DESIGN AND ANALYSIS OF ALGORITHM

23CSE211-Balancing BST

NAME :B.SIDDHARTHA REDDY

ROLL NO : CH.SC.U4CSE24104

1) **Balancing BST- AVL :**

CODE:

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

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

```c
/* Right Rotation */
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    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;
}

/* Left Rotation */
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;
}
```

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

/* Insert with AVL Balancing */
struct Node *insert(struct Node *node, int key)
{
    if (node == NULL)
        return newNode(key);

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

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

    int balance = getBalance(node);

    /* LL Case */
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    /* RR Case */
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
```

```c
    /* LR Case */
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    /* RL Case */
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

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

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

    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = sizeof(arr)/sizeof(arr[0]);

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

    printf("Inorder traversal of balanced AVL Tree:\n");
    inorder(root);

    return 0;
}
```

OUTPUT:

```
Inorder traversal of balanced AVL Tree:
110 111 112 117 122 123 133 141 147 149 151 157
```

Time Complexity:

For your AVL program inserting
`157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133`:

- **Best Case: O(log n)**
- **Average Case: O(log n)**
- **Worst Case: O(log n)**

Because after every insertion, **AVL rotations** keep the tree height balanced, so searching, inserting, and rotating always take logarithmic time.

Space Complexity:

- **O(n)** — space for storing `n` tree nodes
- **O(log n)** — extra stack space due to recursion (tree height)

So overall:

Space Complexity = O(n)

2) Balancing BST-RedBlack Tree:

CODE:

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

enum Color { RED, BLACK };

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

struct Node* root = NULL;

/* Create new node */
struct Node* newNode(int data) {
    struct Node* n = (struct Node*)malloc(sizeof(struct Node));
    n->data = data;
    n->left = n->right = n->parent = NULL;
    n->color = RED;
    return n;
}

/* Left Rotate */
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;
    x->parent = y;
}

/* Right Rotate */
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;
}
```

```c
/* Right Rotate */
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;
}
```

```c
/* Fix Red-Black Tree */
void fixInsert(struct Node* z) {
    while (z->parent != NULL && 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;
                    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;
}

/* Insert Node */
void insert(int data) {
    struct Node* z = newNode(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);
}
```

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

int main() {
    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = sizeof(arr)/sizeof(arr[0]);

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

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

    return 0;
}
```

OUTPUT:

```
Inorder traversal of Red-Black Tree:
110 111 112 117 122 123 133 141 147 149 151 157
```

Time Complexity:

For your Red-Black Tree program inserting
`157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133`:

- **Best Case: O(log n)**
- **Average Case: O(log n)**
- **Worst Case: O(log n)**

A Red-Black Tree always keeps its height **O(log n)** using **node coloring and rotations**, so search and insertion never become linear.

Space Complexity:

- **O(n)** → memory to store `n` nodes
- **O(log n)** → recursive calls (tree height) during traversal

So overall:

Space Complexity = O(n)