Documentation

1. Insert Function:

   Process of insertion is as described. First we check if the root is NULL, if it is NULL then create a root and assign balance factor(bf)=0. Else initialise 3 pointers t,s,p. ‘t’ points to the header node which is an arbitrary node. The RChild of the header is the root node. ‘s’ & ‘p’ points to this root node.

   The entire insertion is a 3 step process:
   - Searching position to insert.
   - Inserting the node.
   - Changing bf of each affected node & Rotation if necessary.

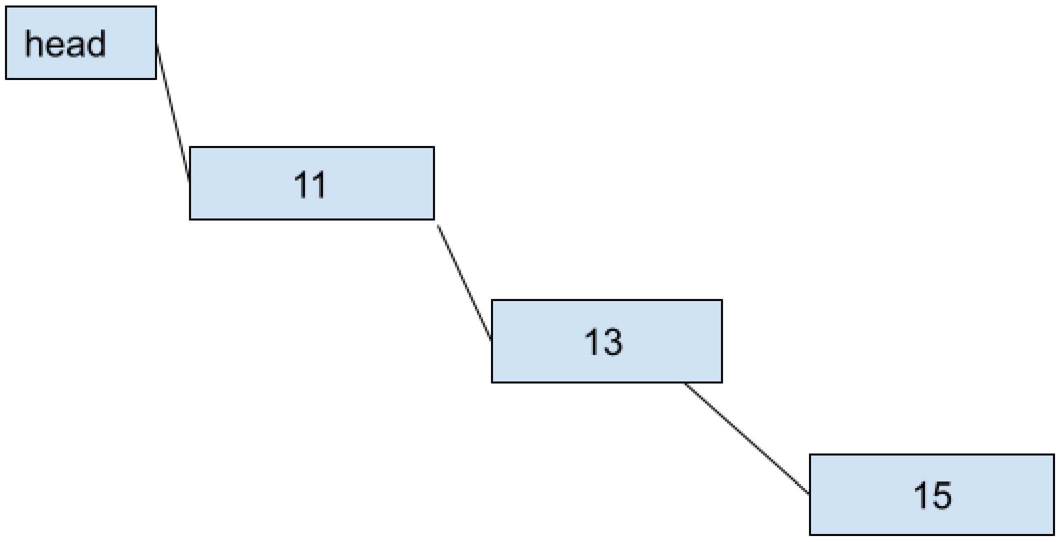
   Searching for a position is easiest. We just traverse through the tree using the ‘p’ pointer. If node is duplicate just return. Else Keep on searching for a suitable position. If found, Give it the new name ‘q’.

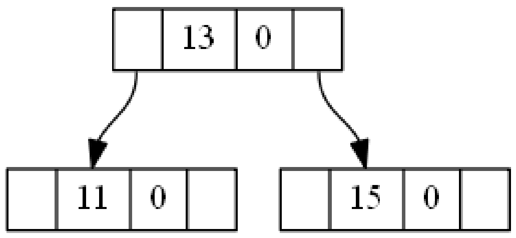
   Assign suitable values to the newly inserted node.

Create a helper variable ‘a’ which would help in finding rotation and balancing the tree. Update ‘p’,’q’,’s’,’t ‘ accordingly. ‘s’ will point to the *point of imbalance.* ‘p’ will update balance factors. Variable ‘rotcount’ will specify a single or double rotation. If required, Rotate it and Finally rebalance after rotation.

Test case 1: Insert 11, 13, 15 in the tree.

- When 11 inserted, bf[11]=0. left=right= NULL.
- When 13 inserted, 13 inserted in the Right subtree. Variable a=-1, bf[11]=-1. And since rotation is not needed as per code, return.
- When 15 is inserted, it will go to the right of 13. Since bf[11]=-1, rotation may occur(s will point to 11). And finally since s[bf]==r[bf], so it is a case of **Single Rotation.**

