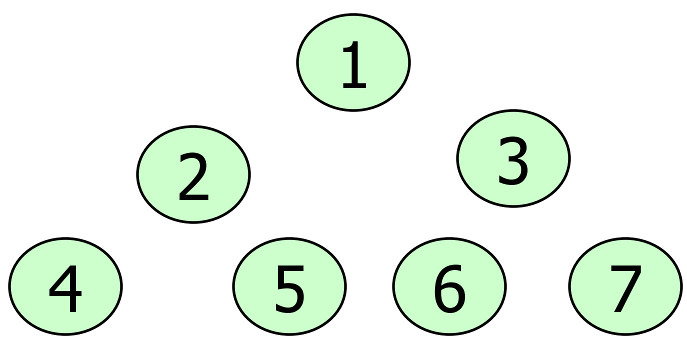
Binary Trees

TreeMap and HashMap

TreeSet and TreeMap were built using binary trees.



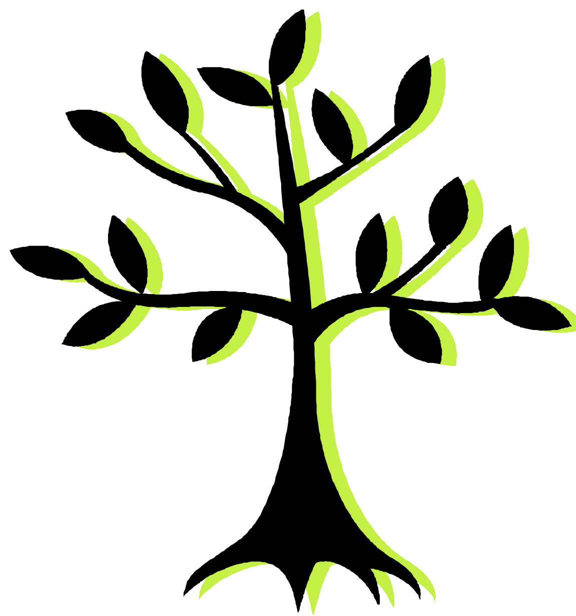
TreeMap

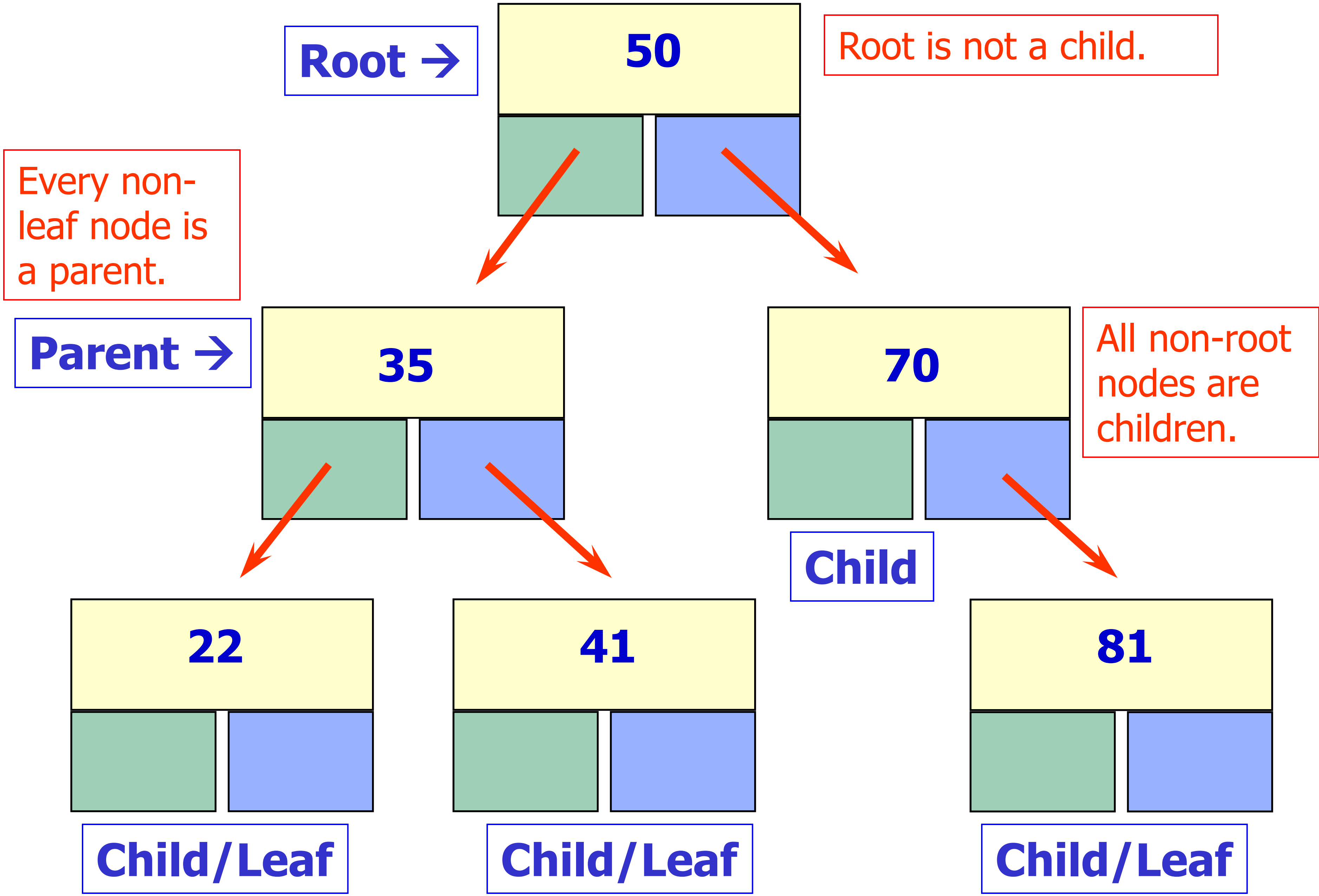
```
Map<Integer,String> map;
map = new TreeMap<Integer,String>();
map.put(1,"one");
map.put(2,"two");
map.put(3,"three");
map.put(4,"four");
map.put(5,"five");
map.put(6,"six");
map.put(7,"seven");

System.out.println(map.get(1));
System.out.println(map.get(13));
System.out.println(map.get(7));
```

