Assignment 2

Before we move on to the questions, I want to describe the tree structure as well as the node structure that we have used for this assignment.

I am considering a dummy head node whose right pointer point to the root of the entire tree and whose left pointer is NULL. This is done so that we can handle the cases where the root of the tree also rotates.

1. [30 points] AVL_Insert(int k) – Inserts a node with key=k. If the element k already exists throw an exception.

   To insert a new element maintaining the balance factor we have to follow the following steps:
   - First, we have to find the balance point in the tree (the point where there can be an imbalance after inserting the element).
   - After finding the point of imbalance and inserting the new added node we are then finding if imbalance has occurred at the point of imbalance.
   - If the imbalance has occurred then we try to rectify it by rotating the subtree where imbalance has occurred.

   Let's explain each step-in details now –

   <u>Finding the balance point:</u>

   While finding whether the node exists in the tree or not we also find the balancing point simultaneously. A node can have imbalance in to only if it's balance factor before insertion is non-zero. If the balance factor of a not before insertion is 0 it means that the node was already balanced before the new node was inserted in the tree. So, inserting a new node will increase it's balance factor to 1 or -1 depending on which subtree of that node it is inserted in. If it is inserted in the left subtree of the node then it will become 1 and if it is inserted in the right subtree of the node then it will become -1.

   Hence, from the above we can safely conclude that if the balance factor of a node is 0 then we cannot consider it as the balancing point. Using this logic we find the node of the tree which my require rotation after the new node is inserted.

   <u>Finding imbalance:</u>

   In order to check whether the balancing point will require any rotation for the insertion of the new node we first need to update the balance factors of all the nodes that lie in the path between the balance point and the newly inserted node. We do not need to update the balance factor of the nodes above the balancing point because their balance factors does not change as after rotation the subtree under the balance point become balanced. The update of the balance factors happens as follows:

   - If the value of the new node (x) is less than the node under consideration (p) then make the balance factor of p = 1 as x is inserted to the left subtree of p.
   - Else if the value of x is greater than p then set balance factor of p = 1 as x is inserted to the right of p.

Any imbalance can occur in the tree if the newly inserted node (x) lies to the same side where there was a partial imbalance. As, we already know that only those nodes where there was a partial imbalance i.e., height of either left subtree or right subtree is 1 more (balance factor = 1 or -1) can be the balance point so if the balance factor of the balance point was already 1 (left partial imbalance) and the x was inserted to the left then there is a complete left imbalance and we need a rotation to fix it. The cases are listed below: (Balance point = s, new node inserted = x)

- Balance factor of s = 1 and value of x < value of s means complete left imbalance has occurred and balancing is required.
- Balance factor of s = -1 and value of x > value of s means complete right imbalance has occurred and balancing is required.
- Balance factor of s = 1 and value of x > value of s means the tree has become balanced so change balance factor of s to 0.
- Balance factor of s = -1 and value of x < value of s means that tree has become balanced so change the balance factor of s to 0.
- Balance factor of s = 0 then change the balance factor of s to 1 or -1 according to the subtree in which x was added.

Balancing / Rotation:

For the final step we need to rotate and balance the tree around the balance point. Before balancing we need to check if we have to do double rotation or single rotation. The conditions for double rotation and single rotation are as follow:

- If the balance factor of the balance point (s) and the parent of the subtree in which the new node is inserted (r) is same then we need to do a single rotation (same sign of the balance factor implies that the new node is added to the same side of balance point and its child).
- If the balance factor of s and r is different then we need to do double rotation (different sign means that the new node is added to different side of the balance point and its child).

After rotation the balance factor of 2 nodes will change, **the balance point (s)** and **the root of the subtree in which the new node is inserted (r)**.

- For single rotation both will become 0 as the tree will be balanced.
- For double rotation following change will occur (a = 1 if the new node is inserted to the left of s and -1 if it is inserted to the right of s).

$$(\mathtt{B(S)},\mathtt{B(R)}) \leftarrow \begin{cases} (-a,0), & \text{if } \mathtt{B(P)} = \ \ a; \\ (\ \ 0,0), & \text{if } \mathtt{B(P)} = \ \ 0; \\ (\ \ 0,a), & \text{if } \mathtt{B(P)} = -a; \end{cases}$$

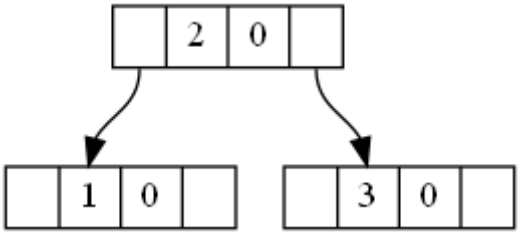
Credits: The Art of Computer Programming Vol 3

Finalizing:

Before termination we also need to reattach all the nodes so that the tree is correct.

