Name	Sujal Dingankar
UID no.	DSE24100720
Experiment No.	7

AIM:	AVL Tree: Perform the following operations on AVL tree data structures: 1. Create 2. Insert (LL , LR , RL , RR rule) 3. Display
THEORY:	**What is a Binary Search Tree (BST) and its Disadvantages?** A Binary Search Tree (BST) is a tree data structure where each node has at most two children, typically referred to as the left and right child. In a BST, for every node: • The left subtree contains nodes with values less than the node's value. • The right subtree contains nodes with values greater than the node's value. **Disadvantages of BST:** Although a BST is efficient for search, insertion, and deletion operations (with an average time complexity of O(log n)), it can become unbalanced. If the nodes are inserted in a sorted order, the BST can degrade into a linked list-like structure, with all nodes either on the left or right, making operations like search, insert, and delete as slow as O(n). This loss of balance is the key disadvantage. **What is an AVL Tree and How Does It Overcome the Disadvantages of BST?** • An **AVL tree** is a self-balancing binary search tree, named after its inventors Adelson-Velsky and Landis. In an AVL tree, the difference in heights between the left and right subtrees (called the balance factor) is at most 1 for every node. This balance is maintained by performing **rotations** whenever nodes are inserted or deleted, ensuring that the tree remains balanced. • By maintaining this balance, AVL trees prevent the worst-case time complexity of BSTs. Even in the worst case, the height of an AVL tree is logarithmic (O(log n)), which guarantees that search, insertion, and deletion operations are consistently fast.

Why Do We Use AVL Trees and What Operations Can We Perform?

We use AVL trees when we need consistently fast search, insertion, and deletion operations. Unlike regular BSTs, AVL trees ensure that the height remains logarithmic, making them more reliable for use in applications where performance is critical.

Operations that can be performed on AVL Trees:

- Search: Just like in a BST, searching in an AVL tree is efficient and has a time complexity of O(log n).
- Insertion: When a new node is inserted, the balance of the tree is checked and if the balance factor becomes more than 1 or less than -1, rotations are performed to restore the balance.
- Deletion: Similar to insertion, after deletion, the tree may become unbalanced, and rotations are used to maintain the AVL property.
- Rotations: There are four types of rotations used to maintain balance: Left rotation, Right rotation, Left-Right rotation, and Right-Left rotation.

Applications of AVL Trees

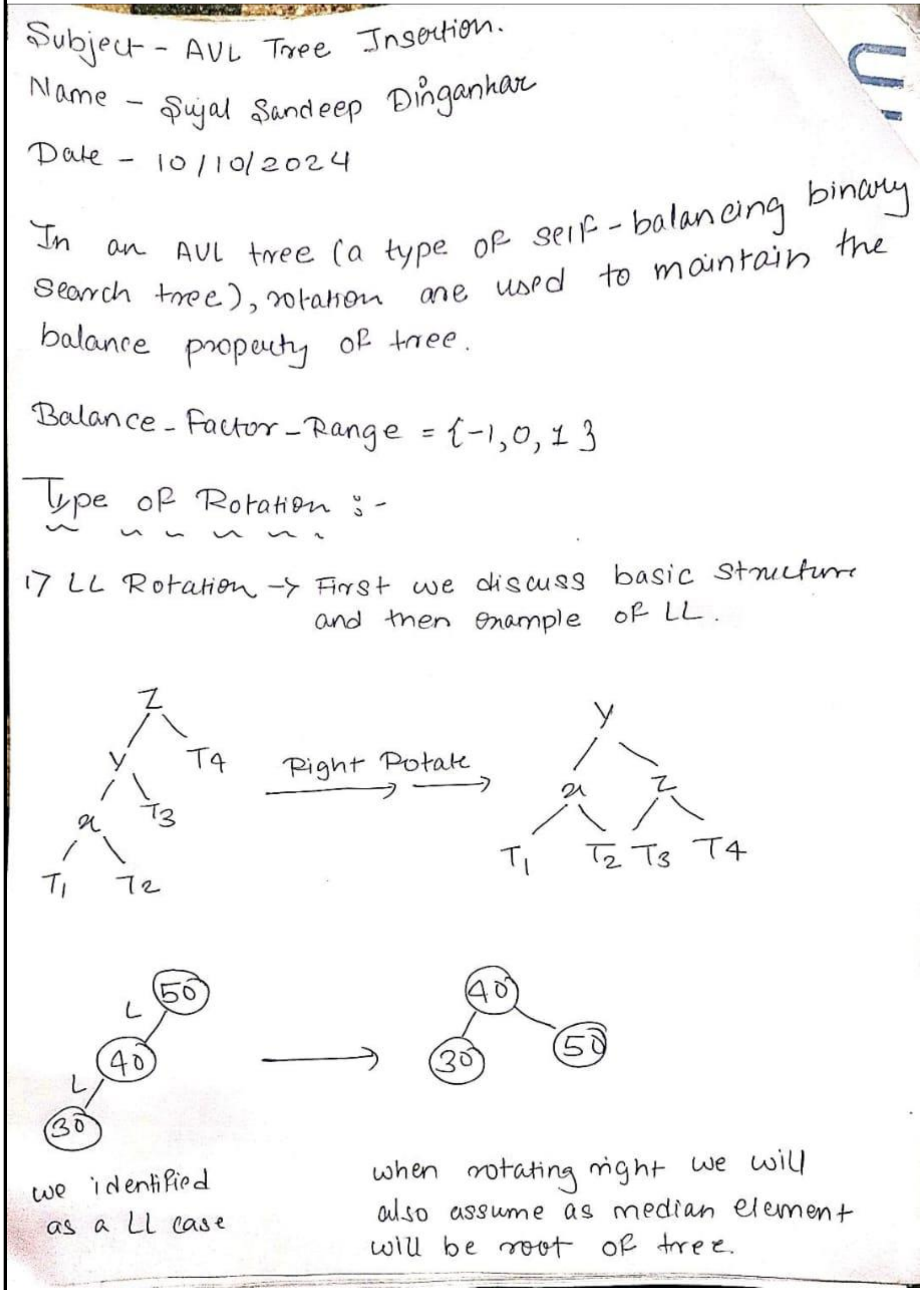
AVL trees are useful in scenarios where data is dynamic and frequent insertions and deletions happen, but balance needs to be maintained for fast access. Some applications include:

- **Databases**: AVL trees are used to index large sets of data to ensure efficient querying, insertion, and deletion.
- **Memory Management**: Operating systems use AVL trees to manage free memory blocks, ensuring efficient allocation and deallocation.
- **File Systems**: Some file systems use AVL trees to manage directories and files for efficient searching and retrieval.
- **Networking**: In network routing algorithms, balanced trees like AVL trees can be used to optimize pathfinding and improve performance.

ALGORITHM:	1. Initialize an empty tree: o Start with an empty AVL tree (root is initially NULL). 2. Menu loop for AVL operations: o Continuously display the menu until the user chooses to exit. 3. Insert Element (Option 1): o When the user selects option 1: 1. Prompt the user to enter the key (data value) for the new node. 2. Insert the new node into the AVL tree: ▪ If the tree is empty, create the new node as the root. ▪ If the tree is not empty, perform a Binary Search Tree (BST) insertion: ▪ Traverse the tree to find the correct position to insert the new node (left if the key is smaller, right if it's larger). ▪ Insert the new node in the correct position. 3. Update node heights and balance the tree: ▪ After insertion, update the height of all ancestor nodes. ▪ Check the balance factor (height difference between left and right subtrees) of each node: ▪ If the balance factor is greater than 1 or less than -1, the tree is unbalanced. ▪ Perform appropriate rotations to balance the tree: ▪ Right Rotation (LL rotation) if the node is left-heavy and its left subtree is also left-heavy. ▪ Left Rotation (RR rotation) if the node is right-heavy and its right subtree is also right-heavy.

	▪ Left-Right Rotation (LR rotation) if the node is left-heavy, but its left subtree is right-heavy. ▪ Right-Left Rotation (RL rotation) if the node is right-heavy, but its right subtree is left-heavy. 4. After rebalancing (if needed), the tree remains balanced. 4. Pre-order Traversal (Option 2): o When the user selects option 2: 1. Traverse the tree in pre-order (root, left, right): ▪ For each node, first visit the node itself, then recursively visit the left subtree, followed by the right subtree. 2. Print the key and balance factor of each node in the tree. 5. Exit (Option 3): o When the user selects option 3, the program will exit.

PROBLEM SOLVING:



Subject - AVL Tree Insertion.

Name - Sujal Sandeep Dinganhar

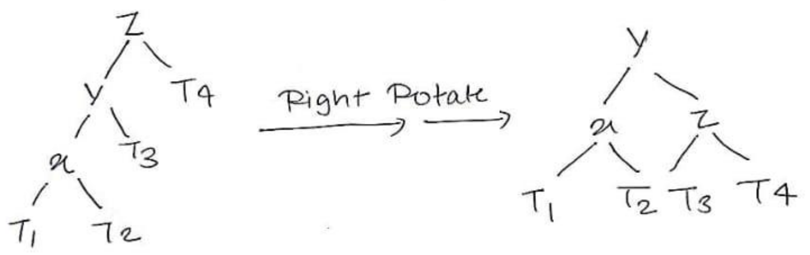
Date - 10/10/2024

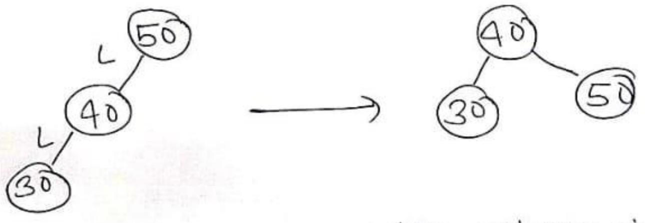
In an AVL tree (a type of self-balancing binary search tree), rotation are used to maintain the balance property of tree.

