ASSINGMENT NO.	**6**
TITLE	Graph using Adjacency list
PROBLEM STATEMENT /DEFINITION	Represent a given graph using adjacency list to perform DFS and BFS. Use the map of the area around the college as the graph. Identify the prominent landmarks as nodes and perform DFS and BFS on that. **OR** There are fight paths between the cities. If there is a fight between the city A and city B then there is an edge between the city A to B. The cost of an edge represents the time required or fuel required to travel from city A to B. Represent this as a graph using adjacency list where every node of graph represented by city name. Perform following operations 1) calculate in degree, out degree of vertices 2) check whether graph is connected or not.
OBJECTIVE	To understand practical implementation and usage of Graph data structure for solving the problems.
OUTCOME	After successful completion of this assignment, students will be able to implement, use, and traverse graph data structure and perform operations on graph.
S/W PACKAGES AND HARDWARE APPARATUS USED	• (64-bit)64-BIT Fedora 17 or latest 64-bit update of equivalent Open source OS • Programming Tools (64-bit) Latest Open source update of Eclipse Programming frame work
REFERENCES	1. E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 2. Sartaj Sahani, ―Data Structures, Algorithms andApplications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. and title 3. Problem definition 4. Learning Objective 5. Learning Outcome 6. Software / Hardware requirement 7. Concepts related Theory 8. Algorithms/ Pseudo Code 9. Class ADT 10. Test cases 11. Conclusion/Analysis

Prerequisites:

- Basic knowledge of linked list, stack, queue, its operations and its implementation
- Object oriented programming features

Learning Objectives:

- To understand practical implementation and usage of Graph data structure for solving the problems.

Learning Outcomes:

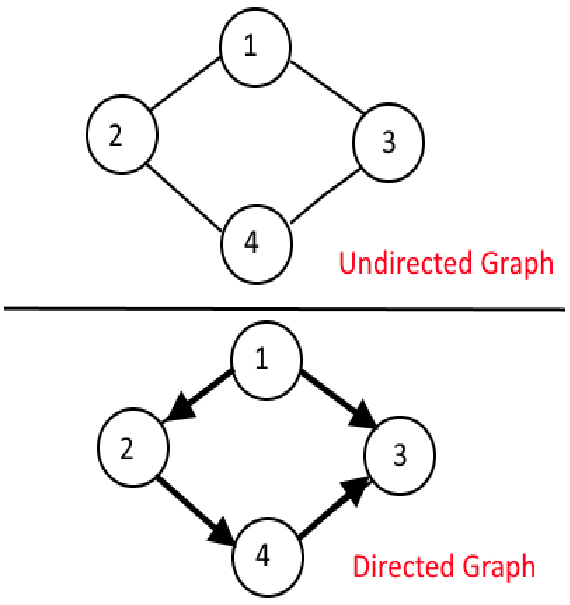
- After successful completion of this assignment, students will be able to implement, use, and traverse graph data structure and perform operations on graph.

Concepts related Theory:

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V). These edges might be weighted or non-weighted.

There can be two kinds of Graphs

- Un-directed Graph — when you can traverse either direction between two nodes.
- Directed Graph — when you can traverse only in the specified direction between two nodes.



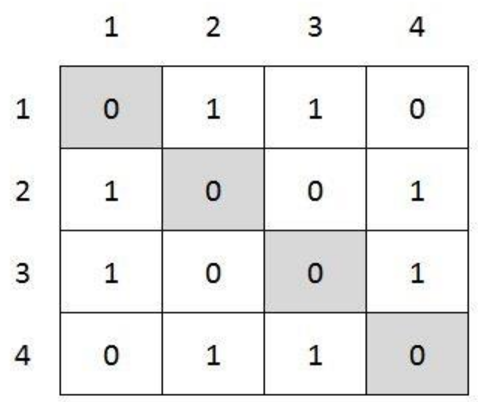
The following two are the most commonly used representations of a graph.

1. Adjacency Matrix

2. Adjacency List

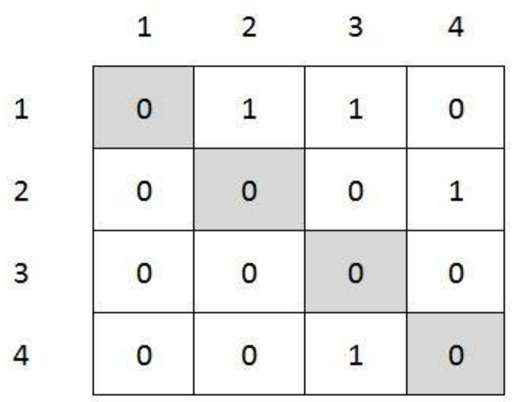
Adjacency Matrix:

Adjacency Matrix is 2-Dimensional Array which has the size VxV, where V are the number of vertices in the graph. See the example below, the Adjacency matrix for the graph shown above.



	1	2	3	4
1	0	1	1	0
2	1	0	0	1
3	1	0	0	1
4	0	1	1	0

Undirected Graph



	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	0	0	0	0
4	0	0	1	0

Directed Graph

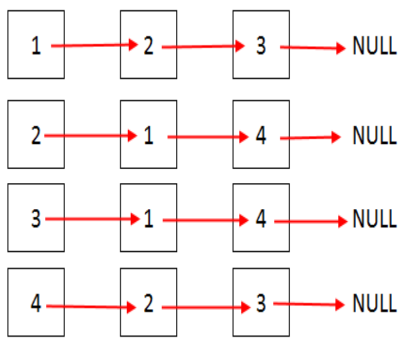
adjMaxtrix[i][j] = 1 when there is edge between Vertex i and Vertex j, else 0.

It’s easy to implement because removing and adding an edge takes only O(1) time.

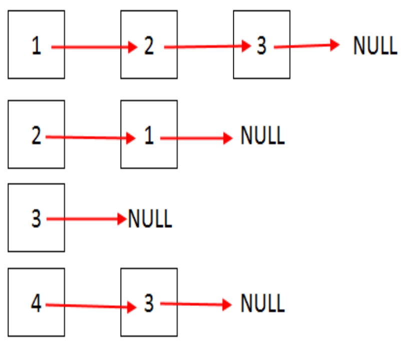
But the drawback is that it takes O(V^2) space even though there are very less edges in the graph.

Adjacency List:

Adjacency List is the Array[] of Linked List, where array size is same as number of Vertices in the graph. Every Vertex has a Linked List. Each Node in this Linked list represents the reference to the other vertices which share an edge with the current vertex. The weights can also be stored in the Linked List Node.



Adjacency List - Undirected Graph



Directed Graph

OPERATION:

The main operation performed by the adjacency list data structure is to report a list of the neighbors of a given vertex. This can be performed in constant time per neighbor. In other words, the total time to report all of the neighbors of a vertex *v* is proportional to the degree of v.

It is also possible, but not as efficient, to use adjacency lists to test whether an edge exists or does not exist between two specified vertices. In an adjacency list in which the neighbors of each vertex are unsorted, testing for the existence of an edge may be performed in time proportional to the minimum degree of the two given vertices, by using a sequential search through the neighbors of this vertex. If the neighbors are represented as a sorted array, binary search may be used instead, taking time proportional to the logarithm of the degree.

Trade-offs:

The main alternative to the adjacency list is the adjacency matrix, a matrix whose rows and columns are indexed by vertices and whose cells contain a Boolean value that indicates whether an edge is present between the vertices corresponding to the row and column of the cell. For a sparse graph (one in which most pairs of vertices are not connected by edges) an adjacency list is significantly more space-efficient than an adjacency matrix (stored as an array): the space usage of the adjacency list is proportional to the number of edges and vertices in the graph, while for an adjacency matrix stored in this way the space is proportional to the square of the number of vertices. However, it is possible to store adjacency matrices more space-efficiently, matching the linear space usage of an adjacency list, by using a hash table indexed by pairs of vertices rather than an array.

The other significant difference between adjacency lists and adjacency matrices is in the efficiency of the operations they perform. In an adjacency list, the neighbors of each vertex may be listed efficiently, in time proportional to the degree of the vertex. In an adjacency matrix, this operation takes time proportional to the number of vertices in the graph, which may be significantly higher than the degree. On the other hand, the adjacency matrix allows testing whether two vertices are adjacent to each other in constant time; the adjacency list is slower to support this operation.

What is a Graph Traversal Algorithm?

Graph traversal is a search technique for finding a vertex in a graph. In the search process, graph traversal is also used to determine the order in which it visits the vertices. Without producing loops, a graph traversal finds the edges to be employed in the search process.

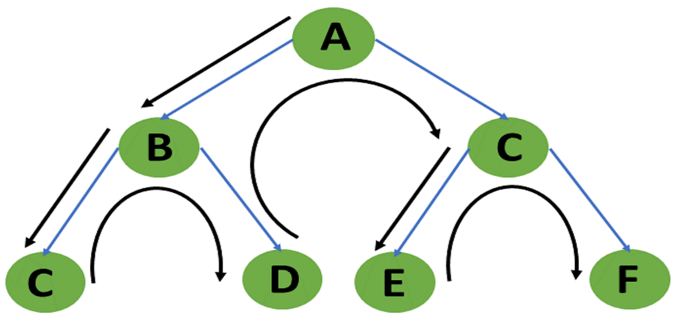
There are two methods to traverse a graph data structure:

Depth-First Search or DFS algorithm

Breadth-First Search or BFS algorithm

Depth-First Search Algorithm?

The depth-first search or DFS algorithm traverses or explores data structures, such as trees and graphs. The algorithm starts at the root node (in the case of a graph, you can use any random node as the root node) and examines each branch as far as possible before backtracking.



When a dead-end occurs in any iteration, the Depth First Search (DFS) method traverses a network in a deathward motion and uses a stack data structure to remember to acquire the next vertex to start a search.

Following the definition of the dfs algorithm, you will look at an example of a depth-first search method for a better understanding.

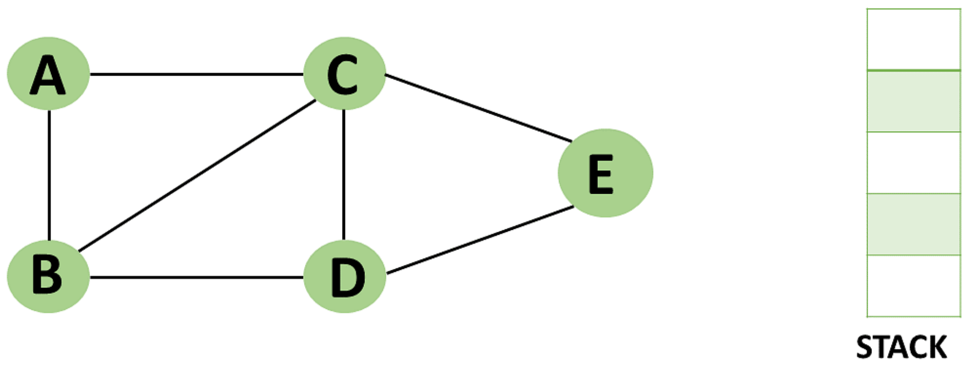
Example of Depth-First Search Algorithm

The outcome of a DFS traversal of a graph is a spanning tree. A spanning tree is a graph that is devoid of loops. To implement DFS traversal, you need to utilize a stack data structure with a maximum size equal to the total number of vertices in the graph.

To implement DFS traversal, you need to take the following stages.
Step 1: Create a stack with the total number of vertices in the graph as the size.
Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to the stack.

