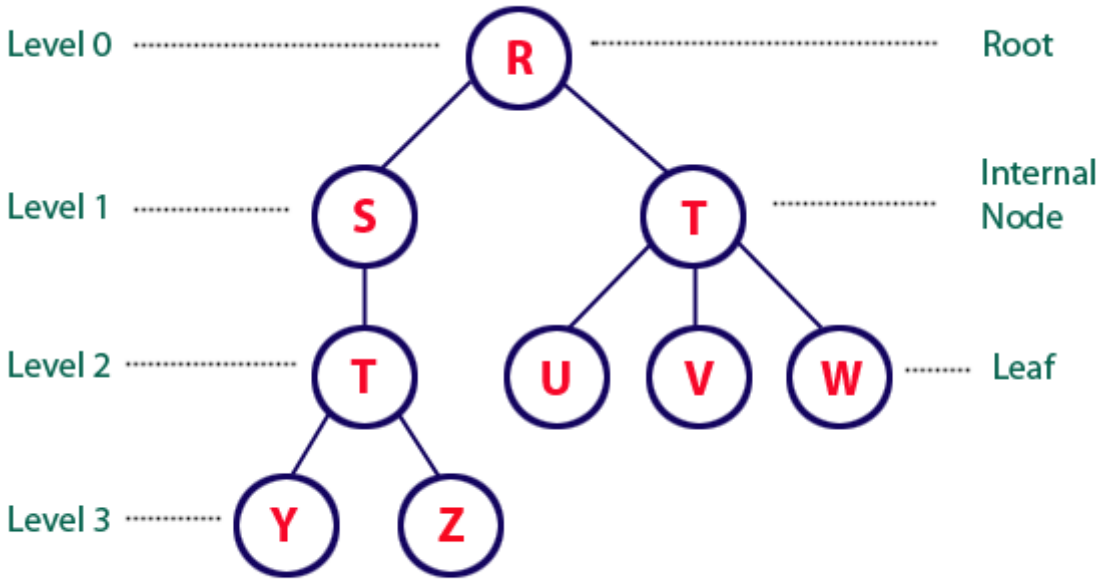
Java Binary Trees

Before getting into the nitty gritty of trees we need to start with a little terminology so that we all know what we are talking about. It may be best to explain the terminology with the help of a tree diagram:



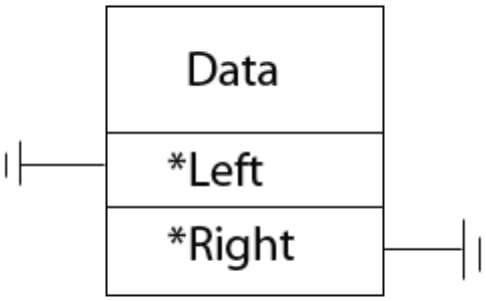
A tree diagram consists of nodes and line segments called **edges** or **branches that** make up family relations (like parent and child). Tree diagrams also consist of geometric relationships such as left, right, up, and down. Finally, we use biological names such as roots and leaves to describe entities of the tree structure.

At the top of any tree is the root node. This is depicted in the above diagram in the node labeled R. We can see that Node R has two children Nodes S and T which are connected to each other by branches (edges). At the bottom of the tree are Nodes U, V, W, Y, and Z. These are called leaf nodes because they have no children. So, by definition a leaf Node is a Node that has no children.

It should also be pointed out that trees have different levels. The top of the tree where the root node is would be level 0 with the level increasing as you move down the branches of the tree.

Binary Trees

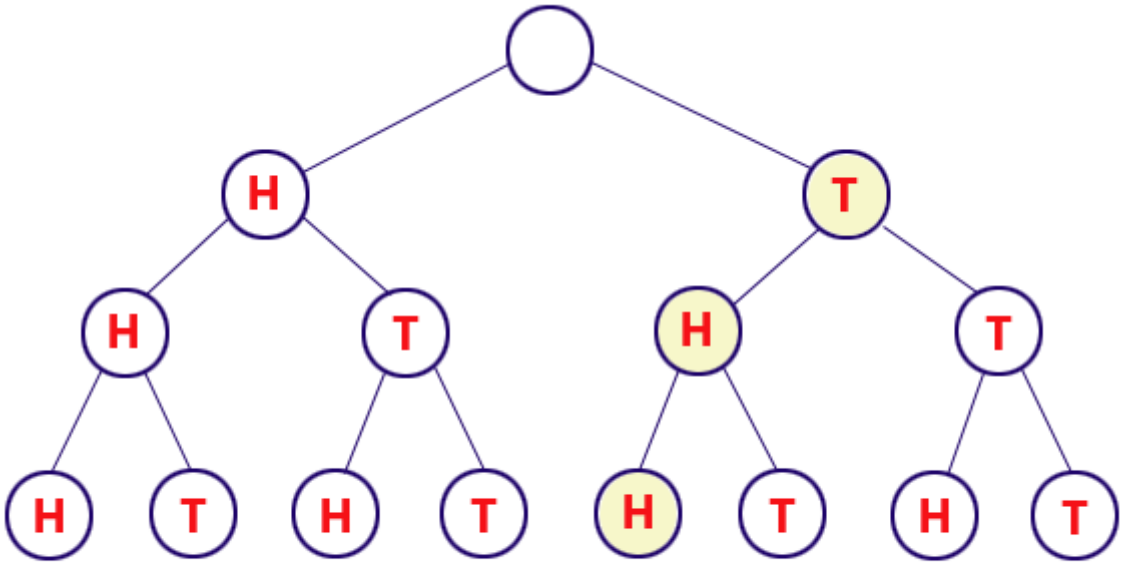
As you can probably guess a binary tree is a tree whose nodes can only have two children. We can conceptualize these nodes by borrowing what we saw with the linked lists:



Looking at the above node you are probably thinking that this looks a lot like a doubly linked list. The fact of the matter is that if you understand a linear linked list then understanding how a binary tree works should be pretty straight forward to you.

Applications

Binary trees are very useful for experiments which involve two possible outcomes like true or false or 1 and 0. An example of this is a tree that shows the possible outcomes of a coin toss:

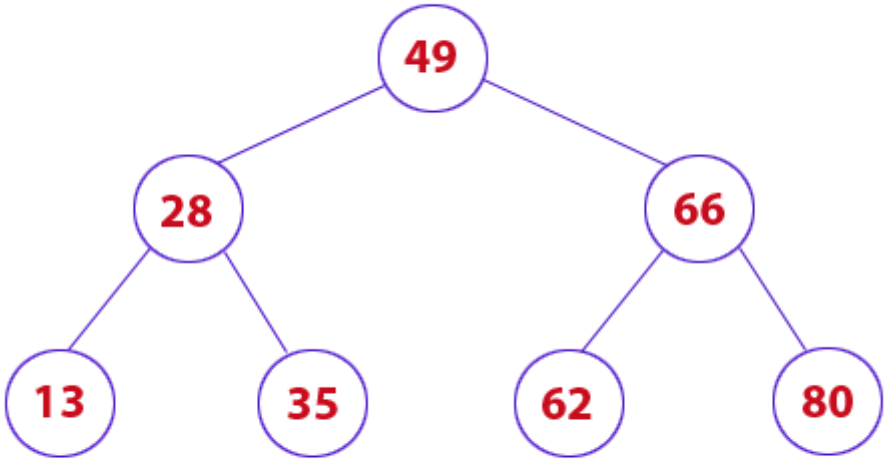


The above tree shows a simulation of three coin flips where the results are Tails, Heads, Heads.

What you should note is that while in this section we are going to look at a linked representation of a tree that it can also be created statically with an array.