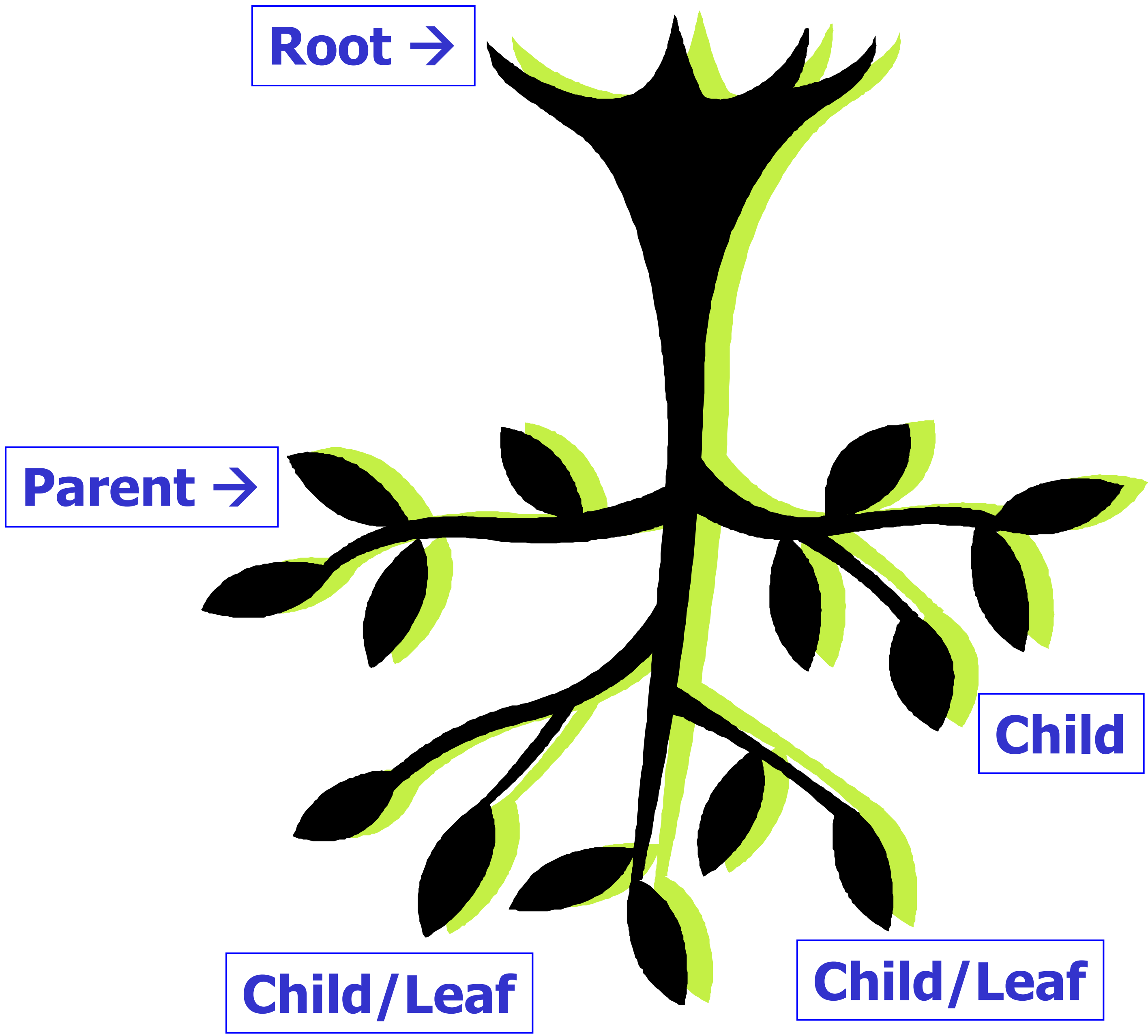
OUTPUT

one
null
seven

What is a tree?

