Chapter 4 Binary Trees

Fei Tan

Chaifetz School of Business
Saint Louis University

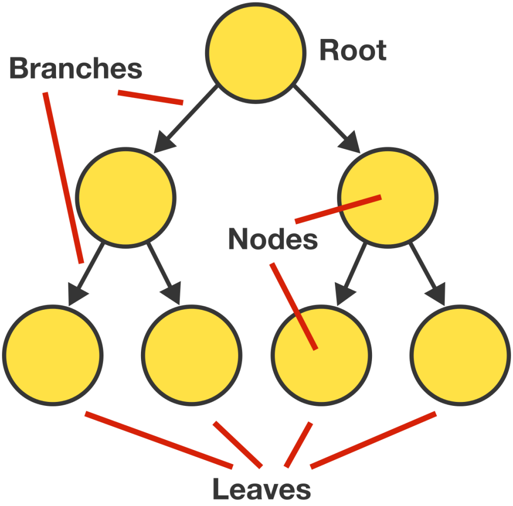
Computer Science Fundamentals
(Source: brilliant.org)

December 3, 2018

The Road Ahead...

- Binary trees find values fast by keeping numbers organized
- What we'll accomplish
  - keep to right & do in-order traversal of tree
  - use tree rotations to balance search trees

Data Structure: Binary Trees



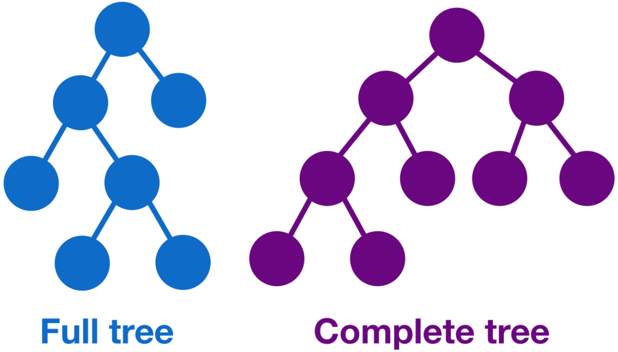
- Binary trees extend linked lists to trees; each node has two or fewer children

Python Code

Definition of binary tree node

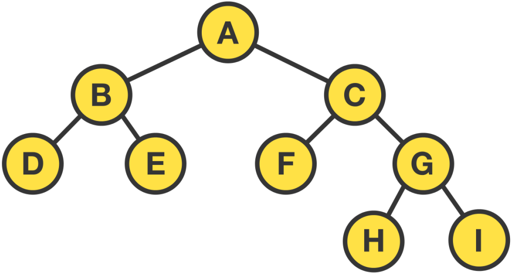
```
class Node:
    def __init__(self, value=None, left=None,
        right=None):
        self.value = value # node information
        self.left = left   # left child (subtree)
        self.right = right # right child (subtree)

    def __str__(self):
        return str(self.value)
```

Full vs. Complete Trees



- Fullness: each node has exactly 0 or 2 children
- Completeness: every level, except last, is completely filled; all nodes are as far left as possible
- A tree can be full (complete) but not complete (full)

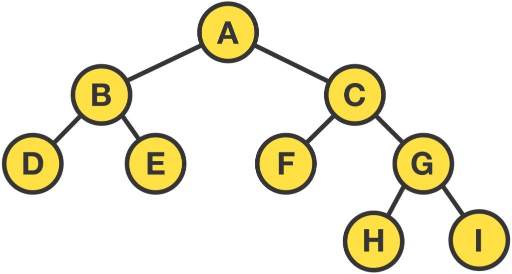
Algorithm: Pre-Order Traversal



Python Code

```
def traverse(tree):
    if tree:
        print(tree.getRootVal())
        traverse(tree.getLeftChild())
        traverse(tree.getRightChild())
#outcome: ABDECFGHI
```

