AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY UNDER SECTION 3 OF UGC ACT, 1956
SCHOOL OF COMPUTING

DESIGN AND ANALYSIS OF ALGORITHMS

LAB WORKBOOK

WEEK - 5

NAME	: KURRA VENKATA GOKUL
ROLL NUMBER	: CH.SC.U4CSE24121
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.

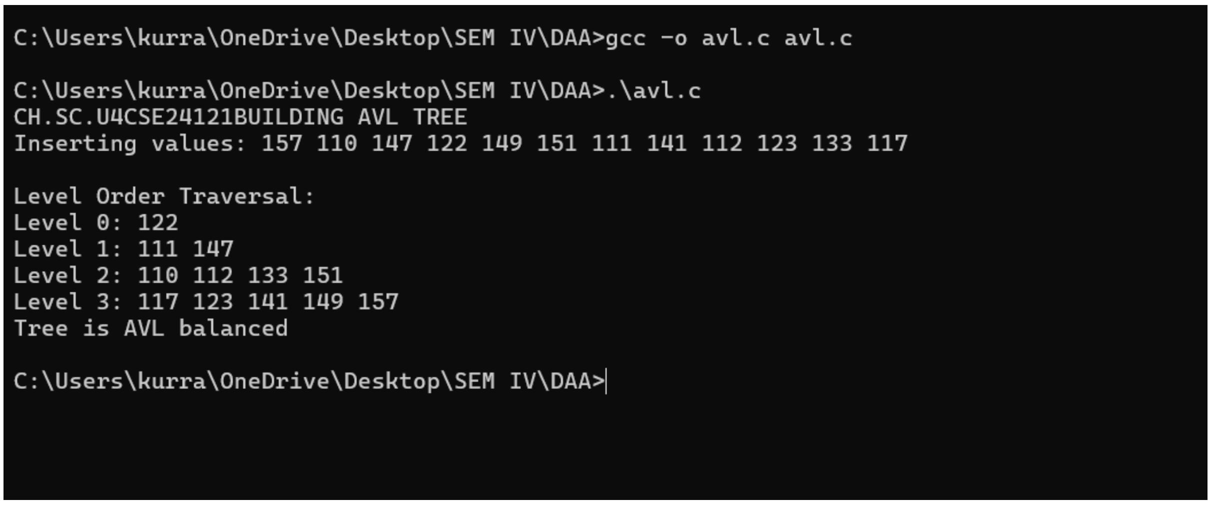
CODE:

```c
// AVL Tree Implementation in C
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *left, *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 data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = 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);
}
struct Node* avlInsert(struct Node* node, int data) {
    if (node == NULL)
        return newNode(data);
    if (data < node->data)
        node->left = avlInsert(node->left, data);
    else if (data > node->data)
        node->right = avlInsert(node->right, data);
    else
        return node;
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);
    // Left Left Case
    if (balance > 1 && data < node->left->data)
        return rightRotate(node);
    // Right Right Case
    if (balance < -1 && data > node->right->data)
        return leftRotate(node);
    // Left Right Case
    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left Case
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
void levelOrder(struct Node* root) {
    if (root == NULL) {
        printf("Tree is empty\n");
        return;
    }
    struct Node* queue[100];
    int front = 0, rear = 0;
    int currentLevel = 0;
    queue[rear++] = root;
    printf("\nLevel Order Traversal:\n");
    while (front < rear) {
        int levelSize = rear - front;
        printf("Level %d: ", currentLevel++);
        for (int i = 0; i < levelSize; i++) {
            struct Node* current = queue[front++];
            printf("%d ", current->data);
            if (current->left != NULL)
                queue[rear++] = current->left;
            if (current->right != NULL)
                queue[rear++] = current->right;
        }
        printf("\n");
    }
}
void printTree(struct Node* root, int space) {
    if (root == NULL)
        return;
    space += 10;
    printTree(root->right, space);
    printf("\n");
    for (int i = 10; i < space; i++)
        printf(" ");
    printf("%d\n", root->data);
    printTree(root->left, space);
}
int main() {
    printf("CH.SC.U4CSE24121");
    struct 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("BUILDING AVL TREE     \n");
    printf("Inserting values: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", values[i]);
        root = avlInsert(root, values[i]);
    }
    printf("\n");
    levelOrder(root);
    if (getBalance(root) >= -1 && getBalance(root) <= 1)
        printf("Tree is AVL balanced\n");
    else
```

```
        printf("Tree is NOT balanced\n");

    return 0;
}
```

OUTPUT:



```
C:\Users\kurra\OneDrive\Desktop\SEM IV\DAA>gcc -o avl.c avl.c

C:\Users\kurra\OneDrive\Desktop\SEM IV\DAA>.\avl.c
CH.SC.U4CSE24121BUILDING AVL TREE
Inserting 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

C:\Users\kurra\OneDrive\Desktop\SEM IV\DAA>
```

