DESIGN AND ANALYSIS OF ALGORITHMS

LAB WORKBOOK

WEEK - 5

NAME : SANTHOSH A

ROLL NUMBER : CH.SC.U4CSE24142

CLASS : CSE-B

1: Construct an AVL tree with these numbers:

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

CODE:

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

/* AVL Tree Node Structure */
struct Node {
    int data;
    struct Node *left, *right;
    int height;
};

/* Utility function to find maximum */
int maxValue(int a, int b) {
    return (a > b) ? a : b;
}

/* Get height of node */
int nodeHeight(struct Node *nodePtr) {
    if (nodePtr == NULL)
        return 0;
    return nodePtr->height;
}

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

/* 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 = maxValue(nodeHeight(y->left), nodeHeight(y->right)) + 1;
    x->height = maxValue(nodeHeight(x->left), nodeHeight(x->right)) + 1;
```

```
42
43      return x;
44  }
45
46  /* Left Rotation */
47  struct Node* leftRotate(struct Node* x) {
48      struct Node* y = x->right;
49      struct Node* T2 = y->left;
50
51      y->left = x;
52      x->right = T2;
53
54      x->height = maxValue(nodeHeight(x->left), nodeHeight(x->right)) + 1;
55      y->height = maxValue(nodeHeight(y->left), nodeHeight(y->right)) + 1;
56
57      return y;
58  }
59
60  /* Get balance factor */
61  int getBalance(struct Node* nodePtr) {
62      if (nodePtr == NULL)
63          return 0;
64      return nodeHeight(nodePtr->left) - nodeHeight(nodePtr->right);
65  }
66
67  /* Insert into AVL Tree */
68  struct Node* insertNode(struct Node* root, int value) {
69      if (root == NULL)
70          return createNewNode(value);
71
72      if (value < root->data)
73          root->left = insertNode(root->left, value);
74      else if (value > root->data)
75          root->right = insertNode(root->right, value);
76      else
77          return root;
78
79      root->height = 1 + maxValue(nodeHeight(root->left), nodeHeight(root->right));
80
81      int balance = getBalance(root);
```

```c

    // LL Case
    if (balance > 1 && value < root->left->data)
        return rightRotate(root);

    // RR Case
    if (balance < -1 && value > root->right->data)
        return leftRotate(root);

    // LR Case
    if (balance > 1 && value > root->left->data) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // RL Case
    if (balance < -1 && value < root->right->data) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

/* Level Order Traversal */
void levelOrder(struct Node* root) {
    if (root == NULL) {
        printf("Tree is empty\n");
        return;
    }

    struct Node* queue[100];
    int front = 0, rear = 0, level = 0;

    queue[rear++] = root;

    printf("\nLevel Order Traversal:\n");

    while (front < rear) {
        int count = rear - front;
```

```c
            printf("Level %d: ", level++);

            for (int i = 0; i < count; i++) {
                struct Node* curr = queue[front++];
                printf("%d ", curr->data);

                if (curr->left != NULL)
                    queue[rear++] = curr->left;
                if (curr->right != NULL)
                    queue[rear++] = curr->right;
            }
            printf("\n");
        }
    }

    /* Main Function */
    int main() {
        struct Node* root = NULL;
        int n, value;

        printf("Enter number of nodes: ");
        scanf("%d", &n);

        printf("Enter %d values:\n", n);
        for (int i = 0; i < n; i++) {
            scanf("%d", &value);
            root = insertNode(root, value);
        }

        levelOrder(root);

        if (getBalance(root) >= -1 && getBalance(root) <= 1)
            printf("\nTree is AVL Balanced\n");
        else
            printf("\nTree is NOT Balanced\n");

        return 0;
    }
```

OUTPUT:

```
Enter number of nodes: 12
Enter 12 values:
157
110
147
122
149
151
111
141
112
123
133
117

Level Order Traversal:
Level 0: 122
Level 1: 111 147
Level 2: 110 112 133 151
Level 3: 117 123 141 149 157

