Jamia Millia Islamia

Advanced Data Structures Lab Report

Ashhar Hasan
13 BCS 0015

Table of Contents

Binary Search Tree Using an Array

```
/* Implement a BST using an array */
#include <iostream>

using namespace std;

struct BST
{
	int info;
	BST* right;
	BST* left;
} *t1, *t2;

// Inserts an element into the BST
BST* insert(BST* root)
{
	int i;
	cout << "Enter the node's value: ";
	cin >> i;
	// Empty tree
	if (root == nullptr)
	{
		root = new BST;
		root->info = i;
		root->left = root->right = nullptr;
	}
	else
	{
		t1 = root;
		t2 = nullptr;
		while ((t1->left != nullptr || t1->right != nullptr) && t2 != t1)
		{
			t2 = t1;
			while (i < t1->info && t1->left != nullptr)
				t1 = t1->left;
			while (i > t1->info && t1->right != nullptr)
				t1 = t1->right;
		}
		t2 = new BST;
		t2->info = i;
		t2->left = t2->right = nullptr;
		if (i < t1->info)
			t1->left = t2;
		else
			t1->right = t2;
	}
	return root;
}
```

```
// Push an element into the stack
int push(BST* stk[], BST* p, int top)
{
	stk[++top] = p;
	return top;
}

// Pop an element from the stack
BST* pop(BST* stk[], int* top)
{
	return stk[(*top)--];
}

// In-Order traversal of the BST
void inorder(BST* t)
{
	BST *p, *stk[20];
	int top = -1;
	p = t;
	do
	{
		while (p != nullptr)
		{
			// Push an element into the stack and move to its left
			top = push(stk, p, top);
			p = p->left;
		}
		// As long as no underflow, keep popping and moving to the right
		if (top != -1)
		{
			p = pop(stk, &top);
			cout << " \t" << p->info;
			p = p->right;
		}
	} while (top != -1 || p != nullptr);
}

int main()
{
	int ch;
	BST* root;
	root = nullptr;
menu:
	cout << "Binary Search Tree Operations" << endl;
	cout << "-----------------------------" << endl;
	cout << "1. Insertion/Creation" << endl;
	cout << "2. In-Order Traversal" << endl;
	cout << "3. Exit" << endl;
	cout << "Enter your choice: ";
	cin >> ch;
```

```
	switch (ch)
	{
		case 1:
			root = insert(root);
			goto menu;
		case 2:
			if (root != nullptr)
				inorder(root);
			else
				cout << "No tree exists!" << endl;
			goto menu;
		case 3:
			return 0;
		default:
			goto menu;
	}
}
```

Binary Search Tree Using a Linked List

```
/* Implement a BST using linked lists */
#include <iostream>

using namespace std;

class BST
{
	struct TreeNode
	{
		int data;
		TreeNode* left;
		TreeNode* right;
	};

public:
	TreeNode* root;

	BST()
	{
		root = nullptr;
	}

	bool IsEmpty()
	{
		return root == nullptr;
	}

	bool Inorder(TreeNode*);
```

```
	bool Insert(int);
};

// Perform an in-order traversal on the tree
bool BST::Inorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		// The extra if avoids recursion if the node is a leaf node
		if (start_node->left)
			Inorder(start_node->left);
		cout << " " << start_node->data << " ";
		if (start_node->right)
			Inorder(start_node->right);
	}
	return true;
}

// Insert an item in the BST
bool BST::Insert(int item)
{
	// Create a new node
	TreeNode* node = new TreeNode;
	node->data = item;
	node->left = nullptr;
	node->right = nullptr;
	TreeNode* parent = nullptr;
	if (IsEmpty())
		root = node;
	// If not, we need to find the proper to-be parent of element
	else
	{
		TreeNode* current = root;
		while (current)
		{
			parent = current;
			if (node->data > current->data)
				current = current->right;
			else
				current = current->left;
		}
		if (node->data < parent->data)
			parent->left = node;
		else
			parent->right = node;
	}
	return true;
}

int main()
{
```

```
	BST bst;
	int choice_i, item_i;
	while (1)
	{
		cout << endl << endl;
		cout << "Binary Search Tree Operations" << endl;
		cout << "-----------------------------" << endl;
		cout << "1. Insertion/Creation" << endl;
		cout << "2. In-Order Traversal" << endl;
		cout << "3. Exit" << endl;
		cout << "Enter your choice: ";
		cin >> choice_i;
		switch (choice_i)
		{
			case 1:
				cout << "Enter Number to be inserted: ";
				cin >> item_i;
				if (bst.Insert(item_i))
					cout << "The element was inserted successfully into the 
tree.";
				break;
			case 2:
				cout << endl;
				cout << "In-Order Traversal" << endl;
				cout << "------------------" << endl;
				bst.Inorder(bst.root);
				break;
			case 3:
				return 0;
			default:
				cout << "Invalid choice! Try again.";
		}
	}
}
```

Traversals of a Binary Search Tree

```
/* Implement In-order, Post-Order and Pre-Order Traversal of a BST
using linked lists */
#include <iostream>

using namespace std;

class BST
{
	struct TreeNode
	{
		int data;
```

```
		TreeNode* left;
		TreeNode* right;
	};

public:
	TreeNode* root;

	BST()
	{
		root = nullptr;
	}

	bool IsEmpty()
	{
		return root == nullptr;
	}

	bool Inorder(TreeNode*);
	bool Preorder(TreeNode*);
	bool Postorder(TreeNode*);
	bool Insert(int);
};

bool BST::Inorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		if (start_node->left)
			Inorder(start_node->left);
		cout << " " << start_node->data << " ";
		if (start_node->right)
			Inorder(start_node->right);
	}
	return true;
}

bool BST::Preorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		cout << " " << start_node->data << " ";
		if (start_node->left)
			Preorder(start_node->left);
		if (start_node->right)
			Preorder(start_node->right);
	}
	return true;
}

bool BST::Postorder(TreeNode* start_node)
{
```

```
	if (start_node != nullptr)
	{
		if (start_node->left)
			Postorder(start_node->left);
		if (start_node->right)
			Postorder(start_node->right);
		cout << " " << start_node->data << " ";
	}
	return true;
}

bool BST::Insert(int item)
{
	TreeNode* node = new TreeNode;
	node->data = item;
	node->left = nullptr;
	node->right = nullptr;
	TreeNode* parent = nullptr;
	if (IsEmpty())
		root = node;
	// If not, we need to find the proper to-be parent of element
	else
	{
		TreeNode* current = root;
		while (current)
		{
			parent = current;
			if (node->data > current->data)
				current = current->right;
			else
				current = current->left;
		}
		if (node->data < parent->data)
			parent->left = node;
		else
			parent->right = node;
	}
	return true;
}

int main()
{
	BST bst;
	int choice_i, item_i;
	while (1)
	{
		cout << endl << endl;
		cout << "Binary Search Tree Operations" << endl;
		cout << "----------------------------" << endl;
		cout << "1. Insertion/Creation" << endl;
		cout << "2. In-Order Traversal" << endl;
```

```
		cout << "3. Pre-Order Traversal" << endl;
		cout << "4. Post-Order Traversal" << endl;
		cout << "5. Exit" << endl;
		cout << "Enter your choice: ";
		cin >> choice_i;
		switch (choice_i)
		{
			case 1:
				cout << "Enter Number to be inserted: ";
				cin >> item_i;
				if (bst.Insert(item_i))
					cout << "The element was inserted successfully into the 
tree.";
				break;
			case 2:
				cout << endl;
				cout << "In-Order Traversal" << endl;
				cout << "------------------" << endl;
				bst.Inorder(bst.root);
				break;
			case 3:
				cout << endl;
				cout << "Pre-Order Traversal" << endl;
				cout << "-------------------" << endl;
				bst.Preorder(bst.root);
				break;
			case 4:
				cout << endl;
				cout << "Post-Order Traversal" << endl;
				cout << "--------------------" << endl;
				bst.Postorder(bst.root);
				break;
			case 5:
				return 0;
			default:
				cout << "Invalid choice! Try again.";
		}
	}
}
```

Insertion of Elements in a Binary Search Tree

```
/* Implement a BST using linked lists with the insertion operation */
#include <iostream>

using namespace std;

class BST
```

```
{
	struct TreeNode
	{
		int data;
		TreeNode* left;
		TreeNode* right;
	};

public:
	TreeNode* root;

	BST()
	{
		root = nullptr;
	}

	bool IsEmpty()
	{
		return root == nullptr;
	}

	bool Inorder(TreeNode*);
	bool Insert(int);
};

bool BST::Inorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		if (start_node->left)
			Inorder(start_node->left);
		cout << " " << start_node->data << " ";
		if (start_node->right)
			Inorder(start_node->right);
	}
	return true;
}

bool BST::Insert(int item)
{
	TreeNode* node = new TreeNode;
	node->data = item;
	node->left = nullptr;
	node->right = nullptr;
	TreeNode* parent = nullptr;
	if (IsEmpty())
		root = node;
	// If not, we need to find the proper to-be parent of element
	else
	{
		TreeNode* current = root;
```

```
		while (current)
		{
			parent = current;
			if (node->data > current->data)
				current = current->right;
			else
				current = current->left;
		}
		if (node->data < parent->data)
			parent->left = node;
		else
			parent->right = node;
	}
	return true;
}

int main()
{
	BST bst;
	int choice_i, item_i;
	while (1)
	{
		cout << endl << endl;
		cout << "Binary Search Tree Operations" << endl;
		cout << "----------------------------" << endl;
		cout << "1. Insertion/Creation" << endl;
		cout << "2. In-Order Traversal" << endl;
		cout << "3. Exit" << endl;
		cout << "Enter your choice: ";
		cin >> choice_i;
		switch (choice_i)
		{
			case 1:
				cout << "Enter Number to be inserted: ";
				cin >> item_i;
				if (bst.Insert(item_i))
					cout << "The element was successfully inserted into the
tree.";
				break;
			case 2:
				cout << endl;
				cout << "In-Order Traversal" << endl;
				cout << "------------------" << endl;
				bst.Inorder(bst.root);
				break;
			case 3:
				return 0;
			default:
				cout << "Invalid choice! Try again.";
		}
	}
```

```
}
```

