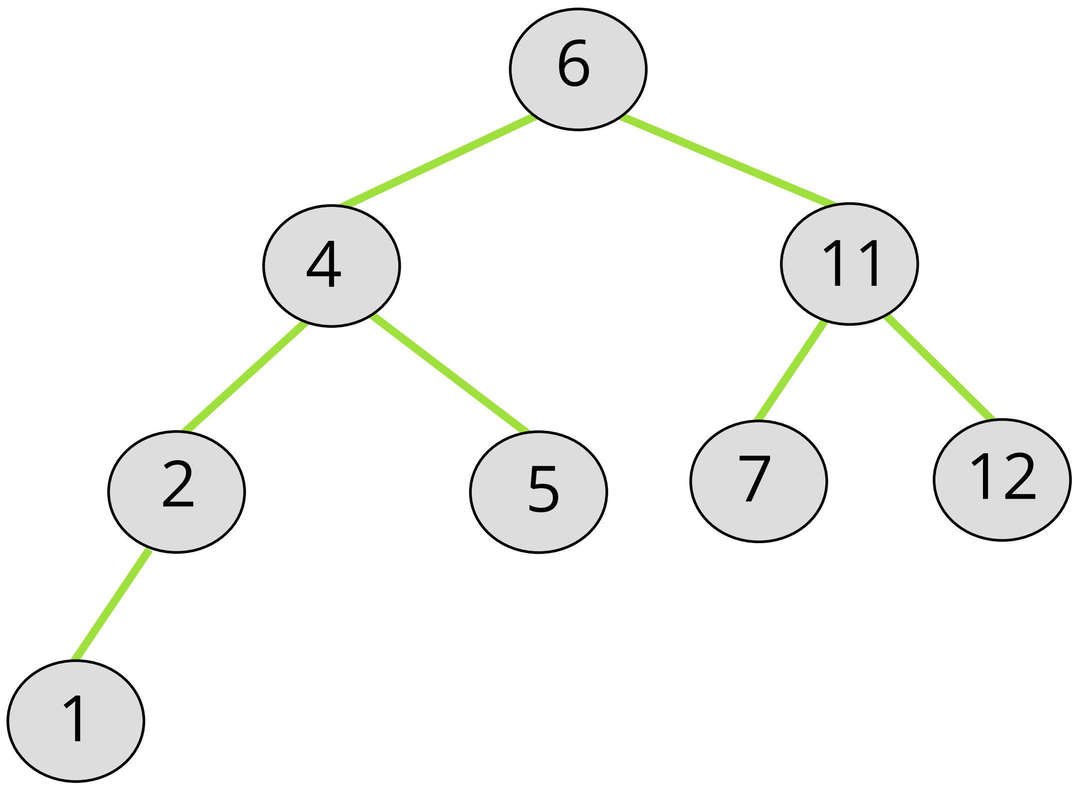
COSC 222 Data Structure

Trees - part 2
- More tree terminology
- Iterative traversals
- Expression trees
- Binary search trees

Binary Tree

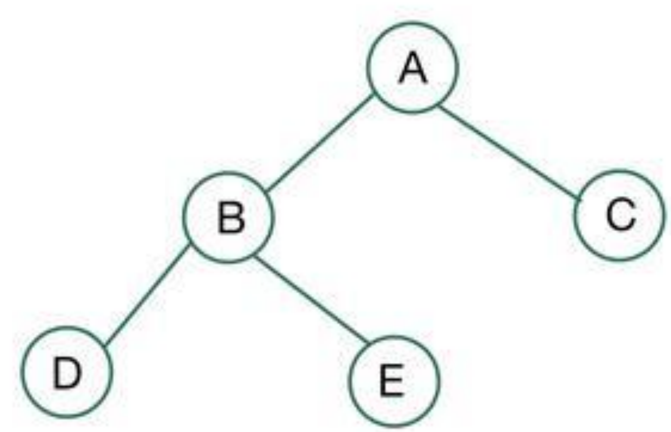
- A binary tree is a tree in which each node has at most two children
- Each child is either the **left child** or the **right child** of its parent



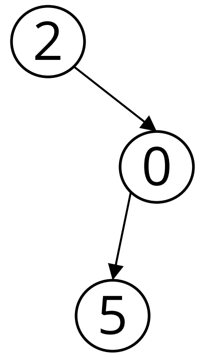
Special kinds of binary trees

Full binary tree

A binary tree where every node has either **0 or 2** children.



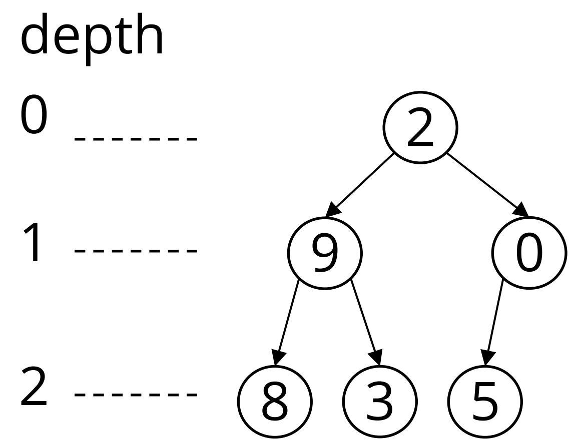
Special kinds of binary trees



Height 2,
minimum number of nodes

Max # of nodes at depth d: 2^d

If height of tree is h:
- min # of nodes: h + 1
- max #of nodes:
  $2^0 + \ldots + 2^h = 2^{h+1} - 1$



Height 2,
maximum number of nodes

Complete binary tree

- Every level, except the last, is completely filled.
- Nodes on bottom level are as far left as possible (no holes.)

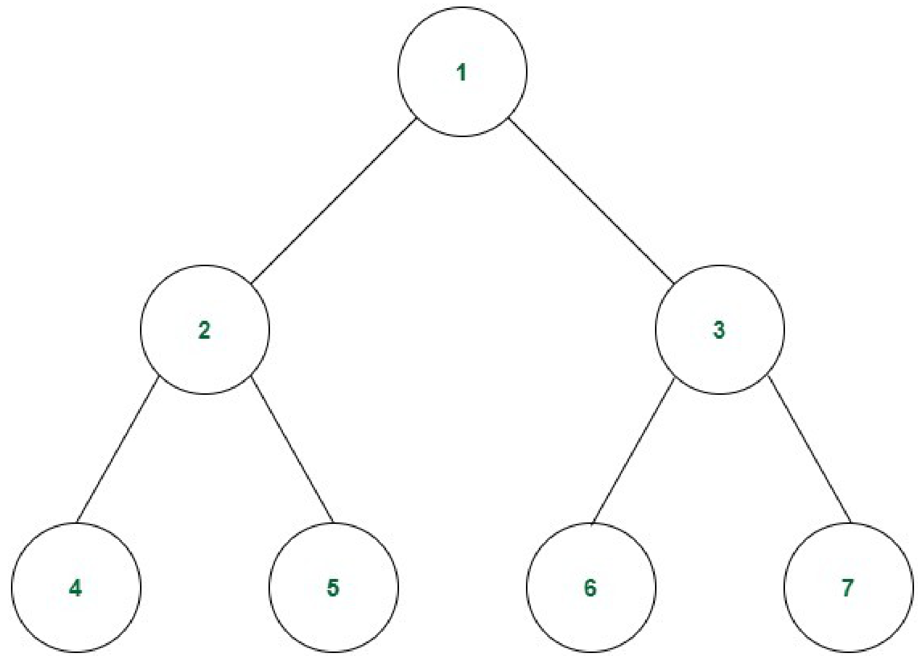
Special kinds of binary trees

Perfect binary tree

A binary tree where:

- every internal node has **2** children *and*
- all leaves have the same depth.

A perfect tree is a full tree.



Iterative Binary Tree Traversals

- In recursive tree traversals, the Java execution stack keeps track of where we are in the tree (by means of the activation records for each call)
- In iterative traversals, the programmer needs to keep track!
  - An iterative traversal uses a container to store references to nodes not yet visited
  - Order of visiting will depend on the type of container being used (stack, queue, etc.)

An Iterative Traversal Algorithm

```
// Assumption: the tree is not empty

Create an empty container to hold references to nodes
       yet to be  visited.

Put reference to the root node in the container.

While the container is not empty {

    Remove a reference x from the container.

    Visit the node x points to.

    Put references to non-empty children of x in the
container.

    }

}
```

Iterative Binary Tree Traversals

- Container is a stack: if we push the right successor of a node before the left successor, we get preorder traversal
- Container is a queue: if we enqueue the left successor before the right, we get a level order traversal
- Exercise: Trace the iterative tree traversal algorithm using as containers
  - a stack
  - a queue

Traversal Analysis

- Consider a binary tree with n nodes
- How many recursive calls are there at most?
  - For each node, 2 recursive calls at most
  - So, 2*n recursive calls at most
- So, a traversal is O(n)

Operations on a Binary Tree

- What might we want to do with a binary tree?
  - Add an element (but where?)
  - Remove an element (but from where?)
  - Is the tree empty?
  - Get size of the tree (i.e. how many elements)
  - Traverse the tree (in preorder, inorder, postorder, level order)

Discussion

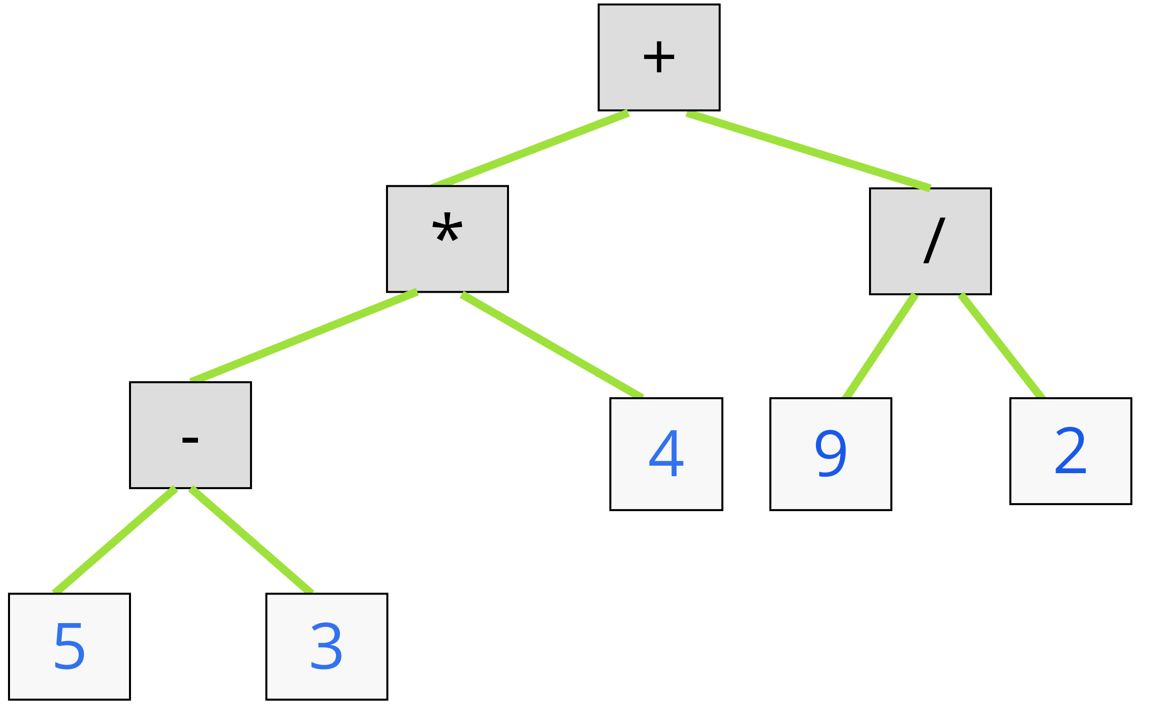
- It is difficult to have a general add operation, until we know the purpose of the tree (we will discuss binary search trees later)
  - We could add “randomly”: go either right or left, and add at the first available spot

Discussion

- Similarly, where would a general remove operation **remove** from?
  - We could arbitrarily choose to remove, say, the leftmost leaf
  - If random choice, what would happen to the children and descendants of the element that was removed? What does the parent of the removed element now point to?
  - What if the removed element is the root?

