2-3 Trees

CS112 Recitation

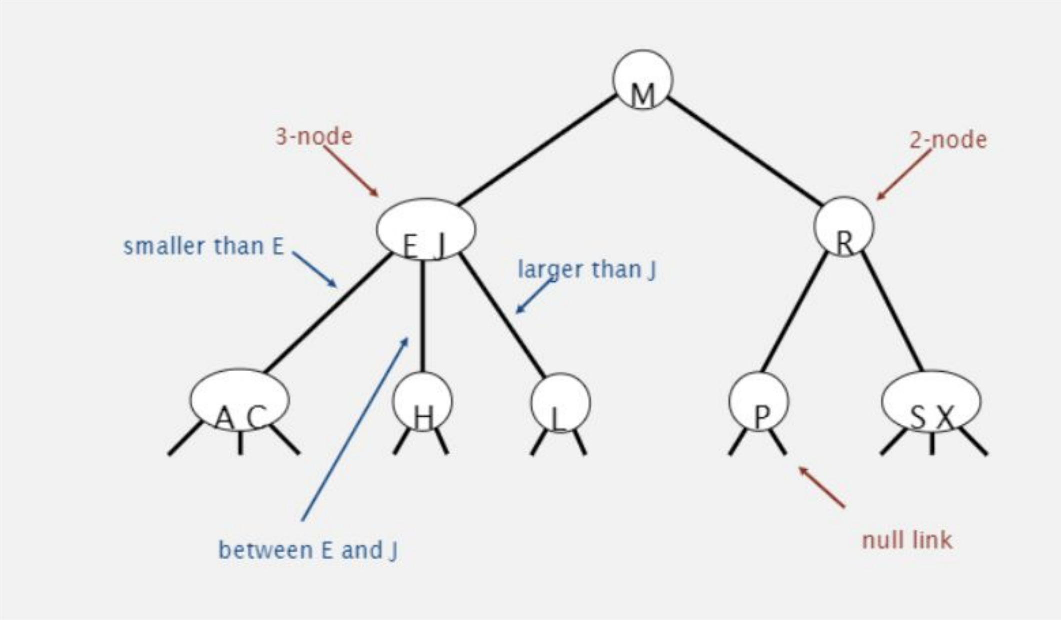
Original slides by Hoai-An Nguyen

Let’s Review: 2-3 Trees, Searching

- A 2-3 Tree is always

____
- Always have either

_ key and _ children, or _ keys and _ children



Warm-Up: Worst case runtime in a 2-3 tree?

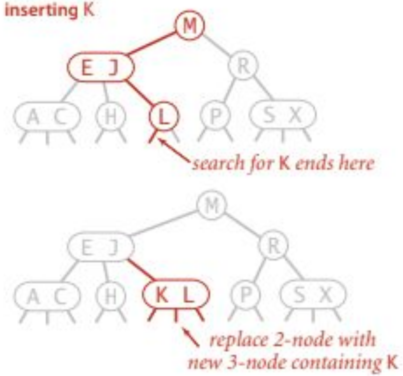
1. What is the worst-case runtime of search and insertion in a 2-3 tree?
2. What is the space complexity of a 2-3 tree?
3. Why are red black trees generally used over 2-3 trees?

Warm-up Answer

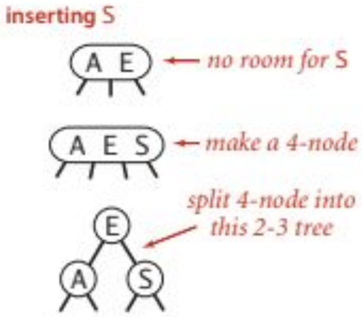
1. O(logn) for both
   a. Since it is balanced, inserting and search takes O(logn). Balancing operations are all constant time.
2. O(n)
3. It's much easier to implement!

