

# DOCUMENTATION FOR ASSIGNMENT II

## Description of the data structure

### AVL Insert(k)

Logic:

Test Cases:

### AVL Delete(k)

Logic:

Test Cases:

### AVL Search(k)

Logic:

Test Cases:

### AVL Print(filename)

Logic:

Test Cases:

### Makefile and valgrind

## Description of the data structure

An AVL tree is a self-balancing binary tree, where the difference of heights between left subtree (LST) and right subtree (RST) of a node is never more than 1. This difference of heights is also called the **balance factor**.

$$\text{balance factor}(\text{node}) = \text{height}(\text{LST of node}) - \text{height}(\text{RST of node})$$

In this program, the balance factor is not calculated by using heights, but is calculated and updated with each modification of the tree.

The tree is defined as a class **AVL\_Tree**. It has a friend class **AVL\_Node**, which defines the component nodes of the AVL tree defined by AVL\_Tree. It is also the friend class of AVL\_Tree.

Each node of the tree has four components:

- **key:** The value associated to the node (type: int)
- **bf:** The balance factor of the node (type: int)
- **LChild** and **RChild:** The pointers to left and right children of the node respectively (type: AVL\_Node\*)

There exists a dummy node **head**, and the root node of the entire tree is the right child of it.

In the main(), a user interface is provided where inputs can be given for the functions, which are described in the document. A sample tree is also provided.

# AVL\_Insert(k)

## Logic:

Inserting an element in the AVL tree is a three step process:

### **1. Finding insertion location & point of imbalance:**

Here, the location where the new element is to be inserted, is found. But parallel to this, the point of imbalance is also found. Point of imbalance will be the last node in the traversal, where the balance factor is non zero (or root). We will initialize it at root.

To find the location of insertion, we compare the values of  $k$  and key of the current node. If at any point  $k == \text{key}(\text{current node})$ , we abort, as element already exists.

If  $k$  is less than key of the current node, current node moves to its left node. Else, it moves to its right node. This comparison will keep happening unless current node is null. Then, we will insert the new node to current node's parent's appropriate child. Eg. if  $k < \text{key of current node's parent}$ , it will be inserted as its left child.

In the meanwhile, before every shift of the current node to its child, we check if the current node's bf is non-zero or not. If it is, it becomes the point of imbalance.

### **2. Adjusting balance factors:**

For adjusting balance factors, we start from the relevant child of the point of imbalance (poi). Eg. if the new node is inserted to the poi's left subtree, we start from the left child of poi.

Then we compare the values of  $k$  and the key of poi. If  $k < \text{key}(\text{poi})$ ,  $\text{bf}(\text{poi}) = 1$ . Else,  $\text{bf}(\text{poi}) = -1$ . We also keep traversing one-by-one in the direction of the inserted node, and keep repeating the process for each node encountered, until we reach the newly inserted node.

### **3. Rotation:**

For this, we need a variable alpha ( $\alpha$ ). It is equal to -1 if the new node is in its right subtree, and +1 otherwise. We also create a pointer to the point of imbalance ( $s$ ), relevant child of poi ( $r$ ), and also to the newly inserted node ( $p$ ).

If bf of s is 0, it will become equal to a. This is because due to insertion, the imbalance is introduced.

Else, if bf of s is -a, it becomes 0, because due to insertion, the tree becomes more balanced.

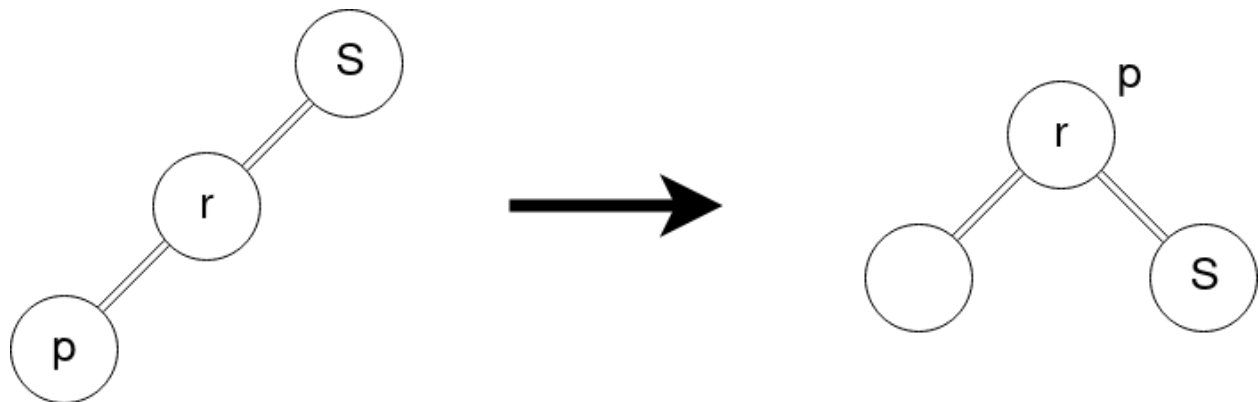
The real transformation happens when the bf of s is a. This means that the insertion is done on the side of the existing imbalance, making it even more imbalanced. To correct this, we first check which kind of imbalance is there.

If bf of r (child of s) is a, this means that Right-Right (RR) or Left-Left (LL) type imbalance is there. This implies there will be a **single rotation**.

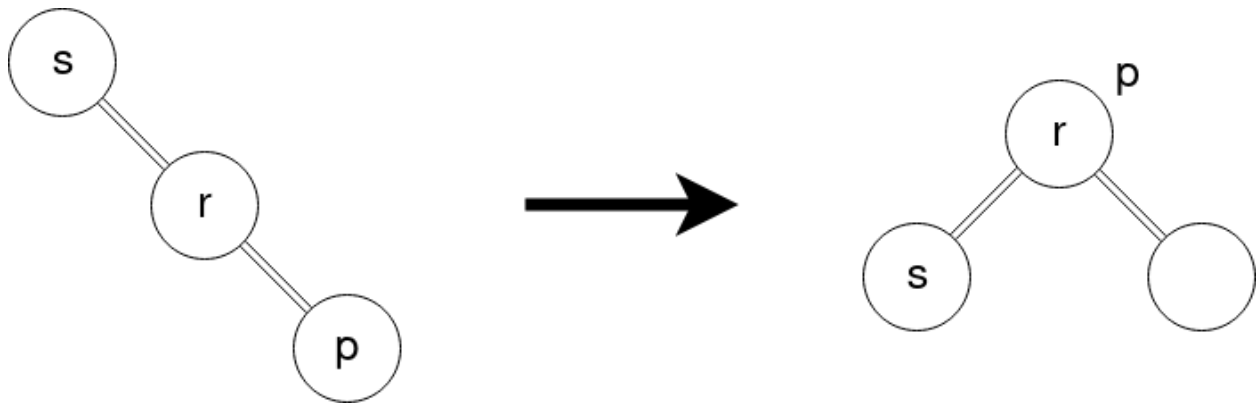
If bf of r is -a, this means that either RL or LR type imbalance is there. This implies that there will be a **double rotation**.

