Toggle the table of contents

Binary search tree

In computer science, a **binary search tree** (**BST**), also called an **ordered** or **sorted binary tree**, is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. The time complexity of operations on the binary search tree is linear with respect to the height of the tree.

Binary search tree		
Type	tree	
Invented	1960	
Invented by	P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard	
Time complexity in **big O notation**		
Algorithm	**Average**	**Worst case**
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Binary search trees allow binary search for fast lookup, addition, and removal of data items. Since the nodes in a BST are laid out so that each comparison skips about half of the remaining tree, the lookup performance is proportional to that of binary logarithm. BSTs were devised in the 1960s for the problem of efficient storage of labeled data and are attributed to Conway Berners-Lee and David Wheeler.

The performance of a binary search tree is dependent on the order of insertion of the nodes into the tree since arbitrary insertions may lead to degeneracy; several variations of the binary search tree can be built with guaranteed worst-case performance. The basic operations include: search, traversal, insert and delete. BSTs with guaranteed worst-case complexities perform better than an unsorted array, which would require linear search time.

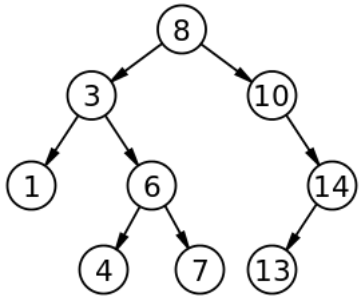


Fig. 1: A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

The complexity analysis of BST shows that, on average, the insert, delete and search takes $O(\log n)$ for n nodes. In the worst case, they degrade to that of a singly linked list: $O(n)$. To address the boundless increase of the tree height with arbitrary insertions and deletions, self-balancing variants of BSTs are introduced to bound the worst lookup complexity to that of the binary logarithm. AVL trees were the first self-balancing binary search trees, invented in 1962 by Georgy Adelson-Velsky and Evgenii Landis.

Binary search trees can be used to implement abstract data types such as dynamic sets, lookup tables and priority queues, and used in sorting algorithms such as tree sort.

History

The binary search tree algorithm was discovered independently by several researchers, including P.F. Windley, Andrew Donald Booth, Andrew Colin, Thomas N. Hibbard.[1][2] The algorithm is attributed to Conway Berners-Lee and David Wheeler, who used it for storing labeled data in magnetic tapes in 1960.[3] One of the earliest and popular binary search tree algorithm is that of Hibbard.[1]

WIKIPEDIA
The Free Encyclopedia

Toggle the table of contents

Hash table

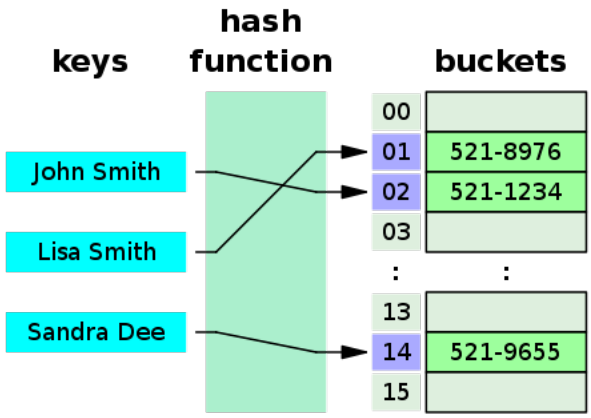
Hash table		
Type	Unordered associative array	
Invented	1953	
Time complexity in big O notation		
Algorithm	**Average**	**Worst case**
Space	Θ(*n*)[1]	O(*n*)
Search	Θ(1)	O(*n*)
Insert	Θ(1)	O(*n*)
Delete	Θ(1)	O(*n*)

In computing, a **hash table**, also known as a **hash map**, is a data structure that implements an associative array, also called a dictionary, which is an abstract data type that maps keys to values.[2] A hash table uses a hash function to compute an *index*, also called a *hash code*, into an array of *buckets* or *slots*, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash *collisions* where the hash function generates the same index for more than one key. Such collisions are typically accommodated in some way.

In a well-dimensioned hash table, the average time complexity 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 constant average cost per operation.[3][4][5]

Hashing is an example of a space-time tradeoff. If memory is infinite, the entire key can be used directly as an index to locate its value with a single memory access. On the other hand, if infinite time is available, values can be stored without regard for their keys, and a binary search or linear search can be used to retrieve the element.[6]:458



A small phone book as a hash table

In many situations, hash tables turn out to be on average 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.