Insertion “cheat sheet”

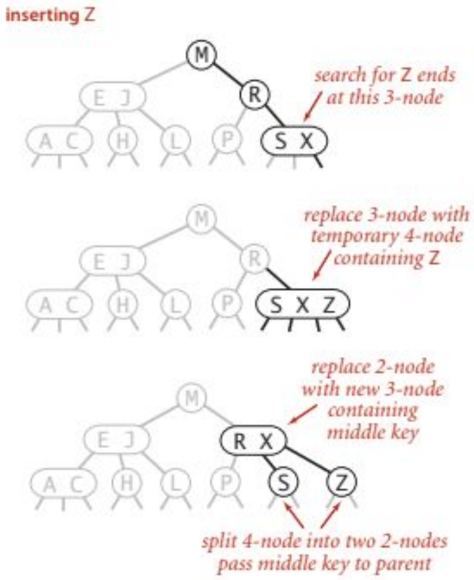
Inserting into an empty tree: trivial



Insert into a 2-node

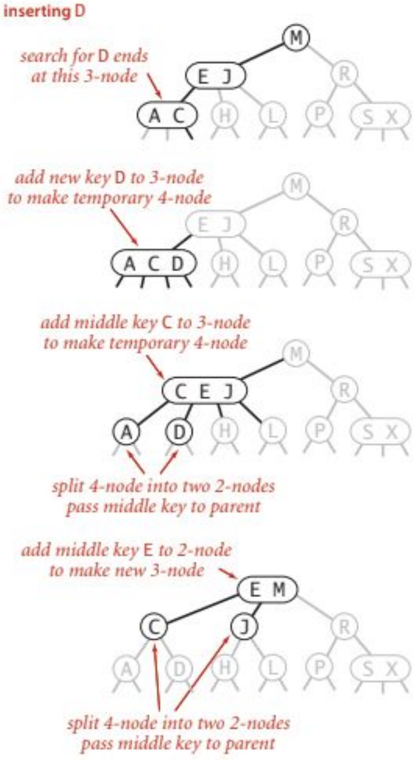


Insert into a single 3-node



Insert into a 3-node whose parent is a 2-node

Insertion “cheat sheet”



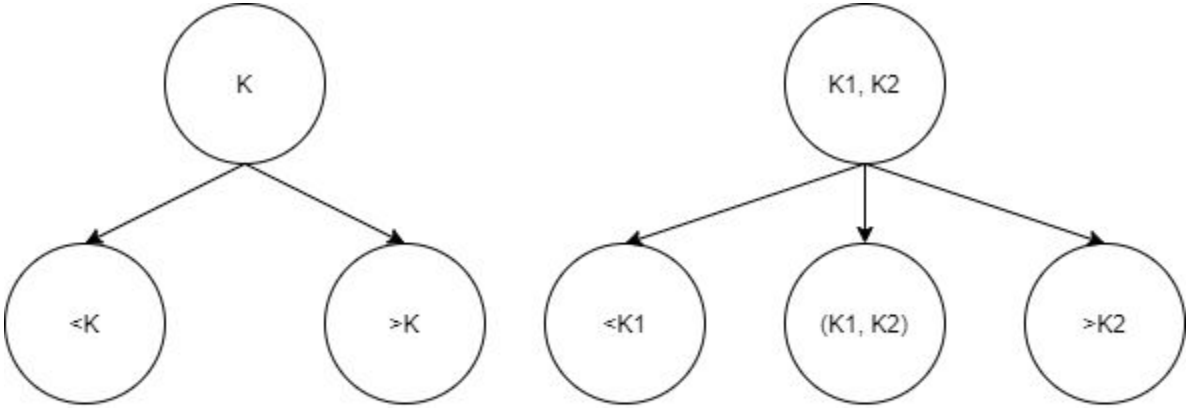
Insert into a 3-node whose parent is a 3-node

General steps for insertion

1. Search until you hit a leaf node
2. Insert into that node, making the node potentially a 3-node or a 4-node
3. Check for violations
    a. If we have a 4-node, we will need to split and rebalance
    b. If you rebalance, check again for violations and rebalance if needed (repeat this until no violations)

Insertion: 2-3 Tree

1. Draw the 2-3 tree that results when you insert the keys E A S Y Q U T I O N in that order into an initially empty tree.