### Single rotation

LL type imbalance is there when a is equal to 1. This is because both bf of s is 1, and r is 1. This makes the imbalance of both s and r towards the left. To remove this imbalance, we do a right rotation, which is as follows:



RR type imbalance is there when a is equal to -1. This is because both bf of s is -1, and r is -1. This makes the imbalance of both s and r towards the right. To remove this imbalance, we do a left rotation, which is as follows:

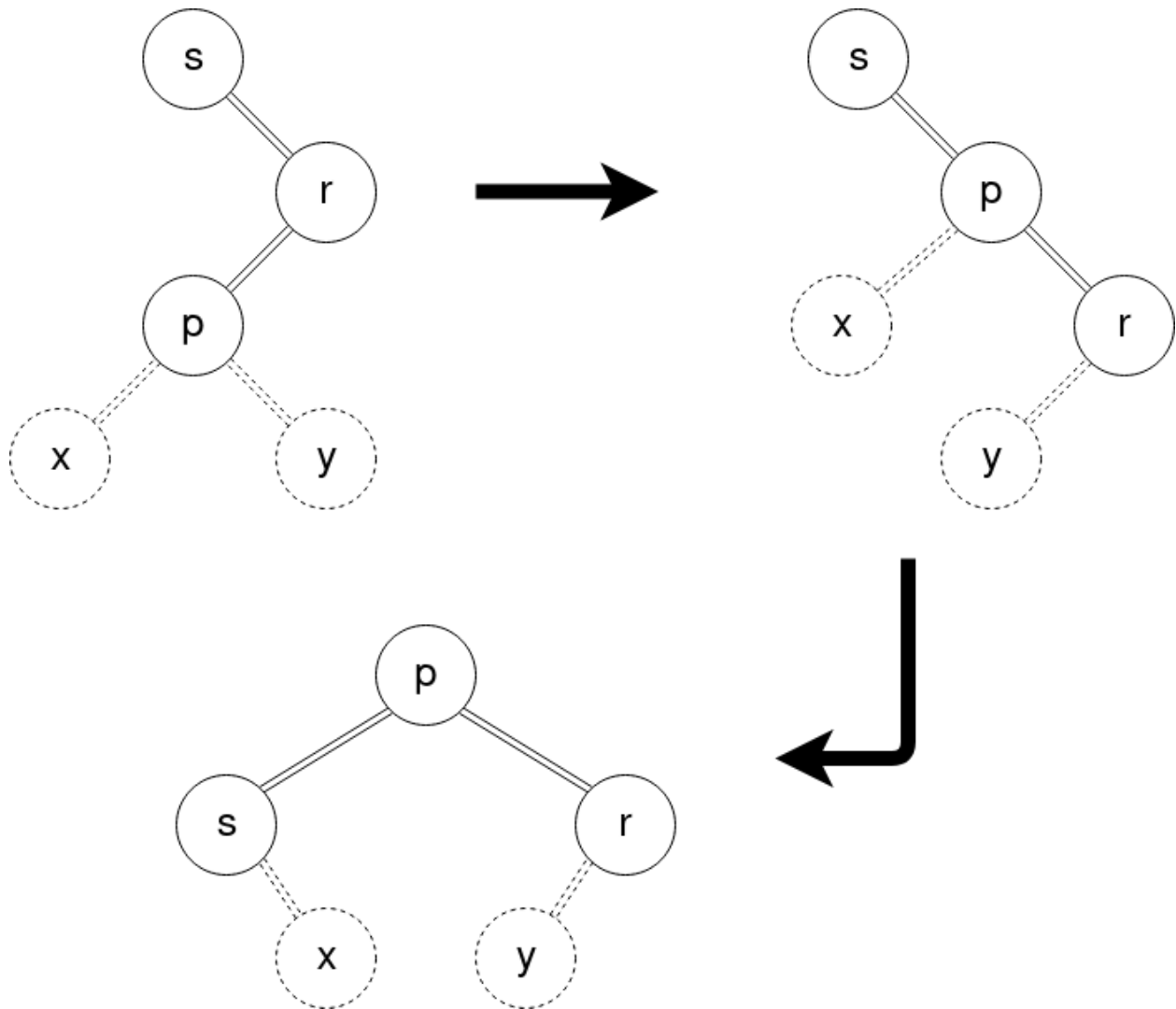


In single rotation, bf of s and r are then changed to 0, as they both are balanced now.

