Trees

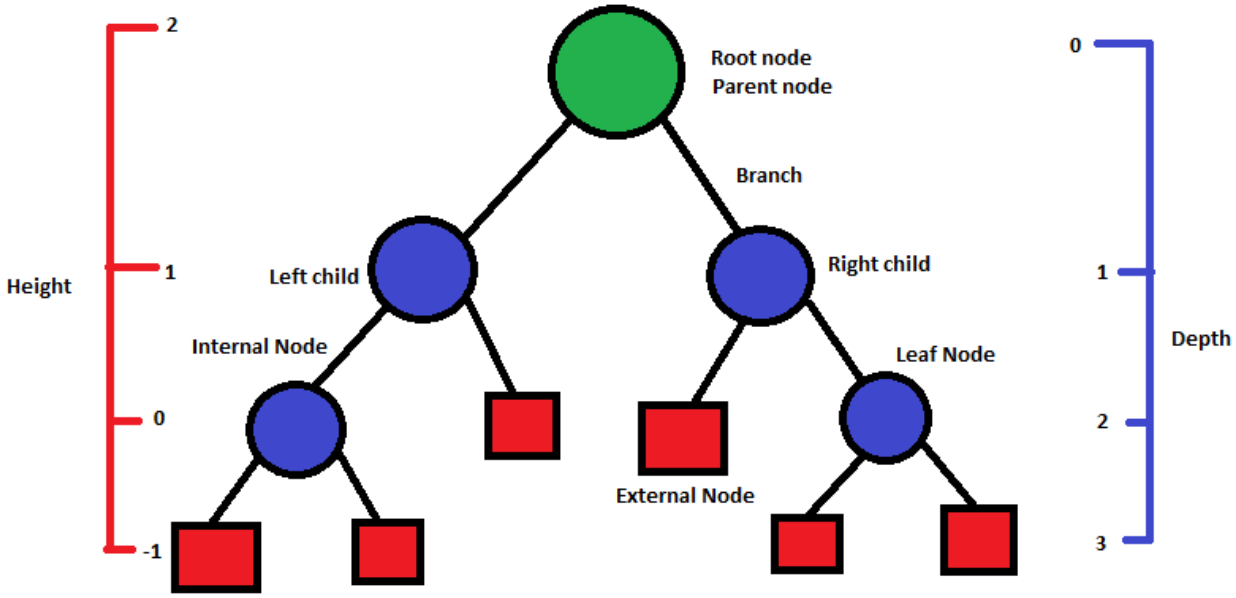
By: Anthony Nguyen

Table of Contents

1. Introduction

Linked lists are easy to implement but unfortunately, searching in its worst case takes O(n) time. A tree is basically a data structure with hierarchies that make it easy to traverse and search. A sorted tree can be searched in O(log n). One thing to consider, however, is that you can't have two of the same value in a tree.

A tree looks like this:



1.1 Terminology

Node: Something that contains a value and pointers for possible child nodes

Child node: Any node that has a node above it that links to it.

Internal node: Any node that contains a value

Parent node: Any node that contains child nodes

External node: A node with no value, used as a sentinel for internal nodes without child internal nodes

Leaf node: An internal node that has two external nodes as children

Depth: Distance from the root to the specific node

Height: Distance from the leaf node to the specific node

External path length: Sum of the depths of all of the external nodes

Internal path length: Sum of the depths of all the internal nodes

N-ary tree: A tree where the parent node can have **n** number of children. (i.e. **Bi**nary tree means that parent nodes can have a maximum of two children).

Root: The top node of the tree

The trees that we mostly deal with are **binary trees**. Binary trees have nodes with two child nodes: a left child node and a right child node.

1.2 Types of Tree Traversals (Binary Trees)

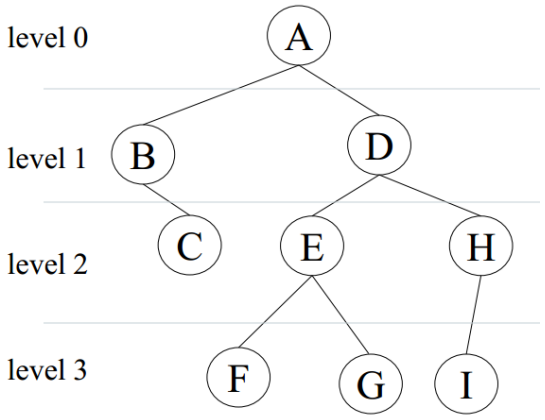
Traversal Steps: The code will check the node's value, left child, and right child, in the specified traversal order. When it checks the node's value, it looks for the value pointer within the node.

Pre order traversal: Traversal that involves checking the value of the node, then recursively going to the left child, and then recursively going to the right child. It checks the value of the node at the beginning hence the word **pre**.

Post order traversal: Traversal that involves recursively going to the left child, recursively going to the right child, and then checking the value of the node. It checks the value of the node at the end hence the word **post**.

In order traversal: Traversal that involves recursively going to the left child, checking the value of the node, and then recursively going to the right child. It checks the value of the node between the left child traversal and right child traversal hence the word **in**.

Breadth order traversal: This one is different. It traverses the nodes by descending height (levels) from left to right.



A,B,D,C,E,H,F,G,I

1.3 AVL Trees

Binary Search Trees

Before we talk about AVL trees, we must know what a **binary search tree** is. If you want your operation to be efficient, you can't just throw these nodes wherever you want. It has to be sorted so that a search algorithm can run properly. A binary search tree is a binary tree, so every parent node has two children. All values in the left subtree of the parent node are less than the value of the parent node, and all values in the right subtree of the parent node are greater than the value of the parent node.

What is an AVL Tree

An AVL tree is a **type of binary search tree** with one distinct quality: the heights of the two children subtrees of any parent node cannot differ by more than one. The difference in height of the two subtrees is known as the **balancing factor**. The balancing factor is taken by subtracting the height of the right subtree from the height of the left subtree. It is **not absolute value** because the negative decides which rotation we're going to use (described later).

1.4 The nodeType Data Structure

The data structure "nodeType" is defined inside either a header file or within the actual *.c file. The node file consists of:

nodeType *_leftChild: This is the node pointer to the left child node.

nodeType *_rightChild: This is the node pointer to the right child node

char *_val: This is the value stored inside the node. In the assignment the value being stored inside the structure is a character, so we use char to declare the value node. This node can be any variable type.

The contents of the node can be accessed like any structure, using **nodeName->_____**.

1.5 The treeType Data Structure

The data structure "treeType" is also defined inside the header file and is used to declare an entire tree. The only member of the treeType data structure is **nodeType *root**, which contains the root node of the tree.

