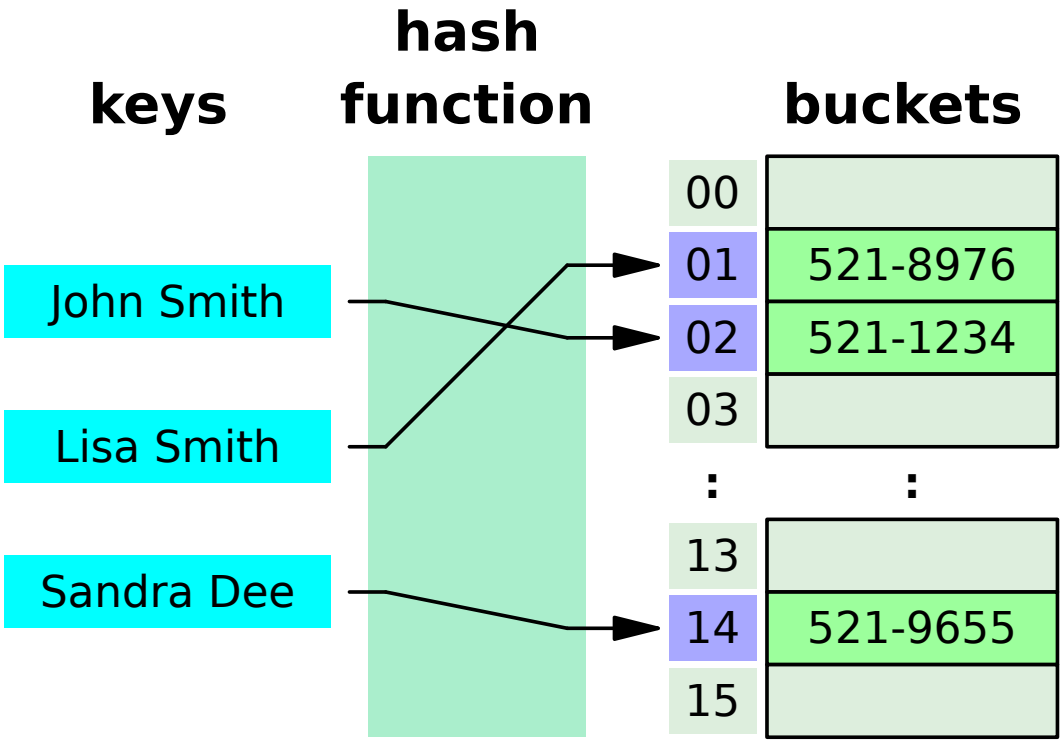
Data Structures, Searching, Sorting

Supporting Information

Contents

Chapter 1

Data structure



A hash table

In computer science, a **data structure** is a particular way of 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, databases use B-tree indexes for small percentages of data retrieval and compilers and databases use dynamic hash tables as look up tables.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Storing and retrieving can be carried out on data stored in both main memory and in secondary memory.

1.1 Overview

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

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

1.2 Examples

Main article: List of data structures

There are numerous types of data structures, generally built upon simpler primitive data types:

- An *array* is a number of elements in a specific order, typically all of the same type. Elements are accessed using an integer index to specify which element is required (although the elements may be of almost any type). Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.
- A *record* (also called a *tuple* or *struct*) is an aggregate data structure. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called *fields* or *members*.
- An *associative array* (also called a *dictionary* or *map*) is a more flexible variation on an array, in which name-value pairs can be added and deleted freely. A hash table is a common implementation of an associative array.
- A *union* type specifies which of a number of permitted primitive types may be stored in its instances, e.g. *float* or *long integer*. Contrast with a record, which could be defined to contain a float *and* an integer; whereas in a union, there is only one value at a time. Enough space is allocated to contain the widest member datatype.
- A *tagged union* (also called a *variant*, *variant record*, *discriminated union*, or *disjoint union*) contains an additional field indicating its current type, for enhanced type safety.
- A *set* is an abstract data structure that can store specific values, in no particular order and with no duplicate values.
- *Graphs* and *trees* are linked abstract data structures composed of *nodes*. Each node contains a value and one or more pointers to other nodes arranged in a hierarchy. Graphs can be used to represent networks, while variants of trees can be used for sorting and searching, having their nodes arranged in some relative order based on their values.
- An *object* contains data fields, like a record, as well as various methods which operate on the contents of the record. In the context of object-oriented programming, records are known as plain old data structures to distinguish them from objects.

1.3 Language support

Most assembly languages and some low-level languages, such as BCPL (Basic Combined Programming Language), lack built-in support for data structures. On the other hand, many high-level programming languages and some higher-level assembly languages, such as MASM, have special syntax or other built-in support for certain data structures,

such as records and arrays. For example, the C and Pascal languages support structs and records, respectively, in addition to vectors (one-dimensional arrays) and multi-dimensional arrays.[3][4]

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

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

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

1.4 See also

- Abstract data type
- Concurrent data structure
- Data model
- Dynamization
- Linked data structure
- List of data structures
- Persistent data structure
- Plain old data structure

1.5 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 Accessed May 21, 2009.*

[2] Entry *data structure* in the Encyclopædia Britannica (2009) Online entry accessed on May 21, 2009.

[3] "The GNU C Manual". Free Software Foundation. Retrieved 15 October 2014.

[4] "Free Pascal: Reference Guide". Free Pascal. Retrieved 15 October 2014.

1.6 Further reading

- Peter Brass, *Advanced Data Structures*, Cambridge University Press, 2008.
- Donald Knuth, *The Art of Computer Programming*, vol. 1. Addison-Wesley, 3rd edition, 1997.
- Dinesh Mehta and Sartaj Sahni *Handbook of Data Structures and Applications*, Chapman and Hall/CRC Press, 2007.
- Niklaus Wirth, *Algorithms and Data Structures*, Prentice Hall, 1985.

1.7 External links

- course on data structures
- Data structures Programs Examples in c,java
- UC Berkeley video course on data structures
- Descriptions from the Dictionary of Algorithms and Data Structures
- Data structures course
- An Examination of Data Structures from .NET perspective
- Schaffer, C. *Data Structures and Algorithm Analysis*

Chapter 2

Array data structure

Not to be confused with Array data type.

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

For example, an array of 10 32-bit integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, … 2036, so that the element with index i has the address $2000 + 4 \times i$.[4]

Because the mathematical concept of a matrix can be represented as a two-dimensional grid, two-dimensional arrays are also sometimes called matrices. In some cases the term "vector" is used in computing to refer to an array, although tuples rather than vectors are more correctly the mathematical equivalent. Arrays are often used to implement tables, especially lookup tables; the word *table* is sometimes used as a synonym of *array*.

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

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

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

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

2.1 History

The first digital computers used machine-language programming to set up and access array structures for data tables, vector and matrix computations, and for many other purposes. Von Neumann wrote the first array-sorting program (merge sort) in 1945, during the building of the first stored-program computer.[6]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, used memory segmentation to perform index-bounds checking in hardware.[7]

Assembly languages generally have no special support for arrays, other than what the machine itself provides. The earliest high-level programming languages, including FORTRAN (1957), COBOL (1960), and ALGOL 60 (1960), had support for multi-dimensional arrays, and so has C (1972). In C++ (1983), class templates exist for multi-dimensional arrays whose dimension is fixed at runtime[3][5] as well as for runtime-flexible arrays.[2]

2.2 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.

2.3 Element identifier and addressing formulas

When data objects are stored in an array, individual objects are selected by an index that is usually a non-negative scalar integer. Indices are also called subscripts. An index *maps* the array value to a stored object.

There are three ways in which the elements of an array can be indexed:

- **0** (*zero-based indexing*): The first element of the array is indexed by subscript of 0.[8]
- **1** (*one-based indexing*): The first element of the array is indexed by subscript of 1.[9]
- **n** (*n-based indexing*): The base index of an array can be freely chosen. Usually programming languages allowing *n-based indexing* also allow negative index values and other scalar data types like enumerations, or characters may be used as an array index.

Arrays can have multiple dimensions, thus it is not uncommon to access an array using multiple indices. For example a two-dimensional array A with three rows and four columns might provide access to the element at the 2nd row and 4th column by the expression A[1, 3] (in a row major language) or A[3, 1] (in a column major language) in the case of a zero-based indexing system. Thus two indices are used for a two-dimensional array, three for a three-dimensional array, and n for an n-dimensional array.

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.

2.3.1 One-dimensional arrays

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

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

Syntax : datatype anArrayname[sizeofArray];

In the given example the array can contain 10 elements of any value available to the int type. In C, the array element indices are 0-9 inclusive in this case. For example, the expressions anArrayName[0] and anArrayName[9] are the first and last elements respectively.

For a vector with linear addressing, the element with index i is located at the address $B + c \times i$, 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.

2.3.2 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, …, ik is

$$B + c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + ck \cdot ik.$$

For example: int a[3][2];

This means that array a has 3 rows and 2 columns, and the array is of integer type. Here we can store 6 elements they are stored linearly but starting from first row linear then continuing with second row. The above array will be stored as a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}.

This formula requires only k multiplications and k 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 ck 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 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.

2.3.3 Dope vectors

The addressing formula is completely defined by the dimension d, the base address B, and the increments c_1, c_2, …, ck. It is often useful to pack these parameters into a record called the array's *descriptor* or *stride vector* or *dope vector*.[2][3] 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.[2]

2.3.4 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 in each row are stored in consecutive positions and all of the elements of a row have a lower address than any of the elements of a consecutive row:

In column-major order (traditionally used by Fortran), the elements in each column are consecutive in memory and all of the elements of a column have a lower address than any of the elements of a consecutive column:

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

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

2.3.5 Resizing

Main article: Dynamic array

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

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

2.3.6 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.

2.4 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.

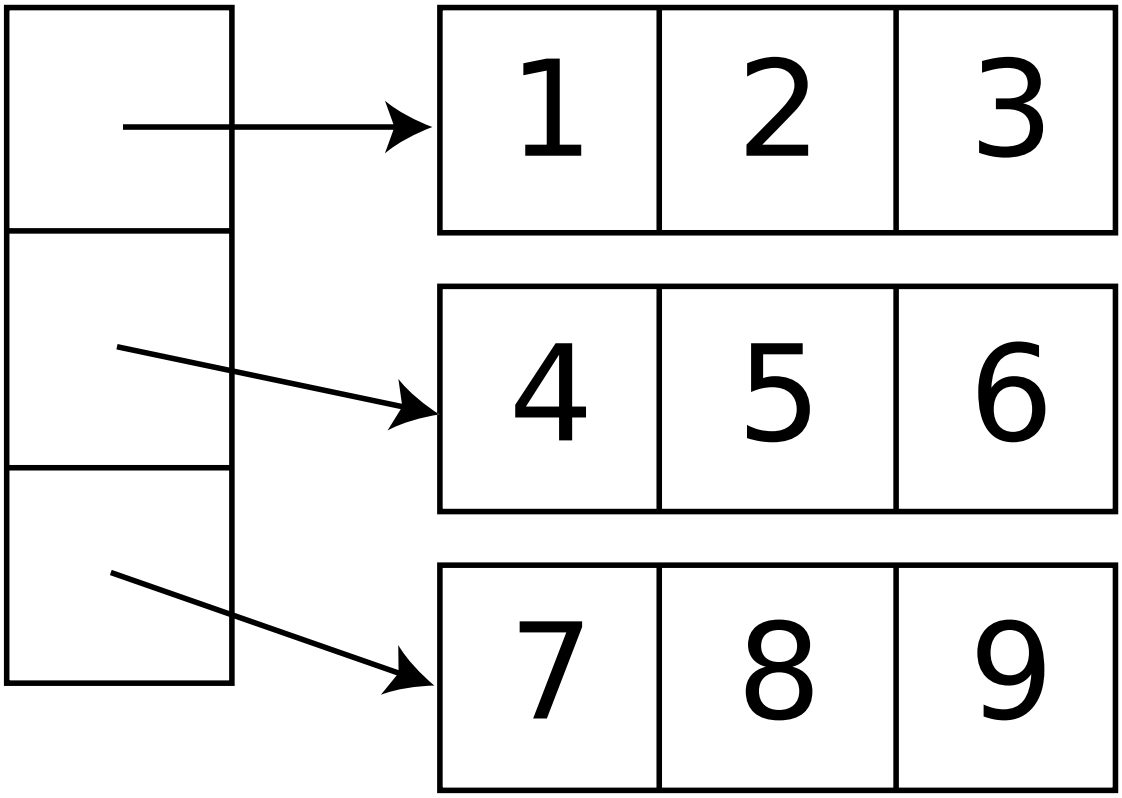
2.4.1 Comparison with other data structures

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,[14] 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.



A two-dimensional array stored as a one-dimensional array of one-dimensional arrays (rows).

An Iliffe vector is an alternative to a multidimensional array structure. It uses a one-dimensional array of references to arrays of one dimension less. For two dimensions, in particular, this alternative structure would be a vector of pointers to vectors, one for each row. Thus an element in row i and column j of an array A would be accessed by double indexing ($A[i][j]$ in typical notation). This alternative structure allows 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.

2.5 Dimension

The dimension of an array is the number of indices needed to select an element. Thus, if the array is seen as a function on a set of possible index combinations, it is the dimension of the space of which its domain is a discrete subset. Thus

a one-dimensional array is a list of data, a two-dimensional array a rectangle of data, a three-dimensional array a block of data, etc.

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

2.6 See also

- Dynamic array
- Parallel array
- Variable-length array
- Bit array
- Array slicing
- Offset (computer science)
- Row-major order
- Stride of an array

2.7 References

[1] Black, Paul E. (13 November 2008). "array". *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Retrieved 22 August 2010.

[2] Bjoern Andres; Ullrich Koethe; Thorben Kroeger; Hamprecht (2010). "Runtime-Flexible Multi-dimensional Arrays and Views for C++98 and C++0x". arXiv:1008.2909 [cs.DS].

[3] Garcia, Ronald; Lumsdaine, Andrew (2005). "MultiArray: a C++ library for generic programming with arrays". *Software: Practice and Experience* **35** (2): 159–188. doi:10.1002/spe.630. ISSN 0038-0644.

[4] David R. Richardson (2002), The Book on Data Structures. iUniverse, 112 pages. ISBN 0-595-24039-9, ISBN 978-0-595-24039-5.

[5] T. Veldhuizen. Arrays in Blitz++. In Proc. of the 2nd Int. Conf. on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), LNCS 1505, pages 223-220. Springer, 1998.

[6] Donald Knuth, *The Art of Computer Programming*, vol. 3. Addison-Wesley

[7] Levy, Henry M. (1984), *Capability-based Computer Systems*, Digital Press, p. 22, ISBN 9780932376220.

[8] "Array Code Examples - PHP Array Functions - PHP code". http://www.configure-all.com/: Computer Programming Web programming Tips. Retrieved 8 April 2011. In most computer languages array index (counting) starts from 0, not from 1. Index of the first element of the array is 0, index of the second element of the array is 1, and so on. In array of names below you can see indexes and values.

[9] "Chapter 6 - Arrays, Types, and Constants". *Modula-2 Tutorial*. http://www.modula2.org/tutor/index.php. Retrieved 8 April 2011. The names of the twelve variables are given by Automobiles[1], Automobiles[2], ... Automobiles[12]. The variable name is "Automobiles" and the array subscripts are the numbers 1 through 12. [i.e. in Modula-2, the index starts by one!]

[10] Gerald Kruse. CS 240 Lecture Notes: Linked Lists Plus: Complexity Trade-offs. Juniata College. Spring 2008.

[11] *Day 1 Keynote - Bjarne Stroustrup: C++11 Style* at *GoingNative 2012* on *channel9.msdn.com* from minute 45 or foil 44

[12] *Number crunching: Why you should never, ever, EVER use linked-list in your code again* at *kjellkod.wordpress.com*

[13] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999), *Resizable Arrays in Optimal Time and Space (Technical Report CS-99-09)*, Department of Computer Science, University of Waterloo

[14] Counted B-Tree

2.8 External links

Chapter 3

Record (computer science)

In computer science, a **record** (also called **struct** or **compound data**)[1] is a basic data structure (a tuple may or may not be considered a record, and vice versa, depending on conventions and the language at hand). A record is a collection of *elements*, typically in fixed number and sequence and typically indexed by serial numbers or identity numbers. The elements of records may also be called *fields* or *members*.

For example, a date could be stored as a record containing a numeric *year* field, a *month* field represented as a string, and a numeric *day-of-month* field. As another example, a Personnel record might contain a *name*, a *salary*, and a *rank*. As yet another example, a Circle record might contain a *center* and a *radius*. In this instance, the center itself might be represented as a Point record containing *x* and *y* coordinates.

Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

A **record type** is a data type that describes such values and variables. Most modern computer languages allow the programmer to define new record types. The definition includes specifying the data type of each field and an identifier (name or label) by which it can be accessed. In type theory, product types (with no field names) are generally preferred due to their simplicity, but proper record types are studied in languages such as System F-sub. Since type-theoretical records may contain first-class function-typed fields in addition to data, they can express many features of object-oriented programming.

Records can exist in any storage medium, including main memory and mass storage devices such as magnetic tapes or hard disks. Records are a fundamental component of most data structures, especially linked data structures. Many computer files are organized as arrays of logical records, often grouped into larger physical records or blocks for efficiency.

The parameters of a function or procedure can often be viewed as the fields of a record variable; and the arguments passed to that function can be viewed as a record value that gets assigned to that variable at the time of the call. Also, in the call stack that is often used to implement procedure calls, each entry is an *activation record* or *call frame*, containing the procedure parameters and local variables, the return address, and other internal fields.

An object in object-oriented language is essentially a record that contains procedures specialized to handle that record; and object types are an elaboration of record types. Indeed, in most object-oriented languages, records are just special cases of objects, and are known as plain old data structures (PODSs), to contrast with objects that use OO features.

A record can be viewed as the computer analog of a mathematical tuple. In the same vein, a record type can be viewed as the computer language analog of the Cartesian product of two or more mathematical sets, or the implementation of an abstract product type in a specific language.

3.1 History

The concept of record can be traced to various types of tables and ledgers used in accounting since remote times. The modern notion of records in computer science, with fields of well-defined type and size, was already implicit in 19th century mechanical calculators, such as Babbage's Analytical Engine.

Records were well established in the first half of the 20th century, when most data processing was done using punched

cards. Typically each record of a data file would be recorded in one punched card, with specific columns assigned to specific fields. Generally, a record was the smallest unit that could be read in from external storage (e.g. card reader, tape or disk).

Most machine language implementations and early assembly languages did not have special syntax for records, but the concept was available (and extensively used) through the use of index registers, indirect addressing, and self-modifying code. Some early computers, such as the IBM 1620, had hardware support for delimiting records and fields, and special instructions for copying such records.

The concept of records and fields was central in some early file sorting and tabulating utilities, such as IBM's Report Program Generator (RPG).

COBOL was the first widespread programming language to support record types,[2] and its record definition facilities were quite sophisticated at the time. The language allows for the definition of nested records with alphanumeric, integer, and fractional fields of arbitrary size and precision, as well as fields that automatically format any value assigned to them (e.g., insertion of currency signs, decimal points, and digit group separators). Each file is associated with a record variable where data is read into or written from. COBOL also provides a MOVE CORRESPONDING statement that assigns corresponding fields of two records according to their names.

The early languages developed for numeric computing, such as FORTRAN (up to FORTRAN IV) and Algol 60, did not have support for record types; but latter versions of those languages, such as Fortran 77 and Algol 68 did add them. The original Lisp programming language too was lacking records (except for the built-in cons cell), but its S-expressions provided an adequate surrogate. The Pascal programming language was one of the first languages to fully integrate record types with other basic types into a logically consistent type system. IBM's PL/1 programming language provided for COBOL-style records. The C programming language initially provided the record concept as a kind of template (struct) that could be laid on top of a memory area, rather than a true record data type. The latter were provided eventually (by the typedef declaration), but the two concepts are still distinct in the language. Most languages designed after Pascal (such as Ada, Modula, and Java) also supported records.

3.2 Operations

A programming language that supports record types usually provides some or all of the following operations:

- Declaration of a new record type, including the position, type, and (possibly) name of each field;
- Declaration of variables and values as having a given record type;
- Construction of a record value from given field values and (sometimes) with given field names;
- Selection of a field of a record with an explicit name;
- Assignment of a record value to a record variable;
- Comparison of two records for equality;
- Computation of a standard hash value for the record.

The selection of a field from a record value yields a value.

Some languages may provide facilities that enumerate all fields of a record, or at least the fields that are references. This facility is needed to implement certain services such as debuggers, garbage collectors, and serialization. It requires some degree of type polymorphism.

In systems with record subtyping, operations on values of record type may also include:

- Adding a new field to a record, setting the value of the new field.
- Removing a field from a record.

In such settings, a specific record type implies that a specific set of fields are present, but values of that type may contain additional fields. A record with fields x, y, and z would thus belong to the type of records with fields x and y, as would a record with fields x, y, and r. The rationale is that passing an (x,y,z) record to a function that expects an (x,y)

record as argument should work, since that function will find all the fields it requires within the record. Many ways of practically implementing records in programming languages would have trouble with allowing such variability, but the matter is a central characteristic of record types in more theoretical contexts.

3.2.1 Assignment and comparison

Most languages allow assignment between records that have exactly the same record type (including same field types and names, in the same order). Depending on the language, however, two record data types defined separately may be regarded as distinct types even if they have exactly the same fields.

Some languages may also allow assignment between records whose fields have different names, matching each field value with the corresponding field variable by their positions within the record; so that, for example, a complex number with fields called real and imag can be assigned to a 2D point record variable with fields X and Y. In this alternative, the two operands are still required to have the same sequence of field types. Some languages may also require that corresponding types have the same size and encoding as well, so that the whole record can be assigned as an uninterpreted bit string. Other languages may be more flexible in this regard, and require only that each value field can be legally assigned to the corresponding variable field; so that, for example, a short integer field can be assigned to a long integer field, or vice-versa.

Other languages (such as COBOL) may match fields and values by their names, rather than positions.

These same possibilities apply to the comparison of two record values for equality. Some languages may also allow order comparisons ('<'and '>'), using the lexicographic order based on the comparison of individual fields.

PL/I allows both of the preceding types of assignment, and also allows *structure expressions*, such as a = a+1; where "a" is a record, or structure in PL/I terminology.

3.2.2 Algol 68's distributive field selection

In Algol 68, if Pts was an array of records, each with integer fields X and Y, one could write Pts.Y to obtain an array of integers, consisting of the Y fields of all the elements of Pts. As a result, the statements Pts[3].Y := 7 and Pts.Y[3] := 7 would have the same effect.

3.2.3 Pascal's "with" statement

In the Pascal programming language, the command with R do S would execute the command sequence S as if all the fields of record R had been declared as variables. So, instead of writing Pt.X := 5; Pt.Y := Pt.X + 3 one could write with Pt do begin X := 5; Y := X + 3 end.

3.3 Representation in memory

The representation of records in memory varies depending on the programming languages. Usually the fields are stored in consecutive positions in memory, in the same order as they are declared in the record type. This may result in two or more fields stored into the same word of memory; indeed, this feature is often used in systems programming to access specific bits of a word. On the other hand, most compilers will add padding fields, mostly invisible to the programmer, in order to comply with alignment constraints imposed by the machine—say, that a floating point field must occupy a single word.

Some languages may implement a record as an array of addresses pointing to the fields (and, possibly, to their names and/or types). Objects in object-oriented languages are often implemented in rather complicated ways, especially in languages that allow multiple class inheritance.

3.4 Examples

The following show examples of record definitions:

- PL/I:

```
declare 1 date, 2 year picture '9999', 2 month picture '99', 2 day picture '99';
```

- C:

```
struct date { int year; int month; int day; };
```

3.5 See also

- Block (data storage)
- Composite data type
- Cons cell
- Data hierarchy
- Data structure alignment
- Object composition
- Row (database)
- struct (C programming language)
- Storage record

3.6 References

[1] Felleisen et al., *How To Design Programs*, MIT Press, 2001

[2] Sebesta, Robert W. *Concepts of Programming Languages* (Third ed.). Addison-Wesley Publishing Company, Inc. p. 218. ISBN 0-8053-7133-8.

Chapter 4

Associative array

"Dictionary (data structure)" redirects here. It is not to be confused with data dictionary.

In computer science, an **associative array**, **map**, **symbol table**, or **dictionary** is an abstract data type composed of a collection of $(key, value)$ pairs, such that each possible key appears just once in the collection.

Operations associated with this data type allow:[1][2]

- the addition of pairs to the collection
- the removal of pairs from the collection
- the modification of the values of existing pairs
- the lookup of the value associated with a particular key

The **dictionary problem** is a classic computer science problem: the task of designing a data structure that maintains a set of data during 'search' 'delete' and 'insert' operations.[3] A standard solution to the dictionary problem is a hash table; in some cases it is also possible to solve the problem using directly addressed arrays, binary search trees, or other more specialized structures.[1][2][4]

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

Associative arrays have many applications including such fundamental programming patterns as memoization and the decorator pattern.[5]

4.1 Operations

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

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

- **Add** or **insert**: add a new $(key, value)$ pair to the collection, binding the new key to its new value. The arguments to this operation are the key and the value.
- **Reassign**: replace the value in one of the $(key, value)$ pairs that are already in the collection, binding an old key to a new value. As with an insertion, the arguments to this operation are the key and the value.
- **Remove** or **delete**: remove a $(key, value)$ pair from the collection, unbinding a given key from its value. The argument to this operation is the key.

- **Lookup**: find the value (if any) that is bound to a given key. The argument to this operation is the key, and the value is returned from the operation. If no value is found, some associative array implementations raise an exception.

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

A multimap generalizes an associative array by allowing multiple values to be associated with a single key.[6] A bidirectional map is a related abstract data type in which the bindings operate in both directions: each value must be associated with a unique key, and a second lookup operation takes a value as argument and looks up the key associated with that value.

4.2 Example

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

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

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

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

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

4.3 Implementation

For dictionaries with very small numbers of bindings, it may make sense to implement the dictionary using an association list, a linked list of bindings. With this implementation, the time to perform the basic dictionary operations is linear in the total number of bindings; however, it is easy to implement and the constant factors in its running time are small.[1][7]

Another very simple implementation technique, usable when the keys are restricted to a narrow range of integers, is direct addressing into an array: the value for a given key k is stored at the array cell $A[k]$, or if there is no binding for k then the cell stores a special sentinel value that indicates the absence of a binding. As well as being simple, this technique is fast: each dictionary operation takes constant time. However, the space requirement for this structure is the size of the entire keyspace, making it impractical unless the keyspace is small.[4]

The most frequently used general purpose implementation of an associative array is with a hash table: an array of bindings, together with a hash function that maps each possible key into an array index. The basic idea of a hash table is that the binding for a given key is stored at the position given by applying the hash function to that key, and that lookup operations are performed by looking at that cell of the array and using the binding found there. However, hash table based dictionaries must be prepared to handle collisions that occur when two keys are mapped by the hash function to the same index, and many different collision resolution strategies have been developed for dealing with this situation, often based either on open addressing (looking at a sequence of hash table indices instead of a single index, until finding either the given key or an empty cell) or on hash chaining (storing a small association list instead of a single binding in each hash table cell).[1][2][4]

Dictionaries may also be stored in binary search trees or in data structures specialized to a particular type of keys such as radix trees, tries, Judy arrays, or van Emde Boas trees, but these implementation methods are less efficient

than hash tables as well as placing greater restrictions on the types of data that they can handle. The advantages of these alternative structures come from their ability to handle operations beyond the basic ones of an associative array, such as finding the binding whose key is the closest to a queried key, when the query is not itself present in the set of bindings.

4.4 Language support

Main article: Comparison of programming languages (mapping)

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 Rexx, Perl, Tcl, JavaScript, Python, Ruby, and Lua, support associative arrays as a primary container type. In many more languages, they are available as library functions without special syntax.

In Smalltalk, Objective-C, .NET,[8] Python, REALbasic, and Swift they are called *dictionaries*; in Perl, Ruby and Seed7 they are called *hashes*; in C++, Java, Go, Clojure, Scala, OCaml, Haskell they are called *maps* (see map (C++), unordered_map (C++), and Map); in Common Lisp and Windows PowerShell, they are called *hash tables* (since both typically use this implementation). In PHP, all arrays can be associative, except that the keys are limited to integers and strings. In JavaScript (see also JSON), all objects behave as associative arrays. In Lua, they are called *tables*, and are used as the primitive building block for all data structures. In Visual FoxPro, they are called *Collections*. The D language also has support for associative arrays [9]

4.5 See also

- Tuple
- Function (mathematics)
- Key-value data store
- JSON

4.6 References

[1] Goodrich, Michael T.; Tamassia, Roberto (2006), "9.1 The Map Abstract Data Type", *Data Structures & Algorithms in Java* (4th ed.), Wiley, pp. 368–371.

[2] Mehlhorn, Kurt; Sanders, Peter (2008), "4 Hash Tables and Associative Arrays", *Algorithms and Data Structures: The Basic Toolbox*, Springer, pp. 81–98.

[3] Anderson, Arne (1989). "Optimal Bounds on the Dictionary Problem". *Proc. Symposium on Optimal Algorithms* (Springer Verlag): 106–114.

[4] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "11 Hash Tables", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 221–252, ISBN 0-262-03293-7 .

[5] Goodrich & Tamassia (2006), pp. 597–599.

[6] Goodrich & Tamassia (2006), pp. 389–397.

[7] "When should I use a hash table instead of an association list?". lisp-faq/part2. 1996-02-20.

[8] "Dictionary<TKey, TValue> Class". MSDN.

[9] "Associative Arrays, the D programming language". Digital Mars.

4.7 External links

- NIST's Dictionary of Algorithms and Data Structures: Associative Array

Chapter 5

Union type

In computer science, a **union** is a value that may have any of several representations or formats; or it is a data structure that consists of a variable that may hold such a value. Some programming languages support special data types, called **union types**, to describe such values and variables. In other words, a union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g., "float or long integer". Contrast with a record (or structure), which could be defined to contain a float *and* an integer; in a union, there is only one value at any given time.

A union can be pictured as a chunk of memory that is used to store variables of different data types. Once a new value is assigned to a field, the existing data is overwritten with the new data. The memory area storing the value has no intrinsic type (other than just bytes or words of memory), but the value can be treated as one of several abstract data types, having the type of the value that was last written to the memory area.

In type theory, a union has a sum type.

Depending on the language and type, a union value may be used in some operations, such as assignment and comparison for equality, without knowing its specific type. Other operations may require that knowledge, either by some external information, or by the use of a tagged union.

5.1 Untagged unions

Because of the limitations of their use, untagged unions are generally only provided in untyped languages or in a type-unsafe way (as in C). They have the advantage over simple tagged unions of not requiring space to store a data type tag.

The name "union" stems from the type's formal definition. If a type is considered as the set of all values that that type can take on, a union type is simply the mathematical union of its constituting types, since it can take on any value any of its fields can. Also, because a mathematical union discards duplicates, if more than one field of the union can take on a single common value, it is impossible to tell from the value alone which field was last written.

However, one useful programming function of unions is to map smaller data elements to larger ones for easier manipulation. A data structure consisting, for example, of 4 bytes and a 32-bit integer, can form a union with an unsigned 64-bit integer, and thus be more readily accessed for purposes of comparison etc.

5.2 Unions in various programming languages

5.2.1 C/C++

In C and C++, untagged unions are expressed nearly exactly like structures (structs), except that each data member begins at the same location in memory. The data members, as in structures, need not be primitive values, and in fact may be structures or even other unions. However, C++ does not allow for a data member to be any type that has a full fledged constructor/destructor and/or copy constructor, or a non-trivial copy assignment operator. For example, it is impossible to have the standard C++ string as a member of a union. (C++11 lifts some of these restrictions.)

Like a structure, all of the members of a union are by default public. The keywords private, public, and protected may be used inside a structure or a union in exactly the same way they are used inside a class for defining private, public, and protected member access.

The primary use of a union is allowing access to a common location by different data types, for example hardware input/output access, perhaps bitfield and word sharing. Unions also provide crude polymorphism. However, there is no checking of types, so it is up to the programmer to be sure that the proper fields are accessed in different contexts. The relevant field of a union variable is typically determined by the state of other variables, possibly in an enclosing struct.

One common C programming idiom uses unions to perform what C++ calls a **reinterpret_cast**, by assigning to one field of a union and reading from another, as is done in code which depends on the raw representation of the values. A practical example is the method of computing square roots using the IEEE representation. This is not, however, a safe use of unions in general.

> Structure and union specifiers have the same form. [. . .] The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.
> —ANSI/ISO 9899:1990 (the ANSI C standard) Section 6.5.2.1

Anonymous union

Unions can also be anonymous; that is, they do not have a name. Their data members are accessed directly. In addition to this, they have certain other restrictions like:

- They must also be declared as static if declared in file scope. If declared in local scope, they must be static or automatic.
- They can have only public members; private and protected members in anonymous unions generate errors.
- They cannot have function members.

Simply omitting the class-name portion of the syntax does not make a union an anonymous union. For a union to qualify as an anonymous union, the declaration must not declare an object. Example:

// anonymous_unions.cpp #include <iostream> using namespace std; int main() { union { int d; char f; }; d = 4; cout << d << endl; f = 'i'; cout << f << endl; }

5.2.2 COBOL

In COBOL, union data items are defined in two ways. The first uses the RENAMES (66 level) keyword, which effectively maps a second alphanumeric data item on top of the same memory location as a preceding data item. In the example code below, data item PERSON-REC is defined as a group containing another group and a numeric data item. PERSON-DATA is defined as an alphanumeric data item that renames PERSON-REC, treating the data bytes continued within it as character data.

01 PERSON-REC. 05 PERSON-NAME. 10 PERSON-NAME-LAST PIC X(12). 10 PERSON-NAME-FIRST PIC X(16). 10 PERSON-NAME-MID PIC X. 05 PERSON-ID PIC 9(9) PACKED-DECIMAL. 01 PERSON-DATA RENAMES PERSON-REC.

The second way to define a union type is by using the REDEFINES keyword. In the example code below, data item VERS-NUM is defined as a 2-byte binary integer containing a version number. A second data item VERS-BYTES is defined as a two-character alphanumeric variable. Since the second item is *redefined* over the first item, the two items share the same address in memory, and therefore share the same underlying data bytes. The first item interprets the two data bytes as a binary value, while the second item interprets the bytes as character values.

01 VERS-INFO. 05 VERS-NUM PIC S9(4) COMP. 05 VERS-BYTES PIC X(2) REDEFINES VERS-NUM.

5.3 Syntax and Example

In C and C++, the syntax is:

union <name> { <datatype> <1st variable name>; <datatype> <2nd variable name>; . . . <datatype> <nth variable name>; } <union variable name>;

A structure can also be a member of a union, as the following example shows:

union name1 { struct name2 { int a; float b; char c; } svar; int d; } uvar;

This example defines a variable uvar as a union (tagged as name1), which contains two members, a structure (tagged as name2) named svar (which in turn contains three members), and an integer variable named d.