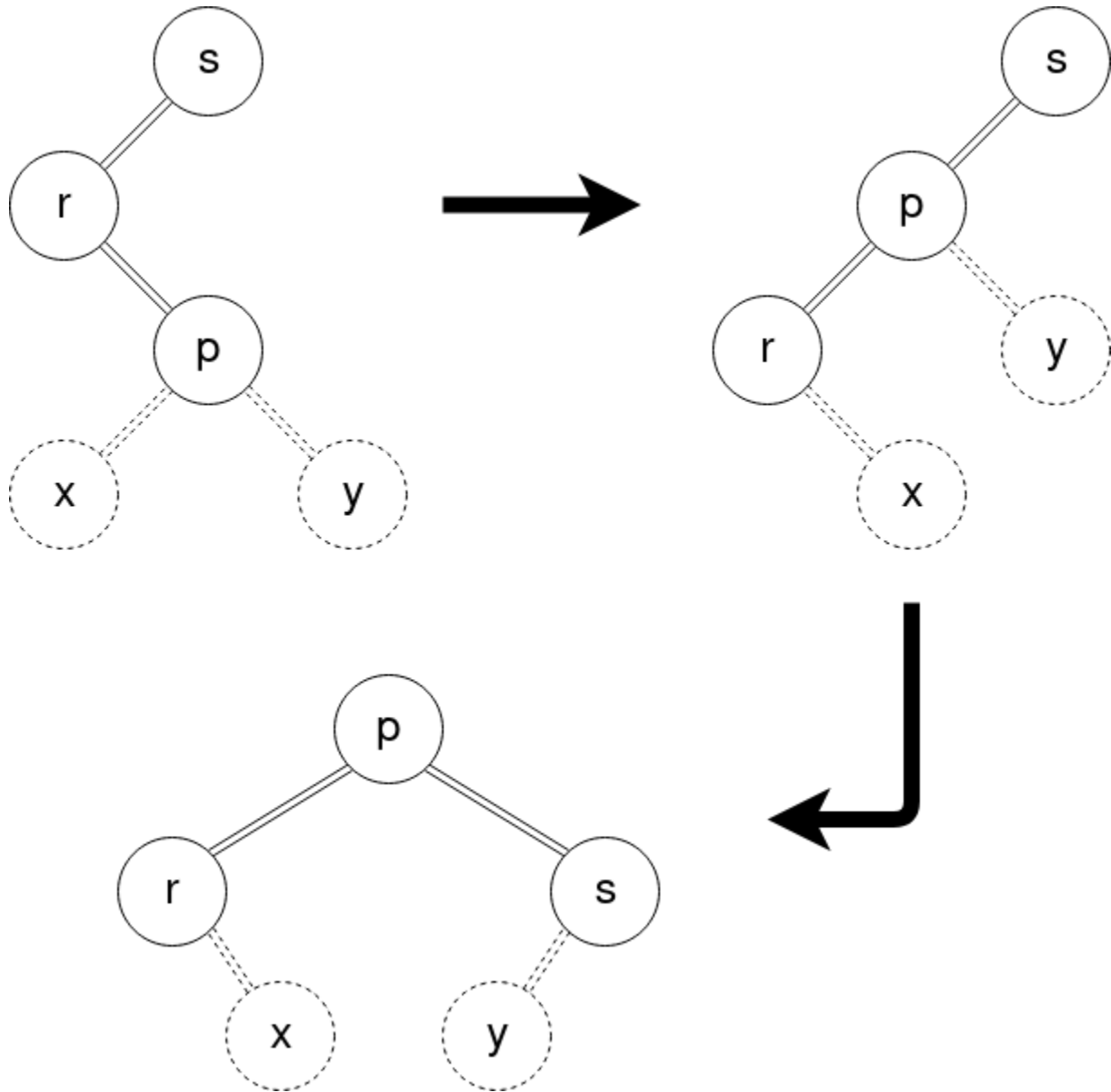
For single rotation, the function **singleRotation()** is called, with a, and addresses of p, r, s as its arguments.

## Double Rotation

RL type imbalance is there when a is equal to 1. This is because bf of s is -1, and r is 1. This makes the imbalance of s towards right and r towards the left. To remove this imbalance, we do a right rotation, followed by a left rotation. They are both done together, which is as follows:



LR type imbalance is there when a is equal to -1. This is because bf of s is 1, and r is -1. This makes the imbalance of s towards left and r towards the right. To remove this imbalance, we do a left rotation, followed by a right rotation. They are both done together, which is as follows:



After the double rotation, we adjust the bf of r and s as following:

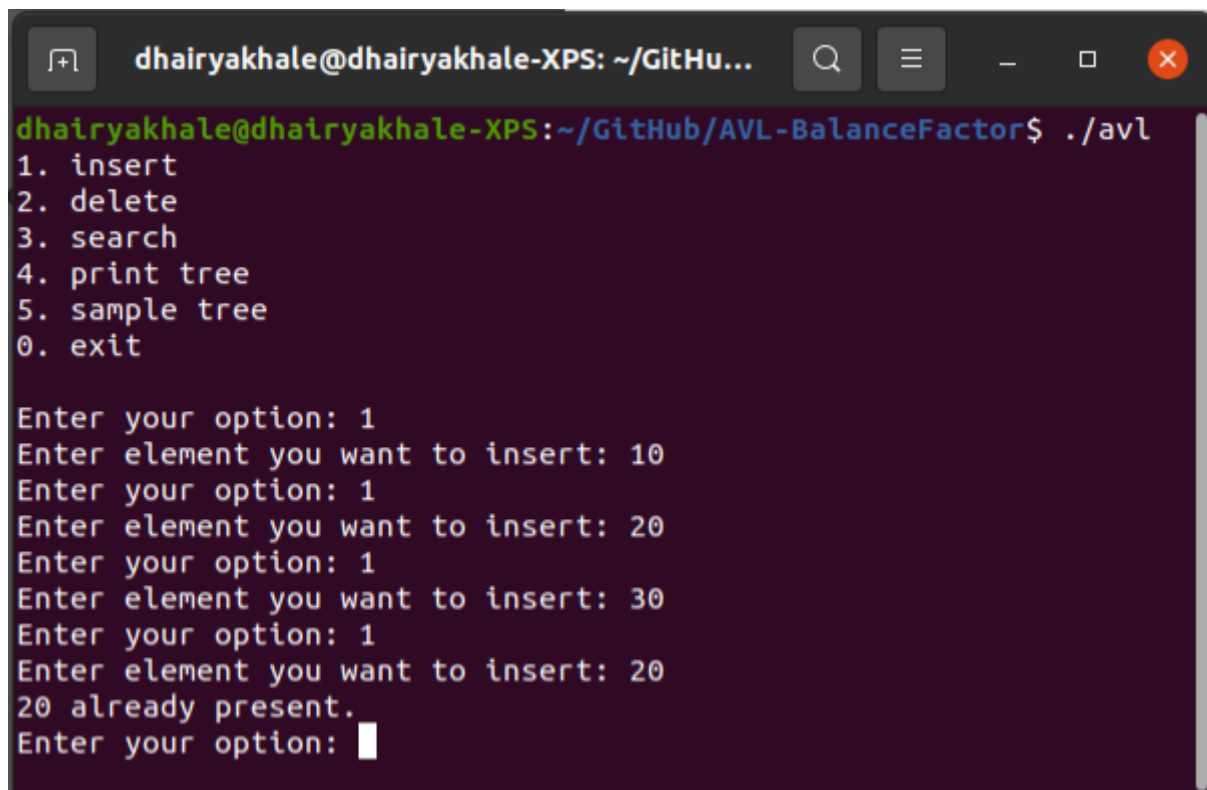
$$(B(\dot{S}), B(R)) \leftarrow \begin{cases} (-a, 0), & \text{if } B(P) = a; \\ (0, 0), & \text{if } B(P) = 0; \\ (0, a), & \text{if } B(P) = -a; \end{cases}$$

And in the end, we set bf of p as 0.

