Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
COMPS Department

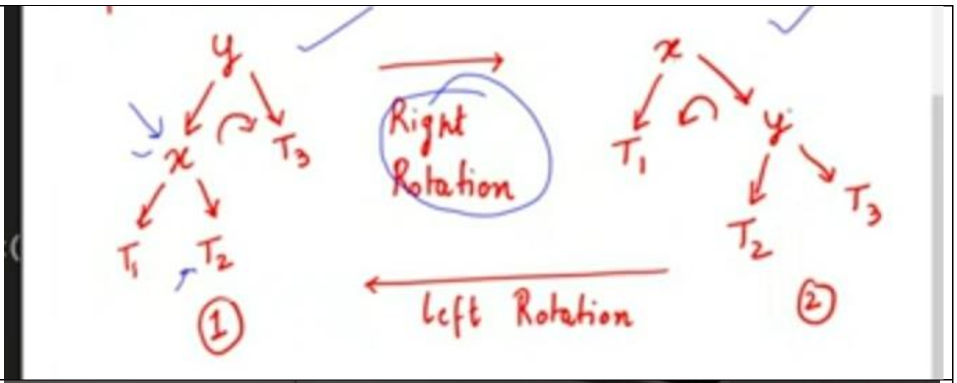
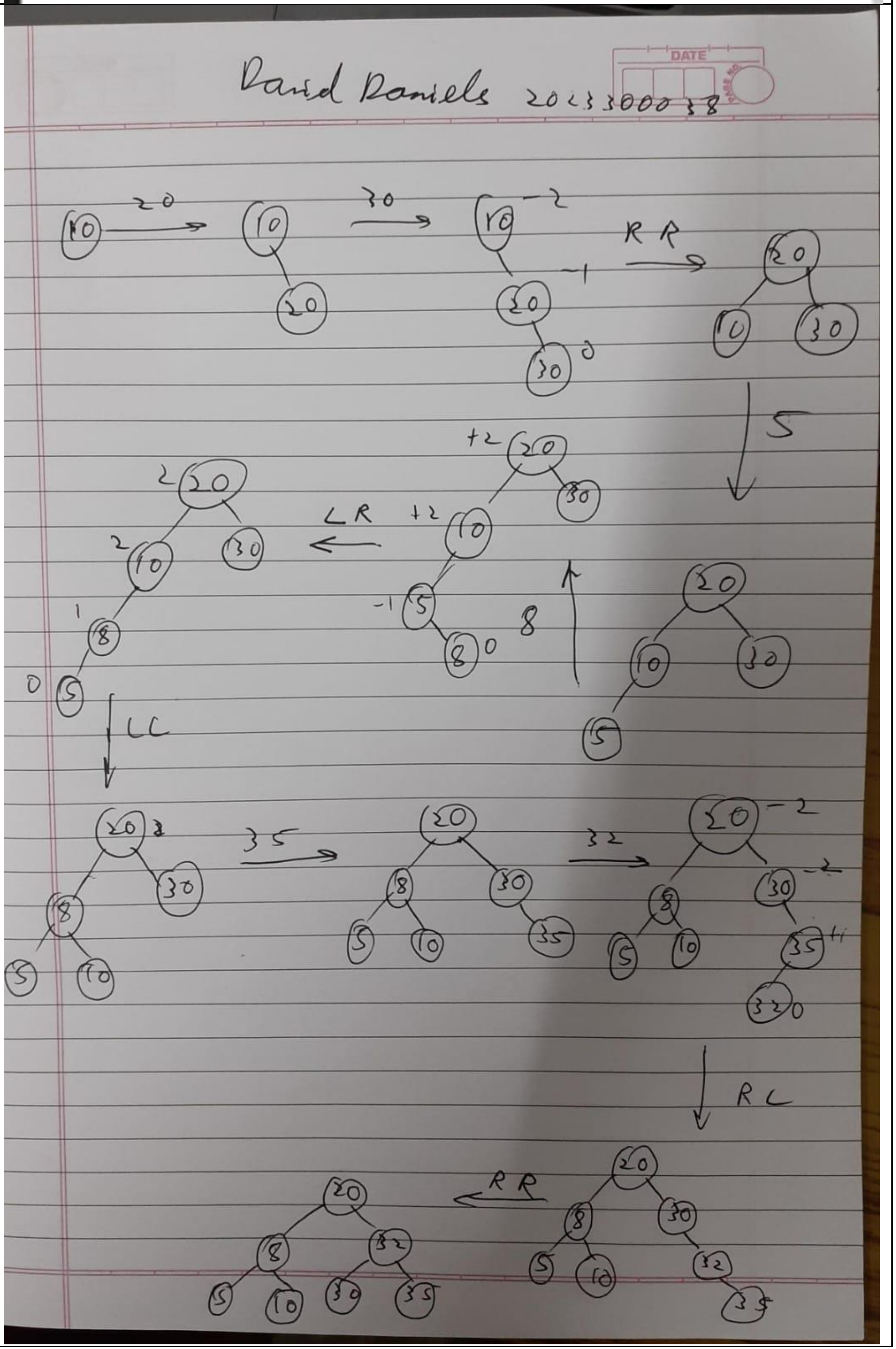
Experiment	7
Aim	**AVL Tree: Perform the following operations on AVL tree data structures: 1. Create 2. Insert (LL , LR , RL , RR rule) 3. Display**
Name	David Daniels
UID	2023300038
Class	Div -A
Batch	C
Date of Submission	8-10-24

