CS 513

DS Lab

Assignment 2

AVL Tree

Abhijeet Padhy

Roll: 214101001

INDEX

Basic Information

This is a console application which starts with the following menu:

```
λ avl_tree_main.exe

This is an implementation of AVL Tree
-------------------------------------------------------
1. Insert an element
2. Delete an element
3. Search for an element
4. Print an image of the tree
5. Insert a series of elements
6. Clone a tree and print it.

Press 0 to quit.
Enter Your Choice:
```

We have the required operations in the menu choosing which we will modify our AVL tree. We can insert a single element, delete an element or search for an element. If we need to insert a series of elements, then option 5 must be chosen.

```
Press 0 to quit.
Enter Your Choice: 5

------------------OPERATION-------------------
Enter the number of elements you want to insert: 6
Insert all the elements:
3 2 4 1 5 6
```

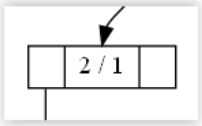
First we need to enter the number of elements we want to insert, followed by the series of elements.

To print an image of the tree, use the option 4. This will create a file graph.gv in the same directory. Use the below command to convert it to a png file named graph.png:
dot -Tpng graph.gv -o graph.png

To clone a tree and then print it, we can use option 6. It makes use of copy constructor. This time the output file will be named cloned_graph.gv. Use the below command to convert it to a png file:

dot -Tpng cloned_graph.gv -o cloned_graph.png

Structure of a node:



Here 2 is the key value of the node and 1 is its balance factor which is the stands for
height(left sub tree) – height(right sub tree)

Insert

The process of insertion is iterative. If element to be inserted is already present in the tree, then an exception is thrown. “Element Already Exists Exception!” is thrown to the caller of this function. If the element is not present, it can either be less than or greater than the current node’s key value. Accordingly we will either move to left sub tree or right sub tree. In this manner, the correct position of the element to be inserted is found keeping in mind the property of a Binary search tree. The element is inserted.

After this step, balance factors are updated.

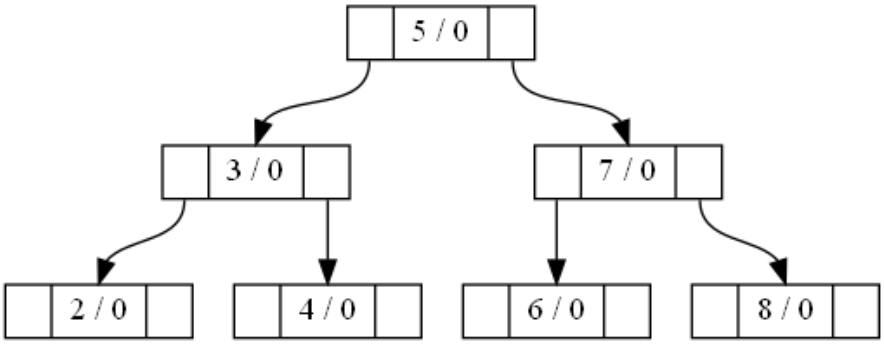
If a rotation is required, appropriate rotation (single rotation or double rotation) is made and again balance factors are updated.

At the end the tree is a height balanced BST with balance factors of any node being +1, 0 or -1.

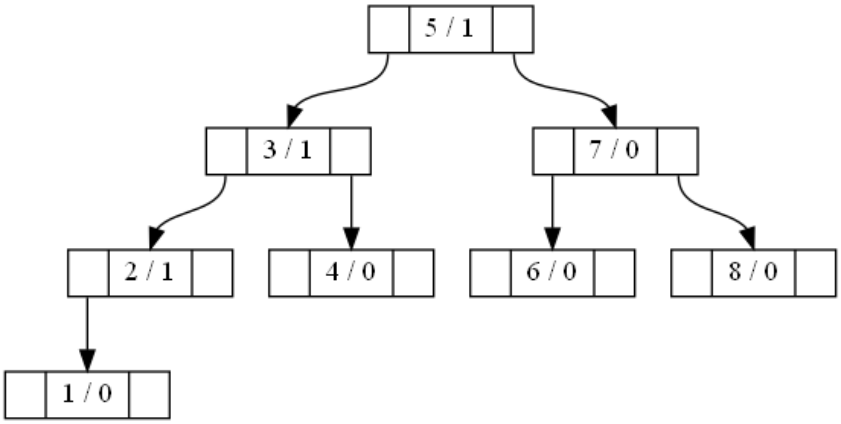
While doing the process of insertion, we maintain four major pointers mainly. P, S, T, and Q. P will move down the tree as we search for the correct place to insert. S will point to the place where rebalancing may be necessary. T will point to the parent of S. Q points to the newly inserted node.

After insertion, we need to adjust balance factors. Balance factors on nodes between S and Q need to be changed from 0 to +/-1. If K < KEY(P), then we set BF(P) = +1, otherwise BF(P) = -1. BF of Q, will be 0, as it is newly inserted and has no children.

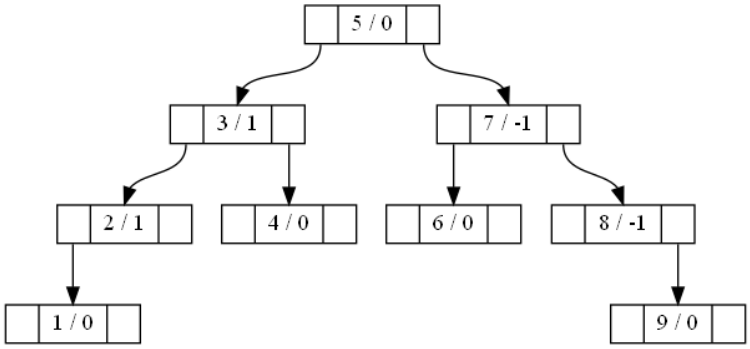
Example1: Consider the tree:



On inserting element 1, the tree will become:



Notice how BF of nodes 3 and 2 were updated according to the position of the newly inserted node. If we further go on to insert 9, the following will be the output:



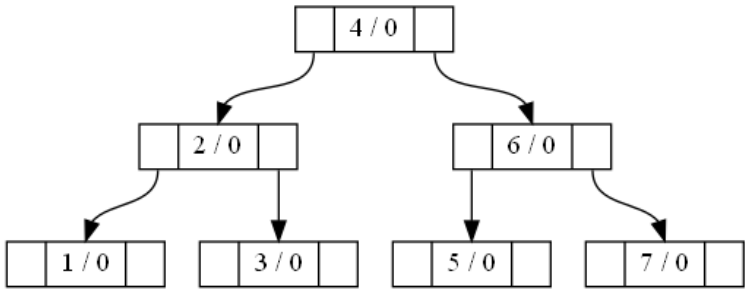
As can be seen the BF of node 7 and 8 were updated to -1 from 0.

We have not changed BF of node S. Based on the BF of node S, following cases will arise:

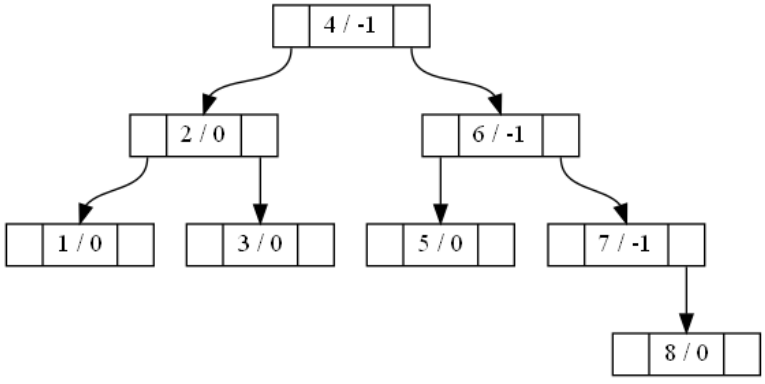
Case1: If BF(S) == 0, this means, the tree has grown higher, and so we need to update the BF here. If new node was added on left sub tree, then BF(S) = +1, otherwise BF(S) = -1.

Example1 explains this case when S was pointed to node 5 after the insertion, and it had a BF of 0. Since node 1 was inserted in the left sub tree of node 5, the BF gets updated to +1. This explains first part of case1, for the second case, look at the below example.

Example2: Now consider the below tree



We insert node 8.