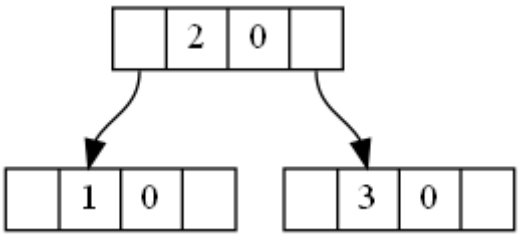
Below I will explain all the cases where rotation can occur:

Sample test case 1: Insert 3, 2, 1



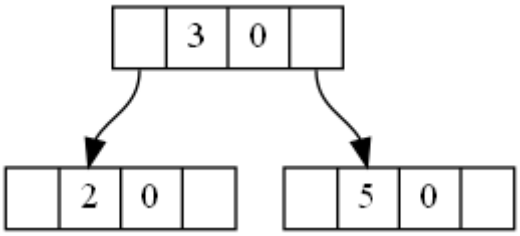
Output 1

Sample test case 2: Insert 1, 2, 3



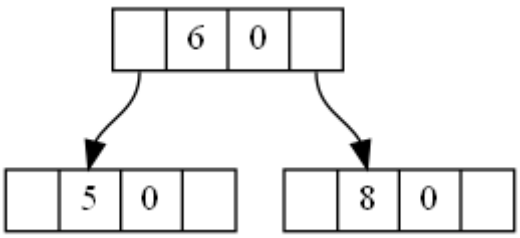
Output 2

Sample test case 3: Insert 5, 2, 3



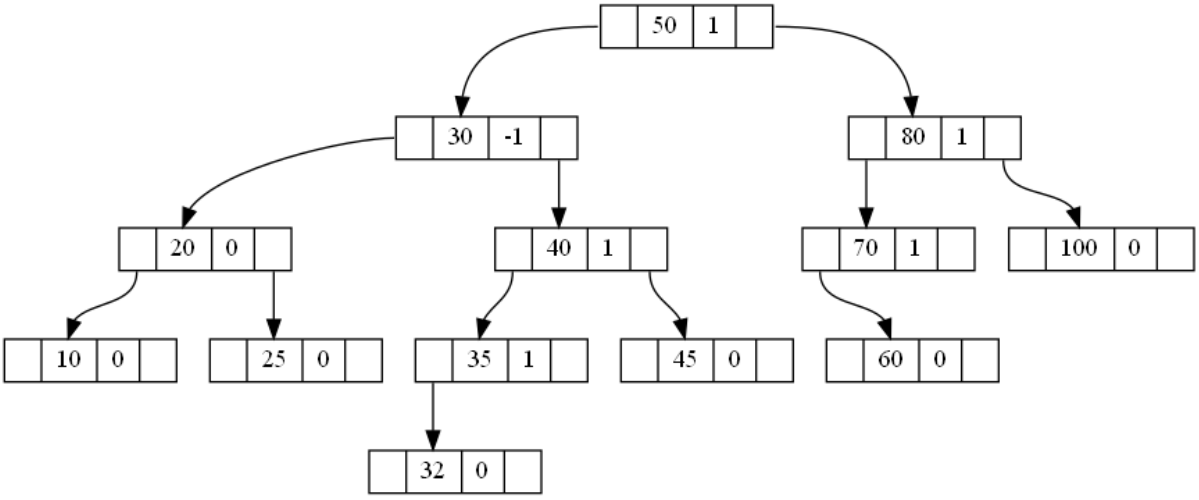
Output 3

Sample test case 4: Insert 5, 8, 6



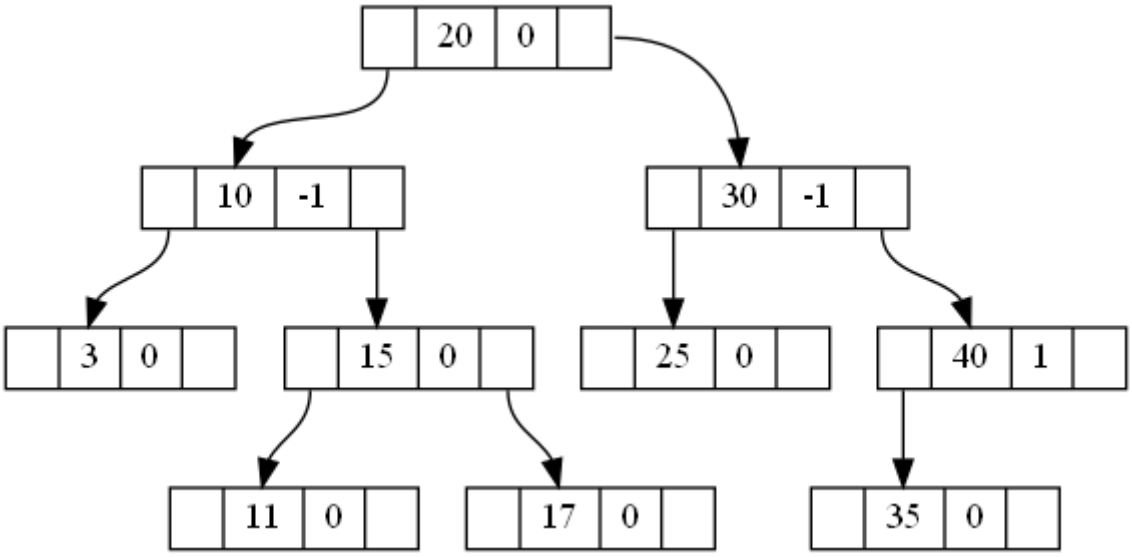
Output 4

Sample test case 5: Insert 50, 30, 80, 20, 40, 70, 100, 10, 25, 35, 45, 60, 32



Output 5

Sample test case 6: Insert 20, 10, 30, 3, 15, 25, 40, 11, 17, 35



Output 6

2. [60 points] AVL_Delete(int k) – Delete the node with key = k. If no node exists throw an exception.

   To delete a node (x) with value equal to the value k we first need to find if that node exists in the tree or not. Also we will need some means of saving the path to the node which has to be deleted if the element exist in the tree because for deletion multiple rotations may be required (O(h) rotations in worst case) unlike insertion of element in the tree. So, deletion will involve the following steps:

   - Searching for the element and saving the path.
   - Balancing all the nodes in the path.

   <u>Searching for the element and saving the path:</u>

   The search is similar to the search we do while inserting but her we also maintain a stack to save the nodes that we are searching to reach the destination. The element to be deleted can be one of the following:

   - Node to be deleted (x) can be the lead node.
   - X can be a node with a single child which is a leaf node.
   - X can be a node which has both leaf and right children.

   Deletion of a node which is leaf node or a node which has a single child which is leaf node is very straight forward, but to delete a node which is the parent of two child nodes we have to apply some logic as mentioned in the steps below:

   - Find the successor of the node to be deleted. While finding the successor also save the path to the successor in the stack that we are keeping to save the path.
