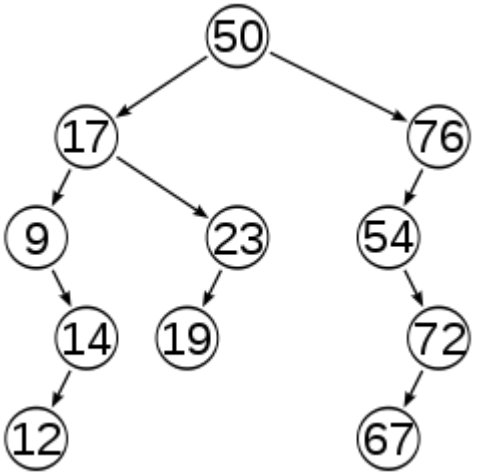
WIKIPEDIA

Self-balancing binary search tree

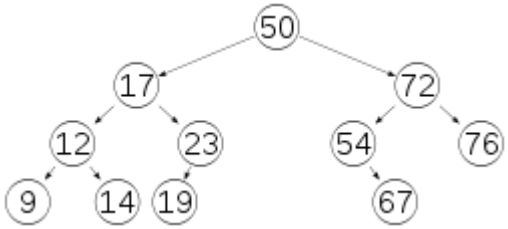
In computer science, a **self-balancing** (or **height-balanced**) **binary search tree** is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.[1]

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.

The red–black tree, which is a type of self-balancing binary search tree, was called symmetric binary B-tree[2] and was renamed but can still be confused with the generic concept of **self-balancing binary search tree** because of the initials.



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



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

Contents

Overview

Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. 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 any tree with n nodes and height h:

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

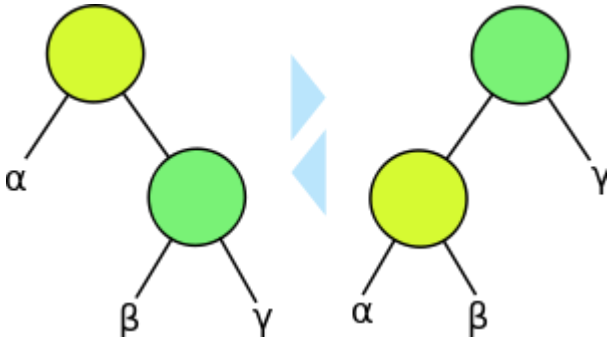
And that implies:

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

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

However, the simplest algorithms for BST item insertion may yield a tree with height n in rather common situations. For example, when the items are inserted in sorted key order, the tree degenerates into a linked list with n nodes. The difference in performance between the two situations may be enormous: for example, when n = 1,000,000, 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.



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

Self-balancing binary trees solve this problem by performing transformations on the tree (such as tree rotations) at key insertion 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.

While it is possible to maintain a BST with minimum height with expected $O(\log n)$ time operations (lookup/insertion/removal), the additional space requirements required to maintain such a structure tend to outweigh the decrease in search time. For comparison, an AVL tree is guaranteed to be within a factor of 1.44 of the optimal height while requiring only two additional bits of storage in a naive implementation.[1] 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

Data structures implementing this type of tree include:

- 2–3 tree
- AA tree
- AVL tree

- B-tree
- Red–black tree
- Scapegoat tree
- Splay tree
- Treap
- Weight-balanced tree

Applications

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

- Search data structure
- Day–Stout–Warren algorithm
- Fusion tree
- Skip list
- Sorting

References

1. 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.
2. Paul E. Black, "red–black tree", in Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, eds. 13 April 2015. (accessed 03 October 2016) Available from: https://xlinux.nist.gov/dads/HTML/redblack.html

External links

- Dictionary of Algorithms and Data Structures: Height-balanced binary search tree (https://xlinux.nist.gov/dads/HTML/heightBalancedTree.html)
- GNU libavl (http://adtinfo.org/), a LGPL-licensed library of binary tree implementations in C, with documentation