Unions may occur within structures and arrays, and vice versa:

struct { int flags; char *name; int utype; union { int ival; float fval; char *sval; } u; } symtab[NSYM];

The number ival is referred to as symtab[i].u.ival and the first character of string sval by either of *symtab[i].u.sval or symtab[i].u.sval[0].

5.4 Difference between Union and Structure

A union is a class all of whose data members are mapped to the same address within its object. The size of an object of a union is, therefore, the size of its largest data member.

In a structure, all of its data members are stored in contiguous memory locations. The size of an object of a struct is, therefore, the size of the sum of all its data members.

This gain in space efficiency, while valuable in certain circumstances, comes at a great cost of safety: the program logic must ensure that it only reads the field most recently written along all possible execution paths. The exception is when unions are used for type conversion: in this case, a certain field is written and the subsequently read field is deliberately different.

An example illustrating this point is:

+-----+-----+ struct { int a; float b; } gives | a | b | +-----+-----+ ^ ^ | | memory location: 150 154 | ˇ +-----+ union { int a; float b; } gives | a | | b | +-----+

Structures are used where an “object” is composed of other objects, like a point object consisting of two integers, those being the x and y coordinates:

typedef struct { int x; // x and y are separate int y; } tPoint;

Unions are typically used in situation where an object can be one of many things but only one at a time, such as a type-less storage system:

typedef enum { STR, INT } tType; typedef struct { tType typ; // typ is separate. union { int ival; // ival and sval occupy same memory. char *sval; }; } tVal;

5.5 See also

- Tagged union
- UNION operator

5.6 External links

- boost::variant, a type-safe alternative to C++ unions
- MSDN: Classes,Structures & Unions, for examples and syntax
- differences, differences between union & structure
- Difference between struct and union in C++

Chapter 6

Set (abstract data type)

In computer science, a **set** is an abstract data type that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set. Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set.

Some set data structures are designed for **static** or **frozen sets** that do not change after they are constructed. Static sets allow only query operations on their elements — 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 deletion of elements from the set.

An abstract data structure is a collection, or aggregate, of data. The data may be booleans, numbers, characters, or other data structures. If one considers the structure yielded by packaging[lower-alpha 1] or indexing,[lower-alpha 2] there are four basic data structures:[1][2]

1. unpackaged, unindexed: bunch
2. packaged, unindexed: set
3. unpackaged, indexed: string (sequence)
4. packaged, indexed: list (array)

In this view, the contents of a set are a bunch, and isolated data items are elementary bunches (elements). Whereas sets *contain* elements, bunches *consist of* elements.

Further structuring may be achieved by considering the multiplicity of elements (sets become multisets, bunches become hyperbunches)[3] or their homogeneity (a record is a set of fields, not necessarily all of the same type).

6.1 Type theory

In type theory, sets are generally identified with their indicator function (characteristic 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.) The characteristic function F of a set S is defined as:

$$F(x) = \begin{cases} 1, & \text{if } x \in S \\ 0, & \text{if } x \notin S \end{cases}$$

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

6.2 Operations

6.2.1 Core set-theoretical operations

One may define the operations of the algebra of sets:

- union(S,T): returns the union of sets S and T.
- intersection(S,T): returns the intersection of sets S and T.
- difference(S,T): returns the difference of sets S and T.
- subset(S,T): a predicate that tests whether the set S is a subset of set T.

6.2.2 Static sets

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

- is_element_of(x,S): checks whether the value x is in the set S.
- is_empty(S): checks whether the set S is empty.
- size(S) or cardinality(S): returns the number of elements in S.
- iterate(S): returns a function that returns one more value of S at each call, in some arbitrary order.
- enumerate(S): returns a list containing the elements of S in some arbitrary order.
- build(x_1,x_2,…,xn,): creates a set structure with values x_1,x_2,…,xn.
- create_from(*collection*): creates a new set structure containing all the elements of the given collection or all the elements returned by the given iterator.

6.2.3 Dynamic sets

Dynamic set structures typically add:

- create(): creates a new, initially empty set structure.
  - create_with_capacity(n): creates a new set structure, initially empty but capable of holding up to n elements.
- add(S,x): adds the element x to S, if it is not present already.
- remove(S, x): removes the element x from S, if it is present.
- 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.

6.2.4 Additional operations

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

- pop(S): returns an arbitrary element of S, deleting it from S.[4]
- pick(S): returns an arbitrary element of S.[5][6][7] Functionally, the mutator pop can be interpreted as the pair of selectors (pick, rest), where rest returns the set consisting of all elements except for the arbitrary element.[8] Can be interpreted in terms of iterate.[lower-alpha 3]

- map(F,S): returns the set of distinct values resulting from applying function F to each element of S.
- filter(P,S): returns the subset containing all elements of S that satisfy a given predicate P.
- fold(A_0,F,S): returns the value $A|S|$ after applying $Ai+1 := F(Ai, e)$ for each element e of S, for some binary operation F. F must be associative and commutative for this to be well-defined.
- clear(S): delete all elements of S.
- equal(S_1, S_2): checks whether the two given sets are equal (i.e. contain all and only the same elements).
- hash(S): returns a hash value for the static set S such that if equal(S_1, S_2) then hash(S_1) = hash(S_2)

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

- sum(S): returns the sum of all elements of S for some definition of "sum". For example, over integers or reals, it may be defined as fold(0, add, S).
- collapse(S): given a set of sets, return the union.[9] For example, collapse({{1}, {2, 3}}) == {1, 2, 3}. May be considered a kind of sum.
- flatten(S): given a set consisting of sets and atomic elements (elements that are not sets), returns a set whose elements are the atomic elements of the original top-level set or elements of the sets it contains. In other words, remove a level of nesting – like collapse, but allow atoms. This can be done a single time, or recursively flattening to obtain a set of only atomic elements.[10] For example, flatten({1, {2, 3}}) == {1, 2, 3}.
- nearest(S,x): returns the element of S that is closest in value to x (by some metric).
- min(S), max(S): returns the minimum/maximum element of S.

6.3 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 operations. A simple implementation is to use a list, ignoring the order of the elements and taking care to avoid repeated values. This is simple but inefficient, as operations like set membership or element deletion are $O(n)$, as they require scanning the entire list.[lower-alpha 4] Sets are often instead implemented using more efficient data structures, particularly various flavors of trees, tries, or hash tables.

As sets can be interpreted as a kind of map (by the indicator function), sets are commonly implemented in the same way as (partial) maps (associative arrays) – in this case in which the value of each key-value pair has the unit type or a sentinel value (like 1) – 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[11] may be used to provide deterministically ordered sets.

Further, in languages that support maps but not sets, sets can be implemented in terms of maps. For example, a common programming idiom in Perl that converts an array to a hash whose values are the sentinel value 1, for use as a set, is:

```
my %elements = map { $_ => 1 } @elements;
```

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

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

6.4 Language support

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

- Java offers the Set interface to support sets (with the HashSet class implementing it using a hash table), and the SortedSet sub-interface to support sorted sets (with the TreeSet class implementing it using a binary search tree).
- Apple's Foundation framework (part of Cocoa) provides the Objective-C classes NSSet, NSMutableSet, NSCountedSet, NSOrderedSet, and NSMutableOrderedSet. The CoreFoundation APIs provide the CFSet and CFMutableSet types for use in C.
- Python has built-in set and frozenset types since 2.4, and since Python 3.0 and 2.7, supports non-empty set literals using a curly-bracket syntax, e.g.: {x, y, z}.
- The .NET Framework provides the generic HashSet and SortedSet classes that implement the generic ISet interface.
- Smalltalk's class library includes Set and IdentitySet, using equality and identity for inclusion test respectively. Many dialects provide variations for compressed storage (NumberSet, CharacterSet), for ordering (OrderedSet, SortedSet, etc.) or for weak references (WeakIdentitySet).
- Ruby's standard library includes a set module which contains Set and SortedSet classes that implement sets using hash tables, the latter allowing iteration in sorted order.
- OCaml's standard library contains a Set module, which implements a functional set data structure using binary search trees.
- The GHC implementation of Haskell provides a Data.Set module, which implements a functional set data structure using binary search trees.
- The Tcl Tcllib package provides a set module which implements a set data structure based upon TCL lists.
- The Swift standard library contains a Set type, since Swift 1.2.

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.

6.4.1 In C++

Main article: set (C++)

In C++, the Standard Template Library (STL) provides the set template class, which implements a sorted set using a binary search tree; SGI's STL also provides the hash_set template class, which implements a set using a hash table.

In sets, the elements themselves are the keys, in contrast to sequenced containers, where elements are accessed using their (relative or absolute) position. Set elements must have a strict weak ordering.

6.5 Multiset

A generalization of the notion of a set is that of a **multiset** or **bag**, which is similar to a set but allows repeated ("equal") values (duplicates). This is used in two distinct senses: either equal values are considered *identical,* and are simply counted, or equal values are considered *equivalent,* and are stored as distinct items. For example, given a list of people (by name) and ages (in years), one could construct a multiset of ages, which simply counts the number of people of a given age. Alternatively, one can construct a multiset of people, where two people are considered equivalent if their ages are the same (but may be different people and have different names), in which case each pair (name, age) must be stored, and selecting on a given age gives all the people of a given age.

Formally, it is possible for objects in computer science to be considered "equal" under some equivalence relation but still distinct under another relation. Some types of multiset implementations will store distinct equal objects as separate items in the data structure; while others will collapse it down to one version (the first one encountered) and keep a positive integer count of the multiplicity of the element.

As with sets, multisets can naturally be implemented using hash table or trees, which yield different performance characteristics.

The set of all bags over type T is given by the expression bag T. If by multiset one considers equal items identical and simply counts them, then a multiset can be interpreted as a function from the input domain to the non-negative integers (natural numbers), generalizing the identification of a set with its indicator function. In some cases a multiset in this counting sense may be generalized to allow negative values, as in Python.

- C++'s Standard Template Library implements both sorted and unsorted multisets. It provides the multiset class for the sorted multiset, as a kind of associative container, which implements this multiset using a self-balancing binary search tree. It provides the unordered_multiset class for the unsorted multiset, as a kind of unordered associative containers, which implements this multiset using a hash table. The unsorted multiset is standard as of C++11; previously SGI's STL provides the hash_multiset class, which was copied and eventually standardized.
- For Java, third-party libraries provide multiset functionality:
  - Apache Commons Collections provides the Bag and SortedBag interfaces, with implementing classes like HashBag and TreeBag.
  - Google Guava provides the Multiset interface, with implementing classes like HashMultiset and TreeMultiset.
- Apple provides the NSCountedSet class as part of Cocoa, and the CFBag and CFMutableBag types as part of CoreFoundation.
- Python's standard library includes collections.Counter, which is similar to a multiset.
- Smalltalk includes the Bag class, which can be instantiated to use either identity or equality as predicate for inclusion test.

Where a multiset data structure is not available, a workaround is to use a regular set, but override the equality predicate of its items to always return "not equal" on distinct objects (however, such will still not be able to store multiple occurrences of the same object) or use an associative array mapping the values to their integer multiplicities (this will not be able to distinguish between equal elements at all).

Typical operations on bags:

- contains(B, x): checks whether the element x is present (at least once) in the bag B
- is_sub_bag(B_1, B_2): checks whether each element in the bag B_1 occurs in B_1 no more often than it occurs in the bag B_2; sometimes denoted as $B_1 \sqsubseteq B_2$.
- count(B, x): returns the number of times that the element x occurs in the bag B; sometimes denoted as $B \# x$.
- scaled_by(B, n): given a natural number n, returns a bag which contains the same elements as the bag B, except that every element that occurs m times in B occurs $n * m$ times in the resulting bag; sometimes denoted as n ⊗ B.
- union(B_1, B_2): returns a bag that containing just those values that occur in either the bag B_1 or the bag B_2, except that the number of times a value x occurs in the resulting bag is equal to (B_1 # x) + (B_2 # x); sometimes denoted as $B_1 \uplus B_2$.

6.5.1 Multisets in SQL

In relational databases, a table can be a (mathematical) set or a multiset, depending on the presence on unicity constraints on some columns (which turns it into a candidate key).

SQL allows the selection of rows from a relational table: this operation will in general yield a multiset, unless the keyword DISTINCT is used to force the rows to be all different, or the selection includes the primary (or a candidate) key.

In ANSI SQL the MULTISET keyword can be used to transform a subquery into a collection expression:

SELECT expression1, expression2... FROM TABLE_NAME...

is a general select that can be used as *subquery expression* of another more general query, while

MULTISET(SELECT expression1, expression2... FROM TABLE_NAME...)

transforms the subquery into a *collection expression* that can be used in another query, or in assignment to a column of appropriate collection type.

6.6 See also

- Bloom filter
- Disjoint set

6.7 Notes

[1] "Packaging" consists in supplying a container for an aggregation of objects in order to turn them into a single object. Consider a function call: without packaging, a function can be called to act upon a bunch only by passing each bunch element as a separate argument, which complicates the function's signature considerably (and is just not possible in some programming languages). By packaging the bunch's elements into a set, the function may now be called upon a single, elementary argument: the set object (the bunch's package).

[2] Indexing is possible when the elements being considered are totally ordered. Being without order, the elements of a multiset (for example) do not have lesser/greater or preceding/succeeding relationships: they can only be compared in absolute terms (same/different).

[3] For example, in Python pick can be implemented on a derived class of the built-in set as follows:
class Set(set): def pick(self): return next(iter(self))

[4] Element insertion can be done in $O(1)$ time by simply inserting at an end, but if one avoids duplicates this takes $O(n)$ time.

6.8 References

[1] Hehner, Eric C. R. (1981), "Bunch Theory: A Simple Set Theory for Computer Science", *Information Processing Letters* **12** (1): 26, doi:10.1016/0020-0190(81)90071-5

[2] Hehner, Eric C. R. (2004), *A Practical Theory of Programming, second edition*

[3] Hehner, Eric C. R. (2012), *A Practical Theory of Programming, 2012-3-30 edition*

[4] Python: pop()

[5] *Management and Processing of Complex Data Structures: Third Workshop on Information Systems and Artificial Intelligence, Hamburg, Germany, February 28 - March 2, 1994. Proceedings,* ed. Kai v. Luck, Heinz Marburger, p. 76

[6] Python Issue7212: Retrieve an arbitrary element from a set without removing it; see msg106593 regarding standard name

[7] Ruby Feature #4553: Add Set#pick and Set#pop

[8] *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning,* Ute Schmid, Springer, Aug 21, 2003, p. 240

[9] *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, S. Margherita, Italy, May 30 - June 3, 1994. Selected Papers, Volume 10,* ed. Egidio Astesiano, Gianna Reggio, Andrzej Tarlecki, p. 38

[10] Ruby: flatten()

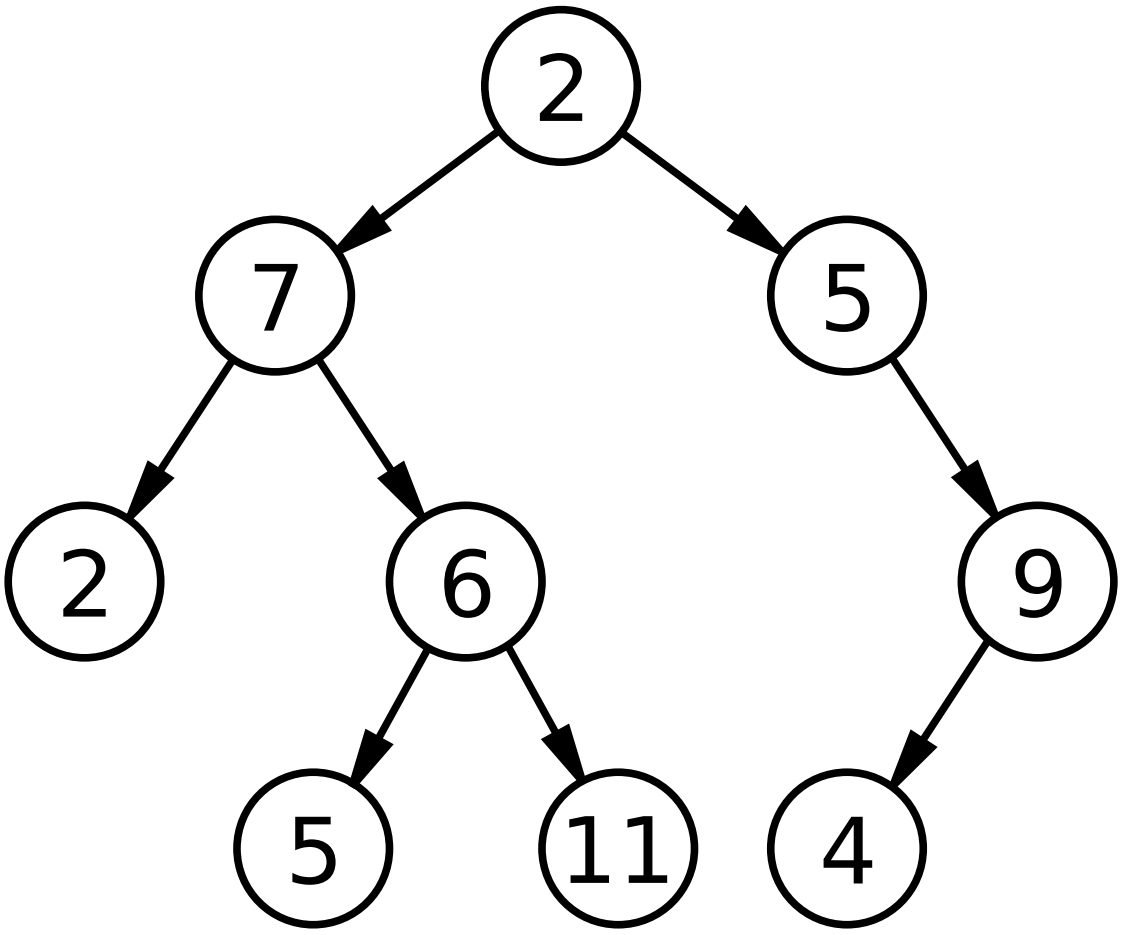
[11] Wang, Thomas (1997), *Sorted Linear Hash Table*

Chapter 7

Tree (data structure)

Not to be confused with trie, a specific type of tree data structure.

In computer science, a **tree** is a widely used abstract data type (ADT) or data structure implementing this ADT that



A simple unordered tree; in this diagram, the node labeled 7 has two children, labeled 2 and 6, and one parent, labeled 2. The root node, at the top, has no parent.

simulates a hierarchical tree structure, with a root value and subtrees of children, represented as a set of linked nodes.

A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

Alternatively, a tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each

node. Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a list of nodes and an adjacency list of edges between nodes, as one may represent a digraph, for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

7.1 Definition

A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

7.2 Terminologies used in Trees

- **Root –** The top node in a tree.
- **Parent –** The converse notion of *child*.
- **Siblings –** Nodes with the same parent.
- **Descendant –** a node reachable by repeated proceeding from parent to child.
- **Ancestor –** a node reachable by repeated proceeding from child to parent.
- **Leaf –** a node with no children.
- **Internal node –** a node with at least one child.
- **External node –** a node with no children.
- **Degree –** number of sub trees of a node.
- **Edge –** connection between one node to another.
- **Path –** a sequence of nodes and edges connecting a node with a descendant.
- **Level –** The level of a node is defined by 1 + the number of connections between the node and the root.
- **Height of tree –**The height of a tree is the number of edges on the longest downward path between the root and a leaf.
- **Height of node –**The height of a node is the number of edges on the longest downward path between that node and a leaf.
- **Depth –**The depth of a node is the number of edges from the node to the tree's root node.
- **Forest –** A forest is a set of $n \geq 0$ disjoint trees.

7.2.1 Data type vs. data structure

There is a distinction between a tree as an abstract data type and as a concrete data structure, analogous to the distinction between a list and a linked list.

As a data type, a tree has a value and children, and the children are themselves trees; the value and children of the tree are interpreted as the value of the root node and the subtrees of the children of the root node. To allow finite trees, one must either allow the list of children to be empty (in which case trees can be required to be non-empty, an "empty tree" instead being represented by a forest of zero trees), or allow trees to be empty, in which case the list of children can be of fixed size (branching factor, especially 2 or "binary"), if desired.

As a data structure, a linked tree is a group of nodes, where each node has a value and a list of references to other nodes (its children). This data structure actually defines a directed graph,[lower-alpha 1] because it may have loops or

several references to the same node, just as a linked list may have a loop. Thus there is also the requirement that no two references point to the same node (that each node has at most a single parent, and in fact exactly one parent, except for the root), and a tree that violates this is "corrupt".

Due to the use of *references* to trees in the linked tree data structure, trees are often discussed implicitly assuming that they are being represented by references to the root node, as this is often how they are actually implemented. For example, rather than an empty tree, one may have a null reference: a tree is always non-empty, but a reference to a tree may be null.

7.2.2 Recursive

Recursively, as a data type a tree is defined as a value (of some data type, possibly empty), together with a list of trees (possibly an empty list), the subtrees of its children; symbolically:

t: v [t[1], ..., t[k]]

(A tree *t* consists of a value *v* and a list of other trees.)

More elegantly, via mutual recursion, of which a tree is one of the most basic examples, a tree can be defined in terms of a forest (a list of trees), where a tree consists of a value and a forest (the subtrees of its children):

f: [t[1], ..., t[k]] t: v f

Note that this definition is in terms of values, and is appropriate in functional languages (it assumes referential transparency); different trees have no connections, as they are simply lists of values.

As a data structure, a tree is defined as a node (the root), which itself consists of a value (of some data type, possibly empty), together with a list of references to other nodes (list possibly empty, references possibly null); symbolically:

n: v [&n[1], ..., &n[k]]

(A node *n* consists of a value *v* and a list of references to other nodes.)

This data structure defines a directed graph,[lower-alpha 2] and for it to be a tree one must add a condition on its global structure (its topology), namely that at most one reference can point to any given node (a node has at most a single parent), and no node in the tree point to the root. In fact, every node (other than the root) must have exactly one parent, and the root must have no parents.

Indeed, given a list of nodes, and for each node a list of references to its children, one cannot tell if this structure is a tree or not without analyzing its global structure and that it is in fact topologically a tree, as defined below.

7.2.3 Type theory

As an ADT, the abstract tree type T with values of some type E is defined, using the abstract forest type F (list of trees), by the functions:

value: $T \to E$

children: $T \to F$

nil: $() \to F$

node: $E \times F \to T$

with the axioms:

value(node(e, f)) = e

children(node(e, f)) = f

In terms of type theory, a tree is an inductive type defined by the constructors *nil* (empty forest) and *node* (tree with root node with given value and children).

7.2.4 Mathematical

Viewed as a whole, a tree data structure is an ordered tree, generally with values attached to each node. Concretely, it is (if required to be non-empty):

- A rooted tree with the “away from root” direction (a more narrow term is an "arborescence"), meaning:
  - A directed graph,
  - whose underlying undirected graph is a tree (any two vertices are connected by exactly one simple path),
  - with a distinguished root (one vertex is designated as the root),
  - which determines the direction on the edges (arrows point away from the root; given an edge, the node that the edge points from is called the *parent* and the node that the edge points to is called the *child*),

together with:

- an ordering on the child nodes of a given node, and
- a value (of some data type) at each node.

Often trees have a fixed (more properly, bounded) branching factor (outdegree), particularly always having two child nodes (possibly empty, hence *at most* two *non-empty* child nodes), hence a “binary tree”.

Allowing empty trees makes some definitions simpler, some more complicated: a rooted tree must be non-empty, hence if empty trees are allowed the above definition instead becomes “an empty tree, or a rooted tree such that ...”. On the other hand, empty trees simplify defining fixed branching factor: with empty trees allowed, a binary tree is a tree such that every node has exactly two children, each of which is a tree (possibly empty).The complete sets of operations on tree must include fork operation.

7.3 Terminology

A **node** is a structure which may contain a value or 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 are drawn growing downwards). 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.

An **internal node** (also known as an **inner node**, **inode** for short, or **branch node**) is any node of a tree that has child nodes. Similarly, an **external node** (also known as an **outer node**, **leaf node**, or **terminal node**) is any node that does not have child nodes.

The topmost node in a tree is called the **root node**. Depending on definition, a tree may be required to have a root node (in which case all trees are non-empty), or may be allowed to be empty, in which case it does not necessarily have a root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin, since as a data structure, one can only pass from parents to children. Note that some algorithms (such as post-order depth-first search) begin at the root, but first visit leaf nodes (access the value of leaf nodes), only visit the root last (i.e., they first access the children of the root, but only access the *value* of the root last). 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, the root node is conventionally 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.

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. The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes, if such are allowed) has depth and height −1.

A **subtree** of a tree T is a tree consisting of a node in T and all of its descendants in T.[lower-alpha 3][1] Nodes thus correspond to subtrees (each node corresponds to the subtree of itself and all its descendants) – the subtree corresponding to the root node is the entire tree, and each node is the root node of the subtree it determines; the subtree corresponding to any other node is called a **proper subtree** (by analogy to a proper subset).

7.4 Drawing Trees

Trees are often drawn in the plane. Ordered trees can be represented essentially uniquely in the plane, and are hence called *plane trees,* as follows: if one fixes a conventional order (say, counterclockwise), and arranges the child nodes in that order (first incoming parent edge, then first child edge, etc.), this yields an embedding of the tree in the plane, unique up to ambient isotopy. Conversely, such an embedding determines an ordering of the child nodes.

If one places the root at the top (parents above children, as in a family tree) and places all nodes that are a given distance from the root (in terms of number of edges: the "level" of a tree) on a given horizontal line, one obtains a standard drawing of the tree. Given a binary tree, the first child is on the left (the "left node"), and the second child is on the right (the "right node").

7.5 Representations

There are many different ways to represent trees; common representations represent the nodes as dynamically allocated records 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).

Indeed, a binary tree can be implemented as a list of lists (a list where the values are lists): the head of a list (the value of the first term) is the left child (subtree), while the tail (the list of second and future terms) is the right child (subtree). This can be modified to allow values as well, as in Lisp S-expressions, where the head (value of first term) is the value of the node, the head of the tail (value of second term) is the left child, and the tail of the tail (list of third and future terms) is the right child.

In general a node in a tree will not have pointers to its parents, but this information can be included (expanding the data structure to also include a pointer to the parent) or stored separately. Alternatively, upward links can be included in the child node data, as in a threaded binary tree.

7.6 Generalizations

7.6.1 Digraphs

If edges (to child nodes) are thought of as references, then a tree is a special case of a digraph, and the tree data structure can be generalized to represent directed graphs by removing the constraints that a node may have at most one parent, and that no cycles are allowed. Edges are still abstractly considered as pairs of nodes, however, the terms *parent* and *child* are usually replaced by different terminology (for example, *source* and *target*). Different implementation strategies exist: a digraph can be represented by the same local data structure as a tree (node with value and list of children), assuming that "list of children" is a list of references, or globally by such structures as adjacency lists.

In graph theory, a tree is a connected acyclic graph; unless stated otherwise, in graph theory trees and graphs are assumed 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.

Given a node in a tree, its children define an ordered forest (the union of subtrees given by all the children, or equivalently taking the subtree given by the node itself and erasing the root). Just as subtrees are natural for recursion (as in a depth-first search), forests are natural for corecursion (as in a breadth-first search).

Via mutual recursion, a forest can be defined as a list of trees (represented by root nodes), where a node (of a tree) consists of a value and a forest (its children):

f: [n[1], ..., n[k]] n: v f

7.7 Traversal methods

Main article: Tree traversal

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; a walk in which a node's left subtree, then the node itself, and finally its right subtree are traversed is called an **in-order** traversal. (This last scenario, referring to exactly two subtrees, a left subtree and a right subtree, assumes specifically a binary tree.) A **level-order** walk effectively performs a breadth-first search over the entirety of a tree; nodes are traversed level by level, where the root node is visited first, followed by its direct child nodes and their siblings, followed by its grandchild nodes and their siblings, etc., until all nodes in the tree have been traversed.

7.8 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
- **Pruning**: Removing a whole section of a tree
- **Grafting**: Adding a whole section to a tree
- Finding the root for any node

7.9 Common uses

- Representing hierarchical data
- Storing data in a way that makes it easily searchable (see binary search tree and tree traversal)
- Representing sorted lists of data
- As a workflow for compositing digital images for visual effects
- Routing algorithms

7.10 See also

- Tree structure
- Tree (graph theory)
- Tree (set theory)
- Hierarchy (mathematics)
- Dialog tree
- Single inheritance

7.10.1 Other trees

- DSW algorithm
- Enfilade
- Left child-right sibling binary tree
- Hierarchical temporal memory

7.11 Notes

[1] Properly, a rooted, ordered directed graph.

[2] Properly, a rooted, ordered directed graph.

[3] This is different from the formal definition of subtree used in graph theory, which is a subgraph that forms a tree – it need not include all descendants. For example, the root node by itself is a subtree in the graph theory sense, but not in the data structure sense (unless there are no descendants).

7.12 References

[1] Weisstein, Eric W., "Subtree", *MathWorld*.

- 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.

7.13 External links

- Data Trees as a Means of Presenting Complex Data Analysis by Sally Knipe
- Description from the Dictionary of Algorithms and Data Structures
- STL-like C++ tree class
- Description of tree data structures from ideainfo.8m.com
- WormWeb.org: Interactive Visualization of the *C. elegans* Cell Tree – Visualize the entire cell lineage tree of the nematode *C. elegans* (javascript)
- [ref : http://www.allisons.org/ll/AlgDS/Tree/]

Chapter 8

Bit field

A **bit field** is a term used in computer programming to store multiple, logical, neighboring bits, where each of the sets of bits, and single bits can be addressed. A bit field is most commonly used to represent integral types of known, fixed bit-width. A well-known usage of bit-fields is to represent a set of bits, and/or series of bits, known as flags. For example, the first bit in a bit field can be used to determine the state of a particular attribute associated with the bit field.

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.

8.1 Implementation

“A bit field is set up with a structure declaration that labels each field and determines its width.”[1] In C and C++ bit fields can be created using unsigned int, signed int, or _Bool (in C99).

You can set, test, and change the bits in the field using a mask, bitwise operators, and the proper membership operator of a struct (. or ->). ORing a value will turn the bits on if they are not already on, and leave them unchanged if they are, e.g. bf.flag |= MASK; To turn a bit off, you can AND its inverse, e.g. bf->flag &= ~MASK; And finally you can toggle a bit (turn it on if it is off and off if it is on) with the XOR operator, e.g. (*bf).flag ^= MASK; To test a bit you can use an AND expression, e.g. (flag_set & MASK) ? true : false;

Having the value of a particular bit can be simply done by left shifting (<<) 1, n amount of times (or, x << n - log2(x) amount of times, where x is a power of 2), where n is the index of the bit you want (the right most bit being the start), e.g. if you want the value of the 4th bit in a binary number, you can do: 1 << 3; which will yield 8, or 2 << 2; etc. The benefits of this become apparent when iterating through a series of bits one at a time in a for loop, or when needing the powers of large numbers to check high bits.

If a language doesn't support bit fields, but supports bit manipulation, you can do something very similar. Since a bit field is just a group of neighboring bits, and so is any other primitive data type, you can substitute the bit field for a primitive, or array of primitives. For example, with a 32 bit integer represented as 32 contiguous bits, you could use it the same way as a bit field with one difference; with a bitfield you can represent a particular bit or set of bits using its named member, and a flag whose value is between 0, and 2 to the nth power, where n is the length of the bits.

8.2 Examples

Declaring a bit field in C:

#include <stdio.h> // opaque and show #define YES 1 #define NO 0 // line styles #define SOLID 1 #define DOTTED 2 #define DASHED 3 // primary colors #define BLUE 4 /* 100 */ #define GREEN 2 /* 010 */ #define RED 1 /* 001 */ // mixed colors #define BLACK 0 /* 000 */ #define YELLOW (RED | GREEN) /* 011 */ #define MAGENTA (RED | BLUE) /* 101 */ #define CYAN (GREEN | BLUE) /* 110 */ #define WHITE (RED | GREEN | BLUE) /* 111 */ const char * colors[8] = {"Black”, “Red”, “Green”, “Yellow”, “Blue”, " Magenta”, “Cyan”, “White"}; // bit

field box properties struct box_props { unsigned int opaque : 1; unsigned int fill_color : 3; unsigned int : 4; // fill to 8 bits unsigned int show_border : 1; unsigned int border_color : 3; unsigned int border_style : 2; unsigned int : 2; // fill to 16 bits };

[2]

Example of emulating bit fields with a primitive and bit operators in C:

/* Each prepocessor directive defines a single bit */ #define KEY_UP (1 << 0) /* 000001 */ #define KEY_RIGHT (1 << 1) /* 000010 */ #define KEY_DOWN (1 << 2) /* 000100 */ #define KEY_LEFT (1 << 3) /* 001000 */ #define KEY_BUTTON1 (1 << 4) /* 010000 */ #define KEY_BUTTON2 (1 << 5) /* 100000 */ int gameControllerStatus = 0; /* Sets the gameCtrollerStatus using OR */ void keyPressed(int key) { gameControllerStatus |= key; } /* Turns the key in gameControllerStatus off using AND and ~ */ void keyReleased(int key) { gameControllerStatus &= ~key; } /* Tests whether a bit is set using AND */ int isPressed(int key) { return gameControllerStatus & key; }

8.3 See also

- Mask (computing)
- Bitboard, used in chess and similar games.
- Bit array
- Flag word

8.4 External links

- Explanation from a book
- Description from another wiki
- Use case in a C++ guide
- C++ libbit bit library (alternative URL)

8.5 References

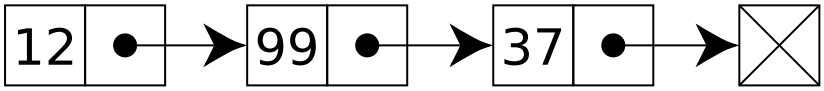
[1] Prata, Stephen (2007). *C primer plus* (5th ed. ed.). Indianapolis, Ind: Sams. ISBN 0-672-32696-5.

[2] Prata, Stephen (2007). *C primer plus* (5th ed. ed.). Indianapolis, Ind: Sams. ISBN 0-672-32696-5.

Chapter 9

Linked list

In computer science, a **linked list** is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.



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

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

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

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

9.1 Advantages

- Linked lists are a dynamic data structure, allocating the needed memory while the program is running.
- Insertion and deletion node operations are easily implemented in a linked list.
- Linear data structures such as stacks and queues are easily executed with a linked list.
- They can reduce access time and may expand in real time without memory overhead.

9.2 Disadvantages

- They have a tendency to waste memory due to pointers requiring extra storage space.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.

- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list.

- Difficulties arise in linked lists when it comes to reverse traversing. Singly linked lists are extremely difficult to navigate backwards, and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

9.3 History

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

LISP, standing for list processor, was created by John McCarthy in 1958 while he was at MIT and in 1960 he published its design in a paper in the Communications of the ACM, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". One of LISP's major data structures is the linked list.

By the early 1960s, the utility of both linked lists and languages which use these structures as their primary data representation was well established. Bert Green of the MIT Lincoln Laboratory published a review article entitled "Computer languages for symbol manipulation" in IRE Transactions on Human Factors in Electronics in March 1961 which summarized the advantages of the linked list approach. A later review article, "A Comparison of list-processing computer languages" by Bobrow and Raphael, appeared in Communications of the ACM in April 1964.

