CSE 332 Spring 2026
Lecture 7: Dictionaries, BSTs

Nathan Brunelle

http://www.cs.uw.edu/332

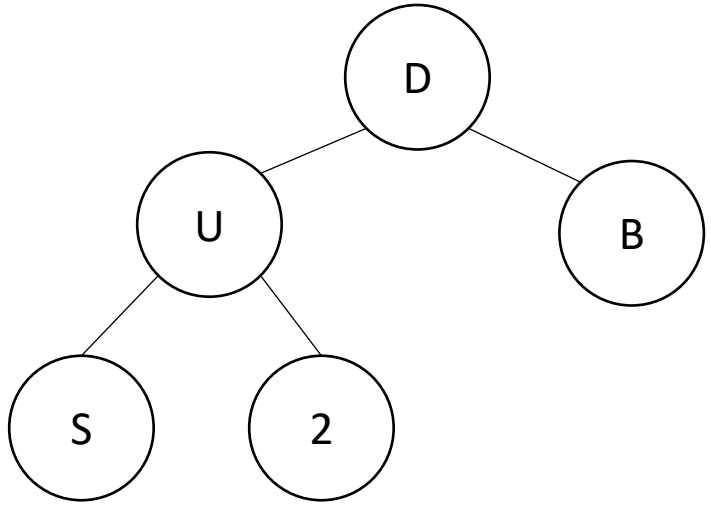
Dictionary (Map) ADT

- Contents:
  - Sets of key+value pairs
  - Keys must be comparable
- Operations:
  - insert(key, value)
    - Adds the (key,value) pair into the dictionary
    - If the key already has a value, overwrite the old value
      - Consequence: Keys cannot be repeated
  - find(key)
    - Returns the value associated with the given key
  - delete(key)
    - Remove the key (and its associated value)

Naïve attempts

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Heap	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (worst)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (expected)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

More Tree “Vocab”



- Traversal:
  - An algorithm for “visiting/processing” every node in a tree
- Pre-Order Traversal:
  - Root, Left Subtree, Right Subtree
  - D U S 2 B
- In-Order Traversal:
  - Left Subtree, Root, Right Subtree
  - S U 2 D B
- Post-Order Traversal
  - Left Subtree, Right Subtree, Root
  - S 2 U B D

Name that Traversal!

```
aOrder(root){
        if (root.left != Null){
                aOrder(root.left);
        }
        if (root.right != Null){
                aOrder(root.right);
        }
        process(root);
}
```

```
bOrder(root){
        process(root);
        if (root.left != Null){
                bOrder(root.left);
        }
        if (root.right != Null){
                bOrder(root.right);
        }
}
```

```
cOrder(root){
        if (root.left != Null){
                cOrder(root.left);
        }
        process(root)
        if (root.right != Null){
                cOrder(root.right);
        }
}
```

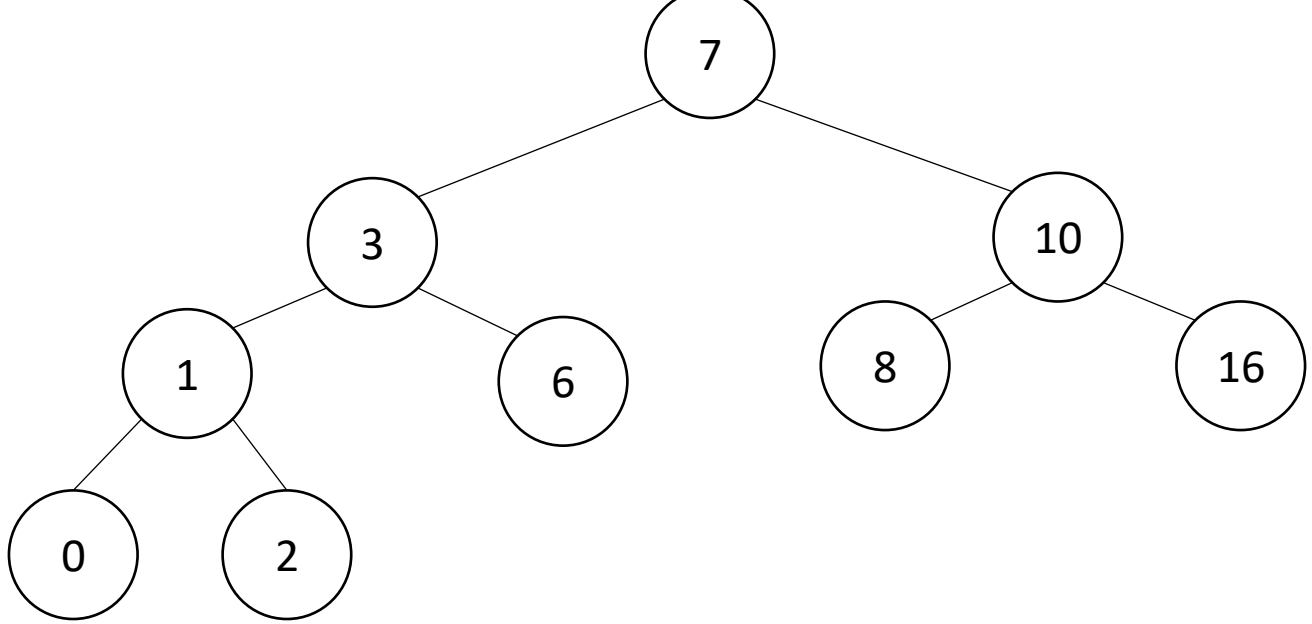
Name that Traversal! (Answers)

```
postOrder(root){
        if (root.left != Null){
                postOrder(root.left);
        }
        if (root.right != Null){
                postOrder(root.right);
        }
        process(root);
}
```

```
preOrder(root){
        process(root);
        if (root.left != Null){
                preOrder(root.left);
        }
        if (root.right != Null){
                preOrder(root.right);
        }
}
```

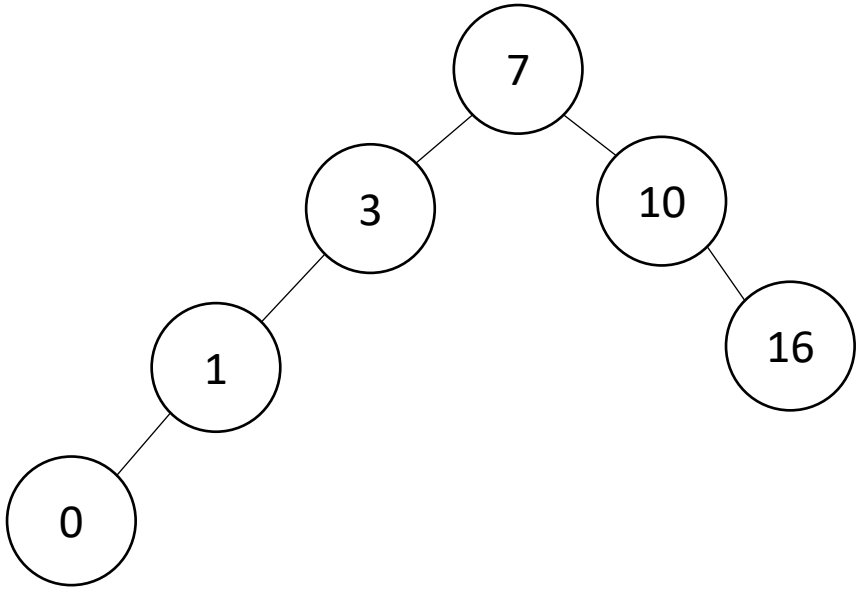
```
inOrder(root){
        if (root.left != Null){
                inOrder(root.left);
        }
        process(root)
        if (root.right != Null){
                inOrder(root.right);
        }
}
```