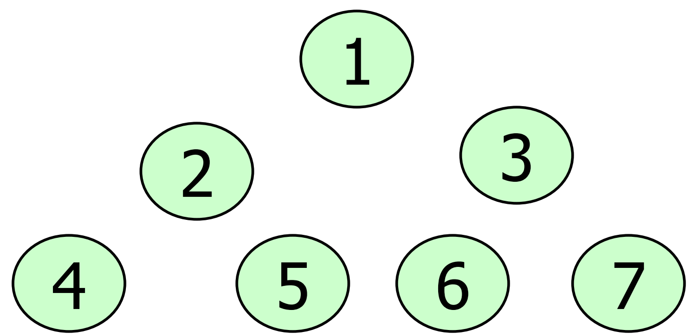



Root →
50
Root is not a child.
Every non-leaf node is a parent.
Parent →
35
70
All non-root nodes are children.
Child
22
41
81
Child/Leaf
Child/Leaf
Child/Leaf


Root →
Parent →
Child
Child/Leaf
Child/Leaf

Binary Tree

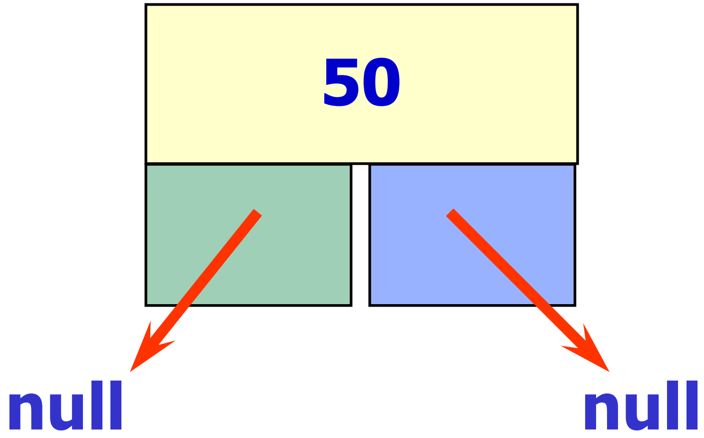
A binary tree is a collection of nodes. Each node has a data value and references to two other nodes. Each node could have a left child and/or a right child.



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);
}
```

The TreeNode Class

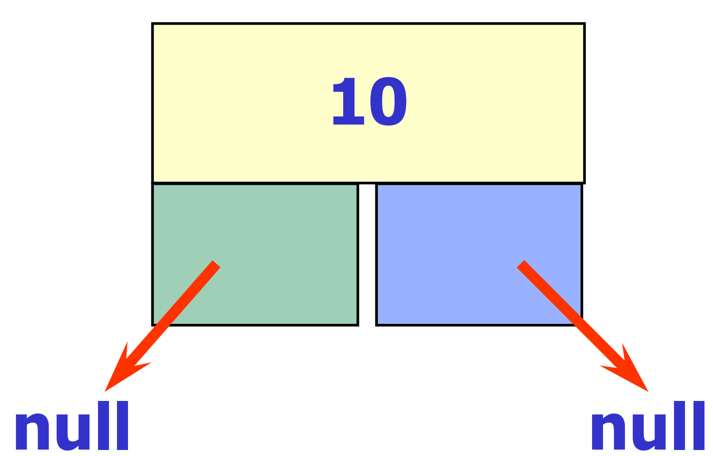
```
public class TreeNode implements Treeable
{
  private Comparable treeNodeValue;
  private TreeNode leftTreeNode;
  private TreeNode rightTreeNode;

  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
}
```

Creating A Single Tree Node

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

OUTPUT

10
null
null



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());
```

