LAB 5

AVL TREE

CODE:

```
  GNU nano 7.2
#include <stdio.h>
#include <stdlib.h>

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

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

int height(struct Node *N) {
    if (N == NULL)
        return 0;
    return N->height;
}

struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
```

```
  GNU nano 7.2
    struct Node* T2 = x->right;

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

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

    return x;
}

struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

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

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

    return y;
}

int getBalance(struct Node* N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct Node* insert(struct Node* node, int data) {
    if (node == NULL)
        return newNode(data);
```

GNU nano 7.2

```
        return newNode(data);

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && data < node->left->data)
        return rightRotate(node);

    if (balance < -1 && data > node->right->data)
        return leftRotate(node);

    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void inorder(struct Node* root) {
    if (root != NULL) {
```

```
  GNU nano 7.2
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

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

int main() {
    struct Node* root = NULL;

    int rollNumbers[] = {
        157, 110, 147, 122, 111, 149,
        151, 141, 123, 112, 117, 133
    };

    int n = sizeof(rollNumbers) / sizeof(rollNumbers[0]);

    for (int i = 0; i < n; i++) {
        root = insert(root, rollNumbers[i]);
    }

    inorder(root);

    return 0;
}
```

OUTPUT:

```
summu@summu-VirtualBox:~$ nano avl.c
summu@summu-VirtualBox:~$ gcc avl.c -o avl
summu@summu-VirtualBox:~$ ./avl
110 111 112 117 122 133 133 141 147 149 151 157
summu@summu-VirtualBox:~$ ./avl
```

TIME COMPLEXITY:

The time complexity of search, insertion, and deletion in an AVL tree is **O(log n)**.

REASON:

For every node, the height difference between left and right subtrees is at most 1.
This strict balancing ensures that the height of the tree is always proportional to log n, so all operations take logarithmic time.

RED ALGORITHM:

CODE:

```
  GNU nano 7.2
#include <stdio.h>
#include <stdlib.h>

typedef enum { RED, BLACK } Color;

struct Node {
    int data;
    Color color;
    struct Node *left, *right, *parent;
};

struct Node* root = NULL;

struct Node* createNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->color = RED;
    node->left = node->right = node->parent = NULL;
    return node;
}

void leftRotate(struct Node* x) {
    struct Node* y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
```

```
  GNU nano 7.2
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

void rightRotate(struct Node* y) {
    struct Node* x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == NULL)
        root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}

void fixInsert(struct Node* z) {
    while (z != root && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            struct Node* y = z->parent->parent->right;
            if (y != NULL && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
```

```
  GNU nano 7.2
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(z->parent->parent);
            }
        } else {
            struct Node* y = z->parent->parent->left;
            if (y != NULL && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(z->parent->parent);
            }
        }
    }
    root->color = BLACK;
}

void insert(int data) {
```

```
  GNU nano 7.2

void insert(int data) {
    struct Node* z = createNode(data);
    struct Node* y = NULL;
    struct Node* x = root;

    while (x != NULL) {
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }

    z->parent = y;
    if (y == NULL)
        root = z;
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;

    fixInsert(z);
}

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

int main() {
```

```
int main() {
    int rollNumbers[] = {
        157, 110, 147, 122, 111, 149,
        151, 141, 123, 112, 117, 133
    };

    int n = sizeof(rollNumbers) / sizeof(rollNumbers[0]);

    for (int i = 0; i < n; i++)
        insert(rollNumbers[i]);

    inorder(root);

    return 0;
}
```

OUTPUT:

```
summu@summu-VirtualBox:~$ nano red.c
summu@summu-VirtualBox:~$ gcc red.c  -o red
summu@summu-VirtualBox:~$ ./red
110 111 112 117 122 123 133 133 141 141 149 151 157
```

TIME COMPLEXITY:

The time complexity of search, insertion, and deletion in a Red-Black tree is O(log n).

REASON:

Red-Black tree maintains balance using node coloring rules instead of strict height conditions.These rules ensure that the longest path from root to leaf is at most twice the shortest path, keeping the tree height bounded by log n and guaranteeing logarithmic time operations.