Deletion of an Element from a Binary Search Tree

```
/* Implement a BST using linked lists with deletion operation  */
#include <iostream>

using namespace std;

class BST
{
	struct TreeNode
	{
		int data;
		TreeNode* left;
		TreeNode* right;
	};

public:
	TreeNode* root;

	BST()
	{
		root = nullptr;
	}

	bool IsEmpty()
	{
		return root == nullptr;
	}

	bool Inorder(TreeNode*);
	bool Preorder(TreeNode*);
	bool Postorder(TreeNode*);
	bool Insert(int);
	bool Remove(int);
	TreeNode* SearchParent(int item);
	TreeNode* Min(TreeNode* start_node);
};

bool BST::Inorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		if (start_node->left)
			Inorder(start_node->left);
		cout << " " << start_node->data << " ";
```

```
		if (start_node->right)
			Inorder(start_node->right);
	}
	return true;
}

bool BST::Preorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		cout << " " << start_node->data << " ";
		if (start_node->left)
			Preorder(start_node->left);
		if (start_node->right)
			Preorder(start_node->right);
	}
	return true;
}

bool BST::Postorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		if (start_node->left)
			Postorder(start_node->left);
		if (start_node->right)
			Postorder(start_node->right);
		cout << " " << start_node->data << " ";
	}
	return true;
}

bool BST::Insert(int item)
{
	TreeNode* node = new TreeNode;
	node->data = item;
	node->left = nullptr;
	node->right = nullptr;
	TreeNode* parent = nullptr;
	if (IsEmpty())
		root = node;
	// If not, we need to find the proper to-be parent of element
	else
	{
		TreeNode* current = root;
		while (current)
		{
			parent = current;
			if (node->data > current->data)
				current = current->right;
			else
```

```
                    current = current->left;
            }
            if (node->data < parent->data)
                  parent->left = node;
            else
                  parent->right = node;
      }
      return true;
}

bool BST::Remove(int item)
{
      bool found = false;
      if (IsEmpty())
            return false;
      // If tree is not empty, find the element
      TreeNode* current = root;
      TreeNode* parent = nullptr;
      while (current)
      {
            if (current->data == item)
            {
                  found = true;
                  break;
            }
            parent = current;
            if (item > current->data)
                  current = current->right;
            else
                  current = current->left;
      }
      if (!found)
            return false;
      // If element was found, there can be 3 cases:
      // 1. We're removing a leaf node
      // 1.1 Is the left child of parent
      // 1.2 Is the right child of parent
      // 2. We're removing a node with only one child
      // 2.1 Only left child present
      // 2.2 Only right child present
      // 3. We're removing a node with two children

      // Node with no child
      if (current->left == nullptr && current->right == nullptr)
      {
            // Is the left child of parent or the right child?
            if (parent->left == current)
                  parent->left = nullptr;
            else
                  parent->right = nullptr;
            delete current;
```

```
		return true;
	}
	// Node with only left child
	if (current->left != nullptr && current->right == nullptr)
	{
		if (parent->left == current)
		{
			parent->left = current->left;
			delete current;
		}
		else
		{
			parent->right = current->left;
			delete current;
		}
		return true;
	}
	// Node with only right child
	if (current->right != nullptr && current->right == nullptr)
	{
		if (parent->left == current)
		{
			parent->left = current->right;
			delete current;
		}
		else
		{
			parent->right = current->right;
			delete current;
		}
		return true;
	}
	// Node with two children
	// ALGORITHM: Replace the deleted node with the smallest value in the right
	// sub-tree, now remove the smallest value from the right sub-tree to
	// remove the duplicate
	if (current->left != nullptr && current->right != nullptr)
	{
		TreeNode* right_subtree = current->right;
		// The right sub-tree has only a single node
		// Replace with it and remove the right sub-tree
		if (right_subtree->right == nullptr && right_subtree->left == nullptr)
		{
			current = right_subtree;
			delete right_subtree;
			current->right = nullptr;
		}
		// Right sub-tree has children, replace with the inorder predecessor
		else
		{
			// The node's right child has a left child
```

```
                    if (current->right->left != nullptr)
                    {
                          TreeNode* minimum = Min(current->right);
                          current->data = minimum->data;
                          delete minimum;
                          TreeNode* minimum_parent = SearchParent(minimum->data);
                          minimum_parent->left = nullptr;
                    }
                    // The node's right child has no left child
                    else
                    {
                          TreeNode* temp_node = current->right;
                          current->data = temp_node->data;
                          current->right = temp_node->right;
                          delete temp_node;
                    }
             }
       }
       return true;
}

BST::TreeNode* BST::SearchParent(int item)
{
       TreeNode* current = new TreeNode;
       while (current != nullptr)
       {
             if (item > current->right->data || item > current->left->data)
                    current = current->right;
             else if (item < current->right->data || item < current->left->data)
                    current = current->left;
             if (item == current->right->data || item == current->left->data)
                    return current;
       }
       return nullptr;
}

BST::TreeNode* BST::Min(TreeNode* start_node)
{
       TreeNode* current = new TreeNode;
       current = start_node;
       // Traverse to the leftmost leaf node
       while (current->left != nullptr)
             current = current->left;
       return current;
}

int main()
{
       BST bst;
       int choice_i, item_i;
       while (1)
```

```
	{
		cout << endl << endl;
		cout << " Binary Search Tree Operations" << endl;
		cout << " -----------------------------" << endl;
		cout << " 1. Insertion/Creation" << endl;
		cout << " 2. In-Order Traversal" << endl;
		cout << " 3. Pre-Order Traversal" << endl;
		cout << " 4. Post-Order Traversal" << endl;
		cout << " 5. Removal" << endl;
		cout << " 6. Exit" << endl;
		cout << " Enter your choice : ";
		cin >> choice_i;
		switch (choice_i)
		{
			case 1:
				cout << " Enter Number to be inserted : ";
				cin >> item_i;
				if (bst.Insert(item_i))
					cout << " The element was inserted into the tree.";
				break;
			case 2:
				cout << endl;
				cout << " In-Order Traversal" << endl;
				cout << " -------------------" << endl;
				bst.Inorder(bst.root);
				break;
			case 3:
				cout << endl;
				cout << " Pre-Order Traversal" << endl;
				cout << " -------------------" << endl;
				bst.Preorder(bst.root);
				break;
			case 4:
				cout << endl;
				cout << " Post-Order Traversal" << endl;
				cout << " --------------------" << endl;
				bst.Postorder(bst.root);
				break;
			case 5:
				cout << " Enter data to be deleted : ";
				cin >> item_i;
				if (bst.Remove(item_i))
					cout << "The element was deleted from the tree.";
				else
					cout << "The element was not found in the tree!";
				break;
			case 6:
				return 0;
			default:
				cout << " Invalid choice! Try again.";
		}
```

```
	}
}
```

Search for an Element in a Binary Search Tree

```
/* Implement a BST using linked lists with the search operation */
#include <iostream>

using namespace std;

class BST
{
	struct TreeNode
	{
		int data;
		TreeNode* left;
		TreeNode* right;
	};

public:
	TreeNode* root;
	// A node that can be used by non-member functions
	TreeNode* node;

	BST()
	{
		root = nullptr;
		node = nullptr;
	}

	bool IsEmpty()
	{
		return root == nullptr;
	}

	bool Inorder(TreeNode*);
	bool Insert(int);
	TreeNode* Search(int);
};

bool BST::Inorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		if (start_node->left)
			Inorder(start_node->left);
		cout << " " << start_node->data << " ";
```

```
		if (start_node->right)
			Inorder(start_node->right);
	}
	return true;
}

bool BST::Insert(int item)
{
	TreeNode* node = new TreeNode;
	node->data = item;
	node->left = nullptr;
	node->right = nullptr;
	TreeNode* parent = nullptr;
	if (IsEmpty())
		root = node;
	// If not, we need to find the proper to-be parent of element
	else
	{
		TreeNode* current = root;
		while (current)
		{
			parent = current;
			if (node->data > current->data)
				current = current->right;
			else
				current = current->left;
		}
		if (node->data < parent->data)
			parent->left = node;
		else
			parent->right = node;
	}
	return true;
}

BST::TreeNode* BST::Search(int item)
{
	TreeNode* current = new TreeNode();
	current = root;
	while (current != nullptr)
	{
		if (item > current->data)
			current = current->right;
		else if (item < current->data)
			current = current->left;
		if (item == current->data)
			return current;
	}
	return nullptr;
}
```

```
int main()
{
	BST bst;
	int choice_i, item_i;
	while (1)
	{
		cout << endl << endl;
		cout << "Binary Search Tree Operations" << endl;
		cout << "-----------------------------" << endl;
		cout << "1. Insertion/Creation" << endl;
		cout << "2. In-Order Traversal" << endl;
		cout << "3. Search for an element" << endl;
		cout << "4. Exit" << endl;
		cout << "Enter your choice: ";
		cin >> choice_i;
		switch (choice_i)
		{
			case 1:
				cout << "Enter Number to be inserted: ";
				cin >> item_i;
				if (bst.Insert(item_i))
					cout << "The element was inserted successfully into the
tree.";
				break;
			case 2:
				cout << endl;
				cout << "In-Order Traversal" << endl;
				cout << "-------------------" << endl;
				bst.Inorder(bst.root);
				break;
			case 3:
				cout << endl;
				cout << "Enter the element to search for: ";
				cin >> item_i;
				bst.node = bst.Search(item_i);
				if (bst.node != nullptr)
				{
					cout << item_i << " found in the tree!" << endl;
					cout << "Left child: " << bst.node->left->data << endl;
					cout << "Right child: " << bst.node->right->data << endl;
				}
				else
					cout << "The element " << item_i
					<< " wasn't found in the tree!" << endl;
				break;
			case 4:
				return 0;
			default:
				cout << "Invalid choice! Try again.";
		}
	}
```

```
}
```