Balance-Factor-Range = {-1, 0, 1}

Type of Rotation :-

1) LL Rotation -> First we discuss basic structure and then example of LL.



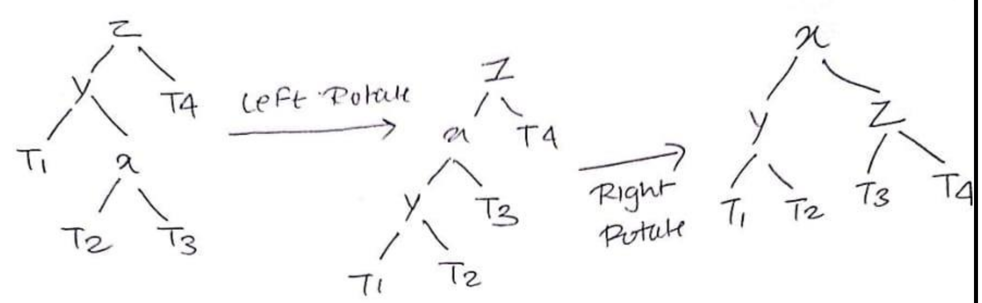


we identified as a LL case

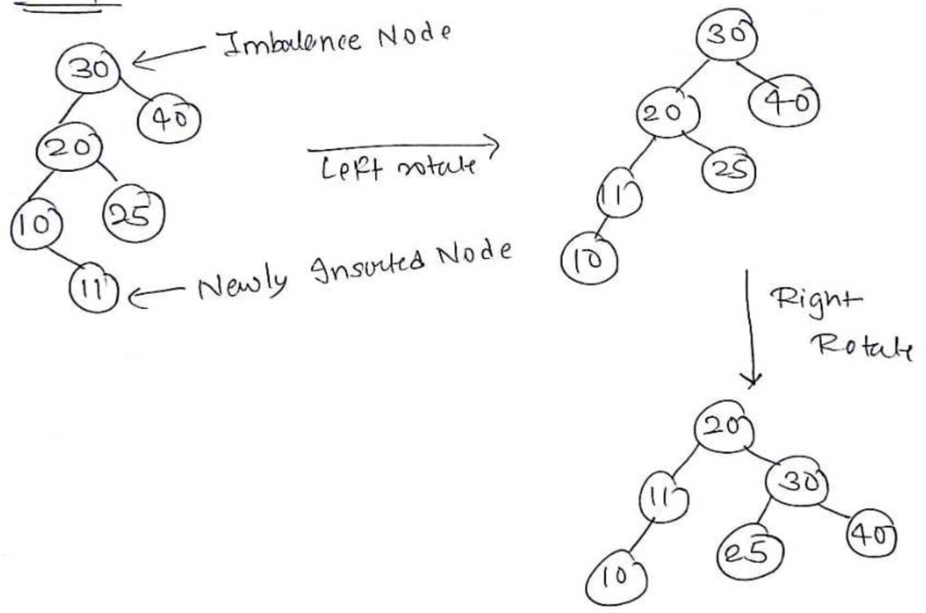
when rotating right we will also assume as median element will be root of tree.

