CS 361, Lecture 20

Jared Saia
University of New Mexico

Outline

- Red-Black Trees

Randomly Built BSTs

- We want to answer the question: "What will be the average depth of a node in a randomly built tree?"
- Define the random variable X to be the depth of a node chosen uniformly at random in the tree
- X takes on n possible values, it takes on each value with probability $1/n$

Our Problem

- For a tree T and node x, let $d(x,T)$ be the depth of node x in T
- Define the total path length, $P(T)$, to be the sum over all nodes x in T of $d(x,T)$
- Then

$$\begin{aligned} E(X) &= \frac{1}{n}\sum_{x\in T} d(x,T) \\ &= \frac{1}{n}P(T) \end{aligned}$$

- Thus we want to show that $P(T) = O(n \log n)$

Analysis

"Shut up brain or I'll poke you with a Q-Tip" - Homer Simpson

- Let T_l, T_r be the left and right subtrees of T respectively. Let n be the number of nodes in T
- Then $P(T) = P(T_l) + P(T_r) + n - 1$. Why?

Analysis

- Let $P(n)$ be the expected total depth of all nodes in a randomly built binary tree with n nodes
- Note that for all i, $0 \leq i \leq n-1$, the probability that T_l has i nodes and T_r has $n-i-1$ nodes is $1/n$.
- Thus $P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1)$

Analysis

$$\begin{aligned}
P(n) &= \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) && (1)\\
&= \frac{1}{n}(\sum_{i=0}^{n-1}(P(i) + P(n-i-1)) + \frac{1}{n}(\sum_{i=0}^{n-1} n - 1)) && (2)\\
&= \frac{1}{n}(\sum_{i=0}^{n-1}(P(i) + P(n-i-1)) + \Theta(n) && (3)\\
&= \frac{2}{n}(\sum_{k=1}^{n-1} P(k)) + \Theta(n) && (4)\\
& && (5)
\end{aligned}$$

Analysis

- We have $P(n) = \frac{2}{n}(\sum_{k=1}^{n-1} P(k)) + \Theta(n)$
- This is the same as the recurrence for randomized Quicksort
- Recall from hw problem 7-2, that the solution to this recurrence is $P(n) = O(n \log n)$

Take Away

- $P(n)$ is the expected total depth of all nodes in a randomly built binary tree with n nodes.
- We've shown that $P(n) = O(n \log n)$
- There are n nodes total
- Thus the expected average depth of a node is $O(\log n)$

Take Away

- The expected average depth of a node in a randomly built binary tree is $O(\log n)$
- This implies that operations like search, insert, delete take expected time $O(\log n)$ for a randomly built binary tree

Warning!

- In many cases, data is not inserted randomly into a binary search tree
- I.e. many binary search trees are not "randomly built"
- For example, data might be inserted into the binary search tree in almost sorted order
- Then the BST would not be randomly built, and so the expected average depth of the nodes would not be $O(\log n)$

What to do?

- A Red-Black tree implements the dictionary operations in such a way that the height of the tree is always $O(\log n)$, where n is the number of nodes
- This will guarantee that no matter how the tree is built that all operations will always take $O(\log n)$ time

What is a RB-Tree

- A RB-Tree is a balanced binary search tree
- The height of the tree is always $O(\log n)$ where n is the number of nodes in the tree

RB Trees

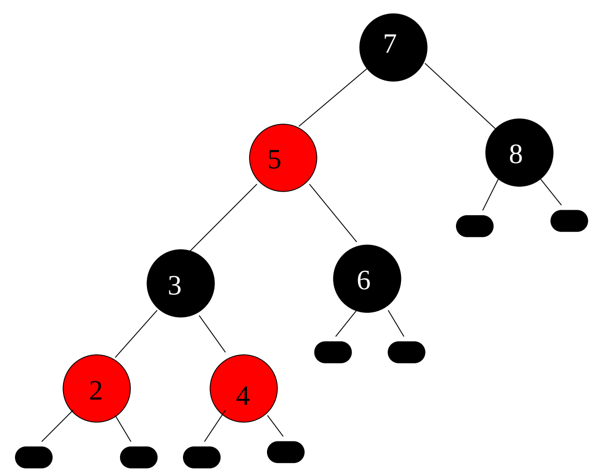
- Each node has a "color" field in addition to a key, left, right, and parent pointer
- If the child or parent of a node does not exist, the corresponding pointer field will contain the value NIL
- We will say that these NIL's are pointers to external nodes (leaves) of the tree, and say that all key-bearing nodes are internal nodes of the tree