   - Swap the value of the successor with the node to be deleted and then delete the successor in the same way that we delete a node that has a single child (which is leaf) or a leaf node.
   - Update the value to be deleted as the successor's value because that is the value that is being deleted.

   <u>Balancing all the nodes in the path:</u>

   After we have successfully deleted the node, now we have to balance all the nodes in the path to the target node. As mentioned before at the start that we are also keeping a dummy node so we iterate till we reach the dummy node. Before deletion we set a variable "a" which will keep track of the side from which the node is deleted.

   - If the value of the deleted node is greater than or equal to the value of a node in the path then we set a = -1.
   - Else if the value of the deleted node is less than the value of the node in the path, we set a = 1.

Using “a” we get to know which side the node is deleted from, and like insertion, there are several cases for which imbalance may occur at a node which lie in the path of the deleted node:

- Balance factor of the node = a; means that there was an imbalance in the node previously and a particular node is deleted from the same side making the node balanced. So, we will just make balance factor of the node = 0 and continue checking the rest of the nodes.
- Balance factor of the node = 0; means that the node was balanced before the deletion happened now after deletion, we need to make a partial imbalance happen in the opposite side of the node as the height of the subtree which is sibling of the subtree from which deletion happened is more. So, we make balance factor of the node as inverse of a (-a).
- Balance factor of the node = inverse of a (-a); means that there was already an imbalance in the opposite side and by deleting an element from the current subtree we make the node more imbalanced thereby needing a rotation. Hence, this case will need a rotation around the node.