In the end of either rotation, the parent of the point of imbalance's child becomes p (since s has shifted from its original location, and p is the new root of resultant subtree).

For double rotation, the function **doubleRotation()** is called, with a, and addresses of p, r, s as its arguments.

### Test Cases:

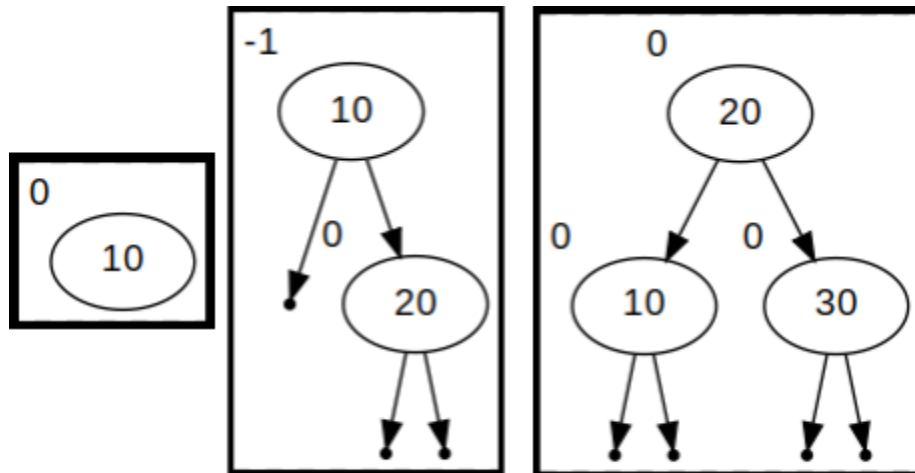


```
dhairyakhale@dhairyakhale-XPS: ~/GitHu...
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ ./avl
1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

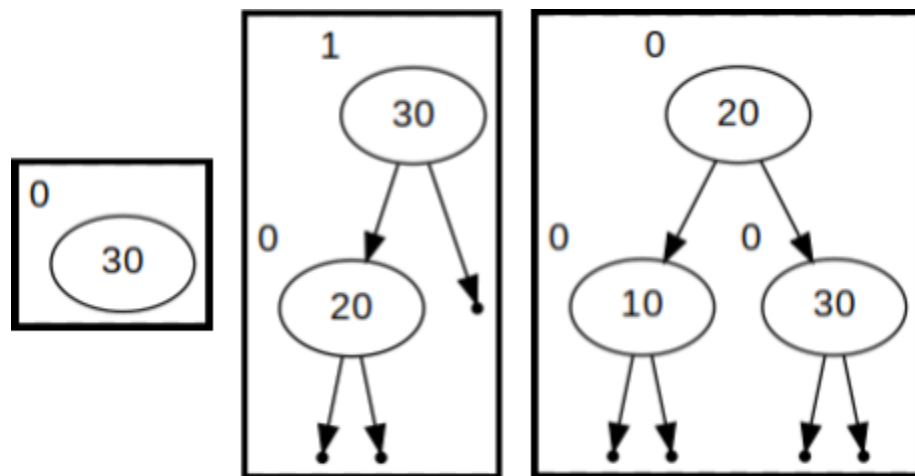
Enter your option: 1
Enter element you want to insert: 10
Enter your option: 1
Enter element you want to insert: 20
Enter your option: 1
Enter element you want to insert: 30
Enter your option: 1
Enter element you want to insert: 20
20 already present.
Enter your option: █
```

Command for insertion

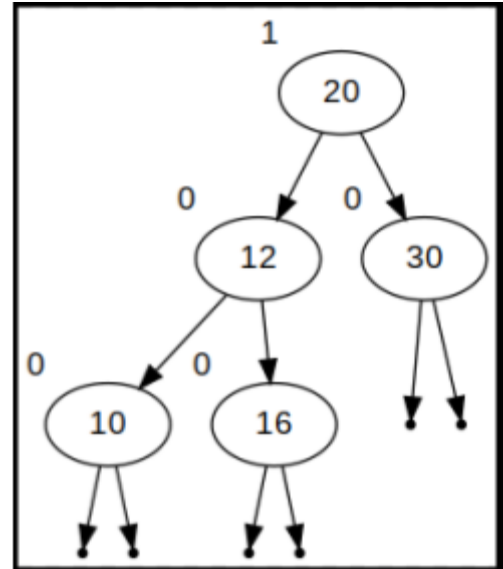
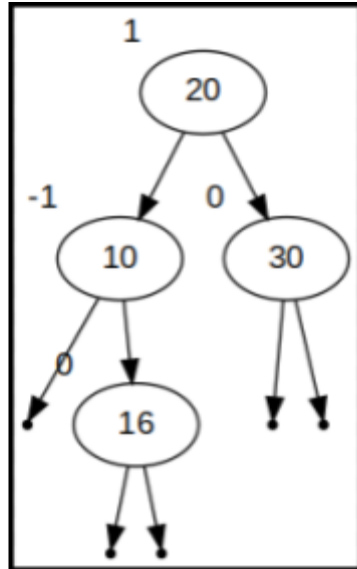
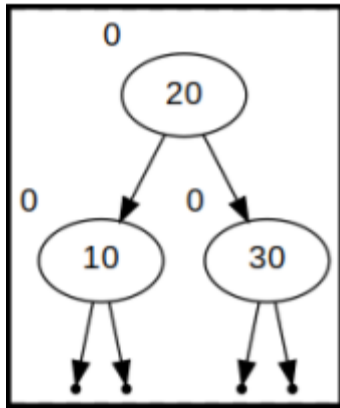




