Algorithm:

Q1.

1. **Node Structure**:
   * Define a structure **Node** containing three fields: **key** to hold the integer value, **left** to store the pointer to the left child, and **right** to store the pointer to the right child.
2. **SplayTree Class**:
   * Define a class **SplayTree** to encapsulate the Splay Tree operations.
   * Declare private member variables: **root** to store the root node of the tree.
   * Define private member functions:
     + **rightRotate**: Perform a right rotation operation on the given node.
     + **leftRotate**: Perform a left rotation operation on the given node.
     + **splay**: Perform the splay operation on the tree with the given key.
     + **findMin**: Find and return the node with the minimum key value in the tree.
     + **findMax**: Find and return the node with the maximum key value in the tree.
     + **deleteNode**: Delete the node with the given key from the tree.
     + **inorder**: Perform an inorder traversal of the tree.
     + **preorder**: Perform a preorder traversal of the tree.
     + **postorder**: Perform a postorder traversal of the tree.
   * Define public member functions:
     + **insert**: Insert a new node with the given key into the tree.
     + **remove**: Remove the node with the given key from the tree.
     + **removeMin**: Remove the node with the minimum key value from the tree.
     + **removeMax**: Remove the node with the maximum key value from the tree.
     + **traverseInorder**: Perform an inorder traversal of the tree and print the keys.
     + **traversePreorder**: Perform a preorder traversal of the tree and print the keys.
     + **traversePostorder**: Perform a postorder traversal of the tree and print the keys.
3. **Main Function**:
   * Inside the **main** function:
     + Create an instance of the **SplayTree** class.
     + Insert some nodes into the tree.
     + Perform various operations like deletion and traversals on the tree.

Q2.

1. **Node Structure**:
   * Define a structure **Node** containing three fields: **key** to hold the integer value, **left** to store the pointer to the left child, and **right** to store the pointer to the right child.
2. **SplayTree Class**:
   * Define a class **SplayTree** to encapsulate the Splay Tree operations.
   * Declare private member variables: **root** to store the root node of the tree.
   * Define private member functions:
     + **rightRotate**: Perform a right rotation operation on the given node.
     + **leftRotate**: Perform a left rotation operation on the given node.
     + **topDownSplay**: Perform the top-down splay operation on the tree with the given key.
     + **bottomUpSplay**: Perform the bottom-up splay operation on the tree with the given key.
     + **splay**: Call either **topDownSplay** or **bottomUpSplay** based on the selected splay approach.
     + **findMin**: Find and return the node with the minimum key value in the tree.
     + **findMax**: Find and return the node with the maximum key value in the tree.
     + **deleteNode**: Delete the node with the given key from the tree.
     + **inorder**: Perform an inorder traversal of the tree.
     + **preorder**: Perform a preorder traversal of the tree.
     + **postorder**: Perform a postorder traversal of the tree.
   * Define public member functions:
     + **insert**: Insert a new node with the given key into the tree.
     + **remove**: Remove the node with the given key from the tree.
     + **removeMin**: Remove the node with the minimum key value from the tree.
     + **removeMax**: Remove the node with the maximum key value from the tree.
     + **traverseInorder**: Perform an inorder traversal of the tree and print the keys.
     + **traversePreorder**: Perform a preorder traversal of the tree and print the keys.
     + **traversePostorder**: Perform a postorder traversal of the tree and print the keys.
3. **Main Function**:
   * Inside the **main** function:
     + Create an instance of the **SplayTree** class.
     + Insert some nodes into the tree using the **insert** function.
     + Perform deletion operations using the **remove**, **removeMin**, or **removeMax** functions.
     + Print the tree using the traversal functions **traverseInorder**, **traversePreorder**, or **traversePostorder**.

Q3.

1. **Node Structure**:
   * Define a structure **Node** containing three fields: **key** to hold the integer value, **left** to store the pointer to the left child, and **right** to store the pointer to the right child.
2. **SplayTree Class**:
   * Define a class **SplayTree** to encapsulate the Splay Tree operations.
   * Declare private member variables: **root** to store the root node of the tree.
   * Define private member functions:
     + **rightRotate**: Perform a right rotation operation on the given node.
     + **leftRotate**: Perform a left rotation operation on the given node.
     + **topDownSplay**: Perform the top-down splay operation on the tree with the given key.
     + **splay**: Call **topDownSplay** to perform splay operation.
     + **findMin**: Find and return the node with the minimum key value in the tree.
     + **findMax**: Find and return the node with the maximum key value in the tree.
     + **deleteNode**: Delete the node with the given key from the tree.
     + **join**: Join two trees into one at the specified key.
     + **split**: Split the tree into two separate trees at the specified key.
     + **inorder**, **preorder**, **postorder**: Traversal functions to print the keys of the tree.
3. **Main Function**:
   * Inside the **main** function:
     + Create instances of the **SplayTree** class: **tree1** and **tree2**.
     + Insert nodes into **tree1** and **tree2**.
     + Perform operations:
       - Splitting **tree1** at a specified key.
       - Joining **tree1** and **tree2** at a specified key.
     + Print the inorder traversal of the trees to visualize the results.

