DESIGN AND ANALYSIS OF ALGORITHMS

LAB WORKBOOK WEEK – 5

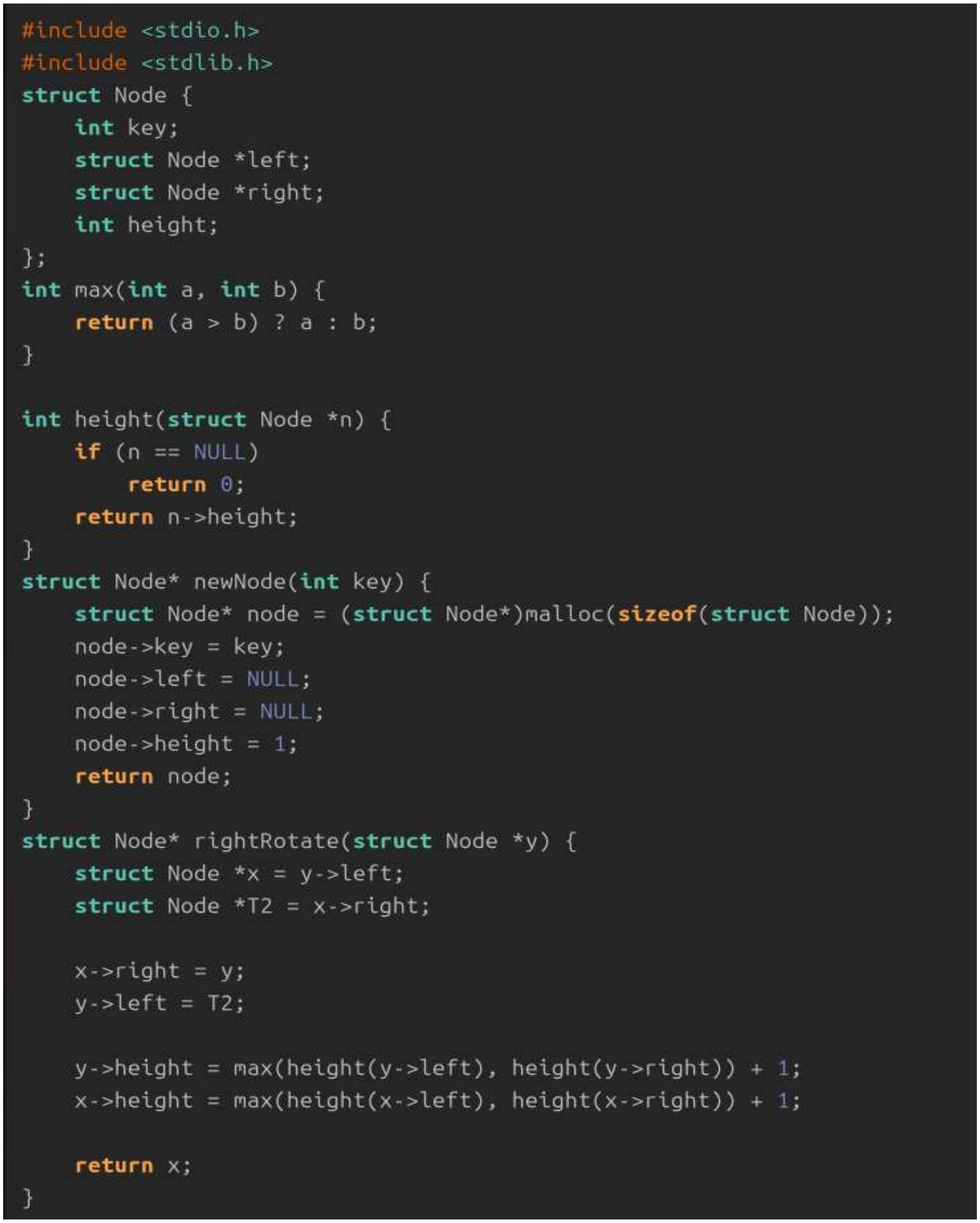
NAME: Konda Bhavani

ROLL NUMBER: CH.SC.U4CSE24120

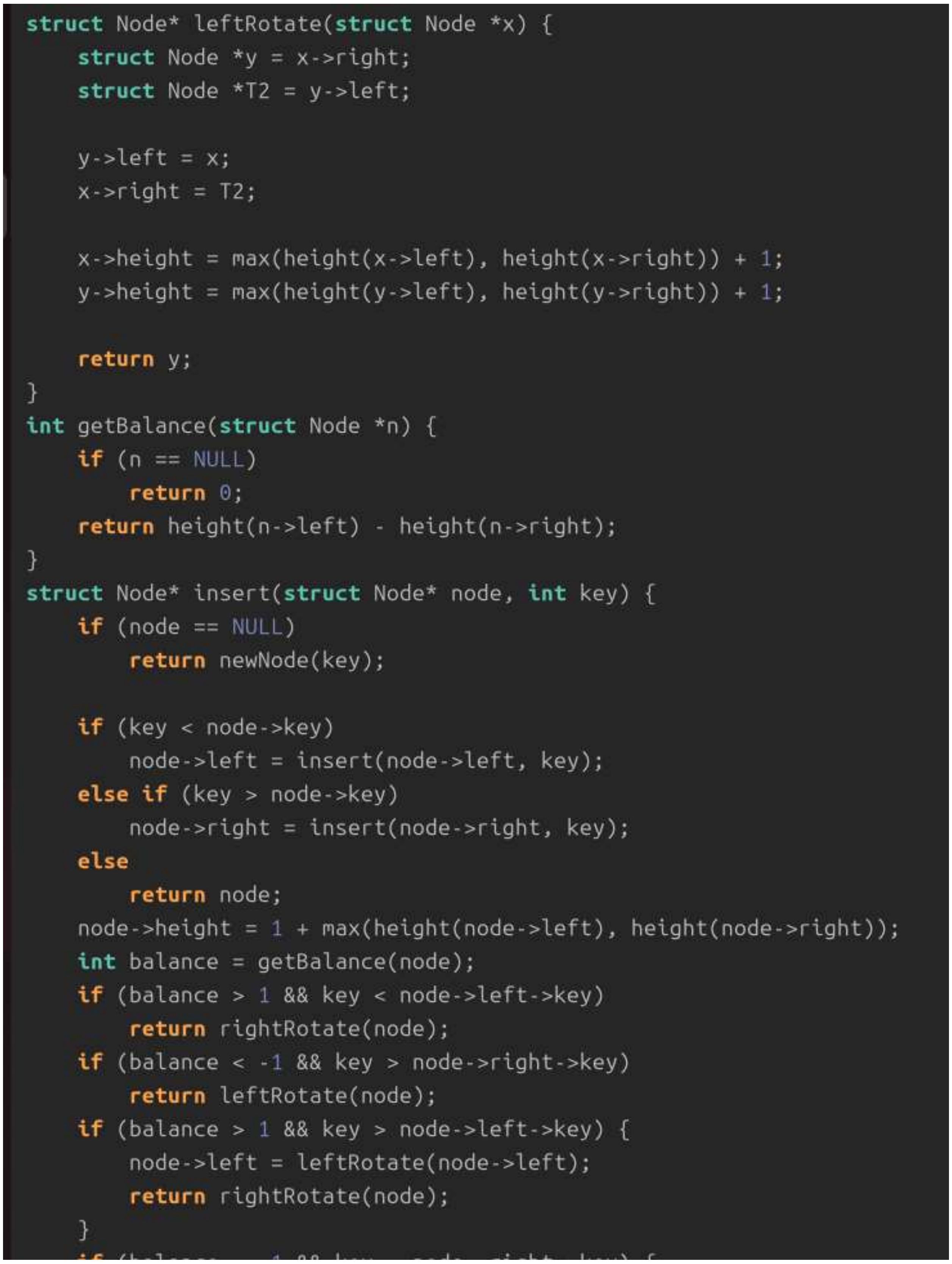