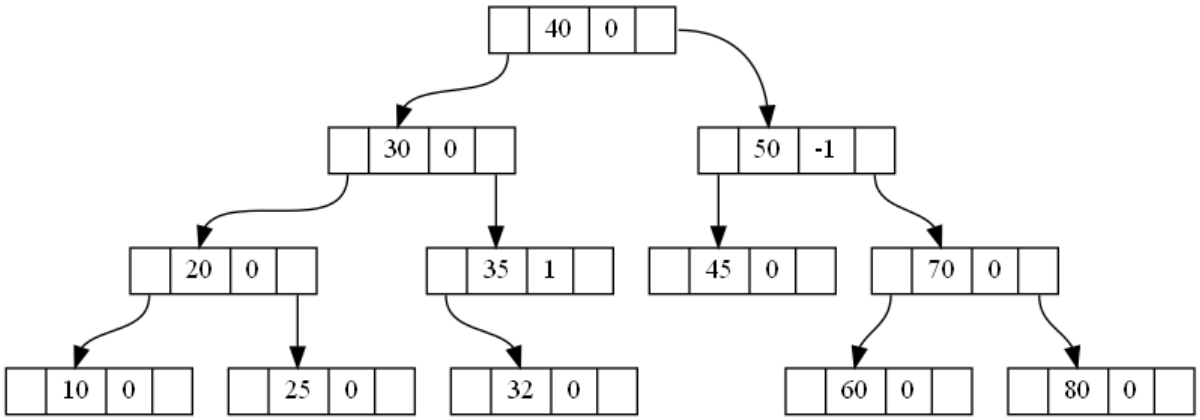
Like insertion here in order to recognise rotation there can be three cases:

- a^{th} subtree of the node requiring balancing has a root with balance factor of -a; needs double rotation.
- a^{th} subtree of the node requiring balancing has a root with balance factor of a; needs a single rotation.
- a^{th} subtree of the node requiring balancing is already balance; need a single rotation.

After the rotation is done successfully there can be a case where the height doesn’t change i.e., the height of the root of the subtree before rotation is as the height after rotation is done. In this case we simply terminate the algorithm as no further checking of other nodes is required.

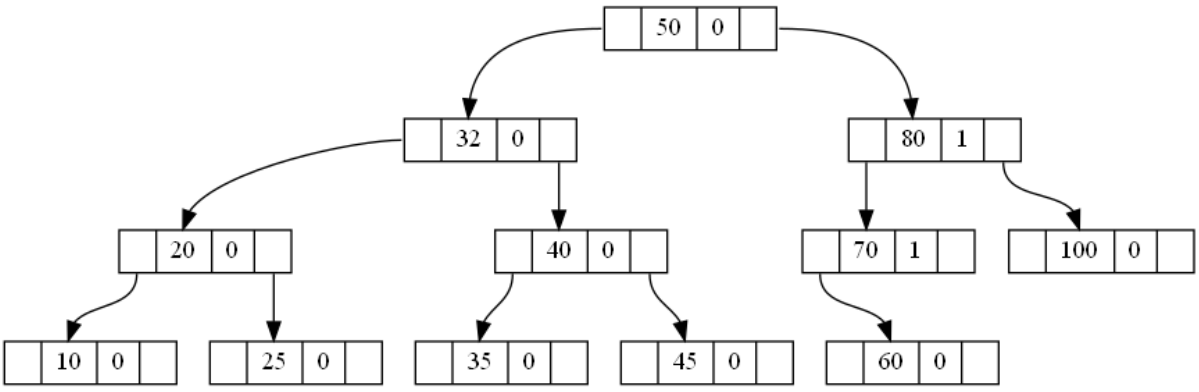
Which rotation we updated the balance factors in the same way that we did for insertion except the case where the a^{th} subtree of the node which has to be adjusted is already balanced. This case can never occur which inserting an element in the tree hence we have not taken this case under consideration which insertion. In this case we keep all the balance factor same except the balance factor of the root of a^{th} subtree which will become = a as after rotation there will be an imbalance created.