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

The TSS/360 operating system, developed by IBM for the System 360/370 machines, used a double linked list for their file system catalog. The directory structure was similar to Unix, where a directory could contain files and/or other directories and extend to any depth.

9.4 Basic concepts and nomenclature

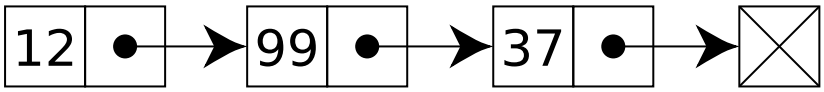
Each record of a linked list is often called an 'element' or 'node'.

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

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

9.4.1 Singly linked list

Singly linked lists contain nodes which have a data field as well as a 'next' field, which points to the next node in line of nodes. Operations that can be performed on singly linked lists include insertion, deletion and traversal.

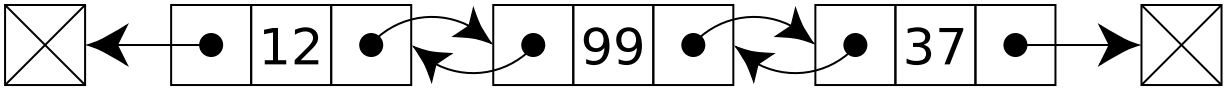


A singly linked list whose nodes contain two fields: an integer value and a link to the next node

9.4.2 Doubly linked list

Main article: Doubly linked list

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



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

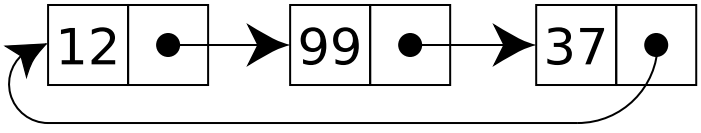
A technique known as XOR-linking allows a doubly linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

9.4.3 Multiply linked list

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.

9.4.4 Circular Linked list

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



A circular linked list

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

9.4.5 Sentinel nodes

Main article: Sentinel node

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.

9.4.6 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.

9.4.7 Hash linking

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

9.4.8 List handles

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

9.4.9 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.

9.5 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.

9.5.1 Linked lists vs. dynamic arrays

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

Linked lists have several advantages over dynamic arrays. Insertion or deletion of an element at a specific point of a list, assuming that we have indexed a pointer to the node (before the one to be removed, or before the insertion point) already, is a constant-time operation (otherwise without this reference it is O(n)), whereas insertion in a dynamic array at random locations will require moving half of the elements on average, and all the elements in the worst case. While one can “delete” an element from an array in constant time by somehow marking its slot as “vacant”, this causes fragmentation that impedes the performance of iteration.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while a dynamic array will eventually fill up its underlying array data structure and will have to reallocate — an expensive operation, one that may not even be possible if memory is fragmented, although the cost of reallocation can be averaged over insertions, and the cost of an insertion due to reallocation would still be amortized O(1). This helps with appending elements at the array’s end, but inserting into (or removing from) middle positions still carries prohibitive costs due to data moving to maintain contiguity. An array from which many elements are removed may also 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 be easily traversed in only

one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays and dynamic arrays is also faster than on linked lists on many machines, because they have optimal locality of reference and thus make good use of data caching.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values, because the storage overhead for the links may exceed by a factor of two or more the size of the data. In contrast, a dynamic array requires only the space for the data itself (and a very small amount of control data).[note 1] It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

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

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

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

A balanced tree has similar memory access patterns and space overhead to a linked list while permitting much more efficient indexing, taking O(log n) time instead of O(n) for a random access. However, insertion and deletion operations are more expensive due to the overhead of tree manipulations to maintain balance. Schemes exist for trees to automatically maintain themselves in a balanced state: AVL trees or red-black trees.

9.5.2 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.

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 those recursive solutions can be adapted 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. The same end-sentinel node may be used 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.

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.

9.5.3 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 fast and easy sequential access to the list in both directions. In a doubly linked list, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly linked list, one must have the *address of the pointer* to that node, which is either the handle for the whole list (in case of the first node) or the link field in the *previous* node. Some algorithms require access in both directions. On the other hand, doubly linked lists do not allow tail-sharing and cannot be used as persistent data structures.

9.5.4 Circularly linked vs. linearly linked

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

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

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

The simplest representation for an empty *circular* list (when such a thing makes sense) is a null pointer, indicating that the list has no nodes. Without 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.

9.5.5 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.[5]

9.6 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.

9.6.1 Linearly linked lists

Singly linked lists

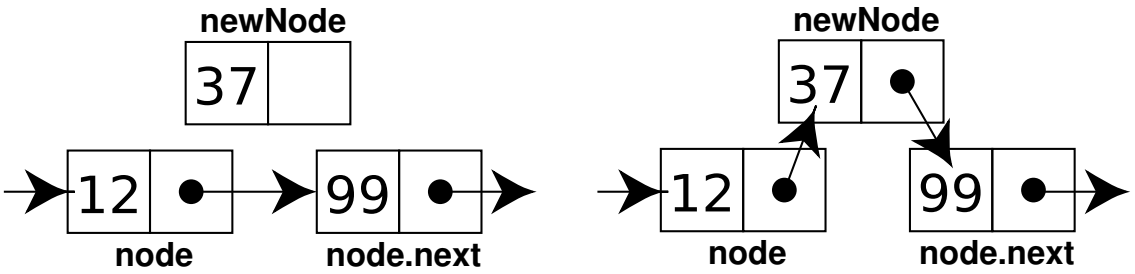
Our node data structure will have two fields. We also keep a variable *firstNode* which always points to the first node in the list, or is *null* for an empty list.

record *Node* { data; *// The data being stored in the node 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 directly; instead, one must keep track of the previous node and insert a node after it.

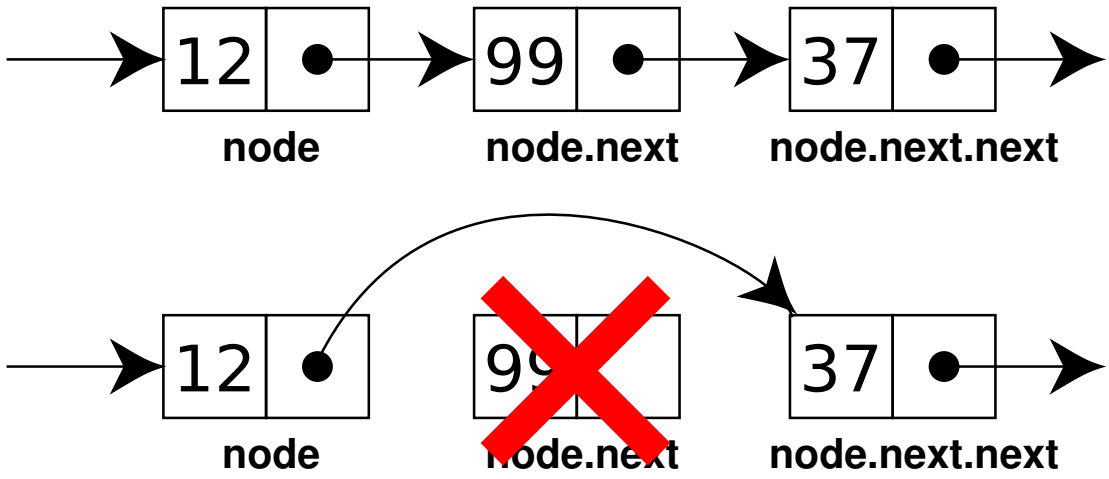


function insertAfter(*Node* node, *Node* newNode) *// insert newNode after node* newNode.next := node.next node.next := newNode

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

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

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.



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

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

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

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

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

9.6.2 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 the test was moved to the beginning of the loop, the procedure would fail whenever the list had only one node.

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

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

9.7 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:

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.

9.8 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.

9.9 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 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.

9.9.1 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.

9.9.2 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).

9.9.3 Random access lists

A random access list is a list with support for fast random access to read or modify any element in the list.[6] One possible implementation is a skew binary random access list using the skew binary number system, which involves a list of trees with special properties; this allows worst-case constant time head/cons operations, and worst-case logarithmic time random access to an element by index.[6] Random access lists can be implemented as persistent data structures.[6]

Random access lists can be viewed as immutable linked lists in that they likewise support the same O(1) head and tail operations.[6]

A simple extension to random access lists is the min-list, which provides an additional operation that yields the minimum element in the entire list in constant time (without mutation complexities).[6]

9.10 Related data structures

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

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

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

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

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

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

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

9.11 Notes

[1] The amount of control data required for a dynamic array is usually of the form $K + B * n$, where K is a per-array constant, B is a per-dimension constant, and n is the number of dimensions. K and B are typically on the order of 10 bytes.

9.12 Footnotes

[1] Gerald Kruse. CS 240 Lecture Notes: Linked Lists Plus: Complexity Trade-offs. Juniata College. Spring 2008.

[2] *Day 1 Keynote - Bjarne Stroustrup: C++11 Style* at *GoingNative 2012* on *channel9.msdn.com* from minute 45 or foil 44

[3] *Number crunching: Why you should never, ever, EVER use linked-list in your code again* at *kjellkod.wordpress.com*

[4] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999), *Resizable Arrays in Optimal Time and Space (Technical Report CS-99-09)*, Department of Computer Science, University of Waterloo

[5] Ford, William and Topp, William *Data Structures with C++ using STL Second Edition* (2002). Prentice-Hall. ISBN 0-13-085850-1, pp. 466-467

[6] C Okasaki, "Purely Functional Random-Access Lists"

9.13 References

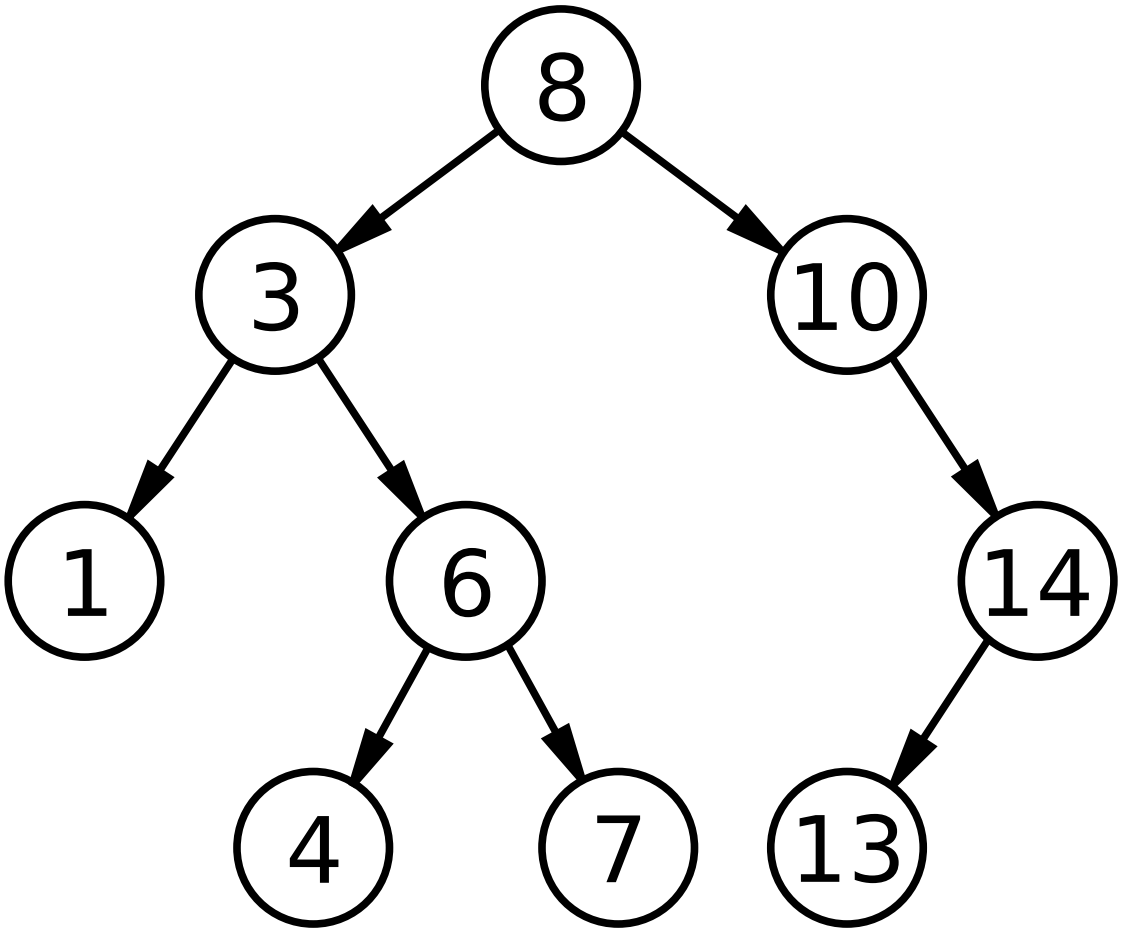
- Juan, Angel (2006). "Ch20 –Data Structures; ID06 - PROGRAMMING with JAVA (slide part of the book "Big Java", by CayS. Horstmann)" (PDF). p. 3
- "Definition of a linked list". National Institute of Standards and Technology. 2004-08-16. Retrieved 2004-12-14.
- Antonakos, James L.; Mansfield, Kenneth 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. doi:10.1109/THFE2.1961.4503292.
- McCarthy, John (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *Communications of the ACM* **3** (4): 184. doi:10.1145/367177.367199.
- 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". 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): 1. doi:10.1016/0066-4138(64)90013-8.
- 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". Retrieved 2009-09-21.

9.14 External links

- Description from the Dictionary of Algorithms and Data Structures
- Introduction to Linked Lists, Stanford University Computer Science Library
- Linked List Problems, Stanford University Computer Science Library
- Open Data Structures - Chapter 3 - Linked Lists
- Patent for the idea of having nodes which are in several linked lists simultaneously (note that this technique was widely used for many decades before the patent was granted)

Chapter 10

Binary search tree



A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

In computer science, **binary search trees** (**BST**), sometimes called **ordered** or **sorted binary trees**, are a class of data structures used to implement lookup tables and dynamic sets. They store data items, known as keys, and allow fast insertion and deletion of such keys, as well as checking whether a key is present in a tree.

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for an key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip over half of the tree, so that each lookup/insertion/deletion takes time proportional to the logarithm of the number of

items stored in the tree. This is much better than the linear time required to find items by key in an unsorted array, but slower than the corresponding operations on hash tables.

10.1 Definition

A binary search tree is a node-based binary tree data structure where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub-tree and smaller than the keys in all nodes in that node's right sub-tree. Each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node. BSTs are also dynamic data structures, and the size of a BST is only limited by the amount of free memory in the operating system. The main advantage of binary search trees is that it remains ordered, which provides quicker search times than many other data structures. The common properties of binary search trees are as follows:[1]

- One node is designated the root of the tree.
- Each internal node contains a key, and can have up to two subtrees.
- The leaves (final nodes) of the tree contain no key.
- Each subtree is itself a binary search tree.
- The left subtree of a node contains only nodes with keys strictly less than the node's key.
- The right subtree of a node contains only nodes with keys strictly greater than the node's 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; they are also easy to code.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays. Some of their disadvantages are as follows:

- The shape of the binary search tree totally depends on the order of insertions, and it can be degenerated.
- When inserting or searching for an element in binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found, i.e., it takes a long time to search an element in a binary search tree.
- The keys in the binary search tree may be long and the run time may increase.
- After a long intermixed sequence of random insertion and deletion, the expected height of the tree approaches square root of the number of keys which grows much faster than $\log n$.

10.2 Operations

Binary search trees support three main operations: insertion of keys, deletion of keys, and lookup (checking whether a key is present). Each requires a *comparator*, a subroutine that computes the total order (linear order) on any two keys. This comparator can be explicitly or implicitly defined, depending on the language in which the binary search tree was implemented. A common comparator is the less-than function, for example, $a < b$, where a and b are keys of two nodes a and b in a binary search tree.

10.2.1 Searching

Searching a binary search tree for a specific key can be a recursive or an iterative process.

We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found before a *null* subtree is reached, then the item must not be present in the tree. This is easily expressed as a recursive algorithm:

function Find-recursive(key, node): *// call initially with node = root* **if** node = Null **or** node.key = key **then return** node **else if** key < node.key **then return** Find-recursive(key, node.left) **else return** Find-recursive(key, node.right)

The same algorithm can be implemented iteratively:

function Find(key, root): current-node := root **while** current-node **is not Null do if** current-node.key = key **then return** current-node **else if** key < current-node.key **then** current-node ← current-node.left **else** current-node ← current-node.right **return Null**

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height* (see tree terminology). On average, binary search trees with n nodes have O(log n) height. However, in the worst case, binary search trees can have O(n) height, when the unbalanced tree resembles a linked list (degenerate tree).

10.2.2 Insertion

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

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

void insert(Node*& root, int data) { if (!root) root = new Node(data); else if (data < root->data) insert(root->left, data); else if (data > root->data) insert(root->right, data); }

The above *destructive* procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

def binary_tree_insert(node, key, value): if node is None: return TreeNode(None, key, value, None) if key == node.key: return TreeNode(node.left, key, value, node.right) if key < node.key: return TreeNode(binary_tree_insert(node.left, key, value), node.key, node.value, node.right) else: return TreeNode(node.left, node.key, node.value, binary_tree_insert(node.right, key, value))

The part that is rebuilt uses O(log n) space in the average case and O(n) in the worst case (see big-O notation).

In either version, this operation requires time proportional to the height of the tree in the worst case, which is O(log n) time in the average case over all trees, but O(n) time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its key.

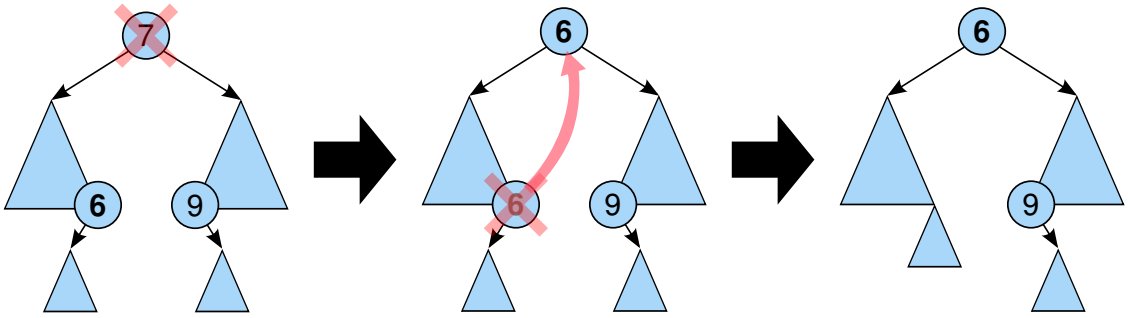
There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

10.2.3 Deletion

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *N*. Do not delete *N*. Instead, choose either its in-order successor node or its in-order predecessor node, *R*. Copy the value of *R* to *N*, then recursively call delete on *R* until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of first 2 cases.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



*Deleting a node with two children from a binary search tree. First the rightmost node in the left subtree, the inorder predecessor **6**, is identified. Its value is copied into the node being deleted. The inorder predecessor can then be easily deleted because it has at most one child. The same method works symmetrically using the inorder successor labelled **9**.*

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 some implementations select one or the other at different times.

Runtime 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.

def find_min(self): # Gets minimum node (leftmost leaf) in a subtree 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): if self.parent: if self == self.parent.left_child: self.parent.left_child = new_value else: self.parent.right_child = new_value if new_value: new_value.parent = self.parent def binary_tree_delete(self, key): if key < self.key: self.left_child.binary_tree_d 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 successor = self.right_child.find_min() self.key = successor.key successor.binary_tree_delete(successor.key elif self.left_child: # if the node has only a *left* child self.replace_node_in_parent(self.left_child) elif self.right_child: # if the node has only a *right* child self.replace_node_in_parent(self.right_child) else: # this node has no children self.replace_node_in_parent(None)

10.2.4 Traversal

Main article: Tree 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. An in-order traversal of a binary search tree will always result in a sorted list of node items (numbers, strings or other comparable items).

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)

With respect to the example defined in the lead section of this article,

- The pre-order traversal is: 8, 3, 1, 6, 4, 7, 10, 14, 13.
- The in-order traversal is: 1, 3, 4, 6, 7, 8, 10, 13, 14.
- The post-order traversal is: 1, 4, 7, 6, 3, 13, 14, 10, 8.

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

10.2.5 Sort

Main article: Tree 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.

The worst-case time of build_binary_tree is $O(n^2)$ —if you feed it a sorted list of values, it chains them into a linked list with no left subtrees. For example, build_binary_tree([1, 2, 3, 4, 5]) yields the tree (1 (2 (3 (4 (5))))).

There are several schemes for overcoming this flaw with simple binary trees; the most common is the self-balancing binary search tree. If this same procedure is done using such a tree, the overall worst-case time is O($n$log n), which is asymptotically optimal for a comparison sort. In practice, the poor cache performance and added overhead in time and space for a tree-based sort (particularly for node allocation) make it inferior to other asymptotically optimal sorts such as heapsort for static list sorting. On the other hand, it is one of the most efficient methods of *incremental sorting*, adding items to a list over time while keeping the list sorted at all times.

10.2.6 Verification

Sometimes we already have a binary tree, and we need to determine whether it is a BST. This is an interesting problem which has a simple recursive solution.

The BST property—every node on the right subtree has to be larger than the current node and every node on the left subtree has to be smaller than (or equal to - should not be the case as only unique values should be in the tree - this also poses the question as to if such nodes should be left or right of this parent) the current node—is the key to figuring out whether a tree is a BST or not. On first thought it might look like we can simply traverse the tree, at every node check whether the node contains a value larger than the value at the left child and smaller than the value on the right child, and if this condition holds for all the nodes in the tree then we have a BST. This is the so-called “Greedy approach,” making a decision based on local properties. But this approach clearly won't work for the following tree:

20 / \ 10 30 / \ 5 40

In the tree above, each node meets the condition that the node contains a value larger than its left child and smaller than its right child hold, and yet it is not a BST: the value 5 is on the right subtree of the node containing 20, a violation of the BST property!

How do we solve this? It turns out that instead of making a decision based solely on the values of a node and its children, we also need information flowing down from the parent as well. In the case of the tree above, if we could remember about the node containing the value 20, we would see that the node with value 5 is violating the BST property contract.

So the condition we need to check at each node is: a) if the node is the left child of its parent, then it must be smaller than (or equal to) the parent and it must pass down the value from its parent to its right subtree to make sure none of the nodes in that subtree is greater than the parent, and similarly b) if the node is the right child of its parent, then it

must be larger than the parent and it must pass down the value from its parent to its left subtree to make sure none of the nodes in that subtree is lesser than the parent.

A simple but elegant recursive solution in C++ can explain this further:

struct TreeNode { int data; TreeNode *left; TreeNode *right; }; bool isBST(TreeNode *node, int minData, int maxData) { if(node == NULL) return true; if(node->data < minData || node->data > maxData) return false; return isBST(node->left, minData, node->data) && isBST(node->right, node->data, maxData); }

The initial call to this function can be something like this:

if(isBST(root, INT_MIN, INT_MAX)) { puts("This is a BST."); } else { puts("This is NOT a BST!"); }

Essentially we keep creating a valid range (starting from [MIN_VALUE, MAX_VALUE]) and keep shrinking it down for each node as we go down recursively.

10.2.7 Priority queue operations

Binary search trees can serve as priority queues: structures that allow insertion of arbitrary key as well as lookup and deletion of the minimum (or maximum) key. Insertion works as previously explained. *Find-min* walks the tree, following left pointers as far as it can without hitting a leaf:

// Precondition: T is not a leaf **function** find-min(T): **while** hasLeft(T): T ← left(T) **return** key(T)

Find-max is analogous: follow right pointers as far as possible. *Delete-min* (*max*) can simply look up the minimum (maximum), then delete it. This way, insertion and deletion both take logarithmic time, just as they do in a binary heap, but unlike a binary heap and most other priority queue implementations, a single tree can support all of *find-min*, *find-max*, *delete-min* and *delete-max* at the same time, making binary search trees suitable as double-ended priority queues.[2]:156

10.3 Types

There are many types of binary search trees. AVL trees and red-black trees are both forms of self-balancing binary search trees. A splay tree is a binary search tree that automatically moves frequently accessed elements nearer to the root. In a treap (*tree heap*), each node also holds a (randomly chosen) priority and the parent node has higher priority than its children. Tango trees are trees optimized for fast searches.

Two other titles describing binary search trees are that of a *complete* and *degenerate* tree.

A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. In complete binary tree, all nodes are far left as possible. It is a tree with n levels, where for each level d <= n - 1, the number of existing nodes at level d is equal to 2^d. This means all possible nodes exist at these levels. An additional requirement for a complete binary tree is that for the nth level, while every node does not have to exist, the nodes that do exist must fill from left to right.

A degenerate tree is a tree where for each parent node, there is only one associated child node. It is unbalanced and, in the worst case, performance degrades to that of a linked list. If your added node function does not handle re-balancing, then you can easily construct a degenerate tree by feeding it with data that is already sorted. What this means is that in a performance measurement, the tree will essentially behave like a linked list data structure.

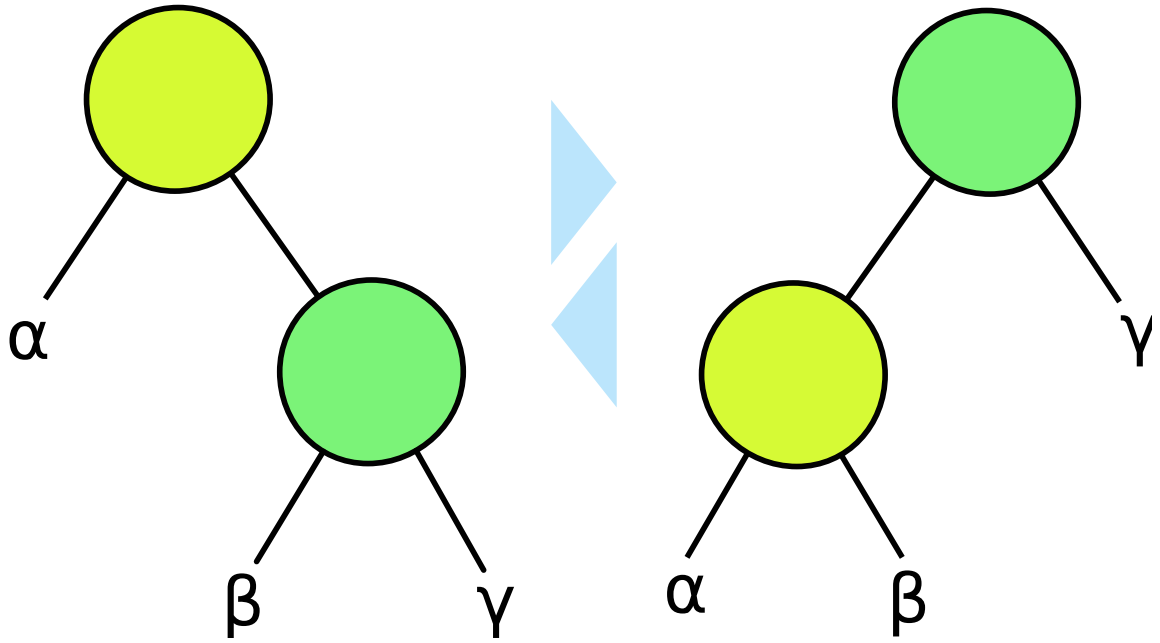
10.3.1 Performance comparisons

D. A. Heger (2004)[3] presented a performance comparison of binary search trees. Treap was found to have the best average performance, while red-black tree was found to have the smallest amount of performance variations.

10.3.2 Optimal binary search trees

Main article: Optimal binary search tree

If we do not plan on modifying a search tree, and we know exactly how often each item will be accessed, we can



Tree rotations are very common internal operations in binary trees to keep perfect, or near-to-perfect, internal balance in the tree.

construct[4] 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 store data elements only 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).

10.4 See also

- Search tree
- Binary search algorithm
- Randomized binary search tree
- Tango trees
- Self-balancing binary search tree
- Geometry of binary search trees
- Red-black tree

- AVL trees
- Day–Stout–Warren algorithm

10.5 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] Mehlhorn, Kurt; Sanders, Peter (2008). *Algorithms and Data Structures: The Basic Toolbox*. Springer.

[3] Heger, Dominique A. (2004), "A Disquisition on The Performance Behavior of Binary Search Tree Data Structures", *European Journal for the Informatics Professional* **5** (5): 67–75

[4] Gonnet, Gaston. "Optimal Binary Search Trees". *Scientific Computation*. ETH Zürich. Retrieved 1 December 2013.

10.6 Further reading

- Paul E. Black, Binary Search Tree at the NIST Dictionary of Algorithms and Data Structures.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "12: Binary search trees, 15.5: Optimal binary search trees". *Introduction to Algorithms* (2nd ed.). MIT Press & McGraw-Hill. pp. 253–272, 356–363. ISBN 0-262-03293-7.
- Jarc, Duane J. (3 December 2005). "Binary Tree Traversals". *Interactive Data Structure Visualizations*. University of Maryland.
- Knuth, Donald (1997). "6.2.2: Binary Tree Searching". *The Art of Computer Programming*. 3: "Sorting and Searching" (3rd ed.). Addison-Wesley. pp. 426–458. ISBN 0-201-89685-0.
- Long, Sean. "Binary Search Tree" (PPT). *Data Structures and Algorithms Visualization-A PowerPoint Slides Based Approach*. SUNY Oneonta.
- Parlante, Nick (2001). "Binary Trees". *CS Education Library*. Stanford University.

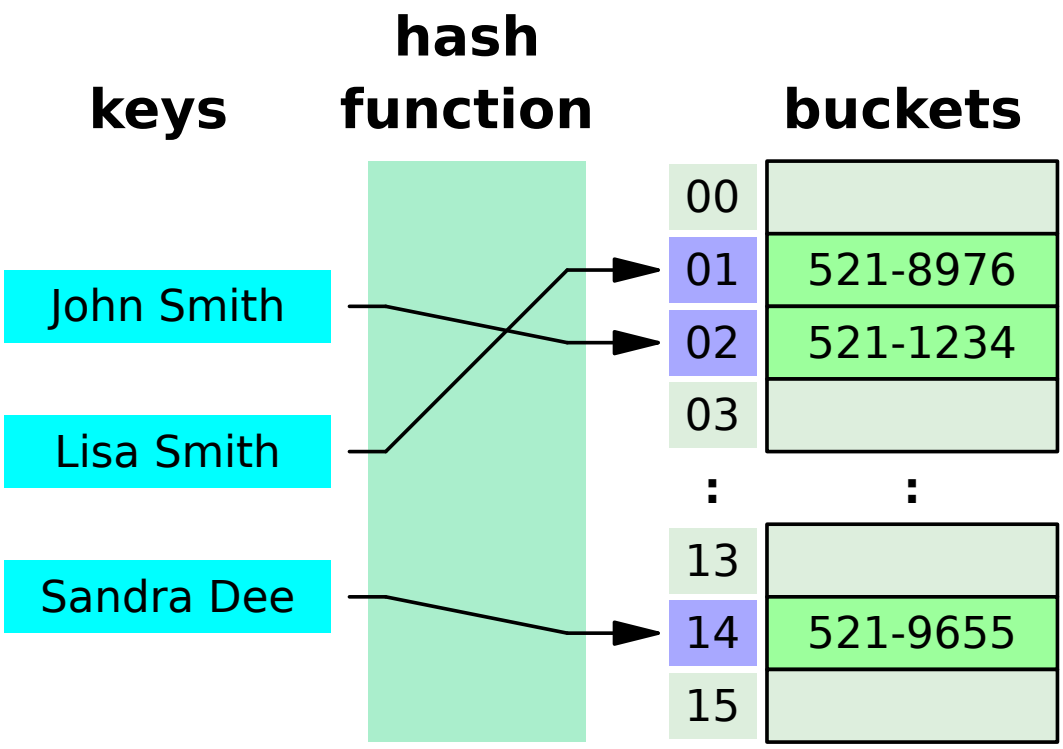
10.7 External links

- Literate implementations of binary search trees in various languages on LiteratePrograms
- Jansens, Dana. "Persistent Binary Search Trees". Computational Geometry Lab, School of Computer Science, Carleton University. C implementation using GLib.
- Binary Tree Visualizer (JavaScript animation of various BT-based data structures)
- Kovac, Kubo. "Binary Search Trees" (Java applet). Korešpondenčný seminár z programovania.
- Madru, Justin (18 August 2009). "Binary Search Tree". *JDServer*. C++ implementation.
- Binary Search Tree Example in Python
- "References to Pointers (C++)". *MSDN*. Microsoft. 2005. Gives an example binary tree implementation.
- Igushev, Eduard. "Binary Search Tree C++ implementation".
- Stromberg, Daniel. "Python Search Tree Empirical Performance Comparison".

Chapter 11

Hash table

Not to be confused with Hash list or Hash tree.
"Rehash" redirects here. For the *South Park* episode, see Rehash (South Park). For the IRC command, see List of Internet Relay Chat commands § REHASH.
In computing, a **hash table** (**hash map**) is a data structure used to implement an associative array, a structure that



A small phone book as a hash table

can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the correct value can be found.

Ideally, the hash function will assign each key to a unique bucket, but this situation is rarely achievable in practice (usually some keys will hash to the same bucket). Instead, most hash table designs assume that *hash collisions*—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the

number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at (amortized[2]) constant average cost per operation.[3][4]

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.

11.1 Hashing

Main article: Hash function

The idea of hashing is to distribute the entries (key/value pairs) across an array of *buckets*. Given a key, the algorithm computes an *index* that suggests where the entry can be found:

index = f(key, array_size)

Often this is done in two steps:

hash = hashfunc(key) index = hash % array_size

In this method, the *hash* is independent of the array size, and it is then *reduced* to an index (a number between 0 and array_size − 1) using the modulo operator (%).

In the case that the array size is a power of two, the remainder operation is reduced to masking, which improves speed, but can increase problems with a poor hash function.

11.1.1 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.

A basic requirement is that the function should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests, e.g., a Pearson's chi-squared test for discrete uniform distributions.[5][6]

The distribution needs to be uniform only for table sizes that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of the table size *s*, then 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.[7]

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[3] is claimed to have particularly poor clustering behavior.[7]

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, the risk of sabotage can also be avoided by cheaper methods (such as applying a secret salt to the data, or using a universal hash function).

11.1.2 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.

11.2 Key statistics

A critical statistic for a hash table is called the *load factor*. This is simply the number of entries divided by the number of buckets, that is, n/k where n is the number of entries and k is the number of buckets.

If the load factor is kept reasonable, the hash table should perform well, provided the hashing is good. If the load factor grows too large, the hash table will become slow, or it may fail to work (depending on the method used). The expected constant time property of a hash table assumes that the load factor is kept below some bound. For a *fixed* number of buckets, the time for a lookup grows with the number of entries and so does not achieve the desired constant time.