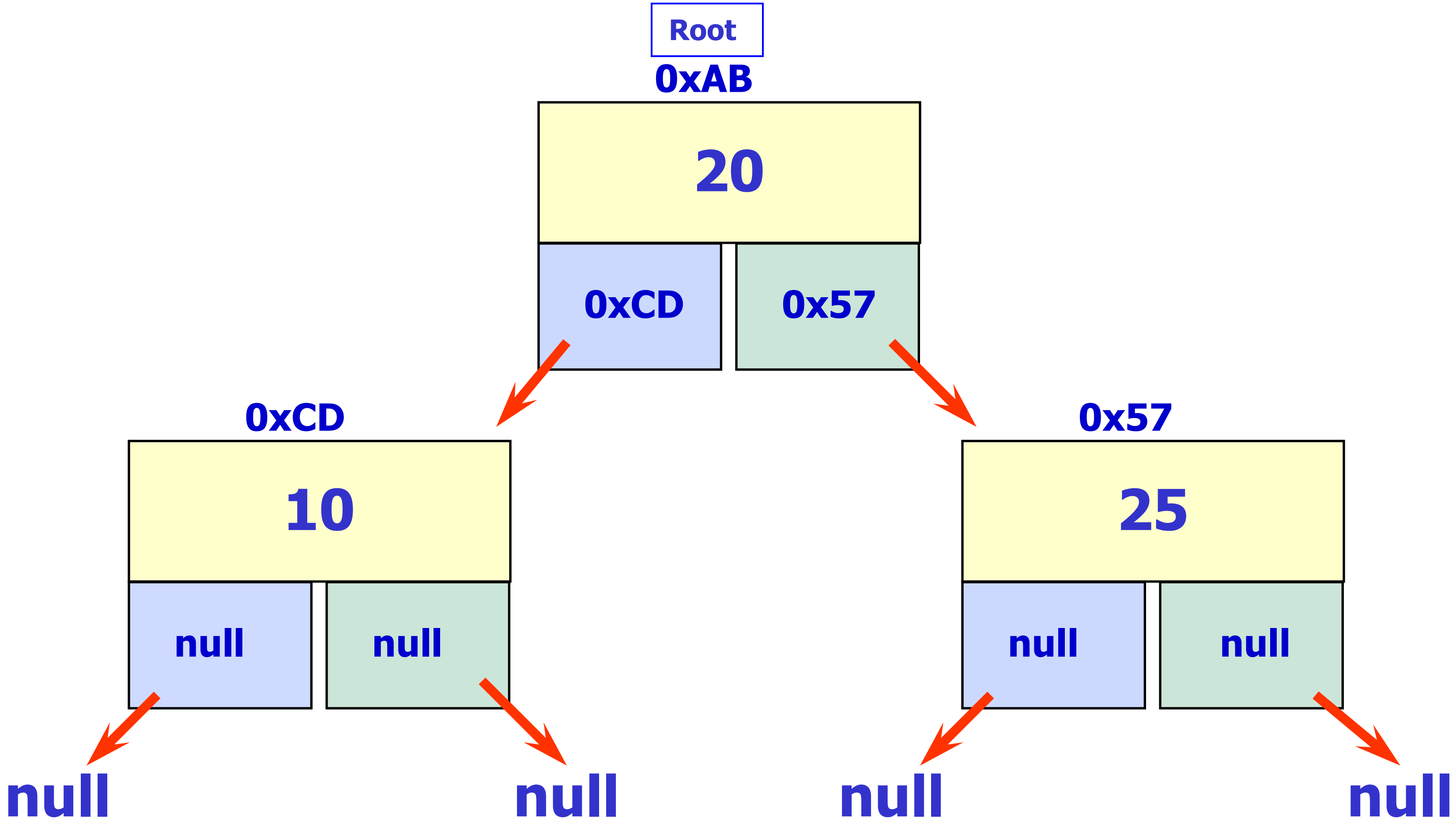
OUTPUT

10

5

20

Linking Tree Nodes



Linking Tree Nodes

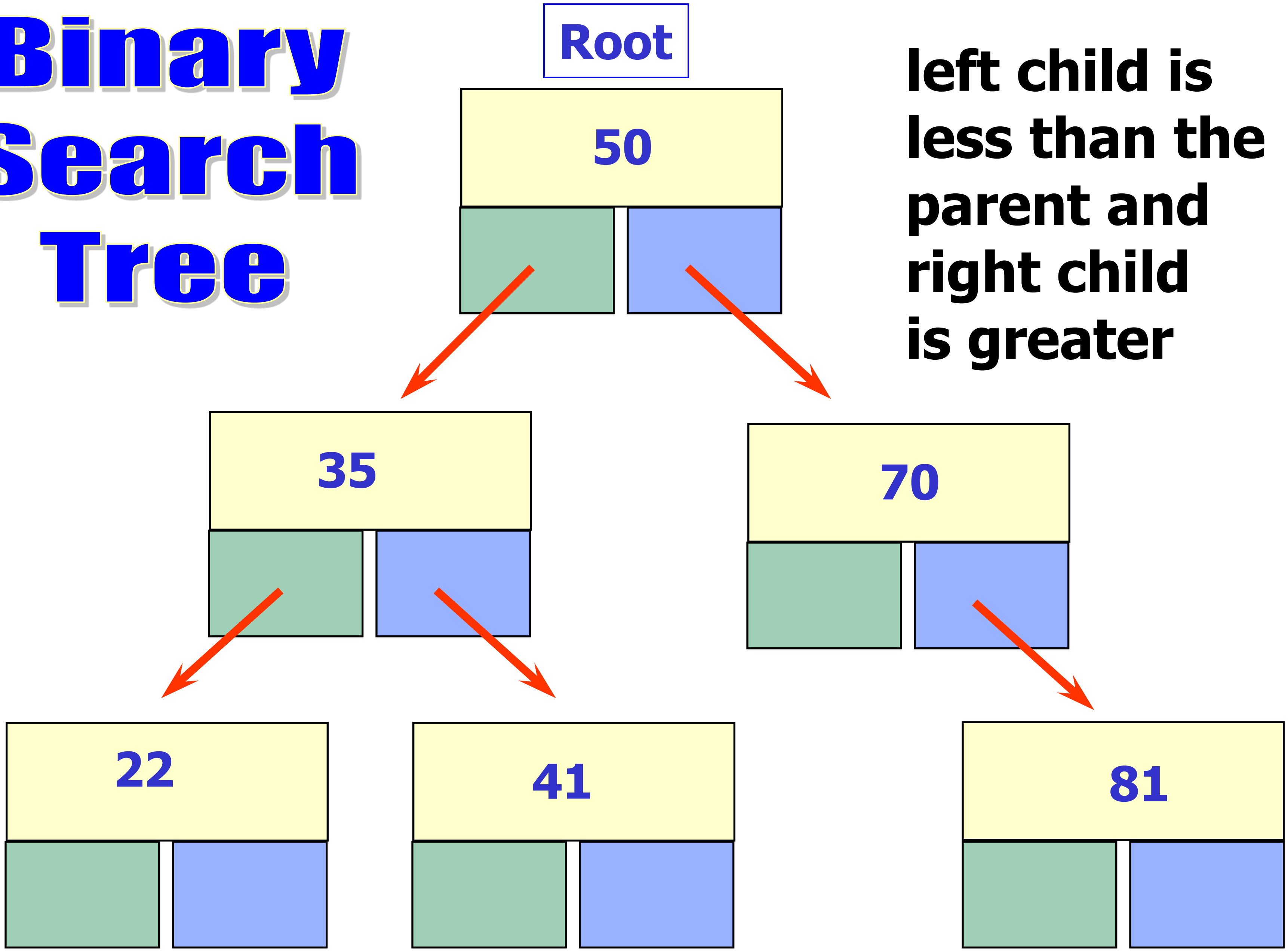
```
TreeNode x = new TreeNode("10",null,null);
TreeNode y = new TreeNode("25", null,null);
TreeNode z = new TreeNode("20", x, y);


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

OUTPUT

20
10
25

Building a Search Tree

Binary Search Tree



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

adding 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.

add - recursive 1