2. Binary Search Tree Operations

2.1 Insertion

Basic insertion of a binary search tree involves recursive operations and comparing nodes to see whether or not the node being inserted is larger or smaller than the compared node. We start off our function taking in the value we want to insert and the entire tree as input variables. We use an integer function because we are using an integer to keep track of whether or not the operation was successful.

```
1.  int insert( char *val, treeType *tree )
2.  {
```

We then declare two nodes. The first node, **n**, is used as the node which will contain the value we want to insert. The second node, **p**, is used as the node which will be compared with the inserted node.

```
3.          nodeType *n=NULL, *p=NULL ;     /* some node pointers */
```

Two integers are declared to keep track of things. The first one, **iCmp**, is used to keep track of which node value is larger. The second one, **iResult**, is used to return a value of 1 or 0 based on success.

```
4.          int iCmp, iResult = 0 ;         /* some int variables */
```

Now we allocate memory for the **n** node that we want to insert, as well as the value of the n node. For the value of the n node, we allocate the memory size of a character multiplied by the length of the string inserted (number of letters, numbers, etc.) plus 1. **Strlen** is a built in function used to find the length of a string.

```
5.          n = malloc( sizeof( nodeType ) ) ; /* allocate a nodeType */
6.          n->_val                            /* allocate the node's _val field */
7.                  = malloc( sizeof(char) * (strlen( val ) + 1) ) ;
```

We then use the **strcpy** internal function to copy the value we want into the node's value field. Note that this step is **only for characters**. For an integer/float only tree, you would simply set the node value equal to the value you want.

```
8.          strcpy( n->_val, val ) ;/* copy val into the node's _val field */
```

We then make the left child and right child of the n node to point to null, because inserted nodes are always leaf nodes.

```
9.          n->_leftChild = NULL ;  /* initialize the children to NULL */
10.         n->_rightChild = NULL ;
```

Now we want to consider if there is an empty tree. If the tree is empty, we want to set the root of the tree to be equal to this node. We then assign a value of 1 to iResult showing successful insertion.

```
11.         if ( tree->_root == NULL )      /* tree is empty, so set this node */
12.         {                               /*    to be the root node       */
13.                 tree->_root = n ; //set the root of the tree to be the node
14.                 iResult = 1 ;           /* success */
15.         }
```

After make sure the tree is not empty, we can decide where to put out new node. We do this by using the find function to find the best existing node that we will attach our new node to.

```
16.         else    /* find correct position for val by searching tree */
17.         {
18.                 p = find( val, (nodeType *) tree->_root ) ;
```

After finding this node, we then use a built in function called **strcmp** to compare the value of the existing node to the new node. The built in function will then return value indicating which value is greater. The function takes in two values as inputs as well as their corresponding type.

```
19.                 iCmp = strcmp( (char *) val, (char *) p->_val ) ;
20.                         /* compare val with _val field of found p node */
```

If the built in function finds that the two values are the same, it will return 0. Since a tree can't have two nodes with the same value, the function will skip and return an iResult of 0. We do this by using a condition statement to make sure the value of **iCmp** calculated from the **strcmp** function is not zero.

```
21.                 if ( iCmp != 0 ) /* val is not currently in the tree */
22.                 {
```

If **iCmp** has a value of **less than zero**, then that means that the value of the new node is less than the value of the existing node. If this is the case, we make the **left child pointer** of the existing node point to the new node.

```
23.                         if ( iCmp < 0 )  /* val is less than p->_val */
24.                         {
25.                                 p->_leftChild = (nodeType *) n ;
26.                         }
```

If **iCmp** has a value of **greater than zero**, then that means that the value of the new node is greater than the value of the existing node. If this is the case, we make the **right child pointer** of the existing node point to the new node.

```
27.                         else            /* val is greater than p->_val */
28.                         {
29.                                 p->_rightChild = (nodeType *) n ;
30.                         }
```

We then give a value of 1 to iResult so that we know that the insertion operation was successful.

```
31.                         iResult = 1 ;   /* success */
32.                 }
33.         }
```

We then return iResult to the main program to show if the insertion was successful or not.

```
34.         return ( iResult ) ;
35. }
```

2.2 Finding the right existing node to attach the new node to

In an already sorted tree, finding the right existing node to attach your new node to is super easy. Basically, you start at the root node. You then see if the inserted value is greater or less than the existing node's value. After that, depending on the result, you would move onto either the right child node or the left child node and repeat. If either of the child nodes are null, then you found the right node to attach your existing node to!

The function type is nodeType because we are returning a node. The input variables of the find function are the inserted character's value, and the node which we are comparing. We always start by inputting the root of the tree.

```
1.  nodeType *find( char *val, nodeType *n )
2.  {
```

We then declare an iCmp variable for the comparison result just like in the previous insert function.

```
3.          int iCmp ;      /*      an int variable */
```

A p node is declared and set equal to the n node being passed for comparison.

```
4.          nodeType *p = (nodeType *) n ;  /* a nodeType pointer */
```

We then check to see if the p value is not empty, to make sure that the tree isn't empty itself.

```
5.          if ( p != NULL )        /* the tree is non empty  */
6.          {
```

Once we've made sure the tree isn't empty, we can compare the existing **p** node to the value being inserted using the built in function **strcmp.** Again strcmp will return a value.

```
7.                  iCmp = strcmp( val, p->_val ) ; /* compare strings */
```

Now the find function will do its analysis. If the value of iCmp is less than zero, then that means the inserted value is less than the value of the existing node. This means that the inserted value will be inserted somewhere to the left of the current node. The condition statement below also checks to see if the left child node is NULL. If it is, then we have our ideal existing node and the value of p gets returned.

```
8.                  if ( ( iCmp < 0 ) && ( p->_leftChild != NULL ) )
9.                  {
```

If the condition statement is passed, then it will recursively run the find function again on the left child of the existing node to search the left branch for the ideal existing node for insertion.

```
10.                         p = find( val, p->_leftChild ) ;
11.                 }
```

If the value of iCmp is greater than zero, the inserted value is greater than the value of the existing node, and the program will then do the same stuff as the previous step for the right child.

```
12.                 else if ( ( iCmp > 0 ) && ( p->_rightChild != NULL ) )
13.                 {
14.                         p = find( val, p->_rightChild ) ;
15.                                         /* search right branch */
16.                 }
17.         }
```

Finally we return our ideal existing node, p, to the insert function.

```
18.         return p ;
19. }
```

2.3 Calculating the Number of Nodes (calcNNodes)

The purpose of the actual function calcNNodes is to run the helper function that will count the nodes **only if the tree is not empty**. That is evident in the condition "if (tree!=NULL)".

```
1.  int calcNNodes( treeType *tree )
2.  {
3.      if ( tree!=NULL) //check to see if the tree is empty or not
4.      {
5.          return( calcNNodesSubtree( tree->_root ) ) ; //runs the helper function to count nodes
6.      }
7.      else return( 0 ) ;
8.  }
```

calcNNodes' helper function: calcNNodesSubtree

The calcNNodes function calls another function, the **helper function**, called **calcNNodesSubtree** to count the number of nodes. The calcNNodes function passes along the **root** of the tree to the helper function because we want to start the traversal there. The helper function accepts a node as input and calls it **root**. The helper function will return the sum of the node count at the very end.