Code:

Q1.

#include <iostream>

using *namespace* std;

*struct* Node {

*int* key;

    Node\* left;

    Node\* right;

};

*class* SplayTree {

*private:*

    Node\* root;

    Node\* rightRotate(Node\* *x*) {

        Node\* y = *x*->left;

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

        y->right = *x*;

        return y;

    }

    Node\* leftRotate(Node\* *x*) {

        Node\* y = *x*->right;

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

        y->left = *x*;

        return y;

    }

    Node\* splay(Node\* *root*, *int* *key*) {

        if (*root* == nullptr || *root*->key == *key*)

            return *root*;

        if (*root*->key > *key*) {

            if (*root*->left == nullptr)

                return *root*;

            if (*root*->left->key > *key*) {

*root*->left->left = splay(*root*->left->left, *key*);

*root* = rightRotate(*root*);

            } else if (*root*->left->key < *key*) {

*root*->left->right = splay(*root*->left->right, *key*);

                if (*root*->left->right != nullptr)

*root*->left = leftRotate(*root*->left);

            }

            return (*root*->left == nullptr) ? *root* : rightRotate(*root*);

        } else {

            if (*root*->right == nullptr)

                return *root*;

            if (*root*->right->key > *key*) {

*root*->right->left = splay(*root*->right->left, *key*);

                if (*root*->right->left != nullptr)

*root*->right = rightRotate(*root*->right);

            } else if (*root*->right->key < *key*) {

*root*->right->right = splay(*root*->right->right, *key*);

*root* = leftRotate(*root*);

            }

            return (*root*->right == nullptr) ? *root* : leftRotate(*root*);

        }

    }

    Node\* findMin(Node\* *node*) {

        if (*node* == nullptr)

            return nullptr;

        while (*node*->left != nullptr)

*node* = *node*->left;

        return *node*;

    }

    Node\* findMax(Node\* *node*) {

        if (*node* == nullptr)

            return nullptr;

        while (*node*->right != nullptr)

*node* = *node*->right;

        return *node*;

    }

    Node\* deleteNode(Node\* *root*, *int* *key*) {

        if (*root* == nullptr)

            return nullptr;

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

        if (*root*->key != *key*)

            return *root*;

        Node\* temp;

        if (*root*->left == nullptr) {

            temp = *root*;

*root* = *root*->right;

        } else {

            temp = *root*;

*root* = splay(*root*->left, *key*);

*root*->right = temp->right;

        }

        delete temp;

        return *root*;

    }

*void* inorder(Node\* *root*) {

        if (*root* != nullptr) {

            inorder(*root*->left);

            cout << *root*->key << " ";

            inorder(*root*->right);

        }

    }

*void* preorder(Node\* *root*) {

        if (*root* != nullptr) {

            cout << *root*->key << " ";

            preorder(*root*->left);

            preorder(*root*->right);

        }

    }

*void* postorder(Node\* *root*) {

        if (*root* != nullptr) {

            postorder(*root*->left);

            postorder(*root*->right);

            cout << *root*->key << " ";

        }

    }

*public:*

    SplayTree() {

        root = nullptr;

    }

*void* insert(*int* *key*) {

        if (root == nullptr) {

            root = new Node;

            root->key = *key*;

            root->left = root->right = nullptr;

            return;

        }

        root = splay(root, *key*);

        if (root->key == *key*)

            return;

        Node\* newNode = new Node;

        newNode->key = *key*;

        if (root->key > *key*) {

            newNode->right = root;

            newNode->left = root->left;

            root->left = nullptr;

        } else {

            newNode->left = root;

            newNode->right = root->right;

            root->right = nullptr;

        }

        root = newNode;

    }

*void* remove(*int* *key*) {

        root = deleteNode(root, *key*);

    }

*void* removeMin() {

        if (root != nullptr) {

            Node\* minNode = findMin(root);

            if (minNode != nullptr)

                root = deleteNode(root, minNode->key);

        }

    }

*void* removeMax() {

        if (root != nullptr) {

            Node\* maxNode = findMax(root);

            if (maxNode != nullptr)

                root = deleteNode(root, maxNode->key);

