



# 2-3 Trees

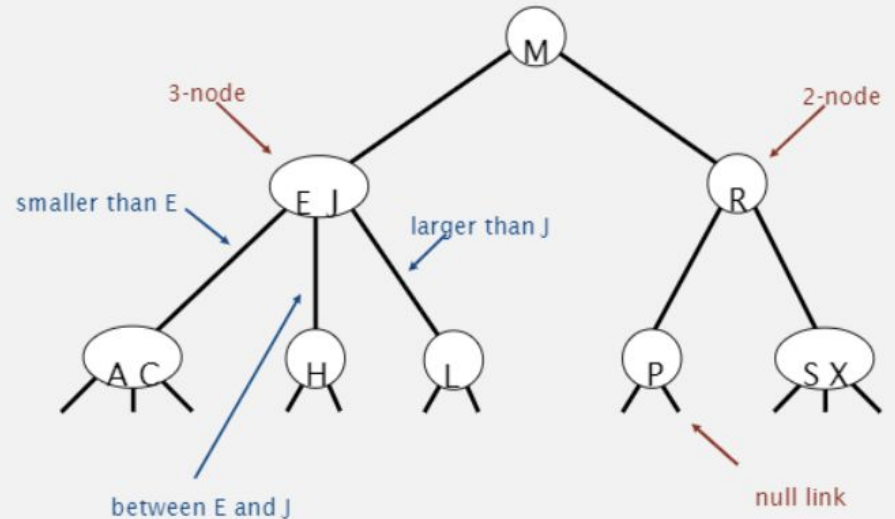
CS112 Recitation

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?

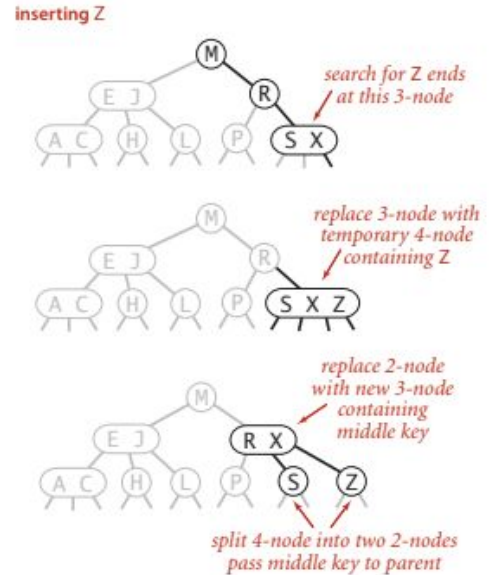
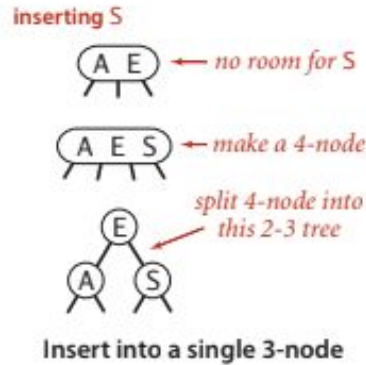
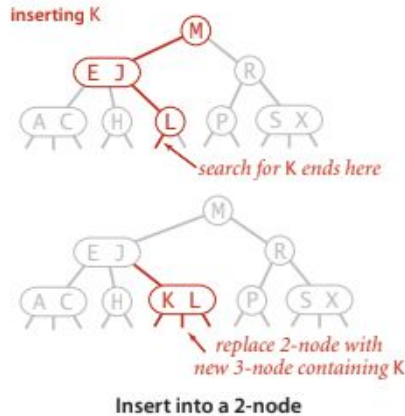
# Warm-up Answer



1.  $O(\log n)$  for both
  - a. Since it is balanced, inserting and search takes  $O(\log n)$ . 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