2> LR (left - Right) Rotation :-

First we discuss basic structure :-

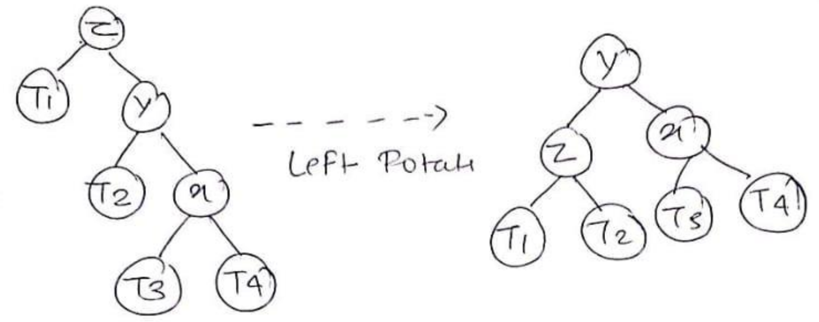


Example :-

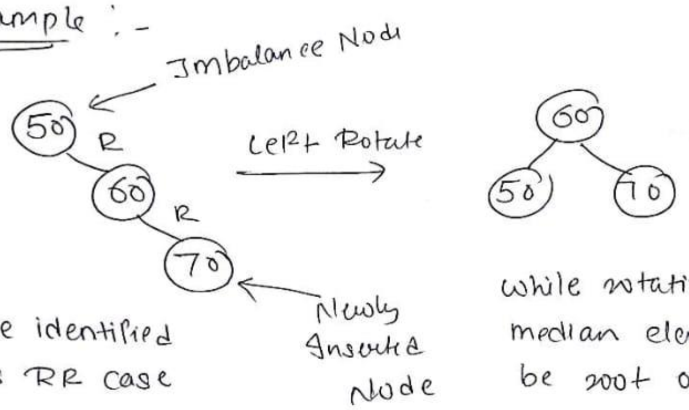


3) RR [Right - Right] Rotation :-

First we disuss basic structure :-



Example :-

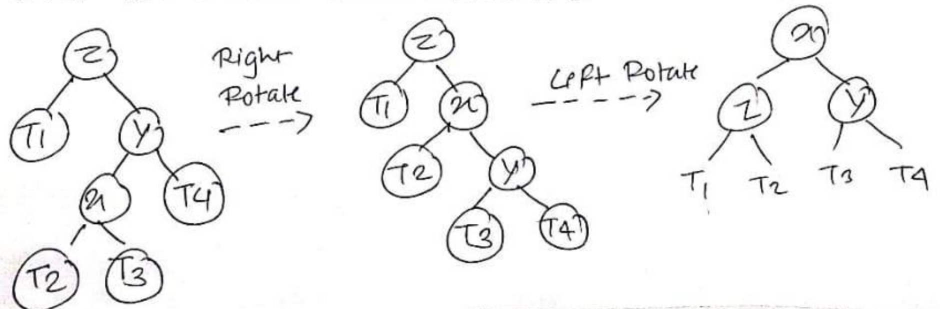


we identified as RR case

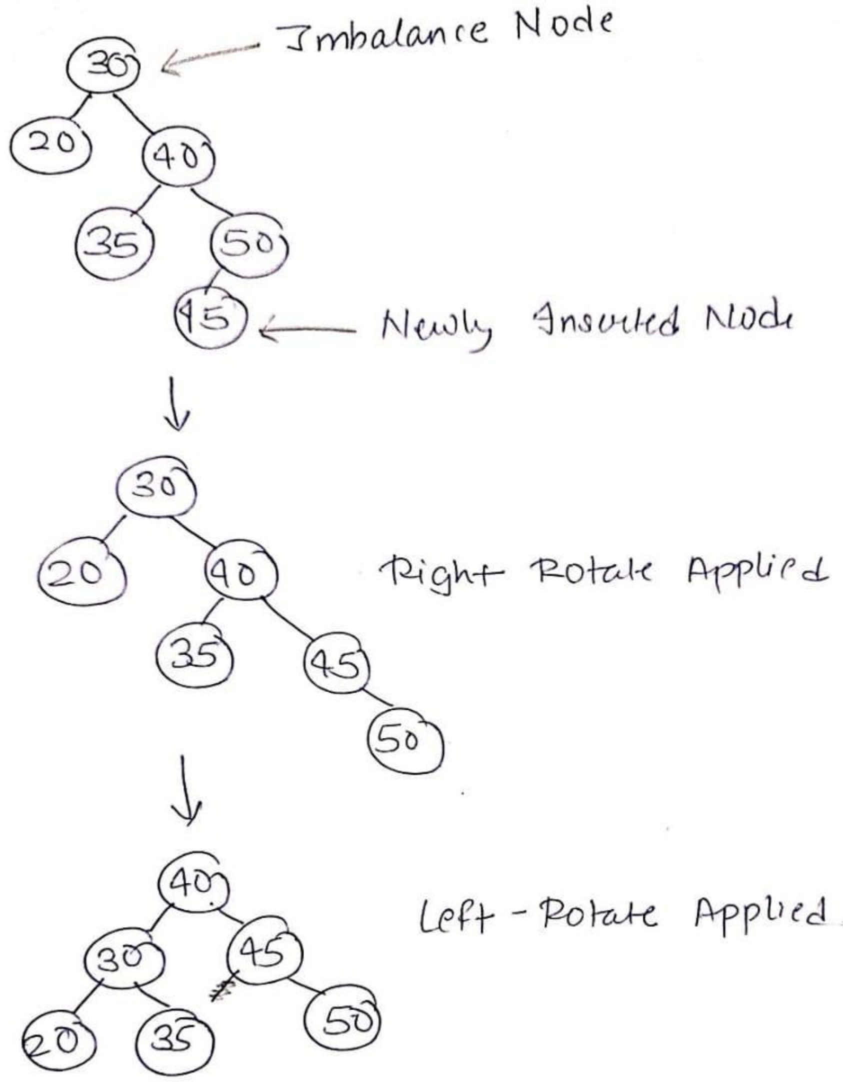
while rotating left median element will be root of tree.

4) RL [Right - Left] Rotation :-

First we discuss basic structure :-



Example :-


Imbalance Node
Newly Inserted Node
Right Rotate Applied
Left - Rotate Applied.

