WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

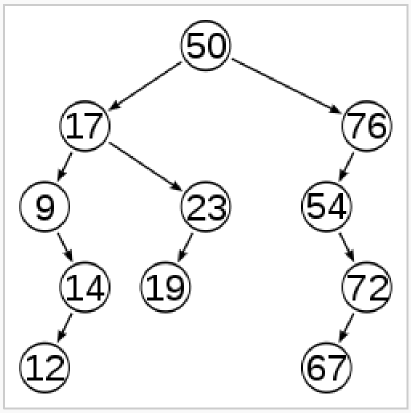
Languages
Deutsch
Español
فارسی
Français
한국어
Italiano
日本語
Српски / srpski
ไทย
Українська
中文
Edit links

Self-balancing binary search tree

From Wikipedia, the free encyclopedia

In computer science, a **self-balancing** (or **height-balanced**) **binary search tree** is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.[1]

These structures provide efficient implementations for mutable ordered lists, and can be used for other abstract data structures such as associative arrays, priority queues and sets.


An example of an **unbalanced** tree; following the path from the root to a node takes an average of 3.27 node accesses

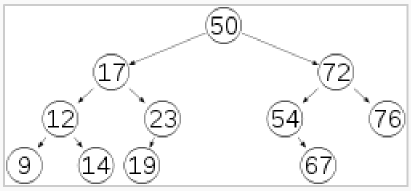
Overview [edit]

Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. A binary tree with height *h* can contain at most $2^0+2^1+\cdots+2^h = 2^{h+1}-1$ nodes. It follows that for a tree with *n* nodes and height *h*:

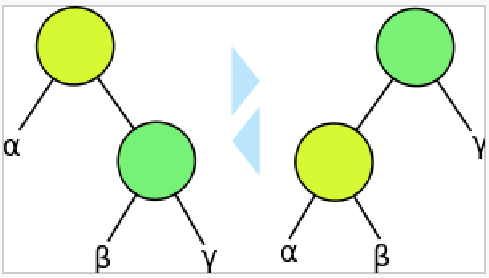
$$n \le 2^{h+1} - 1$$

And that implies:

$$h \ge \lceil \log_2(n+1) - 1 \rceil \ge \lfloor \log_2 n \rfloor.$$


The same tree after being height-balanced; the average path effort decreased to 3.00 node accesses

In other words, the minimum height of a tree with *n* nodes is $\log_2(n)$, rounded down; that is, $\lfloor \log_2 n \rfloor$.[1]



Tree rotations are very common internal operations on self-balancing binary trees to keep perfect or near-to-perfect balance.

However, the simplest algorithms for BST item insertion may yield a tree with height *n* in rather common situations. For example, when the items are inserted in sorted key order, the tree degenerates into a linked list with *n* nodes. The difference in performance between the two situations may be enormous: for *n* = 1,000,000, for example, the minimum height is $\lfloor \log_2(1{,}000{,}000) \rfloor = 19$.

If the data items are known ahead of time, the height can be kept small, in the average sense, by adding values in a random order, resulting in a random binary search tree. However, there are many situations (such as online algorithms) where this randomization is not viable.

Self-balancing binary trees solve this problem by performing transformations on the tree (such as tree rotations) at key times, in order to keep the height proportional to $\log_2(n)$. Although a certain overhead is involved, it may be justified in the long run by ensuring fast execution of later operations.

Maintaining the height always at its minimum value $\lfloor \log_2(n) \rfloor$ is not always viable; it can be proven that any insertion algorithm which did so would have an excessive overhead.[*citation needed*] Therefore, most self-balanced BST algorithms keep the height within a constant factor of this lower bound.

In the asymptotic ("Big-O") sense, a self-balancing BST structure containing *n* items allows the lookup, insertion, and removal of an item in O(log *n*) worst-case time, and ordered enumeration of all items in O(*n*) time. For some implementations these are per-operation time bounds, while for others they are amortized bounds over a sequence of operations. These times are asymptotically optimal among all data structures that manipulate the key only through comparisons.

Implementations [edit]

Popular data structures implementing this type of tree include:

- 2-3 tree
- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

Applications [edit]

Self-balancing binary search trees can be used in a natural way to construct and maintain ordered lists, such as priority queues. They can also be used for associative arrays; key-value pairs are simply inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables. One advantage of self-balancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items *in key order*, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key. Self-balancing BSTs have better worst-case lookup performance than hash tables (O(log n) compared to O(n)), but have worse average-case performance (O(log n) compared to O(1)).

Self-balancing BSTs can be used to implement any algorithm that requires mutable ordered lists, to achieve optimal worst-case asymptotic performance. For example, if binary tree sort is implemented with a self-balanced BST, we have a very simple-to-describe yet asymptotically optimal O(*n* log *n*) sorting algorithm. Similarly, many algorithms in computational geometry exploit variations on self-balancing BSTs to solve problems such as the line segment intersection problem and the point location problem efficiently. (For average-case performance, however, self-balanced BSTs may be less efficient than other solutions. Binary tree sort, in particular, is likely to be slower than merge sort, quicksort, or heapsort, because of the tree-balancing overhead as well as cache access patterns.)

Self-balancing BSTs are flexible data structures, in that it's easy to extend them to efficiently record additional information or perform new operations. For example, one can record the number of nodes in each subtree having a certain property, allowing one to count the number of nodes in a certain key range with that property in O(log *n*) time. These extensions can be used, for example, to optimize database queries or other list-processing algorithms.

See also [edit]

- Day–Stout–Warren algorithm
- Fusion tree
- Skip list
- Sorting

References [edit]

1. ^ [a] [b] Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 6.2.3: Balanced Trees, pp.458–481.

External links [edit]

- Dictionary of Algorithms and Data Structures: Height-balanced binary search tree
- GNU libavl, a LGPL-licensed library of binary tree implementations in C, with documentation

Wikimedia Commons has media related to ***Balanced Trees***.

v · t · e **Tree data structures** [show] v · t · e **Data structures** [show]

Categories: Binary trees | Soviet inventions | Trees (data structures)