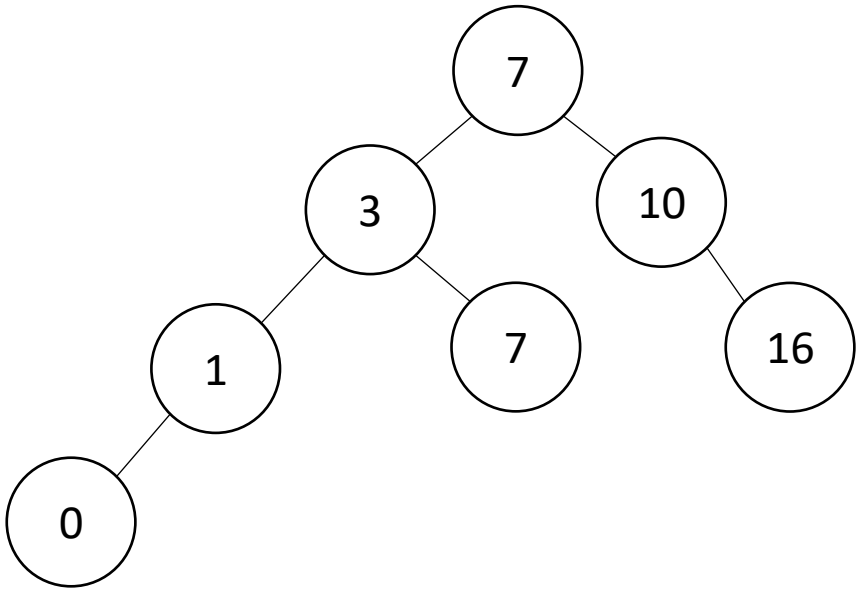
Binary Search Tree

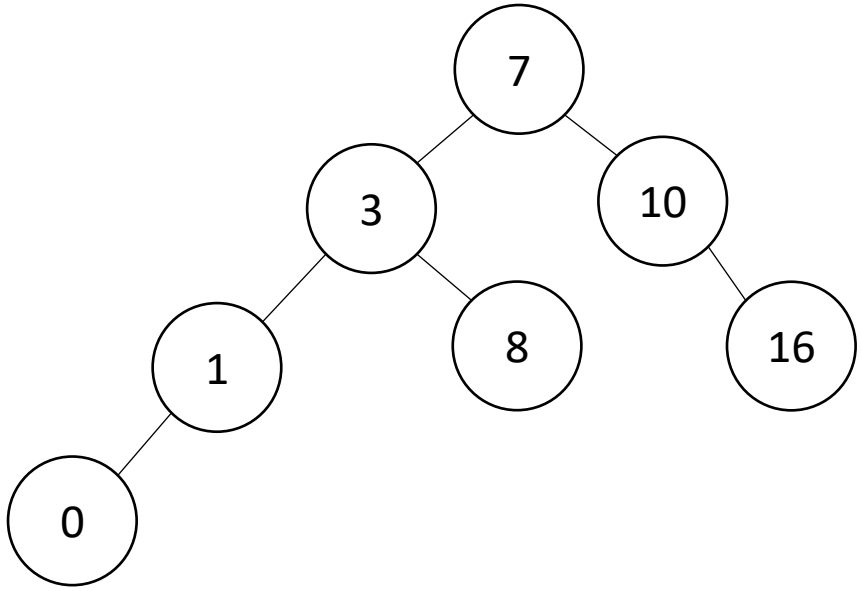


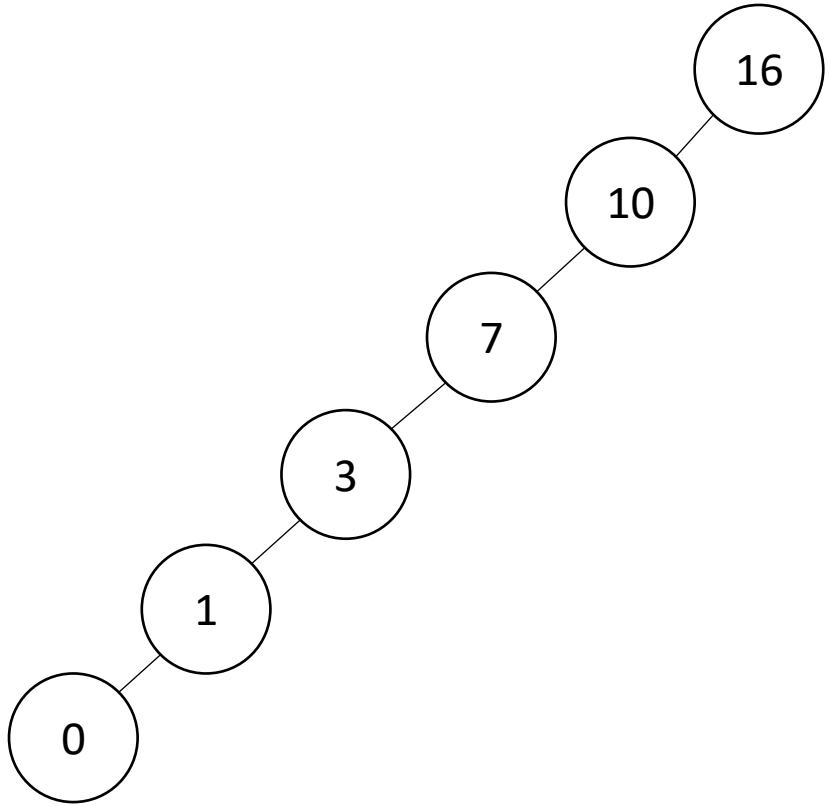
- Binary Tree
  - Definition:
    - Tree where each node has at most 2 children
- Order Property
  - All keys in the left subtree are smaller than the root
  - All keys in the right subtree are larger than the root
  - Consequence: cannot have repeated values

Are these BSTs?

