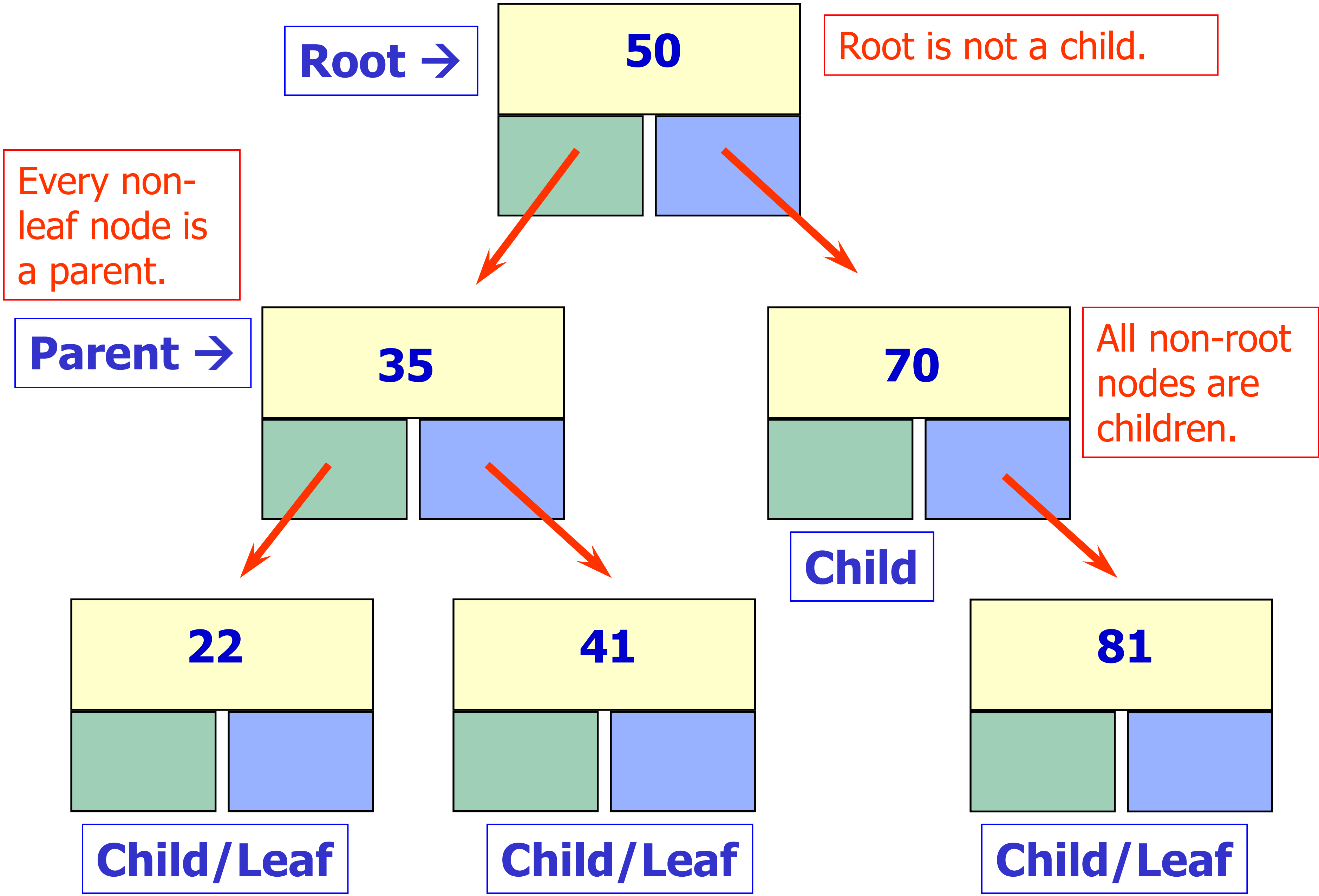
A+ Computer Science

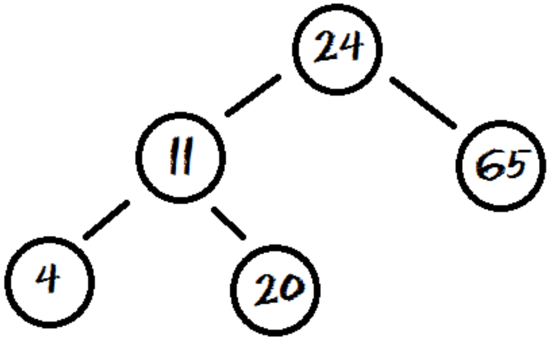
Binary trees

What is a Tree?


Root →
50
Root is not a child.
Every non-leaf node is a parent.
Parent →
35
70
All non-root nodes are children.
Child
22
41
81
Child/Leaf
Child/Leaf
Child/Leaf

Binary Tree

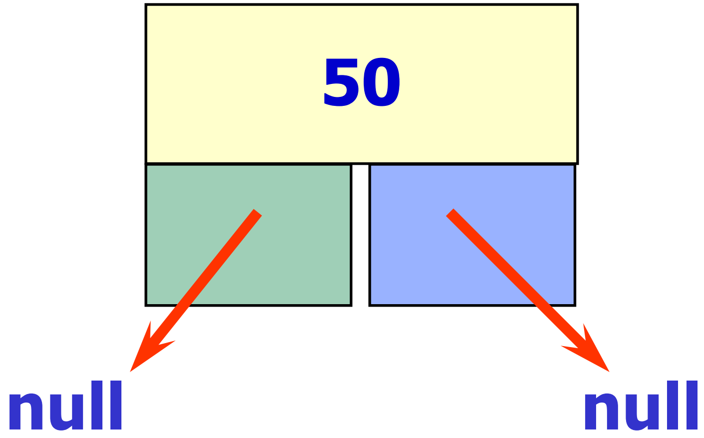
A binary tree is a collection of nodes. Each node has a data value and references to two other nodes. Each node could have a left child and/or a right child.



Simple Node Class

```
public class Node
{
  private Comparable data;
  private Node left;
  private Node right;

  public Node(Comparable dat, Node lft, Node rt)
  {
    data=dat;
    left=lft;
    right=rt;
  }
}
```

A Single Node



A tree node typically has a data component and a reference to a left child and a reference to a right child.

Treeable Interface

```
public interface Treeable
{
  public Object getValue();