Binary Search Trees

A binary search tree is a tree whose left subtree has values lesser values than it's root and the right subtree having values greater than its root. To understand this, it might be best to see an example:



At the root of the tree is a value of 49. As we traverse the left branch we see at the next level is 28 which is smaller that 49. Traversing the left branch further we find another smaller number. If we start at the root and traverse the right branch we see the numbers get larger.

Complexity:
From a complexity standpoint the binary search tree is pretty efficient. Especially if you consider that a linear search has a complexity of **O(n)**. Because the binary tree eliminates half of the numbers at each traversal the complexity is$\mathbf{O(\log_2 n)}$.

Sample Code

At this point it is probably best to look at some sample code. What is being shown here is a modified version of what is being presented in the text. For the sake of simplicity the data section of this tree will be of an int type. Also in this example recursion will be used to traverse the tree. In my opinion it is a much more elegant way of doing things.

Binary Search Tree Operations

There are several operation that need to be implemented to make the binary search tree a useful data structure:

Binary Tree Functions

Operation	Description
insert	Insert a node in the proper location of the tree
remove	Remove a specific node from thetree
contains	Report if the tree contains a specific piece of data.
findMin	Find and return the smallest value in the tree.
findMax	Find and return the largest value in the tree.
isEmpty	Determine if the tree is empty.
printTree	Visit each node in the tree and print the data to the screen.

Creating The Tree

The tree will basically be created in the insert method. Here we want to present the node and the constructor to set the Root of the tree to NULL to indicate that the tree is empty:

```
public class Node
{
    int Data;

    Node left;
    Node right;

    Node(int data)
    {
        this.Data = data;
    }
}
```

Notice that a constructor has been added that sets the data section. This is being done for convenience sake. It will simplify things by having the functionality to set the data section at the time a Node is created.

The constructor for the Tree class is pretty straight forward:

```
public class SearchTree
{
Node Root;

    public SearchTree()
    {
        this.Root = null;
    }

    public static void main(String[] args)
    {
        //TODO code application logic here
    }
}
```

Here we simply need to set the Root of the tree to NULL to indicate that the tree is empty. At this point you have a node that looks like this:

isEmpty

This makes the isEmpty function an easy thing to implement:

```
public boolean isEmpty()
{
    return this.Root == null;
}
```

Insert Into Tree

Because I am taking a recursive approach to creating the binary tree I am creating overloaded functions for most things. One will serve as an interface into the tree and the other will do the actual work. The over loaded functions that serve as the interface will be placed in the public section and those that do the actual work will be placed in the private section. The reason for separating these into public / private is that we want to hide the implementation and only publish the interface. Hopefully this will become clear as you see some code:

```
private void insert(int data, Node n)
{
    if(this.Root == null)
    {
        this.Root = new Node(data);
    }
    else if(data < n.Data)
    {
        if(n.left != null)
            insert(data, n.left);
        else
            n.left = new Node(data);
    }
    else if(data > n.Data)
    {
```

```
            if(n.right != null)
                insert(data, n.right);
            else
                n.right = new Node(data);
        }
    }
```

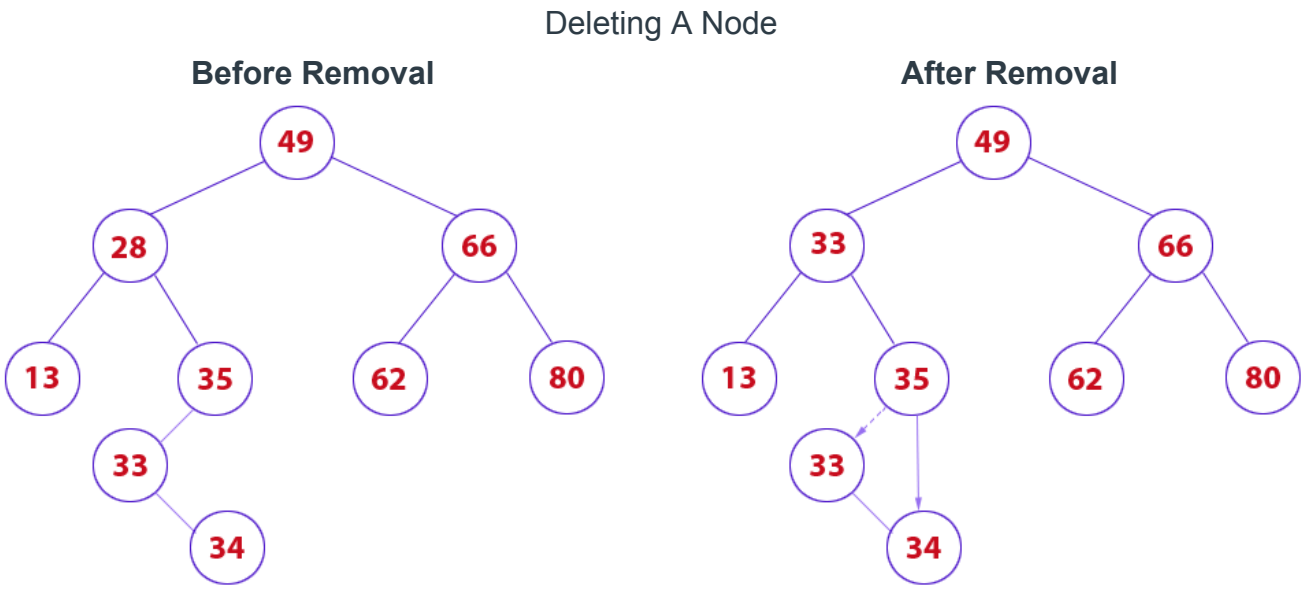
What is being done above is that the correct path is chosen (smaller to the left, larger to the right) until the end of the path is found and then a new node is created and added to that location.

Interface
The public interface to this method looks like this:

```
public void insert(int data)
{
    this.insert(data, this.Root);
}
```

Remove From Tree

Removing from a tree is similar to inserting into the tree in that the tree is traversed by selecting the proper path to go down until the value to removed is found. Deleting can be difficult because you have to be concerned with left and right child nodes. Of course if the node is a leaf removing is somewhat easy.
The typical strategy of removing a node that has both left and right sub-nodes is to replace the data of the node with the data from the smallest data of the right subtree. Once replaced the empty node is deleted by recursively searching for it. This can be seen in the following diagram:



In the above we want to remove the node that has 28 as the data. The right sub-tree is traversed and the smallest value is then copied to the node being removed (28 in this case). Then what is left is to remove the node that was copied. Since it was a leaf node it becomes easy to remove. The following code removes nodes from the tree:

```
private Node remove(int data, Node n)
{
    if(this.Root == null)
        return n; //empty tree do nothing
    if(data < n.Data)
        n.left = this.remove(data, n.left);
    else if(data > n.Data)
        n.right = this.remove(data, n.right);
    else if(n.left != null && n.right != null)
    {
        n.Data = findMin(n.right);
        remove(n.Data, n.right);
    }
    else
    {
        n = (n.left  != null) ? n.left : n.right;
    }
    return n;
}
```

The overloaded interface for this function only takes the data for the node to remove:

```
public void remove(int data)
{
    this.remove(data, this.Root);
}
```

Find Min and Max Values

Finding the min or max values in the tree simply involves traversing the correct path until the leaf is found:

```
private int findMin(Node n)
{
    if(n.left == null)
        return n.Data;
    else
        return findMin(n.left);
}
```

To find the max you simply traverse the right sub-tree. The findMin interface function is as follows:

```
public int findMin()
{
    return this.findMin(this.Root);
}
```

Checking If an Item Exists In the Tree

The contains function traverses the proper path until it finds a match. If NULL is reached before the the data is found the function simply returns true. Here is both the public and private methods:

```
private boolean contains(int data, Node n)
{
    if(n == null)
        return false;
    else if(data < n.Data)
        return contains(data, n.left);
    else if(data > n.Data)
        return contains(data, n.right);
    else
        return true;
}

public boolean contains(int data)
{
    return this.contains(data, this.Root);
}
```

Printing The Tree

Printing the tree involves recursively traversing the left and right sub-tree printing the value of each node as it is visited. In this example we traverse the left path first but you could easily start on the right:

```
private void printTree(Node n)
{
    if(n != null)
    {
        System.out.println(n.Data);
        this.printTree(n.left);
        this.printTree(n.right);
    }
}
public void printTree()
{
    this.printTree(this.Root);
}
```

Binary Trees Finally

If you were able to get a good grasp on recursion then these functions probably seem pretty straight forward to you.