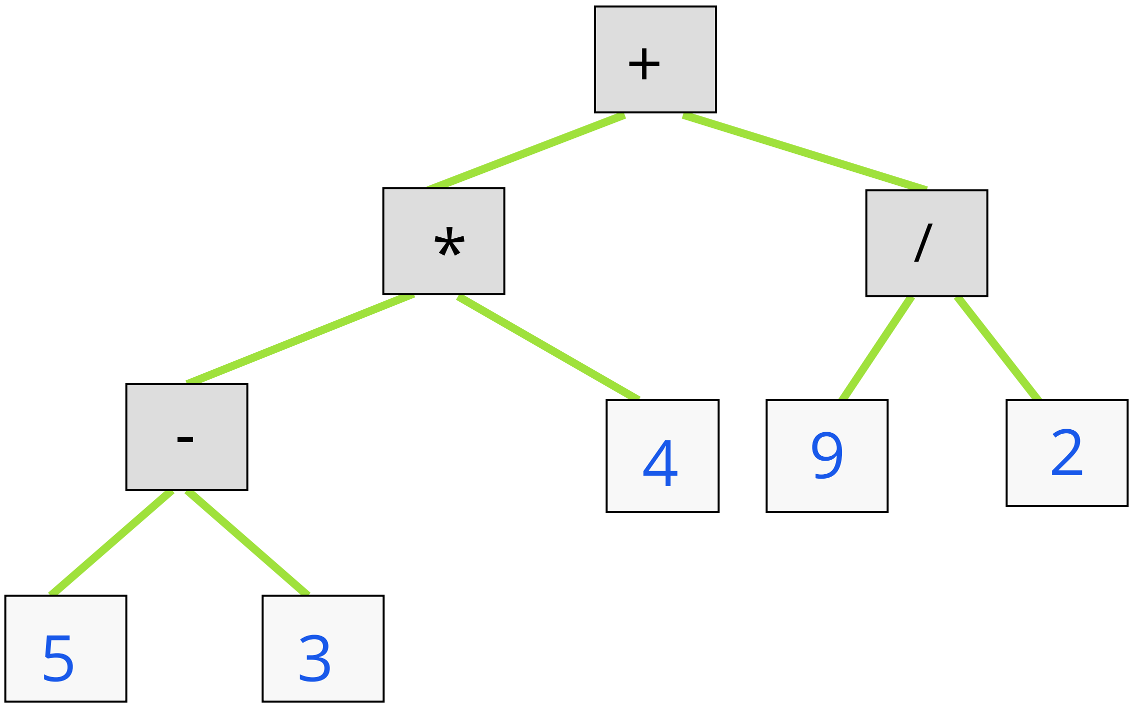
Using Binary Trees: Expression Trees

- Programs that manipulate or evaluate arithmetic expressions can use binary trees to hold the expressions
- An **expression tree** is a binary tree that represents an arithmetic expression composed of binary operators.
  - Example: (5 – 3) * 4 + 9 / 2 ⇨ (((5 – 3) * 4) + (9 / 2))



Expression Trees

- In an expression tree,
  - Constant **operands** are stored in the leaves.
  - **Operators** are stored in the interior nodes (non-leaf node = interior node).

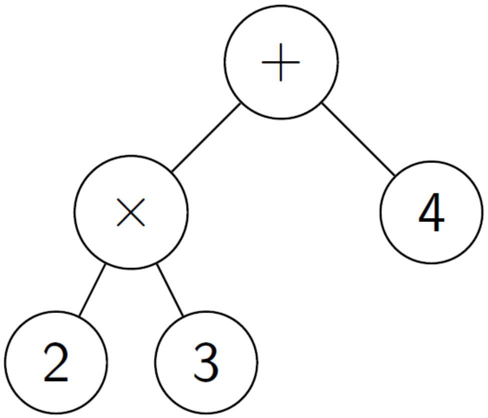


(((5 – 3) * 4) + (9 / 2))

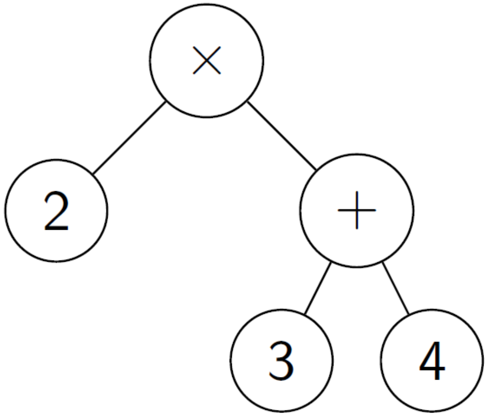
Expression Trees

- For any arithmetic expression, there is only one corresponding expression tree.

Example: Here is the expression tree corresponding to $2 \times 3 + 4$:

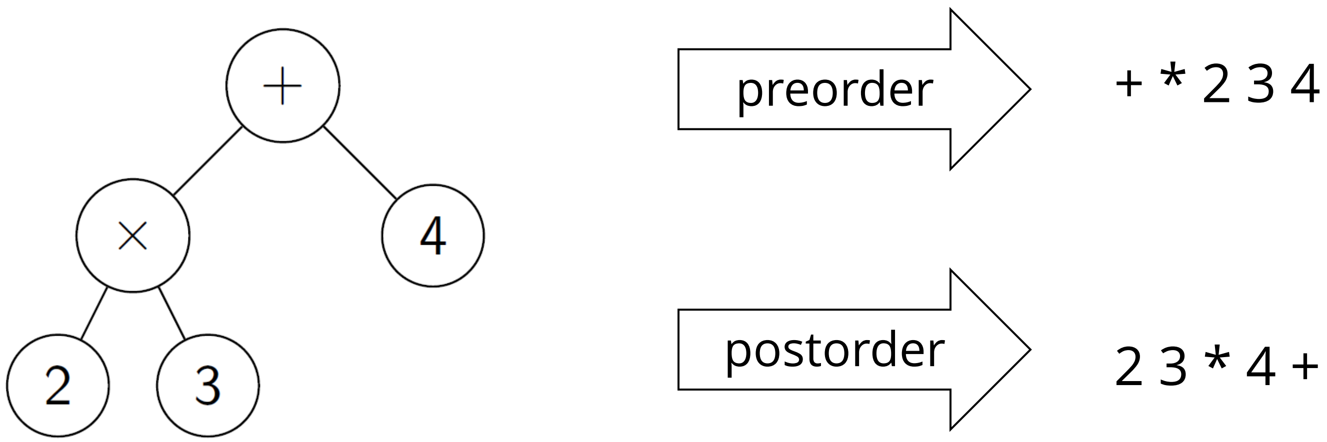


. . . and here is the expression tree corresponding to $2 \times (3 + 4)$:



Expression Trees

- The output of a **preorder** traversal of an expression tree is the corresponding **prefix** expression!
- The output of an **inorder** traversal of an expression tree is the corresponding **infix** expression!
- The output of a **postorder** traversal of an expression tree is the corresponding **postfix** expression!

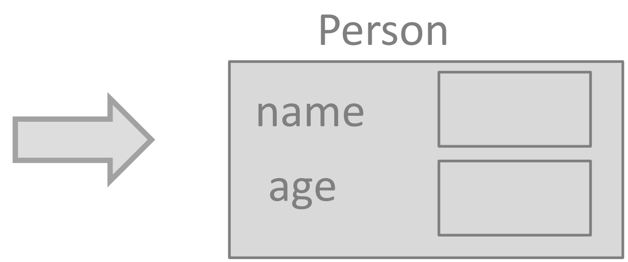


Prefix, infix and postfix notations: http://www.cs.man.ac.uk/~pjj/cs212/fix.html

Creating objects

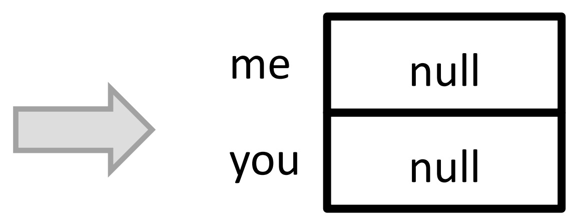
- A class file creates only a “template” for what an object will look like:

```
public class Person {
        //Instance variables
        public String name;
        public int age;
      }
```



- All objects are accessed only via references (pointers, addresses) to the object.
- The special reference null means that there is no object being referred to.

```
Person me, you;
```



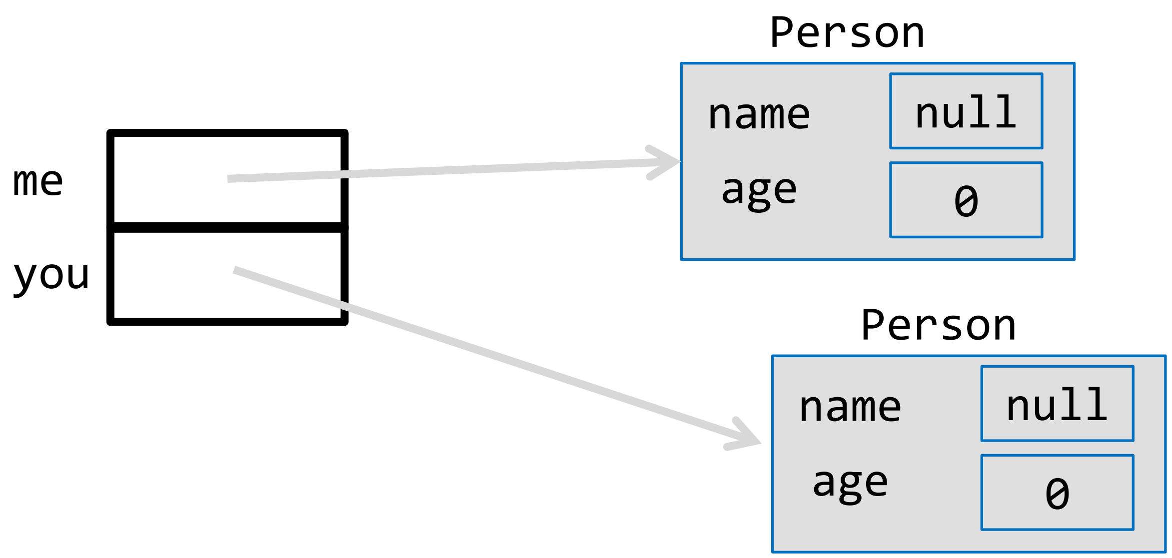
Creating objects

- To create an instance (an actual object), the basic syntax is:

```
new Person( )
```

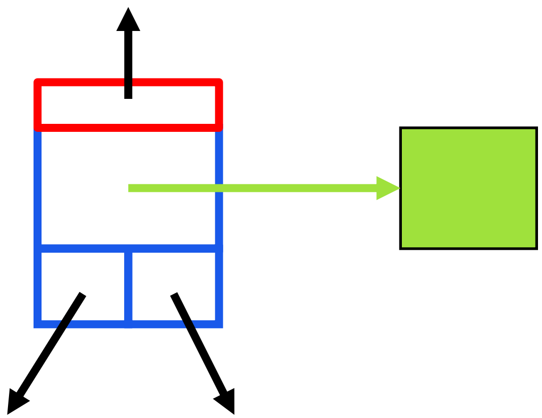
  - This gives a reference to a new object instance, which you usually want to save someplace:

```
me = new Person();
you = new Person();
```



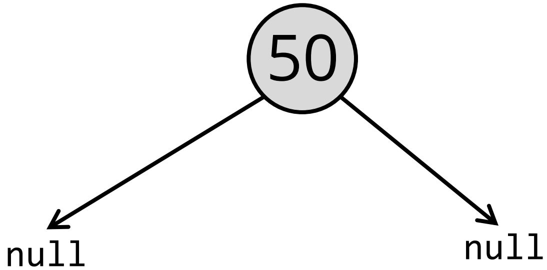
Linked Binary Tree Implementation

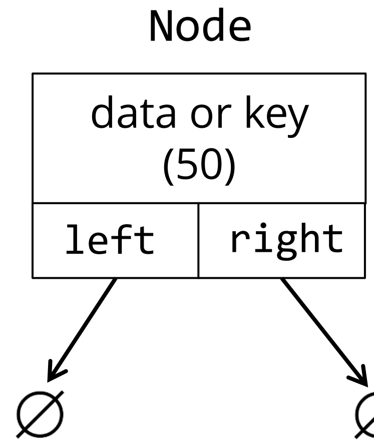
- A binary tree node will contain
  - a reference to a data element
  - references to its left and right children and parent