2 NODE

3 NODE

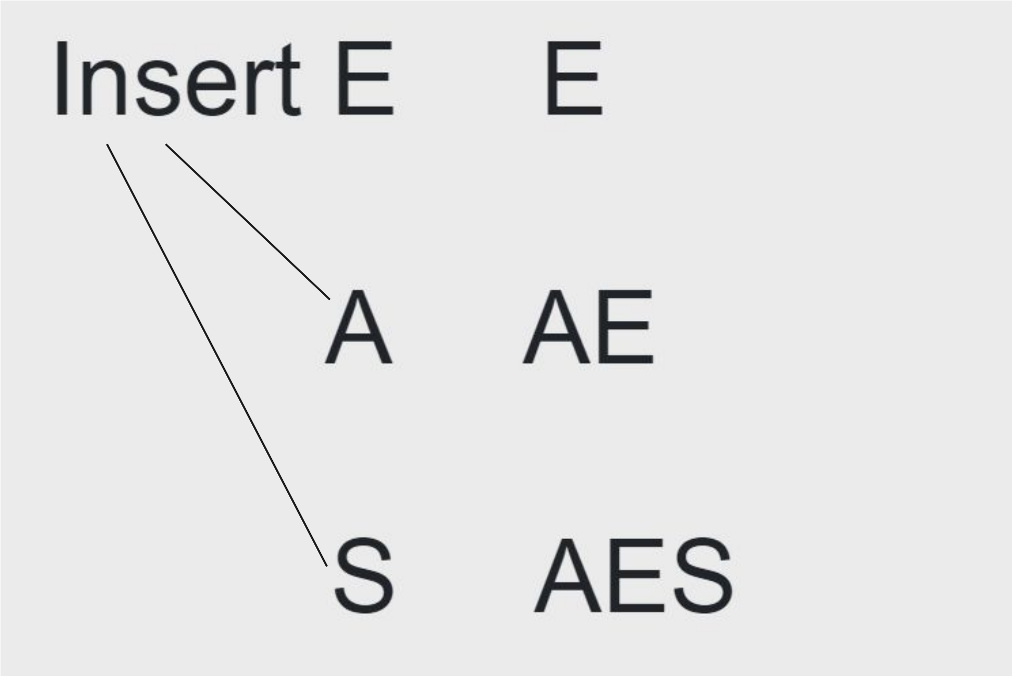
Insert E

Insert A

Insert S

Insert Y

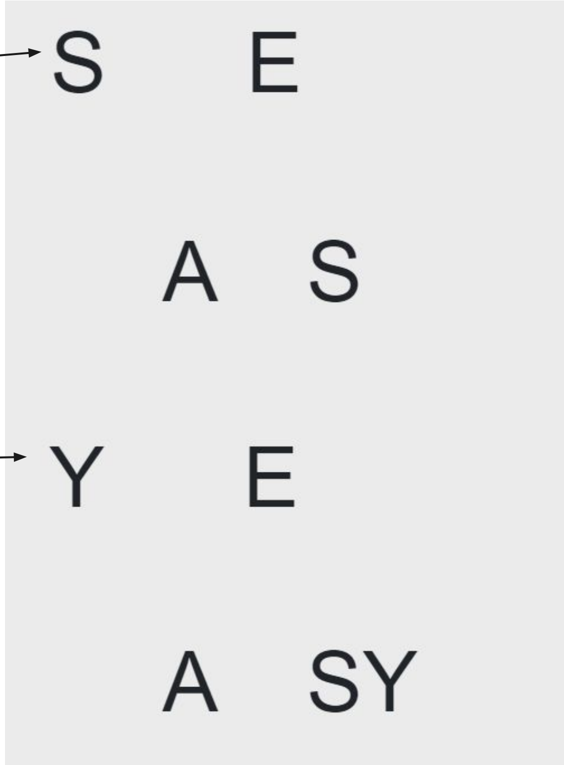
Answer: Insert E, A, S



Answer: Insert S, Y

Rebalance

Insert

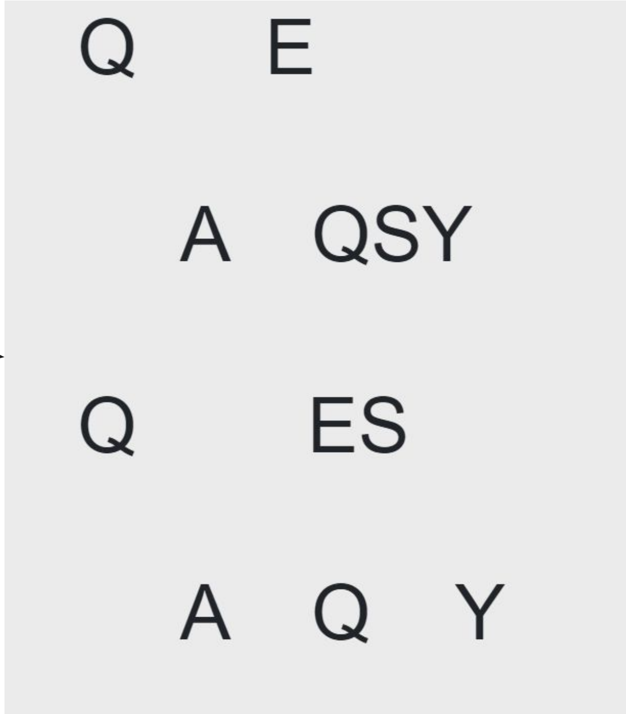


Insert Q

Answer: Insert Q

Insert

Rebalance



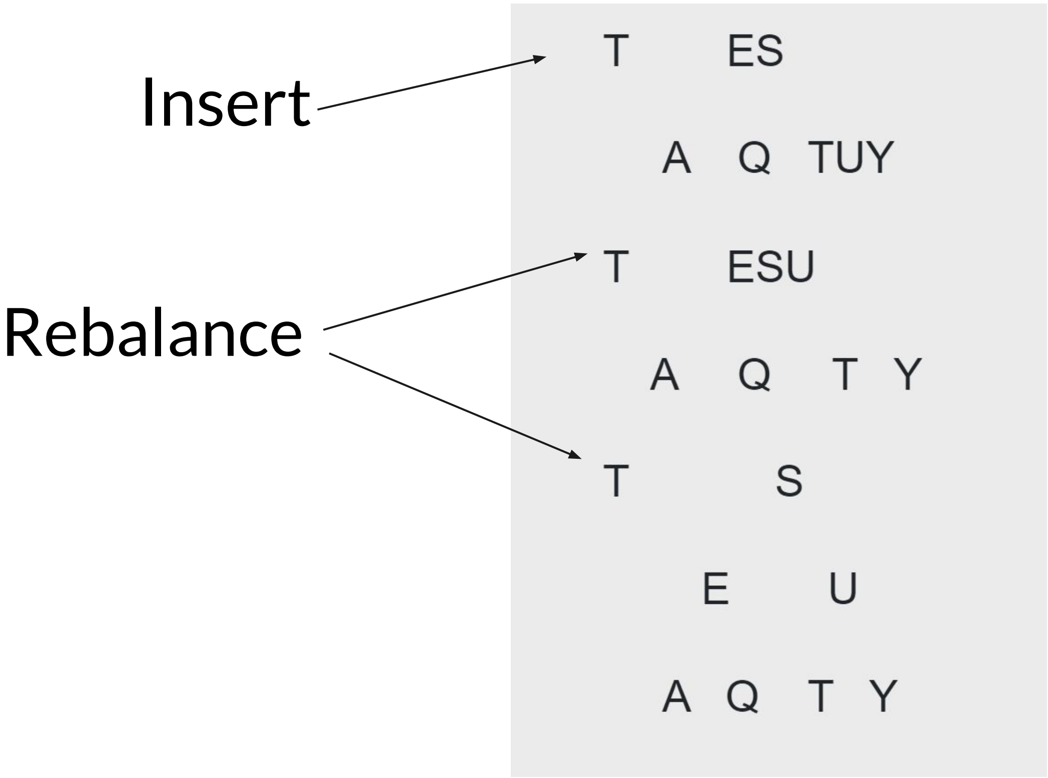