Second to that, one can examine the variance of number of entries per bucket. For example, two tables both have 1000 entries and 1000 buckets; one has exactly one entry in each bucket, the other has all entries in the same bucket. Clearly the hashing is not working in the second one.

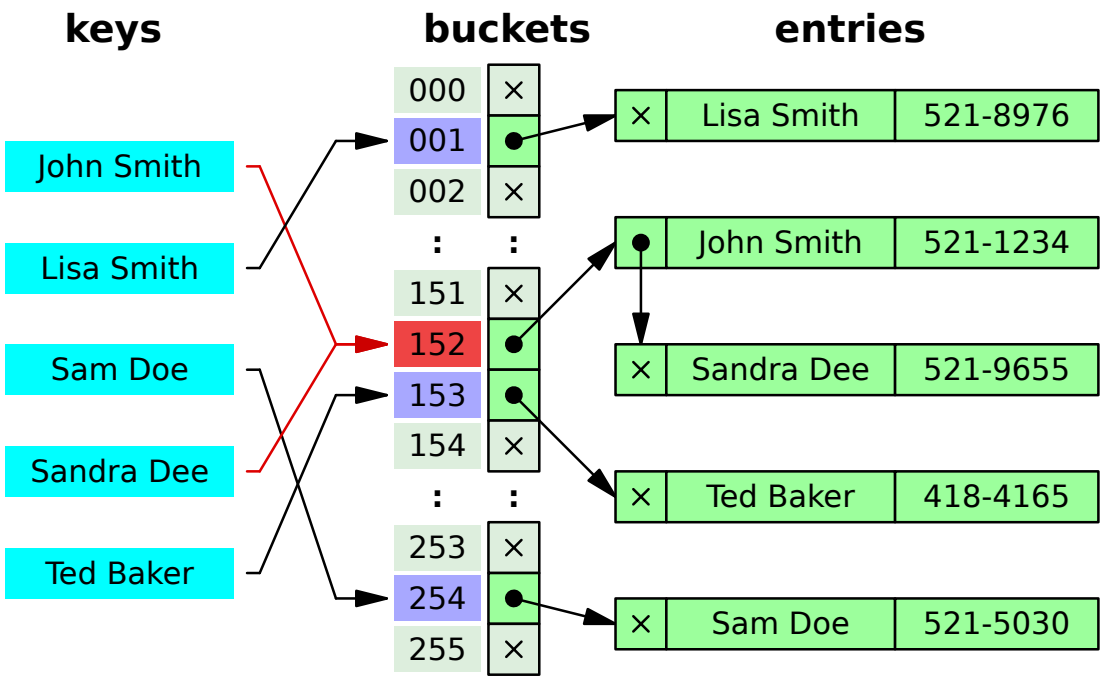
A low load factor is not especially beneficial. As the load factor approaches 0, the proportion of unused areas in the hash table increases, but there is not necessarily any reduction in search cost. This results in wasted memory.

11.3 Collision resolution

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is approximately 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.

11.3.1 Separate chaining



Hash collision resolved by separate chaining.

In the method known as *separate chaining*, each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. Therefore, structures that are efficient in time and space for these cases are preferred. Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable. If these cases happen often, the hashing is not working well, and this needs to be fixed.

Separate chaining with linked lists

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

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

Chained hash tables remain effective even when the number of table entries n is much higher than the number of slots. Their performance degrades more gracefully (linearly) with the load factor. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list.

For separate-chaining, the worst-case scenario is when all entries are 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 a worst-case scenario. 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 head cells

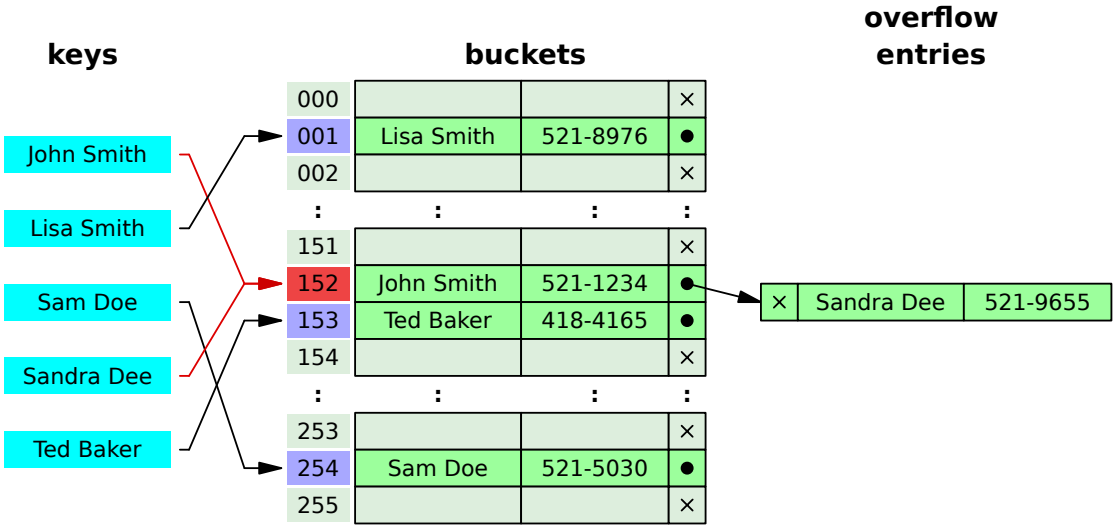
Some chaining implementations store the first record of each chain in the slot array itself.[4] The number of pointer traversals is decreased by one for most cases. The purpose is to increase cache efficiency of hash table access.

The disadvantage is that an empty bucket takes the same space as a bucket with one entry. To save space, such hash tables often have about as many slots as stored entries, meaning that many slots have two or more entries.

Separate chaining with other structures

Instead of a list, one can use any other data structure that supports the required operations. For example, by using a self-balancing tree, the theoretical worst-case time of common hash table operations (insertion, deletion, lookup) can be brought down to 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 must guard against many entries hashed to the same slot (e.g., if one expects extremely non-uniform distributions, or in the case of web sites or other publicly accessible services, which are vulnerable to malicious key distributions in requests).

The variant called array hash table uses a dynamic array to store all the entries that hash to the same slot.[8][9][10] Each newly inserted entry gets appended to the end of the dynamic array that is assigned to the slot. The dynamic array is resized in an *exact-fit* manner, meaning it is grown only by as many bytes as needed. Alternative techniques such as growing the array by block sizes or *pages* were found to improve insertion performance, but at a cost in space. This



Hash collision by separate chaining with head records in the bucket array.

variation makes more efficient use of CPU caching and the translation lookaside buffer (TLB), because slot entries are stored in sequential memory positions. It also dispenses with the next pointers that are required by linked lists, which saves space. Despite frequent array resizing, space overheads incurred by operating system such as memory fragmentation, were found to be small.

An elaboration on this approach is the so-called dynamic perfect hashing,[11] 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 and $n*k$ slots in the average case), this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion.

11.3.2 Open addressing

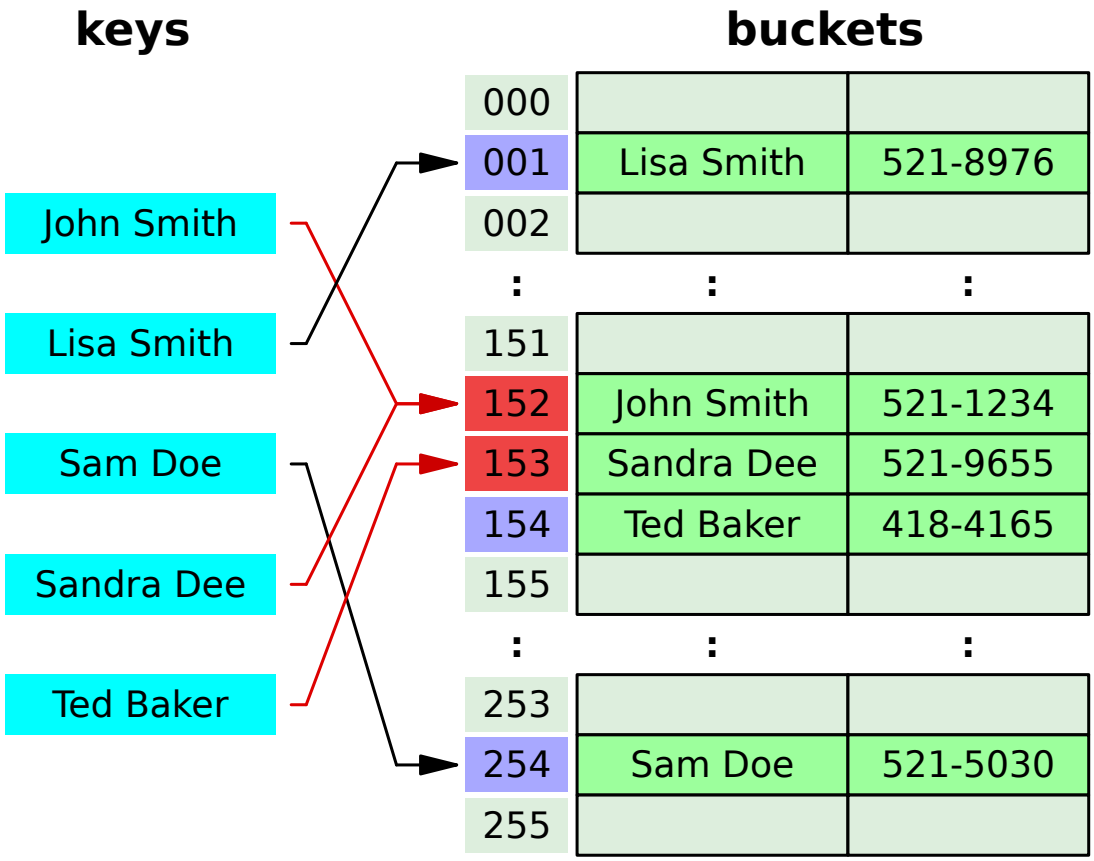
In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, 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.[12] The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called **closed hashing**; it should not be confused with "open hashing" or "closed addressing" that usually mean separate chaining.)

Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by another hash function

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the *same* hash value; whether they are adjacent or nearby is completely irrelevant.



Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith".

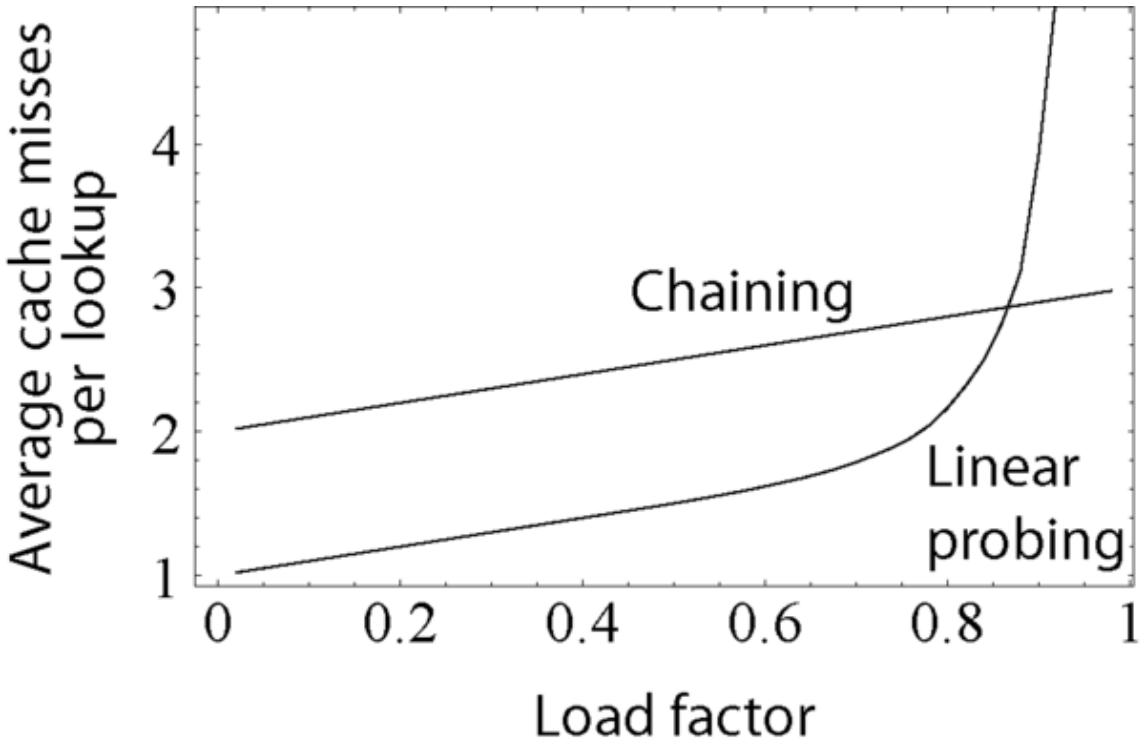
Open addressing only saves memory if the entries are small (less than four times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.

Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (that is, usually the only one). It also has better locality of reference, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups. Hash tables with open addressing are also easier to serialize, because they do not use pointers.

On the other hand, normal open addressing is a poor choice for large elements, because these elements fill entire CPU cache lines (negating the cache advantage), and a large amount of space is wasted on large empty table slots. If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records that can be stored within the table (internal storage) and fit in a cache line. They are particularly suitable for elements of one word or less. If the table is expected to have a high load factor, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

Ultimately, used sensibly, any kind of hash table algorithm is usually fast *enough*; and the percentage of a calculation spent in hash table code is low. Memory usage is rarely considered excessive. Therefore, in most cases the differences between these algorithms are marginal, and other considerations typically come into play.



This graph compares the average number of cache misses required to look up elements in tables with chaining and linear probing. As the table passes the 80%-full mark, linear probing's performance drastically degrades.

Coalesced hashing

A hybrid of chaining and open addressing, coalesced hashing links together chains of nodes within the table itself.[12] 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.

Cuckoo hashing

Another alternative open-addressing solution is cuckoo hashing, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions. It uses two or more hash functions, which means any key/value pair could be in two or more locations. For lookup, the first hash function is used; if the key/value is not found, then the second hash function is used, and so on. If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets, at which point the table is resized. By combining multiple hash functions with multiple cells per bucket, very high space utilisation can be achieved.

Hopscotch hashing

Another alternative open-addressing solution is hopscotch hashing,[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. (This is similar to cuckoo hashing, but with the difference that in this case the empty slot is being moved into the neighborhood, instead of items being moved out with the hope of eventually finding an empty slot.) Each hop brings the open slot closer to the original neighborhood, without invalidating the neighborhood property of any of the buckets along the way. In the end, the open slot has been moved into the neighborhood, and the entry being inserted can be added to it.

11.3.3 Robin Hood hashing

One interesting variation on double-hashing collision resolution is Robin Hood hashing.[14][15] The idea is that a new key may displace a key already inserted, if its probe count is larger than that of the key at the current position. The net effect of this is that it reduces worst case search times in the table. This is similar to ordered hash tables[16] except that the criterion for bumping a key does not depend on a direct relationship between the keys. Since both the worst case and the variation in the number of probes is reduced dramatically, an interesting variation is to probe the table starting at the expected successful probe value and then expand from that position in both directions.[17] 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.[18]

11.3.4 2-choice hashing

2-choice hashing employs 2 different hash functions, $h_1(x)$ and $h_2(x)$, for the hash table. Both hash functions are used to compute two table locations. When an object is inserted in the table, then it is placed in the table location that contains fewer objects (with the default being the $h_1(x)$ table location if there is equality in bucket size). 2-choice hashing employs the principle of the power of two choices.[19]

11.4 Dynamic resizing

The good functioning of a hash table depends on the fact that the table size is proportional to the number of entries. With a fixed size, and the common structures, it is similar to linear search, except with a better constant factor. In some cases, the number of entries may be definitely known in advance, for example keywords in a language. More commonly, this is not known for sure, if only due to later changes in code and data. It is one serious, although common, mistake to not provide *any* way for the table to resize. A general-purpose hash table "class" will almost always have some way to resize, and it is good practice even for simple "custom" tables. An implementation should check the load factor, and do something if it becomes too large (this needs to be done only on inserts, since that is the only thing that would increase it).

To keep the load factor under a certain limit, e.g., under 3/4, many table implementations expand the table when items are inserted. For example, in Java's HashMap class the default load factor threshold for table expansion is 0.75 and in Python's dict, table size is resized when load factor is greater than 2/3.

Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays.

Resizing is accompanied by a full or incremental table **rehash** whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table—followed by a rehash—when items are deleted. From the point of space-time tradeoffs, this operation is similar to the deallocation in dynamic arrays.

11.4.1 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.

For hash tables that shrink and grow frequently, the resizing downward can be skipped entirely. In this case, the table size is proportional to the number of entries that ever were in the hash table, rather than the current number. The disadvantage is that memory usage will be higher, and thus cache behavior may be worse. For best control, a "shrink-to-fit" operation can be provided that does this only on request.

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.

11.4.2 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.

11.4.3 Monotonic keys

If it is known that key values will always increase (or decrease) monotonically, then a variation of consistent hashing can be achieved by keeping a list of the single most recent key value at each hash table resize operation. Upon lookup, keys that fall in the ranges defined by these list entries are directed to the appropriate hash function—and indeed hash table—both of which can be different for each range. Since it is common to grow the overall number of entries by doubling, there will only be O(lg(N)) ranges to check, and binary search time for the redirection would be O(lg(lg(N))). As with consistent hashing, this approach guarantees that any key's hash, once issued, will never change, even when the hash table is later grown.

11.4.4 Other solutions

Linear hashing[20] is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible look-up functions.

Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hashes, where rehashing is prohibitively costly.

11.5 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 k with open addressing has no collisions and holds up to k elements, with a single comparison for successful lookup, and a table of size k with chaining and n keys has the minimum max(0,

n-k) collisions and $O(1 + n/k)$ comparisons for lookup. For the worst choice of hash function, every insertion causes a collision, and hash tables degenerate to linear search, with $\Omega(n)$ amortized comparisons per insertion and up to n 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 n/b^i keys are inserted i or more times, so that the total number of insertions is bounded above by $bn/(b-1)$, which is $O(n)$. By using rehashing to maintain $n < k$, 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 $\Theta(1 + n/k)$ comparisons on average for an unsuccessful lookup, and hashing with open addressing requires $\Theta(1/(1 - n/k))$.[21] Both these bounds are constant, if we maintain $n/k < c$ using table resizing, where c is a fixed constant less than 1.

11.6 Features

11.6.1 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. 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.

11.6.2 Drawbacks

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree. Thus hash tables are not effective when the number of entries is very small. (However, in some cases the high cost of computing the hash function can be mitigated by saving the hash value together with the key.)

For certain string processing applications, such as spell-checking, hash tables may be less efficient than tries, finite automata, or Judy arrays. Also, if each key is represented by a small enough number of bits, then, instead of a hash table, one may use the key directly as the index into an array of values. Note that there are no collisions in this case.

The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. Therefore, there is no efficient way to locate an entry whose key is *nearest* to a given key. Listing all n entries in some specific order generally requires a separate sorting step, whose cost is proportional to $\log(n)$ per entry. In comparison, ordered search trees have lookup and insertion cost proportional to $\log(n)$, but allow finding the nearest key at about the same cost, and *ordered* enumeration of all entries at constant cost per entry.

If the keys are not stored (because the hash function is collision-free), there may be no easy way to enumerate the keys that are present in the table at any given moment.

Although the *average* cost per operation is constant and fairly small, the cost of a single operation may be quite high. In particular, if the hash table uses dynamic resizing, an insertion or deletion operation may occasionally take time proportional to the number of entries. This may be a serious drawback in real-time or interactive applications.

Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays. Compact data structures such as arrays searched with linear search may be faster, if the table is relatively small and keys are compact. The optimal performance point varies from system to system.

Hash tables become quite inefficient when there are many collisions. While extremely uneven hash distributions are extremely unlikely to arise by chance, a malicious adversary with knowledge of the hash function may be able to supply information to a hash that creates worst-case behavior by causing excessive collisions, resulting in very poor performance, e.g., a denial of service attack.[22] In critical applications, universal hashing can be used; a data structure

with better worst-case guarantees may be preferable.[23]

11.7 Uses

11.7.1 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 Perl, Ruby, Python, and PHP.

When storing a new item into a multimap and a hash collision occurs, the multimap unconditionally stores both items.

When storing a new item into a typical associative array and a hash collision occurs, but the actual keys themselves are different, the associative array likewise stores both items. However, if the key of the new item exactly matches the key of an old item, the associative array typically erases the old item and overwrites it with the new item, so every item in the table has a unique key.

11.7.2 Database indexing

Hash tables may also be used as disk-based data structures and database indices (such as in dbm) although B-trees are more popular in these applications.

11.7.3 Caches

Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually erasing the old item that is currently stored in the table and overwriting it with the new item, so every item in the table has a unique hash value.

11.7.4 Sets

Besides recovering the entry that has a given key, many hash table implementations can also tell whether such an entry exists or not.

Those structures can therefore be used to implement a set data structure, which merely records whether a given key belongs to a specified set of keys. In this case, the structure can be simplified by eliminating all parts that have to do with the entry values. Hashing can be used to implement both static and dynamic sets.

11.7.5 Object representation

Several dynamic languages, such as Perl, Python, JavaScript, and Ruby, use hash tables to implement objects. In this representation, the keys are the names of the members and methods of the object, and the values are pointers to the corresponding member or method.

11.7.6 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 *string pool* implemented as a 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.)

11.7.7 String interning

Main article: String interning

11.8 Implementations

11.8.1 In programming languages

Many programming languages provide hash table functionality, either as built-in associative arrays or as standard library modules. In C++11, for example, the unordered_map class provides hash tables for keys and values of arbitrary type.

In PHP 5, the Zend 2 engine uses one of the hash functions from Daniel J. Bernstein to generate the hash values used in managing the mappings of data pointers stored in a hash table. In the PHP source code, it is labelled as DJBX33A (Daniel J. Bernstein, Times 33 with Addition).

Python's built-in hash table implementation, in the form of the dict type, as well as Perl's hash type (%) are used internally to implement namespaces and therefore need to pay more attention to security, i.e., collision attacks. Python sets also use hashes internally, for fast lookup (though they store only keys, not values).[24]

In the .NET Framework, support for hash tables is provided via the non-generic Hashtable and generic Dictionary classes, which store key-value pairs, and the generic HashSet class, which stores only values.

11.8.2 Independent packages

- SparseHash (formerly Google SparseHash) An extremely memory-efficient hash_map implementation, with only 2 bits/entry of overhead. The SparseHash library has several C++ hash map implementations with different performance characteristics, including one that optimizes for memory use and another that optimizes for speed.
- SunriseDD 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 This is an easy-to-use hash table for C structures.

11.9 History

The idea of hashing arose independently in different places. In January 1953, H. P. Luhn wrote an internal IBM memorandum that used hashing with chaining.[25] G. N. Amdahl, E. M. Boehme, N. Rochester, and Arthur Samuel implemented a program using hashing at about the same time. Open addressing with linear probing (relatively prime stepping) is credited to Amdahl, but Ershov (in Russia) had the same idea.[25]

11.10 See also

- Rabin–Karp string search algorithm
- Stable hashing
- Consistent hashing
- Extendible hashing
- Lazy deletion
- Pearson hashing
- PhotoDNA

11.10.1 Related data structures

There are several data structures that use hash functions but cannot be considered special cases of hash tables:

- Bloom filter, memory efficient data-structure designed for constant-time approximate lookups; uses hash function(s) and can be seen as an approximate hash table.
- Distributed hash table (DHT), a resilient dynamic table spread over several nodes of a network.
- Hash array mapped trie, a trie structure, similar to the array mapped trie, but where each key is hashed first.

11.11 References

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). *Introduction to Algorithms* (3rd ed.). Massachusetts Institute of Technology. pp. 253–280. ISBN 978-0-262-03384-8.

[2] Charles E. Leiserson, *Amortized Algorithms, Table Doubling, Potential Method* Lecture 13, course MIT 6.046J/18.410J Introduction to Algorithms—Fall 2005

[3] Knuth, Donald (1998). *'The Art of Computer Programming'*. 3: *Sorting and Searching* (2nd ed.). Addison-Wesley. pp. 513–558. ISBN 0-201-89685-0.

[4] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. 221–252. ISBN 978-0-262-53196-2.

[5] Pearson, Karl (1900). "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling". *Philosophical Magazine, Series 5* **50** (302). pp. 157–175. doi:10.1080/14786440009463897.

[6] Plackett, Robin (1983). "Karl Pearson and the Chi-Squared Test". *International Statistical Review (International Statistical Institute (ISI))* **51** (1). pp. 59–72. doi:10.2307/1402731.

[7] Thomas Wang (1997), Prime Double Hash Table. Retrieved April 27, 2012

[8] Askitis, Nikolas; Zobel, Justin (October 2005). *Cache-conscious Collision Resolution in String Hash Tables*. *Proceedings of the 12th International Conference, String Processing and Information Retrieval (SPIRE 2005)*. 3772/2005. pp. 91–102. doi:10.1007/11575832_11. ISBN 978-3-540-29740-6.

[9] Askitis, Nikolas; Sinha, Ranjan (2010). "Engineering scalable, cache and space efficient tries for strings". *The VLDB Journal* **17** (5): 633–660. doi:10.1007/s00778-010-0183-9. ISSN 1066-8888.

[10] Askitis, Nikolas (2009). *Fast and Compact Hash Tables for Integer Keys*. *Proceedings of the 32nd Australasian Computer Science Conference (ACSC 2009)* **91**. pp. 113–122. ISBN 978-1-920682-72-9.

[11] 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

[12] Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe J. (1990). *Data Structures Using C*. Prentice Hall. pp. 456–461, p. 472. ISBN 0-13-199746-7.

[13] Herlihy, Maurice; Shavit, Nir; Tzafrir, Moran (2008). "Hopscotch Hashing". *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*. Berlin, Heidelberg: Springer-Verlag. pp. 350–364.

[14] Celis, Pedro (1986). *Robin Hood hashing* (Technical report). Computer Science Department, University of Waterloo. CS-86-14.

[15] Goossaert, Emmanuel (2013). "Robin Hood hashing".

[16] Amble, Ole; Knuth, Don (1974). "Ordered hash tables". *Computer Journal* **17** (2): 135. doi:10.1093/comjnl/17.2.135.

[17] 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.

[18] Celis, Pedro (March 1988). *External Robin Hood Hashing* (Technical report). Computer Science Department, Indiana University. TR246.

[19] http://www.eecs.harvard.edu/~{}michaelm/postscripts/handbook2001.pdf

[20] Litwin, Witold (1980). "Linear hashing: A new tool for file and table addressing". *Proc. 6th Conference on Very Large Databases*. pp. 212–223.

[21] Doug Dunham. CS 4521 Lecture Notes. University of Minnesota Duluth. Theorems 11.2, 11.6. Last modified April 21, 2009.

[22] Alexander Klink and Julian Wälde's *Efficient Denial of Service Attacks on Web Application Platforms*, December 28, 2011, 28th Chaos Communication Congress. Berlin, Germany.

[23] Crosby and Wallach's *Denial of Service via Algorithmic Complexity Attacks*.

[24] http://stackoverflow.com/questions/513882/python-list-vs-dict-for-look-up-table

[25] Mehta, Dinesh P.; Sahni, Sartaj. *Handbook of Datastructures and Applications*. pp. 9–15. ISBN 1-58488-435-5.

11.12 Further reading

- Tamassia, Roberto; Goodrich, Michael T. (2006). "Chapter Nine: Maps and Dictionaries". *Data structures and algorithms in Java : [updated for Java 5.0]* (4th ed.). Hoboken, NJ: Wiley. pp. 369–418. ISBN 0-471-73884-0.

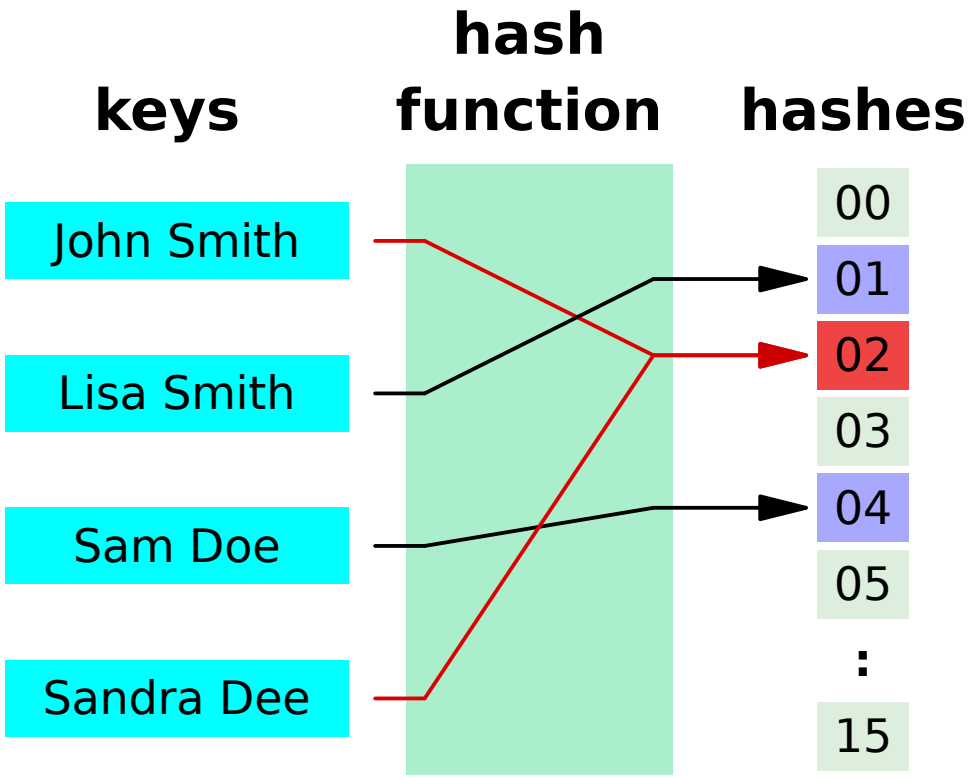
- McKenzie, B. J.; Harries, R.; Bell, T. (Feb 1990). "Selecting a hashing algorithm". *Software Practice & Experience* **20** (2): 209–224. doi:10.1002/spe.4380200207.

11.13 External links

- A Hash Function for Hash Table Lookup by Bob Jenkins.
- Hash Tables by SparkNotes—explanation using C
- Hash functions by Paul Hsieh
- Design of Compact and Efficient Hash Tables for Java
- Libhashish hash library
- NIST entry on hash tables
- Open addressing hash table removal algorithm from ICI programming language, *ici_set_unassign* in set.c (and other occurrences, with permission).
- A basic explanation of how the hash table works by Reliable Software
- Lecture on Hash Tables
- Hash-tables in C—two simple and clear examples of hash tables implementation in C with linear probing and chaining
- Open Data Structures – Chapter 5 – Hash Tables
- MIT's Introduction to Algorithms: Hashing 1 MIT OCW lecture Video
- MIT's Introduction to Algorithms: Hashing 2 MIT OCW lecture Video
- How to sort a HashMap (Java) and keep the duplicate entries
- How python dictionary works

Chapter 12

Hash function



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

A **hash function** is any function that can be used to map digital data of arbitrary size to digital data of fixed size. The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, or simply **hashes**. One practical use is a data structure called a hash table, widely used in computer software for rapid data lookup. Hash functions accelerate table or database lookup by detecting duplicated records in a large file. An example is finding similar stretches in DNA sequences. They are also useful in cryptography. A cryptographic hash function allows one to easily verify that some input data matches a stored hash value, but makes it hard to construct any data that would hash to the same value or find any two unique data pieces that hash to the same value. This principle is used by the PGP algorithm for data validation and by many password checking systems.

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, randomization functions, error-correcting codes, and ciphers. Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimized differently. The Hash Keeper database maintained by the American

National Drug Intelligence Center, for instance, is more aptly described as a catalog of file fingerprints than of hash values.

12.1 Uses

12.1.1 Hash tables

Hash functions are primarily used in hash tables, to quickly locate a data record (e.g., a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets.

Typically, the domain of a hash function (the set of possible keys) is larger than its range (the number of different table indexes), and so it will 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.

12.1.2 Caches

Hash functions are also used to build caches for large data sets stored in slow media. A cache is generally simpler than a hashed search table, since any collision can be resolved by discarding or writing back the older of the two colliding items. This is also used in file comparison.

12.1.3 Bloom filters

Main article: Bloom filter

Hash functions are an essential ingredient of the Bloom filter, a space-efficient probabilistic data structure that is used to test whether an element is a member of a set.

12.1.4 Finding duplicate records

Main article: Hash table

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).

12.1.5 Protecting data

Main article: Security of cryptographic hash functions

A hash value can be used to uniquely identify secret information. This requires that the hash function is collision resistant, which means that it is very hard to find data that generate the same hash value. These functions are categorized into cryptographic hash functions and provably secure hash functions. Functions in the second category are the most secure but also too slow for most practical purposes. Collision resistance is accomplished in part by generating very large hash values. For example SHA-1, one of the most widely used cryptographic hash functions, generates 160 bit values.

12.1.6 Finding similar records

Main article: Locality sensitive hashing

Hash functions can also be used to locate table records whose key is similar, but not identical, to a given key; or pairs of records in a large file which have similar keys. For that purpose, one needs a hash function that maps similar keys to hash values that differ by at most m, where m is a small integer (say, 1 or 2). If one builds a table T of all record numbers, using such a hash function, then similar records will end up in the same bucket, or in nearby buckets. Then one need only check the records in each bucket $T[i]$ against those in buckets $T[i+k]$ where k ranges between $-m$ and m.

This class includes the so-called acoustic fingerprint algorithms, that are used to locate similar-sounding entries in large collection of audio files. For this application, the hash function must be as insensitive as possible to data capture or transmission errors, and to trivial changes such as timing and volume changes, compression, etc.[1]

12.1.7 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.

12.1.8 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.

12.1.9 Standard uses of hashing in cryptography

Main article: Cryptographic hash function

Some standard applications that employ hash functions include authentication, message integrity (using an HMAC (Hashed MAC)), message fingerprinting, data corruption detection, and digital signature efficiency.

12.2 Properties

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. The exact requirements are dependent on the application, for example a hash function well suited to indexing data will probably be a poor choice for a cryptographic hash function.

12.2.1 Determinism

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the data to be hashed, in the mathematical sense of the term. This requirement excludes hash functions that depend on external variable parameters, such as pseudo-random number generators or the time of day. It also excludes functions that depend on the memory address of the object being

hashed, because that address may change during execution (as may happen on systems that use certain methods of garbage collection), although sometimes rehashing of the item is possible.

12.2.2 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. If some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

Note that this criterion only requires the value to be *uniformly distributed*, not *random* in any sense. A good randomizing function is (barring computational efficiency concerns) generally a good choice as a hash function, but the converse need not be true.

Hash tables often contain only a small subset of the valid inputs. For instance, a club membership list may contain only a hundred or so member names, out of the very large set of all possible names. In these cases, the uniformity criterion should hold for almost all typical subsets of entries that may be found in the table, not just for the global set of all possible entries.