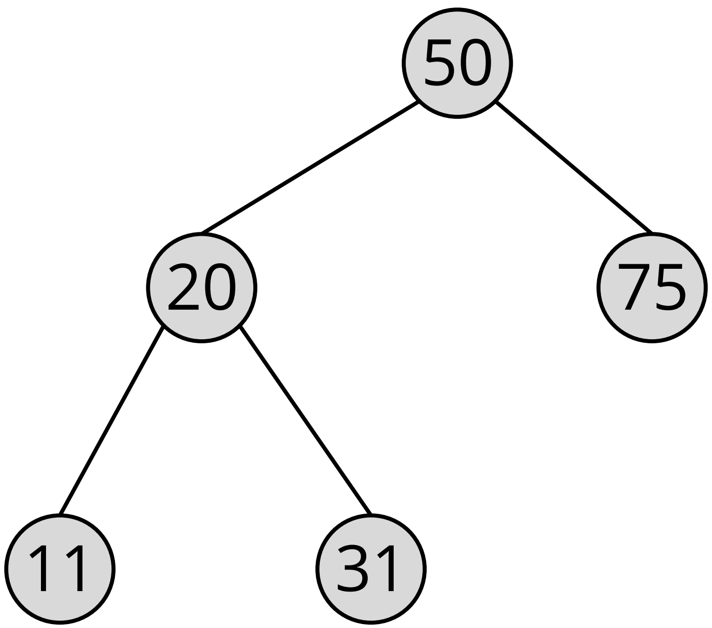


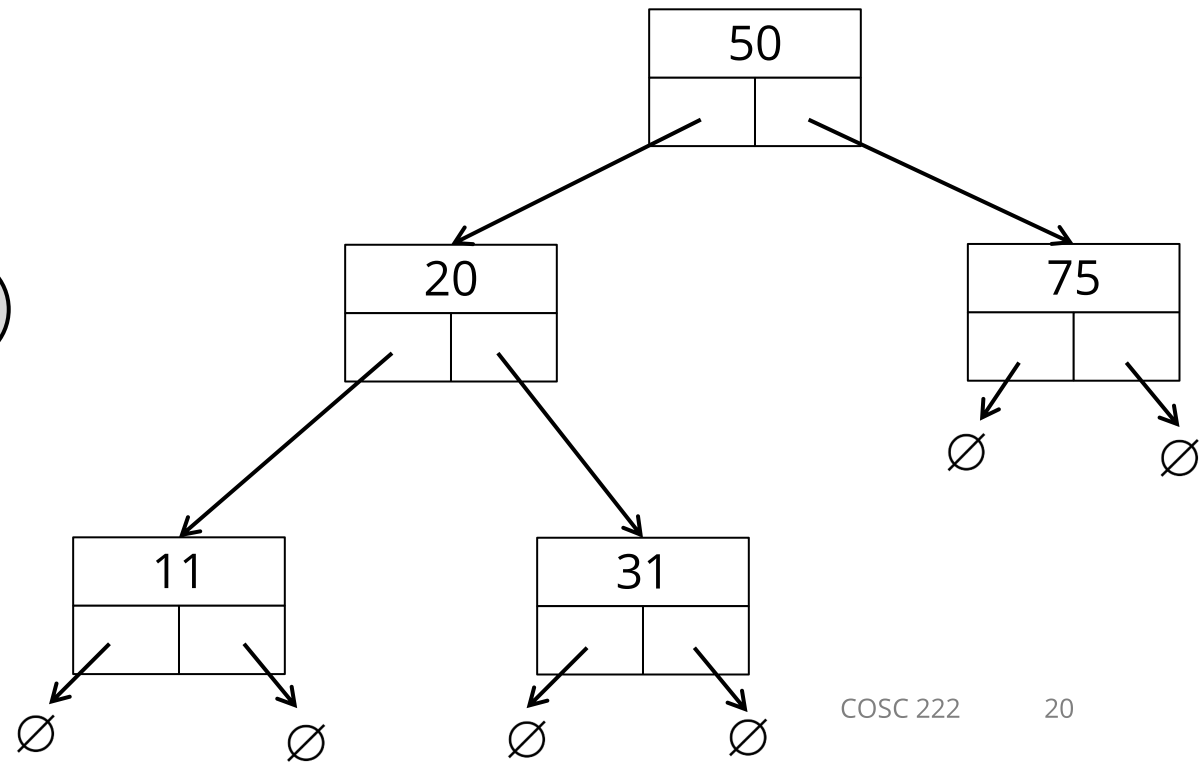
left and right children are binary tree nodes themselves

Binary Trees







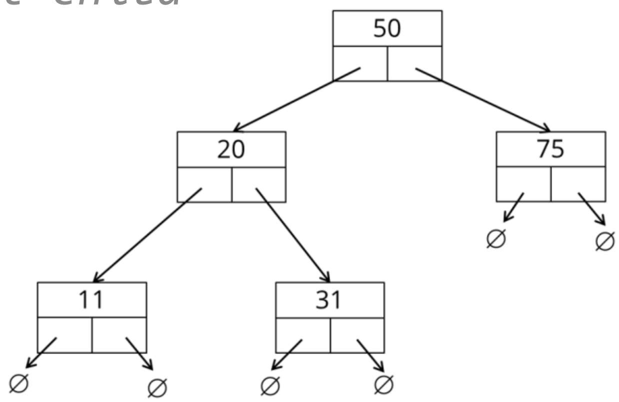


Basic Implementation of Binary Trees

```
public class Node {
    protected int key; // node's data/key
    protected Node left; // pointer to left child
    protected Node right; // pointer to right child

    public Node(int newKey){
        this.key = newKey;
        left = null;
        right = null;
    }

    public Node(int newKey, Node newLeft, Node newRight){
        key = newKey;
        left = newLeft;
        right = newRight;
    }
}
```



Class Binary Tree

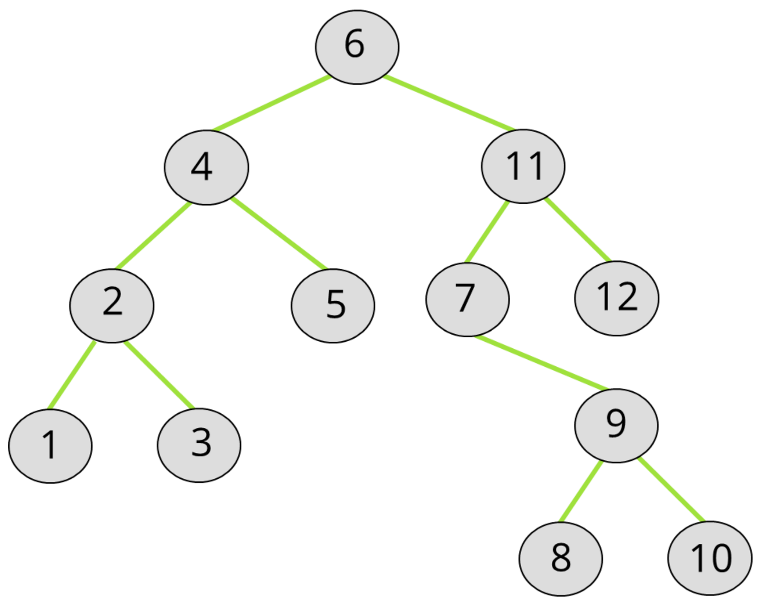
```
public class BST {
    public Node root;

    //========================================
    //  Creates an empty binary search tree.
    //========================================
    public BST(){
        root = null;
    }


…
}
```

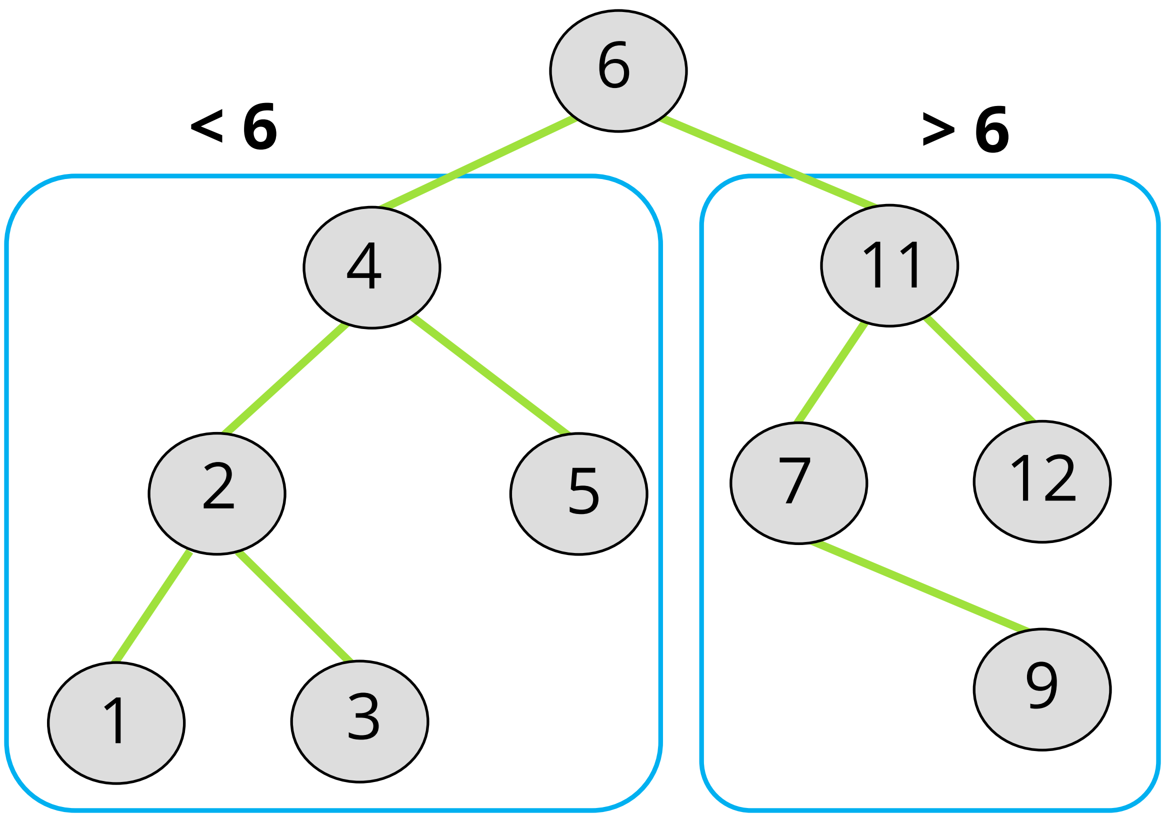
Operations on a Binary Tree

- What might we want to do with a binary tree?
  - Add an element (but where?)
  - Remove an element (but from where?)
  - Search an element
  - Is the tree empty?
  - Get size of the tree (i.e. how many elements)
  - Traverse the tree (in preorder, inorder, postorder, level order)



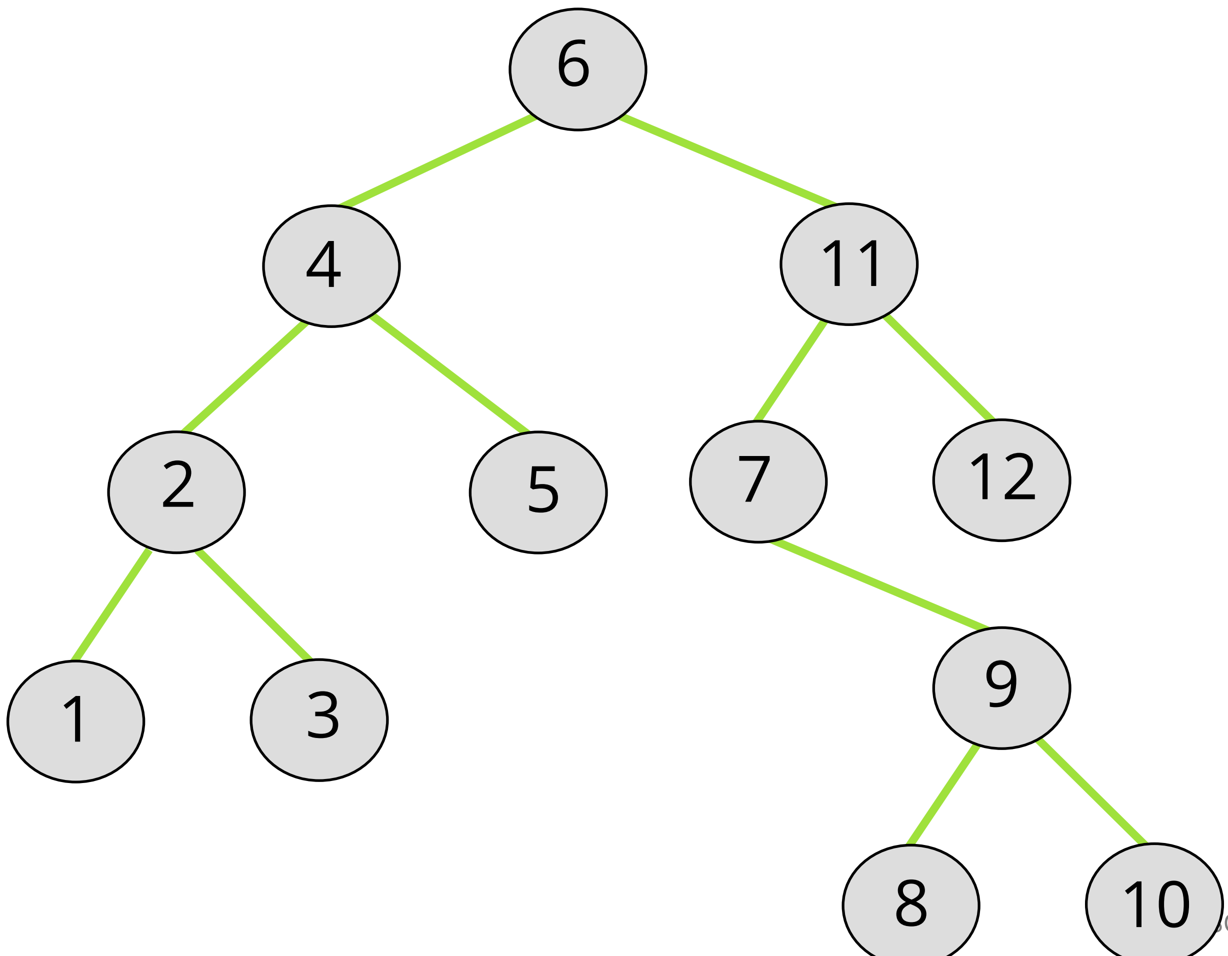
Binary Search Tree (BST)