CLASS: CSE-B

AVL Tree

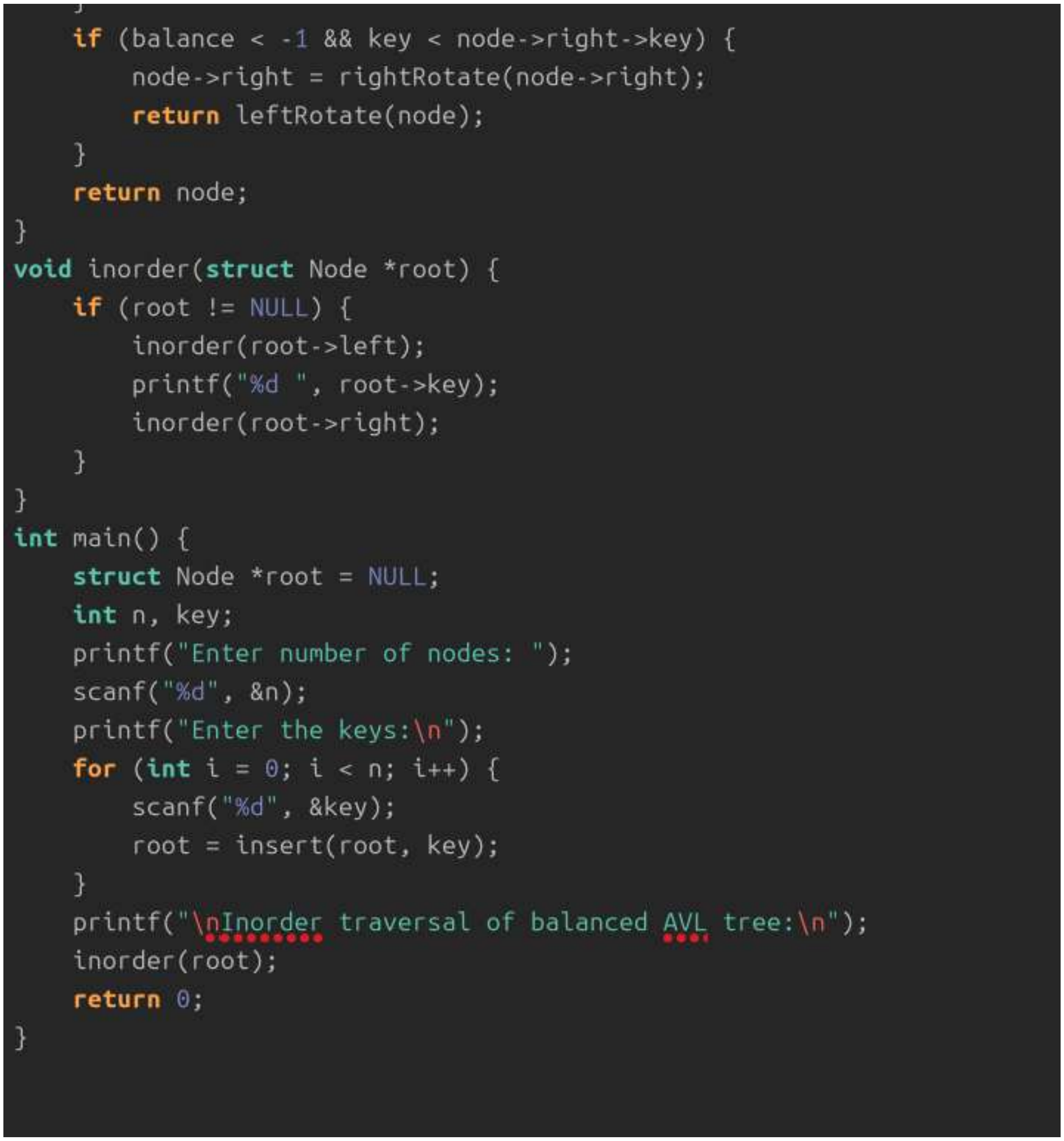
Code:



```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int key;
    struct Node *left;
    struct Node *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 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 = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}
```

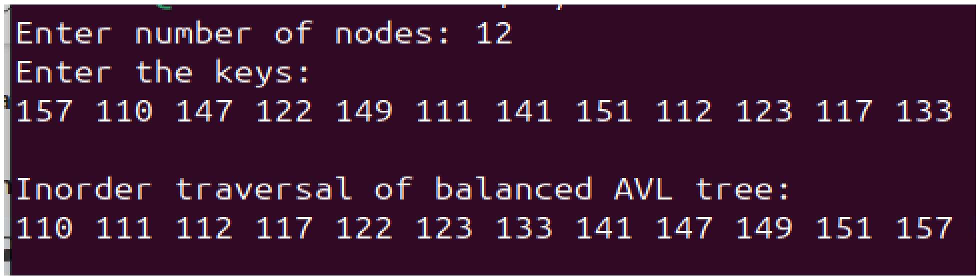


```c
struct Node* leftRotate(struct Node *x) {
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    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* 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 + max(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);
    }
```



```c
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
void inorder(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
int main() {
    struct Node *root = NULL;
    int n, key;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter the keys:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &key);
        root = insert(root, key);
    }
    printf("\nInorder traversal of balanced AVL tree:\n");
    inorder(root);
    return 0;
}
```

Output:



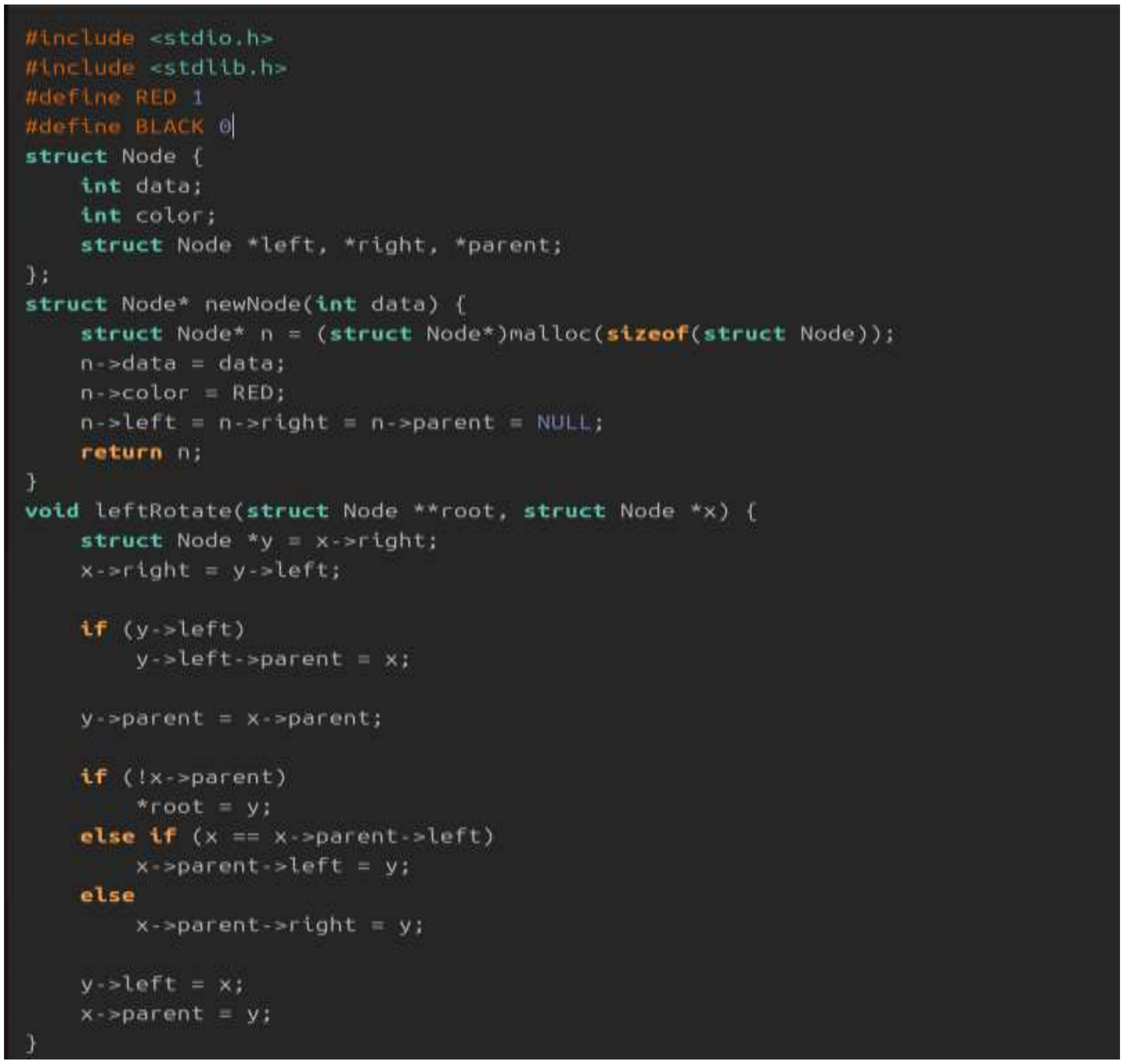
```
Enter number of nodes: 12
Enter the keys:
157 110 147 122 149 111 141 151 112 123 117 133

Inorder traversal of balanced AVL tree:
110 111 112 117 122 123 133 141 147 149 151 157
```