In other words, if a typical set of m records is hashed to n table slots, the probability of a bucket receiving many more than m/n records should be vanishingly small. In particular, if m is less than n, very few buckets should have more than one or two records. (In an ideal "perfect hash function", no bucket should have more than one record; but a small number of collisions is virtually inevitable, even if n is much larger than m – see the birthday paradox).

When testing a hash function, the uniformity of the distribution of hash values can be evaluated by the chi-squared test.

12.2.3 Defined range

It is often desirable that the output of a hash function have fixed size (but see below). If, for example, the output is constrained to 32-bit integer values, the hash values can be used to index into an array. Such hashing is commonly used to accelerate data searches.[2] On the other hand, cryptographic hash functions produce much larger hash values, in order to ensure the computational complexity of brute-force inversion.[3] For example SHA-1, one of the most widely used cryptographic hash functions, produces a 160-bit value.

Producing fixed-length output from variable length input can be accomplished by breaking the input data into chunks of specific size. Hash functions used for data searches use some arithmetic expression which iteratively processes chunks of the input (such as the characters in a string) to produce the hash value.[2] In cryptographic hash functions, these chunks are processed by a one-way compression function, with the last chunk being padded if necessary. In this case, their size, which is called *block size*, is much bigger than the size of the hash value.[3] For example, in SHA-1, the hash value is 160 bits and the block size 512 bits.

Variable range

In many applications, the range of hash values may be different for each run of the program, or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data z, and the number n of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to $2^{32} - 1$), divide the result by n, and use the division's remainder. If n is itself a power of 2, this can be done by bit masking and bit shifting. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and $n - 1$, for any value of n that may occur in the application. Depending on the function, the remainder may be uniform only for certain values of n, e.g. odd or prime numbers.

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

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 – 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.

12.2.4 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.

12.2.5 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."[4]

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 used in Nearest neighbor search.

12.2.6 Non-invertible

In cryptographic applications, hash functions are typically expected to be non-invertible, meaning that it is not possible to reconstruct the input datum x from its hash value h(x) alone without spending great amounts of computing time (see also One-way function).

12.3 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.

12.3.1 Trivial hash function

If the datum to be hashed is small enough, one can use the datum itself (reinterpreted as an integer) as the hashed value. The cost of computing this "trivial" (identity) hash function is effectively zero. This hash function is perfect, as it maps each input to a distinct hash value.

The meaning of "small enough" depends on the size of the type that is used as the hashed value. For example, in Java, the hash code is a 32-bit integer. Thus the 32-bit integer Integer and 32-bit floating-point Float objects can simply use the value directly; whereas the 64-bit integer Long and 64-bit floating-point Double cannot use this method.

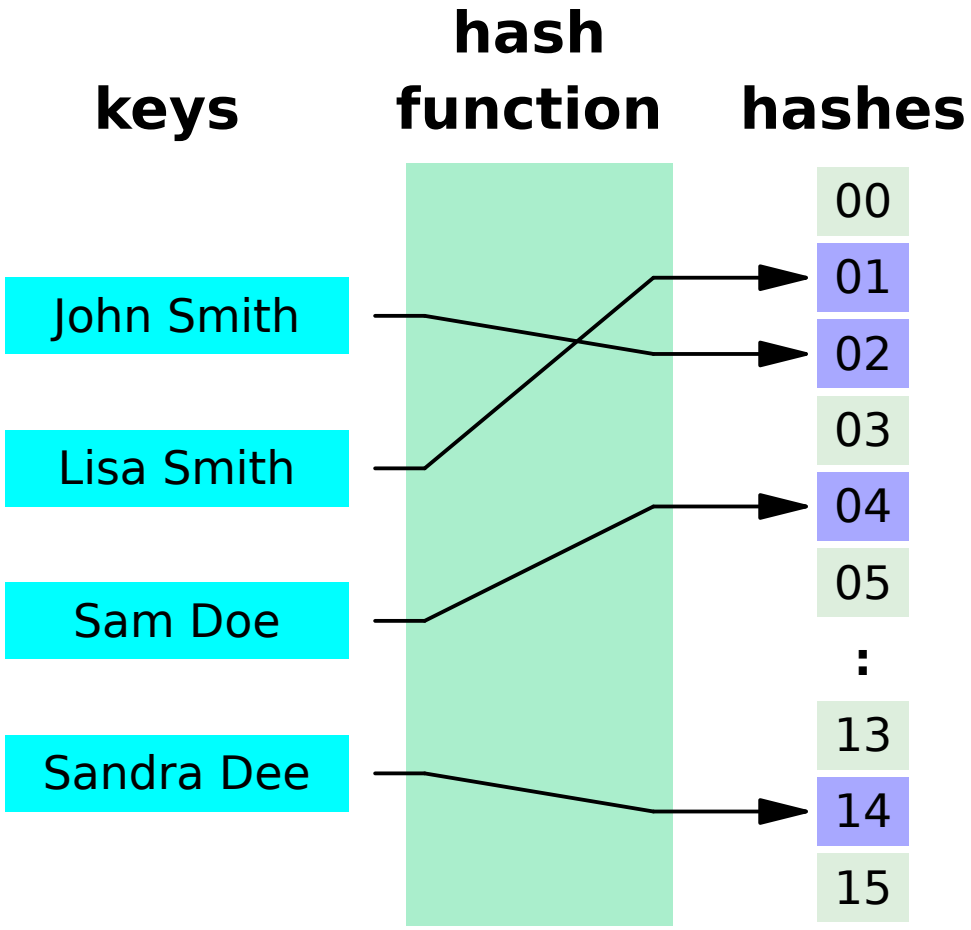
Other types of data can also use this perfect hashing scheme. For example, when mapping character strings between upper and lower case, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for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

(as in ASCII or ISO Latin 1), the table has only 2^8 = 256 entries; in the case of Unicode characters, the table would have 17×2^{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.

12.3.2 Perfect hashing

Main article: Perfect hash function
A hash function that is injective—that is, maps each valid input to a different hash value—is said to be **perfect**. With

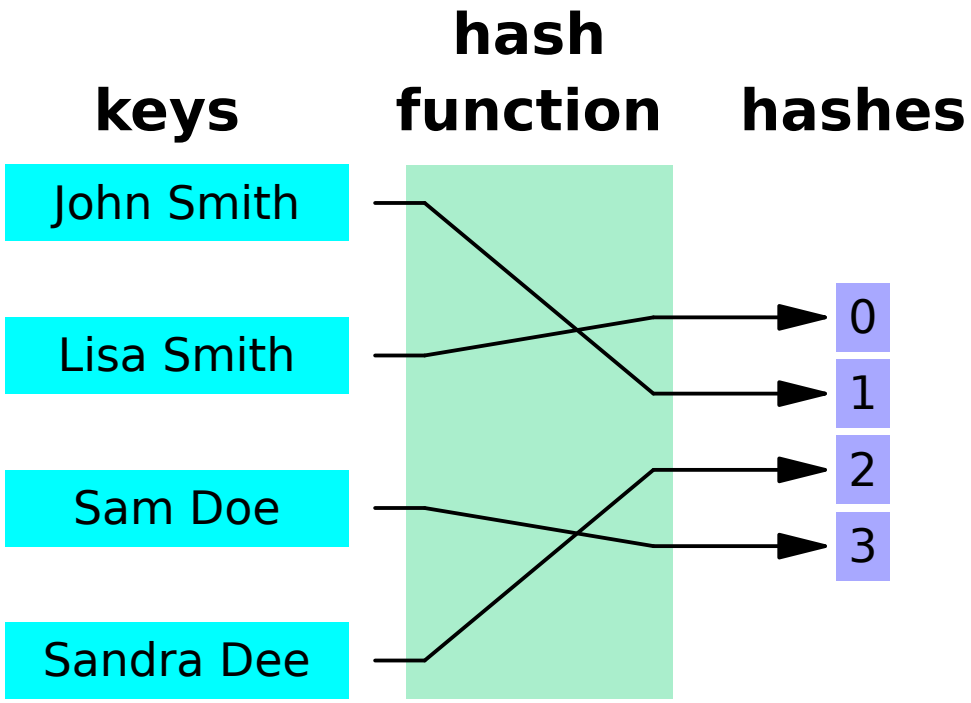


A perfect hash function for the four names shown

such a function one can directly locate the desired entry in a hash table, without any additional searching.

12.3.3 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.



A minimal perfect hash function for the four names shown

12.3.4 Hashing uniformly distributed data

If the inputs are bounded-length strings and each input may independently occur with uniform probability (such as telephone numbers, car license plates, invoice numbers, etc.), then a hash function needs to 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.

$h = z \bmod n$ was used in many of the original random number generators, but was found to have a number of issues. One of which is that as n approaches N, this function becomes less and less uniform.

12.3.5 Hashing data with other distributions

These simple formulas will not do if the input values are not equally likely, or are not independent. For instance, most patrons of a supermarket will live in the same geographic area, so their telephone numbers are likely to begin with the same 3 to 4 digits. In that case, if m is 10000 or so, the division formula $(z \times m) \div M$, which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula $z \bmod m$, which is quite sensitive to the trailing digits, may still yield a fairly even distribution.

12.3.6 Hashing variable-length data

When the data values are long (or variable-length) character strings—such as personal names, web page addresses, or mail messages—their distribution is usually very uneven, with complicated dependencies. For example, text in any natural language has highly non-uniform distributions of characters, and character pairs, very characteristic of the language. For such data, it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way.

In cryptographic hash functions, a Merkle–Damgård construction is usually used. In general, the scheme for hashing

such data is to break the input into a sequence of small units (bits, bytes, words, etc.) and combine all the units *b*[1], *b*[2], …, *b*[*m*] sequentially, as follows

S ← S0; // *Initialize the state.* **for** k **in** 1, 2, ..., m **do** // *Scan the input data units:* S ← F(S, b[k]); // *Combine data unit k into the state.* **return** G(S, n) // *Extract the hash value from the state.*

This schema is also used in many text checksum and fingerprint algorithms. The state variable *S* may be a 32- or 64-bit unsigned integer; in that case, *S0* can be 0, and *G*(*S*,*n*) can be just *S* **mod** *n*. The best choice of *F* is a complex issue and depends on the nature of the data. If the units *b*[*k*] are single bits, then *F*(*S*,*b*) could be, for instance

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

Here *highbit*(*S*) denotes the most significant bit of *S*; the '*' operator denotes unsigned integer multiplication with lost overflow; '^' is the bitwise exclusive or operation applied to words; and *P* is a suitable fixed word.[5]

12.3.7 Special-purpose hash functions

In many cases, one can design a special-purpose (heuristic) hash function that yields many fewer collisions than a good general-purpose hash function. For example, suppose that the input data are file names such as FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, etc., with mostly sequential numbers. For such data, a function that extracts the numeric part *k* of the file name and returns *k* **mod** *n* would be nearly optimal. Needless to say, a function that is exceptionally good for a specific kind of data may have dismal performance on data with different distribution.

12.3.8 Rolling hash

Main article: rolling hash

In some applications, such as substring search, one must compute a hash function *h* for every *k*-character substring of a given *n*-character string *t*; where *k* is a fixed integer, and *n* is *k*. The straightforward solution, which is to extract every such substring *s* of *t* and compute *h*(*s*) separately, requires a number of operations proportional to *k*·*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*.

12.3.9 Universal hashing

A universal hashing scheme is a randomized algorithm that selects a hashing function *h* among a family of such functions, in such a way that the probability of a collision of any two distinct keys is 1/*n*, where *n* is the number of distinct hash values desired—independently of the two keys. Universal hashing ensures (in a probabilistic sense) that the hash function application will behave as well as if it were using a random function, for any distribution of the input data. It will however have more collisions than perfect hashing, and may require more operations than a special-purpose hash function. See also Unique Permutation Hashing.[6]

12.3.10 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.[7]

12.3.11 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.[8] However, this method can provide uniformly distributed hashes even when the keys are chosen by a malicious agent. This feature may help to protect services against denial of service attacks.

12.3.12 Hashing By Nonlinear Table Lookup

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

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

12.3.13 Efficient Hashing Of Strings

- See also Universal hashing of strings

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

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

Another approach[9] is to convert strings to a 32 or 64 bit numeric value and then apply a hash function. One method that avoids the problem of strings having great similarity ("Aaaaaaaaaa" and "Aaaaaaaaab") is to use a Cyclic redundancy check (CRC) of the string to compute a 32- or 64-bit value. While it is possible that two different strings will have the same CRC, the likelihood is very small and only requires that one check the actual string found to determine whether one has an exact match. CRCs will be different for strings such as "Aaaaaaaaaa" and "Aaaaaaaaab". Although, CRC codes can be used as hash values[10] they are not cryptographically secure since they are not collision resistant.[11]

12.4 Locality-sensitive hashing

Locality-sensitive hashing (LSH) is a method of performing probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input items so that similar items are mapped to the same buckets with high probability (the number of buckets being much smaller than the universe of possible input items). This is different from the conventional hash functions, such as those used in cryptography, as in this case the goal is to maximize the probability of "collision" of similar items rather than to avoid collisions.[12]

One example of LSH is MinHash algorithm used for finding similar documents (such as web-pages):

Let h be a hash function that maps the members of A and B to distinct integers, and for any set S define $h_{\min}(S)$ to be the member x of S with the minimum value of $h(x)$. Then $h_{\min}(A) = h_{\min}(B)$ exactly when the minimum hash value of the union $A \cup B$ lies in the intersection $A \cap B$. Therefore,

$$\Pr[h_{\min}(A) = h_{\min}(B)] = J(A,B).$$ where J is Jaccard index.

In other words, if r is a random variable that is one when $h_{\min}(A) = h_{\min}(B)$ and zero otherwise, then r is an unbiased estimator of $J(A,B)$, although it has too high a variance to be useful on its own. The idea of the MinHash scheme is to reduce the variance by averaging together several variables constructed in the same way.

12.5 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.[13]

12.6 List of hash functions

Main article: List of hash functions

- NIST hash function competition
- Bernstein hash[14]
- Fowler-Noll-Vo hash function (32, 64, 128, 256, 512, or 1024 bits)
- Jenkins hash function (32 bits)
- Pearson hashing (64 bits)
- Zobrist hashing
- Almost linear hash function

12.7 See also

Computer science portal

- Bloom filter
- Coalesced hashing
- Cuckoo hashing
- Hopscotch hashing
- Cryptographic hash function
- Distributed hash table
- Geometric hashing
- Hash Code cracker
- Hash table
- HMAC
- Identicon
- Linear hash

- List of hash functions
- Locality sensitive hashing
- MD5
- Perfect hash function
- PhotoDNA
- Rabin–Karp string search algorithm
- Rolling hash
- Transposition table
- Universal hashing
- MinHash
- Low-discrepancy sequence

12.8 References

[1] "Robust Audio Hashing for Content Identification by Jaap Haitsma, Ton Kalker and Job Oostveen"

[2] Sedgewick, Robert (2002). "14. Hashing". *Algorithms in Java* (3 ed.). Addison Wesley. ISBN 978-0201361209.

[3] Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A (1996). *Handbook of Applied Cryptography*. CRC Press. ISBN 0849385237.

[4] "Fundamental Data Structures - Josiang p.132". Retrieved May 19, 2014.

[5] Broder, A. Z. (1993). "Some applications of Rabin's fingerprinting method". *Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag. pp. 143–152.

[6] Shlomi Dolev, Limor Lahiani, Yinnon Haviv, "Unique permutation hashing," Theoretical Computer Science Volume 475, 4 March 2013, Pages 59–65

[7] Bret Mulvey, *Evaluation of CRC32 for Hash Tables*, in *Hash Functions*. Accessed April 10, 2009.

[8] Bret Mulvey, *Evaluation of SHA-1 for Hash Tables*, in *Hash Functions*. Accessed April 10, 2009.

[9] http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7520 Performance in Practice of String Hashing Functions

[10] Peter Kankowski. "Hash functions: An empirical comparison".

[11] Cam-Winget, Nancy; Housley, Russ; Wagner, David; Walker, Jesse (May 2003). "Security Flaws in 802.11 Data Link Protocols". *Communications of the ACM* **46** (5): 35–39. doi:10.1145/769800.769823.

[12] A. Rajaraman and J. Ullman (2010). "Mining of Massive Datasets, Ch. 3.".

[13] Knuth, Donald (1973). *The Art of Computer Programming, volume 3, Sorting and Searching*. pp. 506–542.

[14] "Hash Functions". *cse.yorku.ca*. September 22, 2003. Retrieved November 1, 2012. the djb2 algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c.

12.9 External links

- Hash Functions and Block Ciphers by Bob Jenkins
- The Goulburn Hashing Function (PDF) by Mayur Patel
- Hash Function Construction for Textual and Geometrical Data Retrieval Latest Trends on Computers, Vol.2, pp. 483–489, CSCC conference, Corfu, 2010

Chapter 13

Bubble sort

Bubble sort, sometimes referred to as **sinking sort**, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.[1] It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position.

13.1 Analysis

An example of bubble sort. Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

13.1.1 Performance

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. Even other

$O(n^2)$ sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.

The only significant advantage that bubble sort has over most other implementations, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted is efficiently built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$. By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort have this mechanism too, but it also performs better on a list that is substantially sorted (having a small number of inversions).

Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection.

13.1.2 Rabbits and turtles

The positions of the elements in bubble sort will play a large part in determining its performance. Large elements at the beginning of the list do not pose a problem, as they are quickly swapped. Small elements towards the end, however, move to the beginning extremely slowly. This has led to these types of elements being named rabbits and turtles, respectively.

Various efforts have been made to eliminate turtles to improve upon the speed of bubble sort. Cocktail sort is a bi-directional bubble sort that goes from beginning to end, and then reverses itself, going end to beginning. It can move turtles fairly well, but it retains $O(n^2)$ worst-case complexity. Comb sort compares elements separated by large gaps, and can move turtles extremely quickly before proceeding to smaller and smaller gaps to smooth out the list. Its average speed is comparable to faster algorithms like quicksort.

13.1.3 Step-by-step example

Let us take the array of numbers “5 1 4 2 8”, and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:
(**5** **1** 4 2 8) → (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) → (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) → (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) → (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1** **4** 2 5 8) → (**1** **4** 2 5 8)
(1 **4** **2** 5 8) → (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)
(1 2 4 **5** **8**) → (1 2 4 **5** **8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1** **2** 4 5 8) → (**1** **2** 4 5 8)
(1 **2** **4** 5 8) → (1 **2** **4** 5 8)
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)
(1 2 4 **5** **8**) → (1 2 4 **5** **8**)

13.2 Implementation

13.2.1 Pseudocode implementation

The algorithm can be expressed as (0-based array):

procedure bubbleSort(A : list of sortable items) n = length(A) repeat swapped = false for i = 1 to n-1 inclusive do /* if this pair is out of order */ if A[i-1] > A[i] then /* swap them and remember something changed */ swap(A[i-1],

A[i]) swapped = true end if end for until not swapped end procedure

13.2.2 Optimizing bubble sort

The bubble sort algorithm can be easily optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last n-1 items when running for the n-th time:

procedure bubbleSort(A : list of sortable items) n = length(A) repeat swapped = false for i = 1 to n-1 inclusive do if A[i-1] > A[i] then swap(A[i-1], A[i]) swapped = true end if end for n = n - 1 until not swapped end procedure

More generally, it can happen that more than one element is placed in their final position on a single pass. In particular, after every pass, all elements after the last swap are sorted, and do not need to be checked again. This allows us to skip over a lot of the elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the "swapped" variable:

To accomplish this in pseudocode we write the following:

procedure bubbleSort(A : list of sortable items) n = length(A) repeat newn = 0 for i = 1 to n-1 inclusive do if A[i-1] > A[i] then swap(A[i-1], A[i]) newn = i end if end for n = newn until n = 0 end procedure

Alternate modifications, such as the cocktail shaker sort attempt to improve on the bubble sort performance while keeping the same idea of repeatedly comparing and swapping adjacent items.

13.3 In practice

Although bubble sort is one of the simplest sorting algorithms to understand and implement, its $O(n^2)$ complexity means that its efficiency decreases dramatically on lists of more than a small number of elements. Even among simple $O(n^2)$ sorting algorithms, algorithms like insertion sort are usually considerably more efficient.

Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students. However, some researchers such as Owen Astrachan have gone to great lengths to disparage bubble sort and its continued popularity in computer science education, recommending that it no longer even be taught.[2]

The Jargon File, which famously calls bogosort "the archetypical [sic] perversely awful algorithm", also calls bubble sort "the generic **bad** algorithm".[3] Donald Knuth, in his famous book *The Art of Computer Programming*, concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems", some of which he then discusses.[1]

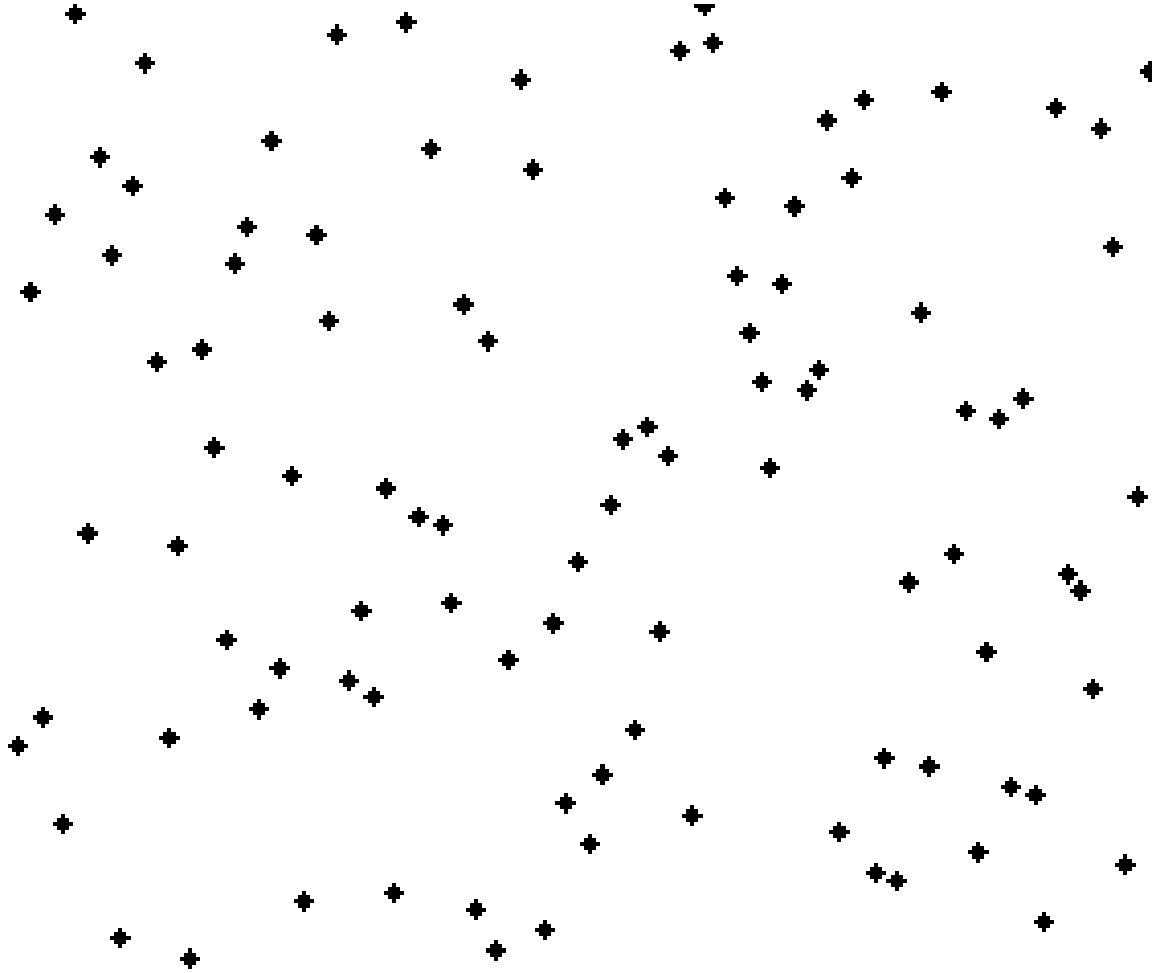
Bubble sort is asymptotically equivalent in running time to insertion sort in the worst case, but the two algorithms differ greatly in the number of swaps necessary. Experimental results such as those of Astrachan have also shown that insertion sort performs considerably better even on random lists. For these reasons many modern algorithm textbooks avoid using the bubble sort algorithm in favor of insertion sort.

Bubble sort also interacts poorly with modern CPU hardware. It requires at least twice as many writes as insertion sort, twice as many cache misses, and asymptotically more branch mispredictions. Experiments by Astrachan sorting strings in Java show bubble sort to be roughly one-fifth as fast an insertion sort and 70% as fast as a selection sort.[2]

In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines. Bubble sort is a stable sort algorithm, like insertion sort.

13.4 Variations

- Odd-even sort is a parallel version of bubble sort, for message passing systems.



A bubble sort, a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order. The list was plotted in a Cartesian coordinate system, with each point (x,y) indicating that the value y is stored at index x. Then the list would be sorted by Bubble sort according to every pixel's value. Note that the largest end gets sorted first, with smaller elements taking longer to move to their correct positions.

- Cocktail sort is another parallel version of the bubble sort
- In some cases, the sort works from right to left (the opposite direction), which is more appropriate for partially sorted lists, or lists with unsorted items added to the end.

13.5 Debate over name

Bubble sort has been occasionally referred to as a "sinking sort".[4]

For example, in Donald Knuth's *The Art of Computer Programming*, Volume 3: *Sorting and Searching* he states in section 5.2.1 'Sorting by Insertion', that [the value] "settles to its proper level" this method of sorting has often been called the *sifting* or *sinking* technique. Furthermore the *larger* values might be regarded as *heavier* and therefore be seen to progressively *sink* to the *bottom* of the list.

13.6 Notes

[1] Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Pages 106–110 of section 5.2.2: Sorting by Exchanging. "[A]lthough the techniques used in the calculations [to analyze the bubble sort] are instructive, the results are disappointing since they tell us that the bubble

sort isn't really very good at all. Compared to straight insertion [...], bubble sorting requires a more complicated program and takes about twice as long!" (Quote from the first edition, 1973.)

[2] Owen Astrachan. Bubble Sort: An Archaeological Algorithmic Analysis. SIGCSE 2003 Hannan Akhtar . (pdf)

[3] http://www.jargon.net/jargonfile/b/bogo-sort.html

[4] Black, Paul E. (24 August 2009). "bubble sort". *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Retrieved 1 October 2014.

13.7 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. Problem 2-2, pg.40.
- Sorting in the Presence of Branch Prediction and Caches
- Fundamentals of Data Structures by Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed ISBN 81-7371-605-6

13.8 External links

- David R. Martin. "Animated Sorting Algorithms: Bubble Sort". – graphical demonstration and discussion of bubble sort
- "Lafore's Bubble Sort". (Java applet animation)
- (sequence A008302 in OEIS) Table (statistics) of the number of permutations of [n] that need k pair-swaps during the sorting.

Chapter 14

Insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Bentley shows a three-line C version, and a five-line optimized version[1]:116
- Efficient for (quite) small data sets
- More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(nk)$ when each element in the input is no more than k places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

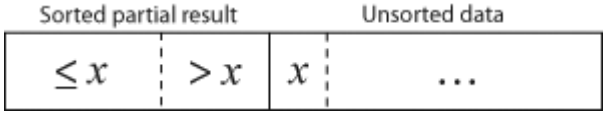
When people manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort.[2]

14.1 Algorithm

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

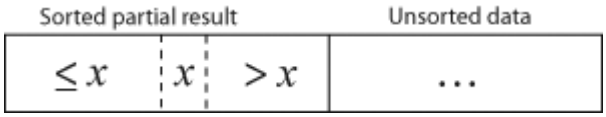
Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after k iterations has the property where the first $k + 1$ entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:



becomes

A graphical example of insertion sort.



with each element greater than x copied to the right as it is compared against x.

The most common variant of insertion sort, which operates on arrays, can be described as follows:

1. Suppose there exists a function called *Insert* designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.

2. To perform an insertion sort, begin at the left-most element of the array and invoke *Insert* to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

Pseudocode of the complete algorithm follows, where the arrays are zero-based:[1]:116

for i ← 1 **to** length(A) - 1 j ← i **while** j > 0 and A[j-1] > A[j] **swap** A[j] and A[j-1] j ← j - 1

The outer loop runs over all the elements except the first one, because the single-element prefix A[0:1] is trivially sorted, so the invariant that the first i+1 entries are sorted is true from the start. The inner loop moves element A[i] to its correct place so that after the loop, the first i+2 elements are sorted.

After expanding the "swap" operation in-place as t ← A[j]; A[j] ← A[j-1]; A[j-1] ← t (where t is a temporary variable), a slightly faster version can be produced that moves A[i] to its position in one go and only performs one assignment in the inner loop body:[1]:116

for i = 1 **to** length(A) - 1 x = A[i] j = i **while** j > 0 and A[j-1] > x A[j] = A[j-1] j = j - 1 A[j] = x

The new inner loop shifts elements to the right to clear a spot for x = A[i].

Note that although the common practice is to implement in-place, which requires checking the elements in-order, the order of checking (and removing) input elements is actually arbitrary. The choice can be made using almost any pattern, as long as all input elements are eventually checked (and removed from the input).

14.2 Best, worst, and average cases

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n^2)).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

Example: The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the key under consideration is underlined. The key that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

14.3 Relation to other sorting algorithms

Insertion sort is very similar to selection sort. As in selection sort, after k passes through the array, the first k elements are in sorted order. For selection sort these are the k smallest elements, while in insertion sort they are whatever the first k elements were in the unsorted array. Insertion sort's advantage is that it only scans as many elements as needed to determine the correct location of the k+1st element, while selection sort must scan all remaining elements to find the absolute smallest element.

Assuming the k+1st element's rank is random, insertion sort will on average require shifting half of the previous k elements, while selection sort always requires scanning all unplaced elements. So for unsorted input, insertion sort will usually perform about half as many comparisons as selection sort. If the input array is reverse-sorted, insertion sort performs as many comparisons as selection sort. If the input array is already sorted, insertion sort performs as few as $n-1$ comparisons, thus making insertion sort more efficient when given sorted or "nearly sorted" arrays.

While insertion sort typically makes fewer comparisons than selection sort, it requires more writes because the inner loop can require shifting large sections of the sorted portion of the array. In general, insertion sort will write to the array O(n^2) times, whereas selection sort will write only O(n) times. For this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading, such as with EEPROM or flash memory.

Some divide-and-conquer algorithms such as quicksort and mergesort sort by recursively dividing the list into smaller sublists which are then sorted. A useful optimization in practice for these algorithms is to use insertion sort for sorting small sublists, where insertion sort outperforms these more complex algorithms. The size of list for which insertion sort has the advantage varies by environment and implementation, but is typically between eight and twenty elements. A variant of this scheme runs quicksort with a constant cutoff K, then runs a single insertion sort on the final array:

proc quicksort(A, lo, hi) **if** hi - lo < K return pivot ← partition(A, lo, hi) quicksort(A, lo, pivot-1) quicksort(A, pivot + 1, hi) **proc** sort(A) quicksort(A, 0, length(A)) insertionsort(A)

This preserves the O(n lg n) expected time complexity of standard quicksort, because after running the quicksort procedure, the array A will be partially sorted in the sense that each element is at most K positions away from its final, sorted position. On such a partially sorted array, insertion sort will run at most K iterations of its inner loop, which is run $n-1$ times, so it has linear time complexity.[1]:121

14.4 Variants

D.L. Shell made substantial improvements to the algorithm; the modified version is called Shell sort. The sorting algorithm compares elements separated by a distance that decreases on each pass. Shell sort has distinctly improved running times in practical work, with two simple variants requiring O($n^{3/2}$) and O($n^{4/3}$) running time.

If the cost of comparisons exceeds the cost of swaps, as is the case for example with string keys stored by reference or with human interaction (such as choosing one of a pair displayed side-by-side), then using *binary insertion sort* may yield better performance. Binary insertion sort employs a binary search to determine the correct location to insert new elements, and therefore performs $\lceil \log_2(n) \rceil$ comparisons in the worst case, which is O(n log n). The algorithm as a whole still has a running time of O(n^2) on average because of the series of swaps required for each insertion.

The number of swaps can be reduced by calculating the position of multiple elements before moving them. For example, if the target position of two elements is calculated before they are moved into the right position, the number of swaps can be reduced by about 25% for random data. In the extreme case, this variant works similar to merge sort.

A variant named *binary merge sort* uses a *binary insertion sort* to sort groups of 32 elements, followed by a final sort using merge sort. It combines the speed of insertion sort on small data sets with the speed of merge sort on large data sets.[3]

To avoid having to make a series of swaps for each insertion, the input could be stored in a linked list, which allows elements to be spliced into or out of the list in constant-time when the position in the list is known. However, searching a linked list requires sequentially following the links to the desired position: a linked list does not have random access, so it cannot use a faster method such as binary search. Therefore, the running time required for searching is O(n) and the time for sorting is O(n^2). If a more sophisticated data structure (e.g., heap or binary tree) is used, the time required for searching and insertion can be reduced significantly; this is the essence of heap sort and binary tree sort.

In 2004 Bender, Farach-Colton, and Mosteiro published a new variant of insertion sort called *library sort* or *gapped insertion sort* that leaves a small number of unused spaces (i.e., "gaps") spread throughout the array. The benefit is that insertions need only shift elements over until a gap is reached. The authors show that this sorting algorithm runs with high probability in O(n log n) time.[4]

If a skip list is used, the insertion time is brought down to O(log n), and swaps are not needed because the skip list is implemented on a linked list structure. The final running time for insertion would be O(n log n).

List insertion sort is a variant of insertion sort. It reduces the number of movements.

14.4.1 List insertion sort code in C

If the items are stored in a linked list, then the list can be sorted with O(1) additional space. The algorithm starts with an initially empty (and therefore trivially sorted) list. The input items are taken off the list one at a time, and then inserted in the proper place in the sorted list. When the input list is empty, the sorted list has the desired result.

struct LIST * SortList1(struct LIST * pList) { // zero or one element in list if(pList == NULL || pList->pNext == NULL) return pList; // head is the first element of resulting sorted list struct LIST * head = NULL; while(pList != NULL) { struct LIST * current = pList; pList = pList->pNext; if(head == NULL || current->iValue < head->iValue) { // insert into the head of the sorted list // or as the first element into an empty sorted list current->pNext = head; head = current; } else { // insert current element into proper position in non-empty sorted list struct LIST * p = head; while(p != NULL) { if(p->pNext == NULL || // last element of the sorted list current->iValue < p->pNext->iValue) // middle of the list { // insert into middle of the sorted list or as the last element current->pNext = p->pNext; p->pNext = current; break; // done } p = p->pNext; } } } return head; }

The algorithm below uses a trailing pointer[5] for the insertion into the sorted list. A simpler recursive method rebuilds the list each time (rather than splicing) and can use O(n) stack space.

struct LIST { struct LIST * pNext; int iValue; }; struct LIST * SortList(struct LIST * pList) { // zero or one element in list if(!pList || !pList->pNext) return pList; /* build up the sorted array from the empty list */ struct LIST * pSorted = NULL; /* take items off the input list one by one until empty */ while (pList != NULL) { /* remember the head */ struct LIST * pHead = pList; /* trailing pointer for efficient splice */ struct LIST ** ppTrail = &pSorted; /* pop head off list */ pList = pList->pNext; /* splice head into sorted list at proper place */ while (!(*ppTrail == NULL || pHead->iValue < (*ppTrail)->iValue)) /* does head belong here? */ { /* no - continue down the list */ ppTrail = &(*ppTrail)->pNext; } pHead->pNext = *ppTrail; *ppTrail = pHead; } return pSorted; }

14.5 References

[1] Jon Bentley (1999). *Programming Pearls*. Addison-Wesley Professional.