```
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)          do I go left?
    tree.setLeft(add(val, tree.getLeft()));
  else if(dirTest>0)     do I go right?
    tree.setRight(add(val, tree.getRight()));
  return tree;
}
```

Check to see which direction to go. Go left or right?

How does this work?

add - recursive 2

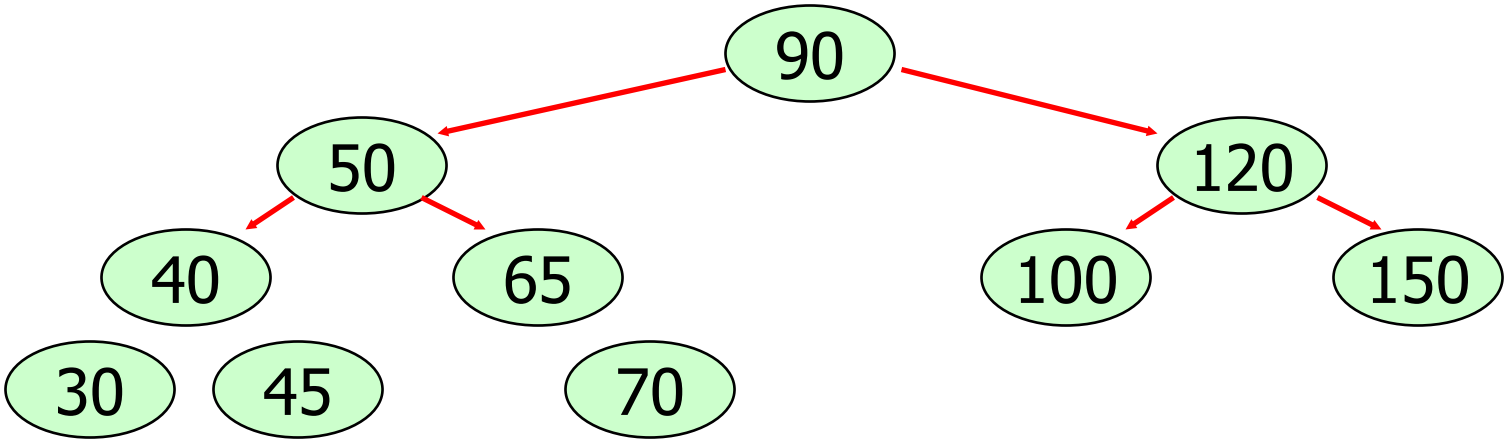
```
private TreeNode add(Comparable val, TreeNode tree)
{
  if (tree == null)
    tree = new TreeNode(val, null, null);
  else if (val.compareTo(tree.getValue()) < 0 )
    tree.setLeft(add(val, tree.getLeft()));
  else if (val.compareTo(tree.getValue()) > 0 )
    tree.setRight(add(val, tree.getRight()));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

Code works the same as 1, but is more compressed.

How does this work?