Maximum and Minimum Elements in a Binary Search Tree

```
/* Find the maximum and minimum elements in a BST */
#include <iostream>

using namespace std;

class BST
{
	struct TreeNode
	{
		int data;
		TreeNode* left;
		TreeNode* right;
	};

public:
	TreeNode* root;

	BST()
	{
		root = nullptr;
	}

	bool IsEmpty()
	{
		return root == nullptr;
	}

	bool Inorder(TreeNode*);
	bool Preorder(TreeNode*);
	TreeNode* SearchParent(int item);
	bool Postorder(TreeNode*);
	bool Insert(int);
	bool Remove(int);
	TreeNode* Max(TreeNode*);
	TreeNode* Min(TreeNode*);
};

bool BST::Inorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		// Avoid recursion once the next element is found to be null
		if (start_node->left != nullptr)
```

```
			Inorder(start_node->left);
		cout << " " << start_node->data << " ";
		if (start_node->right != nullptr)
			Inorder(start_node->right);
	}
	return true;
}

bool BST::Postorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		// Avoid recursion once the next element is found to be null
		if (start_node->left != nullptr)
			Postorder(start_node->left);
		if (start_node->right != nullptr)
			Postorder(start_node->right);
		cout << " " << start_node->data << " ";
	}
	return true;
}

bool BST::Preorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		cout << " " << start_node->data << " ";
		// Avoid recursion once the next element is found to be null
		if (start_node->left != nullptr)
			Preorder(start_node->left);
		if (start_node->right != nullptr)
			Preorder(start_node->right);
	}
	return true;
}

BST::TreeNode* BST::SearchParent(int item)
{
	TreeNode* current = new TreeNode;
	while (current != nullptr)
	{
		if (item > current->right->data || item > current->left->data)
			current = current->right;
		else if (item < current->right->data || item < current->left->data)
			current = current->left;
		if (item == current->right->data || item == current->left->data)
			return current;
	}
	return nullptr;
}
```

```
bool BST::Insert(int item)
{
	TreeNode* new_node = new TreeNode;
	TreeNode* parent = new TreeNode();
	parent = nullptr;
	// Is this a new tree? If yes, new node will become the root
	if (IsEmpty())
		root = new_node;
	// If not, find the proper parent
	else
	{
		// All insertions occur as leaf nodes
		TreeNode* current = root;
		while (current != nullptr)
		{
			parent = current;
			if (new_node->data > current->data)
				current = current->right;
			else
				current = current->left;
		}
		if (new_node->data < parent->data)
			parent->left = new_node;
		else
			parent->right = new_node;
	}
	return true;
}

bool BST::Remove(int item)
{
	bool found = false;
	if (IsEmpty())
		return false;
	// If tree is not empty, find the element
	TreeNode* current = root;
	TreeNode* parent = nullptr;
	while (current)
	{
		if (current->data == item)
		{
			found = true;
			break;
		}
		parent = current;
		if (item > current->data)
			current = current->right;
		else
			current = current->left;
	}
	if (!found)
```

```
	return false;
// If element was found, there can be 3 cases:
// 1. We're removing a leaf node
// 1.1 Is the left child of parent
// 1.2 Is the right child of parent
// 2. We're removing a node with only one child
// 2.1 Only left child present
// 2.2 Only right child present
// 3. We're removing a node with two children

// Node with no child
if (current->left == nullptr && current->right == nullptr)
{
	// Is the left child of parent or the right child?
	if (parent->left == current)
		parent->left = nullptr;
	else
		parent->right = nullptr;
	delete current;
	return true;
}
// Node with only left child
if (current->left != nullptr && current->right == nullptr)
{
	if (parent->left == current)
	{
		parent->left = current->left;
		delete current;
	}
	else
	{
		parent->right = current->left;
		delete current;
	}
	return true;
}
// Node with only right child
if (current->right != nullptr && current->right == nullptr)
{
	if (parent->left == current)
	{
		parent->left = current->right;
		delete current;
	}
	else
	{
		parent->right = current->right;
		delete current;
	}
	return true;
}
```

```
        // Node with two children
        // ALGORITHM: Replace the deleted node with the smallest value in the right
        // sub-tree, now remove the smallest value from the right sub-tree to
        // remove the duplicate
        if (current->left != nullptr && current->right != nullptr)
        {
                TreeNode* right_subtree = current->right;
                // The right sub-tree has only a single node
                // Replace with it and remove the right sub-tree
                if (right_subtree->right == nullptr && right_subtree->left == nullptr)
                {
                        current = right_subtree;
                        delete right_subtree;
                        current->right = nullptr;
                }
                // Right sub-tree has children, replace with the inorder predecessor
                else
                {
                        // The node's right child has a left child
                        if (current->right->left != nullptr)
                        {
                                TreeNode* minimum = Min(current->right);
                                current->data = minimum->data;
                                delete minimum;
                                TreeNode* minimum_parent = SearchParent(minimum->data);
                                minimum_parent->left = nullptr;
                        }
                        // The node's right child has no left child
                        else
                        {
                                TreeNode* temp_node = current->right;
                                current->data = temp_node->data;
                                current->right = temp_node->right;
                                delete temp_node;
                        }
                }
        }
        return true;
}

BST::TreeNode* BST::Max(TreeNode* start_node)
{
        TreeNode* current = new TreeNode();
        current = start_node;
        // Traverse to the rightmost leaf node
        while (current->right != nullptr)
                current = current->right;
        return current;
}

BST::TreeNode* BST::Min(TreeNode* start_node)
```

```
{
	TreeNode* current = new TreeNode;
	current = start_node;
	// Traverse to the leftmost leaf node
	while (current->left != nullptr)
		current = current->left;
	return current;
}

int main()
{
	BST bst;
	int choice_i, item_i;
	while (1)
	{
		cout << endl << endl;
		cout << "Binary Search Tree Operations" << endl;
		cout << "-----------------------------" << endl;
		cout << "1. Insertion/Creation" << endl;
		cout << "2. In-Order Traversal" << endl;
		cout << "3. Pre-Order Traversal" << endl;
		cout << "4. Post-Order Traversal" << endl;
		cout << "5. Minimum Element" << endl;
		cout << "6. Maximum Element" << endl;
		cout << "7. Removal" << endl;
		cout << "8. Exit" << endl;
		cout << "Enter your choice: ";
		cin >> choice_i;
		switch (choice_i)
		{
			case 1:
				cout << "Enter Number to be inserted: ";
				cin >> item_i;
				if (bst.Insert(item_i))
					cout << "The element was inserted in the tree.";
				break;
			case 2:
				cout << endl;
				cout << "In-Order Traversal" << endl;
				cout << "------------------" << endl;
				bst.Inorder(bst.root);
				break;
			case 3:
				cout << endl;
				cout << "Pre-Order Traversal" << endl;
				cout << "-------------------" << endl;
				bst.Preorder(bst.root);
				break;
			case 4:
				cout << endl;
				cout << "Post-Order Traversal" << endl;
```

```
					cout << "--------------------" << endl;
					bst.Postorder(bst.root);
					break;
				case 5:
					cout << endl;
					cout << "Minimum element in the tree is "
						<< bst.Min(bst.root)->data;
					break;
				case 6:
					cout << endl;
					cout << "Maximum element in the tree is "
						<< bst.Max(bst.root)->data;
					break;
				case 7:
					cout << endl;
					cout << "Enter the element to be deleted: ";
					cin >> item_i;
					if (bst.Remove(item_i) == true)
						cout << "The element was deleted successfully.";
					else
						cout << "The element was not found in the tree.";
					break;
				case 8:
					return 0;
				default:
					cout << "Invalid choice! Try again.";
			}
		}
}
```