  public Treeable getLeft();
  public Treeable getRight();
  public void setValue(Comparable value);
  public void setLeft(Treeable left);
  public void setRight(Treeable right);
}
```

TreeNode Class

```
public class TreeNode implements Treeable
{
  private Comparable treeNodeValue;
  private TreeNode leftTreeNode;
  private TreeNode rightTreeNode;

  public TreeNode( ){
    treeNodeValue = null;
    leftTreeNode = null;
    rightTreeNode = null;
  }

  public TreeNode(Comparable value, TreeNode left, TreeNode right){
    treeNodeValue = value;
    leftTreeNode = left;
    rightTreeNode = right;
  }

  //other methods not shown
 //refer to the Treeable interface
}
```

This TreeNode class is similar to the AP TreeNode.

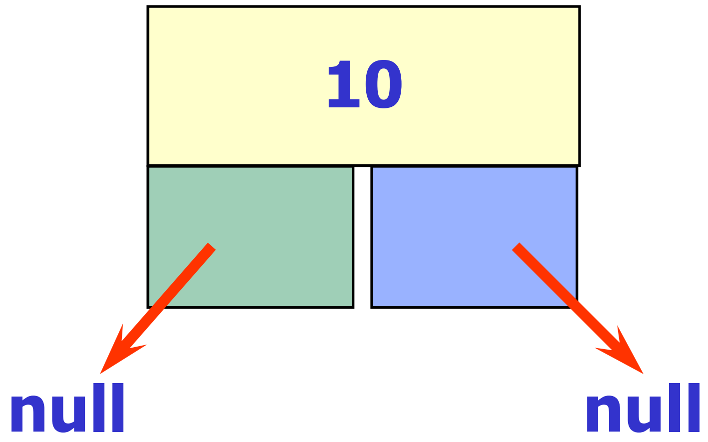
You can obtain the official AP TreeNode class from the college board website. You will be provided with a copy of the AP TreeNode class when you take the AP Computer Science AB exam.