Red-Black Properties

A BST is a red-black tree if it satisfies the RB-Properties

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Example RB-Tree



Black Height

- *Black-height* of a node x, bh(x) is the number of black nodes on any path from, but not including x down to a leaf node.
- Note that the black-height of a node is well-defined since all paths have the same number of black nodes
- The black-height of an RB-Tree is just the black-height of the root

Key Lemma

- *Lemma: A RB-Tree with n internal nodes has height at most* $2\log(n+1)$
- Proof Sketch:
  1. The subtree rooted at the node x contains at least $2^{bh(x)} - 1$ internal nodes
  2. For the root r, $bh(r) \geq h/2$, thus $n \geq 2^{h/2} - 1$. Taking logs of both sides, we get that $h \leq 2\log(n+1)$

Proof

1) The subtree rooted at the node x contains at least $2^{bh(x)} - 1$ internal nodes. Show by induction on the height of x.

- BC: If the height of x is 0, then x is a leaf, and subtree rooted at x does indeed contain $2^0 - 1 = 0$ internal nodes
- IH: For all nodes y of height less than x, the subtree rooted at y contains at least $2^{bh(y)} - 1$ internal nodes.
- IS: Consider a node x which is an internal node with two children(all internal nodes have two children). Each child has black-height of either $bh(x)$ or $bh(x) - 1$ (the former if it is red, the latter if it is black). Since the height of these children is less than x, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. This implies that the subtree rooted at x has at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. This proves the claim.

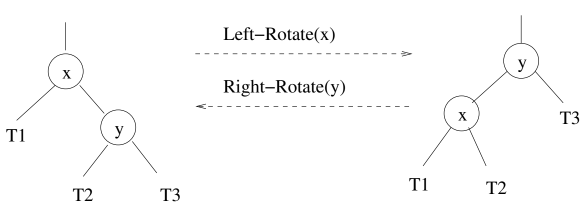
Maintenance?

- How do we ensure that the Red-Black Properties are maintained?
- I.e. when we insert a new node, what do we color it? How do we re-arrange the new tree so that the Red-Black Property holds?
- How about for deletions?

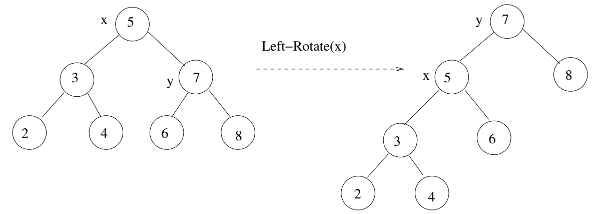
Left-Rotate

- Left-Rotate(x) takes a node x and "rotates" x with its right child
- Right-Rotate is the symmetric operation
- *Both Left-Rotate and Right-Rotate preserve the BST Property*
- We'll use Left-Rotate and Right-Rotate in the RB-Insert procedure

Picture



Example



Binary Search Tree Property

- Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y)$\leq$key(x). If y is a node in the right subtree of x then key(y)$\geq$key(x)

In-Class Exercise

Show that Left-Rotate(x) maintains the BST Property. In other words, show that if the BST Property was true for the tree before the Left-Rotate(x) operation, then it's true for the tree after the operation.

- Show that after rotation, the BST property holds for the entire subtree rooted at x
- Show that after rotation, the BST property holds for the subtree rooted at y
- Now argue that after rotation, the BST property holds for the entire tree