[2] Robert Sedgewick, *Algorithms*, Addison-Wesley 1983 (chapter 8 p. 95)

[3] "Binary Merge Sort".

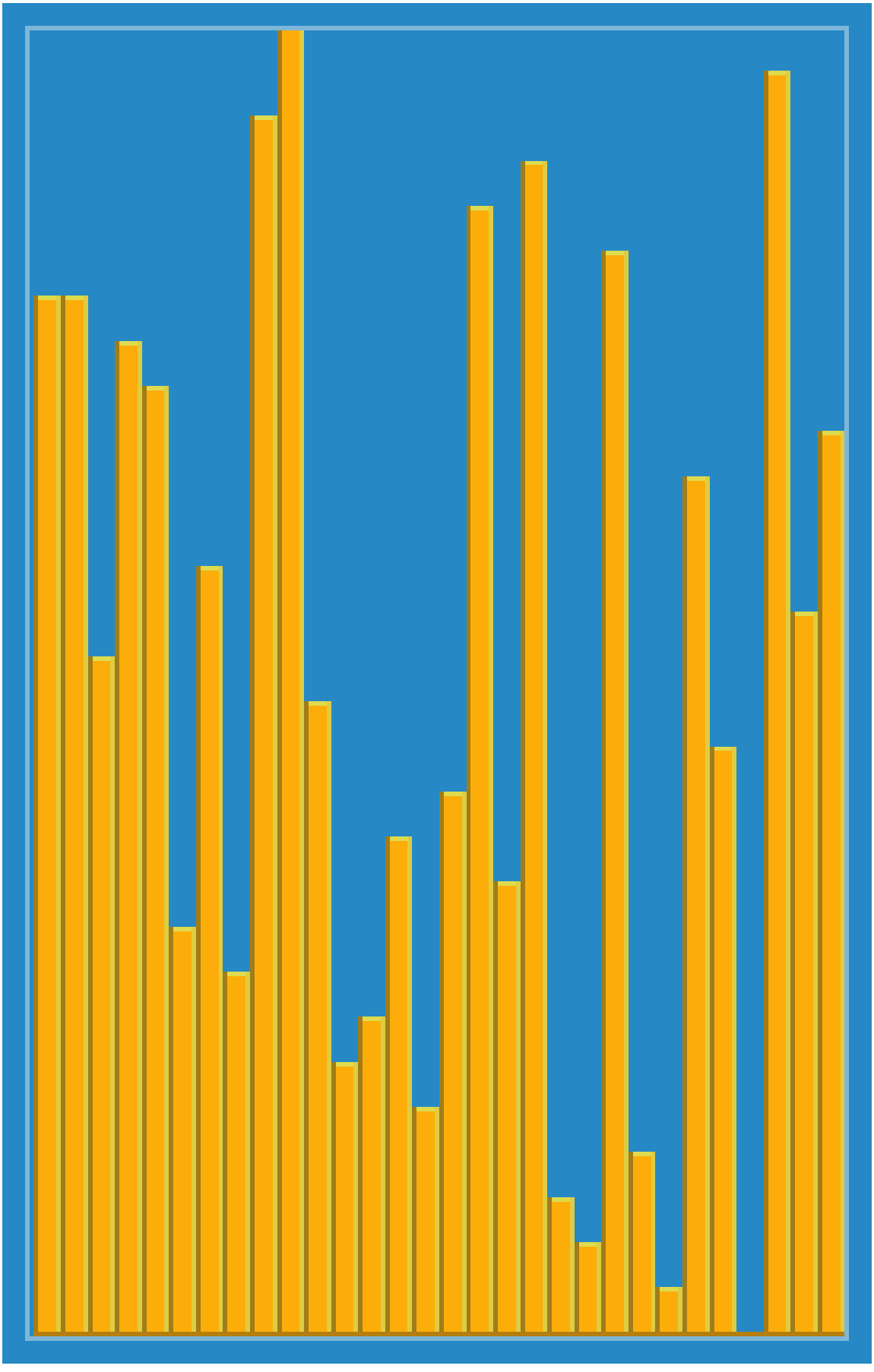
[4] Bender, Michael A.; Farach-Colton, Martín; Mosteiro, Miguel (2004), *Insertion Sort is O(n log n)*, PSU

[5] Hill, Curt (ed.), "Trailing Pointer Technique", *Euler*, Valley City State University, retrieved 22 September 2012.

- Bender, Michael A; Farach-Colton, Martín; Mosteiro, Miguel (2006), *Insertion Sort is O(n log n)* (PDF), SUNYSB; also http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.3758; republished? in *Theory of Computing Systems* (ACM) **39** (3), June 2006 http://dl.acm.org/citation.cfm?id=1132705 Missing or empty |title= (help).
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "2.1: Insertion sort", *Introduction to Algorithms* (second ed.), MIT Press and McGraw-Hill, pp. 15–21, ISBN 0-262-03293-7.
- Knuth, Donald (1998), "5.2.1: Sorting by Insertion", *The Art of Computer Programming*, 3. Sorting and Searching (second ed.), Addison-Wesley, pp. 80–105, ISBN 0-201-89685-0.
- Sedgewick, Robert (1983), "8", *Algorithms*, Addison-Wesley, pp. 95ff, ISBN 978-0-201-06672-2.

14.6 External links

- Adamovsky, John Paul, *Binary Insertion Sort – Scoreboard – Complete Investigation and C Implementation*, Pathcom.
- *Insertion Sort in C with demo*, Electrofriends.
- *Insertion Sort – a comparison with other O(n^2) sorting algorithms*, UK: Core war.
- *Animated Sorting Algorithms: Insertion Sort – graphical demonstration and discussion of insertion sort*, Sorting algorithms.
- *Category:Insertion Sort* (wiki), LiteratePrograms – implementations of insertion sort in various programming languages
- *InsertionSort*, Code raptors – colored, graphical Java applet that allows experimentation with the initial input and provides statistics
- Harrison, *Sorting Algorithms Demo*, CA: UBC – visual demonstrations of sorting algorithms (implemented in Java)
- *Insertion sort* (illustrated explanation), Algo list. Java and C++ implementations.



Animation of the insertion sort sorting a 30 element array.

Chapter 15

Merge sort

In computer science, **merge sort** (also commonly spelled **mergesort**) is an $O(n \log n)$ comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945.[1] A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and Neumann as early as 1948.[2]

15.1 Algorithm

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

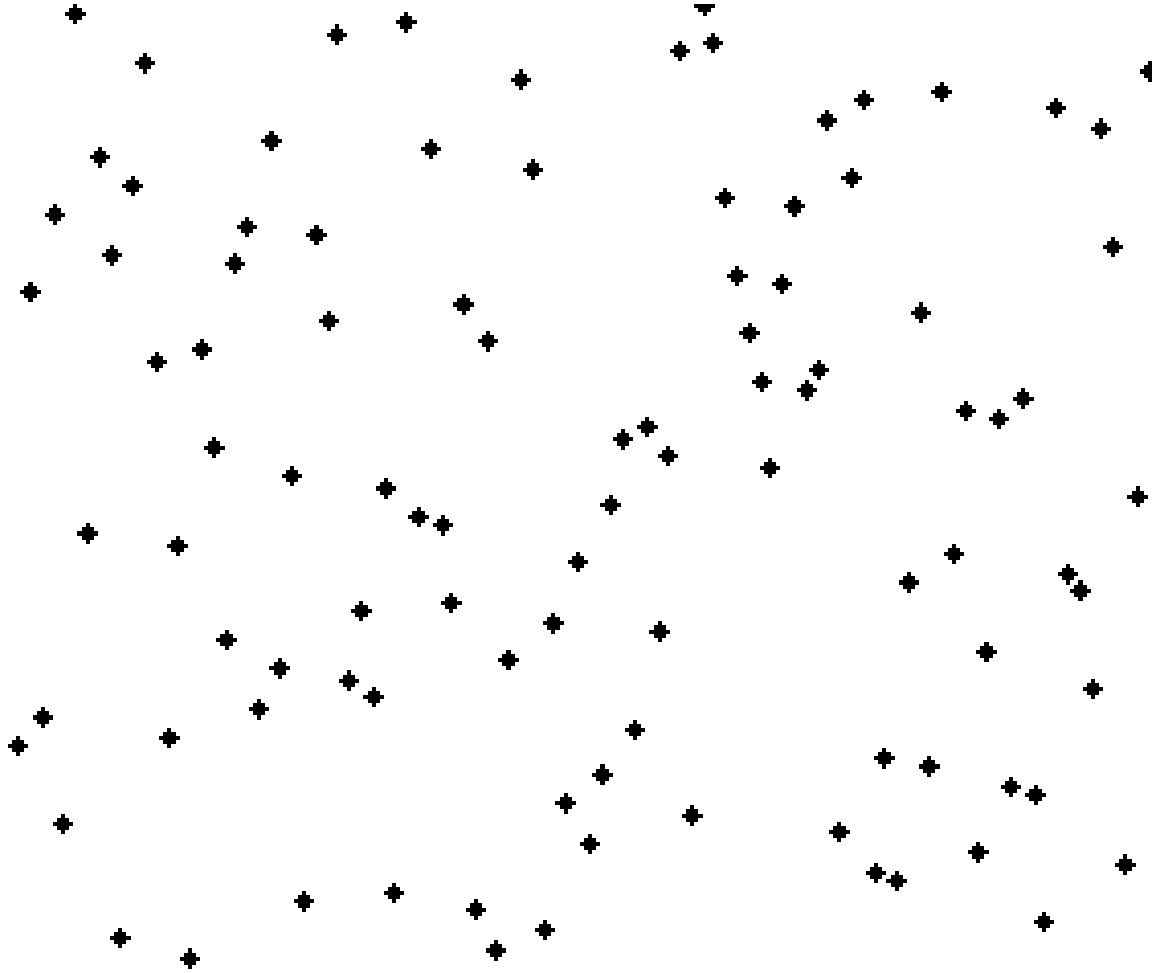
15.1.1 Top-down implementation

Example C like code using indices for top down merge sort algorithm that recursively splits the list (called *runs* in this example) into sublists until sublist size is 1, then merges those sublists to produce a sorted list. The copy back step could be avoided if the recursion alternated between two functions so that the direction of the merge corresponds with the level of recursion.

TopDownMergeSort(A[], B[], n) { TopDownSplitMerge(A, 0, n, B); } // iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set) TopDownSplitMerge(A[], iBegin, iEnd, B[]) { if(iEnd - iBegin < 2) // if run size == 1 return; // consider it sorted // recursively split runs into two halves until run size == 1, // then merge them and return back up the call chain iMiddle = (iEnd + iBegin) / 2; // iMiddle = mid point TopDownSplitMerge(A, iBegin, iMiddle, B); // split / merge left half TopDownSplitMerge(A, iMiddle, iEnd, B); // split / merge right half TopDownMerge(A, iBegin, iMiddle, iEnd, B); // merge the two half runs CopyArray(B, iBegin, iEnd, A); // copy the merged runs back to A } // left half is A[iBegin :iMiddle-1] // right half is A[iMiddle:iEnd-1] TopDownMerge(A[], iBegin, iMiddle, iEnd, B[]) { i0 = iBegin, i1 = iMiddle; // While there are elements in the left or right runs for (j = iBegin; j < iEnd; j++) { // If left run head exists and is <= existing right run head. if (i0 < iMiddle && (i1 >= iEnd || A[i0] <= A[i1])) B[j] = A[i0]; i0 = i0 + 1; else B[j] = A[i1]; i1 = i1 + 1; } } CopyArray(B[], iBegin, iEnd, A[]) { for(k = iBegin; k < iEnd; k++) A[k] = B[k]; }

15.1.2 Bottom-up implementation

Example C like code using indices for bottom up merge sort algorithm which treats the list as an array of n sublists (called *runs* in this example) of size 1, and iteratively merges sub-lists back and forth between two buffers:



Merge sort animation. The sorted elements are represented by dots.

void BottomUpMerge(A[], iLeft, iRight, iEnd, B[]) { i0 = iLeft; i1 = iRight; j; /* While there are elements in the left or right runs */ for (j = iLeft; j < iEnd; j++) { /* If left run head exists and is <= existing right run head */ if (i0 < iRight && (i1 >= iEnd || A[i0] <= A[i1])) { B[j] = A[i0]; i0 = i0 + 1; } else { B[j] = A[i1]; i1 = i1 + 1; } } } void CopyArray(B[], A[], n) { for(i = 0; i < n; i++) A[i] = B[i]; } /* array A[] has the items to sort; array B[] is a work array */ void BottomUpSort(A[], B[], n) { /* Each 1-element run in A is already "sorted". */ /* Make successively longer sorted runs of length 2, 4, 8, 16... until whole array is sorted. */ for (width = 1; width < n; width = 2 * width) { /* Array A is full of runs of length width. */ for (i = 0; i < n; i = i + 2 * width) { /* Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[] */ /* or copy A[i:n-1] to B[] (if(i+width >= n)) */ BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B); } /* Now work array B is full of runs of length 2*width. */ /* Copy array B to array A for next iteration. */ /* A more efficient implementation would swap the roles of A and B */ CopyArray(B, A, n); /* Now array A is full of runs of length 2*width. */ } }

15.1.3 Top-down implementation using lists

Pseudocode for top down merge sort algorithm which recursively divides the input list into smaller sublists until the sublists are trivially sorted, and then merges the sublists while returning up the call chain.

function merge_sort(*list* m) // Base case. A list of zero or one elements is sorted, by definition. **if** length(m) <= 1 **return** m // Recursive case. First, *divide* the list into equal-sized sublists. **var** *list* left, right **var** *integer* middle = length(m) / 2 **for each** x **in** m **before** middle add x to left **for each** x **in** m **after or equal** middle add x to right // Recursively sort both sublists left = merge_sort(left) right = merge_sort(right) // Then merge the now-sorted sublists. **return** merge(left, right)

In this example, the merge function merges the left and right sublists.

function merge(left, right) **var** *list* result **while** notempty(left) **and** notempty(right) **if** first(left) <= first(right) append first(left) to result left = rest(left) **else** append first(right) to result right = rest(right) *// either left or right may have elements left* **while** notempty(left) append first(left) to result left = rest(left) **while** notempty(right) append first(right) to result right = rest(right) **return** result

15.2 Natural merge sort

A natural merge sort is similar to a bottom up merge sort except that any naturally occurring runs (sorted sequences) in the input are exploited. In the bottom up merge sort, the starting point assumes each run is one item long. In practice, random input data will have many short runs that just happen to be sorted. In the typical case, the natural merge sort may not need as many passes because there are fewer runs to merge. In the best case, the input is already sorted (i.e., is one run), so the natural merge sort need only make one pass through the data. Example:

Start : 3--4--2--1--7--5--8--9--0--6 Select runs : 3--4 2 1--7 5--8--9 0--6 Merge : 2--3--4 1--5--7--8--9 0--6 Merge : 1--2--3--4--5--7--8--9 0--6 Merge : 0--1--2--3--4--5--6--7--8--9

Tournament replacement selection sorts are used to gather the initial runs for external sorting algorithms.

15.3 Analysis

In sorting n objects, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than ($n \lceil \lg n \rceil$ - $2^{\lceil \lg n \rceil}$ + 1), which is between ($n \lg n$ - n + 1) and ($n \lg n + n +$ O(lg n)).[3]

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches $\alpha \cdot n$ fewer than the worst case where $\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k+1} \approx 0.2645$.

In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case. In terms of moves, merge sort's worst case complexity is O(n log n)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

Merge sort's most common implementation does not sort in place ; therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only n/2 extra spaces).

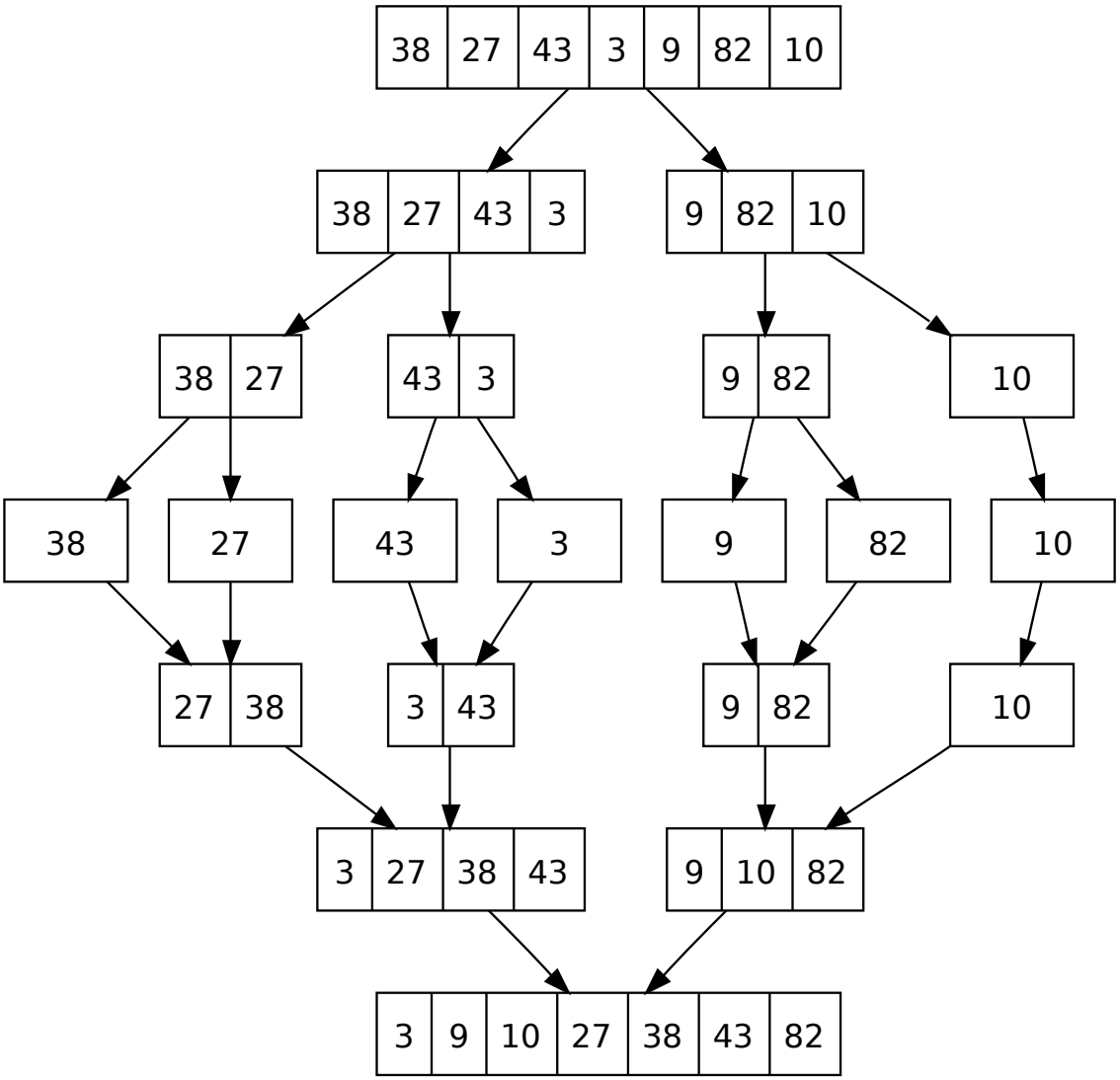
Merge sort also has some demerits. One is its use of 2n locations; the additional n locations are commonly used because merging two sorted sets in place is more complicated and would need more comparisons and move operations. But despite the use of this space the algorithm still does a lot of work: The contents of m are first copied into *left* and *right* and later into the list *result* on each invocation of *merge_sort* (variable names according to the pseudocode above).

15.4 Variants

Variants of merge sort are primarily concerned with reducing the space complexity and the cost of copying.

A simple alternative for reducing the space overhead to n/2 is to maintain *left* and *right* as a combined structure, copy only the *left* part of m into temporary space, and to direct the *merge* routine to place the merged output into m. With this version it is better to allocate the temporary space outside the *merge* routine, so that only one allocation is needed. The excessive copying mentioned previously is also mitigated, since the last pair of lines before the *return result* statement (function *merge* in the pseudo code above) become superfluous.

In-place sorting is possible, and still stable, but is more complicated, and slightly slower, requiring non-linearithmic quasilinear time O(n $\log^2$ n) One way to sort in-place is to merge the blocks recursively.[4] Like the standard merge



A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

sort, in-place merge sort is also a stable sort. In-place stable sorting of linked lists is simpler. In this case the algorithm does not use more space than that already used by the list representation, but the O(log(k)) used for the recursion trace.

An alternative to reduce the copying into multiple lists is to associate a new field of information with each key (the elements in m are called keys). This field will be used to link the keys and any associated information together in a sorted list (a key and its related information is called a record). Then the merging of the sorted lists proceeds by changing the link values; no records need to be moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used. This is a standard sorting technique, not restricted to merge sort.

A variant named *binary merge sort* uses a *binary insertion sort* to sort groups of 32 elements, followed by a final sort using merge sort. It combines the speed of insertion sort on small data sets with the speed of merge sort on large data sets.[5]

15.5 Use with tape drives

An external merge sort is practical to run using disk or tape drives when the data to be sorted is too large to fit into memory. External sorting explains how merge sort is implemented with disk drives. A typical tape drive sort uses

Merge sort type algorithms allowed large data sets to be sorted on early computers that had small random access memories by modern standards. Records were stored on magnetic tape and processed on banks of magnetic tape drives, such as these IBM 729s.

four tape drives. All I/O is sequential (except for rewinds at the end of each pass). A minimal implementation can get by with just 2 record buffers and a few program variables.

Naming the four tape drives as A, B, C, D, with the original data on A, and using only 2 record buffers, the algorithm is similar to Bottom-up implementation, using pairs of tape drives instead of arrays in memory. The basic algorithm can be described as follows:

1. Merge pairs of records from A; writing two-record sublists alternately to C and D.
2. Merge two-record sublists from C and D into four-record sublists; writing these alternately to A and B.
3. Merge four-record sublists from A and B into eight-record sublists; writing these alternately to C and D
4. Repeat until you have one list containing all the data, sorted --- in log2(n) passes.

Instead of starting with very short runs, usually a hybrid algorithm is used, where the initial pass will read many records into memory, do an internal sort to create a long run, and then distribute those long runs onto the output set. The step avoids many early passes. For example, an internal sort of 1024 records will save 9 passes. The internal sort is often large because it has such a benefit. In fact, there are techniques that can make the initial runs longer than the available internal memory.[6]

A more sophisticated merge sort that optimizes tape (and disk) drive usage is the polyphase merge sort.

15.6 Optimizing merge sort

On modern computers, locality of reference can be of paramount importance in software optimization, because multilevel memory hierarchies are used. Cache-aware versions of the merge sort algorithm, whose operations have

been specifically chosen to minimize the movement of pages in and out of a machine's memory cache, have been proposed. For example, the **tiled merge sort** algorithm stops partitioning subarrays when subarrays of size S are reached, where S is the number of data items fitting into a CPU's cache. Each of these subarrays is sorted with an in-place sorting algorithm such as insertion sort, to discourage memory swaps, and normal merge sort is then completed in the standard recursive fashion. This algorithm has demonstrated better performance on machines that benefit from cache optimization. (LaMarca & Ladner 1997)

Kronrod (1969) suggested an alternative version of merge sort that uses constant additional space. This algorithm was later refined. (Katajainen, Pasanen & Teuhola 1996)

Also, many applications of external sorting use a form of merge sorting where the input get split up to a higher number of sublists, ideally to a number for which merging them still makes the currently processed set of pages fit into main memory.

15.7 Parallel merge sort

Merge sort parallelizes well due to use of the divide-and-conquer method. Several parallel variants are discussed in the third edition of Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms*.[7] The first of these can be very easily expressed in a pseudocode with fork and join keywords:

/* *inclusive/exclusive indices* */ **procedure** mergesort(A, lo, hi): **if** lo+1 < hi: /* *if two or more elements* */ mid = ⌊(lo + hi) / 2⌋ **fork** mergesort(A, lo, mid) mergesort(A, mid, hi) **join** merge(A, lo, mid, hi)

This algorithm is a trivial modification from the serial version, but its speedup is not impressive: $\Theta(\log n)$. A parallel merge algorithm to not only parallelize the recursive division of the array, but also the merge operation leads to a better parallel sort that performs well in practice when combined with a fast stable sequential sort, such as insertion sort, and a fast sequential merge as a base case for merging small arrays.[8] Merge sort was one of the first sorting algorithms where optimal speed up was achieved, with Richard Cole using a clever subsampling algorithm to ensure $O(1)$ merge.[9] Other sophisticated parallel sorting algorithms can achieve the same or better time bounds with a lower constant. For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW PRAM with n processors by performing partitioning implicitly.[10] Powers[11] further shows that a pipelined version of Batcher's Bitonic Mergesort at $O(\log^2 n)$ time on a butterfly sorting network is in practice actually faster than his $O(\log n)$ sorts on a PRAM, and he provides detailed discussion of the hidden overheads in comparison, radix and parallel sorting.

15.8 Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$. On typical modern architectures, efficient quicksort implementations generally outperform mergesort for sorting RAM-based arrays. On the other hand, merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl). In Java, the Arrays.sort() methods use merge sort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.[12] Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7,[13] on the Android platform,[14] and in GNU Octave.[15]

15.9 Notes

[1] Knuth (1998, p. 158)

[2] Jyrki Katajainen and Jesper Larsson Träff (1997). "A meticulous analysis of mergesort programs".

[3] The worst case number given here does not agree with that given in Knuth's *Art of Computer Programming, Vol 3*. The discrepancy is due to Knuth analyzing a variant implementation of merge sort that is slightly sub-optimal

[4] A Java implementation of in-place stable merge sort

[5] "Binary Merge Sort".

[6] Selection sort. Knuth's snowplow. Natural merge.

[7] Cormen et al. 2009, pp. 797–805

[8] V. J. Duvanenko, "Parallel Merge Sort", Dr. Dobb's Journal, March 2011

[9] Cole, Richard (August 1988). "Parallel merge sort". *SIAM J. Comput.* **17** (4): 770–785. doi:10.1137/0217049

[10] Powers, David M. W. Parallelized Quicksort and Radixsort with Optimal Speedup, *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.

[11] David M. W. Powers, Parallel Unification: Practical Complexity, Australasian Computer Architecture Workshop, Flinders University, January 1995

[12] OpenJDK Subversion

[13] jjb. "Commit 6804124: Replace "modified mergesort" in java.util.Arrays.sort with timsort". *Java Development Kit 7 Hg repo*. Retrieved 24 Feb 2011.

[14] "Class: java.util.TimSort<T>". *Android JDK Documentation*. Retrieved 19 Jan 2015.

[15] "liboctave/util/oct-sort.cc". *Mercurial repository of Octave source code*. Lines 23-25 of the initial comment block. Retrieved 18 Feb 2013. Code stolen in large part from Python's, listobject.c, which itself had no license header. However, thanks to Tim Peters for the parts of the code I ripped-off.

15.10 References

- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
- Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). "Practical in-place mergesort". *Nordic Journal of Computing* **3**. pp. 27–40. ISSN 1236-6064. Retrieved 2009-04-04.. Also Practical In-Place Mergesort. Also
- Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *Sorting and Searching*. The Art of Computer Programming **3** (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.
- Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field". *Soviet Mathematics - Doklady* **10**. p. 744.
- LaMarca, A.; Ladner, R. E. (1997). "The influence of caches on the performance of sorting". *Proc. 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97)*: 370–379.
- Sun Microsystems. "Arrays API". Retrieved 2007-11-19.
- Sun Microsystems. "java.util.Arrays.java". Retrieved 2007-11-19.

15.11 External links

- Animated Sorting Algorithms: Merge Sort – graphical demonstration and discussion of array-based merge sort
- Dictionary of Algorithms and Data Structures: Merge sort
- Mergesort applet with "level-order" recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.1 - Merge Sort

Chapter 16

Quicksort

Quicksort (sometimes called **partition-exchange sort**) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1960, it is still a very commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.[1]

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.

Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare.

16.1 History

The quicksort algorithm was developed in 1960 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University. At that time, Hoare worked in a project on machine translation for the National Physical Laboratory. He developed the algorithm in order to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.[2]

Quicksort gained widespread adoption, appearing, for example, in Unix as the default library sort function, hence it lent its name to the C standard library function qsort[3] and in the reference implementation of Java. It was analyzed extensively by Robert Sedgewick, who wrote his Ph.D. thesis about the algorithm and suggested several improvements.[3]

16.2 Algorithm

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

The steps are:

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

The base case of the recursion is arrays of size zero or one, which never need to be sorted. In pseudocode, a quicksort that sorts elements lo through hi (inclusive) of an array A can be expressed compactly as[4]:171

quicksort(A, lo, hi): if lo < hi: p := partition(A, lo, hi) quicksort(A, lo, p - 1) quicksort(A, p + 1, hi)
Sorting the entire array is accomplished by calling quicksort(A, 1, length(A)). The partition operation is step 2 from the algorithm description above; it can be defined as:

// lo is the index of the leftmost element of the subarray // hi is the index of the rightmost element of the subarray (inclusive) partition(A, lo, hi) pivotIndex := choosePivot(A, lo, hi) pivotValue := A[pivotIndex] *// put the chosen pivot at A[hi]* swap A[pivotIndex] and A[hi] storeIndex := lo *// Compare remaining array elements against pivotValue = A[hi]* **for** i **from** lo **to** hi−1, **inclusive if** A[i] < pivotValue swap A[i] and A[storeIndex] storeIndex := storeIndex + 1 swap A[storeIndex] and A[hi] // *Move pivot to its final place* **return** storeIndex

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

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

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

16.2.1 Implementation issues

Choice of pivot

In the very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element. Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case. The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot (as recommended by Sedgewick).[5] This "median of three" rule counters the case of sorted (or reverse-sorted) input, and gives a better estimate of the optimal pivot (the true median) than selecting any single element, when no information about the ordering of the input is known.

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

Repeated elements

With a partitioning algorithm such as the one described above (even with one that chooses good pivot values), quicksort exhibits poor performance for inputs that contain many repeated elements. The problem is clearly apparent when all the input elements are equal: at each recursion, the left partition is empty (no input values are less than the pivot), and the right partition has only decreased by one element (the pivot is removed). Consequently, the algorithm takes quadratic time to sort an array of equal values.

To solve this problem (sometimes called the Dutch national flag problem[3]), an alternative linear-time partition routine can be used that separates the values into three groups: values less than the pivot, values equal to the pivot, and values greater than the pivot. (Bentley and McIlroy call this a "fat partition" and note that it was already implemented in the qsort of Version 7 Unix.[3]) The values equal to the pivot are already sorted, so only the less-than and greater-than partitions need to be recursively sorted. In pseudocode, the quicksort algorithm becomes

quicksort(A, lo, hi) **if** lo < hi p := pivot(A, lo, hi) left, right := partition(A, p, lo, hi) *// note: multiple return values* quicksort(A, lo, left) quicksort(A, right, hi)

The best case for the algorithm now occurs when all elements are equal (or are chosen from a small set of $k \ll n$ elements). In the case of all equal elements, the modified quicksort will perform at most two recursive calls on empty subarrays and thus finish in linear time.

Optimizations

Two other important optimizations, also suggested by Sedgewick and widely used in practice are:[6][7]

- To make sure at most $O(\log n)$ space is used, recurse first into the smaller side of the partition, then use a tail call to recurse into the other.
- Use insertion sort, which has a smaller constant factor and is thus faster on small arrays, for invocations on small arrays (i.e. where the length is less than a threshold k determined experimentally). This can be implemented by simply stopping the recursion when less than k elements are left, leaving the entire array k-sorted: each element will be at most k positions away from its final position. Then, a single insertion sort pass[8]:117 finishes the sort in $O(kn)$ time. A separate insertion sort of each small segment as they are identified adds the overhead of starting and stopping many small sorts, but avoids wasting effort comparing keys across the many segment boundaries, where keys will be in order due to the workings of the quicksort process.

Parallelization

Quicksort's divide-and-conquer formulation makes it amenable to parallelization using task parallelism. The partitioning step is accomplished through the use of a parallel prefix sum algorithm to compute an index for each array element in its section of the partitioned array.[9] Given an array of size n, the partitioning step performs $O(n)$ work in $O(\log n)$ time and requires $O(n)$ additional scratch space. After the array has been partitioned, the two partitions can be sorted recursively in parallel. Assuming an ideal choice of pivots, parallel quicksort sorts an array of size n in $O(n \log n)$ work in $O(\log^2 n)$ time using $O(n)$ additional space.

Quicksort has some disadvantages when compared to alternative sorting algorithms, like merge sort, which complicate its efficient parallelization. The depth of quicksort's divide-and-conquer tree directly impacts the algorithm's scalability, and this depth is highly dependent on the algorithm's choice of pivot. Additionally, it is difficult to parallelize the partitioning step efficiently in-place. The use of scratch space simplifies the partitioning step, but increases the algorithm's memory footprint and constant overheads.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds.[10] For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW PRAM with n processors by performing partitioning implicitly.[11]

16.3 Formal analysis

16.3.1 Average-case analysis using discrete probability

To sort an array of n distinct elements, quicksort takes $O(n \log n)$ time in expection, averaged over all $n!$ permutations of n elements with equal probability. Why? For a start, it is not hard to see that the partition operation takes $O(n)$ time.

In the most unbalanced case, each time we perform a partition we divide the list into two sublists of size 0 and $n - 1$ (for example, if all elements of the array are equal). This means each recursive call processes a list of size one less than the previous list. Consequently, we can make $n - 1$ nested calls before we reach a list of size 1. This means that the call tree is a linear chain of $n - 1$ nested calls. The ith call does $O(n - i)$ work to do the partition, and $\sum_{i=0}^{n}(n-i) = O(n^2)$, so in that case Quicksort takes $O(n^2)$ time. That is the worst case: given knowledge of which comparisons are performed by the sort, there are adaptive algorithms that are effective at generating worst-case input for quicksort on-the-fly, regardless of the pivot selection strategy.[12]

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\log_2 n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\log_2 n$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only $O(n)$ time all together (each

call has some constant overhead, but since there are only $O(n)$ calls at each level, this is subsumed in the $O(n)$ factor). The result is that the algorithm uses only $O(n \log n)$ time.

