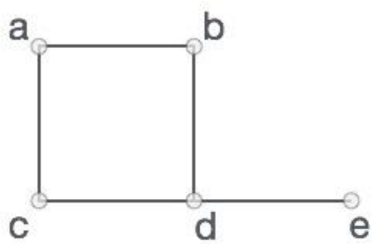
Graph Data Structure

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by 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}

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.

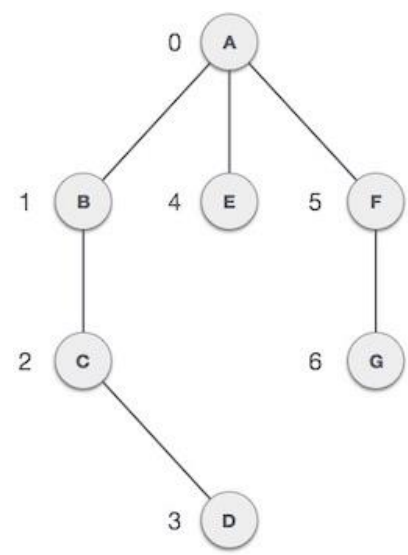
Important Terms

- **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.



Basic Primary Operations of a Graph

- **Add Vertex** − Adds a vertex to the graph.

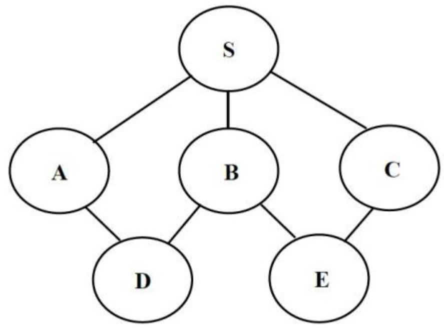
- **Add Edge** − Adds an edge between the two vertices of the graph.

- **Display Vertex** − Displays a vertex of the graph.

```c
#include <stdio.h>
#include <stdlib.h>


#define char VType

#define MAX 10

// Definition and declaration of vertex

typedef struct  {
   VType label;
   bool visited;
}vertex_t;


// Graph variables

// Array of vertices

vertex_t* lstvertices[MAX];

// Adjacency Matrix

int adjMatrix[MAX][MAX];

// Vertex Count

int vertexCount = 0;


// Graph functions

// Add vertex to the vertex list

void addVertex(VType newlabel) {
   vertex_t * vertex = (vertex_t*) malloc(sizeof(vertex_t));
   vertex->label = newlabel;
   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;
}

// Set all vertices as unvisited

void setUnvisited(vertex_t lstVertices[], int vertexCount) {

   int i;
   for (i = 0; i < vertexCount; i++)
         lstvertices[i]->visited == false;

}
```

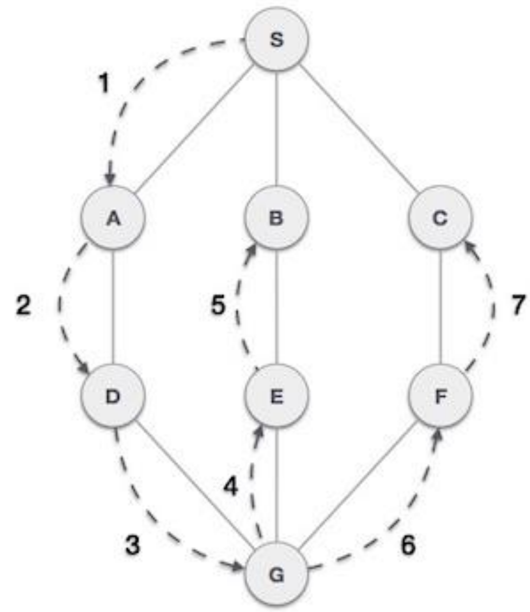
Example main program for the following graph:




```
int main(void) {

	int i, j;
	// Set adjacency matrix to 0
	for (i = 0; i < MAX; i++)
			for (j = 0; j < MAX; j++)
				adjMatrix[i][j] = 0;

	addVertex('S');		// 0
	addVertex('A');		// 1
	addVertex('B');		// 2
	addVertex('C');		// 3
	addVertex('D');		// 4
	addVertex('E');		// 5

	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(2, 5);		// B - E
	addEdge(3, 5);		// C - E

	….


}
```