Printing a Search Tree

Tree Traversals



IN-ORDER = 30 40 45 50 65 70 90 100 120 150

PRE-ORDER = 90 50 40 30 45 65 70 120 100 150

POST-ORDER = 30 45 40 70 65 50 100 150 120 90

REV-ORDER = 150 120 100 90 70 65 50 45 40 30

Binary Tree Traversals

In-Order
LEFT
DATA
RIGHT

Pre-Order
DATA
LEFT
RIGHT

Post-Order
LEFT
RIGHT
DATA

Reverse-Order
RIGHT
DATA
LEFT

Searching a Tree

Searching Trees

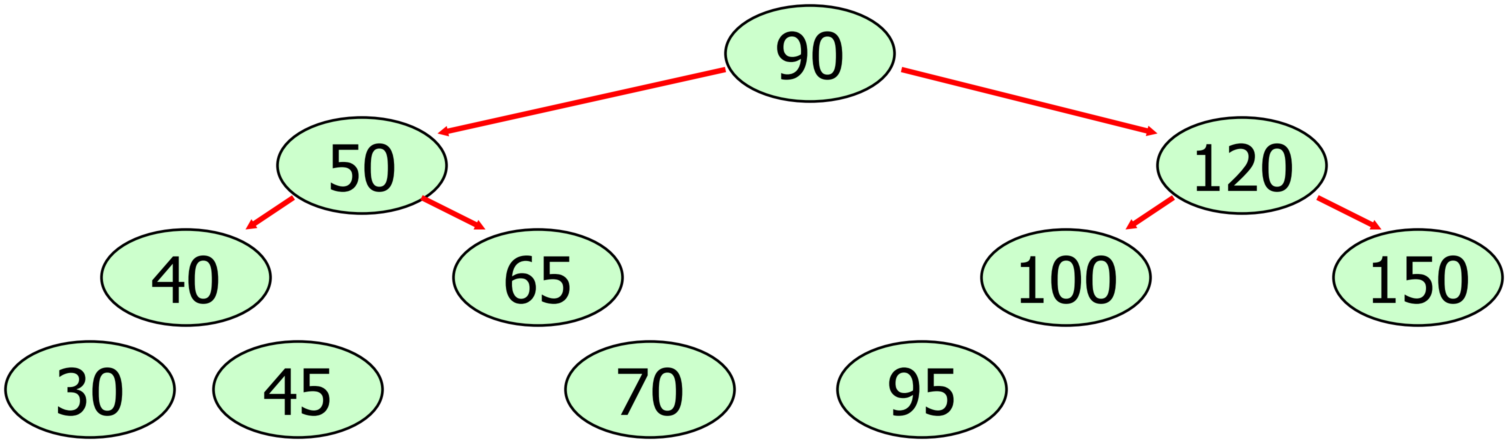
To search a tree, you will use the same basic logic that you used to add a new node.

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.

