**DAY 3 - TREES AND GRAPHS**

**PROBLEM 1** - **Discussion on What is Tree Data Structure. Need for such data Structure. Converting a General Tree to Binary Tree**

Solution:

A tree data structure is a hierarchical structure that consists of nodes connected by edges. It is a widely used data structure in computer science and has various applications. In a tree, each node can have zero or more child nodes, except for the root node, which has no parent.

The need for tree data structures arises in situations where data needs to be organized in a hierarchical manner, such as:

1. Representation of hierarchical relationships: Trees are suitable for representing hierarchical relationships, such as the organization structure of a company, the file system structure of a computer, or the parent-child relationships in a family tree.
2. Searching and sorting: Binary search trees, a specific type of tree, provide efficient searching and sorting operations. They enable fast lookup, insertion, and deletion of elements in logarithmic time complexity.
3. Representation of data with parent-child relationships: Trees allow the representation of data with parent-child relationships, where each node can have multiple children. This is useful in representing XML/HTML documents, parsing expressions, or representing nested data structures like directories and subdirectories.

Converting a general tree to a binary tree involves transforming a tree with multiple children per node into a binary tree with at most two children per node. One common approach is to convert the general tree into a binary tree while preserving the sibling order. Here's an implementation in C++:

C++ Code

#include <iostream>

#include <vector>

// Structure for a tree node

struct TreeNode {

int data;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Function to convert a general tree to a binary tree

TreeNode\* convertToBinaryTree(const std::vector<std::vector<int>>& tree, int rootIndex)

{

std::vector<TreeNode\*> nodes(tree.size(), nullptr); // Array to store created nodes

// Create nodes for each value in the tree

for (int i = 0; i < tree.size(); i++)

{

nodes[i] = new TreeNode(i);

}

// Connect child nodes to their parent nodes

for (int i = 0; i < tree.size(); i++)

{

for (int j = 0; j < tree[i].size(); j++) {

if (j == 0)

{

nodes[i]->left = nodes[tree[i][j]];

} else {

nodes[tree[i][j - 1]]->right = nodes[tree[i][j]];

}

}

}

return nodes[rootIndex];

}

// Function to perform in-order traversal of the binary tree

void inOrderTraversal(TreeNode\* root) {

if (root) {

inOrderTraversal(root->left);

std::cout << root->data << " ";

inOrderTraversal(root->right);

}

}

int main() {

// General tree represented as an adjacency list

std::vector<std::vector<int>> tree = {{1, 2, 3}, {4, 5}, {6, 7}, {}, {}, {8, 9}, {}, {}, {}, {}};

int rootIndex = 0; // Index of the root node

// Convert the general tree to a binary tree

TreeNode\* binaryTreeRoot = convertToBinaryTree(tree, rootIndex);

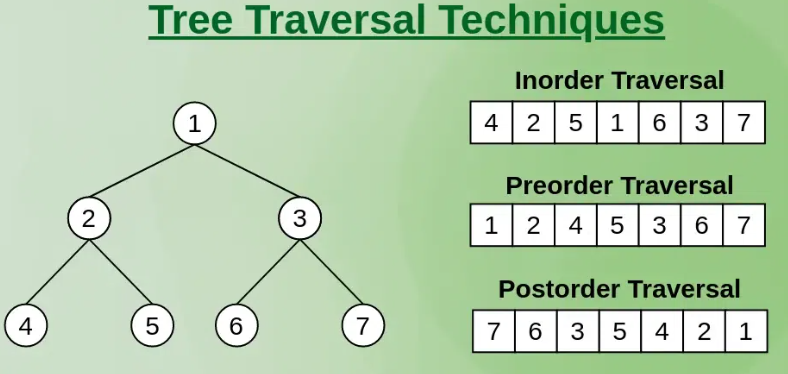
std::cout << "In-order traversal of the binary tree: ";

inOrderTraversal(binaryTreeRoot);

std::cout << std::endl;

return 0;