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

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

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

16.3.2 Average-case analysis using recurrences

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

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

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

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

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

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

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

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

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

This means that, on average, quicksort performs only about 39% worse than in its best case. In this sense it is closer to the best case than the worst case. Also note that a comparison sort cannot use less than $\log_2(n!)$ comparisons on average to sort n items (as explained in the article Comparison sort) and in case of large n, Stirling's approximation yields $\log_2(n!) \approx n(\log_2 n - \log_2 e)$, so quicksort is not much worse than an ideal comparison sort. This fast average runtime is another reason for quicksort's practical dominance over other sorting algorithms.

16.3.3 Analysis of randomized quicksort

Using the same analysis, one can show that randomized quicksort has the desirable property that, for any input, it requires only $O(n \log n)$ expected time (averaged over all choices of pivots). However, there also exists a combinatorial proof.

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

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

By linearity of expectation, the expected value $\mathbb{E}[C]$ of C is $\mathbb{E}[C] = \sum_i \sum_{j<i} \Pr(c_{i,j})$.

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

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

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

16.3.4 Space complexity

The space used by quicksort depends on the version used.

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

- in-place partitioning is used. This unstable partition requires $O(1)$ space.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(\log n)$ space. Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack. This idea, as discussed above, was described by R. Sedgewick, and keeps the stack depth bounded by $O(\log n)$.[5][13]

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

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

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

16.4 Relation to other algorithms

Quicksort is a space-optimized version of the binary tree sort. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls. The algorithms make exactly the same comparisons, but in a different order. An often desirable property of a sorting algorithm is stability - that is the order of elements that compare equal is not changed, allowing controlling order of multikey tables (e.g. directory or folder listings) in a natural way. This property is hard to maintain for in situ (or in place) quicksort (that

uses only constant additional space for pointers and buffers, and logN additional space for the management of explicit or implicit recursion). For variant quicksorts involving extra memory due to representations using pointers (e.g. lists or trees) or files (effectively lists), it is trivial to maintain stability. The more complex, or disk-bound, data structures tend to increase time cost, in general making increasing use of virtual memory or disk.

The most direct competitor of quicksort is heapsort. Heapsort's worst-case running time is always $O(n \log n)$. But, heapsort is assumed to be on average somewhat slower than standard in-place quicksort. This is still debated and in research, with some publications indicating the opposite.[14][15] Introsort is a variant of quicksort that switches to heapsort when a bad case is detected to avoid quicksort's worst-case running time.

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

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

16.4.1 Selection-based pivoting

A selection algorithm chooses the kth smallest of a list of numbers; this is an easier problem in general than sorting. One simple but effective selection algorithm works nearly in the same manner as quicksort, and is accordingly known as quickselect. The difference is that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist which contains the desired element. This change lowers the average complexity to linear or $O(n)$ time, which is optimal for selection, but worst-case time is still $O(n^2)$.

A variant of quickselect, the median of medians algorithm, chooses pivots more carefully, ensuring that the pivots are near the middle of the data (between the 30th and 70th percentiles), and thus has guaranteed linear time – worst-case $O(n)$. This same pivot strategy can be used to construct a variant of quickselect (median of medians quicksort) with worst-case $O(n)$ time. However, the overhead of choosing the pivot is significant, so this is generally not used in practice.

More abstractly, given a worst-case $O(n)$ selection algorithm, one can use it to find the ideal pivot (the median) at every step of quicksort, producing a variant with worst-case $O(n \log n)$ running time. In practical implementations this variant is considerably slower on average, but it is of theoretical interest, showing how an optimal selection algorithm can yield an optimal sorting algorithm.

16.4.2 Variants

Multi-pivot quicksort Instead of partitioning into two subarrays using a single pivot, partition into some s number of subarrays using $s - 1$ pivots. While the dual-pivot case ($s = 3$) was considered by Sedgewick and others already in the mid-1970s, the resulting algorithms were not faster in practice than the "classical" quicksort.[16] However, a version of dual-pivot quicksort developed by Yaroslavskiy in 2009[17] turned out to be fast enough to warrant implementation in Java 7, as the standard algorithm to sort arrays of primitives (sorting arrays of objects is done using Timsort).[18]

Balanced quicksort Choose a pivot likely to represent the middle of the values to be sorted, and then follow the regular quicksort algorithm.

External quicksort The same as regular quicksort except the pivot is replaced by a buffer. First, read the M/2 first and last elements into the buffer and sort them. Read the next element from the beginning or end to balance writing. If the next element is less than the least of the buffer, write it to available space at the beginning. If greater than the greatest, write it to the end. Otherwise write the greatest or least of the buffer, and put the next element in the buffer. Keep the maximum lower and minimum upper keys written to avoid resorting middle elements that are in order. When done, write the buffer. Recursively sort the smaller partition, and loop to sort

the remaining partition. This is a kind of three-way quicksort in which the middle partition (buffer) represents a sorted subarray of elements that are *approximately* equal to the pivot.

Three-way radix quicksort Developed by Sedgewick and also known as **multikey quicksort**, it is a combination of radix sort and quicksort. Pick an element from the array (the pivot) and consider the first character (key) of the string (multikey). Partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character. Recursively sort the "less than" and "greater than" partitions on the same character. Recursively sort the "equal to" partition by the next character (key). Given we sort using bytes or words of length W bits, the best case is O(KN) and the worst case $O(2^K N)$ or at least $O(N^2)$ as for standard quicksort, given for unique keys $N<2^K$, and K is a hidden constant in all standard comparison sort algorithms including quicksort. This is a kind of three-way quicksort in which the middle partition represents a (trivially) sorted subarray of elements that are *exactly* equal to the pivot.

Quick radix sort Also developed by Powers as an o(K) parallel PRAM algorithm. This is again a combination of radix sort and quicksort but the quicksort left/right partition decision is made on successive bits of the key, and is thus O(KN) for N K-bit keys. Note that all comparison sort algorithms effectively assume an ideal K of O(logN) as if k is smaller we can sort in O(N) using a hash table or integer sorting, and if K >> logN but elements are unique within O(logN) bits, the remaining bits will not be looked at by either quicksort or quick radix sort, and otherwise all comparison sorting algorithms will also have the same overhead of looking through O(K) relatively useless bits but quick radix sort will avoid the worst case $O(N^2)$ behaviours of standard quicksort and quick radix sort, and will be faster even in the best case of those comparison algorithms under these conditions of uniqueprefix(K) >> logN. See Powers [19] for further discussion of the hidden overheads in comparison, radix and parallel sorting.

16.4.3 Generalization

Richard Cole and David C. Kandathil, in 2004, discovered a one-parameter family of sorting algorithms, called partition sorts, which on average (with all input orderings equally likely) perform at most $n \log n + O(n)$ comparisons (close to the information theoretic lower bound) and $\Theta(n \log n)$ operations; at worst they perform $\Theta(n \log^2 n)$ comparisons (and also operations); these are in-place, requiring only additional $O(\log n)$ space. Practical efficiency and smaller variance in performance were demonstrated against optimised quicksorts (of Sedgewick and Bentley-McIlroy).[20]

16.5 See also

- Introsort
- Flashsort

16.6 Notes

[1] Skiena, Steven S. (2008). *The Algorithm Design Manual*. Springer. p. 129. ISBN 978-1-84800-069-8.

[2] Shustek, L. (2009). "Interview: An interview with C.A.R. Hoare". *Comm. ACM* **52** (3): 38–41. doi:10.1145/1467247.1467261.

[3] Bentley, Jon L.; McIlroy, M. Douglas (1993). "Engineering a sort function". *Software—Practice and Experience* **23** (11): 1249–1265. doi:10.1002/spe.4380231105.

[4] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.

[5] Sedgewick, Robert (1 September 1998). *Algorithms In C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4* (3 ed.). Pearson Education. ISBN 978-81-317-1291-7. Retrieved 27 November 2012.

[6] qsort.c in GNU libc: ,

[7] http://www.ugrad.cs.ubc.ca/~{}cs260/chnotes/ch6/Ch6CovCompiled.html

[8] Jon Bentley (1999). *Programming Pearls*. Addison-Wesley Professional.

[9] Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, Quicksort and Sorting Lower Bounds, *Parallel and Sequential Data Structures and Algorithms*. 2013.

[10] Miller, Russ; Boxer, Laurence (2000). *Algorithms sequential & parallel: a unified approach*. Prentice Hall. ISBN 978-0-13-086373-7. Retrieved 27 November 2012.

[11] David M. W. Powers, Parallelized Quicksort and Radixsort with Optimal Speedup, *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.

[12] McIlroy, M. D. (1999). "A killer adversary for quicksort". *Software: Practice and Experience* **29** (4): 341–237. doi:10.1002/(SICI)1097-024X(19990410)29:4<341::AID-SPE237>3.3.CO;2-I.

[13] Sedgewick, R. (1978). "Implementing Quicksort programs". *Comm. ACM* **21** (10): 847–857. doi:10.1145/359619.359631.

[14] Hsieh, Paul (2004). "Sorting revisited.". www.azillionmonkeys.com. Retrieved 26 April 2010.

[15] MacKay, David (1 December 2005). "Heapsort, Quicksort, and Entropy". users.aims.ac.za/~{}mackay. Retrieved 26 April 2010.

[16] Wild, Sebastian; Nebel, Markus E. (2012). *Average case analysis of Java 7's dual pivot quicksort*. European Symposium on Algorithms. arXiv:1310.7409.

[17] http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628

[18] "Arrays". *Java Platform SE 7*. Oracle. Retrieved 4 September 2014.

[19] David M. W. Powers, Parallel Unification: Practical Complexity, Australasian Computer Architecture Workshop, Flinders University, January 1995

[20] Richard Cole, David C. Kandathil: "The average case analysis of Partition sorts", European Symposium on Algorithms, 14–17 September 2004, Bergen, Norway. Published: Lecture Notes in Computer Science 3221, Springer Verlag, pp. 240-251.

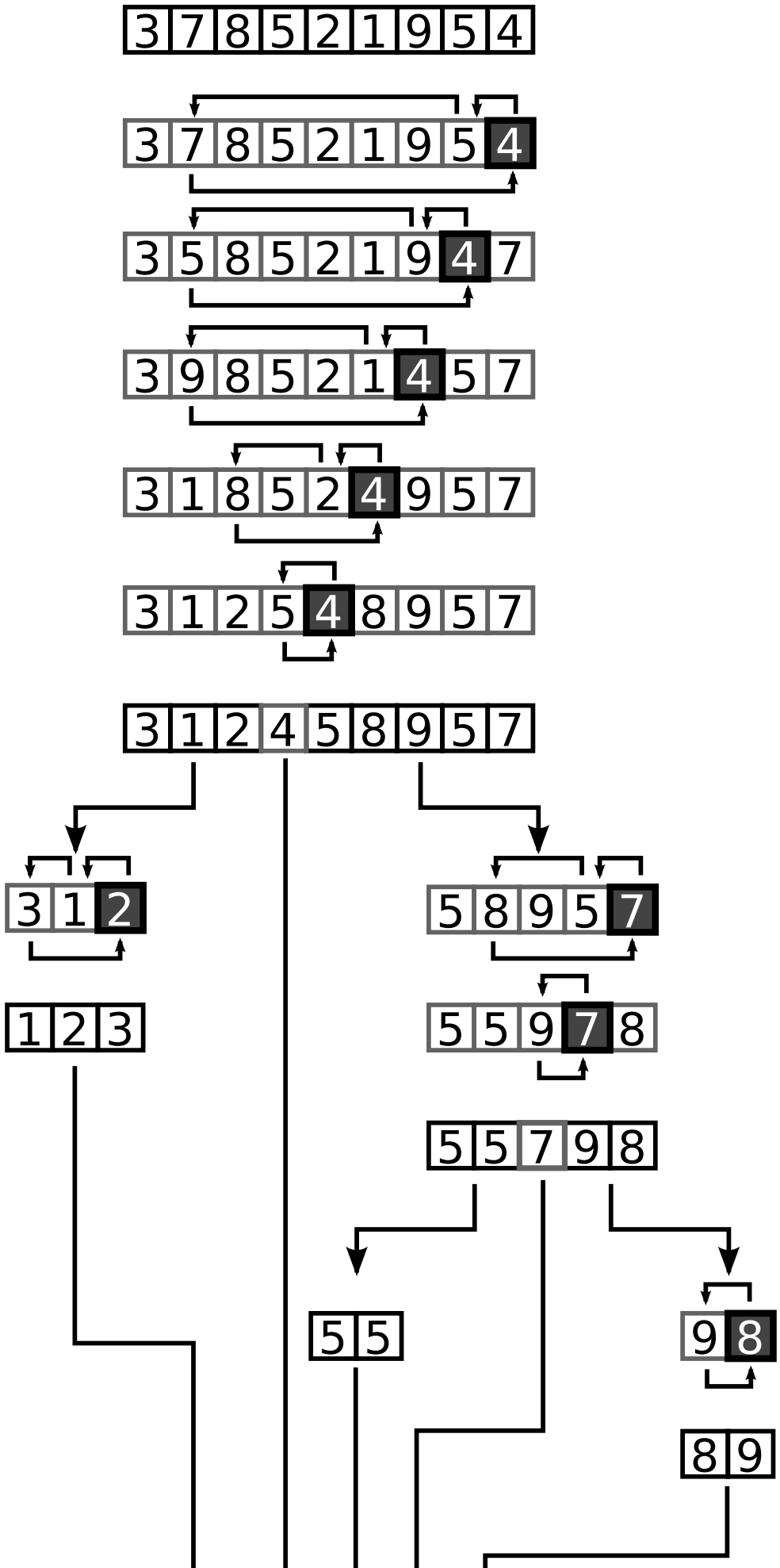
16.7 References

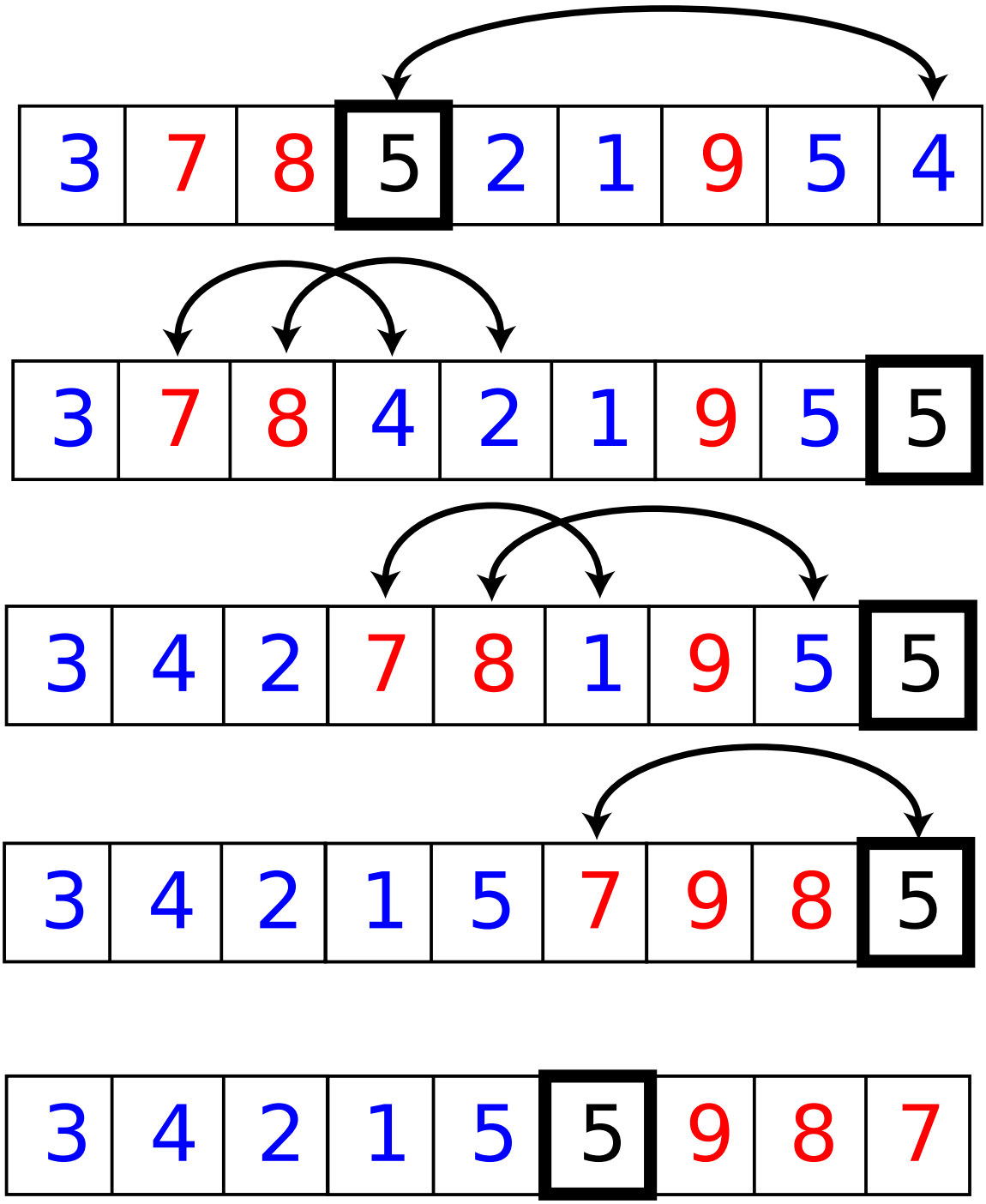
- Sedgewick, R. (1978). "Implementing Quicksort programs". *Comm. ACM* **21** (10): 847–857. doi:10.1145/359619.359631.
- Dean, B. C. (2006). "A simple expected running time analysis for randomized "divide and conquer" algorithms". *Discrete Applied Mathematics* **154**: 1–5. doi:10.1016/j.dam.2005.07.005.
- Hoare, C. A. R. (1961). "Algorithm 63: Partition". *Comm. ACM* **4** (7): 321. doi:10.1145/366622.366642.
- Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". *Comm. ACM* **4** (7): 321. doi:10.1145/366622.366644.
- Hoare, C. A. R. (1961). "Algorithm 65: Find". *Comm. ACM* **4** (7): 321–322. doi:10.1145/366622.366647.
- Hoare, C. A. R. (1962). "Quicksort". *Comput. J.* **5** (1): 10–16. doi:10.1093/comjnl/5.1.10. (Reprinted in Hoare and Jones: *Essays in computing science*, 1989.)
- Musser, David R. (1997). "Introspective Sorting and Selection Algorithms". *Software: Practice and Experience* (Wiley) **27** (8): 983–993. doi:10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#.
- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 113–122 of section 5.2.2: Sorting by Exchanging.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 7: Quicksort, pp. 145–164.
- A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting." Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997. pp. 370–379.
- Faron Moller. Analysis of Quicksort. CS 332: Designing Algorithms. Department of Computer Science, Swansea University.

- Martínez, C.; Roura, S. (2001). "Optimal Sampling Strategies in Quicksort and Quickselect". *SIAM J. Comput.* **31** (3): 683–705. doi:10.1137/S0097539700382108.
- Bentley, J. L.; McIlroy, M. D. (1993). "Engineering a sort function". *Software: Practice and Experience* **23** (11): 1249–1265. doi:10.1002/spe.4380231105.

16.8 External links

- Animated Sorting Algorithms: Quick Sort – graphical demonstration and discussion of quick sort
- Animated Sorting Algorithms: 3-Way Partition Quick Sort – graphical demonstration and discussion of 3-way partition quick sort
- Interactive Tutorial for Quicksort
- Quicksort applet with "level-order" recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.2 - Quicksort
- Literate implementations of Quicksort in various languages on LiteratePrograms
- A colored graphical Java applet which allows experimentation with initial state and shows statistics
- An in-place, stable Quicksort (Java) which runs in $O(n \log n \log n)$ time.


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



In-place partition in action on a small list. The boxed element is the pivot element, blue elements are less or equal, and red elements are larger.

16.9 Text and image sources, contributors, and licenses

16.9.1 Text

- **Data structure** *Source:* http://en.wikipedia.org/wiki/Data%20structure?oldid=647691145 *Contributors:* LC, Ap, -- April, Andre Engels, Karl E. V. Palmen, XJaM, Arvindn, Ghyll, Michael Hardy, TakuyaMurata, Minesweeper, Ahoerstemeier, Nanshu, Kingturtle, Glenn, UserGoogol, Jiang, Edaelon, Nikola Smolenski, Dcoetzee, Chris Lundberg, Populus, Traroth, Mrjeff, Robbot, Noldoaran, Craig Stuntz, Altenmann, Babbage, Mushroom, Seth Ilys, GreatWhiteNortherner, Tobias Bergemann, Giftlite, DavidCary, Esap, Jorge Stolfi, Siroxo, Pgan002, Kjetil r, Lancekt, Jacob grace, Pale blue dot, Andreas Kaufmann, Corti, Wrp103, MisterSheik, Lycurgus, Shanes, Viriditas, Vortexrealm, Obradovic Goran, Helix84, Mdd, Jumbuck, Alansohn, Liao, Tablizer, Yamla, PaePae, ReyBrujo, Derbeth, Forderud, Mahanga, Bushytails, Mindmatrix, Carlette, Ruud Koot, Easyas12c, TreveX, Bluemoose, Abd, Palica, Mandarax, Yoric, Qwertyus, Koavf, Ligulem, GeorgeBills, Margosbot, Fragglet, RexNL, Fresheneesz, Butros, Chobot, Tas50, Banaticus, YurikBot, RobotE, Hairy Dude, Piet Delport, Mipadi, Grafen, Dmoss, Tony1, Googl, Ripper234, Closedmouth, Vicarious, JLaTondre, GrinBot, TuukkaH, SmackBot, Reedy, DCDuring, Thunderboltz, BurntSky, Gilliam, Ohnoitsjamie, EncMstr, MalafayaBot, Nick Levine, Frap, Allan McInnes, Khukri, Ryan Roos, Sethwoodworth, Er Komandante, SashatoBot, Demicx, Soumyasch, Antonielly, SpyMagician, Loadmaster, Noah Salzman, Mr Stephen, Alhoori, Sharcho, Caiaffa, Iridescent, CRGreathouse, Ahy1, FinalMinuet, Requestion, Nnp, Peterdjones, GPhilip, Pascal.Tesson, Qwyrxian, MTA, Thadius856, AntiVandalBot, Widefox, Seaphoto, Jirka6, Dougher, Tom 99, Lanov, MER-C, Wikilolo, Wmbolle, Rhwawn, Nyq, Wwmbes, David Eppstein, User A1, Cpl Syx, Oicumayberight, Gwern, MasterRadius, Rettetast, Lithui, Sanjay742, Rrwright, Marcin Suwalczan, Jimmytharpe, TXiKiBoT, Eve Hall, Vipinhari, Coldfire82, DragonLord, Rozmichelle, Falcon8765, Spinningspark, Spitfire8520, Haiviet, SieBot, Caltas, Eurooppa, Ham Pastrami, Jerryobject, Strife911, Ramkumaran7, Nskillen, DancingPhilosopher, Digisus, Tanvir Ahmmed, ClueBot, Spellsinger180, Justin W Smith, The Thing That Should Not Be, Rodhullandemu, Sundar sando, Garyzx, Adrianwn, Abhishek.kumar.ak, Excirial, Alexbot, Erebus Morgaine, Arjayay, Morel, DumZiBoT, XLinkBot, Paushali, Pgallert, Galzigler, Alexius08, MystBot, Dsimic, Jncraton, MrOllie, EconoPhysicist, Publichealthguru, Tide rolls, يانم, Teles, سعى, Legobot, Luckas-bot, Yobot, Fraggle81, SteelPangolin, Jim1138, Kingpin13, Materialscientist, ArthurBot, Xqbot, Pur3r4ngelw, Miym, DAndC, RibotBOT, Shadowjams, Methcub, Prari, FrescoBot, Liridon, Mark Renier, Hypersonic12, Rameshngbot, MertyWiki, Thompsonb24, Profvalente, FoxBot, Laurențiu Dascălu, Lotje, Bharatshettybarkur, Tbhotch, Thinktdub, Kh31311, Vineetzone, Uriah123, DRAGON BOOSTER, EmausBot, Apoctyliptic, Thecheesykid, ZéroBot, MithrandirAgain, EdEColbert, IGeMiNix, Mentibot, BioPupil, Chandraguptamaurya, Rocketrod1960, Raveendra Lakpriya, ClueBot NG, Aks1521, Widr, Danim, Jorgenev, Orzechowskid, Gmharhar, HMSSolent, Wbm1058, Walk&check, Panchobook, Richfaber, SoniyaR, Yashykt, Cncmaster, Sallupandit, Доктор прагматик, Sgord512, Anderson, Vishnu0919, Varma rockzz, Frosty, Hernan mvs, Forgot to put name, I am One of Many, Bereajan, Gauravxpress, Haeynzen, Gambhir.jagmeet, Richard Yin, Jrachiele, Guturu Bhuvanamitra, TranquilHope, Iliazm and Anonymous: 348
- **Array data structure** *Source:* http://en.wikipedia.org/wiki/Array%20data%20structure?oldid=644965427 *Contributors:* The Anome, Ed Poor, Andre Engels, Tsja, B4hand, Patrick, RTC, Michael Hardy, Norm, Nixdorf, Graue, TakuyaMurata, Alfio, Ellywa, Julesd, Cgs, Poor Yorick, Rossami, Dcoetzee, Dysprosia, Jogloran, Wernher, Fvw, Sewing, Robbot, Josh Cherry, Fredrik, Lowellian, Wikibot, Jleedev, Giftlite, DavidCary, Massysett, BenFrantzDale, Lardarse, Ssd, Jorge Stolfi, Macrakis, Jonathan Grynspan, Lockeownzj00, Beland, Vanished user 1234567890, Karol Langner, Icairns, Simoneau, Jh51681, Andreas Kaufmann, Mattb90, Corti, Jkl, Rich Farmbrough, Guanabot, Qutezuce, ESkog, ZeroOne, Danakil, MisterSheik, G worroll, Spoon!, Army1987, Func, Rbj, Mdd, Jumbuck, Mr Adequate, Atanamir, Krischik, Tauwasser, ReyBrujo, Suruena, Rgrig, Forderud, Beliavsky, Beej71, Mindmatrix, Jimbryho, Ruud Koot, Jeff3000, Grika, Palica, Gerbrant, Graham87, Kbdank71, Zzedar, Ketiltrout, Bill37212, Ligulem, Yamamoto Ichiro, Mike Van Emmerik, Quuxplusone, Intgr, Visor, Sharkface217, Bgwhite, YurikBot, Wavelength, Borgx, RobotE, RussBot, Fabartus, Splash, Piet Delport, Stephenb, Pseudomonas, Kimchi.sg, Dmason, JulesH, Mikeblas, Bota47, JLaTondre, Hide&Reason, Heavyrain2408, SmackBot, Princeatapi, Blue520, Trojo, Brick Thrower, Alksub, Apers0n, Betacommand, GwydionM, Anwar saadat, Keegan, Timneu22, Mcaruso, Tsca.bot, Tamfang, Berland, Cybercobra, Mwtoews, Masterdriverz, Kukini, Smremde, SashatoBot, Derek farn, John, 16@r, Slakr, Beetstra, Dreftymac, Courcelles, George100, Ahy1, Engelec, Wws, Neelix, Simeon, Kaldosh, Travelbird, Mrstonky, Skittleys, Strangelv, Christian75, Narayanese, Epbr123, Sagaciousuk, Trevyn, Escarbot, Thadius856, AntiVandalBot, AbstractClass, JAnDbot, JaK81600, Cameltrader, PhiLho, SiobhanHansa, Magioladitis, VoABot II, Ling.Nut, DAGwyn, David Eppstein, User A1, Squidonius, Gwern, Highegg, Themania, Patar knight, J.delanoy, Slogsweep, Darkspots, Jayden54, Mfb52, Funandtrvl, VolkovBot, TXiKiBoT, Anonymous Dissident, Don4of4, Amog, Redacteur, Nicvaroce, Kbrose, SieBot, Caltas, Garde, Tiptoety, Paolo.dL, Oxymoron83, Svick, Anchor Link Bot, Jlmerrill, ClueBot, LAX, Jackollie, The Thing That Should Not Be, Alksentrs, Rilak, Supertouch, R000t, Liempt, Excirial, Immortal Wowbagger, Bargomm, Thingg, Footballfan190, Johnuniq, SoxBot III, Awbell, Chris glenne, Staticshakedown, SilvonenBot, Henrry513414, Dsimic, Gaydudes, Btx40, EconoPhysicist, SamatBot, Zorrobot, Legobot, Luckas-bot, Yobot, Ptbotgourou, Fraggle81, Peter Flass, Tempodivalse, Obersachsebot, Xqbot, SPTWriter, FrescoBot, Citation bot 2, I dream of horses, HRoestBot, MastiBot, Jandalhandler, Laurențiu Dascălu, Dinamik-bot, TylerWilliamRoss, Merlinsorca, Jfmantis, The Utahraptor, EmausBot, Mfaheem007, Donner60, Ipsign, Ieee andy, EdoBot, Muzadded, Mikhail Ryazanov, ClueBot NG, Widr, Helpful Pixie Bot, Roger Wellington-Oguri, Wbm1058, 111008066it, Mark Arsten, Simba2331, Insidiae, ChrisGualtieri, A'bad group, Makecat-bot, Chetan chopade, Ginsuloft and Anonymous: 238
- **Record (computer science)** *Source:* http://en.wikipedia.org/wiki/Record%20(computer%20science)?oldid=645108207 *Contributors:* Pnm, TakuyaMurata, Minesweeper, Charles Matthews, Dcoetzee, Alexs, Furrykef, RedWolf, Tobias Bergemann, BenFrantzDale, Jorge Stolfi, Gdr, Burgundavia, Andreas Kaufmann, RossPatterson, Rich Farmbrough, CanisRufus, Nickj, Polluks, JeffTan, Suruena, RainbowOfLight, Arneth, RussBot, Jengelh, Mike92591, Luk, SmackBot, Gilliam, Nbarth, Clements, Cybercobra, Loadmaster, Cpiral, Raise exception, Brvman, Philip Trueman, Ilyaroz, Billinghurst, LaxCrosse007, Kbrose, Classicalecon, TheRedPenOfDoom, Razorflame, Hugo Herbelin, WikHead, Addbot, Debresser, Pcap, Peter Flass, AnomieBOT, Erik9bot, Dinamik-bot, Fiftytwo thirty, Gf uip, EmausBot, Leledumbo, ClueBot NG, KLBot2, Simonrodan, RscprinterBot, BattyBot and Anonymous: 35
- **Associative array** *Source:* http://en.wikipedia.org/wiki/Associative%20array?oldid=644061854 *Contributors:* Damian Yerrick, Robert Merkel, Fubar Obfusco, Maury Markowitz, Hirzel, B4hand, Paul Ebermann, Edward, Patrick, Michael Hardy, Shellreef, Graue, Minesweeper, Brianiac, Samuelsen, Bart Massey, Hashar, Dcoetzee, Dysprosia, Silvonen, Bevo, Robbot, Noldoaran, Fredrik, Altenmann, Wlievens, Catbar, Wikibot, Ruakh, EvanED, Jleedev, Tobias Bergemann, Ancheta Wis, Jpo, DavidCary, Mintleaf, Inter, Wolfkeeper, Jorge Stolfi, Macrakis, Pne, Neilc, Kusunose, Karol Langner, Bosmon, Int19h, Andreas Kaufmann, RevRagnarok, Ericamick, LeeHunter, PP Jewel, Kwamikagami, James b crocker, Spoon!, Bobo192, TommyG, Minghong, Alansohn, Mt, Krischik, Sligocki, Kdau, Tony Sidaway, RainbowOfLight, Forderud, TShilo12, Boothy443, Mindmatrix, RzR, Apokrif, Kglavin, Bluemoose, ObsidianOrder, Pfunk42, Yurik, Swmcd, Scandum, Koavf, Agorf, Jeff02, RexNL, Alvin-cs, Wavelength, Fdb, Maerk, Dggoldst, Cedar101, JLaTondre, Ffangs, TuukkaH, SmackBot, KnowledgeOfSelf, MeiStone, Mirzabah, TheDoctor10, Sam Pointon, Brianski, Hugo-cs, Jdh30, Zven, Cfallin, CheesyPuffs144,

Tom, Yintan, Ham Pastrami, Pi is 3.14159, Keilana, Flyer22, TechTony, Redmarkviolinist, Beejaye, Bughunter2, Mygerardromance, NHSKR, Hariva, Denisarona, Thorncrag, Startswithj, Scarlettwharton, ClueBot, Ravek, Justin W Smith, The Thing That Should Not Be, Raghaven, ImperfectlyInformed, Garyzx, Arakunem, Mild Bill Hiccup, Lindsayfgilmour, TobiasPersson, SchreiberBike, Dixie91, Nasty psycho, XLinkBot, Marc van Leeuwen, Avoided, G7mcluvn, Hook43113, Kurniasan, Wolkykim, Addbot, Anandvachhani, MrOllie, Freqsh0, Zorrobot, Jarble, Quantumobserver, Yobot, Fraggle81, KamikazeBot, Shadoninja, AnomieBOT, Jim1138, Materialscientist, Mwe 001, Citation bot, Quantran202, SPTWriter, Mtasic, Binaryedit, Miym, Etienne Lehnart, Sophus Bie, Apollo2991, Constructive editor, Afromayun, Prari, FrescoBot, Meshari alnaim, Ijsf, Mark Renier, Citation bot 1, I dream of horses, Apeculiaz, Patmorin, Carloseow, Vrenator, Zvn, BZRatfink, Arjitmalviya, Vhcomptech, WillNess, Minimac, Jfmantis, RjwilmsiBot, Agrammenos, EmausBot, KralSS, Simply.ari1201, Eniagrom, MaGa, Donner60, Carmichael, Peter Karlsen, 28bot, Sjoerddebruin, ClueBot NG, Jack Greenmaven, Millermk, Rezabot, Widr, MerlIwBot, Helpful Pixie Bot, HMSSolent, BG19bot, WikiPhoenix, Tango4567, Dekai Wu, Computersagar, SaurabhKB, Klilidiplomus, Singlaive, IgushevEdward, Electricmuffin11, TalhaIrfanKhan, Frosty, Smortypi, RossMMooney, Gauravxpress, Noyster, Suzrocksyu, Bryce archer, Melcous, Monkbot, Azx0987, Mahesh Dheravath, Vikas bhatnager, Aswincweety, RationalBlasphemist, Ishanalgorithm and Anonymous: 611