First we want to declare our helper function. This function is declared **outside of the calcNNodes function.**

```
1.  int calcNNodesSubtree( nodeType *root ) //takes a node as input and calls it root
2.  {
```

We then want to declare a variable to use as a counter. We start it at 0.

```
3.      int N = 0 ; //declares a counter variable
```

Because this function will be running recursively, we need a condition to make the function stop running itself. We want the traversal to go backwards (undo recursion) once we get to an **external node** (i.e. the node is NULL). Within the condition statement, to count the external node we assign **N** a value of 1.

```
4.      if ( root == NULL ) N = 1 ; //if the node is external then N has a value of 1
```

Now that we have the terminating statement, we can decide what happens when there is an **internal node** in our **else** statement. The assignment uses **post order traversal** so we will do the same. **First, we want to find the N counter variable for the left side of that internal node.** We do this by running the helper function again and passing the **left child**.

```
5.      else
6.      {
7.          N = calcNNodesSubtree( root->_leftChild ) ;
```

Once we have counted the subtree to the left of the function, we want to **add to the counter variable the number of nodes on the right**, so we run the function again passing off the **right child**.

```
8.          N += calcNNodesSubtree( root->_rightChild ) ;
```

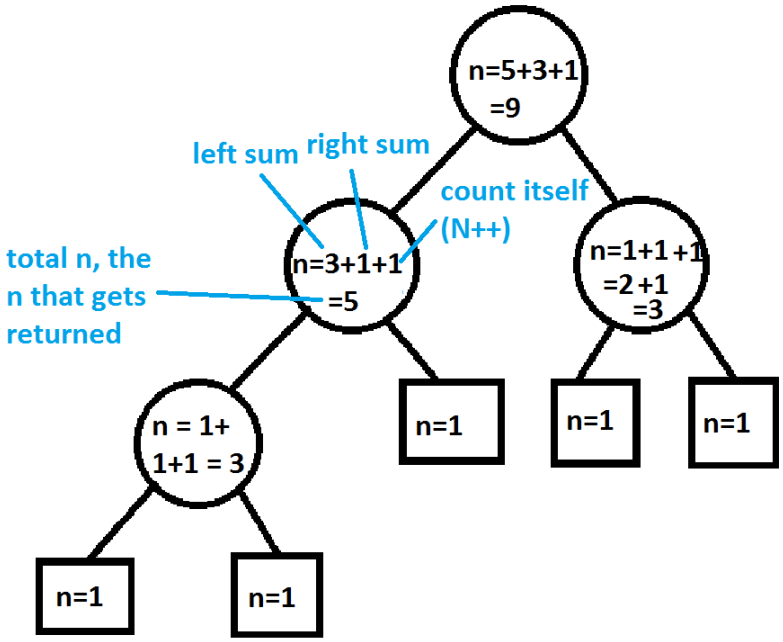
Once it has the sums for both children, it will then count itself by adding 1 to the counter variable.

```
9.          N++ ;
10.     }
```

Finally, the counter variable will be returned to wherever it was called, whether it's a higher level calcNNodesSubtree function to get the sum of the subtree or the calcNNodes function to get the sum of the entire tree.

```
11.     return( N ) ;
12. }
```

It's best to look at a YouTube video to see how this works step by step, but I've provided a diagram that might help show how the number of nodes is calculated. It isn't the easiest to follow but might help.



Inside each node is the **value of n** at **each level of recursion in the function**. It often helps to go line by line in the code and go through the tree.

2.4 Calculating the Number of External Nodes (calcExternalNodes)

The function that calculates the number of external nodes is **very similar** to the calculating the number of nodes function. All you have to do is remove this line from the helper function:

```
9.          N++ ;
```

This line is the line that counts the internal node itself, so you take it out, and there ya go you've got the function for calculating the number of external nodes.

2.5 Calculating the Number of Internal Nodes (calcInternalNodes)

The function that calculates the number of internal nodes is **very similar** to the calculating the number of nodes function. The only change that is made is to the line:

```
4.      if ( root == NULL ) N = 1 ; //if the node is external then N has a value of 1
```

Where you would change N to be equal to 0, because you don't want to count the external nodes. Your new line would look like this:

```
4.      if ( root == NULL ) N = 0 ; //if the node is external then N has a value of 1
```

2.6 Calculating the External Path Length (calcExternalPathLength)

To calculate the external path length, we first write our main function (**calcExternalPathLength)** to call the helper function that will calculate the external path length. The main function takes the tree as an input just like the node counting function.

```
1.  int calcExternalPathLength( treeType *tree )
2.  {
```

Afterwards, it declares a variable for the result and for the initial depth of the root. Both will be set equal to zero.

```
3.      int result = 0, depth = 0 ;
```

Then, just like in the other function, it checks to see if the tree is null. If it isn't, it will call the helper function to find the result, passing off the root node of the tree, and the initial depth.

```
4.      if ( tree != NULL )
5.      {
6.          result = calcEPLSubtree( tree->_root, depth ) ;
7.      }
```

Finally, it will return the resulting length of the tree.

```
8.      return( result ) ;
9.  }
```

calcExternalPathLength's helper function: calcEPLSubtree

The calcEPLSubtree helper function is the function that is used to calculate the external path length. It takes in a node and the node's depth as inputs. It also returns an integer so you declare the function like this:

```
1.  int calcEPLSubtree( nodeType *root, int depth )
```

Once again, the function is declared **outside of the main function**. After declaring the helper function, we declare an integer **e** used to keep track of the total depth (within the level of recursion).

```
2.  {
3.      int e=0 ;
```

Now we want to check if the node sent is external or internal. We only want to return the depths of the external nodes. External nodes will be equal to NULL so we use this as our condition statement.

```
4.      if ( root == NULL ) /* checks if the node is an external node */
```

```
5.      {
```

If the node sent is in fact an external node, then we want the depth of the node to be returned in **e**.

```
6.          e = depth ;
7.      }
```

If the node isn't external, then in the **else** condition it will run the helper function again on the left child. This time we pass off the left child, and a depth of **one greater than the current node** to account for the going down in depth. We set all of this equal to **e** which keeps track of the sum of depths.