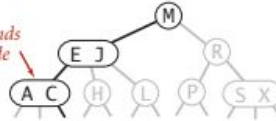


Insert into a 3-node whose parent is a 2-node

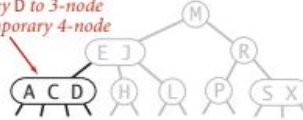
# Insertion “cheat sheet”

inserting D

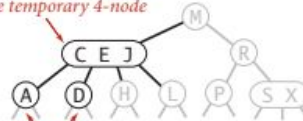
search for D ends  
at this 3-node



add new key D to 3-node  
to make temporary 4-node

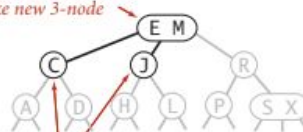


add middle key C to 3-node  
to make temporary 4-node



split 4-node into two 2-nodes  
pass middle key to parent

add middle key E to 2-node  
to make new 3-node



split 4-node into two 2-nodes  
pass middle key to parent

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

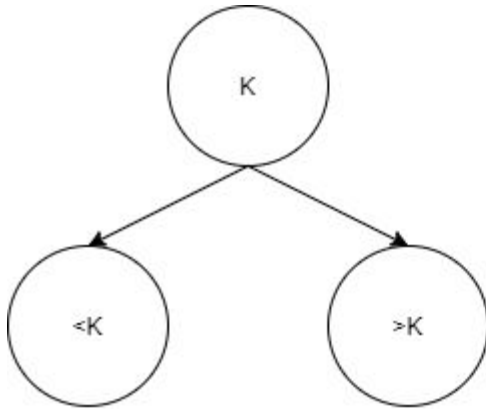
# General steps for insertion



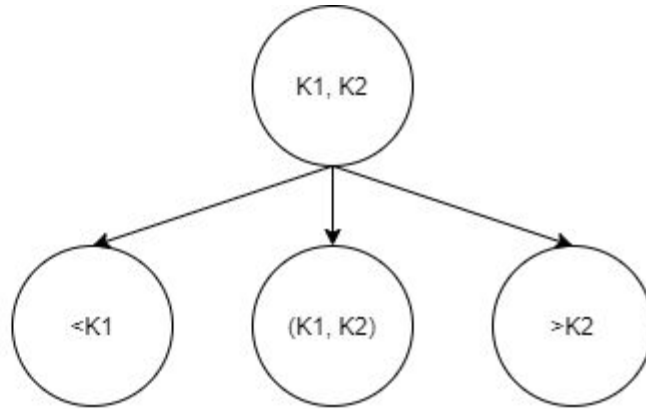
1. Search until you hit a leaf node
2. Insert into that node, making the node potentially a 3-node or a 4-node
3. Check for violations
  - a. If we have a 4-node, we will need to split and rebalance
  - b. If you rebalance, check again for violations and rebalance if needed (repeat this until no violations)