- **Binary search tree** *Source:* http://en.wikipedia.org/wiki/Binary%20search%20tree?oldid=648231058 *Contributors:* Damian Yerrick, Bryan Derksen, Taw, Mrwojo, Spiff, PhilipMW, Michael Hardy, Booyabazooka, Nixdorf, Ixfd64, Minesweeper, Darkwind, LittleDan, Glenn, BAxelrod, Timwi, MatrixFrog, Dcoetzee, Havardk, Dysprosia, Doradus, Maximus Rex, Phil Boswell, Fredrik, Postdlf, Bkell, Hadal, Tobias Bergemann, Enochlau, Awu, Giftlite, P0nc, Ezhiki, Maximaximax, Qleem, Karl-Henner, Qiq, Shen, Andreas Kaufmann, Jin, Grunt, Kate, Oskar Sigvardsson, D6, Ilana, Kulp, ZeroOne, Vdm, Func, LeonardoGregianin, Nicolasbock, HasharBot, Alansohn, Liao, RoySmith, Rudo.Thomas, Pion, Wtmitchell, Evil Monkey, 4c27f8e656bb34703d936fc59ede9a, Oleg Alexandrov, Mindmatrix, LOL, Oliphaunt, Ruud Koot, Trevor Andersen, GregorB, Mb1000, MrSomeone, Qwertyus, Nneonneo, Hathawayc, VKokielov, Ecb29, Mathbot, BananaLanguage, DevastatorIIC, Quuxplusone, Sketch-The-Fox, Butros, Banaticus, Roboto de Ajvol, YurikBot, Wavelength, Michael Slone, Hyad, Taejo, Gaius Cornelius, Oni Lukos, TheMandarin, Salrizvy, Moe Epsilon, BOT-Superzerocool, Googl, Regnaron, Abu adam, Chery, Cedar101, Jogers, LeonardoRob0t, Richardj311, WikiWizard, SmackBot, Bernard François, Gilliam, Ohnoitsjamie, Theone256, Oli Filth, Neurodivergent, DHN-bot, Alexsh, Garoth, Mweber, Allan McInnes, Calbaer, NitishP, Cybercobra, Hcethatsme, MegaHasher, Breno, Nux, Beetstra, Dicklyon, Hu12, Vocaro, Konnetikut, JForget, James pic, CRGreathouse, Ahy1, WeggeBot, Mikeputnam, TrainUnderwater, Jdm64, AntiVandalBot, Jirka6, Lanov, Huttarl, Eapache, JAnDbot, Anoopjohnson, Magioladitis, Abednigo, Allstarecho, Tomt22, Gwern, S3000, MartinBot, Anaxial, Leyo, Mike.lifeguard, Phishman3579, Skier Dude, Joshua Issac, Mgius, Kewlito, Danadocus, Vectorpaladin13, BotKung, One half 3544, Spadgos, MclareN212, Nerdgerl, Rdemar, Davekaminski, Rhanekom, SieBot, YonaBot, Casted, VVVBot, Ham Pastrami, Jerryobject, Swapsy, Djcollom, Svick, Anchor Link Bot, GRHooked, Loren.wilton, Xevior, ClueBot, ChandlerMapBot, Madhan virgo, Theta4, Shailen.sobhee, AgentSnoop, Onomou, XLinkBot, WikHead, Metalmax, MrOllie, Jdurham6, LinkFA-Bot, انا‌م, Matekm, Legobot, Luckas-bot, Yobot, Dimchord, AnomieBOT, The Parting Glass, Burakov, Ivan Kuckir, Tbvdm, LilHelpa, Capricorn42, SPTWriter, Doctordiehard, Wtarreau, Shmomuffin, Dzikasosna, Smallman12q, Kurapix, Adamuu, FrescoBot, 4get, Citation bot 1, Golle95, Aniskhan001, Frankrod44, Cochito, MastiBot, Thesevenseas, Sss41, Vromascanu, Shuri org, Rolpa, Jayaneethatj, Avermapub, MladenWiki, Konstantin Pest, Cyc115, WillNess, Nils schmidt hamburg, RjwilmsiBot, Ripchip Bot, X1024, Your Lord and Master, Nomen4Omen, Meng6, Wmayner, Tolly4bolly, Dan Wang, ClueBot NG, SteveAyre, Jms49, Frietjes, Ontariolot, Solsan88, Nakarumaka, BG19bot, AlanSherlock, Rafikamal, BPositive, RJK1984, Phc1, IgushevEdward, Hdanak, JingguoYao, Yaderbh, RachulAdmas, Frosty, Josell2, SchumacherTechnologies, Farazbhinder, Wulfskin, Embanner, Mtahmed, Jihlim, Kaidul, Cybdestroyer, Jabanabba, Gokayhuz, Mathgorges, Jianhui67, Super fish2, Ryuunoshounen, Dk1027, Azx0987, KH-1, Tshubham, HarshalVTripathi, ChaseKR, Filip Euler and Anonymous: 335

- **Hash table** *Source:* http://en.wikipedia.org/wiki/Hash%20table?oldid=648100229 *Contributors:* Damian Yerrick, AxelBoldt, Zundark, The Anome, BlckKnght, Sandos, Rgamble, LapoLuchini, AdamRetchless, Imran, Mrwojo, Frecklefoot, Michael Hardy, Nixdorf, Pnm, Axlrosen, TakuyaMurata, Ahoerstemeier, Nanshu, Dcoetzee, Dysprosia, Furrykef, Omegatron, Wernher, Bevo, Tjdw, Pakaran, Secretlondon, Robbot, Fredrik, Tomchiukc, R3m0t, Altenmann, Ashwin, UtherSRG, Miles, Giftlite, DavidCary, Wolfkeeper, BenFrantzDale, Everyking, Waltpohl, Jorge Stolfi, Wmahan, Neilc, Pgan002, CryptoDerk, Knutux, Bug, Sonjaaa, Teacup, Beland, Watcher, DNewhall, ReiniUrban, Sam Hocevar, Derek Parnell, Askewchan, Kogorman, Andreas Kaufmann, Kaustuv, Shuchung, T Long, Hydrox, Cfailde, Luqui, Wrp103, Antaeus Feldspar, Khalid, Raph Levien, JustinWick, CanisRufus, Shanes, Iron Wallaby, Krakhan, Bobo192, Davidgothberg, Larry V, Sleske, Helix84, Mdd, Varuna, Baka toroi, Anthony Appleyard, Sligocki, Drbreznjev, DSatz, Akuchling, TShilo12, Nuno Tavares, Woohookitty, LOL, Linguica, Paul Mackay, Davidfstr, GregorB, Meneth, Kbdank71, Tostie14, Rjwilmsi, Scandum, Koavf, Kinu, Filu, Nneonneo, FlaBot, Ecb29, Fragglet, Intgr, Fresheneesz, Antaeus FeIdspar, YurikBot, Wavelength, RobotE, Mongol, RussBot, Me and, CesarB's unpriviledged account, Lavenderbunny, Gustavb, Mipadi, Cryptoid, Mike.aizatsky, Gareth Jones, Piccolomomo, CecilWard, Nethgirb, Gadget850, Bota47, Sebleblanc, Deeday-UK, Sycomonkey, Ninly, Gulliveig, Th1rt3en, CWenger, JLaTondre, ASchmoo, Kungfuadam, SmackBot, Apanag, Obakeneko, PizzaMargherita, Alksub, Eskimbot, RobotJcb, C4chandu, Arpitm, Neurodivergent, EncMstr, Cribe, Deshraj, Tackline, Frap, Mayrel, Radagast83, Cybercobra, Decltype, HFuruseth, Rich.lewis, Esb, Acdx, MegaHasher, Doug Bell, Derek farn, IronGargoyle, Josephsieh, Peter Horn, Pagh, Saxton, Tawkerbot2, Ouishoebean, CRGreathouse, Ahy1, MaxEnt, Seizethedave, Cgma, Not-just-yeti, Thermon, OtterSmith, Ajo Mama, Stannered, AntiVandalBot, Hosamaly, Thailyn, Pixor, JAnDbot, MER-C, Epeefleche, Dmbstudio, SiobhanHansa, Wikilolo, Bongwarrior, QrczakMK, Josephskeller, Tedickey, Schwarzbichler, Cic, Allstarecho, David Eppstein, Oravec, Gwern, Magnus Bakken, Glrx, Narendrak, Tikiwont, Mike.lifeguard, Luxem, NewEnglandYankee, Cobi, Cometstyles, Winecellar, VolkovBot, Simulationelson, Floodyberry, Anurmi, BotKung, Collin Stocks, JimJJewett, Nightkhaos, Spinningspark, Abatishchev, Helios2k6, Kehrbykid, Kbrose, PeterCanthropus, Gerakibot, Triwbe, Digwuren, Svick, JLBot, ObfuscatePenguin, ClueBot, Justin W Smith, ImperfectlyInformed, Adrianwn, Mild Bill Hiccup, Niceguyedc, JJuran, Groxx, Berean Hunter, Eddof13, Johnuniq, Arlolra, XLinkBot, Hetori, Pichpich, Paulsheer, TheTraveler3, MystBot, Karuthedam, Wolkykim, Addbot, Gremel123, CanadianLinuxUser, MrOllie, Numbo3-bot, Om Sao, Zorrobot, Jarble, Frehley, Legobot, Luckas-bot, Yobot, Denispir, KamikazeBot, Dmcomer, AnomieBOT, Erel Segal, Jim1138, Sz-iwbot, Citation bot, ArthurBot, Baliame, Drilnoth, Arbalest Mike, Ched, Shadowjams, Kracekumar, FrescoBot, W Nowicki, X7q, Sae1962, Citation bot 1, Velociostrich, Simonsarris, Iekpo, Trappist the monk, SchreyP, Grapesoda22, Patmorin, Cutelyaware, JeepdaySock, Shafigoldwasser, Kastchei, DuineSidhe, EmausBot, Super48paul, Ibbn, DanielWaterworth, Mousehousemd, ZéroBot, Purplie, Ticklemepink42, Paul Kube, Demonkoryu, Donner60, Carmichael, Phearthecel, Aberdeen01, Teapeat, Rememberway, ClueBot NG, AznBurger, Incompetence, Rawafmail, Cntras, Rezabot, Jk2q3jrklse, Helpful Pixie Bot, Jan Spousta, MusikAnimal, SanAnMan, Pbruneau, AdventurousSquirrel, Triston J. Taylor, CitationCleanerBot, Happyuk, FeralOink, Spacemanaki, Aloksukhwani, Emimull, Deveedutta, Shmageggy, IgushevEdward, AlecTaylor, Mcom320, Razibot, Djszapi, QuantifiedElf, Chip Wildon Forster, Tmferrara, Tuketu7, Monkbot, Iokevins, Oleaster and Anonymous: 425

- **Hash function** *Source:* http://en.wikipedia.org/wiki/Hash%20function?oldid=648837123 *Contributors:* Damian Yerrick, Derek Ross, Taw, BlckKnght, PierreAbbat, Miguel, Imran, David spector, Dwheeler, Hfastedge, Michael Hardy, EddEdmondson, Ixfd64, Mdebets,

- **Bubble sort** *Source:* http://en.wikipedia.org/wiki/Bubble%20sort?oldid=648313032 *Contributors:*
- **Insertion sort** *Source:* http://en.wikipedia.org/wiki/Insertion%20sort?oldid=648822181 *Contributors:*

Simpsons contributor, Ww2censor, Nillerdk, Zvar, Allan McInnes, Oxaric, ThomasMueller, Egli, Ohconfucius, Nmnogueira, John, XreDuex, DavidGrayson, MTSbot, Pipedreambomb, CapitalR, Tawkerbot2, Ahy1, Ivan Pozdeev, VTBassMatt, Seizethedave, Mblumber, MC10, Blaisorblade, Jonas Kölker, Thijs!bot, Kiwi137, Frozenport, I do not exist, Lidden, Hardmath, Ninjakannon, JAnDbot, SiobhanHansa, KBKarma, Pcp071098, Yandman, Tedickey, David Eppstein, MartinBot, Tamer ih, Glrx, Player 03, Ceros, Pharaoh of the Wizards, Gmazeroff, Magicbronson, Coshoi, Buddhikaeport, Lawlzlawlz, Quentonamos, TXiKiBoT, JhsBot, Don4of4, Broadbot, JosephMDecock, Jesin, Billinghurst, Rhanekom, SieBot, Zaradaqaw, Tiddly Tom, Rlneumiller, Flyer22, Ponggr, Steven Zhang, Svick, Neel basu, Sfan00 IMG, ClueBot, DFRussia, Dardasavta, Pakaraki, Garyzx, Okosenkov, Trivialist, Billylikeswikis, Jpmelos, BOTarate, Grofles, XLinkBot, Keynell, Nasradu8, Oğuz Ergin, Baby123412, SilvonenBot, Alexius08, Jordanbray, Mike1341, Addbot, Kangaroosrule, Nuxnut, Alex.mccarthy, Merendoglu, MrOllie, Sapeur, Mike1242, Vasiľ, Jarble, Mess, AnomieBOT, Killiondude, Emdtechnology, RandomAct, Materialscientist, Faizbash, Tryptophan4, XZeroBot, Amaury, Minchenko Michael, HJ Mitchell, Louperibot, MorganGreen, Jonesey95, Barras, Trappist the monk, Freakingtips, Jarajapu, Dinamik-bot, Acidburnjd, Brambleclawx, Vpshastry, H.ehsaan, Jfmantis, 1qaz-pl, Autumnalmonk, RA0808, K6ka, Josve05a, Nordald, Goutamrocks, Aleks80, E.ruzi, Pdvyas, BioPupil, ChuispastonBot, Swfung8, Klrste, Josh Kehn, Baltar, Gaius, ClueBot NG, This lousy T-shirt, Strcat, Widr, Helpful Pixie Bot, Villaone56, Rax2095, Marvon7Newby, Pratyya Ghosh, Trunks175, ChrisGualtieri, SmokingCrop, Will Faught, Seanhalle, Rajarammallya, Amarpalsinghkapoor, EAtsala, Love debian, Dastoger Bashar, Davinci8x8, Monkbot, Faham123, Kushal khadka, Aswincweety, Dalton Quinn, TranquilHope, Xakari, Ankitsachan333 and Anonymous: 344

- **Merge sort** *Source:* http://en.wikipedia.org/wiki/Merge%20sort?oldid=648626827 *Contributors:* Damian Yerrick, Lee Daniel Crocker, Mav, The Anome, JohnOwens, Booyabazooka, Pnm, Shellreef, Dcljr, TakuyaMurata, Eric119, Minesweeper, Rl, Timwi, Dcoetzee, Ww, Kbk, Daniel Quinlan, Jogloran, Joshk, Furrykef, Thue, Garo, Robbot, Fredrik, Tomchiukc, Romanm, JerryFriedman, Jleedev, Tobias Bergemann, Decrypt3, Giftlite, Frencheigh, T0m, JF Bastien, Neilc, Chowbok, Pgan002, Knutux, Onco p53, Wilagobler, Nickls, Sam nead, Naku, Oskar Sigvardsson, Rfl, Discospinster, Rich Farmbrough, Guanabot, Rspeer, ArnoldReinhold, Sperling, Kenb215, Danakil, MisterSheik, Adambro, Jpl, Nyenyec, BrokenSegue, Mikewebkist, Haham hanuka, Ossi, Jumbuck, Alansohn, Sligocki, Swift, Caesura, ReyBrujo, Mattjohnson, Dr. Gonzo, WojciechSwiderski, Forderud, Kenyon, Bobrayner, Daniel Geisler, Soultaco, Mntlchaos, Nuno Tavares, Merlinme, Jacobolus, Ruud Koot, Vorn, Fred J, GregorB, Tommunist, Palica, Pfalstad, Wbeek, Qwertyus, GrundyCamellia, Grammarbot, Rjwilmsi, Vexis, Bayard, Bernard van der Wees, Bubba73, Nguyen Thanh Quang, SLi, StuartBrady, FlaBot, Ewlyahoocom, DevastatorIIC, Comocomocomocomo, Intgr, CJLL Wright, Chobot, DVdm, Garas, Cactus.man, Dbagnall, Hyad, Jengelh, Zhaladshar, Mipadi, Fashnek, Mikeblas, Black Falcon, Lt-wiki-bot, Abu adam, Cedar101, GraemeL, Donhalcon, Thijswijs, SmackBot, Damonkohler, Kalebdf, Renku, Delldot, Scott Paeth, Gilliam, Ohnoitsjamie, Andy M. Wang, Sirex98, Oli Filth, Silly rabbit, Nbarth, DHN-bot, Nixeagle, Avb, EdSchouten, Allan McInnes, Easwarno1, HoserHead, Zophar1, Cybercobra, Duckbill, OranL, Oxaric, ThomasMueller, Itmozart, Andrei Stroe, Nmnogueira, Cuzelac, Sir Nicholas de Mimsy-Porpington, Apoorbo, Noah Salzman, Dammit, MTSbot, Negrulio, PavelY, UncleDouggie, GDallimore, Amiodusz, Jafet, Xueshengyao, CRGreathouse, Ahy1, Bakanov, Mblumber, Thijs!bot, Towopedia, Radiozilla, Ablonus, AntiVandalBot, WinBot, VineetKumar, Jirka6, Wootery, Martinkunev, SiobhanHansa, Antientropic, CobaltBlue, Dima1, Eleschinski2000, Cic, Destynova, Dbingham, Pkrecker, Maju wiki, Glrx, Ceros, J.delanoy, Acalamari, Faridani, Sanjay742, ScottBurson, Vanished user 39948282, Alain Amiouni, TXiKiBoT, Daztekk, Amahdy, Liko81, JhsBot, Sychen, SashaMarievskaya, Dmcq, Rhanekom, SieBot, Coffee, Artagnon, Fizo86, Ctxppc, Garrettw87, Hariva, Xhackeranywhere, Novas0x2a, ClueBot, DFRussia, PipepBot, Fyyer, Garyzx, N26ankur, DragonBot, Excirial, Bender2k14, TobiasPersson, XLinkBot, EAspenwood, Oğuz Ergin, C. A. Russell, Bill wang1234, Hoof1341, Addbot, Fyrael, Miskaton, Rcgldr, MrOllie, Download, Worch, Lightbot, Ninjatummen, Geeker87, Yobot, Tohd8BohaithuGh1, Fraggle81, Timeroot, NotARusski, Naderra, AnomieBOT, Erel Segal, Kingpin13, Citation bot, ArthurBot, JimVC3, Octotron, Locobot, A3ng3l, Shadowjams, Captain Fortran, Kracekumar, FrescoBot, X7q, Mctpyt, Ahmadsh, Mecej4, Citation bot 1, Base698, Mutinus, RedBot, Tuttu4u, Michele bon, Schorzman78, Deanonwiki, Patmorin, TBloemink, RjwilmsiBot, Wintonian, EmausBot, Immunize, Dewritech, Shashwat2691, Bor75, Yodamgod, Sven nestle2, Mlhetland, Duvavic1, Eserra, Swfung8, Klrste, Josh Kehn, ClueBot NG, Jheld88, Strcat, Hofmic, Widr, Jk2q3jrklse, Riemann'sZeta, Ravishanker.bit, EmadIV, Villaone56, Pratyya Ghosh, Colemanfoley, Deepakabhyankar, Sam.ldite, Maeln, Mark viking, Apoorva181192, SigbertW, Alcat33, Kaushik0402, Monkbot, Harshitm26, Navstar55, Ansuman04, Merocastle, Engheta, Dalton Quinn, Beijingxilu, Swrobinson26 and Anonymous: 399
- **Quicksort** *Source:* http://en.wikipedia.org/wiki/Quicksort?oldid=648926577 *Contributors:* AxelBoldt, The Anome, Arvindn, Hannes Hirzel, Hfastedge, NTF, Spiff, Booyabazooka, Modster, Nixdorf, Shellreef, Liftarn, Meekohi, Tompagenet, Chinju, Zeno Gantner, Cyde, TakuyaMurata, Eric119, CesarB, Notheruser, Kingturtle, Randywombat, Kragen, LittleDan, Michael Shields, Evercat, Timwi, Dcoetzee, Ww, Dysprosia, Kbk, Doradus, Wik, Saltine, Bartosz, Populus, McKay, Lord Emsworth, Raul654, Pakaran, Jusjih, Jamesday, MH, Murray Langton, Fredrik, R3m0t, RedWolf, Altenmann, Romanm, Kuszi, Sverdrup, Rursus, Bkell, Intangir, Vikreykja, Carlj7, Crowst, Pifactorial, Tobias Bergemann, Connelly, Decrypt3, Centrx, Giftlite, Fennec, Jao, Juliand, Dinomite, Ferkelparade, Dissident, Karnan, Eequor, Chameleon, Shashmik11, Neilc, Stevietheman, Pgan002, Knutux, OverlordQ, Phe, Quarl, Panzi, Sam Hocevar, Gscshoyru, Aerion, Aroben, Bluear, Mike Rosoft, Oskar Sigvardsson, Discospinster, Chrischan, Rsanchezsaez, Rspeer, Too Old, Xezbeth, AlanBarrett, ESkog, ZeroOne, AdmN, Danakil, Diego UFCG, Shanes, Spearhead, Diomidis Spinellis, Aaronbrick, Nwerneck, Richss, Func, Sandare, BrokenSegue, Dila, Mh26, Jadrian, Benbread, 4v4l0n42, HasharBot, Alansohn, Bputman, Kanie, Jason Davies, PatrickFisher, Comrade009, Sligocki, Swift, Snowolf, Furrfu, Fx2, Pgimeno, Dan100, Forderud, Kbolino, Postrach, Simetrical, LOL, DanBishop, Mihai Damian, Ruud Koot, RolandH, Jeff3000, Hdante, Bluemoose, Sf222, GregorB, Fxer, Palica, Pmcjones, Pete142, Graham87, Magister Mathematicae, Qwertyus, Sjakkalle, Rjwilmsi, Bayard, Pako, Nneonneo, Bubba73, Mathbot, Mathiastck, Quuxplusone, Fresheneesz, Compotatoj, CiaPan, CJLL Wright, NevilleDNZ, Chobot, Daekharel, CarlosHoyos, YurikBot, Dmccarty, Eleassar, NawlinWiki, Nothing1212, Jpbowen, Mikeblas, LodeRunner, Dbfirs, Berlinguyinca, Adicarlo, Cmcfarland, Black Falcon, Lars Trebing, SamuelRiv, Cdiggins, Abu adam, Cedar101, Donhalcon, Wisgary, Composingliger, Darrel francis, Dancraggs, SmackBot, NickyMcLean, Roger Hui, KnowledgeOfSelf, Scott Paeth, Btwied, UrsaFoot, DomQ, Senfo, Ohnoitsjamie, JMiall, Bluebot, Oli Filth, Jazmatician, Timneu22, Nbarth, DHN-bot, Tavianator, Nick Levine, Rrburke, Allan McInnes, Runefurb, Cybercobra, Jóna Þórunn, Andrei Stroe, Ugur Basak Bot, Nmnogueira, Alexander256, N Vale, Tanadeau, Feradz, Sir Nicholas de Mimsy-Porpington, StephenDow, Domokato, Beetstra, Norm mit, Skapur, Elharo, Arto B, Sabb0ur, Owen214, Ylloh, Ahy1, Ipeirotis, Jokes Free4Me, NickW557, Croc hunter, Mblumber, MC10, Blaisorblade, Wfaxon, Chrislk02, ColdShine, Thijs!bot, Wikid77, John Comeau, HalHal, Gpollock, AntiVandalBot, Widefox, Seaphoto, Михајло Анђелковић, Arcturus4669, Lordmetroid, Nik 0 0, Gdo01, PhilKnight, SiobhanHansa, Magioladitis, VoABot II, Necklace, Yandman, Rami R, CobaltBlue, Abednigo, Stdazi, User A1, Chrisdone, Maju wiki, Gwern, Shentino, Doccolinni, Glrx, Themania, Narendrak, Francis Tyers, Faridani, Indeed123, Daniel5Ko, Elcielo917, Dr.RMills, KylieTastic, Entropy, Neohaven, VolkovBot, Alain Amiouni, Lhuang3, Soliloquial, Philip Trueman, PGSONIC, Jnoring, Calwiki, Nxavar, Ocolon, Sychen, Perpetuo2, David Bunde, Dmcq, Helios2k6, A Meteorite, Rhanekom, SieBot, Udirock, Theclawen, OKBot, Svick, ClueBot, Vladkornea, Czarkoff, DragonBot, Tim32, KittyKAY4, Versus22, DumZiBoT, BarretB, XLinkBot, Stickee, Yulin11, Snasna, C. lorenz, Oğuz Ergin, Dsimic, Addbot, Bertrc, MrOllie, Download, Jevy1234, Glane23, Worch, TheCois, Rrufai, Tide rolls, Ninjatummen, Kapildalwani, Ettrig, Fryed-peach, Boulevardier, Obscuranym, KamikazeBot, AnomieBOT, 1exec1, Quantumelixir, Materialscientist, Bhaveshnande,

ThomasTomMueller, LilHelpa, SPTWriter, Scebert, Znupi, RibotBOT, Nearffxx, Brayan Jaimes, Aaditya 7, SKATEMANKING, Noisylo65, FrescoBot, C7protal, HJ Mitchell, Mecej4, Biker Biker, I dream of horses, Frankrod44, Alcauchy, Sir Edward V, Darren Strash, Ungzd, Pushingbits, Patmorin, Weedwhacker128, Fastilysock, Rony fhebrian, Jfmantis, Haker4o, Dmwpowers, JustAHappyCamper, John lindgren, John of Reading, Zarrandreas, Dewritech, Mswen, Bollyjeff, Tolly4bolly, NovellGuy, Empaisley, ChuispastonBot, Rdargent, Bastetswarrior, Solde9, Swfung8, Cumthsc, Integr8e, Josh Kehn, Petrb, Timkaler, ClueBot NG, Jack Greenmaven, Vacation9, Srchulo, Thiagohirai, Hamza1886, Larosek, Bengski68, Alexgotsis, Frze, Frantisek.jandos, Snow Blizzard, Jalestro, Jamesmontalvo3, JingguoYao, Mark L MacDonald, Lugia2453, Naveenkodali123, François Robere, Nonsanity, Shivajivarma, Akaloger, Krishna553, Lyuflamb, Apoorva181192, Phanikumarv, Swifty slow, Anselmotalotta, Harshitm26, Minhaskamal, 1706abhi, Victor veitch, Fezzerof, Mansur56, Abouelsaoud, Serbanalex2202 and Anonymous: 630

16.9.2 Images

- **File:8bit-dynamiclist.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1d/8bit-dynamiclist.gif *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* Seahen
- **File:Array_of_array_storage.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/0/01/Array_of_array_storage.svg *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:BinaryTreeRotations.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/43/BinaryTreeRotations.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Josell7
- **File:Binary_search_tree.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/da/Binary_search_tree.svg *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Binary_search_tree_delete.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/46/Binary_search_tree_delete.svg *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Binary_tree.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/f7/Binary_tree.svg *License:* Public domain *Contributors:* Own work *Original artist:* Derrick Coetzee
- **File:Bubble-sort-example-300px.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/c8/Bubble-sort-example-300px.gif *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Swfung8
- **File:Bubble_sort_animation.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/3/37/Bubble_sort_animation.gif *License:* CC BY-SA 2.5 *Contributors:* Own work by uploader using Matlab *Original artist:* The original uploader was Nmnogueira at English Wikipedia
- **File:Bubblesort-edited-color.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/8/83/Bubblesort-edited-color.svg *License:* CC0 *Contributors:* Own work *Original artist:* Pmdumuid
- **File:CPT-LinkedLists-addingnode.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/4b/CPT-LinkedLists-addingnode.svg *License:* Public domain *Contributors:*
- Singly_linked_list_insert_after.png *Original artist:* Singly_linked_list_insert_after.png: Derrick Coetzee
- **File:CPT-LinkedLists-deletingnode.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/d4/CPT-LinkedLists-deletingnode.svg *License:* Public domain *Contributors:*
- Singly_linked_list_delete_after.png *Original artist:* Singly_linked_list_delete_after.png: Derrick Coetzee
- **File:Circularly-linked-list.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/df/Circularly-linked-list.svg *License:* Public domain *Contributors:* Own work *Original artist:* Lasindi
- **File:Commons-logo.svg** *Source:* http://upload.wikimedia.org/wikipedia/en/4/4a/Commons-logo.svg *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Directed_Graph_Edge.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/d7/Directed_Graph_Edge.svg *License:* Public domain *Contributors:* Own work *Original artist:* Johannes Rössel (talk)
- **File:Directed_graph,_cyclic.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1c/Directed_graph%2C_cyclic.svg *License:* Public domain *Contributors:* Transfered from de.wikipedia Transfer was stated to be made by User:Ddxc. *Original artist:* Original uploader was David W. at de.wikipedia
- **File:Directed_graph,_disjoint.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/7/78/Directed_graph%2C_disjoint.svg *License:* Public domain *Contributors:* Transfered from de.wikipedia Transfer was stated to be made by User:Ddxc. *Original artist:* Original uploader was David W. at de.wikipedia
- **File:Directed_graph_with_branching.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/0/0d/Directed_graph_with_branching.jpg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Milgesch
- **File:Doubly-linked-list.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/5/5e/Doubly-linked-list.svg *License:* Public domain *Contributors:* Own work *Original artist:* Lasindi
- **File:Graph_single_node.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/e/e2/Graph_single_node.svg *License:* Public domain *Contributors:* Transfered from de.wikipedia Transfer was stated to be made by User:Ddxc. *Original artist:* Original uploader was David W. at de.wikipedia
- **File:Hash_table_3_1_1_0_1_0_0_SP.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/7/7d/Hash_table_3_1_1_0_1_0_0_SP.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash_table_4_1_0_0_0_0_0_LL.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/2/2e/Hash_table_4_1_0_0_0_0_0_LL.svg *License:* Public domain *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash_table_4_1_1_0_0_0_0_LL.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/7/71/Hash_table_4_1_1_0_0_0_0_LL.svg *License:* Public domain *Contributors:* Own work *Original artist:* Jorge Stolfi

- **File:Hash_table_4_1_1_0_0_1_0_LL.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/5/58/Hash_table_4_1_1_0_0_1_0_LL.svg *License:* Public domain *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash_table_5_0_1_1_1_1_0_LL.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/5/5a/Hash_table_5_0_1_1_1_1_0_LL.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash_table_5_0_1_1_1_1_0_SP.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/b/bf/Hash_table_5_0_1_1_1_1_0_SP.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash_table_5_0_1_1_1_1_1_LL.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/d0/Hash_table_5_0_1_1_1_1_1_LL.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash_table_average_insertion_time.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1c/Hash_table_average_insertion_time.png *License:* Public domain *Contributors: Author's Own Work. Original artist:* Derrick Coetzee (User:Dcoetzee)
- **File:IBM_729_Tape_Drives.nasa.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/6/6c/IBM_729_Tape_Drives.nasa.jpg *License:* Public domain *Contributors:* http://en.wikipedia.org/wiki/File:IBM_729_Tape_Drives.nasa.jpg Originally uploaded 14:32, 22 August 2004 (UTC) by ArnoldReinhold (talk • contribs) to en:wiki. *Original artist:* NASA
- **File:Insertion-sort-example-300px.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/0/0f/Insertion-sort-example-300px.gif *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Swfung8
- **File:Insertion_sort.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/42/Insertion_sort.gif *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Simpsons contributor
- **File:Insertionsort-after.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/d9/Insertionsort-after.png *License:* Public domain *Contributors:* Transferred from en.wikipedia; transferred to Commons by User:RHaworth using CommonsHelper. *Original artist:* Original uploader was Dcoetzee at en.wikipedia
- **File:Insertionsort-before.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/3/32/Insertionsort-before.png *License:* Public domain *Contributors:* Transferred from en.wikipedia; transferred to Commons by User:RHaworth using CommonsHelper. *Original artist:* Original uploader was Dcoetzee at en.wikipedia
- **File:Insertionsort-edited.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/7/7e/Insertionsort-edited.png *License:* Public domain *Contributors:* I (crashmatrix (talk | contribs)) created this work entirely by myself. *Original artist:* crashmatrix (talk | contribs). Original uploader was Crashmatrix at en.wikipedia
- **File:Internet_map_1024.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/d2/Internet_map_1024.jpg *License:* CC BY 2.5 *Contributors:* Originally from the English Wikipedia; description page is/was here. *Original artist:* The Opte Project
- **File:Merge-sort-example-300px.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/cc/Merge-sort-example-300px.gif *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Swfung8
- **File:Merge_sort_algorithm_diagram.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/e/e6/Merge_sort_algorithm_diagram.svg *License:* Public domain *Contributors:* Transferred from en.wikipedia; transferred to Commons by User:Eric Bauman using CommonsHelper. *Original artist:* Original uploader was VineetKumar at en.wikipedia
- **File:Merge_sort_animation2.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/c5/Merge_sort_animation2.gif *License:* CC BY-SA 2.5 *Contributors:* en.wikipedia *Original artist:* CobaltBlue
- **File:Office-book.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/a/a8/Office-book.svg *License:* Public domain *Contributors:* This and myself. *Original artist:* Chris Down/Tango project
- **File:Partition_example.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/8/84/Partition_example.svg *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Question_book-new.svg** *Source:* http://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg *License:* Cc-by-sa-3.0 *Contributors:*

  Created from scratch in Adobe Illustrator. Based on Image:Question book.png created by User:Equazcion *Original artist:* Tkgd2007
- **File:Quicksort-diagram.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/a/af/Quicksort-diagram.svg *License:* Public domain *Contributors:* Own work *Original artist:* Znupi
- **File:Singly-linked-list.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/6/6d/Singly-linked-list.svg *License:* Public domain *Contributors:* Own work *Original artist:* Lasindi
- **File:Sorting_quicksort_anim.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/6/6a/Sorting_quicksort_anim.gif *License:* CC-BY-SA-3.0 *Contributors:* originally upload on the English Wikipedia *Original artist:* Wikipedia:en:User:RolandH
- **File:Text_document_with_red_question_mark.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/a/a4/Text_document_with_red_question_mark.svg *License:* Public domain *Contributors:* Created by bdesham with Inkscape; based upon Text-x-generic.svg from the Tango project. *Original artist:* Benjamin D. Esham (bdesham)
- **File:Wiki_letter_w_cropped.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1c/Wiki_letter_w_cropped.svg *License:* CC-BY-SA-3.0 *Contributors:*
- Wiki_letter_w.svg *Original artist:* Wiki_letter_w.svg: Jarkko Piiroinen
- **File:Wikibooks-logo-en-noslogan.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/df/Wikibooks-logo-en-noslogan.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Bastique, User:Ramac et al.
- **File:Wikibooks-logo.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/fa/Wikibooks-logo.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Bastique, User:Ramac et al.
- **File:Wikiquote-logo.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/fa/Wikiquote-logo.svg *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Wikisource-logo.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/4c/Wikisource-logo.svg *License:* CC BY-SA 3.0 *Contributors:* Rei-artur *Original artist:* Nicholas Moreau

- **File:Wikiversity-logo-Snorky.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1b/Wikiversity-logo-en.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Snorky
- **File:Wikiversity-logo.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/9/91/Wikiversity-logo.svg *License:* CC BY-SA 3.0 *Contributors:* Snorky (optimized and cleaned up by verdy_p) *Original artist:* Snorky (optimized and cleaned up by verdy_p)
- **File:Wiktionary-logo-en.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/f8/Wiktionary-logo-en.svg *License:* Public domain *Contributors:* Vector version of Image:Wiktionary-logo-en.png. *Original artist:* Vectorized by Fvasconcellos (talk · contribs), based on original logo tossed together by Brion Vibber

16.9.3 Content license

- Creative Commons Attribution-Share Alike 3.0