Theory	In computer science, an **AVL tree** (named after inventors **A**delson-**V**elsky and **L**andis) is a self-balancing binary search tree. It is a special type of Binary Search Tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations. In a binary tree the *balance factor* of a node X is defined to be the height difference BF(node) = Height(Left Subtree) – Height(Right Subtree) of its two child sub-trees rooted by node X. In an AVL tree, The balance factor of a node must always be between 1 and -1. If it goes beyond that, We must apply a rotation. There are many types of rotations The basic operations performed on the AVL Tree structures include all the operations performed on a binary search tree, since the AVL Tree at its core is actually just a binary search tree holding all its properties. Therefore, basic operations performed on an AVL Tree are − **Insertion** and **Deletion** and traversal. To traverse an AVL tree is same as a binary tree (preorder , postorder,inorder) In the Implementation of an AVL tree, we will have an int height of node along with the data. **Advantages of AVL Tree:** 1. AVL trees can self-balance themselves and therefore provides time complexity as O(Log n) for search, insert and delete.

	2. It is a BST only (with balancing), so items can be traversed in sorted order. 3. Since the balancing rules are strict AVL trees in general have relatively less height and hence the search is faster. **Disadvantages of AVL Tree:** 1. It is difficult to implement compared to normal BST 3. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed. **Applications of AVL Tree:** 1. AVL Tree is used as a first example self balancing BST in teaching DSA as it is easier to understand and implement 2. Applications, where insertions and deletions are less common but frequent data lookups along with other operations of BST like sorted traversal, floor, ceil, min and max. 1. Red Black tree is more commonly implemented in language libraries like map in C++, set in C++, TreeMap in Java and TreeSet in Java. 4. AVL Trees can be used in a real time environment where predictable and consistent performance is required.
Algorithm	1. **Insertion Algorithm**: o The insertion follows the typical binary search tree (BST) insertion, where we insert the new node in the correct position by recursively comparing it with the root and traversing left or right accordingly. o After insertion, the height of each node along the path from the inserted node to the root is updated. 2. **Balancing the Tree**: o After inserting the node, we check the balance factor of each ancestor node and apply rotations if necessary to restore the AVL property. 3. **Balance Factor Calculation**: o After updating the heights of nodes, the balance factor is recalculated as: get_height(n->left) - get_height(n->right) o Depending on the balance factor, one of the four rotation cases is triggered: ▪ bf > 1 (Left Heavy) → Either LL or LR rotation is needed. Depending on the node value ▪ bf < -1 (Right Heavy) → Either RR or RL rotation is needed. Depending on the node value 4. **Handling Four Rotation Cases**: o **Left-Left (LL) Rotation**: ▪ This occurs when a node is inserted into the left subtree of the left child of the unbalanced node (i.e., when the balance factor of the left child is