The complete code for the tree presented in this lecture can be found here:

```
package searchtree;
```

```
/**
 *
 * @author gstev_000
 */
public class SearchTree
{
Node Root;

    public SearchTree()
    {
        this.Root = null;
    }
    public boolean isEmpty()
    {
        return this.Root == null;
    }
    private void insert(int data, Node n)
    {
        if(this.Root == null)
        {
             this.Root = new Node(data);
        }
        else if(data < n.Data)
        {
            if(n.left != null)
                insert(data, n.left);
            else
                n.left = new Node(data);
        }
        else if(data > n.Data)
        {
            if(n.right != null)
                insert(data, n.right);
            else
                n.right = new Node(data);
        }
    }
    public void insert(int data)
    {
        this.insert(data, this.Root);
    }
    private Node remove(int data, Node n)
    {
        if(this.Root == null)
            return n; //empty tree do nothing
        if(data < n.Data)
           n.left = this.remove(data, n.left);
        else if(data > n.Data)
           n.right = this.remove(data, n.right);
        else if(n.left != null && n.right != null)
        {
            n.Data = findMin(n.right);
            remove(n.Data, n.right);
        }
        else
        {
           n = (n.left != null) ? n.left : n.right;

        }
        return n;
    }
    public void remove(int data)
    {
        this.remove(data, this.Root);
    }
```

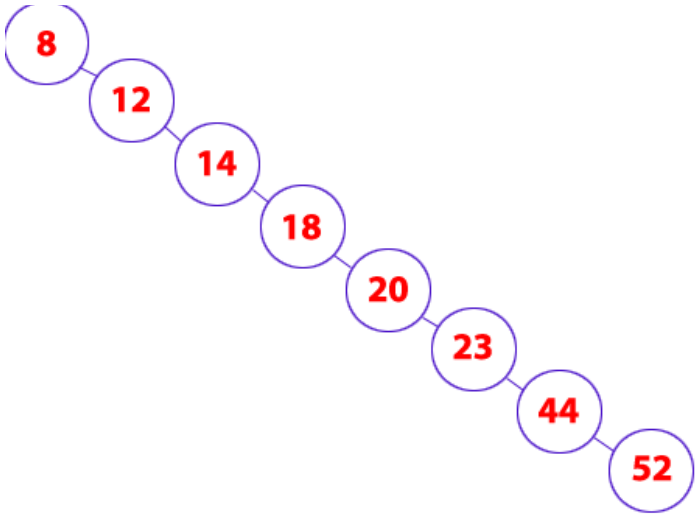
```
    private int findMin( Node n )
    {
        if( n.left == null )
            return n.Data;
        else
            return findMin( n.left );
    }
    public int findMax()
    {
        return this.findMax(this.Root);
    }

    public int findMin()
    {
        return this.findMin(this.Root);
    }
    private int findMax( Node n )
    {
        if( n.right == null )
            return n.Data;
        else
            return findMax(n.right);
    }
    private boolean contains(int data, Node n)
    {
         if(n == null)
             return false;
         else if(data < n.Data)
             return contains(data, n.left);
         else if(data > n.Data)
             return contains(data, n.right);
         else
             return true;
    }
    public boolean contains(int data)
    {
        return this.contains(data, this.Root);

    }
    private void printTree(Node n)
    {
        if(n != null)
        {
            System.out.println(n.Data);
            this.printTree(n.left);
            this.printTree(n.right);
        }

    }
    public void printTree()
    {
        this.printTree(this.Root);
    }

    public static void main(String args)
    {
        SearchTree st = new SearchTree();
        st.insert(10);
        st.insert(15);
        st.insert(20);
        st.insert(12);
        st.insert(25);
         if(st.contains(20))
            System.out.println("Found");
```

```
        else
            System.out.println("Not Found");
        st.printTree();
        System.out.println("Remove Node\n");
        st.remove(20);
        st.printTree();

         if(st.contains(20))
            System.out.println("Found");
        else
            System.out.println("Not Found");
    }
}
```

```
public static void main(String[] args)
{
    SearchTree st = new SearchTree();
    st.insert(10);
    st.insert(15);
    st.insert(20);
    st.insert(12);
    st.insert(25);
    if(st.contains(20))
        System.out.println("Found");
    else
        System.out.println("Not Found");
    st.printTree();
    System.out.println("Remove Node\n");
    st.remove(200);
    st.printTree();

    if(st.contains(20))
        System.out.println("Found");
    else
        System.out.println("Not Found");
```

Java AVL Trees

AVL trees were created in 1962 by two Russian mathematicians, G.M. Andelson-Velsik and E.M. Landis. As you can see the tree structure was named after them. The concept presented in the binary tree are very relevant to those of the AVL. The idea behind an AVL tree is that the tree be balanced. Why would balancing a tree be so important. Well, take a look at the following tree:



As you can see, this is nothing more than a linear list which gives the search efficiency a value of **O(n)** which is really not all that efficient and defeats the purpose of using a tree structure. What we need to do is balance the tree to make it more efficient.

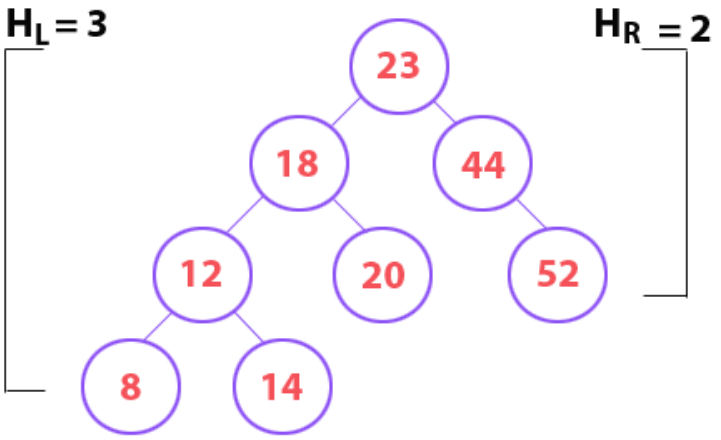
Definition:

An AVL tree is a tree that is either empty or consists of two AVL sub-trees T_L and T_R whose height differs by no more than one. Given this we can create at the formula.

$|H_L - H_R| <= 1$

To check the balance of an AVL tree. If you are unsure about what this formula means it is simply that the absolute difference of the height of the left sub-tree and the right sub-tree be no more than one.

Let's take a look at the above tree after being balanced:



As you can see the height of the left tree is 3 and the height of the right tree is 2. This gives us a difference of 1 (**balance factor of 1**) . While this is not 100% balanced it does put it in the range of what a balanced AVL tree should be.