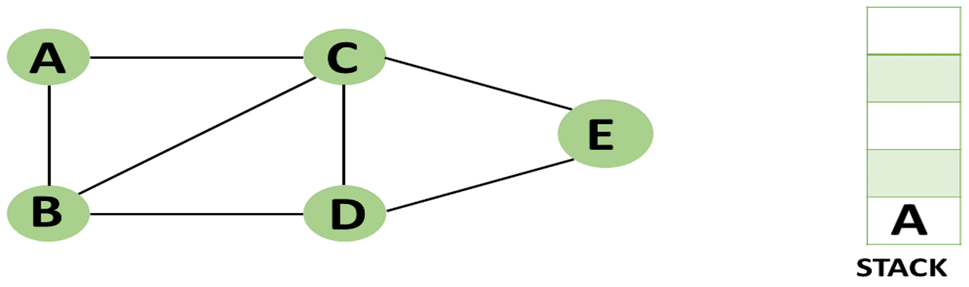
Step 3 - Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.
Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.
Step 5 - If there are no new vertices to visit, go back and pop one from the stack using backtracking.
Step 6 - Continue using steps 3, 4, and 5 until the stack is empty.
Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

Consider the following graph as an example of how to use the dfs algorithm.



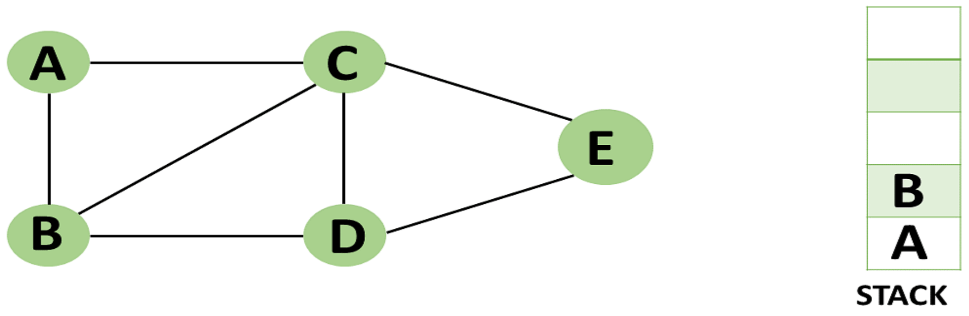
Step 1: Mark vertex A as a visited source node by selecting it as a source node.

- You should push vertex A to the top of the stack.



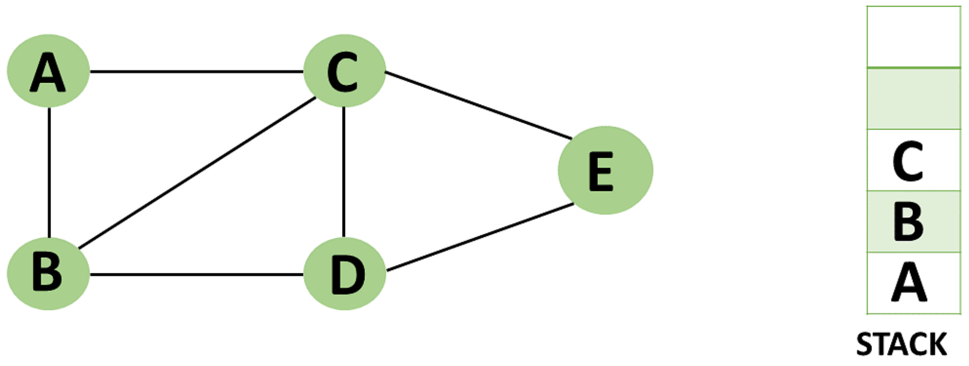
Step 2: Any nearby unvisited vertex of vertex A, say B, should be visited.

- You should push vertex B to the top of the stack.



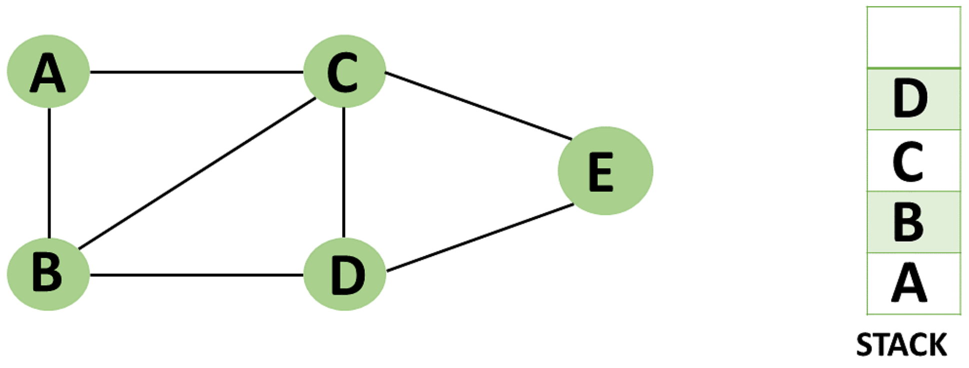
Step 3: From vertex C and D, visit any adjacent unvisited vertices of vertex B. Imagine you have chosen vertex C, and you want to make C a visited vertex.

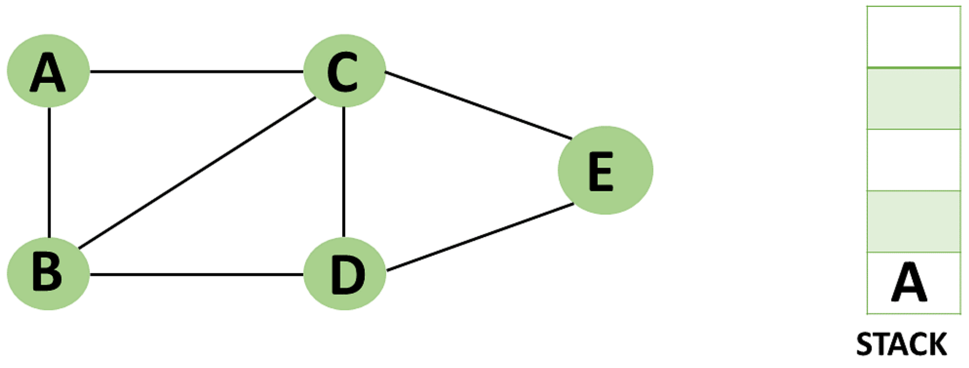
- Vertex C is pushed to the top of the stack.



Step 4: You can visit any nearby unvisited vertices of vertex C, you need to select vertex D and designate it as a visited vertex.

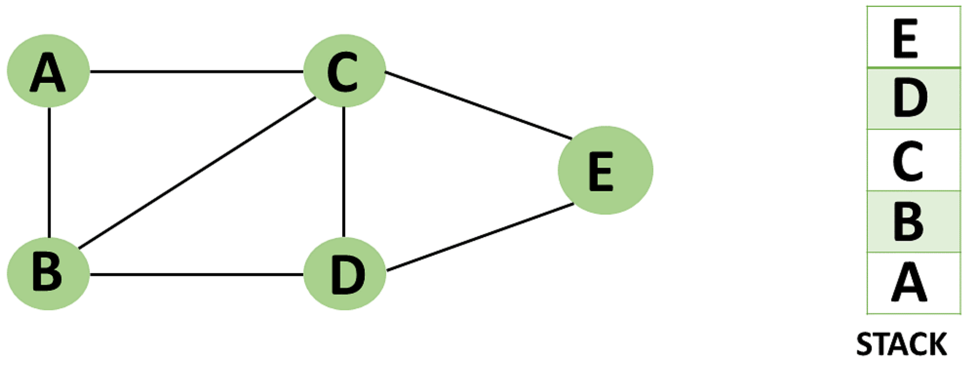
- Vertex D is pushed to the top of the stack.



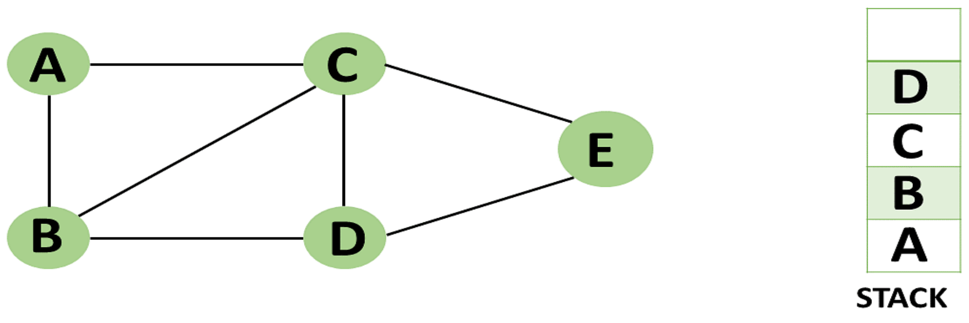


Step 5: Vertex E is the lone unvisited adjacent vertex of vertex D, thus marking it as visited.

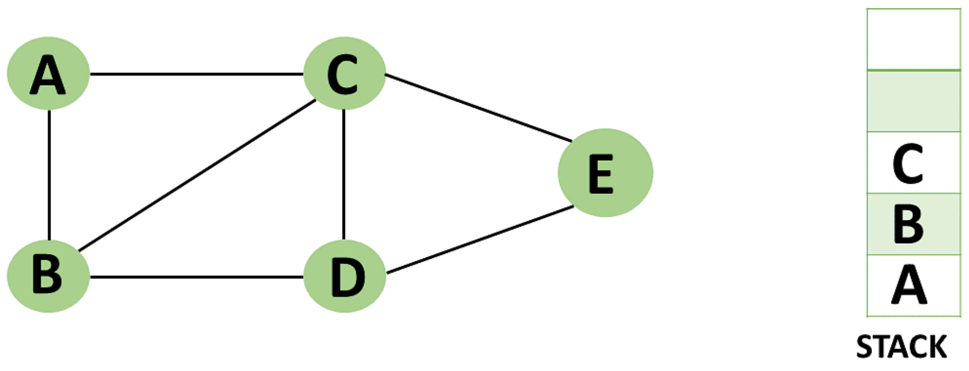
- Vertex E should be pushed to the top of the stack.



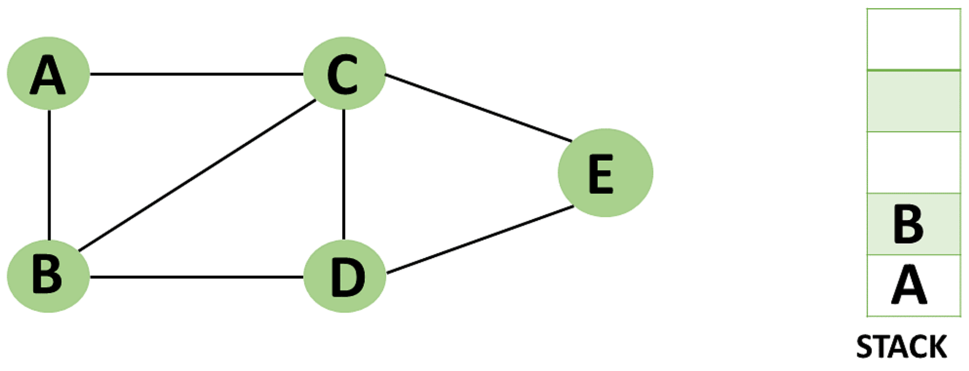
Step 6: Vertex E's nearby vertices, namely vertex C and D have been visited, pop vertex E from the stack.