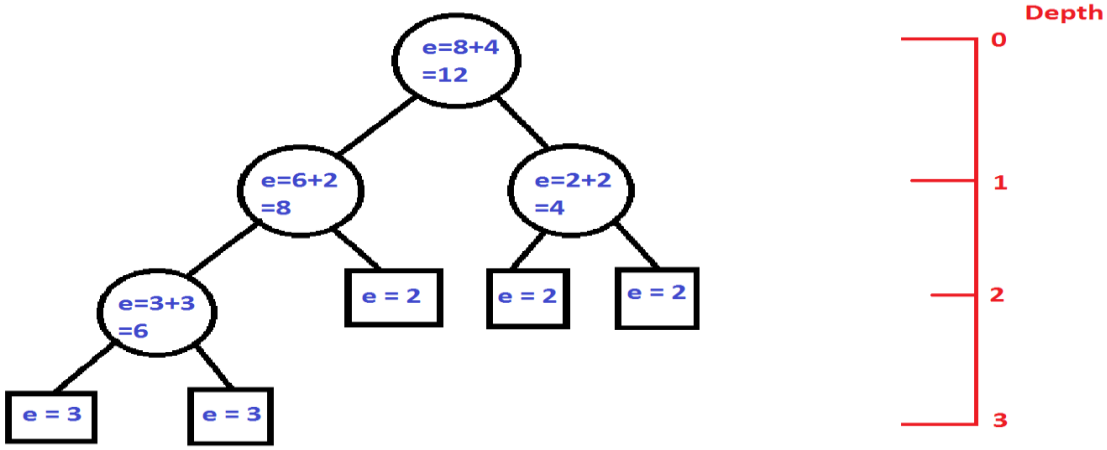
```
8.      else            /* not an external node */
9.      {
10.         e = calcEPLSubtree( root->_leftChild, depth+1 ) ;
```

After we have the sum of depths on the left, we want to find the sum of depths on the right. We do this by calling the helper function again to find the sum of depths on the right child, passing off the right child and again depth+1. We want to add the sum of depths of the right to the already existing sum of depths (**e**).

```
11.         e += calcEPLSubtree( root->_rightChild, depth+1 ) ;
12.     }
```

Finally, we return our sum of depths at that level to either the previous level of the helper function if called recursively, or will return the entire sum of external depths (external path length) to the main function.

```
13.     return( e ) ;
14. }
```



Fun fact: Because we're checking the node first, left child second, and right child third, this method is a **pre order traversal**.

2.7 Calculating the Internal Path Length (calcInternalPathLength)

Calculating the internal path length is very similar to calculating the external path length. You start off with a main function (**calcInternalPathLength**) that calls the helper function (**calcIPLSubtree)** if the tree is not empty. The helper function takes the same inputs as well.

```
1.  int calcInternalPathLength( treeType *tree )
2.  {
3.      int result = 0, depth = 0 ;
4.      if ( tree != NULL )
5.      {
6.          result = calcIPLSubtree( tree->_root, depth ) ;
7.      }
8.      return( result ) ;
9.  }
```

calcInternalPathLength's helper function: calcIPLSubtree

The helper function for calculating the internal path length is very similar to the helper function for calculating the external path length. The initialization and initial declarations are the same (but with the different function name and different sum of depths tracking variable name).

```
1.  int calcIPLSubtree( nodeType *root, int depth )
2.  {
3.      int i=0 ; //variable to keep track of sum of depths within the call
```

It then checks to see if the node is null to make sure it is not an external node. If it is an external node, the root will be null, and the function will return a value of zero.

```
4.      if ( root != NULL ) /* not an external node */
5.      {
```

Once we pass our condition statement, we know that the node is an internal node. Now we do essentially what is a **pre order traversal**. First, we add the depth of the current node to **i**.

```
6.          i = depth ;
```

Next, we **add the sum of depths of the left child** to i. We do this by running the function again and passing off the left child node and depth+1.

```
7.          i += calcIPLSubtree( root->_leftChild, depth+1 ) ;
```

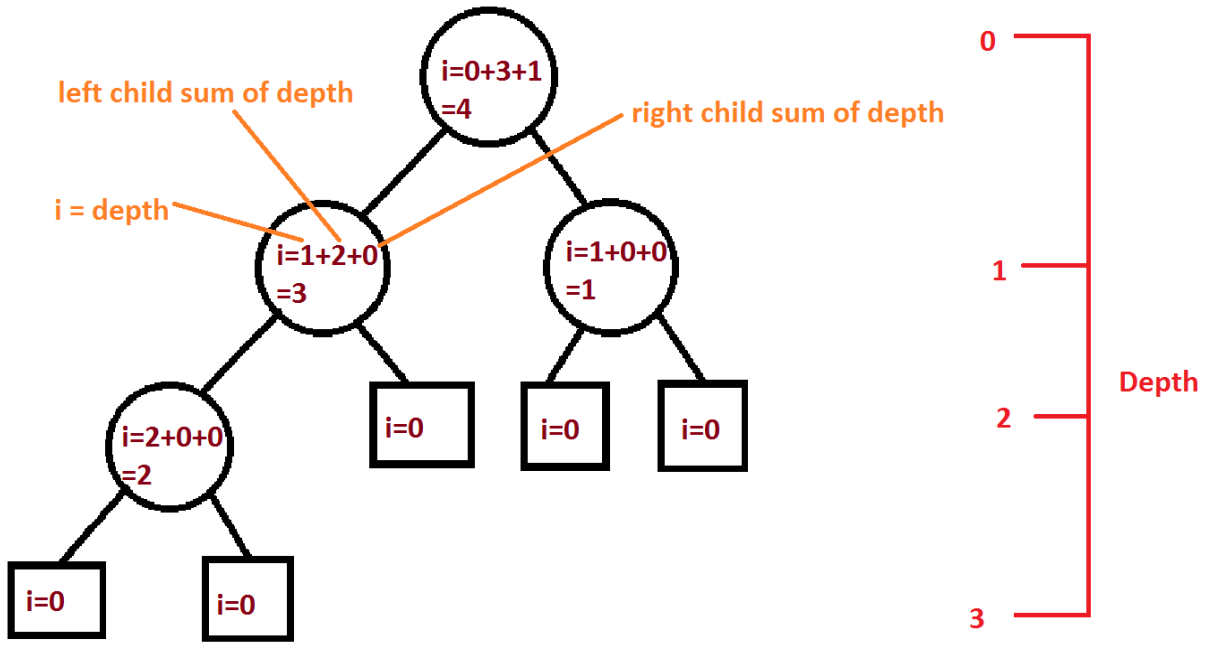
Finally, we **add the sum of depths of the right child** to i. We do this by running the function again and passing off the right child node and depth+1.

```
8.          i += calcIPLSubtree( root->_rightChild, depth+1 ) ;
9.      }
```

We finish off the helper function by returning the value of i to the previously called recursive function or back to the main function.

```
10.     return( i ) ;
11. }
```

Diagram:



2.8 Freeing the Tree

To free the tree, memory needs to be de-allocated from each node before free the actual tree itself. That is why another helper function needs to be called to do this before we can go ahead and do the function free(tree). First, you declare your function, taking in the tree as your input variable, and using void because nothing is returned.

```
1.  void freeTree( treeType *tree )
2.  {
```

Then, like in previous methods, it checks to see if the tree is empty first. If it isn't, it goes on.

```
3.      if ( tree != NULL )
4.      {
```