Balancing Trees

Any time you add a node to the tree there is a possibility that the tree will become unbalanced. There are basically four cases you need to consider to balance the tree:

Tree Balance Cases

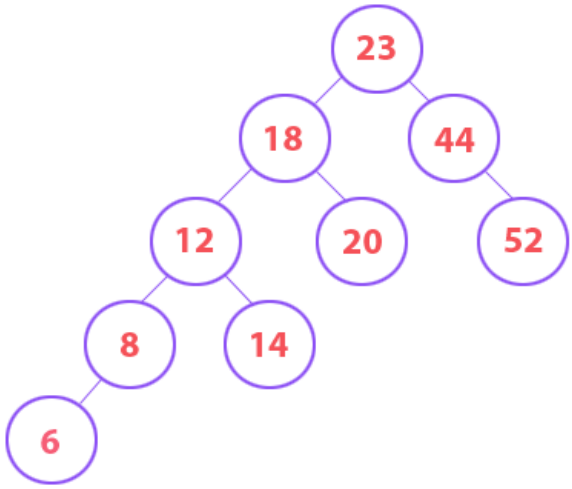
Case	Description
Left of Left	A subtree of a tree that is left high also become left high.
Right of Right	A subtree of a tree that is right high also has also become right high.
Right of Left	A subtree of a tree that is left high has become right high.
Left of Right	A subtree of a tree that is right high has become left high.

Rotations

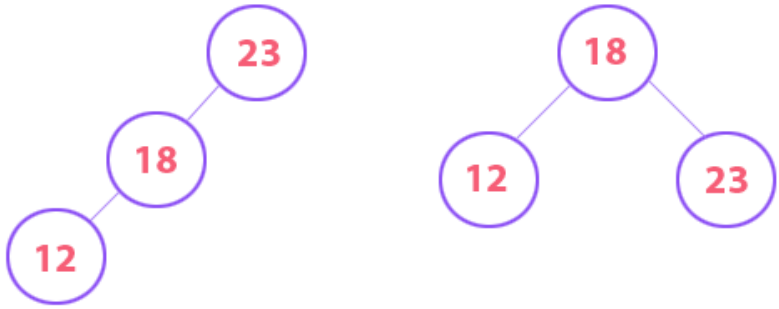
When a tree becomes out of balance it can be put back into balance by rotating the nodes. To understand what is meant by rotating let's take each of the four cases and show how rotating the nodes can put the tree back into balance.

Left of Left

When the tree becomes out of balance because of a left insertion of a left high subtree. The tree can be brought back into balance by rotating the node to the right. Let's take the above balanced tree and add a value of 6 to it which will increase the height of the left subtree by 1:



In the above we can balance the tree by rotating the out of balance node to the right. As a simple example let's look at rotating the root node to the right:



Out of balance Right rotation

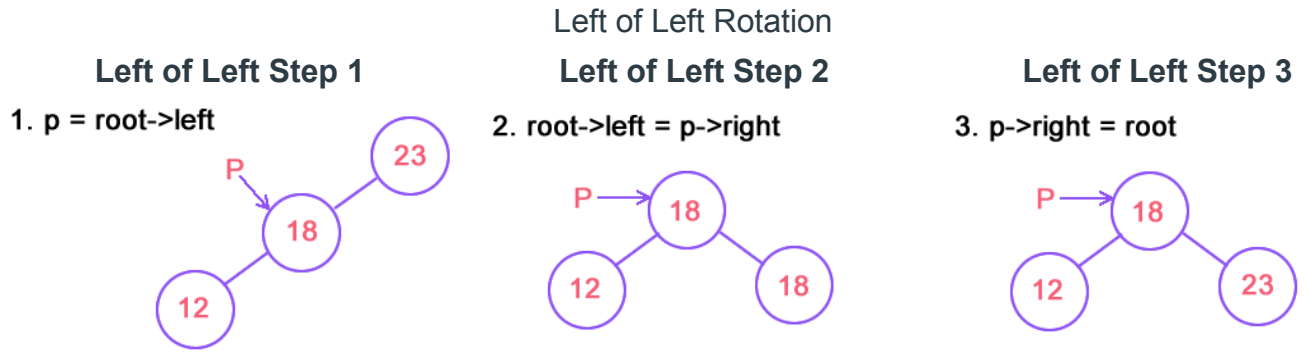
This rotation is achieved using the following steps

1. Node 18 becomes the new root
2. Node 23 takes ownership of Node 18 right child as its left child. The value is NULL.
3. Node 18 takes ownership of Node 12 as it's right child.

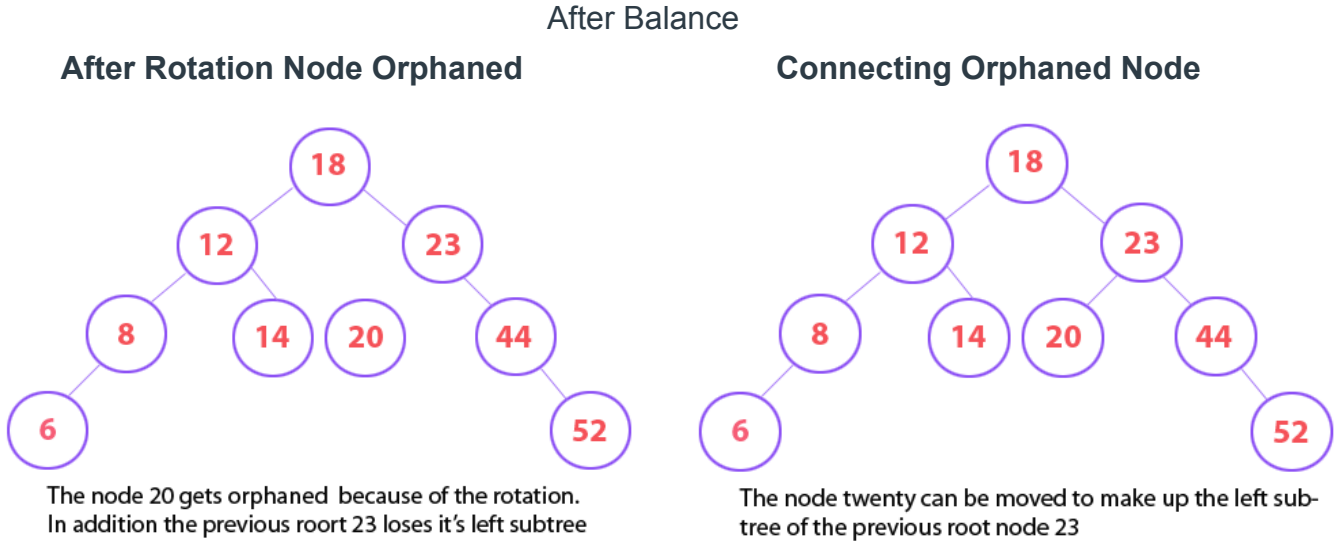
The code to achieve this is as follows:

```
private Node rotateRight(Node root)
{
    Node p = root.left;
    root.left = p.right;
    p.right = root;
    return p;
}
```

Let's break this rotation down step-by-step:

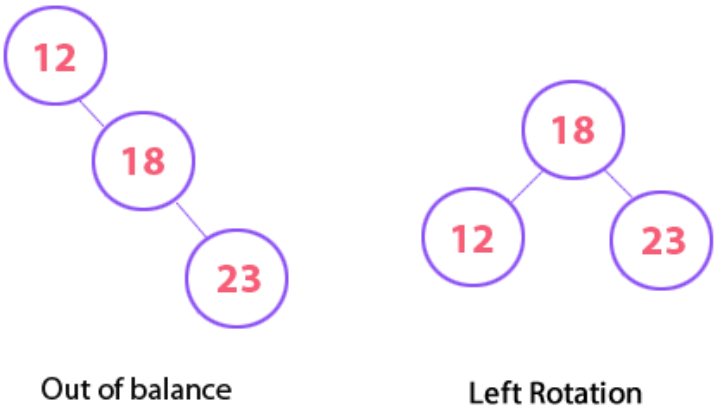


After this rotation the tree now looks like this:



After the rotation there is a problem What to do with the node containing 23. When the root node 23 is rotated to the right it loses it's left subtree. The node containing 20 can then simply be attached as left sub-tree of node 23.

