Balanced Binary Search Trees

A binary search tree can implement any of the basic dynamic-set operations in $O(h)$ time. These operations are $O(\log n)$ if tree is "balanced".

There has been lots of research on developing algorithms to keep binary search trees balanced, including:

Those that insert nodes as is done in the BST insert, then rebalance tree:
- Red-Black trees
- AVL trees
- Splay trees

Those that allow more than one key per node of the search tree:
- 2-3 trees
- 2-3-4 trees
- B-trees

Red-Black Trees (RBT) (Ch. 13)

Red-Black tree: BST in which each node is colored red or black. Constraints on the coloring and connection of nodes ensure that no root to leaf path is more than twice as long as any other, so tree is approximately balanced.

Each RBT node contains fields left, right, parent, color, and key.

Red-Black Properties

Red-Black tree properties:
1) Every node is either red or black.
2) The root is black.
3) Every leaf contains NIL and is black.
4) If a node is red, then both its children are black.
5) For each node $x$, all paths from $x$ to its descendant leaves contain the same number of black nodes.

Black Height $bh(x)$

Black-height of a node $x$: $bh(x)$ is the number of black nodes (including the NIL leaf) on the path from $x$ to a leaf, not counting $x$ itself.

Every node has a black-height, $bh(x)$.

For all NIL leaves, $bh(x) = 0$.

For root $x$, $bh(x) = bh(T)$.

Claim: Any node with height $h$ has black-height $\geq h/2$.

Red-Black Tree Height

Lemma: A red-black tree with $n$ internal nodes has height at most $2\log(n+1)$.

Proof:
Claim 1: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal nodes.

Proof is by induction on the height of the node $x$.

Basis: height of $x$ is 0 with $bh(x) = 0$. Then $x$ is a leaf and its subtree contains $2^0 - 1 = 0$ internal nodes.

Inductive step: Consider a node $x$ that is an internal node with 2 children. Each child of $x$ has $bh$ either equal to $bh(x)$ (red child) or $bh(x)-1$ (black child).

Red-Black Tree Height

Claim 1: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal nodes.

We can apply the IHOP to the children of $x$ to find that the subtree rooted at each child of $x$ has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at $x$ has at least $2(2^{bh(x)-1} - 1) + 1$ internal nodes $= 2^{bh(x)} - 1$ internal nodes.
Red-Black Tree Height

**Lemma:** A red-black tree with $n$ internal nodes has height at most $2\lg(n+1)$.

**Rest of proof of lemma:** Let $h$ be the height of the tree. By property 4 of RBTs, at least 1/2 the nodes on any root to leaf path are black. Therefore, the black-height of the root must be at least $h/2$.

Thus, by claims, $n \geq 2^{h/2} - 1$, so $n+1 \geq 2^{h/2}$, and, taking the $\lg$ of both sides, $\lg(n+1) \geq h/2$, which means that $h \leq 2\lg(n+1)$.

---

Operations on Red-Black Trees

All non-modifying bst operations (min, max, succ, pred, search) run in $O(h) = O(\lg n)$ time on red-black trees.

Insertion and deletion are more complex.

If we insert a node, what color do we make the new node?
* If red, the node might violate property 4.
* If black, the node might violate property 5.

If we delete a node, what color was the node that was removed?
* Red? OK, since we won't have changed any black-heights, nor will we have created 2 red nodes in a row. Also, if node removed was red, it could not have been the root.
* Black? Could violate property 4, 5, or 2.

---

Red-Black Tree Rotations

Algorithms to restore Red-Black tree property to tree after Tree-Insert and Tree-Delete include right and left rotations and re-coloring nodes.

We will not cover these RBT algorithms.

The number of rotations for insert and delete are constant, but they may take place at every level of the tree, so therefore the running time of insert and delete is $O(\lg(n))$.

---

AVL Trees

Developed by Russians Adelson-Velsky and Landis (hence AVL). This algorithm is not covered in our text, but the operations insert/delete are easier to understand than red-black tree insert/delete.

**Goal:** keep the height of a binary search tree low.

**Definition:** An AVL tree is a BST in which the balance factor of every node is either 0, +1, or -1. The balance factor of node $x$ is the difference in heights of nodes in $x$'s left and right subtrees (the height of the right subtree minus the height of the left subtree at each node).
AVL Trees

In the tree below, all nodes have a balance factor of 0, meaning that the subtrees at all nodes are balanced.

Inserting or deleting a node from an AVL tree can cause some node to have a balance factor of +2 or -2, in which case the algorithm causes rotations of particular branches to restore the 0, +1 or -1 balance factor at each node.

Inserting or deleting a node is done according to the regular BST insert or delete procedure and then the nodes are rotated (if necessary) to rebalance the tree. E.g., suppose we had the BST formed by inserting keys 20, 18, 22, 17, 19, 25. Now insert 23.

Example of AVL rotations

There are 4 basic types of rotation in an AVL tree:

- LL Rotation
- RR Rotation
- LR Rotation
- RL Rotation

The LR rotation is a combination of 2 separate rotations: The first rotation produces a tree with an LL imbalance and then the tree is balanced using the rules for an LL imbalance.

The RL rotation is a combination of 2 separate rotations: The first rotation produces a tree with an RR imbalance and then the tree is balanced using the rules for an RR imbalance.
Each unbalanced state requires at most two rotations (or "transplants"), with the changing of a constant number of pointers.