        }

    }

*void* traverseInorder() {

        inorder(root);

        cout << endl;

    }

*void* traversePreorder() {

        preorder(root);

        cout << endl;

    }

*void* traversePostorder() {

        postorder(root);

        cout << endl;

    }

};

*int* main() {

    SplayTree tree;

    tree.insert(10);

    tree.insert(5);

    tree.insert(15);

    tree.insert(20);

    tree.insert(7);

    cout << "Inorder traversal: ";

    tree.traverseInorder();

    tree.remove(7);

    cout << "After deleting 7: ";

    tree.traverseInorder();

    cout << "Preorder traversal: ";

    tree.traversePreorder();

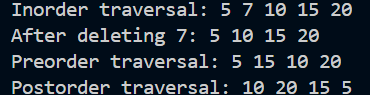
    cout << "Postorder traversal: ";

    tree.traversePostorder();

    return 0;

}

Output:



Q2.

#include <iostream>

using *namespace* std;

*struct* Node {

*int* key;

    Node\* left;

    Node\* right;

};

*class* SplayTree {

*private:*

    Node\* root;

    Node\* rightRotate(Node\* *x*) {

        Node\* y = *x*->left;

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

        y->right = *x*;

        return y;

    }

    Node\* leftRotate(Node\* *x*) {

        Node\* y = *x*->right;

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

        y->left = *x*;

        return y;

    }

    // Top-down splay

    Node\* topDownSplay(Node\* *root*, *int* *key*) {

        if (*root* == nullptr || *root*->key == *key*)

            return *root*;

        if (*root*->key > *key*) {

            if (*root*->left == nullptr)

                return *root*;

            if (*root*->left->key > *key*) {

*root*->left->left = topDownSplay(*root*->left->left, *key*);

*root* = rightRotate(*root*);

            } else if (*root*->left->key < *key*) {

*root*->left->right = topDownSplay(*root*->left->right, *key*);

                if (*root*->left->right != nullptr)

*root*->left = leftRotate(*root*->left);

            }

            return (*root*->left == nullptr) ? *root* : rightRotate(*root*);

        } else {

            if (*root*->right == nullptr)

                return *root*;

            if (*root*->right->key > *key*) {

*root*->right->left = topDownSplay(*root*->right->left, *key*);

                if (*root*->right->left != nullptr)

*root*->right = rightRotate(*root*->right);

            } else if (*root*->right->key < *key*) {

*root*->right->right = topDownSplay(*root*->right->right, *key*);

*root* = leftRotate(*root*);

            }

            return (*root*->right == nullptr) ? *root* : leftRotate(*root*);

        }

    }

    // Bottom-up splay

    Node\* bottomUpSplay(Node\* *root*, *int* *key*) {

        if (*root* == nullptr || *root*->key == *key*)

            return *root*;

        if (*root*->key > *key*) {

            if (*root*->left == nullptr)

                return *root*;

            if (*root*->left->key > *key*) {

*root*->left->left = bottomUpSplay(*root*->left->left, *key*);

*root* = rightRotate(*root*);

            } else if (*root*->left->key < *key*) {

*root*->left->right = bottomUpSplay(*root*->left->right, *key*);

                if (*root*->left->right != nullptr)

*root*->left = rightRotate(*root*->left);

            }

            return (*root*->left == nullptr) ? *root* : rightRotate(*root*);

        } else {

            if (*root*->right == nullptr)

                return *root*;

            if (*root*->right->key > *key*) {

*root*->right->left = bottomUpSplay(*root*->right->left, *key*);

                if (*root*->right->left != nullptr)

*root*->right = leftRotate(*root*->right);

            } else if (*root*->right->key < *key*) {

*root*->right->right = bottomUpSplay(*root*->right->right, *key*);

*root* = leftRotate(*root*);

            }

            return (*root*->right == nullptr) ? *root* : leftRotate(*root*);

        }

    }

    Node\* splay(Node\* *root*, *int* *key*) {

        return topDownSplay(*root*, *key*); // Change to bottomUpSplay(root, key) for bottom-up splay

    }

    Node\* findMin(Node\* *node*) {

        if (*node* == nullptr)

            return nullptr;

        while (*node*->left != nullptr)

*node* = *node*->left;

        return *node*;

    }

    Node\* findMax(Node\* *node*) {

        if (*node* == nullptr)

            return nullptr;

        while (*node*->right != nullptr)

*node* = *node*->right;

        return *node*;

    }

    Node\* deleteNode(Node\* *root*, *int* *key*) {

        if (*root* == nullptr)

            return nullptr;