# 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



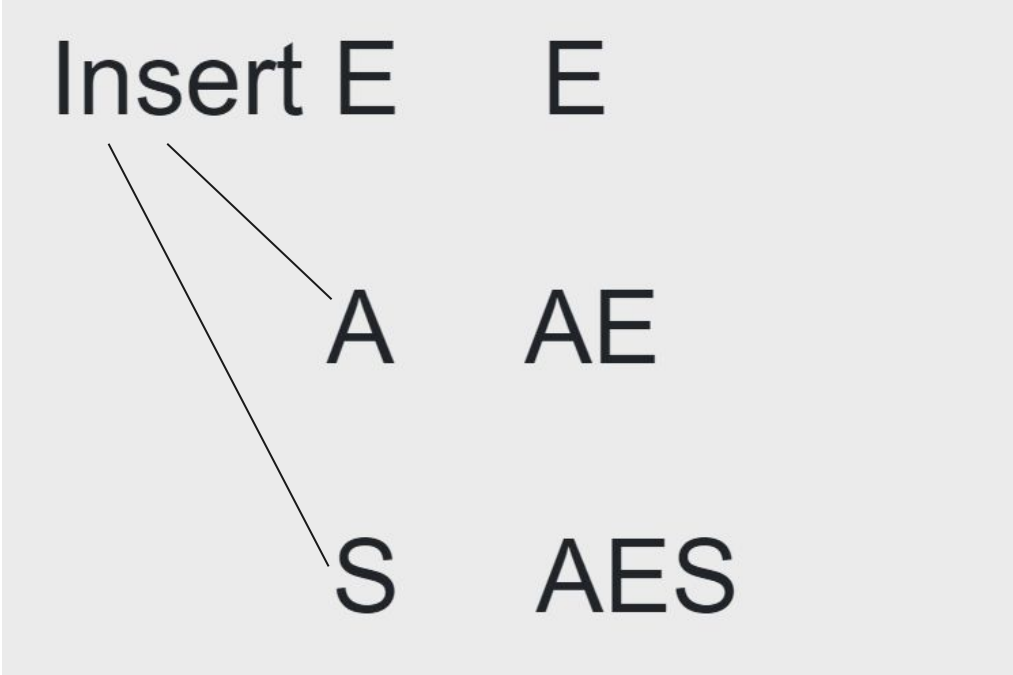
# Insert S



# Insert Y



Answer: Insert E, A, S



Insert E	E
A	AE
S	AES

# Answer: Insert S, Y



Rebalance → S E

A S

Insert → Y E

A SY

# Insert Q



# Answer: Insert Q

Insert →

Q      E

A      QSY

Rebalance →

Q      ES

A      Q      Y



# Insert U



Answer: Insert U

Insert →

U

ES

A

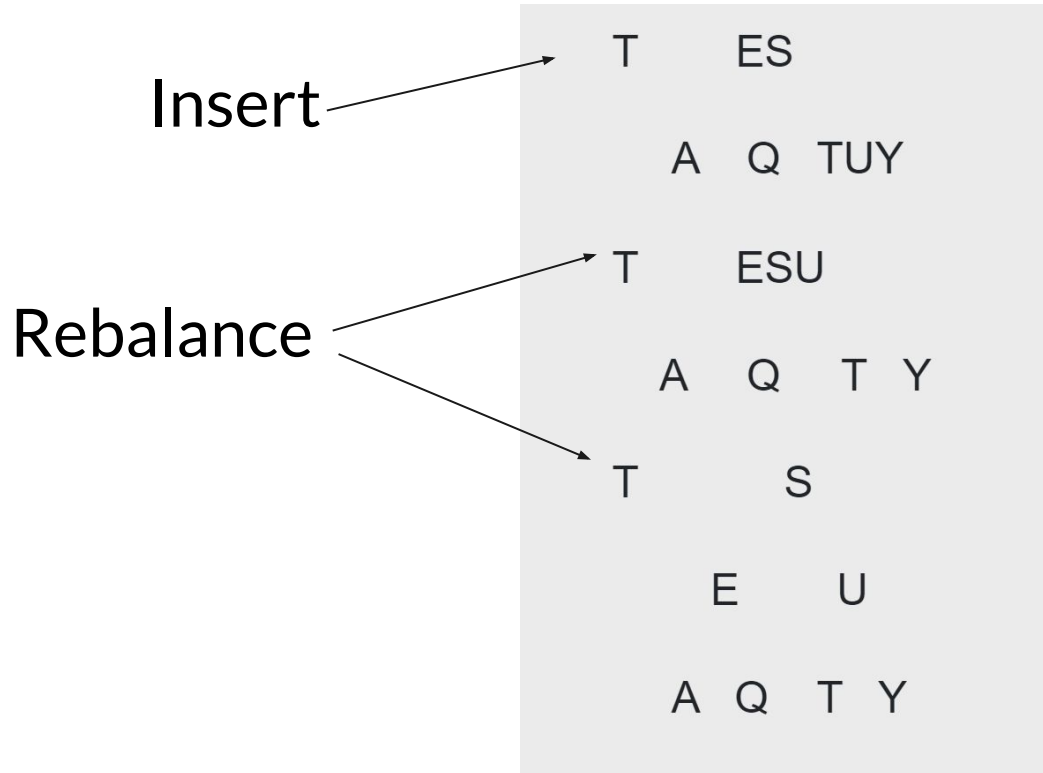
Q

UY

# Insert T



# Answer: Insert T



# Insert I



# Insert O



# Answer: Insert I, O

Insert

Rebalance

The diagram illustrates a B-tree structure with three levels. The root node contains 'I' and 'S'. It has three pointers leading to three leaf nodes. The first leaf node contains 'E' and 'U'. The second leaf node contains 'A', 'IQ', 'T', and 'Y'. The third leaf node contains 'O' and 'S'. An arrow from the word 'Insert' points to the first leaf node and the second leaf node. Below this, the word 'Rebalance' has an arrow pointing to the second leaf node. The rebalanced state shows the root node containing 'I' and 'S'. The first leaf node now contains 'E' and 'U'. The second leaf node now contains 'A', 'IOQ', 'T', and 'Y'. The third leaf node now contains 'O' and 'S'. The leaf node that previously contained 'E' and 'U' is now merged into the second leaf node, and the leaf node that previously contained 'O' and 'S' is now merged into the third leaf node.