Creating a Single TreeNode

```
Treeable node = new TreeNode("10", null,null);
out.println(node.getValue());
out.println(node.getLeft());
out.println(node.getRight());
```

OUTPUT

10
null
null



onetreenode.java

Linking Tree Nodes

Linking Tree Nodes

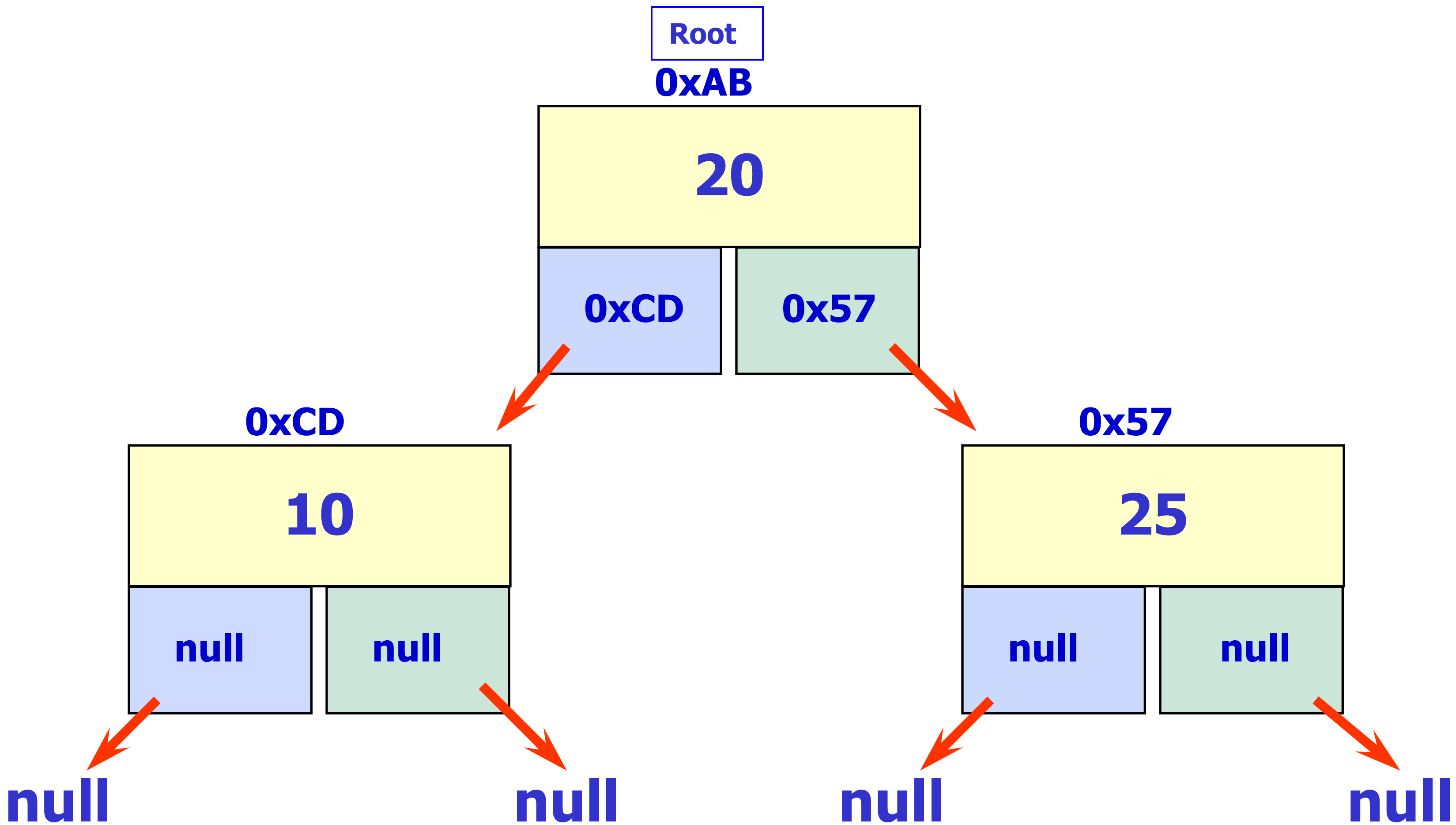
```
TreeNode node = new TreeNode("10",
                    new TreeNode("5", null,null),
                    new TreeNode("20", null,null));

out.println(node.getValue());
out.println(node.getLeft().getValue());
out.println(node.getRight().getValue());
```

