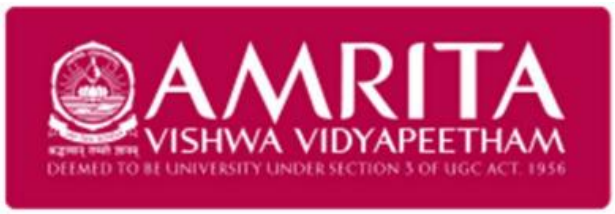

SCHOOL OF COMPUTING

DESIGN AND ANALYSIS OF ALGORITHMS

LAB WORKBOOK

WEEK - 5

NAME : MADDU RAAHITHYA YADAV

ROLL NUMBER : CH.SC.U4CSE24124

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

struct AVLNode {
    int key;
    struct AVLNode *lc, *rc;
    int ht;
};

int maximum(int x, int y) {
    return (x > y) ? x : y;
}

int getHeight(struct AVLNode *ptr) {
    if (ptr == NULL)
        return 0;
    return ptr->ht;
}

struct AVLNode* createNode(int value) {
    struct AVLNode* temp = (struct AVLNode*)malloc(sizeof(struct AVLNode));
    temp->key = value;
    temp->lc = temp->rc = NULL;
    temp->ht = 1;
    return temp;
}

struct AVLNode* rotateRight(struct AVLNode* parent) {
    struct AVLNode* child = parent->lc;
    struct AVLNode* temp = child->rc;

    child->rc = parent;
    parent->lc = temp;

    parent->ht = maximum(getHeight(parent->lc), getHeight(parent->rc)) + 1;
    child->ht = maximum(getHeight(child->lc), getHeight(child->rc)) + 1;
```

```
37
38      return child;
39  }
40
41  struct AVLNode* rotateLeft(struct AVLNode* parent) {
42      struct AVLNode* child = parent->rc;
43      struct AVLNode* temp = child->lc;
44
45      child->lc = parent;
46      parent->rc = temp;
47
48      parent->ht = maximum(getHeight(parent->lc), getHeight(parent->rc)) + 1;
49      child->ht = maximum(getHeight(child->lc), getHeight(child->rc)) + 1;
50
51      return child;
52  }
53
54  int balanceFactor(struct AVLNode* ptr) {
55      if (ptr == NULL)
56          return 0;
57      return getHeight(ptr->lc) - getHeight(ptr->rc);
58  }
59
60  struct AVLNode* insertAVL(struct AVLNode* root, int value) {
61      if (root == NULL)
62          return createNode(value);
63
64      if (value < root->key)
65          root->lc = insertAVL(root->lc, value);
66      else if (value > root->key)
67          root->rc = insertAVL(root->rc, value);
68      else
69          return root;
70
71      root->ht = 1 + maximum(getHeight(root->lc), getHeight(root->rc));
```

```c

    int bf = balanceFactor(root);

    if (bf > 1 && value < root->lc->key)
        return rotateRight(root);

    if (bf < -1 && value > root->rc->key)
        return rotateLeft(root);

    if (bf > 1 && value > root->lc->key) {
        root->lc = rotateLeft(root->lc);
        return rotateRight(root);
    }

    if (bf < -1 && value < root->rc->key) {
        root->rc = rotateRight(root->rc);
        return rotateLeft(root);
    }

    return root;
}

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

    struct AVLNode* q[100];
    int front = 0, rear = 0, level = 0;

    q[rear++] = root;

