**ASSIGNMENT-6 (DSA)**

**Sneha Kondawar**

1. #include <stdio.h>

#include <stdlib.h>

typedef enum Color {

    RED,

    BLACK

} Color;

typedef struct Node {

    char key;

    Color color;

    struct Node \*left, \*right, \*parent;

} Node;

Node \*root = NULL;

Node\* createNode(char key) {

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    newNode->key = key;

    newNode->left = newNode->right = newNode->parent = NULL;

    newNode->color = RED; // New nodes are always red

    return newNode;

}

void rotateLeft(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 rotateRight(Node \*x) {

    Node \*y = x->left;

    x->left = y->right;

    if (y->right != NULL) {

        y->right->parent = x;

    }

    y->parent = x->parent;

    if (x->parent == NULL) {

        root = y;

    } else if (x == x->parent->right) {

        x->parent->right = y;

    } else {

        x->parent->left = y;

    }

    y->right = x;

    x->parent = y;

}

void insertFixUp(Node \*z) {

    while (z->parent != NULL && z->parent->color == RED) {

        if (z->parent == z->parent->parent->left) {

            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;

                    rotateLeft(z);

                }

                z->parent->color = BLACK;

                z->parent->parent->color = RED;

                rotateRight(z->parent->parent);

            }

        } else {

            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;

                    rotateRight(z);

                }

                z->parent->color = BLACK;

                z->parent->parent->color = RED;

                rotateLeft(z->parent->parent);

            }

        }

    }

    root->color = BLACK;

}

void insert(char key) {

    Node \*y = NULL;

    Node \*x = root;

    while (x != NULL) {

        y = x;

        if (key < x->key) {

            x = x->left;

        } else {

            x = x->right;

        }

    }

    Node \*z = createNode(key);

    z->parent = y;

    if (y == NULL) {

        root = z;

    } else if (z->key < y->key) {

        y->left = z;

    } else {

        y->right = z;

    }

    insertFixUp(z);

}

void inorderTraversal(Node \*node) {

    if (node != NULL) {

        inorderTraversal(node->left);

        printf("%c ", node->key);

        inorderTraversal(node->right);

    }

}

int main() {

    insert('F');

    insert('S');

    insert('Q');

    insert('K');

    insert('C');

    insert('L');

    insert('H');

    insert('T');

    insert('V');

    insert('W');

    insert('M');

    insert('R');

    insert('N');

    insert('P');

    insert('A');

    insert('B');

    insert('X');

    insert('Y');

    insert('D');

    insert('Z');

    insert('E');

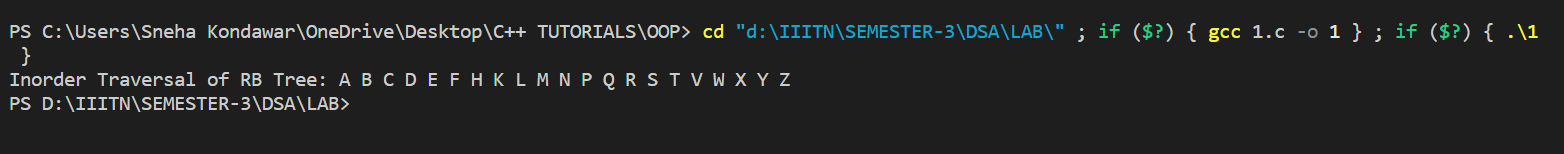
    printf("Inorder Traversal of RB Tree: ");

    inorderTraversal(root);

    printf("\n");

    return 0;

}



2. #include <stdio.h>

#include <stdlib.h>

// ... (definitions for Red-Black Tree structure and functions from the previous example)

typedef enum Color {

    RED,

    BLACK

} Color;

typedef struct Node {

    char key;

    Color color;

    struct Node \*left, \*right, \*parent;

} Node;

Node \*root = NULL;

Node\* createNode(char key) {

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    newNode->key = key;

    newNode->left = newNode->right = newNode->parent = NULL;

    newNode->color = RED; // New nodes are always red

    return newNode;

}

void rotateLeft(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 rotateRight(Node \*x) {

    Node \*y = x->left;

    x->left = y->right;

    if (y->right != NULL) {

        y->right->parent = x;

    }

    y->parent = x->parent;

    if (x->parent == NULL) {

        root = y;

    } else if (x == x->parent->right) {

        x->parent->right = y;

    } else {

        x->parent->left = y;

    }

    y->right = x;

    x->parent = y;

}

void insertFixUp(Node \*z) {

    while (z->parent != NULL && z->parent->color == RED) {

        if (z->parent == z->parent->parent->left) {

            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;

                    rotateLeft(z);

                }

                z->parent->color = BLACK;

                z->parent->parent->color = RED;

                rotateRight(z->parent->parent);

            }

        } else {

            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;

                    rotateRight(z);

                }

                z->parent->color = BLACK;

                z->parent->parent->color = RED;

                rotateLeft(z->parent->parent);