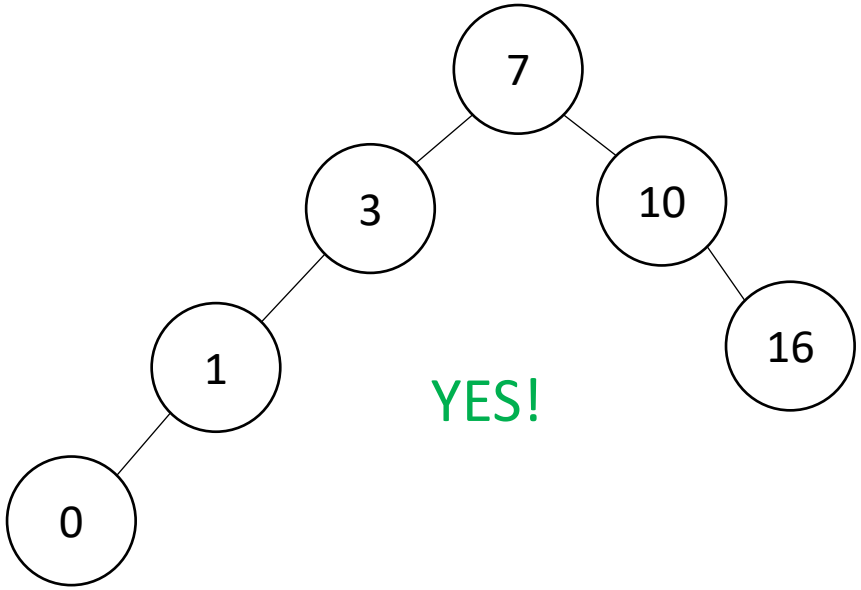




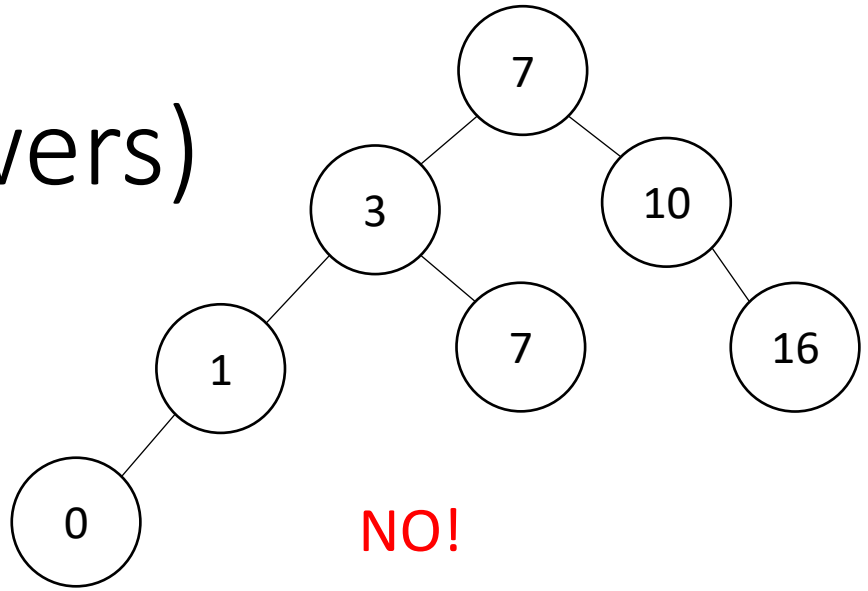




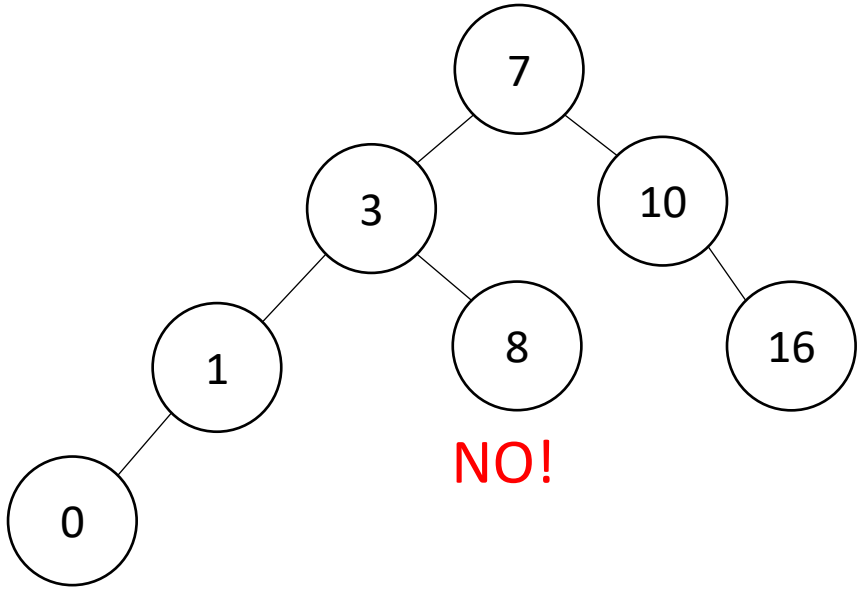
Are these BSTs? (Answers)



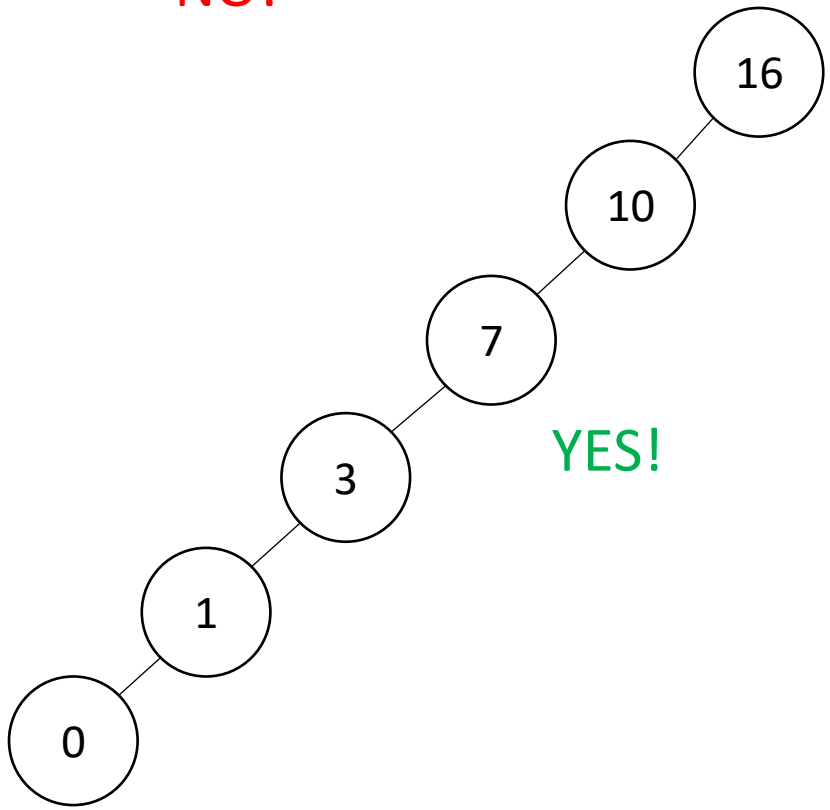
YES!



NO!



NO!



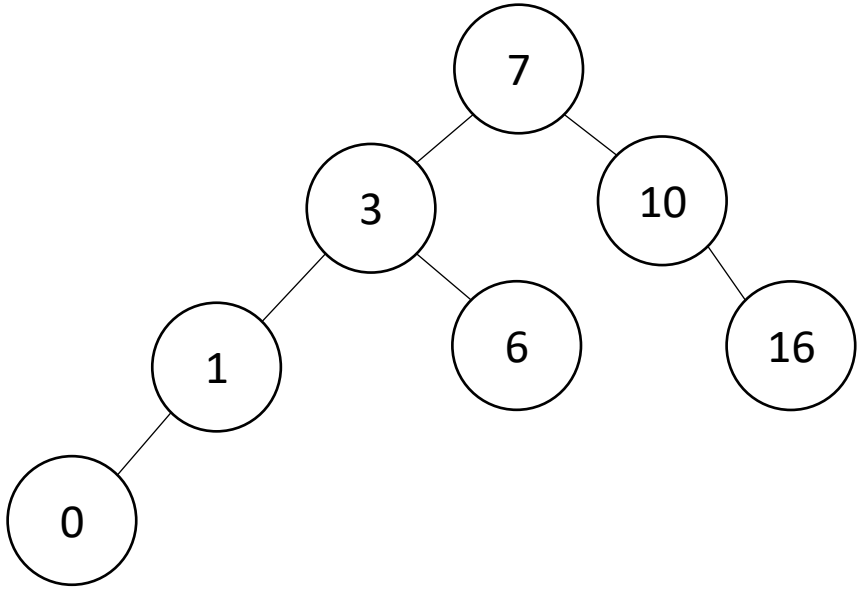
YES!

Aside: Why not use an array?

