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries

Caches

Hash functions are also used to build caches for large data sets stored in slow media. A cache is generally simpler than a hashed search table, since any collision can be resolved by discarding or writing back the older of the two colliding items.

Bloom filters

Hash functions are an essential ingredient of the Bloom filter, a compact data structure that provides an enclosing approximation to a set of keys.

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 of 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 (as in the MusicBrainz song labeling service). For this application, the hash function must be as insensitive as possible to data capture or transmission errors, and to "trivial" changes such as timing and volume changes, compression, etc.^[2]

Finding similar substrings

The same techniques can be used to find equal or similar stretches in a large collection of strings, such as a document repository or a genomic database. In this case, the input strings are broken into many small pieces, and a hash function is used to detect potentially equal pieces, as above.

The Rabin-Karp algorithm is a relatively fast string searching algorithm that works in $O(n)$ time on average. It is based on the use of hashing to compare strings.

Geometric hashing

This principle is widely used in computer graphics, computational geometry and many other disciplines, to solve many proximity problems in the plane or in three-dimensional space, such as finding closest pairs in a set of points, similar shapes in a list of shapes, similar images in an image database, and so on. In these applications, the set of all inputs is some sort of metric space, and the hashing function can be interpreted as a partition of that space into a grid of *cells*. The table is often an array with two or more indices (called a *grid file*, *grid index*, *bucket grid*, and similar names), and the hash function returns an index tuple. This special case of hashing is known as geometric hashing or *the grid method*. Geometric hashing is also used in telecommunications (usually under the name vector quantization) to encode and compress multi-dimensional signals.

Properties

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. Note that different requirements apply to the other related concepts (cryptographic hash functions, checksums, etc.).

Low cost

The cost of computing a hash function must be small enough to make a hashing-based solution advantageous with regard to alternative approaches. For instance, binary search can locate an item in a sorted table of n items with $\log_2 n$ key comparisons. Therefore, a hash table solution will be more efficient than binary search only if computing the hash function for one key costs less than performing $\log_2 n$ key comparisons. However, this example does not take sorting the data set into account. Even very fast sorting algorithms such as merge sort take an average of $n \log n$ time to sort a set of data, and so the efficiency of a binary search solution is reduced as the frequency with which items are added to the data set increases. One advantage of hash tables is that they do not require sorting, which keeps the cost of the hash function constant regardless of the rate at which items are added to the data set.

Determinism

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the hashed data, in the mathematical sense of the term. This requirement excludes hash functions that depend on external variable parameters, such as pseudo-random number generators that depend on the time of day. It also excludes functions that depend on the memory address of the object being hashed, if that address may change during processing (as may happen in systems that use certain methods of garbage collection), although sometimes rehashing of the item can be done.

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-square test.^[3]

Variable range

In many applications, the range of hash values may be different for each run of the program, or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data z , and the number n of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to $2^{32}-1$), divide the result by n , and use the division's remainder. If n is itself a power of 2, this can be done by bit masking and bit shifting. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and $n-1$, for any n that may occur in the application. Depending on the function, the remainder may be uniform only for certain n , e.g. odd or prime numbers.

It is possible to relax the restriction of the table size being a power of 2 and not having to perform any modulo, remainder or division operation -as these operation are considered computational costly in some contexts. When n is much lesser than 2^b take a pseudo random number generator (PRNG) function $P(key)$, uniform on the interval $[0, 2^b-1]$. Consider the ratio $q = 2^b / n$. Now the hash function can be seen as the value of $P(key) / q$. Rearranging the calculation and replacing the 2^b -division by bit shifting right ($>>$) b times you end up with hash function $n * P(key) \gg b$.

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.

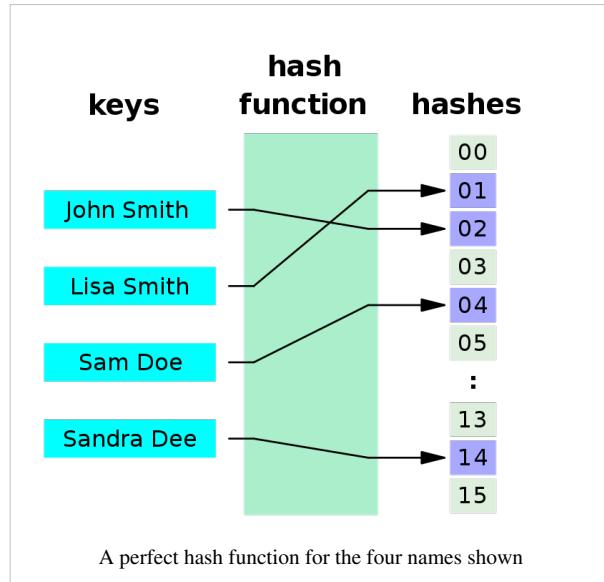
The meaning of "small enough" depends on how much memory is available for the hash table. A typical PC (as of 2008) might have a gigabyte of available memory, meaning that hash values of up to 30 bits could be accommodated. However, there are many applications that can get by with much less. For example, 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.

Unfortunately, perfect hash functions are effective only in situations where the inputs are fixed and entirely known in advance, such as mapping month names to the integers 0 to 11, or words to the entries of a dictionary. A perfect function for a given set of n keys, suitable for use in a hash table, can be found in time proportional to n , can be represented in less than $3*n$ bits, and can be evaluated in a constant number of operations. There are generators that produce optimized executable code to evaluate a perfect hash for a given input set.

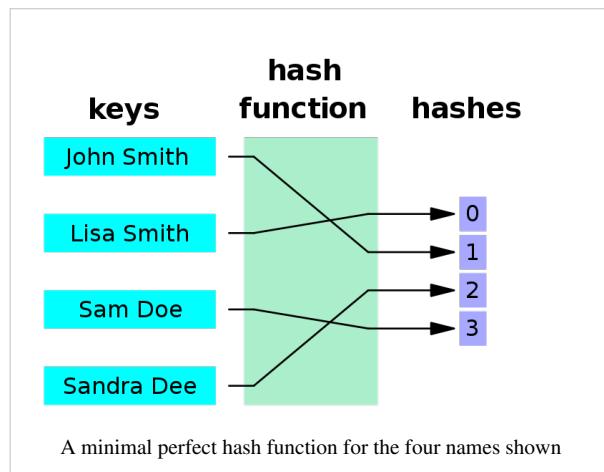


Minimal perfect hashing

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.



Hashing data with other distributions

These simple formulas will not do if the input values are not equally likely, or are not independent. For instance, most patrons of a supermarket will live in the same geographic area, so their telephone numbers are likely to begin with the same 3 to 4 digits. In that case, if n is 10000 or so, the division formula $(z \times n) \div N$, which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula $z \bmod n$, which is quite sensitive to the trailing digits, may still yield a fairly even distribution of hash values.

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 $b[1], b[2], \dots, b[m]$ 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, $S0$ 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 $\text{highbit}(S) = 0$ then return $2 * S + b$ else **return** $(2 * S + b) \wedge P$ Here $\text{highbit}(S)$ denotes the most significant bit of S ; the '*' operator denotes unsigned integer multiplication with lost overflow; ' \wedge ' 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.

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 much larger than 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—individually 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.

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)
- MurmurHash (32 or 64 bits)
- Pearson hashing (8 bits)
- Zobrist hashing

See also

- Bloom filter
- Coalesced hashing
- Cuckoo hashing
- Cryptographic hash function
- Distributed hash table
- Geometric hashing
- Hash table
- HMAC
- Linear hash
- List of hash functions
- Locality sensitive hashing
- Perfect hash function
- Rabin-Karp string search algorithm
- Rolling hash
- Transposition table
- Universal hashing

External links

- General purpose hash function algorithms (C/C++/Pascal/Java/Python/Ruby) ^[8]
- Hash Functions and Block Ciphers by Bob Jenkins ^[9]
- Integer Hash Function ^[10] by Thomas Wang
- The Goulburn Hashing Function ^[11] (PDF) by Mayur Patel
- Hash Functions ^[23] by Paul Hsieh
- The Murmur Hash Function ^[12] by Austin Appleby
- Hash generator ^[13] Generates hashes online for 30+ different algorithms
- Online Char (ASCII), HEX, Binary, Base64, etc... Encoder/Decoder with MD2, MD4, MD5, SHA1+2, etc. hashing algorithms ^[14]
- FNV ^[15] Fowler, Noll, Vo Hash Function
- Hash Generator ^[16] Online Hash Generator (CRCs, MD2, MD4, MD5, SHA1, Tiger, Snefru, RipeMD, Whirlpool, Haval, Oracle, Cisco PIX, NTLM, Fletcher, RAdmin, etc) About 123 algorithms and 200 modes
- Ajax-based Hash Generator ^[17] Online Hash Generator with instant hash computation while typing
- qDecoder's C/C++ hash functions ^[18] – opensource library
- hashr ^[19] A tool to create hashes in more than 40 hashing algorithms. Available through website or Firefox extension
- MIT's Introduction to Algorithms: Hashing 1 ^[32] MIT OCW lecture Video
- MIT's Introduction to Algorithms: Hashing 2 ^[33] MIT OCW lecture Video
- The Art of Hashing ^[20] The Art of Hashing
- Easy Hash ^[21] Simple hash generator & cracker which supports over 130 algorithms

References

- [1] Knuth, Donald (1973). *The Art of Computer Programming, volume 3, Sorting and Searching*. pp. 506–542.
- [2] "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:zSzzSzwww.extra.research.philips.comzSznatlabzSzdownloadzSzaudiofpzSzcbmi01audiohashv1.0.pdf>/haitsma01robust.pdf
- [3] Bret Mulvey, Hash Functions (<http://bretm.home.comcast.net/~bretm/hash/>). Accessed April 11, 2009
- [4] A. Z. Broder. Some applications of Rabin's fingerprinting method. In Sequences II: Methods in Communications, Security, and Computer Science, pp. 143–152. Springer-Verlag, 1993
- [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] <http://www.cse.yorku.ca/~oz/hash.html>
- [8] <http://www.partow.net/programming/hashfunctions/index.html>
- [9] <http://burtleburtle.net/bob/hash/index.html>
- [10] <http://www.concentric.net/~Ttwang/tech/inthash.htm>
- [11] <http://www.oocities.com/drone115b/Goulburn06.pdf>
- [12] <http://murmurhash.googlepages.com/>
- [13] <http://www.utilitiesman.com/computers/security/encrypt>
- [14] <http://www.paulschou.com/tools/xlate/>
- [15] <http://isthe.com/chongo/tech/comp/fnv/>
- [16] <http://www.sinfocol.org/herramientas/hashes.php>
- [17] <http://hash.stephan-brumme.com/>
- [18] <http://www.qdecoder.org/goto/qHash.html>
- [19] <http://rogeriopvl.com/hashr>
- [20] http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx
- [21] http://ziin.pl/en/easy_hash

Open addressing

Open addressing, or **closed hashing**, is a method of collision resolution in hash tables. With this method a hash collision is resolved by **probing**, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. ^[1] Well known probe sequences include:

Linear probing

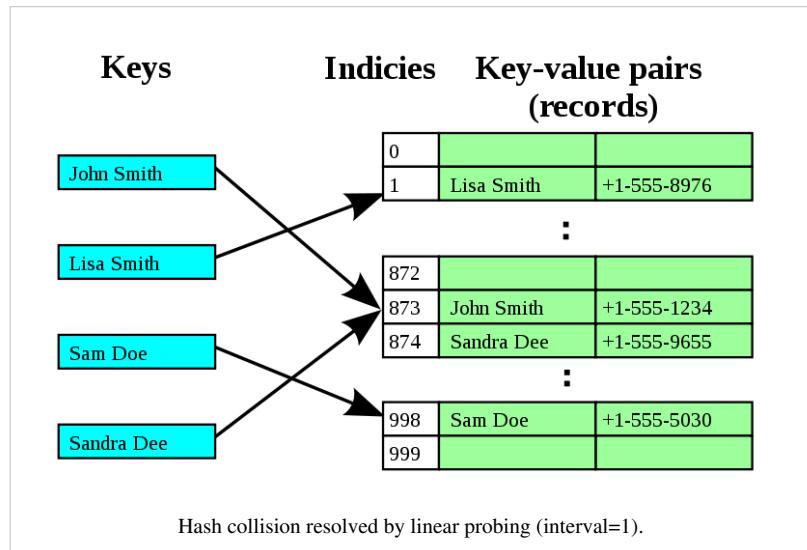
in which the interval between probes is fixed — often at 1.

Quadratic probing

in which the interval between probes increases linearly (hence, the indices are described by a quadratic function).

Double hashing

in which the interval between probes is fixed for each record but is computed by another hash function.



The main tradeoffs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poor cache performance but exhibits virtually no clustering; quadratic probing falls in-between in both areas. Double hashing can also require more computation than other forms of probing. Some open addressing methods, such as last-come-first-served hashing and cuckoo hashing move existing keys around in the array to make room for the new key. This gives better maximum search times than the methods based on probing.

A critical influence on performance of an open addressing hash table is the *load factor*; that is, the proportion of the slots in the array that are used. As the load factor increases towards 100%, the number of probes that may be required to find or insert a given key rises dramatically. Once the table becomes full, probing algorithms may even fail to terminate. Even with good hash functions, load factors are normally limited to 80%. A poor hash function can exhibit poor performance even at very low load factors by generating significant clustering. What causes hash functions to cluster is not well understood, and it is easy to unintentionally write a hash function which causes severe clustering.

Example pseudo code

The following pseudocode is an implementation of an open addressing hash table with linear probing and single-slot stepping, a common approach that is effective if the hash function is good. Each of the **lookup**, **set** and **remove** functions use a common internal function **find_slot** to locate the array slot that either does or should contain a given key.

```

record pair { key, value }
var pair array slot[0..num_slots-1]

function find_slot(key)
    i := hash(key) modulo num_slots
    // search until we either find the key, or find an empty slot.
    while ( (slot[i] is occupied) and ( slot[i].key ≠ key ) ) do
        i := (i + 1) modulo num_slots
    repeat
    return i

function lookup(key)
    i := find_slot(key)
    if slot[i] is occupied // key is in table
        return slot[i].value
    else                      // key is not in table
        return not found

function set(key, value)
    i := find_slot(key)
    if slot[i] is occupied
        slot[i].value := value
    else
        if the table is almost full
            rebuild the table larger (note 1)
            i := find_slot(key)
        slot[i].key    := key
        slot[i].value := value

```

Another example showing open addressing technique. Presented function is converting each part(4) of an internet protocol address, where NOT is bitwise NOT, XOR is bitwise XOR, OR is bitwise OR, AND is bitwise AND and << and >> are shift-left and shift-right:

```
// key_1,key_2,key_3,key_4 are following 3-digit numbers - parts of ip
address xxx.xxx.xxx.xxx

function ip(key parts)
    j := 1
    do
        key := (key_2 << 2)
        key := (key + (key_3 << 7))
        key := key + (j OR key_4 >> 2) * (key_4) * (j + key_1)
XOR j
        key := key AND _prime_           // _prime_ is a prime number
        j := (j+1)
    while collision
    return key
```

note 1

Rebuilding the table requires allocating a larger array and recursively using the **set** operation to insert all the elements of the old array into the new larger array. It is common to increase the array size exponentially, for example by doubling the old array size.

```
function remove(key)
    i := find_slot(key)
    if slot[i] is unoccupied
        return // key is not in the table
    j := i
    loop
        mark slot[i] as unoccupied
        r2:
        j := (j+1) modulo num_slots
        if slot[j] is unoccupied
            exit loop
        k := hash(slot[j].key) modulo num_slots
        // `k lies cyclically in ]i,j]'
        // | i.k.j |
        // |....j i.k.| or | .k..j i...|
        if ( (i<=j) ? ((i<k)&&(k<=j)) :
((i<k) || (k<=j)) )
            goto r2;
        slot[i] := slot[j]
        i := j
```

note 2

For all records in a cluster, there must be no vacant slots between their natural hash position and their current position (else lookups will terminate before finding the record). At this point in the pseudocode, *i* is a vacant slot that might be invalidating this property for subsequent records in the cluster. *j* is such a subsequent record. *k* is the raw hash where the record at *j* would naturally land in the hash table if there were no collisions. This

test is asking if the record at j is invalidly positioned with respect to the required properties of a cluster now that i is vacant.

Another technique for removal is simply to mark the slot as deleted. However this eventually requires rebuilding the table simply to remove deleted records. The methods above provide $O(1)$ updating and removal of existing records, with occasional rebuilding if the high water mark of the table size grows.

The $O(1)$ remove method above is only possible in linearly probed hash tables with single-slot stepping. In the case where many records are to be deleted in one operation, marking the slots for deletion and later rebuilding may be more efficient.

See also

- Lazy deletion - a method of deleting from a hash table using open addressing.

References

- [1] Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe J. (1990), *Data Structures Using C*, Prentice Hall, pp. pp. 456–461, pp. 472, ISBN 0-13-199746-7

Lazy deletion

In computer science, **lazy deletion** refers to a method of deleting elements from a hash table that uses open addressing. In this method, deletions are done by marking an element as deleted, rather than erasing it entirely. Deleted locations are treated as empty when inserting and as occupied during a search.

The problem with this scheme is that as the number of delete/insert operations increases the cost of a successful search increases. To improve this when an element is searched and found in the table the element is relocated to the first location marked for deletion that was probed during the search. Instead of finding an element to relocate when the deletion occurs, the relocation occurs lazily during the next search.^[1]

References

- [1] Celis, Pedro; Franco, John (1995), *The Analysis of Hashing with Lazy Deletions* (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.39.9637>), Computer Science Department, Indiana University, Technical Report CS-86-14,

bbb

Linear probing

Linear probing is a scheme in computer programming for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location.^[1] This is accomplished using two values - one as a starting value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the *stepsize*, is repeatedly added to the starting value until a free space is found, or the entire table is traversed.

```
newLocation = (startingValue + stepSize) % arraySize
```

This algorithm, which is used in open-addressed hash tables, provides good memory caching (if stepsize is equal to one), through good locality of reference, but also results in clustering, an unfortunately high probability that where there has been one collision there will be more. The performance of linear probing is also more sensitive to input distribution when compared to double hashing.

Given an ordinary hash function $H(x)$, a linear probing function would be:

$$H(x, i) = (H(x) + i) \pmod{n}.$$

Here $H(x)$ is the starting value, n the size of the hash table, and the *stepsize* is i in this case.

See also

- Double hashing
- Hash collision
- Hash function
- Quadratic probing
- Collision resolution

External links

- How Caching Affects Hashing^[2] by Gregory L. Heileman and Wenbin Luo 2005.

References

- [1] Dale, Nell (2003). *C++ Plus Data Structures*. Sudbury, MA: Jones and Bartlett Computer Science. ISBN 0-7637-0481-4.
- [2] <http://www.siam.org/meetings/alenex05/papers/13gheileman.pdf>

Quadratic probing

Quadratic probing is a scheme in computer programming for resolving collisions in hash tables.

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This algorithm is used in open-addressed hash tables. Quadratic probing provides good memory caching because it preserves some locality of reference; however, linear probing has greater locality and, thus, better cache performance. Quadratic probing better avoids the clustering problem that can occur with linear probing, although it is not immune.

Quadratic probing is used in the Berkeley Fast File System to allocate free blocks. The allocation routine chooses a new cylinder-group when the current is nearly full using quadratic probing, because of the speed it shows in finding unused cylinder-groups.

Quadratic Probing Algorithm

Let $h(k)$ be a hash function that maps an element k to an integer in $[0, m - 1]$, where m is the size of the table.

Let the i^{th} probe position for a value k be given by the function $h(k, i) = (h(k) + c_1i + c_2i^2) \pmod{m}$, where $c_2 \neq 0$. If $c_2 = 0$, then $h(k, i)$ degrades to a linear probe. For a given hash table, the values of c_1 and c_2 remain constant.

Example: If $h(k, i) = (h(k) + i + i^2) \pmod{m}$, then the probe sequence will be $h(k), h(k) + 2, h(k) + 6, \dots$

For $m = 2^n$, a good choice for the constants are $c_1 = c_2 = 1/2$, as the values $h(k, i)$ for i in $[0, m - 1]$ are all distinct.^[1] This leads to a probe sequence of $h(k), h(k) + 1, h(k) + 3, h(k) + 6, \dots$ where the values increase by 1, 2, 3,

For prime $m > 2$, most choices of c_1 and c_2 will make $h(k, i)$ distinct for i in $[0, (m - 1)/2]$. Such choices include $c_1 = c_2 = 1/2$, $c_1 = c_2 = 1$, and $c_1 = 0, c_2 = 1$. Because there are only about $m/2$ distinct probes for a given element, it is difficult to guarantee that insertions will succeed when the load factor is $< 1/2$.

See also

- Hash tables
- Hash collision
- Double hashing
- Linear probing
- Hash function
- Collision resolution

References

Thomas H. Cormen

Vladimir Batagelj: The quadratic hash method when the table size is not a prime number. Communications of the ACM, Volume 18 , Issue 4 (April 1975), Pages: 216 - 217 <http://portal.acm.org/citation.cfm?id=360737>

References

- [1] Proof: assume there exist i, j such that i, j in $[0, m - 1]$ and $(i + i^2)/2 = (j + j^2)/2 \pmod{m}$. Then
 $i + i^2 = j + j^2 \pmod{2m}$, $i^2 - j^2 + i - j = 0 \pmod{2m}$,
 $(i - j)(i + j) + (i - j) = 0 \pmod{2m}$, and $(i - j)(i + j + 1) = 0 \pmod{2m}$. Since $2m$ is a power of 2, and only one of the two factors can be even, we must have $i - j = 0 \pmod{2m}$ or
 $i + j + 1 = 0 \pmod{2m}$. The latter is not possible with i, j in $[0, m - 1]$, and the former implies that $i = j$.

Double hashing

Double hashing is a computer programming technique used in hash tables to resolve hash collisions, cases when two different values to be searched for produce the same hash key. It is a popular collision-resolution technique in open-addressed hash tables.

Like linear probing, it uses one hash value as a starting point and then repeatedly steps forward an interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is decided using a second, independent hash function (hence the name double hashing). Unlike linear probing and quadratic probing, the interval depends on the data, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering. In other words, given independent hash functions h_1 and h_2 , the j th location in the bucket sequence for value k in a hash table of size m is:

$$h(k, j) = (h_1(k) + j \cdot h_2(k)) \pmod{m}$$

Disadvantages

Linear probing and, to a lesser extent, quadratic probing are able to take advantage of the data cache by accessing locations that are close together. Double hashing has larger intervals and is not able to achieve this advantage.

Like all other forms of open addressing, double hashing becomes linear as the hash table approaches maximum capacity. The only solution to this is to rehash to a larger size.

On top of that, it is possible for the secondary hash function to evaluate to zero. For example, if we choose $k=5$ with the following function:

$$h_2(k) = 5 - (k \pmod{7})$$

The resulting sequence will always remain at the initial hash value. One possible solution is to change the secondary hash function to:

$$h_2(k) = (k \pmod{7}) + 1$$

This ensures that the secondary hash function will always be non zero.

See also

- Hash collision
- Hash function
- Linear probing
- Cuckoo hashing

External links

- How Caching Affects Hashing^[2] by Gregory L. Heileman and Wenbin Luo 2005.

Cuckoo hashing

Cuckoo hashing is a scheme in computer programming for resolving hash collisions of values of hash functions in a table. Cuckoo hashing was first described by Rasmus Pagh and Flemming Friche Rodler in 2001.^[1] The name derives from the behavior of some species of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches.

Theory

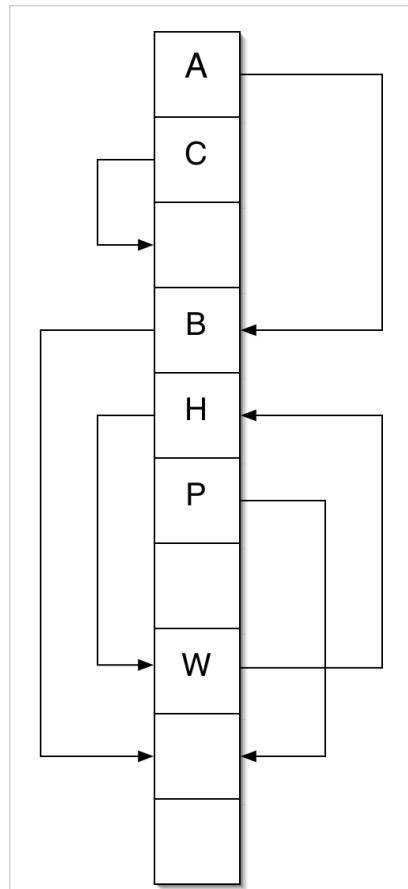
The basic idea is to use two hash functions instead of only one. This provides two possible locations in the hash table for each key.

When a new key is inserted, a greedy algorithm is used: The new key is inserted in one of its two possible locations, "kicking out", that is, displacing, any key that might already reside in this location. This displaced key is then inserted in its alternative location, again kicking out any key that might reside there, until a vacant position is found, or the procedure enters an infinite loop. In the latter case, the hash table is rebuilt in-place^[2] using new hash functions.

Lookup requires inspection of just two locations in the hash table, which takes constant time in the worst case (*see* Big O notation). This is in contrast to many other hash table algorithms, which may not have a constant worst-case bound on the time to do a lookup.

It can also be shown that insertions succeed in expected constant time,^[1] even considering the possibility of having to rebuild the table, as long as the number of keys is kept below half of the capacity of the hash table, i.e., the load factor is below 50%.

Generalizations of cuckoo hashing that use more than 2 alternative hash functions can be expected to utilize a larger part of the capacity of the hash table efficiently while sacrificing some lookup and insertion speed. Using just three hash functions increases the load to 91%. Another generalization of cuckoo hashing consists in using more than one key per bucket. Using just 2 keys per bucket permits a load factor above 80%.



Cuckoo hashing example. The arrows show the alternative location of each key.

A new item would be inserted in the location of A by moving A to its alternative location, currently occupied by B, and moving B to its alternative location which is currently vacant. Insertion of a new item in the location of H would not succeed: Since H is part of a cycle (together with W), the new item would get kicked out again.

Other algorithms that use multiple hash functions include the Bloom filter. Cuckoo hashing can be used to implement a data structure equivalent to a Bloom filter.

A study by Zukowski et al.^[3] has shown that cuckoo hashing is much faster than chained hashing for small, cache-resident hash tables on modern processors. Kenneth Ross^[4] has shown bucketized versions of cuckoo hashing (variants that use buckets that contain more than one key) to be faster than conventional methods also for large hash tables, when space utilization is high. However as of 2007 cuckoo hashing remains largely unknown outside the research community.

A simplified generalization of cuckoo hashing called skewed-associative cache is used in some CPU caches.

See also

- Perfect hashing
- Linear probing
- Double hashing
- Hash collision
- Hash function
- Quadratic probing

References

- [1] Pagh, Rasmus; Rodler, Flemming Friche (2001) (PDF, PS). *Cuckoo Hashing* (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.4189>). doi:10.1.1.25.4189. . Retrieved 2008-10-16.
- [2] Pagh and Rodler. "There is no need to allocate new tables for the rehashing: We may simply run through the tables to delete and perform the usual insertion procedure on all keys found not to be at their intended position in the table."
- [3] Zukowski, Marcin; Heman, Sandor; Boncz, Peter (2006-06) (pdf). *Architecture-Conscious Hashing* (<http://www.cs.cmu.edu/~damon2006/pdf/zukowski06archconscioushashing.pdf>). Proceedings of the International Workshop on Data Management on New Hardware (DaMoN). . Retrieved 2008-10-16.
- [4] Ross, Kenneth (2006-11-08) (pdf). *Efficient Hash Probes on Modern Processors* ([http://domino.research.ibm.com/library/cyberdig.nsf/papers/DF54E3545C82E8A585257222006FD9A2/\\$File/rc24100.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/DF54E3545C82E8A585257222006FD9A2/$File/rc24100.pdf)). IBM Research Report RC24100. RC24100. . Retrieved 2008-10-16.
- A cool and practical alternative to traditional hash tables (<http://www.ru.is/faculty/ulfar/CuckooHash.pdf>), U. Erlingsson, M. Manasse, F. Mcsherry, 2006.
- Cuckoo Hashing for Undergraduates, 2006 (<http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf>), R. Pagh, 2006.
- Cuckoo Hashing, Theory and Practice (<http://mybiasedcoin.blogspot.com/2007/06/cuckoo-hashing-theory-and-practice-part.html>) (Part 1, Part 2 (http://mybiasedcoin.blogspot.com/2007/06/cuckoo-hashing-theory-and-practice-part_15.html) and Part 3 (http://mybiasedcoin.blogspot.com/2007/06/cuckoo-hashing-theory-and-practice-part_19.html)), Michael Mitzenmacher, 2007.
- [INaor, Moni (<http://www.wisdom.weizmann.ac.il/~naor/>); Segev, Gil; Wieder, Udi (2008). "History-Independent Cuckoo Hashing" (http://www.wisdom.weizmann.ac.il/~naor/PAPERS/cuckoo_hi_abs.html). *International Colloquium on Automata, Languages and Programming (ICALP)*. Reykjavik, Iceland. Retrieved 2008-07-21.

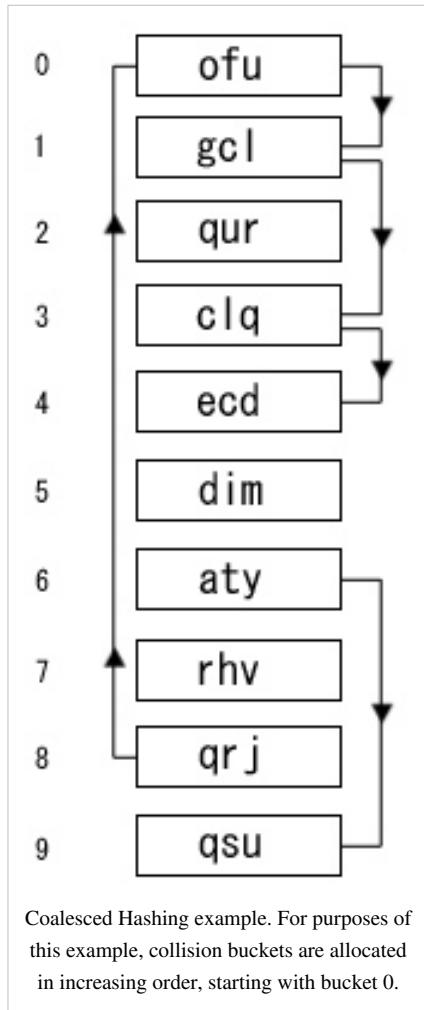
External links

- Static cuckoo hashtable generator for C/C++ (<http://www.theiling.de/projects/lookupstable.html>)
- Cuckoo hashtable written in Java (<http://lmonson.com/blog/?p=100>)

Coalesced hashing

Coalesced hashing, also called **coalesced chaining**, is a strategy of collision resolution in a hash table that forms a hybrid of separate chaining and open addressing. In a separate chaining hash table, items that hash to the same address are placed on a list (or "chain") at that address. This technique can result in a great deal of wasted memory because the table itself must be large enough to maintain a load factor that performs well (typically twice the expected number of items), and extra memory must be used for all but the first item in a chain (unless list headers are used, in which case extra memory must be used for all items in a chain).

Given a sequence "qrj," "aty," "qur," "dim," "ofu," "gcl," "rvh," "clq," "ecd," "qsu" of randomly generated three character long strings, the following table would be generated (using Bob Jenkins' One-at-a-Time hash algorithm^[1]) with a table of size 10:



(null)			
"clq"			
"qur"			
(null)			
(null)			
"dim"			
"aty"	"qsu"		
"rvh"			
"qrj"	"ofu"	"gcl"	"ecd"
(null)			
(null)			

This strategy is effective, efficient, and very easy to implement. However, sometimes the extra memory use might be prohibitive, and the most common alternative, open addressing, has uncomfortable disadvantages that decrease performance. The primary disadvantage of open addressing is primary and secondary clustering, in which searches may access long sequences of used buckets that contain items with different hash addresses; items with one hash

address can thus lengthen searches for items with other hash addresses.

One solution to these issues is coalesced hashing. Coalesced hashing uses a similar technique as separate chaining, but instead of allocating new nodes for the linked list, buckets in the actual table are used. The first empty bucket in the table at the time of a collision is considered the collision bucket. When a collision occurs anywhere in the table, the item is placed in the collision bucket and a link is made between the chain and the collision bucket. It is possible for a newly inserted item to collide with items with a different hash address, such as the case in the example above when item "clq" is inserted. The chain for "clq" is said to "coalesce" with the chain of "qrj," hence the name of the algorithm. However, the extent of coalescing is minor compared with the clustering exhibited by open addressing. For example, when coalescing occurs, the length of the chain grows by only 1, whereas in open addressing, search sequences of arbitrary length may combine.

An important optimization, to reduce the effect of coalescing, is to restrict the address space of the hash function to only a subset of the table. For example, if the table has size M with buckets numbered from 0 to $M - 1$, we can restrict the address space so that the hash function only assigns addresses to the first N locations in the table. The remaining $M - N$ buckets, called the *cellar*, are used exclusively for storing items that collide during insertion. No coalescing can occur until the cellar is exhausted.

The optimal choice of N relative to M depends upon the load factor (or fullness) of the table. A careful analysis shows that the value $N = 0.86 \times M$ yields near-optimum performance for most load factors.^[2] Other variants for insertion are also possible that have improved search time. Deletion algorithms have been developed that preserve randomness, and thus the average search time analysis still holds after deletions.^[2]

Insertion in C:

```
/*
htab is the hash table,
N is the size of the address space of the hash function, and
M is the size of the entire table including the cellar.
Collision buckets are allocated in decreasing order, starting with
bucket M-1. */

int insert ( char key[] )
{
    unsigned h = hash ( key, strlen ( key ) ) % N;

    if ( htab[h] == NULL ) {
        /* Make a new chain */
        htab[h] = make_node ( key, NULL );
    } else {
        struct node *it;
        int cursor = M-1;

        /* Find the first empty bucket */
        while ( cursor >= 0 && htab[cursor] != NULL )
            --cursor;

        /* The table is full, terminate unsuccessfully */
        if ( cursor == -1 )
            return -1;

        htab[cursor] = make_node ( key, NULL );
    }
}
```

```

/* Find the last node in the chain and point to it */
it = htab[h];

while ( it->next != NULL )
    it = it->next;

    it->next = htab[cursor];
}

return 0;
}

```

One benefit of this strategy is that the search algorithm for separate chaining can be used without change in a coalesced hash table.

Lookup in C:

```

char *find ( char key[] )
{
    unsigned h = hash ( key, strlen ( key ) ) % N;

    if ( htab[h] != NULL ) {
        struct node *it;

        /* Search the chain at index h */
        for ( it = htab[h]; it != NULL; it = it->next ) {
            if ( strcmp ( key, it->data ) == 0 )
                return it->data;
        }
    }

    return NULL;
}

```

Performance

Coalesced chaining avoids the effects of primary and secondary clustering, and as a result can take advantage of the efficient search algorithm for separate chaining. If the chains are short, this strategy is very efficient and can be highly condensed, memory-wise. As in open addressing, deletion from a coalesced hash table is awkward and potentially expensive, and resizing the table is terribly expensive and should be done rarely, if ever.

References

- [1] <http://burtleburtle.net/bob/>
- [2] J. S. Vitter and W.-C. Chen, *Design and Analysis of Coalesced Hashing*, Oxford University Press, New York, NY, 1987, ISBN 0-19-504182-8

Perfect hash function

A **perfect hash function** for a set S is a hash function that maps distinct elements in S to distinct integers, with no collisions. A perfect hash function with values in a limited range can be used for efficient lookup operations, by placing keys from S (or other associated values) in a table indexed by the output of the function.

A perfect hash function for a specific set S that can be evaluated in constant time, and with values in a small range, can be found by a randomized algorithm in a number of operations that is proportional to the size of S . The minimal size of the description of a perfect hash function depends on the range of its function values: The smaller the range, the more space is required. Any perfect hash functions suitable for use with a hash table require at least a number of bits that is proportional to the size of S .

Using a perfect hash function is best in situations where there is a frequently queried large set, S , which is seldom updated. Efficient solutions to performing updates are known as dynamic perfect hashing, but these methods are relatively complicated to implement. A simple alternative to perfect hashing, which also allows dynamic updates, is cuckoo hashing.

A **minimal perfect hash function** is a perfect hash function that maps n keys to n consecutive integers—usually $[0..n-1]$ or $[1..n]$. A more formal way of expressing this is: Let j and k be elements of some set K . F is a minimal perfect hash function iff $F(j) = F(k)$ implies $j=k$ and there exists an integer a such that the range of F is $a..a+|K|-1$. It has been proved that any minimal perfect hash scheme requires at least 1.44 bits/key. However the smallest currently use around 2.5 bits/key. Some of these space-efficient perfect hash functions have been implemented in cmph library [1] and Sux4J [2].

A minimal perfect hash function F is **order preserving** if keys are given in some order a_1, a_2, \dots , and for any keys a_j and a_k , $j < k$ implies $F(a_j) < F(a_k)$. Order-preserving minimal perfect hash functions require necessarily $\Omega(n \log n)$ bits to be represented.

A minimal perfect hash function F is **monotone** if it preserves the lexicographical order of the keys. Monotone minimal perfect hash functions can be represented in very little space. Several implementations of monotone minimal perfect hash functions are available in Sux4J [2].

See also

- Pearson hashing

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 11.5: Perfect hashing, pp. 245–249.
- Fabiano C. Botelho, Rasmus Pagh and Nivio Ziviani. "Perfect Hashing for Data Management Applications" [3].
- Fabiano C. Botelho and Nivio Ziviani. "External perfect hashing for very large key sets" [4]. 16th ACM Conference on Information and Knowledge Management (CIKM07), Lisbon, Portugal, November 2007.
- Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. "Monotone minimal perfect hashing: Searching a sorted table with O(1) accesses" [5]. In Proceedings of the 20th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA), New York, 2009. ACM Press.
- Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. "Theory and practise of monotone minimal perfect hashing" [6]. In Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, 2009.

External links

- Minimal Perfect Hashing [7] by Bob Jenkins
- gperf [8] is a Free software C and C++ perfect hash generator
- cmph [9] is another Free Software implementing many perfect hashing methods
- Sux4J [2] is another Free Software implementing many perfect hashing methods in Java

References

- [1] <http://cmph.sourceforge.net/>
- [2] <http://sux4j.dsi.unimi.it/>
- [3] <http://arxiv.org/pdf/cs/0702159>
- [4] <http://homepages.dcc.ufmg.br/~nivio/papers/cikm07.pdf>
- [5] <http://vigna.dsi.unimi.it/ftp/papers/MonotoneMinimalPerfectHashing.pdf>
- [6] <http://vigna.dsi.unimi.it/ftp/papers/TheoryAndPractiseMonotoneMinimalPerfectHashing.pdf>
- [7] <http://burtleburtle.net/bob/hash/perfect.html>
- [8] <http://www.gnu.org/software/gperf/>
- [9] <http://cmph.sourceforge.net/index.html>

Universal hashing

Universal hashing is a randomized algorithm for selecting a hash function F with the following property: for any two distinct inputs x and y , the probability that $F(x)=F(y)$ (i.e., that there is a hash collision between x and y) is the same as if F was a random function. Thus, if F has function values in a range of size r , the probability of any particular hash collision should be **at most** $1/r$. There are universal hashing methods that give a function F that can be evaluated in a handful of computer instructions.

Introduction

Hashing was originally used to implement hash tables, taking an input such as a string and returning an index into the table for an object corresponding to the input. Since then, hashing has found other uses. For example, two inputs might be compared by checking to see if their hash values are the same. In general, a hash function is a function from a set of possible inputs, U , to a set of outputs, which is usually taken to be $\{1, \dots, N\}$ for some N .

However, hash functions are many-to-one mappings. That is, they are not injective. If we are very unfortunate, we might have a set of inputs which all map to the same value. Universal hashing is all about making it an extremely unlikely event that the possible hash values are not equally used.

Universal hashing

When trying to prove that a hashing algorithm works well, one needs to show that it successfully distributes the input values it is given and they aren't all mapped to the same value. But, for any given hash function h , we know that there are some input values for which it doesn't work well, so we cannot expect to prove anything of that sort for an explicitly given h .

The solution to this problem is to select h from some class of functions at random, ensuring that any given input is unlikely to be a bad input for our chosen hash. The obvious class of such functions is the set of functions from the set of possible input keys, K , to the set of outputs, $\{1, \dots, N\}$.

In this case, it is a theorem that the expected time required to perform any m operations (insert, delete, lookup) on a hash-table based on h is of order $O(m)^{[1]}$.

Unfortunately, this set is very large so picking a random function from it is computationally very expensive. Moreover, the function would also (almost certainly) be very complicated to describe, negating any speed benefits that the hash function might have given.

Fortunately, to prove the theorem above, the only property of the set of functions used was the fact that, for inputs $x \neq y$, the probability that $h(x) = h(y)$ is at most $1/N$, a property that we can guarantee in practice.

Definition

Let K be a set of keys, V be a set of values and H be a set of functions from K to V . Then H is called a **2-universal family** of hash functions if, for all x, y in K with $x \neq y$,

$$\Pr_{h \in H} [h(x) = h(y)] \leq \frac{1}{|V|}.$$

Note that this is the property that was described above^[2].

A more stringent requirement is that, for all distinct x, y in K and x', y' in V ,

$$\Pr_{h \in H} [h(x) = x' \text{ and } h(y) = y'] = \frac{1}{|V|^2}.$$

If this holds then H is called a **strongly 2-universal family**. Without qualification, the latter definition is probably intended. This definition can be generalised to a **k-universal family**, picking k distinct elements of K and of V and requiring that the probability is $|V|^{-k}$.

Examples

Suppose $K = \{0, \dots, N_K - 1\}$ and $V = \{0, \dots, N_V - 1\}$ where $N_K > N_V$. Then choose a prime $p > N_K$. Define

$$h(x) = g(x) \bmod N_V \text{ and}$$

$$g(x) = ((ax + b) \bmod p)$$

with $a, b \in \mathbb{Z}/p\mathbb{Z}$ and $a \neq 0$. Then H is a 2-universal family of hash functions from K to V .

To see why, first observe that if $h(x) = h(y)$, then $g(x) \equiv g(y) \pmod{N_V}$. This can happen in at most $p(p-1)/N_V$ ways by first choosing one of the p choices for $g(x)$ and then choosing one of the $\lfloor p/N_V \rfloor = \lceil p/N_V \rceil - 1 \leq (p-1)/N_V$ choices for $g(y)$ that are equivalent to our choice for $g(x)$ modulo N_V (where the equality follows because p is prime and $p > N_V$). If $g(x) = g(y)$, then it has nothing to do with a and b because

$$g(x) = g(y)$$

$$\iff ax + b \equiv ay + b \pmod{p}$$

$$\iff ax + b - b \equiv ay + b - b \pmod{p}$$

$$\iff ax \equiv ay \pmod{p}$$

$$\iff a^{-1}ax \equiv a^{-1}ay \pmod{p}$$

$$\iff x \equiv y \pmod{p}.$$

Since $\mathbb{Z}/p\mathbb{Z}$ is a field, b has an additive inverse and $a \neq 0$ has a multiplicative inverse, so the above steps are valid.

Because there are $p-1$ choices for a and p choices for b , the probability that two hashing functions in this family will hash to the same value when $x \neq y$ is at most $\frac{p(p-1)}{p(p-1)} = \frac{1}{n}$, which means that this family of hash functions is 2-universal.

Uses

Universal hashing has numerous uses in computer science, for example in cryptography and in implementations of hash tables. Since the function is randomly chosen, an adversary hoping to create many hash collisions is unlikely to succeed.

See also

- Universal one-way hash function

References

[1] Carter and Wegman, p146 or Miltersen, p3

[2] Carter and Wegman used the notation *universal*₂ for this definition

- Knuth, Donald Ervin (1998). *[The Art of Computer Programming], Vol. III: Sorting and Searching* (2e ed.). Reading, Mass ; London: Addison-Wesley. ISBN 0-201-89685-0.
- Carter and Wegman (1977). "Universal Classes of Hash Functions". *Journal of Computer and System Sciences* **18** (2): 143–154. doi:10.1016/0022-0000(79)90044-8.
- Miltersen, Peter Bro. "Universal Hashing" (<http://www.webcitation.org/5hmOaVISI>) (PDF). Archived from the original (<http://www.daimi.au.dk/~bromille/Notes/un.pdf>) on 24th June 2009.

Linear hashing

Linear hashing is a dynamic hash table algorithm invented by Witold Litwin (1980),^[1] and later popularized by Paul Larson. Linear hashing allows for the expansion of the hash table one slot at a time. The frequent single slot expansion can very effectively control the length of the collision chain. The cost of hash table expansion is spread out across each hash table insertion operation, as opposed to be incurred all at once.^[2] Therefore linear hashing is well suited for interactive applications.

Algorithm Details

A hash function controls the address calculation of linear hashing. In linear hashing, the address calculation is always bounded by a size that is a power of two * N, where N is the chosen original number of buckets. The number of buckets is given by $N * 2^{\text{Level}}$ e.g. Level 0, N; Level 1, 2N; Level 2 , 4N.

```
address(level, key) = hash(key) mod N * (2level)
```

The 'split' variable controls the read operation, and the expansion operation.

A read operation would use $\text{address}(\text{level}, \text{key})$ if $\text{address}(\text{level}, \text{key})$ is greater than or equal to the 'split' variable. Otherwise, $\text{address}(\text{level}+1, \text{key})$ is used. This takes into account buckets numbered less than split have been rehashed with $\text{address}(\text{level}+1, \text{key})$ after its contents split between two new buckets (the first bucket writing over the contents of the old single bucket prior to the split).

A linear hashing table expansion operation would consist of rehashing the entries at one slot location indicated by the 'split' variable to either of two target slot locations of $\text{address}(\text{level}+1, \text{key})$. This intuitively is consistent with the assertion that if $y = x \bmod M$ and $y' = x \bmod M * 2$, then $y' = y$ or $y' = y + M$.

The 'split' variable is incremented by 1 at the end of the expansion operation. If the 'split' variable reaches $N * 2^{\text{level}}$, then the 'level' variable is incremented by 1, and the 'split' variable is reset to 0.

Thus the hash buckets are expanded round robin, and seem unrelated to where buckets overflow at the time of expansion. Overflow buckets are used at the sites of bucket overflow (the normal bucket has a pointer to the

overflow bucket), but these are eventually reabsorbed when the round robin comes to the bucket with the overflow bucket, and the contents of that bucket and the overflow bucket are redistributed by the future hash function ($\text{hash(key)} \bmod N * (2^{\text{level}+1})$).