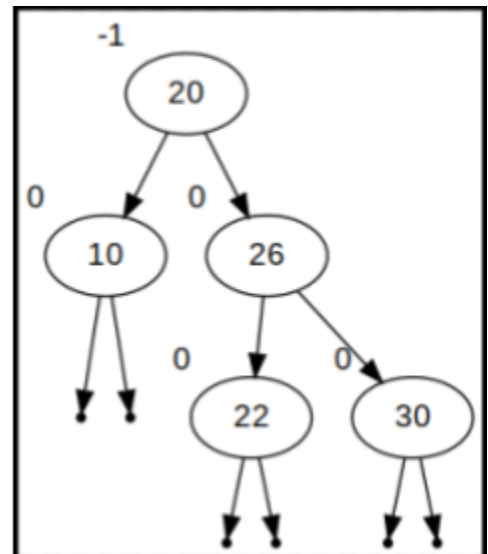
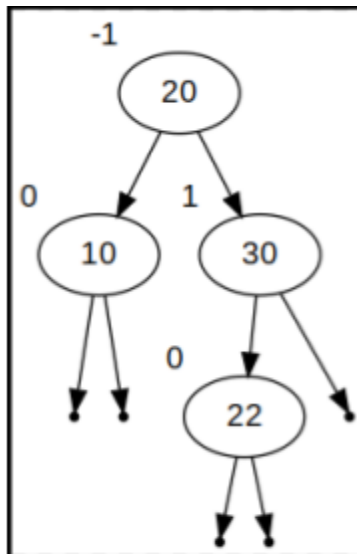
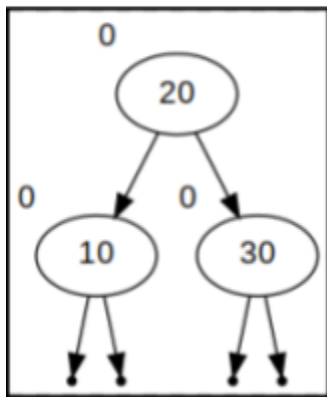
RR imbalance, Left rotation



LL imbalance, Right rotation



RL imbalance, RL rotation



LR imbalance, LR rotation

# AVL\_Delete(k)

## Logic:

Deleting an element from an AVL tree is also a three step process.

### **1. Finding location for deletion:**

Here, the node which is to be deleted, is found. But along with this, we also keep pushing the parents encountered in the traversal into a stack.

To find the node for deletion, we compare the values of k and key of the current node.

We start from the root. If k is less than key of the current node, current node moves to its left node. Else, it moves to its right node. This comparison will keep happening unless key of the current node is equal to k. If at no point k is equal to key of the current node, and current node becomes null, that means the node does not exist in the tree. So we abort.

### **2. Adjustment after deletion:**

When the node for deletion is obtained, we check how many children it has. There are three cases:

#### **1. No children**

When the node for deletion has no children, it is easy to delete, as no new connections have to be formed. So in that case, if it is the right child of their parent, the parent's right child becomes null, and similarly if the node is their parent's left child. Then, the node is deleted.

For this, the function **Del0Child()** is called with the node and its parent passed as arguments.

#### **2. One child**

Here, similarly to before, we see if the node is their parent's left child, or right child. If the node is their parent's left child, the child of node is connected to the parent as their left child. If the node is their parent's right child, the child of node is connected to the parent as their right child. Then, the node is deleted.

Along with this, the bf of the parent becomes equal to the bf of their new child.

For this, the function **Del1Child()** is called with the node and its parent passed as arguments.

### 3. Two children

When two children are there, we go to the inorder successor of the element (leftmost node of right subtree), and copy the value to the node. We also push all the parents of the inorder successor.

After this, we apply the deletion to the inorder successor, according to the number of children it has (0 or 1).

For this, either **Del0Child()** or **Del1Child()** is called, with the node and its parent passed as arguments.

### 3. Rotation

Now, for each element in stack (all the parents of deleted node), we will do the rotation method as specified in **AVL\_Insert()**, but with few changes:

- If bf of parent is 0, the new bf becomes -a, because imbalance is introduced due to deletion. And the imbalance is introduced in the direction opposite to where it was deleted from.
- If bf of parent is a, the new bf becomes 0, because imbalance is removed due to deletion.
- If bf of parent is -a, that means deletion of the node is from the side opposite to imbalance, which creates more imbalance. Because of this, the rotation methods of **AVL\_Insert** are applied, but -a is passed in them, as as opposed to insertion, deletion from place with less imbalance creates more imbalance overall.
- In the special case where bf(r) is 0, a single rotation is applied but bf(s) is not changed, as it might still be imbalanced. Also, bf of r becomes a.

After the rotation, if the resulting tree has bf not equal to zero, we will not cover all parents and return after connecting parent of point of imbalance and p, as the parents will be unchanged due to their children still being not fully balanced.

Else, we will connect the parent of poi and p, and keep going.

For single rotation (except the special case specified above), the function **singleRotation()** is called, with a, and addresses of p, r, s as its arguments.

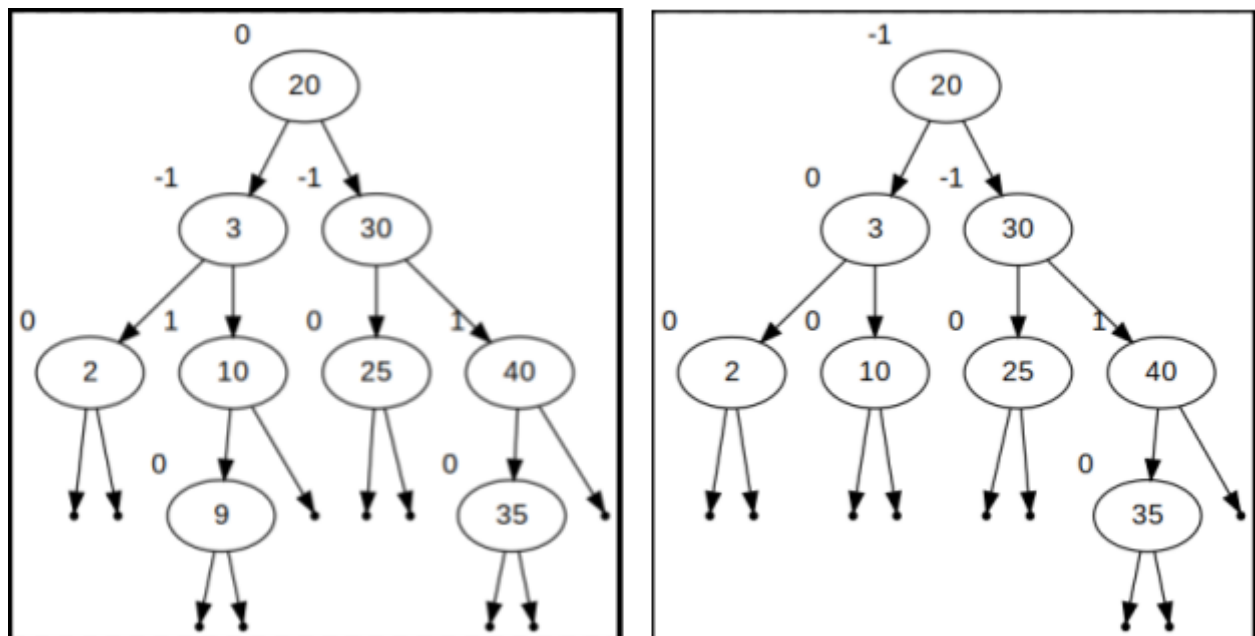
For double rotation, the function **doubleRotation()** is called, with a, and addresses of p, r, s as its arguments.