- We represented a heap using an array, finding children/parents by index
- We will represent BSTs with nodes and references. Why?
  - We might have “gaps” in our tree
  - Memory!
    - 2^n

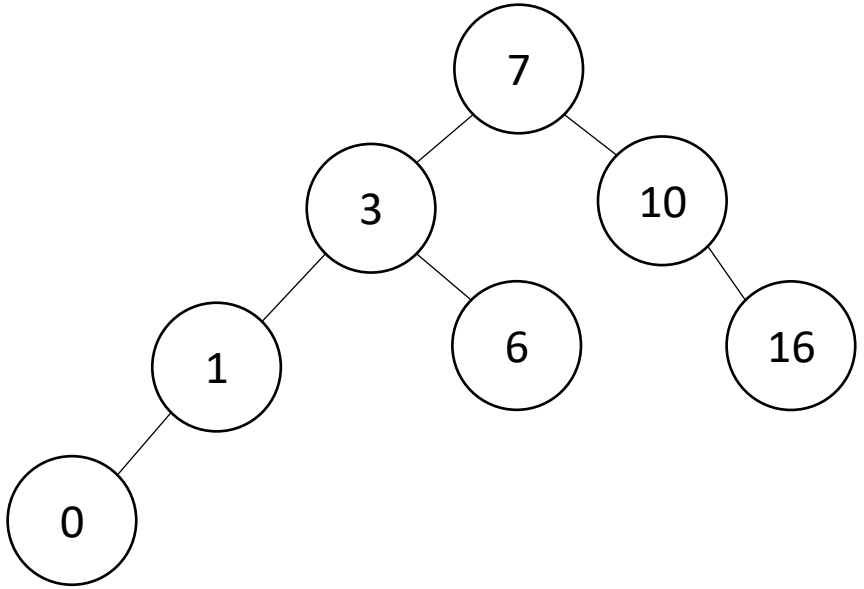
Find Operation (recursive)

```
find(key, root){
        if (root == Null){
                return Null;
        {
        if (key == root.key){
                return root.value;
        }
        if (key < root.key){
                return find(key, root.left);
        }
        if (key > root.key){
                return find(key, root.right);
        }
        return Null;
}
```



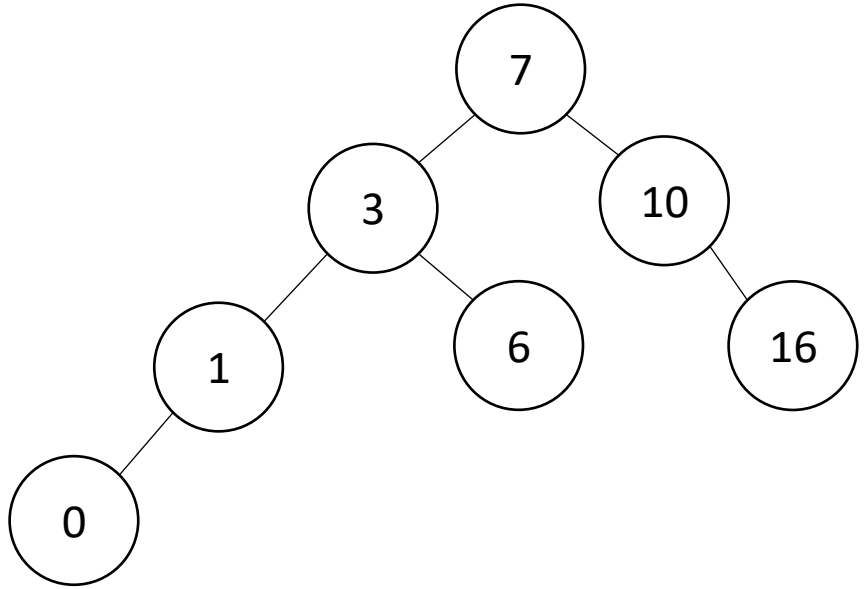
Find Operation (iterative)

```
find(key, root){
        while (root != Null && key != root.key){
                if (key < root.key){
                        root = root.left;
                }
                else if (key > root.key){
                        root = root.right;
                }
        }
        if (root == Null){
                return Null;
        }
        return root.value;
}
```



Insert Operation (recursive)

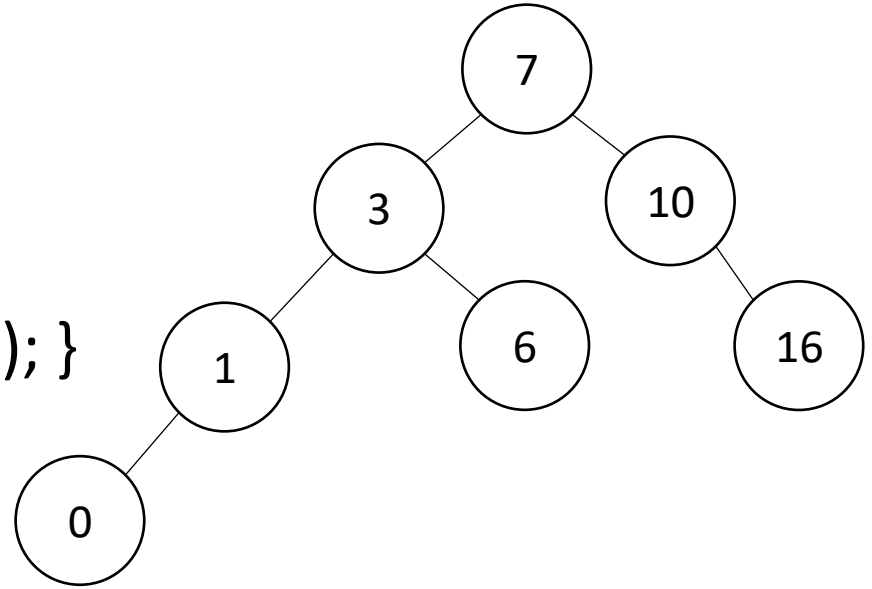
```
insert(key, value, root){
        root = insertHelper(key, value, root);
}
insertHelper(key, value, root){
        if(root == null)
                return new Node(key, value);
        if (root.key < key)
                root.right = insertHelper(key, value, root.right);
        else
                root.left = insertHelper(key, value, root.left);
        return root;
}
```



Note: Insert happens only at the leaves!

Insert Operation (iterative)

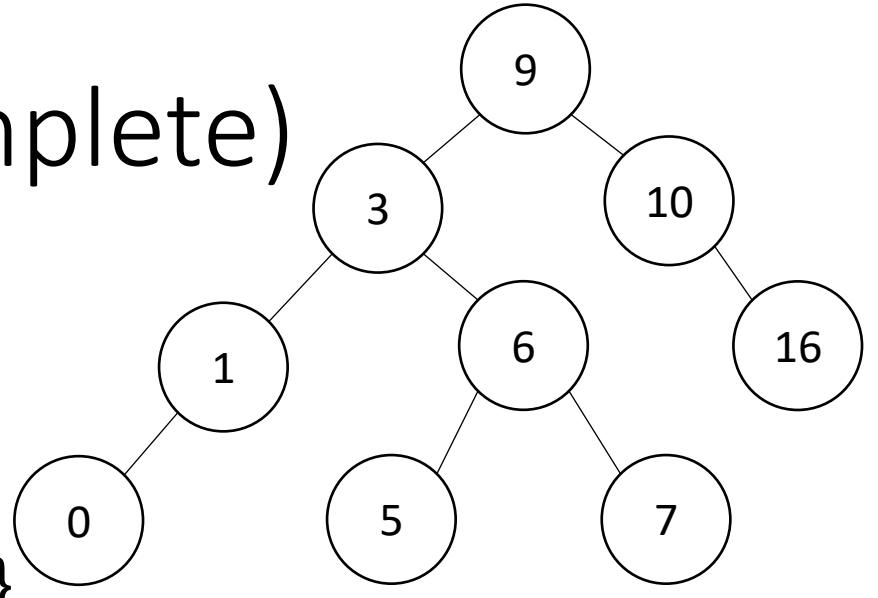
```
insert(key, value, root){
        if (root == Null){ this.root = new Node(key, value); }
        parent = Null;
        while (root != Null && key != root.key){
                parent = root;
                if (key < root.key){ root = root.left; }
                else if (key > root.key){ root = root.right; }
        }
        if (root != Null){ root.value = value; }
        else if (key < parent.key){ parent.left = new Node(key, value); }
        else{ parent.right = new Node (key, value); }
}
```