Time Complexity:
Recursive traversal follows the height of the tree.
Tree height is maintained as logarithmic due to strict balancing.
Operations (search/insert/delete) traverse at most log n levels.
Time Complexity = O(log n)

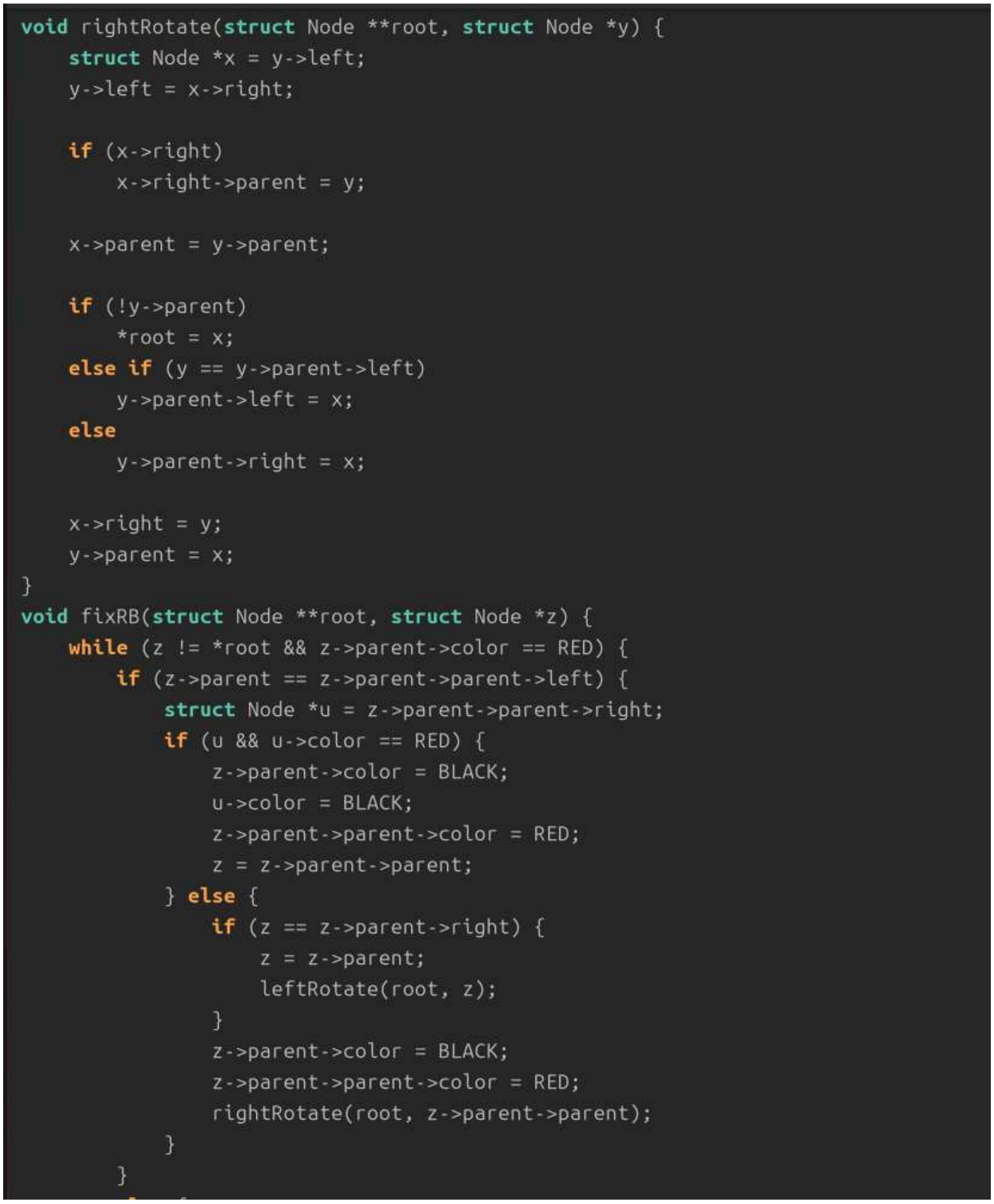
Space Complexity:
Recursive calls are used during insertion and deletion.
Recursion depth depends on height of the tree.
Height of AVL tree is log n.
Space Complexity = O(log n)