Tree is AVL Balanced
```

Time Complexity for

(i) Search: O(log N) : Tree height is O(log n), search follows one path from root to leaf.

(ii) Insertion: O(log N) : BST insertion O(log n) + at most 2 rotations O(1) = O(log n).

(iii) Deletion: O (log N) : BST deletion O(log n) + rebalancing up to O(log n) rotations = O(log n)

(iv) Traversal: O(N) : Must visit all n nodes exactly once.

(v) Rotation: O(1) : Only changes a constant number of pointers.

Space Complexity: O(N) : Stores all the N nodes along with the data, pointer and height.

2: Construct a Red-Black tree with the numbers:

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

Print the tree showing R (red) or B (black) for each node.

CODE:

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

#define RED 1
#define BLACK 0

struct RBNode {
    int value;
    int shade;
    struct RBNode *lchild, *rchild, *parent;
};

struct RBNode *treeRoot = NULL;

/* Create node */
struct RBNode* createRBNode(int val) {
    struct RBNode* temp = (struct RBNode*)malloc(sizeof(struct RBNode));
    temp->value = val;
    temp->shade = RED;
    temp->lchild = temp->rchild = temp->parent = NULL;
    return temp;
}

/* Left rotate */
void rotateLeftRB(struct RBNode *curr) {
    struct RBNode *child = curr->rchild;
    curr->rchild = child->lchild;

    if (child->lchild != NULL)
        child->lchild->parent = curr;

    child->parent = curr->parent;

    if (curr->parent == NULL)
        treeRoot = child;
    else if (curr == curr->parent->lchild)
        curr->parent->lchild = child;
    else
        curr->parent->rchild = child;

    child->lchild = curr;
```

```
    curr->parent = child;
}

/* Right rotate */
void rotateRightRB(struct RBNode *curr) {
    struct RBNode *child = curr->lchild;
    curr->lchild = child->rchild;

    if (child->rchild != NULL)
        child->rchild->parent = curr;

    child->parent = curr->parent;

    if (curr->parent == NULL)
        treeRoot = child;
    else if (curr == curr->parent->lchild)
        curr->parent->lchild = child;
    else
        curr->parent->rchild = child;

    child->rchild = curr;
    curr->parent = child;
}

/* Fix violations */
void fixRBInsert(struct RBNode *node) {
    while (node != treeRoot && node->parent->shade == RED) {

        if (node->parent == node->parent->parent->lchild) {
            struct RBNode *uncle = node->parent->parent->rchild;

            if (uncle != NULL && uncle->shade == RED) {
                node->parent->shade = BLACK;
                uncle->shade = BLACK;
                node->parent->parent->shade = RED;
                node = node->parent->parent;
            } else {
                if (node == node->parent->rchild) {
                    node = node->parent;
                    rotateLeftRB(node);
```

```
                }
                node->parent->shade = BLACK;
                node->parent->parent->shade = RED;
                rotateRightRB(node->parent->parent);
            }
        } else {
            struct RBNode *uncle = node->parent->parent->lchild;

            if (uncle != NULL && uncle->shade == RED) {
                node->parent->shade = BLACK;
                uncle->shade = BLACK;
                node->parent->parent->shade = RED;
                node = node->parent->parent;
            } else {
                if (node == node->parent->lchild) {
                    node = node->parent;
                    rotateRightRB(node);
                }
                node->parent->shade = BLACK;
                node->parent->parent->shade = RED;
                rotateLeftRB(node->parent->parent);
            }
        }
    }
    treeRoot->shade = BLACK;
}

/* Insert */
void insertRB(int val) {
    struct RBNode *newNode = createRBNode(val);
    struct RBNode *parentPtr = NULL;
    struct RBNode *trav = treeRoot;

    while (trav != NULL) {
        parentPtr = trav;
        if (val < trav->value)
```

```
            trav = trav->lchild;
        else
            trav = trav->rchild;
    }

    newNode->parent = parentPtr;

    if (parentPtr == NULL)
        treeRoot = newNode;
    else if (val < parentPtr->value)
        parentPtr->lchild = newNode;
    else
        parentPtr->rchild = newNode;

    fixRBInsert(newNode);
}

/* Inorder traversal */
void inorderRB(struct RBNode *node) {
    if (node != NULL) {
        inorderRB(node->lchild);
        printf("%d(%s) ", node->value,
               node->shade == RED ? "R" : "B");
        inorderRB(node->rchild);
    }
}

/* Tree structure printing */
void printRBTree(struct RBNode* root, int space) {
    if (root == NULL)
        return;

    space += 10;

    printRBTree(root->rchild, space);

    printf("\n");
    for (int i = 10; i < space; i++)
        printf(" ");
```

```
157
158         printf("%d(%s)", root->value,
159                 root->shade == RED ? "R" : "B");
160
161         printRBTree(root->lchild, space);
162     }
163
164     int main() {
165         int n, val;
166
167         printf("Enter number of nodes: ");
168         scanf("%d", &n);
169
170         printf("Enter %d values:\n", n);
171         for (int i = 0; i < n; i++) {
172             scanf("%d", &val);
173             insertRB(val);
174         }
175
176         printf("\nInorder Traversal:\n");
177         inorderRB(treeRoot);
178
179         printf("\n\nTree Structure:\n");
180         printRBTree(treeRoot, 0);
181
182         return 0;
183     }
```

OUTPUT:

```
Enter number of nodes: 12
Enter 12 values:
157
110
147
122
149
151
111
141
112
123
133
117

Inorder Traversal:
110(B) 111(R) 112(B) 117(R) 122(B) 123(R) 133(B) 141(R) 147(R) 149(R) 151(B) 157(R)

Tree Structure:

                        157(R)
                151(B)
                        149(R)
        147(R)
                        141(R)
                133(B)
                        123(R)
122(B)
                        117(R)
                112(B)
        111(R)
                110(B)
```

Time Complexity for

(i) Search: O(log N) : Red-Black properties guarantee tree height ≤ $2*\log_2(n+1)$, so we traverse at most logarithmic levels from root to leaf.

(ii) Insertion: O(log N) : BST insertion takes O(log n) to find the position, and fixing violations requires at most O(log n) recoloring plus at most 2 rotations (O(1) each).

(iii) Deletion: O (log N) : BST deletion takes O(log n) to find the node, and fixing violations requires at most O(log n) recoloring plus at most 3 rotations (O(1) each).

(iv) Traversal: O(N) : We must visit every node exactly once in inorder /preorder/ postorder traversal, and there are n nodes.

(v) Rotation: O(1) : It only updates a constant number of pointers (parent, left, right) regardless of tree size.

Space Complexity: O(N) : Stores all the N nodes along with the data, pointer and colour information.