## Test Cases:

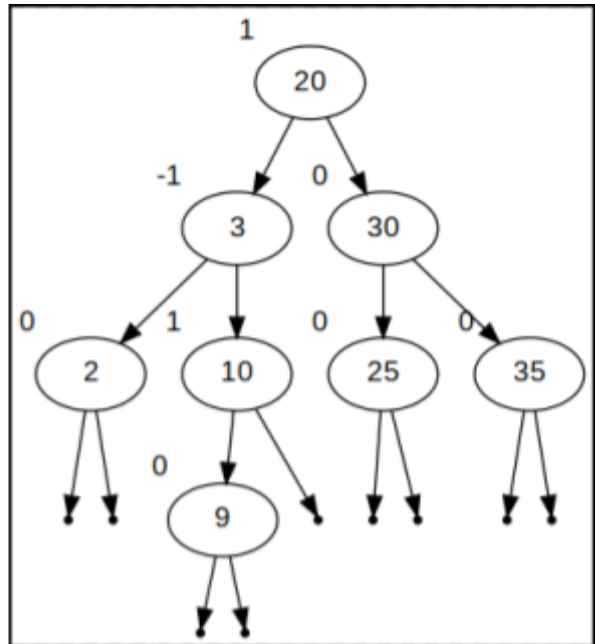
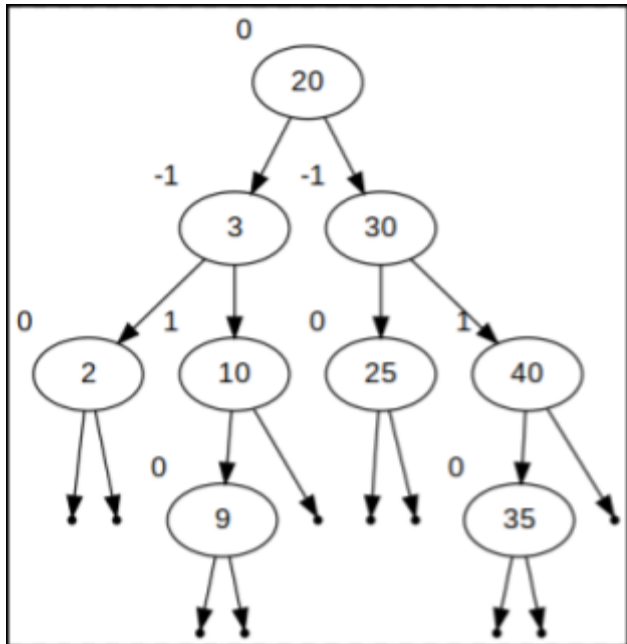
```
dhairyakhale@dhairyakhale-XPS: ~/GitHub/AVL-BalanceFactor
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ ./avl
1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 5
Find file in same directory, and open.
Enter your option: 2
Enter element you want to delete: 35
Enter your option: 2
Enter element you want to delete: 10
Enter your option: 2
Enter element you want to delete: 20
Enter your option: 2
Enter element you want to delete: 420
420 not found for deletion.
Enter your option: █
```

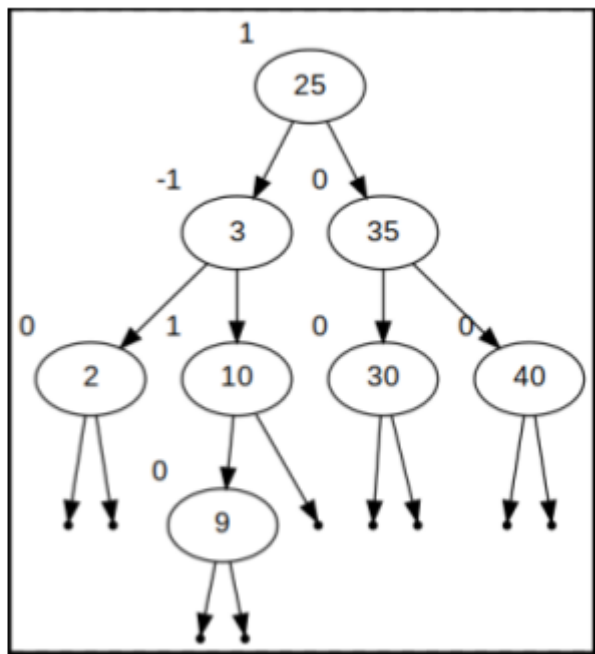
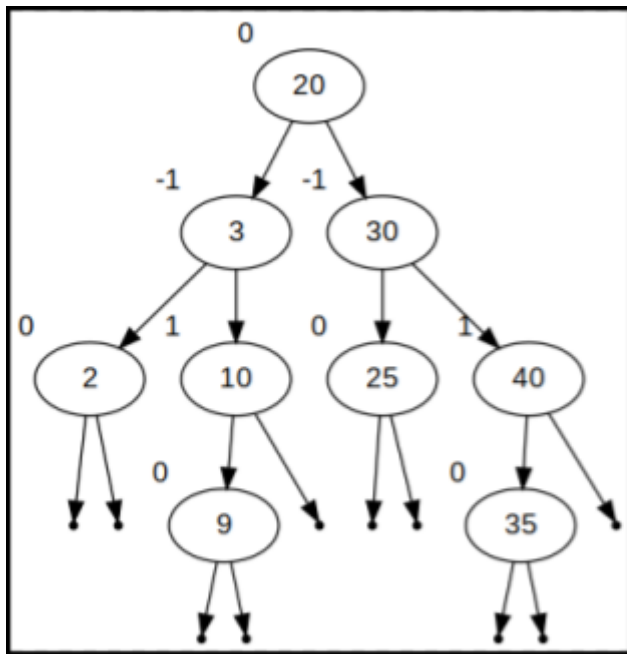
Command for deletion



Deletion of a leaf node (9)



Deletion of a node with one child (40)



Deletion of a node with 2 children (20)

## AVL\_Search(k)

### Logic:

Searching in an AVL tree is exactly the same as searching in a regular Binary Search Tree.