}



**PROBLEM 2 -** **Implement Tree Traversal for a Binary Tree. Generate a Tree from the given Traversal methods.**

Solution: To implement tree traversal for a binary tree, we typically use three methods: in-order traversal, pre-order traversal, and post-order traversal. Here's an implementation of these traversal methods in C++:

#include <iostream>

// Structure for a tree node

struct TreeNode {

int data;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}

};

// In-order traversal: left -> root -> right

void inOrderTraversal(TreeNode\* root) {

if (root) {

inOrderTraversal(root->left);

std::cout << root->data << " ";

inOrderTraversal(root->right);

}

}

// Pre-order traversal: root -> left -> right

void preOrderTraversal(TreeNode\* root) {

if (root) {

std::cout << root->data << " ";

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

}

// Post-order traversal: left -> right -> root

void postOrderTraversal(TreeNode\* root) {

if (root) {

postOrderTraversal(root->left);

postOrderTraversal(root->right);

std::cout << root->data << " ";

}

}

// Generate a binary tree from in-order and pre-order traversals

TreeNode\* generateTreeFromTraversal(int inOrder[], int preOrder[], int inStart, int inEnd, int& preIndex) {

if (inStart > inEnd) {

return nullptr;

}

// Create a new node with the current pre-order element

TreeNode\* node = new TreeNode(preOrder[preIndex++]);

// Find the index of the current node in the in-order traversal

int inIndex;

for (int i = inStart; i <= inEnd; i++) {

if (inOrder[i] == node->data) {

inIndex = i;

break;

}

}

// Recursively generate left and right subtrees

node->left = generateTreeFromTraversal(inOrder, preOrder, inStart, inIndex - 1, preIndex);

node->right = generateTreeFromTraversal(inOrder, preOrder, inIndex + 1, inEnd, preIndex);

return node;

}

int main() {

int inOrder[] = {4, 2, 5, 1, 6, 3, 7};

int preOrder[] = {1, 2, 4, 5, 3, 6, 7};

int size = sizeof(inOrder) / sizeof(inOrder[0]);

int preIndex = 0;

TreeNode\* root = generateTreeFromTraversal(inOrder, preOrder, 0, size - 1, preIndex);

std::cout << "In-order traversal: ";

inOrderTraversal(root);

std::cout << std::endl;

std::cout << "Pre-order traversal: ";

preOrderTraversal(root);

std::cout << std::endl;

std::cout << "Post-order traversal: ";

postOrderTraversal(root);

std::cout << std::endl;

return 0;

}

**PROBLEM 3** - **Create a function to determine if a binary tree is symmetric (its left and right subtrees are mirror images of each other).**

**Solution -** Here's an implementation of a function in C++ to determine if a binary tree is symmetric. The function uses a recursive approach to compare the left and right subtrees.

#include <iostream>

// Definition for a binary tree node.

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

// Helper function to check if two trees are mirror images of each other

bool isMirror(TreeNode\* leftTree, TreeNode\* rightTree) {

// Base cases: if both trees are empty, they are mirror images

if (leftTree == nullptr && rightTree == nullptr)

return true;

// If one of the trees is empty or the values at the current nodes don't match, they are not mirror images

if (leftTree == nullptr || rightTree == nullptr || leftTree->val != rightTree->val)

return false;

// Recursively check if the left subtree of the left tree is a mirror image of the right subtree of the right tree,

// and if the right subtree of the left tree is a mirror image of the left subtree of the right tree.

return isMirror(leftTree->left, rightTree->right) && isMirror(leftTree->right, rightTree->left);

}

// Function to check if a binary tree is symmetric

bool isSymmetric(TreeNode\* root) {

// If the tree is empty, it is symmetric

if (root == nullptr)

return true;

// Check if the left and right subtrees are mirror images of each other

return isMirror(root->left, root->right);

}

// Test the function

int main() {

// Create a symmetric binary tree

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(2);

root->left->left = new TreeNode(3);

root->left->right = new TreeNode(4);

root->right->left = new TreeNode(4);

root->right->right = new TreeNode(3);

// Check if the tree is symmetric

if (isSymmetric(root))

std::cout << "The binary tree is symmetric." << std::endl;

else

std::cout << "The binary tree is not symmetric." << std::endl;

return 0;