The degenerate case, which is unlikely with a randomized hash function, is that enough entries are hashed to the same bucket so that there is enough entries to overflow more than one overflow bucket (assuming overflow bucket size = normal bucket size), before being absorbed when that bucket's turn to split comes in the round robin.

The point of the algorithm seems to be that overflow is preempted by gradually increasing the number of available buckets, and overflow buckets are eventually reabsorbed during a later split, which must eventually happen because splitting occurs round robin.

There is some flexibility in choosing how often the expansion operations are performed. One obvious choice is to perform the expansion operation each time no more slots are available at the target slot location. Another choice is to control the expansion with a programmer defined load factor.

The hash table array for linear hashing is usually implemented with a dynamic array algorithm.

Adoption in language systems

Griswold and Townsend^[3] discussed the adoption of linear hashing in the Icon language. They discussed the implementation alternatives of dynamic array algorithm used in linear hashing, and presented performance comparisons using a list of Icon benchmark applications.

External links

- Sorted Linear Hash Table^[4]
-  This article incorporates public domain material from the NIST document "linear hashing"^[5] by Paul E. Black (Dictionary of Algorithms and Data Structures).

See also

- Extendible hashing
- Consistent hashing

References

- [1] Litwin, Witold (1980), "Linear hashing: A new tool for file and table addressing" (<http://www.cs.cmu.edu/afs/cs.cmu.edu/user/christos/www/courses/826-resources/PAPERS+BOOK/linear-hashing.PDF>) (PDF), *Proc. 6th Conference on Very Large Databases*: 212–223,
- [2] Larson, Per-Åke (April 1988), "Dynamic Hash Tables" (<http://doi.acm.org/10.1145/42404.42410>), *Communications of the ACM* **31**: 446–457, doi:10.1145/42404.42410,
- [3] Griswold, William G.; Townsend, Gregg M. (April 1993), "The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon" (<http://citeseer.ist.psu.edu/griswold93design.html>), *Software - Practice and Experience* **23** (4): 351–367,
- [4] <http://www.concentric.net/~Ttwang/tech/sorthash.htm>
- [5] <http://www.nist.gov/dads/HTML/linearHashing.html>

Extendible hashing

Extendible hashing is a type of hash system which treats a hash as a bit string, and uses a trie for bucket lookup.^[1] Because of the hierarchical nature of the system, re-hashing is an incremental operation (done one bucket at a time, as needed). This means that time-sensitive applications are less affected by table growth than by standard full-table rehashes.

Example

This is a more simplistic example from Fagin et al. (1979).

Assume that the hash function $h(k)$ returns a binary number. The first i bits of each string will be used as indices to figure out where they will go in the "directory" (hash table). Additionally, i is the smallest number such that the first i bits of all keys are different.

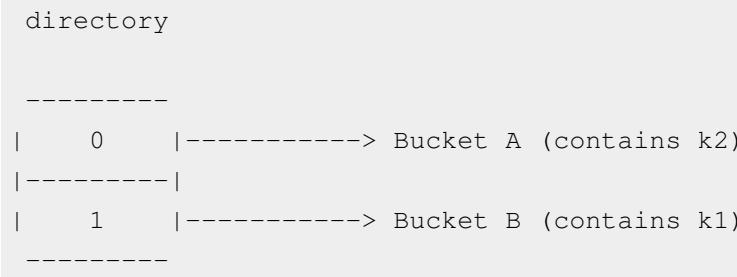
Keys to be used:

$$h(k_1) = 100100$$

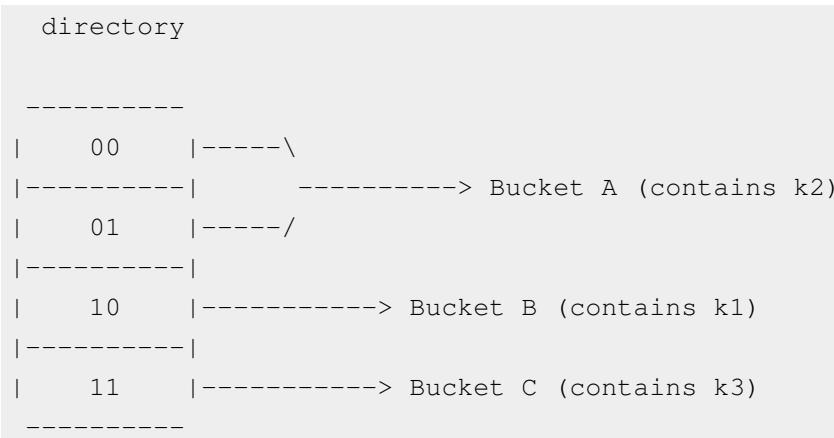
$$h(k_2) = 010110$$

$$h(k_3) = 110110$$

Let's assume that for this particular example, the bucket size is 1. The first two keys to be inserted, k_1 and k_2 , can be distinguished by the most significant bit, and would be inserted into the table as follows:



Now, if k_3 were to be hashed to the table, it wouldn't be enough to distinguish all three keys by one bit (because k_3 and k_1 have 1 as their leftmost bit. Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:



And so now k_1 and k_3 have a unique location, being distinguished by the first two leftmost bits. Because k_2 is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.

Same example, with further detail

(can be skipped if more general idea only needed)

$$h(k_4) = 011110$$

Now, k_4 needs to be inserted, and it has the first two bits as 01..(1110), and using a 2 bit depth in the directory, this maps from 01 to Bucket A. Bucket A is full (max size 1), so it must be split; because there is more than one pointer to Bucket A, there is no need to increase the directory size.

What is needed is information about :

- 1) the key size that maps the directory (the global depth), and
- 2) the key size that has previously mapped the bucket (the local depth).

In order to distinguish the two action cases:

- 1) Doubling the directory when a bucket becomes full
- 2) Creating a new bucket , and re-distributing the entries between the old and the new bucket.

Examining the initial case of an extendible hash structure, if each directory entry points to one bucket, then the local depth should be equal to the global depth.

The number of directory entries is equal to $2^{\text{global depth}}$, and the initial number of buckets is equal to $2^{\text{local depth}}$.

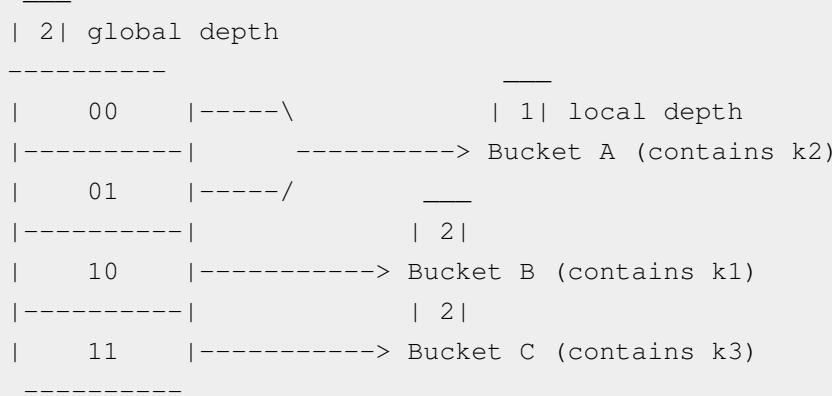
Thus if global depth = local depth = 0, then $2^0 = 1$, so an initial directory of one pointer to one bucket.

Back to the two action cases :

If the local depth is equal to the global depth , then there is only one pointer to the bucket, and there is no other directory pointers that can map to the bucket, so the directory must be doubled (case 1).

if the bucket is full, if the local depth is less than the global depth, then there exists more than one pointer from the directory to the bucket, and the bucket can be split (case 2).

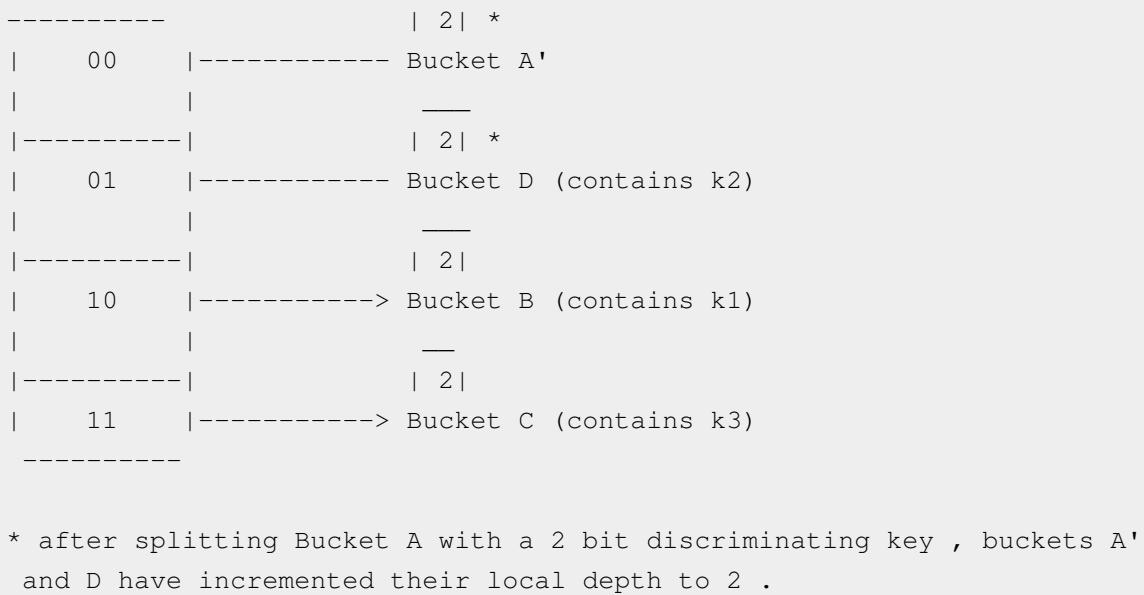
directory



Key 01 points to Bucket A , and Bucket A's local depth of 1 is less than the directory's global depth of 2, which means keys hashed to Bucket A have only used a 1 bit prefix (i.e. 0), and the bucket needs to have its contents split using keys $1 + 1 = 2$ bits in length ; in general, for any local depth d where d is less than D , the global depth, then d must be incremented after a bucket split, and the new d used as the number of bits of each entry's key to redistribute the entries of the former bucket into the new buckets.

directory





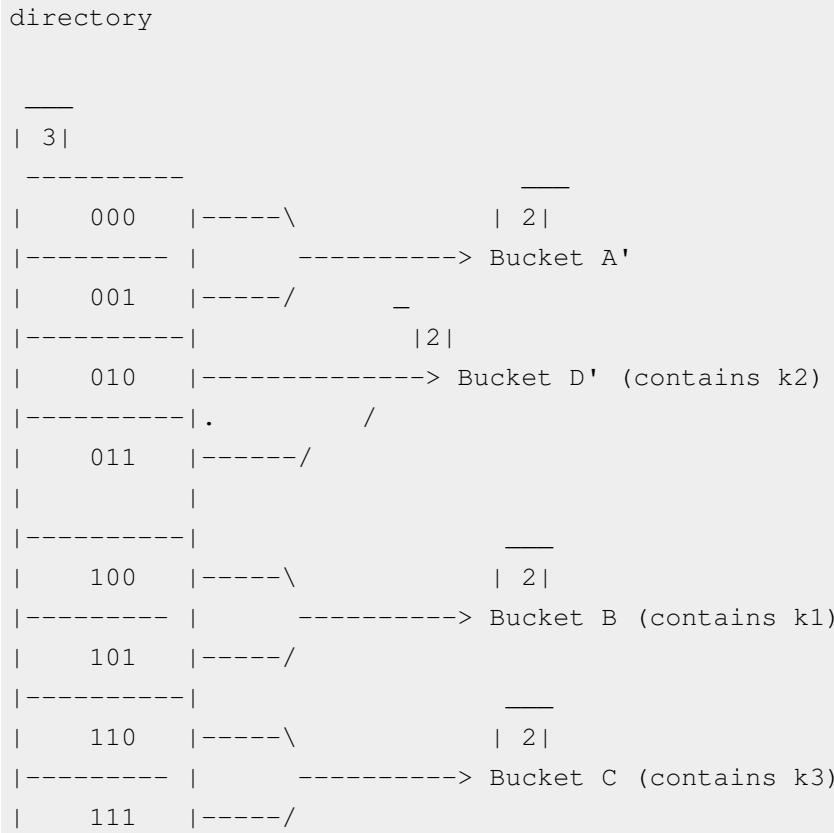
Now, $h(k_4)=011110$

is tried again , with 2 bits 01.. , and now key 01 points to a new bucket but there is still k2 in it ($h(k2)=010110$ and also begins with 01).

If k2 had been 000110 , with key 00, there would have been no problem, because k2 would have remained in the new bucket A' and bucket D would have been empty.

(This would have been the most likely case by far when buckets are of greater size than 1 and the newly split buckets would be exceedingly unlikely to overflow, unless all the entries were all rehashed to one bucket again. But just to emphasise the role of the depth information , the example will be pursued logically to the end.)

So Bucket D needs to be split, but a check of its local depth, which is 2, is the same as the global depth , which is 2, so the directory must be split again , in order to hold keys of sufficient detail, e.g. 3 bits.

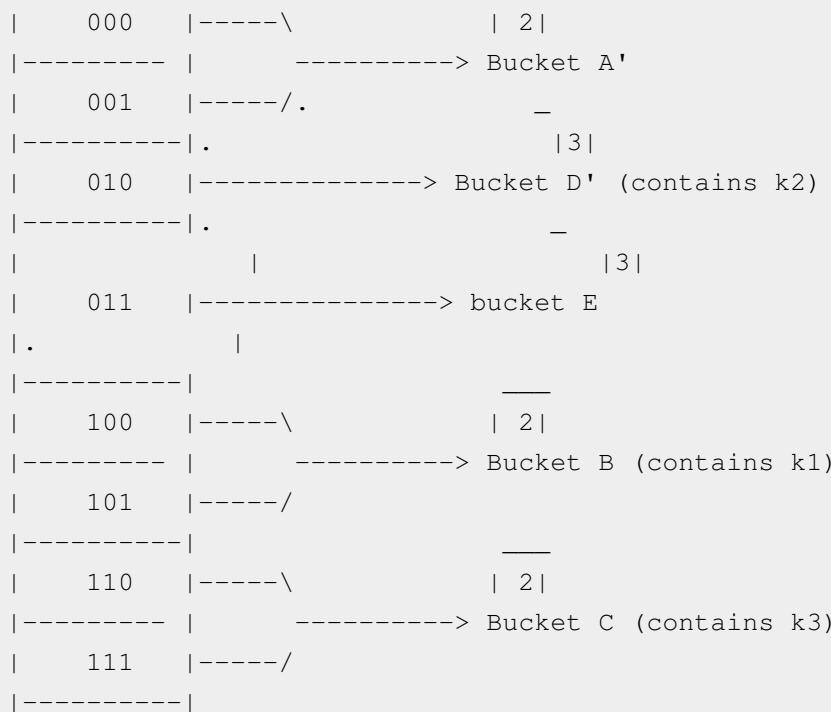


|-----|

1. Bucket D needs to split due to being full.
2. As D's local depth = the global depth , the directory must double to increase bit detail of keys.
3. global depth has incremented after directory split to 3.
4. The new entry k4 is rekeyed with global depth 3 bits and ends up in D which has local depth 2,
which can now be incremented to 3 and D can be split to D' and E.
5. the contents of the split bucket D, k2, has been re-keyed with 3 bits, and it ends up in D.
6. K4 is retried and it ends up in E which has a spare slot.

directory

-----| 3 |-----



Now, $h(k_2) = 010110$ is in D and $h(k_4) = 011110$ is tried again , with 3 bits 011.. , and it points to bucket D which already contains k2 so is full; D's local depth is 2 but now the global depth is 3 after the directory doubling , so now D can be split into bucket's D' and E , the contents of D , k2 has it's $h(k_2)$ retried with a new global depth bitmask of 3 and k2 ends up in D', then the new entry k4 is retried with $h(k_4)$ bitmasked using the new global depth bit count of 3 and this gives 011 which which now points to a new bucket E which is empty ! (wheh!) so K4 goes in Bucket E !.

A Python, Java and C++ in-memory implementation of extendible hashing

Sometimes it is easier to visualize the algorithm with working code. Below is the same version of the extendible hashing algorithm in Python, Java, and C++, with the disc block / memory Page association, caching and consistency issues removed. This part is the whole point of doing extendible hashing, except where an in-memory hashing technique is needed, where the cost of rehashing the contents of a overfilled hash table is undesirable. Note a problem exists if the depth exceeds the bit size of an integer, because then doubling of the directory or splitting of a bucket won't allow entries to be rehashed to different buckets.

Python example

```
PAGE_SZ = 20

class Page:

    def __init__(self):
        self.m = {}
        self.d = 0

    def full(self):
        return len(self.m) > PAGE_SZ

    def put(self, k, v):
        self.m[k] = v

    def get(self, k):
        return self.m.get(k)

class EH:

    def __init__(self):
        self.gd = 0
        p = Page()
        self.pp= [p]

    def get_page(self,k):
        h = hash(k)
        p = self.pp[ h & (( 1 << self.gd) -1) ]
        return p

    def put(self, k, v):
        p = self.get_page(k)
        if p.full() and p.d == self.gd:
            self.pp *= 2
            self.gd += 1

        if p.full() and p.d < self.gd:
            p.put(k,v);
```

```
p1 = Page()
p2 = Page()
for k2,v2 in p.m.items():
    h = hash(k2)
    h = h & ((1 << self.gd) -1)
    if (h >> p.d) & 1 == 1:
        p2.put(k2,v2)
    else:
        p1.put(k2,v2)
for i,x in enumerate(self.pp):
    if x == p:
        if (i >> p.d) & 1 == 1:
            self.pp[i] = p2
        else:
            self.pp[i] = p1

    p1.d = p.d + 1
    p2.d = p1.d
else:
    p.put(k, v)

def get(self, k):
    p = self.get_page(k)
    return p.get(k)

if __name__ == "__main__":
    eh = EH()
    N = 10000
    l = list(range(N))

    import random
    random.shuffle(l)
    for x in l:
        eh.put(x,x)
    print l

    for i in range(N):
        print eh.get(i)
```

Java example

here's the same code in Java, translated with side to side editors.

```
package ext_hashing;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Collections;

public class EH2<K, V> {
    static class Page<K, V> {
        static int PAGE_SZ = 400;
        private Map<K, V> m = new HashMap<K, V>();
        int d = 0;

        boolean full() {
            return m.size() > PAGE_SZ;
        }

        void put(K k, V v) {
            m.put(k, v);
        }

        V get(K k) {
            return m.get(k);
        }
    }

    int gd = 0;

    List<Page<K, V>> pp = new ArrayList<Page<K, V>>();

    public EH2() {
        pp.add(new Page<K, V>());
    }

    Page<K, V> getPage(K k) {
        int h = k.hashCode();
        Page<K, V> p = pp.get(h & ((1 << gd) - 1));
        return p;
    }

    void put(K k, V v) {
        Page<K, V> p = getPage(k);
        if (p.full() && p.d == gd) {
```

```
        List<Page<K, V>> pp2 = new ArrayList<EH2.Page<K, V>>(pp);
        pp.addAll(pp2);
        ++gd;
    }

    if (p.full() && p.d < gd) {
        p.put(k, v);
        Page<K, V> p1, p2;
        p1 = new Page<K, V>();
        p2 = new Page<K, V>();
        for (Map.Entry<K, V> entry : p.m.entrySet()) {
            K k2 = entry.getKey();
            V v2 = entry.getValue();

            int h = k2.hashCode() & ((1 << gd) - 1);

            if ((h >> p.d) & 1 == 1)
                p2.put(k2, v2);
            else
                p1.put(k2, v2);
        }

        for (int i = 0; i < pp.size(); ++i)
            if (pp.get(i) == p) {
                if ((i >> p.d) & 1 == 1)
                    pp.set(i, p2);
                else
                    pp.set(i, p1);
            }

        p1.d = p.d + 1;
        p2.d = p1.d;

    } else
        p.put(k, v);
}

public V get(K k) {
    return getPage(k).get(k);
}

public static void main(String[] args) {

    int N = 5000000;

    List<Integer> l = new ArrayList<Integer>(N);
    for (int i = 0; i < N; ++i) {
```

```

        l.add(i);
    }

    Collections.shuffle(l);

    EH2<Integer, Integer> eh = new EH2<Integer, Integer>();
    for (Integer x : l) {
        eh.put(x, x);
    }

    for (int i = 0; i < N; ++i) {
        System.out.printf("%d:%d , ", i, eh.get(i));
        if (i % 10 == 0)
            System.out.println();
    }

}
}

```

C++ example

The C++ version avoids managing pointers by using an array index as the page pointers in the directory. These examples contrasts the language features of the 3 languages, with respect to a version of the extendible hashing algorithm, which isn't mathematically complex.

The C++ version actually worked 35% better on a mingw intel dual core 2MB setup for 60 million integers, than the simple unordered_map , so it seems small unordered_maps work better despite the overhead of a directory.

```

using namespace std;
#include <map>
#include <unordered_map>

/* to use unordered_map, compile with something like g++ -std=c++0x ..
 */

#include <vector>
#include <cstdlib>
#include <cstring>
#include <iostream>
#ifndef EH_H
#define EH_H
static const int PAGE_SZ=400;

template <class K, class V> class Page {

```

- unordered_map<K,V> m;
- int d ;

```

public:
    Page() : d(0) {

```

```
    }

    bool full() {
        return m.size() > PAGE_SZ;
    }

    void put( K k, V v) {
        m[k] = v;
    }

    V& get( K k) {
        return m[k];
    }

    int depth() { return d; }
    void set_depth(int _d) { d= _d; }

    void get_list( vector<pair<K,V>>& v) {
        for (typename std::unordered_map<K,V>::iterator i =
m.begin(); i != m.end(); ++i) {
            v.push_back(*i);
        }
    }

    void clear() { m.clear(); }

    void hash( const K& k , int &h) {
        memcpy( (void*)&h, (void*)&k, 4);
    }
};

template <class K,class V> class EH {
    int gd;
    vector<int> pp ;
    vector<Page<K,V>> pageFile;
    public:
        EH() : gd(0) {
            pageFile.push_back( Page<K,V>());
            pp.push_back(0);
        }

        ~EH() {
    }
```

```
}

Page<K,V>& get_page( K k, int& ix) {
    int h= 0;
    pageFile[0].hash(k,h);
    //cout << "\nh=" << h << "\n ";
    return pageFile[ix= pp[ h & ( ( 1 << gd) - 1) ]];
}

void put(K k, V v) {
    int ix =0;
    Page<K,V>& p = get_page(k,ix);
    if (p.full() && p.depth() == gd) {
        int sz = pp.size();
        for(int i = 0; i < sz ; ++i)
            pp.push_back(pp[i]);
        ++gd;
    }

    if (p.full() && p.depth() < gd) {
        p.put(k,v);
        Page<K,V> p2 ;
        vector<pair<K,V>> v;
        p.get_list(v);
        int e = v.size();
        p.clear();
        for(int j = 0; j < e; ++j) {
            K k2 = v[j].first;
            V v2 = v[j].second;
            int h = 0;
            p.hash(k2,h);
            h = h & ( (1<< gd) -1);
            if ( ( h >> p.depth()) & 1 == 1)
                p2.put(k2,v2);
        else
            p.put(k2,v2);

        vector<int> l;
        int sz = pp.size();
        for(int i = 0; i < sz; ++i)
            if (pp[i] == ix)
                l.push_back(i);

        for (int i = 0; i < l.size(); ++i ) {
            if ( ( l[i] >> p.depth()) & 1 == 1)
                pp[i] = pageFile.size();
        else
```

```
                pp[i] = ix;
            }
            pageFile.push_back(p2);
        }
        p2.set_depth(p.depth() + 1);
        p.set_depth(p2.depth());
    }

} else
    p.put(k, v);
}

V& operator[](K k) {
    int ix= 0;
    return get_page(k, ix).get(k);
}
};

#endif

// main.cpp

using namespace std;
#include <iostream>
#include <unordered_map>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include "exhash.h"
template <class T> int run_main(int argc, char** argv) {
    int N = 5000000;
    cout << "argc = " << argc << endl;

    if (argc > 1)
        N = atoi ( argv[1] );

    T* x = new T[N];
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
    time_t t0, t1, t2;
    t0=time(&t0);

// EH worked 35% better on dual core intel 2MB mingw than unordered_map
with 60 million test integers !

    //unordered_map<int, int> testmap;
```

```

EH<T, T > testmap;
for (int i = 0; i < N; ++i )
    x[i] = (T) i;

random_shuffle(x, x + N);

t1 = time(&t1);
for (int i = 0; i < N; ++i )
    testmap[x[i]] = x[i];

for (int i = 0; i < N; ++i ) {
    //cout << i << ":" << testmap[i] << " ";
    //if (i % 10 == 0) cout << endl;

    if ( abs( testmap[i] - i) >= 1 )
        cout << i << " testmap[i] not equal" << testmap[i] << endl;
}

t2 = time(&t2);
cout << "N= " << N << " log10(N)= " << log10(N) << endl;
cout << "random generation " << t1 - t0 << endl;
cout << "hash and print " << (t2-t1) << endl;
delete [] x;
return 0;
}

int main( int argc, char** argv) {
    if (argc > 2 && argv[2][1] == 'd') {
        cout << "running with double" << endl;
        return run_main<double>(argc, argv);
    }
    return run_main<int>(argc, argv);
}

```

Significant Bits

The detailed example above uses the *most significant bits* of the hash code as an index into the hash directory, as presented in Fagin et al. (1979). This has the advantage of making the pictures easier to see. Alternatively, using the *least significant bits* makes it more efficient to expand the table, as the entire directory can be copied as one block (Ramakrishnan & Gehrke (2003)).

The python code above takes this *least significant bits* approach.

References

- Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R. (September, 1979), "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems* **4** (3): 315–344, doi:10.1145/320083.320092
- Ramakrishnan, R.; Gehrke, J. (2003), *Database Management Systems, 3rd Edition: Chapter 11, Hash-Based Indexing*, pp. 373–378

See also

- Trie
- Hash table
- Stable hashing
- Consistent hashing
- Linear hashing

External links

-  This article incorporates public domain material from the NIST document "Extendible hashing" [2] by Paul E. Black (Dictionary of Algorithms and Data Structures).
- Extendible Hashing [3] at University of Nebraska
- Extendible Hashing notes [4] at Arkansas State University
- {<http://www.smckearney.com/adb/notes/lecture.extendible.hashing.pdf>} Extendible hashing notes, explains the need to record bit length (k) in the bucket header at time of bucket creation or splitting, in order to determine whether directory doubling is required (where k = K (the directory's bit length)), or just splitting of the bucket , with entries of bit length (k+1) last bit 0 in one bucket, and bit length (k+1) last bit 1 in the other bucket, and then making both buckets' record their bucket bit length as k+1. k is the local depth, and K is the global depth, terms used in one reference.

References

- [1] Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R. (September, 1979), "Extendible Hashing - A Fast Access Method for Dynamic Files" (<http://doi.acm.org/10.1145/320083.320092>), *ACM Transactions on Database Systems* **4** (3): 315–344, doi:10.1145/320083.320092,
- [2] <http://www.nist.gov/dads/HTML/extendibleHashing.html>
- [3] <http://www.isqa.unomaha.edu/haworth/isqa3300/fs009.htm>
- [4] <http://www.csm.astate.edu/~rossa/datastruc/Extend.html>

2-choice hashing

2-choice hashing, also known as **2-choice chaining**, is a variant of a hash table in which keys are added by hashing with two hash functions. The key is put in the array position with the fewer (colliding) keys. Some collision resolution scheme is needed, unless keys are kept in buckets. The average-case cost of a successful search is $O(2 + (m-1)/n)$, where m is the number of keys and n is the size of the array. The most collisions is $\log_2 \ln n + \theta(m/n)$ with high probability.

See also

- 2-left hashing

 This article incorporates public domain material from the NIST document "2-choice hashing" [1] by Paul E. Black (Dictionary of Algorithms and Data Structures).

References

[1] <http://www.nist.gov/dads/HTML/twoChoiceHashing.html>

Pearson hashing

Pearson hashing^[1] is a hash function designed for fast execution on processors with 8-bit registers. Given an input consisting of any number of bytes, it produces as output a single byte that is strongly dependent^[1] on every byte of the input. Its implementation requires only a few instructions, plus a 256-byte lookup table containing a permutation of the values 0 through 255.

This hash function is a CBC-MAC that uses an 8-bit random block cipher implemented via the permutation table. An 8-bit block cipher has negligible cryptographic security, so the Pearson hash function is not cryptographically strong; but it offers these benefits:

- It is extremely simple.
- It executes quickly on resource-limited processors.
- There is no simple class of inputs for which collisions (identical outputs) are especially likely.
- Given a small, privileged set of inputs (e.g., reserved words for a compiler), the permutation table can be adjusted so that those inputs yield distinct hash values, producing what is called a perfect hash function.

The algorithm was originally described by the following pseudocode, which computes the hash of message C with length n using the permutation table T of size t :

```
h := 0 for i in 1..n loop index := h xor C[i - 1] h := T[index mod t] end loop return h
```

References

[1] "Fast Hashing of Variable-Length Text Strings". Peter K. Pearson, *Communications of the ACM* 33(6), 677 (1990) — ACM full text (requires subscription) (<http://portal.acm.org/citation.cfm?id=78978>)

Fowler–Noll–Vo hash function

Fowler–Noll–Vo is a non-cryptographic hash function created by Glenn Fowler, Landon Curt Noll, and Phong Vo.

Overview

The current versions are FNV-1 and FNV-1a, which supply a means of creating non-zero **FNV offset basis**. FNV currently comes in 32-, 64-, 128-, 256-, 512-, and 1024-bit flavors. For pure FNV implementations, this is determined solely by the availability of **FNV primes** for the desired bit length; however, the FNV webpage discusses methods of adapting one of the above versions to a smaller length that may or may not be a power of two.
[1] [2]

The FNV hash algorithms and sample FNV source code^[3] have been released into the public domain.^[4]

FNV is not a cryptographic hash.

The hash

One of FNV's key advantages is that it is very simple to implement. Start with an initial hash value of **FNV offset basis**. For each byte in the input, multiply *hash* by the **FNV prime**, then XOR it with the byte from the input. The alternate algorithm, FNV-1a, reverses the multiply and XOR steps.

FNV-1 hash

The FNV-1 hash algorithm is as follows:^[5]

```
hash = FNV_offset_basis
for each octet_of_data to be hashed
    hash = hash × FNV_prime
    hash = hash XOR octet_of_data
return hash
```

In the above pseudocode, all variables are unsigned integers. All variables, except for *octet_of_data*, have the same number of bits as the FNV hash. The variable, *octet_of_data*, is an 8 bit unsigned integer.

As an example, consider the 64-bit FNV-1 hash:

- All variables, except for *octet_of_data*, are 64-bit unsigned integers.
- The variable, *octet_of_data*, is an 8 bit unsigned integer.
- The **FNV_offset_basis** is the 64-bit **FNV offset basis** value: 14695981039346656037.
- The **FNV_prime** is the 64-bit **FNV prime** value: 1099511628211.
- The multiply (indicated by the \times symbol) returns the lower 64-bits of the product.
- The XOR is an 8-bit operation that modifies only the lower 8-bits of the hash value.
- The *hash* value returned is an 64-bit unsigned integer.

The values for **FNV prime** and **FNV offset basis** may be found in this table.^[6]

FNV-1a hash

The FNV-1a hash differs from the FNV-1 hash by only the order in which the multiply and XOR is performed: [7]

```
hash = FNV_offset_basis
for each octet_of_data to be hashed
    hash = hash XOR octet_of_data
    hash = hash × FNV_prime
return hash
```

The above pseudocode has the same assumptions that were noted for the FNV-1 pseudocode. The small change in order leads to much better avalanche characteristics. [8]

See also

- Pearson hashing (uses a constant linear permutation instead of a constant prime seed)
- Jenkins hash function
- MurmurHash

External links

- Landon Curt Noll's webpage on FNV [9]

References

- [1] Changing the FNV hash size - xor-folding (<http://www.isthe.com/chongo/tech/comp/fnv/index.html#xor-fold>)
- [2] Changing the FNV hash size - non-powers of 2 (<http://www.isthe.com/chongo/tech/comp/fnv/index.html#other-folding>)
- [3] <http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-source>
- [4] FNV put into the public domain (http://www.isthe.com/chongo/tech/comp/fnv/index.html#public_domain)
- [5] The core of the FNV hash (<http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1>)
- [6] Parameters of the FNV-1 hash (<http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param>)
- [7] FNV-1a alternate algorithm (<http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1a>)
- [8] Avalanche (<http://murmurhash.googlepages.com/avalanche>)
- [9] <http://www.isthe.com/chongo/tech/comp/fnv/index.html>

Bitstate hashing

Bitstate hashing is a hashing method invented in 1968 by Morris.^[1] It is used for state hashing, where each state (e.g. of an automaton) is represented by a number and it is passed to some hash function.

The result of the function is then taken as the index to an array of bits (a bit-field), where one looks for 1 if the state was already seen before or stores 1 by itself if not.

It usually serves as a yes–no technique without a need of storing whole state bit representation.

A shortcoming of this framework is losing precision like in other hashing techniques. Hence some tools use this technique with more than one hash function so that the bit-field gets widened by the number of used functions, each having its own row. And even after all functions return values (the indices) point to fields with contents equal to 1, the state may be uttered as visited with much higher probability.

Use

- It is utilized in SPIN model checker for decision whether a state was already visited by nested-depth-first search searching algorithm or not. They mention savings of 98% of memory in the case of using one hash function (175 MB to 3 MB) and 92% when two hash functions are used (13 MB). The state coverage dropped to 97% in the former case.^[2]

References

- [1] Morris, R. (1968). *Scatter Storage Techniques*
[2] Holzmann, G. J. (2003) Addison Wesley. *Spin Model Checker, The: Primer and Reference Manual*

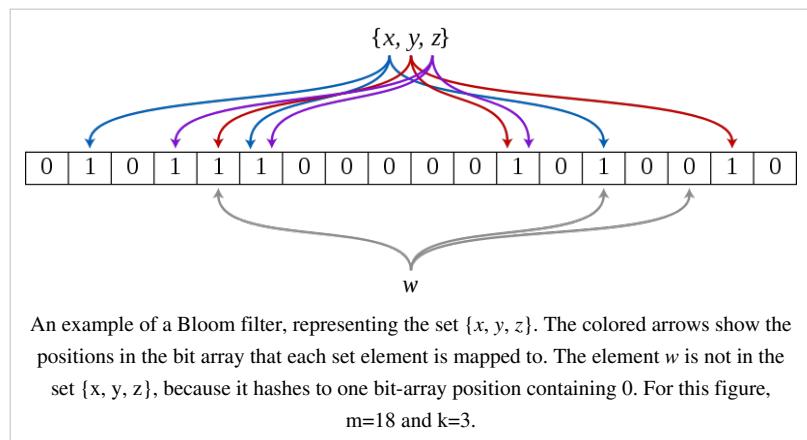
Bloom filter

The **Bloom filter**, conceived by Burton Howard Bloom in 1970,^[1] is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not. Elements can be added to the set, but not removed (though this can be addressed with a counting filter). The more elements that are added to the set, the larger the probability of false positives.

Algorithm description

An **empty Bloom filter** is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution.

To **add** an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.



To **query** for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions are 0, the element is not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements.

The requirement of designing k different independent hash functions can be prohibitive for large k . For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields. Alternatively, one can pass k different initial values (such as 0, 1, ..., $k - 1$) to a hash function that takes an initial value; or add (or append) these values to the key. For larger m and/or k , independence among the hash functions can be relaxed with negligible increase in false positive rate (Dillinger & Manolios (2004a), Kirsch & Mitzenmacher (2006)). Specifically, Dillinger & Manolios (2004b) show the effectiveness of using enhanced double hashing or triple hashing, variants of double hashing, to derive the k indices using simple arithmetic on two or three indices computed with independent hash functions.

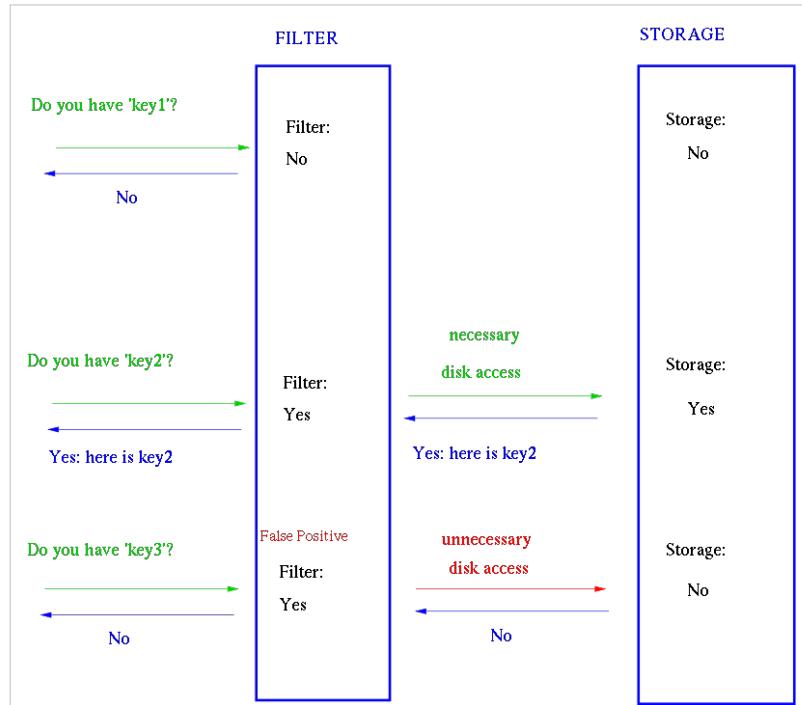
Unfortunately, removing an element from this simple Bloom filter is impossible. The element maps to k bits, and although setting any one of these k bits to zero suffices to remove it, this has the side effect of removing any other elements that map onto that bit, and we have no way of determining whether any such elements have been added. Such removal would introduce a possibility for false negatives, which are not allowed.

Removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which are not permitted. This approach also limits the semantics of removal since re-adding a previously removed item is not possible.

However, it is often the case that all the keys are available but are expensive to enumerate (for example, requiring many disk reads). When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.

Space and time advantages

While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers. A Bloom filter with 1% error and an optimal value of k , on the other hand, requires only about 9.6 bits per element — regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. If a 1% false positive rate seems too high, each time we add about 4.8 bits per element we decrease it by ten times.



Bloom filter used to speed up answers in a key-value storage system. Values are stored on a disk which has slow access times. Bloom filter decisions are much faster. However some unnecessary disk accesses are made when the filter reports a positive (in order to weed out the false positives). Overall answer speed is better with the Bloom filter than without the Bloom filter. Use of a Bloom filter for this purpose, however, does increase memory usage.

However, if the number of potential values is small and many of them can be in the set, then the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element. Note also that hash tables gain a space and time advantage if they begin ignoring collisions and only store whether each bucket contains an entry; in this case, they have effectively become Bloom filters with $k = 1$.

Bloom filters also have the unusual property that the time needed to either add items or to check whether an item is in the set is a fixed constant, $O(k)$, completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its k lookups are independent and can be parallelized.

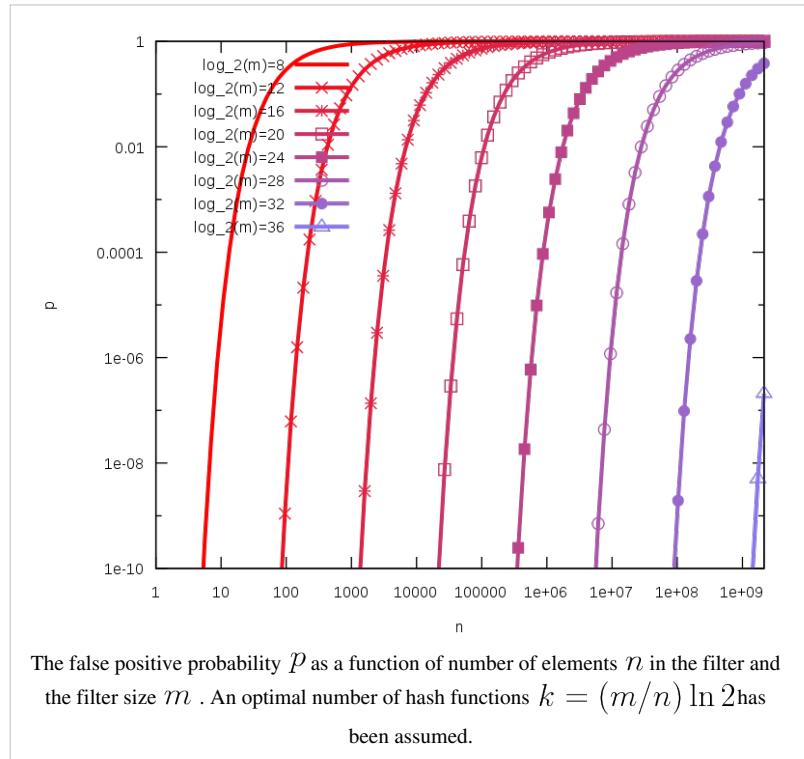
To understand its space efficiency, it is instructive to compare the general Bloom filter with its special case when $k = 1$. If $k = 1$, then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter (k greater than 1) allows many more bits to be set while still maintaining a low false positive rate; if the parameters (k and m) are chosen well, about half of the bits will be set, and these will be apparently random, minimizing redundancy and maximizing information content.

Probability of false positives

Assume that a hash function selects each array position with equal probability. If m is the number of bits in the array, the probability that a certain bit is not set to one by a certain hash function during the insertion of an element is then

$$1 - \frac{1}{m}.$$

The probability that it is not set by any of the hash functions is



$$\left(1 - \frac{1}{m}\right)^k.$$

If we have inserted n elements, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn};$$

the probability that it is 1 is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$

Now test membership of an element that is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is then

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

Obviously, the probability of false positives decreases as m (the number of bits in the array) increases, and increases as n (the number of inserted elements) increases. For a given m and n , the value of k (the number of hash functions) that minimizes the probability is

$$\frac{m}{n} \ln 2 \approx \frac{9m}{13n} \approx 0.7 \frac{m}{n},$$

which gives the false positive probability of

$$2^{-k} \approx 0.6185^{m/n}.$$

Taking the optimal number of hashes, the false positive probability (when ≤ 0.5) can be rewritten and bounded (Starobinski, Trachtenberg & Agarwal (2003))

$$\frac{m}{n} \geq \frac{1}{\ln 2}.$$

This means that in order to maintain a fixed false positive probability, the length of a Bloom filter must grow linearly with the number of elements being filtered.

The required number of bits m given n (the number of inserted elements) and a desired false positive probability p , and assuming the optimal value of k is

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

Interesting properties

- Unlike sets based on hash tables, any Bloom filter can represent the entire universe of elements. In this case, all bits are 1. Another consequence of this property is that **add** never fails due to the data structure "filling up," although the false positive rate increases steadily as elements are added.
- Union and intersection of Bloom filters with the same size and set of hash functions can be implemented with bitwise OR and AND operations, respectively.

Example

Google BigTable uses Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.^[2]

The Squid Web Proxy Cache uses Bloom filters for cache digests^{[3][4]}

The Venti archival storage system uses Bloom filters to detect previously-stored data.^[5]

Alternatives

Classic Bloom filters use $1.44 \log 2(1/\epsilon)$ bits of space per inserted key, where ϵ is the false positive rate of the Bloom filter. However the space that is strictly necessary for any data structure playing the same role as a Bloom filter is only $\log 2(1/\epsilon)$ per key (Pagh, Pagh & Rao 2005). Hence Bloom filters use 44% more space than a hypothetical equivalent optimal data structure. The number of hash functions used to achieve a given false positive rate ϵ is proportional to $1/\epsilon$ which is not optimal as it has been proved that an optimal data structure would need only a constant number of hash functions independent of the false positive rate.

Stern & Dill (1996) describe a probabilistic structure based on hash tables, hash compaction, which Dillinger & Manolios (2004b) identify as significantly more accurate than a Bloom filter when each is configured optimally. Dillinger and Manolios, however, point out that the reasonable accuracy of any given Bloom filter over a wide range of numbers of additions makes it attractive for probabilistic enumeration of state spaces of unknown size. Hash compaction is, therefore, attractive when the number of additions can be predicted accurately; however, despite being very fast in software, hash compaction is poorly-suited for hardware because of worst-case linear access time.

Putze, Sanders & Singler (2007) have studied some variants of Bloom filters that are either faster or use less space than classic Bloom filters. The basic idea of the fast variant is to locate the k hash values associated with each key into one or two blocks having the same size as processor's memory cache blocks (usually 64 bytes). This will presumably improve performance by reducing the number of potential memory cache misses. The proposed variants have however the drawback of using about 32% more space than classic Bloom filters.

The space efficient variant relies on using a single hash function that generates for each key a value in the range $[0, n/\epsilon]$ where ϵ is the requested false positive rate. The sequence of values is then sorted and compressed using Golomb coding (or some other compression technique) to occupy a space close to $n \log 2(1/\epsilon)$ bits. To query the Bloom filter for a given key, it will suffice to check if its corresponding value is stored in the Bloom filter.

Decompressing the whole Bloom filter for each query would make this variant totally unusable. To overcome this problem the sequence of values is divided into small blocks of equal size that are compressed separately. At query time only half a block will need to be decompressed on average. Because of decompression overhead, this variant may be slower than classic Bloom filters but this may be compensated by the fact that a single hash function need to be computed.

Another alternative to classic Bloom filter is the one based on space efficient variants of cuckoo hashing. In this case once the hash table is constructed, the keys stored in the hash table are replaced with short signatures of the keys. Those signatures are strings of bits computed using a hash function applied on the keys.

Extensions and applications

Counting filters

Counting filters provide a way to implement a *delete* operation on a Bloom filter without recreating the filter afresh. In a counting filter the array positions (buckets) are extended from being a single bit, to an n-bit counter. In fact, regular Bloom filters can be considered as counting filters with a bucket size of one bit. Counting filters were introduced by Fan et al. (1998).

The insert operation is extended to *increment* the value of the buckets and the lookup operation checks that each of the required buckets is non-zero. The delete operation, obviously, then consists of decrementing the value of each of the respective buckets.

Arithmetic overflow of the buckets is a problem and the buckets should be sufficiently large to make this case rare. If it does occur then the increment and decrement operations must leave the bucket set to the maximum possible value in order to retain the properties of a Bloom filter.