- A **binary search tree** is a binary tree:
  - All nodes in the left subtree have values that are less than the value in that node, and
  - All values in the right subtree are greater (assume no duplicates)



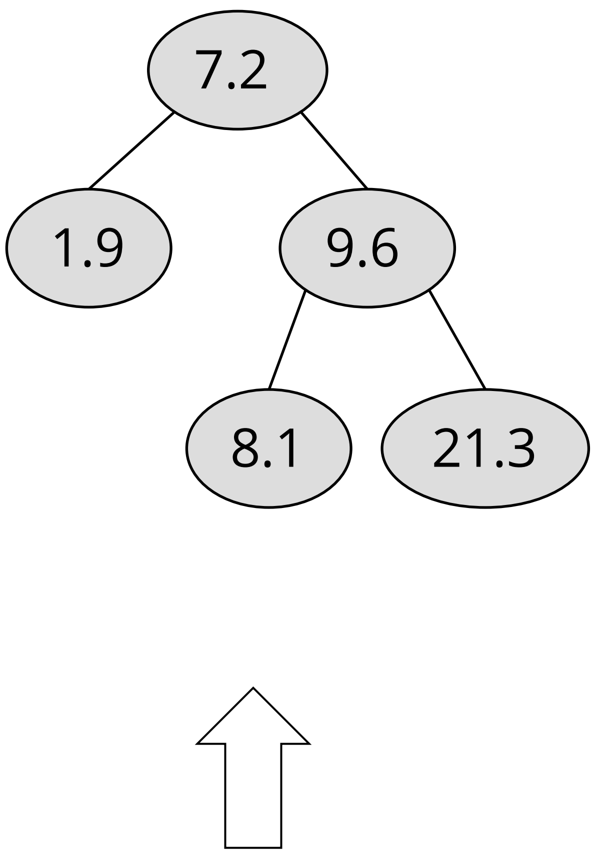
Binary Search Tree

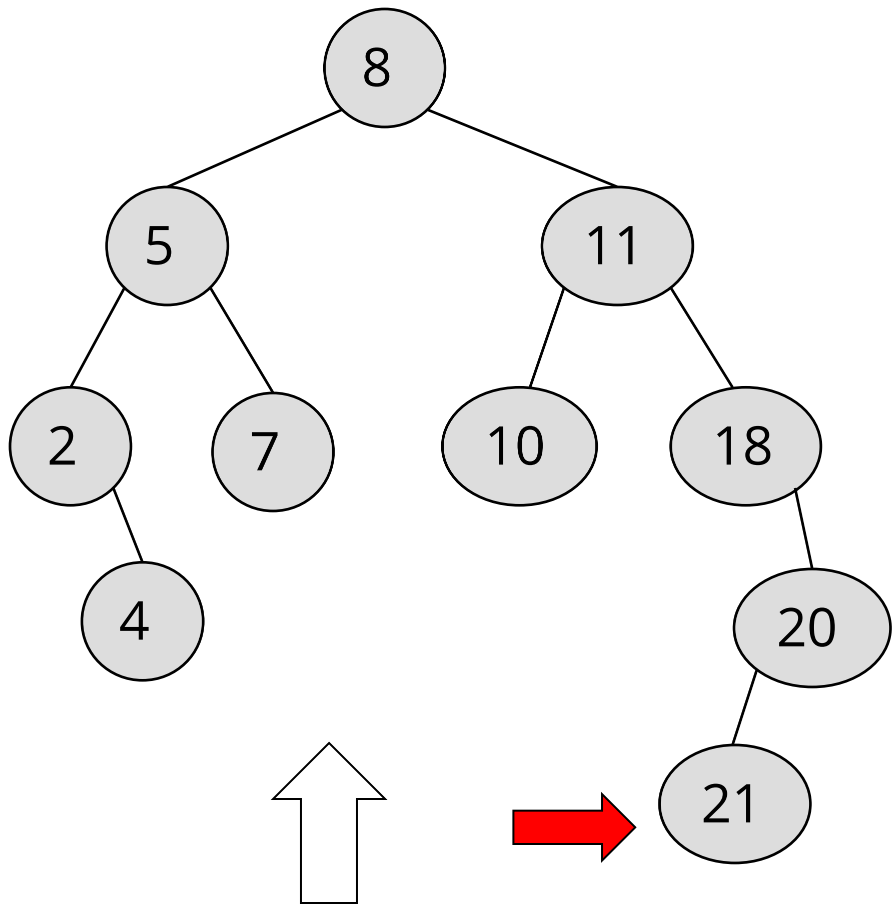
- What if duplicates are allowed?
- All values in left child subtree < value in node j <= all values in the right tree child subtree



BST examples

Which of the trees shown are valid binary search trees?



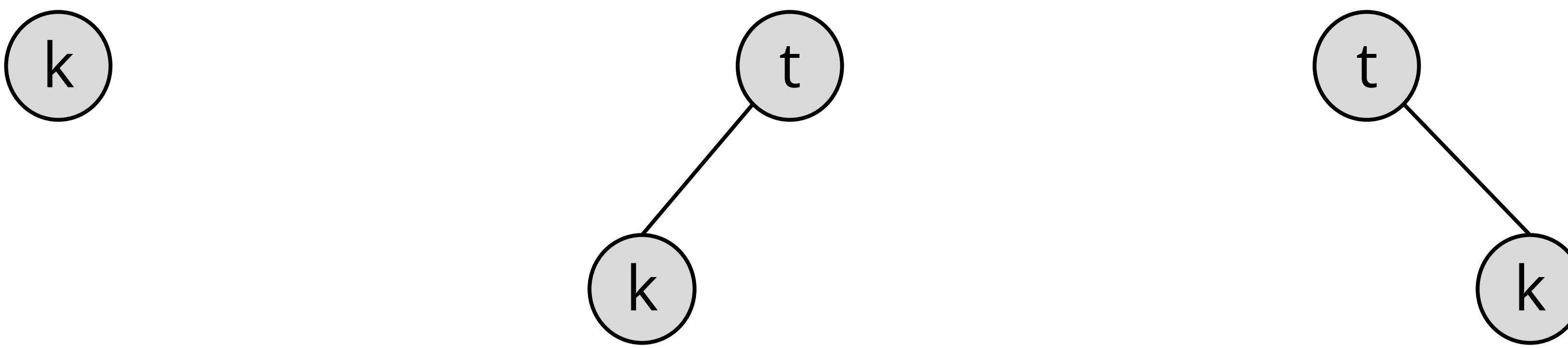


Valid

Not valid

Adding to a BST

- To insert k into tree t:
  - **`t`** is empty, replace **`t`** by a tree consisting of a single node with value **`k`**.
  - If **`k`** is less than the value at the root of **`t`**, insert **`k`** into the left subtree of **`t`**.
  - If **`k`** is greater than the value at the root of **`t`**, insert **`k`** into the right subtree of **`t`**.

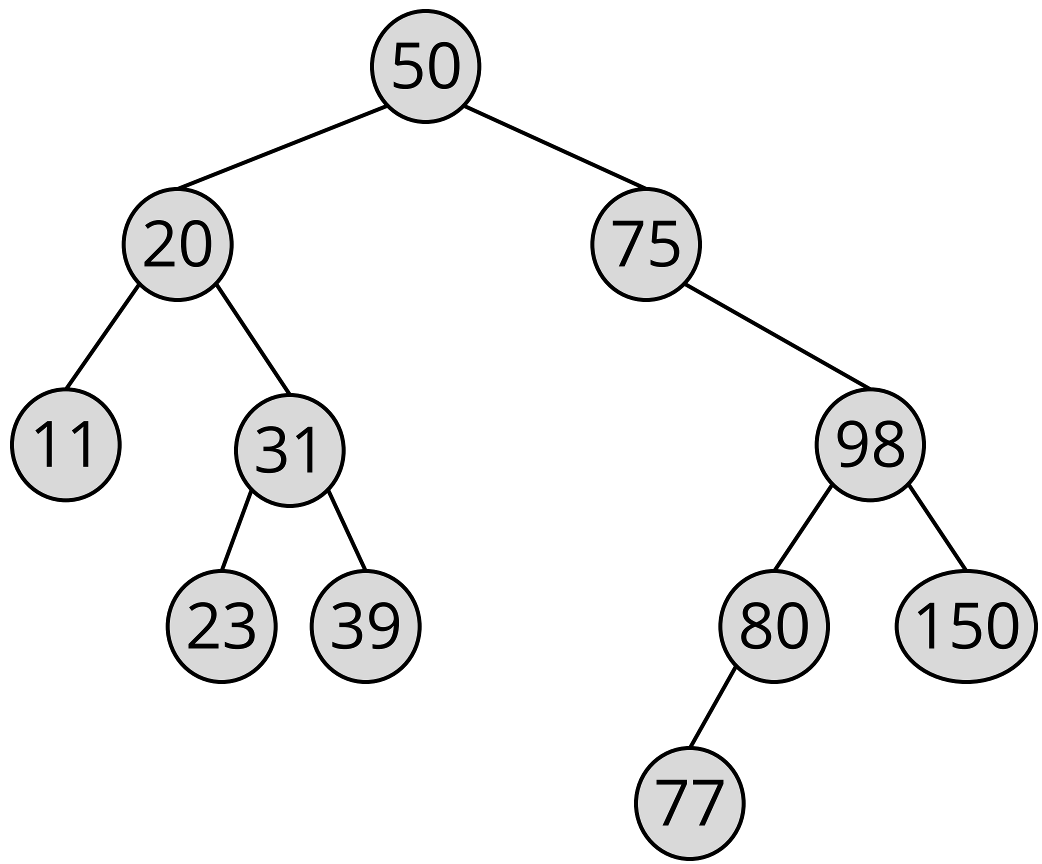


`t` is empty　　**k < root of t**　　**k > root of t**

Adding to a BST

- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77



The add method

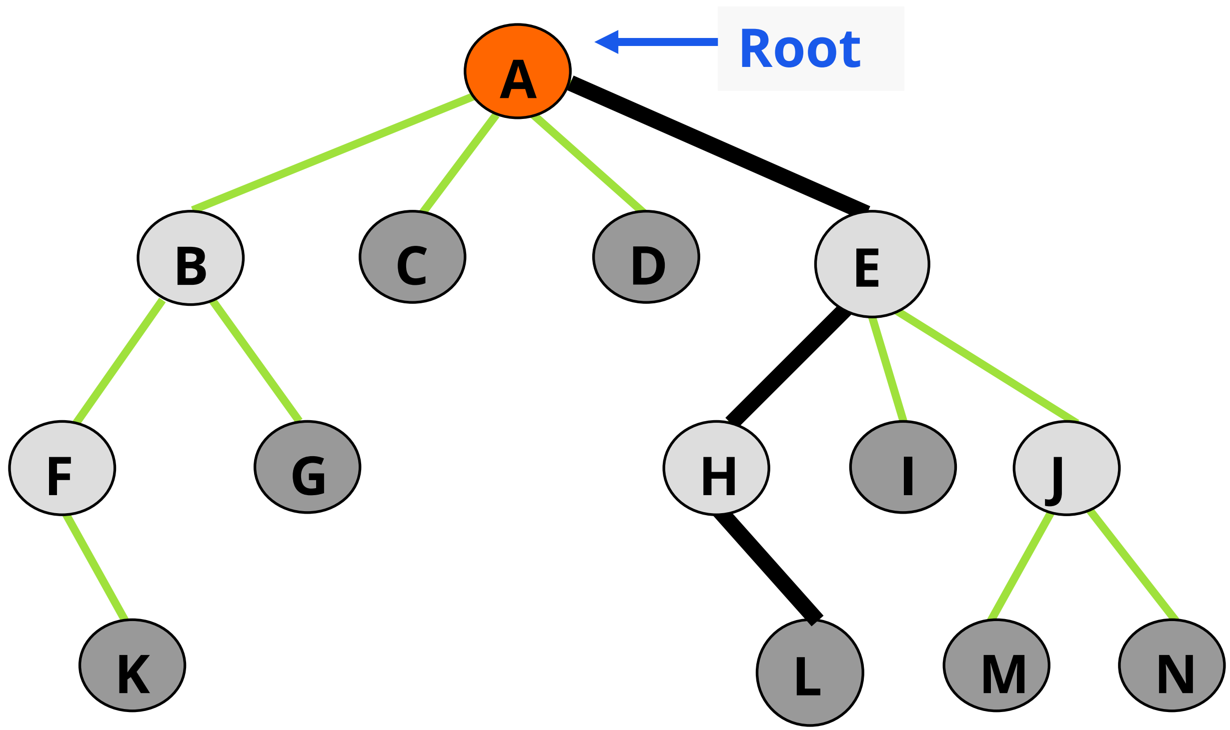
```
public void insert(int key)
{
    root = insertRecursive(root, key);
}

public Node insertRecursive(Node current, int key){

    if (current == null){
      return new Node(key);
    } else if (key < current.key){
        current.left = insertRecursive(current.left, key);
    } else if (key > current.key){
        current.right = insertRecursive(current.right, key);
    }
    return current;
}
```