We then call the helper function to free the nodes of the tree, passing along the root node of the tree first.

```
5.          freeSubtree( tree->_root ) ;
```

Once we free all of the nodes, we can then free the tree using the **free(____)** function.

```
6.          free( tree ) ;
7.      }
8.  }
```

freeTree's helper function freeSubtree

To free all the nodes from a tree, we need to use **post order traversal**. The actual function takes in a node as its input (calls it root). Because it does not return anything, it is a **void** function.

```
1.  void freeSubtree( nodeType *root )
2.  {
```

Because external nodes are NULL already, there is no point in freeing the space for an external node. We check to see if the node is an external node or not by checking if it is equal to NULL. If it isn't, the condition is satisfied and we move on.

```
3.      if ( root != NULL )
4.      {
```

We are using post order traversal, so before the actual node can be freed, the left and right branches need to be freed first. Here we run the helper function again recursively to free the left children of the node.

```
5.          freeSubtree( root->_leftChild ) ;
```

We then run the function again to free all the right children of the node.

```
6.          freeSubtree( root->_rightChild ) ;
```

Finally, we free the node itself.

```
7.          free( root ) ;
8.      }
9.  }
```

2.9 Finding the Height

We need to calculate the height to find stuff like the balance factor. What the program does to calculate height is measure the height of all of the paths, and then it takes the longest path and uses it as the height.

Height returns an integer value so we use the integer function. We also take in a node to measure the height of.

```
1. int calcHeight( nodeType *root )
2. {
```

First we need to make sure that the node (called 'root') isn't empty. If it is, then it must have a height of 0, and a value of 0 will be returned.

```
3.         if(root==NULL)
4.         return(0);
```

If the node is not empty, then it will return a macro called **max** which is defined at the top. The definition looks like this:

```
1. #ifndef max
2.         #define max( a, b ) ( ((a) > (b)) ? (a) : (b) )
3. #endif
```

What is does is it compares the value inserted to the left with the value inserted to the right. It will then keep whichever value is higher. As inputs, the max macro takes in the recursively run **calcHeight** function used on the left child, and it takes in the recursively run **calcHeight** function used on the right child plus 1. What's returned is essentially the longest height.

```
5.         else
6.         return(max(calcHeight(root->_leftChild),calcHeight(root->_rightChild) + 1));
7. }
```

3. AVL Tree Balancing

3.1 Finding the Balance Factor

Before we start balancing, we need to find the balance factor at the root node of the tree or subtree to decide which type of balancing we are going to do. There are four types of balancing: LL, LR, RL, and RR. The type of balancing done is decided by the balance factor.

The balance factor function finds the height of the left child and the height of the right child and subtracts them.

The balance factor is always an integer so we use an integer function. It also takes in the root node of the tree or subtree as input.

```
1. int calcBalanceFactor(nodeType *root )
2. {
```

It then sees if the root node is NULL just in case an empty node was passed. If this is the case, it will return 0.

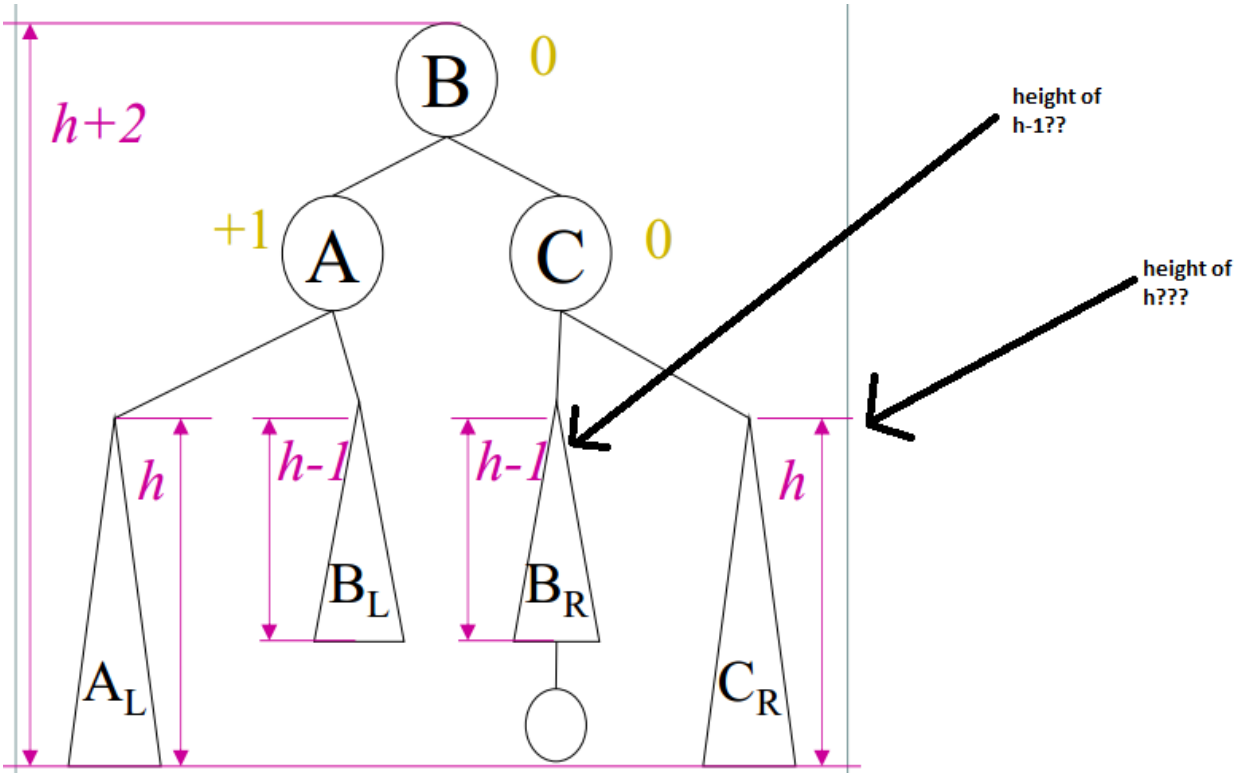
```
3. if (root==NULL) return(0);
```

If the root node isn't null, then it will do the balance factor operation. To make this simple (or more complicated depending on how you like things), they put the entire operation inside the return function. We return the result of the left subtree height minus the right subtree height.

```
4. else
5. return(calcHeight(root->_leftChild) - calcHeight(root->_rightChild));
6. }
```

Once we have our balance factors, we can start to do our balancing.

Side Note: WHAT IS THIS NOTATION???