The size of counters is usually 3 or 4 bits. Hence counting Bloom filters use 3 to 4 times more space than static Bloom filters. In theory, an optimal data structure equivalent to a counting Bloom filter should not use more space than a static Bloom filter.

Another issue with counting filters is limited scalability. Because the counting Bloom filter table cannot be expanded, the maximal number of keys to be stored simultaneously in the filter must be known in advance. Once the designed capacity of the table is exceeded the false positive rate will grow rapidly as more keys are inserted.

Bonomi et al. (2006) introduced a data structure based on d-left hashing that is functionally equivalent but uses approximately half as much space as counting Bloom filters. The scalability issue does not occur in this data structure. Once the designed capacity is exceeded, the keys could be reinserted in a new hash table of double size.

The space efficient variant by Putze, Sanders & Singler (2007) could also be used to implement counting filters by supporting insertions and deletions.

Data synchronization

Bloom filters can be used for approximate data synchronization as in Byers et al. (2004). Counting Bloom filters can be used to approximate the number of differences between two sets and this approach is described in Agarwal & Trachtenberg (2006).

Bloomier filters

Chazelle et al. (2004) designed a generalization of Bloom filters that could associate a value with each element that had been inserted, implementing an associative array. Like Bloom filters, these structures achieve a small space overhead by accepting a small probability of false positives. In the case of "Bloomier filters", a *false positive* is defined as returning a result when the key is not in the map. The map will never return the wrong value for a key that *is* in the map.

The simplest Bloomier filter is near-optimal and fairly simple to describe. Suppose initially that the only possible values are 0 and 1. We create a pair of Bloom filters A_0 and B_0 which contain, respectively, all keys mapping to 0 and all keys mapping to 1. Then, to determine which value a given key maps to, we look it up in both filters. If it is in neither, then the key is not in the map. If the key is in A_0 but not B_0 , then it does not map to 1, and has a high probability of mapping to 0. Conversely, if the key is in B_0 but not A_0 , then it does not map to 0 and has a high probability of mapping to 1.

A problem arises, however, when *both* filters claim to contain the key. We never insert a key into both, so one or both of the filters is lying (producing a false positive), but we don't know which. To determine this, we have another, smaller pair of filters A_1 and B_1 . A_1 contains keys that map to 0 and which are false positives in B_0 ; B_1 contains keys that map to 1 and which are false positives in A_0 . But whenever A_0 and B_0 both produce positives, at most one of these cases must occur, and so we simply have to determine which if any of the two filters A_1 and B_1 contains the key, another instance of our original problem.

It may so happen again that both filters produce a positive; we apply the same idea recursively to solve this problem. Because each pair of filters only contains keys that are in the map *and* produced false positives on all previous filter pairs, the number of keys is extremely likely to quickly drop to a very small quantity that can be easily stored in an ordinary deterministic map, such as a pair of small arrays with linear search. Moreover, the average total search time is a constant, because almost all queries will be resolved by the first pair, almost all remaining queries by the second pair, and so on. The total space required is independent of n , and is almost entirely occupied by the first filter pair.

Now that we have the structure and a search algorithm, we also need to know how to insert new key/value pairs. The program must not attempt to insert the same key with both values. If the value is 0, insert the key into A_0 and then test if the key is in B_0 . If so, this is a false positive for B_0 , and the key must also be inserted into A_1 recursively in the same manner. If we reach the last level, we simply insert it. When the value is 1, the operation is similar but with A and B reversed.

Now that we can map a key to the value 0 or 1, how does this help us map to general values? This is simple. We create a single such Bloomier filter for each bit of the result. If the values are large, we can instead map keys to hash values that can be used to retrieve the actual values. The space required for a Bloomier filter with n -bit values is typically slightly more than the space for $2n$ Bloom filters.

A very simple way to implement Bloomier filters is by means of minimal perfect hashing. A minimal perfect hash function h is first generated for the set of n keys. Then an array is filled with n pairs (signature,value) associated with each key at the positions given by function h when applied on each key. The signature of a key is a string of r bits computed by applying a hash function g of range 2^r on the key. The value of r is chosen such that $2^r \geq 1/\epsilon$, where ϵ is the requested false positive rate. To query for a given key, hash function h is first applied on the key. This will give a position into the array from which we retrieve a pair (signature,value). Then we compute the signature of the key using function g . If the computed signature is the same as retrieved signature we return the retrieved value. The probability of false positive is $1/2^r$.

Another alternative to implement static bloomier and bloom filters based on matrix solving has been simultaneously proposed in Porat (2008) , Dietzfelbinger & Pagh (2008) and Charles & Chellapilla (2008). The space usage of this method is optimal as it needs only $\log_2(\epsilon)$ bits per key for a bloom filter. However time to generate the bloom or bloomier filter can be very high. The generation time can be reduced to a reasonable value at the price of a small increase in space usage.

Dynamic Bloomier filters have been studied by Mortensen, Pagh & Pătrașcu (2005). They proved that any dynamic Bloomier filter needs at least around $\log(l)$ bits per key where l is the length of the key. A simple dynamic version of Bloomier filters can be implemented using two dynamic data structures. Let the two data structures be noted S1 and S2. S1 will store keys with their associated data while S2 will only store signatures of keys with their associated data. Those signatures are simply hash values of keys in the range $[0, n/\epsilon]$ where n is the maximal number of keys to be stored in the Bloomier filter and ϵ is the requested false positive rate. To insert a key in the Bloomier filter, its

hash value is first computed. Then the algorithm checks if a key with the same hash value already exists in S2. If this is not the case, the hash value is inserted in S2 along with data associated with the key. If the same hash value already exists in S2 then the key is inserted into S1 along with its associated data. The deletion is symmetric: if the key already exists in S1 it will be deleted from there, otherwise the hash value associated with the key is deleted from S2. An issue with this algorithm is on how to store efficiently S1 and S2. For S1 any hash algorithm can be used. To store S2 golomb coding could be applied to compress signatures to use a space close to $\log 2(1/\epsilon)$ per key.

Compact approximators

Boldi & Vigna (2005) proposed a lattice-based generalization of Bloom filters. A **compact approximator** associates to each key an element of a lattice (the standard Bloom filters being the case of the Boolean two-element lattice). Instead of a bit array, they have an array of lattice elements. When adding a new association between a key and an element of the lattice, they maximize the current content of the k array locations associated to the key with the lattice element. When reading the value associated to a key, they minimize the values found in the k locations associated to the key. The resulting value approximates from above the original value.

Stable Bloom Filters

Deng & Rafiei (2006) proposed Stable Bloom filters as a variant of Bloom filters for streaming data. The idea is that since there is no way to store the entire history of a stream (which can be infinite), Stable Bloom continuously evicts the stale information to make room for those more recent elements. The authors show that a tight upper bound of false positive rates is guaranteed, and the method is superior in terms of both accuracy and time efficiency when a small space and an acceptable false positive rate are given.

Scalable Bloom Filters

Almeida et al. (2007) proposed a variant of Bloom filters that can adapt dynamically to the number of elements stored, while assuring a minimum false positive probability. The technique is based on sequences of standard bloom filters with increasing capacity and tighter false positive probabilities, so as to ensure that a maximum false positive probability can be set beforehand, regardless of the number of elements to be inserted.

References

- Agarwal, Sachin; Trachtenberg, Ari (2006), "Approximating the number of differences between remote sets", *IEEE Information Theory Workshop* (Punta del Este, Uruguay): 217, doi:10.1109/ITW.2006.1633815
- Ahmadi, Mahmood; Wong, Stephan (2007), "A Cache Architecture for Counting Bloom Filters" [6], *15th international Conference on Netwroks (ICON-2007)*
- Almeida, Paulo; Baquero, Carlos; Preguica, Nuno; Hutchison, David (2007), "Scalable Bloom Filters" [7], *Information Processing Letters* **101** (6): 255–261, doi:10.1016/j.ipl.2006.10.007
- Byers, John W.; Considine, Jeffrey; Mitzenmacher, Michael; Rost, Stanislav (2004), "Informed content delivery across adaptive overlay networks", *IEEE/ACM Transactions on Networking* **12** (5): 767, doi:10.1109/TNET.2004.836103
- Bloom, Burton H. (1970), "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM* **13** (7): 422–426, doi:10.1145/362686.362692
- Boldi, Paolo; Vigna, Sebastiano (2005), "Mutable strings in Java: design, implementation and lightweight text-search algorithms", *Science of Computer Programming* **54** (1): 3–23, doi:10.1016/j.scico.2004.05.003
- Bonomi, Flavio; Mitzenmacher, Michael; Panigrahy, Rina; Singh, Sushil; Varghese, George (2006), "An Improved Construction for Counting Bloom Filters", *Algorithms – ESA 2006, 14th Annual European Symposium*, **4168**, pp. 684–695, doi:10.1007/11841036_61

- Broder, Andrei; Mitzenmacher, Michael (2005), "Network Applications of Bloom Filters: A Survey" [8], *Internet Mathematics* **1** (4): 485–509
- Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson; Wallach, Deborah; Burrows, Mike; Chandra, Tushar; Fikes, Andrew *et al.* (2006), "Bigtable: A Distributed Storage System for Structured Data" [9], *Seventh Symposium on Operating System Design and Implementation*
- Charles, Denis; Chellapilla, Kumar (2008), "Bloomier Filters: A second look" [10], *The Computing Research Repository (CoRR)*
- Chazelle, Bernard; Kilian, Joe; Rubinfeld, Ronitt; Tal, Ayellet (2004), "The Bloomier filter: an efficient data structure for static support lookup tables" [11], *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 30–39
- Cohen, Saar; Matias, Yossi (2003), "Spectral Bloom Filters" [12], *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 241–252, doi:10.1145/872757.872787
- Deng, Fan; Rafiei, Davood (2006), "Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters" [13], *Proceedings of the ACM SIGMOD Conference*, pp. 25–36
- Dharmapurikar, Sarang; Song, Haoyu; Turner, Jonathan; Lockwood, John (2006), "Fast packet classification using Bloom filters" [14], *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pp. 61–70, doi:10.1145/1185347.1185356
- Dietzfelbinger, Martin; Pagh, Rasmus (2008), "Succinct Data Structures for Retrieval and Approximate Membership" [15], *The Computing Research Repository (CoRR)*
- Dillinger, Peter C.; Manolios, Panagiotis (2004a), "Fast and Accurate Bitstate Verification for SPIN" [16], *Proceedings of the 11th International Spin Workshop on Model Checking Software*, Springer-Verlag, Lecture Notes in Computer Science 2989
- Dillinger, Peter C.; Manolios, Panagiotis (2004b), "Bloom Filters in Probabilistic Verification" [17], *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, Springer-Verlag, Lecture Notes in Computer Science 3312
- Donnet, Benoit; Baynat, Bruno; Friedman, Timur (2006), "Retouched Bloom Filters: Allowing Networked Applications to Flexibly Trade Off False Positives Against False Negatives" [18], *CoNEXT 06 – 2nd Conference on Future Networking Technologies*
- Eppstein, David; Goodrich, Michael T. (2007), "Space-efficient straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters", *Algorithms and Data Structures, 10th International Workshop, WADS 2007*, Springer-Verlag, Lecture Notes in Computer Science 4619, pp. 637–648, arXiv:0704.3313
- Fan, Li; Cao, Pei; Almeida, Jussara; Broder, Andrei (2000), "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", *IEEE/ACM Transactions on Networking* **8** (3): 281–293, doi:10.1109/90.851975. A preliminary version appeared at SIGCOMM '98.
- Kirsch, Adam; Mitzenmacher, Michael (2006), "Less Hashing, Same Performance: Building a Better Bloom Filter" [19], *Algorithms – ESA 2006, 14th Annual European Symposium*, **4168**, Springer-Verlag, Lecture Notes in Computer Science 4168, pp. 456–467, doi:10.1007/11841036
- Mortensen, Christian Worm; Pagh, Rasmus; Pătrașcu, Mihai (2005), "On dynamic range reporting in one dimension", *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, pp. 104–111, doi:10.1145/1060590.1060606
- Pagh, Anna; Pagh, Rasmus; Rao, S. Srinivasa (2005), "An optimal Bloom filter replacement" [20], *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 823–829

- Porat, Ely (2008), "An Optimal Bloom Filter Replacement Based on Matrix Solving" ^[21], *The Computing Research Repository (CoRR)*
- Putze, F.; Sanders, P.; Singler, J. (2007), "Cache-, Hash- and Space-Efficient Bloom Filters" ^[22], *Experimental Algorithms, 6th International Workshop, WEA 2007, 4525*, Springer-Verlag, Lecture Notes in Computer Science 4525, pp. 108–121, doi:10.1007/978-3-540-72845-0
- Sethumadhavan, Simha; Desikan, Rajagopalan; Burger, Doug; Moore, Charles R.; Keckler, Stephen W. (2003), "Scalable hardware memory disambiguation for high ILP processors" ^[23], *36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, MICRO-36*, pp. 399–410, doi:10.1109/MICRO.2003.1253244
- Shanmugasundaram, Kulesh; Brönnimann, Hervé; Memon, Nasir (2004), "Payload attribution via hierarchical Bloom filters", *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pp. 31–41, doi:10.1145/1030083.1030089
- Starobinski, David; Trachtenberg, Ari; Agarwal, Sachin (2003), "Efficient PDA Synchronization", *IEEE Transactions on Mobile Computing* 2 (1): 40, doi:10.1109/TMC.2003.1195150
- Stern, Ulrich; Dill, David L. (1996), "A New Scheme for Memory-Efficient Probabilistic Verification" ^[24], *Proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification: IFIP TC6/WG6.1 Joint International Conference*, Chapman & Hall, IFIP Conference Proceedings, pp. 333–348

External links

- Table of false-positive rates for different configurations ^[25] from a University of Wisconsin–Madison website
- Bloom Filters and Social Networks with Java applet demo ^[26] from a Sun Microsystems website
- Interactive Processing demonstration ^[27] from ashcan.org
- "More Optimal Bloom Filters," Ely Porat (Nov/2007) Google TechTalk video ^[28] on YouTube

Implementations

- Implementation in C++ and Object Pascal ^[8] from partow.net
- Implementation in C ^[29] from literateprograms.org
- Implementation in Erlang ^[30] from sites.google.com
- Implementation in Haskell ^[31] from haskell.org
- Implementation in C# ^[32] from codeplex.com
- Implementation in Java ^[33] from tu-dresden.de
- Implementation in Javascript ^[34] from la.ma.la
- Implementation in Lisp ^[35] from lemonodor.com
- Implementation in Perl ^[36] from cpan.org
- Implementation in PHP ^[37] from code.google.com
- Implementation in Python, Scalable Bloom Filter ^[38] from python.org
- Implementation in Ruby ^[39] from rubyinside.com
- Implementation in Tcl ^[40] from kocjan.org

References

- [1] Donald Knuth. "[[The Art of Computer Programming (<http://www-cs-faculty.stanford.edu/~knuth/err3.txt>)], Errata for Volume 3 (2nd ed.)]. .
- [2] (Chang et al. 2006).
- [3] <http://wiki.squid-cache.org/SquidFaq/CacheDigests>
- [4] Wessels, Duane (January 2004). "10.7 Cache Digests". *Squid: The Definitive Guide* (1st ed.). O'Reilly Media. p. 172. ISBN 0596001622.
"Cache Digests are based on a technique first published by Pei Cao, called Summary Cache. The fundamental idea is to use a Bloom filter to represent the cache contents."
- [5] <http://plan9.bell-labs.com/magic/man2html/8/venti>
- [6] http://www.ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=4444031&arnumber=4444089&count=113&index=57
- [7] <http://gsd.di.uminho.pt/members/cbm/ps/dbloom.pdf>
- [8] <http://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf>
- [9] <http://labs.google.com/papers/bigtable.html>
- [10] <http://arxiv.org/pdf/0807.0928v1>
- [11] <http://www.ee.technion.ac.il/~ayellet/Ps/nelson.pdf>
- [12] <http://www.sigmod.org/sigmod03/eproceedings/papers/r09p02.pdf>
- [13] <http://webdocs.cs.ualberta.ca/~drafiei/papers/DupDet06Sigmod.pdf>
- [14] <http://www.arl.wustl.edu/~sarang/ancs6819-dharmapurikar.pdf>
- [15] <http://arxiv.org/pdf/0803.3693v1>
- [16] <http://www.cc.gatech.edu/fac/Pete.Manolios/research/spin-3spin.html>
- [17] <http://www.cc.gatech.edu/fac/Pete.Manolios/research/bloom-filters-verification.html>
- [18] http://adetti.iscte.pt/events/CONEXT06/Conext06_Proceedings/papers/13.html
- [19] <http://www.eecs.harvard.edu/~kirsch/pubs/bbbf/esa06.pdf>
- [20] <http://www.it-c.dk/people/pagh/papers/bloom.pdf>
- [21] <http://arxiv.org/pdf/0804.1845v1>
- [22] <http://algo2.iti.uni-karlsruhe.de/singler/publications/cacheefficientbloomfilters-wea2007.pdf>
- [23] <http://www.cs.utexas.edu/users/simha/publications/lsq.pdf>
- [24] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.4101>
- [25] <http://www.cs.wisc.edu/~cao/papers/summary-cache/node8.html>
- [26] http://blogs.sun.com/bblfish/entry/my_bloomin_friends
- [27] <http://tr.ashcan.org/2008/12/bloomers.html>
- [28] <http://www.youtube.com/watch?v=947gWqwkhu0>
- [29] [http://en.literateprograms.org/Bloom_filter_\(C\)](http://en.literateprograms.org/Bloom_filter_(C))
- [30] <http://sites.google.com/site/scalablebloomfilters/>
- [31] <http://hackage.haskell.org/cgi-bin/hackage-scripts/package/bloomfilter>
- [32] <http://codeplex.com/bloomfilter>
- [33] http://wwwse.inf.tu-dresden.de/xsiena/bloom_filter
- [34] <http://la.ma.la/misc/js/bloomfilter/>
- [35] <http://lemonodor.com/archives/000881.html>
- [36] <http://search.cpan.org/dist/Bloom-Filter/>
- [37] <http://code.google.com/p/php-bloom-filter/>
- [38] <http://pypi.python.org/pypi/pybloom/1.0.2>
- [39] <http://www.rubyinside.com/bloom-filters-a-powerful-tool-599.html>
- [40] <http://www.kocjan.org/tclmentor/61-bloom-filters-in-tcl.html>

Locality preserving hashing

In computer science, a **locality preserving hashing** is a hash function f that maps a point or points in a multidimensional coordinate space to a scalar value, such that if we have three points A , B and C such that

$$|A - B| < |B - C| \Rightarrow |f(A) - f(B)| < |f(B) - f(C)|.$$

In other words, these are hash functions where the relative distance between the input values is preserved in the relative distance between of the output hash values; input values that are closer to each other will produce output hash values that are closer to each other.

This is in contrast to cryptographic hash functions and checksums, which are designed to have maximum output difference between adjacent inputs.

Locality preserving hashes are related to space-filling curves

External links

- Locality-preserving hashing in multidimensional spaces ^[1]
- Locality-preserving hash functions for general purpose parallel computation ^[2]

References

- [1] <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=E79A120CE822B57F2198689803F42CF6?doi=10.1.1.50.4927&rep=rep1&type=pdf>
[2] <http://unclaw.com/chin/scholarship/index.htm#parallelalgorithms>

Morton number

In mathematical number theory and computer science, a **Morton number** is a single integer value constructed by interleaving the bits or digits of one or more source numbers. This is often useful for constructing a single hash index from a pair (or more) of input numbers.

In number theory, Morton numbers are useful in proofs, often in examples which map multiple dimensions to one, or vice versa. For example, an infinite 2D grid of integer coordinates can have a single unique Morton number computed for each coordinate, and those Morton numbers give a one-to-one mapping of the infinite 2D coordinates to a 1D coordinate—along the Z-order curve—proving that the infinite number of integer pairs has the same cardinality as the integers.

External links

- Bits interleaving in C ^[1]
- python example ^[2]

References

- [1] <http://graphics.stanford.edu/~seander/bithacks.html#InterleaveTableObvious>
[2] <http://www.codexon.com/posts/morton-codes/>

Zobrist hashing

Zobrist hashing (also referred to as **Zobrist keys** or **Zobrist signatures** ^[1]) is a hash function construction used in computer programs that play abstract board games, such as chess and Go, to implement transposition tables, a special kind of hash table that is indexed by a board position and used to avoid analyzing the same position more than once. Zobrist hashing is named for its inventor, Albert Zobrist.^[2]

Zobrist hashing starts by randomly generating bitstrings for each possible element of a board game. Given a certain board position, it breaks up the board into independent components, finds out what state each component is in, and combines the bitstrings representing those elements together using bitwise XOR. If the bitstrings are long enough, different board positions will almost certainly hash to different values; however longer bitstrings require proportionally more computer resources to manipulate. Many game engines store only the hash values in the transposition table, omitting the position information itself entirely to reduce memory usage, and assuming that hash collisions will not occur, or will not greatly influence the results of the table if they do.

As an example, in chess, each of the 64 squares can at any time be empty, or contain one of the 6 game pieces, which are either black or white. That is, each square can be in one of $1 + 6 \times 2 = 13$ possible states at any time. Thus one needs to generate at most $13 \times 64 = 832$ random bitstrings. Given a position, one obtains its Zobrist hash by finding out which pieces are on which squares, and combining the relevant bitstrings together.

The position of a board can be updated simply by XORing out the bitstring(s) for states which have changed, and XORing in the bitstrings for the new states. For instance, if a pawn on a chessboard square is replaced by a rook from another square, the resulting position would be produced by XORing the existing hash with the bitstrings for:

```
'pawn at this square'      (XORing out the pawn at this square)
'rook at this square'      (XORing in the rook at this square)
'rook at source square'    (XORing out the rook at the source square)
'nothing at source square' (XORing in nothing at the source square).
```

This makes Zobrist hashing very efficient for traversing a game tree.

In computer go, this technique is also used for superko detection.

See also

- Alpha-beta pruning

References

- [1] Zobrist keys: a means of enabling position comparison. (<http://web.archive.org/web/20070822204038/http://www.seanet.com/~brucemo/topics/zobrist.htm>)
- [2] Albert L. Zobrist, *A New Hashing Method with Application for Game Playing* (<https://www.cs.wisc.edu/techreports/1970/TR88.pdf>), Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, (1969).

Rolling hash

A **rolling hash** is a hash function where the input is hashed in a window that moves through the input.

A few hash functions allow a rolling hash to be computed very quickly -- the new hash value is rapidly calculated given only the old hash value, the old value removed from the window, and the new value added to the window -- similar to the way a moving average function can be computed much more quickly than other low-pass filters.

One of the main applications is the Rabin-Karp string search algorithm, which uses the rolling hash described below.

Another popular application is rsync program which uses a checksum based on Mark Adler's adler-32 as its rolling hash.

Rabin-Karp rolling hash

The Rabin-Karp string search algorithm is normally used with a very simple rolling hash function that only uses multiplications and additions:

$$H = c_1 a^{k-1} + c_2 a^{k-2} + c_3 a^{k-3} + \dots + c_k a^0 \text{ where } a \text{ is a constant and } c_1, \dots, c_k \text{ are the input characters.}$$

In order to avoid manipulating huge H values, all math is done modulo n . The choice of a and n is critical to get good hashing; see linear congruential generator for more discussion.

Removing and adding chars simply involves adding or subtracting the first or last term. Shifting all chars by one position to the left requires multiplying the entire sum H by a . Shifting all chars by one position to the right requires dividing the entire sum H by a . Note that in modulo arithmetic, a can be chosen to have a multiplicative inverse a^{-1} by which H can be multiplied to get the result of the division without actually performing a division.

Cyclic polynomial

Hashing by cyclic polynomial^[1] —sometimes called Buzhash^[2]—is also simple, but it has the benefit of avoiding multiplications, using barrel shifts instead. It presumes that there is some hash function h from characters to integers in the interval $[0, 2^L)$. This hash function might be simply an array or a hash table mapping characters to random integers. Let the function s be a cyclic binary rotation (or barrel shift): it rotates the bits by 1 to the left, pushing the latest bit in the first position. E.g., $s(10011) = 00111$. Let \oplus be the bit-wise exclusive or. The hash values are defined as

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \dots \oplus s(h(c_{k-1})) \oplus h(c_k)$$

where the multiplications by powers of two can be implemented by binary shifts. The result is a number in $[0, 2^L)$.

Computing the hash values in a rolling fashion is done as follows. Let H be the previous hash value. Rotate H once: $H \leftarrow s(H)$. If c_1 is the character to be removed, rotate it k times: $s^k(h(c_1))$. Then simply set

$$H \leftarrow H \oplus s^k(h(c_1)) \oplus h(c_{k+1})$$

where c_{k+1} is the new character.

Independence

At best, rolling hash values are pairwise independent^[3]. Similarly, at best the (randomized) Rabin-Karp rolling hash values are independent. Hashing n-grams by cyclic polynomials achieves pairwise independence: simply keep the first $L - n + 1$ bits.

Computational complexity

All rolling hash functions are linear in the number of characters, but their complexity with respect to the length of the window (k) varies. Rabin-Karp rolling hash requires the multiplications of two k -bit numbers, integer multiplication is in $O(k \log k 2^{O(\log^* k)})$ ^[4]. Hashing n-grams by cyclic polynomials can be done in linear time^[3].

Software

- ngramhashing^[5] is a Free software C++ implementation of several rolling hash functions

References

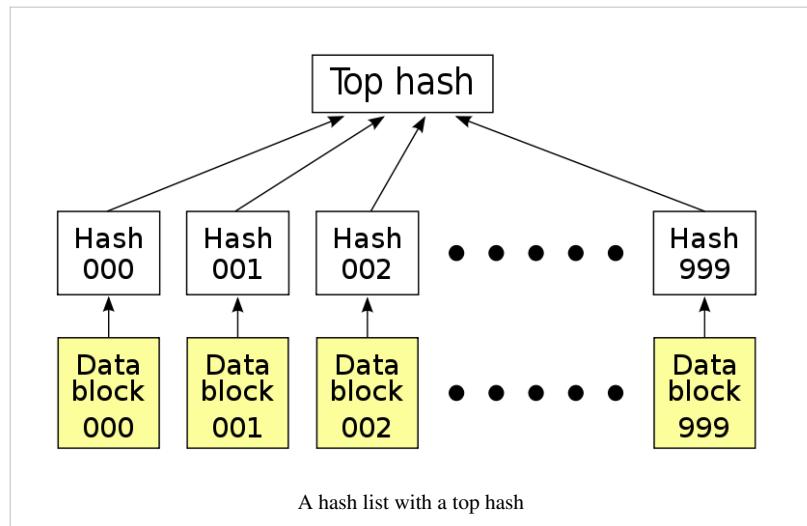
- [1] Jonathan D. Cohen, Recursive hashing functions for n-grams, ACM Trans. Inf. Syst. 15 (3), 1997
- [2] <https://www.se.auckland.ac.nz/courses/SOFTENG250/archive/2006/assignments/Hashing/BuzHash.java>
- [3] Daniel Lemire, Owen Kaser: Recursive n-gram hashing is pairwise independent, at best, arXiv:0705.4676
- [4] M. Fürer, Faster integer multiplication, in: STOC '07, 2007, pp. 57–66.
- [5] <http://code.google.com/p/ngramhashing/>

Hash list

In computer science, a **hash list** is typically a list of hashes of the data blocks in a file or set of files. Lists of hashes are used for many different purposes, such as fast table lookup (hash tables) and distributed databases (distributed hash tables). This article covers hash lists that are used to guarantee data integrity.

A hash list is an extension of the old concept of hashing an item (for instance, a file). A hash list is usually sufficient for most needs, but a more advanced form of the concept is a hash tree.

Hash lists can be used to protect any kind of data stored, handled and transferred in and between computers. An important use of hash lists is to make sure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and to check that the other peers do not "lie" and send fake blocks.



Usually a cryptographic hash function such as SHA-1 is used for the hashing. If the hash list only needs to protect against unintentional damage less secure checksums such as CRCs can be used.

Hash lists are better than a simple hash of the entire file since, in the case of a data block being damaged, this is noticed, and only the damaged block needs to be redownloaded. With only a hash of the file, many undamaged blocks would have to be redownloaded, and the file reconstructed and tested until the correct hash of the entire file is obtained. Hash lists also protect against nodes that try to sabotage by sending fake blocks, since in such a case the damaged block can be acquired from some other source.

Often, an additional hash of the hash list itself (a *top hash*, also called *root hash* or *master hash*) is used. Before downloading a file on a p2p network, in most cases the top hash is acquired from a trusted source, for instance a friend or a web site that is known to have good recommendations of files to download. When the top hash is available, the hash list can be received from any non-trusted source, like any peer in the p2p network. Then the received hash list is checked against the trusted top hash, and if the hash list is damaged or fake, another hash list from another source will be tried until the program finds one that matches the top hash.

In some systems (like for example BitTorrent), instead of a top hash the whole hash list is available on a web site in a small file. Such a "torrent file" contains a description, file names, a hash list and some additional data.

See also

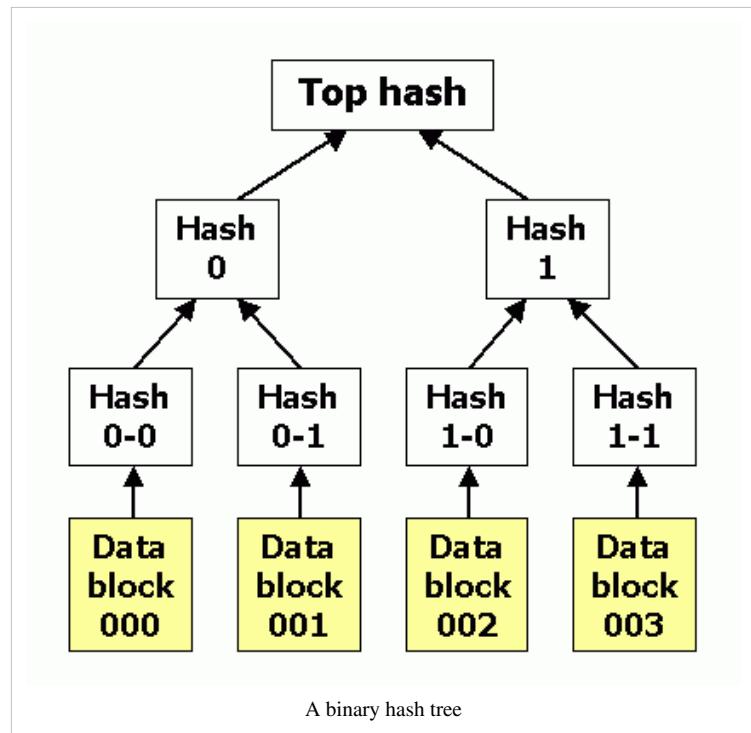
- Hash tree
- Hash table
- Hash chain
- Ed2k: URI scheme, which uses an MD4 top hash of an MD4 hash list to uniquely identify a file
- Cryptographic hash function
- List

Hash tree

In cryptography and computer science **Hash trees** or **Merkle trees** are a type of data structure which contains a tree of summary information about a larger piece of data – for instance a file – used to verify its contents. Hash trees are an extension of hash lists, which in turn are an extension of hashing. Hash trees where the underlying hash function is Tiger are often called **Tiger trees** or **Tiger tree hashes**.

Uses

Hash trees can be used to protect any kind of data stored, handled and transferred in and between computers. Currently the main use of hash trees is to make sure that data blocks received from other peers in a peer-to-peer network are received



undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks. Suggestions have been made to use hash trees in trusted computing systems. Sun Microsystems has used Hash Trees in the ZFS filesystem^[1]. Hash Trees are used in Google Wave protocol^[2] and in tarsnap backup system.

Hash trees were invented in 1979 by Ralph Merkle^[3]. The original purpose was to make it possible to efficiently handle many Lamport one-time signatures. Lamport signatures are believed to still be secure in the event that quantum computers become reality. Unfortunately each Lamport key can only be used to sign a single message. But combined with hash trees they can be used for many messages and then become a fairly efficient digital signature scheme.

How hash trees work

A hash tree is a tree of hashes in which the leaves are hashes of data blocks in, for instance, a file or set of files. Nodes further up in the tree are the hashes of their respective children. For example, in the picture *hash 0* is the result of hashing *hash 0-0* and then *hash 0-1*. That is, $\text{hash } 0 = \text{hash}(\text{hash } 0-0 + \text{hash } 0-1)$ where + denotes concatenation.

Most hash tree implementations are binary (two child nodes under each node) but they can just as well use many more child nodes under each node.

Usually, a cryptographic hash function such as SHA-1, Whirlpool, or Tiger is used for the hashing. If the hash tree only needs to protect against unintentional damage, much less secure checksums such as CRCs can be used.

In the top of a hash tree there is a *top hash* (or *root hash* or *master hash*). Before downloading a file on a p2p network, in most cases the top hash is acquired from a trusted source, for instance a friend or a web site that is known to have good recommendations of files to download. When the top hash is available, the hash tree can be received from any non-trusted source, like any peer in the p2p network. Then, the received hash tree is checked against the trusted top hash, and if the hash tree is damaged or fake, another hash tree from another source will be tried until the program finds one that matches the top hash.

The main difference from a hash list is that one branch of the hash tree can be downloaded at a time and the integrity of each branch can be checked immediately, even though the whole tree is not available yet. This can be an advantage since it is efficient to split files up in very small data blocks so that only small blocks have to be redownloaded if they get damaged. If the hashed file is very big, such a hash tree or hash list becomes fairly big. But if it is a tree, one small branch can be downloaded quickly, the integrity of the branch can be checked, and then the downloading of data blocks can start.

There are several additional tricks, benefits and details regarding hash trees. See the references and external links below for more in-depth information.

Tiger tree hash

The Tiger tree hash is probably the most widely used form of hash tree. It uses a binary hash tree (two child nodes under each node), usually has a data block size of 1024-bytes and uses the cryptographically secure Tiger hash.

Tiger tree hashes are used in the Gnutella, Gnutella2, and Direct Connect P2P file sharing protocols and in file sharing applications such as Phex, BearShare, LimeWire, Shareaza, DC++ and Valknut.

See also

- Cryptographic hash function
- Binary tree which is a tree data structure.
- Ralph Merkle – The cryptographer who invented hash trees (Merkle trees).
- Lamport signature – Hash trees were originally invented to make it possible to handle Lamport signatures efficiently.

- Hash table - Not to be confused with...
- Hash trie
- Linked Timestamping

References

- Merkle tree patent 4,309,569 ^[4] – Explains both the hash tree structure and the use of it to handle many one-time signatures.
- Tree Hash EXchange format (THEX) ^[5] – A detailed description of Tiger trees.
- Efficient Use of Merkle Trees ^[6] – RSA labs explanation of the original purpose of Merkle trees: To handle many Lamport one-time signatures.

[1] Jeff Bonwick's Blog *ZFS End-to-End Data Integrity* (http://blogs.sun.com/bonwick/entry/zfs_end_to_end_data)

[2] Google Wave Federation Protocol *Wave Protocol Verification Paper* (<http://www.waveprotocol.org/whitepapers/wave-protocol-verification>)

[3] R. C. Merkle, *A digital signature based on a conventional encryption function*, Crypto '87

[4] <http://www.google.com/patents?q=4309569>

[5] <http://web.archive.org/web/20080316033726/http://www.open-content.net/specs/draft-jchapweske-thex-02.html>

[6] <http://www.rsasecurity.com/rsalabs/node.asp?id=2003>

External links

- <http://www.codeproject.com/cs/algorithms/thexlsx.asp> – Tiger Tree Hash (TTH) source code in C# – by Gil Schmidt
- <http://sourceforge.net/projects/tigertree/> – Tiger Tree Hash (TTH) implementations in C and Java
- <http://codingplayground.blogspot.com/2009/03/hashing-shingling-and-hashtrees.html> a Boost implementation in C++ by Antonio Gulli
- RHash (<http://rhash.sourceforge.net/>), an open source command-line tool, which can calculate TTH and magnet links with TTH.

Prefix hash tree

A **prefix hash tree** (PHT) is a distributed data structure that enables more sophisticated queries over a distributed hash table (DHT). The prefix hash tree uses the lookup interface of a DHT to construct a trie-based data structure that is both efficient (updates are doubly logarithmic in the size of the domain being indexed), and resilient (the failure of any given node in a prefix hash tree does not affect the availability of data stored at other nodes).

External links

- <http://berkeley.intel-research.net/sylvia/pht.pdf> - *Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables*
- <http://pier.cs.berkeley.edu> - PHT was developed as part of work on the PIER project.

See also

- Prefix tree
- P-Grid

Hash trie

In computer science, **hash trie** refers to two kinds of data structure:

- A space-efficient implementation of a sparse trie, in which the descendants of each node may be interleaved in memory. (The name is suggested by a similarity to a closed hash table.)^[1]
- An ordinary trie used to store hash values, for example, in an implementation of a hash tree.
- A data structure which "combines features of hash tables and LC-tries in order to perform efficient lookups and updates"^[2]

References

- [1] Liang, Frank (June 1983), *Word hy-phen-a-tion by com-pu-ter* (<http://www.tug.org/docs/liang-liang-thesis.pdf>), Frank M. Liang, Ph.D. thesis, Stanford University., , retrieved 2010-03-28
- [2] Thomas, Roshan; Mark, Brian; Johnson, Tommy; Croall, James (2004), *High-speed Legitimacy-based DDoS Packet Filtering with Network Processors: A Case Study and Implementation on the Intel IXP1200* (<http://nopl.gmu.edu/pubs/BookContrib/ThomasMarkJC-NPW04.pdf>), , retrieved 2009-05-03

Hash array mapped trie

A **hash array mapped trie**^[1] (HAMT) is an implementation of an associative array that combines the characteristics of a hash table and an array mapped trie^[2].

Problems with HAMTs

Implementation of a HAMT involves the use of the population count function, which counts the number of ones in the binary representation of a number. This operation is available in many instruction set architectures (where it is sometimes called "CTPOP"), but it is only available in some high-level languages. Although population count can be implemented in software in O(1) time using a series of shift and add instructions, doing so may perform the operation an order of magnitude slower.

Implementations

The programming language Clojure uses a persistent variant of hash array mapped tries for its native hash map type.^[3]

References

- [1] Bagwell, P. (2001) Ideal Hash Trees (<http://lampwww.epfl.ch/papers/idealhashtrees.pdf>). Technical Report, 2001.
- [2] Bagwell, P. (2000) Fast And Space Efficient Trie Searches (<http://citeseer.ist.psu.edu/rd/61998956,282572,1,0,25,Download/http:qSqqSqlampwww.epfl.chqSqpapersqSqtresearches.pdf.gz>). Technical Report, 2000.
- [3] Java source file of Clojure's hash map type. (<http://github.com/richhickey/clojure/blob/14316ae2110a779ffc8ac9c3da3f1c41852c4289/src/jvm/clojure/lang/PersistentHashMap.java>)

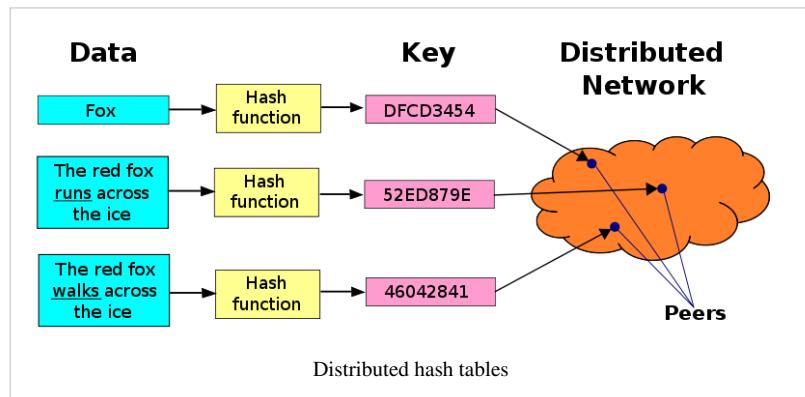
Distributed hash table

Distributed hash tables (DHTs) are a class of decentralized distributed systems that provide a lookup service similar to a hash table; *(key, value)* pairs are stored in the DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows DHTs to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

DHTs form an infrastructure that can be used to build more complex services, such as distributed file systems, peer-to-peer file sharing and content distribution systems, cooperative web caching, multicast, anycast, domain name services, and instant messaging. Notable distributed networks that use DHTs include BitTorrent's distributed tracker, the Kad network, the Storm botnet, YaCy, and the Coral Content Distribution Network.

History

DHT research was originally motivated, in part, by peer-to-peer systems such as Napster, Gnutella, and Freenet, which took advantage of resources distributed across the Internet to provide a single useful application. In particular, they took advantage of increased bandwidth and hard disk capacity to provide a file sharing service.



These systems differed in how they *found* the data their peers contained:

- Napster had a central index server: each node, upon joining, would send a list of locally held files to the server, which would perform searches and refer the querier to the nodes that held the results. This central component left the system vulnerable to attacks and lawsuits.
- Gnutella and similar networks moved to a flooding query model—in essence, each search would result in a message being broadcast to every other machine in the network. While avoiding a single point of failure, this method was significantly less efficient than Napster.
- Finally, Freenet was also fully distributed, but employed a heuristic key-based routing in which each file was associated with a key, and files with similar keys tended to cluster on a similar set of nodes. Queries were likely to be routed through the network to such a cluster without needing to visit many peers. However, Freenet did not guarantee that data would be found.

Distributed hash tables use a more structured key-based routing in order to attain both the decentralization of Gnutella and Freenet, and the efficiency and guaranteed results of Napster. One drawback is that, like Freenet, DHTs only directly support exact-match search, rather than keyword search, although that functionality can be layered on top of a DHT.

The first DHT implementation—the Beyond Borders system^[1] —was introduced in 1998 and was based on Plaxton, Rajaraman, and Richa's algorithm.^[2] In 2001, four systems—CAN, Chord,^[3] Pastry, and Tapestry—ignited DHTs as a popular research topic, and this area of research remains active. Outside academia, DHT technology has been adopted as a component of BitTorrent and in the Coral Content Distribution Network.

Properties

DHTs characteristically emphasize the following properties:

- Decentralization: the nodes collectively form the system without any central coordination.
- Scalability: the system should function efficiently even with thousands or millions of nodes.
- Fault tolerance: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.

A key technique used to achieve these goals is that any one node needs to coordinate with only a few other nodes in the system – most commonly, $O(\log n)$ of the n participants (see below) – so that only a limited amount of work needs to be done for each change in membership.

Some DHT designs seek to be secure against malicious participants^[4] and to allow participants to remain anonymous, though this is less common than in many other peer-to-peer (especially file sharing) systems; see anonymous P2P.

Finally, DHTs must deal with more traditional distributed systems issues such as load balancing, data integrity, and performance (in particular, ensuring that operations such as routing and data storage or retrieval complete quickly).

Structure

The structure of a DHT can be decomposed into several main components.^[5] ^[6] The foundation is an abstract **keyspace**, such as the set of 160-bit strings. A **keyspace partitioning** scheme splits ownership of this keyspace among the participating nodes. An **overlay network** then connects the nodes, allowing them to find the owner of any given key in the keyspace.

Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings. To store a file with given *filename* and *data* in the DHT, the SHA-1 hash of *filename* is generated, producing a 160-bit key k , and a message $\text{put}(k, \text{data})$ is sent to any node participating in the DHT. The message is forwarded from node to node through the overlay network until it reaches the single node responsible for key k as specified by the keyspace partitioning. That node then stores the key and the data. Any other client can then retrieve the contents of the file by again hashing *filename* to produce k and asking any DHT node to find the data associated with k with a message $\text{get}(k)$. The message will again be routed through the overlay to the node responsible for k , which will reply with the stored *data*. The keyspace partitioning and overlay network components are described below with the goal of capturing the principal ideas common to most DHTs; many designs differ in the details.

Keyspace partitioning

Most DHTs use some variant of consistent hashing to map keys to nodes. This technique employs a function $\delta(k_1, k_2)$ which defines an abstract notion of the *distance* from key k_1 to key k_2 , which is unrelated to geographical distance or network latency. Each node is assigned a single key called its *identifier* (ID). A node with ID i owns all the keys for which i is the closest ID, measured according to δ .

Example. The Chord DHT treats keys as points on a circle, and $\delta(k_1, k_2)$ is the distance traveling clockwise around the circle from k_1 to k_2 . Thus, the circular keyspace is split into contiguous segments whose endpoints are the node identifiers. If i_1 and i_2 are two adjacent IDs, then the node with ID i_2 owns all the keys that fall between i_1 and i_2 .

Consistent hashing has the essential property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. Contrast this with a traditional hash table in which addition or removal of one bucket causes nearly the entire keyspace to be remapped. Since any change in ownership typically corresponds to bandwidth-intensive movement of objects stored in the DHT from one node to another, minimizing such reorganization is required to efficiently support high rates of churn (node arrival and failure).

Overlay network

Each node maintains a set of links to other nodes (its *neighbors* or routing table). Together these links form the overlay network. A node picks its neighbors according to a certain structure, called the network's topology.

All DHT topologies share some variant of the most essential property: for any key k , the node either has a node ID which owns k or has a link to a node whose node ID is *closer* to k , in terms of the keyspace distance defined above. It is then easy to route a message to the owner of any key k using the following greedy algorithm (that is not necessarily globally optimal): at each step, forward the message to the neighbor whose ID is closest to k . When there is no such neighbor, then we must have arrived at the closest node, which is the owner of k as defined above. This style of routing is sometimes called key-based routing.

Beyond basic routing correctness, two important constraints on the topology are to guarantee that the maximum number of hops in any route (route length) is low, so that requests complete quickly; and that the maximum number of neighbors of any node (maximum node degree) is low, so that maintenance overhead is not excessive. Of course, having shorter routes requires higher maximum degree. Some common choices for maximum degree and route length are as follows, where n is the number of nodes in the DHT, using Big O notation:

- Degree $O(1)$, route length $O(n)$
- Degree $O(\log n)$, route length $O(\log n / \log \log n)$
- Degree $O(\log n)$, route length $O(\log n)$
- Degree $O(\sqrt{n})$, route length $O(1)$

The third choice is the most common, even though it is not quite optimal in terms of degree/route length tradeoff, because such topologies typically allow more flexibility in choice of neighbors. Many DHTs use that flexibility to pick neighbors which are close in terms of latency in the physical underlying network.

Maximum route length is closely related to diameter: the maximum number of hops in any shortest path between nodes. Clearly the network's route length is at least as large as its diameter, so DHTs are limited by the degree/diameter tradeoff^[7] which is fundamental in graph theory. Route length can be greater than diameter since the greedy routing algorithm may not find shortest paths.^[8]

Algorithms for overlay networks

Aside from routing, there exist many algorithms which exploit the structure of the overlay network for sending a message to all nodes, or a subset of nodes, in a DHT.^[9] These algorithms are used by applications to do overlay multicast, range queries, or to collect statistics.