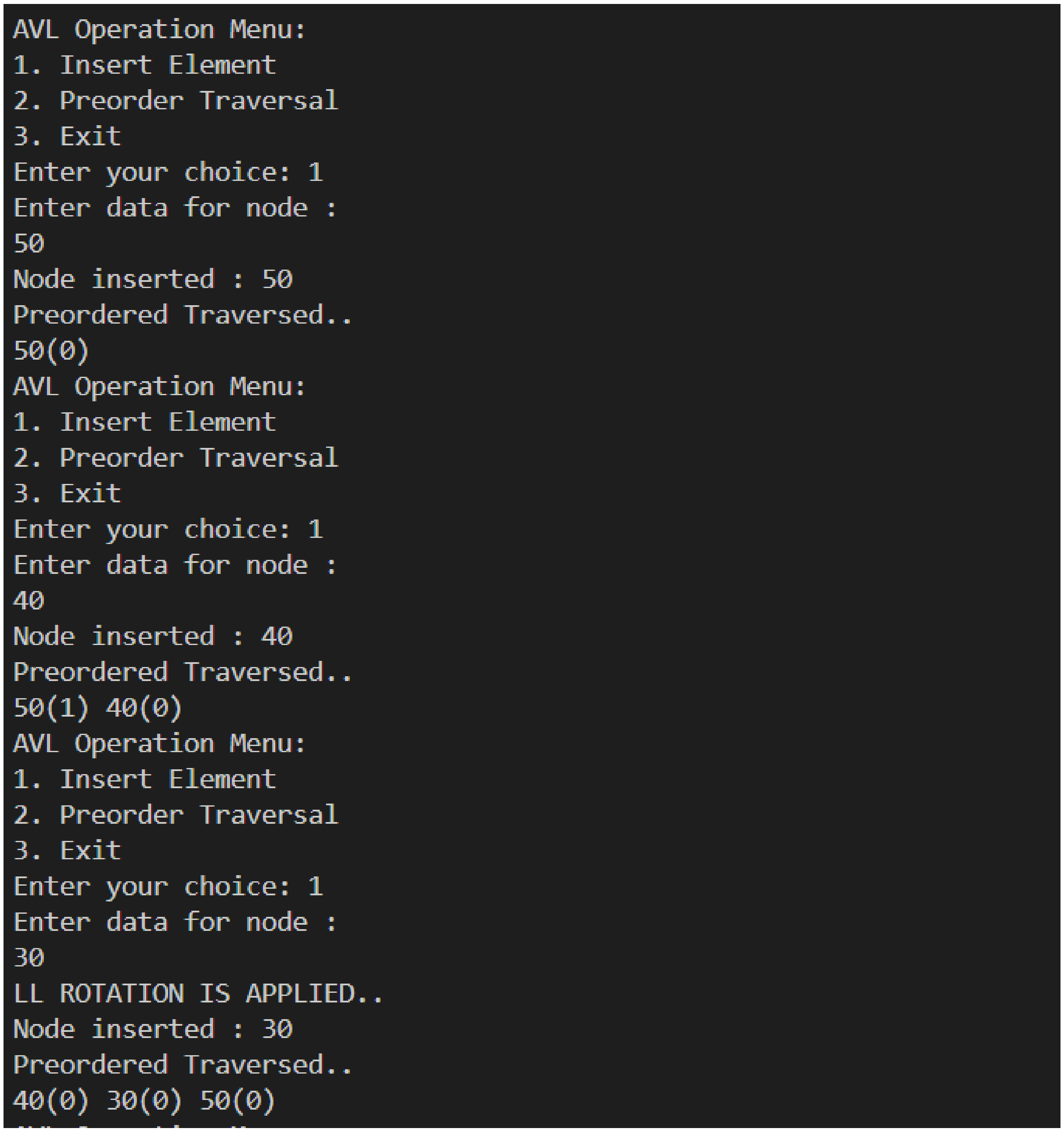
PROGRAM :-

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

struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// Calculate height of the node
int height(struct Node *n) {
    if (n == NULL)
        return 0;
    return n->height;
}

// Calculate max of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Create a new node
struct Node *createNode(int key) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->height = 1;
    return newNode;
}

// Get balance factor of node n
int balanceFactor(struct Node *n) {
    if (n == NULL)
        return 0;
    return height(n->left) - height(n->right);
}

// Perform right rotation
struct Node *rightRotate(struct Node *y) {
    printf("LL ROTATION IS APPLIED..\n");
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// Perform left rotation
struct Node *leftRotate(struct Node *x) {
    printf("RR ROTATION IS APPLIED..\n");
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}
```

```
// Insertion in AVL tree
struct Node *insertion(struct Node *root, int key) {
    // Perform standard BST insertion
    if (root == NULL)
        return createNode(key);

    if (key < root->key)
        root->left = insertion(root->left, key);
    else if (key > root->key)
        root->right = insertion(root->right, key);
    else
        return root; // Duplicate keys not allowed

    // Update height of current node
    root->height = 1 + max(height(root->left), height(root->right));

    // Get balance factor of this node
    int balance = balanceFactor(root);

    // Left Left Case (RR rotation)
    if (balance > 1 && key < root->left->key)
        return rightRotate(root);

    // Right Right Case (LL rotation)
    if (balance < -1 && key > root->right->key)
        return leftRotate(root);