Height of a Tree

- **Height of a (non-empty) tree** : A tree's (or subtree's) length of the longest path from the root to a leaf



Height = 3

Measuring the Height of a Binary Tree

- An empty tree has height of -1

- The height of a binary tree rooted at node a can be found recursively by
  - calculating the height of the left subtree of a,
  - calculating the height of the right subtree of a, and
  - returning the maximum of the two, plus one.

```
height(a) = 1 + max{height(a.left), height(a.right)}
```

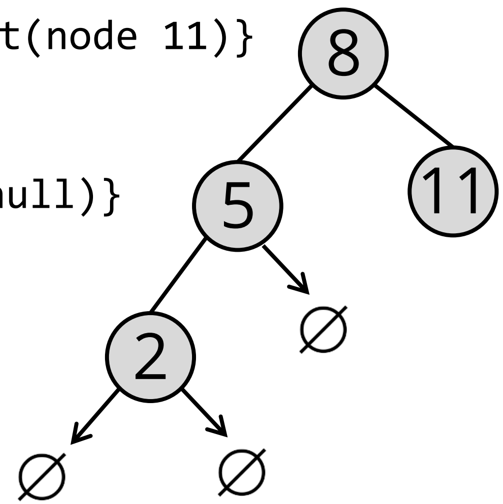
Measuring Height Example

`height(a) = 1 + max{height(a.left), height(a.right)}`

`1 + max{height(node 5), height(node 11)}`

`1 + max{height(node 2), height(null)}`

`1 + max{height(null), height(null)}`



`1 + max{height(node 5), height(node 11)}`

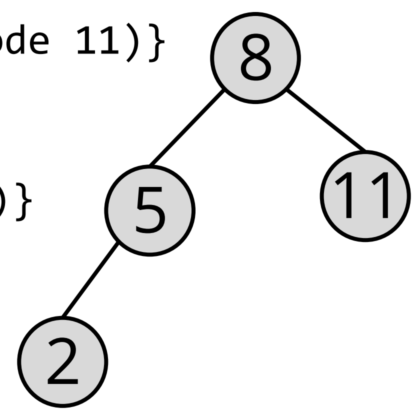
1

`1 + max{height(node 2), height(null)}`

0 -1

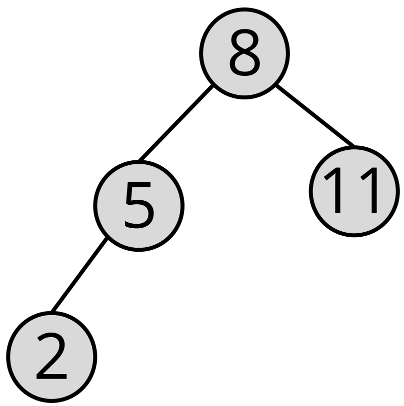
`1 + max{height(null), height(null)}`

-1 -1



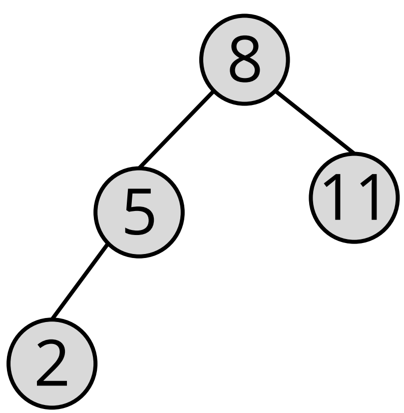
Measuring Height Example

`height(a) = 1 + max{height(a.left), height(a.right)}`



`1 + max{1, height(node 11)}`

`1 + max{height(null), height(null)}`



`1 + max{1, height(node 11)}`

0

`1 + max{height(null), height(null)}`

-1 -1

`height(8) = 2`

Measuring Height Example

```
public int height(){
    return heightSubtree(root);
}

public int heightSubtree(Node current){
    int height = -1;
    if (root == null)
        return height;
    else if (current !=null) {
        int leftHeight = heightSubtree(current.left);
        int rightHeight = heightSubtree(current.right);
        height = 1 + Math.max(leftHeight, rightHeight);
    }
    return height;
}
```

Time Complexity of Measuring Height

- How long does it take to measure the height?
  - Inside the recursive function for each node, we perform O(1) operations.
  - Each node is visited exactly once.
  - In total, the time complexity will be $n \times O(1) = O(n)$.

Tree Traversal: Preorder

```
public void preorderTraversal(){
    if (root == null)
        System.out.println("Tree is empty");
    else
        preorderTraversalRecursive(root);
}


public void preorderTraversalRecursive(Node current){
    if (current != null) {
        System.out.println("Visit " + current.key);
        preorderTraversalRecursive(current.left);
        preorderTraversalRecursive(current.right);
    }
}
```

Tree Traversal: Inorder

```java
public void inorderTraversal(){
    if (root == null)
        System.out.println("Tree is empty");
    else
        inorderTraversalRecursive(root);
}


public void inorderTraversalRecursive(Node current){
    if (current != null) {
        inorderTraversalRecursive(current.left);
        System.out.println("Visit " + current.key);
        inorderTraversalRecursive(current.right);
    }
}
```

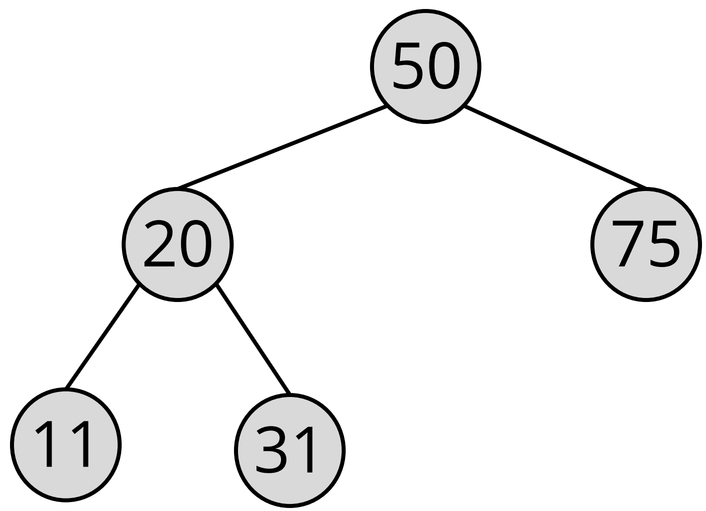
Tree Traversal: Postorder

```java
public void postorderTraversal(){
    if (root == null)
        System.out.println("Tree is empty");
    else
        postorderTraversalRecursive(root);
}


public void postorderTraversalRecursive(Node current){
    if (current != null) {
        postorderTraversalRecursive(current.left);
        postorderTraversalRecursive(current.right);
        System.out.println("Visit " + current.key);
    }
}
```

Inorder traversal Output

- The output of an inorder traversal of a binary search tree is the values of the tree in sortedorder.



Output of an inorder traversal: 11 → 20 → 31 → 50 → 75

Time Complexity of Tree Traversal

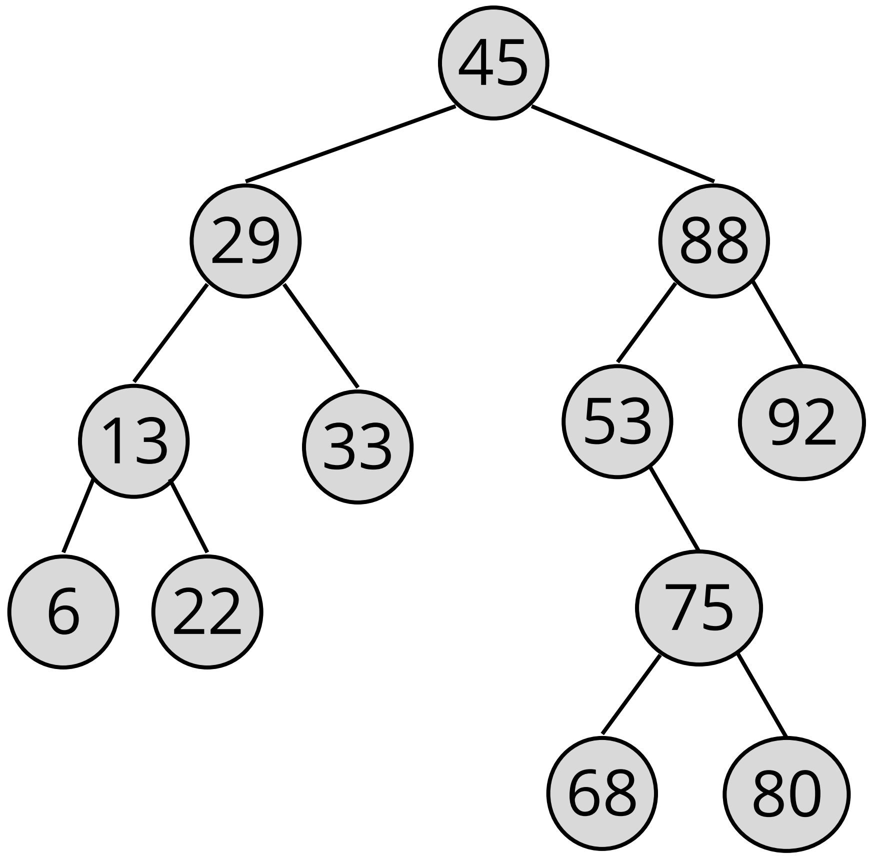
- All in-order, pre-order, and post-order traversals take O(n)
  - each node is visited exactly once
  - visiting a node takes O(1) time
- worst case = average case = best case for tree traversal.

Searching for a Key in the Tree

- If keys in a tree are unordered, **searching for a key** *s* in a tree rooted at node a is similar to performing a pre-order traversal:
  - check whether s is the key of a,
  - recursively search for s in the left subtree of a,
  - recursively search for s in the right subtree of a.