DHT implementations

Most notable differences encountered in practical instances of DHT implementations include at least the following:

- The address space is a parameter of DHT. Several real world DHTs use 128 bit or 160 bit key space
- Some real-world DHTs use hash functions other than SHA1.
- In the real world the key k could be a hash of a file's *content* rather than a hash of a file's *name*, so that renaming of the file does not prevent users from finding it.
- Some DHTs may also publish objects of different types. For example, key k could be node ID and associated data could describe how to contact this node. This allows publication of presence information and often used in IM applications, etc. In simplest case ID is just a random number which is directly used as key k (so in a 160-bit DHT ID will be a 160 bit number, usually randomly chosen). In some DHTs publishing of nodes IDs is also used to optimize DHT operations.
- Redundancy can be added to improve reliability. The $(k, data)$ key pair can be stored in more than one node corresponding to the key. Usually, rather than selecting just one node, real world DHT algorithms select i suitable nodes, with i being an implementation-specific parameter of the DHT. In some DHT designs, nodes agree to handle a certain keyspace range, the size of which may be chosen dynamically, rather than hard-coded.
- Some advanced DHTs like Kademlia perform iterative lookups through the DHT first in order to select a set of suitable nodes and send $put(k, data)$ messages only to those nodes, thus drastically reducing useless traffic, since published messages are only sent to nodes which seem suitable for storing the key k ; and iterative lookups cover just a small set of nodes rather than the entire DHT, reducing useless forwarding. In such DHTs forwarding of $put(k, data)$ messages may only occur as part of a self-healing algorithm: if a target node receives a $put(k, data)$ message but believes that k is out of its handled range and a closer node (in terms of DHT keyspace) is known, the message is forwarded to that node. Otherwise, data are indexed locally. This leads to a somewhat self-balancing DHT behavior. Of course, such an algorithm requires nodes to publish their presence data in the DHT so the iterative lookups can be performed.

Examples

DHT protocols and implementations

- Apache Cassandra
- BitTorrent DHT - based on Kademlia as provided by Khashmir.^[10]
- CAN (Content Addressable Network)
- Chord (DHT)
- Kademlia
- Pastry (DHT)
- P-Grid
- Tapestry (DHT)

Applications employing DHTs

- Codeen: Web caching
- Coral Content Distribution Network
- Freenet: A censorship-resistant anonymous network
- Dijjer: Freenet-like distribution network
- FAROO: Peer-to-peer web search engine
- GNUnet: Freenet-like distribution network including a DHT implementation
- JXTA: Opensource P2P platform
- YaCy: distributed search engine
- maidsafe: c++ implementation of Kademlia (BSD license), with NAT traversal and crypto libraries

See also

- memcached: a high-performance, distributed memory object caching system
- Prefix hash tree: sophisticated querying over DHTs

External links

- Distributed Hash Tables, Part 1^[11] by Brandon Wiley.
- Distributed Hash Tables links^[12] Carles Pairot's Page on DHT and P2P research
- Tangosol Coherence^[13] includes a structure similar to a DHT, though all nodes have knowledge of the other participants
- kademlia.scs.cs.nyu.edu^[14] Archive.org snapshots of kademlia.scs.cs.nyu.edu
- Hazelcast^[15] open source DHT implementation

References

- [1] Renu Tewari, Mike Dahlin, Harrick Vin, and John Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet (<ftp://ftp.cs.utexas.edu/pub/techreports/tr98-04.ps>). University of Texas at Austin Computer Sciences Department Technical Report number TR98-04, February 1998.
- [2] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment (<http://doi.acm.org/10.1145/258492.258523>). In Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures, Newport, Rhode Island, pages 311-320, June 1997.
- [3] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems (<http://www.cs.berkeley.edu/~istoica/papers/2003/cacm03.pdf>). In Communications of the ACM, February 2003.
- [4] Guido Urdaneta, Guillaume Pierre and Maarten van Steen. A Survey of DHT Security Techniques (http://www.globule.org/publi/SDST_acmcs2009.html). ACM Computing Surveys, 2009.
- [5] Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach (<http://www.wisdom.weizmann.ac.il/~naor/PAPERS/dh.pdf>). Proc. SPAA, 2003.
- [6] Gurmeet Singh Manku. Dipsea: A Modular Distributed Hash Table (<http://www-db.stanford.edu/~manku/phd/index.html>). Ph. D. Thesis (Stanford University), August 2004.
- [7] The (Degree,Diameter) Problem for Graphs (http://maite71.upc.es/grup_de_grafs/table_g.html)
- [8] Gurmeet Singh Manku, Moni Naor, and Udi Wieder. Know thy Neighbor's Neighbor: the Power of Lookahead in Randomized P2P Networks (<http://citeseer.ist.psu.edu/naor04know.html>). Proc. STOC, 2004.
- [9] Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables (<http://www.sics.se/~ali/thesis/>). KTH-Royal Institute of Technology, 2006.
- [10] Tribler wiki (<http://www.tribler.org/trac/wiki/Khashmir>) retrieved January 2010.
- [11] <http://linuxjournal.com/article/6797>
- [12] <http://deim.urv.cat/~cpairot/dhts.html>
- [13] <http://wiki.tangosol.com/display/COH32UG/Partitioned+Cache+Service>
- [14] [http://web.archive.org/web/*/<http://kademlia.scs.cs.nyu.edu/>](http://web.archive.org/web/*/http://kademlia.scs.cs.nyu.edu/)
- [15] <http://code.google.com/p/hazelcast/>

Consistent hashing

Consistent hashing is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped. By using consistent hashing, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of slots.

History

Consistent hashing was introduced in 1997 as a way of distributing requests among a changing population of web servers. Each slot is then represented by a node in a distributed system. The addition (joins) and removal (leaves/failures) of nodes only requires K/n items to be re-shuffled when the number of slots/nodes change. More recently it has been used to reduce the impact of partial system failures in large web applications as to allow for robust caches without incurring the system wide fallout of a failure ^[1] ^[2].

More recently, consistent hashing has been applied in the design of distributed hash tables (DHTs). DHTs use consistent hashing to partition a keyspace among a distributed set of nodes, and additionally provide an overlay network which connects nodes such that the node responsible for any key can be efficiently located.

Technique

Consistent hashing is based on mapping items to a real angle (or equivalently a point on the edge of a circle). Slots correspond to angle ranges. Slots can be added or removed by either slightly readjusting all the angle ranges or just a subset of them (with the condition that every angle is assigned to one slot).

External links

- Understanding Consistent hashing ^[3]
- Consistent hashing ^[4] by Tom White, including implementation in Java
- Consistent hashing implementation in Erlang ^[5]
- Ketama: Consistent Hashing (Audioscrobbler) ^[6]

References

- [1] Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. (1997). "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web" (<http://portal.acm.org/citation.cfm?id=258660>). *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. ACM Press New York, NY, USA. pp. 654–663. doi:10.1145/258533.258660. . Retrieved 2008-06-17.
- [2] Karger, D.; Sherman, A.; Berkheimer, A.; Bogstad, B.; Dhanidina, R.; Iwamoto, K.; Kim, B.; Matkins, L.; Yerushalmi, Y. (1999). "Web Caching with Consistent Hashing" (<http://www8.org/w8-papers/2a-webserver/caching/paper2.html>). *Computer Networks* **31** (11): 1203–1213. doi:10.1016/S1389-1286(99)00055-9. . Retrieved 2008-06-17.
- [3] <http://www.spiteful.com/2008/03/17/programmers-toolbox-part-3-consistent-hashing/>
- [4] <http://www.lexemtech.com/2007/11/consistent-hashing.html>
- [5] http://bitbucket.org/basho/riak/src/tip/apps/riak_core/src/chash.erl
- [6] <http://www.audioscrobbler.net/development/ketama/>

Stable hashing

Stable hashing is a tool used to implement randomized load balancing and distributed lookup in peer-to-peer computer systems.

See also

- Hash function
- Consistent hashing

Koorde

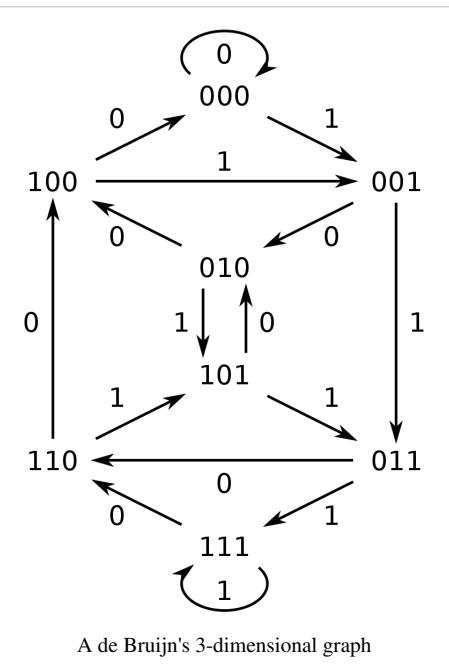
In peer-to-peer networks, **Koorde** is a Distributed hash table (DHT) system based on the Chord DHT and the De Bruijn graph (De Bruijn sequence). Inheriting the simplicity of Chord, Koorde meets $O(\log n)$ hops per node (where n is the number of nodes in the DHT), and $O(\log n / \log \log n)$ hops per lookup request with $O(\log n)$ neighbors per node.

The Chord concept is based on a wide range of identifiers (i.e. 2^{160}) in a structure of a ring where an identifier can stand for both node and data. Node-successor is responsible for the whole range of IDs between itself and its predecessor.

De Bruijn's graphs

Koorde is based on Chord but also on De Bruijn graph (De Bruijn sequence). In a d -dimensional de Bruijn graph, there are $3d$ nodes, each of which has a unique d -bit ID. The node with ID i is connected to nodes $2i$ modulo $2d$ and $2i+1$ modulo $2d$. Thanks to this property, the routing algorithm can route to any destination in d hops by successively „shifting in” the bits of the destination ID but only if the dimensions of the distance between modulo $1d$ and $3d$ are equal.

Routing a message from node m to node k is accomplished by taking the number m and shifting in the bits of k one at a time until the number has been replaced by k . Each shift corresponds to a routing hop to the next intermediate address; the hop is valid because each node's neighbors are the two possible outcomes of shifting a 0 or 1 onto its own address. Because of the structure of de Bruijn graphs, when the last bit of k has been shifted, the query will be at node k . Node k responds whether key k exists.



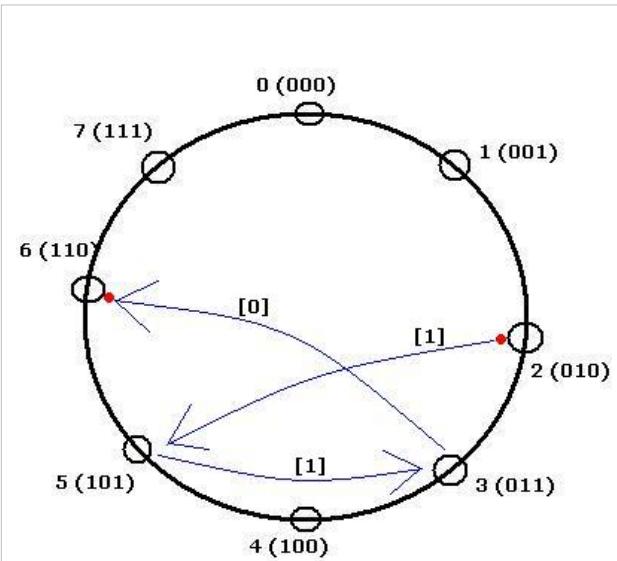
Routing example

For example, when a message needs to be routed from node “2” (which is “010”) to “6” (which is “110”), the steps are following:

Step 1) Node #2 routes the message to Node #5 (using its connection to $2i+1 \bmod 8$), shifts the bits left and puts “1” as the youngest bit (right side).

Step 2) Node #5 routes the message to Node #3 (using its connection to $2i+1 \bmod 8$), shifts the bits left and puts “1” as the youngest bit (right side).

Step 3) Node #3 routes the message to Node #6 (using its connection to $2i \bmod 8$), shifts the bits left and puts “0” as the youngest bit (right side).



Example of the way Koorde routes from Node2 to Node6 using a 3-dimensional, binary graph.

Non-constant degree Koorde

The d -dimensional de Bruijn can be generalized to base k , in which case node i is connected to nodes $k * i + j$ modulo kd , $0 \leq j < k$. The diameter is reduced to $\Theta(\log k n)$. Koorde node i maintains pointers to k consecutive nodes beginning at the predecessor of $k * i$ modulo kd . Each de Bruijn routing step can be emulated with an expected constant number of messages, so routing uses $O(\log k n)$ expected hops- For $k = \Theta(\log n)$, we get $\Theta(\log n)$ degree and $\Theta(\log n / \log \log n)$ diameter.

```
function n.lookup(k, shift, i)
{
  if k  $\in$  (n, s) return (s);
  else if i  $\in$  (n, s) return
    (p.lookup(k, shift << 1, i  $\circ$  topBit(shift)));
  else return (s.lookup(k, shift, i));
}
```

The Koorde lookup algorithm at node **n**. **k** is the key. **i** is the imaginary De Bruijn node. **p** is the reference to the predecessor of **2n**. **s** is the reference to the successor of **n**.

Koorde lookup algorithm.

References

- "Internet Algorithms" by Greg Plaxton, Fall 2003: [1]
- "Koorde: A simple degree-optimal distributed hash table" by M. Frans Kaashoek and David R. Karger: [2]
- Chord and Koorde descriptions: [3]

References

- [1] <http://web.archive.org/web/20040929211835/http://www.cs.utexas.edu/users/plaxton/c395t/slides/DynamicTopologies-2.pdf>
- [2] <http://iptps03.cs.berkeley.edu/final-papers/koorde.ps>
- [3] http://www.cs.jhu.edu/~scheideler/courses/600.348_F03/lecture_10.pdf

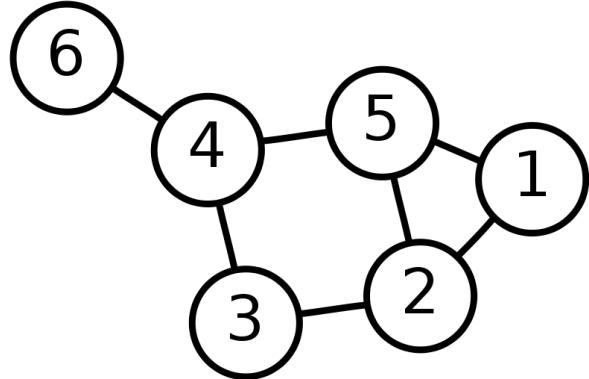
Graphs

Graph

In computer science, a **graph** is an abstract data structure that is meant to implement the graph concept from mathematics.

A graph data structure consists mainly of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (x,y) is said to **point** or **go from x to y** . The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).



A labeled graph of 6 vertices and 7 edges.

Operations

The basic operations provided by a graph data structure G usually include

- $\text{adjacent}(G, x, y)$: tests whether there is an edge from node x to node y .
- $\text{neighbors}(G, x)$: lists all nodes y such that there is an edge from x to y .
- $\text{add}(G, x, y)$: adds to G the edge from x to y , if it is not there.
- $\text{delete}(G, x, y)$: removes the edge from x to y , if it is there.
- $\text{get_node_value}(G, x)$: returns the value associated with the node x .
- $\text{set_node_value}(G, x, a)$: sets the value associated with the node x to a .

Structures that associate values to the edges usually provide also

- $\text{get_edge_value}(G, x, y)$: returns the value associated to the edge (x, y) .
- $\text{set_edge_value}(G, x, y, v)$: sets the value associated to the edge (x, y) to v .

Representations

Different data structures for the representation of graphs are used in practice, e.g.:

- **Adjacency list** - An adjacency list is implemented as an array of lists, with one list of destination nodes for each source node.
- **Incidence list** - A variant of the adjacency list that allows for the description of the edges at the cost of additional edges.
- **Adjacency matrix** - A two-dimensional Boolean matrix, in which the rows and columns represent source and destination vertices and entries in the matrix indicate whether an edge exists between the vertices associated with that row and column.
- **Incidence matrix** - A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The array entries indicate if both are related, i.e. incident.

Adjacency lists are preferred for sparse graphs; otherwise, an adjacency matrix is a good choice.

For graphs with some regularity in the placement of edges, a symbolic graph is a possible choice of representation.

Algorithms

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd-Warshall algorithm.

A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford-Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph.

External links

- Algraf Project: Graphical tool to draw graphs, apply several algorithms to them and export to XML ^[1]
- Boost Graph Library: a powerful C++ graph library ^[2]
- Graph Data Structures ^[3] (PDF, 280 KiB)
- Graphviz - Graph Visualization Software (Open Source) ^[4]
- Tulip - Graph/Data Visualization Software (LGPL) ^[5]
- Java Universal Network/Graph Framework (JUNG) ^[6]
- Java Graph and algorithm API (Annas) ^[7]
- NetworkX - a Python Graph package ^[8]
- Perl graph routines ^[9]
- Prefuse - Java framework for interactive data visualizations ^[10]
- QuickGraph: Graph Data Structures And Algorithms for .NET ^[11]

References

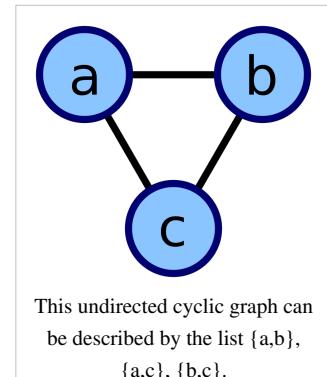
- [1] <http://algraf.es.kz>
- [2] <http://www.boost.org/libs/graph>
- [3] http://hamilton.bell.ac.uk/swdev2/notes/notes_18.pdf
- [4] <http://www.graphviz.org/>
- [5] <http://www.tulip-software.org/>
- [6] <http://jung.sourceforge.net/>
- [7] <http://annas.googlecode.com/>
- [8] <http://networkx.lanl.gov/>
- [9] <http://search.cpan.org/search?query=Graph&mode=all>
- [10] <http://prefuse.org/>
- [11] <http://www.codeplex.com/quickgraph>

Adjacency list

In graph theory, an **adjacency list** is the representation of all edges or arcs in a graph as a list.

If the graph is undirected, every entry is a set (or multiset) of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc.

Typically, adjacency lists are unordered.



Application in computer science

The graph pictured above has this adjacency list representation:		
a	adjacent to	b,c
b	adjacent to	a,c
c	adjacent to	a,b

In computer science, an adjacency list is a data structure for representing graphs. In an adjacency list representation, we keep, for each vertex in the graph, a list of all other vertices which it has an edge to (that vertex's "adjacency list"). For instance, the representation suggested by van Rossum, in which a hash table is used to associate each vertex with an array of adjacent vertices, can be seen as an example of this type of representation. Another example is the representation in Cormen et al. in which an array indexed by vertex numbers points to a singly-linked list of the neighbors of each vertex.

One difficulty with the adjacency list structure is that it has no obvious place to store data associated with the edges of a graph, such as the lengths or costs of the edges. To remedy this, some texts, such as that of Goodrich and Tamassia, advocate a more object oriented variant of the adjacency list structure, sometimes called an incidence list, which stores for each vertex a list of objects representing the edges incident to that vertex. To complete the structure, each edge must point back to the two vertices forming its endpoints. The extra edge objects in this version of the adjacency list cause it to use more memory than the version in which adjacent vertices are listed directly, but these extra edges are also a convenient location to store additional information about each edge (e.g. their length).

Trade-offs

The main alternative to the adjacency list is the adjacency matrix. For a graph with a sparse adjacency matrix an adjacency list representation of the graph occupies less space, because it does not use any space to represent edges that are *not* present. Using a naive array implementation of adjacency lists on a 32-bit computer, an adjacency list for an undirected graph requires about $8e$ bytes of storage, where e is the number of edges: each edge gives rise to entries in the two adjacency lists and uses four bytes in each.

On the other hand, because each entry in an adjacency matrix requires only one bit, they can be represented in a very compact way, occupying only $n^2/8$ bytes of contiguous space, where n is the number of vertices. Besides just avoiding wasted space, this compactness encourages locality of reference.

Noting that a graph can have at most n^2 edges (allowing loops) we can let $d = e/n^2$ denote the *density* of the graph. Then, if $8e > n^2/8$, the adjacency list representation occupies more space, which is true when $d > 1/64$. Thus a graph must be very sparse for an adjacency list representation to be more memory efficient than an adjacency matrix. However, this analysis is valid only when the representation is intended to store the connectivity structure of the graph without any numerical information about its edges.

Besides the space trade-off, the different data structures also facilitate different operations. It is easy to find all vertices adjacent to a given vertex in an adjacency list representation; you simply read its adjacency list. With an adjacency matrix you must instead scan over an entire row, taking $O(n)$ time. If you, instead, want to perform a neighbor test on two vertices (i.e., determine if they have an edge between them), an adjacency matrix provides this at once. However, this neighbor test in an adjacency list requires time proportional to the number of edges associated with the two vertices.

References

- Joe Celko (2004). *Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann. excerpt from Chapter 2: "Adjacency List Model" [1]. ISBN 1-55860-920-2.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2001). *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill. pp. 527–529 of section 22.1: Representations of graphs. ISBN 0-262-03293-7.
- David Eppstein (1996). "ICS 161 Lecture Notes: Graph Algorithms" [2].
- Michael T. Goodrich and Roberto Tamassia (2002). *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons. ISBN 0-471-38365-1.
- Guido van Rossum (1998). "Python Patterns — Implementing Graphs" [3].

References

- [1] <http://www.SQLSummit.com/AdjacencyList.htm>
- [2] <http://www.ics.uci.edu/~eppstein/161/960201.html>
- [3] <http://www.python.org/doc/essays/graphs/>

Adjacency matrix

In mathematics and computer science, an **adjacency matrix** is a means of representing which vertices of a graph are adjacent to which other vertices. Another matrix representation for a graph is the incidence matrix.

Specifically, the adjacency matrix of a finite graph G on n vertices is the $n \times n$ matrix where the nondiagonal entry a_{ij} is the number of edges from vertex i to vertex j , and the diagonal entry a_{ii} , depending on the convention, is either once or twice the number of edges (loops) from vertex i to itself. Undirected graphs often use the former convention of counting loops twice, whereas directed graphs typically use the latter convention. There exists a unique adjacency matrix for each graph (up to permuting rows and columns), and it is not the adjacency matrix of any other graph. In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric.

The relationship between a graph and the eigenvalues and eigenvectors of its adjacency matrix is studied in spectral graph theory.

Examples

- Here is an example of a labeled graph and its adjacency matrix. The convention followed here is that an adjacent edge counts 1 in the matrix for an undirected graph. (X,Y coordinates are 1-6)

Labeled graph	Adjacency matrix
	$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$

- The adjacency matrix of a complete graph is all 1's except for 0's on the diagonal.
- The adjacency matrix of an empty graph is a zero matrix.

Adjacency matrix of a bipartite graph

The adjacency matrix A of a bipartite graph whose parts have r and s vertices has the form

$$A = \begin{pmatrix} O & B \\ B^T & O \end{pmatrix},$$

where B is an $r \times s$ matrix and O is an all-zero matrix. Clearly, the matrix B uniquely represents the bipartite graphs, and it is commonly called its biadjacency matrix.

Formally, let $G = (U, V, E)$ be a bipartite graph or bigraph with parts $U = u_1, \dots, u_r$ and $V = v_1, \dots, v_s$. An $r \times s$ 0-1 matrix B is called the **biadjacency matrix** if $B_{i,j} = 1$ iff $(u_i, v_j) \in E$.

If G is a bipartite multigraph or weighted graph then the elements $B_{i,j}$ are taken to be the number of edges between or the weight of (u_i, v_j) respectively.

Properties

The adjacency matrix of an undirected simple graph is symmetric, and therefore has a complete set of real eigenvalues and an orthogonal eigenvector basis. The set of eigenvalues of a graph is the **spectrum** of the graph.

Suppose two directed or undirected graphs G_1 and G_2 with adjacency matrices A_1 and A_2 are given. G_1 and G_2 are isomorphic if and only if there exists a permutation matrix P such that

$$PA_1P^{-1} = A_2.$$

In particular, A_1 and A_2 are similar and therefore have the same minimal polynomial, characteristic polynomial, eigenvalues, determinant and trace. These can therefore serve as isomorphism invariants of graphs. However, two graphs may possess the same set of eigenvalues but not be isomorphic – one cannot 'hear' (reconstruct, or 'inverse-scatter') the shape of a graph.

If A is the adjacency matrix of the directed or undirected graph G , then the matrix A^n (i.e., the matrix product of n copies of A) has an interesting interpretation: the entry in row i and column j gives the number of (directed or undirected) walks of length n from vertex i to vertex j . This implies, for example, that the number of triangles in an undirected graph G is exactly the trace of A^3 divided by 6.

The main diagonal of every adjacency matrix corresponding to a graph without loops has all zero entries.

For (d) -regular graphs, d is also an eigenvalue of A , for the vector $v = (1, \dots, 1)$, and G is connected if and only if the multiplicity of d is 1. It can be shown that $-d$ is also an eigenvalue of A if G is connected bipartite graph. The above are results of Perron–Frobenius theorem.

Variations

The Seidel adjacency matrix or **(0,-1,1)-adjacency matrix** of a simple graph has zero on the diagonal and entry $a_{ij} = -1$ if ij is an edge and $+1$ if it is not. This matrix is used in studying strongly regular graphs and two-graphs.

A distance matrix is like a higher-level adjacency matrix. Instead of only providing information about whether or not two vertices are connected, also tells the distances between them. This assumes the length of every edge is 1. A variation is where the length of an edge is not necessarily 1.

Data structures

When used as a data structure, the main alternative for the adjacency matrix is the adjacency list. Because each entry in the adjacency matrix requires only one bit, they can be represented in a very compact way, occupying only $n^2/8$ bytes of contiguous space, where n is the number of vertices. Besides just avoiding wasted space, this compactness encourages locality of reference.

On the other hand, for a sparse graph, adjacency lists win out, because they do not use any space to represent edges which are *not* present. Using a naïve array implementation on a 32-bit computer, an adjacency list for an undirected graph requires about $8e$ bytes of storage, where e is the number of edges.

Noting that a simple graph can have at most n^2 edges, allowing loops, we can let $d = e/n^2$ denote the *density* of the graph. Then, $8e > n^2/8$, or the adjacency list representation occupies more space, precisely when $d > 1/64$.

Thus a graph must be sparse indeed to justify an adjacency list representation.

Besides the space tradeoff, the different data structures also facilitate different operations. Finding all vertices adjacent to a given vertex in an adjacency list is as simple as reading the list. With an adjacency matrix, an entire row must instead be scanned, which takes $O(n)$ time. Whether there is an edge between two given vertices can be determined at once with an adjacency matrix, while requiring time proportional to the minimum degree of the two vertices with the adjacency list.

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2001), *Introduction to Algorithms*, second edition. MIT Press and McGraw-Hill. ISBN 0-262-03293-7. Section 22.1: Representations of graphs, pp. 527–531.
- Chris Godsil and Gordon Royle (2001), *Algebraic Graph Theory*. New York: Springer-Verlag. ISBN 0-387-95241-1

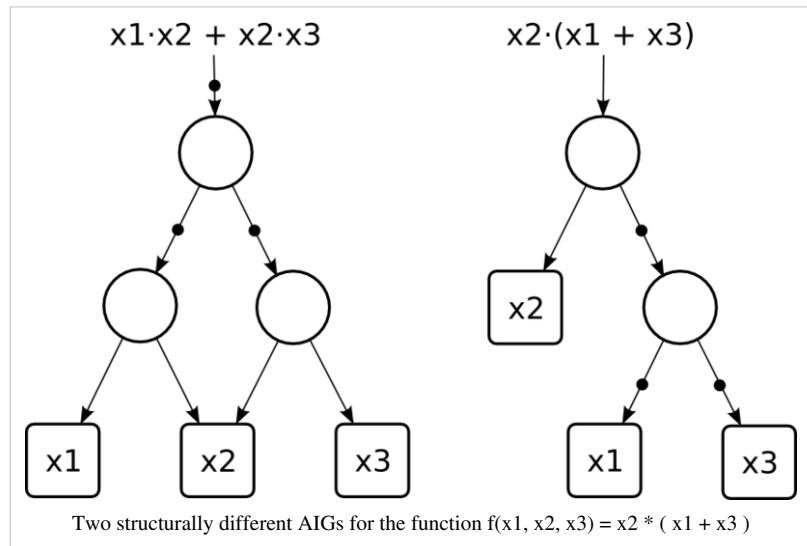
External links

- Fluffschack (<http://www.x2d.org/java/projects/fluffschack.jnlp>) — an educational Java web start game demonstrating the relationship between adjacency matrices and graphs.

And-inverter graph

An **and-inverter graph (AIG)** is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit or network. An AIG consists of two-input nodes representing logical conjunction, terminal nodes labeled with variable names, and edges optionally containing markers indicating logical negation. This representation of a logic function is rarely structurally efficient for large circuits, but is an efficient representation for manipulation of boolean functions. Typically, the abstract graph is represented as a data structure in software.

Conversion from the network of logic gates to AIGs is fast and scalable. It only requires that every gate be expressed in terms of AND gates and inverters. This conversion does not lead to unpredictable increase in memory use and runtime. This makes the AIG an efficient representation in comparison with either the binary decision diagram (BDD) or the "sum-of-products" ($\Sigma\Omega\Pi$) form, that is, the canonical form in Boolean algebra known as the disjunctive normal form (DNF). The BDD and DNF may also be viewed as circuits, but they involve formal constraints that deprive them of scalability. For example, $\Sigma\Omega\Pi$ s are circuits with at most two levels while BDDs are canonical, that is, they require that input variables be evaluated in the same order on all paths.



Circuits composed of simple gates, including AIGs, are an "ancient" research topic. The interest in AIGs started in the late 1950s^[1] and continued in the 1970s when various local transformations have been developed. These transformations were implemented in several logic synthesis and verification systems, such as Darringer et al.^[2] and Smith et al.^[3], which reduce circuits to improve area and delay during synthesis, or to speed up formal equivalence checking. Several important techniques were discovered early at IBM, such as combining and reusing multi-input logic expressions and subexpressions, now known as structural hashing.

Recently there has been a renewed interest in AIGs as a functional representation for a variety of tasks in synthesis and verification. That is because representations popular in the 1990s (such as BDDs) have reached their limits of scalability in many of their applications. Another important development was the recent emergence of much more

efficient boolean satisfiability (SAT) solvers. When coupled with *AIGs* as the circuit representation, they lead to remarkable speedups in solving a wide variety of boolean problems.

AIGs found successful use in diverse EDA applications. A well-tuned combination of *AIGs* and boolean satisfiability made an impact on formal verification, including both model checking and equivalence checking.^[4] Another recent work shows that efficient circuit compression techniques can be developed using *AIGs*.^[5] There is a growing understanding that logic and physical synthesis problems can be solved using *AIGs* simulation and boolean satisfiability compute functional properties (such as symmetries^[6]) and node flexibilities (such as don't-cares, resubstitutions, and SPFDS^[7]). This work shows that *AIGs* are a promising *unifying* representation, which can bridge logic synthesis, technology mapping, physical synthesis, and formal verification. This is, to a large extent, due to the simple and uniform structure of *AIGs*, which allow rewriting, simulation, mapping, placement, and verification to share the same data structure.

In addition to combinational logic, *AIGs* have also been applied to sequential logic and sequential transformations. Specifically, the method of structural hashing was extended to work for *AIGs* with memory elements (such as D-type flip-flops with an initial state, which, in general, can be unknown) resulting in a data structure that is specifically tailored for applications related to retiming.^[8]

Ongoing research includes implementing a modern logic synthesis system completely based on *AIGs*. The prototype called ABC^[9] features an *AIG* package, several *AIG*-based synthesis and equivalence-checking techniques, as well as an experimental implementation of sequential synthesis. One such technique combines technology mapping and retiming in a single optimization step. It should be noted that these optimizations can be implemented using networks composed of arbitrary gates, but the use of *AIGs* makes them more scalable and easier to implement.

Implementations

- Logic Synthesis and Verification System ABC^[9]
- A set of utilities for *AIGs* AIGER^[10]
- OpenAccess Gear^[11]

See also

- Binary decision diagram
- Logical conjunction

This article is adapted from a column in the ACM SIGDA ^[12] *e-newsletter* ^[13] *by Alan Mishchenko* ^[14]
Original text is available here^[15].

References

- [1] L. Hellerman (June 1963). "A catalog of three-variable Or-Inverter and And-Inverter logical circuits". *IEEE Trans. Electron. Comput.* **EC-12**: 198–223. doi:10.1109/PGEC.1963.263531.
- [2] A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan (1981). "Logic synthesis through local transformations". *IBM J. Of Research and Development* **25** (4): 272–280.
- [3] G. L. Smith, R. J. Bahnsen, H. Halliwell (1982). "Boolean comparison of hardware and flowcharts". *IBM J. Of Research and Development* **26** (1): 106–116.
- [4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai (2002). "Robust boolean reasoning for equivalence checking and functional property verification". *IEEE Trans. CAD* **21** (12): 1377–1394.
- [5] P. Bjesse and A. Boralv. "DAG-aware circuit compression for formal verification". *Proc. ICCAD '04*. pp. 42–49.
- [6] K.-H. Chang, I. L. Markov, V. Bertacco. "Post-placement rewiring and rebuffering by exhaustive search for functional symmetries". *Proc. ICCAD '05* pages=56–63.
- [7] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske (May 2006). "Using simulation and satisfiability to compute flexibilities in Boolean networks". *IEEE Trans. CAD* **25** (5): 743–755..
- [8] J. Baumgartner and A. Kuehlmann. "Min-area retiming on flexible circuit structures". *Proc. ICCAD'01*. pp. 176–182.

- [9] <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [10] <http://fmv.jku.at/aiger/index.html>
- [11] <http://opendatools.si2.org/oagear/>
- [12] <http://www.sigda.org>
- [13] <http://www.sigda.org/newsletter/index.html>
- [14] <http://www.eecs.berkeley.edu/~alanmi/>
- [15] http://www.sigda.org/newsletter/2006/eNews_061215.html

Binary decision diagram

In the field of computer science, a **binary decision diagram (BDD)** or **branching program**, like a negation normal form (NNF) or a propositional directed acyclic graph (PDAG), is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression.

Definition

A Boolean function can be represented as a rooted, directed, acyclic graph, which consists of decision nodes and two terminal nodes called 0-terminal and 1-terminal. Each decision node is labeled by a Boolean variable and has two child nodes called low child and high child. The edge from a node to a low (high) child represents an assignment of the variable to 0 (1). Such a **BDD** is called 'ordered' if different variables appear in the same order on all paths from the root. A BDD is said to be 'reduced' if the following two rules have been applied to its graph:

- Merge any isomorphic subgraphs.
- Eliminate any node whose two children are isomorphic.

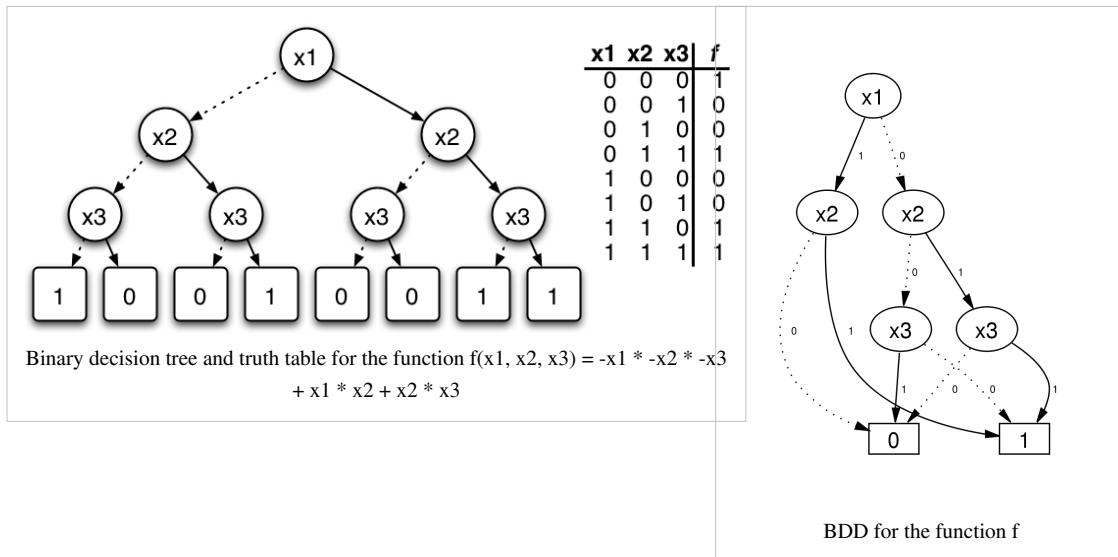
In popular usage, the term **BDD** almost always refers to **Reduced Ordered Binary Decision Diagram (ROBDD)** in the literature, used when the ordering and reduction aspects need to be emphasized. The advantage of an ROBDD is that it is canonical (unique) for a particular function and variable order.^[1] This property makes it useful in functional equivalence checking and other operations like functional technology mapping.

A path from the root node to the 1-terminal represents a (possibly partial) variable assignment for which the represented Boolean function is true. As the path descends to a low child (high child) from a node, then that node's variable is assigned to 0 (1).

Example

The left figure below shows a binary decision *tree* (the reduction rules are not applied), and a truth table, each representing the function $f(x_1, x_2, x_3)$. In the tree on the left, the value of the function can be determined for a given variable assignment by following a path down the graph to a terminal. In the figures below, dotted lines represent edges to a low child, while solid lines represent edges to a high child. Therefore, to find $(x_1=0, x_2=1, x_3=1)$, begin at x_1 , traverse down the dotted line to x_2 (since x_1 has an assignment to 0), then down two solid lines (since x_2 and x_3 each have an assignment to one). This leads to the terminal 1, which is the value of $f(x_1=0, x_2=1, x_3=1)$.

The binary decision *tree* of the left figure can be transformed into a binary decision *diagram* by maximally reducing it according to the two reduction rules. The resulting **BDD** is shown in the right figure.



History

The basic idea from which the data structure was created is the Shannon expansion. A switching function is split into two sub-functions (cofactors) by assigning one variable (cf. *if-then-else normal form*). If such a sub-function is considered as a sub-tree, it can be represented by a *binary decision tree*. Binary decision diagrams (BDD) were introduced by Lee^[2], and further studied and made known by Akers^[3] and Boute^[4].

The full potential for efficient algorithms based on the data structure was investigated by Randal Bryant at Carnegie Mellon University: his key extensions were to use a fixed variable ordering (for canonical representation) and shared sub-graphs (for compression). Applying these two concepts results in an efficient data structure and algorithms for the representation of sets and relations^{[5] [6]}. By extending the sharing to several BDDs, i.e. one sub-graph is used by several BDDs, the data structure *Shared Reduced Ordered Binary Decision Diagram* is defined^[7]. The notion of a BDD is now generally used to refer to that particular data structure.

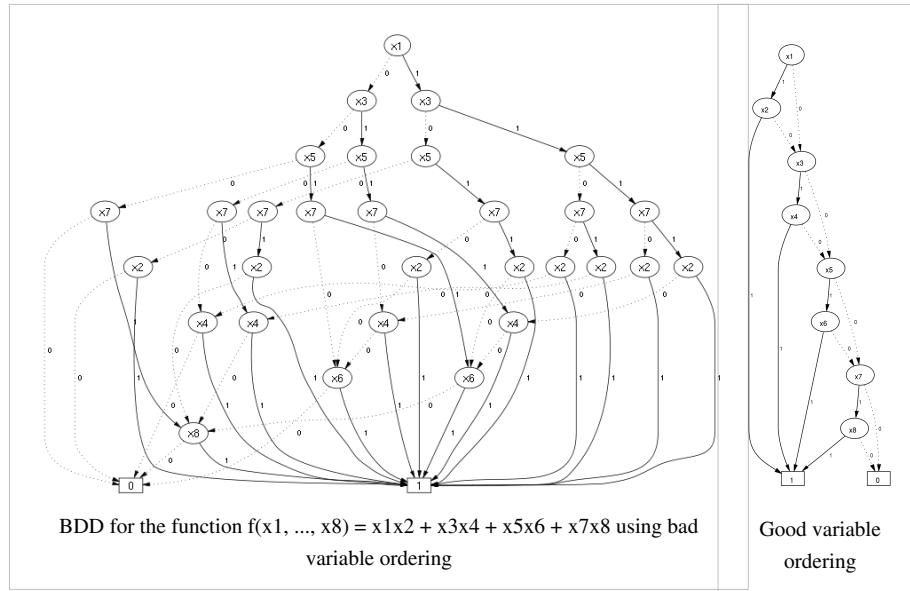
Applications

BDDs are extensively used in CAD software to synthesize circuits (logic synthesis) and in formal verification. There are several lesser known applications of BDD, including Fault tree analysis, Bayesian Reasoning and Product Configuration.

Every arbitrary BDD (even if it is not reduced or ordered) can be directly implemented by replacing each node with a 2 to 1 multiplexer; each multiplexer can be directly implemented by a 4-LUT in a FPGA. It is not so simple to convert from an arbitrary network of logic gates to a BDD (unlike the and-inverter graph).

Variable ordering

The size of the BDD is determined both by the function being represented and the chosen ordering of the variables. For a boolean function $f(x_1, \dots, x_n)$ then depending upon the ordering of the variables we would end up getting a graph whose number of nodes would be linear (in n) at the best and exponential at the worst case. Let us consider the Boolean function $f(x_1, \dots, x_{2n}) = x_1x_2 + x_3x_4 + \dots + x_{2n-1}x_{2n}$. Using the variable ordering $x_1 < x_3 < \dots < x_{2n-1} < x_2 < x_4 < \dots < x_{2n}$, the BDD needs 2^{n+1} nodes to represent the function. Using the ordering $x_1 < x_2 < x_3 < x_4 < \dots < x_{2n-1} < x_{2n}$, the BDD consists of $2n$ nodes.



It is of crucial importance to care about variable ordering when applying this data structure in practice. The problem of finding the best variable ordering is NP-hard.^[8] For any constant $c > 1$ it is even NP-hard to compute a variable ordering resulting in an OBDD with a size that is at most c times larger than an optimal one.^[9] However there exist efficient heuristics to tackle the problem.

There are functions for which the graph size is always exponential — independent of variable ordering. This holds e.g. for the multiplication function (an indication as to the apparent complexity of factorization).

Researchers have of late suggested refinements on the BDD data structure giving way to a number of related graphs: BMD (Binary Moment Diagrams), ZDD (Zero Suppressed Decision Diagram), FDD (Free Binary Decision Diagrams), PDD (Parity decision Diagrams), MTBDDs (Multiple terminal BDDs), etc.

Logical operations on BDDs

Many logical operations on BDDs can be implemented by polynomial-time graph manipulation algorithms.

- conjunction
- disjunction
- negation
- existential abstraction
- universal abstraction

However, repeating these operations several times, for example forming the conjunction or disjunction of a set of BDDs, may in the worst case result in an exponentially big BDD. This is because any of the preceding operations for two BDDs may result in a BDD with a size proportional to the product of the BDDs' sizes, and consequently for several BDDs the size may be exponential.

See also

- Boolean satisfiability problem
- Data structure
- Model checking
- Negational normal form (NNF)
- Propositional directed acyclic graph (PDAG)
- Radix tree
- Binary key - a method of species identification in biology using binary trees

References

- [1] Graph-Based Algorithms for Boolean Function Manipulation, Randal E. Bryant, 1986
- [2] C. Y. Lee. "Representation of Switching Circuits by Binary-Decision Programs". Bell Systems Technical Journal, 38:985–999, 1959.
- [3] Sheldon B. Akers. Binary Decision Diagrams (<http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1675141>), IEEE Transactions on Computers, C-27(6):509–516, June 1978.
- [4] Raymond T. Boute, "The Binary Decision Machine as a programmable controller". EUROMICRO Newsletter, Vol. 1(2):16–22, January 1976.
- [5] Randal E. Bryant. " Graph-Based Algorithms for Boolean Function Manipulation (<http://www.cs.cmu.edu/~bryant/pubdir/ieetc86.ps>)". IEEE Transactions on Computers, C-35(8):677–691, 1986.
- [6] R. E. Bryant, " Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams" (<http://www.cs.cmu.edu/~bryant/pubdir/acmcs92.ps>), ACM Computing Surveys, Vol. 24, No. 3 (September, 1992), pp. 293–318.
- [7] Karl S. Brace, Richard L. Rudell and Randal E. Bryant. " Efficient Implementation of a BDD Package" (<http://portal.acm.org/citation.cfm?id=123222&coll=portal&dl=ACM>). In Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC 1990), pages 40–45. IEEE Computer Society Press, 1990.
- [8] Beate Bollig, Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete (doi:10.1109/12.537122), IEEE Transactions on Computers, 45(9):993—1002, September 1996.
- [9] Detlef Sieling. "The nonapproximability of OBDD minimization." Information and Computation 172, 103–138. 2002.
- R. Ubar, "Test Generation for Digital Circuits Using Alternative Graphs (in Russian)", in Proc. Tallinn Technical University, 1976, No.409, Tallinn Technical University, Tallinn, Estonia, pp. 75–81.

Further Reading

- D. E. Knuth, "The Art of Computer Programming Volume 4, Fascicle 1: Bitwise tricks & techniques; Binary Decision Diagrams" (Addison-Wesley Professional, March 27, 2009) viii+260pp, ISBN 0-321-58050-8. Draft of Fascicle 1b (<http://www-cs-faculty.stanford.edu/~knuth/fasc1b.ps.gz>) available for download.
- H. R. Andersen " An Introduction to Binary Decision Diagrams (<http://www.configit.com/fileadmin/Configit/Documents/bdd-eap.pdf>)," Lecture Notes, 1999, IT University of Copenhagen.
- Ch. Meinel, T. Theobald, " Algorithms and Data Structures in VLSI-Design: OBDD - Foundations and Applications" (http://www.hpi.uni-potsdam.de/fileadmin/hpi/FG_ITS/books/OBDD-Book.pdf), Springer-Verlag, Berlin, Heidelberg, New York, 1998. Complete textbook available for download.