    // Left Right Case (LR rotation)
    if (balance > 1 && key > root->left->key) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Left Case (RL rotation)
    if (balance < -1 && key < root->right->key) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Pre-order traversal of the tree
void preOrder(struct Node *root) {
    if (root != NULL) {
        preOrder(root->left);
        printf("%d(%d) " , root->key,balanceFactor(root));
        preOrder(root->right);
    }
}

int main() {

    int choice;
    struct Node *root = NULL;



    do{
        printf("\nAVL Operation Menu:\n");
        printf("1. Insert Element\n");
        printf("2. Preorder Traversal\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
        {
            int key;
            printf("Enter data for node :\n");
            scanf("%d",&key);
            root = insertion(root,key);
            printf("Node inserted : %d \n",key);
            preOrder(root);
        }
            break;
        case 2:
            printf("Preordered Traversed..\n");
            preOrder(root);
            break;
        case 3:
            break;
        default:
            break;
        }
    }while (choice != 3);

    return 0;
}
```

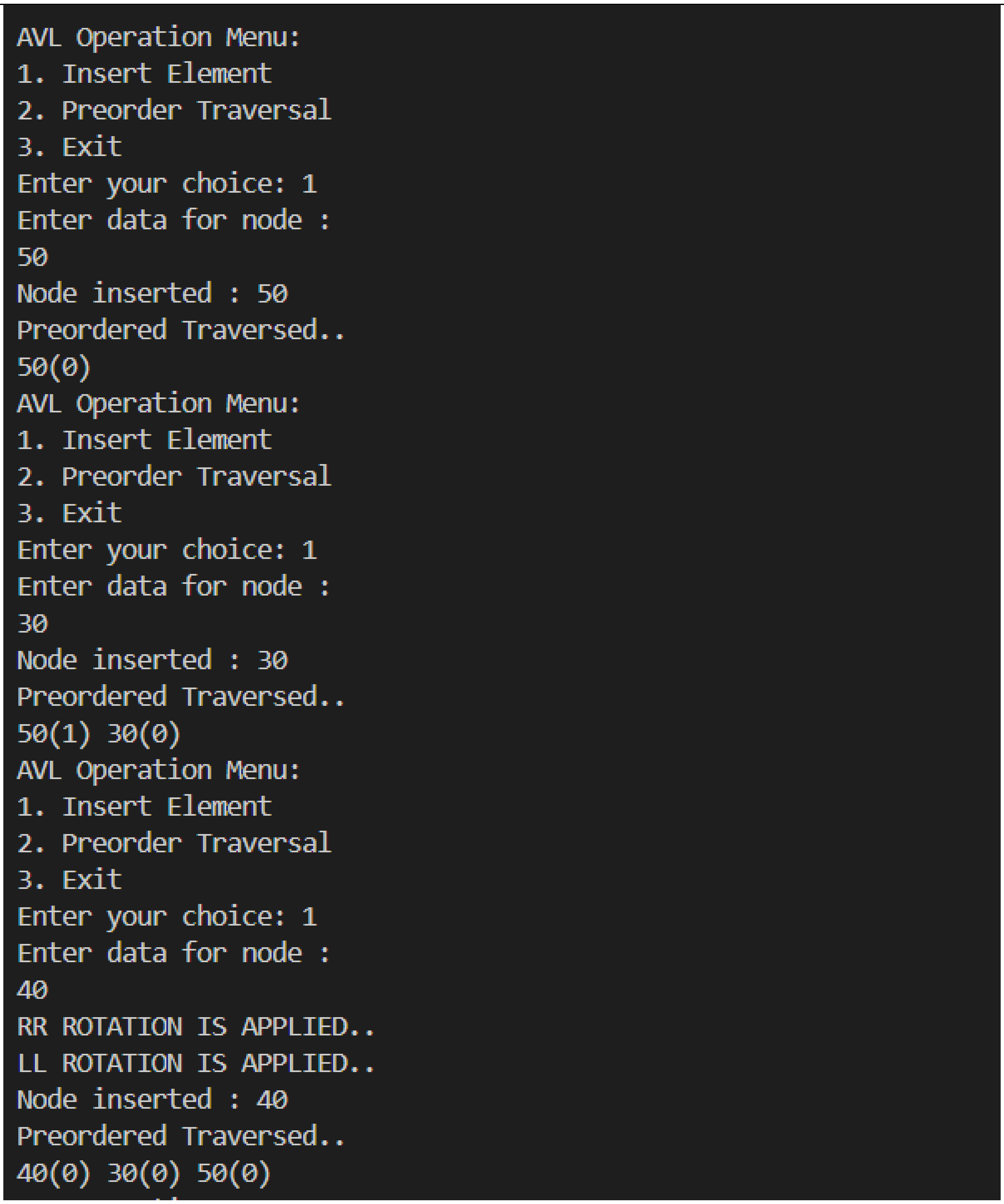
RESULT :-

1. LL Rotation :-



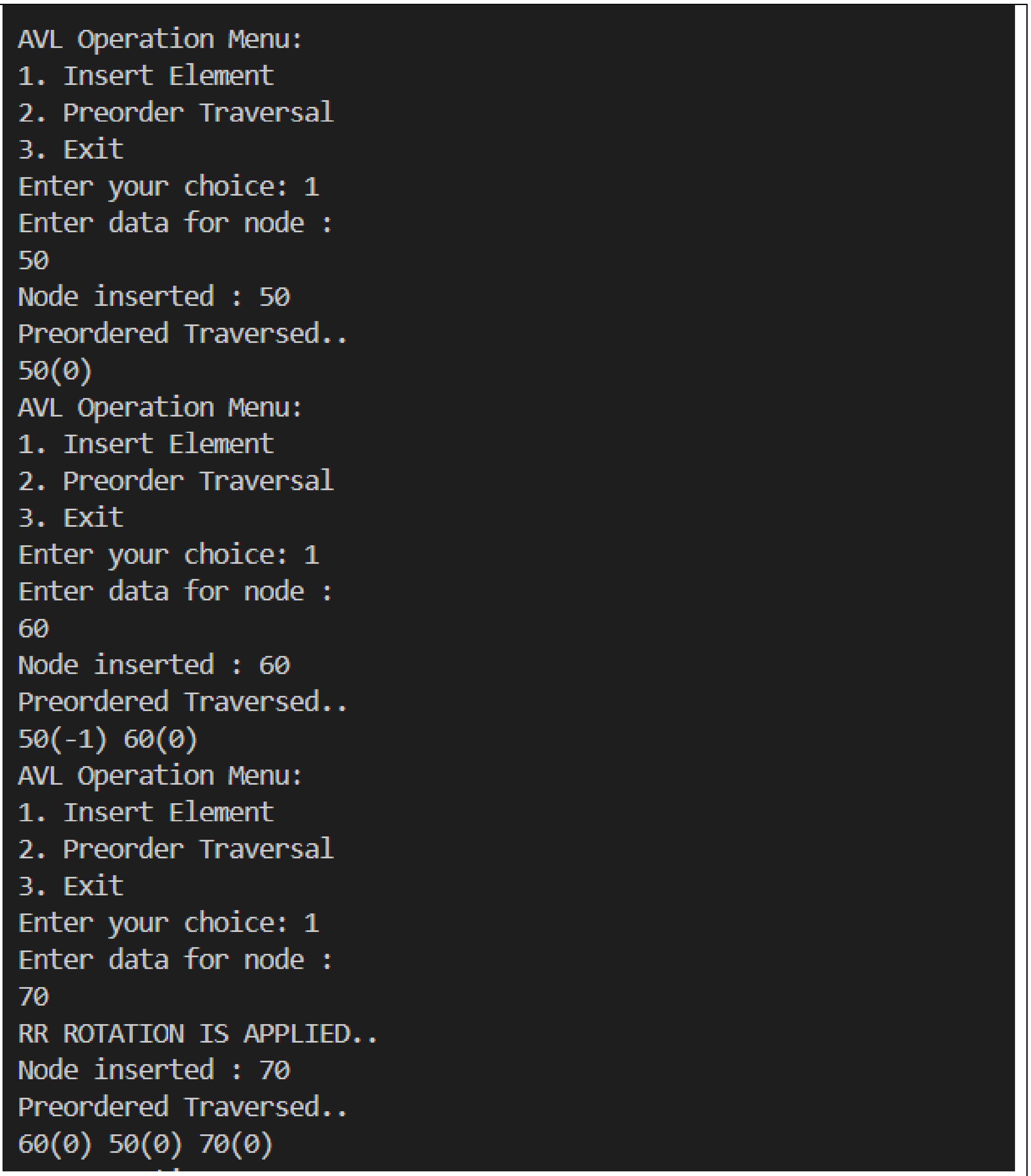
```
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
50
Node inserted : 50
Preordered Traversed..
50(0)
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
40
Node inserted : 40
Preordered Traversed..
50(1) 40(0)
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
30
LL ROTATION IS APPLIED..
Node inserted : 30
Preordered Traversed..
40(0) 30(0) 50(0)
```

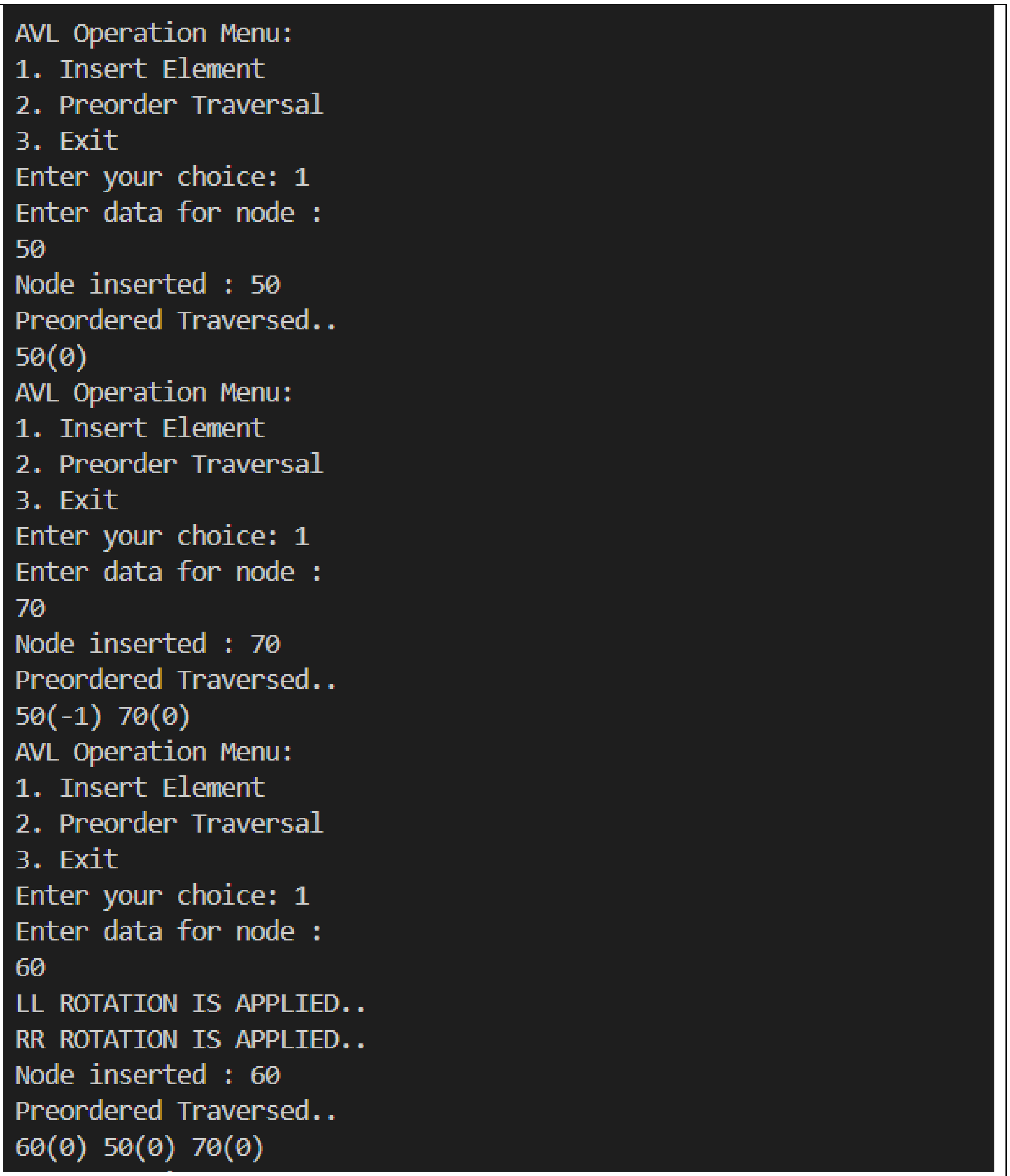
2. LR Rotation :-