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

        if (*root*->key != *key*)

            return *root*;

        Node\* temp;

        if (*root*->left == nullptr) {

            temp = *root*;

*root* = *root*->right;

        } else {

            temp = *root*;

*root* = splay(*root*->left, *key*);

*root*->right = temp->right;

        }

        delete temp;

        return *root*;

    }

*void* inorder(Node\* *root*) {

        if (*root* != nullptr) {

            inorder(*root*->left);

            cout << *root*->key << " ";

            inorder(*root*->right);

        }

    }

*void* preorder(Node\* *root*) {

        if (*root* != nullptr) {

            cout << *root*->key << " ";

            preorder(*root*->left);

            preorder(*root*->right);

        }

    }

*void* postorder(Node\* *root*) {

        if (*root* != nullptr) {

            postorder(*root*->left);

            postorder(*root*->right);

            cout << *root*->key << " ";

        }

    }

*public:*

    SplayTree() {

        root = nullptr;

    }

*void* insert(*int* *key*) {

        if (root == nullptr) {

            root = new Node;

            root->key = *key*;

            root->left = root->right = nullptr;

            return;

        }

        root = splay(root, *key*);

        if (root->key == *key*)

            return;

        Node\* newNode = new Node;

        newNode->key = *key*;

        if (root->key > *key*) {

            newNode->right = root;

            newNode->left = root->left;

            root->left = nullptr;

        } else {

            newNode->left = root;

            newNode->right = root->right;

            root->right = nullptr;

        }

        root = newNode;

    }

*void* remove(*int* *key*) {

        root = deleteNode(root, *key*);

    }

*void* removeMin() {

        if (root != nullptr) {

            Node\* minNode = findMin(root);

            if (minNode != nullptr)

                root = deleteNode(root, minNode->key);

        }

    }

*void* removeMax() {

        if (root != nullptr) {

            Node\* maxNode = findMax(root);

            if (maxNode != nullptr)

                root = deleteNode(root, maxNode->key);

        }

    }

*void* traverseInorder() {

        inorder(root);

        cout << endl;

    }

*void* traversePreorder() {

        preorder(root);

        cout << endl;

    }

*void* traversePostorder() {

        postorder(root);

        cout << endl;

    }

};

*int* main() {

    SplayTree tree;

    tree.insert(10);

    tree.insert(5);

    tree.insert(15);

    tree.insert(20);

    tree.insert(7);

    cout << "Inorder traversal: ";

    tree.traverseInorder();

    tree.remove(7);

    cout << "After deleting 7: ";

}

Output:



Q3

#include <iostream>

using *namespace* std;

*struct* Node {

*int* key;

    Node\* left;

    Node\* right;

};

*class* SplayTree {

*private:*

    Node\* root;

    Node\* rightRotate(Node\* *x*) {

        Node\* y = *x*->left;

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

        y->right = *x*;

        return y;

    }

    Node\* leftRotate(Node\* *x*) {

        Node\* y = *x*->right;

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

        y->left = *x*;

        return y;

    }

    Node\* topDownSplay(Node\* *root*, *int* *key*) {

        if (*root* == nullptr || *root*->key == *key*)

            return *root*;

        if (*root*->key > *key*) {

            if (*root*->left == nullptr)

                return *root*;

            if (*root*->left->key > *key*) {

*root*->left->left = topDownSplay(*root*->left->left, *key*);

*root* = rightRotate(*root*);

            } else if (*root*->left->key < *key*) {

*root*->left->right = topDownSplay(*root*->left->right, *key*);

                if (*root*->left->right != nullptr)

*root*->left = leftRotate(*root*->left);

            }

            return (*root*->left == nullptr) ? *root* : rightRotate(*root*);

        } else {

            if (*root*->right == nullptr)

                return *root*;

            if (*root*->right->key > *key*) {

*root*->right->left = topDownSplay(*root*->right->left, *key*);

                if (*root*->right->left != nullptr)

*root*->right = rightRotate(*root*->right);

            } else if (*root*->right->key < *key*) {

*root*->right->right = topDownSplay(*root*->right->right, *key*);

*root* = leftRotate(*root*);

            }

            return (*root*->right == nullptr) ? *root* : leftRotate(*root*);

        }

    }

    Node\* splay(Node\* *root*, *int* *key*) {

        return topDownSplay(*root*, *key*);

    }

    Node\* findMin(Node\* *node*) {

        if (*node* == nullptr)

            return nullptr;

        while (*node*->left != nullptr)

*node* = *node*->left;

        return *node*;