    - positive, and the newly inserted node is also on the left).
    - To fix this, we perform a **right rotation**.
  - **Right-Right (RR) Rotation**:
    - This occurs when a node is inserted into the right subtree of the right child of the unbalanced node (i.e., when the balance factor of the right child is negative, and the newly inserted node is also on the right).
    - To fix this, we perform a **left rotation**.
  - **Left-Right (LR) Rotation**:
    - This occurs when a node is inserted into the right subtree of the left child of the unbalanced node (i.e., when the balance factor of the left child is positive, but the newly inserted node is on the right).
    - This requires two rotations: first a **left rotation** on the left child, followed by a **right rotation** on the unbalanced node.
  - **Right-Left (RL) Rotation**:
    - This occurs when a node is inserted into the left subtree of the right child of the unbalanced node (i.e., when the balance factor of the right child is negative, but the newly inserted node is on the left).
    - This requires two rotations: first a **right rotation** on the right child, followed by a **left rotation** on the unbalanced node.

5. **Rotation Functions**:
   - **Right Rotation (LL Case)**:
     - It performs a rightward rotation around a node that has become left-heavy (LL case). The left child of the node becomes the new root, and the node itself becomes the right child of the new root.
   - **Left Rotation (RR Case)**:
     - It performs a leftward rotation around a node that has become right-heavy (RR case). The right child of the node becomes the new root, and the node itself becomes the left child of the new root.

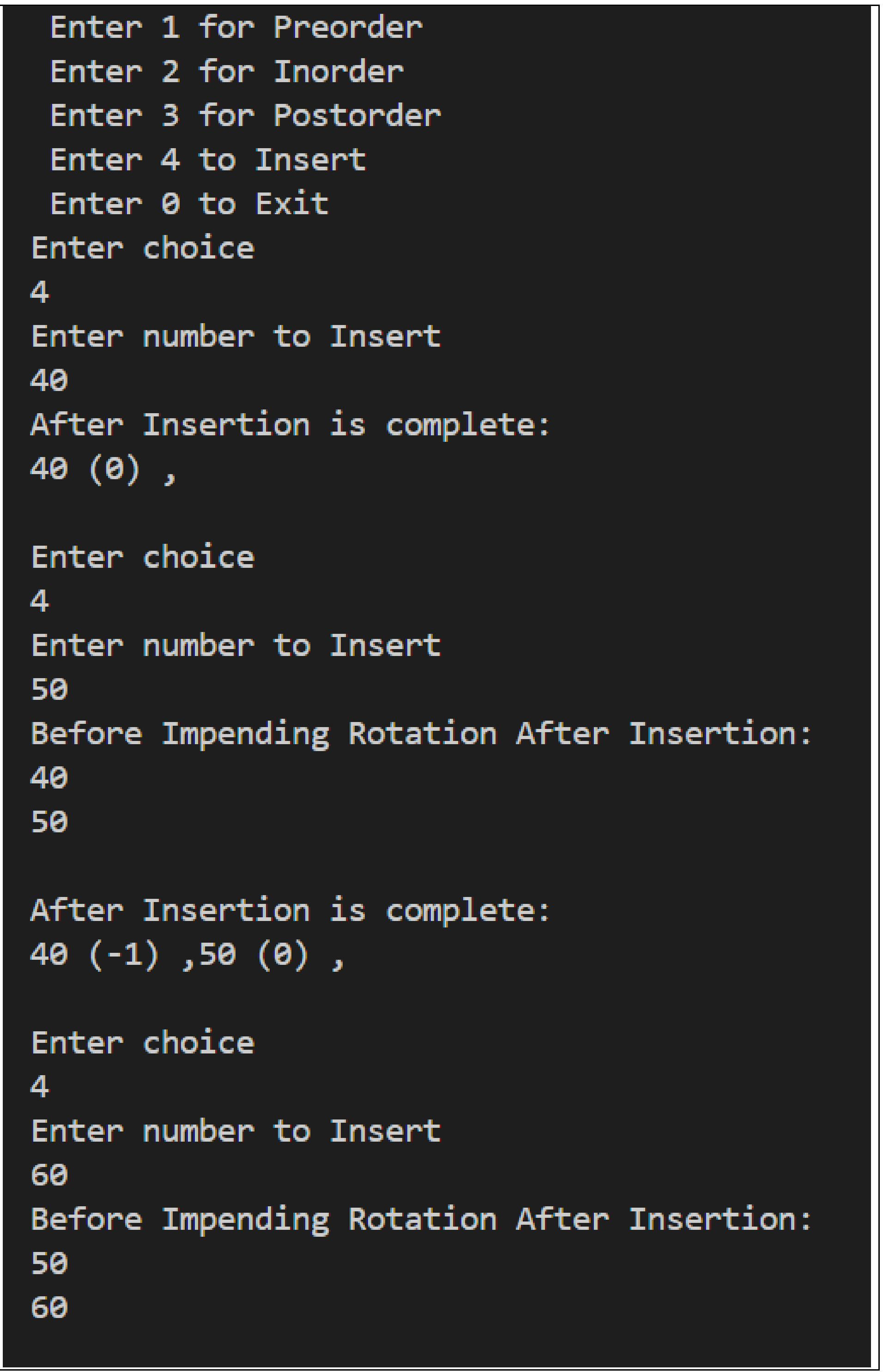
	 y x T1 T2 T3 Right Rotation x y T1 T2 T3 Left Rotation 1 2
Problem Solving	 David Daniels 2023300038 DATE PAGE NO. 20 30 -2 -1 0 RR 5 8 +2 LR -1 LL 35 32 -2 +1 RL RR