Red-Black Tree:

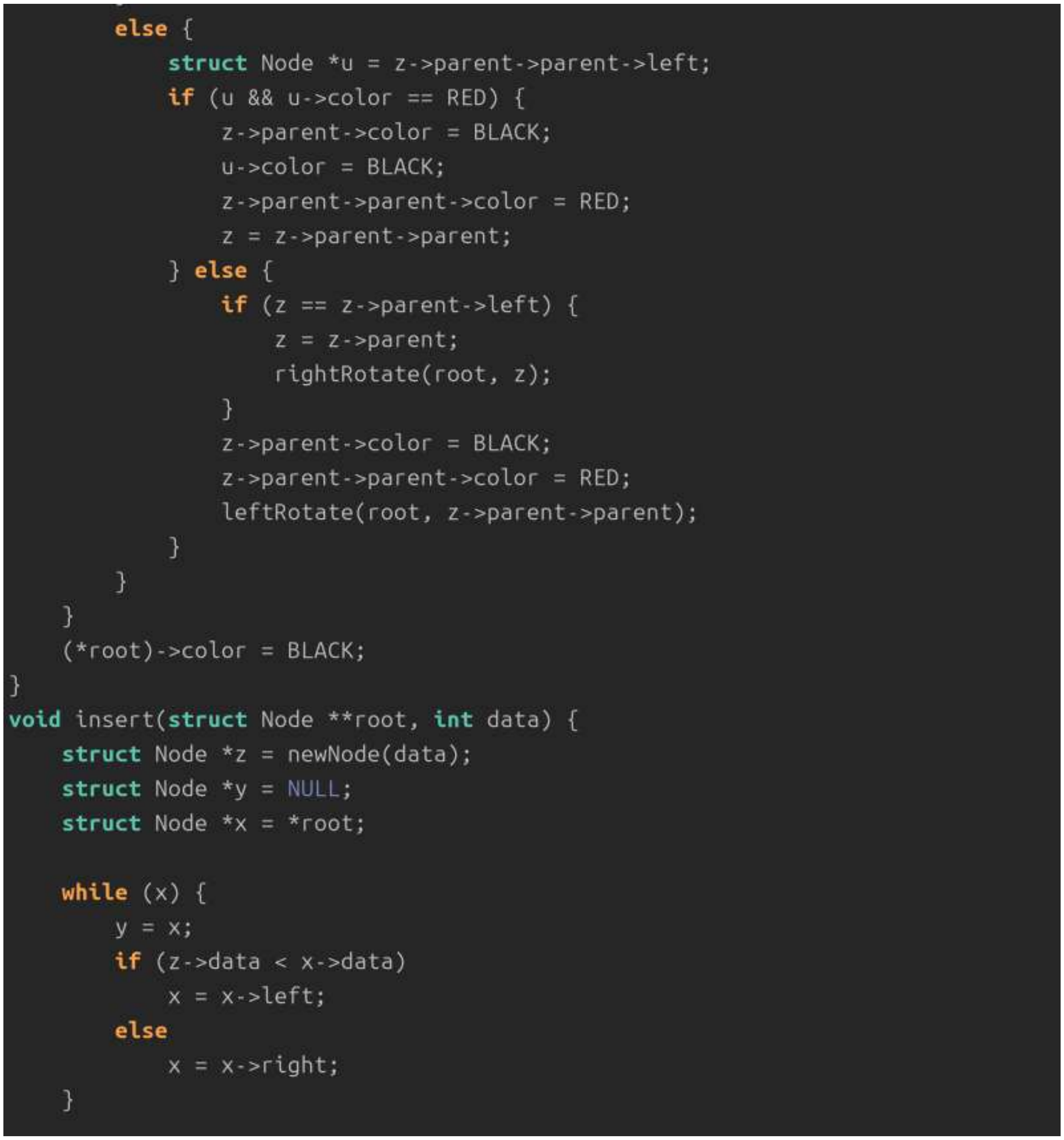
Code:



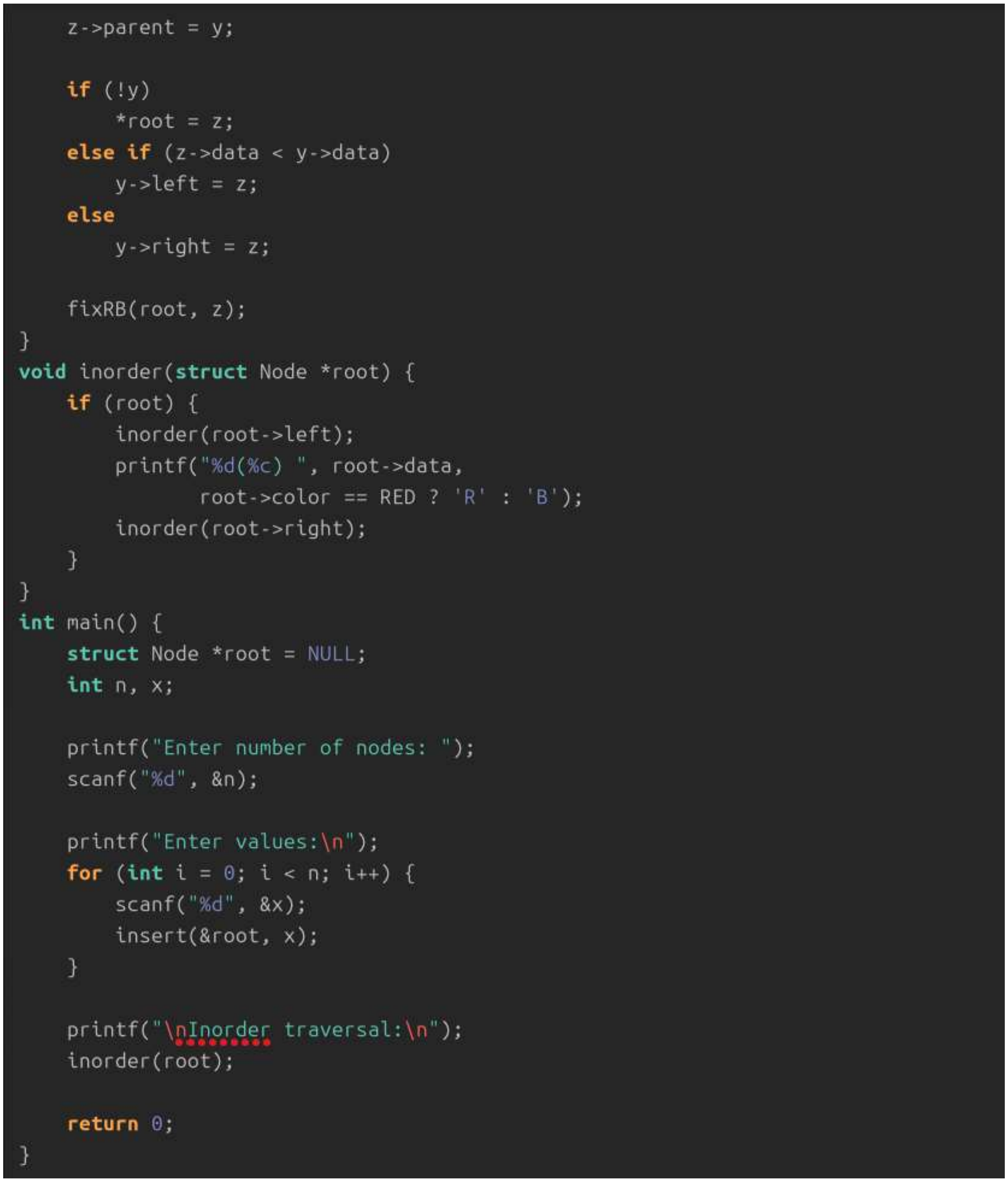
```c
#include <stdio.h>
#include <stdlib.h>
#define RED 1
#define BLACK 0
struct Node {
    int data;
    int color;
    struct Node *left, *right, *parent;
};
struct Node* newNode(int data) {
    struct Node* n = (struct Node*)malloc(sizeof(struct Node));
    n->data = data;
    n->color = RED;
    n->left = n->right = n->parent = NULL;
    return n;
}
void leftRotate(struct Node **root, struct Node *x) {
    struct Node *y = x->right;
    x->right = y->left;

    if (y->left)
        y->left->parent = x;

    y->parent = x->parent;

    if (!x->parent)
        *root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;

    y->left = x;
    x->parent = y;
}
```



```c
void rightRotate(struct Node **root, struct Node *y) {
    struct Node *x = y->left;
    y->left = x->right;

    if (x->right)
        x->right->parent = y;

    x->parent = y->parent;

    if (!y->parent)
        *root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;

    x->right = y;
    y->parent = x;
}
void fixRB(struct Node **root, struct Node *z) {
    while (z != *root && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            struct Node *u = z->parent->parent->right;
            if (u && u->color == RED) {
                z->parent->color = BLACK;
                u->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(root, z->parent->parent);
            }
        }
```