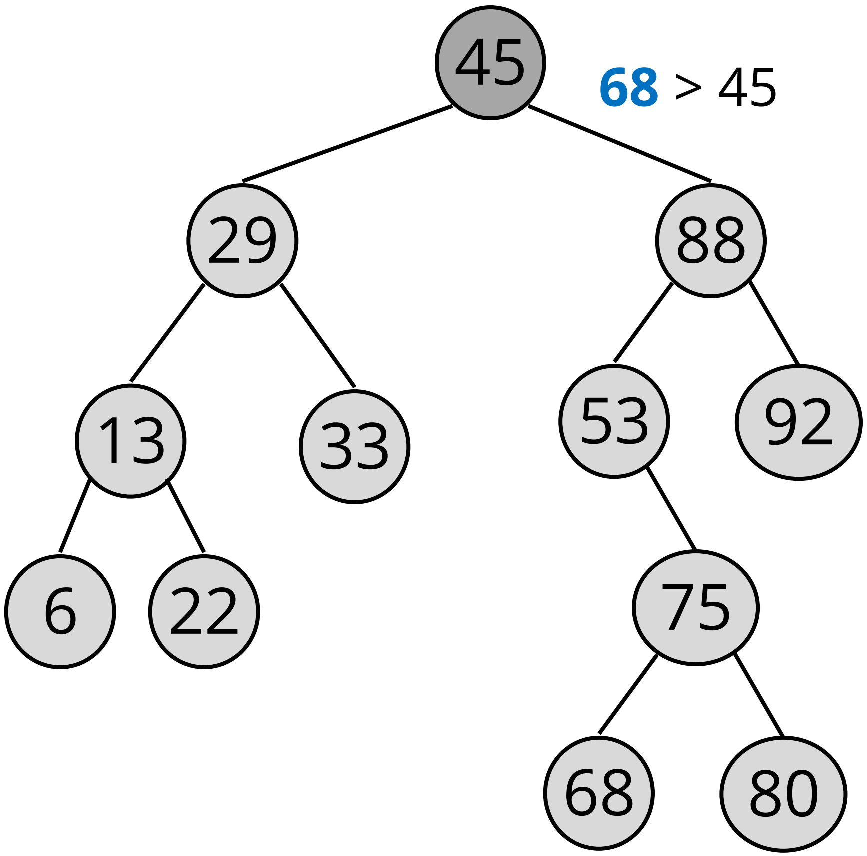
Searching for a Key in the Tree

- Find 68 in the following tree



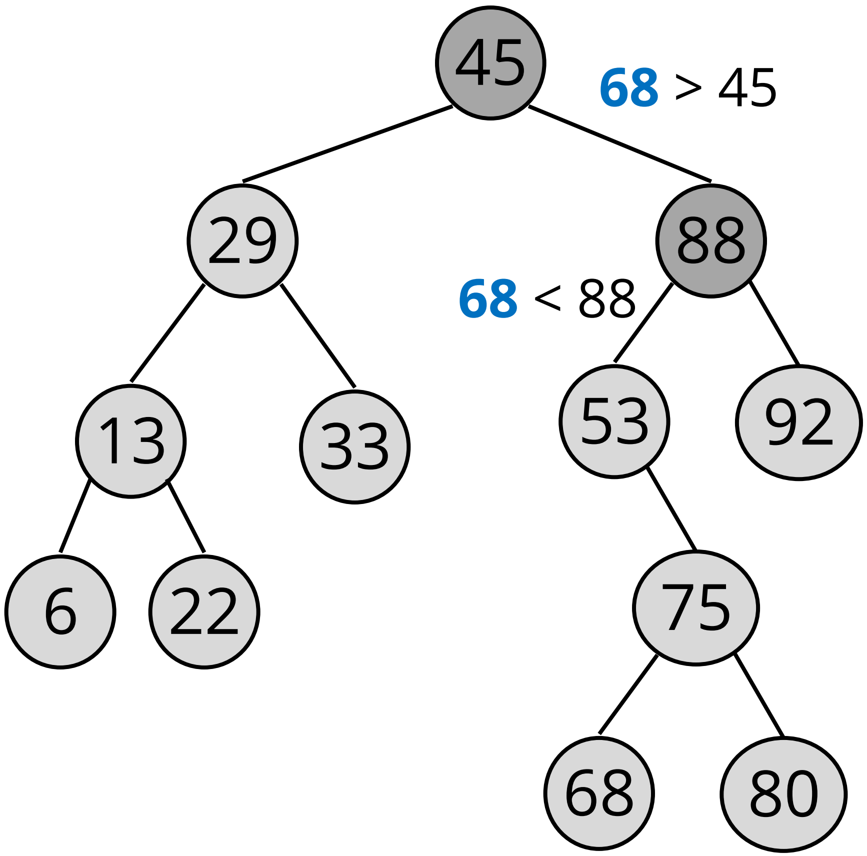
Searching for a Key in the Tree

- Find 68 in the following tree



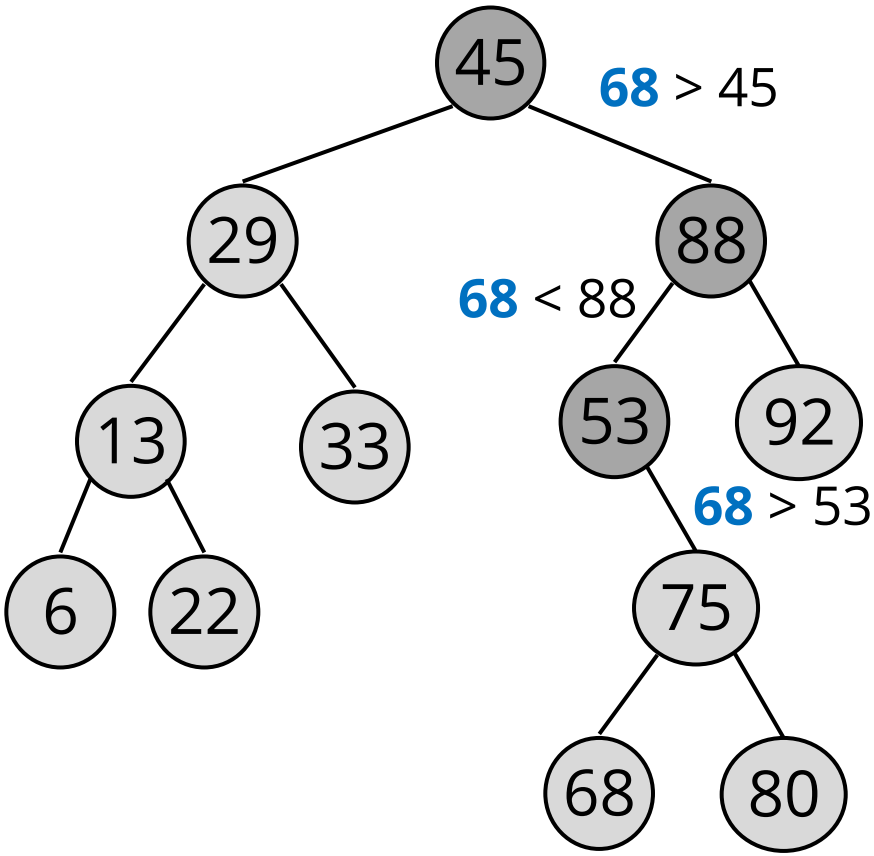
Searching for a Key in the Tree

- Find 68 in the following tree



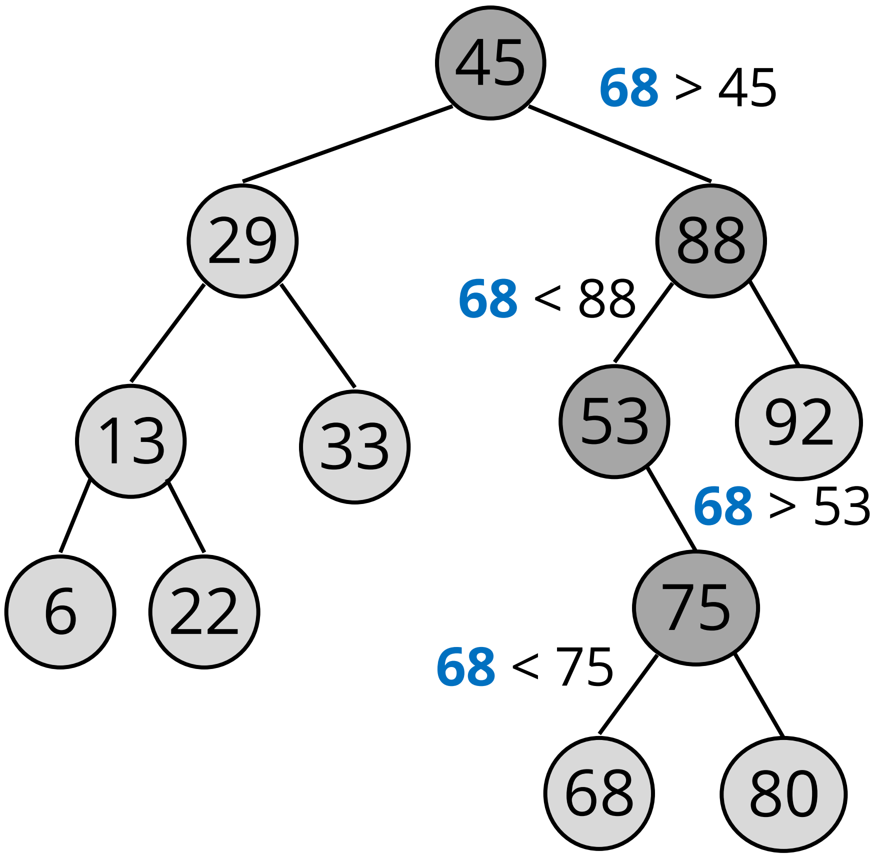
Searching for a Key in the Tree

- Find 68 in the following tree



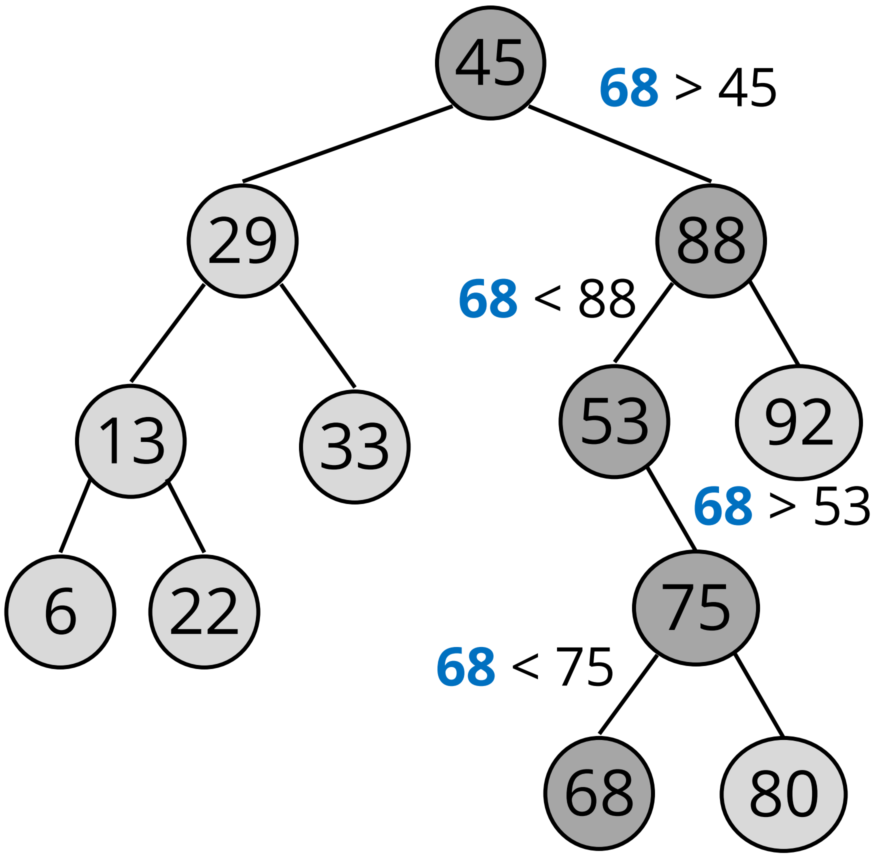
Searching for a Key in the Tree

- Find 68 in the following tree



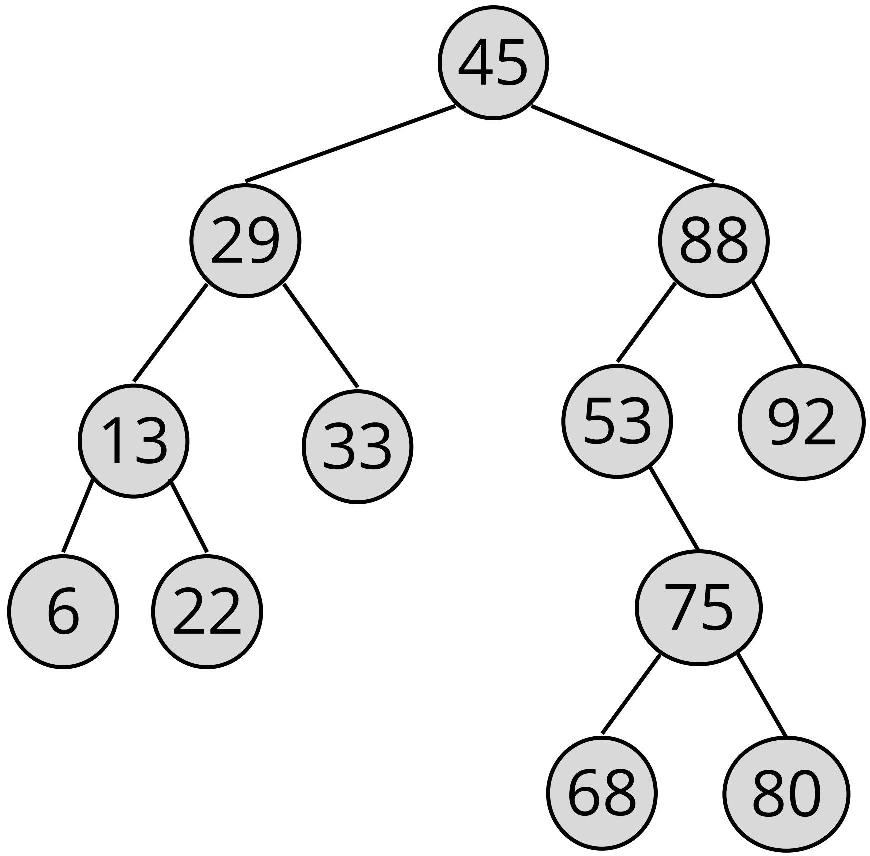
Searching for a Key in the Tree

- Find 68 in the following tree



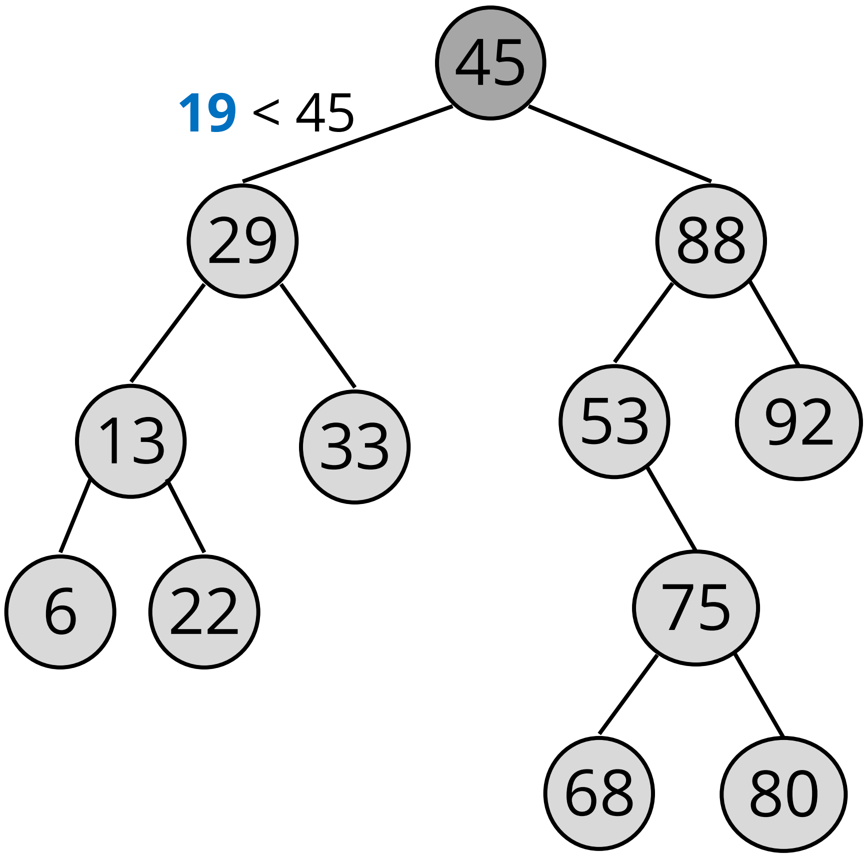
Searching for a Key in the Tree

- Find 19 in the following tree



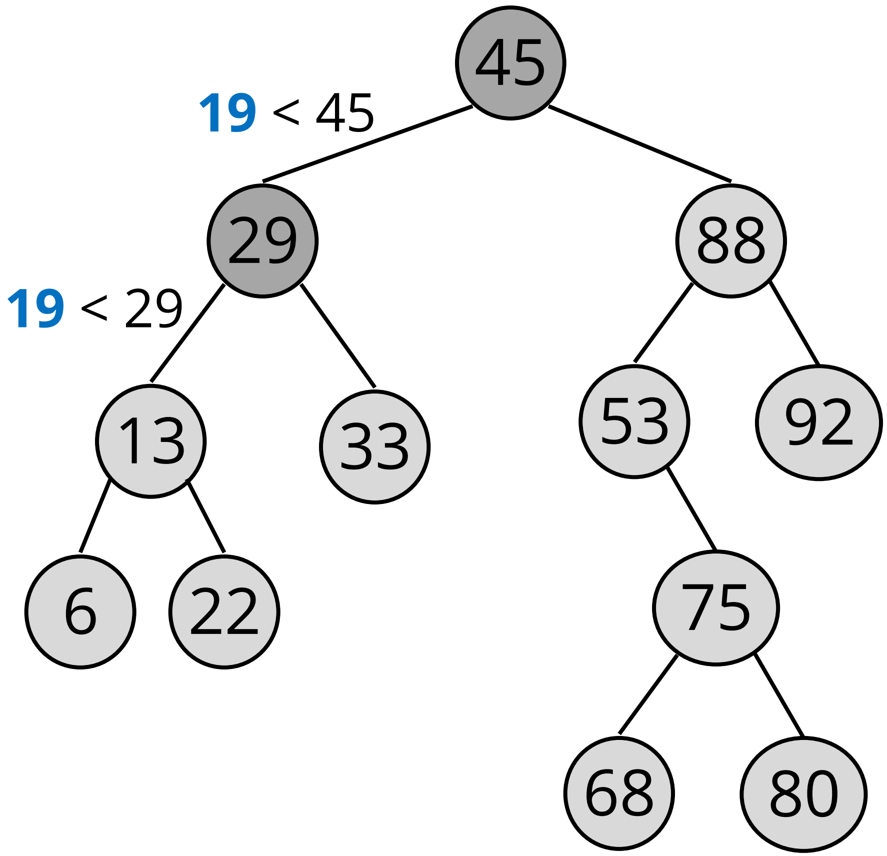
Searching for a Key in the Tree

- Find 19 in the following tree



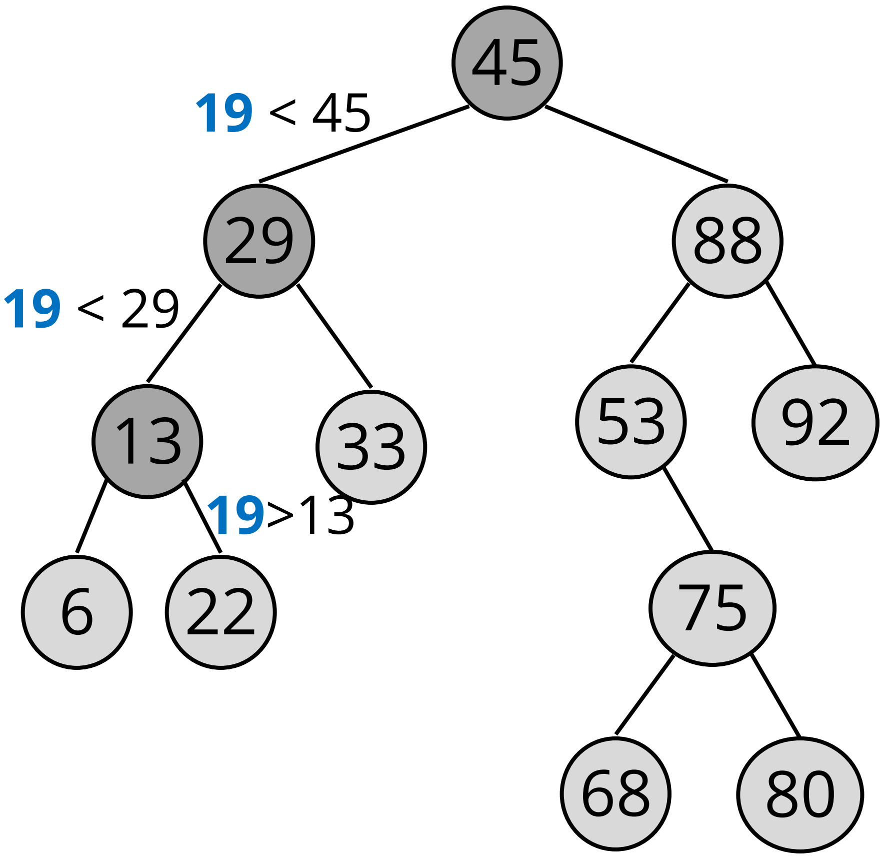
Searching for a Key in the Tree

- Find 19 in the following tree



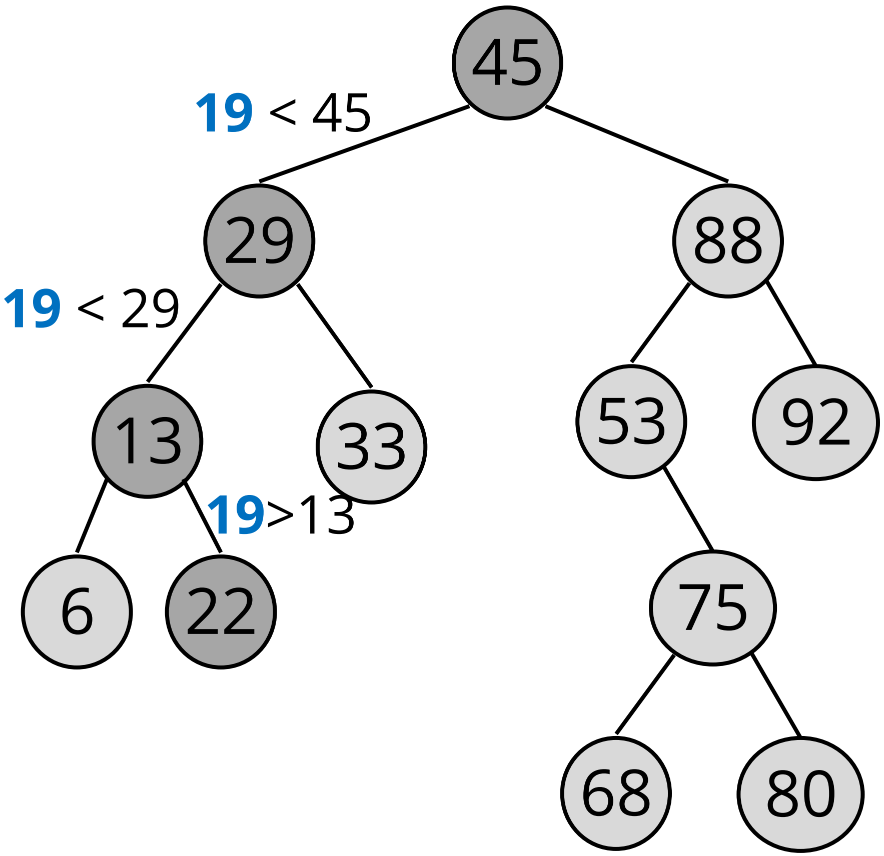