}

Here are the steps for the implementation:

1. We define the TreeNode struct, representing a node in the binary tree. It has an integer value (val) and pointers to its left and right child nodes.
2. We implement a helper function isMirror that takes two tree nodes as parameters and checks if the trees rooted at those nodes are mirror images of each other. The function uses a recursive approach.
3. In the isMirror function, we have base cases to handle empty trees (both leftTree and rightTree are nullptr), and another case to handle situations where one of the trees is empty or the values at the current nodes don't match. In such cases, the trees are not mirror images.
4. The recursive step in the isMirror function checks if the left subtree of the left tree is a mirror image of the right subtree of the right tree, and if the right subtree of the left tree is a mirror image of the left subtree of the right tree.
5. Next, we implement the isSymmetric function, which takes the root of the binary tree as a parameter. It checks if the tree is empty and returns true in that case.

**PROBLEM 4 –**

Design a program to check if a binary tree is a binary search tree (BST)

#include <iostream>

// Binary Tree Node Structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

// Function to check if a binary tree is a binary search tree

bool isBST(Node\* root, Node\* minNode = nullptr, Node\* maxNode = nullptr) {

// Base case: an empty tree is considered a valid BST

if (root == nullptr)

return true;

// Check if the current node violates the BST property

if ((minNode && root->data <= minNode->data) || (maxNode && root->data >= maxNode->data))

return false;

// Recursively check the left and right subtrees

return isBST(root->left, minNode, root) && isBST(root->right, root, maxNode);

}

int main() {

// Create a sample binary tree

Node\* root = new Node(4);

root->left = new Node(2);

root->right = new Node(6);

root->left->left = new Node(1);

root->left->right = new Node(3);

root->right->left = new Node(5);

root->right->right = new Node(7);

// Check if the binary tree is a BST

if (isBST(root))

std::cout << "The binary tree is a binary search tree (BST).\n";

else

std::cout << "The binary tree is not a binary search tree (BST).\n";

// Clean up the dynamically allocated memory

delete root->left->left;

delete root->left->right;

delete root->right->left;

delete root->right->right;

delete root->left;

delete root->right;

delete root;

return 0;

}

Explanation:

1. We define a **Node** structure to represent each node of the binary tree. Each **Node** contains a data value and pointers to its left and right child nodes.
2. The **isBST** function checks if the given **root** node is part of a valid BST. It takes three parameters: the **root** node, a **minNode** representing the minimum value allowed in the subtree, and a **maxNode** representing the maximum value allowed in the subtree.
3. The function starts with a base case: if the **root** node is **nullptr**, it means the current subtree is empty, so it is considered a valid BST, and the function returns **true**.
4. We then check if the current **root** violates the BST property. If the **minNode** is not **nullptr** and the data of the **root** is less than or equal to the **minNode** data, or if the **maxNode** is not **nullptr** and the data of the **root** is greater than or equal to the **maxNode** data, it means the BST property is violated, and the function returns **false**.
5. If the current **root** does not violate the BST property, we recursively call the **isBST** function for the left and right subtrees. For the left subtree, the maximum allowed value becomes the current **root** (as no value greater than the current **root** should be present in the left subtree). For the right subtree, the minimum allowed value becomes the current **root**

**PROBLEM 5 –**

**Develop a data structure program that allows efficient insertion and retrieval of elements in a self-balancing binary search tree. [Here red-black tree]**

In C++, you can use the built-in **std::set** container from the Standard Template Library (STL) to implement a self-balancing binary search tree (BST). The **std::set** is implemented as a **red-black tree**, which is a self-balancing binary search tree.