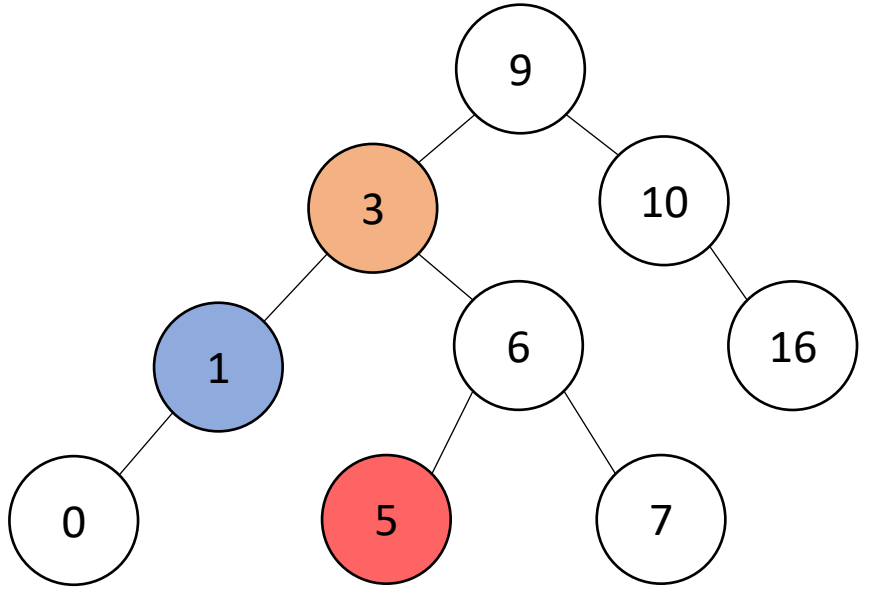
Note: Insert happens only at the leaves!

Delete Operation (iterative, incomplete)

```
delete(key, root){
        while (root != Null && key != root.key){
                if (key < root.key){ root = root.left; }
                else if (key > root.key){ root = root.right; }
        }
        if (root == Null){ return; }
        // Now root is the node to delete, what happens next?
}
```



Delete – 3 Cases

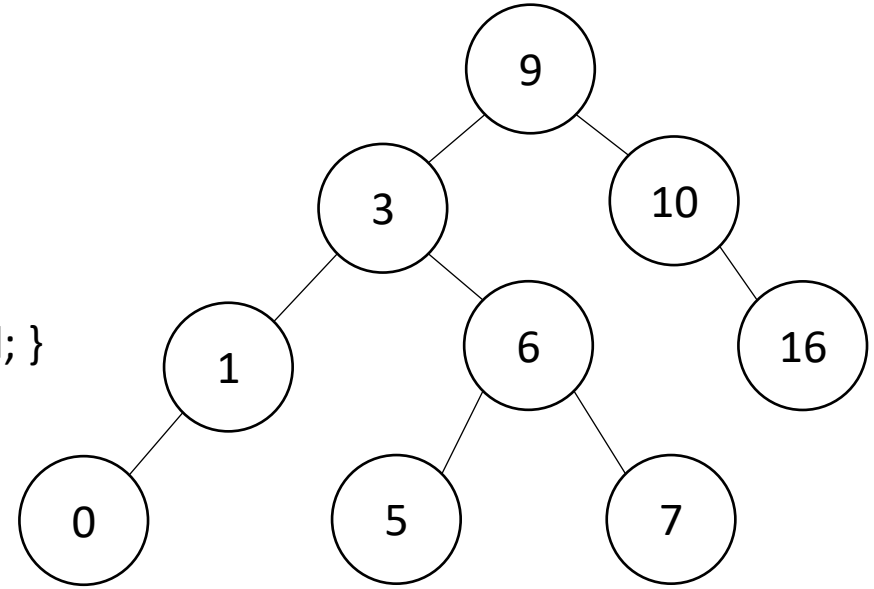


- 0 Children (i.e. it's a leaf)
- 1 Child
  - Replace the deleted node with its child
- 2 Children
  - Replace the deleted with the largest node to its left or else the smallest node to its right

Finding the Max and Min

- Max of a BST:
  - Right-most Thing

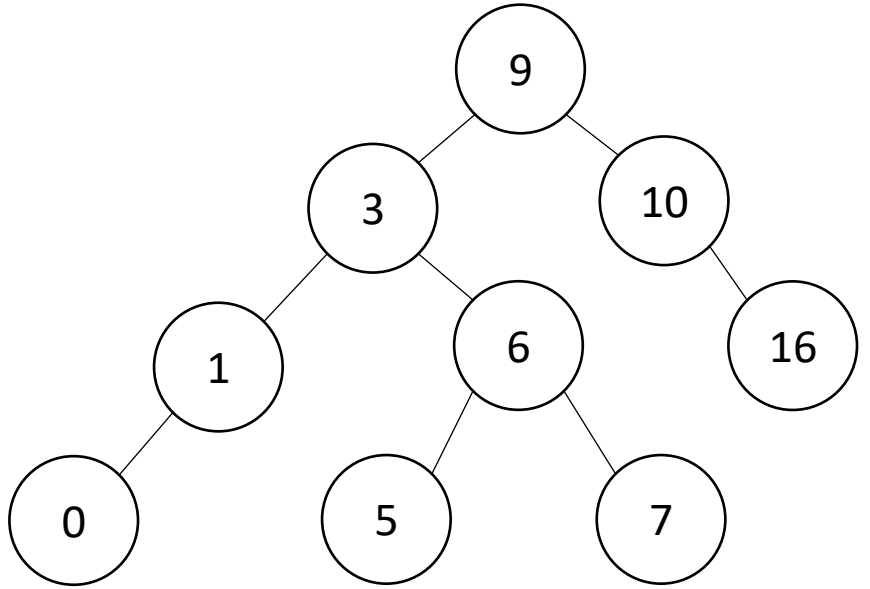
```
maxNode(root){
        if (root == Null){ return Null; }
        while (root.right != Null){
                root = root.right;
        }
        return root;
}
```



- Min of a BST:
  - Left-most Thing

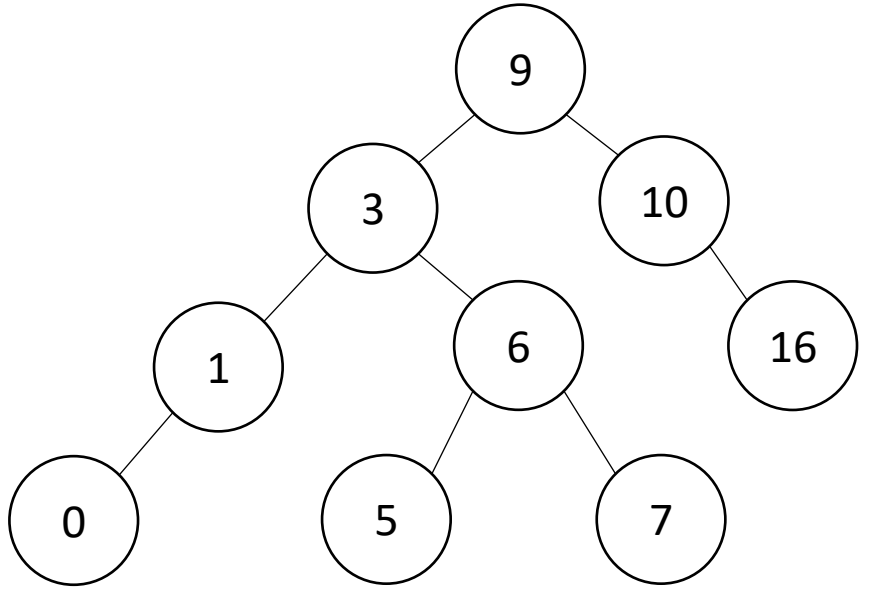
```
minNode(root){
        if (root == Null){ return Null; }
        while (root.left != Null){
                root = root.left;
        }
        return root;
}
```