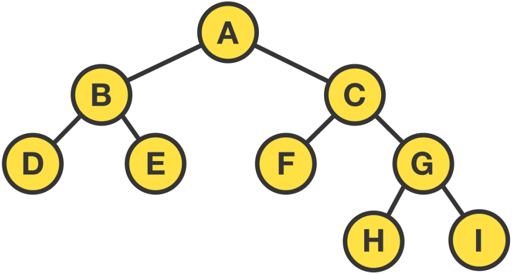
Algorithm: In-Order Traversal



Python Code

```
def traverse(tree):
    if tree:
        traverse(tree.getLeftChild())
        print(tree.getRootVal())
        traverse(tree.getRightChild())
#outcome: DBEAFCHGI
```

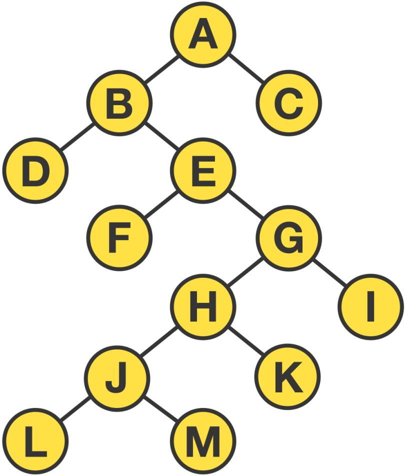
Algorithm: Post-Order Traversal



Python Code

```
def traverse(tree):
    if tree:
        traverse(tree.getLeftChild())
        traverse(tree.getRightChild())
        print(tree.getRootVal())
#outcome: DEBFHIGCA
```

Algorithms: Depth- vs. Breadth-First Search



- DFS: ABDEFGHJLMKIC (pre-order)
- BFS: ABCDEFGHIJKLM (left to right)

Python Code

DFS implementation

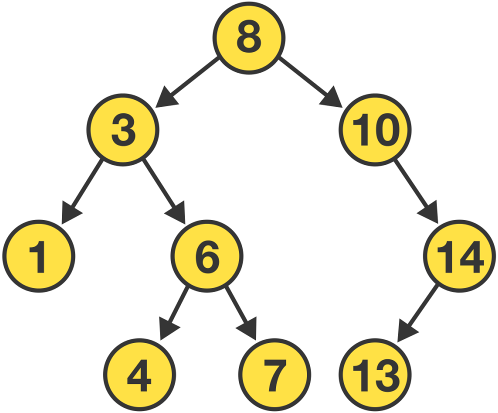
```python
def DFS(node):
    if not node:
        return []
    result = []
    if node.left:
        # append left subtree's values
        result = result + DFS(node.left)
    if node.value:
        # append this node's value
        result.append(node.value)
    if node.right:
        # append right subtree's values
        result = result + DFS(node.right)
    return result
```

Python Code (Cont'd)

BFS implementation

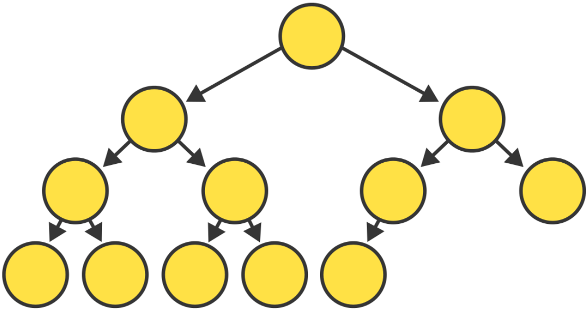
```
def BFS(node):
    result = []
    nodeList = [node]
    # nodeList contains nodes for height n
    while nodeList:
        # nextNodeList contains nodes for height n
            +1
        nextNodeList = []
        for subnode in nodeList:
            # append current node's value
            result.append(subnode.value)
            # append current node's children
            if subnode.left:
                nextNodeList.append(subnode.left)
            if subnode.right:
                nextNodeList.append(subnode.right)
        nodeList = nextNodeList
    return result
```

Data Structure: Binary Search Trees (BST)