Program(Code)	Code
	(see below)

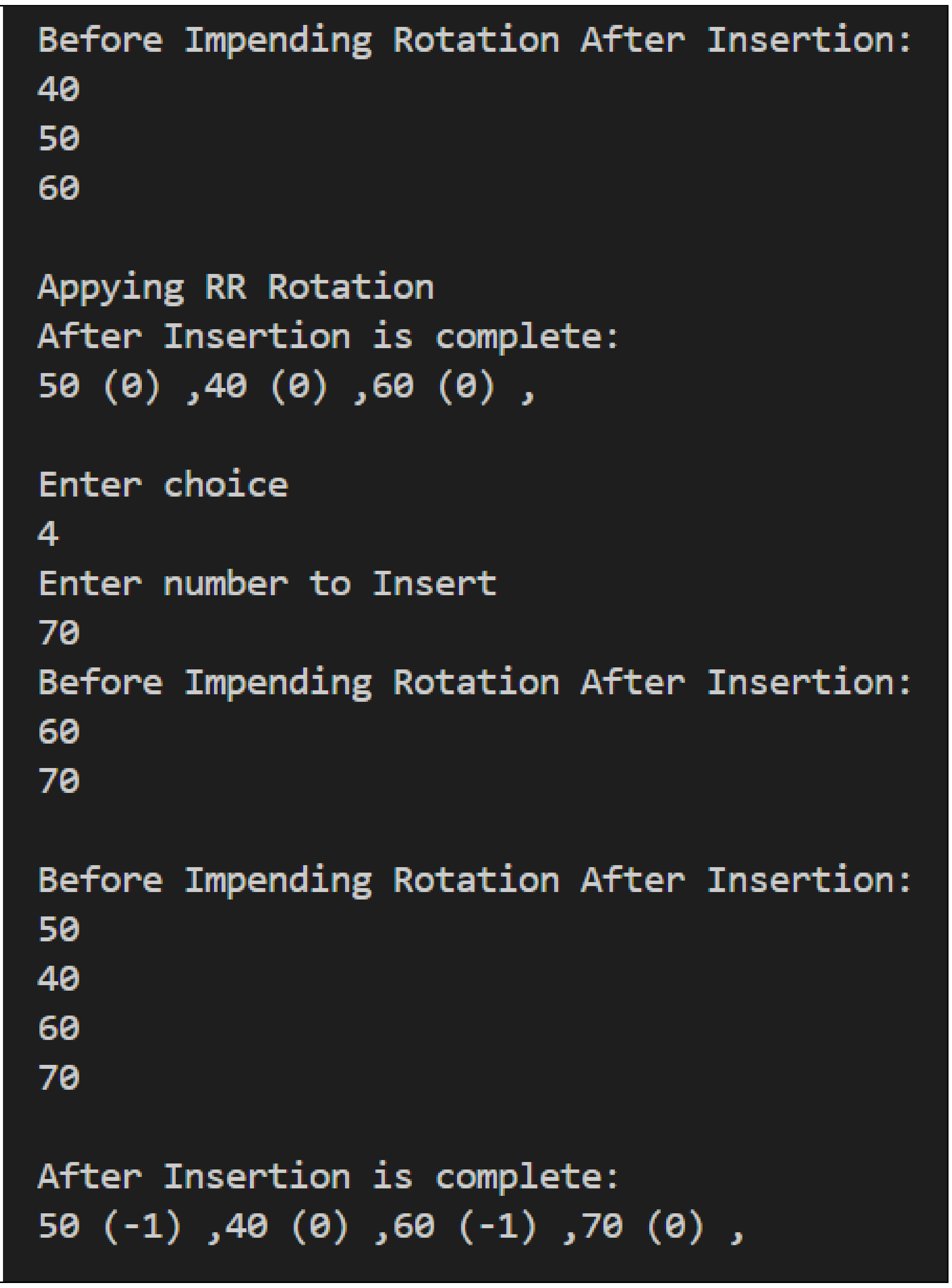
Program(Code)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct node
{
   int data;
   struct node * left;
   struct node * right;
   int height;
}node;