Insert U

Answer: Insert U

Insert

Insert T

Answer: Insert T



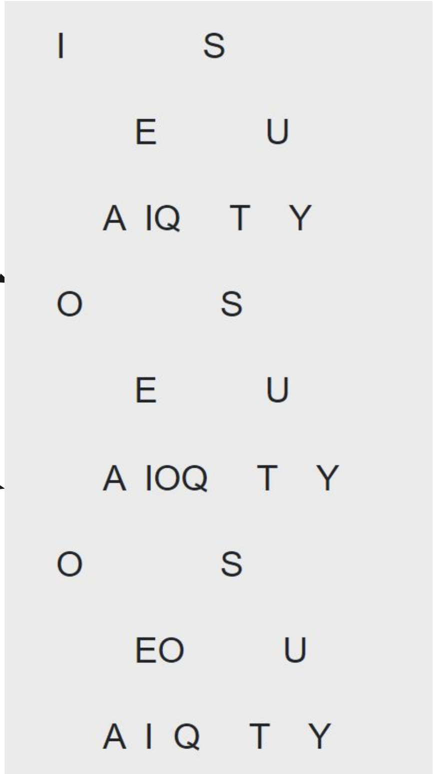
Insert I

Insert O

Answer: Insert I, O

Insert

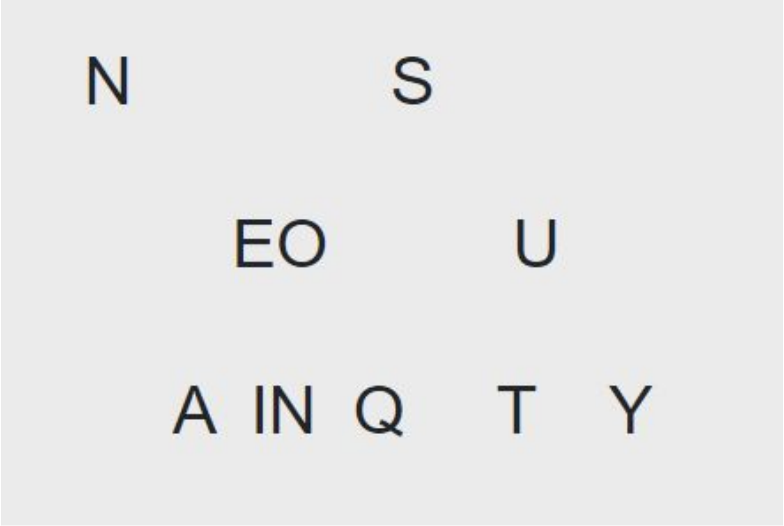
Rebalance



Insert N

Answer: Insert N

Insert



Check if a Binary Tree is Balanced

