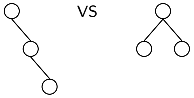
Balanced Trees

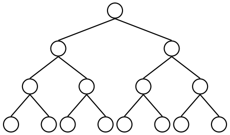
BST Performance

- Insertion, deletion, and searching are all ok if a tree branches a lot because they depend on the height of the tree



- A way to describe this is how balanced the tree is
  - A tree is balanced if any leaf is not “too much further” from the root than any other leaf
  - The definition of “too much further” depends on the kind of balanced tree

Absolute Best Case



- Height: $h = 3$;
- Number of Nodes: $n = 15$
- $h = \lfloor \log_2(n) \rfloor = \lfloor lg(n) \rfloor$
- To search in a best case tree, we need to probe at most $h = \lfloor lg(n) \rfloor$ nodes
  - Compare to a list, where we have to probe at most n nodes...

Side Break: Logs

It's round, it's heavy, it's wood

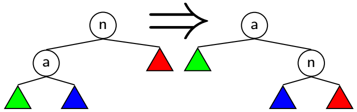
- The log operation undoes the power operation
  - If $a^b = x$, then $log_b(x) = a$
- As Computer Sciency type people, we like to work in base 2, and very frequently use $\log_2$
- As Computer Sciency type people, we are lazy, so we abbreviate $\log_2$ as *lg*

Balancing Trees

- Operations like insertion and deletion change the height of the tree
- To maintain the balance of the tree, we can rotate the tree around nodes
- Two kinds of rotations
  - Right
  - Left

Right Rotation

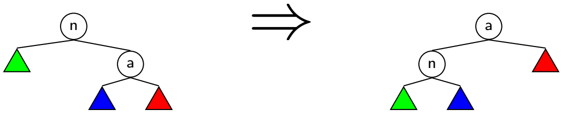
To rotate right around node n



- Reduces left height, increases right height

Right Rotation

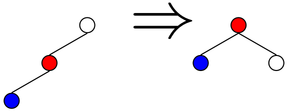
To rotate right around node n



- Reduces right height, increases left height

How Does This Help?

Let's rotate right around white



Much more balanced!

- We will use rotations in both red-black trees and AVL Trees

RB Trees

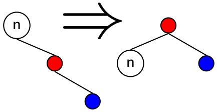
- R-B Trees assign a color to every node and use rules to ensure:
  - A node is either red or black
  - The root is black
  - All leaves (null) are black
  - If a node is red, both children are black
  - *Every path from root to a leaf has the same number of black nodes*
    - This is called the black height

AVL Trees

- AVL Trees have a simpler definition
    - In an AVL Tree the heights of the left and right subtrees of any node shall differ by no more than 1
- To accomplish this, AVL Trees use the concept of a Balance Factor (BF)
    - The Balance Factor of a tree rooted at node n is the height of n's right subtree - the height of n's left subtree
    - An important note: An empty tree has a height of -1 (by the definition we use in this class)
        - If a tree has no root, it's just a hole in the ground, so there is literally negative tree
- In an AVL Tree, the BF of a node may be either -1, 0, or 1. Any other cases require us to rebalance the tree
- There are four cases where we need to rotate to rebalance

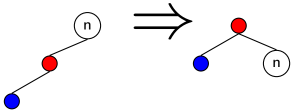
RR

- In the RR case, n's right child is heavier (BF(n)=2) and n.right's right child is heavier (BF(n.right)=1)
- This can be fixed with a left rotation around n



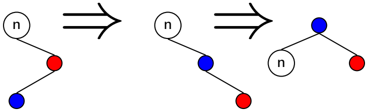
LL

- In the LL case, n's left child is heavier (BF(n)=-2) and n.left's right child is heavier (BF(n.left)=-1)
- This can be fixed with a right rotation around n



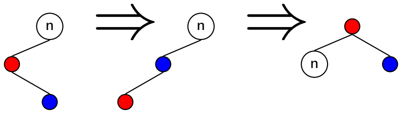
RL

- In the RL case, n's right child is heavier (BF(n)=2) and n.right's left child is heavier (BF(n.right)=-1)
- This can be fixed with a right rotation around n.right followed by a left rotation around n



LR

- In the LR case, n's right child is heavier (BF(n)=-2) and n.left's right child is heavier (BF(n.left)=1)
- This can be fixed with a left rotation around n.left followed by a right rotation around n



Case Summary

	Description	Rotations
RR	$BF(n) = 2, BF(n.right) = 1$	Left around n
LL	$BF(n) = -2, BF(n.left) = -1$	Right around n
RL	$BF(n) = 2, BF(n.right) = -1$	Right around n.right, left around n
LR	$BF(n) = -2, BF(n.left) = 1$	Left around n.right, right around n

Any time you modify the tree (insertion, deletion), begin at the node you just inserted and removed, then work your way up the tree. Do NOT wait until all operations are done.