            }

        }

    }

    root->color = BLACK;

}

void insert(char key) {

    Node \*y = NULL;

    Node \*x = root;

    while (x != NULL) {

        y = x;

        if (key < x->key) {

            x = x->left;

        } else {

            x = x->right;

        }

    }

    Node \*z = createNode(key);

    z->parent = y;

    if (y == NULL) {

        root = z;

    } else if (z->key < y->key) {

        y->left = z;

    } else {

        y->right = z;

    }

    insertFixUp(z);

}

void inorderTraversal(Node \*node) {

    if (node != NULL) {

        inorderTraversal(node->left);

        printf("%c ", node->key);

        inorderTraversal(node->right);

    }

}

// Helper function to find the minimum node in a Red-Black Tree

Node \*minimum(Node \*x) {

    while (x->left != NULL) {

        x = x->left;

    }

    return x;

}

// Helper function to replace a node with another node

void transplant(Node \*u, Node \*v) {

    if (u->parent == NULL) {

        root = v;

    } else if (u == u->parent->left) {

        u->parent->left = v;

    } else {

        u->parent->right = v;

    }

    if (v != NULL) {

        v->parent = u->parent;

    }

}

// Helper function for deletion fix-up

void deleteFixUp(Node \*x) {

    while (x != root && (x == NULL || x->color == BLACK)) {

        if (x == x->parent->left) {

            Node \*w = x->parent->right;

            if (w != NULL && w->color == RED) {

                w->color = BLACK;

                x->parent->color = RED;

                rotateLeft(x->parent);

                w = x->parent->right;

            }

            if ((w->left == NULL || w->left->color == BLACK) && (w->right == NULL || w->right->color == BLACK)) {

                w->color = RED;

                x = x->parent;

            } else {

                if (w->right == NULL || w->right->color == BLACK) {

                    if (w->left != NULL) {

                        w->left->color = BLACK;

                    }

                    w->color = RED;

                    rotateRight(w);

                    w = x->parent->right;

                }

                w->color = x->parent->color;

                x->parent->color = BLACK;

                if (w->right != NULL) {

                    w->right->color = BLACK;

                }

                rotateLeft(x->parent);

                x = root;

            }

        } else {

            Node \*w = x->parent->left;

            if (w != NULL && w->color == RED) {

                w->color = BLACK;

                x->parent->color = RED;

                rotateRight(x->parent);

                w = x->parent->left;

            }

            if ((w->right == NULL || w->right->color == BLACK) && (w->left == NULL || w->left->color == BLACK)) {

                w->color = RED;

                x = x->parent;

            } else {

                if (w->left == NULL || w->left->color == BLACK) {

                    if (w->right != NULL) {

                        w->right->color = BLACK;

                    }

                    w->color = RED;

                    rotateLeft(w);

                    w = x->parent->left;

                }

                w->color = x->parent->color;

                x->parent->color = BLACK;

                if (w->left != NULL) {

                    w->left->color = BLACK;

                }

                rotateRight(x->parent);

                x = root;

            }

        }

    }

    if (x != NULL) {

        x->color = BLACK;

    }

}

// Delete a node from the Red-Black Tree

void deleteNode(char key) {

    Node \*z = root;

    while (z != NULL) {

        if (key == z->key) {

            break;

        }

        if (key < z->key) {

            z = z->left;

        } else {

            z = z->right;

        }

    }

    if (z == NULL) {

        printf("Node with key %c not found.\n", key);

        return;

    }

    Node \*y = z;

    Color yOriginalColor = y->color;

    Node \*x;

    if (z->left == NULL) {

        x = z->right;

        transplant(z, z->right);

    } else if (z->right == NULL) {

        x = z->left;

        transplant(z, z->left);

    } else {

        y = minimum(z->right);

        yOriginalColor = y->color;

        x = y->right;

        if (y->parent == z) {

            if (x != NULL) {

                x->parent = y;

            }

        } else {

            transplant(y, y->right);

            y->right = z->right;

            y->right->parent = y;

        }

        transplant(z, y);

        y->left = z->left;

        y->left->parent = y;

        y->color = z->color;

    }

    free(z);

    if (yOriginalColor == BLACK) {

        deleteFixUp(x);

    }

}

int main() {

    // Create and insert nodes into the Red-Black Tree as shown in the previous example

    // ... (insertion code)

        insert('F');

    insert('S');

    insert('Q');

    insert('K');

    insert('C');

    insert('L');

    insert('H');

    insert('T');

    insert('V');

    insert('W');

    insert('M');

    insert('R');

    insert('N');

    insert('P');

    insert('A');

    insert('B');

    insert('X');

    insert('Y');

    insert('D');

    insert('Z');

    insert('E');

    printf("Inorder Traversal of RB Tree (before deletion): ");

    inorderTraversal(root);

    printf("\n");

    // Delete a node from the Red-Black Tree

    deleteNode('S');

    printf("Inorder Traversal of RB Tree (after deletion): ");

    inorderTraversal(root);

    printf("\n");

    deleteNode('Z');