In a balanced tree a binary tree, the left and right subtrees of every node differ in height by no more than 1. Write a method that determines whether a tree is balanced.

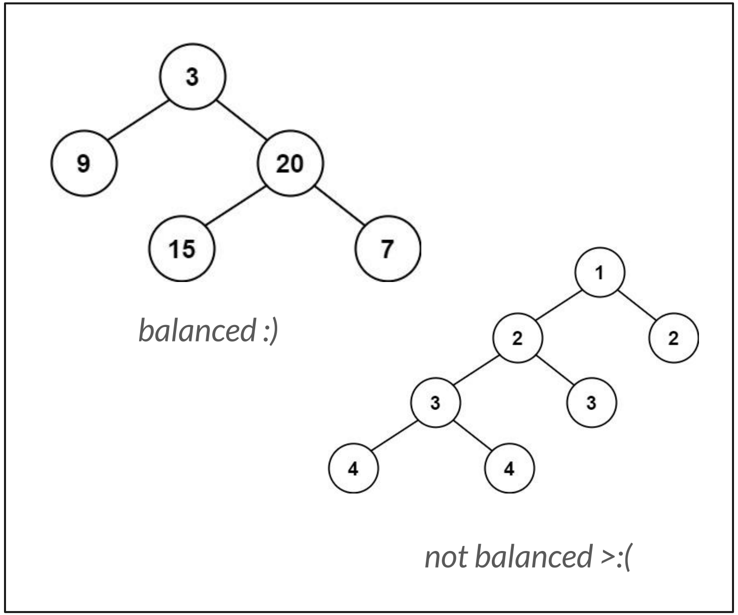
Use the following Node class:

```
public class Node     {
        int val; // associated value
        Node left, right;  // links to subtrees
}
```