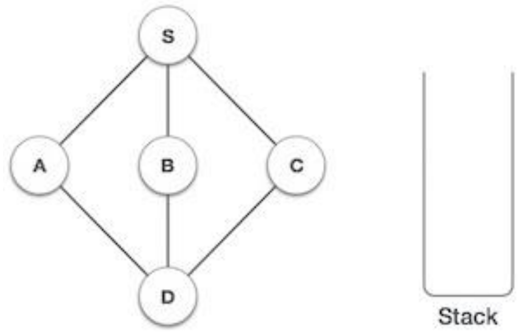
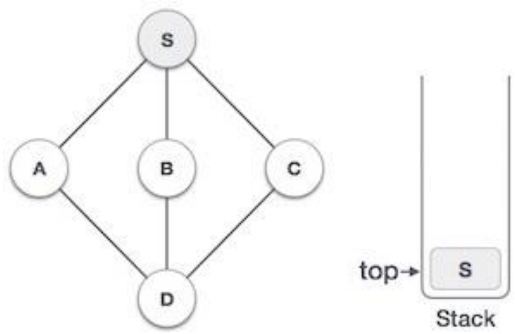
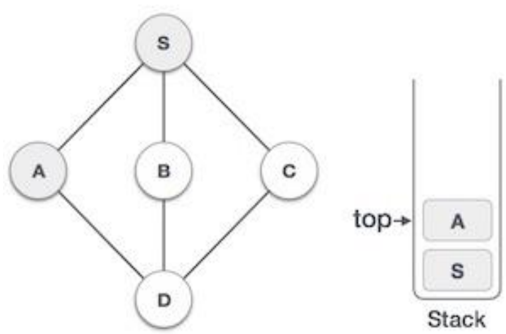
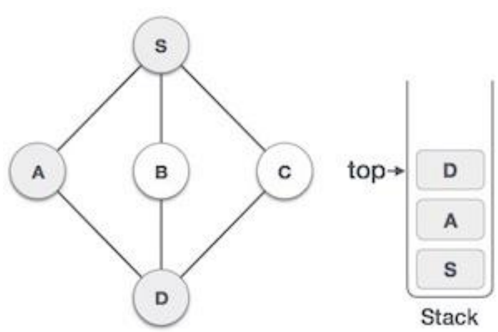
Depth First Traversal

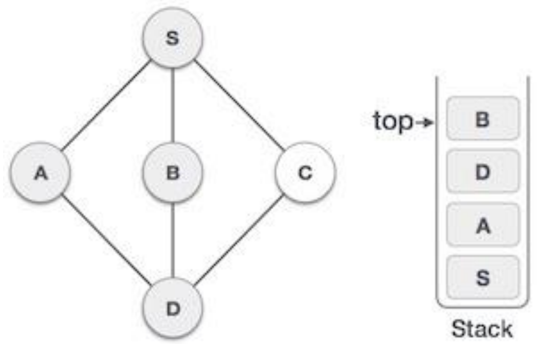
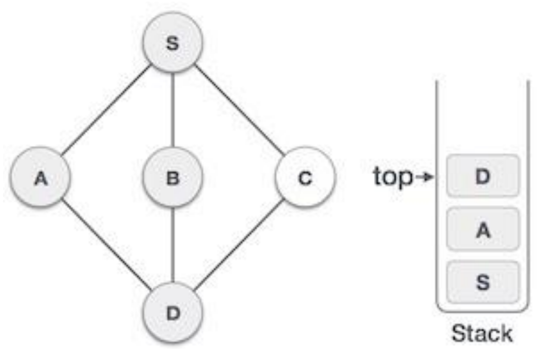
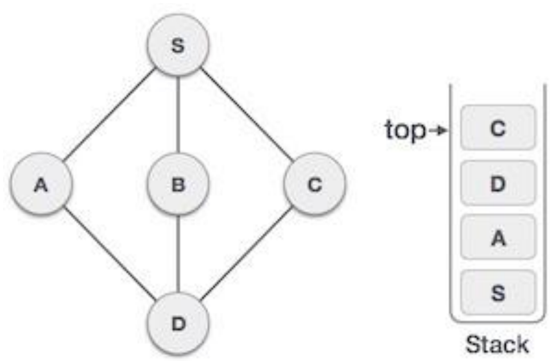
Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



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.

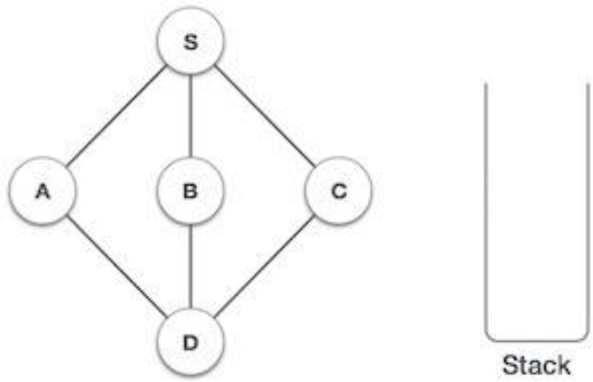
Step	Traversal	Description
1.		Initialize the stack.
2.		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark **A** as visited and put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