int max(int a, int b)
{
   return a>b?a:b;
}

void preorder(node * root)
{
   if (root!=NULL)
   {
      printf("%d \n",root->data);
      preorder(root->left);
      preorder(root->right);
   }
}

void postorder(node * root)
{
   if (root!=NULL)
   {
      postorder(root->left);
      postorder(root->right);
      printf("%d \n",root->data);
   }
}

void inorder(node * root)
{
   if (root!=NULL)
   {
      inorder(root->left);
      printf("%d \n",root->data);
      inorder(root->right);
   }
}

int get_height(node * n)
{
   return n?n->height:0;
```

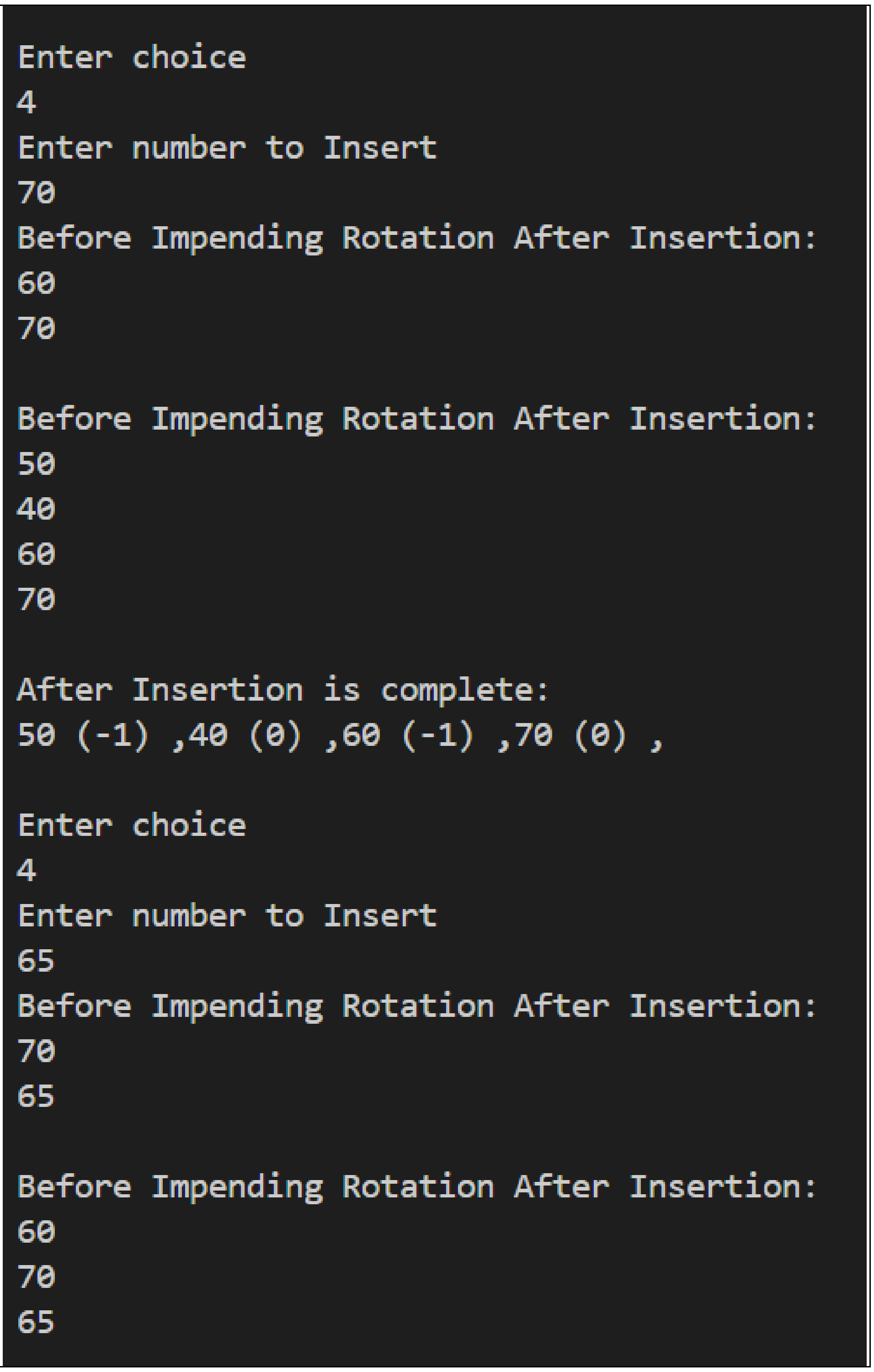
```
}

int get_bal_f(node * n)
{
   return n?get_height(n->left) - get_height(n->right):0;
}

node * create_node(int d)
{
   node * ptr = (node * ) malloc(sizeof(node));
   ptr->left=NULL;
   ptr->right=NULL;
   ptr->data=d;
   ptr->height=1;
   return ptr;
}

void preorder_with_bal(node * root)
{
   if (root!=NULL)
   {
      printf("%d (%d) ,", root->data, get_bal_f(root));
      preorder_with_bal(root->left);
      preorder_with_bal(root->right);
   }
}

node * right_rotate(node * y)
{
   struct node *x = y->left;
   struct node *T2 = x->right;

   x->right = y;
   y->left = T2;

   y->height = max(get_height(y->left),get_height(y->right)) + 1;
   x->height = max(get_height(x->left),get_height(x->right)) + 1;

   return x;
}

node * left_rotate(node * x)
{
   struct node *y = x->right;
   struct node *T2 = y->left;

   y->left = x;
   x->right = T2;

   x->height = max(get_height(x->left),get_height(x->right)) + 1;
   y->height = max(get_height(y->left),get_height(y->right)) + 1;

   return y;
}

node * insert(node * root , int data)
{
   if (root==NULL)
      return create_node(data);
   if (data <root->data)
      root->left=insert(root->left,data);
   else if (data>root->data)
      root->right=insert(root->right,data);

   //we are using recursion
   //dont worry the height will be updated from leaf node onwards (on
the path we are on)
   root->height= 1+ max( get_height(root->left), get_height(root-
>right));
   int bf = get_bal_f(root);

   printf("Before Impending Rotation After Insertion: \n");
   preorder(root);
   printf("\n");