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

```
107
108         while (front < rear) {
109             int count = rear - front;
110             printf("Level %d: ", level++);
111
112             for (int i = 0; i < count; i++) {
113                 struct AVLNode* curr = q[front++];
114                 printf("%d ", curr->key);
115
116                 if (curr->lc)
117                     q[rear++] = curr->lc;
118                 if (curr->rc)
119                     q[rear++] = curr->rc;
120             }
121             printf("\n");
122         }
123     }
124
125     void printAVLTree(struct AVLNode* root, int space) {
126         if (root == NULL)
127             return;
128
129         space += 10;
130
131         printAVLTree(root->rc, space);
132
133         printf("\n");
134         for (int i = 10; i < space; i++)
135             printf(" ");
136         printf("%d", root->key);
137
138         printAVLTree(root->lc, space);
139     }
140
141     int main() {
```

```c

int main() {
    struct AVLNode* treeRoot = NULL;

    int dataList[] = {157, 110, 147, 122, 149, 151, 111, 141, 112, 123, 133, 117};
    int size = sizeof(dataList) / sizeof(dataList[0]);
    printf("values: ");

    for (int i = 0; i < size; i++) {
        printf("%d ", dataList[i]);
        treeRoot = insertAVL(treeRoot, dataList[i]);
    }

    printf("\n");
    displayLevelOrder(treeRoot);

    printf("\nTree Structure:\n");
    printAVLTree(treeRoot, 0);

    if (balanceFactor(treeRoot) >= -1 && balanceFactor(treeRoot) <= 1)
        printf("\nTree is AVL balanced\n");
    else
        printf("\nTree is NOT balanced\n");

    return 0;
}
```

OUTPUT:

```
PS D:\raahithya\4TH SEM\DAA\week5> gcc AVLtree.c -o results
PS D:\raahithya\4TH SEM\DAA\week5> ./results
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 Structure:

                        157
                151
                        149
        147
                        141
                133
                        123
122
                        117
                112
        111
                110
Tree is AVL balanced
```

Time Complexity for

(i) Search – O(log N)
The tree is balanced, so its height is **log N**. Searching moves from root to leaf along one path.

(ii) Insertion – O(log N)
Insertion follows BST rules in **log N** time. Rotations (if needed) take constant time.

(iii) Deletion – O(log N)
Deletion takes **log N** time, and rebalancing also takes at most **log N** time.

(iv) Traversal – O(N)
All nodes are visited once, so time taken is **N**.

(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
#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 new 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 rotation
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)
```

```
36          curr->parent->lchild = child;
37      else
38          curr->parent->rchild = child;
39
40      child->lchild = curr;
41      curr->parent = child;
42  }
43
44  // Right rotation
45  void rotateRightRB(struct RBNode *curr) {
46      struct RBNode *child = curr->lchild;
47      curr->lchild = child->rchild;
48
49      if (child->rchild != NULL)
50          child->rchild->parent = curr;
51
52      child->parent = curr->parent;
53
54      if (curr->parent == NULL)
55          treeRoot = child;
56      else if (curr == curr->parent->lchild)
57          curr->parent->lchild = child;
58      else
59          curr->parent->rchild = child;
60
61      child->rchild = curr;
62      curr->parent = child;
63  }
64
65  // Fix Red-Black properties
66  void fixRBInsert(struct RBNode *node) {
67      while (node != treeRoot && node->parent->shade == RED) {
68
```

```
69          if (node->parent == node->parent->parent->lchild) {
70              struct RBNode *uncle = node->parent->parent->rchild;
71
72              if (uncle != NULL && uncle->shade == RED) {
73                  node->parent->shade = BLACK;
74                  uncle->shade = BLACK;
75                  node->parent->parent->shade = RED;
76                  node = node->parent->parent;
77              } else {
78                  if (node == node->parent->rchild) {
79                      node = node->parent;
80                      rotateLeftRB(node);
81                  }
82                  node->parent->shade = BLACK;
83                  node->parent->parent->shade = RED;
84                  rotateRightRB(node->parent->parent);
85              }
86          } else {
87              struct RBNode *uncle = node->parent->parent->lchild;
88
89              if (uncle != NULL && uncle->shade == RED) {
90                  node->parent->shade = BLACK;
91                  uncle->shade = BLACK;
92                  node->parent->parent->shade = RED;
93                  node = node->parent->parent;
94              } else {
95                  if (node == node->parent->lchild) {
96                      node = node->parent;
97                      rotateRightRB(node);
98                  }
99                  node->parent->shade = BLACK;
100                 node->parent->parent->shade = RED;
101                 rotateLeftRB(node->parent->parent);
```