Depth First Traversal in C

For our reference purpose, we shall follow our example and take this as our graph model −



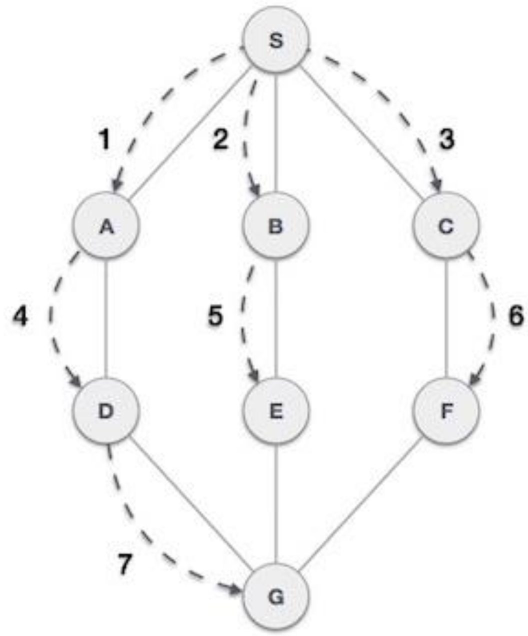
Example: Implement the given graph and apply the depth-first search algorithm.

Your code will produce the following result:

S A D B C

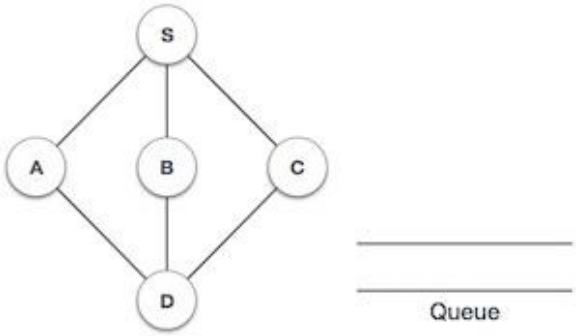
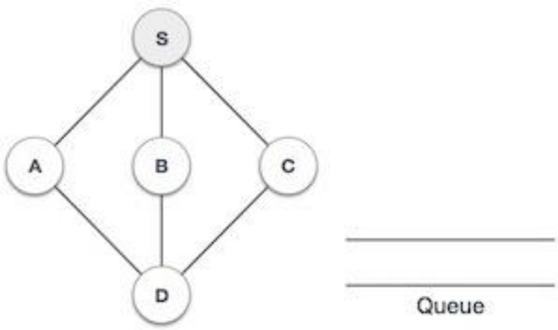
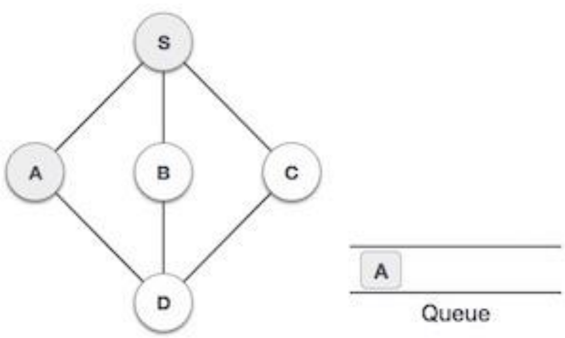
Breadth First Traversal