Predecessor and Successor of an Element in a Binary Search Tree

```
/* Find the successors and predecessors of a given node in a BST */
#include <iostream>

using namespace std;

class BST
{
	struct TreeNode
	{
		int data;
		TreeNode* left;
		TreeNode* right;
	};

public:
```

```
	TreeNode* root;

	BST()
	{
		root = nullptr;
	}

	bool IsEmpty()
	{
		return root == nullptr;
	}

	bool Inorder(TreeNode*);
	bool Preorder(TreeNode*);
	bool Postorder(TreeNode*);
	bool Insert(int);
	bool Remove(int);
	TreeNode* Max(TreeNode*);
	TreeNode* Min(TreeNode*);
	TreeNode* Predecessor(int, int);
	TreeNode* Successor(int, int);
	TreeNode* Search(int);
	TreeNode* SearchParent(int);
};

bool BST::Inorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		// Avoid recursion once the next element is found to be null
		if (start_node->left != nullptr)
			Inorder(start_node->left);
		cout << " " << start_node->data << " ";
		if (start_node->right != nullptr)
			Inorder(start_node->right);
	}
	return true;
}

bool BST::Postorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		// Avoid recursion once the next element is found to be null
		if (start_node->left != nullptr)
			Postorder(start_node->left);
		if (start_node->right != nullptr)
			Postorder(start_node->right);
		cout << " " << start_node->data << " ";
	}
	return true;
```

```
}

bool BST::Preorder(TreeNode* start_node)
{
	if (start_node != nullptr)
	{
		cout << " " << start_node->data << " ";
		// Avoid recursion once the next element is found to be null
		if (start_node->left != nullptr)
			Preorder(start_node->left);
		if (start_node->right != nullptr)
			Preorder(start_node->right);
	}
	return true;
}

bool BST::Insert(int item)
{
	TreeNode* new_node = new TreeNode;
	TreeNode* parent = new TreeNode;
	// Is this a new tree? If yes, new node will become the root
	if (IsEmpty())
		root = new_node;
	// If not, find the proper parent
	else
	{
		// All insertions occur as leaf nodes
		TreeNode* current = root;
		while (current != nullptr)
		{
			parent = current;
			if (new_node->data > current->data)
				current = current->right;
			else
				current = current->left;
		}
		if (new_node->data < parent->data)
			parent->left = new_node;
		else
			parent->right = new_node;
	}
	return true;
}

bool BST::Remove(int item)
{
	bool found = false;
	if (IsEmpty())
	{
		cout << "This tree is empty!" << endl;
		return false;
```

```
}
// If tree is not empty, find the element
TreeNode* current = root;
TreeNode* parent = nullptr;
while (current)
{
	if (current->data == item)
	{
		found = true;
		break;
	}
	parent = current;
	if (item > current->data)
		current = current->right;
	else
		current = current->left;
}
if (!found)
{
	cout << "Data not found in the tree!" << endl;
	return false;
}
// If element was found, there can be 3 cases:
// 1. We're removing a leaf node
// 1.1 Is the left child of parent
// 1.2 Is the right child of parent
// 2. We're removing a node with only one child
// 2.1 Only left child present
// 2.2 Only right child present
// 3. We're removing a node with two children

// Node with no child
if (current->left == nullptr && current->right == nullptr)
{
	// Is the left child of parent or the right child?
	if (parent->left == current)
		parent->left = nullptr;
	else
		parent->right = nullptr;
	delete current;
	return true;
}
// Node with only left child
if (current->left != nullptr && current->right == nullptr)
{
	if (parent->left == current)
	{
		parent->left = current->left;
		delete current;
	}
	else
```

```
        {
                parent->right = current->left;
                delete current;
        }
        return true;
}
// Node with only right child
if (current->right != nullptr && current->right == nullptr)
{
        if (parent->left == current)
        {
                parent->left = current->right;
                delete current;
        }
        else
        {
                parent->right = current->right;
                delete current;
        }
        return true;
}
// Node with two children
// ALGORITHM: Replace the deleted node with the smallest value in the right
// sub-tree, now remove the smallest value from the right sub-tree to
// remove the duplicate
if (current->left != nullptr && current->right != nullptr)
{
        TreeNode* right_subtree = current->right;
        // The right sub-tree has only a single node
        // Replace with it and remove the right sub-tree
        if (right_subtree->right == nullptr && right_subtree->left == nullptr)
        {
                current = right_subtree;
                delete right_subtree;
                current->right = nullptr;
        }
        // Right sub-tree has children, replace with the inorder predecessor
        else
        {
                // The node's right child has a left child
                if (current->right->left != nullptr)
                {
                        TreeNode* minimum = Min(current->right);
                        current->data = minimum->data;
                        delete minimum;
                        TreeNode* minimum_parent = SearchParent(minimum->data);
                        minimum_parent->left = nullptr;
                }
                // The node's right child has no left child
                else
                {
```

```
                    TreeNode* temp_node = current->right;
                    current->data = temp_node->data;
                    current->right = temp_node->right;
                    delete temp_node;
                }
            }
    }
    return true;
}

BST::TreeNode* BST::Search(int item)
{
    TreeNode* current = new TreeNode;
    while (current != nullptr)
    {
        if (item > current->data)
            current = current->right;
        else if (item < current->data)
            current = current->left;
        if (item == current->data)
            return current;
    }
    return nullptr;
}

BST::TreeNode* BST::SearchParent(int item)
{
    TreeNode* current = new TreeNode;
    while (current != nullptr)
    {
        if (item > current->right->data || item > current->left->data)
            current = current->right;
        else if (item < current->right->data || item < current->left->data)
            current = current->left;
        if (item == current->right->data || item == current->left->data)
            return current;
    }
    return nullptr;
}

BST::TreeNode* BST::Max(TreeNode* start_node)
{
    TreeNode* current = new TreeNode;
    current = start_node;
    // Traverse to the rightmost leaf node
    while (current->right != nullptr)
        current = current->right;
    return current;
}

BST::TreeNode* BST::Min(TreeNode* start_node)
```

```
{
	TreeNode* current = new TreeNode;
	current = start_node;
	// Traverse to the leftmost leaf node
	while (current->left != nullptr)
		current = current->left;
	return current;
}

BST::TreeNode* BST::Predecessor(int item, int mode)
{
	// Search for the start_node
	TreeNode* start_node = new TreeNode;
	start_node = Search(item);
	switch (mode)
	{
		// Pre-Order Predecessor
		case 1:
		{
			// If start_node is root of tree, predecessor is undefined
			if (start_node == root)
				return nullptr;
			// If start_node has left sibling ls then predecessor is
			// the rightmost descendant of ls
			TreeNode* parent = SearchParent(start_node->data);
			if (parent->right == start_node && parent->left != nullptr)
				return Max(parent->left);
			// Else the parent
			return parent;
		}
		// In-Order Predecessor
		case 2:
		{
			// If start_node has left  child l, then predecessor is rightmost
			// descendant of l
			if (start_node->left != nullptr)
				return Max(start_node->left);
			// Predecessor is the closest ancestor v of start_node such that
			// start_node is in the right subtree of v
			TreeNode* parent = SearchParent(start_node->data);
			if (parent->right == start_node)
				return parent;
			// If not an immediate right child, recurse
			TreeNode* subtree_node = parent;
			while (true)
			{
				if (subtree_node->right == parent)
					return subtree_node;
				parent = subtree_node;
				// If we reached the top and still couldn't find it, give up
				if (parent == root)
```

```
                        return nullptr;
                    subtree_node = SearchParent(subtree_node->data);
                }
            }
            // Post-Order Predecessor
            case 3:
            {
                // If start_node has a right child
                // then the predecessor is the right child
                if (start_node->right != nullptr)
                    return start_node->right;
                // If start_node has a left child
                // then the predecessor is the left child
                if (start_node->left != nullptr)
                    return start_node->left;
                // If start_node has left sibling then predecessor is the sibling
                TreeNode* parent = SearchParent(start_node->data);
                if (parent->left != nullptr)
                    return parent->left;
                // If start_node has an ancestor which:
                // is a right child AND has a left sibling vls then pred is vls
                TreeNode* ancestor = SearchParent(parent->data);
                if (ancestor->right == parent && ancestor->left != nullptr)
                    return ancestor->left;
                // If not an immediate ancestor, recurse
                while (true)
                {
                    if (ancestor->right == parent && ancestor->left != nullptr)
                        return ancestor->left;
                    parent = ancestor;
                    // If we reached the top and still couldn't find it, give up
                    if (parent == root)
                        return nullptr;
                    ancestor = SearchParent(ancestor->data);
                }
            }
            default:
                break;
        }
        return nullptr;
}

BST::TreeNode* BST::Successor(int item, int mode)
{
        // Search for the start_node
        TreeNode* start_node = new TreeNode;
        start_node = Search(item);
        switch (mode)
        {
            // Pre-Order Successor
            case 1:
```

```
{
        // If start_node has a left child then successor is the left child.
        if (start_node->left != nullptr)
                return start_node->left;
        // If start_node has a right child then successor is right child
        if (start_node->right != nullptr)
                return start_node->right;
        // If the start_node is a leaf
        // 1. And a left child and has a right sibling rs, rs is successor
        TreeNode* parent = SearchParent(item);
        if (start_node == parent->left)
                if (parent->right != nullptr)
                        return parent->right;
        // 2. start_node has an ancestor which is a left child and
        // has a right sibling then the sibling is the successor
        TreeNode* ancestor = SearchParent(parent->data);
        if (ancestor->left == parent && ancestor->right != nullptr)
                return ancestor->right;
        // If not an immediate ancestor, recurse
        while (true)
        {
                if (ancestor->left == parent && ancestor->right != nullptr)
                        return ancestor->right;
                parent = ancestor;
                // If we reached the top and still couldn't find it, give up
                if (parent == root)
                        return nullptr;
                ancestor = SearchParent(ancestor->data);
        }
}
// In-Order Successor
case 2:
{
        // If start_node has a right child then successor is the leftmost
        // descendant of start_node
        if (start_node->right != nullptr)
                return Min(start_node);
        // Else the closest ancestor of start_node such that start_node
        // is in the left subtree of the ancestor
        TreeNode* parent = SearchParent(start_node->data);
        if (parent->right == start_node)
                return parent;
        // If not an immediate right child, recurse
        TreeNode* subtree_node = parent;
        while (true)
        {
                if (subtree_node->right == parent)
                        return subtree_node;
                parent = subtree_node;
                // If we reached the top and still couldn't find it, give up
                if (parent == root)
```

```
                                break;
                        subtree_node = SearchParent(subtree_node->data);
                }
                return nullptr;
        }
        // Post-Order Successor
        case 3:
        {
                // If start_node is the root, the successor is undefined
                if (start_node == root)
                        return nullptr;
                // If start_node is a right child, the successor is it's parent
                TreeNode* parent = SearchParent(item);
                if (start_node == parent->right)
                        return parent;
                // If start_node is a left child and has a right sibling rs
                // the successor is the leftmost leaf in the rs's subtree
                if (start_node == parent->left)
                        if (parent->right != nullptr)
                                return Min(start_node);
                // Else the successor is the parent of start_node
                return parent;
        }
        default:
                break;
    }
    return nullptr;
}

int main()
{
    BST bst;
    int choice_i, item_i, mode_i;
    while (true)
    {
        cout << endl << endl;
        cout << "Binary Search Tree Operations" << endl;
        cout << "-----------------------------" << endl;
        cout << "1. Insertion/Creation" << endl;
        cout << "2. In-Order Traversal" << endl;
        cout << "3. Pre-Order Traversal" << endl;
        cout << "4. Post-Order Traversal" << endl;
        cout << "5. Predecessor" << endl;
        cout << "6. Successor" << endl;
        cout << "7. Removal" << endl;
        cout << "8. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice_i;
        switch (choice_i)
        {
                case 1:
```

```
            cout << "Enter Number to be inserted: ";
            cin >> item_i;
            bst.Insert(item_i);
            break;
    case 2:
            cout << endl;
            cout << "In-Order Traversal" << endl;
            cout << "------------------" << endl;
            bst.Inorder(bst.root);
            break;
    case 3:
            cout << endl;
            cout << "Pre-Order Traversal" << endl;
            cout << "-------------------" << endl;
            bst.Preorder(bst.root);
            break;
    case 4:
            cout << endl;
            cout << "Post-Order Traversal" << endl;
            cout << "--------------------" << endl;
            bst.Postorder(bst.root);
            break;
    case 5:
            cout << endl;
            cout << "Enter the element whose predecessor you want: "
                    << endl;
            cin >> item_i;
            cout << "1. Pre-Order predecessor" << endl;
            cout << "2. In-Order predecessor" << endl;
            cout << "3. Post-Order predecessor" << endl;
            cin >> mode_i;
            cout << "The predecessor of " << item_i << " is "
                    << bst.Predecessor(item_i, mode_i);
            break;
    case 6:
            cout << endl;
            cout << "Enter the element whose successor you want: "
                    << endl;
            cin >> item_i;
            cout << "1. Pre-Order successor" << endl;
            cout << "2. In-Order successor" << endl;
            cout << "3. Post-Order successor" << endl;
            cin >> mode_i;
            cout << "The successor of " << item_i << " is "
                    << bst.Successor(item_i, mode_i);
            break;
    case 7:
            cout << "Enter data to be deleted: ";
            cin >> item_i;
            bst.Remove(item_i);
            break;
```

```
			case 8:
				return 0;
			default:
				cout << "Invalid choice! Try again.";
		}
	}
}
```