Use the header:

```
public boolean isBalanced( Node root )
```

Hint: use a (recursive) helper method!



```
public boolean isBalanced (Node root) {

        if (root == null)
                return ___________________________

        if (_____________________________________________________________)
                return false;

        return _______________________________________________________;
        // remember, every single node in the tree needs to be balanced for the tree to be balanced
}


private int getDepth(Node root) {
        if (root == null)
                return 0;

        return Math.max(getDepth(root.left), getDepth(root.right)) + 1;
}
```

Check if a Binary Tree is Balanced Solution

```
public boolean isBalanced(Node root) {
  if (root == null)
      return true;
  if (Math.abs(getDepth(root.left) - getDepth(root.right)) >1)
      return false;
  return isBalanced(root.left) && isBalanced(root.right);
}

private int getDepth(Node root){
    if (root == null)
        return 0;
    return Math.max(getDepth(root.left), getDepth(root.right)) + 1;
}
```

This is a fairly intuitive, top-down approach. However, is there a way we could optimize our algorithm? What work is being repeated, and how can we eliminate redundancy?

Optimizing isBalanced()

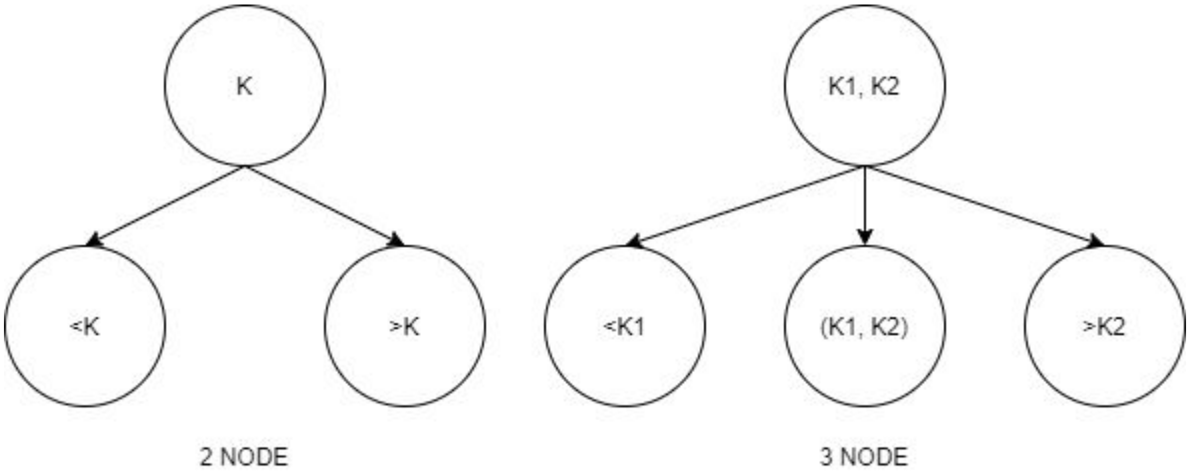
In the previous page's top-down approach, we start at the root and call getDepth() for every single node in the tree. We end up traversing subtrees multiple times, and this repeated work is unnecessary.

What if we instead utilize a bottom-up approach? If leaf nodes have a height of 0, then we can recurse down to them and keep track of the subtree's height as the recursive steps return.

Consider this on your own time, and watch this video explanation for further clarification: https://www.youtube.com/watch?v=LU4fGD-fgJQ

Insert: 2-3 Tree Q2 (Optional)

Find an insertion order for the keys S E A R C H X M that leads to a 2-3 tree of height 1.



Insert S E A R C H X M

Insert S E A R C H X M

Insert: 2-3 Tree Q2 Answer (Optional)

Insertion order: S E A R C H X M

Insert S S

Insert E SE

Insert A AES

E

A S

Insert R E

A RS

Insert C E

AC RS

Insert H E

AC HRS

ER

AC H S

Insert X ER

AC H SX

Insert M ER

AC HM SX

Good Work!

Go to https://dynrec.cs.rutgers.edu/live/

Enter the Quiz Code:

Have a great spring break! :)