Design Analysis and Algorithm – Lab Work

Week 5

Question 1: Write a program to Illustrate Balancing of BST using AVL Tree

{157,110,147,122,111,149,151,141,123,112,117,133}.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int key;
    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 key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    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;
    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);
}
```

```c
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);
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        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]);
    struct Node* root = NULL;
    for (int i = 0; i < n; i++)
        root = insert(root, arr[i]);
    printf("Inorder Traversal of AVL Tree:\n");
    inorder(root);
    return 0;
}
```

Output:

```
PS C:\Users\rohit_ytwyq5k\OneDrive - Amrita Vishwa Vidyapeetham- Chennai Campus\4th Semester\Design Analysis And Algorithm\Week 5> gcc AVL.c
PS C:\Users\rohit_ytwyq5k\OneDrive - Amrita Vishwa Vidyapeetham- Chennai Campus\4th Semester\Design Analysis And Algorithm\Week 5> ./a
Inorder Traversal of AVL Tree:
110 111 112 117 122 123 133 141 147 149 151 157
```

Time Complexity:

The AVL Tree maintains balance by performing rotations after every insertion.
For each insertion:

- Searching the correct position takes **O(log n)** time because the tree height is log n.
- After insertion, updating heights and checking balance factors is done while returning from recursion, which also takes **O(log n)** time.
- Rotations (LL, RR, LR, RL) take **constant time O(1)**.

Thus, each insertion takes **O(log n)** time.
For inserting *n* elements, the total time required is:
O(n log n)

Space Complexity:

The space occupied by the AVL Tree program mainly depends on the **number of nodes created** and the **recursion depth during insertion**.
Each node stores an integer key, two pointers (left and right), and an integer height. For an input of size *n*, the tree stores *n* nodes, so the basic storage requirement is **O(n)**.
During insertion, the program uses recursion to insert nodes and rebalance the tree. Since an AVL Tree is always height-balanced, its height is **log n**. Therefore, the **maximum recursion depth** during insertion is **log n**.
Hence, the extra memory used due to recursion is **O(log n)**.
Therefore, the Space Complexity of AVL Tree insertion is:

- **O(n)** for node storage
- **O(log n)** extra space due to recursion

Question 2: Write a program to Illustrate Balancing of BST using Red-Black Tree

{157,110,147,122,111,149,151,141,123,112,117,133}.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#define RED 1
#define BLACK 0
struct Node {
    int data;
    int 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;
    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;
                    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) {
    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 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:

```
PS C:\Users\rohit_ytwyq5k\OneDrive - Amrita Vishwa Vidyapeetham- Chennai Campus\4th Semester\Design Analysis And Algorithm\Week 5> gcc Red-Black.c
PS C:\Users\rohit_ytwyq5k\OneDrive - Amrita Vishwa Vidyapeetham- Chennai Campus\4th Semester\Design Analysis And Algorithm\Week 5> ./a
Inorder Traversal of Red-Black Tree:
110 111 112 117 122 123 133 141 147 149 151 157
```

Space Complexity:

The Red-Black Tree program allocates memory dynamically for each node.
Each node contains:

- Data value
- Color (RED or BLACK)
- Three pointers (left, right, parent)

For an input size of *n*, the tree stores *n* nodes, so the total storage required is **O(n)**.
Unlike AVL Trees, Red-Black Trees **do not use recursion for balancing**; balancing is handled using loops and rotations. Therefore, **no recursion stack** is involved during insertion.
Hence, the Space Complexity of Red-Black Tree insertion is:

- **O(n)** for node storage
- **O(1)** extra space (no recursion)

Time Complexity:

For each insertion:

- Finding the correct position in the tree takes **O(log n)** time because the height of a Red-Black Tree is always bounded by **2 log n**.
- Fixing Red-Black property violations involves recoloring and at most **two rotations**, each taking **O(1)** time.

Thus, the total time for a single insertion is **O(log n)**.
For inserting *n* elements, the total time taken is:
O(n log n)