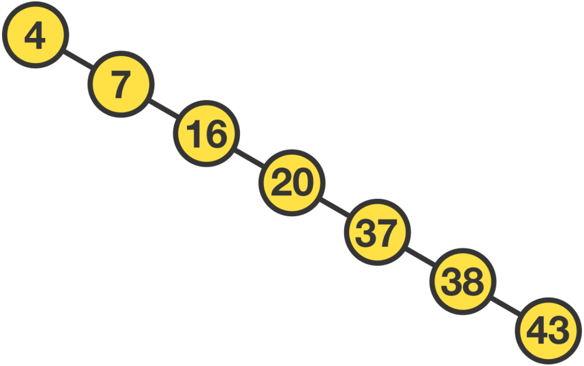
- All elements in any left sub-tree $<$ parent
- All elements in any right sub-tree $>$ any element in left sub-tree

BST Optimal Performance



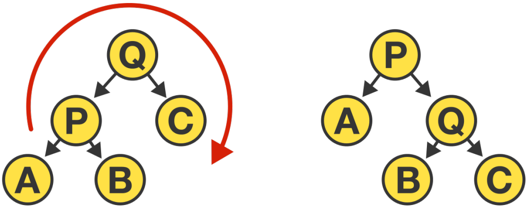
- Time complexity of complete/balanced BST: $O(\log N)$

BST Worst Performance

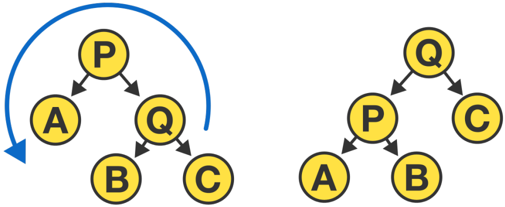


- Time complexity of maximally unbalanced BST: $O(N)$

Basic Tree Rotations

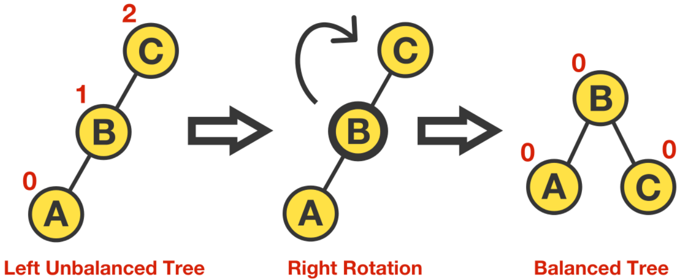


Right rotation



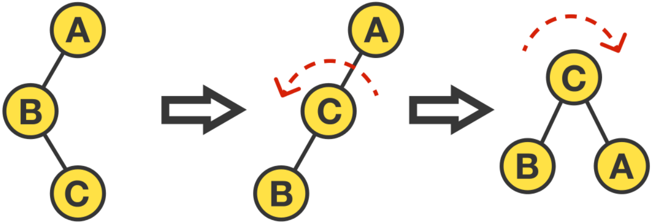
Left rotation

Algorithm: AVL Rotation



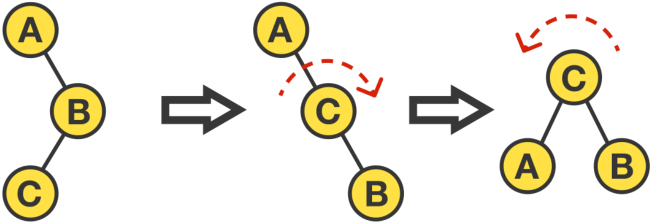
- Adelson-Velskii and Landis (AVL) self-balancing BST
- Branch pattern: left-left

Algorithm: AVL Rotation (Cont'd)