Sample test case 1: Delete 100 from Output 5



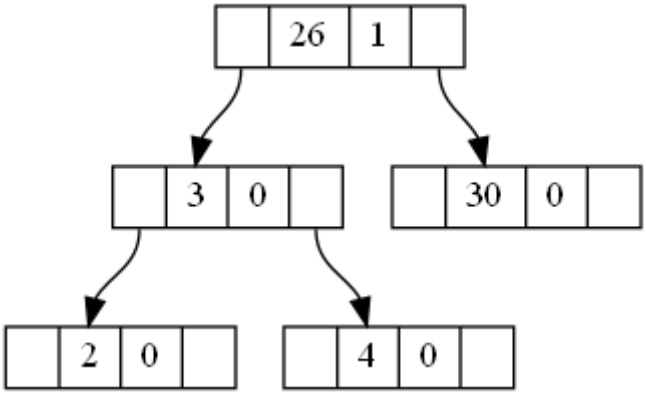
Output 7

Sample test case 2: Delete 30 from Output 5

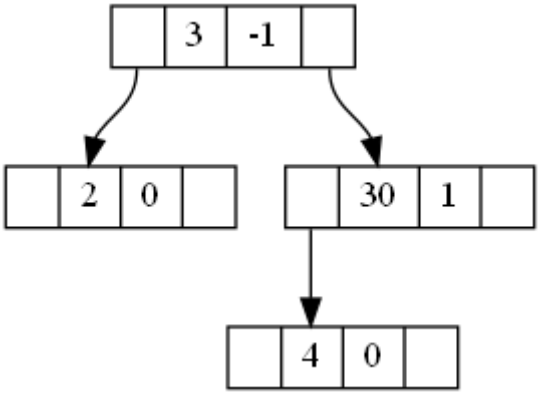


Output 8

Sample test case 3: Insert 26, 3, 30, 2, 4 then Delete 26

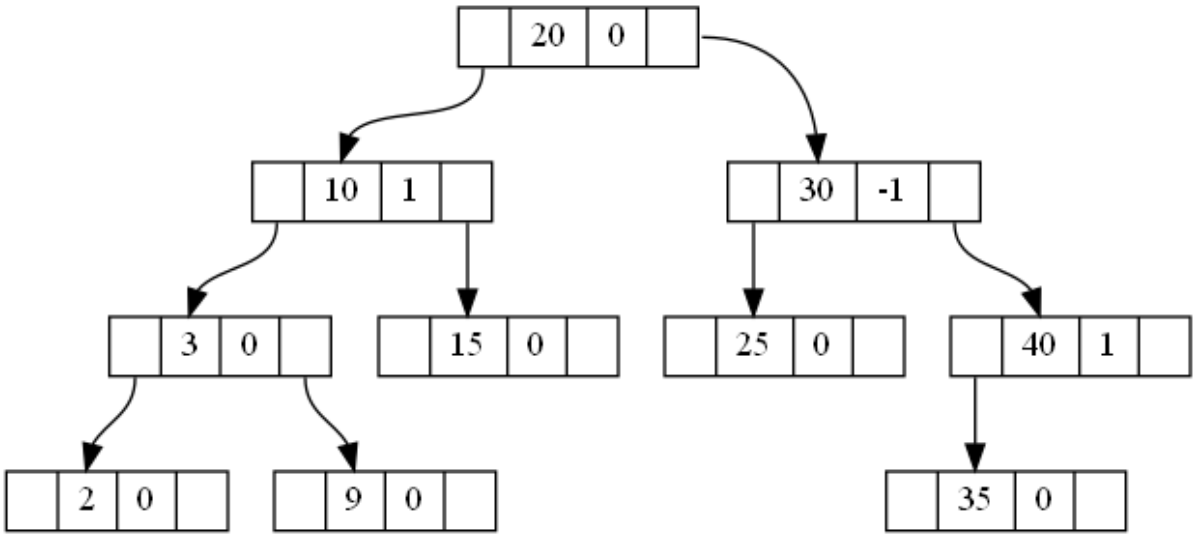


Before delete

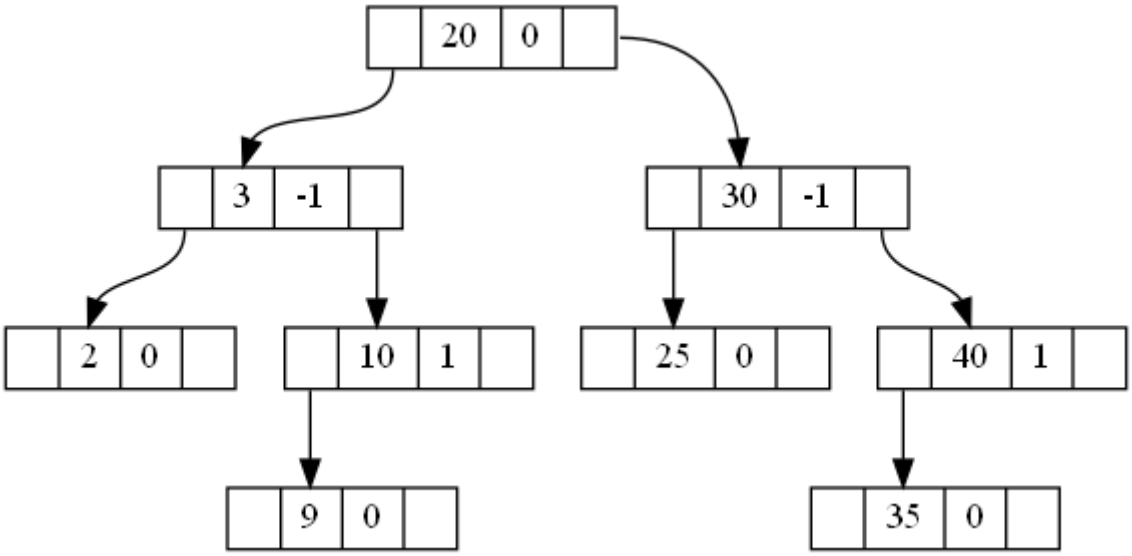


After delete

Sample test case 4: Insert 20, 10, 30, 3, 15, 25, 40, 2, 9, 35 then delete 15