External links

Available OBDD Packages

- ABCD (<http://fmv.jku.at/abcd/>): The ABCD package by Armin Biere, Johannes Kepler Universität, Linz.
- BuDDy (<http://sourceforge.net/projects/buddy/>): A BDD package by Jørn Lind-Nielsen
- CMU BDD (<http://www-2.cs.cmu.edu/~modelcheck/bdd.html>), BDD package, Carnegie Mellon University, Pittsburgh
- CrocoPat (<http://mtc.epfl.ch/~beyer/CrocoPat/>), BDD package and a high-level querying language, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
- CUDD (<http://vlsi.colorado.edu/~fabio/CUDD/>): BDD package, University of Colorado, Boulder
- Biddy (<http://lms.uni-mb.si/biddy/>): multi-platform academic BDD package, University of Maribor, Slovenia

- JavaBDD (<http://javabdd.sourceforge.net>), a Java port of BuDDy that also interfaces to CUDD, CAL, and JDD
- The Berkeley CAL (http://embedded.eecs.berkeley.edu/Research/cal_bdd/) package which does breadth-first manipulation
- TUD BDD (<http://www.rs.tu-darmstadt.de/Stefan-Hoereth.90.0.html>): A BDD package and a world-level package by Stefan Höreth
- Vahidi's JDD (<http://javaddlib.sourceforge.net/jdd/>), a java library that supports common BDD and ZBDD operations
- Vahidi's JBDD (<http://javaddlib.sourceforge.net/jbdd/>), a Java interface to BuDDy and CUDD packages
- Maiki & Boaz BDD-PROJECT (<http://www.bdd-project.com>), a web application for BDD reduction and visualization
- A. Costa BFunc (<http://www.dei.isep.ipp.pt/~acc/bfunc/>), includes a BDD boolean logic simplifier supporting up to 32 inputs / 32 outputs (independently)
- DDD (<http://ddd.lip6.fr>): A C++ library with support for integer valued and hierarchical decision diagrams.
- JINC (<http://www.jossowski.de/projects/jinc/jinc.html>): A C++ library developed at University of Bonn, Germany, supporting several BDD variants and multi-threading.
- OBDD (<http://hackage.haskell.org/cgi-bin/hackage-scripts/package/obdd>): A Haskell package for OBDD
- Configit Product Modeler (<http://www.configit.com/>): A BDD-based tool for product configuration developed by Configit, Copenhagen.

Binary moment diagram

A **binary moment diagram** (BMD) is a generalization of the binary decision diagram (BDD) to linear functions over domains such as booleans (like BDDs), but also to integers or to real numbers.

They can deal with boolean functions with complexity comparable to BDDs, but also some functions that are dealt with very inefficiently in a BDD are handled easily by BMD, most notably multiplication.

The most important properties of BMD is that, like with BDDs, each function has exactly one canonical representation, and many operations can be efficiently performed on these representations.

The main features that differentiate BMDs from BDDs are using linear instead of pointwise diagrams, and having weighted edges.

The rules that ensure the canonicity of the representation are:

- Decision over variables higher in the ordering may only point to decisions over variables lower in the ordering.
- No two nodes may be identical (in normalization such nodes all references to one of these nodes should be replaced by references to another)
- No node may have all decision parts equivalent to 0 (links to such nodes should be replaced by links to their always part)
- No edge may have weight zero (all such edges should be replaced by direct links to 0)
- Weights of the edges should be coprime. Without this rule or some equivalent of it, it would be possible for a function to have many representations, for example $2x + 2$ could be represented as $2 \cdot (1 + x)$ or $1 \cdot (2 + 2x)$.

Pointwise and linear decomposition

In pointwise decomposition, like in BDDs, on each branch point we store result of all branches separately. An example of such decomposition for an integer function ($2x + y$) is:

$$\begin{cases} \text{if } x \begin{cases} \text{if } y, 3 \\ \text{if } \neg y, 2 \end{cases} \\ \text{if } \neg x \begin{cases} \text{if } y, 1 \\ \text{if } \neg y, 0 \end{cases} \end{cases}$$

In linear decomposition we provide instead a default value and a difference:

$$\begin{cases} \text{always } \begin{cases} \text{always } 0 \\ \text{if } y, +1 \\ \text{if } x, +2 \end{cases} \end{cases}$$

It can easily be seen that the latter (linear) representation is much more efficient in case of additive functions, as when we add many elements the latter representation will have only $O(n)$ elements, while the former (pointwise), even with sharing, exponentially many.

Edge weights

Another extension is using weights for edges. A value of function at given node is a sum of the true nodes below it (the node under always, and possibly the decided node) times the edges' weights.

For example $(4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0)$ can be represented as:

1. Result node, always $1 \times$ value of node 2, if x_2 add $4 \times$ value of node 4
2. Always $1 \times$ value of node 3, if x_1 add $2 \times$ value of node 4
3. Always 0, if x_0 add $1 \times$ value of node 4
4. Always $1 \times$ value of node 5, if y_2 add +4
5. Always $1 \times$ value of node 6, if y_1 add +2
6. Always 0, if y_0 add +1

Without weighted nodes a much more complex representation would be required:

1. Result node, always value of node 2, if x_2 value of node 4
2. Always value of node 3, if x_1 value of node 7
3. Always 0, if x_0 value of node 10
4. Always value of node 5, if y_2 add +16
5. Always value of node 6, if y_1 add +8
6. Always 0, if y_0 add +4
7. Always value of node 8, if y_2 add +8
8. Always value of node 9, if y_1 add +4
9. Always 0, if y_0 add +2
10. Always value of node 11, if y_2 add +4
11. Always value of node 12, if y_1 add +2
12. Always 0, if y_0 add +1

Zero suppressed decision diagram

A **zero suppressed decision diagram** (ZSDD or ZDD) is a version of binary decision diagram (BDD) where instead of nodes being introduced when the positive and the negative part are different, they are introduced when negative part is different from constant 0. Zero suppressed decisions diagrams are also commonly referred to as **zero suppressed binary decision diagram** (ZBDD).

They are useful when dealing with functions which are almost everywhere 0.

Available packages

- CUDD^[1]: A BDD package written in C that implements BDDs and ZBDDs, University of Colorado, Boulder
- JDD^[2], A java library that implements common BDD and ZBDD operations

References

[1] <http://vlsi.colorado.edu/~fabio/CUDD/>

[2] <http://javaddlib.sourceforge.net/jdd/>

- Shin-ichi Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1600231)", DAC '93: Proceedings of the 30th international conference on Design automation, 1993
- Ch. Meinel, T. Theobald, "Algorithms and Data Structures in VLSI-Design: OBDD - Foundations and Applications" (http://www.hpi.uni-potsdam.de/fileadmin/hpi/FG_ITS/books/OBDD-Book.pdf), Springer-Verlag, Berlin, Heidelberg, New York, 1998.

External links

- Alan Mishchenko, [http://www.eecs.berkeley.edu/~alanmi/publications/2001/tech01_zdd.pdf An Introduction to Zero-Suppressed Binary Decision Diagrams"]

Propositional directed acyclic graph

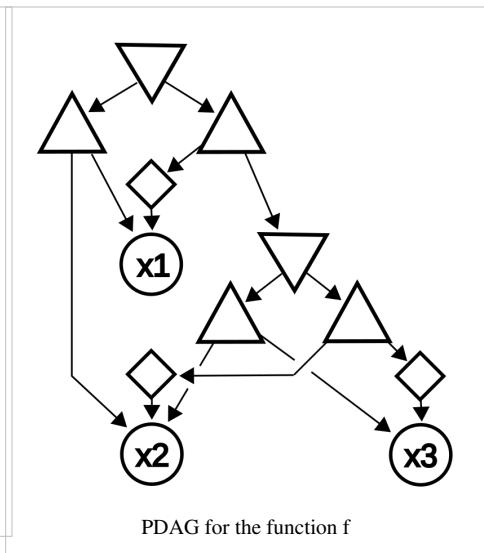
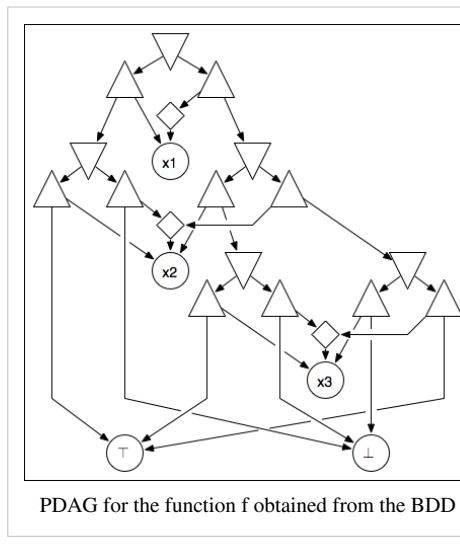
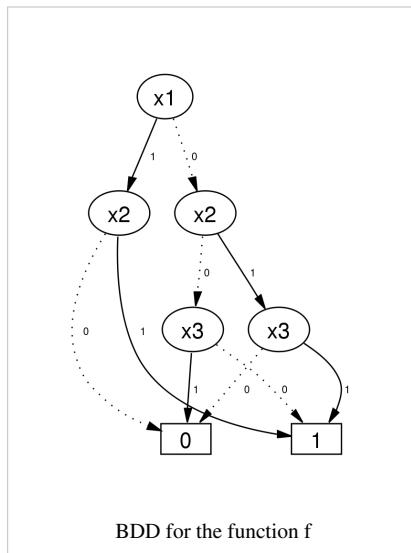
A **propositional directed acyclic graph (PDAG)** is a data structure that is used to represent a Boolean function. A Boolean function can be represented as a rooted, directed acyclic graph of the following form:

- Leaves are labeled with \top (true), \perp (false), or a Boolean variable.
- Non-leaves are \triangle (logical and), ∇ (logical or) and \diamond (logical not).
- \triangle - and ∇ -nodes have at least one child.
- \diamond -nodes have exactly one child.

Leaves labeled with \top (\perp) represent the constant Boolean function which always evaluates to 1 (0). A leaf labeled with a Boolean variable x is interpreted as the assignment $x = 1$, i.e. it represents the Boolean function which evaluates to 1 if and only if $x = 1$. The Boolean function represented by a \triangle -node is the one that evaluates to 1, if and only if the Boolean function of all its children evaluate to 1. Similarly, a ∇ -node represents the Boolean function that evaluates to 1, if and only if the Boolean function of at least one child evaluates to 1. Finally, a \diamond -node represents the complementary Boolean function its child, i.e. the one that evaluates to 1, if and only if the Boolean function of its child evaluates to 0.

PDAG, BDD, and NNF

Every **binary decision diagram (BDD)** and every **negation normal form (NNF)** is also a PDAG with some particular properties. The following pictures represent the Boolean function $f(x_1, x_2, x_3) = -x_1 * -x_2 * -x_3 + x_1 * x_2 + x_2 * x_3$:



See also

- Data structure
- Boolean satisfiability problem
- Proposition

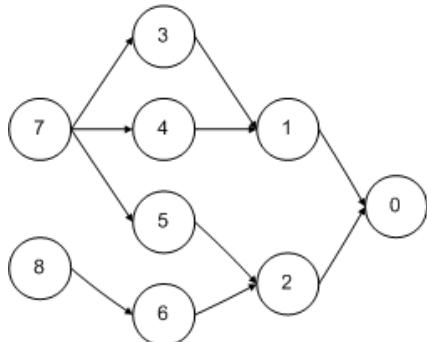
References

- M. Wachter & R. Haenni, "Propositional DAGs: a New Graph-Based Language for Representing Boolean Functions", KR'06, 10th International Conference on Principles of Knowledge Representation and Reasoning, Lake District, UK, 2006.
- M. Wachter & R. Haenni, "Probabilistic Equivalence Checking with Propositional DAGs", Technical Report iam-2006-001, Institute of Computer Science and Applied Mathematics, University of Bern, Switzerland, 2006.
- M. Wachter, R. Haenni & J. Jonczy, "Reliability and Diagnostics of Modular Systems: a New Probabilistic Approach", DX'06, 18th International Workshop on Principles of Diagnosis, Peñaranda de Duero, Burgos, Spain, 2006.

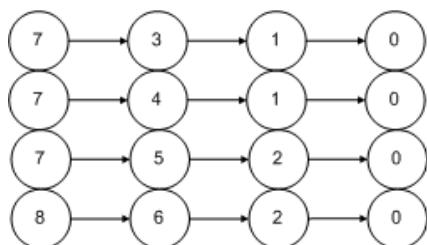
Graph-structured stack

In computer science, a **graph-structured stack** is a directed acyclic graph where each directed path represents a stack. The graph-structured stack is an essential part of Tomita's algorithm, where it replaces the usual stack of a pushdown automaton. This allows the algorithm to encode the nondeterministic choices in parsing an ambiguous grammar, sometimes with greater efficiency.

In the following diagram, there are four stacks: {7,3,1,0}, {7,4,1,0}, {7,5,2,0}, and {8,6,2,0}.



Another way to simulate nondeterminism would be to duplicate the stack as needed. The duplication would be less efficient since vertices would not be shared. For this example, 16 vertices would be needed instead of 9.



Scene graph

A **scene graph** is a general data structure commonly used by vector-based graphics editing applications and modern computer games. Examples of such programs include AutoCAD, Adobe Illustrator, Acrobat 3D, X3D, VRML97, OpenSceneGraph, OpenSG, and CorelDRAW.

The scene graph is a structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene. The definition of a scene graph is fuzzy because programmers who implement scene graphs in applications and in particular the games industry take the basic principles and adapt these to suit particular applications. This means there is no consensus as to what a scene graph should be.

A scene graph is a collection of nodes in a graph or tree structure. A node may have many children but often only a single parent, with the effect of a parent applied to all its child nodes; an operation performed on a group automatically propagates its effect to all of its members. In many programs, associating a geometrical transformation matrix (see also transformation and matrix) at each group level and concatenating such matrices together is an efficient and natural way to process such operations. A common feature, for instance, is the ability to group related shapes/objects into a compound object which can then be moved, transformed, selected, etc. as easily as a single object.

It also happens that in some scene graphs, a node can have a relation to any node including itself, or at least an extension that refers to another node (for instance Pixar's PhotoRealistic RenderMan because of its usage of Reyes rendering algorithm, or Adobe Systems's Acrobat 3D for advanced interactive manipulation).

Scene graphs in graphics editing tools

In vector-based graphics editing, each node in a scene graph represents some atomic unit of the document, usually a shape such as an ellipse or Bezier path. Although shapes themselves (particularly paths) can be decomposed further into nodes such as spline nodes, it is practical to think of the scene graph as composed of shapes rather than going to a lower level of representation.

Another useful and user-driven node concept is the layer. A layer acts like a transparent sheet upon which any number of shapes and shape groups can be placed. The document then becomes a set of layers, any of which can be conveniently made invisible, dimmed, and/or locked (made read-only). Some applications place all layers in a linear list while others support sublayers (i.e., layers within layers, to any desired depth).

Internally, there may be no real structural difference between layers and groups at all, since they are both just nested scene graphs. If differences are needed, a common type declaration in C++ would be to make a generic scene graph class, and then derive layers and groups as subclasses. A visibility member, for example, would be a feature of a layer but not necessarily of a group.

Scene graphs in games and 3D applications

Scene graphs are useful for modern games using 3D graphics and increasingly large worlds or levels. In such applications, nodes in a scene graph (generally) represent entities or objects in the scene.

For instance, a game might define a logical relationship between a knight and a horse so that the knight is considered an extension to the horse. The scene graph would have a 'horse' node with a 'knight' node attached to it.

As well as describing the logical relationship, the scene graph may also describe the spatial relationship of the various entities: the knight moves through 3D space as the horse moves.

In these large applications, memory requirements are major considerations when designing a scene graph. For this reason many large scene graph systems use instancing to reduce memory costs and increase speed. In our example above, each knight is a separate scene node, but the graphical representation of the knight (made up of a 3D mesh, textures, materials and shaders) is instanced. This means that only a single copy of the data is kept, which is then

referenced by any 'knight' nodes in the scene graph. This allows a reduced memory budget and increased speed, since when a new knight node is created, the appearance data does not need to be duplicated.

Scene graph implementation

The simplest form of scene graph uses an array or linked list data structure, and displaying its shapes is simply a matter of linearly iterating the nodes one by one. Other common operations, such as checking to see which shape intersects the mouse pointer (e.g., in a GUI-based applications) are also done via linear searches. For small scene graphs, this tends to suffice.

Larger scene graphs cause linear operations to become noticeably slow and thus more complex underlying data structures are used, the most popular being a tree. This is the most common form of scene graph. In these scene graphs the composite design pattern is often employed to create the hierarchical representation of group-nodes and leaf-nodes.

Group Nodes - Can have any number of child nodes attached to it. Group nodes include transformations and switch nodes.

Leaf Nodes - Are nodes that are actually rendered or see the effect of an operation. These include objects, sprites, sounds, lights and anything that could be considered 'rendered' in some abstract sense.

Scene graph operations and dispatch

Applying an operation on a scene graph requires some way of dispatching an operation based on a node's type. For example, in a render operation, a transformation group-node would accumulate its transformation (generally this is matrix multiplication but could involve operations with vector displacement and quaternions or Euler angles instead), while an object leaf-node would send the object off for rendering to the renderer (some implementations might render the object directly but this can integrate the underlying rendering API - e.g. OpenGL or DirectX too tightly and rigidly - it is better to separate the scene graph and renderer systems as this promotes portability). In order to accomplish this type of dispatching, several different approaches can be taken.

In Object-Oriented languages such as C++ this can easily be achieved by virtual functions, with each virtual function representing an operation that can be performed on a node. Virtual functions are simple to write, but it is usually impossible to add new operations to nodes without access to the source code. Alternatively, the **visitor pattern** can be used. This has a similar disadvantage in that it is similarly difficult to add new node types.

Other techniques involve the use of RTTI (Run-Time Type Information). The operation can be realised as a class which is passed the current node; it then queries the nodes type using RTTI and looks up the correct operation in an array of callbacks or functors. This requires that the map of types to functors or callbacks be initialized at runtime, but offers more flexibility, speed and extensibility.

Variations on these techniques exist and new methods can offer added benefits - one alternative is scene graph rebuilding where the scene graph is rebuilt for each of the operations performed, this however can be very slow but produces a highly optimised scene graph. This demonstrates that a good scene graph implementation depends heavily on the application in which it is used.

Traversals

Traversals are the key to the power of applying operations to scene graphs. A traversal generally consists of starting at some arbitrary node (often the root of the scene graph), applying the operation(s) (often the updating and rendering operations are applied one after the other), and recursively moving down the scene graph(tree) to the child nodes, until a leaf node is reached. At this point many scene graph engines then traverse back up the tree, applying a similar operation. For example, considering a render operation that takes transformations into account: while recursively traversing down the scene graph hierarchy a pre-render operation is called. If the node is a transformation

node it adds its own transformation to the current transformation matrix. Once the operation finishes traversing all the children of a node, it calls the node's post-render operation, so that the transformation node can undo the transformation. This approach drastically reduces the necessary amount of matrix multiplication.

Some scene graph operations are actually more efficient when nodes are traversed in a different order - this is where some systems implement scene graph rebuilding to reorder the scene graph into an easier to parse format or tree.

For example, in 2D cases, scene graphs typically render themselves by starting at the tree's root node and then recursively drawing the child nodes. The tree's leaves represent the most foreground objects. Since drawing proceeds from back to front with closer objects simply overwriting farther ones, the process is known as employing the Painter's algorithm. In 3D systems, which often employ depth buffers, it is more efficient to draw the closest objects first, since farther objects often need only be depth-tested instead of actually rendered, because they are occluded by nearer objects.

Scene graph and bounding volume hierarchies (BVHs)

Bounding Volume Hierarchies (BVHs) are useful for numerous tasks - including efficient culling and speeding up collision detection between objects. A BVH is a spatial structure but doesn't have to partition the geometry (see spatial partitioning, below).

A BVH is a tree of bounding volumes (often spheres, axis-aligned bounding boxes or/and oriented bounding boxes). At the bottom of the hierarchy the size of the volume is just large enough to encompass a single object tightly (or possibly even some smaller fraction of an object in high resolution BVHs). As one ascends the hierarchy each node has its own volume which tightly encompasses all the volumes beneath it. At the root of the tree is a volume that encompasses all the volumes in the tree (the whole scene).

BVHs are useful for speeding up collision detection between objects. If an object's bounding volume does not intersect a volume higher in the tree then it cannot intersect any object below that node (so they are all rejected very quickly).

Obviously there are some similarities between BVHs and scene graphs. A scene graph can easily be adapted to include/become a BVH - if each node has a volume associated or there is a purpose built 'bound node' added in at convenient location in the hierarchy. This may not be the typical view of scene graph but there are benefits to including a BVH in a scene graph.

Scene graphs and spatial partitioning

An effective way of combining spatial partitioning and scene graphs is by creating a scene leaf node that contains the spatial partitioning data. This data is usually static and generally contains non-moving level data in some partitioned form. Some systems may have the systems separate and render them separately, this is fine and there are no real advantages to either method. In particular it is bad to have the scene graph contained within the spatial partitioning system, this is because the scene graph is better thought of as the grander system to the spatial partitioning.

When it is useful to combine them

In short: Spatial partitioning will/should considerably speed up the processing and rendering time of the scene graph.

Very large drawings, or scene graphs that are generated solely at runtime (as happens in ray tracing rendering programs), require defining of group nodes in a more automated fashion. A raytracer, for example, will take a scene description of a 3D model and build an internal representation that breaks up its individual parts into bounding boxes (also called bounding slabs). These boxes are grouped hierarchically so that ray intersection tests (as part of visibility determination) can be efficiently computed. A group box that does not intersect an eye ray, for example, can entirely skip having to test any of its members.

A similar efficiency holds in 2D applications as well. If the user has magnified a document so that only part of it is visible on his computer screen, and then scrolls said document, it is useful to use a bounding box (or in this case, a bounding rectangle scheme) to quickly determine which scene graph elements are visible and thus actually need to be drawn.

Depending on the particulars of the application's drawing performance, a large part of the scene graph's design can be impacted by rendering efficiency considerations. In 3D video games such as Quake, for example, binary space partitioning (BSP) trees are heavily favored to minimize visibility tests. BSP trees, however, take a very long time to compute from design scene graphs, and must be recomputed if the design scene graph changes so the levels tend to remain static and dynamic characters aren't generally considered in the spatial partitioning scheme.

Scene graphs for dense regular objects such as heightfields and polygon meshes tend to employ quadtrees and octrees, which are specialized variants of a 3D bounding box hierarchy. Since a heightfield occupies a box volume itself, recursively subdividing this box into eight subboxes (hence the 'oct' in octree) until individual heightfield elements are reached is efficient and natural. A quadtree is simply a 2D octree.

X3D

X3d is a royalty-free open standards file format and run-time architecture to represent and communicate 3D scenes and objects using XML. It is an ISO ratified standard that provides a system for the storage, retrieval and playback of real time graphics content embedded in applications, all within an open architecture to support a wide array of domains and user scenarios.

PHIGS

PHIGS was the first commercial scene graph specification, and became an ANSI standard in 1988. Disparate implementations were provided by Unix hardware vendors. The "HOOPS 3D Graphics System" [1] appears to have been the first commercial scene graph library provided by a single software vendor. It was designed to run on disparate lower-level 2D and 3D interfaces, with the first major production version (v3.0) completed in 1991. Shortly thereafter, Silicon Graphics released IRIS Inventor 1.0 (1992), which was a scene graph built on top of the IRIS GL 3D API. It was followed up with Open Inventor in 1994, a portable scene graph built on top of OpenGL. More 3D scene graph libraries can be found in Category:3D Scenegraph APIs.

See also

- Graph theory
- Graph (data structure)
- Tree (data structure)
- Space partitioning

References

Books

- Leler, Wm and Merry, Jim (1996) *3D with HOOPS*, Addison-Wesley
- Wernecke, Josie (1994) *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, ISBN 0-201-62495-8 (Release 2)

Web sites and articles

- Strauss, Paul (1993). "IRIS Inventor, a 3D Graphics Toolkit" [2]
- Helman, Jim; Rohlf, John (1994). "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics" [3]
- Carey, Rikk and Bell, Gavin (1997). "The Annotated VRML 97 Reference Manual" [4]
- PEXTimes [5]
- Bar-Zeev, Avi. "Scenegraphs: Past, Present, and Future" [6]
- X3d
- OpenSceneGraph [7]
- OpenSG [8]
- Java3D [9]; Aviatrix3D [10], LG3D [11] ...
- jReality [12]
- Visualization Library [13]
- "Scenegraphs for 3D Software Development" [14]

References

- [1] <http://www.hoops3d.com>
- [2] <http://portal.acm.org/citation.cfm?id=165889>
- [3] <http://portal.acm.org/citation.cfm?id=192262>
- [4] <http://www.jwave.vt.edu/~engineer/vrml97book/ch1.htm>
- [5] <http://www.jch.com/jch/vrml/PEXTimes.txt>
- [6] <http://www.realityprime.com/scenegraph.php>
- [7] <http://www.openscenegraph.org>
- [8] <http://www.opensg.org>
- [9] <https://java3d.dev.java.net>
- [10] <http://aviatrix3d.j3d.org>
- [11] <https://lg3d.dev.java.net>
- [12] <http://www.jreality.de>
- [13] <http://www.visualizationlibrary.com>
- [14] <http://eureka3d.com/blog/?p=18>

Appendix

Big O notation

In mathematics, computer science, and related fields, **big O notation** (also known as **Big Oh notation**, **Landau notation**, **Bachmann–Landau notation**, and **asymptotic notation**) describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. Big O notation allows its users to simplify functions in order to concentrate on their growth rates: different functions with the same growth rate may be represented using the same O notation.

Although developed as a part of pure mathematics, this notation is now frequently also used in the analysis of algorithms to describe an algorithm's usage of computational resources: the worst case or average case running time or memory usage of an algorithm is often expressed as a function of the length of its input using big O notation. This allows algorithm designers to predict the behavior of their algorithms and to determine which of multiple algorithms to use, in a way that is independent of computer architecture or clock rate. Big O notation is also used in many other fields to provide similar estimates.

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.

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, for sufficiently large values of x , $f(x)$ is at most a constant times $g(x)$ 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 $x > x_0$:

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

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 ($T(1,000,000) = 1,000,000^2 = U(1,000,000)$). 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

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

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 $O(c^n)$ are very different. The latter grows much, much faster, no matter how big the constant c is (as long as it is greater than one). A function that grows faster than any power of n 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^2n^2 , and the big O notation ignores the constant c^2 . This can be written as $c^2n^2 \in O(n^2)$. 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

$$\begin{aligned} f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) &\Rightarrow f_1 f_2 \in O(g_1 g_2) \\ f \cdot O(g) &\in O(fg) \end{aligned}$$

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 $|h(x)| \leq Cg(x)$ 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 slower 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:

$$\begin{aligned} f(m) &= O(m^n), \\ g(n) &= O(m^n). \end{aligned}$$

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

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

rather than the less explicit

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

Complex usages

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

$$(n+1)^2 = n^2 + O(n)$$

$$(n+O(n^{1/2}))(n+O(\log n))^2 = n^3 + O(n^{5/2})$$

$$n^{O(1)} = O(e^n).$$

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.

Orders of common functions

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.

See table of common time complexities for a more comprehensive list.

Notation	Name	Example
$O(1)$	constant	Determining if a number is even or odd; using a constant-size lookup table or hash table
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree
$O(n^c)$, $0 < c < 1$	fractional power	Searching in a kd-tree
$O(n)$	linear	Finding an item in an unsorted list or a malformed tree (worst case); adding two n -digit numbers
$O(n \log n) = O(\log n!)$	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-adjoining grammar parsing; maximum matching for bipartite graphs
$L_n[\alpha, c]$, $0 < \alpha < 1 = e^{(c+\alpha(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$	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 traveling 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; 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 $O(n^{c+a})$ for any $a > 0$, 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 $f(x)$. It assumes that f and g are both functions of one variable. Formally, it states

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

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	As $n \rightarrow \infty$, eventually...	Definition
$f(n) \in O(g(n))$	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, n_0 \forall n > n_0 f(n) \leq g(n) \cdot k $ or $\exists k > 0, n_0 \forall n > n_0 f(n) \leq g(n) \cdot k$
$f(n) \in \Omega(g(n))$ (Older math papers sometimes use this in the weaker sense that $f = o(g)$ is false)	Big Omega	f is bounded below by g (up to constant factor) asymptotically	$ f(n) \geq g(n) \cdot k$ for some k	$\exists k > 0, n_0 \forall n > n_0 g(n) \cdot k \leq f(n) $
$f(n) \in \Theta(g(n))$	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 k_1, k_2	$\exists k_1, k_2 > 0, n_0 \forall n > n_0 g(n) \cdot k_1 < f(n) < 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 \leq f(n) $
$f(n) \sim g(n)$	on the order of; "twiddles"	f is equal to g asymptotically	$f(n)/g(n) \rightarrow 1$	$\forall \varepsilon > 0 \exists n_0 \forall n > n_0 f(n)/g(n) - 1 < \varepsilon$

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 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 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.

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., bullets 2 and 3 below) are usually strongly preferred over laxness of bound (i.e., bullet 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 (bullets 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 $o(n^\epsilon)$ for any constant k and any $\epsilon > 0$).

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

It is often useful to bound the running time of graph algorithms. Unlike most other computational problems, for a graph $G = (V, E)$ there are two relevant parameters describing the size of the input: the number $|V|$ of vertices in the graph and the number $|E|$ of edges in the graph. Inside asymptotic notation (and only there), it is common to use the symbols V and E , when someone really means $|V|$ and $|E|$. We adopt this convention here to simplify asymptotic functions and make them easily readable. The symbols V and E are never used inside asymptotic notation with their literal meaning, so this abuse of notation does not risk ambiguity. For example $O(E + V \log V)$ means $O((E, V) \mapsto |E| + |V| \cdot \log |V|)$ for a suitable metric of graphs. Another common convention—referring to the values $|V|$ and $|E|$ by the names n and m , respectively—sidesteps this ambiguity.

History

The notation 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 notation was popularized in the work of number theorist Edmund Landau; hence it is sometimes called a Landau symbol. It was popularized in computer science by Donald Knuth, who (re)introduced the related Omega and Theta notations.^[6] He also noted that the (then obscure) Omega notation had been introduced by Hardy and Littlewood^[7] under a slightly different meaning, and proposed the current definition. Hardy's symbols were (in terms of the modern O notation)

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

other similar symbols were sometimes used, such as \preceq and $\prec\!\prec$.

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.

See also

- Asymptotic expansion: Approximation of functions generalizing Taylor's formula
- Asymptotically optimal: A phrase frequently used to describe an algorithm that has an upper bound asymptotically within a constant of a lower bound for the problem
- Limit superior and limit inferior: An explanation of some of the limit notation used in this article
- Nachbin's theorem: A precise method of bounding complex analytic functions so that the domain of convergence of integral transforms can be stated
- Big O in probability notation : O_{p,o_p}

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.
- Marian Slodicka (Slodde vo de maten) & Sandy Van Wontergem. *Mathematical Analysis I*. University of Ghent, 2004.
- 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* [8]
- Paul E. Black, "big-O notation" [9], 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" [10], 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 " [11], 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 " [12], 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 " [13], 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 [14]

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/books?id=_tnwmvHmVwMC&pg=PA5&vq=%22The+trouble-is%22). Amsterdam: North-Holland. pp. 5–7. ISBN 9780486642215..
- [3] Ronald Graham, Donald Knuth, and Oren Patashnik (1994). *Concrete Mathematics* (<http://books.google.co.uk/books?id=pntQAAAAMAAJ&dq=editions:ISBN0201558025>) (2 ed.). Reading, Massachusetts: Addison-Wesley. p. 446. ISBN 9780201558029..
- [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] 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.
- [7] G. H. Hardy and J. E. Littlewood, *Some problems of Diophantine approximation*, Acta Mathematica 37 (1914), p. 225
- [8] <http://www.andrew.cmu.edu/~avigad/Papers/bigO.pdf>
- [9] <http://www.nist.gov/dads/HTML/bigOnotation.html>
- [10] <http://www.nist.gov/dads/HTML/littleOnotation.html>
- [11] <http://www.nist.gov/dads/HTML/omegaCapital.html>
- [12] <http://www.nist.gov/dads/HTML/omega.html>
- [13] <http://www.nist.gov/dads/HTML/theta.html>
- [14] <http://www.soe.ucsc.edu/classes/cmps102/Spring04/TantaloAsymp.pdf>

Amortized analysis

In computer science, especially analysis of algorithms, **amortized analysis** finds the average running time per operation over a worst-case *sequence* of operations. Amortized analysis differs from average-case performance in that probability is not involved; amortized analysis guarantees the time per operation over worst-case performance.

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

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

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

There are several techniques used in amortized analysis:

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

- The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.

Common use

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

References

- Allan Borodin and Ran El-Yaniv (1998). *Online Computation and Competitive Analysis* [1]. Cambridge University Press. pp. 20,141.
- 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 17: Amortized Analysis, pp.405–430.

References

[1] <http://www.cs.technion.ac.il/~rani/book.html>

Locality of reference

In computer science, **locality of reference**, also known as the **principle of locality**, is the phenomenon of the same value or related storage locations being frequently accessed. There are two basic types of reference locality. Temporal locality refers to the reuse of specific data and/or resources within relatively small time durations. Spatial locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, e.g., traversing the elements in a one-dimensional array.

Locality is merely one type of predictable behavior that occurs in computer systems. Systems which exhibit strong *locality of reference* phenomenon, are good candidates for performance optimization through the use of techniques, like the cache and prefetching technology concerning the memory, or like the advanced branch predictor at the pipelining of processors.

Locality of reference

The locality of reference, also known as the locality principle [1], is the phenomenon, that the collection of the data locations referenced in a short period of time in a running computer, often consists of relatively well predictable clusters. Important special cases of locality are *temporal*, *spatial*, *equidistant* and *branch* locality.

- **Temporal locality:** if at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. There is a temporal proximity between the adjacent references to the same memory location. In this case it is common to make efforts to store a copy of the referenced data in special memory storage, which can be accessed faster. Temporal locality is a very special case of the spatial locality, namely when the prospective location is identical to the present location.
- **Spatial locality:** if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. There is a spatial proximity between the memory locations, referenced at almost the same time. In this case it is common to make efforts to guess, how big neighbourhood around the current reference is worthwhile to prepare for faster access.

- **Equidistant locality:** it is halfway between the spatial locality and the branch locality. Consider a loop accessing locations in an equidistant pattern, i.e. the path in the **spatial-temporal coordinate space** is a dotted line. In this case, a simple linear function can predict which location will be accessed in the near future.
- **Branch locality:** if there are only few amount of possible alternatives for the prospective part of the path in the **spatial-temporal coordinate space**. This is the case when an instruction loop has a simple structure, or the possible outcome of a small system of conditional branching instructions is restricted to a small set of possibilities. Branch locality is typically not a spatial locality since the few possibilities can be located far away from each other.

In order to make benefit from the very frequently occurring *temporal* and *spatial* kind of locality, most of the information storage systems are hierarchical; see below. The *equidistant* locality is usually supported by the diverse nontrivial increment instructions of the processors. For the case of *branch* locality, the contemporary processors have sophisticated branch predictors, and on the base of this prediction the memory manager of the processor tries to collect and preprocess the data of the plausible alternatives.

Reasons for locality

There are several reasons for locality. These reasons are either goals to achieve or circumstances to accept, depending on the aspect. The reasons below are not disjoint; in fact, the list below goes from the most general case to special cases.

- **Predictability:** In fact, locality is merely one type of predictable behavior in computer systems. Luckily, many of the practical problems are decidable and hence the corresponding program can behave predictably, if it is well written.
- **Structure of the program:** Locality occurs often because of the way in which computer programs are created, for handling decidable problems. Generally, related data is stored in nearby locations in storage. One common pattern in computing involves the processing of several items, one at a time. This means that if a lot of processing is done, the single item will be accessed more than once, thus leading to temporal locality of reference. Furthermore, moving to the next item implies that the next item will be read, hence spatial locality of reference, since memory locations are typically read in batches.
- **Linear data structures:** Locality often occurs because code contains loops that tend to reference arrays or other data structures by indices. Sequential locality, a special case of spatial locality, occurs when relevant data elements are arranged and accessed linearly. For example, the simple traversal of elements in a one-dimensional array, from the base address to the highest element would exploit the sequential locality of the array in memory.^[1] The more general *equidistant locality* occurs when the linear traversal is over a longer area of adjacent data structures having identical structure and size, and in addition to this, not the whole structures are in access, but only the mutually corresponding same elements of the structures. This is the case when a matrix is represented as a sequential matrix of rows and the requirement is to access a single column of the matrix.

Use of locality in general

If most of the time the substantial portion of the references aggregate into clusters, and if the shape of this system of clusters can be well predicted, then it can be used for speed optimization. There are several ways to make benefit from locality. The common techniques for optimization are:

- to increase the locality of references. This is achieved usually on the software side.
- to exploit the locality of references. This is achieved usually on the hardware side. The *temporal* and *spatial* locality can be capitalized by hierarchical storage hardwares. The *equidistant* locality can be used by the appropriately specialized instructions of the processors, this possibility is not only the responsibility of hardware, but the software as well, whether its structure is suitable for compiling a binary program which calls the specialized instructions in question. The *branch* locality is a more elaborate possibility, hence more developing effort is needed, but there is much larger reserve for future exploration in this kind of locality than in all the remaining ones.

Use of spatial and temporal locality: hierarchical memory

Hierarchical memory is a hardware optimization that takes the benefits of spatial and temporal locality and can be used on several levels of the memory hierarchy. Paging obviously benefits from *temporal and spatial locality*. A cache is a simple example of exploiting temporal locality, because it is a specially designed faster but smaller memory area, generally used to keep recently referenced data and data near recently referenced data, which can lead to potential performance increases. Data in cache does not necessarily correspond to data that is spatially close in main memory; however, data elements are brought into cache one cache line at a time. This means that spatial locality is again important: if one element is referenced, a few neighboring elements will also be brought into cache. Finally, temporal locality plays a role on the lowest level, since results that are referenced very closely together can be kept in the machine registers. Programming languages such as C allow the programmer to suggest that certain variables are kept in registers.

Data locality is a typical memory reference feature of regular programs (though many irregular memory access patterns exist). It makes the hierarchical memory layout profitable. In computers, memory is divided up into a hierarchy in order to speed up data accesses. The lower levels of the memory hierarchy tend to be slower, but larger. Thus, a program will achieve greater performance if it uses memory while it is cached in the upper levels of the memory hierarchy and avoids bringing other data into the upper levels of the hierarchy that will displace data that will be used shortly in the future. This is an ideal, and sometimes cannot be achieved.

Typical memory hierarchy (access times and cache sizes are approximations of typical values used as of 2006 for the purpose of discussion; actual values and actual numbers of levels in the hierarchy vary):

- CPU registers (8-32 registers) – immediate access
- L1 CPU caches (32 KiB to 128 KiB) – fast access
- L2 CPU caches (128 KiB to 12 MiB) – slightly slower access
- Main physical memory (RAM) (256 MiB to 4 GiB) – slow access
- Disk (file system) (1 GiB to 1 TiB) – very slow
- Remote Memory (such as other computers or the Internet) (Practically unlimited) – speed varies

Modern machines tend to read blocks of lower memory into the next level of the memory hierarchy. If this displaces used memory, the operating system tries to predict which data will be accessed least (or latest) and move it down the memory hierarchy. Prediction algorithms tend to be simple to reduce hardware complexity, though they are becoming somewhat more complicated.

Spatial and temporal locality example: matrix multiplication

A common example is matrix multiplication:

```
for i in 0..n
    for j in 0..m
        for k in 0..p
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

When dealing with large matrices, this algorithm tends to shuffle data around too much. Since memory is pulled up the hierarchy in consecutive address blocks, in the C programming language it would be advantageous to refer to several memory addresses that share the same row (spatial locality). By keeping the row number fixed, the second element changes more rapidly. In C and C++, this means the memory addresses are used more consecutively. One can see that since j affects the column reference of both matrices C and B , it should be iterated in the innermost loop (this will fix the row iterators, i and k , while j moves across each column in the row). This will not change the mathematical result, but it improves efficiency. By switching the looping order for j and k , the speedup in large matrix multiplications becomes dramatic. (In this case, 'large' means, approximately, more than 100,000 elements in each matrix, or enough addressable memory such that the matrices will not fit in L1 and L2 caches.)

Temporal locality can also be improved in the above example by using a technique called *blocking*. The larger matrix can be divided into evenly-sized sub-matrices, so that the smaller blocks can be referenced (multiplied) several times while in memory.

```
for (ii = 0; ii < SIZE; ii += BLOCK_SIZE)
    for (kk = 0; kk < SIZE; kk += BLOCK_SIZE)
        for (jj = 0; jj < SIZE; jj += BLOCK_SIZE)
            for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++)
                for (k = kk; k < kk + BLOCK_SIZE && k < SIZE;
                    k++)
                    for (j = jj; j < jj + BLOCK_SIZE && j < SIZE;
                        j++)
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The temporal locality of the above solution is provided because a block can be used several times before moving on, so that it is moved in and out of memory less often. Spatial locality is improved because elements with consecutive memory addresses tend to be pulled up the memory hierarchy together.

See also

- Burst mode (computing)
- Row-major order
- File system fragmentation
- Cache-oblivious algorithm
- Partitioned global address space

Bibliography

- P.J. Denning, S.C. Schwartz, Communications of the ACM, Volume 15 , Issue 3 (March 1972), Pages:191-198

References

[1] Aho, Lam, Sethi, and Ullman. "Compilers: Principles, Techniques & Tools" 2nd ed. Pearson Education, Inc. 2007

Standard Template Library

C++ Standard Library
<ul style="list-style-type: none"> • ios • iostream • iomanip • fstream • sstream
Standard Template Library
<ul style="list-style-type: none"> • vector • deque • list • map • set • stack • queue • bitset • algorithm • functional • iterator
C Standard Library
<ul style="list-style-type: none"> • cassert • ctype • errno • limits • locale • cmath • csetjmp • csignal • cstdarg • cstddef • cstdio • cstdint • cstdlib • cstring • ctime

The **Standard Template Library (STL)** is a software library partially included in the C++ Standard Library. It provides *containers*, *iterators*, *algorithms*, and *functors*. More specifically, the C++ Standard Library is based on the STL published by SGI. Both include some features not found in the other. SGI's STL is rigidly specified as a set of headers, while ISO C++ does not specify header content, and allows implementation either in the headers, or in a true library.

Overview

The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL achieves its results through the use of templates. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL.

The STL was created as the first library of generic algorithms and data structures for C++, with four ideas in mind: generic programming, abstractness without loss of efficiency, the Von Neumann computation model, and value semantics.

Contents

Containers

The STL contains sequence containers and associative containers. The standard sequence containers include *vector*, *deque*, and *list*. The standard associative containers are *set*, *multiset*, *map*, and *multimap*. There are also container adaptors *queue*, *priority_queue*, and *stack*, that are containers with specific interface, using other containers as implementation.

Container	Description
Simple Containers	
pair	The pair container is a simple associative container consisting of a 2-tuple of data elements or objects, called 'first' and 'second', in that fixed order. The STL 'pair' can be assigned, copied and compared. The array of objects allocated in a map or hash_map (described below) are of type 'pair' by default, where all the 'first' elements act as the unique keys, each associated with their 'second' value objects.
Sequences (Arrays / Linked Lists) - ordered collections	
vector	a dynamic array, like C array (i.e., capable of random access) with the ability to resize itself automatically when inserting or erasing an object. Inserting and removing an element to/from back of the vector at the end takes amortized constant time. Inserting and erasing at the beginning or in the middle is linear in time. A specialization for type bool exists, which optimizes for space by storing bool values as bits.
list	a doubly-linked list; elements are not stored in contiguous memory. Opposite performance from a vector. Slow lookup and access (linear time), but once a position has been found, quick insertion and deletion (constant time).
deque (double ended queue)	a vector with insertion/erase at the beginning or end in amortized constant time, however lacking some guarantees on iterator validity after altering the deque.
Container adaptors	
queue	Provides FIFO queue interface in terms of push/pop/front/back operations. Any sequence supporting operations <code>front()</code> , <code>back()</code> , <code>push_back()</code> , and <code>pop_front()</code> can be used to instantiate queue (e.g. <i>list</i> and <i>deque</i>).
priority_queue	Provides priority queue interface in terms of push/pop/top operations (the element with the highest priority is on top). Any random-access sequence supporting operations <code>front()</code> , <code>push_back()</code> , and <code>pop_back()</code> can be used to instantiate priority_queue (e.g. <i>vector</i> and <i>deque</i>). Elements should additionally support comparison (to determine which element has a higher priority and should be popped first).
stack	Provides LIFO stack interface in terms of push/pop/top operations (the last-inserted element is on top). Any sequence supporting operations <code>back()</code> , <code>push_back()</code> , and <code>pop_back()</code> can be used to instantiate stack (e.g. <i>vector</i> , <i>list</i> , and <i>deque</i>).
Associative containers - unordered collections	

set	a mathematical set; inserting/erasing elements in a set does not invalidate iterators pointing in the set. Provides set operations union, intersection, difference, symmetric difference and test of inclusion. Type of data must implement comparison operator < or custom comparator function must be specified. Implemented using a self-balancing binary search tree.
multiset	same as a set, but allows duplicate elements.
map	an associative array; allows mapping from one data item (a key) to another (a value). Type of key must implement comparison operator < or custom comparator function must be specified. Implemented using a self-balancing binary search tree.
multimap	same as a map, but allows duplicate keys.
hash_set hash_multiset hash_map hash_multimap	similar to a set, multiset, map, or multimap, respectively, but implemented using a hash table; keys are not ordered, but a hash function must exist for the key type. These containers are not part of the C++ Standard Library, but are included in SGI's STL extensions, and are included in common libraries such as the GNU C++ Library in the <code>__gnu_cxx</code> namespace. These are scheduled to be added to the C++ standard as part of TR1, with the slightly different names of <code>unordered_set</code> , <code>unordered_multiset</code> , <code>unordered_map</code> and <code>unordered_multimap</code> .
Other types of containers	
bitset	stores series of bits similar to a fixed-sized vector of bools. Implements bitwise operations and lacks iterators. Not a Sequence.
valarray	another C-like array like vector, but is designed for high speed numerics at the expense of some programming ease and general purpose use. It has many features that make it ideally suited for use with vector processors in traditional vector supercomputers and SIMD units in consumer-level scalar processors, and also ease vector mathematics programming even in scalar computers.

Iterators

The STL implements five different types of iterators. These are *input iterators* (which can only be used to read a sequence of values), *output iterators* (which can only be used to write a sequence of values), *forward iterators* (which can be read, written to, and move forward), *bidirectional iterators* (which are like forward iterators but can also move backwards) and *random access iterators* (which can move freely any number of steps in one operation).