    }

    Node\* findMax(Node\* *node*) {

        if (*node* == nullptr)

            return nullptr;

        while (*node*->right != nullptr)

*node* = *node*->right;

        return *node*;

    }

    Node\* deleteNode(Node\* *root*, *int* *key*) {

        if (*root* == nullptr)

            return nullptr;

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

        if (*root*->key != *key*)

            return *root*;

        Node\* temp;

        if (*root*->left == nullptr) {

            temp = *root*;

*root* = *root*->right;

        } else {

            temp = *root*;

*root* = splay(*root*->left, *key*);

*root*->right = temp->right;

        }

        delete temp;

        return *root*;

    }

    Node\* join(Node\* *T1*, Node\* *T2*, *int* *key*) {

        if (*T1* == nullptr) {

            return *T2*;

        }

        if (*T2* == nullptr) {

            return *T1*;

        }

*T1* = splay(*T1*, *key*);

*T1*->right = *T2*;

        return *T1*;

    }

    pair<Node\*, Node\*> split(Node\* *root*, *int* *key*) {

        if (*root* == nullptr)

            return {nullptr, nullptr};

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

        if (*root*->key <= *key*) {

            Node\* rightSubtree = *root*->right;

*root*->right = nullptr;

            return {*root*, rightSubtree};

        } else {

            Node\* leftSubtree = *root*->left;

*root*->left = nullptr;

            return {leftSubtree, *root*};

        }

    }

*void* inorder(Node\* *root*) {

        if (*root* != nullptr) {

            inorder(*root*->left);

            cout << *root*->key << " ";

            inorder(*root*->right);

        }

    }

*void* preorder(Node\* *root*) {

        if (*root* != nullptr) {

            cout << *root*->key << " ";

            preorder(*root*->left);

            preorder(*root*->right);

        }

    }

*void* postorder(Node\* *root*) {

        if (*root* != nullptr) {

            postorder(*root*->left);

            postorder(*root*->right);

            cout << *root*->key << " ";

        }

    }

*public:*

    SplayTree() {

        root = nullptr;

    }

*void* insert(*int* *key*) {

        if (root == nullptr) {

            root = new Node;

            root->key = *key*;

            root->left = root->right = nullptr;

            return;

        }

        root = splay(root, *key*);

        if (root->key == *key*)

            return;

        Node\* newNode = new Node;

        newNode->key = *key*;

        if (root->key > *key*) {

            newNode->right = root;

            newNode->left = root->left;

            root->left = nullptr;

        } else {

            newNode->left = root;

            newNode->right = root->right;

            root->right = nullptr;

        }

        root = newNode;

    }

*void* remove(*int* *key*) {

        root = deleteNode(root, *key*);

    }

*void* removeMin() {

        if (root != nullptr) {

            Node\* minNode = findMin(root);

            if (minNode != nullptr)

                root = deleteNode(root, minNode->key);

        }

    }

*void* removeMax() {

        if (root != nullptr) {

            Node\* maxNode = findMax(root);

            if (maxNode != nullptr)

                root = deleteNode(root, maxNode->key);

        }

    }

*void* traverseInorder() {

        inorder(root);

        cout << endl;

    }

*void* traversePreorder() {

        preorder(root);

        cout << endl;

    }

*void* traversePostorder() {

        postorder(root);

        cout << endl;

    }

*void* splitTree(*int* *key*) {

        pair<Node\*, Node\*> splitTrees = split(root, *key*);

        cout << "Tree with keys <= " << *key* << ": ";

        inorder(splitTrees.first);

        cout << endl;

        cout << "Tree with keys > " << *key* << ": ";

        inorder(splitTrees.second);

        cout << endl;

    }

*void* joinTrees(SplayTree& *tree2*, *int* *key*) {

        root = join(root, *tree2*.root, *key*);

    }

};

*int* main() {

    SplayTree tree1;

    tree1.insert(10);

    tree1.insert(5);

    tree1.insert(15);

    tree1.insert(20);

    tree1.insert(7);

    cout << "Inorder traversal of tree1: ";

    tree1.traverseInorder();

    SplayTree tree2;

    tree2.insert(25);

    tree2.insert(30);

    tree2.insert(35);

    tree2.insert(40);

    tree2.insert(45);

    cout << "Inorder traversal of tree2: ";

    tree2.traverseInorder();

    cout << "Splitting tree1 at key 15: " << endl;

    tree1.splitTree(15);

    cout << "Joining tree1 and tree2 at key 25: " << endl;

    tree1.joinTrees(tree2, 25);

    cout << "Inorder traversal of joined tree: ";

    tree1.traverseInorder();

    return 0;

}

Output:

