AMRITA VISHWA VIDYAPEETHAM

DESIGN AND ANALYSIS OF ALGORITHMS

LAB WORKBOOK

WEEK - 5

NAME : ROHITH S

ROLL NUMBER : CH.SC.U4CSE24140

CLASS : CSE-B

Question 1:

Construct an AVL tree with these numbers:

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

(i) Print the tree showing each level.

(ii) Check that all nodes are balanced.

CODE:

```c
//CH.SC.U4CSE24140 - Rohith S
#include<stdio.h>
#include<stdlib.h>

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

int maximum(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=maximum(height(y->left),height(y->right))+1;
    x->height=maximum(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=maximum(height(x->left),height(x->right))+1;
    y->height=maximum(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 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+maximum(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;
}

int isBalanced(struct Node*root){
    if(root==NULL)
        return 1;

    int bf=getBalance(root);

    if(bf>1||bf<-1)
        return 0;

    return isBalanced(root->left)&&isBalanced(root->right);
}

int treeHeight(struct Node*root){
    if(root==NULL)
        return 0;
    return 1+maximum(treeHeight(root->left),treeHeight(root->right));
}
```

```c
void printLevel(struct Node*root,int level){
    if(root==NULL)
        return;

    if(level==1)
        printf("%d ",root->key);
    else{
        printLevel(root->left,level-1);
        printLevel(root->right,level-1);
    }
}

void printTreeLevels(struct Node*root){
    int h=treeHeight(root);
    for(int i=1;i<=h;i++){
        printf("Level %d: ",i-1);
        printLevel(root,i);
        printf("\n");
    }
}

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

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

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

    printf("AVL Tree Level-wise:\n");
    printTreeLevels(root);

    if(isBalanced(root))
        printf("\nAll nodes are balanced. AVL Tree is valid.\n");
    else
        printf("\nTree is NOT balanced.\n");

    return 0;
}
```

OUTPUT:

```
AVL Tree Level-wise:
Level 0: 122
Level 1: 111 147
Level 2: 110 112 133 151
Level 3: 117 123 141 149 157

All nodes are balanced. AVL Tree is valid.
PS C:\Users\123sr\OneDrive\Desktop\COLLEGE STUFF\SEM IV\DAA LAB\Week 5>
```

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.

Question 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. Check that:

(i) Root is black

(ii) No red node has red children

CODE:

```c
//CH.SC.U4CSE24140 - Rohith S
#include<stdio.h>
#include<stdlib.h>

#define RED 1
#define BLACK 0

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

struct Node*root=NULL;

struct Node*newNode(int data){
    struct Node*node=(struct Node*)malloc(sizeof(struct Node));
    node->data=data;
    node->color=RED;
    node->left=NULL;
    node->right=NULL;
    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=newNode(data);
    struct Node*y=NULL;
    struct Node*x=root;

    while(x!=NULL){
        y=x;
```

```
106             if(z->data<x->data)
107                 x=x->left;
108             else
109                 x=x->right;
110         }
111
112         z->parent=y;
113
114         if(y==NULL)
115             root=z;
116         else if(z->data<y->data)
117             y->left=z;
118         else
119             y->right=z;
120
121         fixInsert(z);
122     }
123
124     int treeHeight(struct Node*node){
125         if(node==NULL)
126             return 0;
127         int lh=treeHeight(node->left);
128         int rh=treeHeight(node->right);
129         return (lh>rh?lh:rh)+1;
130     }
131
132     void printLevel(struct Node*node,int level){
133         if(node==NULL)
134             return;
135         if(level==1)
136             printf("%d(%c) ",node->data,node->color==RED?'R':'B');
137         else{
138             printLevel(node->left,level-1);
139             printLevel(node->right,level-1);
140         }
```

```
}

void printTree(){
    int h=treeHeight(root);
    for(int i=1;i<=h;i++){
        printf("Level %d: ",i-1);
        printLevel(root,i);
        printf("\n");
    }
}

int checkRootBlack(){
    if(root==NULL)
        return 1;
    return root->color==BLACK;
}

int checkNoRedRed(struct Node*node){
    if(node==NULL)
        return 1;

    if(node->color==RED){
        if((node->left!=NULL && node->left->color==RED) ||
           (node->right!=NULL && node->right->color==RED))
            return 0;
    }

    return checkNoRedRed(node->left) && checkNoRedRed(node->right);
}
```

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

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

    printf("Red-Black Tree Level-wise:\n");
    printTree();

    printf("\nCheck Conditions:\n");

    if(checkRootBlack())
        printf("(i) Root is Black: YES\n");
    else
        printf("(i) Root is Black: NO\n");

    if(checkNoRedRed(root))
        printf("(ii) No Red node has Red child: YES\n");
    else
        printf("(ii) No Red node has Red child: NO\n");

    return 0;
}
```

OUTPUT:

```
Red-Black Tree Level-wise:
Level 0: 122(B)
Level 1: 111(R) 147(R)
Level 2: 110(B) 112(B) 133(B) 151(B)
Level 3: 117(R) 123(R) 141(R) 149(R) 157(R)

Check Conditions:
(i) Root is Black: YES
(ii) No Red node has Red child: YES
PS C:\Users\123sr\OneDrive\Desktop\COLLEGE STUFF\SEM IV\DAA LAB\Week 5>
```

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) recolouring 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) recolouring plus at most 3 rotations (O(1) each).

(iv) Traversal: O(N) : We must visit every node exactly once in in order /preorder/ post order 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.