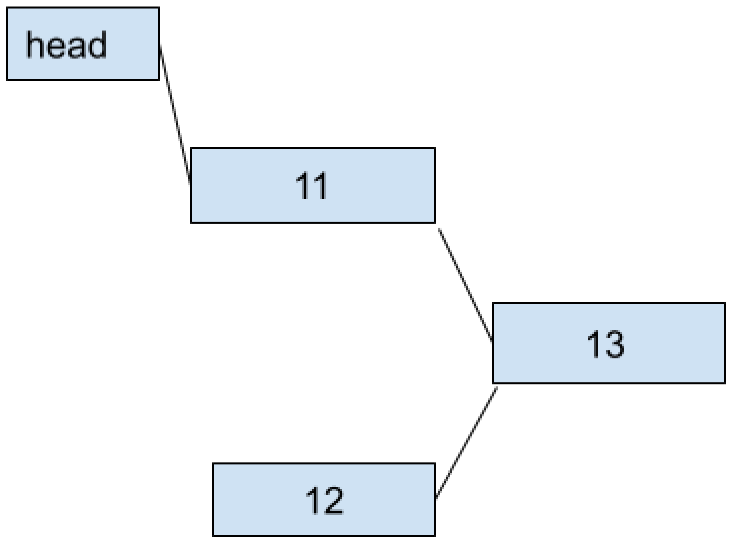




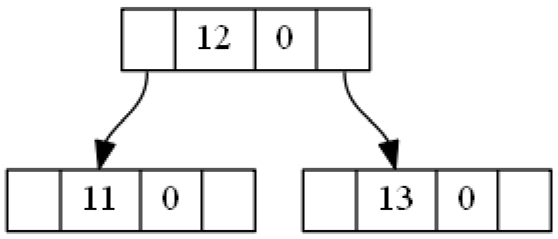
Above is output from running my program. Similar is the case of LL rotation.

Test case 2: Insert 11 13 12

- When 11 inserted, bf[11]=0. left=right= NULL.
- When 13 inserted, 13 inserted in the Right subtree. Variable a=-1. bf[11]=-1. And since rotation is not needed as per code, return.
- When 12 is inserted, it will go to the left of 13. Since bf[11]=-1, rotation may occur(s will point to 11). And finally since s[bf]==-(r[bf]). So it is a case of **Double Rotation.**



Following is my program output. Similar is case of LR Rotation



2. Delete function:

Deletion of a node is also a 3 step process. In deletion we require a *path stack* for storing the path, Which will help us change the balancing factor.

- Search for element

- Delete the node as per BST Deletion.
- Update bf of ancestors and balance if necessary.

Searching for a position is easiest. We just traverse through the tree using the ‘p’ pointer. While doing so we add each node to the *path stack.*

The intended node to be deleted could be either a leaf, one child or 2 child node. 1 child node and leaf node are almost the same. In 2 child nodes, we first find Inorder successor or predecessor and then do processing of swap and then delete either using leaf deletion or 1 child deletion.

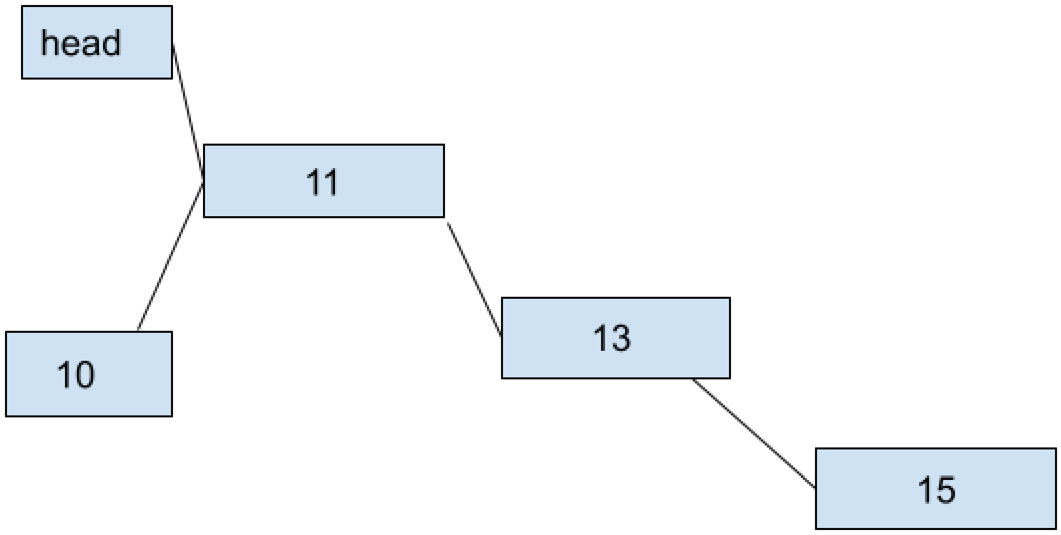
In the deletion step we set *parent to deleted child link*=NULL and free the pointer.