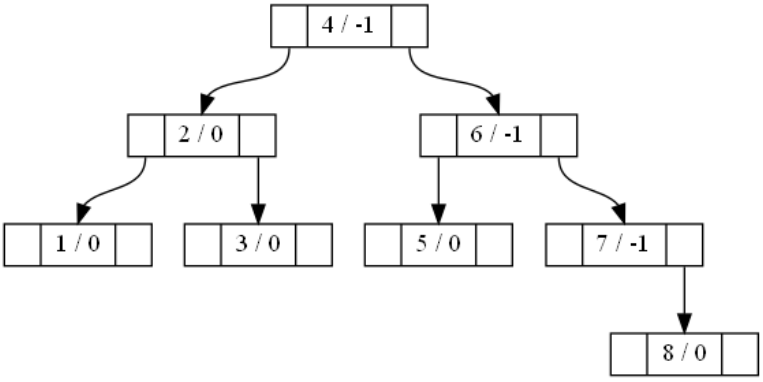


The S pointer points to node 4 in this case whose BF is 0. Since node 8 gets inserted on the right sub tree of node 4, the BF gets updated to -1 after insertion. This explains the second part of case1.

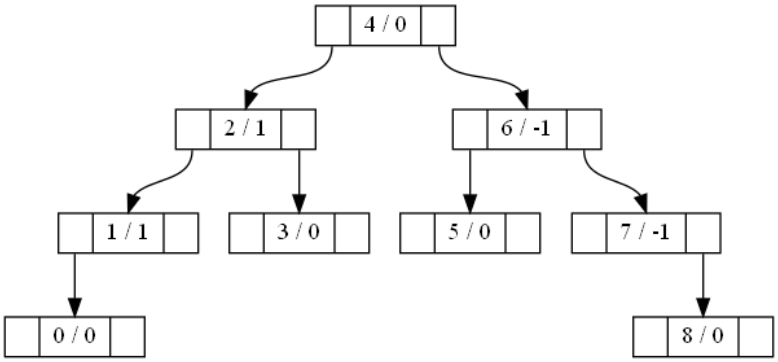
Case2: If BF(S) = +1 and insertion occurs on right sub tree of S, then make the BF of S to 0. On the other hand if BF(S) = -1 and insertion occurs on left sub tree of S, then make the BF of S to 0. This happens because the tree has got more balanced.

Consider diagram2 of Example1 after we have already inserted node 1 and we are going to insert node 9. In this case S points to node 5 whose BF is +1. We inserted node 9 into the right sub tree of node 5, so BF of node 5 gets updated to 0 as can be seen in the third diagram of Example1. Thus this demonstrates the first part of case2. For the second part, look at the below example.

Example3: Now consider the below tree:



We insert node 0:



The S pointer points to node 4 in this case whose BF is -1. We insert node 0 on the left sub tree of node 4. So BF of node 4 gets updated to 0. This explains the second part of case2.

Case3: In case1 or case2, there was no need of rotation. So it was sufficient to adjust the BFs only. But in case3, there will be rotations.

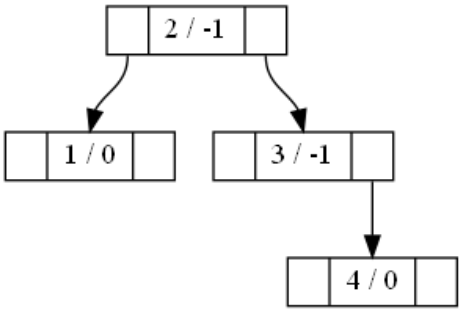
If K is less than key of S, then R stores Left Child of S. If K is greater than key of S, then R stores Right Child of S.

Case3a: Single Rotation is needed

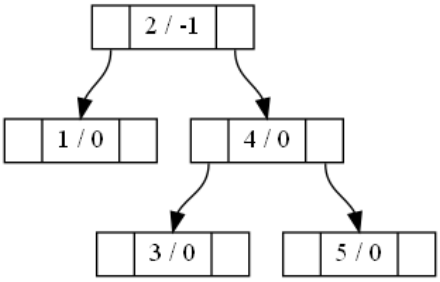
1. If bf of R is 1 and the inserted element is on the right sub tree of S. (Example4 below)
2. If bf of R is -1 and the inserted element is on the left sub tree of S. (Example 5 below)

The BF of both S and R nodes will be set to 0.

Example4: Consider the below tree:

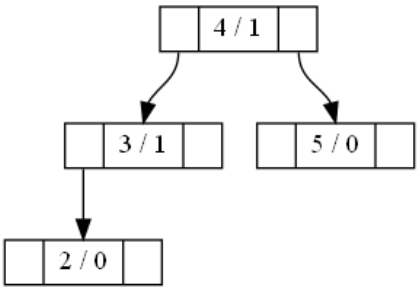


If we insert 5, there will be a requirement of single left rotation:

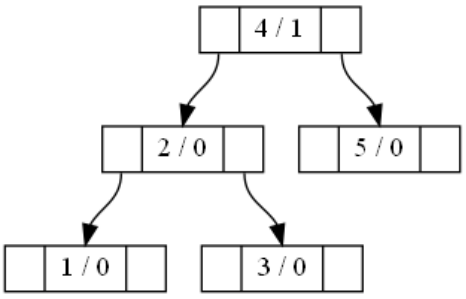


Note that S points to node 3, R points to Node 4. The BFs of both of them have been made 0 in the resultant tree.

Example5: Consider the below tree:



If we insert 1, there will be a requirement of single right rotation:



Note that S points to node 3, R points to Node 2. The BFs of both of them have been made 0 in the resultant tree.

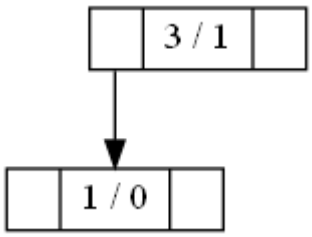
Case3b: Double Rotation is needed:

1. If BF of R is +1 and element is inserted on left sub tree of S. P is the right child of R.
2. If BF of R is -1 and element is inserted on the right sub tree of S. P is the left child of R.

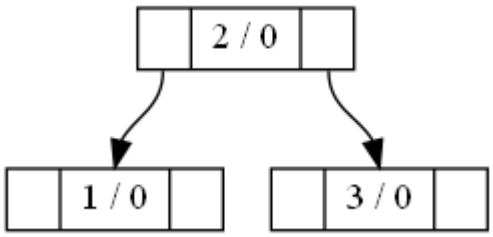
If key is inserted on left sub tree of S, a = -1
If key is inserted on right sub tree of S, a = +1
With the above definitions of a, S, P, R, the BFs of these nodes get modified according to the following conditions:

```
if(P->bf == -a){
    S->bf = a;
    R->bf = 0;
}else if(P->bf == 0){
    S->bf = 0;
    R->bf = 0;
}else if(a == 1){
    S->bf = 0;
    R->bf = -a;
}
P->bf = 0;
```

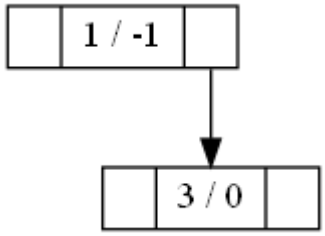
Example6: Left Right Rotation



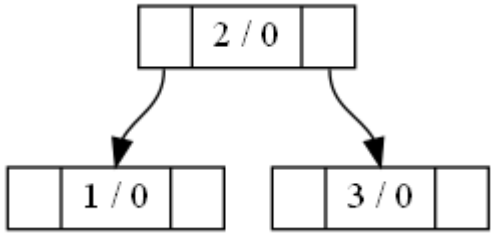
Insert 2



Example7: Right Left Rotation

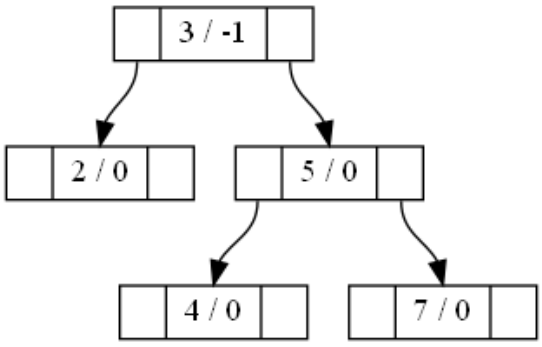


Insert 2

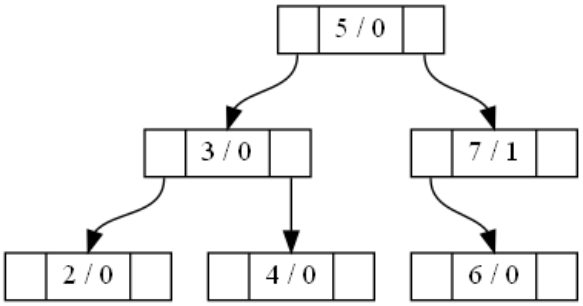


Some additional Examples:

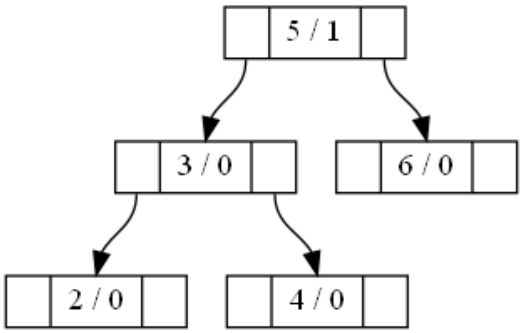
Example8: Single Rotation



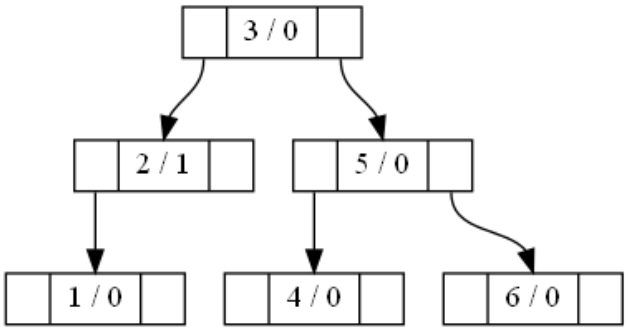
Insert 6



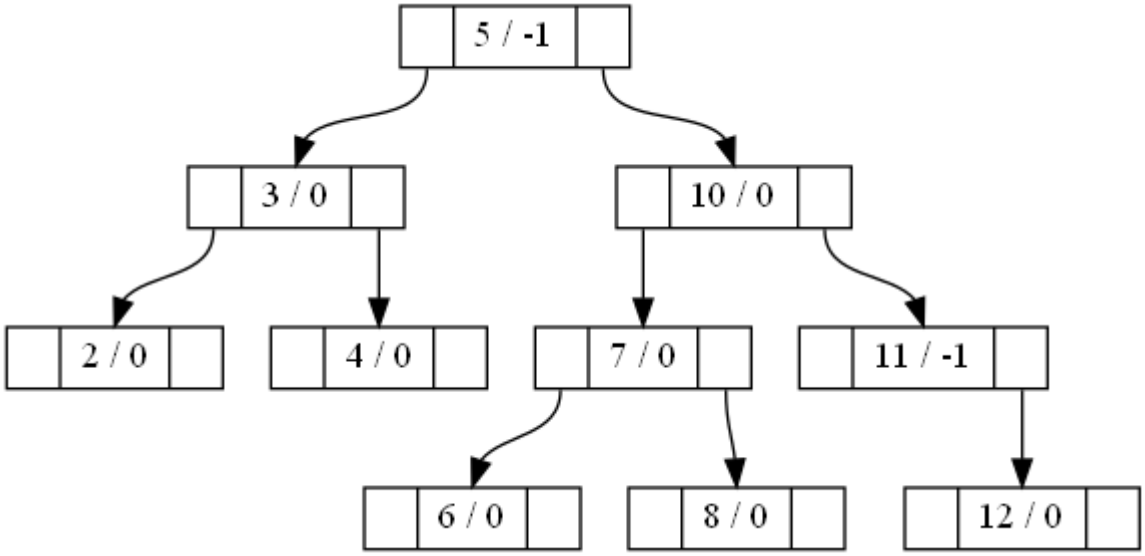
Example9: Single Rotation



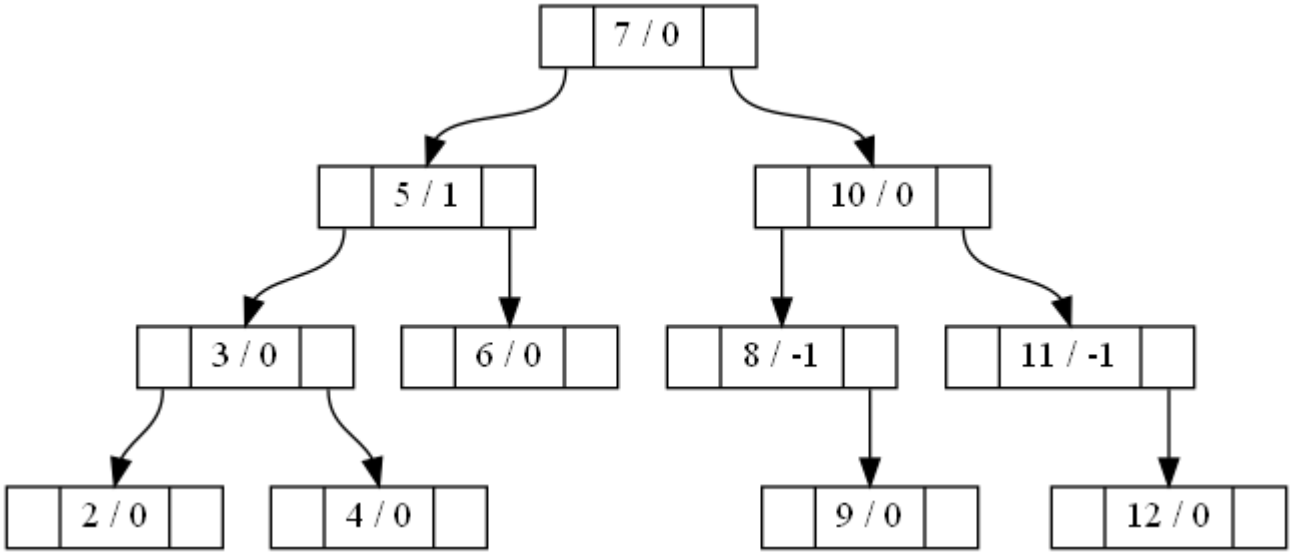
Insert 1



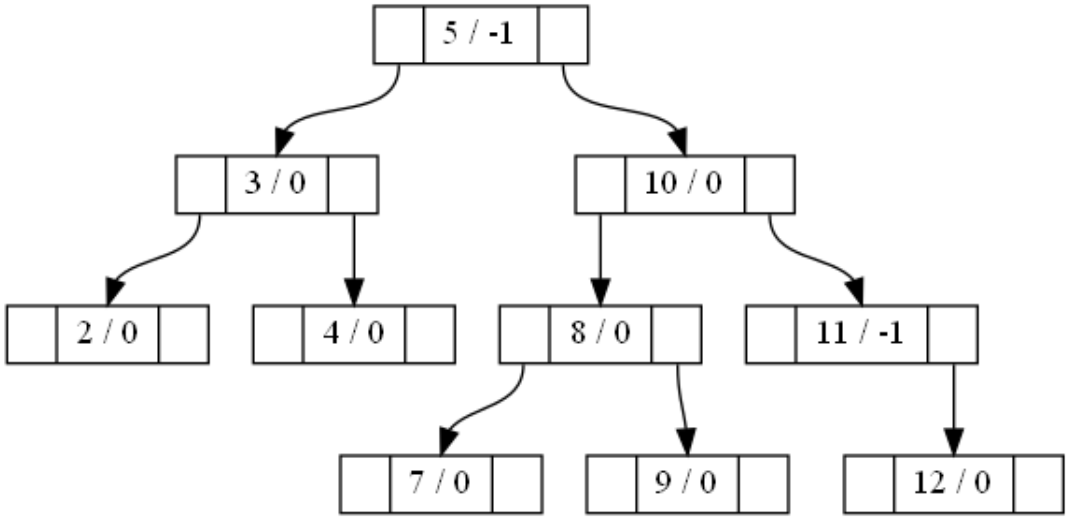
Example9: Double Rotation



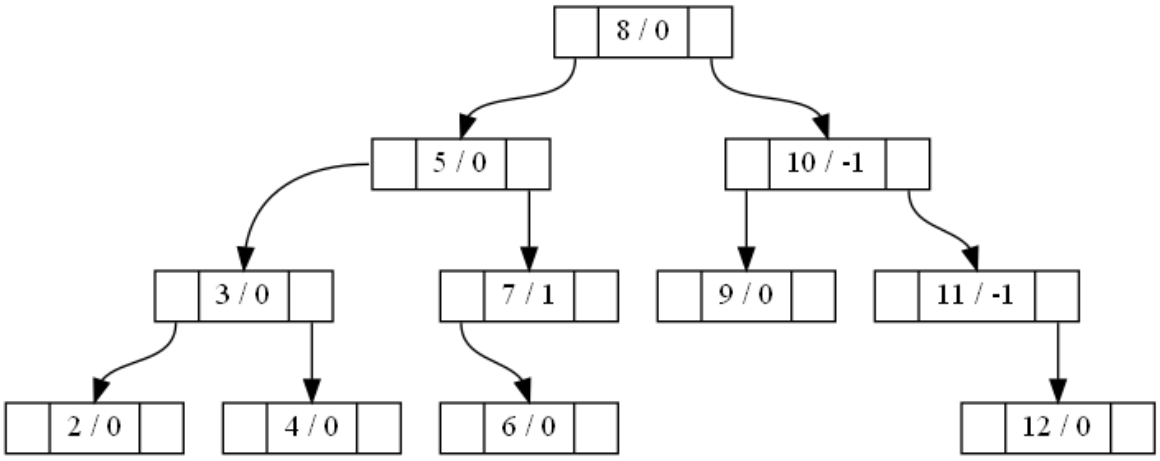
Insert 9:



Example10: Double Rotation



Insert 6



Delete

We first proceed with the deletion of a node. At first three cases will arise:

1. Leaf Node: Delete directly
2. Node has only one child: The parent will point to the child of the node to be deleted.
3. Node has two children: The node's key value is replaced by the in order successor and the latter is then deleted which is either a leaf node or a node with a single child.

While we search for the node to be deleted, we store the path in a stack.

After the deletion step has occurred, we proceed to correct the BFs and do the required rotations.

For a particular node on the path, the variable a stores either -1 or +1. If node to be deleted is present on the left sub tree of the node on the path, a stores -1, otherwise it stores +1.

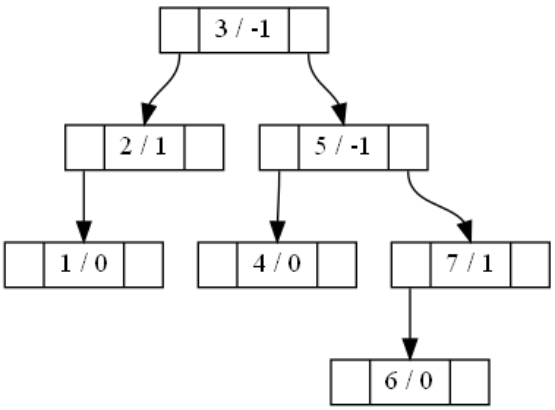
We traverse all the nodes on the stack one by one and correct the BFs for them.

Case1: If BF is equal to –a, we set the BF of the node to 0. Continue for next node on stack.

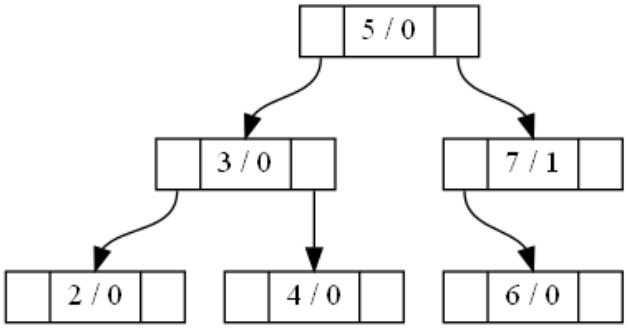
Case2: If BF is equal to 0, we set the BF of the node to a. Terminate.

Case3: If BF is equal to a, rebalancing is required. Since a lot of cases arise, we will check each case via examples:

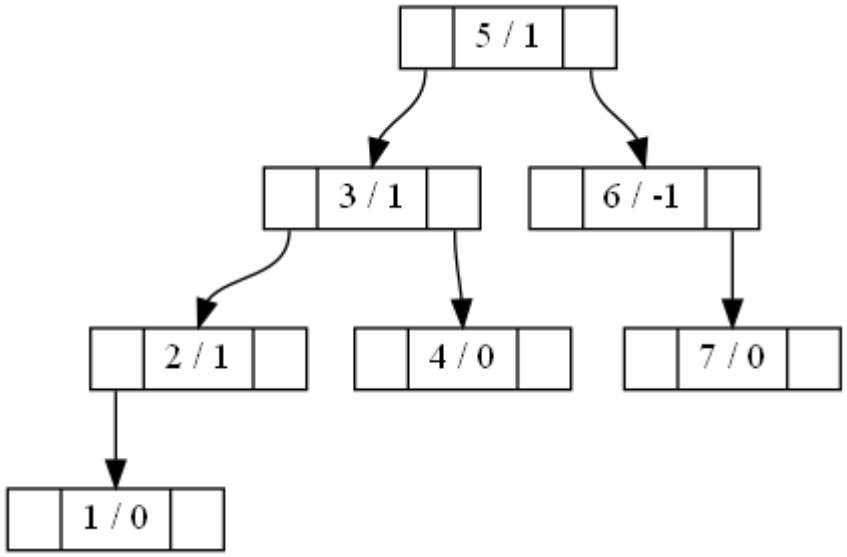
Case 3.1a:



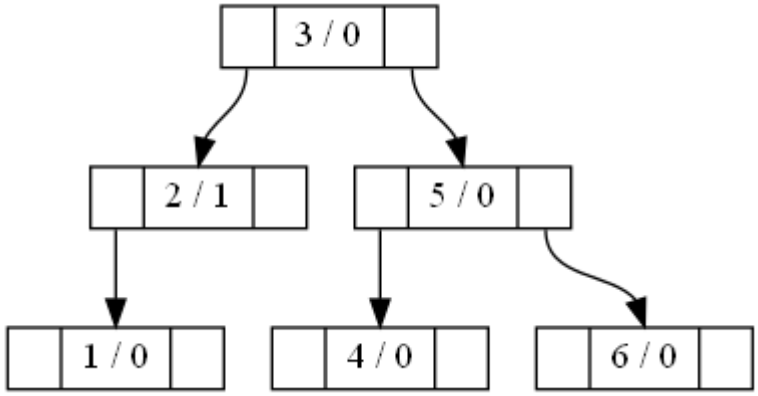
Delete 1



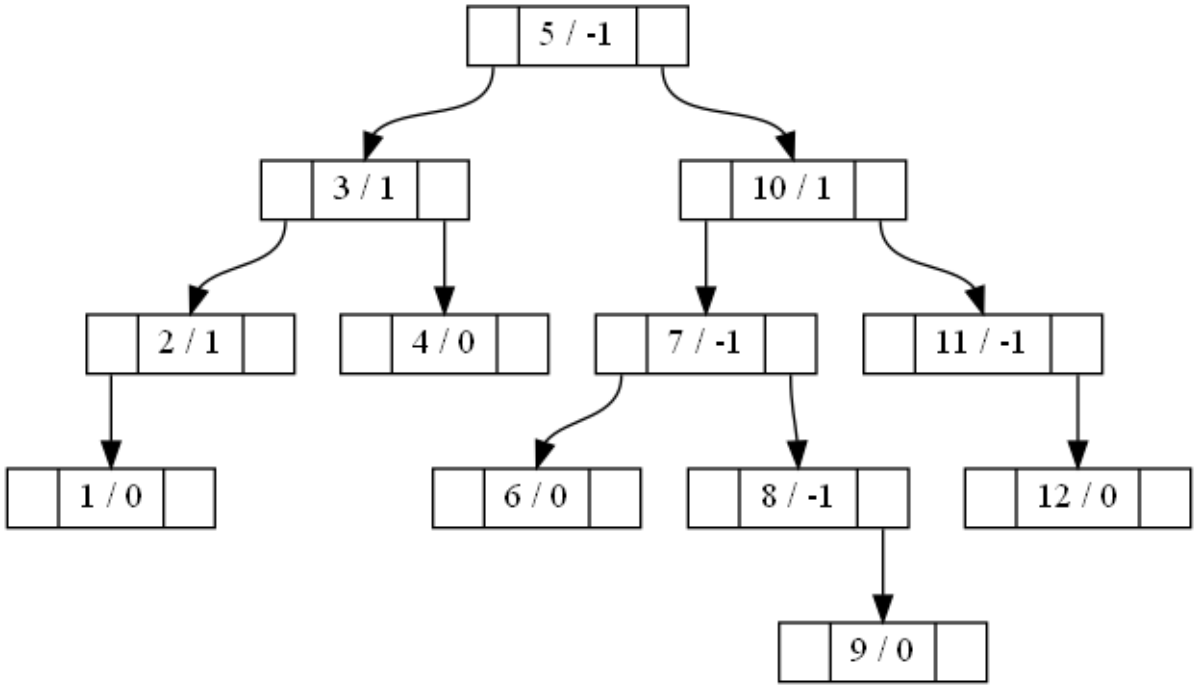
Case 3.1b:



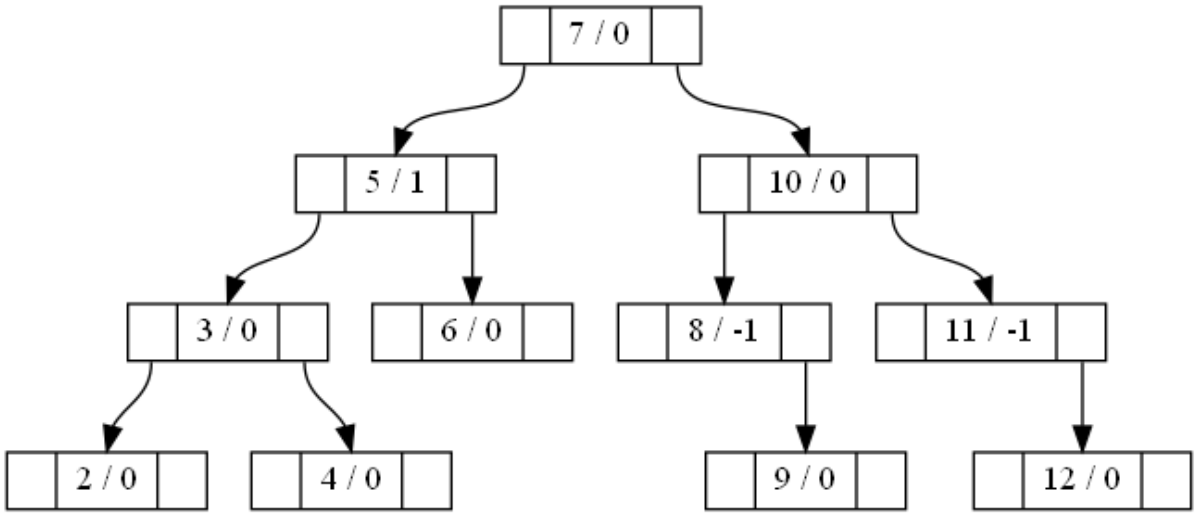
Delete 7:



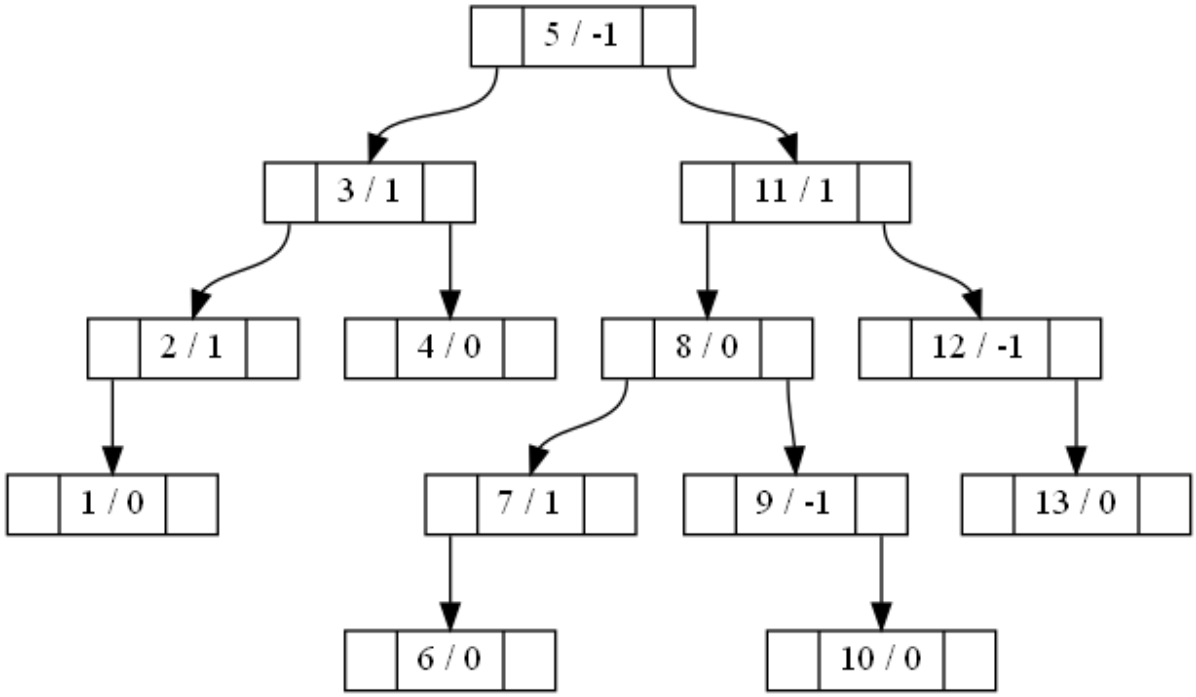
Case3.2a+:



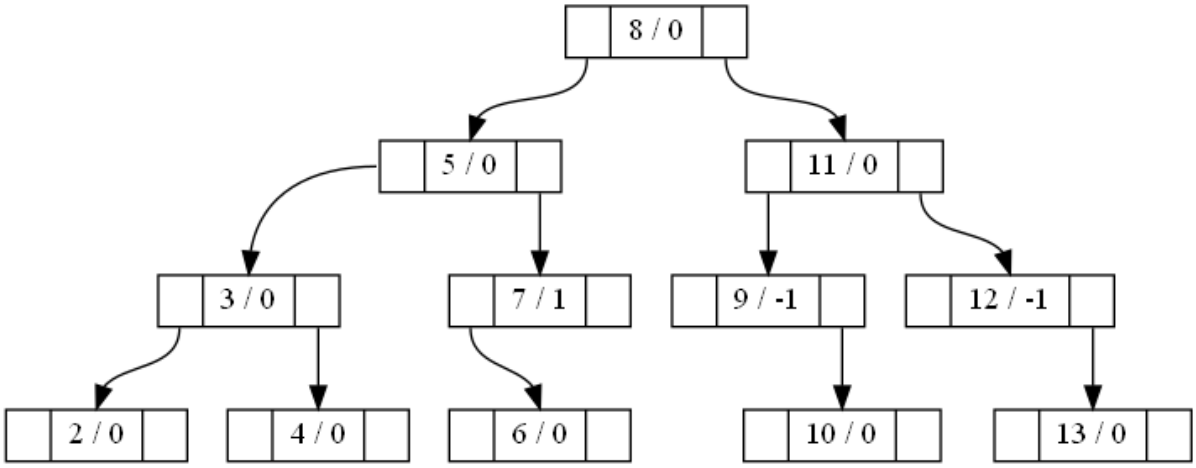
Delete 1:



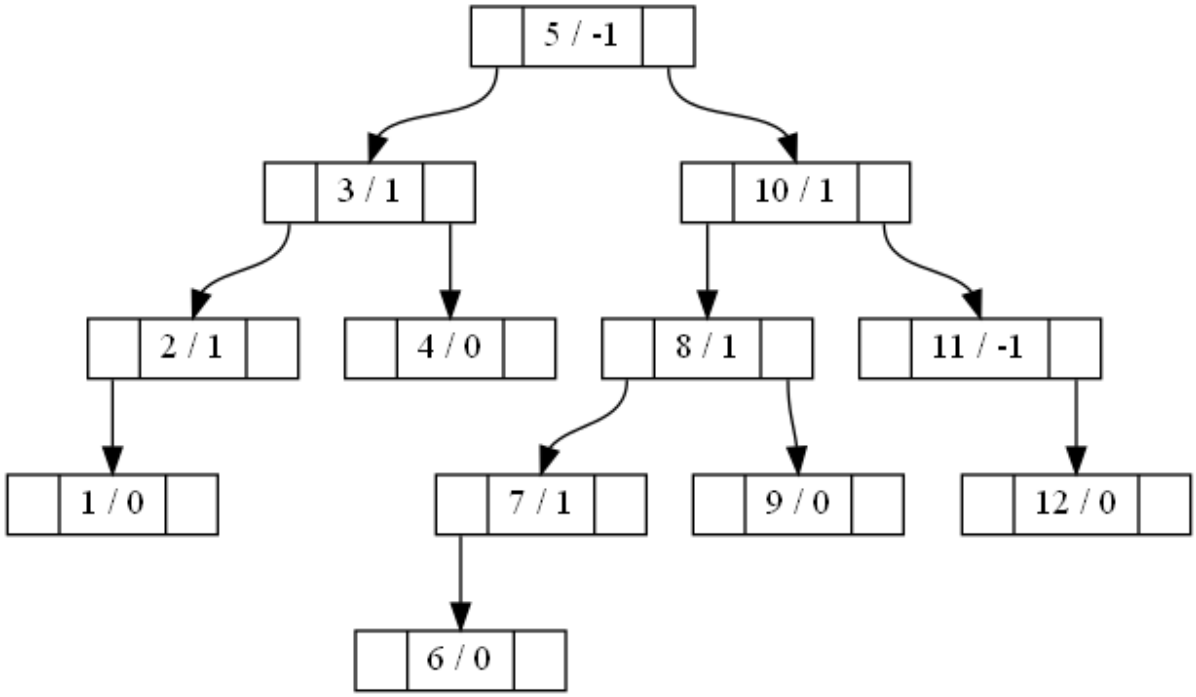
Case3.2a0:



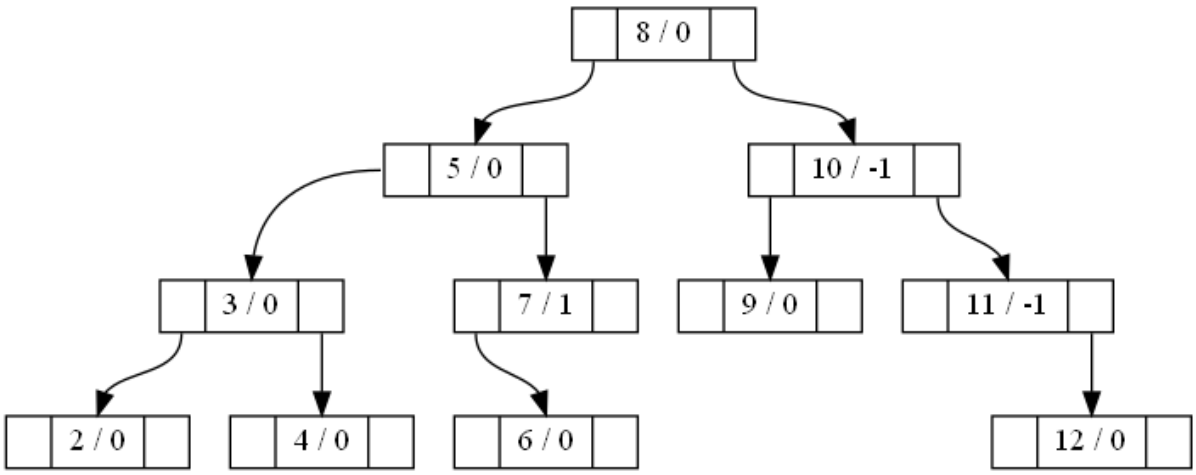
Delete 1:



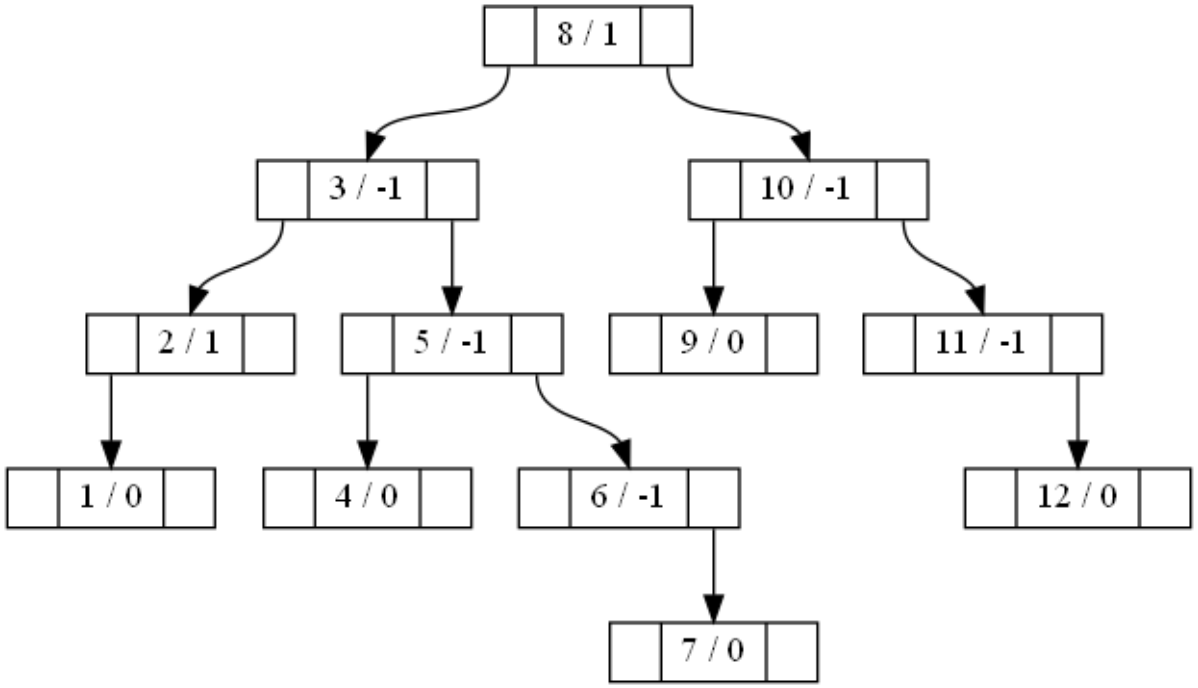
Case 3.2a-:



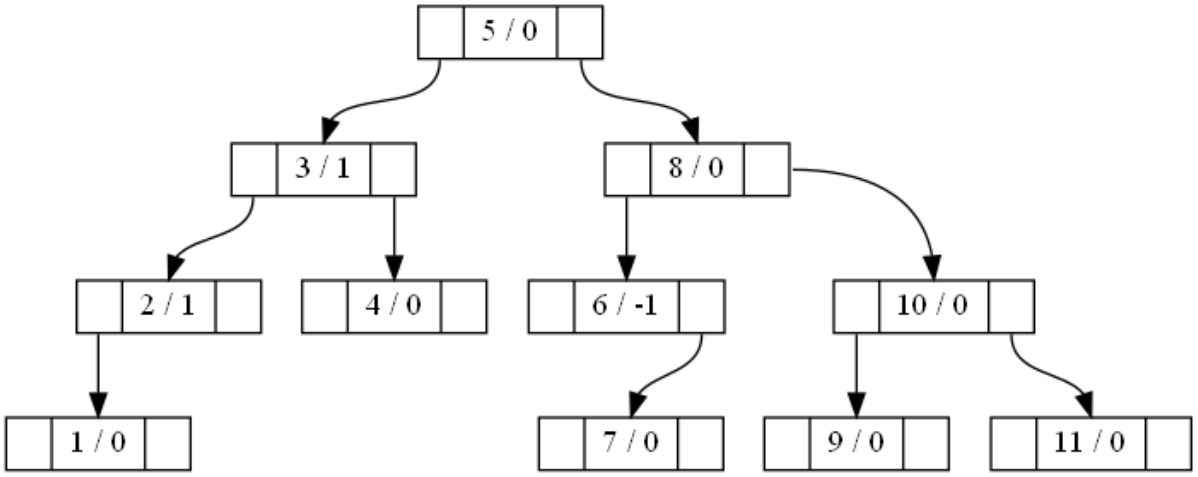
Delete 1:



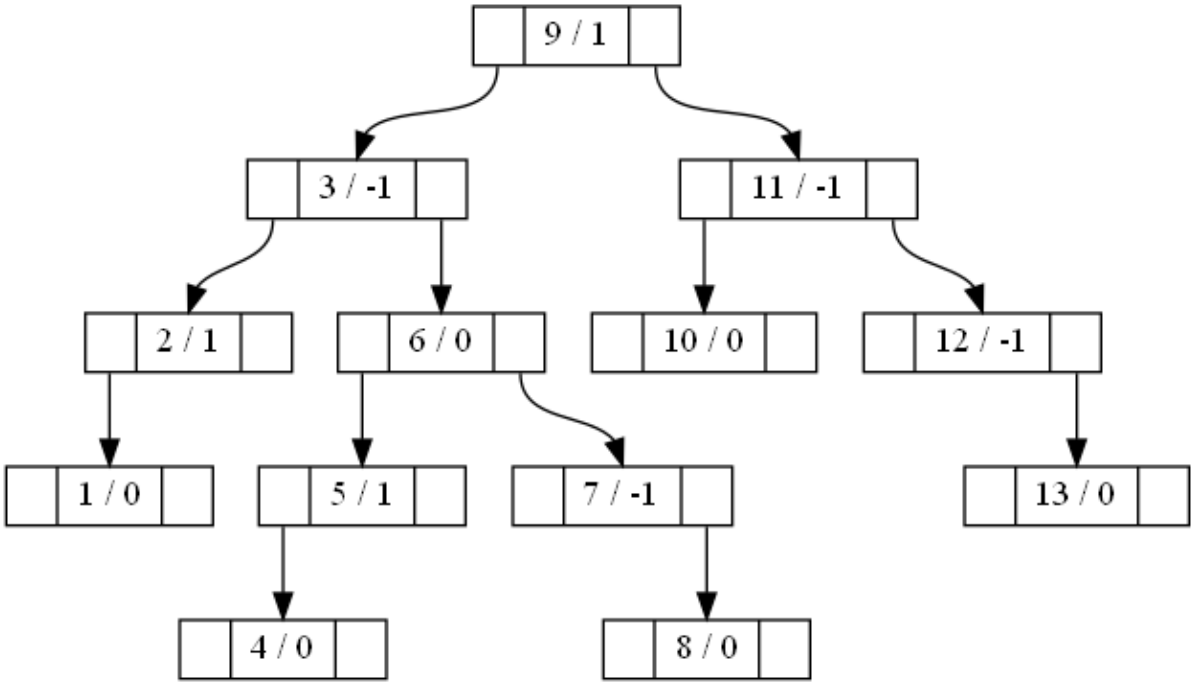
Case 3.2b+:



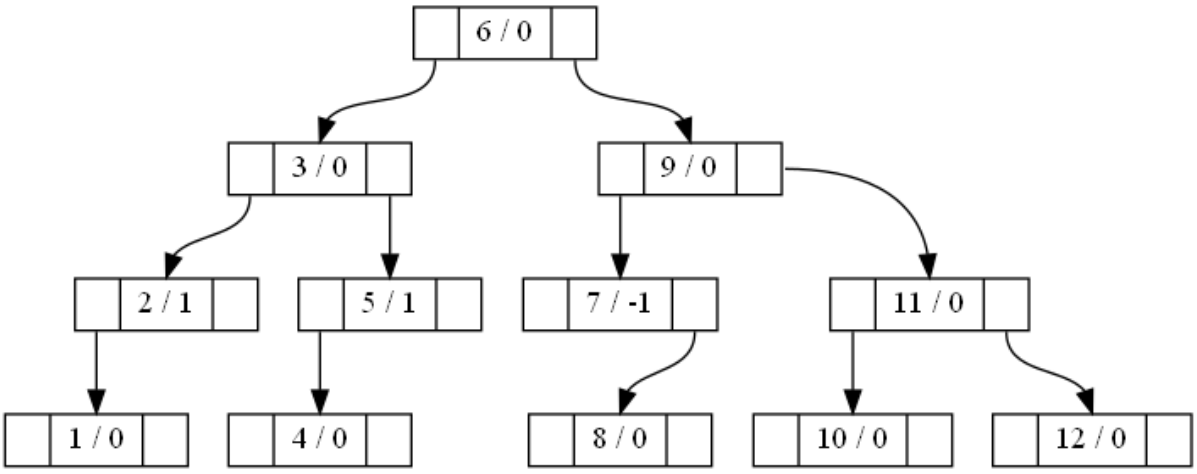
Delete 12:



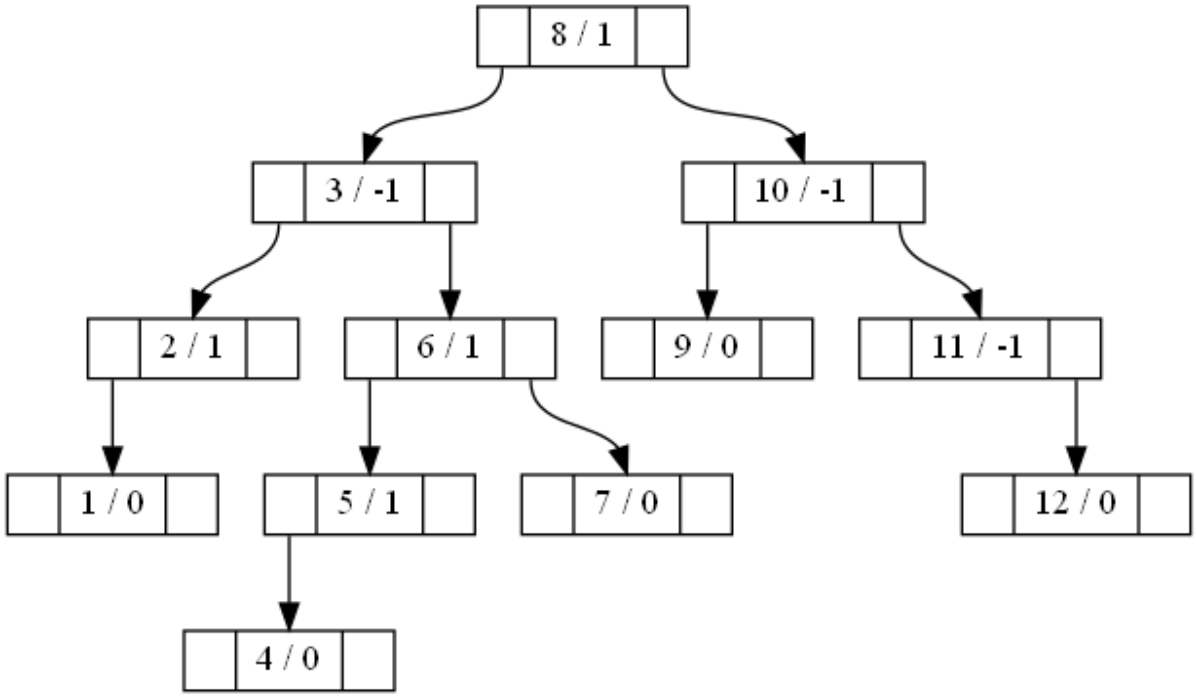
Case 3.2b0:



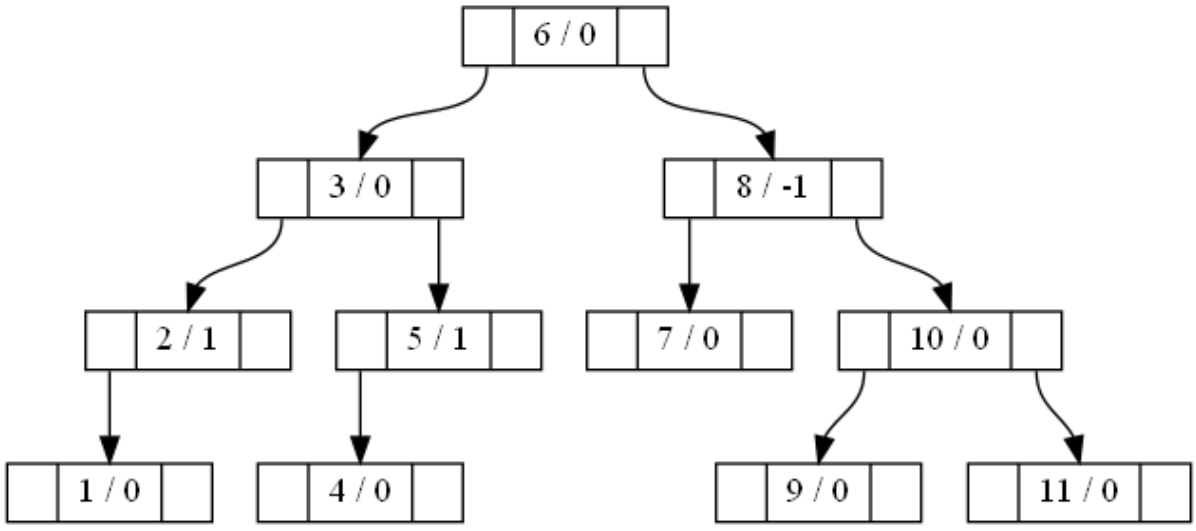
Delete 13:



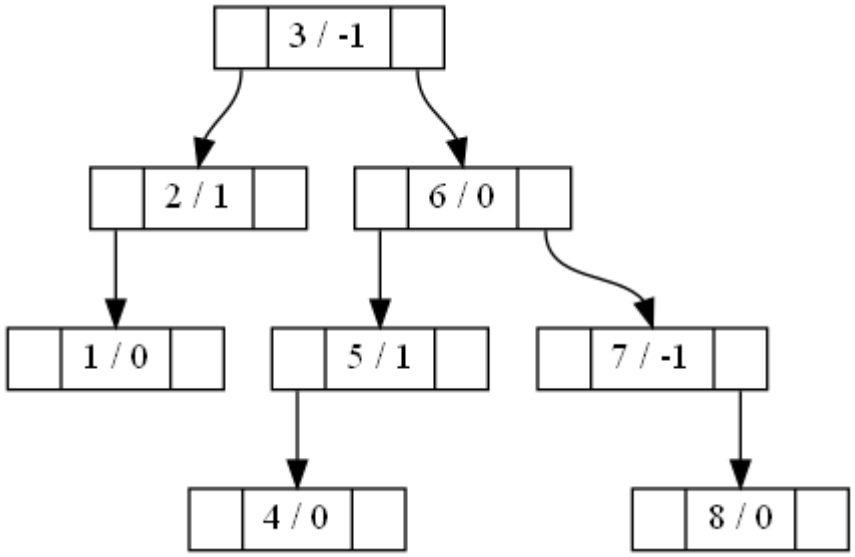
Case 3.2b-:



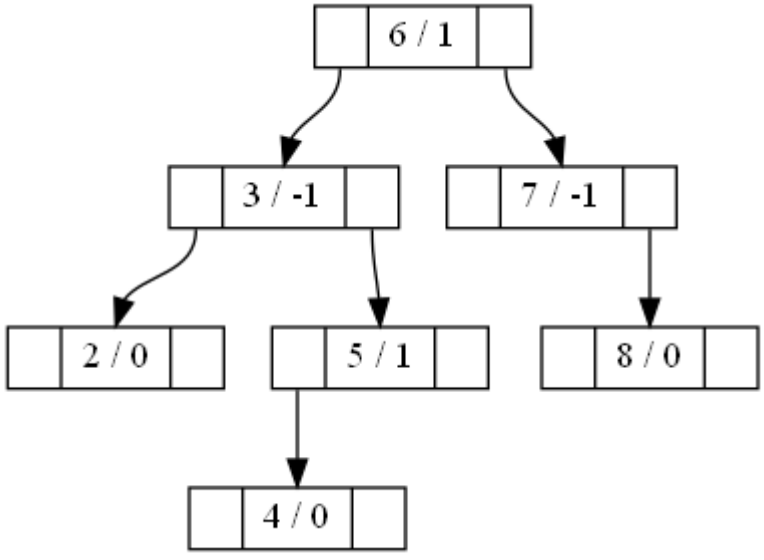
Delete 12:



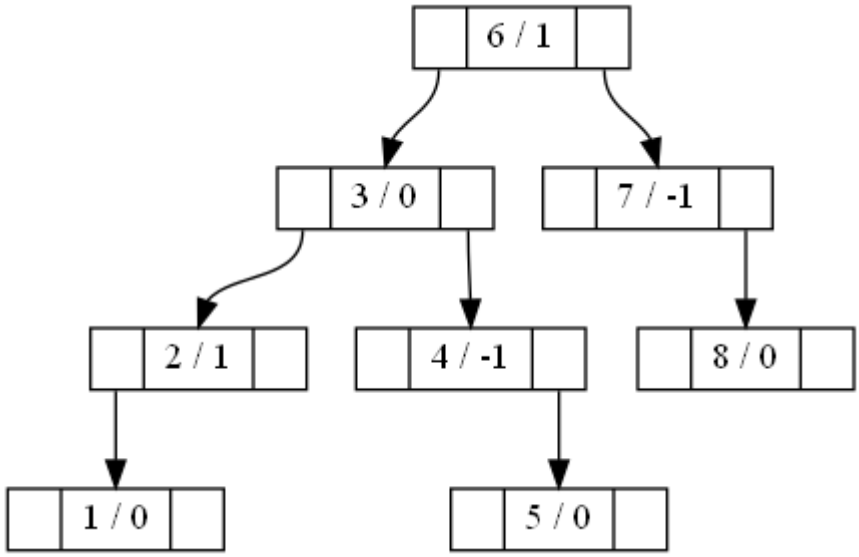
Case 3.3a:



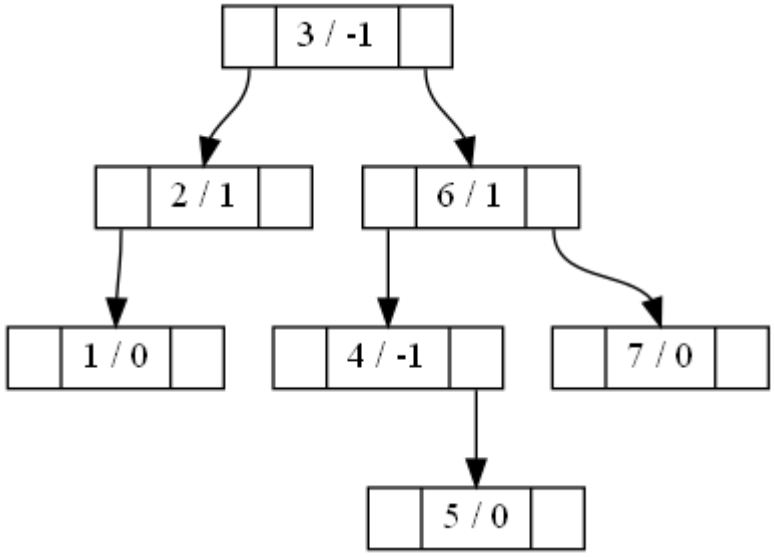
Delete 1:



Case 3.3b:



Delete 8:



Search

Searching is very simple. We search for the element recursively. If element value is equal to the current node's values, return true as the element is found. If element's value is less than node's key value, search recursively in left sub tree. Otherwise search in right sub tree. If node is NULL, return false. So if we cannot find the element, we will ultimately reach NULL and hence return false.

Print Tree

The purpose of this function is to generate .gv file which will be parsed by graphwiz utility to produce an image of the tree. We traverse the tree in a preorder manner. When we arrive at a node in our traversal, we add node in the .gv file. Then we recur for left and right subtrees. When we return from recursive call from a sub tree, we add an edge from the root node to the left or right sub trees' root node. At the end we have the complete .gv file. From outside of the program, run the following command to convert a .gv file to .png file which is our required image.

dot -Tpng graph.gv -o graph.png

Constructor

It simply sets the root pointer to NULL.

Copy Constructor

This creates a new tree which is an exact clone of an existing tree. We pass the tree to be cloned in argument. Copy constructor creates a new tree object, but the child nodes are not present yet. To clone all the tree nodes, we call AVL_Clone method. When the AVL_Clone method is called, a new node is created if the current node in the original tree is not NULL. Then key value is copied from the original node to new node. Then if there is to be a left sub tree, then left recursion is made and root node of left sub tree is set as left child of the new node. Similarly we go for right sub tree recursion. We follow a pre order traversal in this function. Finally we have an exact clone of the existing tree.

Destructor

This is called before we exit the program using delete(ptr) function. This helps to free all the dynamically allocated memory during program runtime. The destructor calls the AVL_ClearTree function whose work is to recursively delete all the tree nodes. At the end, the tree object is also deleted.

If some allocated memory is not freed after program exit, there will be memory leak which will be detected using valgrind.

Assignment operator overloading

Suppose we have two tree objects obj1 and obj2. We need to copy the contents of obj2 to obj1. In such a case we will use assignment operator overloading.

obj1 = obj2;

This will call the operator= method. First we check if both the objects under consideration are different. If they are same, simply return. Otherwise call the AVL_ClearTree function on obj1 to clear the existing tree of obj1 and then call AVL_Clone function on obj2. This will make a clone of obj2 and then we store the clone in obj1. Thus obj1 will now have a clone of the tree stored in obj2.

Memory Leak Check

There was no memory leak in the program as verified by valgrind tool.

```
==3181== Memcheck, a memory error detector
==3181== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3181== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3181== Command: ./avl_tree_main
==3181==
==3181==
==3181== HEAP SUMMARY:
==3181==     in use at exit: 0 bytes in 0 blocks
==3181==   total heap usage: 36 allocs, 36 frees, 77,272 bytes allocated
==3181==
==3181== All heap blocks were freed -- no leaks are possible
==3181==
==3181== For lists of detected and suppressed errors, rerun with: -s
==3181== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```