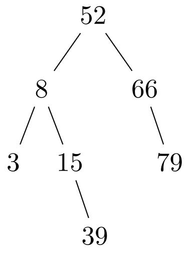
Trees

January 3, 2019

1 Binary Trees

- A Binary Tree is a collection of nodes
  - Each node has a left and a right pointer
  - A root node indicates the start of the tree
  - If the root is *None*, the tree is empty
- A Binary Search Tree is a binary tree where the value of any node is:
  - Greater than all the values in it's left subtree
  - Less than all the values in it's right subtree



2 Defining a Binary Search Tree

```
class Node:
    def __init__(self, item, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

class BST:
    def __init__(self):
        self.root = None

    def rec_add(self, node, item):
        if node == None:
            return Node(item)
        elif item < node.item:
            node.left = self.rec_add(node.left, item)
        else:
            node.rigth = self.rec_add(node.right, item)
        return node

    def add(self, item):
        self.root = self.rec_add(self.root, item)
```

---

3 Tree Traversals

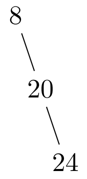
- Tree Traversal is just a way to access every element in the tree
- This can be done in 3 orders;
  - InOrder
  - PreOrder
  - PostOrder

---

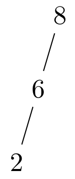
4 AVL trees

- An *AVL* tree is a Binary Search Tree where the height is maintained to $O(logN)$
  - This is done by adjusting the tree when it becomes unbalanced
  - A tree is considered unbalanced when the difference between the 2 child nodes is greater than 1
  - The *add* and *delete* operations need to be restructured to maintain balance

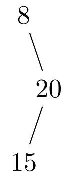
There are 4 ways that a tree can become unbalanced



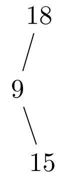
Right-Right



Left-Left

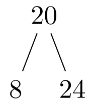


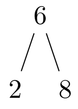
Right-Left

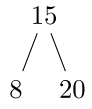


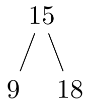
Left-Right

In each case, the tree can be balanced by replacing the root with the middle element









5 AVL adding an element

- Add the node to the tree
- Starting at this node, move up the tree towards the root, checking to see if any node is unbalanced
  - If it is, find out which case it is and perform the corresponding restructuring
  - Do this every time an element is added
  - It seems like a lot of work, but it's only $O(logN)$
  - No more than one restructuring is required per insert