OUTPUT

10
5
20

treeone.java

Linking Tree Nodes



Linking Tree Nodes

```
TreeNode x = new TreeNode("10",null,null);
TreeNode y = new TreeNode("25", null,null);
TreeNode z = new TreeNode("20", x, y);

out.println(z.getValue());
out.println(z.getLeft().getValue());
out.println(z.getRight().getValue());
```

OUTPUT

20

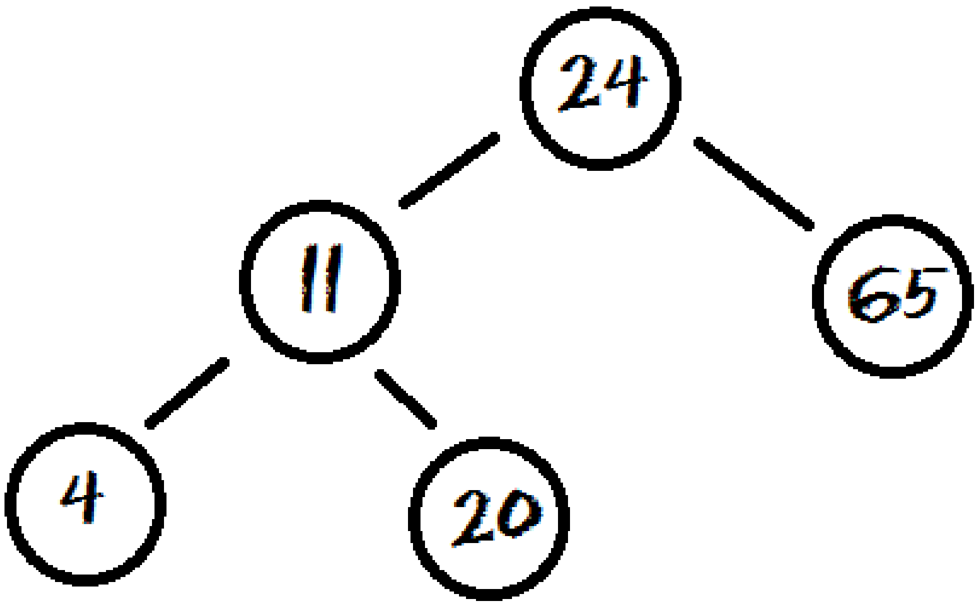
10

25

treetwo.java

Building a Search Tree

Binary Search Tree



left child is less than the parent and right child is greater

Adding Tree Nodes

Every item that is added to a search tree is first compared to the root. If the item is larger than the root, a recursive call is made on the right sub tree. If the item is smaller than the root, a recursive call is made on the left sub tree. This process continues until a null reference is found.