Implementation of a Huffman Tree to Perform Huffman Encoding of a String

```
#include <iostream>
#include <string>
#include <fstream>
#include <array>

// Maximum height of the Huffman Tree
#define MAX_TREE_HT 100

using namespace std;

class Huffman
{
public:
	// A Huffman Tree node
	struct minHeapNode
	{
		char data;
		int freq;
		minHeapNode* left;
		minHeapNode* right;
	};

	// The Huffman Tree itself
	struct minHeap
	{
		int current_size;
		int capacity;
		minHeapNode** array; // Array of minHeap node pointers
	};

	// Allocate a new minHeap node with passed character and freq
	minHeapNode* newNode(unsigned char, int);
	// Create a minHeap of passed capacity
	minHeap* create(int);
	// Swap two minHeap nodes
	void swapNode(minHeapNode**, minHeapNode**);
	// Heapify
```

```
	void heapify(minHeap*, int);

	// Returns true if current size of minHeap is 1
	bool isSizeOne(minHeap* min_heap)
	{
		if (min_heap->current_size == 1)
			return true;
		return false;
	}

	// Find minimum node
	minHeapNode* getMin(minHeap*);
	// Insert a new node in the minHeap
	void insert(minHeap*, minHeapNode*);
	// Build a minHeap
	void build(minHeap*);

	// Return true if it a leaf
	bool isLeaf(minHeapNode* root)
	{
		if (!root->left && !root->right)
			return true;
		return false;
	}

	// Creates a minHeap of capacity equal to size and insert all character of
	// data[] in minHeap. Initially size of minHeap is equal to capacity.
	minHeap* generate(unsigned char[], int[], int);
	// The main function that builds Huffman tree
	minHeapNode* buildHuffmanTree(unsigned char[], int[], int);
	// Print the huffman coded input
	void printCodes(minHeapNode*, int[], int);
	// Builds a Huffman Tree and print codes by traversing it
	void huffmanCodes(unsigned char[], int[], int);
};

Huffman::minHeapNode* Huffman::newNode(unsigned char data, int freq)
{
	// Initialise a node with the passed parameters
	minHeapNode* new_node = new minHeapNode;
	new_node->left = new_node->right = nullptr;
	new_node->data = data;
	new_node->freq = freq;
	return new_node;
}

void printArray(int arr[], int n)
{
	for (int i = 0; i < n; ++i)
		cout << arr[i] << " ";
	cout << endl;
```

```
}

Huffman::minHeap* Huffman::create(int capacity)
{
	minHeap* min_heap = new minHeap;
	min_heap->current_size = 0; // current size is 0
	min_heap->capacity = capacity;
	min_heap->array = new minHeapNode*;
	// (minHeapNode**)malloc(min_heap->capacity * sizeof(minHeapNode*));
	return min_heap;
}

void Huffman::swapNode(minHeapNode** a, minHeapNode** b)
{
	minHeapNode* t = *a;
	*a = *b;
	*b = t;
}

void Huffman::heapify(minHeap* minHeap, int idx)
{
	int smallest = idx;
	int left = 2 * idx + 1;
	int right = 2 * idx + 2;
	// The standard heapify algorithm
	if (left < minHeap->current_size &&
		minHeap->array[left]->freq < minHeap->array[smallest]->freq)
		smallest = left;

	if (right < minHeap->current_size &&
		minHeap->array[right]->freq < minHeap->array[smallest]->freq)
		smallest = right;

	if (smallest != idx)
	{
		swapNode(&minHeap->array[smallest], &minHeap->array[idx]);
		heapify(minHeap, smallest);
	}
}

Huffman::minHeapNode* Huffman::getMin(minHeap* min_heap)
{
	minHeapNode* minimum = min_heap->array[0];
	min_heap->array[0] = min_heap->array[min_heap->current_size - 1];
	min_heap->current_size--;
	heapify(min_heap, 0);
	return minimum;
}

void Huffman::insert(minHeap* min_heap, minHeapNode* min_heapNode)
{
```

```
	++min_heap->current_size;
	int i = min_heap->current_size - 1;
	while (i && min_heapNode->freq < min_heap->array[(i - 1) / 2]->freq)
	{
		min_heap->array[i] = min_heap->array[(i - 1) / 2];
		i = (i - 1) / 2;
	}
	min_heap->array[i] = min_heapNode;
}

void Huffman::build(minHeap* min_heap)
{
	int n = min_heap->current_size - 1;
	for (int i = (n - 1) / 2; i >= 0; --i)
		heapify(min_heap, i);
}

Huffman::minHeap* Huffman::generate(unsigned char data[], int freq[], int size)
{
	minHeap* min_heap = create(size);
	for (int i = 0; i < size; ++i)
		min_heap->array[i] = newNode(data[i], freq[i]);
	min_heap->current_size = size;
	build(min_heap);
	return min_heap;
}

Huffman::minHeapNode* Huffman::buildHuffmanTree(unsigned char data[], int freq[], int size)
{
	minHeapNode *left, *right, *top;

	// Create a min heap of capacity equal to size.  Initially, there are
	// modes equal to size.
	minHeap* minHeap = generate(data, freq, size);

	// Iterate while size of heap doesn't become 1
	while (!isSizeOne(minHeap))
	{
		// Extract the two minimum freq items from min heap
		left = getMin(minHeap);
		right = getMin(minHeap);

		// Create a new internal node with freq equal to the
		// sum of the two nodes frequencies. Make the two extracted node as
		// left and right children of this new node.
		// Add this node to the min heap
		top = newNode('$', left->freq + right->freq);
		// Unused symbol $ to mark internal nodes
		top->left = left;
		top->right = right;
		insert(minHeap, top);
```

```
	}
	// The remaining node is the root node and the tree is complete.
	return getMin(minHeap);
}

void Huffman::printCodes(minHeapNode* root, int arr[], int top)
{
	// Assign 0 to left edge and recurse
	if (root->left)
	{
		arr[top] = 0;
		printCodes(root->left, arr, top + 1);
	}

	// Assign 1 to right edge and recurse
	if (root->right)
	{
		arr[top] = 1;
		printCodes(root->right, arr, top + 1);
	}

	// If this is a leaf node, then it contains one of the input
	// characters, print the character and its code from arr[]
	if (isLeaf(root))
	{
		cout << root->data << " ";
		printArray(arr, top);
	}
}

void Huffman::huffmanCodes(unsigned char data[], int freq[], int size)
{
	minHeapNode* root = buildHuffmanTree(data, freq, size);
	int arr[MAX_TREE_HT], top = 0;
	printCodes(root, arr, top);
}

int main()
{
	cout << "Enter the filename of the file to be Huffman coded: ";
	string inputFile;
	cin >> inputFile;
	ifstream inFile(inputFile);
	string inputbuffer;
	unsigned char tokens[255];
	int freq[255];
	for (int i = 0; i < 255; i++)
	{
		freq[i] = 0;
		tokens[i] = i;
	}
```

```
	Huffman hf;
	while (!inFile.eof())
	{
		getline(inFile, inputbuffer);
		// Find frequency of all possible 256 characters
		for (unsigned int i = 0; i < inputbuffer.length(); i++)
		{
			tokens[inputbuffer[i]] = inputbuffer[i];
			freq[inputbuffer[i]]++;
		}
		unsigned char new_tokens[255];
		int new_freq[255];
		int numoftokens = 0;
		// Filter out only those tokens which have a non-zero frequency
		for (int i = 0; i < 255; i++)
		{
			if (freq[i] != 0)
			{
				new_tokens[numoftokens] = tokens[i];
				new_freq[numoftokens] = freq[i];
				numoftokens++;
			}
		}
		int size = sizeof new_tokens / sizeof new_tokens[0];
		// Encode and print the coded output
		hf.huffmanCodes(new_tokens, new_freq, numoftokens);
	}
}
```