It is possible to have bidirectional iterators act like random access iterators, as moving forward ten steps could be done by simply moving forward a step at a time a total of ten times. However, having distinct random access iterators offers efficiency advantages. For example, a vector would have a random access iterator, but a list only a bidirectional iterator.

Iterators are the major feature which allow the generality of the STL. For example, an algorithm to reverse a sequence can be implemented using bidirectional iterators, and then the same implementation can be used on lists, vectors and deques. User-created containers only have to provide an iterator which implements one of the 5 standard iterator interfaces, and all the algorithms provided in the STL can be used on the container.

This generality also comes at a price at times. For example, performing a search on an associative container such as a map or set can be much slower using iterators than by calling member functions offered by the container itself. This is because an associative container's methods can take advantage of knowledge of the internal structure, which is opaque to algorithms using iterators.

Algorithms

A large number of algorithms to perform operations such as searching and sorting are provided in the STL, each implemented to require a certain level of iterator (and therefore will work on any container which provides an interface by iterators).

Functors

The STL includes classes that overload the function operator (`operator()`). Classes that do this are called functors or function objects. They are useful for keeping and retrieving state information in functions passed into other functions. Regular function pointers can also be used as functors.

A particularly common type of functor is the predicate. For example, algorithms like `find_if` take a unary predicate that operates on the elements of a sequence. Algorithms like `sort`, `partial_sort`, `nth_element` and all sorted containers use a binary predicate which must provide a strict weak ordering, that is, it must behave like a membership test on a transitive, irreflexive and antisymmetric binary relation. If none is supplied, these algorithms and containers use less^[1] by default, which in turn calls the less-than-operator `<`.

Criticisms

Quality of compiler

The Quality of Implementation (QoI) of the C++ compiler has a large impact on usability of STL (and templated code in general):

- Error messages involving templates tend to be very long and difficult to decipher. This problem has been considered so severe that a number of tools have been written which simplify and prettyprint STL-related error messages to make them more comprehensible.
- Careless use of STL templates can lead to code bloat. This has been countered with special techniques within STL implementation (using `void*` containers internally) and by improving optimization techniques used by compilers.
- Template instantiation tends to increase compilation time and memory usage (even by an order of magnitude). Until the compiler technology improves enough this problem can be only partially eliminated by very careful coding and avoiding certain idioms.

STL design issues

1. Design flaws due to limitations in the C++ language

- Initialization of STL containers with constants within the source code is not as easy as data structures inherited from C (addressed in C++0x with initializer lists).

2. Perceived flaws due to the requirement to maximize speed and minimize space usage

- STL containers are not intended to be used as base classes (their destructors are deliberately non-virtual); deriving from a container is a common mistake.^{[2] [3]}

3. Other flaws, caused either by trade-offs in design or which have become more visible over time

- The concept of iterators as implemented by STL can be difficult to understand at first: for example, if a value pointed to by the iterator is deleted, the iterator itself is then no longer valid. This is a common source of errors. Most implementations of the STL provide a debug mode which is slower but can locate such errors if used. However, at present no better replacement for iterators has been suggested and a similar problem exists in other languages, for example Java.
- Certain iteration patterns do not map to the STL iterator model. For example, callback enumeration APIs cannot be made to fit the STL model without the use of coroutines, which are platform-dependent and/or unavailable, and are outside the C++ standard.
- Compiler compliance does not guarantee that Allocator objects, used for memory management for containers, will work with state dependent behavior. For example, a portable library can't define an allocator type that will pull memory from different pools using different allocator objects of that type. (Meyers, p. 50)
- The set of algorithms is not complete — for example, the `copy_if` algorithm was left out by oversight (Stroustrup, p. 530) (added in C++0x: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2666.pdf>).
- The interface of some containers (in particular `string`) is bloated (Sutter and Alexandrescu, p. 79); others are considered insufficient.
- Hashing containers were left out of the original standard, but have been added in Technical Report 1, a recent extension to C++.

History

The architecture of STL is largely the creation of one person, Alexander Stepanov. In 1979 he began working out his initial ideas of generic programming and exploring their potential for revolutionizing software development. Although David Musser had developed and advocated some aspects of generic programming already by year 1971, it was limited to a rather specialized area of software development (computer algebra).

Stepanov recognized the full potential for generic programming and persuaded his then-colleagues at General Electric Research and Development (including, primarily, David Musser and Deepak Kapur) that generic programming should be pursued as a comprehensive basis for software development. At the time there was no real support in any programming language for generic programming.

The first major language to provide such support was Ada, with its generic units feature. By 1987 Stepanov and Musser had developed and published an Ada library for list processing that embodied the results of much of their research on generic programming. However, Ada had not achieved much acceptance outside the defense industry and C++ seemed more likely to become widely used and provide good support for generic programming even though the language was relatively immature. Another reason for turning to C++, which Stepanov recognized early on, was the C/C++ model of computation which allows very flexible access to storage via pointers is crucial to achieving generality without losing efficiency.

Much research and experimentation were needed, not just to develop individual components, but to develop an overall architecture for a component library based on generic programming. First at AT&T Bell Laboratories and later at Hewlett-Packard Research Labs, Stepanov experimented with many architectural and algorithm formulations, first in C and later in C++. Musser collaborated in this research and in 1992 Meng Lee joined Stepanov's project at HP and became a major contributor.

This work undoubtedly would have continued for some time being just a research project or at best would have resulted in an HP proprietary library if Andrew Koenig of Bell Labs had not become aware of the work and asked Stepanov to present the main ideas at a November 1993 meeting of the ANSI/ISO committee for C++ standardization. The committee's response was overwhelmingly favorable and led to a request from Koenig for a formal proposal in time for the March 1994 meeting. Despite the tremendous time pressure, Alex and Meng were able to produce a draft proposal that received preliminary approval at that meeting.

The committee had several requests for changes and extensions (some of them major), and a small group of committee members met with Stepanov and Lee to help work out the details. The requirements for the most significant extension (associative containers) had to be shown to be consistent by fully implementing them, a task Stepanov delegated to Musser. It would have been quite easy for the whole enterprise to spin out of control at this point, but again Stepanov and Lee met the challenge and produced a proposal that received final approval at the July 1994 ANSI/ISO committee meeting. (Additional details of this history can be found in Stevens.) Subsequently, the Stepanov and Lee document 17 was incorporated into the ANSI/ISO C++ draft standard (1, parts of clauses 17 through 27). It also influenced other parts of the C++ Standard Library, such as the string facilities, and some of the previously adopted standards in those areas were revised accordingly.

In spite of STL's success with the committee, there remained the question of how STL would make its way into actual availability and use. With the STL requirements part of the publicly available draft standard, compiler vendors and independent software library vendors could of course develop their own implementations and market them as separate products or as selling points for their other wares. One of the first edition's authors, Atul Saini, was among the first to recognize the commercial potential and began exploring it as a line of business for his company, Modena Software Incorporated, even before STL had been fully accepted by the committee.

The prospects for early widespread dissemination of STL were considerably improved with Hewlett-Packard's decision to make its implementation freely available on the Internet in August 1994. This implementation, developed by Stepanov, Lee, and Musser during the standardization process, became the basis of many implementations offered

by compiler and library vendors today.

List of STL implementations

- libstdc++ from gnu (was part of libg++)
- STLPort, based on SGI STL
- Rogue Wave standard library (HP, SGI, SunSoft, Siemens-Nixdorf)
- Dinkum STL library by P.J. Plauger

See also

- List of C++ template libraries
- C++0x
- Boost C++ Libraries

References

- Alexander Stepanov and Meng Lee, The Standard Template Library. HP Laboratories Technical Report 95-11(R.1), November 14, 1995. (Revised version of A. A. Stepanov and M. Lee: The Standard Template Library, Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.) [4]
- Alexander Stepanov (2007) (PDF). *Notes on Programming* [5]. Stepanov reflects about the design of the STL.
- Nicolai M. Josuttis (2000). *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley. ISBN 0-201-37926-0.
- Scott Meyers (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley. ISBN 0-201-74962-9.
- Al Stevens (March 1995). "Al Stevens Interviews Alex Stepanov" [6]. *Dr. Dobb's Journal*. Retrieved 2007-07-18.
- Bjarne Stroustrup (2000). *The C++ Programming Language* (3rd ed.). Addison-Wesley. ISBN 0-201-70073-5.
- Herb Sutter and Andrei Alexandrescu (2005). *C++ Coding Standards*. Addison-Wesley. ISBN 0-321-11358-6.
- David Vandevoorde and Nicolai M. Josuttis (2002). *C++ Templates: The Complete Guide*. Addison-Wesley Professional. ISBN 0-201-73484-2.
- Matthew Wilson (February 2004). "Callback Enumeration APIs & the Input Iterator Concept" [7]. *Dr. Dobb's Journal*.
- Atul Saini and David R. Musser, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Foreword by Alexander Stepanov; [Copyright Modena Software Inc.] Addison-Wesley ISBN 0-201-63398-1

External links

- STL programmer's guide [8] official guide from SGI
- STL Reference [9] lists STL containers and their members
- C/C++ reference [10] includes a section on the STL
- Apache (formerly Rogue Wave) C++ Standard Library Class Reference [11]
- Apache (formerly Rogue Wave) C++ Standard Library User Guide [12]
- Bjarne Stroustrup on The emergence of the STL [13] (Page 5, Section 3.1)
- STLPort [14], a very popular open-source implementation of the STL

References

- [1] <http://www.sgi.com/tech/stl/less.html>
- [2] Meyers, Scott (2005). *Effective C++ Third Edition - 55 Specific Ways to Improve Your Designs*. Addison Wesley. ISBN 0-321-33487-6.
- [3] Sutter, Herb; Alexandrescu, Andrei (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley.
- [4] <http://www.stepanovpapers.com>
- [5] <http://www.stepanovpapers.com/notes.pdf>
- [6] <http://www.sgi.com/tech/stl/drdobbs-interview.html>
- [7] <http://www.ddj.com/cpp/184401766>
- [8] <http://www.sgi.com/tech/stl/>
- [9] <http://www.cplusplus.com/reference/stl/>
- [10] <http://cppreference.com/>
- [11] <http://stdcxx.apache.org/doc/stdlibref/index.html>
- [12] <http://stdcxx.apache.org/doc/stdcbug/index.html>
- [13] <http://www.research.att.com/~bs/DnE2005.pdf>
- [14] <http://stlport.sourceforge.net/Home.shtml>

Article Sources and Contributors

Data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=360219271> *Contributors:* -- April, 195.149.37.xxx, 24.108.14.xxx, Abd, Abhishek.kumar.ak, Ahoerstemeier, Ahy1, Alansohn, Alexius08, Alhoori, Allan McInnes, Altenmann, Andre Engels, Andreas Kaufmann, Antonielly, Ap, Apoctylptic, Aravindvijayanpala, Arvindn, Babbage, Bharatshettybarkur, Bluemoose, BurntSky, Bushytails, CRGreathouse, Caiaffa, Carlette, Chris Lundberg, Closedmouth, Coldfire82, Conversion script, Corti, Cpl Syx, Craig Stuntz, Cubathy, DCDuring, DavidCary, Dcoetzee, Demix, Derbeth, Digitus, Dmoss, Easys12c, EconoPhysicist, Edaelon, Er Komandante, Espa, Eurooppa, Eve Hall, FinalMinuet, Fraggle, Frap, Fresheneesz, GPhilip, Garyzx, GeorgeBills, Ghyll, Giftlite, Gilliam, Glenn, Googl, Hairy Dude, HaiViet, Ham Pastrami, Helix84, Hypersonic12, Iridescent, JLaTondre, Jiang, Jimmytharpe, Jirkka6, Jorge Stolfi, Karl E. V. Palmen, Kh31311, Khukri, Kingpin13, Kingturtle, Kjetil r, LC, Lanek, Lanov, Laurențiu Dascălu, Liao, Ligulem, Lithui, Loadmaster, MTA, Mahanga, Marcin Suwalczan, Mark Renier, MasterRadius, Mdd, MertyWiki, Michael Hardy, Mindmatrix, Minesweeper, Mipadi, MisterSheik, Miym, Mr Stephen, MrJeff, Mushroom, Nanshu, Nick Levine, Nmp, Noah Salzman, Noldoaran, Nskillen, Obradovic Goran, Ohnoitsjamie, Oicumayberight, Orderud, PaePae, Pale blue dot, Pascal.Tesson, Paushali, Pgallert, Piet Delport, Populus, Prari, Pur3r4ngelw, Ramkumararan7, Reedy, Requestion, Rettetast, RexNL, ReyBrujo, Rhawn, Ripper234, Rrrwright, Ruud Koot, Ryan Roos, Sanjay742, Seth Ilys, Shadowjams, Shanes, Sharcho, Siroxo, Soumyasch, Spellsinger180, Strife911, Sundar sando, Tablizer, TakuyaMurata, Tanvir Ahmed, Tas50, Teles, Thadius856, The Thing That Should Not Be, Thinkdub, Thompson24, Thunderboltz, Tobias Bergemann, Tom 99, Traroth, TreveX, TuukkaH, Uriah123, User A1, UserGoogol, Vicarious, Vineetzone, Viriditas, Vortexrealm, Widefox, Wikilolo, Wmbolle, Wrp103, XJamRastafire, Yama, Yoric, یعنی، عیسی، 278 anonymous edits

Linked data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=359739654> *Contributors:* Andreas Kaufmann, Gmiller4th, Jorge Stolfi, Katieh5584, Miym, Spartlow, 15 anonymous edits

Succinct data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=350084479> *Contributors:* Algotime, Andreas Kaufmann, Camilo Sanchez, JyBy, Kmanmike15, Ryk, Shalom Yechiel, 4 anonymous edits

Implicit data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=349993858> *Contributors:* Andreas Kaufmann, Dcoetzee, Emilkeyder, Ilyathemuromets, RainbowCrane, Ruud Koot, Schizobullet, 1 anonymous edits

Compressed data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=350086433> *Contributors:* Algotime, Andreas Kaufmann, Eekster

Search data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=351753423> *Contributors:* Andreas Kaufmann, Clayhalliwell, Devendermishra, GTBacchus, Hosamaly, Jorge Stolfi, Quibik, Trigger hurt, Vicarious, X7q, 5 anonymous edits

Static and dynamic data structures *Source:* <http://en.wikipedia.org/w/index.php?oldid=359008400> *Contributors:* Altenmann, D6, Dcoetzee, Rich Farmbrough, Thejoshwolfe, Thusithajayalath, 9 anonymous edits

Persistent data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=355027889> *Contributors:* Allen3, Cdiggins, Charles Matthews, Chris0804, Dcoetzee, Edward, Haeleth, Harej, Hooperbloob, JIBB, Ken Hirsch, MarSch, Peruvianllama, Pomte, Qwertys, Ratan203, Scarverwiki, Seunosewa, Spoon!, Tobias Bergemann, Tomaxer, Totemsmartin, 18 anonymous edits

Concurrent data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=357875823> *Contributors:* Andreas Kaufmann, Hyang04, Ronjhones, Senthryl, Uriah123, 11 anonymous edits

Abstract data type *Source:* <http://en.wikipedia.org/w/index.php?oldid=359208522> *Contributors:* 209.157.137.xxx, A5b, Adrianwn, Aitias, Andreas Kaufmann, Anna Lincoln, Antonielly, Aqualung, Ark, Armin Rigo, B4hand, Baudway, BenRG, Bluebusy, Boing! said Zebedee, Brick Thrower, Cadacril, Chevymontcarlo, Chris the speller, Cob1, Conversion script, Corti, Daniel Brockman, Debresser, Dismantle101, Dreadstar, Dunks58, Elphf, Everton137, Garyzx, Ghettoblaster, Giftlite, Gmfirf, Haakon, Ideogram, Japanese SeaRobin, Jonathan.mark.lingard, Jorge Stolfi, Jpvinal, Kbdkank71, Kbrose, Kendrick Hang, Knutux, Leif, Lights, Liztanp, Mark Renier, Marudubshink, Merphant, Miaow Miaow, Michael Hardy, Mild Bill Hiccup, Mr Adequate, Nhey24, Noldoaran, Only2sea, PeterV1510, Pink18, Pohta ce-am pohtit, Populus, R. S. Shaw, RJHall, Rich Farmbrough, Ruud Koot, SAE1962, Sagaciousuk, SchfiftyThree, Sean Whitton, Sector7agent, Silvonen, Skysmith, SolKarma, SpallettaAC1041, Spoon!, Svick, The Arbiter, The Thing That Should Not Be, Tobias Bergemann, Tuukkah, W7cook, Wapcaplet, Wernher, Widefox, Wrp103, Yerpo, 186 anonymous edits

List *Source:* <http://en.wikipedia.org/w/index.php?oldid=347750999> *Contributors:* Alihaq717, Altenmann, Andrea Kaufmann, Andrew Eisenberg, Angela, BradBeattie, Brick Thrower, Calexico, Chevan, Chowbok, Christian List, Classicalegen, Cmdrjameson, Crater Creator, Cybercobra, Daniel Brockman, Delirium, Denisp, Dgreen34, Dijxtra, Dismantle101, Docu, Drag, Eao, Ed Cormany, Elf, Elwikipedista, EugeneZelenko, Falk Lieder, Fredrik, Glenn, HQCentral, Ham Pastrami, Hyacinth, Jan Hidders, Jareha, Jeff3000, Jeffrey Mall, Jorge Stolfi, Josh Parris, Joswig, Ketiltrottr, Liao, ManN, Mav, Mic, Michael Hardy, Mickeymousechen, Mike.nicholaides, Mipadi, Neilc, Noldoaran, P0nc, Paddy3118, Palmard, Patrick, Paul G, Paul foord, Peak, Pohta ce-am pohtit, Poor Yorick, Prumpf, Puckly, R. S. Shaw, Ruud Koot, Salix alba, Samuelsen, Spoon!, Stormie, TShilo12, TakuyaMurata, The Thing That Should Not Be, Tokek, VictorAnyakin, WODUP, Wavelength, Wmahan, Wnissen, Wwwolf, XJamRastafire, ZeroOne, 81 anonymous edits

Stack *Source:* <http://en.wikipedia.org/w/index.php?oldid=360214108> *Contributors:* 1exec1, 202.144.44.xxx, 2mem, Aaron Rotenberg, Aavviof, Adam majewski, Adam78, Agateller, Ahluka, Aillema, Aitias, Al Lemos, Alxeedo, Andre Engels, Andreas Kaufmann, Andrej, Angusmcellar, Arch dude, Arkahot, Arvindn, BenFrantzDale, Bobo192, Boivie, Bookmaker, Borgx, Bsmntbomddoo, Caerwine, Calliopejen1, CanisRufus, Ceriak, Cheusov, Chris the speller, Christian List, Cjhoyle, Clx321, Cmcormick8, Colin meier, Conversion script, Corti, CosineKitty, Dan Granahan, David.Federman, Davidhorman, Dcoetzee, Dillesca, Dinoen, ENeville, EatMyShortz, F15x28, Faysol037, Fernandopabon, Finlay McWalter, Fragment, Fredrik, FrontLine, Funandtrvl, Funky Monkey, Geeg, Gggh, Ghettoblaster, Giftlite, Gonzen, Graham87, Grue, Guy Peters, Gwernol, Hackwrench, Ham Pastrami, Hangfromthefloor, Hariva, Hgfernan, Hook43113, IanOsgood, Individual IX, Ionutzmovie, Ifxf64, Jake Nelson, James Foster, JensMueller, Jesse Viviano, Jheiv, Johnnuniq, Jzlae, Karl-Henner, Kbdkank71, Klower, KnightRider, Liao, Loadmaster, LobStoR, Luciform, Manassehkatz, Marc Mongenet, Mark Renier, MartinHarper, Maxim Razin, Maximimax, Mbessey, Mdd, MegaHasher, Mentifisto, Michael Hardy, Michael Stone, Mindmatrix, Mipadi, Mpkr, Modster, Mohinib27, Murray Langton, Musiphil, Myasuda, Netkinetic, NoirNoir, Noldoaran, Notheruser, Obradovic Goran, Patrick, PeterJeremy, Physicistjedi, Pion, Poccil, Pomte, Postrach, Quantran202, R. S. Shaw, RDBrown, RTC, Raviemani, Reikon, RevRagnarok, ReyBrujo, Robbe, Rustamabd, Ruud Koot, Sanjay742, Seth Manapio, SimenH, SiobhanHansa, Slgrandson, Spoon!, Stan Shebs, Stephemb, Stevenbird, TakuyaMurata, Tasc, The Anome, Thumperward, Traroth, Tsf, Tuukkah, Unara, VT Bass Matt, VampWillow, Vystrix Nethox, Whosasking, Wikidan829, Wikilolo, WiseWoman, Wj32, Wlievens, Xdenizen, Yammesicka, Zchenyu, 222 anonymous edits

Queue *Source:* <http://en.wikipedia.org/w/index.php?oldid=358023900> *Contributors:* 16@r, Ahoerstemeier, Almgklor, Andre Engels, Andreas Kaufmann, Arsenic99, Atiflz, Banditlord, BenFrantzDale, Bobo2000, Brain, Caerwine, Caesura, Calliopejen1, Carlosguitar, Cdills, Chairboy, Chu Jetcheng, Ckatz, Contactabnash, Conversion script, Corti, Dabear, DavidLevinson, Dcoetzee, Discospinster, EatMyShortz, Edward, Egerhart, Emperorbma, Ewlyahocom, Fredrik, Furykef, Gbduende, Ggia, Giftlite, Glenn, GrahamDavies, Gunslinger47, Ham Pastrami, Hariva, Helix84, Honza Záruba, Howcheng, Indl, Ifxf64, Jesin, Jguk, JohJak2, Joseph.w.s, JosephBarillari, Jusjih, Keilana, Kenyon, Kletos, Ksull10, LapoLuchini, Liao, Loupete, Lperez2029, M2MM4M, Mark Renier, Marry314113, Massysett, Maw, Maxwellterry, Mehrenberg, Metasquares, Michael Hardy, Mike1024, MikeDunlavey, Miklcet, Mindmatrix, Mpkr, Noldoaran, Nwbeeson, Patrick, Peng, PhilipR, Pissant, PrometheeFeu, PseudoSudo, Qwertys, Rachael1, Rasmus Faber, Rdsmith4, Ruby.red.roses, Ruud Koot, Sanjay742, SensuiShinobu1234, Sharcho, SimenH, SiobhanHansa, So Says Chappy, Spoon!, SpuriousQ, Stassats, Stephemb, Thadius856, TheSuperslacker, TobiasPersson, Traroth, Tsemii, Uruiamme, VT Bass Matt, Vegpuff, W3bb, Wikibarista, Wikilolo, Woohookitty, Wrp103, Zoney, Zotel, Zvar, ۱۴۴ anonymous edits

Deque *Source:* <http://en.wikipedia.org/w/index.php?oldid=359803547> *Contributors:* Aamirlang, Andreas Kaufmann, Anonymous Dissident, Bcbell, BenFrantzDale, CesarB, Conversion script, Cparker, Dcoetzee, Drmrgrv, Esrogs, Fabartus, Fbergo, Felix C, Stegerman, Ffaarr, Fox, Frecklefoot, Fredrik, Funnyfarmofdoom, Furykef, Ham Pastrami, Hawk777, Jengelh, KILNA, Kbrose, Kurykh, Merovingian, Mindmatrix, Mwhitlock, Naraht, Offby1, Omicronpersei8, Psiphiorg, Pt, Puetzk, Rasmus Faber, Ripper234, Rosen, Rrmsjp, Schellhammer, Silly rabbit, Sj, Sneftel, Spoon!, SpuriousQ, The Anome, TimBentley, VictorAnyakin, Wikiboff, Zoicon5, 94 anonymous edits

Priority queue *Source:* <http://en.wikipedia.org/w/index.php?oldid=360105876> *Contributors:* Andreas Kaufmann, BACbKA, Bdolan, BlackFingolfin, Bobo192, Byrial, Chrisahn, Clicketyclack, Codrdan, Conversion script, Corp, CosineKitty, Cybercobra, David Eppstein, Dbeardsl, Dcoetzee, Decrypt3, El C, Emeraldemon, EnOreg, Frecklefoot, Fredrik, Ghettoblaster, Gilliam, Hdante, HenryAyoola, Ilingod, Itus15q4user, Jeff02, Jeltz, John Reed Riley, Jutiphan, Kenyon, Kowey, Krungie factor, LastKingpin, LeeG, Mbloore, Mentifisto, Michael Hardy, Nixdorf, Nyenec, Omicronpersei8, Orderud, PaulWright, Pdelong, RHaden, Red Act, Rhanekom, Riedl, Robbins, Ruud Koot, RyanGerbil10, Sanxiyn, Silly rabbit, Spl, Spoon!, SpuriousQ, The Anome, TimBentley, VictorAnyakin, Wikiboff, Zoicon5, 94 anonymous edits

Map *Source:* <http://en.wikipedia.org/w/index.php?oldid=357759948> *Contributors:* Agorf, Ajo Mama, Alansohn, Altenmann, Alvin-cs, AmiDaniel, Andreas Kaufmann, Anna Lincoln, Antonielly, AvramYU, B4hand, Bart Massey, Bartledan, Bevo, Bluemoose, Bobo192, Boothy443, Bosmon, Brianiac, Brianski, Catbar, Cfallin, Chaos5023, CheesyPuffs144, Comet-berkeley, Countchoc, CultureDrone, Cybercobra, Damian Yerrick, DavidDouthitt, Davidwhite544, Dcoetzee, Debresser, Decltype, Deineka, Dggoldst, Doc aberdeen, Doug Bell, Dreftymac, Dysprosia, EdC, Edward, Ericamick, EvanED, Floatingdecimal, Fredrik, Fubar Obfuscous, George100, Graue, Hashar, Hirzel, Hugo-cs, Int19h, Inter, Irishjugg, JForget, JLaTondre, James b crocker, Jdh30, Jeff02, Jerazol, Jerryobject, Jesdisciple, Jleedev, Jokes Free4Me, JonathanCross, Jorge Stolfi, Jpo, Karol Langner, Kdau, Kglatin, KnowledgeOfSelf, Krischik, Kusunose, LeeHunter, Macrakis, Maerk, Margin1522, Maslin, Maury Markowitz, Michael Hardy, Minesweeper, Minghong, Mintleaf, Mirzabah, MrSteve, Mt, Neilc, Nemo20000, Nick Levine, Noldoaran, ObsidianOrder, Orbnauticus, Orderud, PP Jewel, Paddy3118, PanagosTheOther, Patrick, Paul Ebermann, Pfast, Pfunk42, Pgr94, PhiLho, Pimlottc, Pne, Pohta ce-am pohtit, Radagast83, RainbowOfLight,

RevRagnarok, RexNL, Robert Merkel, Ruakh, RzR, Sam Pointon, Samuelsen, Shellreef, Signalhead, Silvonen, Sligocki, Spoon!, Swmcd, TShilo12, TheDoctor10, Tobe2199, Tony Sidaway, TuukkaH, Vegard, Wlievens, Wmbolle, Wolfkeeper, Yurik, Zven, 228 anonymous edits

Bidirectional map *Source:* <http://en.wikipedia.org/w/index.php?oldid=356063694> *Contributors:* Andreas Kaufmann, Cobi, Cyc, GregorB, Mattbierner, 1 anonymous edits

Multimap *Source:* <http://en.wikipedia.org/w/index.php?oldid=340038464> *Contributors:* Aminorex, Andreas Kaufmann, Bantman, Bluebusy, Cybercobra, Enochlau, Fuhghettaboutit, GCarty, JakobVoss, Macha, Spoon!, Svick, TheDJ, TuukkaH, Wmbolle, 10 anonymous edits

Set *Source:* <http://en.wikipedia.org/w/index.php?oldid=355767283> *Contributors:* Andreas Kaufmann, ClassicaLecon, Cybercobra, Damian Yerrick, Davecrosby uk, Dcoetzee, Dreadstar, EdH, Fredrik, Gracenotes, Hetur, Hosamaly, Incnis Mrsi, Jorge Stolfi, Linforest, Loupeter, Lt basketball, Lvr, MegaHasher, Mintguy, Modster, Mxn, Otus, Oxymoron83, Patrick, Pfunk42, Pohta ce-am pohit, QmunkE, Quintaylor, RJFJR, Rhamekom, SoSaysChappy, Spoon!, TakuyaMurata, Twri, Tyamatth, WikiIolo, 20 anonymous edits

Tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=360131466> *Contributors:* 209.157.137.xxx, 2help, Abce2, Adrianwn, Afrozenator, Alfredo J. Herrera Lago, Alvestrand, Andrevan, Anonymous Dissident, AxelSelD, BAxelrod, Bernard Fran ois, BlckKnight, Bobo192, Boleslav Bobcik, Bryan Derksen, Can't sleep, clown will eat me, Cbraga, Cfallin, Conversion script, Corti, Creidieki, Cruccone, Cybercobra, Dcandete, Dcoetzee, Defza, DevastatorIIC, Digitalme, Dillard421, Dysprosia, Eh kia, Evercat, Fabrigamos, Fabricationary, FleetCommand, Fredrik, Fresheneesz, Garyzx, Giftlife, Giraffedata, Glenn, Ham Pastrami, Happynomad, Hashbrowncipher, Hotlorp, INKbusse, Iamscked, Iamtheudeus, Ifroelich, Jitse Niesen, Jiy, Jorge Stolfi, JulesH, Kausikghatak, KellyCoinGuy, Kineticman, Kjetil r, Knutux, KoRnholio8, Kpeeters, Kurykh, Ladsroup, Liao, Lisamh, Loisel, MER-C, MagiMaster, Majorly, Mark Renier, Mattisgo, McKaysalisbury, Mdd, Mdd4696, Mdebets, Mentifisto, Mercury1992, Michael A. White, Mike314113, Mindmatrix, Mnernex, Mnogo, Mrwojo, Myanw, NHRHS2010, Night Gyr, Nikai, Nivix, Nsaa, Nuno Tavares, Ollydb, Paul August, Perique des Pallettes, Pi is 3.14159, Pingveno, Pradeepvaishy, Qwertys, R. S. Shaw, RazorICE, Reetep, Rich Farmbrough, Ripper234, Robertbowerman, RoyBoy, Rp, Ruakh, Rudd Koot, RzR, Sara wiki, Sbowers3, Seb, Shamus1331, Silly rabbit, SiobhanHansa, Skittleys, Skraz, SlowJog, Solbris, SteveWitham, T Long, Taemyr, Tarquin, The Anome, Thomas Bj rkan, Thunderboltz, Tobias Bergemann, Tomhubbard, Trusilver, Two Bananas, Twri, Utcursch, VKokielov, Vdm, WhyBeNormal, Wsload, Xevior, 208 anonymous edits

Array data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=360030134> *Contributors:* 16@r, 209.157.137.xxx, AbstractClass, Ahy1, Alfio, Alksentr, Alksub, Andre Engels, Andreas Kaufmann, Anonymous Dissident, Anwar saadat, Apers0n, Army1987, Atanamir, Awbell, B4hand, Badanedwa, Bargomm, Beej71, Beetstra, Beland, Beliavsky, BenFrantzDale, Berland, Betacommand, Bill37212, Blue250, Borgx, Bradjamesbrown, Brick Thrower, Btx40, Caltas, Cameltrader, Cgs, Chris glenne, Conversion script, Corti, Cybercobra, DAGwyn, Danakil, Darkspots, DavidCary, Dcoetzee, Derek farn, Dmason, Dreftymac, Dysprosia, ESkog, EconoPhysicist, Ed Poor, Engelec, Fabartus, Footballfan190, Fredrik, Funandtrvl, Func, Fvw, G worroll, Garde, Gaydudes, George100, Gerbrant, Giftlife, Graue, Grika, GwydionM, Henry513414, Hide&Reason, Highegg, Icarins, Immortal Wowbagger, Intrg, J.delanoy, JLaTondre, JaK81600, Jackollie, Jeff3000, Jh51681, Jimbryho, Jkl, Jleede, Jlmerrill, Jogloran, John, Johnuniq, Jonathan Grynspan, Jorge Stolfi, Josh Cherry, JulesH, Kaldosh, Karol Langner, Kbdank71, Kbrose, Ketiltrout, Kimchi.sg, Krischik, Kukini, LAX, Lardarse, Laurentiu Dasc u, Liempt, Ligulem, Ling.Nut, Lockeownzj00, Lowellian, Macrakis, Massyett, Masterdriverz, Mattb90, Matthewsim, Mcaruso, Mdd, Merlinssor, Mfahaem007, Mfb52, Michael Hardy, Mike Van Emmerik, Mikeblas, MisterSheik, Mr Adequate, Mrstonky, Mtwoitoes, Narayanese, Nicvaroce, Nixdorf, Norm, Orderud, Oxymoron83, Patrick, Phi.lho, Piet Delport, Poor Yorick, Princeatapi, Pseudomonas, Qutezuze, Quuxpluseone, RTC, Rbj, Redacteur, ReyBrujo, Rgrig, Rilak, Rossami, Rudd Koot, SPTWriter, Sagaciousuk, Sewing, Sharkface217, Simeon, SiobhanHansa, Skittleys, Slakr, Slogswipe, Smremde, Spoon!, Squidonius, Ssd, Stephenb, Strangely, Suruema, TakuyaMurata, Tamfang, Tawwasser, Thadius856, The Anome, The Thing That Should Not Be, Themania, Thinggg, Timneu22, Travelbird, Trevyn, Trojo, User A1, Visor, Vistro, Wernher, Wws, Yamamoto Ichiro, ZeroOne, Zzedar, 277 anonymous edits

Row-major order *Source:* <http://en.wikipedia.org/w/index.php?oldid=351292638> *Contributors:* AaronWL, BenFrantzDale, Drf5n, Jungrung, Mcld, Mikeblas, Mstahl, Nzroller, Pearl, Sintaku, Splintercellguy, Stevenj, Tedernst, Welsh, Zawersh, 33 anonymous edits

Dope vector *Source:* <http://en.wikipedia.org/w/index.php?oldid=302518652> *Contributors:* Amalas, Andreas Kaufmann, Dougher, Finn-Zoltan, Gennaro Prota, Phresnel, Sjorford, Wik, Xexo, 6 anonymous edits

Iliffe vector *Source:* <http://en.wikipedia.org/w/index.php?oldid=306189573> *Contributors:* Alynnna Kasmira, Andreas Kaufmann, Cmdrjameson, Doradus, Feyd, Lkinkade, Mgreenbe, NapoliRoma, Ospalh, RainbowCrane, Wotnarg, 9 anonymous edits

Dynamic array *Source:* <http://en.wikipedia.org/w/index.php?oldid=354368669> *Contributors:* Alex.vatchenko, Andreas Kaufmann, Beetstra, Cobi, Cybercobra, Damian Yerrick, David Eppstein, Dcoetzee, Decltype, Dpm64, Forbsey, Fresheneesz, Furykef, Garyzx, Green caterpillar, Icep, Ixfd64, Jorge Stolfi, Karol Langner, MegaHasher, MisterSheik, Moxfyre, Octahedron80, Ryk, SPTWriter, Simonykill, Spoon!, Wikilolo, Wrp103, ZeroOne, 阿卜杜勒·薩海, 23 anonymous edits

Hashed array tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=297042548> *Contributors:* Cobi, Dcoetzee, Garyzx, MegaHasher, Queenmomcat, RainbowCrane, Surachit

Gap buffer *Source:* <http://en.wikipedia.org/w/index.php?oldid=356137436> *Contributors:* Alf, Andreas Kaufmann, Charles Matthews, Damian Yerrick, Dpm64, Hosamaly, Hydrostatic, J04n, Jaberwocky6669, Jacobslusser, Jogloran, LittleDan, MisterSheik, Pgdn002, Populus, 8 anonymous edits

Circular buffer *Source:* <http://en.wikipedia.org/w/index.php?oldid=359846736> *Contributors:* Amikake3, Andreas Kaufmann, Anonymi, Astronomouth7303, Calliopejen1, Cburnett, DrZoomEN, Hosamaly, Jennavecia, Joeyadams, Julesd, KiloByte, Lucius Annaeus Seneca, Malcohol, Marokwitz, Mike65535, OlivierEM, OrlinKolev, Parthashome, Paulitex, Rhanekom, Relf, Shabble, Silly rabbit, Ybungabill, Zoxic, 魏文, 48 anonymous edits

Sparse array *Source:* <http://en.wikipedia.org/w/index.php?oldid=352308066> *Contributors:* 16@r, Cadillac, Charles Matthews, Engelec, Hgranqvist, Incnis Mrsi, Karol Langner, KurtRaschke, MisterSheik, Monedula, Pako, RainbowCrane, Rjwilmsi, 14 anonymous edits

Bit field *Source:* <http://en.wikipedia.org/w/index.php?oldid=353818381> *Contributors:* Andreas Kaufmann, AndrewHowse, Bovlb, Calliopejen1, Cic, Damian Yerrick, Davnor, Derek farn, Exe, Furykef, Glen Pepicelli, Jengelh, Jhodapp1, Mskfisher, Nbarth, Ozzmosis, Pdbe, Rjwilmsi, Rudd Koot, Samuel Blanning, Scott Paeth, Skwa, Solsticedhiver, Spoon!, WikiTobi, 32 anonymous edits

Bit array *Source:* <http://en.wikipedia.org/w/index.php?oldid=358735226> *Contributors:* AJim, Andreas Kaufmann, Archimerged, Bluebusy, Bluemoose, Bovlb, Cobi, Cocciasik, Davnor, Dcoetzee, Doug Bell, Furykef, Glen Pepicelli, Hack-Man, Intgr, JLaTondre, Jacobolus, JesseW, Karol Langner, Kubanczyk, Notinasmaid, Onomou, Orderud, Paul August, Pcordes, R. S. Shaw, Rrigr, RomainThibaux, Sam Hocevar, Skwa, Spoon!, StuartBrady, TheChrisD, TomJF, 33 anonymous edits

Bitboard *Source:* <http://en.wikipedia.org/w/index.php?oldid=341010234> *Contributors:* Andreas Kaufmann, Avalon, Bigtimepeace, Brainix, Bubba73, CharlesGillingham, DGJM, Damian Yerrick, Danielmachin, Ddxc, Dissident, Doug Bell, Dr.Szlachedzki, Dwheeler, Dylan Lake, Epolk, Furykef, Glen Pepicelli, IanOsgood, Jleede, Kaimiddleton, Kenneth Cochran, LokiClock, M412k, MER-C, MRRProgrammer, Marudubshinki, Mik03k, Mynameisnotpi, Nibuod, Notheruser, Nv8200p, Pawnkingthree, Pearle, PrometheusX303, Psu256, QTCaptain, Quackor, RJFJR, RoyBoy, Slamb, Srlefle, Tromp, Trovatore, WiiWillieWiki, Zargulon, ZeroOne, 98 anonymous edits

Parallel array *Source:* <http://en.wikipedia.org/w/index.php?oldid=299808024> *Contributors:* AlanBarrett, Alik Kirillovich, Andreas Kaufmann, Charles Matthews, Dcoetzee, Garyzx, GregorB, Ironholds, Karol Langner, MisterSheik, Peter bertok, Ragzouken, TakuyaMurata, TheProgrammer, Thorwald, Veledan, 6 anonymous edits

Lookup table *Source:* <http://en.wikipedia.org/w/index.php?oldid=356366663> *Contributors:* Andreas Kaufmann, Angela, Arteitle, Berland, CRGreathouse, Charles Matthews, Cheese Sandwich, David@wickrill.com, Davidmaxwaterman, Dcoetzee, Dylan Lake, Dysprosia, Falcor84, Fingew, Fredrik, Gazoot, Giftlife, Girolamo Savonarola, Graemec2, JLaTondre, Jelsova, Jengelh, KSmrq, Kdakin, Kencf016, Khalid hassani, Kostmo, Melesse, Michael Hardy, MrDomino, Nbarth, NicM, Omegatron, Peter S., Poco a poco, Qef, Ralian, Sadi, Sterrys, SteveBaker, Supernoob, Yflicker, ZAB, 99 anonymous edits

Linked list *Source:* <http://en.wikipedia.org/w/index.php?oldid=359998106> *Contributors:* 12.234.213.xxx, 16@r, 65.91.64.xxx, 75th Trombone, ATren, Achurch, Adamking, Adrian, Aeons, Agrammenos, Ahy1, Amog, Anandvachhani, Andreas Kaufmann, Angela, Antaeus Feldspar, Apeculiaz, Apollo2991, Arakunem, Arivne, Arneth, BL, Beejaye, Beetstra, BenFrantzDale, BigDunc, Binaryedit, Bluezy, Bobo192, Borgx, Bughunter2, Celticicer, Cesarsorm, Christian *Yendi* Severin, Clayhalliwell, Cloudw81, Cobi, ColdFusion650, Colonies Chris, Constructive editor, Conversion script, Corti, Creidieki, CryptoDerk, Cybercobra, Danarmstrong, Dark knight ita, Darkfilac, David Eppstein, Dcoetzee, Deeptrivia, Deshrail, Dillesca, Dixie91, Docboat, Drini, Elf, Enochlau, Etienne Lehnhart, Fawcett5, Ferdinand Pienaar, FlashSheridan, Fredrik, Freqsh0, Fubar Obsfuso, G7mluvn, Garyzx, Haeleth, Ham Pastrami, HappyInGeneral, Hariva, Hook43113, IE, IceKarma, Ijs1, Ingr, Iridescent, J0hn7On, JFreeman, Jan.Smolik, Japo, Jarsyl, Javawizard, Jengelh, Jin, JohnnyDog, JonHarder, Jorge Stolfi, Justin W Smith, Karingo, Karl-Henner, Katieh5584, Kbrose, Kelmar, Kenny Moens, Kenny sh, Kenyon, Kku, Koweja, KralSS, Lasindi, Leafboat, Levin, Lindsayfgleimour, Lord Pistachio, MER-C, Mantipula, Marek69, Mark Renier, MatrixFrog, Mboverload, Mc6809e, Mdd, MegaHasher, Meshari alnaim, Mietchen, Mindmatrix, Minesweeper, MisterSheik, Miym, Mtasic, Mwe 001, Mygerardromance, Nasty psycho, Neil916, Neilc, NewName, Nickj, Nixdorf, Noah Salzman, PavelY, Pengo, Perique des Palottes, PhilKnight, Pi is 3.14159, Poccil, Quantran202, Quantumobserver, Quentin mcalmott, R. S. Shaw, RJFJR, Raghaven, Ravek, Raven4x4, Redmarkviolinist, RichF, RickK, Rnt20, Rudd Koot, SPTWriter, Sam Hocevar, Sam Pointon, Samber, Quandoninja, Shanes, Silsor, Silvonen, SiobhanHansa, Smack, Smiler jerg, Sonett72, Spoon!, Sprocter, Ste4k, Stevage, Stevertigo, StuartBrady, Stw, Supadawg, Ta bu shi da yu, Tabletop, TakuyaMurata, TedPostol, Tetracube, Thadius856, The Thing That Should Not Be, TheCoffee, Theodore Kloba, Thue, Tiddly Tom, TimBentley, Tobias Bergemann, TobiasPersson, Tomaxer, Traroth, Trusilver, Tyomitch, Unixxx, Upnishad, Uriyan,

VTBassMatt, Versageek, Vhcomptech, Vipinhari, Waltercruz, Wereon, Wikilolo, Willemo, Wizmo, Wjl2, Wolfrock, Wrp103, Ww, Xenochria, Yacht, Zvn, 521 anonymous edits

XOR linked list *Source:* <http://en.wikipedia.org/w/index.php?oldid=354367506> *Contributors:* Ais523, Andreas Kaufmann, Cacycle, Cjoev, Cyp, Damian Yerrick, Dav4is, Dcoetzee, Eequor, Evil saltine, Haeleth, Itai, J04n, Jketola, Jon Awbrey, Nick, Nixdorf, Pakaran, Paul August, Quotemstr, Quuxplusone, Sander123, Siroxo, Tomaxer, Zoicon5, 20 anonymous edits

Unrolled linked list *Source:* <http://en.wikipedia.org/w/index.php?oldid=348148903> *Contributors:* Afrikaner, Andreas Kaufmann, Costaluz, Damian Yerrick, DavidCary, Dcoetzee, Decora, EatMyShortz, Fx2, Garyzx, Johnbod, Kdau, Neilc, Potatoswatter, Warren, Yuriybrisk, 5 anonymous edits

VList *Source:* <http://en.wikipedia.org/w/index.php?oldid=347671026> *Contributors:* Andreas Kaufmann, Cdiggins, Dawdler, Dcoetzee, Fredrik, Irrbloss, Jbolden1517, Ken Hirsch, MegaHasher, Ripounet, Sandstein, WriterHound, 12 anonymous edits

Skip list *Source:* <http://en.wikipedia.org/w/index.php?oldid=360151225> *Contributors:* Alan Dawrst, Almkglor, Altenmann, Andreas Kaufmann, Antaeus Feldspar, CRGreathouse, Carlsot, Cereblia, Chadmcaniel, Charles Matthews, Cybercobra, Dcoetzee, Devyni, Doradus, Dysprosia, Fredrik, Intgr, Jorge Stolfi, Jrockway, Kukolar, Laurienne Bell, Menahem.fuchs, MinorContributor, Mrwojo, Nkour, Naldoaran, Nsfmc, Pet3ris, Piet Delpot, Populus, R. S. Shaw, Rdhettinger, Rhanekom, Sanchom, Silly rabbit, Stevenj, Viebel, Wojciech mula, Xcez-be, Zr2d2, 128 anonymous edits

Jump list *Source:* <http://en.wikipedia.org/w/index.php?oldid=359036796> *Contributors:* Andreas Kaufmann, Cobi, JohnBlackburne, Ruud Koot, 1 anonymous edits

Self-organizing list *Source:* <http://en.wikipedia.org/w/index.php?oldid=291842784> *Contributors:* Aclwon, Andreas Kaufmann, JonHarder, PigFlu Oink, VTBassMatt, 4 anonymous edits