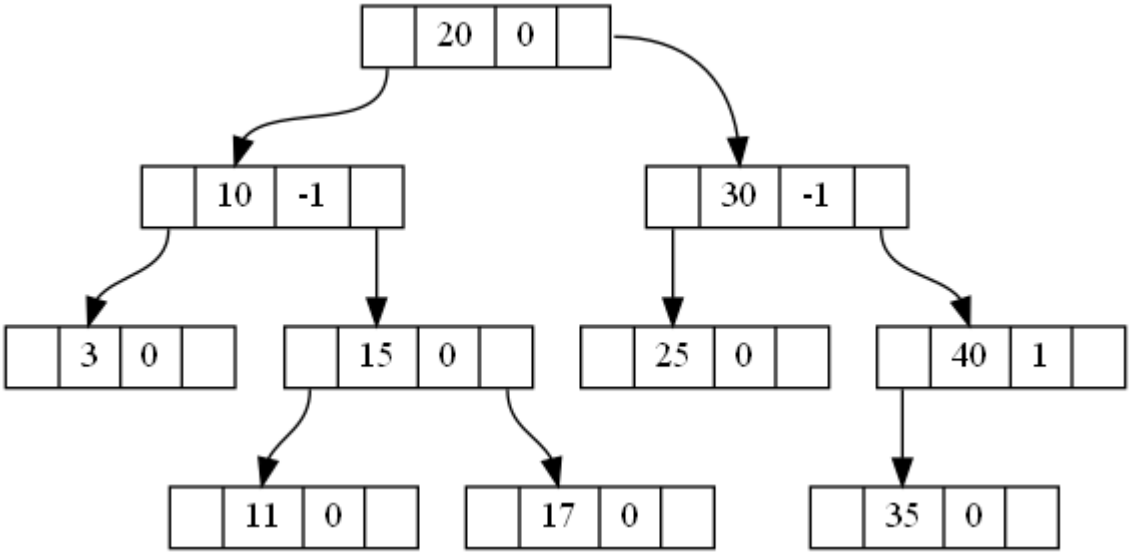


Before delete

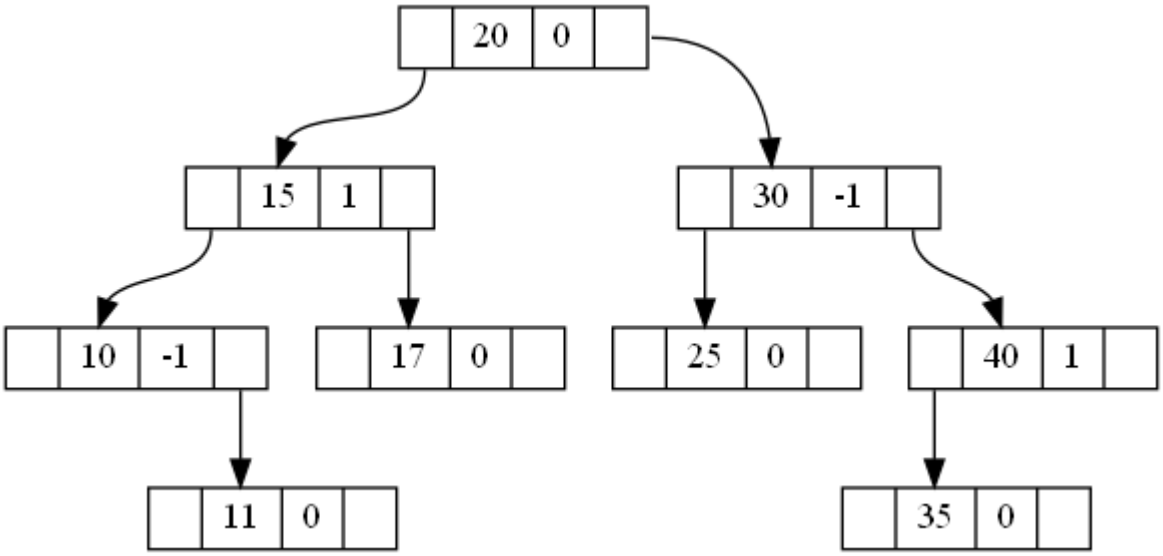


After delete

Sample test case 5: Insert 20, 10, 30, 3, 15, 25, 40, 11, 17, 35 then delete 3

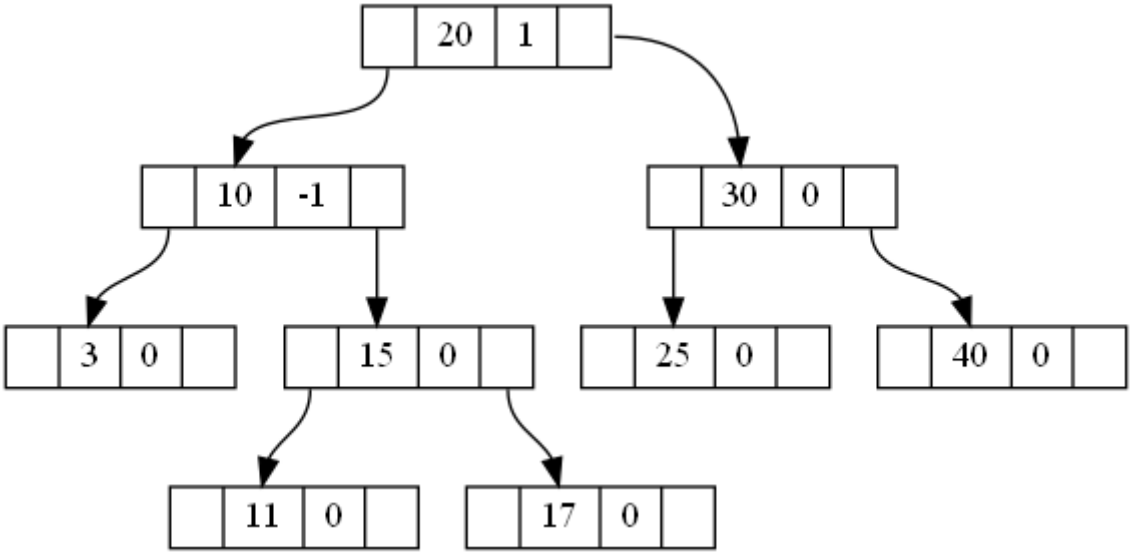


Before delete

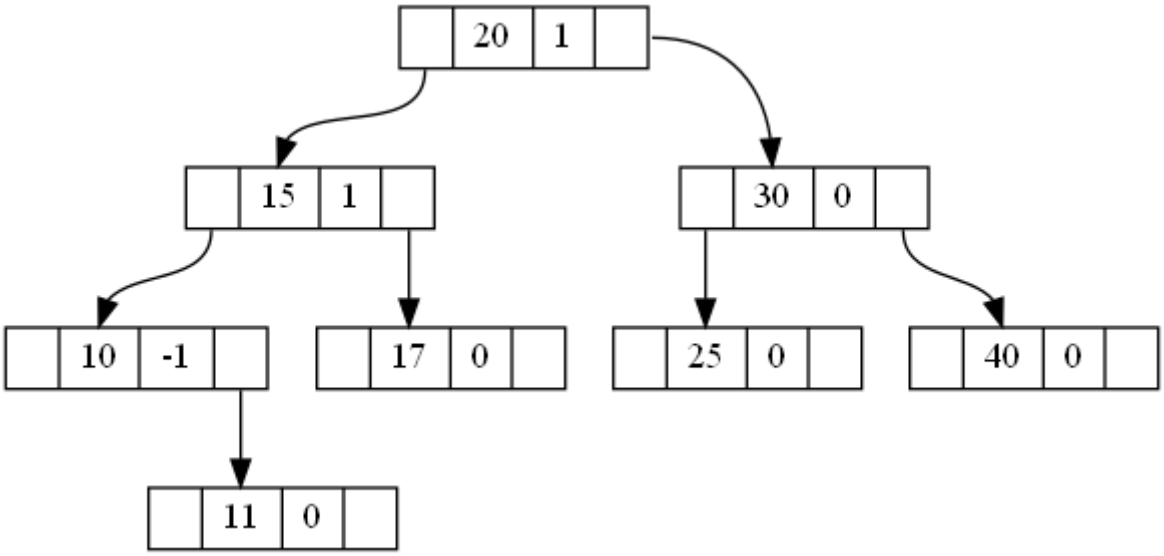


After delete

Sample test case 6: Insert 20, 10, 30, 3, 15, 25, 40, 11, 17 then delete 3.



Before delete



After delete