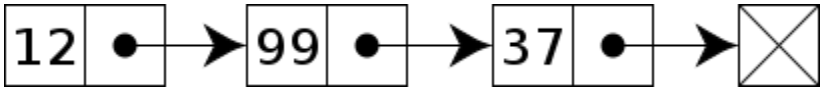
History

The idea of hashing arose independently in different places. In January 1953, Hans Peter Luhn wrote an internal IBM memorandum that used hashing with chaining. Open addressing was later proposed by A. D. Linh building on Luhn's paper.[7]:15 Around the same time, Gene Amdahl, Elaine M. McGraw, Nathaniel Rochester, and Arthur Samuel of IBM Research implemented hashing for the IBM 701 assembler.[8]:124

WIKIPEDIA
The Free Encyclopedia

Toggle the table of contents

Linked list



A linked list is a sequence of nodes that contain two fields: data (an integer value here as an example) and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

In computer science, a **linked list** is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence. In its most basic form, each node contains data, and a reference (in other words, a *link*) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. More complex variants add additional links, allowing more efficient insertion or removal of nodes at arbitrary positions. A drawback of linked lists is that data access time is a linear function of the number of nodes for each linked list (I.e., the access time linearly increases as nodes are added to a linked list.) because nodes are serially linked so a node needs to be accessed first to access the next node (so difficult to pipeline). Faster access, such as random access, is not feasible. Arrays have better cache locality compared to linked lists.

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, stacks, queues, associative arrays, and S-expressions, though it is not uncommon to implement those data structures directly without using a linked list as the basis.

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

On the other hand, since simple linked lists by themselves do not allow random access to the data or any form of efficient indexing, many basic operations—such as obtaining the last node of the list, finding a node that contains a given datum, or locating the place where a new node should be inserted—may require iterating through most or all of the list elements.

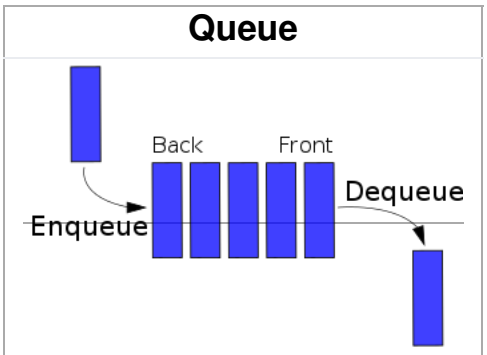
History

Linked lists were developed in 1955–1956, by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation and Carnegie Mellon University as the primary data structure for their Information Processing Language (IPL). 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

WIKIPEDIA
The Free Encyclopedia

Toggle the table of contents

Queue (abstract data type)



Representation of a FIFO (first in, first out) queue

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(1)
Delete	O(1)	O(1)

In computer science, a **queue** is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence. By convention, the end of the sequence at which elements are added is called the back, tail, or rear of the queue, and the end at which elements are removed is called the head or front of the queue, analogously to the words used when people line up to wait for goods or services.

The operation of adding an element to the rear of the queue is known as *enqueue*, and the operation of removing an element from the front is known as *dequeue*. Other operations may also be allowed, often including a *peek* or *front* operation that returns the value of the next element to be dequeued without dequeuing it.

The operations of a queue make it a first-in-first-out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. A queue is an example of a linear data structure, or more abstractly a sequential collection. Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.

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

Queue implementation

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

Fixed-length arrays are limited in capacity, but it is not true that items need to be copied towards the head of the queue. The simple trick of turning the array into a closed circle and letting the head and tail drift around endlessly in that circle makes it unnecessary to ever move items stored in the array. If n is the size of the array, then computing indices modulo n will turn the array into a circle. This is still the conceptually simplest way to construct a queue in a high-level language, but it does admittedly slow things down a little, because the array indices must be compared to zero and the array size, which is comparable to the time taken to check whether an array index is out of bounds, which some languages do, but this will certainly be the method of

Red–black tree

In computer science, a **red–black tree** is a specialised binary search tree data structure noted for fast storage and retrieval of ordered information, and a guarantee that operations will complete within a known time. Compared to other self-balancing binary search trees, the nodes in a red-black tree hold an extra bit called "color" representing "red" and "black" which is used when re-organising the tree to ensure that it is always approximately balanced. [3]

When the tree is modified, the new tree is rearranged and "repainted" to restore the coloring properties that constrain how unbalanced the tree can become in the worst case. The properties are designed such that this rearranging and recoloring can be performed efficiently.