Tree Operations

Tree Operations



WIDTH - 7

HEIGHT - 3

NUMLEAVES - 5

NUMLEVELS - 4

NUMNODES - 11

ISFULL – NO

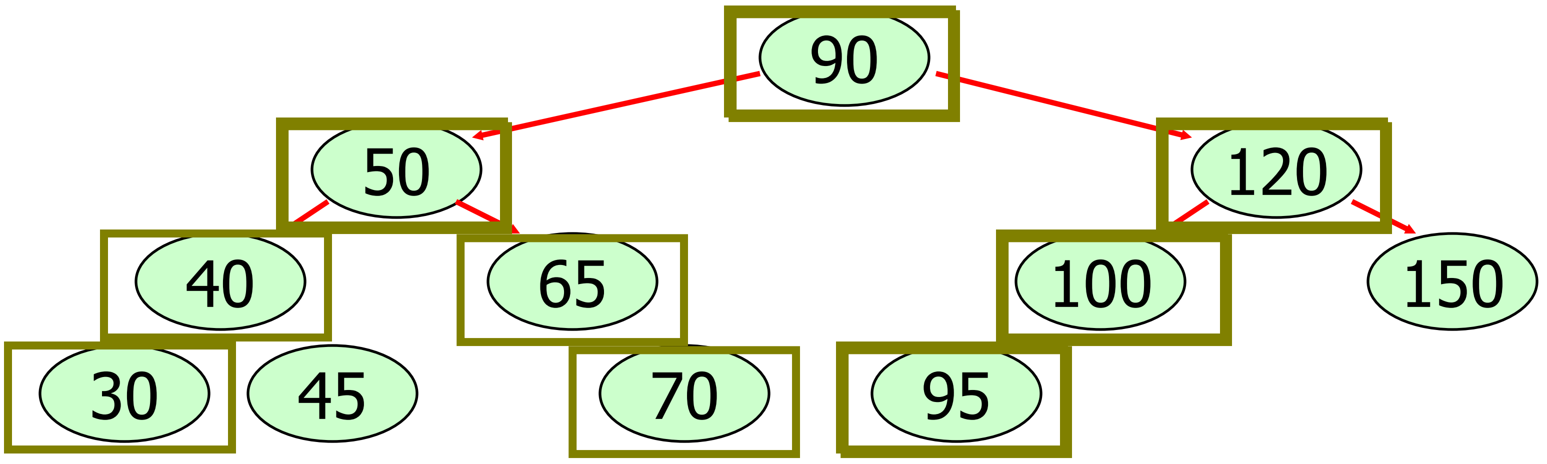
Binary Trees

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

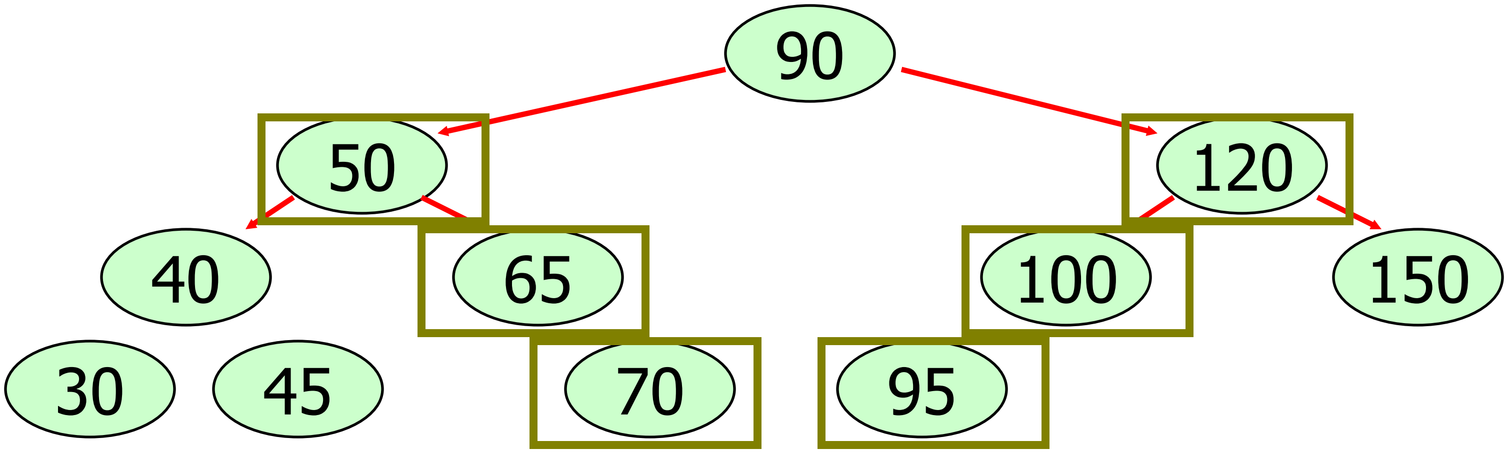
Level - a group of equal nodes
the root is level - 0
the children of the root - level - 1

Width



WIDTH - 7

Height



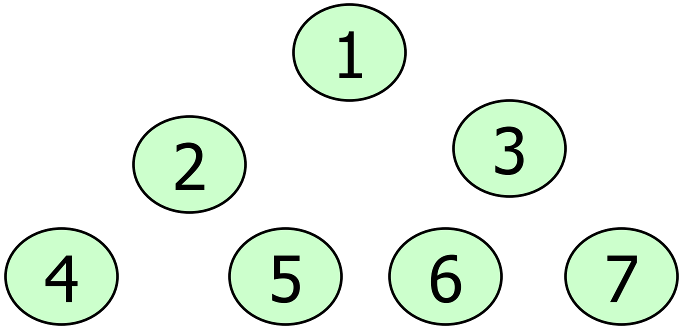
HEIGHT - 3

A FULL BINARY TREE

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.

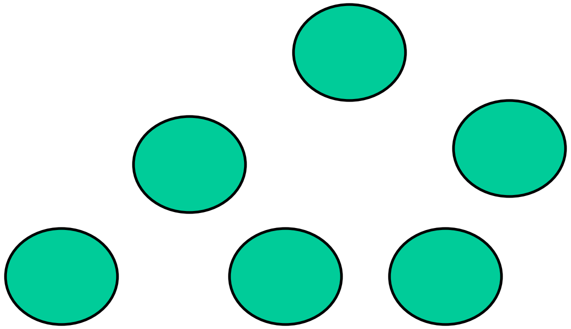
7 is the # of nodes

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



A Complete Tree

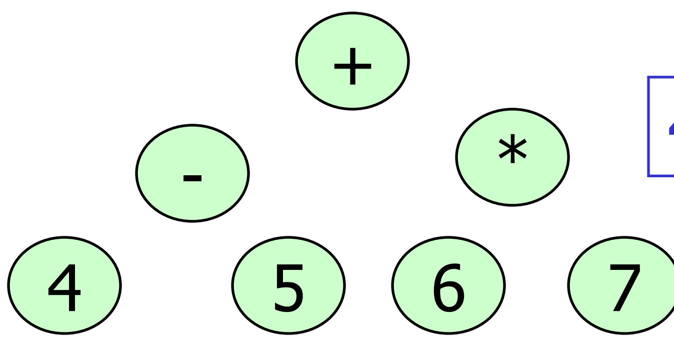
In a complete tree, every level that can be filled is filled. Any levels that are not full have all nodes shifted as far left as possible.



Fancy Trees

Expression Tree

A binary expression tree is a binary tree in which each parent node contains an operator and each leaf contains a number.