It should be noted that a left rotation is simply a mirror of the right:



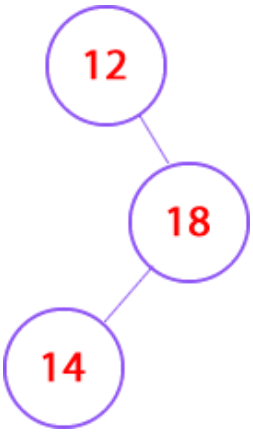
The code for a left rotation is then:

```
private Node rotateLeft(Node root)
{
    Node p = root.right;
    root.left = p.left;
    p.left = root;
    return p;
}
```

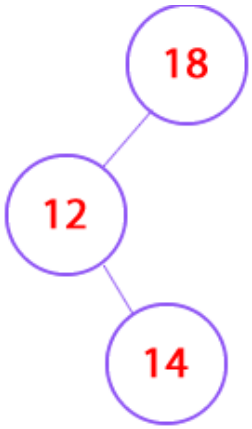
Double Rotations

Left RIght (Double Left) Rotation)

There are times when a single rotation is not enough. Take a look at the following example:

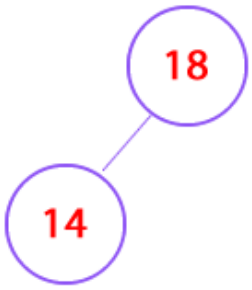


You would think by looking at this that this can be balanced by performing a left rotation. Let's take a look at what happens after performing that action:



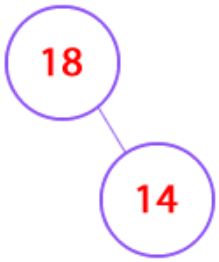
After the completion of the rotation we simply have the same situation, only a mirror of the previous sub-tree. The reason for this situation is that the right sub-tree was left heavy (had a negative balance) so performing a left rotation had no effect.

To solve this problem we must perform a right rotation on the right sub-tree. This means that we are **not** rotating on the current root, we are rotating on the right child.



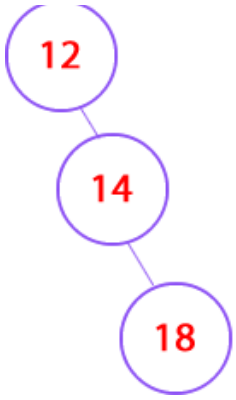
Original Right Sub-tree

After performing a right rotation on the right sub-tree we end up with:

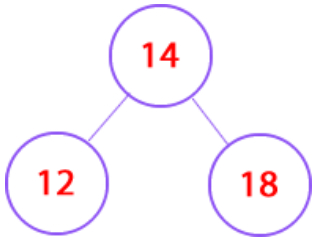


After Rotation

This sets us up to make a left rotation starting from the root:

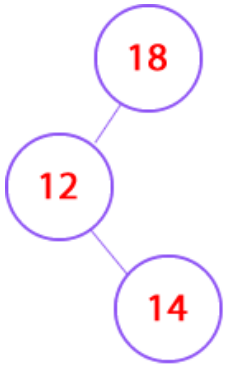


After this rotation we end up with a balanced sub-tree:

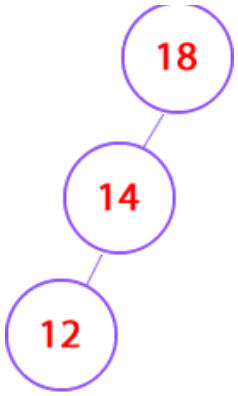


Right-left (Double Left) Rotation:

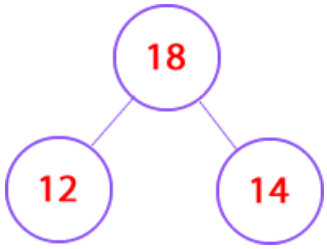
A double right (right-left) rotation is needed when balancing a tree which has a left sub-tree that is right heavy. The double left rotation is simply a mirror of the double right rotation:



Again if we make a right rotation on the root we simply end up with a mirror of what we have now. The solution then is to first perform a left rotation on the left sub-tree. This will leave us with with a left heavy sub-tree:



The sub-tree can now be balanced by performing a right rotation at the root node 18:



You may think that creating these double rotate functions is going to be a difficult task. In reality we have already developed left and right rotate functions. All we need to do is pass the appropriate node to rotate from.

Insert

Inserting into an AVL Tree is very similar to inserting into a binary tree. The big difference is that you need to keep track of the height of the tree. There are several ways to do this but probably the easiest way is to simply add a variable to the Node to hold the height. Here is the Node for the AVLTree:

```
public class AvlNode
{
    int element;
    AvlNode left;
    AvlNode right;
    int height;

    AvlNode(int data, AvlNode r, AvlNode l)
    {
        this.left = l;
        this.right = r;
        this.element = data;
    }
}
```

A couple of other things to notice, the first is that a class is used instead of a struct. Using a class allows us to add a constructor which is useful for adding data to the Node at the time it is created.

The other thing to notice that the AVLTree class has been made a friend of the AVLNode class. This will allow the Node class to have access to the private data of the AVLTree class. While using a friend may not seem like the best way to approach this, it is still better than making parts of the AVLNode class public.

Inserting an AVLNode into the tree then is a pretty straight forward thing to do. Here is code that shows how to do it:

```
private AvlNode insert(int x, AvlNode t)
{
    if(t == null)
        return new AvlNode(x, null, null);

    if(x < t.element)
        t.left = insert(x, t.left);
    else if(x > t.element)
        t.right = insert(x, t.right);

    return balance(t);
}
```

The second if statement checks to see if the data being inserted is smaller. If it is then the left sub-tree is traversed. The else if statement does the exact opposite. It traverses the right sub-tree if the data being inserted is greater than the right. The last call to the method balance checks the height of the tree and performs rotation if necessary:

```
private AvlNode insert(AvlNode t)
{
    if(t == null)
        return t;

    if(height(t.left) - height(t.right) > ALLOWED_IMBALANCE)
        if(height(t.left.left) >= height(t.left.right))
            t = rotateWithLeftChild(t);
        else
            t = doubleWithLeftChild(t);
    else
    if(height(t.right) - height(t.left) > ALLOWED_IMBALANCE)
        if(height(t.right.right) >= height(t.right.left))
            t = rotateWithRightChild(t);
        else
            t = doubleWithRightChild(t);

    t.height = Math.max(height(t.left), height(t.right)) + 1;
    return t;
}
```

The height method is pretty simple. It simply return the height of the node or -1 if the node is null. Here is the code:

```
private int height(AvlNode t)
{
```

```
    return t == null ? -1 : t.height;
}
```

Removing A Node

Removing a node is a little more tricky than inserting because you have to take into account a leaf node (Node with no children) This needs to be handled differently. Here is the remove function:

```
private AvlNode remove(int x, AvlNode t)
{
    if(t == null)
        return t; //Item not found; do nothing

    if(x < t.element)
        t.left = remove(x, t.left);
    else if(x > t.element)
        t.right = remove(x, t.right);
    else if(t.left != null && t.right != null) //Two children
    {
        t.element = findMin(t.right).element;
        t.right = remove(t.element, t.right);
    }
    else //one or no children
        t = (t.left != null ) ? t.left : t.right;
    return balance(t);
}
```

Summary

A tree is considered unbalanced if the root node has a **balance factor** of 2. This means that height of one sub-tree has at least two more nodes than the height of the other. We can follow a few simple rules to determine which rotation we should use:

IF tree is right heavy

{

  IF tree's right subtree is left heavy

  {

    Perform Double Left rotation

  }

  ELSE

  {

    Perform Single Left rotation

  }

}

ELSE IF tree is left heavy

{

IF tree's left subtree is right heavy

{

Perform Double Right rotation

}

ELSE

{

Perform Single Right rotation

}

}

Below is a video set and some sample code on how to create an AVL tree. The code for AVL trees can be a little convoluted because you are having to keep track of the balance factor when nodes are added and removed.