Here's an example program that demonstrates the usage of **std::set** for efficient insertion and retrieval:

#include <iostream>

#include <set>

int main() {

std::set<int> mySet;

// Insert elements into the set

mySet.insert(10);

mySet.insert(5);

mySet.insert(15);

mySet.insert(3);

mySet.insert(8);

// Retrieve elements from the set

std::cout << "Set elements: ";

for (const auto& element : mySet) {

std::cout << element << " ";

}

std::cout << std::endl;

// Check if an element exists in the set

int key = 5;

if (mySet.find(key) != mySet.end()) {

std::cout << key << " exists in the set." << std::endl;

} else {

std::cout << key << " does not exist in the set." << std::endl;

}

return 0;

}

In this program, we create a **std::set** named **mySet** to store integers. We then insert elements into the set using the **insert()** function. The **std::set** container automatically maintains the tree structure and balances it using the red-black tree algorithm.

To retrieve elements from the set, we can use a range-based for loop to iterate over the set and print its elements.

To check if a specific element exists in the set, we can use the **find()** function, which returns an iterator to the element if found, or **mySet.end()** if not found.

The self-balancing property of the red-black tree ensures that insertion and retrieval operations have a time complexity of O(log n) on average, where n is the number of elements in the set. This makes it an efficient data structure for handling dynamic datasets with efficient search and insertion operations.

**Problem 6 - Design an efficient program to find the kth smallest element in a binary search tree.**

Solution –

#include <iostream>

#include <stack>

using namespace std;

// Definition for a binary tree node.

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int value) : val(value), left(nullptr), right(nullptr) {}

};

int kthSmallest(TreeNode\* root, int k) {

stack<TreeNode\*> nodes;

while (root != nullptr || !nodes.empty()) {

while (root != nullptr) {

nodes.push(root);

root = root->left;

}

root = nodes.top();

nodes.pop();

if (--k == 0) {

return root->val;

}

root = root->right;

}

return -1; // Invalid case

}

// Test program

int main() {

// Create a binary search tree

TreeNode\* root = new TreeNode(5);

root->left = new TreeNode(3);

root->right = new TreeNode(7);

root->left->left = new TreeNode(2);

root->left->right = new TreeNode(4);

root->right->left = new TreeNode(6);

root->right->right = new TreeNode(8);

int k = 3;

int kthSmallestElement = kthSmallest(root, k);

cout << "The " << k << "th smallest element in the BST is: " << kthSmallestElement << endl;

return 0;

}

In this program, we use an iterative approach and a stack to perform an in-order traversal of the binary search tree. The in-order traversal visits the nodes in ascending order, which allows us to find the kth smallest element efficiently.

We start by initializing an empty stack and a while loop. In each iteration, we traverse to the leftmost node of the current subtree, pushing all the encountered nodes onto the stack. Once we reach the leftmost node, we pop it from the stack and check if it is the kth smallest element. If it is, we return its value. Otherwise, we move to the right subtree and repeat the process. We continue this until we find the kth smallest element or the stack becomes empty.

In the main function, we create a binary search tree as an example and call the **kthSmallest** function with the root node and the desired value of k. Finally, we print the result.

Feel free to modify the program to suit your needs or integrate it into a larger codebase.

**Problem – 7**

Develop a program that allows efficient insertion and deletion of elements in a self-balancing binary AVL tree.

Solution –

Basic implementation of an AVL tree in C++ that supports efficient insertion and deletion operations. Here's the code:

#include <iostream>

// Structure for a node in the AVL tree

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

// Function to calculate the height of a node

int getHeight(Node\* node) {

if (node == nullptr)

return 0;

return node->height;

}

// Function to calculate the balance factor of a node

int getBalance(Node\* node) {

if (node == nullptr)

return 0;

return getHeight(node->left) - getHeight(node->right);

}

// Function to update the height of a node

void updateHeight(Node\* node) {

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

node->height = 1 + std::max(leftHeight, rightHeight);

}

// Function to perform a right rotation

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

updateHeight(y);

updateHeight(x);

return x;

}

// Function to perform a left rotation

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

updateHeight(x);

updateHeight(y);

return y;

}

// Function to insert a key into the AVL tree

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

// Perform the normal BST insertion

if (root == nullptr) {

Node\* newNode = new Node;

newNode->key = key;

newNode->left = nullptr;

newNode->right = nullptr;

newNode->height = 1;

return newNode;