Binary tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359108897> *Contributors:* 7, ABCD, Aarslankhalid, Abu adam, Adam majewski, Ahoerstemeier, Ahy1, Airplaneman, Alex.vatchenko, Alienus, Allefant, Altenmann, Andreas Kaufmann, AndrewKepert, AnotherPerson2, Antaeus Feldspar, Aroundthewayboy, Ateeq.s, B4hand, Behrangsa, Bender2k14, Bhadani, BigDunc, BlckKngh, Bluebusy, Bobo192, Bojan1989, Brentsmith101, Brianga, Brucelee, Caltas, Cbraga, Charles Matthews, Chicodroid, Ck lostword, Classicaeleon, Cneplyr, Coelacan, Conversion script, Cybercobra, Cyhawk, David Eppstein, David Shay, David-Sarah Hopwood, Dcoetzee, Dguido, Djcollom, Dkasaki, Dominus, Dontdoit, Doug s, Dr. Sunglasses, Drano, DuaneBailey, Dysprosia, Ekeb, Encognito, Ferkelparade, Frankrod44, Frozendice, FvpD, Garyzx, Giftlite, Gilliam, Gimmetrow, Gsmodi, Happenstein, Heirpixel, IanS1967, Ilana, Itchy Archibald, Itman, Jafet, Jerome Charles Potts, Jonfore, Josephskeller, Jruderman, Justin W Smith, Kamath.nakul, Kamirao, Karagounis, Khlobin, Kgashok, Kgautam28, Kuru, LC, Liao, Liftarn, LightningDragon, Linas, LithiumBreather, Loisel, Loolo, Lotje, MONGO, Mark Renier, Martimp23, Matir, Maurice Carbonaro, Mcld, Mdnahas, Mhayes46, Michael Angelkovich, Michael Hardy, Michael Slone, Mike Christie, Minesweeper, Mjm1964, Mrwojo, Neomagic100, Nippoo, Naldoaran, Nonexistent, Oblivious, Oli Filth, Opelio, Orphic, Otterdam, ParticleMan, Pgk, Philip Trueman, Pit, Ppelleti, RG2, RadioFan2 (usurped), Rege, Reinderien, Rhanekom, Roreuqnoq, Rspeer, Rzelnik, SGBailey, Sapeur, Shentino, Shmomuffin, Shoujun, Smallpond, SmartGuy, Someone else, SpaceFlight89, Spottedowl, Taemyr, TakuyaMurata, Tarquin, Tarrahaticas, Vegpuff, Waggers, Wtarreau, Wælgæst wæfre, XJamRastafire, Xevior, Yuubinbako, Zero sharp, Zetawood, Zipdisc, Zvn, 311 anonymous edits

Binary search tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=360053986> *Contributors:* 4get, Abednigo, Abu adam, Adamuu, AgentSnoop, Ahy1, Alansohn, Alexsh, Allan McInnes, Andreas Kaufmann, Anoopjohnson, Awu, BAxelrod, Banaticus, Beetstra, Bernard François, Booabayzooka, Bryan Derksen, Butros, Calbaer, Capricorn42, Casted, Chery, Cochito, Conversion script, Cybercobra, D6, Damian Yerrick, Danadocus, Dcoetzee, DevastatorJIC, Dimechord, Djcollom, Doctordiehard, Doradus, Dysprosia, Dzikasosna, Ecb29, Enochlau, Evil Monkey, Ezhiki, Frankrod44, Fredrik, Func, GRHooked, Gaius Cornelius, Garoth, Giftlite, Glenn, Googl, Gorfy, GregorB, Grunt, Hadal, Ham Pastrami, Hathawaye, Havardk, Hyad, Ilana, Ixfd64, JForget, James pic, Jdm04, Jdurlamh, Jin, Jokers, Karl-Henner, Kate, Kewlito, Kragen, Kulp, LOL, Lanov, Liao, LilHelpa, LittleDan, Loren.wilton, Madhan virgo, MatrixFrog, Maximimax, Maximus Rex, Mb1000, MelareN212, MegaHasher, Metalmax, Mgjus, Michael Hardy, Michael Slone, Mikeputnam, Mindmatrix, Minesweeper, MrOllie, MrSomeone, Mrwojo, Mweber, Nerdgerl, Neurodivergent, Nils schmid hamburg, Nixdorf, Nux, Ohnoitsjamie, Oleg Alexandrov, Oli Filth, Oni Lukos, Onomou, Oskar Sigvardsson, P0nc, Phil Boswell, PhilipMW, Pion, Postdlf, Qiq, Gleem, Quuxplusone, Qwertys, Rdemar, Regnaron, Rhanekom, Richardj311, RoySmith, Rudo.Thomas, Ruud Koot, S3000, SPTWriter, Salrizvy, Shen, Shmomuffin, Sketch-The-Fox, Skier Dude, Spadgos, Spiff, Swapzy, Taejo, Taw, Tbvdm, The Parting Glass, TheMandarin, Theone256, Thesevenseas, Timwi, Tomt22, TrainUnderwater, Trevor Andersen, VKokielov, Vdm, Vectorpaladin13, Vocaro, WikHead, WikiWizard, Wtarreau, Wtmitchell, X1024, Xevior, ZeroOne, ۲۷۸ anonymous edits

Self-balancing binary search tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=347913160> *Contributors:* ABCD, ActuallyRationalThinker, Andreas Kaufmann, Angela, Baluba, Bluebusy, CRGreathouse, Chochopk, Cybercobra, DJ Clayworth, David Eppstein, Dcoetzee, Diberri, Drinloth, Dysprosia, Enochlau, Fredrik, Intgr, Jacob grace, Jacobolus, Jafet, Japanese Searobin, Jeltz, Jon Awbrey, Jorge Stolfi, Kdau, Lamro, Light current, Michael Hardy, Momet, Moskvax, MrDrBob, Neilc, Naldoaran, Pgan002, Plastikspork, Plyd, RJFJR, Ripe, Shlonif, Sriganeshs, SteveJothen, Wolfkeeper, 33 anonymous edits

Tree rotation *Source:* <http://en.wikipedia.org/w/index.php?oldid=352968869> *Contributors:* Abarry, Adamuu, Altenmann, Andreas Kaufmann, B4hand, BlckKngh, Boykobb, Castorvx, Chub, Conversion script, David Eppstein, Dcoetzee, Dysprosia, Hyperionred, Kjkjava, Leonard G., Mav, Michael Devore, Michael Hardy, Mr Bound, Mtanti, Neilc, Oleg Alexandrov, Pako, Ramasamy, SCriBu, Salvar, Skaraoke, Trainra, Vegasprof, Waylonflinn, Wizzar, Woblosch, Xevior, ۲۷۰ anonymous edits

Weight-balanced tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=327648690> *Contributors:* Andreas Kaufmann, Bhadani, Bobmath, Ciphergoth, DVD R W, JaGa, Jointds, KenE, Norm mit, Pascal.Tesson, Snarius, The former 134.250.72.176, 8 anonymous edits

Threaded binary tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=327764889> *Contributors:* A3 nm, Amicon, Andreas Kaufmann, Brad101, Edlee, Jaccos, Jagat sastry, Konstantin Veretennicov, MER-C, Michael Hardy, MoraSique, Mr2001, Pearle, Ppelleti, R. S. Shaw, Ragzouken, Sdenn, Themania, 13 anonymous edits

AVL tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359623415> *Contributors:* Adamuu, Aent, Agrawalyogesh, Alex Kapranoff, AlexGreat, Altenmann, Anant sogani, Andreas Kaufmann, Apanag, Astral, Auntoff, Binnacle, Bkil, Blackllotus, Bluebusy, Byrial, Castorvx, Caviare, ChrisMPI1, Conversion script, Cybercobra, Daewollama, Damian Yerrick, David.Mestel, Dcamp314, Dcoetzee, Docboat, Doradus, Drinloth, Dtribbbins, Dysprosia, Enviroboy, Euchiasmus, Evil Monkey, Flyingspuds, Fredrik, Gaius Cornelius, Geek84, Geoff55, Gnowor, Greenrd, Greg Tyler, Gruu, Gulllive, Gurch, Gökhan, II MusLiM HYBRID II, Iamnitin, Intgr, J.delany, Jeepday, Jeff02, Jennavecia, Jirkfa, JII, Joeyadams, Kain2396, Kdau, Kenyon, Kgasso, Kingpin13, Kjolk, Ksull10, Kukolar, LOL, Lankiveil, Larry V, Leuko, M, MarkHeily, MattY10, McKaysalsibus, Mellerho, Merovingian, Michael Hardy, Michael M Clarke, Michael miceli, Mikm, Minesweeper, Mjko, Mnogo, Mohammad ahad, Momet, Mr.Berna, Msanchez1978, Mtanti, Mzruya, NawlinWiki, Neilc, Nguyễn Hữ Dung, Nixdorf, Naldoaran, Nysin, Obradovic Goran, Ommyi-Pangaeus, Orimosenzon, Paul D. Anderson, Pavel Vozenilek, Pgan002, Pnorcks, Poor Yorick, RJFJR, Resper, Rockslave, ST47, Seyen, Shmomuffin, Smalljim, Srivesh, Ste4k, Tamfang, Tobias Bergemann, Toby Douglass, Tphyahoo, Tsemii, Tsoft, UnwashedMeme, Uw.Antony, Vektor330, Vlad.c.manea, Xevior, Yksyks, 261 anonymous edits

Red-black tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=358898693> *Contributors:* 206.176.2.xxx, Abu adam, Adamuu, Ahoerstemeier, Ahy1, AlanUS, Altenmann, Andreas Kaufmann, Aplusbi, Belfry, BenFrantzDale, Binnacle, Bioskope, Blackllotus, Blow, Bonadea, Bovineone, Brona, C. A. Russell, Cabubanna, Caviare, Churnett, Cjcollier, Connelly, Consed, Conversion script, Cybercobra, Cyp, David Eppstein, Dcoetzee, Drak, Dreamyshade, Drebs, Dysprosia, Dlugosz, Enochlau, Ephr123, Fawcett5, Fcp2007, Fraggle, Fredrik, Ghakko, Ghevwill, Giftlite, Gimbo13, Girafedata, H2gbob, Hao2lian, Hariva, Hawke666, Hermal, Hnn79, Humpback, IDdogbert, Iav, JMBucknall, Jaxl, Jenghel, Jleeped, Jodaw, Joshibschman, Jzcold, Karakak, Karl-Henner, Karlhendrikse, Kenyon, Kukolar, Laurier12, Leonard G., LiDaobing, Linuxrocks123, Loading, Lukax, Lunakeet, MatrixFrog, Maxis ftw, Mgrand, Michael Hardy, Mindmatrix, Minesweeper, Mnogo, N3bulous, Narlami, Nishantj, Notheruser, OMouse, Offliner, Pgan002, Pgk, Phil Boswell, Potatoswatter, Pqrstuv, RJFJR, Regnaron, Ripper234, Roleplayer, Ruud Koot, Ryan Stone, SGreen, SLi, Schellhammer, Sdenn, Sepreece, Sesquianual, Shizhao, SickTwist, Silly rabbit, Silverdirk, Simoncropp, Smangano, Smilindog2000, Sreeakshay, Stanislav Nowak, Stdzi, Stone628, Storables, Strebe, Supertigerman, Tbvdm, TheWarlock, Themania, Timwi, Toby Douglass, Tristero, Uniwalk, Urod, Versus22, Warut, Will2k, Xevior, Xiteer, Xmarios, Yakov1122, Zehntor, Zetawood, 261 anonymous edits

AA tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=358899993> *Contributors:* AliasXYZ, Ash211, Bporopat, Charles Matthews, Confuzzled, Cybercobra, Damian Yerrick, Darguz Parsilyan, Dcoetzee, Firsfron, Floodyberry, Gazpacho, Goochelaar, HyperQuantum, Koavf, Kukolar, MONGO, Mecki78, Mnogo, Nroets, Optikos, Ppelleti, Qutezuce, RJFJR, Rkleckner, RobertG, Stanislav Nowak, Triona, Why Not A Duck, 35 anonymous edits

Scapegoat tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=358901133> *Contributors:* AbsolutBildung, Andreas Kaufmann, Aplusbi, Cybercobra, Dcoetzee, Firsfron, Jarsyl, Joey Parrish, Kukolar, MegaHasher, Mnogo, Rich Farmbrough, Robert Ullmann, Ruakh, Sam Hocevar, Slike2, SteveJothen, Themania, Wknight94, 17 anonymous edits

Splay tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359308274> *Contributors:* Abu adam, Ahy1, Andreas Kaufmann, Anna Lincoln, Apankrat, Aplusbi, Atavi, Axlapre, Christopher Mahan, Coldzero1120, Conversion script, Cybercobra, CyborgTosser, Dcoetzee, Dodno, Drz, Dysprosia, Dlugosz, Foober, Fredrik, Freeside3, Fresheneesz, Gscshoyru, Gwern, HPRappaport, Hannan1212, Honza Záruba, Jamie King, Jwililia3, KConWiki, Karl Dickman, Kukolar, Ligulem, Lqs, Martlau, Mav, MegaHasher, Michael Hardy, MladenWiki, Mnogo, Momet, Nixdorf, Octahedron80, Offliner, Pako, Phil Boswell, Qutezuce, Rhanekom, Rich Farmbrough, Roman Munich, Romanc19s, Russelj9, Safek, Shmomuffin, Siyah.deepak, Silly rabbit, Silvonen, Snielfeld, Stephan Schulz, Tabletop, That Guy, From That Show!, Tjdw, Unyoyega, Urhixidur, VTBassMatt, Vecter, Versus, Wiml, Wj32, Wolfkeeper, Ybungabill, Yonkeltron, Zuphilip, 72 anonymous edits

T-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=358898567> *Contributors:* Acm, Andreas Kaufmann, Bryan Derksen, Charles Matthews, Cybercobra, Damian Yerrick, Dlugosz, Fuhghettaboutit, Gaius Cornelius, GreatWhiteNorthermer, Hholzgra, Hmendonca, Jamie Lokier, Jonah.harris, Kroepke, Mnogo, Paxcoder, Psychonaut, Ted nw, The Thing That Should Not Be, WriterHound, 18 anonymous edits

Rope *Source:* <http://en.wikipedia.org/w/index.php?oldid=335385891> *Contributors:* Almkglor, Andreas Kaufmann, ArtemGr, Chkno, Dcoetzee, Doradus, Emersoni, Fridolin, Jokes Free4Me, Mrwojo, Mstroeck, Pengo, Snaxe920, Spoon!, Vdm, 5 anonymous edits

Top Trees *Source:* <http://en.wikipedia.org/w/index.php?oldid=288103932> *Contributors:* Afrozenator, Andreas Kaufmann, Cobi, Cybercobra, Xezbeth, 3 anonymous edits

Tango Trees *Source:* <http://en.wikipedia.org/w/index.php?oldid=248971495> *Contributors:* Iridescent, Malcolma, Rayhe, Vecter, 1 anonymous edits

van Emde Boas tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=333313409> *Contributors:* Adrianwn, B4hand, BACbKA, Charles Matthews, Cybercobra, Cyhawk, David Cooke, Dbenbenn, Dcoetzee, Doradus, Fresheneesz, Fx4m, Gailcarmichael, Gulliveig, Jeff02, Kragen, Michael Hardy, Nickj, Patmorin, Phil Boswell, Piet Delport, Quuxplusone, Qwertyus, Svick, 8 anonymous edits

Cartesian tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359870041> *Contributors:* Andreas Kaufmann, Cobi, Cybercobra, David Eppstein, Gifflite, GrEp, Pichpitch, Tabletop, 6 anonymous edits

Treap *Source:* <http://en.wikipedia.org/w/index.php?oldid=356991976> *Contributors:* AHMartin, Andreas Kaufmann, Arbor, Benmq, C02134, Cdb273, Cshinyee, Cybercobra, David Eppstein, Eequor, Gustavb, Hans Adler, Hdante, Itai, James.nvc, Jleedev, Jogloran, Jsaxton86, Jörg Knappen, KnowledgeOfSelf, Kukolar, MegaHasher, Milkmandan, Miym, Oleg Alexandrov, Pfh, Poor Yorick, Qef, RainbowCrane, Regnaron, Ruud Koot, Saccade, Wsloand, 19 anonymous edits

B-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=358898386> *Contributors:* 128.139.197.xxx, ABCD, AaronSw, Abrech, Ahy1, Alaric, Alfalfahotshots, AlistairMcMillan, Altenmann, Altes, AlyM, Anakin101, Andreas Kaufmann, AnnaFrance, Antimatter15, Appoose, Aubrey Jaffer, BAxelrod, Battamer, Beeson, Bezenek, Bovineone, Bryan Derksen, Btwied, CanadianLinuxUser, Carbundle, Cbraga, Chadloeder, Charles Matthews, Chmod007, Ciphergoth, Ck lostsword, ConvityGoddess, Conversion script, Curps, Cutter, Cybercobra, Daev, Dcoetzee, Decrease789, Dlae, Dmn, Dpotter, Dravecky, Dysprosia, EEMIV, Ed g2s, Eddvella, Edward, Fabriciodosanjossilva, FatalError, Fgdafsdgfdsgfd, Flying Bishop, Fragglet, Fredrik, FrepypSpang, Fresheneesz, Gdr, Gifflite, Glrx, GoodPeriodGal, Ham Pastrami, Hao2lian, Hariva, Hbent, I do not exist, Inquisitus, JCLately, JWSchmidt, Jacosi, Jirkaf, Jjdawson7, Joahnnes, Joe07734, Jorge Stolfi, Jy00912345, Kate, Kinu, Knutux, Kovianyo, Kpjias, Kukolar, Lamdk, Lee J Haywood, Leibniz, Levin, Loadmaster, Luna Santin, MIT Trekkie, MachineRebel, Makkuuro, Malbrain, Merit 07, Mhss, Michael Angelkovich, Michael Hardy, Michal Nebyla, Mikeblas, Mindmatrix, Minesweeper, Mnogo, MoAgnome, MorgothX, Mrnaz, Mrwojo, Neile, Nishantjr, Norm mit, Oli Filth, Paushali, Peter bertok, Pgjan002, Postrach, Priyank bolia, PrologFan, Psyphen, Ptheoch, Quadrescence, Qutezuce, Qz, R. S. Shaw, Rich Farmbrough, Rp, Rpajares, Ruud Koot, Sandeep.a.v, Sandman@llgp.org, SickTwist, Simon04, Slady, Slake, Spiff, Ssboho, Stephan Leclercq, Ta bu shi da yu, The Fifth Horseman, Tjdw, Tobias Bergemann, Uday, Uw.Antony, Verbal, Wantnot, Wipe, Wkailey, Wolfkeeper, Wsloand, Wtmitchell, 296 anonymous edits

B+ tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=360216690> *Contributors:* Afaviram, Ahy1, Arny, Arrow, Ash211, Bluebusy, Boing! said Zebedee, Bornhj, Bovineone, Bryan Derksen, Capi, Cesarb, Cherezov, Cmdrjameson, ColinTempler, Cybercobra, Cychoi, Daniell, Decrease789, Dmn, DomQ, Eddie595, Encyclops, Eurleif, Fresheneesz, Gifflite, GregorB, Gurch, Happyvalley, Hongooi, I do not exist, Imachchu, Immunize, Inkling, Itmozart, James.walmsley, Jbalint, Josh Cherry, Jsnx, Julther, Jwang01, Kastayra, Kl4m, Knutties, LOL, Leibniz, Leksey, Leijohn, Lightst, LrdChaos, Luc4, MRLacey, Marc omorain, Matthew D Dillon, Max613, Mdmkolbe, Mfedyk, Mhss, MikeDierken, Mikeblas, Mogentiana, Mqchen, Mrcowden, Muro de Aguas, Neile, Nishantjr, Nqzero, Nuworld, Nyenye, Obradovic Goran, Oleg Alexandrov, OliviaGuest, Penumbra2000, PhilippWeissenbacher, Pinethicket, Priyank bolia, RHaworth, Raatikka, Scrool, SheffieldSteel, Snarius, Ste4k, TZOTZIY, TheProject, Thunderpenguin, Tim Starling, Tlesher, Tresiden, Tuxthepenguin933, Txomin, Unixguy, Vikreykjia, Yamla, Yellowstone6, Yungoe, Zenohockey, 185 anonymous edits

B*-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=342508692> *Contributors:* ATopt, AlistairMcMillan, Bovineone, Bryan Derksen, CesarB, Charles Matthews, Cybercobra, Fragglet, Gatte, Gdr, Goplat, Hotmop, Inkling, Intgr, Isaac Dupree, Jamelan, Jodawi, Kanenas, Nick, Oleg Alexandrov, Pvshnik, Qutezuce, Svick, Wiml, Yaml, 10 anonymous edits

B sharp tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=357644587> *Contributors:* ATopt, CapitalR, I do not exist, Loadmaster, Qutezuce, RJFJR, Rich Farmbrough, Swpb, 5 anonymous edits

Dancing tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=326411909> *Contributors:* 16@r, AlistairMcMillan, Andreas Kaufmann, ArnoldReinhold, Audriusa, Bsiegel, Computer Guru, Cybercobra, DanielCD, Dlugosz, FrenchIsAwesome, Inkling, Koody, PeterSymonds, Qutezuce, Rjwilmsi, Ronocdh, Royboycrashfan, TheParanoidOne, 10 anonymous edits

2-3 tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=357779608> *Contributors:* ABF, Altenmann, Apanag, Asenine, Awotter, Chrismiceli, Curtisf14, Cybercobra, DGaw, Japo, Jodawi, Slady, Utcursch, V2desco, Wsloand, 26 anonymous edits

2-3-4 tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=352676792> *Contributors:* ABCD, AceDevil, Alfalfahotshots, Andmatt, Andreas Kaufmann, Aopf, Apham, Ashdurbat, Chrismiceli, Cmuhlenk, Cybercobra, DHN, David Eppstein, Dfletter, Drbreznjev, Erickstark, Ghakko, GromXXVII, Heida Maria, HostZ, Jengelh, Jodawi, Jogloran, Jvendries, Michael Devore, Nishantjr, Oli Filth, OliD, Paul Ebermann, Ruud Koot, Schellhammer, Slady, Talrias, Terronis, Twthmoses, 50 anonymous edits

Queaps *Source:* <http://en.wikipedia.org/w/index.php?oldid=353944961> *Contributors:* Malcolmra, Parruda, Rettetast, 4 anonymous edits

Fusion tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=347689806> *Contributors:* CesarB, Charles Matthews, Cybercobra, Dcoetzee, Decoratrix, Lamro, Oleg Alexandrov, ZeroOne, 1 anonymous edits

Bx-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=334892467> *Contributors:* Bxtree, Cheesewire, Chensu, Cybercobra, GregorB, Rjwilmsi, 9 anonymous edits

Heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=357705178> *Contributors:* Abdull, Abu adam, Ahy1, Alwar.sumit, AncientPC, Andreas Kaufmann, Apokrif, Aragorn2, Ascánder, B4hand, B7j0c, Bender2k14, Bluebusy, Boleslav Bobcik, Bradgib, C7protal, Charles Matthews, Chipchap, Christian List, Cobi, Conversion script, Cool 1 love, Cybercobra, Da monster under your bed, DarkPhoenix, David-Sarah Hopwood, DavidCary, DavidHarkness, Dcoetzee, Derbth, Derek Ross, Doradus, DrBob, Drpaula, Dysprosia, Flamholz, Frecklefoot, Fredrik, Fresheneesz, Garyzx, Gruntler, Ham Pastrami, Ilyathemuromets, ImPerfection, Iron Wallaby, Itmozart, Jafet, Jfroelich, Jimbleak, Jirkaf, Jll, Jmmbatista, Jogloran, Jurohi, Ketil3, Kibiru, Kragen, Krun, Kukolar, LC, LOL, LeoNerd, Loadmaster, Mahanga, Mark Renier, Mdd, Mellum, Mloun, Mpkr, NawlinWiki, Noldoaran, Notheruser, Oleg Alexandrov, Oli Filth, Oskar Sigvardsson, OverLeg, Paddu, Pgjan002, Pinguin.tk, Prasi90, Prophile, Psyphen, Public Menace, Qutezuce, Quuxplusone, Reedy, Rhanekom, Ruud Koot, SabbeRubbish, Sbisolo, Schellhammer, Shizhao, Silly rabbit, Sligocki, Spoon!, Suruena, Tgdwyer, Thumperward, Timwi, Tom W.M., Vecter, Vikingstad, Wiesmann, Wikitamino, Wsloand, Xcez-be, Yuval madar, 102 anonymous edits

Binary heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=356996257> *Contributors:* Altenmann, Amossin, Andreas Kaufmann, Antaeus Feldspar, B4hand, Bluebusy, Brighterorange, Bryanharris, Bsdlogical, CiaPan, Codetiger, Cpflames, DHN, Danielcer, David Eppstein, Dcoetzee, Derek Ross, Djcmackay, Doug Bell, Dysprosia, Foober, Fredrik, Fresheneesz, Htonl, Hydrox, Ilmari Karonen, Indy256, Inquisitus, Iron Wallaby, Jaded-view, Japo, Kb, Kenyon, Kragen, Kukolar, Kyokpae, LOL, Laurens, Levin, Liao, Loftpo, Lourakis, Mahanga, Matt77, Mdouze, MonoNexo, Nixdorf, Notheruser, Nuttycoconut, Oli Filth, Pdelong, Pit, Platyk, R27182818, Rich Farmbrough, Ruud Koot, Sam Hocevar, Schellhammer, Scott tucker, Seshu pv, Shd, Sladen, Snoyes, Taw, Tdeoras, Theone256, VKokielov, Vektor330, Velle, Vikingstad, Wsloand, 92 anonymous edits

Binomial heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=355183139> *Contributors:* Aham1234, Arthena, Biscuittin, Brona, Cdang, CiaPan, Creidieki, Dcoetzee, Doradus, Dysprosia, Fragglet, Fredrik, Fresheneesz, Googl, Hairy Dude, Karlheg, LOL, Lemontea, MarkSweep, Martin TB, Matt.smart, Maximus Rex, Michael Hardy, NeonMerlin, Npansare, OOo.Rax, Peterwhy, Poor Yorick, Qwertyus, Reb42, Sapeur, Stebulus, Theone256, TonyW, Vecter, Vmanor, Wsloand, 37 anonymous edits

Fibonacci heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=354805524> *Contributors:* Aaron Rotenberg, Adam Zivner, AnnedeKoning, Antiuser, Aquiel, Arjun024, Arkitus, Bporopat, Brona, Charles Matthews, Creidieki, David Eppstein, Dcoetzee, DekuDekuplex, DerHexer, Dysprosia, Fredrik, Fresheneesz, Gimmetrow, Hao2lian, Japanese Searobin, Jirkaf, LOL, Lars Washington, Michael Hardy, Mild Bill Hiccup, Miym, Mkorpela, MorganGreen, Offliner, P0nc, Poor Yorick, Qutezuce, Ravik, Ruud Koot, Safenner1, The Real Marauder, Thw1309, Wik, Wsloand, Zeno Gantner, 71 anonymous edits

2-3 heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=236199484> *Contributors:* ABCD, Bluemask, Charles Matthews, Dr. Weazel, Fredrik, PerryTachett, RainbowOfLight, Ruud Koot, Wsloand, 5 anonymous edits

Pairing heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=359243559> *Contributors:* Bredawson, David Eppstein, Drdisque, Geoffrey.foster, Hoofinasia, Ruud Koot, SAMJAM, Sneftel, Tgdwyer, Wsloand, 3 anonymous edits

Beap *Source:* <http://en.wikipedia.org/w/index.php?oldid=221204252> *Contributors:* Charles Matthews, Doradus, Hu, Ilyathemuromets, Rjwilmsi, Ruud Koot, Wsloand, 2 anonymous edits

Leftist tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359121204> *Contributors:* Akurat, Buss, Cmdrjameson, Computergeeksjw, Fresheneesz, Hathawayc, Mahanga, Npansare, Ruud Koot, Virtualphnt, WillNess, X1987x, 18 anonymous edits

Skew heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=348900743> *Contributors:* Fuhghettaboutit, Leduytien, Miym, Quinntaylor, Ruud Koot, SabbeRubbish, Wsloand, 6 anonymous edits

Soft heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=315938487> *Contributors:* Agthorr, Bender2k14, Bondegezou, Cobi, Dcoetzee, Doradus, Fredrik, Just Another Dan, Ruud Koot, Wsloand, 9 anonymous edits

Ternary heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=357655204> *Contributors:* BahaFura, Derek Ross, Destynova, G8ken, Nathanjd, 6 anonymous edits

D-ary heap *Source:* <http://en.wikipedia.org/w/index.php?oldid=332003056> *Contributors:* Derek Ross, Freshenesz, Greenrd, JanniePieters, LeaW, M2Ys4U, Miyagawa, Shalom Yechiel, Skier Dude, Slemm, 7 anonymous edits

trie *Source:* <http://en.wikipedia.org/w/index.php?oldid=358218958> *Contributors:* 97198, Altenmann, Andreas Kaufmann, Antaeus Feldspar, Anupchowdary, Blahedo, Booyabazooka, Bryan Derksen, Coding.mike, Conversion script, Cowgod14, Cybercobra, Danielx, Danny Rathjens, Dantheon, David Eppstein, Dbenbenn, Dcoetzee, Deineka, Denshade, Diomidis Spinellis, Doradus, Drpaule, Dscholte, Dysprosia, EatMyShortz, Edlee, Electrum, Enrique.benimeli, Eug, Francis Tyers, Fredrik, FuzziusMaximus, Gaius Cornelius, Gdr, Gerbrant, GeypycGn, Giftlite, Gmaxwell, Ham Pastrami, Honza Záruba, Hugewolf, Jim baker, Johnny Zoo, JustinWick, Jyoti.mickey, KMeyer, Kate, Kwamikagami, Leaflord, LiDaobing, Loreto, Matt Gies, Micahcowan, Mostafa.vafii, MrOllie, Nad, Neurodivergent, Nosbig, Otus, Piet Delpot, Pombredanne, Qwertyus, Raanoo, Rjwilmsi, Rl, Runtime, Sergio1, Shoujun, Simetrical, Slamb, Sperxios, Stillnotlef, Superm401, Svick, TMott, Taral, Teacup, Tobias Bergemann, Tr00st, Watcher, Wolfkeeper, Yaframa, 許仁毅, 136 anonymous edits

Radix tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359608831> *Contributors:* AaronSw, Adoniscik, Ahy1, Andreas Kaufmann, Bhyde, Brim, Burkeaj, CesarB, Cobi, Coffee2theorems, Cwitty, Cybercobra, DBeyer, DavidDecotigny, Dcoetzee, Dogcow, Drachmae, Edward, Gulliveig, Gwalla, Hesamwls, Hetori, Itman, Jamelan, Jy00912345, MegaHasher, Modify, Morteju, Nausher, Noosphere, Piet Delpot, Qutezue, Rgruijan, Rocchini, Safek, Sameemir, SparsityProblem, 64 anonymous edits

Suffix tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=356362240> *Contributors:* 12hugo34, Alfio, Andreas Kaufmann, AxelBoldt, Bbi5291, Bcat, Beetstra, Blahma, Charles Matthews, Christian Kreibich, CobaltBlue, Cybercobra, David Eppstein, Dcoetzee, Delirium, Deselaers, Dionyziz, Dmitriy V, Ffaarr, Garyzx, Giftlite, Illya.havsiyevych, Jamelan, Jemfinch, Jhclark, Jogloran, Johnbibby, Kh3rd, Leafman, Luismsgomes, MaxEnt, Mechonbarsa, NVar, Neajc, Nils Grimsmo, Nux, Oleg Alexandrov, P0nc, R. S. Shaw, Requestion, RomanPszonka, Ru.spider, Safek, Sho Uemura, Shoujun, Sky Attacker, Squash, Sundar, TheMandarin, TheTaxman, TripleF, Vector, Wsloand, Xevior, Xutaodeng, 56 anonymous edits

Suffix array *Source:* <http://en.wikipedia.org/w/index.php?oldid=350481289> *Contributors:* Andreas Kaufmann, Arnabdortg, BenRG, Bkl, Chris83, Cobi, EchoBlaze94, Gailcarmichael, Gaius Cornelius, Garyzx, Giftlite, Jwarhol, Karol Langner, Kiwibird, Malbrain, Mboverload, MeltBanana, Nils Grimsmo, Nroets, Singleheart, Tobias Bergemann, TripleF, Viksit, Wolfgang-gerlach, 39 anonymous edits

Compressed suffix array *Source:* <http://en.wikipedia.org/w/index.php?oldid=343750366> *Contributors:* Andreas Kaufmann, RHaworth, Stringologist

FM-index *Source:* <http://en.wikipedia.org/w/index.php?oldid=327138736> *Contributors:* Andreas Kaufmann, Fingerz, Raven1977, 2 anonymous edits

Generalised suffix tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=342777990> *Contributors:* Aditya, Andreas Kaufmann, Bbi5291, Bernhard Bauer, Dcoetzee, Giftlite, JanCeuleers, Joey-das-WBF, Malik Shabazz, Michael Hardy, Nils Grimsmo, Wsloand, 16 anonymous edits

B-trie *Source:* <http://en.wikipedia.org/w/index.php?oldid=309764300> *Contributors:* IForTheMoney, Andreas Kaufmann, Debackerl, Hetori, Merosonox, MindstormsKid, R'n'B, 1 anonymous edits

Judy array *Source:* <http://en.wikipedia.org/w/index.php?oldid=349302346> *Contributors:* Alynnna Kasmira, Andreas Kaufmann, Cowgod14, Dcoetzee, Djmckee1, Doug Bell, EIFY, Ejrh, Freshenesz, Garyzx, Gmaxwell, Mellery, Minghong, Nigosh, RainbowCrane, Rolingyid, The Anome, 11 anonymous edits

Directed acyclic word graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=357112504> *Contributors:* Andreas Kaufmann, Archie172, Balrog-kun, Bkonrad, Bokeh.Senssei, Chimz, Chkno, Damian Yerrick, David Eppstein, Gwernol, JonHarder, MC10, Norman Ramsey, Radagast83, Rl, Rofl, Thelb4, Watcher, 19 anonymous edits

Ternary search tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=350014691> *Contributors:* Antaeus Feldspar, Arkanosis, Booyabazooka, Cwolvesheep, Cybercobra, Jds, Klapautius, Maghnuis, Magioladitis, Potatoswatter, Raanoo, Richardj311, Selket, Shadypalm88, SkyWalker, 7 anonymous edits

And-or tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=328992086> *Contributors:* Addshore, Classicaelecon, Fratrep, Garion96, H taammoli, Hooperbloob, Intgr, Jamjam337, Jpbowen, Justmeherenow, Logperson, Nfwu, Pete.Hurd, Routa-olio, Rwww, Ryan Roos, 3 anonymous edits

(a,b)-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=288107241> *Contributors:* AlkyAlky, Bovineone, Cacadril, Centrx, Cybercobra, Gardar Rurak, JonHarder, Linas, Lockley, MER-C, Nick, Onodevo, Sapeur, Skapur, Skysmith, 2 anonymous edits

Link/cut tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=303194528> *Contributors:* Giftlite, GregorB, Martlau, Rkleckner, 5 anonymous edits

SPQR tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=350417273> *Contributors:* Andreas Kaufmann, Auntof6, Cybercobra, David Eppstein, Eng 104*, Harrigan, Kintaro, MaxEnt, Riedel, Rjwilmsi, Twri, 1 anonymous edits

Spaghetti stack *Source:* <http://en.wikipedia.org/w/index.php?oldid=358434566> *Contributors:* Alf, Antandrus, Congruence, Dcoetzee, Dreftymac, Edward, EmptyString, Eric119, Eyal0, Grenavitar, It's-is-not-a-genitive, Karada, Lindsey Kuper, Masharabinovich, Pitchpitch, Ruud Koot, Skaller, Somercet, Tagishsimon, The Anome, TimBentley, Timwi, Tyc20, Woohoo kitty, 17 anonymous edits

Disjoint-set data structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=355496304> *Contributors:* Adrianwn, Alksentr, Andreas Kaufmann, Archimerged, Backpackadam, Bigaln2, Bkkbrad, Boydski, Charles Matthews, Chipchap, Clodyle, David Eppstein, Dcoetzee, Deewiant, Dominus, Edward Vielmetti, Freshenesz, Gfonsecab, Grendelhan, IanLiu, Intgr, Isaac Dupree, Jamelan, Jonel, Kasei-jin, Kevtrice, LOL, Lambiam, LittleDan, MasterAchilles, Msh210, NawlinWiki, Nikastr, Nyenyc, Oaf2, Oli Filth, Pakaran, Qutezue, Qwertyus, Rbrewer42, ReyBrujo, Ripper234, Salix alba, Sceptre, Sean22190, Shmilymann, Spike Wilbury, Spirko, Stellmach, Superjoe30, Tevildo, The Anome, Williacb, 58 anonymous edits

Space partitioning *Source:* <http://en.wikipedia.org/w/index.php?oldid=340903198> *Contributors:* Altenmann, Arminahmady, Bernhard Bauer, CBM, Flyinglemon, Frostyandy2k, Glen, Kjmathew, M-le-mot-dit, Michael Hardy, Nlu, Oleg Alexandrov, PaulJones, Reedbeta, Ruud Koot, Timrb, Yumf, 8 anonymous edits

Binary space partitioning *Source:* <http://en.wikipedia.org/w/index.php?oldid=359160150> *Contributors:* Abdull, Altenmann, Amanaplanacanalpanama, Amritchowdhury, Angela, Ariadie, B4hand, Brucenaylor, Bryan Derksen, Cbraga, Chan siuman, Charles Matthews, CyberSkull, Cybercobra, DanielPharos, Dcoetzee, Dysprosia, Fredrik, Frenchigh, GregorB, Immibis, Jafet, Jamesontai, Kdau, Kelvie, KnightRider, LOL, LogiNevermore, M-le-mot-dit, Mdob, Michael Hardy, Mild Bill Hiccup, Noxin911, NuclearFriend, Obiwhonn, Oleg Alexandrov, Operator link, Palmin, Percivall, Prikikipedia, QuasarTE, Reedbeta, Spodi, Stephan Leeds, Tabletop, Tarquin, Twri, Wiki alf, WikiLaurent, Wmahan, Wonghang, Yar Kramer, Zetawoof, 49 anonymous edits

Segment tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=360199288> *Contributors:* AdultSwim, Alfredo J. Herrera Lago, Csernica, Cybercobra, Dean p foster, Dhruvbird, Diomidis Spinellis, Dstrash, Jj137, Mikhail Dvorkin, Optigan13, Portalian, Rahmtin rotabi, X7q, XLerate, Yintianjiao, 15 anonymous edits

Interval tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359201649> *Contributors:* Breuwi, Cobi, CommonsDelinker, Cosmoskramer, Da monster under your bed, David Eppstein, Dcoetzee, Dean p foster, Digulla, Favonian, Jamelan, Kungi01, Mkcmke, Mloskot, Porao, RJFJR, Raboof, Raduberinde, Rodgling, Svick, X7q, Zhybear, 31 anonymous edits

Range tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=360199894> *Contributors:* Breuwi, Caesura, Cybercobra, Dcoetzee, Infvwl, RJFJR, Sanchom, Smalljim, 10 anonymous edits

Bin *Source:* <http://en.wikipedia.org/w/index.php?oldid=333053281> *Contributors:* Agent007ravi, AndrewHowse, Andycjp, Lifebaka, Michael Hardy, Ospalh, Yumf, 4 anonymous edits

kd-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359484623> *Contributors:* Achurch, Algomaster, Altenmann, Amit man, Andreas Kaufmann, Angus Lepper, Arkitus, Bajejohannes, BenFrantzDale, Borax, Braddodson, Brandonwood, Bryan Derksen, Bytnar, C. lorenz, Casbah, CesarB, Connelly, Cybercobra, David Eppstein, Dcoetzee, Don Reba, EQ5afN2M, Equendil, Favonian, Ficuep, Formerly the IP-Address 24.22.227.53, Genieser, Giftlite, Grendelhan, Ilyathemuromets, Jokes Free4Me, Karlhendrikse, KiwiSunset, MYguel, Madduck, Mcld, Miym, MladenWiki, MusicScience, Neile, NikolasCo, Nintendude, Reedbeta, Revpj, Riitoken, Rjwilmsi, Ruud Koot, SJP, Skinkie, SteinbDJ, Svick, Szabolcs Nagy, The Thing That Should Not Be, TheParanoidOne, Ts3r0, Uranor, User A1, Waldir, Woohoo kitty, Wsloand, Wwheeler, Yumf, 118 anonymous edits

Implicit kd-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=345196923> *Contributors:* Andreas Kaufmann, Cybercobra, Genieser, Iridescent, Karlhendrikse, Rjwilmsi, Rockfang, Tesi1700, Tim Q. Wells, 2 anonymous edits

min/max kd-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=345197156> *Contributors:* Andreas Kaufmann, Genieser, Jokes Free4Me, Rockfang, Signalhead, 3 anonymous edits

Adaptive k-d tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=95886887> *Contributors:* Onodevo, Qutezuce, Wsloand

Quadtree *Source:* <http://en.wikipedia.org/w/index.php?oldid=347073110> *Contributors:* ALoopingIcon, Aburad, Aeturnus, AllanBz, Arthena, Backpackadam, Bryan Derksen, Cdc, Cuaxdon, Cybercobra, Danny, David Eppstein, Dcoetze, Dgreenheck, Dysprosia, Enochlau, Fredrik, Gaius Cornelius, GeorgeMoney, Gimmetrow, Happyvalley, Hermann.tropf, Interiot, Johnflux, Lancekt, Maglev2, Miko3k, Nickblack, Oleg Alexandrov, Peterjanbroomans, Piet Delport, RHaden, RJFJR, Ronz, Shencoop, Spottedowl, TechnologyTrial, Tomo, Tony1212, Wojciech mula, Wsloand, Yrithind, 39 anonymous edits

Octree *Source:* <http://en.wikipedia.org/w/index.php?oldid=352424138> *Contributors:* 23u982j32r29j92, Alienskull, Alksub, Andreas Kaufmann, Arthena, AySz88, Balabiot, Bkkbrad, Bryan Derksen, Cesarc, Claystu, Cybercobra, DragonflySixtyseven, Dysprosia, Eike Welk, Exe, Ffaar, Fredrik, JaGa, Jacob grace, June8th, Kierano, Litherum, MIT Trekkie, MagiMaster, Melena, Noerrorsfound, Nomis80, Nothings, Olsonse, Rahul220, Ruud Koot, Sadangel, Scott Paeth, Scott5114, SimonP, Sscomp2004, TechnologyTrial, Viriditas, WhiteTimberwolf, Wsloand, 67 anonymous edits

Linear octrees *Source:* <http://en.wikipedia.org/w/index.php?oldid=222531225> *Contributors:* Andreas Kaufmann, Leontios, Pegship, Rahul220, 1 anonymous edits

Z-order *Source:* <http://en.wikipedia.org/w/index.php?oldid=359404721> *Contributors:* Andreas Kaufmann, BenFrantzDale, Black Falcon, Bluap, CesarB, David Eppstein, DnetSvg, Edratzer, Einstein9073, Ephraim33, Fisherjs, Giftite, Hermann.tropf, Hesperian, Kku, Lambiam, Lendorien, Magioladitis, Paul Foxworthy, Robertd, VivaEmilyDavies, Ynhockey, Zotel, 14 anonymous edits

UB-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=309499977> *Contributors:* Andreas Kaufmann, Cybercobra, Hermann.tropf, Honnza, Rich Farmbrough, Svick, Travelbird, Welsh, 5 anonymous edits

R-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359974143> *Contributors:* AAA!, Alejos, AllanBz, Altenmann, Alynnna Kasmira, Avono, Bernhard Bauer, CALR, Curps, Cutelyaware, Cybercobra, Elwood j blues, EncMstr, FatalError, Foobaz, Freekh, G.hartig, Gwern, Hadal, Happyvalley, Huisho, Hydrogen Iodide, Imbcmthd, Jason.surratt, Jodawi, MIT Trekkie, MacTed, Minghong, Nick Pisarro, Jr., Oik, Okoky, Peterjanbroomans, Pqrstuv, Qutezuce, Radim Baca, Skinkie, Soumyasch, Sperrios, Stolee, The Anome, Tony1212, Twri, Ziyuang, 65 anonymous edits

R+ tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=353698227> *Contributors:* Alksentrs, AllanBz, Bluap, Brian0918, CesarB, Creidieki, Fuzzie, Happyvalley, J.pellicoli, Jodawi, Kerowyn, Minghong, NikolasCo, Nintendude, Pearle, RJFJR, Tim Starling, Turnstep, Westquote, 14 anonymous edits

R* tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359974617> *Contributors:* Frouaix, Happyvalley, Jodawi, Minghong, Neilc, The Anome, Virtuald, 15 anonymous edits

Hilbert R-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=354258341> *Contributors:* Andreas Kaufmann, BD2412, D6, Damsgård, Erechtheus, Hammertime, JCarlos, Jodawi, Michael Hardy, Mwtwoes, Niczar, Niteris, Okoky, Robin K. Albrecht, SimonD, Twri, 26 anonymous edits

X-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=359738783> *Contributors:* Cybercobra, Danceswithzerglings, Drumguy8800, Hyperfusion, Mild Bill Hiccup, 5 anonymous edits

Metric tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=289951463> *Contributors:* Brighterorange, Bryan Derksen, Cybercobra, Itman, Nsk92, Staffelde, 4 anonymous edits

VP-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=333666162> *Contributors:* Bryan Derksen, Cybercobra, Itman, Mcld, Piet Delport, Sbjesse, User A1, 3 anonymous edits

BK-tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=348534082> *Contributors:* Bryan Derksen, Cybercobra, Endpoint, Fikus, Itman, Piet Delport, Quuxplusone, Staffelde, Volkan YAZICI, Wossi, 6 anonymous edits

Hash table *Source:* <http://en.wikipedia.org/w/index.php?oldid=359641766> *Contributors:* ASchmoo, AdamRetchless, Adrianwn, Ahoerstemeier, Ahy1, Ajo Mama, Akuchling, Alksub, Allstarecho, Altenmann, Andreas Kaufmann, Antaeus Feldspar, Antaeus Feldspar, Anthony Appleyard, Anurmi, Apanag, Arbaest Mike, Arpitm, Ashwin, AxelBoldt, Baka toroi, BenFrantzDale, Bevo, BlckKnight, Bobo192, Bug, C4chanu, CRGreathouse, CanadianLinuxUser, CanisRufus, Cesarc's unprivileged account, Cfайлde, Cgma, Ched Davis, Cic, Cobi, Cometstyles, Conversion script, Crike, CryptoDerk, Cryptoid, Cybercobra, DNewhall, DSatz, Damian Yerrick, David Eppstein, Davidfstr, Davidgothberg, Dcoetze, Decltype, Derek Parnell, Derek farn, Deshraj, Digwuren, Dimcomer, Doug Bell, Drbreznjev, Drinloth, Dysprosia, Ecb29, Edof13, EncMstr, Esb, Everyking, Filu, Floodyberry, Fragglet, Frecklefoot, Fredrik, Frehley, Freshenees, Furykef, Giftite, Glrx, GregorB, Gremel123, Gulliveig, Gustavb, HFuruseth, Helios2k6, Helix84, Hetori, Hosamaly, Iekpo, Imran, Ingr, Iron Wallaby, Jjuruan, JLaTondre, Jorge Stolfi, Josephskeller, JustinWick, Karuthedam, Kaustuv, Kbndam71, Kbrose, Khalid, Kinu, Knutux, Krakhan, Kungfuadam, LOL, LapoLuchini, Larry V, Lavenderbunny, Linguita, Luqui, Luxem, MER-C, Magnus Bakken, MaxEnt, Mayrel, Mdd, MegaHasher, Meneth, Michael Hardy, Mike.aizatksy, Miles, Mipadi, Mousehousemd, Mrwojo, Nanshu, Narendrik, Neile, Nethgirb, Neurodivergent, Nightkaos, Nixdorf, Nneonneo, Not-just-yeti, Nuno Tavares, ObfuscatePenguin, Om Sao, Omegatron, Oravec, Ouishiobean, Pagh, Pakaran, Peter Horn, Piccolomomo, Pichpich, PizzaMargherita, QrczakMK, R3m0t, Radagast83, Raph Levien, Rgamble, Rjwilmsi, Sam Hocevar, Sandos, Scandum, Schwarzbichler, Sebleblanc, Secretlondon, Seizethedave, Shafgoldwasser, Shanes, Shuchung, Simonsarris, SiobhanHansa, Sleske, Sligocki, Sonjaa, Stannered, T Long, TShilo12, Tackline, TakuyaMurata, Teacup, Th1rt3en, The Anome, TheTraveler3, Thermon, Ticklemepink42, Tikiwont, Tjdw, Tomchiuke, Tostie14, Triwbe, UtterSRG, Varuna, Velociostrich, Waltpohl, Watcher, Wernher, Wikilolo, Winecellar, Wmahan, Wolfkeeper, Woohooikit, Wrp103, Zundark, 355 anonymous edits