Searching for a Key in the Tree

- Find 19 in the following tree



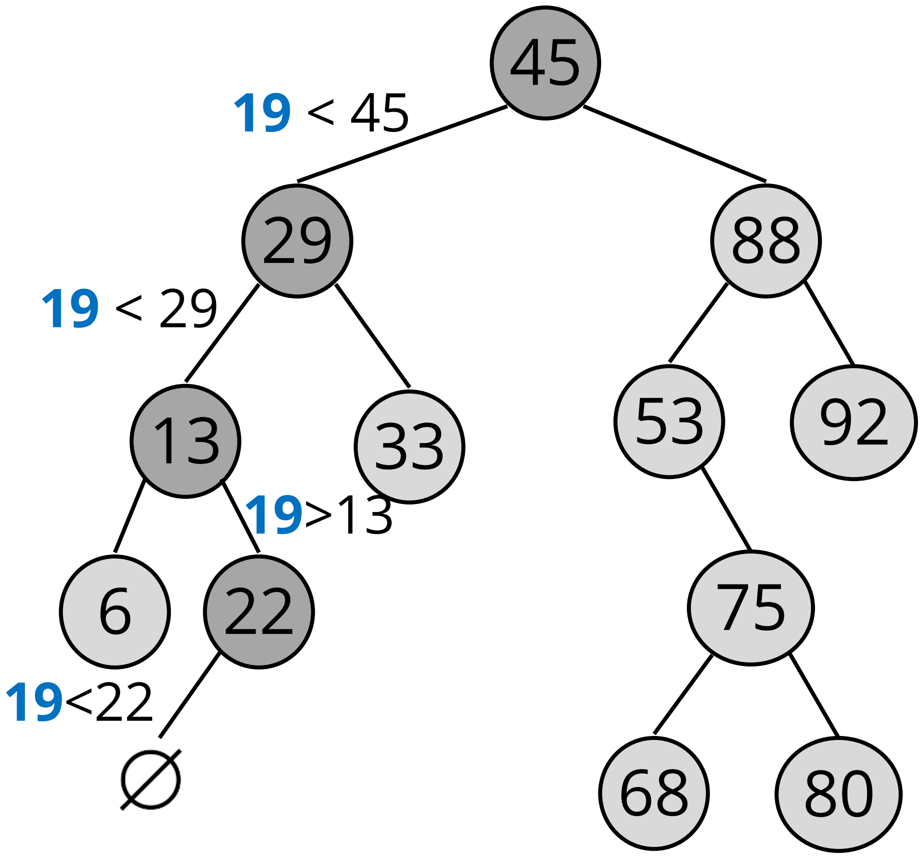
Searching for a Key in the Tree

- Find 19 in the following tree



Searching for a Key in the Tree

- Find 19 in the following tree



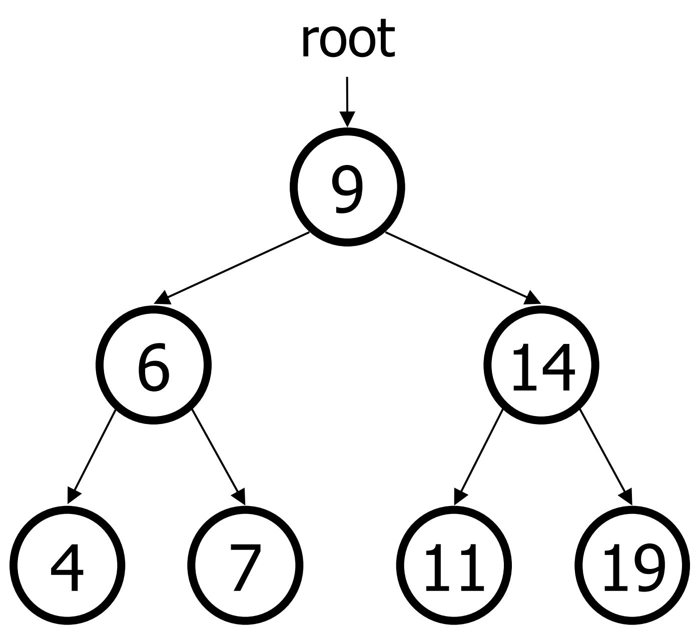
Searching for a Key in the Tree

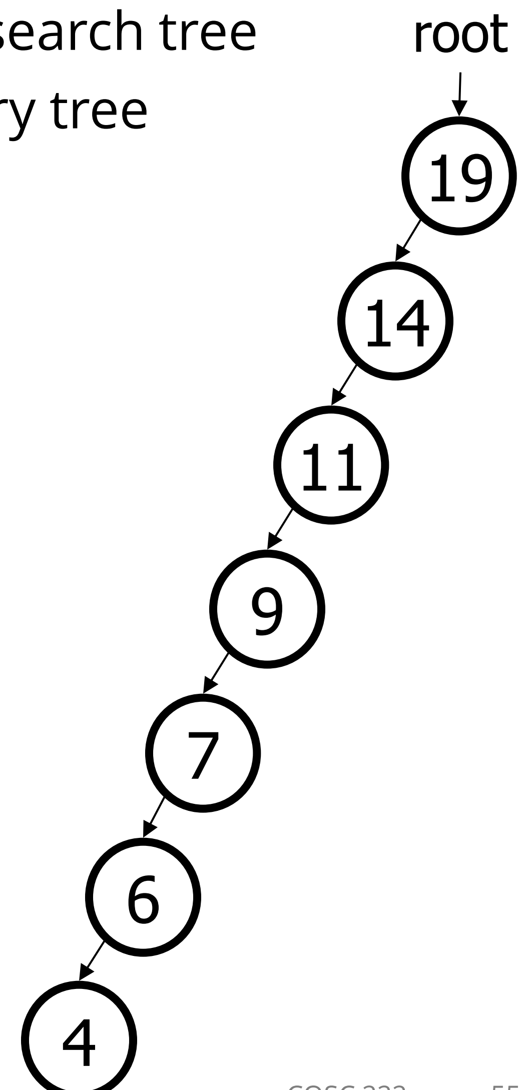
```
public boolean search(int item) {
    return recursiveSearch(root, item);
}


public boolean recursiveSearch(Node current, int key){
    if (current == null)
        return false;
    else if (current.key == key)
        return true;
    else if (key < current.key)
        return recursiveSearch(current.left, key);
    else
        return recursiveSearch(current.right, key);
}
```

Search Time Complexity

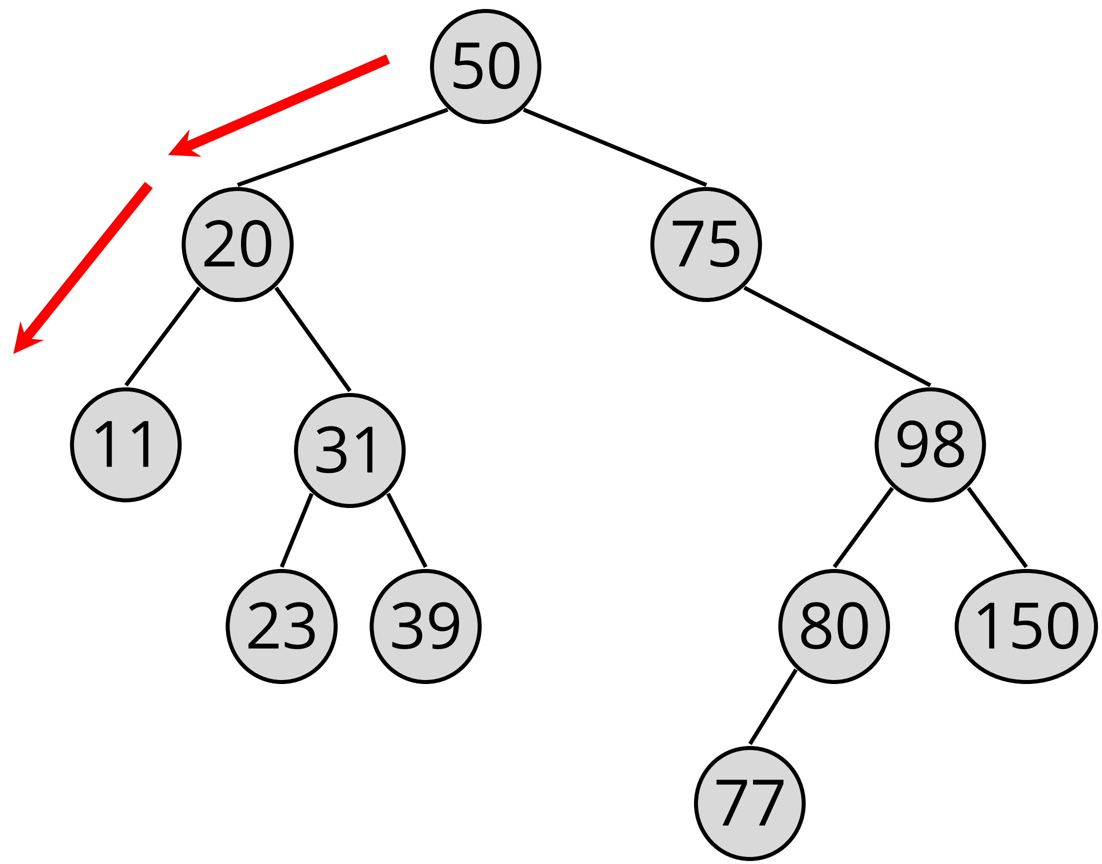
- The BSTs below contain the same elements.
  - require O(log n) time on a balanced binary search tree
  - O(n) worst-case time on an unordered binary tree





Finding Minimum in a BST