Implementation of an Expression Tree

```
/* Evaluate a postfix expression using expression tree */
#include <iostream>

using namespace std;

struct tree
{
	char data;
	tree* left;
	tree* right;
};

int top = -1;
tree* stack[20];
tree* node;
```

```
void push(tree* node)
{
	stack[++top] = node;
}

tree* pop()
{
	return stack[top--];
}

int check(char c)
{
	// Return 2 if operator otherwise 1
	if (c == '+' || c == '-' || c == '/' || c == '*')
		return 2;
	return 1;
}

int cal(tree* node)
{
	int ch;
	// Check if operand or operator
	ch = check(node->data);
	// If it is an operand, convert it to the corresponding integer by
	// subtracting 48 from it's ascii value
	if (ch == 1)
		return (node->data - 48);
	if (ch == 2)
	{
		if (node->data == '+')
			return (cal(node->left) + cal(node->right));
		if (node->data == '-')
			return (cal(node->right) - cal(node->left));
		if (node->data == '*')
			return (cal(node->left) * cal(node->right));
		if (node->data == '/')
			return (cal(node->right) / cal(node->left));
	}
}

void operands(char b)
{
	node = new tree;
	node->data = b;
	node->left = nullptr;
	node->right = nullptr;
	push(node);
}

void operators(char a)
{
```

```
        node = new tree;
        node->data = a;
        node->left = pop();
        node->right = pop();
        push(node);
}

// Perform in-order traversal to evaulate the expression tree
void traverse(tree* node)
{
        if (node != nullptr)
        {
                traverse(node->right);
                printf("%c", node->data);
                traverse(node->left);
        }
}

int main()
{
        int i, p, ans;
        char s[20];
        cout << "Enter the expression tree in postfix form: ";
        fgets(s, 19, stdin);
        for (i = 0; s[i] != '\n'; i++)
        {
                p = check(s[i]);
                if (p == 1)
                        operands(s[i]);
                else if (p == 2)
                        operators(s[i]);
        }
        ans = cal(stack[top]);
        cout << endl << "The value of the postfix expression = " << ans << endl;
        cout << "The actual traversal will be:" << endl;
        traverse(stack[top]);
}
```

Implementation of a B-Tree and it’s Traversal

```
/* C++ Program to Implement B-Tree */
#include <iostream>

using namespace std;

struct BTreeNode
{
```

```
	int *data;
	BTreeNode **child_ptr;
	bool leaf;
	int n;
}*root = nullptr, *np = nullptr, *x = nullptr;

BTreeNode* init()
{
	int i;
	np = new BTreeNode;
	np->data = new int[5];
	np->child_ptr = new BTreeNode*[6];
	np->leaf = true;
	np->n = 0;
	for (i = 0; i < 6; i++)
		np->child_ptr[i] = nullptr;
	return np;
}

void traverse(BTreeNode* p)
{
	cout << endl;
	int i;
	for (i = 0; i < p->n; i++)
	{
		if (p->leaf == false)
			traverse(p->child_ptr[i]);
		cout << " " << p->data[i];
	}
	if (p->leaf == false)
		traverse(p->child_ptr[i]);
	cout << endl;
}

void sort(int* p, int n)
{
	int i, j, temp;
	for (i = 0; i < n; i++)
	{
		for (j = i; j <= n; j++)
		{
			if (p[i] > p[j])
			{
				temp = p[i];
				p[i] = p[j];
				p[j] = temp;
			}
		}
	}
}
```

```
int split_child(BTreeNode* x, int i)
{
	int j, mid;
	BTreeNode *np1, *np3, *y;
	np3 = init();
	np3->leaf = true;
	if (i == -1)
	{
		mid = x->data[2];
		x->data[2] = 0;
		x->n--;
		np1 = init();
		np1->leaf = false;
		x->leaf = true;
		for (j = 3; j < 5; j++)
		{
			np3->data[j - 3] = x->data[j];
			np3->child_ptr[j - 3] = x->child_ptr[j];
			np3->n++;
			x->data[j] = 0;
			x->n--;
		}
		for (j = 0; j < 6; j++)
			x->child_ptr[j] = nullptr;
		np1->data[0] = mid;
		np1->child_ptr[np1->n] = x;
		np1->child_ptr[np1->n + 1] = np3;
		np1->n++;
		root = np1;
	}
	else
	{
		y = x->child_ptr[i];
		mid = y->data[2];
		y->data[2] = 0;
		y->n--;
		for (j = 3; j < 5; j++)
		{
			np3->data[j - 3] = y->data[j];
			np3->n++;
			y->data[j] = 0;
			y->n--;
		}
		x->child_ptr[i + 1] = y;
		x->child_ptr[i + 1] = np3;
	}
	return mid;
}

void insert(int a)
{
```

```
int i, temp;
x = root;
if (x == nullptr)
{
	root = init();
	x = root;
}
else
{
	if (x->leaf == true && x->n == 5)
	{
		split_child(x, -1);
		x = root;
		for (i = 0; i < x->n; i++)
		{
			if (a > x->data[i] && a < x->data[i + 1])
			{
				i++;
				break;
			}
			if (a < x->data[0])
				break;
		}
		x = x->child_ptr[i];
	}
	else
	{
		while (x->leaf == false)
		{
			for (i = 0; i < x->n; i++)
			{
				if (a > x->data[i] && a < x->data[i + 1])
				{
					i++;
					break;
				}
				if (a < x->data[0])
					break;
			}
			if (x->child_ptr[i]->n == 5)
			{
				temp = split_child(x, i);
				x->data[x->n] = temp;
				x->n++;
			}
			else
				x = x->child_ptr[i];
		}
	}
}
x->data[x->n] = a;
```

```
	sort(x->data, x->n);
	x->n++;
}

int main()
{
	int i, n, t;
	cout << "Enter the no of elements to be inserted: ";
	cin >> n;
	for (i = 0; i < n; i++)
	{
		cout << "Enter the element: ";
		cin >> t;
		insert(t);
	}
	cout << "Traversal of constructed tree:" << endl;
	traverse(root);
}
```

Bubble Sort

```
/* Implement Bubble Sort */
#include <iostream>

using namespace std;

int main()
{
	cout << "Enter the number of elements in the array: ";
	int n;
	cin >> n;
	int array[50];
	for (int i = 0; i < n; i++)
	{
		cout << "Enter the " << i + 1 << "th element: ";
		cin >> array[i];
	}
	for (int x = 0; x < n; x++)
	{
		for (int y = 0; y < n - 1; y++)
		{
			if (array[y] > array[y + 1])
			{
				int temp = array[y + 1];
				array[y + 1] = array[y];
				array[y] = temp;
			}
```

```
            }
        }
        cout << "The Sorted Array is:" << endl;
        for (int i = 0; i < n; i++)
            cout << " " << array[i] << " ";
}
```

Merge Sort

```
/* Implement Merge Sort */
#include <iostream>

using namespace std;

int max(int x, int y)
{
        if (x > y)
            return x;
        return y;
}

// Left is the index of the leftmost element of the subarray.
// Right is one past the index of the rightmost element
void merge(int* input, int left, int right, int* scratch)
{
        // The non-base case, if anything other than this, it must be the base case
        // And in that case we will just return
        if (right != left + 1)
        {
            int i = 0;
            int length = right - left;
            int midpoint_distance = length / 2;
            int l = left, r = left + midpoint_distance;

            // Sort each subarray
            merge(input, left, left + midpoint_distance, scratch);
            merge(input, left + midpoint_distance, right, scratch);

            // Merge the arrays together using scratch for temporary storage
            for (i = 0; i < length; i++)
            {
                // Check to see if any elements remain in the left array
                // If so, we check if there are elements left in the right array
                // If so, we compare them. Otherwise, we know that the merge must
                // use the element from the left array
                if (l < left + midpoint_distance &&
                    (r == right || max(input[l], input[r]) == input[l]))
```

```
                {
                        scratch[i] = input[l];
                        l++;
                }
                else
                {
                        scratch[i] = input[r];
                        r++;
                }
            }
            // Copy the sorted subarray back to the input
            for (i = left; i < right; i++)
                input[i] = scratch[i - left];
        }
}

bool mergesort(int* input, int size)
{
        int* scratch = new int;
        // (int *)malloc(size * sizeof(int));
        if (scratch != nullptr)
        {
                merge(input, 0, size, scratch);
                free(scratch);
                return true;
        }
        return false;
}

int main()
{
        int array[100], n;
        cout << "Enter the size of the array: ";
        cin >> n;
        for (int i = 0; i < n; i++)
        {
                cout << "Enter the " << i + 1 << "th element: ";
                cin >> array[i];
        }
        if (mergesort(array, n))
        {
                cout << "The sorted array is:" << endl;
                for (int i = 0; i < n; i++)
                        cout << " " << array[i] << " ";
        }
        else
                cout << "The sorting failed!" << endl;
}
```

