CS 513

DS Lab

Assignment 2

Abhijeet Padhy

Roll: 214101001

# INDEX

1. Insert
2. Delete
3. Search
4. Print Tree
5. Constructor
6. Destructor
7. Assignment Operator Overloading

# 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.

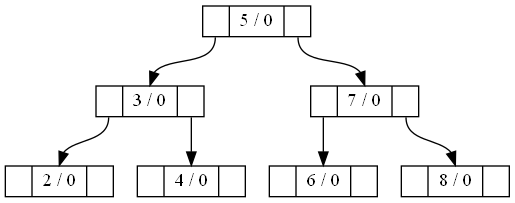
If a rotation is required, appropriate rotation (single rotation or double rotation) is made and again balance factors are updated.

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

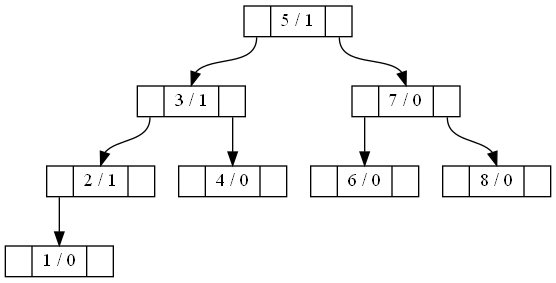
While doing the process of insertion, we maintain four major pointers mainly. P, S and T. P will move down the tree as we search for the correct place to insert. S will point to the place where rebalancing may be necessary. T will point to the parent of S. Q points to the newly inserted node.

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.

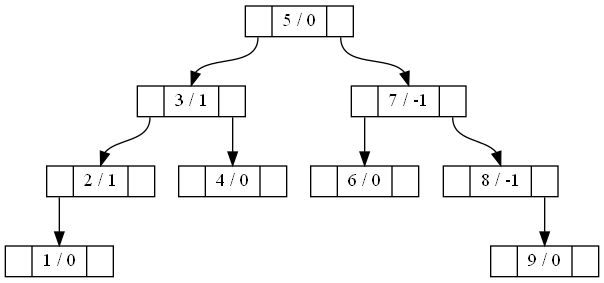
**Example1**: Consider the tree:



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:

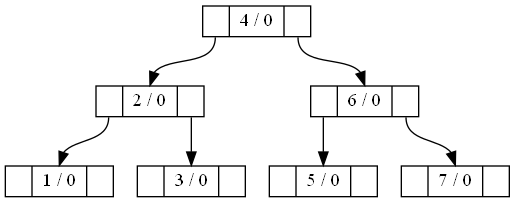


As can be seen the BF of node 7 and 8 were updated to -1 from 0.

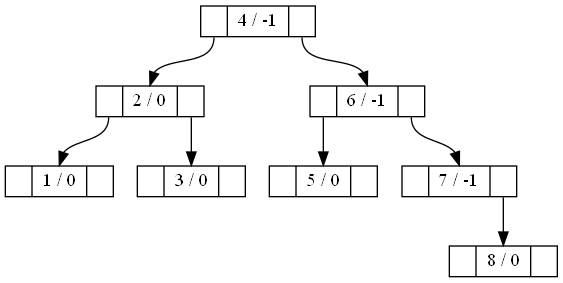
We have not changed BF of node S. Based on the BF of node S, following cases will arise:

**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.

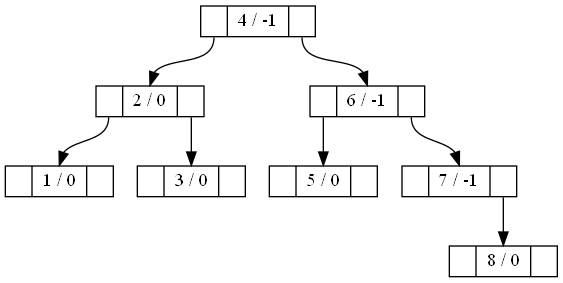


The S pointer points to node 4 in this case whose BF is 0. Since node 8 gets inserted on the right sub tree of node 4, the BF gets updated to -1 after insertion. This explains the second part of case1.

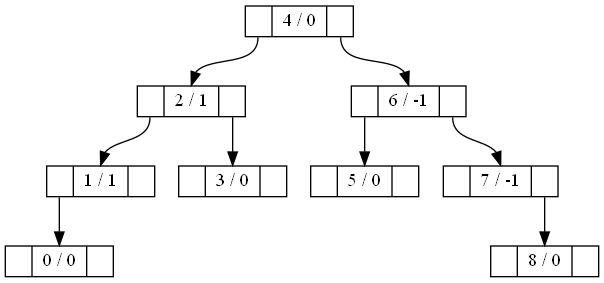
**Case2**: If BF(S) = +1 and insertion occurs on right sub tree of S, then make the BF of S to 0. On the other hand if BF(S) = -1 and insertion occurs on left sub tree of S, then make the BF of S to 0. This happens because the tree has got more balanced.

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.

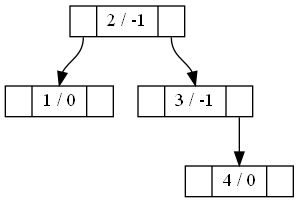
If K is less than key of S, then R stores Left Child of S. If K is greater than key of S, then R stores Right Child of S.

**Case3a**: Single Rotation is needed

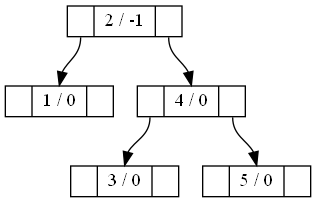
1. If bf of R is 1 and the inserted element is on the right sub tree of S. (Example4 below)
2. If bf of R is -1 and the inserted element is on the left sub tree of S. (Example 5 below)

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

**Example4**: Consider the below tree:

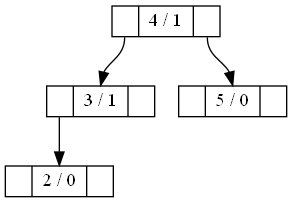


If we insert 5, there will be a requirement of single left rotation:

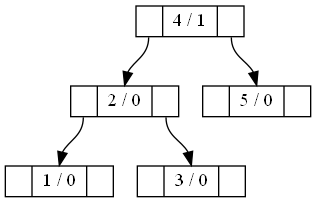


Note that S points to node 3, R points to Node 4. The BFs of both of them have been made 0 in the resultant tree.

**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.

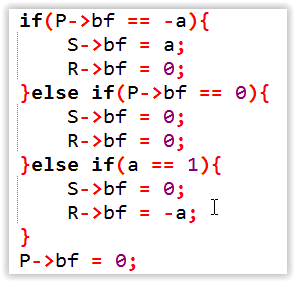
**Case3b**: Double Rotation is needed:

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.

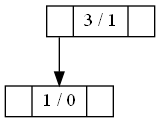
If key is inserted on left sub tree of S, a = -1

If key is inserted on right sub tree of S, a = +1

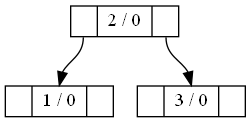
With the above definitions of a, S, P, R, the BFs of these nodes get modified according to the following conditions:



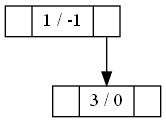
**Example6**: Left Right Rotation



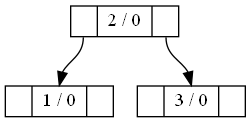
Insert 2



**Example7**: Right Left Rotation



Insert 2



# 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.

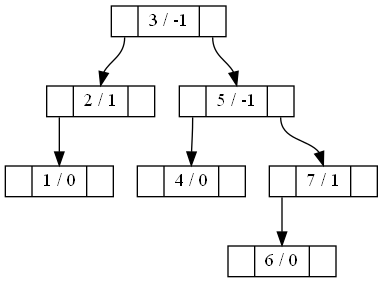
We traverse all the nodes on the stack one by one and correct the BFs for them.

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

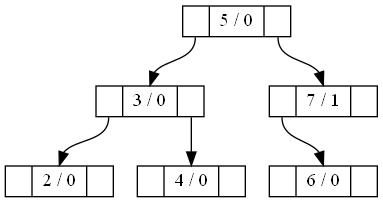
**Case2**: If BF is equal to 0, we set the BF of the node to a. Terminate.

**Case3**: If BF is equal to a, rebalancing is required.

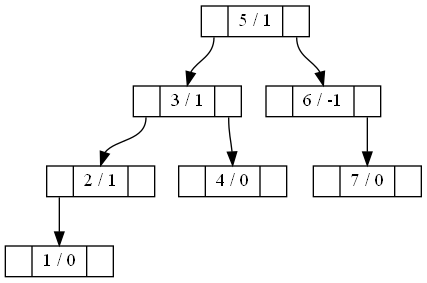
**Case 3.1a:**



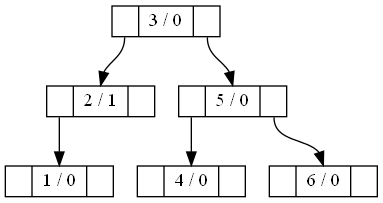
Delete 1



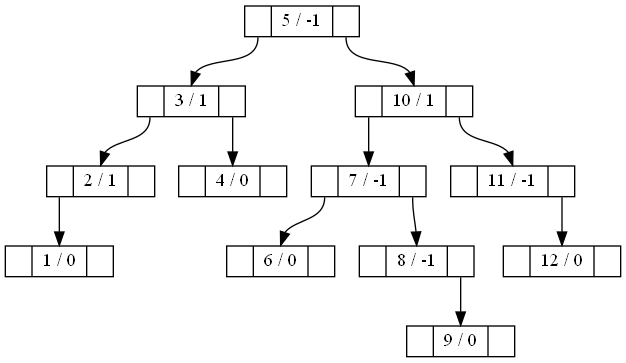
**Case 3.1b:**

****

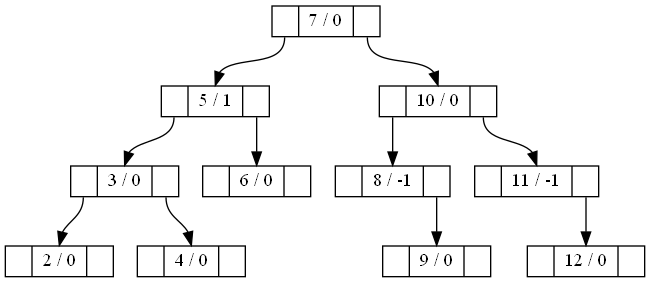
Delete 7:



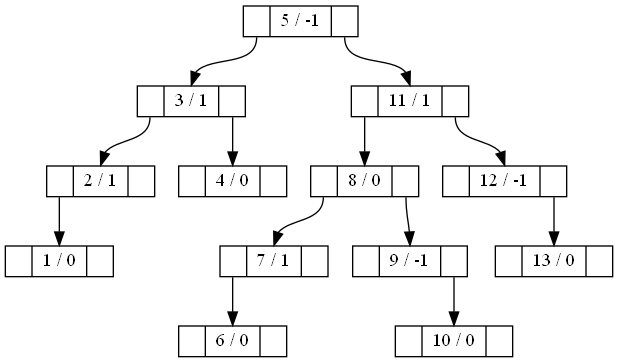
Case3.2a+:



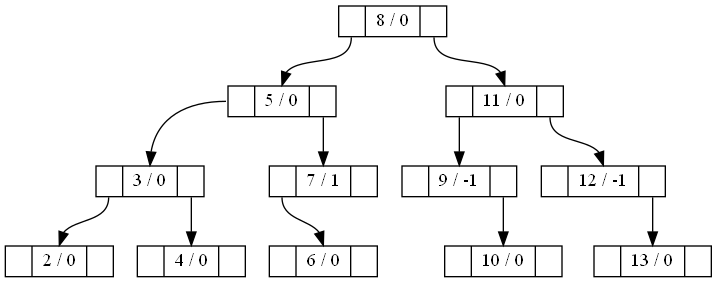
Delete 1:



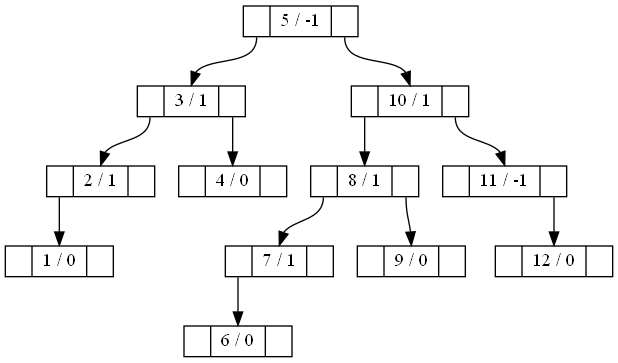
Case3.2a0:



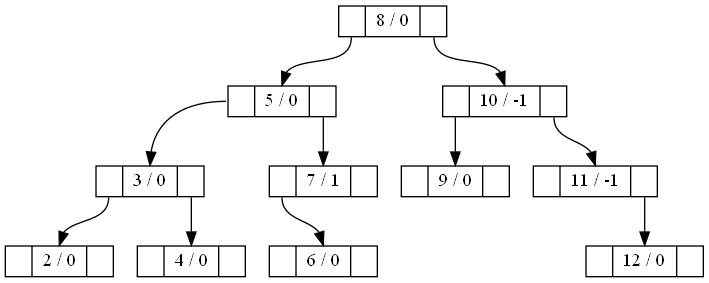
Delete 1:



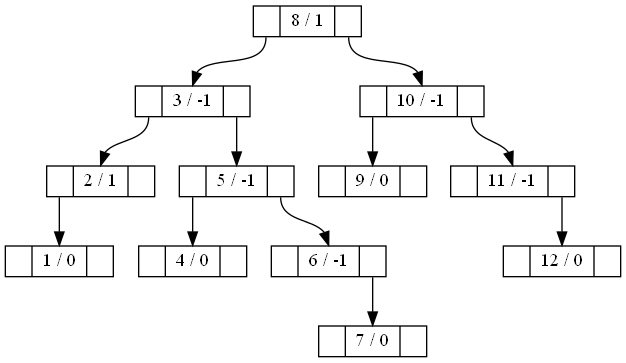
Case 3.2a-:



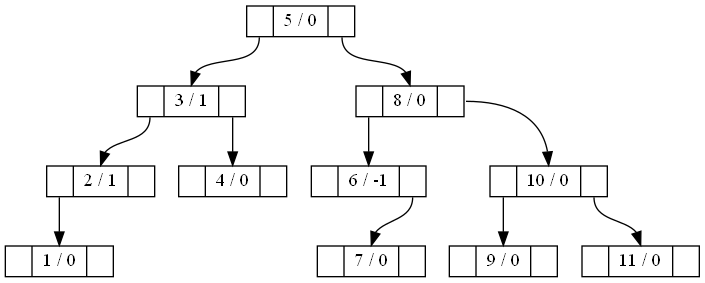
Delete 1:



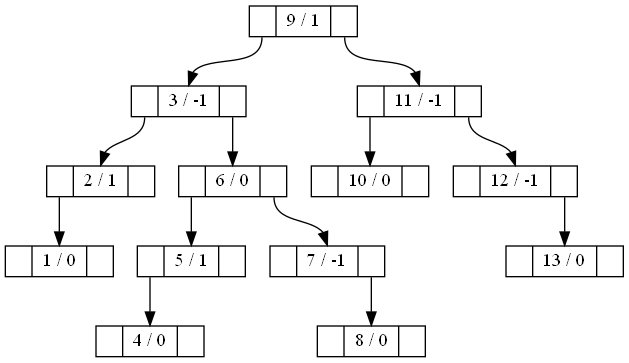
Case 3.2b+:



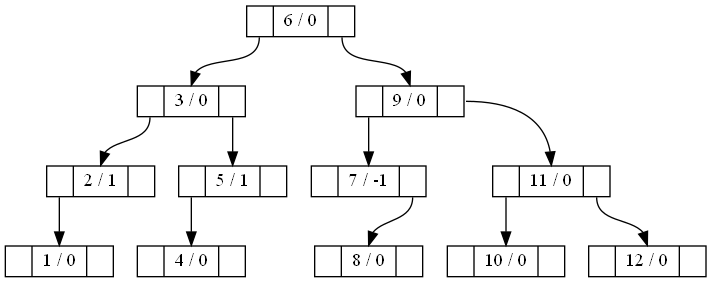
Delete 12:



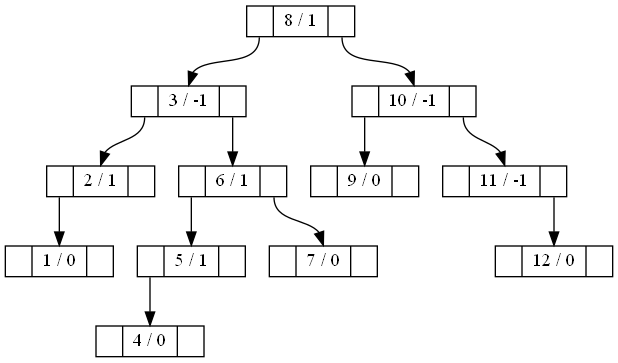
Case 3.2b0:



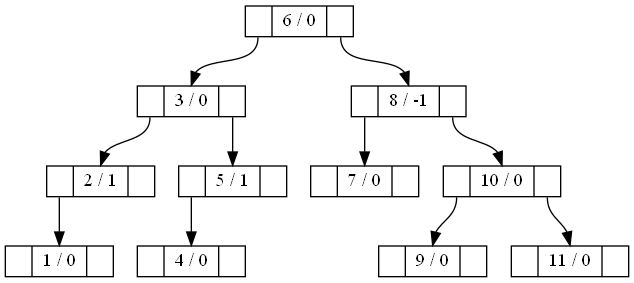
Delete 13:



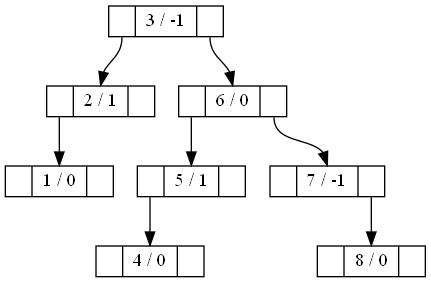
Case 3.2b-:



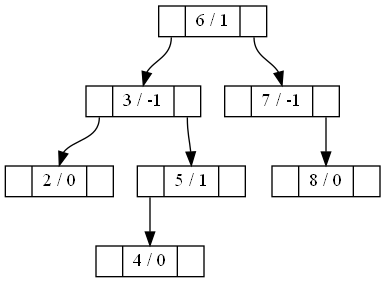
Delete 12:



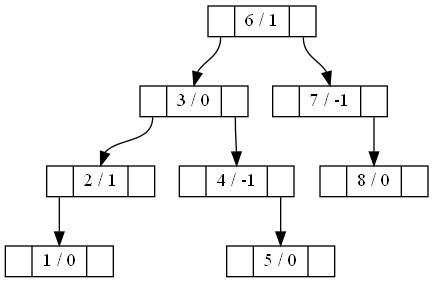
Case 3.3a:



Delete 1:



Case 3.3b:



Delete 8:

