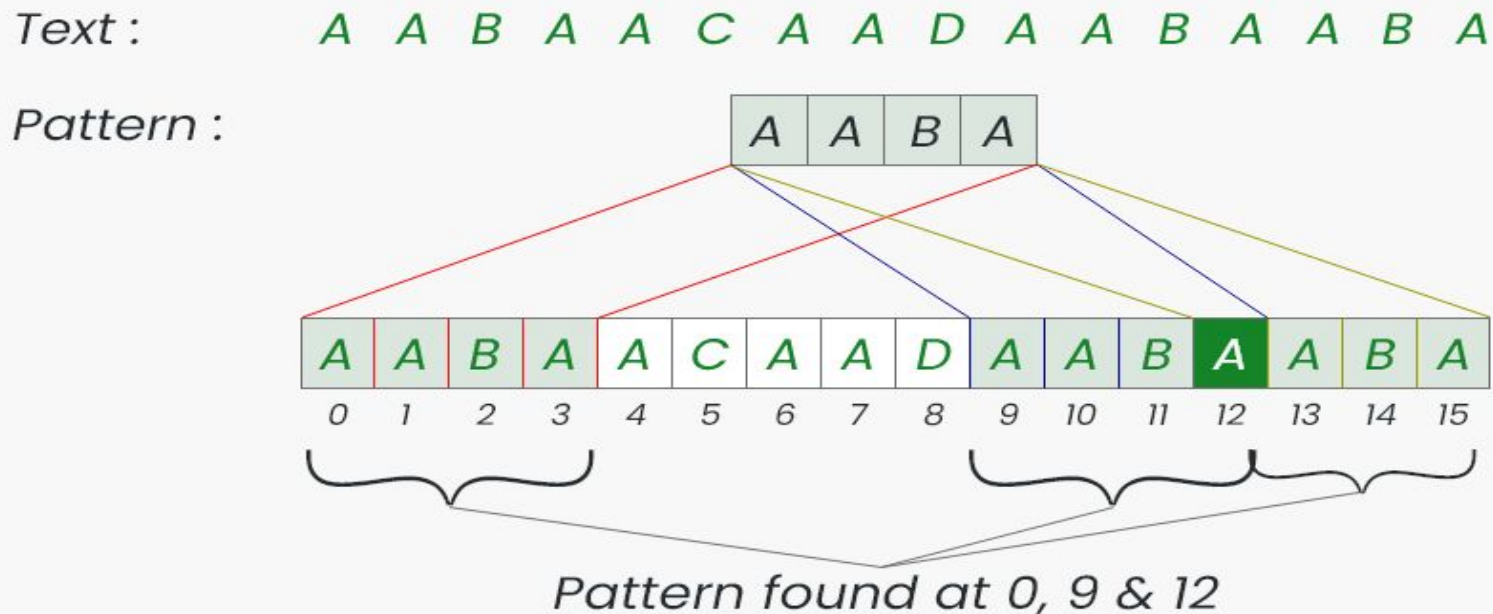


UNIT V

Pattern Searching

Pattern searching algorithms are essential tools in computer science and data processing. These algorithms are designed to efficiently find a particular pattern within a larger set of data. A string matching algorithm is an algorithm that searches for occurrences of a pattern string within a larger text string. These algorithms are used in various applications like searching for specific words in a document, finding malicious code in software, or identifying patterns in biological data.



Purpose:

The primary goal is to find all or specific occurrences of a pattern within a larger text.

Input:

The algorithm takes two strings as input: the pattern string (the one being searched for) and the text string (the larger string where the search is performed).

Output:

The algorithm typically returns the starting positions (indices) of all occurrences of the pattern within the text string.

Common Algorithms:

Several string matching algorithms exist, each with its strengths and weaknesses in terms of time complexity and performance. Some notable examples include:

Important Pattern Searching Algorithms:

Naive String Matching : A Simple Algorithm that works in $O(m \times n)$ time where m is the length of the pattern and n is the length of the text.

Knuth-Morris-Pratt (KMP) Algorithm It preprocesses pattern and works in $O(m + n)$ Time.

Rabin-Karp Algorithm : It uses hashing to compare the pattern with the text. It works in $O(m \times n)$ Time in worst case,

Aho-Corasick Algorithm : A deterministic finite automaton (DFA) based algorithm and works in $O(m + n)$ time.

Naive algorithm for Pattern Searching

Given **text** string with length n and a **pattern** with length m , the task is to print all occurrences of **pattern** in **text**.

Note: You may assume that $n > m$.

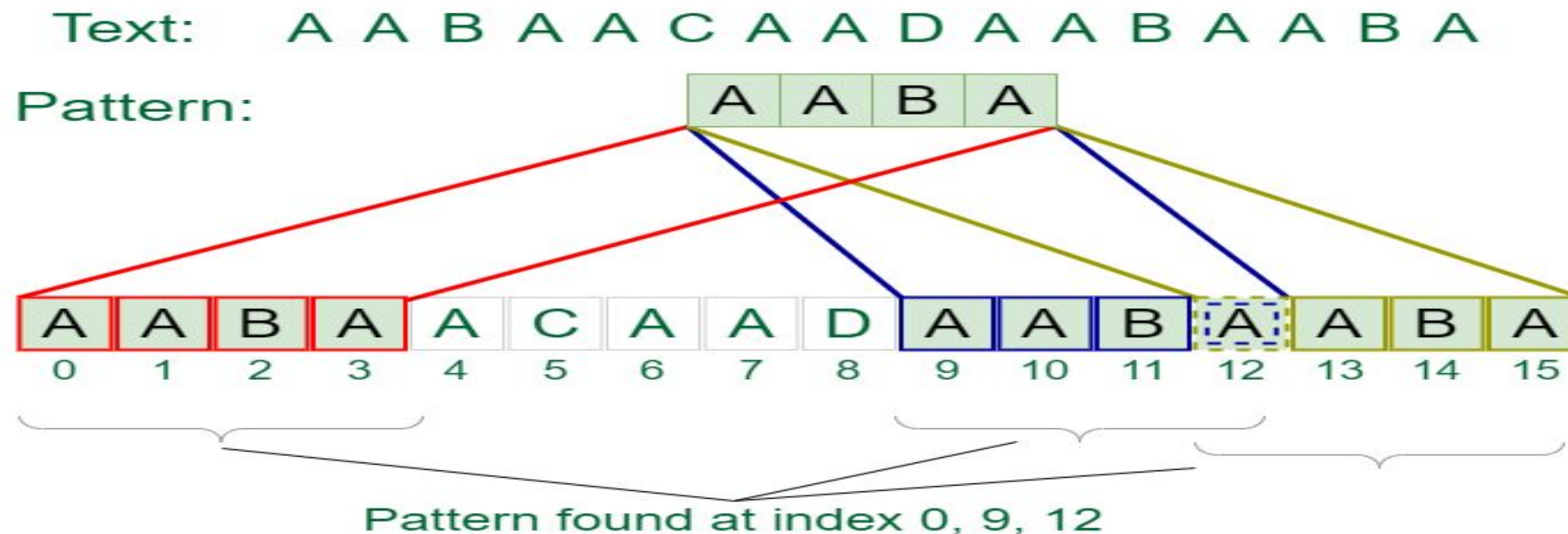
Examples:

Input: text = "THIS IS A TEST TEXT", pattern = "TEST"

Output: Pattern found at index 10

Input: text = "AABAACAADAABAABA", pattern = "AABA"

Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12



```
#include <iostream>
#include <string>
using namespace std;
void search(string& pat, string& txt) {
    int M = pat.size();
    int N = txt.size();
    // A loop to slide pat[] one by one
    for (int i = 0; i <= N - M; i++) {
        int j;
        // For current index i, check for pattern match
        for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j]) {
                break;
            }
        }
        // If pattern matches at index i
        if (j == M) {
            cout << "Pattern found at index " << i << endl;
        }
    }
}
```

// Driver's Code

```
int main() {
```

```
    // Example 1
```

```
    string txt1 = "AABAACAADAABAABA";
```

```
    string pat1 = "AABA";
```

```
    cout << "Example 1: " << endl;
```

```
    search(pat1, txt1);
```

```
    // Example 2
```

```
    string txt2 = "agd";
```

```
    string pat2 = "g";
```

```
    cout << "\nExample 2: " << endl;
```

```
    search(pat2, txt2);
```

```
    return 0;
```

```
}
```

Output :

Pattern found at index 0

Pattern found at index 9

Pattern found at index 13

Time Complexity: $O(N^2)$

Auxiliary Space: $O(1)$

KMP Algorithm for Pattern Searching

Given two strings **txt** and **pat**, the task is to return all indices of **occurrences** of **pat** within **txt**.

Examples:

Input: **txt** = "ab~~cab~~", **pat** = "ab"

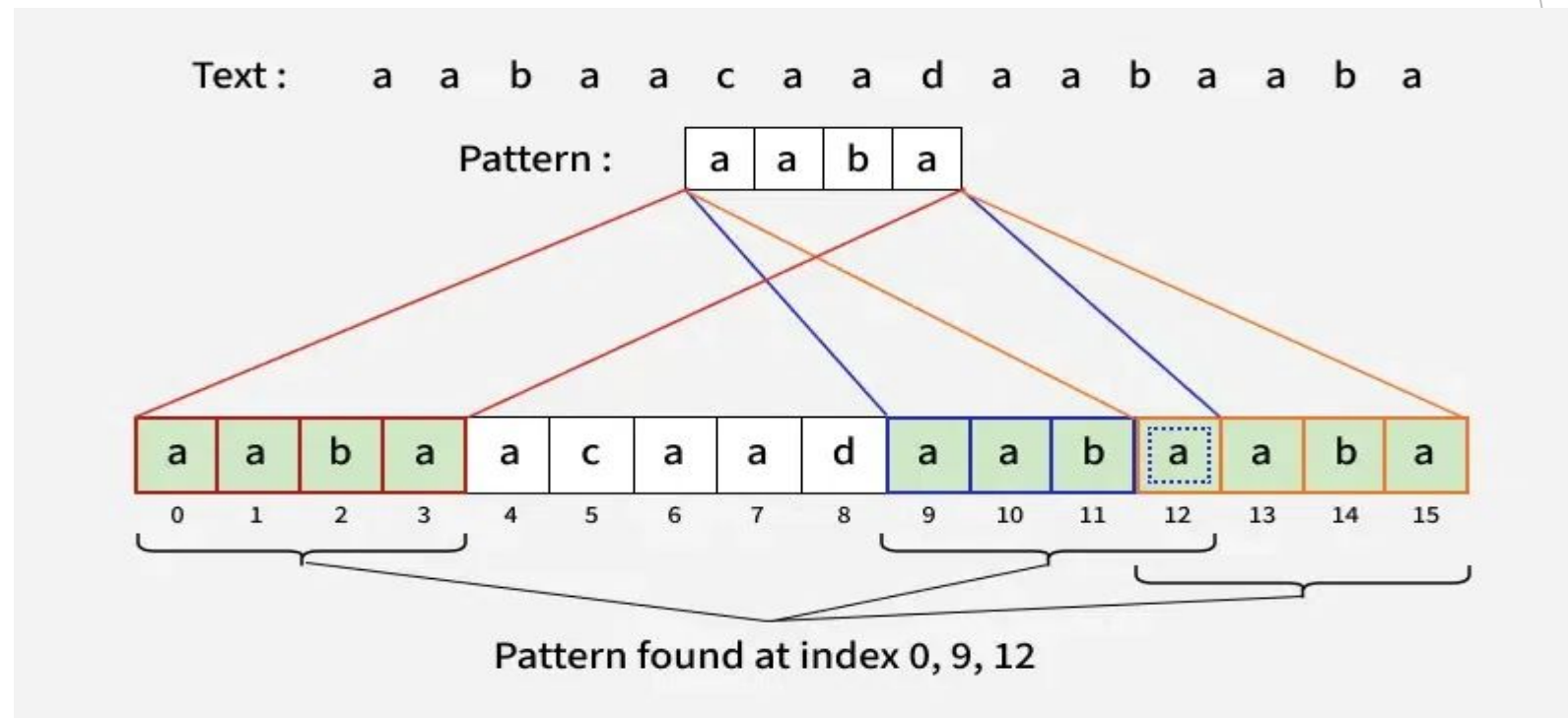
Output: [0, 3]

Explanation: The string "ab" occurs twice in txt, first occurrence starts from index 0 and second from index 3.

Input: **txt**= "aabaacaadaabaaba", **pat** = "aaba"

Output: [0, 9, 12]

Explanation:



Rabin-Karp Algorithm for Pattern Searching

Given a text $T[0 \dots n-1]$ and a pattern $P[0 \dots m-1]$, write a function `search(char P[], char T[])` that prints all occurrences of $P[]$ present in $T[]$ using Rabin Karp algorithm. You may assume that $n > m$.

Examples:

Input: $T[] = \text{"THIS IS A TEST TEXT"}, P[] = \text{"TEST"}$

Output: Pattern found at index 10

Input: $T[] = \text{"AABAACAADAABAABA"}, P[] = \text{"AABA"}$

Output: Pattern found at index 0

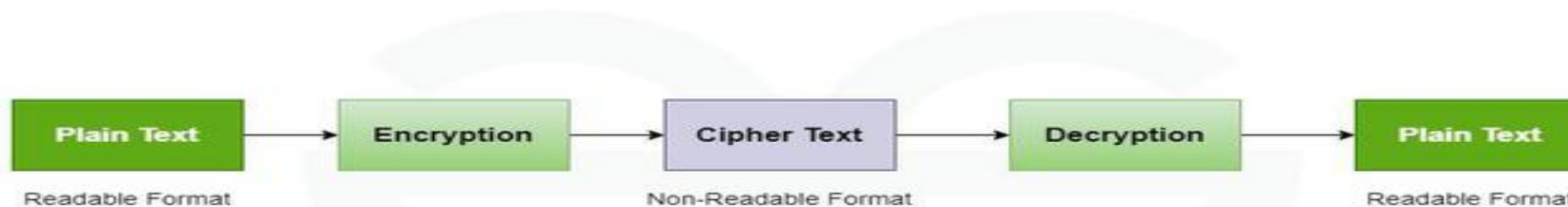
Pattern found at index 9

Pattern found at index 12

Cryptography :Cryptography is a technique of securing information and communications through the use of codes so that only those persons for whom the information is intended can understand and process it. Thus, preventing unauthorized access to information. The prefix “crypt” means “hidden” and the suffix “graphy” means “writing”.

In Cryptography, the techniques that are used to protect information are obtained from mathematical concepts and a set of rule-based calculations known as algorithms to convert messages in ways that make it hard to decode them.

These algorithms are used for cryptographic key generation, digital signing, and verification to protect data privacy, web browsing on the internet and to protect confidential transactions such as credit card and debit card transactions.



Features Of Cryptography

Confidentiality: Information can only be accessed by the person for whom it is intended and no other person except him can access it.

Integrity: Information cannot be modified in storage or transition between sender and intended receiver without any addition to information being detected.

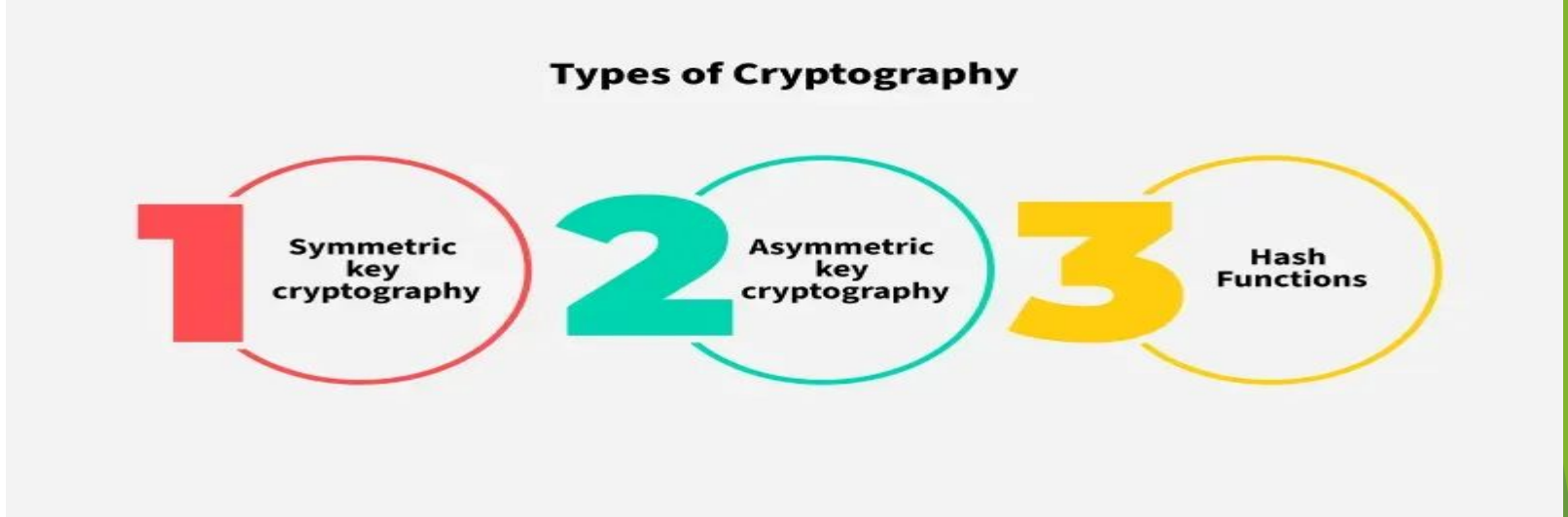
Non-repudiation: The creator/sender of information cannot deny his intention to send information at a later stage.

Authentication: The identities of the sender and receiver are confirmed. As well destination/origin of the information is confirmed.

Interoperability: Cryptography allows for secure communication between different systems and platforms.

Adaptability: Cryptography continuously evolves to stay ahead of security threats and technological advancements.

Types Of Cryptography :



1. Symmetric Key Cryptography

It is an encryption system where the sender and receiver of a message use a single common key to encrypt and decrypt messages. [Symmetric Key cryptography](#) is faster and simpler but the problem is that the sender and receiver have to somehow exchange keys securely. The most popular symmetric key cryptography systems are [Data Encryption Systems \(DES\)](#) and [Advanced Encryption Systems \(AES\)](#).



2. Hash Functions

There is no usage of any key in this algorithm. A hash value with a fixed length is calculated as per the plain text which makes it impossible for the contents of plain text to be recovered. Many operating systems use hash functions to encrypt passwords.

3. Asymmetric Key Cryptography

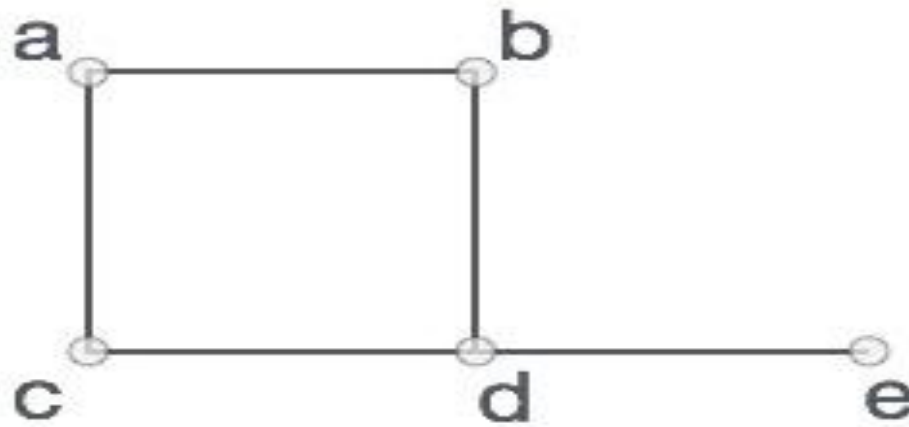
In Asymmetric Key Cryptography, a pair of keys is used to encrypt and decrypt information. A sender's public key is used for encryption and a receiver's private key is used for decryption. Public keys and Private keys are different. Even if the public key is known by everyone the intended receiver can only decode it because he alone knows his private key. The most popular asymmetric key cryptography algorithm is the RSA algorithm.



Graph :

A graph is an abstract data type (ADT) which consists of a set of objects that are connected to each other via links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$

The Graph Abstract Data Type

The graph abstract data type (ADT) is defined as follows:

`Graph()` creates a new, empty graph.

`addVertex(vert)` adds an instance of `Vertex` to the graph.

`addEdge(fromVert, toVert)` Adds a new, directed edge to the graph that connects two vertices.

`addEdge(fromVert, toVert, weight)` Adds a new, weighted, directed edge to the graph that connects two vertices.

`getVertex(vertKey)` finds the vertex in the graph named `vertKey`.

`getVertices()` returns the list of all vertices in the graph.

`in` returns `True` for a statement of the form `vertex in graph`, if the given vertex is in the graph, `False` otherwise.

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

Vertex – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

Edge – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

Adjacency – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Representations of Graph

Here are the two most common ways to represent a graph : For simplicity, we are going to consider only unweighted graphs in this post.

Adjacency Matrix

Adjacency List

Adjacency Matrix Representation

An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

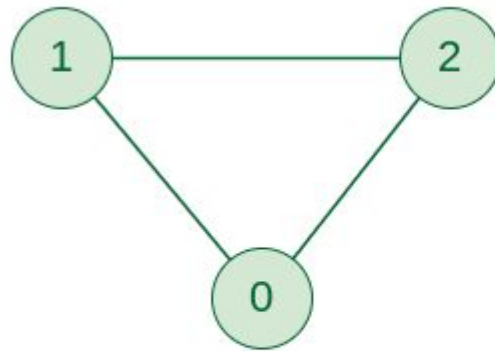
Let's assume there are n vertices in the graph So, create a 2D matrix `adjMat[n][n]` having dimension $n \times n$.

If there is an edge from vertex i to j , mark `adjMat[i][j]` as 1.

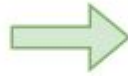
If there is no edge from vertex i to j , mark `adjMat[i][j]` as 0.

Representation of Undirected Graph as Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 to both cases (`adjMat[destination]` and `adjMat[destination]`) because we can go either way.



Undirected Graph



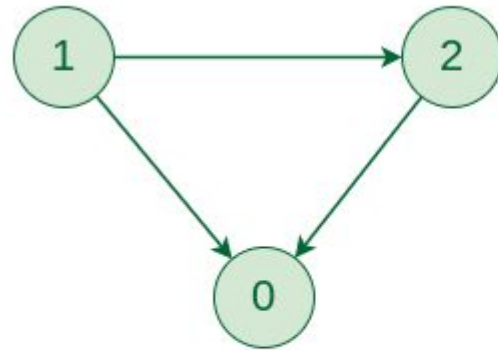
	0	1	2
0		1	1
1	1		1
2	1	1	

Adjacency Matrix

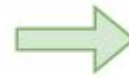
Graph Representation of Undirected graph to Adjacency Matrix

Representation of Directed Graph as Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 for that particular `adjMat[destination]`.



Directed Graph



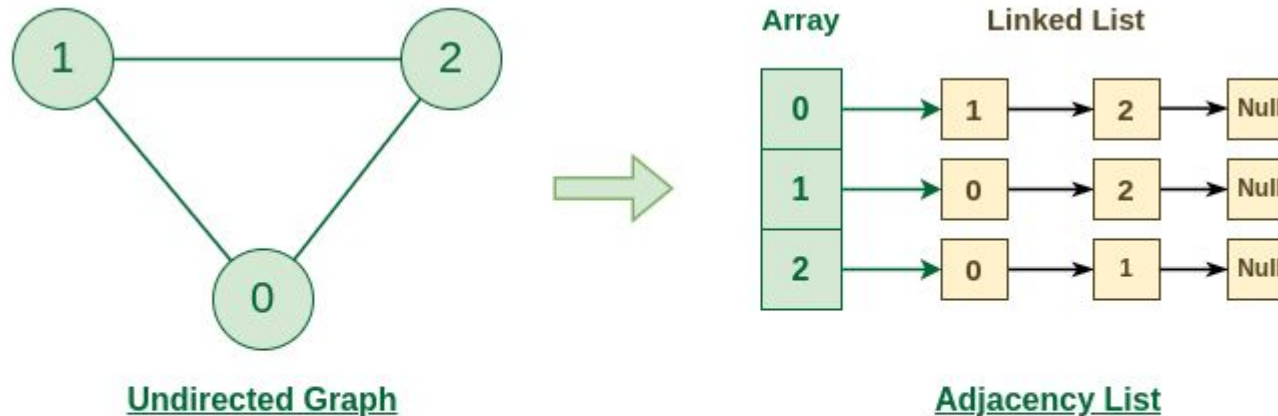
	0	1	2
0			
1	1		1
2	1		

Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

Representation of Undirected Graph as Adjacency list:

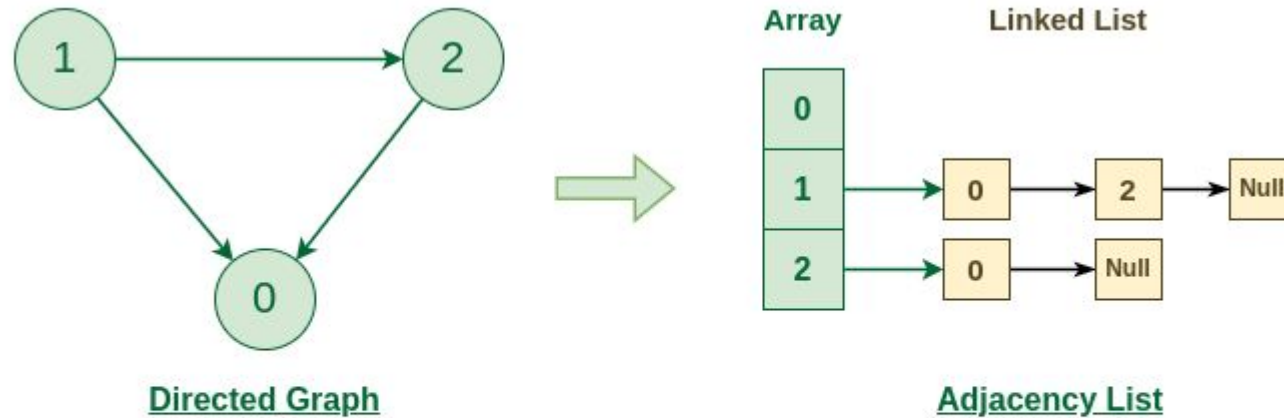
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Undirected graph to Adjacency List

Representation of Directed Graph as Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Directed graph to Adjacency List

Operations of Graphs

The primary operations of a graph include creating a graph with vertices and edges, and displaying the said graph. However, one of the most common and popular operation performed using graphs are Traversal, i.e. visiting every vertex of the graph in a specific order.

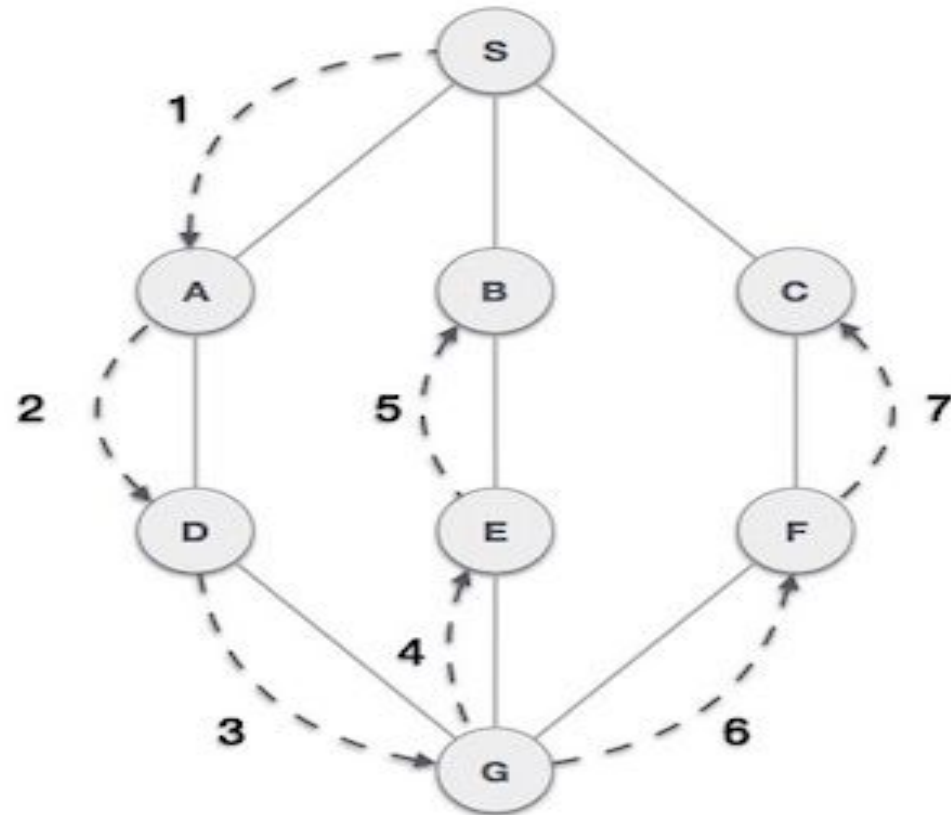
There are two types of traversals in Graphs –

- Depth First Search Traversal

- Breadth First Search Traversal

Depth First Search (DFS) Algorithm

Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

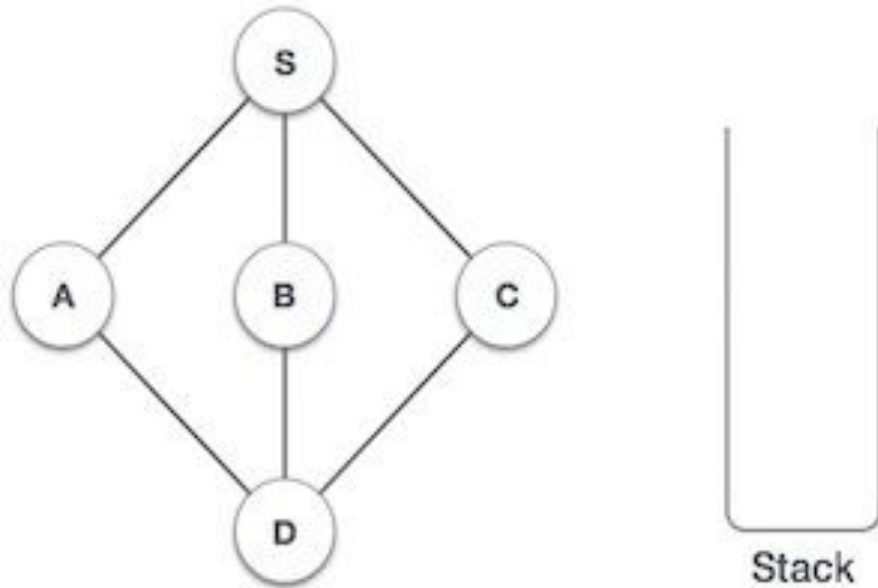


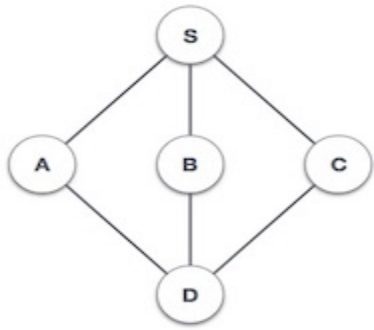
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.



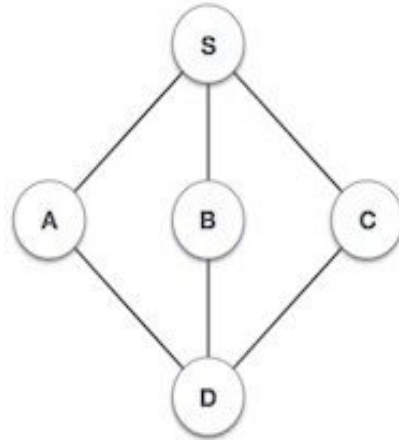


1



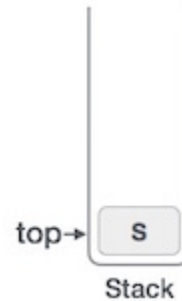
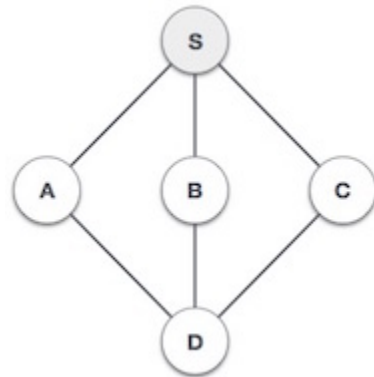
Traversal

Description



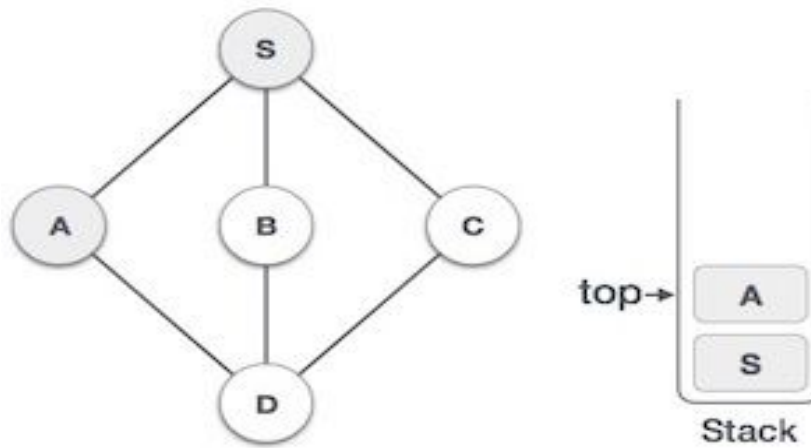
Initialize the stack.

2



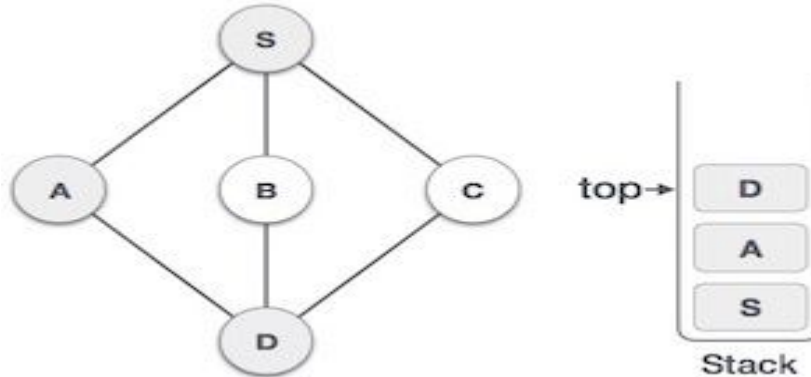
Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3



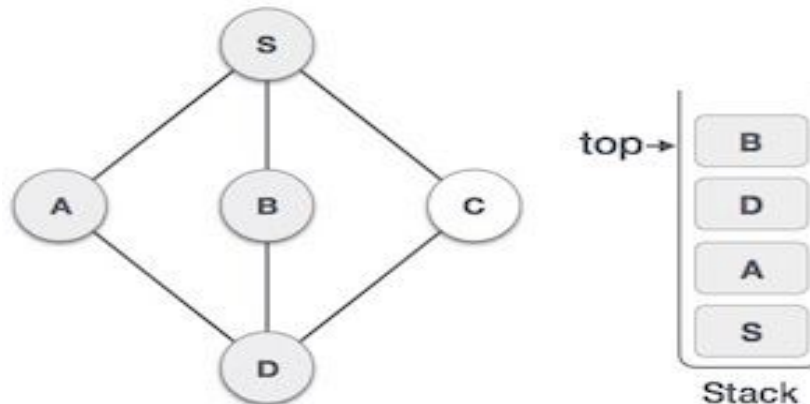
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4



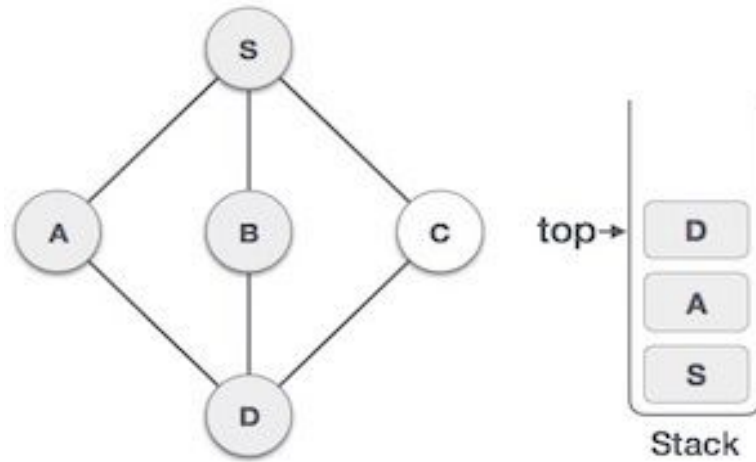
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5



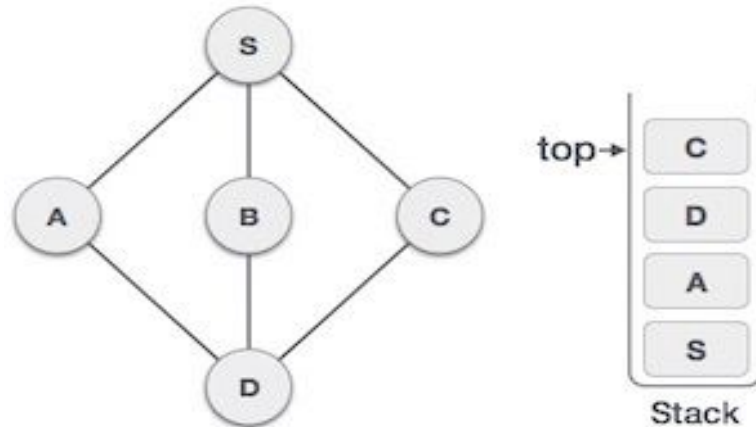
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.


Complexity of DFS Algorithm

Time Complexity


The time complexity of the DFS algorithm is represented in the form of $O(V + E)$, where V is the number of nodes and E is the number of edges.

Space Complexity

The space complexity of the DFS algorithm is $O(V)$.

The background of the slide features an abstract design on the right side, composed of several overlapping, semi-transparent green triangles and polygons of varying shades, creating a dynamic, layered effect. The left side of the slide is a solid white background where the code is located.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex {
    char label;
    bool visited;
};
//stack variables
int stack[MAX];
int top = -1;
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
```

The background of the slide features a series of overlapping, semi-transparent green triangles and polygons of various shades, creating a modern, abstract geometric pattern on the right side.

```
//stack functions
void push(int item) {
    stack[++top] = item;
}
int pop() {
    return stack[top--];
}
int peek() {
    return stack[top];
}
bool isEmpty() {
    return top == -1;
}

//graph functions
//add vertex to the vertex list
```

```
void addVertex(char label) {
    struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
    vertex->label = label;
    vertex->visited = false;
    lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
void addEdge(int start,int end) {
    adjMatrix[start][end] = 1;
    adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
    printf("%c ",lstVertices[vertexIndex]->label);
}
```


//get the adjacent unvisited vertex

```
int getAdjUnvisitedVertex(int vertexIndex) {
```

```
    int i;
```

```
    for(i = 0; i < vertexCount; i++) {
```

```
        if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false) {
```

```
            return i;
```

```
        }
```

```
    }
```

```
    return -1;
```

```
}
```

```
void depthFirstSearch() {
```

```
    int i;
```

```
    //mark first node as visited
```

```
    lstVertices[0]->visited = true;
```

```
    //display the vertex
```

```
    displayVertex(0);
```

```
    //push vertex index in stack
```

```
    push(0);
```

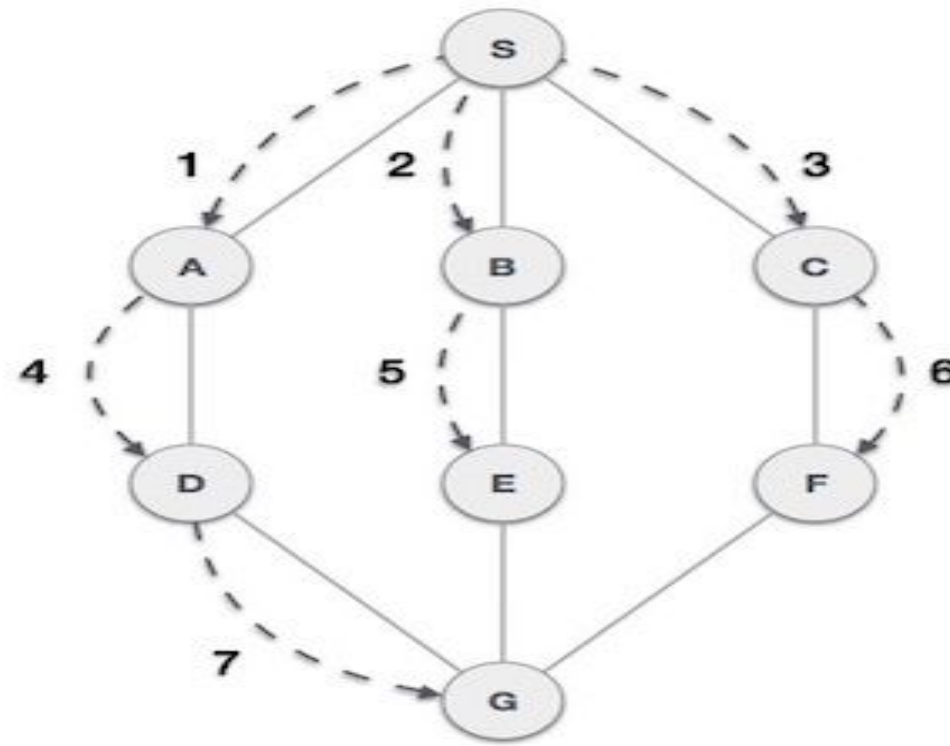
```
while(!isEmpty()) {  
    //get the unvisited vertex of vertex which is at top of the stack  
    int unvisitedVertex = getAdjUnvisitedVertex(peek());  
    //no adjacent vertex found  
    if(unvisitedVertex == -1) {  
        pop();  
    } else {  
        lstVertices[unvisitedVertex]->visited = true;  
        displayVertex(unvisitedVertex);  
        push(unvisitedVertex);  
    }  
}  
//stack is empty, search is complete, reset the visited flag  
for(i = 0; i < vertexCount; i++) {  
    lstVertices[i]->visited = false;  
}  
}
```

```
int main() {  
    int i, j;  
    for(i = 0; i < MAX; i++) { // set adjacency  
        for(j = 0; j < MAX; j++) // matrix to 0  
            adjMatrix[i][j] = 0;    }  
    addVertex('S'); // 0  
    addVertex('A'); // 1  
    addVertex('B'); // 2  
    addVertex('C'); // 3  
    addVertex('D'); // 4  
    addEdge(0, 1); // S - A  
    addEdge(0, 2); // S - B  
    addEdge(0, 3); // S - C  
    addEdge(1, 4); // A - D  
    addEdge(2, 4); // B - D  
    addEdge(3, 4); // C - D  
    printf("Depth First Search: ");  
    depthFirstSearch();  
    return 0;  
}
```

Breadth First Search (BFS) Algorithm :

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion to search a graph data structure for a node that meets a set of criteria. It uses a queue to remember the next vertex to start a search, when a dead end occurs in any iteration.

Breadth First Search (BFS) algorithm starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

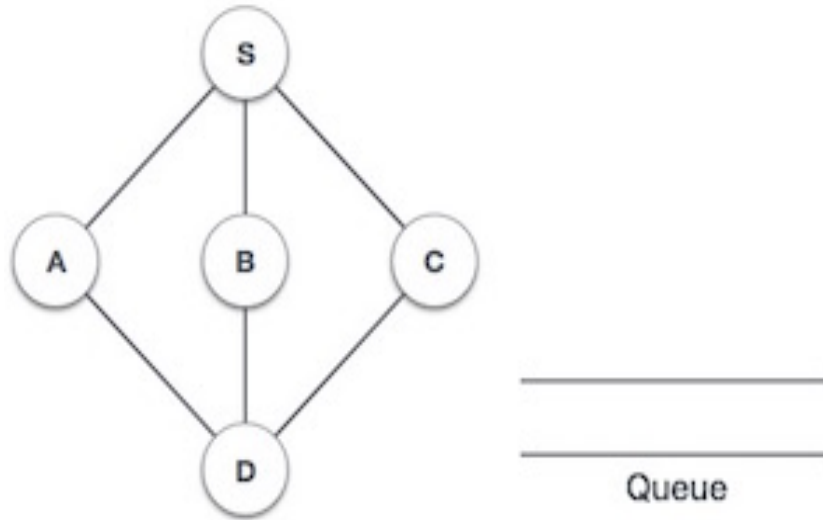
Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

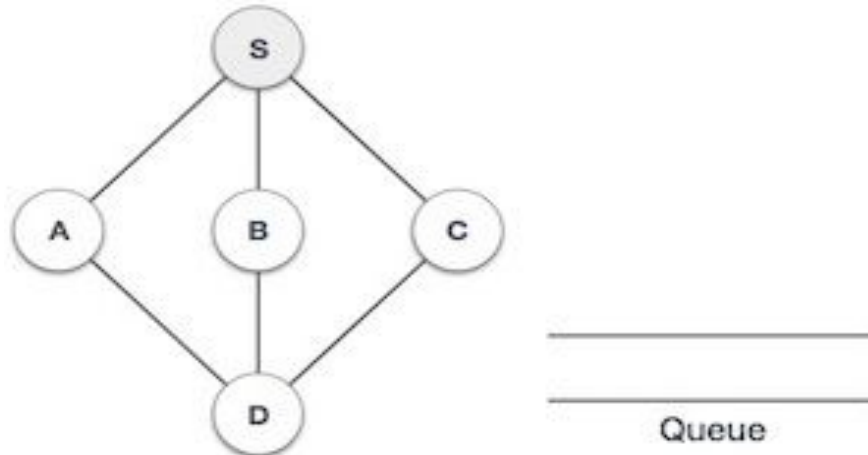
Step	Traversal	Description
------	-----------	-------------

1



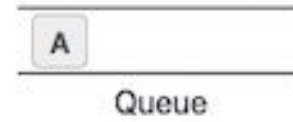
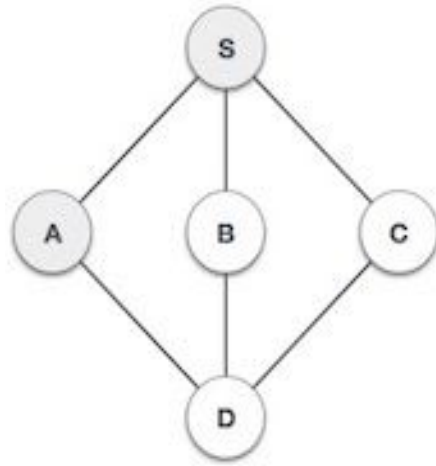
Initialize the queue.

2



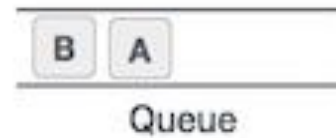
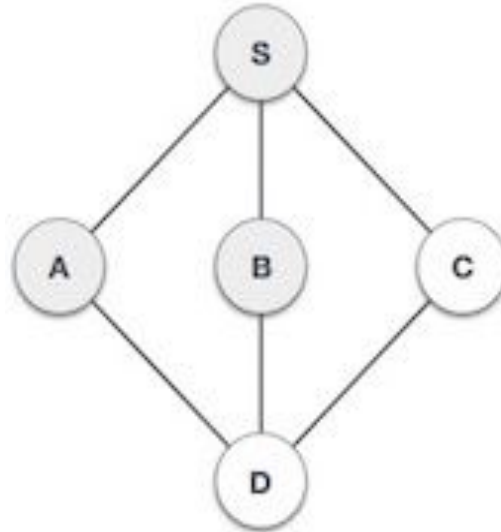
We start from visiting **S** (starting node), and mark it as visited.

3



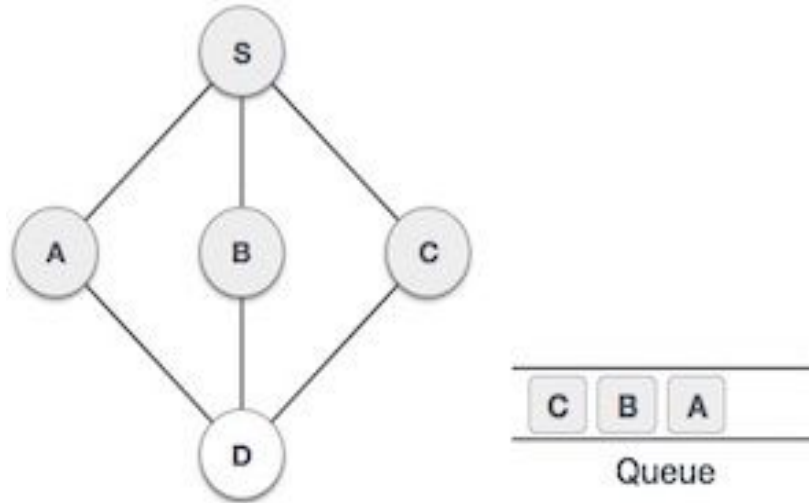
We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

4



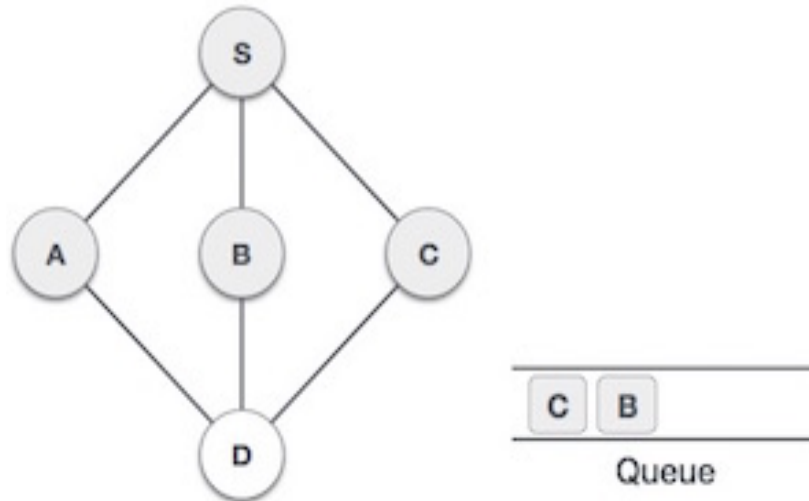
Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5



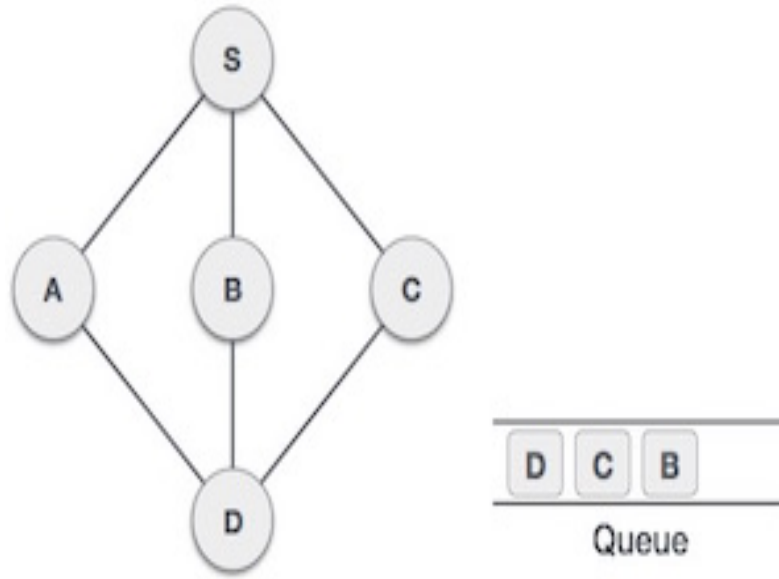
Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6



Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

7



From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

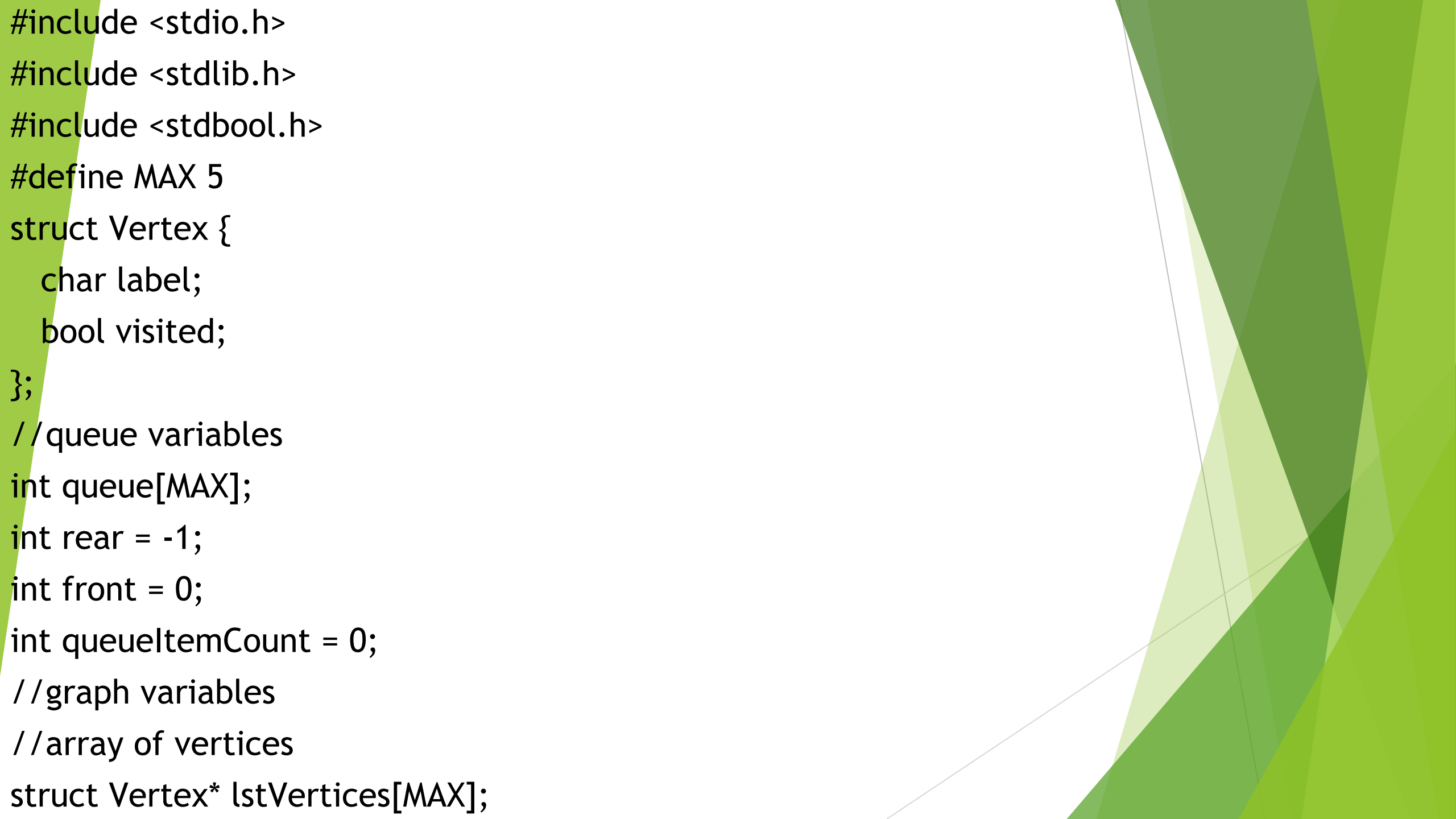
Complexity of BFS Algorithm

Time Complexity

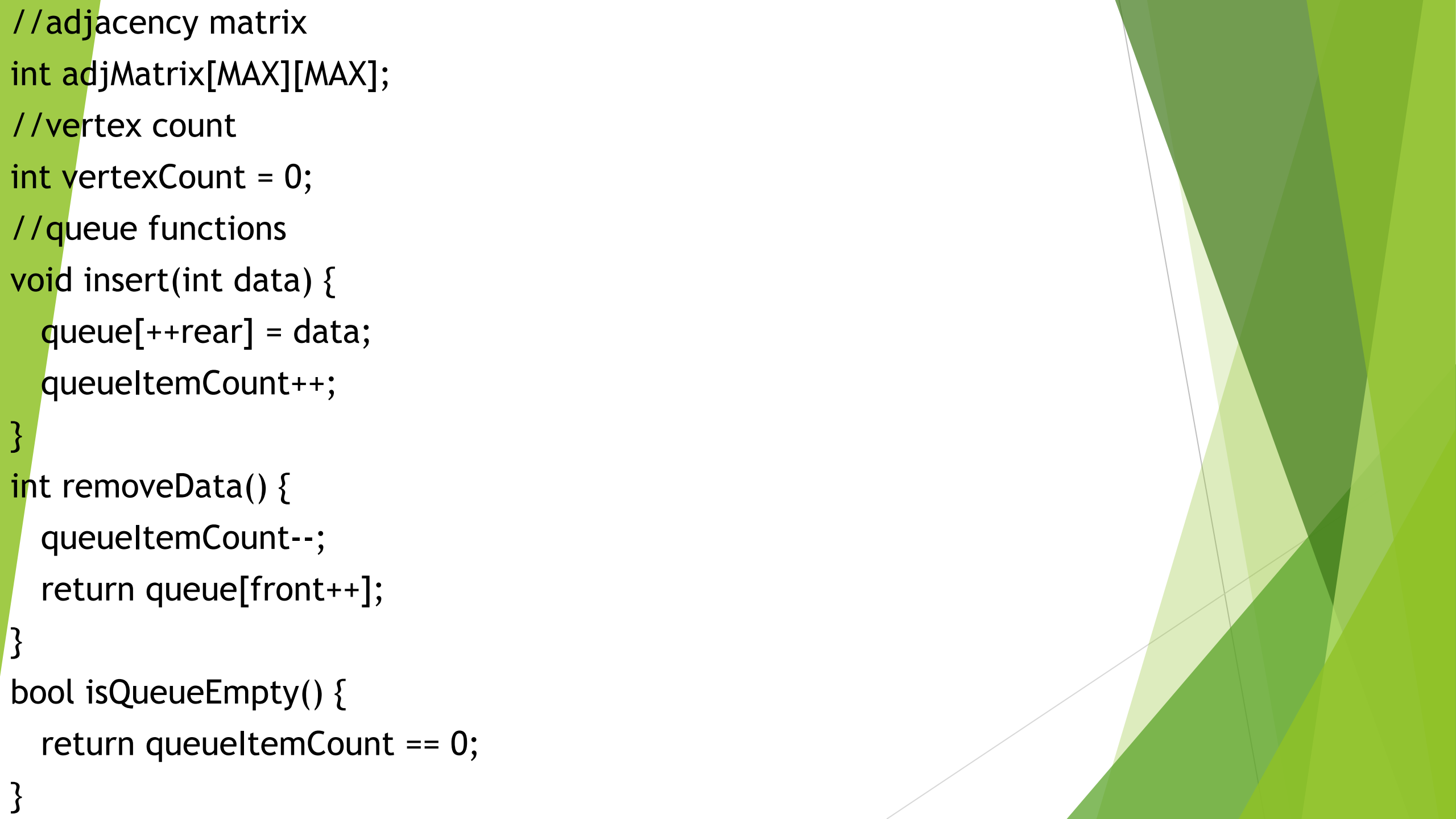
The time complexity of the BFS algorithm is represented in the form of $O(V + E)$, where V is the number of nodes and E is the number of edges.

Space Complexity

The space complexity of the BFS algorithm is $O(V)$.

The background of the slide features a series of overlapping, semi-transparent green triangles and polygons of various shades, creating a modern, abstract geometric pattern on the right side.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex {
    char label;
    bool visited;
};
//queue variables
int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];
```

The background features a series of overlapping, semi-transparent green triangles and polygons of various shades, creating a modern, abstract geometric pattern on the right side of the slide.

```
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
//queue functions
void insert(int data) {
    queue[++rear] = data;
    queueItemCount++;
}
int removeData() {
    queueItemCount--;
    return queue[front++];
}
bool isEmpty() {
    return queueItemCount == 0;
}
```

```
//graph functions
//add vertex to the vertex list
void addVertex(char label) {
    struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
    vertex->label = label;
    vertex->visited = false;
    lstVertices[vertexCount++] = vertex;
}
//add edge to edge array
void addEdge(int start,int end) {
    adjMatrix[start][end] = 1;
    adjMatrix[end][start] = 1;
}
//display the vertex
void displayVertex(int vertexIndex) {
    printf("%c ",lstVertices[vertexIndex]->label);
}
```

//get the adjacent unvisited vertex

```
int getAdjUnvisitedVertex(int vertexIndex) {  
    int i;  
    for(i = 0; i<vertexCount; i++) {  
        if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)  
            return i;  
    }  
    return -1;  
}
```

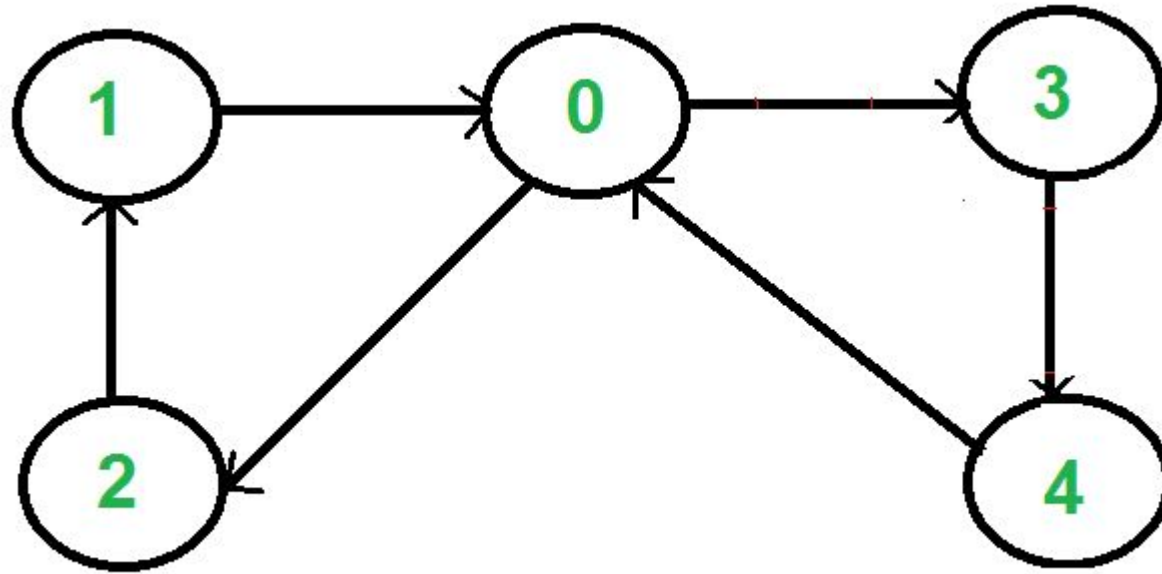
```
void breadthFirstSearch() {  
    int i;  
    //mark first node as visited  
    lstVertices[0]->visited = true;  
    //display the vertex  
    displayVertex(0);  
    //insert vertex index in queue  
    insert(0);  
    int unvisitedVertex;
```

```
while(!isEmpty()) {  
    //get the unvisited vertex of vertex which is at front of the queue  
    int tempVertex = removeData();  
    //no adjacent vertex found  
    while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {  
        lstVertices[unvisitedVertex]->visited = true;  
        displayVertex(unvisitedVertex);  
        insert(unvisitedVertex);  
    }  
}  
//queue is empty, search is complete, reset the visited flag  
for(i = 0;i<vertexCount;i++) {  
    lstVertices[i]->visited = false;  
}  
}
```

```
int main() {  
    int i, j;  
    for(i = 0; i<MAX; i++) { // set adjacency  
        for(j = 0; j<MAX; j++) // matrix to 0  
            adjMatrix[i][j] = 0;  
    }  
    addVertex('S'); // 0  
    addVertex('A'); // 1  
    addVertex('B'); // 2  
    addVertex('C'); // 3  
    addVertex('D'); // 4  
    addEdge(0, 1); // S - A  
    addEdge(0, 2); // S - B  
    addEdge(0, 3); // S - C  
    addEdge(1, 4); // A - D  
    addEdge(2, 4); // B - D  
    addEdge(3, 4); // C - D  
    printf("\nBreadth First Search: ");  
    breadthFirstSearch();  
    return 0;  
}
```


Directed Graph :

A directed graph is defined as a type of graph where the edges have a direction associated with them.



Characteristics of Directed Graph

Directed graphs have several characteristics that make them different from undirected graphs. Here are some key characteristics of directed graphs:

Directed edges: In a directed graph, edges have a direction associated with them, indicating a one-way relationship between vertices.

Indegree and Outdegree: Each vertex in a directed graph has two different degree measures: indegree and outdegree. Indegree is the number of incoming edges to a vertex, while outdegree is the number of outgoing edges from a vertex.

Cycles: A directed graph can contain cycles, which are paths that start and end at the same vertex and contain at least one edge. Cycles can be important for understanding feedback loops or other patterns in the graph.

Paths and reachability: Paths in a directed graph follow the direction of the edges, and can be used to analyze reachability between vertices.

Applications of Directed Graph

Directed graphs have many applications across a wide range of fields. Here are some examples:

Social networks: Social networks are often modeled as directed graphs, where each person is a vertex and relationships such as friendships or following are represented as edges.

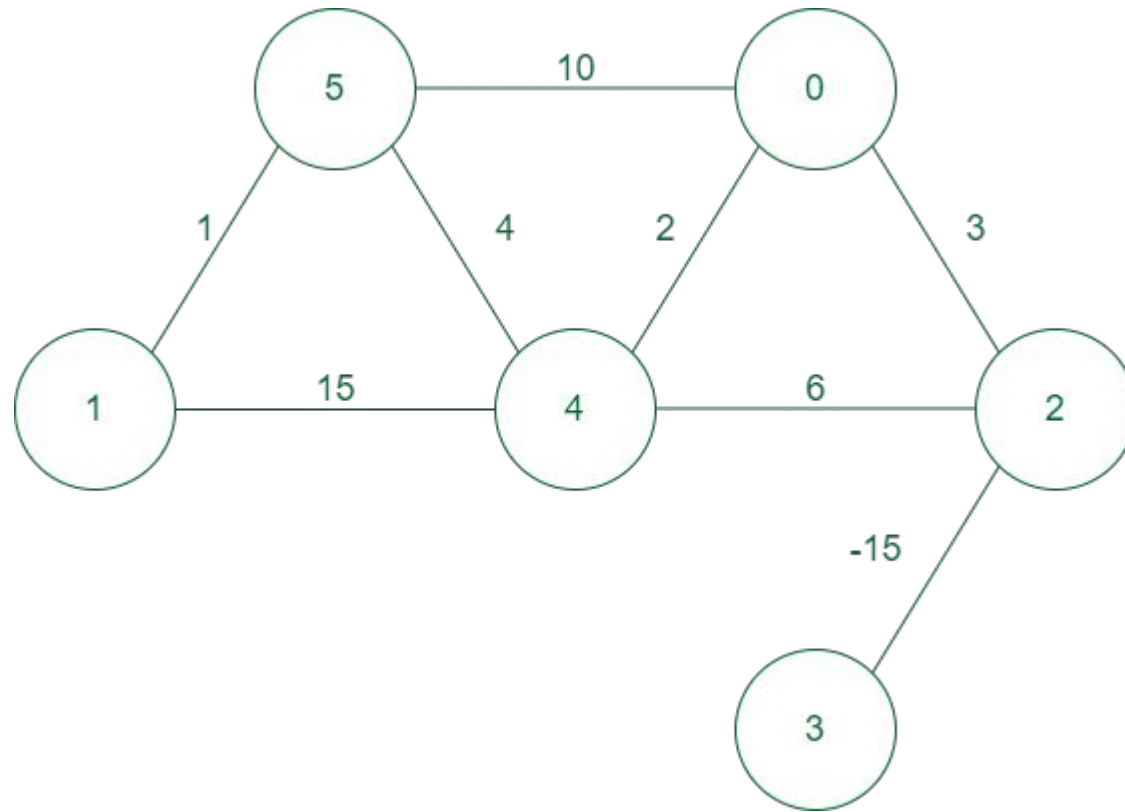
Transportation networks: Transportation systems such as roads, airports, or subway systems can be modeled as directed graphs, with vertices representing locations and edges representing connections between them.

Computer networks: Computer networks such as the internet can be represented as directed graphs, with vertices representing devices such as computers or routers and edges representing connections between them.

Project management: Project management can be modeled as a directed graph, with vertices representing tasks and edges representing dependencies between them.

Weighted Graph :

A **weighted graph** is defined as a special type of graph in which the edges are assigned some weights which represent cost, distance, and many other relative measuring units.



Applications of Weighted Graph:

2D matrix games: In 2d matrix, games can be used to find the optimal path for maximum sum along starting to ending points and many variations of it can be found online.

Spanning trees: Weighted graphs are used to find the minimum spanning tree from graph which depicts the minimal cost to traverse all nodes in the graph.

Constraints graphs: Graphs are often used to represent constraints among items. Used in scheduling, product design, asset allocation, circuit design, and artificial intelligence.

Dependency graphs: Directed weighted graphs can be used to represent dependencies or precedence order among items. Priority will be assigned to provide a flow in which we will solve the problem or traverse the graph from highest priority to lowest priority. Such graphs are often used in large projects in laying out what components rely on other components and are used to minimize the total time or cost to completion while abiding by the dependencies.

Compilers: Weighted graphs are used extensively in compilers. They can be used for type inference, for so-called data flow analysis, and many other purposes such as query optimization in database languages.

Artificial Intelligence: Weighted graphs are used in artificial intelligence for decision-making processes, such as in game trees for determining the best move in a game.

Image Processing: Weighted graphs are used in image processing for segmentation, where the weight of the edges represents the similarity between two pixels.

Natural Language Processing: Weighted graphs are used in natural language processing for text classification, where the weight of the edges represents the similarity between two words.

END