Adding Tree Nodes

```
private TreeNode add(Comparable val, TreeNode tree)
{
  if (tree == null)
     return new TreeNode(val, null, null);

  int dirTest = val.compareTo(tree.getValue());
  if(dirTest<0)
     tree.setLeft(add(val, tree.getLeft()));
  else if(dirTest>0)
     tree.setRight(add(val, tree.getRight()));
  return tree;
}
```

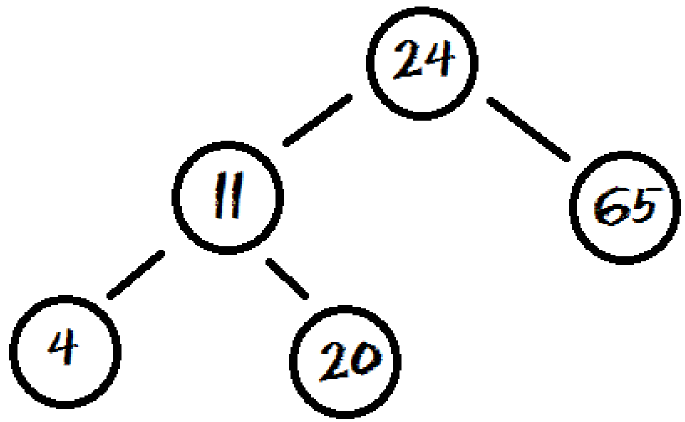
Adding Tree Nodes

```
private TreeNode add(Comparable val, TreeNode tree)
{
  if (tree == null)
    tree = new TreeNode(val, null, null);
  else if (val.compareTo(tree.getValue()) < 0 )
    tree.setLeft(add(val, tree.getLeft()));
  else if (val.compareTo(tree.getValue()) > 0 )
    tree.setRight(add(val, tree.getRight()));
  return tree;
}
```

addprintone.java

Displaying Tree Nodes

Tree Traversals



IN-ORDER = 4 11 20 24 65

PRE-ORDER = 24 11 4 20 65

POST-ORDER = 4 20 11 65 24

REV-ORDER = 65 24 20 11 4

Tree Traversals
In Order

```
private void inOrder(TreeNode tree)
{
  if (tree != null){
        inOrder(tree.getLeft());
        out.print(tree.getValue() + " ");
        inOrder(tree.getRight());
  }
}
```

Tree Traversals
In Order

```
private void inOrder(TreeNode tree)
{
  if (tree == null)
    return;
  inOrder(tree.getLeft());
  out.print(tree.getValue() + " ");
  inOrder(tree.getRight());
}
```

Tree Traversals
In Order

```
private String inOrder(TreeNode tree)
{
  if (tree != null)
    return inOrder(tree.getLeft())
        + tree.getValue() + " " +
        inOrder(tree.getRight());
  return "";
}
```

addprinttwo.java

Searching a Tree

Searching for Values

To search a tree, you will use the same basic logic that you used to add a new node.

First, compare the current node to the search value and see if it is a match. If it is not a match, check to see if you need to search the left sub tree or the right sub tree. Repeat.

Sounds like a binary search.