}

if (key < root->key)

root->left = insert(root->left, key);

else if (key > root->key)

root->right = insert(root->right, key);

else

return root; // Duplicate keys are not allowed in AVL tree

// Update the height of the ancestor node

updateHeight(root);

// Check the balance factor and rebalance the tree if needed

int balance = getBalance(root);

// Left Left Case

if (balance > 1 && key < root->left->key)

return rotateRight(root);

// Right Right Case

if (balance < -1 && key > root->right->key)

return rotateLeft(root);

// Left Right Case

if (balance > 1 && key > root->left->key) {

root->left = rotateLeft(root->left);

return rotateRight(root);

}

// Right Left Case

if (balance < -1 && key < root->right->key) {

root->right = rotateRight(root->right);

return rotateLeft(root);

}

return root;

}

// Function to find the node with the minimum key value

Node\* findMinNode(Node\* root) {

Node\* current = root;

while (current->left != nullptr)

current = current->left;

return current;

}

// Function to delete a key from the AVL tree

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

// Perform the normal BST deletion

if (root == nullptr)

return root;

if (key < root->key)

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

else if (key > root->key)

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

else {

// Node to be deleted found

// Node with only one child or no child

if (root->left == nullptr || root->right == nullptr) {

Node\* temp = root->left ? root->left : root->right;

// No child case

if (temp == nullptr) {

temp = root;

root = nullptr;

} else // One child case

\*root = \*temp; // Copy the contents of the non-empty child

delete temp;

} else {

// Node with two children, get the inorder successor

Node\* temp = findMinNode(root->right);

// Copy the inorder successor's data to this node

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

}

// If the tree had only one node then return

if (root == nullptr)

return root;

// Update the height of the current node

updateHeight(root);

// Check the balance factor and rebalance the tree if needed

int balance = getBalance(root);

// Left Left Case

if (balance > 1 && getBalance(root->left) >= 0)

return rotateRight(root);

// Left Right Case

if (balance > 1 && getBalance(root->left) < 0) {

root->left = rotateLeft(root->left);

return rotateRight(root);

}

// Right Right Case

if (balance < -1 && getBalance(root->right) <= 0)

return rotateLeft(root);

// Right Left Case

if (balance < -1 && getBalance(root->right) > 0) {

root->right = rotateRight(root->right);

return rotateLeft(root);

}

return root;

}

// Function to print the AVL tree in inorder traversal

void printInorder(Node\* root) {

if (root == nullptr)

return;

printInorder(root->left);

std::cout << root->key << " ";

printInorder(root->right);

}

// Driver code

int main() {

Node\* root = nullptr;

// Inserting elements into the AVL tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

std::cout << "Inorder traversal after insertions: ";

printInorder(root);

std::cout << std::endl;

// Deleting elements from the AVL tree

root = deleteNode(root, 30);

root = deleteNode(root, 40);

std::cout << "Inorder traversal after deletions: ";

printInorder(root);

std::cout << std::endl;

return 0;

}

This implementation provides the basic functionality of an AVL tree, including efficient insertion and deletion operations. The **insert** function takes a root node and a key to be inserted. It performs the normal BST insertion and then checks the balance factor of each node to ensure the AVL tree property is maintained. If the balance factor is violated, appropriate rotations are performed to rebalance the tree.

The **deleteNode** function takes a root node and a key to be deleted. It performs the normal BST deletion and then checks the balance factor of each node to rebalance the tree if needed.

The **rotateRight** and **rotateLeft** functions are used to perform right and left rotations, respectively, to rebalance the tree.

The **printInorder** function is provided to print the elements of the AVL tree in ascending order.

In the example in the **main** function, elements are inserted into the AVL tree, and then some elements are deleted. The inorder traversal is printed before and after the insertions and deletions to demonstrate the correctness of the operations.

Note: This implementation assumes unique keys, as duplicate keys are not allowed in an AVL tree.