Delete Operation (iterative)




```
delete(key, root){
        while (root != Null && key != root.key){
                if (key < root.key){ root = root.left;  }
                else if (key > root.key){ root = root.right; }
        }
        if (root == Null){ return; }
        if (root has no children){
                make parent point to Null Instead;
        }
        if (root has one child){
                make parent point to that child instead;
        }
        if (root has two children){
                make parent point to either the max from the left or min from the right
        }
}
```

Delete Operation (recursive)




```
delete(key, root){
        if (root == Null){ return; } // key not present
        if (root.key == key){
                if (root has no children) { return Null; }
                if (root has one child) { return that child; }
                if (root has two children) {return removeMax(root.left);}
        }
        if (root.key < key) { root.right = delete(key, root.right); }
        else { root.left = delete(key, root.left); }
}
```

Worst Case Analysis

- For each of Find, insert, Delete:
  - Worst case running time matches height of the tree
- What is the maximum height of a BST with n nodes?
  - $\Theta(n)$

Improving the worst case

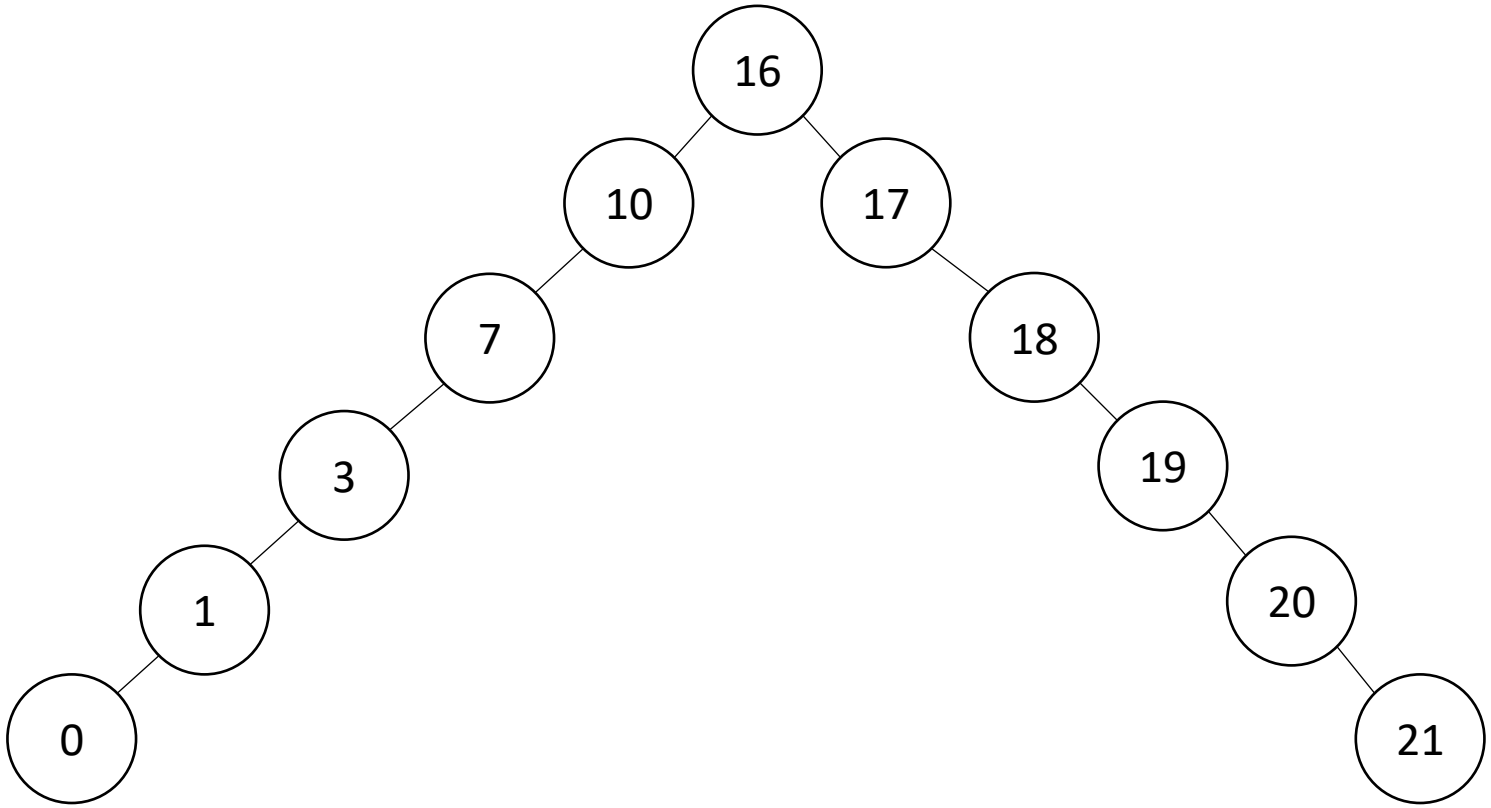
- How can we get a better worst case running time?
  - Add rules about the shape of our BST
- AVL Tree
  - A BST with some shape rules
    - Algorithms need to change to accommodate those

“Balanced” Binary Search Trees

- We get better running times by having “shorter” trees
- Trees get tall due to them being “sparse” (many one-child nodes)
- Idea: modify how we insert/delete to keep the tree more “full”
  - Encourage Branches!

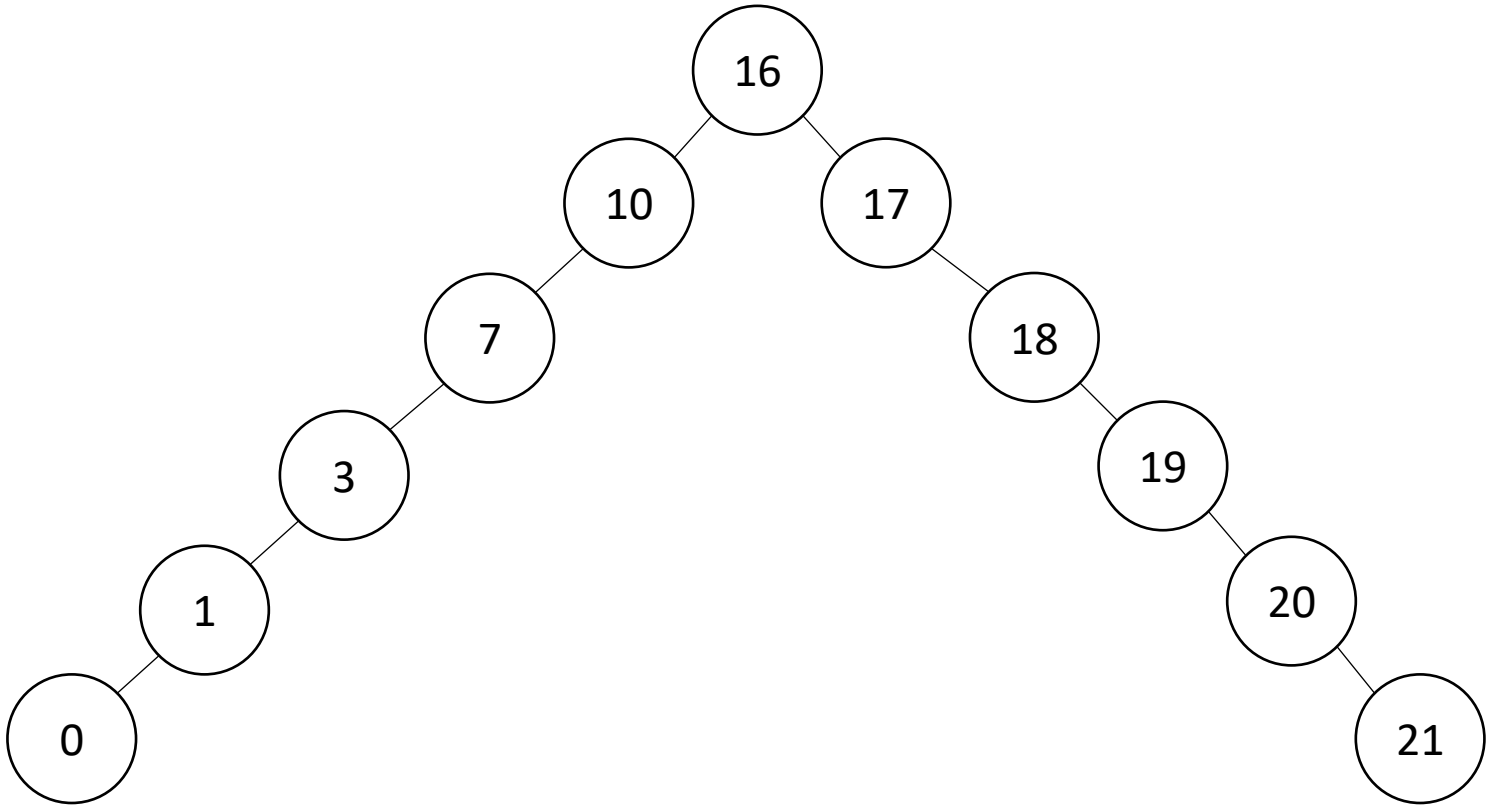
Idea 1: Both Subtrees of Root have same # Nodes