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:

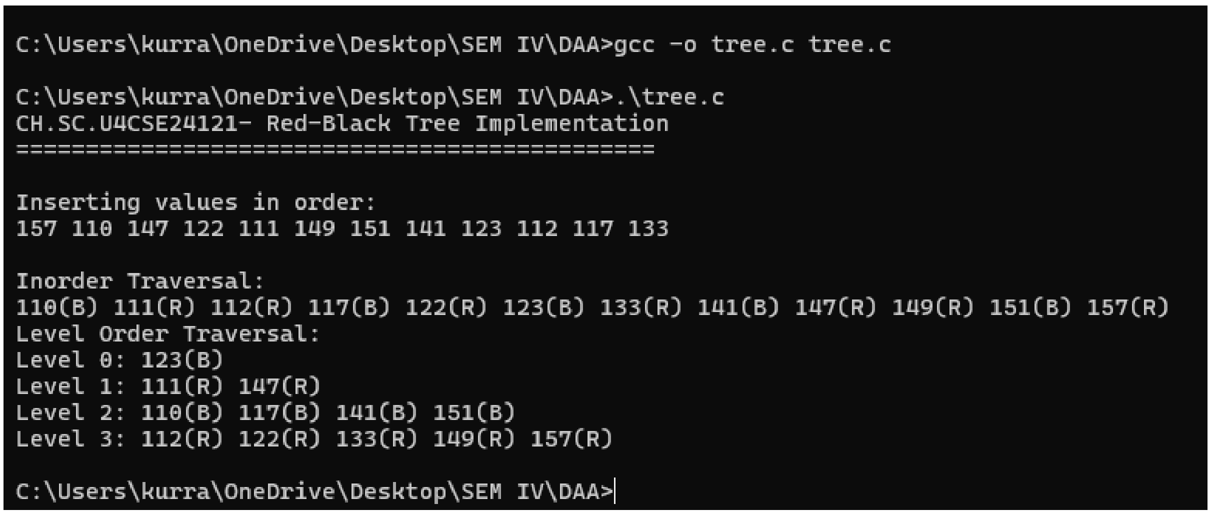
```
//Red-Black Tree Implementation in C
```

```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; // New node is always 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; // Uncle
            // Case 1: Uncle is RED
            if (y != NULL && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            }
            // Case 2 & 3
            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);
}
int height(struct Node* node) {
    if (node == NULL) return 0;
    int leftHeight = height(node->left);
    int rightHeight = height(node->right);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}
void printLevel(struct Node* node, int level, int currentLevel) {
    if (node == NULL) {
        if (currentLevel == level) printf("   ");
        return;
    }
    if (currentLevel == level) {
        printf("%d(%s) ", node->data, node->color == RED ? "R" : "B");
    } else if (currentLevel < level) {
        printLevel(node->left, level, currentLevel + 1);
        printLevel(node->right, level, currentLevel + 1);
    }
}
void levelOrder() {
    int h = height(root);
    printf("\n\nLevel Order Traversal:\n");
    for (int i = 0; i < h; i++) {
        printf("Level %d: ", i);
        printLevel(root, i, 0);
        printf("\n");
    }
```

```c
}
void printTree(struct Node* root, int space, char* prefix) {
    if (root == NULL) return;
    space += 10;
    printTree(root->right, space, "/");
    printf("\n");
    for (int i = 10; i < space; i++) printf(" ");
    printf("%s", prefix);
    printf("%d(%s)\n", root->data, root->color == RED ? "R" : "B");
    printTree(root->left, space, "\\");
}
void inorder(struct Node *node) {
    if (node != NULL) {
        inorder(node->left);
        printf("%d(%s) ", node->data,
               node->color == RED ? "R" : "B");
        inorder(node->right);
    }
}
int main() {
    printf("CH.SC.U4CSE24121 - Red-Black Tree Implementation\n");
    printf("================================================\n\n");
    int keys[] = {157, 110, 147, 122, 111, 149, 151, 141, 123, 112, 117, 133};
    int n = sizeof(keys) / sizeof(keys[0]);
    printf("Inserting values in order:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", keys[i]);
        insert(keys[i]);
    }
    printf("\n\nInorder Traversal:\n");
    inorder(root);
    levelOrder();
    printf("\nTree Structure:\n");
    printf("===============\n");
    printTree(root, 0, "");

    return 0;
}
```

OUTPUT:



```
C:\Users\kurra\OneDrive\Desktop\SEM IV\DAA>gcc -o tree.c tree.c

C:\Users\kurra\OneDrive\Desktop\SEM IV\DAA>.\tree.c
CH.SC.U4CSE24121- Red-Black Tree Implementation
==============================================

Inserting values in order:
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)

C:\Users\kurra\OneDrive\Desktop\SEM IV\DAA>
```

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.