In updating the balance factor we just pop the element from the stack and check for a specific value of the balance factor and finally decide whether to update the balancing factor or not.

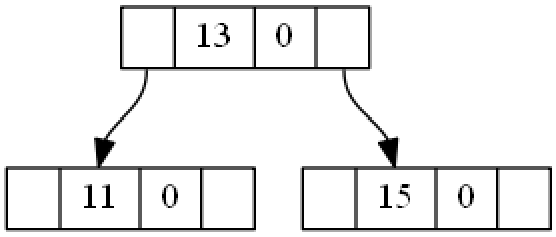
If bf of a node is 1 or -1 rotation might be needed otherwise simply updating the balancing factor would suffice.

Rotation step is the same as in insertion.

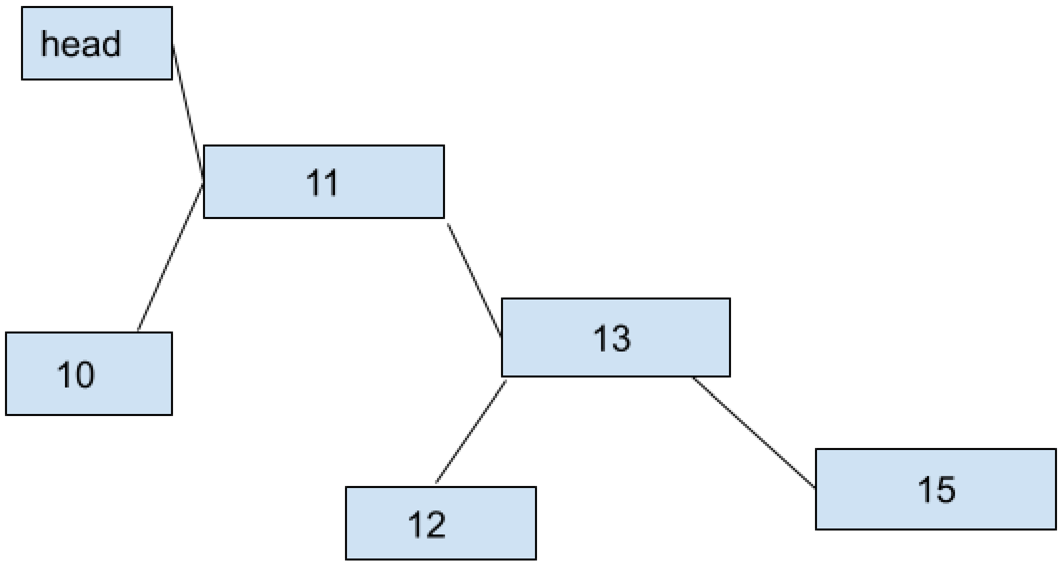
Test case 1: Delete 10. In following tree [Balancing required in ancestor]



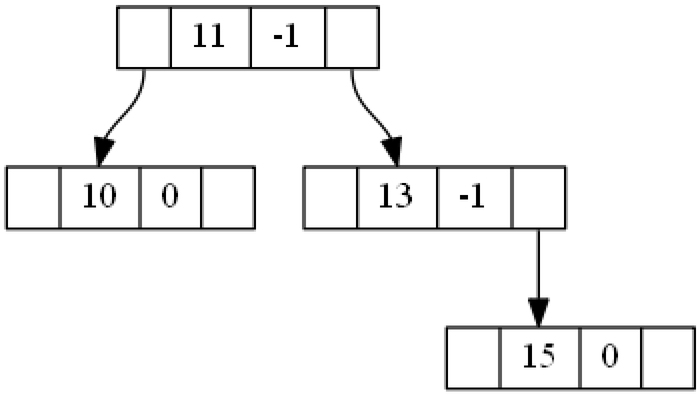
Following is my program output



Test case 2: Delete 12. In the following tree [NO Balancing required in ancestor].



Following is my code output



3. Searching

Searching in AVL trees happens in O(log n). Search procedure is easy, just compare with the elements from the root. If the key is less than the root's data, search the left subtree, else search the right subtree and continue until the key is found. If NULL is reached, output: *element is not present.*

4.Print tree:

This function outputs a **DOT file** which can be used to visualise the tree. File handling functions like open, close, write etc have been used. I used a level-order traversal rather than a recursion to identify left and right of a specific node and update dot file accordingly.