Idea 1: Both Subtrees of Root have same # Nodes - Issue



Idea 2: Both Subtrees of Root have same height

Idea 2: Both Subtrees of Root have same height - Issue



Idea 3: Both Subtrees of every Node have same # Nodes

Idea 3: Both Subtrees of every Node have same # Nodes - Issue

Not all tree sizes are possible!

For example, cannot have a tree of size 3.

Idea 4: Both Subtrees of every Node have same height

Idea 4: Both Subtrees of every Node have same height - Issue

Not all tree sizes are possible!

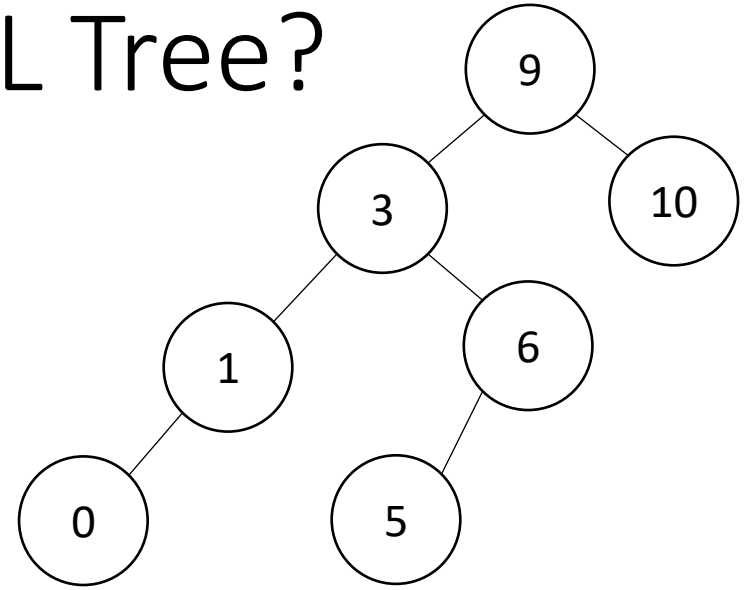
For example, cannot have a tree of size 3.

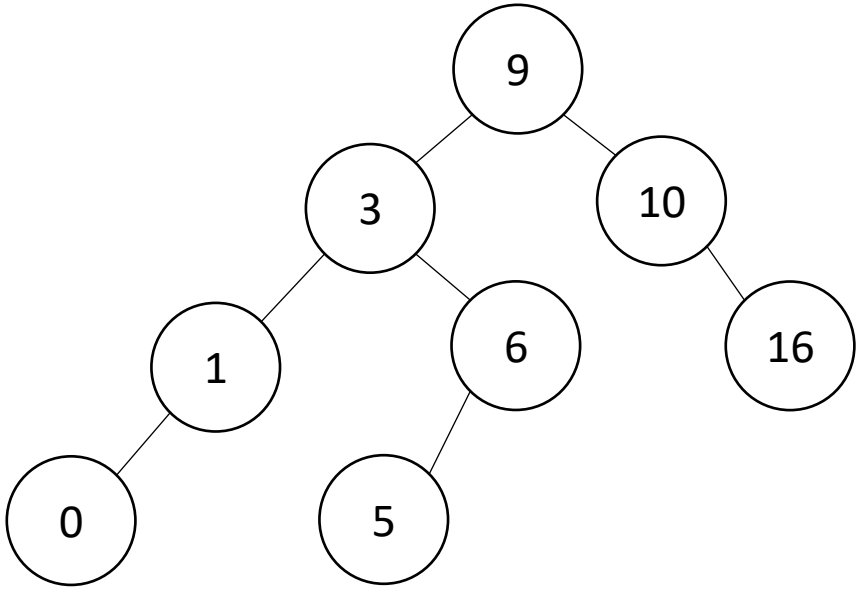
AVL Tree

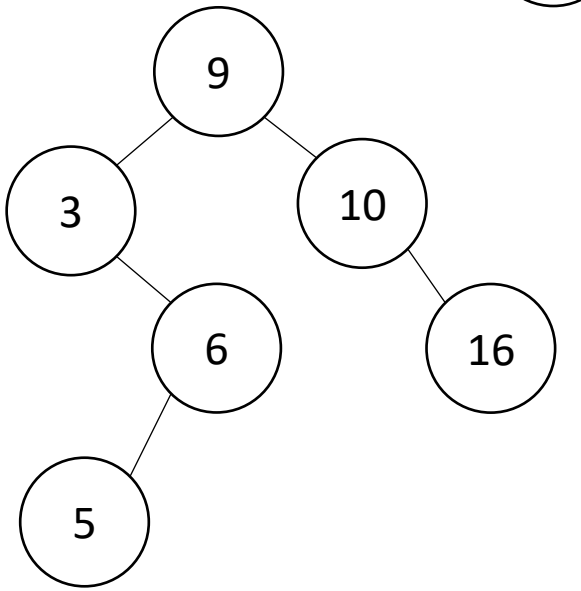
- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
  - height of left subtree and height of right subtree off by at most 1
  - Not too weak (ensures trees are short)
  - Not too strong (works for any number of nodes)

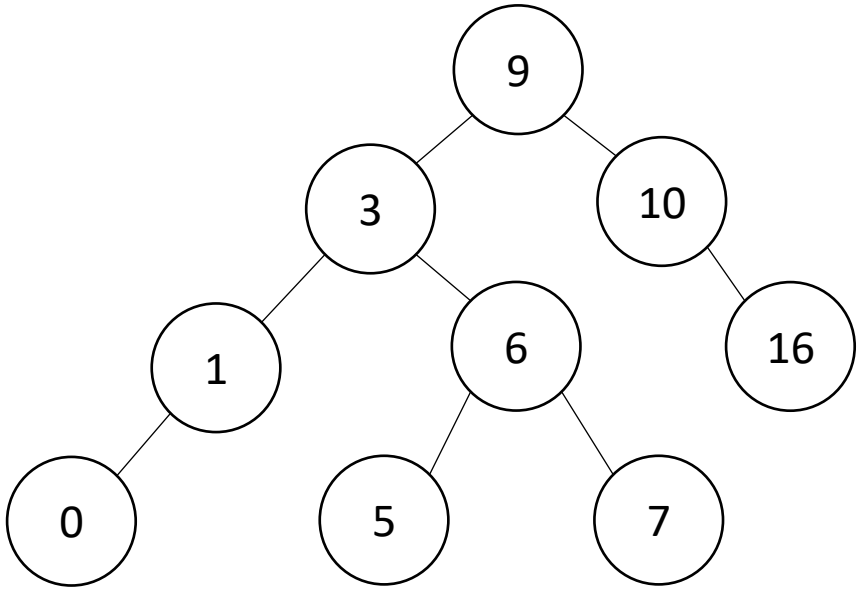
- Idea of AVL Tree:
  - When you insert/delete nodes, if tree is “out of balance” then modify the tree
  - Modification = “rotation”

Is it an AVL Tree?