- The **minimum value** in the tree is found by starting at the root and repeatedly moving to the left child until you reach a leaf.

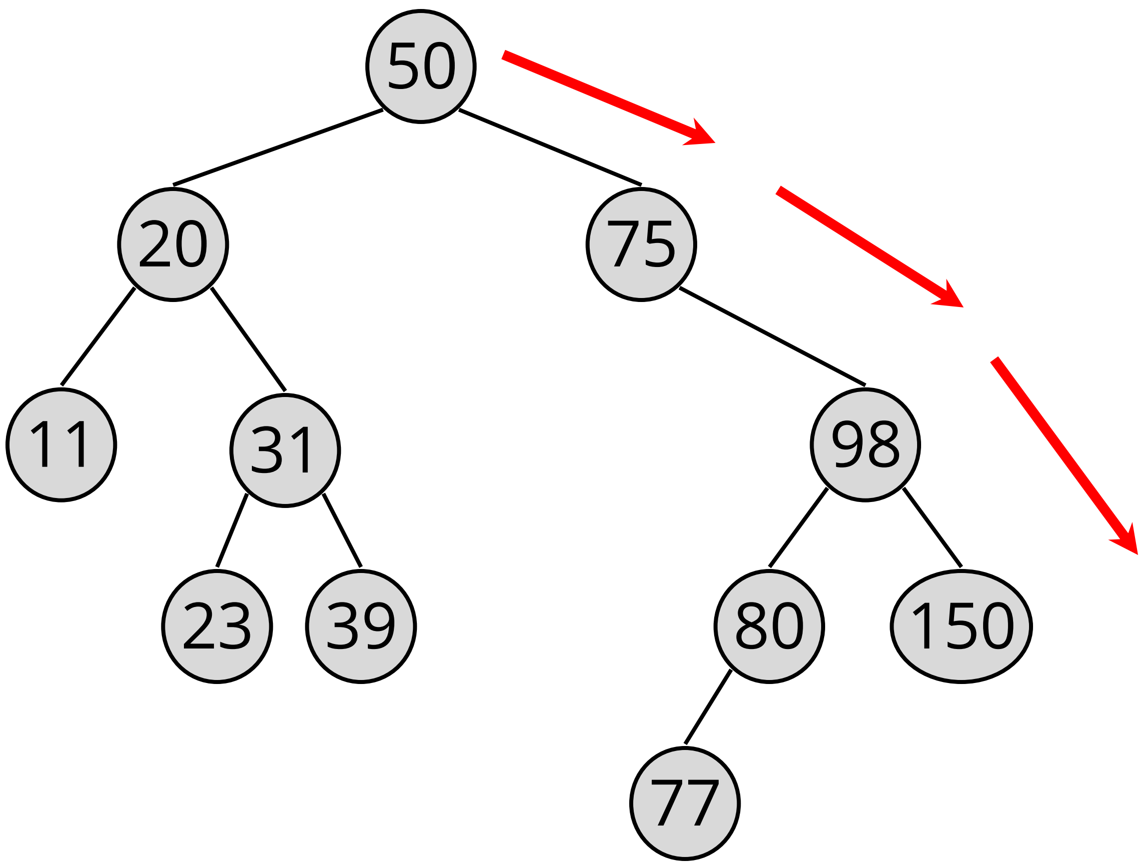


Finding Minimum in a BST

```java
public int minimum(){
    Node current = root;
    while (current.left!=null)
        current = current.left;
    return current.key;
}
```

Finding Maximum in a BST

- The **maximum value** is found by starting at the root and repeatedly moving to the right child until you reach a leaf.



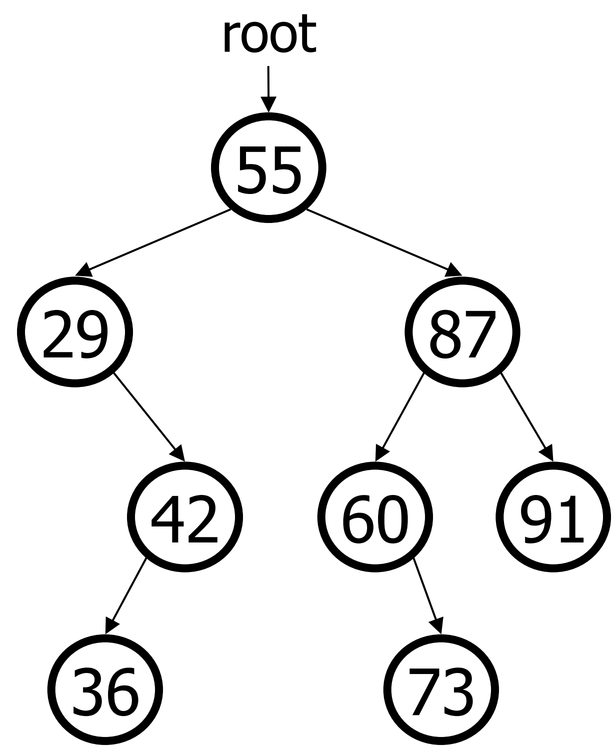
Finding Maximum in a BST

```
public int maximum(){
    Node current = root;
    while (current.right!=null)
        current = current.right;
    return current.key;
}
```

Removing from a BST

- How can we remove a value from a BST in such a way as to maintain proper BST ordering?

```
tree.remove(73);
tree.remove(29);
tree.remove(87);
tree.remove(55);
```



Deletion is somewhat more difficult, depending on the location of the value that you want to delete.

Questions?