The (re-)balancing is not perfect, but guarantees searching in Big O time of $O(\log n)$, where n is the number of entries (or keys) in the tree. The insert and delete operations, along with the tree rearrangement and recoloring, are also performed in $O(\log n)$ time.[4]

Red–black tree		
Type	Tree	
Invented	1978	
Invented by	Leonidas J. Guibas and Robert Sedgewick	
Complexities in big O notation		
Space complexity		
Space	$O(n)$	
Time complexity		
Function	**Amortized**	**Worst Case**
Search	$O(\log n)$[1]	$O(\log n)$[1]
Insert	$O(1)$[2]	$O(\log n)$[1]
Delete	$O(1)$[2]	$O(\log n)$[1]

Tracking the color of each node requires only one bit of information per node because there are only two colors. The tree does not contain any other data specific to it being a red–black tree, so its memory footprint is almost identical to that of a classic (uncolored) binary search tree. In some cases, the added bit of information can be stored at no added memory cost.

History

In 1972, Rudolf Bayer[5] invented a data structure that was a special order-4 case of a B-tree. These trees maintained all paths from root to leaf with the same number of nodes, creating perfectly balanced trees. However, they were not *binary* search trees. Bayer called them a "symmetric binary B-tree" in his paper and later they became popular as 2–3–4 trees or even 2–4 trees.[6]

In a 1978 paper, "A Dichromatic Framework for Balanced Trees",[7] Leonidas J. Guibas and Robert Sedgewick derived the red–black tree from the symmetric binary B-tree.[8] The color "red" was chosen because it was the best-looking color produced by the color laser printer available to the authors while working at Xerox PARC.[9] Another response from Guibas states that it was because of the red and black pens available to them to draw the trees.[10]

In 1993, Arne Andersson introduced the idea of a right leaning tree to simplify insert and delete operations.[11]

In 1999, Chris Okasaki showed how to make the insert operation purely functional. Its balance function needed to take care of only 4 unbalanced cases and one default balanced case.[12]

Toggle the table of contents

Stack (abstract data type)

In computer science, a **stack** is an abstract data type that serves as a collection of elements, with two main operations:

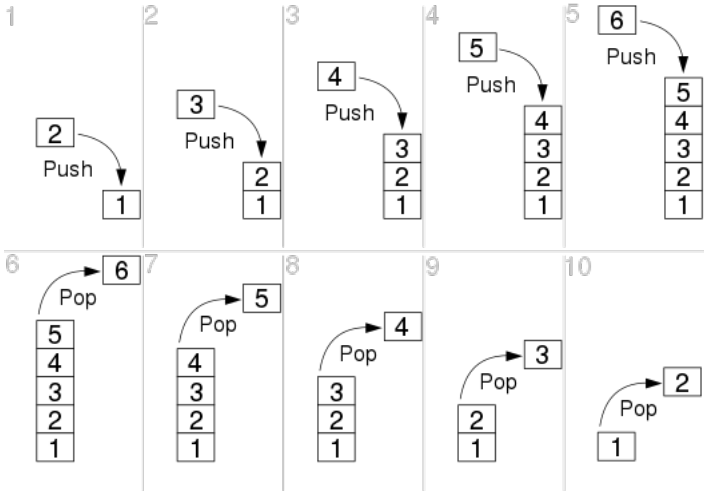
- **Push**, which adds an element to the collection, and
- **Pop**, which removes the most recently added element that was not yet removed.

Additionally, a peek operation can, without modifying the stack, return the value of the last element added. Calling this structure a *stack* is by analogy to a set of physical items stacked one atop another, such as a stack of plates.

Similar to a stack of plates, adding or removing is only possible at the top.

The order in which an element added to or removed from a stack is described as **last in, first out**, referred to by the acronym **LIFO**.[nb 1] As with a stack of physical objects, this structure makes it easy to take an item off the top of the stack, but accessing a datum deeper in the stack may require taking off multiple other items first.[1]

Considered as a linear data structure, or more abstractly a sequential collection, the push and pop operations occur only at one end of the structure, referred to as the *top* of the stack. This data structure makes it possible to implement a stack as a singly linked list and as a pointer to the top element. A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept another element, the stack is in a state of stack overflow.



Simple representation of a stack runtime with *push* and *pop* operations.

A stack is needed to implement depth-first search.

History

Stacks entered the computer science literature in 1946, when Alan M. Turing used the terms "bury" and "unbury" as a means of calling and returning from subroutines.[2][3] Subroutines and a 2-level stack had already been implemented in Konrad Zuse's Z4 in 1945.[4][5]

Klaus Samelson and Friedrich L. Bauer of Technical University Munich proposed the idea of a stack called