```
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
50
Node inserted : 50
Preordered Traversed..
50(0)
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
30
Node inserted : 30
Preordered Traversed..
50(1) 30(0)
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
40
RR ROTATION IS APPLIED..
LL ROTATION IS APPLIED..
Node inserted : 40
Preordered Traversed..
40(0) 30(0) 50(0)
```

3. RR Rotation :-

	
	```
	AVL Operation Menu:
	1. Insert Element
	2. Preorder Traversal
	3. Exit
	Enter your choice: 1
	Enter data for node :
	50
	Node inserted : 50
	Preordered Traversed..
	50(0)
	AVL Operation Menu:
	1. Insert Element
	2. Preorder Traversal
	3. Exit
	Enter your choice: 1
	Enter data for node :
	60
	Node inserted : 60
	Preordered Traversed..
	50(-1) 60(0)
	AVL Operation Menu:
	1. Insert Element
	2. Preorder Traversal
	3. Exit
	Enter your choice: 1
	Enter data for node :
	70
	RR ROTATION IS APPLIED..
	Node inserted : 70
	Preordered Traversed..
	60(0) 50(0) 70(0)
	```
	4. RL Rotation :-

```
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
50
Node inserted : 50
Preordered Traversed..
50(0)
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
70
Node inserted : 70
Preordered Traversed..
50(-1) 70(0)
AVL Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Exit
Enter your choice: 1
Enter data for node :
60
LL ROTATION IS APPLIED..
RR ROTATION IS APPLIED..
Node inserted : 60
Preordered Traversed..
60(0) 50(0) 70(0)
```

<u>Conclusion</u> :-

From this experiment with AVL Trees, I learned how to maintain a balanced binary search tree using self-balancing rotations. By ensuring the tree's height remains balanced, AVL trees guarantee efficient search, insertion, and deletion operations with O(log n) time complexity. This experience enhanced my understanding of the importance of balance factors and the role of AVL trees in optimizing dynamic data structures for performance-critical applications like databases and memory management.