**GRAPH PROBLEMS**

**Problem 8 - Implementation of BFS for Graph using Adjacency List implementation**

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Function to add an edge to the graph

void addEdge(vector<int> adj[], int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

// Function to perform Breadth-First Search traversal

void BFS(vector<int> adj[], int V, int startVertex) {

vector<bool> visited(V, false); // Track visited vertices

queue<int> q; // Queue for BFS traversal

visited[startVertex] = true; // Mark the start vertex as visited

q.push(startVertex); // Enqueue the start vertex

while (!q.empty()) {

int currVertex = q.front(); // Get the current vertex

q.pop();

cout << currVertex << " "; // Process the current vertex

// Traverse all adjacent vertices of the current vertex

for (int adjVertex : adj[currVertex]) {

if (!visited[adjVertex]) {

visited[adjVertex] = true; // Mark the adjacent vertex as visited

q.push(adjVertex); // Enqueue the adjacent vertex

}

}

}

}

int main() {

int V = 5; // Number of vertices in the graph

vector<int> adj[V]; // Adjacency list representation

// Add edges

addEdge(adj, 0, 1);

addEdge(adj, 0, 2);

addEdge(adj, 1, 3);

addEdge(adj, 2, 4);

addEdge(adj, 3, 4);

int startVertex = 0; // Starting vertex for BFS traversal

cout << "BFS traversal starting from vertex " << startVertex << ": ";

BFS(adj, V, startVertex);

return 0;

}

In this code, we have an **addEdge** function to add edges between vertices in the graph. The **BFS** function performs the Breadth-First Search traversal, starting from the **startVertex**. It uses a queue to keep track of the vertices to be visited. The **visited** vector is used to mark the visited vertices to avoid revisiting them.

In the **main** function, we create a graph with 5 vertices and add edges between them. We then specify the starting vertex for the BFS traversal and call the **BFS** function.

When you run this code, it will output the BFS traversal starting from the specified vertex. For the given example, the output will be:

**BFS traversal starting from vertex 0: 0 1 2 3 4**

This indicates that the BFS traversal starts from vertex 0 and visits vertices 1, 2, 3, and 4 in that order.

**Problem 9 - Design an algorithm to convert a graph represented as a collection of edges to an adjacency matrix, considering scalability and memory usage using c++.**

#include <iostream>

#include <vector>

// Function to convert graph to adjacency matrix

std::vector<std::vector<int>> convertToAdjacencyMatrix(const std::vector<std::pair<int, int>>& edges, int numVertices) {

std::vector<std::vector<int>> adjacencyMatrix(numVertices, std::vector<int>(numVertices, 0));

// Iterate over the edges and populate the adjacency matrix

for (const auto& edge : edges) {

int src = edge.first;

int dest = edge.second;

adjacencyMatrix[src][dest] = 1;

adjacencyMatrix[dest][src] = 1; // Assuming an undirected graph

}

return adjacencyMatrix;

}

// Function to display adjacency matrix

void displayAdjacencyMatrix(const std::vector<std::vector<int>>& adjacencyMatrix) {

for (const auto& row : adjacencyMatrix) {

for (int val : row) {

std::cout << val << " ";

}

std::cout << std::endl;

}

}

int main() {

// Example usage

std::vector<std::pair<int, int>> edges = {{0, 1}, {0, 2}, {1, 2}, {2, 3}};

int numVertices = 4;

std::vector<std::vector<int>> adjacencyMatrix = convertToAdjacencyMatrix(edges, numVertices);

displayAdjacencyMatrix(adjacencyMatrix);

return 0;

}

In this algorithm, we use a 2D vector (**adjacencyMatrix**) to represent the adjacency matrix. The **convertToAdjacencyMatrix** function takes a vector of edges and the number of vertices as input. It initializes the adjacency matrix with all zeros and then iterates over the edges to set the corresponding matrix entries to 1 (assuming an undirected graph). Finally, it returns the resulting adjacency matrix.

The **displayAdjacencyMatrix** function is used to print the adjacency matrix for visualization purposes.

In the **main** function, we provide an example usage of the algorithm by creating a vector of edges and specifying the number of vertices. The resulting adjacency matrix is then displayed using the **displayAdjacencyMatrix** function.