I	S			
E	U			
A	IQ	T	Y	
O	S			
E	U			
A	IOQ	T	Y	
O	S			
EO	U			
A	I	Q	T	Y

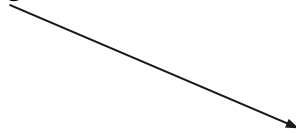
# Insert N





# Answer: Insert N

Insert



N			S		
		EO		U	
	A	IN	Q	T	Y

# 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.

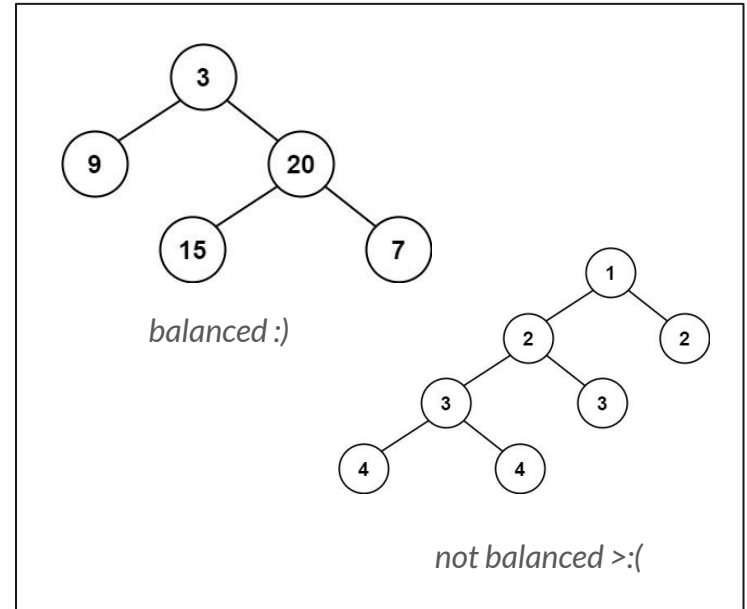
Use the following Node class:

```
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

---

```
1  public boolean isBalanced(Node root) {
2      if (root == null)
3          return true;
4      if (Math.abs(getDepth(root.left) - getDepth(root.right)) > 1)
5          return false;
6      return isBalanced(root.left) && isBalanced(root.right);
7  }
8
9  private int getDepth(Node root){
10     if (root == null)
11         return 0;
12     return Math.max(getDepth(root.left), getDepth(root.right)) + 1;
13 }
```

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()



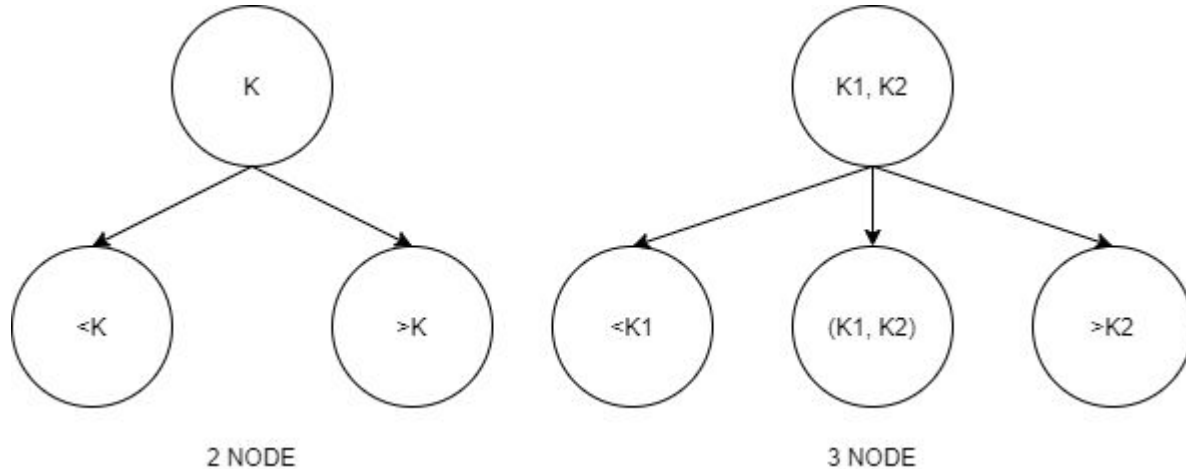
In the previous page's top-down approach, we start at the root and call `getDepth()` for every single node in the tree. We end up traversing subtrees multiple times, and this repeated work is unnecessary.

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 SEARCH XM



# Insert SEARCH XM





# 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! :)