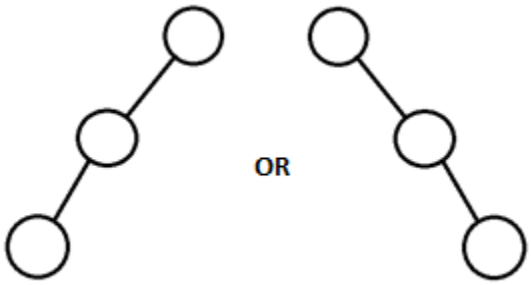
Well it turns out that those triangle things are used to represent an entire subtree. Subtrees have varying heights which is why these things vary in size, and some can have height **h** and some can have a height of **h-1**.

3.2 Balancing

When a new node is inserted, it may change the balance factor to something greater than 1 or something less than -1. If this is the case, we need to balance our tree. Now, since our nodes are inserted one at a time, the tree balancing algorithm will make sure that the balance factor is never more than 2 or less than -2 before balancing. Otherwise, you messed up.

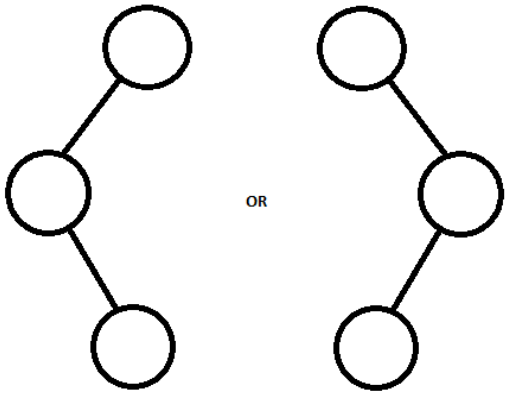
There are four types of balancing, but this can be further simplified to two types of balancing with different directions.

Simple balancing rotations are the left-left and right-right rotations. Only one round of balancing needs to happen when you do these types of rotations. You use this type of balancing when the path of the three nodes in the subtree looks something like this:



In CS, they call this type of path a **zig-zig pattern**, as in the path goes the same way both times.

Double balancing rotations are the left-right and right-left rotations. These are done when the path isn't straight and one rotation is not enough to balance the tree. You use double balancing when your three node path looks something like this:

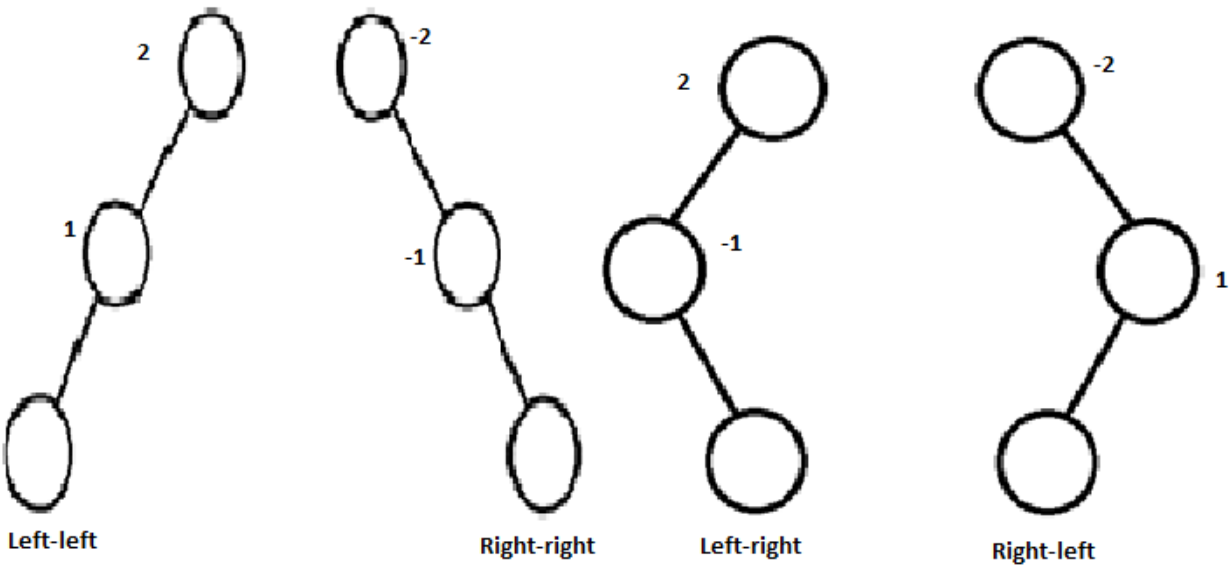


In CS, they call this type of path a **zig-zag pattern**, as the path goes one way and then the other.

In the code, what happens is the function will find the balance factor at the top node. If the balance factor is >1, then it knows the imbalance is in the left subtree, and it will again find the balance factor of the left child of that top node. If the balance factor at that node is 1, then it will know to do a left-left rotation, because there is a node to the left. If the balance factor at that node is -1, then it will know to

do a left-right rotation, as the balance factor indicates that there is a node at the right but not at the left.

If the balance factor at the root node is <-1, then it knows the imbalance is in the right subtree, and it will again find the balance factor of the right child node of the root node. If the balance factor of the right child node is -1, then it will know to do a right-right rotation because the imbalance is at the right path both times. If the balance factor at that node is 1, then it will to a right-left rotation.



If for whatever reason the balance factor at one of the subtrees is >2 or <-2, it will recursively run itself on either the left or right child depending on the balance factor of the node, so it balances the subtree.

We'll go more in depth with balancing when we get to our balancing functions.

The Balancing Function

Because the balancing function returns the new root of the function, we want to make this function a **nodeType** function. We pass the original root of the tree or subtree to this function.

```
1.  nodeType *AVLBalance(nodeType *root)
2.  {
```

Next, we create a node called **temp** to hold the root node.

```
3.      nodeType * temp = root;
```

Now we calculate our balance factor using the previous **calcBalanceFactor** function. To save lines, we declare an integer variable called BF and set it equal to this function on the same line. The function takes in the temp node as input.

```
4.      int BF = calcBalanceFactor(temp);
```

It then does the root!=null check. I'm not going to explain this one again.

```
5.      if (root != NULL)
6.      {
```

Now the function will do its analysis in the condition statements. If the balance factor is greater than one, it will go ahead and calculate the balance factor in the left child node, because it knows the imbalance is in the left.

```
7.          if (BF>1)
8.          {
9.              int LBF = calcBalanceFactor(temp->_leftChild);
10.             printf("Balance Factor at Left Child : %d \n",LBF);
```

Now it does the analysis on the left balance factor. If it is between 0 and 2 (leaving only 1), then it knows that a left-left rotation needs to be done, so it will call the left-left rotation function. **No it is NOT a built in function, we write it ourselves later on.** The left-left rotation will take the root node as input.

```
11.             if ((LBF>0) && (LBF<2))
12.             {
13.                 temp = LLRotation(temp);
14.             }
```

If the left balance factor is between -2 and 0, then it will knows we need to do a left-right rotation, so it will go ahead and call the left-right rotation function.

```
15.             else if ((LBF>-2) && (LBF <0))
16.             {
17.                 temp= LRRotation(temp);
18.             }
```