```
102                 }
103             }
104         }
105         treeRoot->shade = BLACK;
106     }
107
108     // Insert node
109     void insertRB(int val) {
110         struct RBNode *newNode = createRBNode(val);
111         struct RBNode *parentPtr = NULL;
112         struct RBNode *trav = treeRoot;
113
114         while (trav != NULL) {
115             parentPtr = trav;
116             if (val < trav->value)
117                 trav = trav->lchild;
118             else
119                 trav = trav->rchild;
120         }
121
122         newNode->parent = parentPtr;
123
124         if (parentPtr == NULL)
125             treeRoot = newNode;
126         else if (val < parentPtr->value)
127             parentPtr->lchild = newNode;
128         else
129             parentPtr->rchild = newNode;
130
131         fixRBInsert(newNode);
132     }
133
```

```c
// Height calculation
int treeHeight(struct RBNode* node) {
    if (node == NULL) return 0;
    int lh = treeHeight(node->lchild);
    int rh = treeHeight(node->rchild);
    return (lh > rh ? lh : rh) + 1;
}

// Print specific level
void printLevelRB(struct RBNode* node, int level, int curr) {
    if (node == NULL) {
        if (curr == level) printf("   ");
        return;
    }
    if (curr == level)
        printf("%d(%s) ", node->value, node->shade == RED ? "R" : "B");
    else {
        printLevelRB(node->lchild, level, curr + 1);
        printLevelRB(node->rchild, level, curr + 1);
    }
}

// Level order traversal
void levelOrderRB() {
    int h = treeHeight(treeRoot);
    printf("\nLevel Order Traversal:\n");
    for (int i = 0; i < h; i++) {
        printf("Level %d: ", i);
        printLevelRB(treeRoot, i, 0);
        printf("\n");
    }
}

// Inorder traversal
void inorderRB(struct RBNode *node) {
    if (node != NULL) {
```

```
170             inorderRB(node->lchild);
171             printf("%d(%s) ", node->value,
172                    node->shade == RED ? "R" : "B");
173             inorderRB(node->rchild);
174         }
175     }
176
177     int main() {
178         int n, input;
179
180         printf("Enter number of nodes: ");
181         scanf("%d", &n);
182
183         printf("Enter %d values:\n", n);
184         for (int i = 0; i < n; i++) {
185             scanf("%d", &input);
186             insertRB(input);
187         }
188
189         printf("\nInorder Traversal:\n");
190         inorderRB(treeRoot);
191
192         levelOrderRB();
193
194         return 0;
195     }
196
```

OUTPUT:

```
PS D:\raahithya\4TH SEM\DAA\week5> gcc redblacktree.c -o results
PS D:\raahithya\4TH SEM\DAA\week5> ./results
Enter number of nodes: 12
Enter 12 values:
157
110
147
122
149
151
111
141
147
147
147

147

Inorder Traversal:
Inorder Traversal:
110(B) 111(B) 112(B) 122(B) 123(R) 133(B) 141(R) 147(B) 147(R) 149(B) 151(R) 157(B)
Level Order Traversal:
Level Order Traversal:
Level 0: 122(B)
Level 1: 111(B) 147(B)
Level 2: 110(B) 112(B) 133(B) 151(R)
Level 3:             123(R) 141(R) 149(B) 157(B)
Level 4:             147(R)
```

Time Complexity for

(i) Search – O(log N)
Red-Black Tree is balanced, so its height is proportional to log N. Searching moves from root to leaf.

(ii) Insertion – O(log N)
Finding the position takes log N time. Fixing Red-Black violations takes log N time, with only a few rotations**.**

(iii) Deletion – O(log N)
Deleting a node takes log N time. Rebalancing also takes log N time with limited rotations.

(iv) Traversal – O(N)
Every node is visited once during traversal.

(v) Rotation – O(1)
Rotation changes only a few pointers, so time is constant.

Space Complexity

O(N)
The tree stores N nodes, each containing data, color, and pointers.