   //recursion always gives the lowest unbalanced node

   //LL
   if (bf>1 && root->left!=NULL && data < root->left->data)
   {
      printf("Appying LL Rotation \n");
      return right_rotate(root);
   }

   //RR
   if (bf<-1 && root->right!=NULL && data > root->right->data )
   {
      printf("Appying RR Rotation \n");
      return left_rotate(root);
   }

   //LR
   if (bf>1 && data > root->left->data)
   {
      printf("Appying LR Rotation \n");
      root->left = left_rotate(root->left);
      return right_rotate(root);
   }
   //RL
   if (bf < -1 && data < root->right->data)
   {
   printf("Applying RL Rotation \n");
   root->right = right_rotate(root->right);
   return left_rotate(root);
   }

   return root;
}

void free_tree (node * root)
{
   if (root==NULL)
   {
      return;
   }

   free_tree(root->left);
   free_tree(root->right);
   free(root);
}

void choice(node * root)
{
   int choice=0,num=0;
   printf(" Enter 1 for Preorder \n Enter 2 for Inorder \n Enter 3 for
Postorder \n Enter 4 to Insert \n Enter 0 to Exit \n");

   while (true)
   {

   printf("Enter choice \n");
      scanf("%d",&choice);
      if (choice==0)
      {
         printf("\nGoodbye\n");
         break;
      }
   switch (choice)
   {
   case 1:
      {
         printf("Preorder Is: \n");
         preorder(root);
         printf("\n\n");
         break;
      }
   case 2:
      {
            printf("Inorder Is: \n");
            inorder(root);
            printf("\n\n");
            break;
        }
    case 3:
        {
            printf("Postorder Is: \n");
            postorder(root);
            printf("\n\n");
            break;
        }
    case 4:
        {
            printf("Enter number to Insert \n");
            scanf("%d", &num);
            root=insert(root,num);
            printf("After Insertion is complete: \n");
            preorder_with_bal(root);
            printf("\n\n");
            break;
        }
    default:
        {
            printf("ERROR”);
            break;
        }
    }
}
}

int main(int argc, char const *argv[])
{
    node * root =NULL;
    choice(root);
    free_tree(root);
    return 0;
}
```

Output	

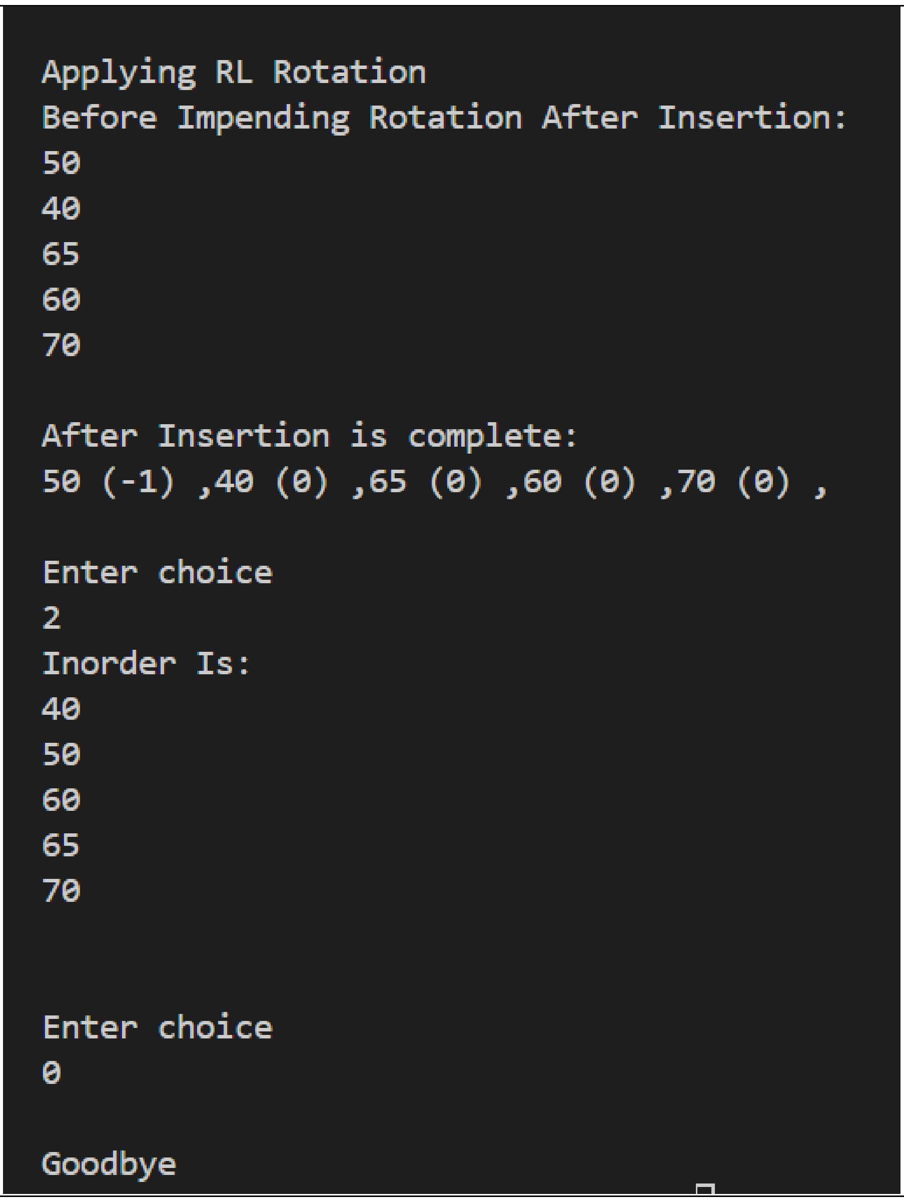
```
 Enter 1 for Preorder
 Enter 2 for Inorder
 Enter 3 for Postorder
 Enter 4 to Insert
 Enter 0 to Exit
Enter choice
4
Enter number to Insert
40
After Insertion is complete:
40 (0) ,

Enter choice
4
Enter number to Insert
50
Before Impending Rotation After Insertion:
40
50

After Insertion is complete:
40 (-1) ,50 (0) ,

Enter choice
4
Enter number to Insert
60
Before Impending Rotation After Insertion:
50
60
```



```
Before Impending Rotation After Insertion:
40
50
60

Appying RR Rotation
After Insertion is complete:
50 (0) ,40 (0) ,60 (0) ,

Enter choice
4
Enter number to Insert
70
Before Impending Rotation After Insertion:
60
70

Before Impending Rotation After Insertion:
50
40
60
70

After Insertion is complete:
50 (-1) ,40 (0) ,60 (-1) ,70 (0) ,
```



```
Enter choice
4
Enter number to Insert
70
Before Impending Rotation After Insertion:
60
70

Before Impending Rotation After Insertion:
50
40
60
70

After Insertion is complete:
50 (-1) ,40 (0) ,60 (-1) ,70 (0) ,

Enter choice
4
Enter number to Insert
65
Before Impending Rotation After Insertion:
70
65

Before Impending Rotation After Insertion:
60
70
65
```



```
Applying RL Rotation
Before Impending Rotation After Insertion:
50
40
65
60
70

After Insertion is complete:
50 (-1) ,40 (0) ,65 (0) ,60 (0) ,70 (0) ,

Enter choice
2
Inorder Is:
40
50
60
65
70


Enter choice
0

Goodbye
```

Conclusion	Thus, we have seen the operations of creating, inserting, and displaying elements in an AVL tree, while adhering to the balancing rules of LL, LR, RL, and RR. These rotations ensure that the tree remains balanced after each insertion, maintaining optimal search, insert, and delete operations in logarithmic time. By implementing and testing various insertion scenarios, we gained a deeper understanding of how AVL trees dynamically balance themselves to avoid skewed structures, which can lead to inefficient performance. Thus we have understood how to maintain balance in a binary search tree using avl tree.