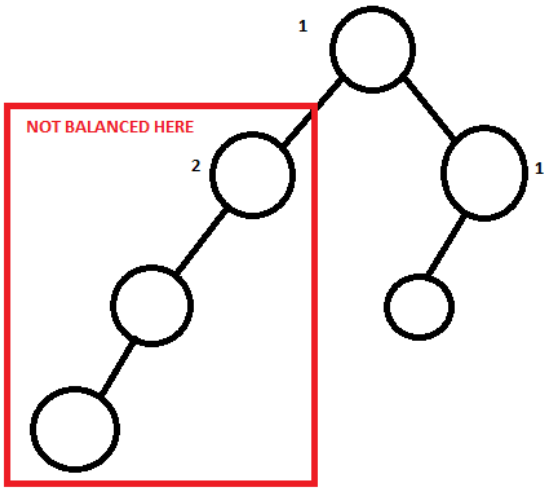
If the left balance factor is not zero and it is not between the previous 2, it means the imbalance at the subtree is greater than 2. If this is the case, it will run the AVL balance again recursively on the left child.

```
19.             else if (LBF != 0)
20.                 temp->_leftChild = AVLBalance(temp->_leftChild);
21.         }
```

Now we get back to the scenario where the balance factor for the root node is less than -1. It will basically do the same thing except as the left, except with new variable names (RBF instead of LBF), and the other types of balancing. Follow the code and you'll see this.

```
22.         else if (BF<-1)
23.         {
24.             int RBF = calcBalanceFactor(temp->_rightChild);
25.             printf("Balance Factor at Right Child : %d \n",RBF);
26.             if (RBF<0 && RBF >-2)
27.             {
28.                 temp = RRRotation(temp);
29.             }
30.             else if (RBF<2 && RBF >0)
31.             {
32.                 temp= RLRotation(temp);
33.             }
34.             else if (RBF != 0)
35.                 temp->_rightChild= AVLBalance(temp->_rightChild);
36.         }
```

If the balance factor is balanced at the root, then the balance factor will be between -2 and 2. If this is the case, **we still need to balance the subtrees.** Why? The reason is that just because the root is balanced doesn't mean the entire tree is balanced.



Didn't see that one coming did ya.

We do this by writing an **else condition** which will recursively run the AVLBalance program on the left child and the right child of the node.

```
37.         else
38.         {
39.             temp->_leftChild = AVLBalance(temp->_leftChild);
40.             temp->_rightChild= AVLBalance(temp->_rightChild);
41.         }
42.     }
```

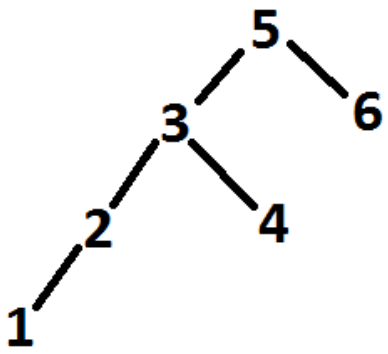
```
43. return (temp);
44. }
```

Now that we have our function that will call the balancing's, we can do our rotations.

3.3 Left-left rotation

The left-left rotation is done when there is an imbalance in the left child and in the left subtree of the left child. The balance factor would be 2 at the root and 1 at the left subtree.

Consider this tree here:



The balance factor of the root is 2, and the balance factor of the left child is 1, so we do a left-left rotation.

The function itself is a **nodeType function**, because we want it to return the new root of the tree. It takes in the old root as the input variable.

```
1.  nodeType * LLRotation(nodeType *oldroot)
2.  {
```

We then do our check to make sure the old root isn't null.

```
3.   if ( oldroot!=NULL)
4.  {
```

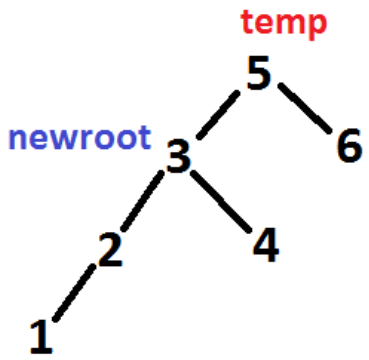
Now that we know the old root isn't null, we can start the balancing steps.

First we are going to give names to our key nodes that we are going to use for the rotation. Let's call the original root **temp**, the new root **newroot**. We set the old root equal to the temp variable.

```
    5.   nodeType *temp = oldroot;
6.   nodeType *newroot;
```

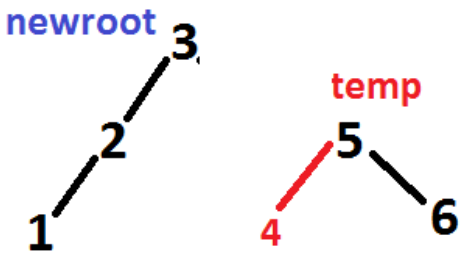
Now that we have our names for the two key nodes, we start messing around with the tree pointers.

1. We first want to make the **new root the left child of the old root**.



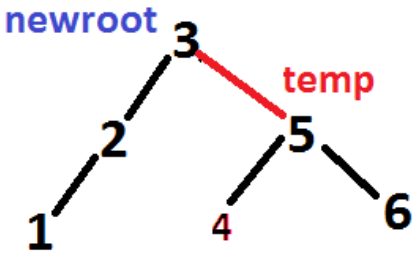
7. `newroot = temp->_leftChild;`

2. We then want to make the **left child pointer of temp point to the right child of the new root.**



8. `temp->_leftChild = newroot->_rightChild;`

3. Finally, we want the **right child pointer of the new root to point to the temp.**



9. `newroot->_rightChild = temp;`

And there it is, your tree is balanced. Finally, we return the new root. If the operation fails we return null.

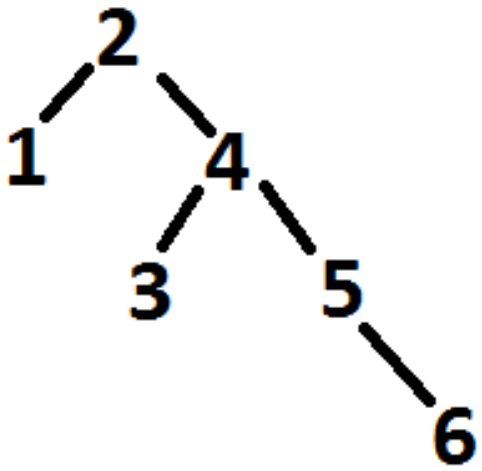
10. `return(newroot) ;`

```
11. }
12. else return(NULL) ;
13. }
```

3.4 Right-Right Rotation

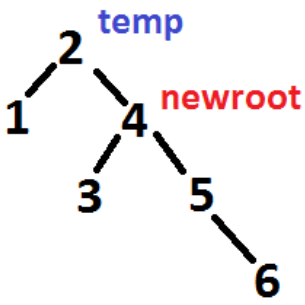
RR rotation is very similar to LL rotation. It is basically a mirror of the LL rotation. The right-right rotation is done when there is an imbalance in the right child and in the right subtree of the right child. The balance factor would be -2 at the root and -1 at the right subtree.