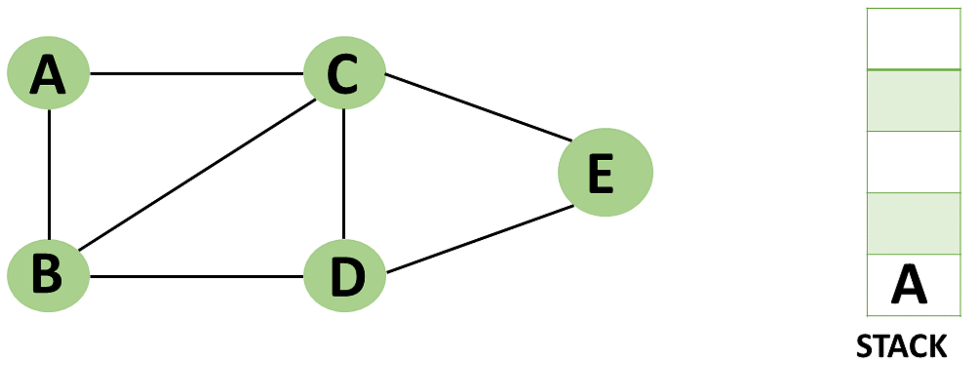
Step 7: Now that all of vertex D's nearby vertices, namely vertex B and C, have been visited, pop vertex D from the stack.



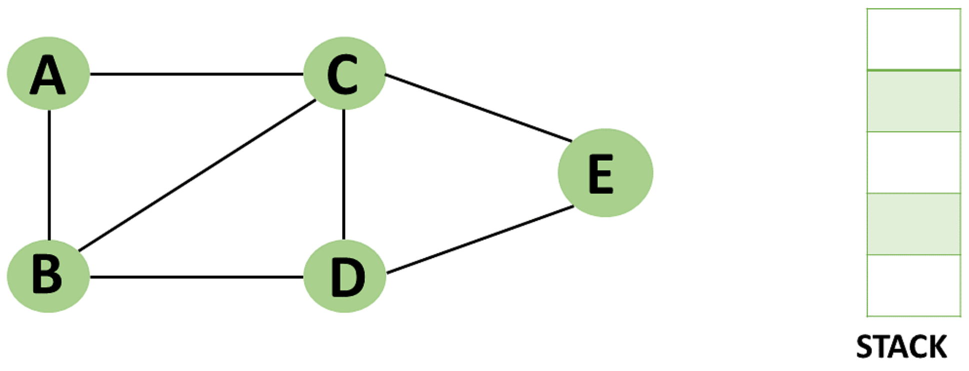
Step 8: Similarly, vertex C's adjacent vertices have already been visited; therefore, pop it from the stack.



Step 9: There is no more unvisited adjacent vertex of b, thus pop it from the stack.



Step 10: All of the nearby vertices of Vertex A, B, and C, have already been visited, so pop vertex A from the stack as well.



```
DFS-iterative (G, s):                                   //Where G is graph and s
is source vertex
      let S be stack
      S.push( s )            //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not
visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                     S.push( w )
                    mark w as visited


    DFS-recursive(G, s):
        mark s as visited
        for all neighbours w of s in Graph G:
            if w is not visited:
                DFS-recursive(G, w)
```

Breadth first search: Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

BFS algorithm A standard BFS implementation puts each vertex of the graph into one of two categories: Visited and Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

```
BFS (G, s)                   //Where G is the graph and s is the
source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its
neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour
will be visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                        Q.enqueue( w )             //Stores w in
Q to further visit its neighbour
                    mark w as visited.
```

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node this will also solved weather the graph is connected or not(graph connectivity problem)

TEST CASES:

Check there adjacency list find all the edges that are directly connected to a cities or not.

Conclusion:

Thus we have studied adjacency list representation of the graph successfully for cities and performed traversal on graph.

Review Questions:

1. What is Graph? Explain its uses.
2. Explain Adjacency List.
3. Explain Adjacency Matrix.
4. Explain why stack and queues are used in dfs and bfs?
5. What is the difference between undirected and directed graph?
6. Explain Sparse graph.