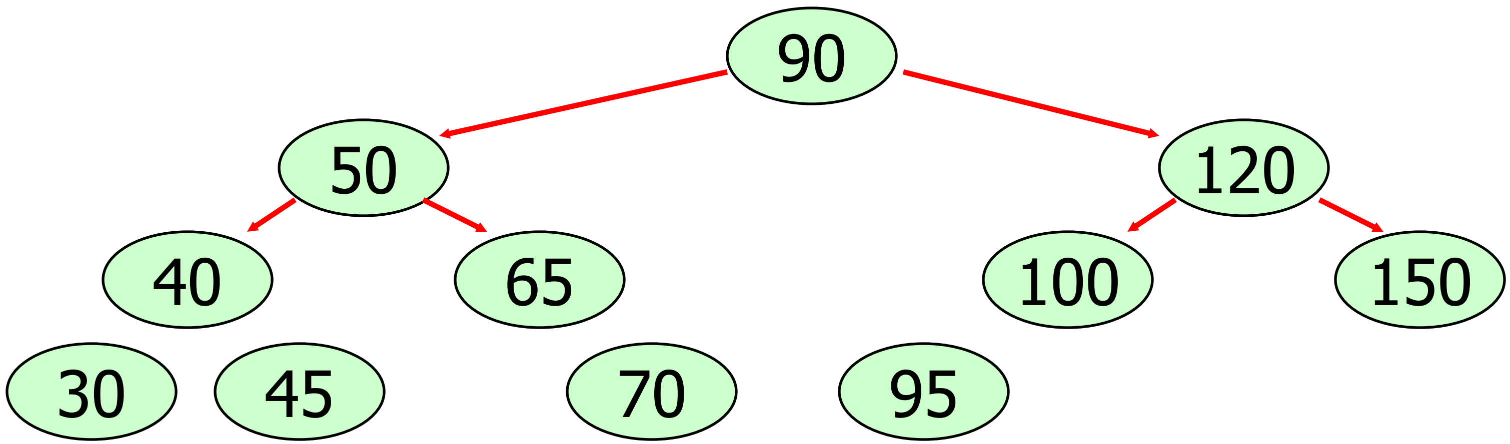
Searching for Values

```
private boolean search(Comparable val, TreeNode tree)
{
   if(tree != null)
   {
      int dirTest = val.compareTo(tree.getValue());
      if(dirTest == 0 )
         return true;
      else if (dirTest < 0)
         return search(val, tree.getLeft());
      else if (dirTest > 0)
         return search(val, tree.getRight());
   }
   return false;
}
```

contains.java

Processing Tree Nodes

Processing Tree Nodes



WIDTH - 7

HEIGHT - 3

NUMLEAVES - 5

NUMLEVELS - 4

NUMNODES - 11

ISFULL – NO

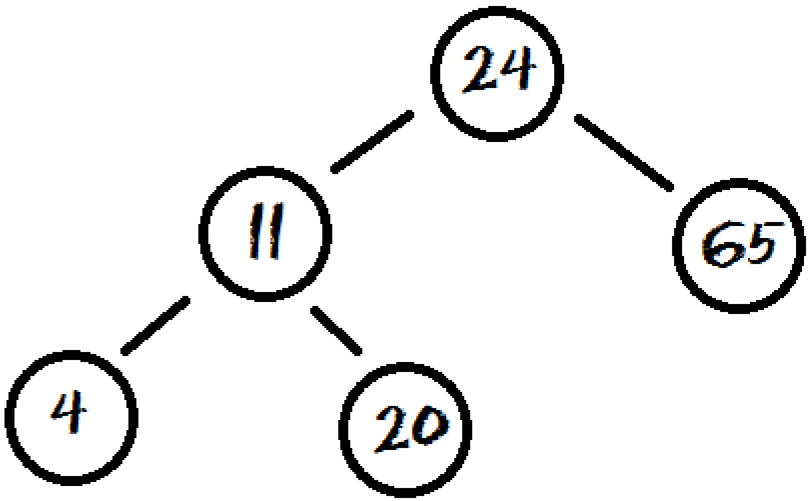
Processing Tree Nodes

Width - dist between two furthest leaves in the tree – does not have to go through the root

Height – longest path from root to a leaf # of links from root to farthest leaf

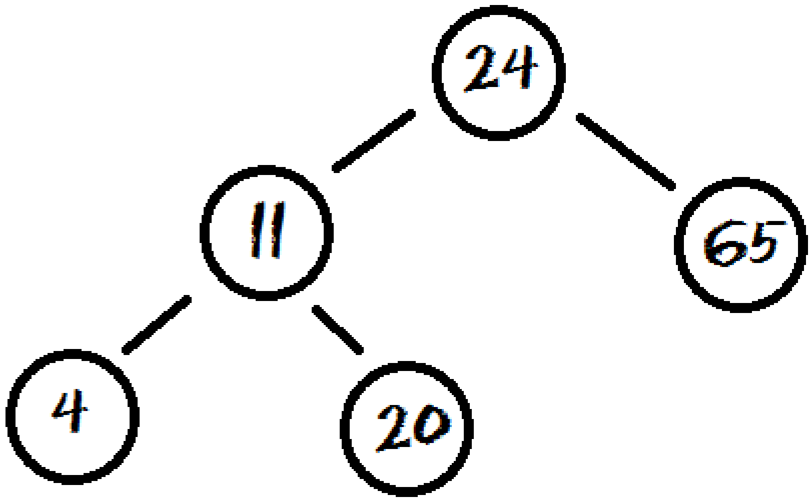
Level - a group of equal nodes
the root is level - 0
the children of the root - level - 1