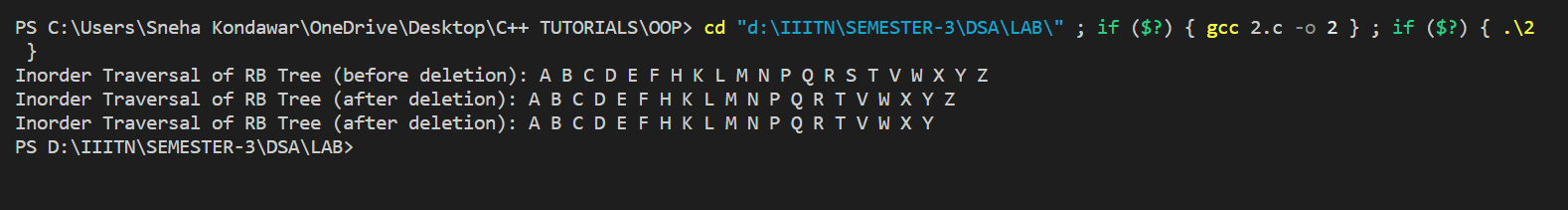
    printf("Inorder Traversal of RB Tree (after deletion): ");

    inorderTraversal(root);

    printf("\n");

    return 0;

}



3. #include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*leftChild, \*rightChild;

};

struct node\* newNode(int data){

struct node\* Node = (struct node\*)malloc(sizeof(struct node));

Node->data = data;

Node->leftChild = Node->rightChild = NULL;

return (Node);

}

struct node\* rightRotate(struct node \*x){

struct node \*y = x->leftChild;

x->leftChild = y->rightChild;

y->rightChild = x;

return y;

}

struct node\* leftRotate(struct node \*x){

struct node \*y = x->rightChild;

x->rightChild = y->leftChild;

y->leftChild = x;

return y;

}

struct node\* splay(struct node \*root, int data){

if (root == NULL || root->data == data)

return root;

if (root->data > data) {

if (root->leftChild == NULL) return root;

if (root->leftChild->data > data) {

root->leftChild->leftChild = splay(root->leftChild->leftChild, data);

root = rightRotate(root);

} else if (root->leftChild->data < data) {

root->leftChild->rightChild = splay(root->leftChild->rightChild, data);

if (root->leftChild->rightChild != NULL)

root->leftChild = leftRotate(root->leftChild);

}

return (root->leftChild == NULL)? root: rightRotate(root);

} else {

if (root->rightChild == NULL) return root;

if (root->rightChild->data > data) {

root->rightChild->leftChild = splay(root->rightChild->leftChild, data);

if (root->rightChild->leftChild != NULL)

root->rightChild = rightRotate(root->rightChild);

} else if (root->rightChild->data < data) {

root->rightChild->rightChild = splay(root->rightChild->rightChild, data);

root = leftRotate(root);

}

return (root->rightChild == NULL)? root: leftRotate(root);

}

}

struct node\* insert(struct node \*root, int k){

if (root == NULL) return newNode(k);

root = splay(root, k);

if (root->data == k) return root;

struct node \*newnode = newNode(k);

if (root->data > k) {

newnode->rightChild = root;

newnode->leftChild = root->leftChild;

root->leftChild = NULL;

} else {

newnode->leftChild = root;

newnode->rightChild = root->rightChild;

root->rightChild = NULL;

}

return newnode;

}

struct node\* deletenode(struct node\* root, int data){

struct node\* temp;

if (root == NULL)

return NULL;

root = splay(root, data);

if (data != root->data)

return root;

if (!root->leftChild) {

temp = root;

root = root->rightChild;

} else {

temp = root;

root = splay(root->leftChild, data);

root->rightChild = temp->rightChild;

}

free(temp);

return root;

}

int height(struct node\* node)

{

if (node == NULL)

return 0;

else {

// Compute the height of each subtree

int lheight = height(node->leftChild);

int rheight = height(node->rightChild);

// Use the larger one

if (lheight > rheight)

return (lheight + 1);

else

return (rheight + 1);

}

}

void printCurrentLevel(struct node\* root, int level)

{

if (root == NULL)

return;

if (level == 1)

printf("%d ", root->data);

else if (level > 1) {

printCurrentLevel(root->leftChild, level - 1);

printCurrentLevel(root->rightChild, level - 1);

}

}

void printLevelOrder(struct node\* root)

{

int h = height(root);

int i;

for (i = 1; i <= h; i++){

printCurrentLevel(root, i);

printf("\n");

}

}

int main(){

struct node\* root = newNode(34);

root = insert(root,15);

root = insert(root,40);

root = insert(root,12);

root = insert(root,14);

root = insert(root,59);

printf("The Splay tree is \n");

printLevelOrder(root);

root = deletenode(root, 40);

printf("\nThe Splay tree after deletion is \n");

printLevelOrder(root);

return 0;

}