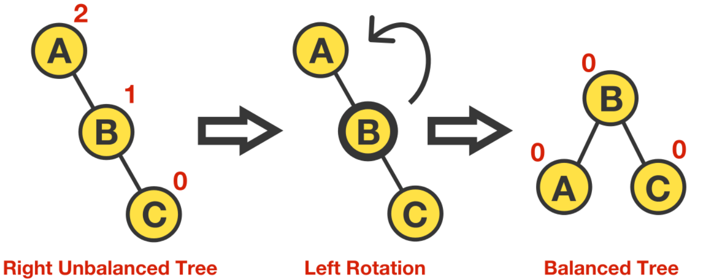
- Adelson-Velskii and Landis (AVL) self-balancing BST
- Branch pattern: left-right

Algorithm: AVL Rotation (Cont'd)



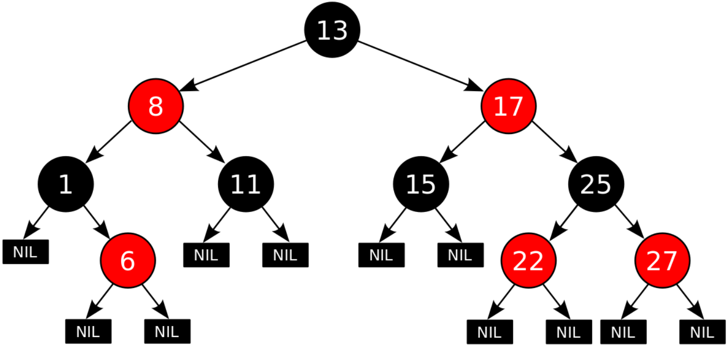
- Adelson-Velskii and Landis (AVL) self-balancing BST
- Branch pattern: right-left

Algorithm: AVL Rotation (Cont'd)



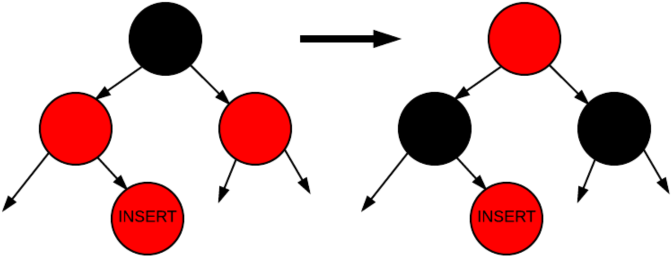
- Adelson-Velskii and Landis (AVL) self-balancing BST
- Branch pattern: right-right

Data Structure: Red-Black Trees (RBT)



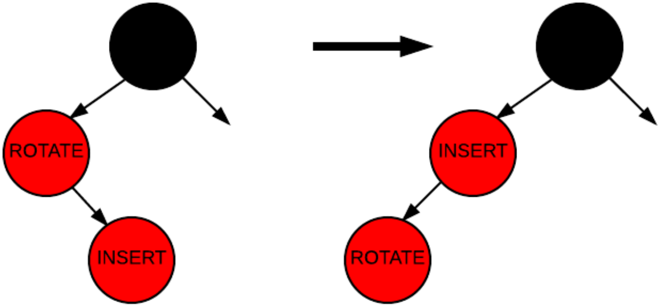
- Self-balancing BST; slower lookup but faster insertion & deletion than AVL trees
- Each node is red/black; newly inserted node is red; red node's parent is black
- Time complexity: $O(\log N)$

Algorithm: RBT Insertion (Case 1)



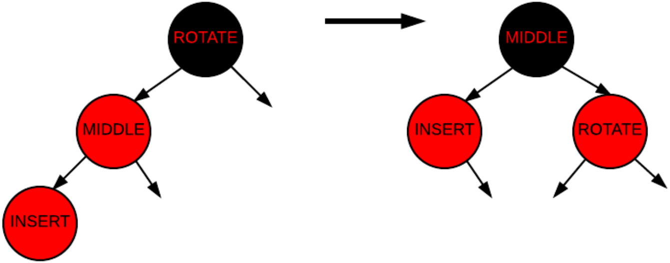
- Same for deletion

Algorithm: RBT Insertion (Case 2)



- Then apply case 3
- Same for deletion

Algorithm: RBT Insertion (Case 3)



- Same for deletion