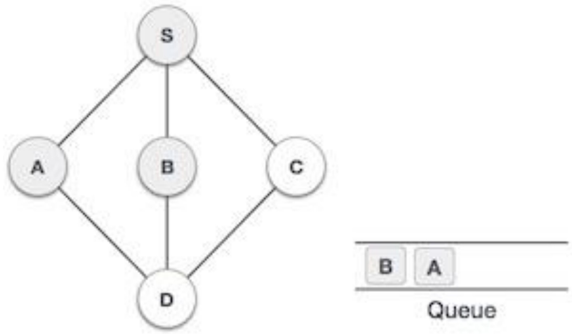
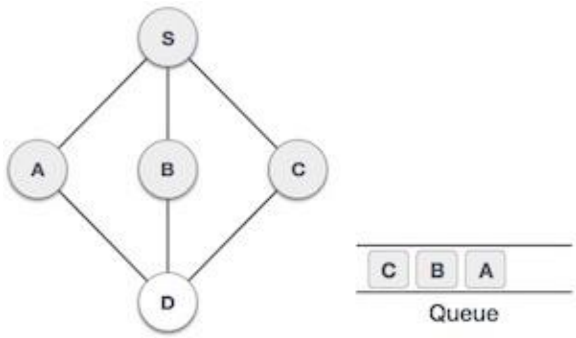
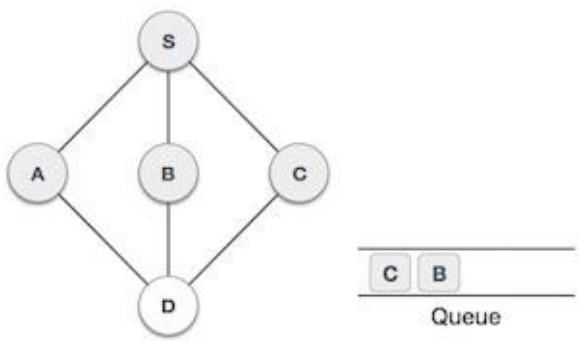
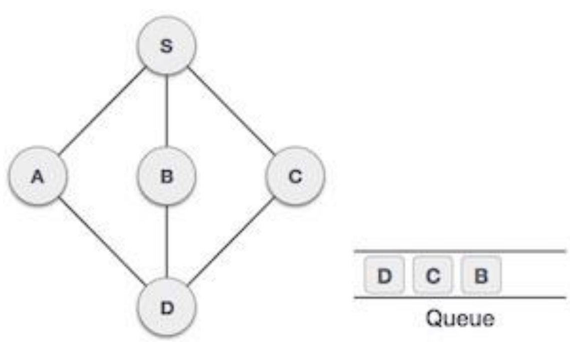
Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



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

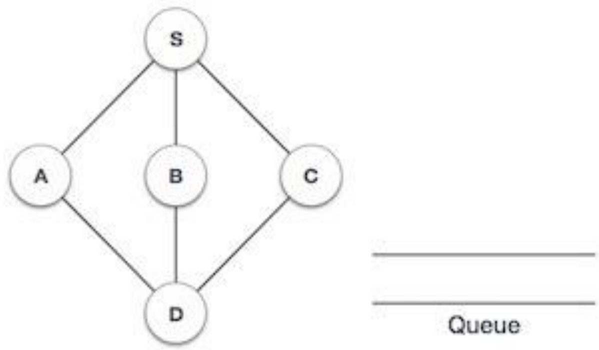
Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting **S**(starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

<table>
<tr><td>4.</td><td>
</td><td>Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.</td></tr>
<tr><td>5.</td><td>
</td><td>Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.</td></tr>
<tr><td>6.</td><td>
</td><td>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</td></tr>
<tr><td>7.</td><td>
</td><td>From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.</td></tr>
</table>

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.

Breadth First Traversal in C

For our reference purpose, we shall follow our example and take this as our graph model −



Example: Implement the given graph and apply the depth-first search algorithm.

Your code will produce the following result:

S A B C D

Reference: https://www.tutorialspoint.com/index.htm

```
// Stack function for the peek value

int peek(stack_t st) {
    return pop(&st);
}


void depthFirstSearch(void)
{
      int unvisitedVertex;
      stack_t S;
      initializeS(&S);

      setUnvisited(listVertices, vertexCount);

      //mark first node as visited
      lstVertices[0]->visited = TRUE;

      //display the vertex
      printf("%c ", lstVertices[0]->label);

      //push vertex index in stack
      push(&S, 0);

      while (!isEmptyS(&S))
      {
          //get the unvisited vertex of vertex which is at top of the stack
          unvisitedVertex = getAdjUnvisitedVertex(peek(S));

          //no adjacent vertex found
          if (unvisitedVertex == -1)
              pop(&S);
          else
          {
              lstVertices[unvisitedVertex]->visited = true;
              printf("%c ", lstVertices[unvisitedVertex]->label);
              push(&S, unvisitedVertex);
          }
      }

}
```

```c
// Queue function for the peek value

int peek(queue_t q) {
   return remove(&q);
}



void breadthFirstSearch(void)
{
     int unvisitedVertex;

     queue_t Q;
     initializeQ(&Q);

     setUnvisited(listVertices, vertexCount);

     //mark first node as visited
     lstVertices[0]->visited = TRUE;

     //display the vertex
     printf("%c ", lstVertices[0]->label);

     //insert vertex index in queue
     insert(&Q, 0);

     while (!isEmptyQ(&Q))
     {
        //get the unvisited vertex of vertex which is at front of the queue
        unvisitedVertex = getAdjUnvisitedVertex(peek(Q));

        if (unvisitedVertex == -1)
          remove(&Q);
        else
        {
          lstVertices[unvisitedVertex]->visited = true;
          printf("%c ", lstVertices[unvisitedVertex]->label);
          insert(&Q, unvisitedVertex);
        }
     }
}
```