4-5+6*7

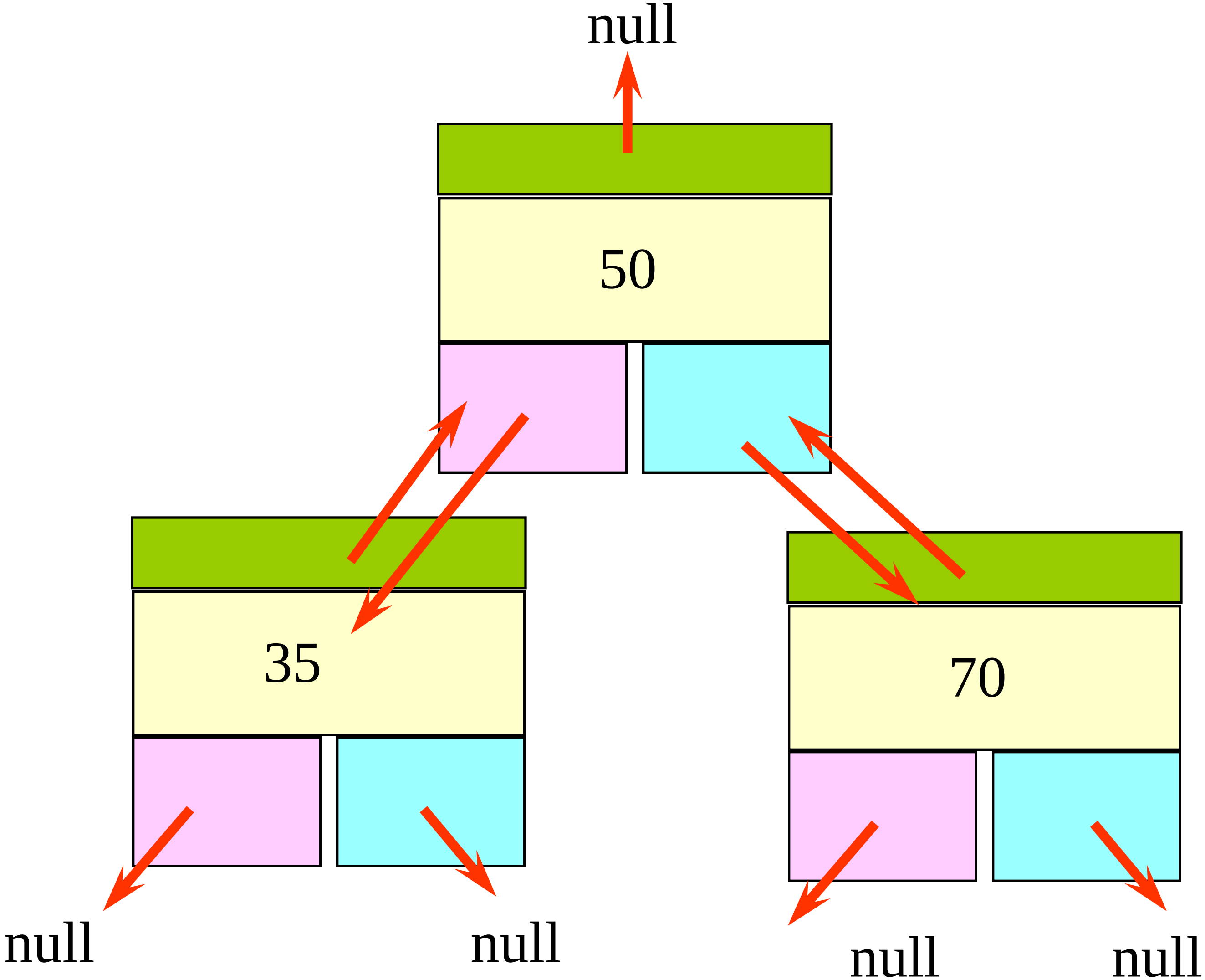
Threaded Tree

A threaded binary tree is a binary tree with an additional reference in each node that is used to point from a child back to its parent.

Threaded Tree

```
public class ThreadedTreeNode {

   private Comparable treeNodeValue;
   private ThreadedTreeNode leftTreeNode;
   private ThreadedTreeNode rightTreeNode;
   private ThreadedTreeNode parentTreeNode;

   //constructors and methods not shown

}
```

Threaded Tree



BigO

Big-O Notation

Big-O notation is an assessment of an algorithm's efficiency. Big-O notation helps gauge the amount of work that is taking place.

Common Big O Notations :

$O(1)$
$O(2^N)$
$O(N \, Log_2 N)$
$O(Log_2 N)$
$O(Log_2 N)$
$O(N^2)$
$O(N)$
$O(N^3)$

Binary Search Tree

traverse all nodes	O(N)
search for an item	$O(\log_2 N)$
remove any item location unknown	$O(\log_2 N)$
get any item location unknown	$O(\log_2 N)$
add item at the end	$O(\log_2 N)$
add item at the front	O(1)

A binary tree node has a reference to its left and right nodes. Nodes are ordered.

These notations assume the tree is balanced or near balanced.