Heap Sort

```
#include <iostream>

using namespace std;

const int MAX = 10;

class Heap
{
	int arr[MAX];
	int count;
public:
	Heap()
	{
		count = 0;
		for (int i = 0; i < MAX; i++)
			arr[MAX] = 0;
	}

	void Add(int num);
	void MakeHeap(int);
	void HeapSort();
	void Display();
};

void Heap::Add(int num)
{
	if (count < MAX)
	{
		arr[count] = num;
		count++;
	}
	else
		cout << endl << "Array is full!" << endl;
}

void Heap::MakeHeap(int c)
{
	for (int i = 1; i < c; i++)
	{
		int val = arr[i];
		int s = i;
		int f = (s - 1) / 2;
		while (s > 0 && arr[f] < val)
		{
			arr[s] = arr[f];
```

```
                s = f;
                f = (s - 1) / 2;
            }
            arr[s] = val;
        }
}

void Heap::HeapSort()
{
        for (int i = count - 1; i > 0; i--)
        {
                int ivalue = arr[i];
                arr[i] = arr[0];
                arr[0] = ivalue;
                MakeHeap(i);
        }
}

void Heap::Display()
{
        for (int i = 0; i < count; i++)
                cout << arr[i] << "\t";
        cout << endl;
}

int main()
{
        Heap arr;
        int size;
        cout << "Enter the size of the heap: ";
        cin >> size;
        for (int i = 0; i < size; i++)
        {
                int elem;
                cout << "Enter " << i + 1 << "th element: ";
                cin >> elem;
                arr.Add(elem);
        }
        arr.MakeHeap(size);
        cout << endl << "Heap Sort:" << endl;
        cout << endl << "Before Sorting:" << endl;
        arr.Display();
        arr.HeapSort();
        cout << endl << "After Sorting:" << endl;
        arr.Display();
}
```

Depth First Search in a Graph

```
/* Implement DFS */
#include <iostream>
#include <fstream>

using namespace std;

struct node
{
	int info;
	struct node* next;
};

class stack
{
	struct node* top;
public:
	stack()
	{
		top = nullptr;
	}

	void push(int);
	int pop();

	bool isEmpty()
	{
		return (top == nullptr);
	}

	void display();
};

void stack::push(int data)
{
	node* p;
	if ((p = new node) == nullptr)
	{
		cout << "Memory Exhausted";
		exit(0);
	}
	p = new node;
	p->info = data;
	p->next = nullptr;
	if (top != nullptr)
		p->next = top;
	top = p;
}
```

```
int stack::pop()
{
	struct node* temp;
	int value;
	if (top == nullptr)
	{
		cout << "\nThe stack is Empty" << endl;
		return -1;
	}
	temp = top;
	top = top->next;
	value = temp->info;
	delete temp;
	return value;
}

void stack::display()
{
	struct node* p = top;
	if (top == nullptr)
		cout << "\nNothing to Display\n";
	else
	{
		cout << "\nThe contents of Stack\n";
		while (p != nullptr)
		{
			cout << p->info << endl;
			p = p->next;
		}
	}
}

class Graph
{
	int n;
	int** A;
public:
	Graph(int size = 2);
	~Graph();

	bool isConnected(int x, int y)
	{
		return (A[x - 1][y - 1] == 1);
	}

	void addEdge(int x, int y)
	{
		A[x - 1][y - 1] = A[y - 1][x - 1] = 1;
	}
```

```
	void DFS(int, int);
};

Graph::Graph(int size)
{
	int i, j;
	if (size < 2)
		n = 2;
	else
		n = size;
	A = new int*[n];
	for (i = 0; i < n; ++i)
		A[i] = new int[n];
	for (i = 0; i < n; ++i)
		for (j = 0; j < n; ++j)
			A[i][j] = 0;
}

Graph::~Graph()
{
	for (int i = 0; i < n; ++i)
		delete[] A[i];
	delete[] A;
}

void Graph::DFS(int x, int required)
{
	stack s;
	// Boolean array to track visited nodes
	bool* visited = new bool[n + 1];
	int i;
	// Mark all as unvisited
	for (i = 0; i <= n; i++)
		visited[i] = false;
	s.push(x);
	visited[x] = true;
	if (x == required)
		return;
	cout << "Depth first Search starting from vertex ";
	cout << x << " : " << endl;
	while (!s.isEmpty())
	{
		int k = s.pop();
		if (k == required)
			break;
		cout << k << " ";
		for (i = n; i >= 0; --i)
			if (isConnected(k, i) && !visited[i])
			{
				s.push(i);
				visited[i] = true;
```

```
			}
	}
	cout << endl;
	delete[] visited;
}

int main()
{
	FILE* inFile;
	fopen_s(&inFile, "vertex.txt", "r+");
	int size, matrix[20][20];
	fscanf_s(inFile, "%d", &size);
	for (int i = 1; i <= size; i++)
	{
		for (int j = 1; j <= size; j++)
		{
			fscanf_s(inFile, "%d", &matrix[i][j]);
			cout << matrix[i][j] << " ";
		}
		cout << endl;
	}
	Graph g(size);
	for (int i = 1; i <= size; i++)
		for (int j = 1; j <= size; j++)
			if (matrix[i][j] != 0)
				g.addEdge(i, j);
	cout << "Enter the starting node: ";
	int source;
	cin >> source;
	cout << "Enter the destination node: ";
	int dest;
	cin >> dest;
	cout << endl << endl;
	g.DFS(source, dest);
}
```

Breadth First Search in a Graph

```
/* Implement BFS */
#include <iostream>
#include <ctime>

using namespace std;

struct node
{
	int info;
```

```
	node* next;
};

class Queue
{
	node* front;
	node* rear;
public:
	Queue()
	{
		front = nullptr;
		rear = nullptr;
	}

	~Queue()
	{
		delete front;
	}

	bool isEmpty()
	{
		return (front == nullptr);
	}

	void push(int);
	int pop();
	void display();
};

void Queue::display()
{
	node* p = new node;
	p = front;
	if (front == nullptr)
	{
		cout << "\nNothing to Display\n";
	}
	else
	{
		while (p != nullptr)
		{
			cout << endl << p->info;
			p = p->next;
		}
	}
}

void Queue::push(int data)
{
	node* temp = new node();
	temp->info = data;
```

```
	temp->next = nullptr;
	if (front == nullptr)
		front = temp;
	else
		rear->next = temp;
	rear = temp;
}

int Queue::pop()
{
	node* temp = new node();
	int value;
	if (front == nullptr)
	{
		cout << "\nQueue is Emtpty!" << endl;
		return -1;
	}
	temp = front;
	value = temp->info;
	front = front->next;
	delete temp;
	return value;
}

class Graph
{
	int n;
	int** A;
public:
	Graph(int size = 2);
	~Graph();

	bool isConnected(int u, int v)
	{
		return (A[u - 1][v - 1] == 1);
	}

	void addEdge(int u, int v)
	{
		A[u - 1][v - 1] = A[v - 1][u - 1] = 1;
	}

	void BFS(int);
};

Graph::Graph(int size)
{
	int i, j;
	if (size < 2)
		n = 2;
	else
```

```
		n = size;
	A = new int*[n];
	for (i = 0; i < n; ++i)
		A[i] = new int[n];
	for (i = 0; i < n; ++i)
		for (j = 0; j < n; ++j)
			A[i][j] = 0;
}

Graph::~Graph()
{
	for (int i = 0; i < n; ++i)
		delete[] A[i];
	delete[] A;
}

void Graph::BFS(int s)
{
	Queue Q;
	// Keeps track of visited vertices
	bool* explored = new bool[n + 1];
	// Initialise all vertices as unexplored
	for (int i = 1; i <= n; ++i)
		explored[i] = false;

	// Push initial vertex to the queue
	Q.push(s);
	explored[s] = true;
	cout << "Breadth first search starting from vertex ";
	cout << s << " : " << endl;

	while (!Q.isEmpty())
	{
		int v = Q.pop();
		// Display the explored vertices
		cout << v << " ";
		// From the explored vertex v try to explore all the connected vertices
		for (int w = 1; w <= n; ++w)
			// Explore the vertex w if it is connected to v and if it is unexplored
			if (isConnected(v, w) && !explored[w])
			{
				Q.push(w);
				explored[w] = true;
			}
	}
	cout << endl;
	delete[] explored;
}

int main()
{
```

```
	FILE* inFile;
	fopen_s(&inFile, "vertex.txt", "r+");
	int size, matrix[20][20];
	fscanf_s(inFile, "%d", &size);
	for (int i = 1; i <= size; i++)
	{
		for (int j = 1; j <= size; j++)
		{
			fscanf_s(inFile, "%d", &matrix[i][j]);
			cout << matrix[i][j] << " ";
		}
		cout << endl;
	}
	Graph g(size);
	for (int i = 1; i <= size; i++)
		for (int j = 1; j <= size; j++)
			if (matrix[i][j] != 0)
				g.addEdge(i, j);
	cout << "Enter the starting node: ";
	int source;
	cin >> source;
	/*cout << "Enter the destination node: ";
	int dest;
	cin >> dest;*/
	cout << endl << endl;
	// Explores all vertices findable from the source
	g.BFS(source);
}
```