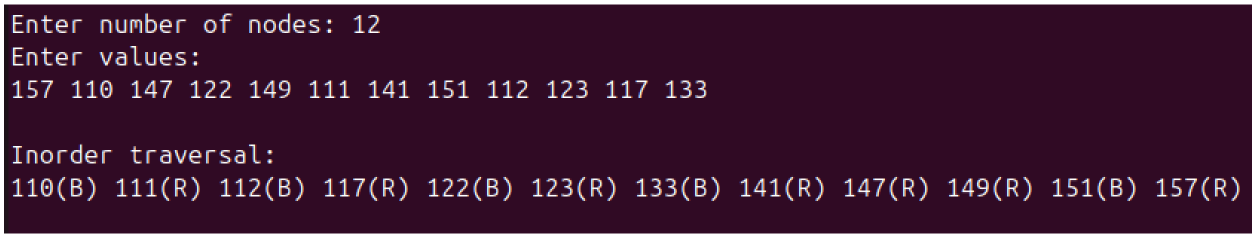


```c
        else {
            struct Node *u = z->parent->parent->left;
            if (u && u->color == RED) {
                z->parent->color = BLACK;
                u->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(root, z->parent->parent);
            }
        }
    }
    (*root)->color = BLACK;
}
void insert(struct Node **root, int data) {
    struct Node *z = newNode(data);
    struct Node *y = NULL;
    struct Node *x = *root;

    while (x) {
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }
```



```c
    z->parent = y;

    if (!y)
        *root = z;
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;

    fixRB(root, z);
}
void inorder(struct Node *root) {
    if (root) {
        inorder(root->left);
        printf("%d(%c) ", root->data,
               root->color == RED ? 'R' : 'B');
        inorder(root->right);
    }
}
int main() {
    struct Node *root = NULL;
    int n, x;

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

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

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

    return 0;
}
```

Output:



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

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)
```

Time Complexity:

Operations traverse the height of the tree.
Tree height is kept logarithmic using color properties.
Search, insertion, and deletion take at most log n steps.
Time Complexity = O(log n)

Space Complexity:

Recursive calls or iterative traversal use stack space.
Recursion depth depends on height of the tree.
Height of Red–Black tree is log n.
Space Complexity = O(log n)