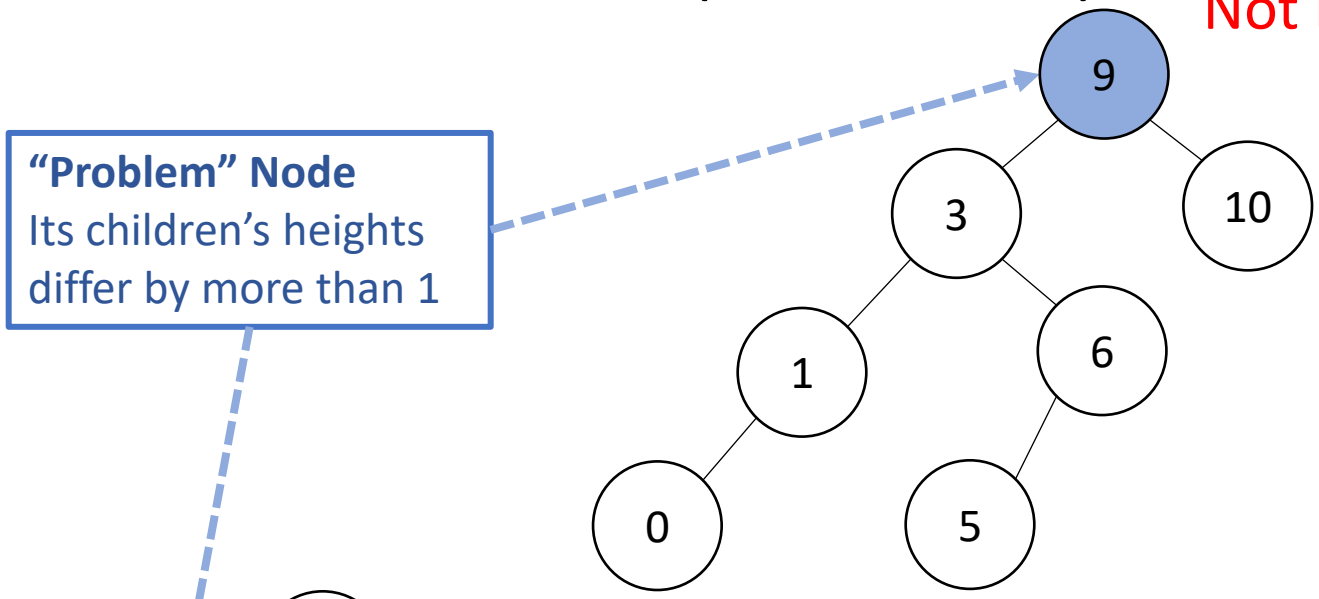




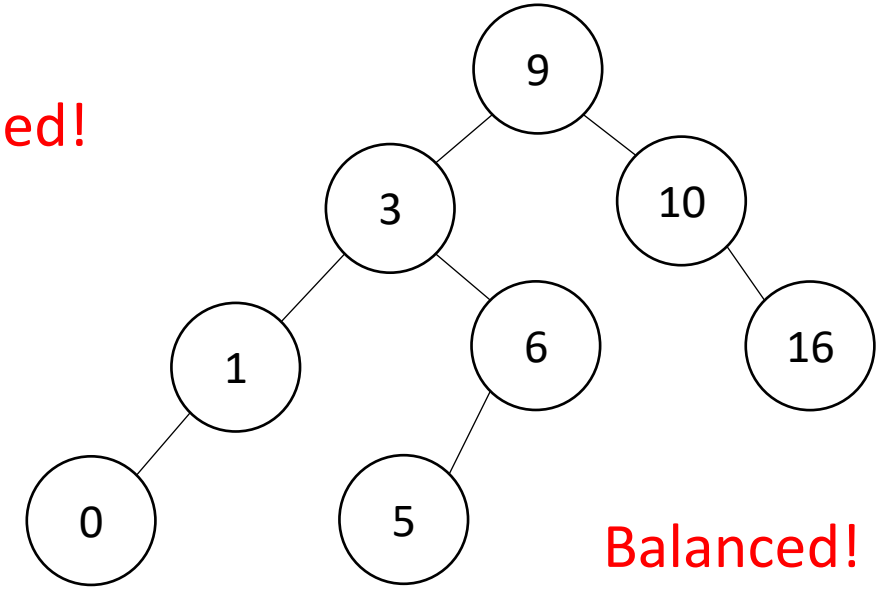




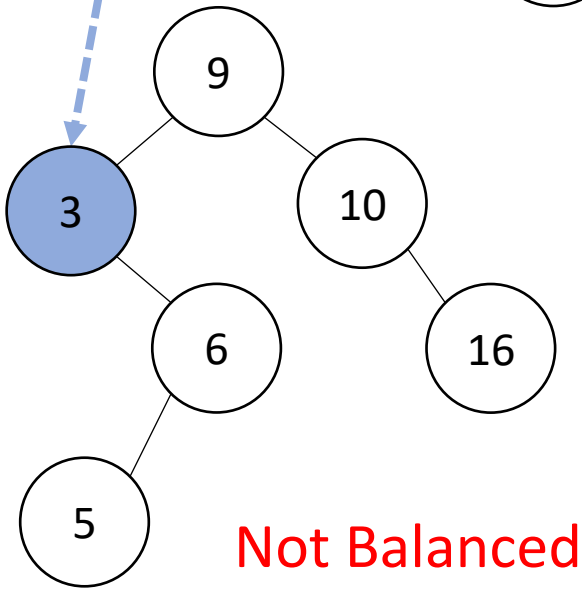
Is it an AVL Tree? (Answers)



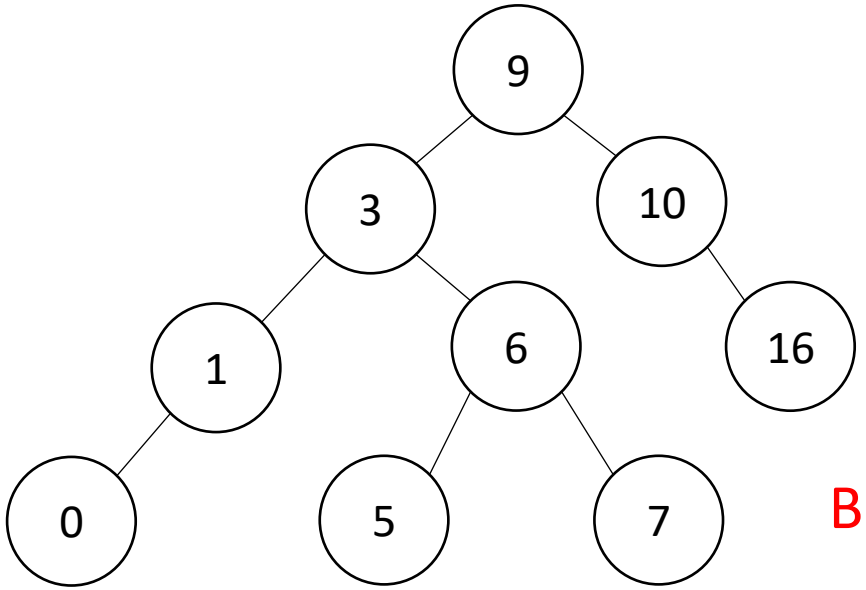
Not Balanced!



Balanced!



Not Balanced!



Balanced!