Code Listing

The following shows the complete listing of the AVL Tree:

```
package avltree;

public class AvlTree
{
    private AvlNode root;

    public AvlTree( )
    {
        root = null;
    }

    /**
     * Insert into the tree; duplicates are ignored.
     *
     */
    public void insert( int x )
    {
        root = insert( x, root );
    }

    /**
     * Remove from the tree. Nothing is done if x is not found.
     *
     */
    public void remove( int x )
    {
        root = remove( x, root );
```

```
    }

    /**
     * Internal method to remove from a subtree.
     * @param x the item to remove.
     * @param t the node that roots the subtree.
     * @return the new root of the subtree.
     */
    private AvlNode remove( int x, AvlNode  t )
    {
        if( t == null )
            return t;   // Item not found; do nothing

        if( x < t.element )
            t.left = remove( x, t.left );
        else if( x  > t.element )
            t.right = remove( x, t.right );
        else if( t.left != null && t.right != null ) // Two children
        {
            t.element = findMin( t.right ).element;
            t.right = remove( t.element, t.right );
        }
        else  // one or no children
            t = ( t.left != null ) ? t.left : t.right;
        return balance( t );
    }

    /**
     * Find the smallest item in the tree.
     * @return smallest item or null if empty.
     */
    public int findMin( )
    {
        return findMin( root ).element;
    }

    /**
     * Find the largest item in the tree.
     * @return the largest item of null if empty.
     */
    public int findMax( )
    {
        return findMax( root ).element;
    }

    /**
     * Find an item in the tree.
     * @param x the item to search for.
     * @return true if x is found.
     */
    public boolean contains( int x )
    {
        return contains( x, root );
    }

    /**
     * Make the tree logically empty.
     */
    public void makeEmpty( )
    {
        root = null;
    }

    /**
     * Test if the tree is logically empty.
```

```
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return root == null;
    }

    /**
     * Print the tree contents in sorted order.
     */
    public void printTree( )
    {
        if( isEmpty( ) )
            System.out.println( "Empty tree" );
        else
            printTree( root );
    }

    private static final int ALLOWED_IMBALANCE = 1;

    // Assume t is either balanced or within one of being balanced
    private AvlNode balance( AvlNode t )
    {
        if( t == null )
            return t;

        if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )
            if( height( t.left.left ) >= height( t.left.right ) )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
        else
        if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )
            if( height( t.right.right ) >= height( t.right.left ) )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );

        t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
        return t;
    }

    public void checkBalance( )
    {
        checkBalance( root );
    }

    private int checkBalance( AvlNode t )
    {
        if( t == null )
            return -1;

        if( t != null )
        {
            int hl = checkBalance( t.left );
            int hr = checkBalance( t.right );
            if( Math.abs( height( t.left ) - height( t.right ) ) > 1 ||
                    height( t.left ) != hl || height( t.right ) != hr )
                System.out.println( "OOPS!!" );
        }

        return height( t );
    }
```

```
    /**
     * Internal method to insert into a subtree.
     * @param x the item to insert.
     * @param t the node that roots the subtree.
     * @return the new root of the subtree.
     */
    private AvlNode insert( int x, AvlNode t )
    {
        if( t == null )
            return new AvlNode( x, null, null );

        if( x < t.element )
            t.left = insert( x, t.left );
        else if( x  > t.element )
            t.right = insert( x, t.right );

        return balance( t );
    }

    /**
     * Internal method to find the smallest item in a subtree.
     * @param t the node that roots the tree.
     * @return node containing the smallest item.
     */
    private AvlNode findMin( AvlNode t )
    {
        if( t == null )
            return t;

        while( t.left != null )
            t = t.left;
        return t;
    }

    /**
     * Internal method to find the largest item in a subtree.
     * @param t the node that roots the tree.
     * @return node containing the largest item.
     */
    private AvlNode findMax( AvlNode t )
    {
        if( t == null )
            return t;

        while( t.right != null )
            t = t.right;
        return t;
    }

    /**
     * Internal method to find an item in a subtree.
     * @param x is item to search for.
     * @param t the node that roots the tree.
     * @return true if x is found in subtree.
     */
    private boolean contains( int x, AvlNode t )
    {
        while( t != null )
        {

            if( x < t.element )
                t = t.left;
            else if( x > t.element )
                t = t.right;
            else
```

```
                return true;    // Match
            }

            return false;   // No match
        }

        /**
         * Internal method to print a subtree in sorted order.
         * @param t the node that roots the tree.
         */
        private void printTree( AvlNode t )
        {
            if( t != null )
            {
                printTree( t.left );
                System.out.println( t.element );
                printTree( t.right );
            }
        }

        /**
         * Return the height of node t, or -1, if null.
         */
        private int height( AvlNode t )
        {
            return t == null ? -1 : t.height;
        }

        /**
         * Rotate binary tree node with left child.
         * For AVL trees, this is a single rotation for case 1.
         * Update heights, then return new root.
         */
        private AvlNode rotateWithLeftChild( AvlNode k2 )
        {
            AvlNode k1 = k2.left;
            k2.left = k1.right;
            k1.right = k2;
            k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
            k1.height = Math.max( height( k1.left ), k2.height ) + 1;
            return k1;
        }

        /**
         * Rotate binary tree node with right child.
         * For AVL trees, this is a single rotation for case 4.
         * Update heights, then return new root.
         */
        private AvlNode rotateWithRightChild( AvlNode k1 )
        {
            AvlNode k2 = k1.right;
            k1.right = k2.left;
            k2.left = k1;
            k1.height = Math.max( height( k1.left ), height( k1.right ) ) + 1;
            k2.height = Math.max( height( k2.right ), k1.height ) + 1;
            return k2;
        }

        /**
         * Double rotate binary tree node: first left child
         * with its right child; then node k3 with new left child.
         * For AVL trees, this is a double rotation for case 2.
         * Update heights, then return new root.
         */
        private AvlNode doubleWithLeftChild( AvlNode k3 )
```

```
    {
        k3.left = rotateWithRightChild( k3.left );
        return rotateWithLeftChild( k3 );
    }

    /**
     * Double rotate binary tree node: first right child
     * with its left child; then node k1 with new right child.
     * For AVL trees, this is a double rotation for case 3.
     * Update heights, then return new root.
     */
    private AvlNode doubleWithRightChild( AvlNode k1 )
    {
        k1.right = rotateWithLeftChild( k1.right );
        return rotateWithRightChild( k1 );
    }
        // Test program
    public static void main( String [ ] args )
    {
        AvlTree avl= new AvlTree();
        avl.insert(23);
        avl.insert(18);
        avl.insert(44);
        avl.insert(6);
        avl.insert(12);
        avl.insert(52);
        avl.insert(14);
        avl.insert(8);

        avl.printTree();
        System.out.println("After Remove");

        avl.remove(52);
        avl.printTree();
    }
}
```

```
package avltree;

public class AvlNode
{
    int element;
    AvlNode left;
    AvlNode right;
    int height;

    AvlNode(int data, AvlNode r, AvlNode l)
    {
        this.left = l;
        this.right = r;
        this.element = data;
    }
}
```