Processing Tree Nodes



WIDTH - 4

Processing Tree Nodes



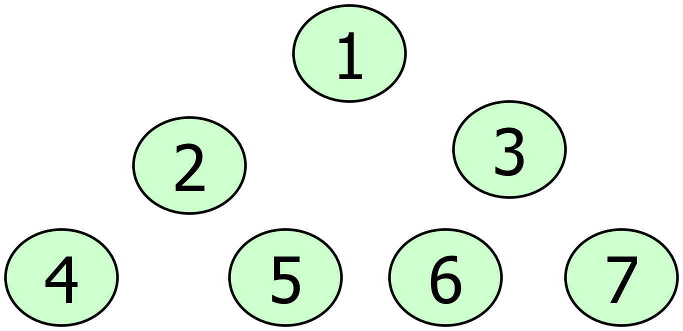
HEIGHT - 2

Processing Tree Nodes

In a full binary tree, every parent has exactly two children or no children at all.   The number of nodes in the tree will equal 2 raised to the number of levels -1 if the tree if full.

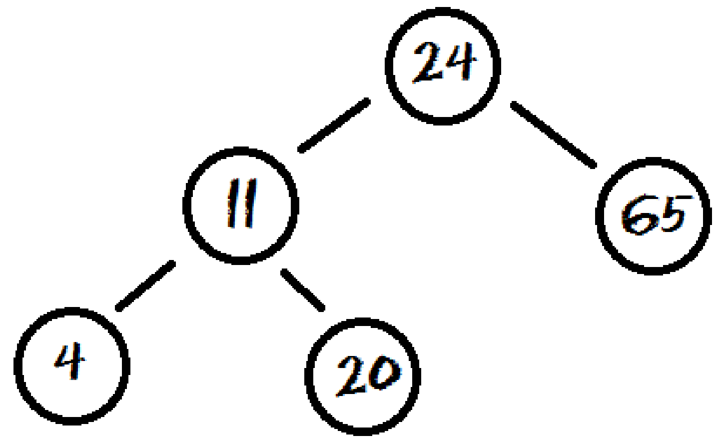
7 is the # of nodes

$2^{3(\text{number of levels})} - 1 = 7$



Processing Tree Nodes

In a complete tree, every level that can be filled is filled. Any levels that are not full have all nodes shifted as far left as possible.



Processing Tree Nodes

```
int  getNumLevels(TreeNode  tree)
{
   if (tree==null)
      return something;
   else
   {
      numLeft = getNumLevels of the left
      numRight = getNumLevels of the right
      if (numLeft > numRight)
          return 1 + numLeft;
      else
          return 1 + numRight;
   }
}
```

Processing Tree Nodes

```
public int getNumLevels()
{
    return getNumLevels(root);
}

private int getNumLevels(TreeNode tree)
{
   if(tree==null)   return 0;
   else
   {
     int numLeft = getNumLevels(tree.getLeft());
     int numRight = getNumLevels(tree.getRight());
     if(numLeft > numRight)
        return 1 + numLeft;
     return 1 + numRight;
   }
}
```

Processing Tree Nodes

```
public int getNumLevels()
{
    return getNumLevels(root);
}

private int getNumLevels(TreeNode tree)
{
  if(tree==null)   return 0;
  else
  {
    return
          1 + Math.max(getNumLevels(tree.getLeft() ,
                       getNumLevels(tree.getRight());
  }
}
```

numlevels.java

Work on Programs!

Crank
Some Code!

A+ Computer Science

Binary trees