If a node is inserted into a balanced AVL tree, the tree is rebalanced by changing at most \( ? \) pointers.

**AVL Trees**

When a node is inserted and causes an imbalance as shown on the last slide, two rotations take place to restore the balance in the tree.

This is the RL case

Two rotations produces the balanced tree shown below.

This is the RR case

Correcting the imbalance after each insert/delete ensures that all dictionary operations on an AVL tree will take \( \Omega \) time in the worst-case.

**AVL rotation practice**

For each of these trees, show the balance factor at each node and indicate whether the tree is balanced.

**Inserting nodes into AVL tree**

Insert the following nodes into an AVL tree, in the order specified. Show the balance factor at each node as you add each one. When an imbalance occurs, specify the rotations needed to restore the AVL property. Nodes = \( <30, 40, 35, 21, 28, 29> \)

**Splay Trees**

Developed by Daniel Sleator and Robert Tarjan. Not covered in our text. Maintains a binary tree but makes no attempt to keep the height low. Uses a "Least Recently Used" (LRU) policy so that recently accessed elements are quick to access again because they are closer to the root. Basic operation is called splaying. Splaying the tree for a certain element rearranges it so that the splayed element is placed at the root of the tree. Performs a standard BST search and then does a sequence of rotations to bring the element to the root of the tree. Frequently accessed nodes will move nearer to the root where they can be accessed more quickly. These trees are particularly useful for implementing caches and garbage collection algorithms.

**Splay Trees**

Has amortized performance of \( O(\log(n)) \) if the same keys are searched for repeatedly.

*Amortized performance* means that some operations can be very expensive, but those expensive operations can be averaged with the costs of many fast operations.
Splaying

There are three types of splay steps (x is the node searched for and p is x’s parent):

**Zig Step:** This step is done when p is the root. The tree is rotated on the edge between x and p.

**Zig-zig Step:** This step is done when p is not the root and x and p are either both right children or are both left children.

**Zig-Zag Step:** This step is done when p is not the root and x is a right child and p is a left child or vice versa. (Looks like the LR in AVL trees, right?)

Splay trees yield $O(\log(n))$ running time only if the pattern of searches produces few rotations. This is a probabilistic algorithm. A very adversarial sequence of searches could result in a higher running time.

---

2-3 Trees

Developed by John Hopcroft in 1974. This algorithm is not covered in our text.

Another set of procedures to keep the height of a binary search tree low.

Definition: A 2-3 tree is a tree that can have nodes of two kinds: 2-nodes and 3-nodes. A 2-node contains a single key K and has two children, exactly like any other binary search tree node. A 3-node contains two ordered keys $K_1$ and $K_2$ ($K_1 < K_2$) and has three children. The leftmost child is the root of a subtree with keys less than $K_1$, the middle child is the root of a subtree with keys between $K_1$ and $K_2$, and the rightmost child is the root of a subtree with keys greater than $K_2$. The last requirement of a 2-3 tree is that all its leaves must be on the same level, i.e., a 2-3 tree is always perfectly height-balanced.

---

2-3 Trees

Search for a key k in a 2-3 tree:

1. Start at the root.
2. If the root is a 2-node, treat it exactly the same as a search in a regular binary search tree. Stop if k is equal to the root’s key, go to the left child if k is smaller, and to the right child if k is larger.
3. If the root is a 3-node, stop if k is equal to either of the root’s keys, go to the left child if k is less than $K_1$, to the middle child if k is greater than $K_1$ but less than $K_2$, and go to the right child if k is greater than $K_2$.
4. Treat each new node on the search path exactly the same as the root.

---

2-3 Trees

Insert a key k in a 2-3 tree: (node always inserted as leaf)

1. Start at the root.
2. Search for k until reaching a leaf.
   a) If leaf is a 2-node, insert k in proper position in leaf, either before or after key that already exists in the leaf, making the leaf a 3-node.
   b) If leaf is a 3-node, split the leaf in two: the smallest of the 3 keys (2 old ones and 1 new one) is put in the first leaf, the largest key is put in the second leaf, and the middle key is promoted to the old leaf’s parent. This may cause overload on the parent leaf and can lead to several node splits along the chain of the leaf’s ancestors. The splits go up the tree and the height expands upwards.
Inserting nodes into 2-3 tree

Insert the following nodes into a 2-3 tree, in the order specified. When an overload occurs, specify the changes needed to restore the 2-3 property. Nodes = 50 60 70 40 30 20 10 80 90

What is the smallest number of keys in a tree of height h? What is the largest number of keys in a tree of height h?

A 2-3 tree of height h with the smallest number of keys is ??? (height = ???). A 2-3 tree of height h with largest number of keys is ??? (height = ???). Therefore, all operations are ???.

B-Trees

Developed by Bayer and McCreight in 1972.

Our text covers these trees in Chapter 18.

B-trees are balanced search trees designed to work well on magnetic disks or other secondary-storage devices to minimize disk I/O operations.

Internal nodes can have a variable number of child nodes within some pre-defined range, m. B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes.

You are not expected to know anything more about B-Trees than that they are a strategy for maintaining a balanced search tree and are useful for accessing secondary-storage devices.

End of balanced bst lecture