Hash function *Source:* <http://en.wikipedia.org/w/index.php?oldid=360084132> *Contributors:* 10metreh, Adolphus79, Agent X2, Alexcollins, Altenmann, Andipi, Andrei Stroe, Antaeus Feldspar, Apoc2400, Appleseed, Appirlpn, Atlant, BlckKnight, Bomac, Boredzo, Bradjamesbrown, BrianRice, Brona, Capricorn42, Cesarc's unprivileged account, Chalst, Chuck Simmons, Claygate, Coffee2theorems, Connelly, Conversion script, Cuddlyable3, CyberSkull, Damian Yerrick, David spector, Davidgothberg, Dcoetze, Demosta, Derek Ross, Dfinkel, Dkf11, Doradus, Dranorter, Drostie, Dwheeler, EddEdmondson, Eisnel, Emc2, EngineerScotty, Epbr123, Erebus Morgaine, Eurosong, Evand, Everyking, Fender123, Folic Acid, Folkor, Francis Tyers, FusionNow, GVOPLIT, Garyyx, Ggia, Giftite, Gulliveig, Gurch, Hamsterlopiethecus, Harmil, Helix84, Hfaestedge, Iamhigh, InShanee, Ingr, Ixf6d4, J-Wiki, Jdelanoy, Jachto, JakeD409, Jbalint, Jc, Jeaise, Jediknil, Jeltz, JesseStone, Jfroelich, Jopxton, Jorge Stolfi, KHAaaaaaaaAN, Karl-Henner, Karuthedam, Kazvorpal, Keith Cascio, KelSYC, Kgfeischmann, Kirrages, Knutux, Kotha arun2005, KrizbyB, Kurivaima, Kuru, Kusunose, L337 kyblndmr, LOL, Lambiam, Leedeth, Linas, Ljl., M2millenium, MIT Trekkie, Malycatenar, Mark Wegman, MathStuf, Matt Crypto, Maurice Carbonaro, Mdebets, MegaHasher, Mgghtg, Michael Hardy, Midnightcomm, Miguel, Mikeblas, Mild Bill Hiccup, Mmmready, Mnbf9rca, Ms2ger, Nanshu, Nguyen Thanh Quang, Ninly, Noldoaran, Nyttend, Obli, Octahedron80, Oli Filth, OverlordQ, Pabix, Patelm, Paul Richter, Paulschou, Peter bertok, Peyre, Pfunk42, Phil Spectre, PhilHibbs, PierreAbbat, Pndfam05, Powerdes, Pseudomonas, QmunkE, Quota, Qwertys, Raghaw, RattusMaximus, Rich Farmbrough, Rjwilmsi, SamHartman, SavantEdge, Schmitelye, ShakingSpirit, ShawnVW, Shingra, Sjones23, Sklender, Snoops, Stangaa, StealthFox, Stevelothen, Svnsvn, Tachyon01, Talrias, Taw, That Guy, From That Show!, TomViza, Tonysan, Toolan, TruthIIIPower, Twri, Ultimus, Updatehelper, Utcursh, Vanis, Vstarre, Watcher, WereSpielChequers, Whi Trillium, Wikilolo, Wjaguar, Wolfkeeper, Ww, Xelgen, Zeno of Elea, शौ. 347 anonymous edits

Open addressing *Source:* <http://en.wikipedia.org/w/index.php?oldid=359641902> *Contributors:* Cobi, DataWraith, Freshenees, Gildos, LOL, O keyes, Skwa, 5 anonymous edits

Lazy deletion *Source:* <http://en.wikipedia.org/w/index.php?oldid=351047115> *Contributors:* A Generic User, Alksub, Freshenees, The Doink, 6 anonymous edits

Linear probing *Source:* <http://en.wikipedia.org/w/index.php?oldid=288261822> *Contributors:* Andreas Kaufmann, C. A. Russell, Cesarc's unprivileged account, Enochlau, Gazpacho, Jngnyc, JonHarder, Linas, Negruilio, OliviaGuest, RJFJR, Sbluen, SpuriousQ, Tas50, Themania, 24 anonymous edits

Quadratic probing *Source:* <http://en.wikipedia.org/w/index.php?oldid=343543713> *Contributors:* Andreas Kaufmann, Bavla, C. A. Russell, Cesarc's unprivileged account, Cybercobra, David Eppstein, Dcoetze, Enochlau, Jdanz, Kmgrparyush, Mikeblas, Oleg Alexandrov, Philip Trueman, Robertvan1, Ryk, Simeon, ZeroOne, 30 anonymous edits

Double hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=338216359> *Contributors:* Amiceli, Angela, AxelBoldt, CesarB, Cesarc's unprivileged account, Cobblet, DasBrose, Dcoetze, Gurch, Hashbrowncipher, Imposing, JForget, Only2sea, Pfunk42, RJFJR, Usrmme h8er, Zawersh, 41 anonymous edits

Cuckoo hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=344245441> *Contributors:* Arvindh, CRGreathouse, Cesarc's unprivileged account, Cwolfsheep, Cybercobra, Dcoetze, Ej, Hermel, Hetori, Jafet, LiranKatzir, Lmnonson26, Mandyhan, Mark cummins, Neile, Nyh, Pagh, S3000, Themania, Thumperward, Valentas.Kurauskas, Wjaguar, Zawersh, Zr2d2, 32 anonymous edits

Coalesced hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=339004477> *Contributors:* Algotime, Andreas Kaufmann, Basawala, Cesarc's unprivileged account, Cic, Confuzzled, Dcoetze, Freshenees, Ian Pitchford, Jafet, Jll, Oleg Alexandrov, Pmdboi, Tassedeth, Zawersh, 8 anonymous edits

Perfect hash function *Source:* <http://en.wikipedia.org/w/index.php?oldid=359642499> *Contributors:* 4hodnt, Burschik, Cesarc's unprivileged account, Cimon Avaro, Cobi, Dcoetze, Dtrebbien, Dlugosz, E David Moyer, Fredrik, G121, Itman, JMCorey, Johndburger, LOL, MegaHasher, Mudd1, Neile, Pagh, Ruud Koot, Salrzv, Spl, Wikilolo, 34 anonymous edits

Universal hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=351130933> *Contributors:* ArnoldReinhold, Arnstein87, Bender2k14, CharlesHBennett, Copland Stalker, Copysan, Cyberjoac, Dmharvey, Dwmalone, EmilJ, Francois.boutines, Guruparan18, Jafet, Neilc, Pagh, Rjwilmsi, Rswarbrick, SeanMack, Superm401, Twintop, Winxa, 20 anonymous edits

Linear hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=355757842> *Contributors:* BrokenSegue, CanadianLinuxUser, CeserB's unprivileged account, David Eppstein, Dcoetzee, ENGIMa, Gail, JultheP, MegaHasher, Res2216firestar, Rjwilmsi, TenPoundHammer, Zawersh, 36 anonymous edits

Extendible hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=359799000> *Contributors:* Alex Kapranoff, Amiceli, Boing! said Zebedee, Gloomy Coder, JustinWick, MegaHasher, Musicant, Spamduck, Tommy2010, Wolfeye90, Zawersh, 56 anonymous edits

2-choice hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=258474388> *Contributors:* Asparagus, JaGa, Onodevo, Picchipich, Zawersh

Pearson hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=344115778> *Contributors:* Abednigo, Avocado, Bkonrad, Charles Matthews, Chieffuggybear, Creative1985, Dwmalone, Dlugosz, Ed Poor, Epolk, Fredrik, Frigoris, LittleDan, Oli Filth, Pbannister, PeterPearson, Phil Boswell, Quuxplusone, The Anome, 10 anonymous edits

Fowler–Noll–Vo hash function *Source:* <http://en.wikipedia.org/w/index.php?oldid=357037097> *Contributors:* Apoc2400, Bachrach44, Boredzo, Bubba73, CRGReathouse, Cybercobra, Damian Yerrick, David spector, Elf-friend, Enochlau, HumphreyW, Jorge Stolfi, Landon Curt Noll, Ospalh, Phil Spectre, Raph Levien, Raymondwinn, Rbarreira, Ron Ritzman, TruthIIIPower, Uzume, Woohookitty, 31 anonymous edits

Bitstate hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=287405139> *Contributors:* Andreas Kaufmann, Jirislaby, Mild Bill Hiccup, Paalappo, 1 anonymous edits

Bloom filter *Source:* <http://en.wikipedia.org/w/index.php?oldid=359269939> *Contributors:* Agl, Alexmadon, Andrea Kaufmann, AndreasBWagner, Argav, Babbage, Bblfish, Bramp, Brighterorange, Connnett, CeserB's unprivileged account, Charles Matthews, Chocolateboy, Chronulator, Cybercobra, David Eppstein, Dcoetzee, Drae, Drafie, Drangon, Dtrebbien, Edward, Emorisseyy, EvilGrin, Fredrik, Furrykef, Galaa2, Galaxiaad, Gharb, Gtoal, Igodard, James A. Donald, Jerz4835, Justin W Smith, Kronos04, Labongo, Lindsay658, Marokwitz, Michael Hardy, Mindmatrix, Msikma, NavenduAin, Neilc, Nhahmada, Payrard, Pfunk42, Quansstro, Quuxplusone, Requestion, Rlaufuer, Rubicanto, Russianspy3, Ryan Reich, RzR, Shakeelmahate, ShaunMacPherson, Shreevatsa, Stev0, SteveJothen, Subrabbit, Susvolans, The Anome, Trachten, Vsriram, Willpwill, Wirthi, Wwwolf, X7q, Xiphoris, 162 anonymous edits

Locality preserving hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=346843169> *Contributors:* Adverick, BenFrantzDale, Cataclysm, Elaborating, Jitse Niesen, Ligulem, Lourakis, Mdd, Zorakoid, 2 anonymous edits

Morton number *Source:* <http://en.wikipedia.org/w/index.php?oldid=320361974> *Contributors:* Asztal, Lourakis, Salih, Veritaba, 1 anonymous edits

Zobrist hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=357056898> *Contributors:* Charles Matthews, Dlugosz, Evand, Giftlite, IanOsgood, Jafet, Julian Cardona, Matthew Woodcraft, Qwertus, Shirifan, Stephen Morley, WVhybrid, ZeroOne, 10 anonymous edits

Rolling hash *Source:* <http://en.wikipedia.org/w/index.php?oldid=347220745> *Contributors:* Bo Lindbergh, CeserB's unprivileged account, DanielLemire, Eug, Hegariz, Jafet, Martlau, 28 anonymous edits

Hash list *Source:* <http://en.wikipedia.org/w/index.php?oldid=354162812> *Contributors:* CeserB's unprivileged account, D climacus, DataWraith, Davidgothberg, Digitalme, Grm wnr, Harryboyles, Homerjay, IstvanWolf, Jeff3000, JonLS, K;ngdfhg, M, Ruud Koot, 10 anonymous edits

Hash tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=354177034> *Contributors:* Blaisorblade, CeserB's unprivileged account, Ciphergoth, Cybercobra, DataWraith, Davidgothberg, GargoyleMT, Gdr, Giftlite, Gil Schmidt, Httmlapps, Intgr, Jim Douglas, Kazvorpal, Kmag, Kroepke, M, Marudubshinki, Matt Crypto, Nysin, Purebill, Rashless, Rhys.rhaven, RistoLaanoja, Ruud Koot, Sperxios, Thattommyhall, Thelest, Thv, Tr00st, Ullner, Zack, 34 anonymous edits

Prefix hash tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=330831311> *Contributors:* Dbenbenn, Intgr, Miym, Nad, The Thing That Should Not Be, 2 anonymous edits

Hash trie *Source:* <http://en.wikipedia.org/w/index.php?oldid=352502258> *Contributors:* Evand, Gdr, Magioladitis, OrangeDog, Pombredanne, Tr00st

Hash array mapped trie *Source:* <http://en.wikipedia.org/w/index.php?oldid=357102759> *Contributors:* Andreas Kaufmann, D6, IMNeme, Jrw@pobox.com, Wynand.winterbach, 1 anonymous edits

Distributed hash table *Source:* <http://en.wikipedia.org/w/index.php?oldid=359720051> *Contributors:* Alfio, Allquixotic, Anescent, Anthony, Apoc2400, Athymik, Baa, Bernhard Bauer, Bpringlemeir, Br, Bryan Derksen, Bubba hotep, Cacheonix, Cburnett, Charles Matthews, Chrismiceli, Chy168, Cslin, Cojoco, Corti, CosineKitty, Crystallina, Cwolfsheep, CyberSkull, DJPhazer, DerHexer, Diberri, Dito, Ehn, Elf, EmilSit, Enzo Aquarius, Eras-mus, Eric.weigle, Erik Sandberg, F, Frick, FlyHigh, GPHemsley, Garas, Godrickwok, Goofi1781, Gpiere, Greywiz, Haakon, Hadal, Happyrabbit, Harryboyles, Hbpencil, Hu12, Imeshev, Intgr, Irrbloss, Itai, Jamesday, Jda, Jensoan, Jnlin, JonHarder, Kernesky, Kbrose, Khalid hassani, KnightRider, Knuckles, Luqui, M.B, MUzzammils, Mabdul, Md4567, Meneth, Mgrinich, Michael miceli, MikeSchmid111, Minority Report, Miym, Monkeyjunk, Nad, Naff89, Nazlfrag, Nealmcb, NeiIN, Nethgirb, Nhorton, Noodles-sb, OPless, Ohnoitsjamie, Oskilian, Pkg, ReinoutS, Roger pack, Ronz, Seano1, Search4Lancer, ShaunMacPherson, Shentino, Squallo, Ssspera, TakuyaMurata, The Anome, The Belgain, TheJosh, Thingg, Thv, Tombeo, UtherSRG, X1987x, Xiong Chiamiov, Xosé, Xubupt, Yramesh, 232 anonymous edits

Consistent hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=359386182> *Contributors:* Argv0, CeserB's unprivileged account, Dirkx, DreamOfMirrors, Edratzer, Egil, Gareth Jones, Harald Haugland, Harrigan, Miym, Nethgirb, Pbrandao, Rene Mas, Skalet, 9 anonymous edits

Stable hashing *Source:* <http://en.wikipedia.org/w/index.php?oldid=332240728> *Contributors:* Alasdair, Alex Kapranoff, Bigtop, CeserB's unprivileged account, Ssilverm, The Anome, TheBlueFlamingo, 1 anonymous edits

Koerde *Source:* <http://en.wikipedia.org/w/index.php?oldid=358290602> *Contributors:* Bchocie, Bladegirl, D6, JPG-GR, Jorgeantibes, Krose, Kyu, Suhhy, Wekkolin, 5 anonymous edits

Graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=358429462> *Contributors:* 31stCenturyMatt, Aaronzat, Alink, Andreas Kaufmann, Any Key, Bluebusy, Booyabazooka, C4Cypher, Chochopk, Chrisholland, Cybercobra, David Eppstein, Dcoetzee, Dysprosia, Epimetheus, FedericoMenaQuintero, Gallando, Giftlite, Gmelli, Graphicalx, Gvanrossum, Hariva, Hobsonlane, Jojit fb, Jon Awbrey, Jorge Stolfi, Juliancolton, Kate4341, Kazubon, Krose, KellyCoinGuy, Kendrick Hang, Labraun90, Liao, Max Terry, NerdyNSK, Nmz787, Obradovic Goran, Ovi 1, P0nc, Pbirnie, Pieleric, R. S. Shaw, RG2, Rabarberski, Rborrego, Rd232, Rhanekom, Saimhe, Salix alba, Simonfairfax, SiobhanHansa, Skippydo, Sphung, TFloto, Timwi, Tyir, Zoicon5, ZorroIII, 88 anonymous edits

Adjacency list *Source:* <http://en.wikipedia.org/w/index.php?oldid=360145533> *Contributors:* Andreas Kaufmann, Ash211, Beetstra, Bobbertface, Booyabazooka, Chmod007, Cobi, David Eppstein, Dcoetzee, Dysprosia, Fredrik, Garyzx, Hariva, Hobsonlane, Iridescent, Jamelan, Jwpurple, Kku, Krazelke, MathMartin, NSR, Oleg Alexandrov, Only2sea, Rdude, SPTWriter, Schneelocke, ThE cRaCkEr, Twri, 23 anonymous edits

Adjacency matrix *Source:* <http://en.wikipedia.org/w/index.php?oldid=355979063> *Contributors:* Abdull, Abha Jain, Aleph4, Arthouse, AxelBoldt, Beetstra, BenFrantzDale, Bitsianshash, Booyabazooka, Burn, Calle, Churnett, David Eppstein, Dcoetzee, Debresser, Dreadstar, Dysprosia, ElBenevolente, Fredrik, Garyzx, Gauge, Giftlite, JackSchmidt, JamesBWatson, Jean Honorio, Jokers, Juffi, Kneufeld, Lipedia, MarkSweep, MathMartin, Mbogelund, Mdrine, Michael Hardy, Miym, Morre, Natelewis, Oleg Alexandrov, Only2sea, Periergeia, Phils, Reina riemann, Rhetth, Rich Farmbrough, RobinK, SPTWriter, Salgueiro, Schneelocke, Slawomir Bialy, TakuyaMurata, Tbackstr, Tim Q. Wells, Timendum, Tomo, Ttzz, Twri, Yoav HaCohen, YuryKirienko, Zaslav, سعی، 56 anonymous edits

And-inverter graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=327328966> *Contributors:* Aaron Hurst, Alan Mishchenko, Andreas Kaufmann, Andy Crews, Appraiser, Gregbard, Igor Markov, Jjiang, Jon Awbrey, Jonathan de Boyne Pollard, Kettiltrout, Michael Hardy, Mikeblas, Pigorsch, Rjwilmsi, Trovatore, 3 anonymous edits

Binary decision diagram *Source:* <http://en.wikipedia.org/w/index.php?oldid=358665124> *Contributors:* Ajo Mama, Amccosta, Andreas Kaufmann, Andris, AshtonBenson, Bluemoose, Bobke, Boute, Brighterorange, Charles Matthews, David.Monniaux, Derek farn, Dirk Beyer, EmilJ, Greenrd, GregorB, Gtrmp, Hermel, Heron, Hitanshu D, J04n, Jason Reliner, Esq., Jay.Here, Jcarroll, Jkl, Kschutte, Kakesson, Karlbrace, Karlk, Karsten Strehl, Laudaka, LouScheffer, Matumio, McCart42, Mdd, Nouiz, Pce3@ij.net, Rohit.nadig, Ryan Clark, Sade, SailorH, Sam Hocevar, Sirutan, Taw, Trovatore, Twri, Uli, YannTM, 91 anonymous edits

Binary moment diagram *Source:* <http://en.wikipedia.org/w/index.php?oldid=184571998> *Contributors:* Andreas Kaufmann, Jon Awbrey, Taw, 2 anonymous edits

Zero suppressed decision diagram *Source:* <http://en.wikipedia.org/w/index.php?oldid=212084345> *Contributors:* Andreas Kaufmann, Doradus, Eep², Esbenrunre, Jon Awbrey, Taw, TubularWorld, Waggers, 2 anonymous edits

Propositional directed acyclic graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=297664119> *Contributors:* Aagtbdoua, Andreas Kaufmann, BD2412, Brighterorange, DRap, Nbarth, RUN, Selket, Trovatore, 4 anonymous edits

Graph-structured stack *Source:* <http://en.wikipedia.org/w/index.php?oldid=332032272> *Contributors:* Andreas Kaufmann, Charles Matthews, Djspiewak, Fredrik, Jaredwf, SpaceMoose, TimBentley, 2 anonymous edits

Scene graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=347303142> *Contributors:* Altenmann, Amatheny, Andreas Kaufmann, Anita.havale, Ap, BAxelrod, C4Cypher, CALR, CTho, CamTarn, Cbraga, Cmdrjameson, Cpu11, Davepape, Docu, DzzD, EmanWilm, Engwar, Fredrik, GregorB, J04n, JLaTondre, Jmdyck, KickAssClown, Kmote, Lancekt, Leroyluc, Lockewownzj00, Lowellian, Mav, Mayalld, Mecanismo, Michael Hardy, Mortene, Mushin, Nknight, P0lyglut, PeterV1510, Reinyday, Ricvelozo, Rufous, Stephanie40, SynergyBlades, Tommstein, Tony Sidaway, WillC2 45220, Wsriley, Zundark, 64 anonymous edits

Big O notation *Source:* <http://en.wikipedia.org/w/index.php?oldid=359785314> *Contributors:* 4v4l0n42, A-Ge0, A. Pichler, ABCD, Abdull, Adashiel, Addps4cat, Aelvin, Ahmad Faridi, Ahoerstemeier, Alan smithe, Alex Selby, Alksentr, AllanBz, Altenmann, AnOddName, Andre Engels, Andreas Kaufmann, Anonymous Dissident, Anthony Appleyard, Apanag, Arthur Rubin, Arvindn, Ascánder, AvicAWB, AxelBoldt, B4hand, Bagsc, Barak Sh, Baronjonas, Ben pcc, BenFrantzDale, Bergstra, Bird of paradox, Bkell, Booyabazooka, Borgx, Brion VIBBER, Bryner, Bubba73, Buster79, C45207, CBKA(Topsails), CRGreathouse, Calculuslover, Charlow3, Charles Matthews, CharlesGillingham, ChazBeckett, ChrisForno, ClemMcGann, Colfulus, Compotatoj, Connnelly, Cookie4869, CosineKitty, Corps, Cybercobra, CyborgTosser, D4g0thur, DFS454, Dachshund, Dadudadu, Damian Yerrick, Danakil, Danny, Dark Charles, David Eppstein, Davidwt, Deljr, Doctzee, Deeparnab, Den fjärrtrade ankan, Derlay, Dhuss, Diberri, Dionyziz, DniQ, Doradus, Dr. Universe, Draco flavus, Drpaule, Duagloth, Dysprosia, EconoPhysicist, El C, Elephant in a tornado, Eleveneleven, EmilJ, Enochlau, Epachamo, Eric119, Ernie shoemaker, FauxFaux, Felix Wiemann, Fennec, Fibonacci, FilipeS, Flouran, Foxjwill, Fredrik, Fvw, Gadig, Gene Ward Smith, GeordieMcBain, Giftlite, Gilliam, Gjd001, Glassmage, Gracenotes, Gremagor, Gutza, H.ehsaan, Haham hanuka, Hans Adler, Hdante, Head, HenningThielemann, Henrygb, Hermel, Hlg, Intgr, Isis, Ixfd64, JHMM13, JIP, Jacobolus, James.S, Jaredwf, Javit, Jeronimo, Jleedev, JoeKearney, JohnWStockwell, Jolfsa123, Jonathanzung, Jowwan2005, Jpkotta, Jthillik, Justin W Smith, Jwh335, Kan8eDie, Katsushi, KneeLess, KoenDelaere, Koffieyahoo, LC, LOL, Lambiam, Lamro, Leithp, LeonardoGregorian, Leycec, Linas, Ling.Nut, Luqui, MFH, MIT Trekkie, Macrakis, Mad Jack, Maksim-e, MarkOlah, MathMartin, Matiasholte, McKay, Mcstrother, Melcombe, Michael Hardy, Michael Rogers, Michael Slone, Miguel, Mike Schwartz, Mitchoyoshitaka, Miym, Mobius, Modeha, Mpaganu, Myrpislon, Mstuomel, Mn, NephestTheFirst, Neilc, Nejko, NeoUrfahrner, Nils Grimsmo, NovaDog, O Pavlos, Oleg Alexandrov, Oliphant, Opelio, Optikos, Ott2, PGWG, Patrick, Patrick Lucas, Paul August, PaulTanenbaum, Paxcoder, Pcufl, Pete4512, Plutor, Poor Yorick, Prumpf, Quendus, Qwfp, R.e.b., R3m0t, Raknar44, Rebread, Reinderen, RexNL, Rfl, Riceplaytexas, Rjwilmst, RobertBorgersen, RobinK, Rovenhot, Ruud Koot, Sabalka, SchfiftyThree, Scirurina, ScotsmanRS, Sean22190, Shalom Yechiel, Shellgirl, Shizhao, Shoessss, Shreevatsa, Simetrical, Simon Fenney, Skaraoke, Sligoocki, Smig, Sophus Bie, Spitzak, Stefan.karpinski, Stephen Compall, StevenJ, Stevertigo, Stimp, Sydbarrett74, Syncategoremeta, TNARasslin, Taemyr, TakuyaMurata, Tardis, Tarotcards, The Anome, Thenub314, Tide rolls, Timwi, Tony Fox, Tossa, Tritium6, Ultimus, Universalss, User A1, Vecter, Vedant, VictorAnyakin, Whosyourjudas, Writer on wiki, Yarin Kaul, ZAB, Zack, Zero sharp, ZeroOne, Zowch, Zundark, Zvika, île flottante, 536 anonymous edits

Amortized analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=357664300> *Contributors:* Altenmann, Andreas Kaufmann, BarretBonden, Brazzy, Brona, Caesura, Dcoetzee, EconoPhysicist, Giftlite, Jimp, Laubrau, Mantipula, Michael Hardy, Nneonneo, Oravec, Pierre de Lyon, Poor Yorick, PrologFan, Quetzuce, Rbonvall, Safek, Stevo2001, TakuyaMurata, Torzsmokus, User A1, 15 anonymous edits

Locality of reference *Source:* <http://en.wikipedia.org/w/index.php?oldid=351413542> *Contributors:* 16@r, 18.94, Adrianwn, Andreas Kaufmann, BMF81, BenFrantzDale, Brianhe, Charles Matthews, Costaluz, Cpiral, Cyberjoac, DTM, Derek farn, Ecb29, Ee00224, Einstein9073, Felix Andrews, Firsfron, Fredrik, Helwr, Intgr, Ixfd64, JPG-GR, JonHarder, Jruderman, KatelynJohann, Kbdkank71, Kenyon, Kurt Jansson, Mboverload, Mirror Vax, Naroza, Not-just-yet, Phils, Piet Delport, Prohlep, R'n'B, Radagast83, Randomalous, ShakataGaNai, Stephen Morley, TakuyaMurata, The Anome, Themusicgod1, Trjonescp, Uncle G, Uttar, Werner, Zawersh, 43 anonymous edits

Standard Template Library *Source:* <http://en.wikipedia.org/w/index.php?oldid=355487849> *Contributors:* Ahannani, AleyCZ, Alfio, Alksentr, Alvin-es, Andres, Aragorn2, Avinashm, Barfooz, BenFrantzDale, Benhoyt, Bobbyi, Brainfsck, Bwgstaff, Chemisor, Chris-gore, Cibu, Curps, DanielKO, David Eppstein, Decltype, Deineka, Deryck Chan, Discospinster, Dreftymac, Drrrngrvy, Emuka, Enerjazzer, Flex, Frecklefoot, Furrykef, Gaius Cornelius, Geregen2, Glenn, Gustavo.mori, Hadas583, Hydrogen Iodide, Ian Pitchford, J04n, JTN, Jesse Viviano, Jibjibjib, Jjcolotti, Jkt, Jorend, Josh Cherry, K3rb, Keno, Lawrencegold, LiDaobing, LilHelpa, MarSch, Marc Mongenet, Martin Moene, Merphant, Minghong, Mirror Vax, Modster, Moritz, Mrjeff, Mushroom, Muthudesigner, Neilc, Nneonneo, Norm mit, Nwbeeson, Olydbg, Omnicog, Orderud, Oğuz Ergin, Pavel Vozenilek, Pedram.salehpoor, Pieleric, Piet Delport, Pt, Pulseczar, R2100, Rijkbenik, Rjwilmst, Rookkey, RoySmith, Sairahulreddy, Saziel, Sdorrance, Sebor, Shoaler, Smalljim, Smyth, Soumyasch, Spoon!, Spurymoses, Stoph, Streetraider, Tacvek, Tavianator, Thumperward, TingChong Ma, Tobias Bergemann, Toffile, Tony Sidaway, Torc2, Tyler Oderkirk, Uiweo, Vald, Vincenzo.romano, Whaa?, Wiki0709, Wmbolle, Xerxesnine, Zigger, Zundark, Zzuuzz, 135 anonymous edits

Image Sources, Licenses and Contributors

Image:Singly-linked-list.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Singly-linked-list.svg> *License:* Public Domain *Contributors:* User:Lasindi

Image:Singly linked list.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Singly_linked_list.png *License:* Public Domain *Contributors:* User:Dcoetzee

Image:Data stack.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Data_stack.svg *License:* Public Domain *Contributors:* User:Boivie

Image:ProgramCallStack2.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:ProgramCallStack2.png> *License:* Public Domain *Contributors:* Agateller, LobStoR, Obradovic Goran

Image:PD-icon.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:PD-icon.svg> *License:* Public Domain *Contributors:* User:Duesentrieb, User:Rfl

Image:Data Queue.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Data_Queue.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Foroa, Vegpuff

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

Image:Array of array storage.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Array_of_array_storage.svg *License:* Public Domain *Contributors:* User:Dcoetzee

Image:HashedArrayTree16.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:HashedArrayTree16.svg> *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Surachit at w:User:Surachiten.wikipedia

Image:Circular buffer.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - empty.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_empty.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - XX1XXXX.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_XX1XXXX.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - XX123XX.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_XX123XX.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - XXXX3XX.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_XXXX3XX.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - 6789345.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_6789345.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - 6789AB5.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_6789AB5.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - X789ABX.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_X789ABX.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - XX123XX with pointers.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_XX123XX_with_pointers.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - 6789AB5 with pointers.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_6789AB5_with_pointers.svg *License:* GNU Free Documentation License *Contributors:* User:Cburnett

Image:Circular buffer - 6789AB5 full.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_6789AB5_full.svg *License:* GNU Free Documentation License *Contributors:* , modifications

Image:Circular buffer - 6789AB5 empty.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Circular_buffer_-_6789AB5_empty.svg *License:* GNU Free Documentation License *Contributors:* , modifications

Image:Abramowitz&Stegun.page97.agr.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Abramowitz&Stegun.page97.agr.jpg> *License:* Creative Commons Attribution 2.5 *Contributors:* User:ArnoldReinhold, User:Trojancowboy

Image:Interpolation example linear.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Interpolation_example_linear.svg *License:* Public Domain *Contributors:* User:Berland

Image:Circularly-linked-list.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Circularly-linked-list.svg> *License:* Public Domain *Contributors:* User:Lasindi

Image:Doubly-linked-list.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Doubly-linked-list.svg> *License:* Public Domain *Contributors:* User:Lasindi

Image:Singly linked list insert after.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Singly_linked_list_insert_after.png *License:* unknown *Contributors:* Derrick Coetzee

Image:Singly linked list delete after.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Singly_linked_list_delete_after.png *License:* unknown *Contributors:* Derrick Coetzee

File:VList example diagram.png *Source:* http://en.wikipedia.org/w/index.php?title=File:VList_example_diagram.png *License:* Public Domain *Contributors:* User:Dcoetzee

Image:Skip_list.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Skip_list.svg *License:* Public Domain *Contributors:* User:Wojciech mula

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

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:* User:Cyhawk

Image: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

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

Image:AVLtree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:AVLtree.svg> *License:* Public Domain *Contributors:* User:Mikm

Image:Tree rotation.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_rotation.png *License:* GNU Free Documentation License *Contributors:* User:Ramasamy

Image:Tree Rotations.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_Rotations.gif *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* C. A. Russell, Mtanti, 1 anonymous edits

Image:Tree Rebalancing.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_Rebalancing.gif *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Original uploader was Mtanti at en.wikipedia

Image:Weight_balanced_tree2.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Weight_balanced_tree2.jpg *License:* Public Domain *Contributors:* KenE

Image:Threaded tree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Threaded_tree.svg *License:* Public Domain *Contributors:* User:R. S. Shaw

Image:Red-black tree example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Red-black_tree_example.svg *License:* GNU Free Documentation License *Contributors:* Cburrett, German, Grafite, S Sepp, Wknight94, 5 anonymous edits

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:* GNU Free Documentation License *Contributors:* Grafite, 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, Deekkar 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 Deekkar, 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:Red Black Shape Cases.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Red_Blk_Shape_Cases.svg *License:* GNU Free Documentation License *Contributors:* User:Why Not A Duck

Image:AA Tree Shape Cases.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:AA_Tree_Shape_Cases.svg *License:* GNU Free Documentation License *Contributors:* User:Why Not A Duck

Image:AA Tree Skew2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:AA_Tree_Skew2.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Why Not A Duck

Image:AA Tree Split2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:AA_Tree_Split2.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Why Not A Duck

Image:Zig.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Zig.gif> *License:* GNU Free Documentation License *Contributors:* Dcoetzee, Regnaron, Remember the dot

Image:Zigzag.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Zigzag.gif> *License:* GNU Free Documentation License *Contributors:* Dcoetzee, Regnaron, Remember the dot

Image:Zigzag.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Zigzag.gif> *License:* GNU Free Documentation License *Contributors:* Dcoetzee, Regnaron, Remember the dot

Image:T-tree-1.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:T-tree-1.png> *License:* unknown *Contributors:* Original uploader was Hholzgra at en.wikipedia

Image:T-tree-2.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:T-tree-2.png> *License:* unknown *Contributors:* Hholzgra

Image:Top tree.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Top_tree.jpg *License:* Public Domain *Contributors:* Afrozenator

File:Cartesian tree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Cartesian_tree.svg *License:* Public Domain *Contributors:* User:David Eppstein

File:Cartesian tree range searching.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Cartesian_tree_range_searching.svg *License:* Public Domain *Contributors:* User:David Eppstein

File:Bracketing pairs.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bracketing_pairs.svg *License:* Public Domain *Contributors:* User:David Eppstein

Image:TrepAlphaKey.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:TrepAlphaKey.svg> *License:* Creative Commons Zero *Contributors:* User:Qef

Image:B-tree.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:B-tree.png> *License:* unknown *Contributors:* Fresheneesz, Qef, 1 anonymous edits

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

Image:Btree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Btree.svg> *License:* GNU Free Documentation License *Contributors:* Mikm, Nagae, 3 anonymous edits

Image:2-3-4 tree 2-node.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:2-3-4_tree_2-node.svg *License:* Public Domain *Contributors:* User:Chrismiceli

Image:2-3-4-tree 3-node.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:2-3-4-tree_3-node.svg *License:* Public Domain *Contributors:* User:Chrismiceli

Image:2-3-4 tree insert 1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:2-3-4_tree_insert_1.svg *License:* Public Domain *Contributors:* User:Chrismiceli

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

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

Image: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.

Image: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

Image:Heap add step1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_add_step1.svg *License:* Public Domain *Contributors:* User:Ilmari Karonen

Image:Heap add step2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_add_step2.svg *License:* Public Domain *Contributors:* User:Ilmari Karonen

Image:Heap add step3.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_add_step3.svg *License:* Public Domain *Contributors:* User:Ilmari Karonen

Image:Heap remove step1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_remove_step1.svg *License:* Public Domain *Contributors:* User:Ilmari Karonen

Image:Heap remove step2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Heap_remove_step2.svg *License:* Public Domain *Contributors:* User:Ilmari Karonen

Image:Binomial_Trees.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binomial_Trees.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Lemontea, Ma-Lik

Image:Binomial-heap-13.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Binomial-heap-13.svg> *License:* GNU Free Documentation License *Contributors:* User:D0ktorz

Image:Binomial heap merge1.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binomial_heap_merge1.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Lemontea

Image:Binomial heap merge2.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Binomial_heap_merge2.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Lemontea

Image:Fibonacci_heap.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Fibonacci_heap.png *License:* GNU Free Documentation License *Contributors:* User:Brona, User:Brona/Images/fibonacci_heap.tex

Image:Fibonacci heap extractmin1.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Fibonacci_heap_extractmin1.png *License:* GNU Free Documentation License *Contributors:* User:Brona, User:Brona/Images/fibonacci_heap.tex

Image:Fibonacci heap extractmin2.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Fibonacci_heap_extractmin2.png *License:* GNU Free Documentation License *Contributors:* User:Brona, User:Brona/Images/fibonacci_heap.tex

Image:Fibonacci heap-decreasekey.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Fibonacci_heap-decreasekey.png *License:* GNU Free Documentation License *Contributors:* User:Brona, User:Brona/Images/fibonacci_heap.tex

Image:beap.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Beap.jpg> *License:* GNU Free Documentation License *Contributors:* User:OrphanBot

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:* User:Computergeeksjw

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:SkewHeapMerge1.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:SkewHeapMerge1.svg> *License:* Public Domain *Contributors:* User:Quintaylor

Image:SkewHeapMerge2.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:SkewHeapMerge2.svg> *License:* Public Domain *Contributors:* User:Quintaylor

Image:SkewHeapMerge3.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:SkewHeapMerge3.svg> *License:* Public Domain *Contributors:* User:Quintaylor

Image:SkewHeapMerge4.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:SkewHeapMerge4.svg> *License:* Public Domain *Contributors:* User:Quintaylor

Image:SkewHeapMerge5.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:SkewHeapMerge5.svg> *License:* Public Domain *Contributors:* User:Quintaylor

Image:SkewHeapMerge6.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:SkewHeapMerge6.svg> *License:* Public Domain *Contributors:* User:Quintaylor

Image:SkewHeapMerge7.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:SkewHeapMerge7.svg> *License:* Public Domain *Contributors:* User:Quintaylor

Image:Min_T Heap.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Min_T_Heap.png *License:* Public Domain *Contributors:* User:G8ken

Image:Max T Heap.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Max_T_Heap.png *License:* Public Domain *Contributors:* User:G8ken

Image:trie example.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Trie_example.svg *License:* Public Domain *Contributors:* User:Superm401

Image:Patricia trie.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Patricia_trie.svg *License:* Creative Commons Attribution 2.5 *Contributors:* Claudio Rocchini

Image:Suffix tree BANANA.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Suffix_tree_BANANA.svg *License:* Public Domain *Contributors:* User:Nux

Image:Suffix tree ABAB_BABA.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Suffix_tree_ABAB_BABA.svg *License:* Public Domain *Contributors:* User:Joey-das-WBF, User:Nils Grimsmo

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

Image:And-or tree.JPG *Source:* http://en.wikipedia.org/w/index.php?title=File:And-or_tree.JPG *License:* unknown *Contributors:* Logperson

File:SPQR tree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:SPQR_tree.svg *License:* Public Domain *Contributors:* User:David Eppstein

Image:spaghettistack.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Spaghettistack.svg> *License:* Public Domain *Contributors:* User:Ealf

Image:Binary space partition.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Binary_space_partition.png *License:* Public Domain *Contributors:* User:Fredrik

Image:Segment tree instance.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Segment_tree_instance.gif *License:* Public Domain *Contributors:* Alfredo J. Herrera Lago

Image:bin computational geometry.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Bin_computational_geometry.png *License:* Public Domain *Contributors:* McLoaf, Qef, Yumf

Image:3dtree.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:3dtree.png> *License:* GNU General Public License *Contributors:* Btyner, 2 anonymous edits

Image:Kdtree 2d.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Kdtree_2d.svg *License:* GNU Free Documentation License *Contributors:* KiwiSunset

Image:Tree_0001.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Tree_0001.svg *License:* Public Domain *Contributors:* User:MYguel

Image:KDTree-animation.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:KDTree-animation.gif> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:User_A1

Image:Implicitmaxkdtree.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Implicitmaxkdtree.gif> *License:* Public Domain *Contributors:* Genieser

Image:Point quadtree.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Point_quadtree.svg *License:* Public Domain *Contributors:* User:David Eppstein

Image:Quad_tree_bitmap.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Quad_tree_bitmap.svg *License:* Public Domain *Contributors:* User:Wojciech mula

Image:Octree2.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Octree2.svg> *License:* GNU Free Documentation License *Contributors:* User:WhiteTimberwolf

File:Four-level Z.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Four-level_Z.svg *License:* GNU Free Documentation License *Contributors:* User:David Eppstein, User:Hesperian

File:Lebesgue-3d-step2.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Lebesgue-3d-step2.png> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:RobertdWc

File:Lebesgue-3d-step3.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Lebesgue-3d-step3.png> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:RobertdWc

File:Z-curve.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Z-curve.svg> *License:* Public Domain *Contributors:* User:David Eppstein

File:Bigmin.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Bigmin.svg> *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* User:DnetSvg

Image:R-tree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:R-tree.svg> *License:* Public Domain *Contributors:* User:Skinkie

Image:RTree-Visualization-3D.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:RTree-Visualization-3D.svg> *License:* Public Domain *Contributors:* User:Chire

Image:figure1_left.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Figure1_left.gif *License:* Public Domain *Contributors:* Okoky

Image:figure1_right.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Figure1_right.gif *License:* Public Domain *Contributors:* Okoky

Image:figure2_Hilbert.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Figure2_Hilbert.gif *License:* Public Domain *Contributors:* Okoky

Image:figure3_data_rects.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Figure3_data_rects.gif *License:* Public Domain *Contributors:* Okoky

Image:figure4_file_structure.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Figure4_file_structure.gif *License:* Public Domain *Contributors:* Okoky

Image:Hash table 3 1 1 0 1 0 0 SP.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_3_1_1_0_1_0_0_SP.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Jorge Stolfi

Image:Hash table 5 0 1 1 1 1 LL.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_5_0_1_1_1_1_LL.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Jorge Stolfi

Image:Hash table 5 0 1 1 1 0 LL.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_5_0_1_1_1_0_LL.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Jorge Stolfi

Image:Hash table 5 0 1 1 1 0 SP.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_5_0_1_1_1_0_SP.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Jorge Stolfi

Image:Hash table average insertion time.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_table_average_insertion_time.png *License:* Public Domain *Contributors:* User:Dcoetze

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:* User: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:* User: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:* User:Jorge Stolfi

Image:HASHTB12.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:HASHTB12.svg> *License:* Public Domain *Contributors:* Helix84, Simeon87, Velociostrich, Xhienne

Image:cuckoo.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Cuckoo.png> *License:* GNU Free Documentation License *Contributors:* Joelholdsworth, Pagh

Image:CoalescedHash.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:CoalescedHash.jpg> *License:* Public Domain *Contributors:* Confuzzled, Nv8200p

File:Bloom filter.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bloom_filter.svg *License:* Public Domain *Contributors:* User:David Eppstein

File:Bloom filter speed.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bloom_filter_speed.svg *License:* Public Domain *Contributors:* Alexmadon

File:Bloom filter fp probability.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bloom_filter_fp_probability.svg *License:* Creative Commons Attribution 3.0 *Contributors:* User:Jezr4835

Image:Hash_list.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_list.svg *License:* Public Domain *Contributors:* Hash_list.png: Original uploader was Davidgothberg at en.wikipedia conversion to SVG: DataWraith (talk)

Image:Hash tree.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Hash_tree.png *License:* Public Domain *Contributors:* Davidgothberg, Qef

File:DHT_en.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:DHT_en.svg *License:* Public Domain *Contributors:* Jinlin

File:De_brujin_graph-for_binary_sequence_of_order_4.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:De_brujin_graph-for_binary_sequence_of_order_4.svg *License:* unknown *Contributors:* SVG version: Wekkolin, Original version: english wikipedia user Michael_Hardy

Image:Koorde lookup routing.JPG *Source:* http://en.wikipedia.org/w/index.php?title=File:Koerde_lookup_routing.JPG *License:* Public Domain *Contributors:* Suhhy

Image:Koorde lookup code.JPG *Source:* http://en.wikipedia.org/w/index.php?title=File:Koerde_lookup_code.JPG *License:* Public Domain *Contributors:* Suhhy

Image:6n-graf.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:6n-graf.svg> *License:* Public Domain *Contributors:* User:AzaToth

Image:Simple cycle graph.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Simple_cycle_graph.svg *License:* Public Domain *Contributors:* Booyabazooka

Image:6n-graph2.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:6n-graph2.svg> *License:* Public Domain *Contributors:* Booyabazooka, Dcoetzee, 1 anonymous edits

Image:And-inverter-graph.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:And-inverter-graph.png> *License:* GNU Free Documentation License *Contributors:* Pigorsch

Image:BDD.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:BDD.png> *License:* GNU Free Documentation License *Contributors:* Original uploader was IMeowbot at en.wikipedia

Image:BDD simple.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:BDD_simple.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* User:Dirk Beyer

Image:BDD Variable Ordering Bad.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:BDD_Variable_Ordering_Bad.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* User:Dirk Beyer

Image:BDD Variable Ordering Good.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:BDD_Variable_Ordering_Good.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* User:Dirk Beyer

File:BDD simple.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:BDD_simple.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* User:Dirk Beyer

File:BDD2pdag.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:BDD2pdag.png> *License:* GNU Free Documentation License *Contributors:* Original uploader was RUN at en.wikipedia

File:BDD2pdag simple.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:BDD2pdag_simple.svg *License:* GNU Free Documentation License *Contributors:* w:User:SelketUser:Selket and w:User:RUNUser:RUN (original)

Image:Graphstructuredstack jaredwf.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Graphstructuredstack_jaredwf.png *License:* Public Domain *Contributors:* User:BotMultichillIT

Image:Stacks jaredwf.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Stacks_jaredwf.png *License:* unknown *Contributors:* jaredwf

License

Creative Commons Attribution-Share Alike 3.0 Unported
<http://creativecommons.org/licenses/by-sa/3.0/>