NAME: CHANTATI SAI PURNISHA MAHI

ROLL NO:CH.SC.U4CSE24156

(Design and Analysis Algorithms)

WEEK 5:-

AVL TREE:-

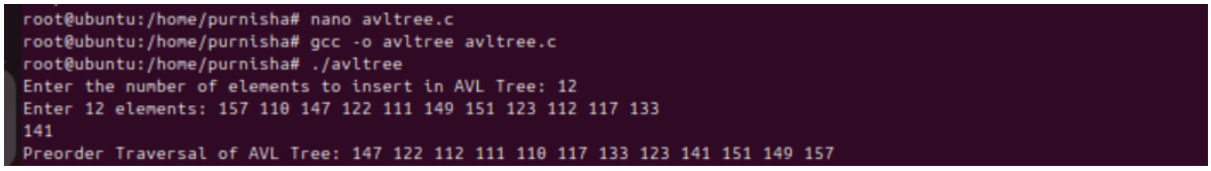
CODE:-

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

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

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

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

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

struct AVLNode* rightRotate(struct AVLNode* y) {
    struct AVLNode* x = y->left;
    struct AVLNode* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right) + 1);
```

```c
struct AVLNode* leftRotate(struct AVLNode* x) {
    struct AVLNode* y = x->right;
    struct AVLNode* 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 AVLNode* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

struct AVLNode* insert(struct AVLNode* node, int data) {
    if (node == NULL)
        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)
```

```c
void preOrder(struct AVLNode* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main() {
    struct AVLNode* root = NULL;
    int n;
    printf("Enter the number of elements to insert in AVL Tree: ");
    scanf("%d", &n);
    int data;
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &data);
        root = insert(root, data);
    }
    printf("Preorder Traversal of AVL Tree: ");
    preOrder(root);
    printf("\n");
    return 0;
}
```

OUPUT:-



TIME COMPLEXITY:- O(n log n)

JUSTIFICATION:-

Insertion of one element in an AVL tree takes **O(log n)** time because the tree remains height-balanced.
For **n elements**, total insertion time = **n × log n**.
Preorder traversal visits each node once, which takes **O(n)** time.
Overall time complexity is dominated by insertion, so it is **O(n log n)**.

SPACE COMPLEXITY:- O(n)

JUSTIFICATION:-

Each node stores:

- int key → 4 bytes
- left pointer → 8 bytes
- right pointer → 8 bytes
- height → 4 bytes

For **n nodes**, memory grows linearly.
Recursive function calls also use stack space

proportional to tree height **O(log n)**.
Hence, total space complexity is **O(n)**.

REDBALCKTREE:-

CODE:-

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

#define RED 0
#define BLACK 1

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

struct RBNode* TNULL;

struct RBNode* newRBNode(int data) {
    struct RBNode* node = (struct RBNode*)malloc(sizeof(struct RBNode));
    node->data = data;
    node->left = node->right = node->parent = TNULL;
    node->color = RED;
    return node;
}

void leftRotate(struct RBNode** root, struct RBNode* x) {
    struct RBNode* y = x->right;
    x->right = y->left;
    if (y->left != TNULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == TNULL)
        *root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}
```

```
void rightRotate(struct RBNode** root, struct RBNode* x) {
    struct RBNode* y = x->left;
    x->left = y->right;
    if (y->right != TNULL)
        y->right->parent = x;
    y->parent = x->parent;
    if (x->parent == TNULL)
        *root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;
    y->right = x;
    x->parent = y;
}

void fixInsert(struct RBNode** root, struct RBNode* k) {
    struct RBNode* u;
    while (k->parent->color == RED) {
        if (k->parent == k->parent->parent->right) {
            u = k->parent->parent->left;
            if (u->color == RED) {
                u->color = BLACK;
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                k = k->parent->parent;
            } else {
                if (k == k->parent->left) {
                    k = k->parent;
                    rightRotate(root, k);
                }
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                leftRotate(root, k->parent->parent);
            }
        } else {
            u = k->parent->parent->right;
            if (u->color == RED) {
```

```
        } else {
            u = k->parent->parent->right;
            if (u->color == RED) {
                u->color = BLACK;
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                k = k->parent->parent;
            } else {
                if (k == k->parent->right) {
                    k = k->parent;
                    leftRotate(root, k);
                }
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                rightRotate(root, k->parent->parent);
            }
        }
        if (k == *root)
            break;
    }
    (*root)->color = BLACK;
}

void insert(struct RBNode** root, int data) {
    struct RBNode* node = newRBNode(data);
    struct RBNode* y = TNULL;
    struct RBNode* x = *root;

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

    node->parent = y;
    if (y == TNULL)
```

```
    if (y == TNULL)
        *root = node;
    else if (node->data < y->data)
        y->left = node;
    else
        y->right = node;

    if (node->parent == TNULL) {
        node->color = BLACK;
        return;
    }

    if (node->parent->parent == TNULL)
        return;

    fixInsert(root, node);
}

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

int main() {
    int n, data;
    TNULL = (struct RBNode*)malloc(sizeof(struct RBNode));
    TNULL->color = BLACK;
    TNULL->left = TNULL->right = TNULL->parent = NULL;

    struct RBNode* root = TNULL;

    printf("Enter the number of elements to insert in Red-Black Tree: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &data);
        insert(&root, data);
    }

    printf("Inorder Traversal of Red-Black Tree: ");
    inorder(root);
    printf("\n");

    return 0;
}
```

OUPUT:-

```
root@ubuntu:/home/purnisha# nano redblacktree.c
root@ubuntu:/home/purnisha# gcc -o redblacktree redblacktree.c
root@ubuntu:/home/purnisha# ./redblacktree
Enter the number of elements to insert in Red-Black Tree: 12
Enter 12 elements: 157 110 147 122 111 149 151 141 123 112 147 133
Inorder Traversal of Red-Black Tree: 110 111 112 122 123 133 141 147 147 149 151 157
```

TIME COMPLEXITY:- O(n log n)

JUSTIFICATION:-

Insertion of one element in a Red-Black Tree takes **O(log n)** time due to balancing rules.
For **n elements**, total insertion time = **n × log n**.
Inorder traversal visits each node exactly once,

which takes **O(n)** time.
Overall time complexity is **O(n log n)**.

SPACE COMPLEXITY:- O(n)

JUSTIFICATION:-

Each node stores:

- int data → 4 bytes
- color → 4 bytes
- left pointer → 8 bytes
- right pointer → 8 bytes
- parent pointer → 8 bytes

Memory required increases linearly with number of nodes **n**.
Recursive traversal uses stack space proportional to tree height **O(log n)**.
Therefore, space complexity is **O(n)**.