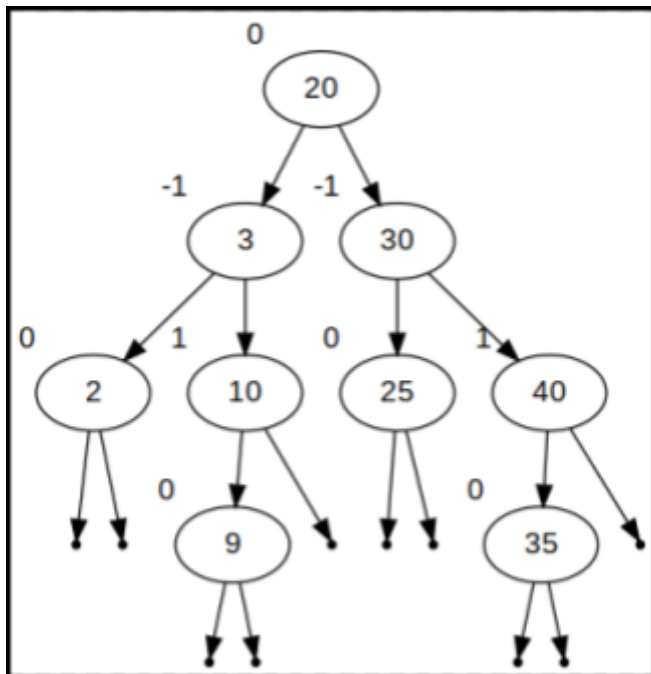
We make a node pointer, and point it to the root node. We then compare  $\text{key}(\text{node})$  and  $k$ .

If  $k < \text{key}(\text{node})$ , node becomes its left child. If  $k > \text{key}(\text{node})$ , node becomes its right child. This process keeps repeating till the node is null.

If  $k == \text{key}(\text{node})$ , that means node is what we were searching for. So we return true, and exit.

If node becomes null, that means the node we were searching for does not exist in the tree. Hence, we return false.

### Test Cases:



Searching in the sample tree

```
dhairyakhal...
Enter your option: 3
Enter element you want to search: 10
10 found!
Enter your option: 3
Enter element you want to search: 25
25 found!
Enter your option: 3
Enter element you want to search: 798
798 not found.
Enter your option: 3
Enter element you want to search: 87
87 not found.
Enter your option: █
```

## AVL\_Print(filename)

### Logic:

Here, we needed the use of graphviz software. So we had to basically make a graphviz compatible file by writing data out of the cpp file.

For this, we used 3 functions:

1. **printTree\_null**
2. **printTree\_main**
3. **AVL\_Print**
4. **inorderLabel**

The first function is just used to write the command which makes the input argument node connect to a dot through an arrow.

The second function is basically creating all the edges.

It starts from root, and creates all the left edges by looping while the left child is not null. It associated to each destination and source node a label and name, both specified in the inorderLabel function. When the null is reached, it calls the printTree\_null() function, where the node is connected to a dot.

Similarly, all the right edges are created by looping while the right child is not null.

The second function also has a static int nullc. This is passed with post-increment to the printTree\_null() function, so that every null node is named uniquely.

The third function is basically a driver function.

It first creates and opens a file in write mode. The name of the file is specified via the argument of the function (filename). It also creates a label associated with each node, which tells the bf of that node.

It then gives some starting commands in the graph file.

If root is null, it does not input anything in the graph file. Else if root has left and right null, it just feeds a single node value into the file. Else, it calls the printTree\_main() function.

It then gives ending commands to the file, and closes the file.

The fourth function associates a unique label and unique name to each non-null node. This helps in printing, and is required for purposes of the graphviz software.



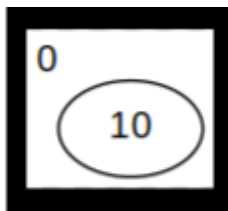
After closing, we open the cmd, convert the .gv file to .svg image, and save the image in the current working directory using the system() function.

### Test Cases:

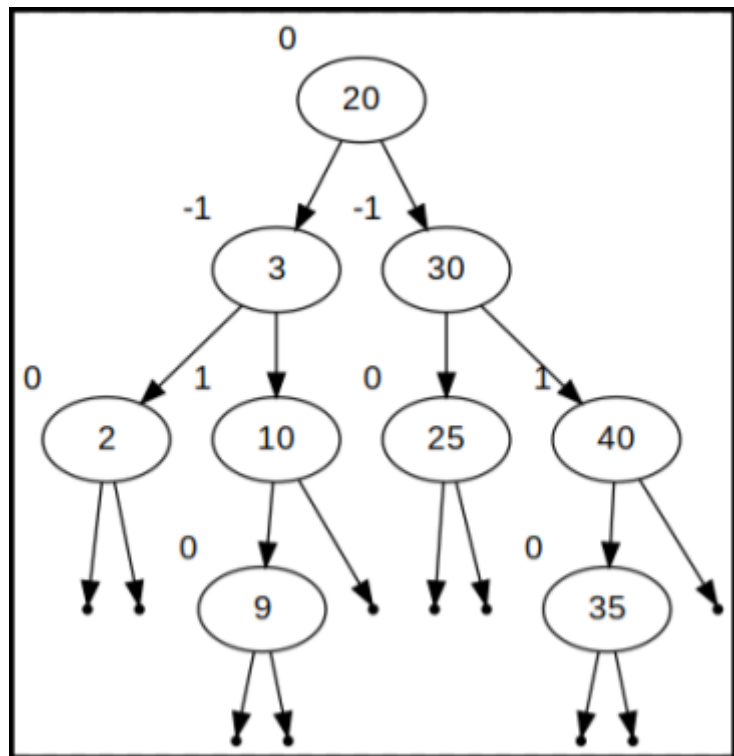
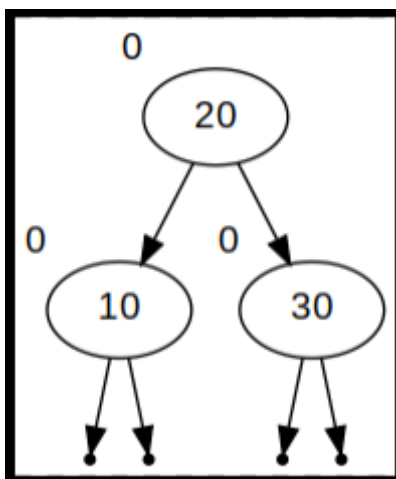
```
dhairyakhale@dhairyakhale-XPS: ~/GitHu...
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ ./avl
1. insert
2. delete
3. search
4. print tree
5. sample tree
6. exit

Enter your option: 4
Enter name of file: myfile
Find file in same directory, and open.
Enter your option: █
```