Java Splay Trees

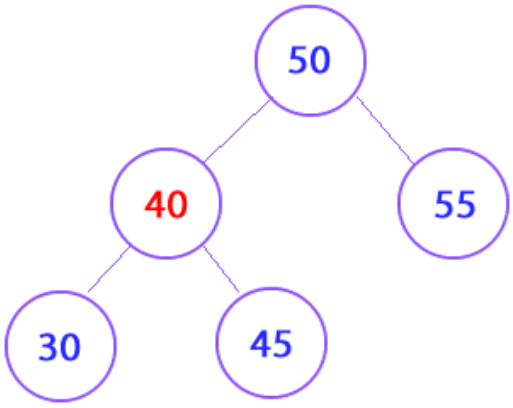
A splay tree is an implementation of a balanced tree that is designed to make common searches more efficient. It is typical for an item that has been recently searched to be searched again on a frequent basis. Studies have shown that in practice 90% of searches to the tree are to access 10% of the data items (This is called the **90-10 rule**) Because of this when a tree is searched and the item is located, the splay tree reorganizes so that the searched element is moved to the root of the tree. Generally speaking, if a small number of elements is being searched heavily they will tend to be found near the top of the tree making them quickly found. What is important about the splay tree is that it offers worst case time **O(M log n)** for a sequence of M operations.
Splay trees are also somewhat easier to implement because they do not require the maintenance of keeping track of the height or balance factor.

Moving a node to the root is called splaying which consists of single rotations in pairs in an order that depends on the link between the child, parent, and grandparent nodes. There are three kinds of rotations that are used to move an element to the root:

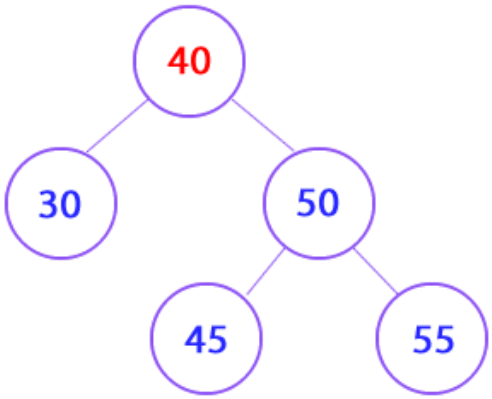
1. Simple Rotation (Zig and Zag)
2. Zig-Zag and Zag-Zag
3. Zig-Zag

Simple Rotation

The simple tree rotation uses the AVL tree rotations to move the searched element to the root of the tree.



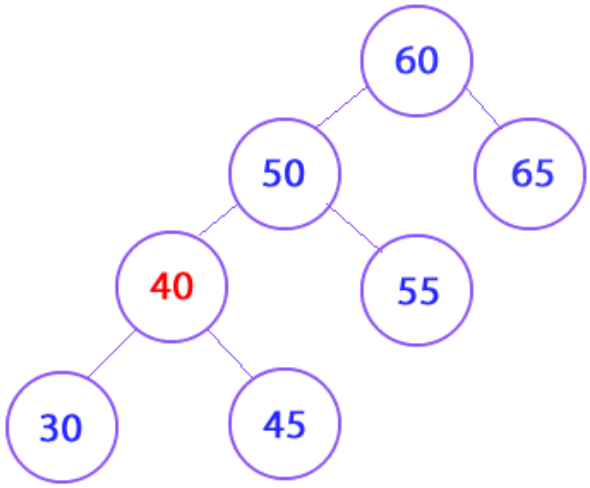
In this situation the node with 40 is the element being searched. A simple rotation will move the element to the root of the tree:



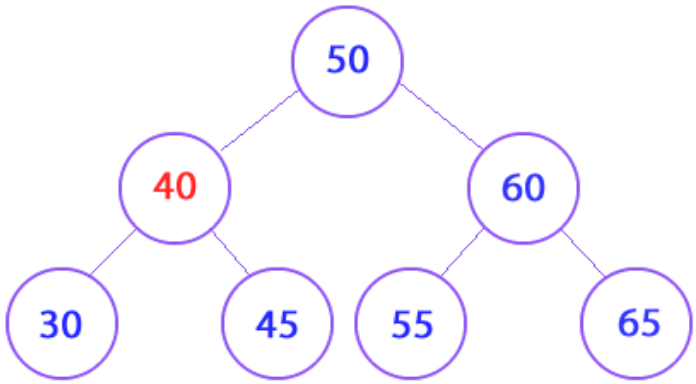
The node with 40 then is the node being splayed. This is called the zig case. It is when a left child is simply rotated to the root. The zag case then would be when the right child is simply rotated to the root.

Zig-Zag and Zag-Zag Rotation

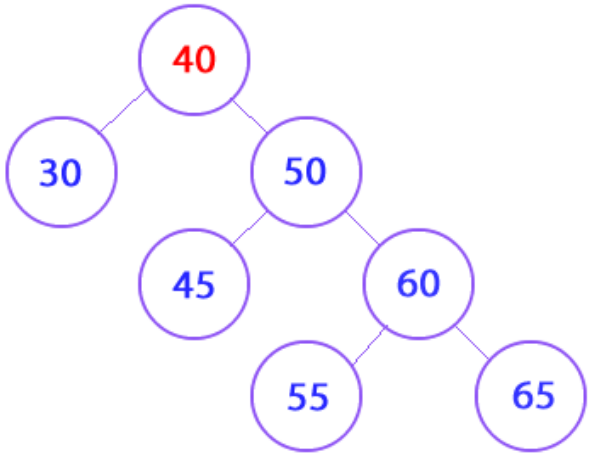
Nodes that are further down the tree are rotated in pairs so that the nodes on the path from the splayed node to the root move closer to the root on average. With the zig-zag case the splayed node is the left child of a left child:



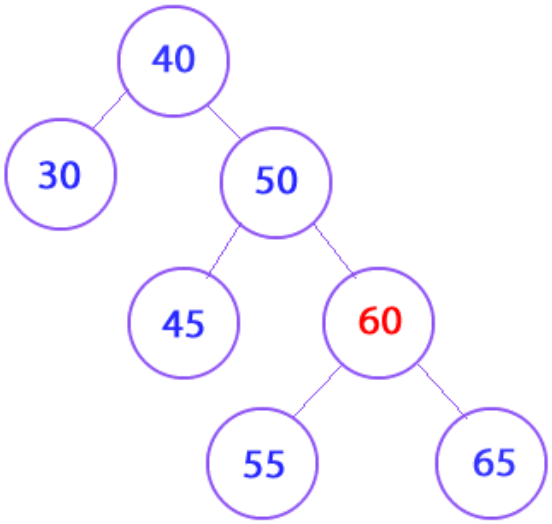
In this situation we rotate the node with 50 around the node with 60 to give us a semi-splay:



Finally, the node with 40 is rotated about the node with 50 to give us the full splay:



The zag-zag rotation is a mirror of the zig-zig. This is when the node is a right child of a right child:

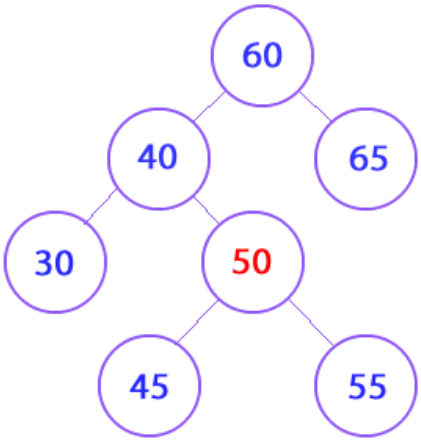


Zig-Zag Rotation

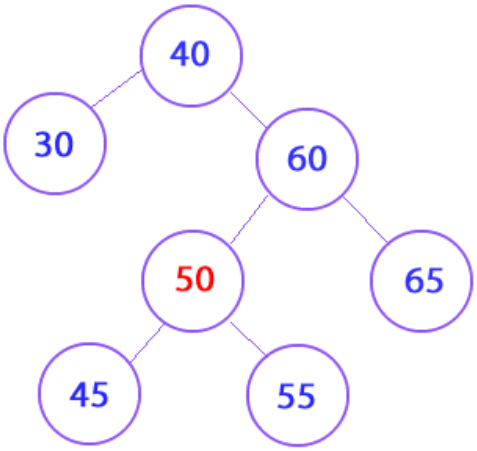
In the zig-zag case the node is the left child of a right child or a right child of a left child:

Zig-Zag Case

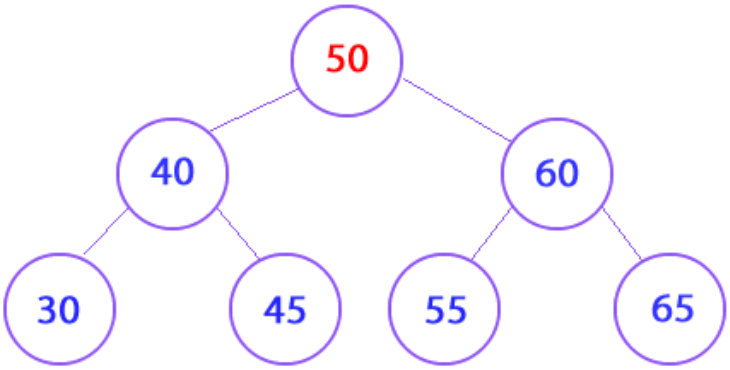
Before Rotation



After Rotation



This rotation produces a tree that has a height that is less than its original height which improves the overall balance of the tree:



The code to insert into a splay tree is as follows:

```
public void insert(int x)
{
    //Create a new node for insertion
    if(newNode == null)
        newNode = new Node();
    //set the data
    newNode.element = x;
    //If the root is null set the root and its
    //children
    if(root == nullNode)
    {
        newNode.left = newNode.right = nullNode;
        root = newNode;
    }
    else
    {
        //Move the last accessed node to the root
        //by performing a top-down splay
        root = splay(x, root);
        if(x < root.element)
        {
            newNode.left = root.left;
            newNode.right = root;
            root.left = nullNode;
            root = newNode;
        }
        else if(x > root.element)
        {
            newNode.right = root.right;
            newNode.left = root;
            root.right = nullNode;
            root = newNode;
        }
        else
            System.out.println("Duplicate Item Found In Tree");
    }
    newNode = null; //So next insert will call new
}
```

Splaying the Tree

To splay the tree a series of rotations in performed until the last searched item is at the root. Notice the use of the for ever loop to accomplish this:

```
private Node splay(int x, Node root)
{
    Node leftTreeMax, rightTreeMin;

    header.left = header.right = nullNode;
    leftTreeMax = rightTreeMin = header;

    nullNode.element = x; //Guarantee a match

    for( ; ; )
    {
        if(x < root.element)
        {
            if(x < root.left.element)
                root = rotateWithLeftChild(root);
            if(root.left == nullNode)
                break;
            //Link Right
            rightTreeMin.left = root;
            rightTreeMin = root;
            root = root.left;
        }
        else if(x > root.element)
        {
            if(x > root.right.element)
                root = rotateWithRightChild(root);
            if(root.right == nullNode)
                break;
            //Link Left
            leftTreeMax.right = root;
            leftTreeMax = root;
            root = root.right;
        }
        else
            break;
    }

    leftTreeMax.right = root.left;
    rightTreeMin.left = root.right;
    root.left = header.right;
    root.right = header.left;
    return root;
}
```

Code Listing

Below is the complete listing for the splay tree.

```
package splaytree;

/**
 *
```

```
 * @author gstev_000
 */
public class SplayTree
{
    private Node root;
    private Node nullNode;
    private static Node newNode = null;
    private static Node header = new Node( ); // For splay
    private static final int NOTFOUND  = -1;

    public SplayTree()
    {
       nullNode = new  Node();
       nullNode.left = nullNode.right = nullNode;
       this.root = nullNode;
    }
    public void insert(int x )
    {
        // Create a new node for insertion
        if( newNode == null )
            newNode = new Node( );
        // set the data
        newNode.element = x;
        // If the root is null set the root and it's
        // children
        if( root == nullNode )
        {
            newNode.left = newNode.right = nullNode;
            root = newNode;
        }
        else
        {
            // Move the last accessed node to the root
            // by performing a top-down splay
            root = splay( x, root );
            if( x < root.element )
            {
                newNode.left = root.left;
                newNode.right = root;
                root.left = nullNode;
                root = newNode;
            }
            else if( x > root.element)
            {
                newNode.right = root.right;
                newNode.left = root;
                root.right = nullNode;
                root = newNode;
            }
            else
                System.out.println("Duplicate Item Found In Tree");
        }
        newNode = null;   // So next insert will call new
    }

     private Node splay( int x, Node root )
     {
        Node leftTreeMax, rightTreeMin;

        header.left = header.right = nullNode;
        leftTreeMax = rightTreeMin = header;

        nullNode.element = x;   // Guarantee a match

        for( ; ; )
```

```
        {
            if( x <  root.element  )
            {
                if( x <  root.left.element )
                    root = rotateWithLeftChild( root );
                if( root.left == nullNode )
                    break;
                // Link Right
                rightTreeMin.left = root;
                rightTreeMin = root;
                root = root.left;
            }
            else if( x > root.element )
            {
                if( x > root.right.element )
                    root = rotateWithRightChild( root );
                if( root.right == nullNode )
                    break;
                // Link Left
                leftTreeMax.right = root;
                leftTreeMax = root;
                root = root.right;
            }
            else
                break;
        }

        leftTreeMax.right = root.left;
        rightTreeMin.left = root.right;
        root.left = header.right;
        root.right = header.left;
        return root;
    }
    public void remove( int x )
    {
        Node newTree;

            // If x is found, it will be at the root
        root = splay( x, root );
        if( root.element != x )
            System.out.println("Item Not In Tree");

        if( root.left == nullNode )
            newTree = root.right;
        else
        {
            // Find the maximum in the left subtree
            // Splay it to the root; and then attach right child
            newTree = root.left;
            newTree = splay( x, newTree );
            newTree.right = root.right;
        }
        root = newTree;
    }

    private Node rotateWithLeftChild( Node k2 )
    {
        Node k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        return k1;
    }

    private Node rotateWithRightChild( Node k1 )
    {
```

```
        Node k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        return k2;
    }
     public int find( int x )
    {
        root = splay( x, root );

        if( isEmpty( ) || root.element != x )
            return NOTFOUND;

        return root.element;
    }
     public boolean isEmpty( )
    {
        return root == nullNode;
    }
    private void printTree(Node root)
    {
        if( root != root.left )
       {
            this.printTree( root.left );
            System.out.println( root.element);
            this.printTree( root.right );
       }
     }
    public void printTree()
    {
        this.printTree(this.root);
    }
    public static void main(String[] args)
    {
        SplayTree st = new SplayTree();

        st.insert(60);
        st.insert(50);
        st.insert(55);
        st.insert(65);
        st.insert(40);
        st.insert(45);
        st.insert(30);

        st.printTree();

        int x = st.find(40);

        System.out.println(x);
        System.out.println("Tree Print");
        st.printTree();
    }
}
```

Select "Next" to continue working on this unit