We'll be using this tree here as a guide:



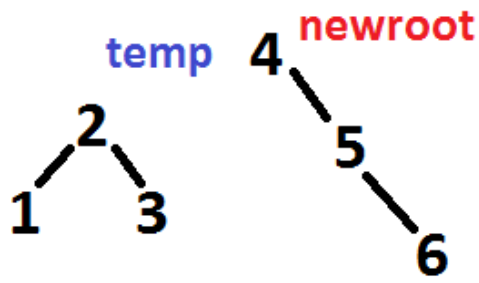
The top bit (lines 1-6) of the function is pretty much the same as LL except for the function name so we'll skip that.

1. We first want to make the **new root the left child of the old root**.



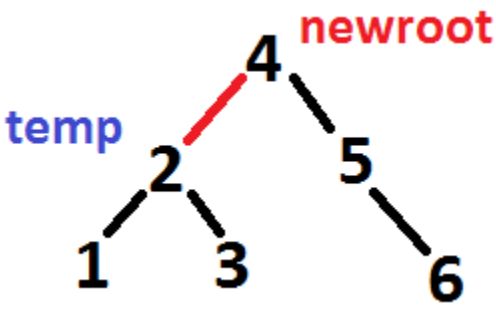

```
7.   newroot = temp->_rightChild;
```

2. We then want to make the **right child pointer of temp point to the left child of the new root.**




```
8.   temp->_rightChild = newroot->_leftChild;
```

3. Finally, we want the **right child pointer of the new root to point to the temp.**




```
9.   newroot->_leftChild = temp;
```

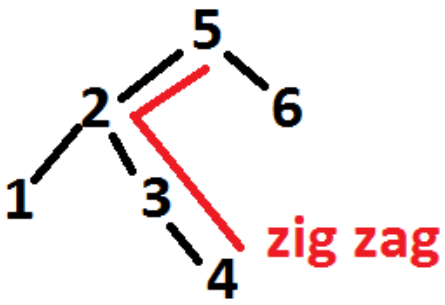
Lines 10-13 are exactly the same as LL.

3.5 LR Rotation

The left-right rotation is done when there is an imbalance in the left child and in the right subtree of the left child. The balance factor would be 2 at the root and -1 at the left subtree.

This is where the naming comes from. Left right means that there is an imbalance on the left of the root and on the right of the left subtree. It is NOT the order of calling of the functions.

We will use this tree here as an example.



Notice how the pattern is zig zag.

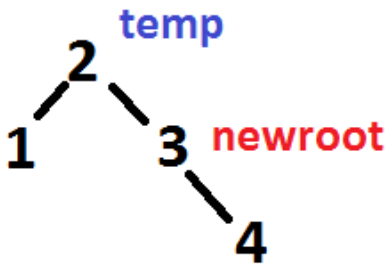
We call our function the same way as before and we set the temp variable to be equal to the old root just like before.

```
1. nodeType *LRRotation(nodeType *oldroot)
2. {
3. nodeType *temp = oldroot;
```

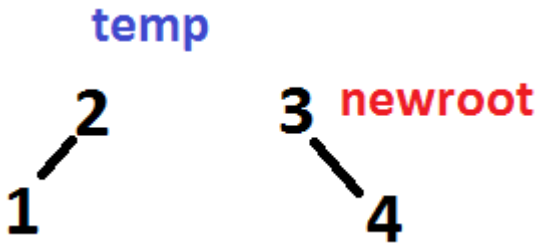
Next, we run the **right rotation** on the left subtree of the root. We return the new root for the temp variable.

```
4. temp = RRRotation(oldroot->_leftChild);
```

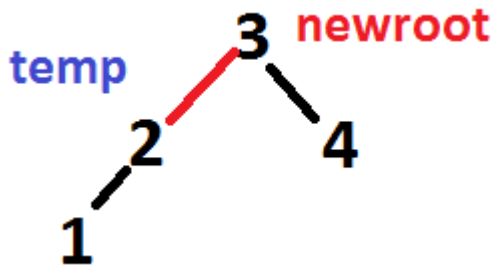
Within the RR rotation function:



Step 1:

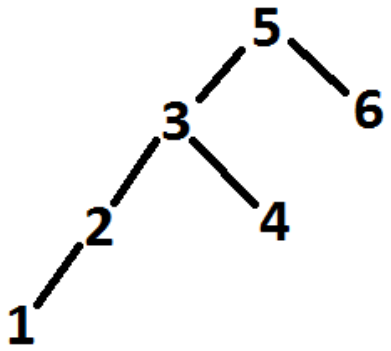


Step 2:



Step 3:

REMEMBER THAT TEMP HERE IS NOT THE TEMP IN THE RL FUNCTION!!! IT IS THE ONE INSIDE THE RR!!

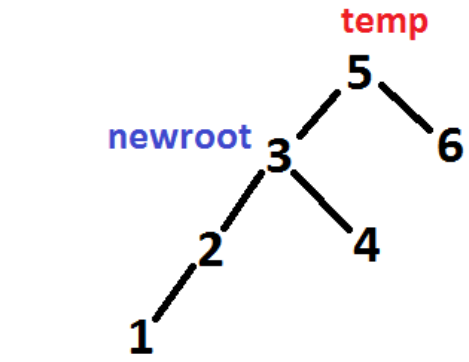


We end up with this:

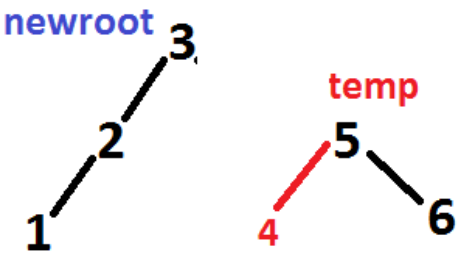
We then do a LL rotation at the root level. We send the old root as the input variable.

```
5.  temp = LLRotation(oldroot);
```

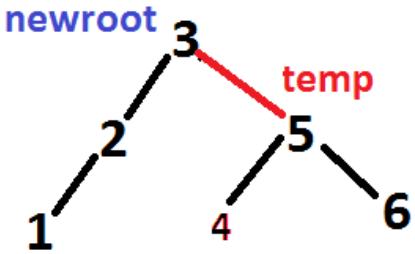
Within the function:



Step 1:



Step 2:



Step 3:

We then return our temp as the new root.

```
6.  return(temp);
7.  }
```

3.6 RL Rotation

RL pretty much a bass akwards version of LR. You run recursively the LL rotation on the right subtree, and then you run the RR rotation at the root level. Code is almost exactly the same except for those changes.