1. **Write a C program to implement Red black tree.**

Code:

```
#include <stdio.h>

#include <stdlib.h>


// Define colors

#define RED 1

#define BLACK 0


// Structure for a node in the Red-Black Tree

typedef struct RBNode {

   int data;

   int color; // RED or BLACK

   struct RBNode *left, *right, *parent;

} RBNode;


// Function prototypes

RBNode* createNode(int data);

void rotateLeft(RBNode** root, RBNode* node);
```

```
void rotateRight(RBNode** root, RBNode* node);

void insertFixUp(RBNode** root, RBNode* node);

void insertNode(RBNode** root, int data);

void inorderTraversal(RBNode* root);

void freeTree(RBNode* root);


// Create a new node with given data

RBNode* createNode(int data) {

   RBNode* newNode = (RBNode*)malloc(sizeof(RBNode));

   newNode->data = data;

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

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

   return newNode;

}


// Left rotation

void rotateLeft(RBNode** root, RBNode* node) {

   RBNode* rightChild = node->right;
```

```
node->right = rightChild->left;

if (rightChild->left != NULL) {
    rightChild->left->parent = node;
}

rightChild->parent = node->parent;

if (node->parent == NULL) {
    *root = rightChild;
} else if (node == node->parent->left) {
    node->parent->left = rightChild;
} else {
    node->parent->right = rightChild;
}

rightChild->left = node;
node->parent = rightChild;
```

```
}

// Right rotation
void rotateRight(RBNode** root, RBNode* node) {
    RBNode* leftChild = node->left;
    node->left = leftChild->right;

    if (leftChild->right != NULL) {
        leftChild->right->parent = node;
    }

    leftChild->parent = node->parent;

    if (node->parent == NULL) {
        *root = leftChild;
    } else if (node == node->parent->right) {
        node->parent->right = leftChild;
    } else {
```

```
        node->parent->left = leftChild;
    }

    leftChild->right = node;
    node->parent = leftChild;
}

// Fix the Red-Black Tree after insertion
void insertFixUp(RBNode** root, RBNode* node) {
    while (node != *root && node->parent->color == RED) {
        if (node->parent == node->parent->parent->left) {
            RBNode* uncle = node->parent->parent->right;

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

```
    } else {
        if (node == node->parent->right) {
            node = node->parent;
            rotateLeft(root, node);
        }
        node->parent->color = BLACK;
        node->parent->parent->color = RED;
        rotateRight(root, node->parent->parent);
    }
} else {
    RBNode* uncle = node->parent->parent->left;

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

```
            if (node == node->parent->left) {
                node = node->parent;
                rotateRight(root, node);
            }
            node->parent->color = BLACK;
            node->parent->parent->color = RED;
            rotateLeft(root, node->parent->parent);
        }
    }
  }

  (*root)->color = BLACK;
}

// Insert a new node into the Red-Black Tree
void insertNode(RBNode** root, int data) {
  RBNode* newNode = createNode(data);
  RBNode* parent = NULL;
```

```
RBNode* current = *root;


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


newNode->parent = parent;


if (parent == NULL) {
    *root = newNode;
} else if (data < parent->data) {
    parent->left = newNode;
} else {
```

```
        parent->right = newNode;
    }

    insertFixUp(root, newNode);
}

// Inorder traversal of the Red-Black Tree
void inorderTraversal(RBNode* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
            printf("%d (%s) ", root->data, root->color == RED ? "RED" :
"BLACK");
        inorderTraversal(root->right);
    }
}

// Free the memory allocated for the Red-Black Tree
void freeTree(RBNode* root) {
```

```
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}

// Main function to demonstrate Red-Black Tree operations
int main() {
    RBNode* root = NULL;

    insertNode(& root, 10);
    insertNode(& root, 20);
    insertNode(& root, 30);
    insertNode(& root, 15);
    insertNode(& root, 25);

    printf("Inorder Traversal of Red-Black Tree:\n");
```

```
    inorderTraversal(root);
    printf("\n");

    freeTree(root);
    return 0;
}
```

Output: Inorder Traversal of Red-Black Tree:

10 (BLACK) 15 (RED) 20 (BLACK) 25 (RED) 30 (BLACK)

2 .Write a C program to implement the Splay tree.

Code:

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

// Node structure for the splay tree
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
```

```
} Node;


// Function to create a new node
Node* createNode(int key) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}


// Function to perform right rotation
Node* rightRotate(Node* root) {
    Node* newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    return newRoot;
}
```

```
// Function to perform left rotation
Node* leftRotate(Node* root) {
    Node* newRoot = root->right;
    root->right = newRoot->left;
    newRoot->left = root;
    return newRoot;
}

// Splay function to bring the key to the root
Node* splay(Node* root, int key) {
    if (root == NULL || root->key == key) {
        return root;
    }

    if (key < root->key) {
        if (root->left == NULL) return root;
```

```
    // Key is in the left subtree
    if (key < root->left->key) {
        // Zig-Zig (Left Left)
        root->left->left = splay(root->left->left, key);
        root = rightRotate(root);
    } else if (key > root->left->key) {
        // Zig-Zag (Left Right)
        root->left->right = splay(root->left->right, key);
        if (root->left->right != NULL) {
            root->left = leftRotate(root->left);
        }
    }
    return (root->left == NULL) ? root : rightRotate(root);
} else {
    if (root->right == NULL) return root;

    // Key is in the right subtree
    if (key > root->right->key) {
```

```
            // Zig-Zig (Right Right)
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        } else if (key < root->right->key) {
            // Zig-Zag (Right Left)
            root->right->left = splay(root->right->left, key);
            if (root->right->left != NULL) {
                root->right = rightRotate(root->right);
            }
        }
        return (root->right == NULL) ? root : leftRotate(root);
    }
}

// Function to insert a new key
Node* insert(Node* root, int key) {
    if (root == NULL) return createNode(key);
```

```
    root = splay(root, key);

    if (root->key == key) return root;

    Node* newNode = createNode(key);

    if (key < root->key) {
        newNode->right = root;
        newNode->left = root->left;
        root->left = NULL;
    } else {
        newNode->left = root;
        newNode->right = root->right;
        root->right = NULL;
    }

    return newNode;
}
```

```
// Function to search for a key
Node* search(Node* root, int key) {
    return splay(root, key);
}

// Function to find the minimum value in the splay tree
Node* findMin(Node* root) {
    while (root->left != NULL) root = root->left;
    return root;
}

// Function to delete a key
Node* delete(Node* root, int key) {
    if (root == NULL) return NULL;

    root = splay(root, key);
```

```
    if (key != root->key) return root;


    Node* temp;
    if (root->left == NULL) {
        temp = root;
        root = root->right;
    } else {
        temp = root;
        Node* newRoot = splay(root->left, key);
        newRoot->right = root->right;
        root = newRoot;
    }


    free(temp);
    return root;
}


// Function to print the tree in-order
```

```
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
}

// Main function for testing
int main() {
    Node* root = NULL;

    // Insert elements
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
```

```
    // Print in-order traversal
    printf("In-order traversal: ");
    inorder(root);
    printf("\n");

    // Search for a key
    root = search(root, 20);
    printf("Root after searching for 20: %d\n", root->key);

    // Delete a key
    root = delete(root, 30);
    printf("In-order traversal after deleting 30: ");
    inorder(root);
    printf("\n");

    return 0;
}
```

Output: **In-order traversal: 10 20 25 30 40 50**

Root after searching for 20: 20

In-order traversal after deleting 30: 10 20 25 40 50