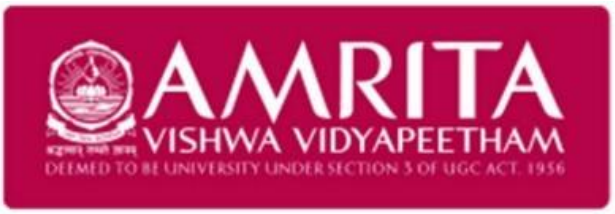

SCHOOL OF COMPUTING

DESIGN AND ANALYSIS OF ALGORITHMS LAB

WORKBOOK

WEEK - 5

NAME : Ganath Avinash G.R

ROLL NUMBER : CH.SC.U4CSE24118

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
//Ganath Avinash CH.SC.U4CSE24118

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

typedef struct Node {
    int data;
    struct Node *left;
    struct Node *right;
    int height;
} Node;

int max(int a, int b) {
    return (a > b) ? a : b;
}

int getHeight(Node *node) {
    if (node == NULL) return 0;
    return node->height;
}

Node *createNode(int data) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    node->height = 1;
    return node;
}

int getBalance(Node *node) {
    if (node == NULL) return 0;
    return getHeight(node->left) - getHeight(node->right);
}

Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *t2 = x->right;

    x->right = y;
    y->left = t2;

    y->height = 1 + max(getHeight(y->left), getHeight(y->right));
    x->height = 1 + max(getHeight(x->left), getHeight(x->right));

    return x;
```

```c
Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *t2 = y->left;

    y->left = x;
    x->right = t2;

    x->height = 1 + max(getHeight(x->left), getHeight(x->right));
    y->height = 1 + max(getHeight(y->left), getHeight(y->right));

    return y;
}

Node *insertAVL(Node *root, int data) {
    if (root == NULL) return createNode(data);

    if (data < root->data)
        root->left = insertAVL(root->left, data);
    else if (data > root->data)
        root->right = insertAVL(root->right, data);
    else
        return root;

    root->height = 1 + max(getHeight(root->left), getHeight(root->right));
    int balance = getBalance(root);

    if (balance > 1 && data < root->left->data)
        return rightRotate(root);

    if (balance < -1 && data > root->right->data)
        return leftRotate(root);

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

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

    return root;
}
```

```c
void levelOrder(Node *root) {
    if (root == NULL) {
        printf("Tree is empty\n");
        return;
    }

    Node *queue[200];
    int front = 0, rear = 0;

    queue[rear++] = root;
    int level = 0;

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

    while (front < rear) {
        int levelSize = rear - front;
        printf("Level %d: ", level++);

        for (int i = 0; i < levelSize; i++) {
            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");
    }
}

int isAVL(Node *root) {
    if (root == NULL) return 1;
    int bal = getBalance(root);
    if (bal < -1 || bal > 1) return 0;
    return isAVL(root->left) && isAVL(root->right);
}
```

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

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

    printf("Inserted values: ");

    for (int i = 0; i < n; i++) {
        printf("%d ", values[i]);
        root = insertAVL(root, values[i]);
    }

    printf("\n");
    levelOrder(root);

    if (isAVL(root))
        printf("\nTree is AVL Balanced\n");
    else
        printf("\nTree is NOT AVL Balanced\n");

    return 0;
}
```

OUTPUT:

```
PS C:\Users\Ganath Avinash\OneDrive\ドキュメント\Back-end\DAA> cd
AVL_QN.c -o AVL_QN } ; if ($?) { .\AVL_QN }
Inserted 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
PS C:\Users\Ganath Avinash\OneDrive\ドキュメント\Back-end\DAA>
```

Time Complexity for

1. **Search: O(log N) :** Tree height is O(log n), search follows one path from root to leaf.
2. **Insertion: O(log N) :** BST insertion O(log n) + at most 2 rotations O(1) = O(log n).
3. **Traversal: O(N) :** Must visit all n nodes exactly once.
4. **Rotation: O(1) :** Only changes constant number of pointers. (Didn't do Deletion)

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
//Ganath Avinash CH.SC.U4CSE24118

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

#define RED 1
#define BLACK 0

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

Node *root = NULL;

Node *createNode(int data) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->data = data;
    node->color = RED;
    node->left = node->right = node->parent = NULL;
    return node;
}
```

```
void leftRotate(Node *x) {
    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(Node *y) {
    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(Node *z) {
    while (z != root && z->parent != NULL && z->parent->color == RED) {

        if (z->parent == z->parent->parent->left) {
            Node *uncle = z->parent->parent->right;

            if (uncle != NULL && uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->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 {
            Node *uncle = z->parent->parent->left;

            if (uncle != NULL && uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->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;
}
```

```c
void insert(int data) {
    Node *z = createNode(data);
    Node *parent = NULL;
    Node *curr = root;

    while (curr != NULL) {
        parent = curr;
        if (z->data < curr->data)
            curr = curr->left;
        else
            curr = curr->right;
    }

    z->parent = parent;

    if (parent == NULL)
        root = z;
    else if (z->data < parent->data)
        parent->left = z;
    else
        parent->right = z;

    fixInsert(z);
}

int height(Node *node) {
    if (node == NULL) return 0;
    int hl = height(node->left);
    int hr = height(node->right);
    return (hl > hr ? hl : hr) + 1;
}

void printLevel(Node *node, int targetLevel, int currLevel) {
    if (node == NULL) {
        if (currLevel == targetLevel) printf("   ");
        return;
    }

    if (currLevel == targetLevel) {
        printf("%d(%c) ", node->data, node->color == RED ? 'R' : 'B');
        return;
    }

    printLevel(node->left, targetLevel, currLevel + 1);
    printLevel(node->right, targetLevel, currLevel + 1);
```

```c
void levelOrder() {
    int h = height(root);
    printf("\nLevel Order Traversal:\n");

    for (int i = 0; i < h; i++) {
        printf("Level %d: ", i);
        printLevel(root, i, 0);
        printf("\n");
    }
}

void inorder(Node *node) {
    if (node == NULL) return;

    inorder(node->left);
    printf("%d(%c) ", node->data, node->color == RED ? 'R' : 'B');
    inorder(node->right);
}

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

    printf("Red-Black Tree Insertion\n");
    printf("Inserting values: ");

    for (int i = 0; i < n; i++) {
        printf("%d ", keys[i]);
        insert(keys[i]);
    }

    printf("\n\nInorder Traversal:\n");
    inorder(root);

    printf("\n");
    levelOrder();

    return 0;
}
```

OUTPUT:

```
PS C:\Users\Ganath Avinash\OneDrive\ドキュメント\Back-end\DAA> cd "c:\Users\Ganath A
\RedBlack }
Red-Black Tree Insertion
Inserting values: 157 110 147 122 111 149 151 141 123 112 117 133

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

Level Order Traversal:
Level 0: 123(B)
Level 1: 111(R) 147(R)
Level 2: 110(B) 117(B) 141(B) 151(B)
Level 3:       112(R) 122(R) 133(R)    149(R) 157(R)
PS C:\Users\Ganath Avinash\OneDrive\ドキュメント\Back-end\DAA>
```

Time Complexity for

1. **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.
2. **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).
3. **Traversal: O(N) :** We must visit every node exactly once in inorder /preorder/ postorder traversal, and there are n nodes.
4. **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.