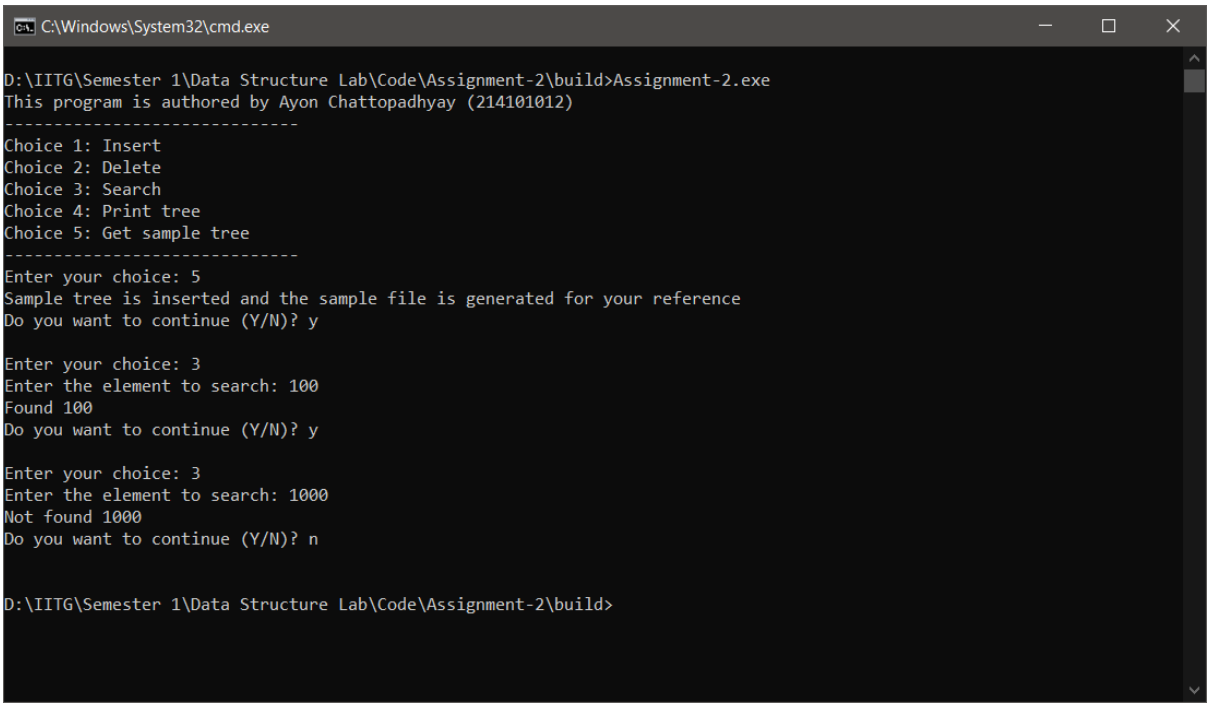
3. [5 points] AVL_serach(int k) – Finds the key k and returns true (false) if k is present (not present).

Search is implemented in the same way as we search an element in a binary search tree as follows:

- If the value of the node is less than the value to be searched, check its right subtree.
- Else if the value of the node is greater than the value to be searched, check its left subtree.
- Else if the value is equal then return true.

This check is continued till we reach a leaf. If still we don't find it then we return false.

Sample test case 1: Search 100 and 1000 in Output 5



Output 9

4. [10 points] AVL_Print(const char *filename) – Prints the tree structure with key and bf.

   This is basically inorder traversal of the BST with some manipulations so that we get a proper .png file using graphviz.