Command for printing a tree



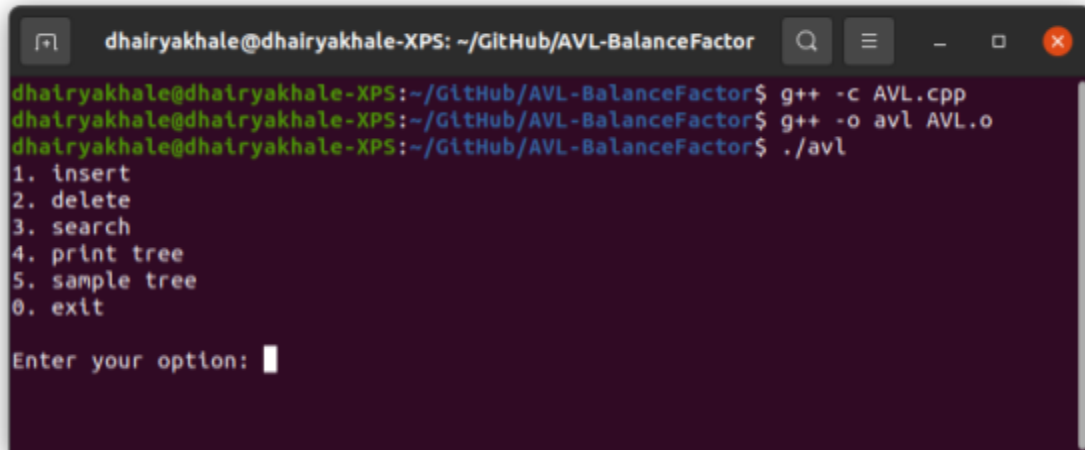
A tree with only one node



A sample tree

## Makefile and valgrind

For ease of running the code, a makefile is provided. Thus, the user does not need to compile and run the code repeatedly, just access the makefile. The makefile is named **avl**.

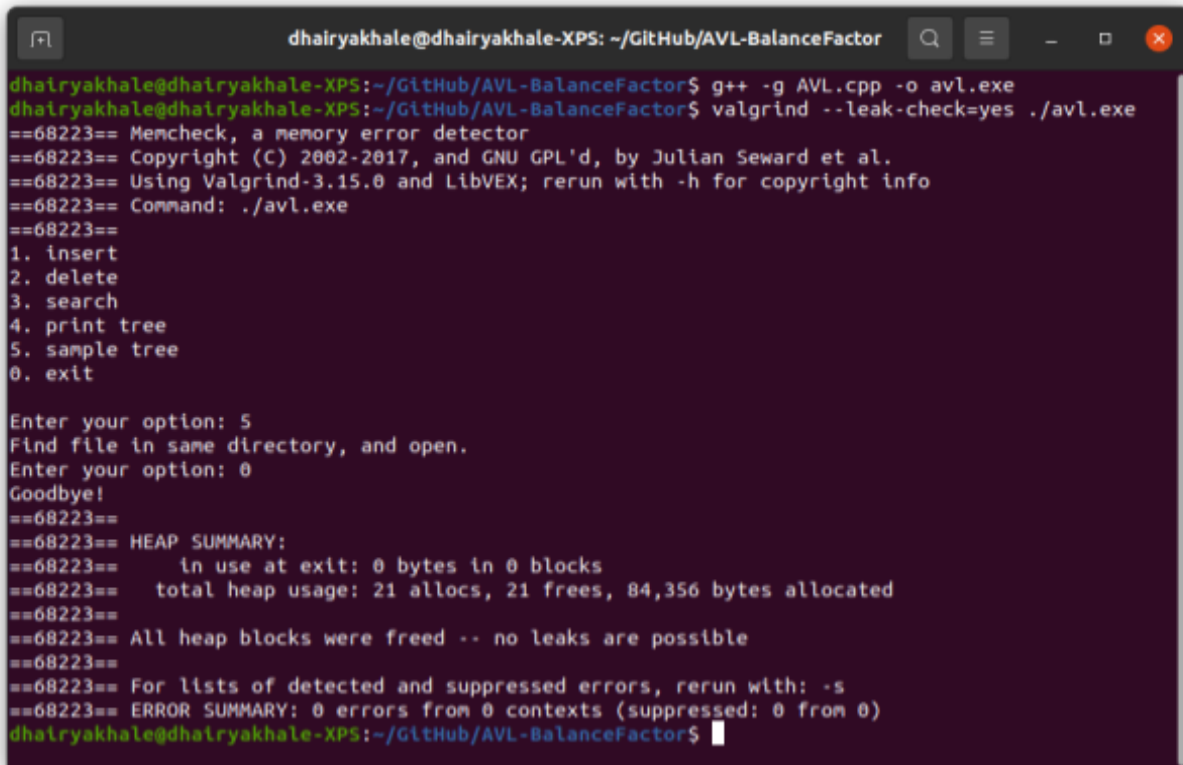


```
dhairyakhale@dhairyakhale-XPS: ~/GitHub/AVL-BalanceFactor
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ g++ -c AVL.cpp
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ g++ -o avl AVL.o
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ ./avl
1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 
```

Creation of makefile

**Valgrind** is a tool which allows us to check if our program is leaking any memory or not. This tool was used on our code and it was observed that no memory was leaked.



```
dhairyakhale@dhairyakhale-XPS: ~/GitHub/AVL-BalanceFactor
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ g++ -g AVL.cpp -o avl.exe
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ valgrind --leak-check=yes ./avl.exe
==68223== Memcheck, a memory error detector
==68223== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==68223== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==68223== Command: ./avl.exe
==68223==
1. insert
2. delete
3. search
4. print tree
5. sample tree
0. exit

Enter your option: 5
Find file in same directory, and open.
Enter your option: 0
Goodbye!
==68223==
==68223== HEAP SUMMARY:
==68223==    in use at exit: 0 bytes in 0 blocks
==68223==   total heap usage: 21 allocs, 21 frees, 84,356 bytes allocated
==68223==
==68223== All heap blocks were freed -- no leaks are possible
==68223==
==68223== For lists of detected and suppressed errors, rerun with: -s
==68223== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dhairyakhale@dhairyakhale-XPS:~/GitHub/AVL-BalanceFactor$ 
```

Running valgrind on the program. The sample tree option also covers an instance of deletion.