Dijkstra’s Algorithm to Find Shortest Distance between All Pairs of Vertices in a Graph

```
/* Find the shortest path between a given vertex and
all other vertices in a graph */
#include <iostream>
#define PERM 1
#define TEMP 0
#define INF INT_MAX

using namespace std;

struct node
{
	int pre;
	int dist;
	bool status;
};
```

```
int dijkstra(int s, int d, int path[], int weight[][10], int* len, int n)
{
	// s is the source, d is the destination
	// path will store the shortest path, len will give the length of the path
	struct node node[10];
	int i, current, min, u, v, count = 0, newdist;
	*len = 0;
	// Set all vertices' status to temporary and assign infinite weights
	for (i = 1; i <= n; i++)
	{
		node[i].pre = 0;
		node[i].status = TEMP;
		node[i].dist = INF;
	}
	// Force the first vertex to be selected
	node[s].pre = 0;
	node[s].dist = 0;
	node[s].status = PERM;
	current = s;
	// Until we reach the destination node, keep repeating
	while (current != d)
	{
		for (i = 1; i <= n; i++)
		{
			// For all unvisited vertices find the tentative distance and
			// assign whichever is smaller among the tentative distance and
			// the currently assigned distance
			if (weight[current][i] > 0 && node[i].status == TEMP)
			{
				newdist = node[current].dist + weight[current][i];
				if (newdist < node[i].dist)
				{
					node[i].dist = newdist;
					node[i].pre = current;
				}
			}
		}
		current = 0;
		min = INF;
		// Select the unvisited vertex with the smallest tentative distance
		// Mark it as the current vertex
		for (i = 1; i <= n; i++)
		{
			if (node[i].status == TEMP && node[i].dist < min)
			{
				min = node[i].dist;
				current = i;
			}
		}
		// If there is no such vertex, the algorithm was unable to find a path
```

```
		if (current == 0)
			return 0;
		// Mark the selected vertex as visited
		node[current].status = PERM;
	}
	// Populate the path matrix and the count of elements in path
	while (current != 0)
	{
		count++;
		path[count] = current;
		current = node[current].pre;
	}
	// Find the path length by traversing in reverse order
	for (i = count; i > 1; i--)
	{
		u = path[i];
		v = path[i - 1];
		*len += weight[u][v];
	}
	// Return the number of elements in the path matrix
	return count;
}

int main()
{
	int i, j, n, weight[10][10], s, d, len, count, path[20];
	char ch1;
	// Open the file containing the adjacency matrix
	FILE* f;
	fopen_s(&f, "vertex.txt", "r+");
	fscanf_s(f, "%d", &n);
	// Read the weights from the file
	for (i = 1; i <= n; i++)
	{
		for (j = 1; j <= n; j++)
			fscanf_s(f, "%d", &weight[i][j]);
	}
	// Print the adjacency matrix
	cout << "\nWeight Matrix:" << endl;
	for (i = 1; i <= n; i++)
	{
		for (j = 1; j <= n; j++)
			cout << " " << weight[i][j] << "\t";
		cout << endl;
	}
	// Ask for the source node
	cout << "\nEnter the source node: ";
	cin >> ch1;
	// Convert the entered character into it corresponding alphabet
	s = ch1 - 'a' + 1;
	// Run the algorithm over the set of all possible destination vertices
```

```
	for (int c = 1; c <= n; c++)
	{
		d = c;
		count = dijkstra(s, d, path, weight, &len, n);
		// If the path exists, print it out
		if (len > 0)
		{
			cout << "\n\nShortest Path between " << char(s + 'a' - 1)
				<< " (SOURCE) and " << char(d + 'a' - 1)
				<< " (DESTINATION) is: ";
			for (i = count; i >= 1; i--)
				cout << char(path[i] + 'a' - 1);
			cout << "\n\nLength of the shortest path is: " << len;
		}
		else
			cout << "\n\nNo Path exists between the nodes "
			<< char(s + 'a' - 1) << " and " << char(d + 'a' - 1);
	}
}
```

Prim's Algorithm to Find Minimum Spanning Tree

```
/* Implement Prim's algorithm to find the MST of a graph */
#include <iostream>
#include <stdio.h>

using namespace std;

int main()
{
	int i, j, k, n, x, u = 0, v = 0, small, smallest, pos = 0, total = 0;
	int weight[10][10], visited[10], parent[10];
	/*
		n is the number of vertices in the graph
		parent is the constructed MST
	*/
	// Open the file containing the adjacency matrix
	FILE* f;
	f = fopen("vertex.txt", "r+");
	fscanf(f, "%d", &n);
	// Read all the weights
	for (i = 1; i <= n; i++)
		for (j = 1; j <= n; j++)
			fscanf(f, "%d", &weight[i][j]);
	// Print the weight matrix
	for (i = 1; i <= n; i++)
	{
```

```
		for (j = 1; j <= n; j++)
			cout << " " << weight[i][j] << "\t";
		cout << endl;
	}
	/*
		Algorithm:
		Mark all nodes as unvisited and assign infinite weights to each vertex
		Select the first vertex
		While there are elements not yet selected in the MST, do:
			Pick an unvisited vertex not yet in the MST and,
			which has the shortest edge joining it to a vertex in the MST
			For all the neighbours of the vertex:
				Update the minimum edge length with the value
				of the edge length of the vertex selected earlier
			Add the picked vertex to the MST
	*/
	// Set all vertices as unvisited and the MST as empty
	for (i = 1; i <= n; i++)
	{
		parent[i] = -1;
		visited[i] = 0;
	}
	// Pick the first vertex
	x = 1;
	parent[x] = 1;
	visited[x] = 1;
	// Repeat over all remaining vertices
	for (i = 2; i <= n; i++)
	{
		k = 1;
		// The weight of the shortest edge between a vertex in the MST
		// and an unselected vertex
		smallest = 9999;
		// For all vertices cuurently in the MST
		while (parent[k] != -1)
		{
			// The weight of the vertex nearest to a vertex in the MST
			// in a given row
			small = 999;
			for (j = 1; j <= n; j++)
			{
				// Find vertex nearest (with minimum edge weight) to k
				if (weight[k][j] > 0 && visited[j] == 0 && weight[k][j] < small)
				{
					small = weight[k][j];
					pos = j;
					u = parent[k];
				}
			}
			// If the previously decided minimum weight has been beaten
			// update the global minimum
```

```
                        if (small <= smallest)
                        {
                                smallest = small;
                                v = pos;
                        }
                        k++;
                }
                // If we reach here means we have found the shortest edge between
                // a vertex in the MST and an unselected vertex
                // Update the cost of the MST, add the chosen vertex to the MST
                // and mark it as visited
                total += smallest;
                parent[x++] = v;
                visited[v] = 1;
                cout << "\nEdge from " << char(u + 'a' - 1) << "->" << char(v + 'a' - 1) << endl;
        }
        cout << "\n Weight of the Minimum Spanning Tree : " << total;
}
```

Kruskal's Algorithm to Find Minimum Spanning Tree

```
#include <iostream>

using namespace std;

int main()
{
        int i, j, k, n, visit, l, v, count = 0, count1, vst, p;
        int weight[10][10], visited[10];
        int dup1, dup2;
        FILE* f;
        fopen_s(&f, "vertex.txt", "r+");
        fscanf_s(f, "%d", &n);
        // Read all the weights
        for (i = 1; i <= n; i++)
                for (j = 1; j <= n; j++)
                        fscanf_s(f, "%d", &weight[i][j]);
        // Print the weight matrix
        for (i = 1; i <= n; i++)
        {
                for (j = 1; j <= n; j++)
                        cout << " " << weight[i][j] << "\t";
                cout << endl;
        }
        for (i = 1; i <= n; i++)
                for (j = 1; j <= n; j++)
                        if (weight[i][j] == 0)
                                weight[i][j] = INT_MAX;
```

```
    visit = 1;
    while (visit < n)
    {
        v = INT_MAX;
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                if (weight[i][j] != INT_MAX && weight[i][j] < v
                    && weight[i][j] != -1)
                {
                    int count = 0;
                    for (p = 1; p <= n; p++)
                    {
                        if (visited[p] == i || visited[p] == j)
                            count++;
                    }
                    if (count >= 2)
                    {
                        for (p = 1; p <= n; p++)
                            if (weight[i][p] != INT_MAX && p != j)
                                dup1 = p;
                        for (p = 1; p <= n; p++)
                            if (weight[j][p] != INT_MAX && p != i)
                                dup2 = p;

                        if (weight[dup1][dup2] == -1)
                            continue;
                    }
                    l = i;
                    k = j;
                    v = weight[i][j];
                }
        cout << endl;
        cout << "Edge from " << char(l + 'a' - 1) << "->" << char(k + 'a' - 1)
            << endl;
        weight[l][k] = -1;
        weight[k][l] = -1;
        visit++;
        count1 = 0;
        for (i = 1; i <= n; i++)
        {
            if (visited[i] == l)
                count++;
            if (visited[i] == k)
                count1++;
        }
        if (count == 0)
            visited[++vst] = l;
        if (count1 == 0)
            visited[++vst] = k;
    }
}
```