

GRAPHS

Graphs are mathematical structures that represent pairwise relationships between objects. A graph is a flow structure that represents the relationship between various objects. It can be visualized by using the following two basic components:

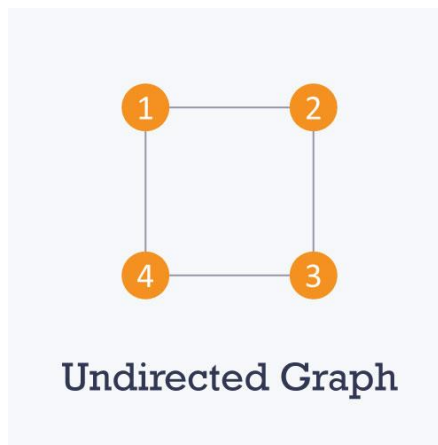
- **Nodes:** These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- **Edges:** Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

Types of nodes

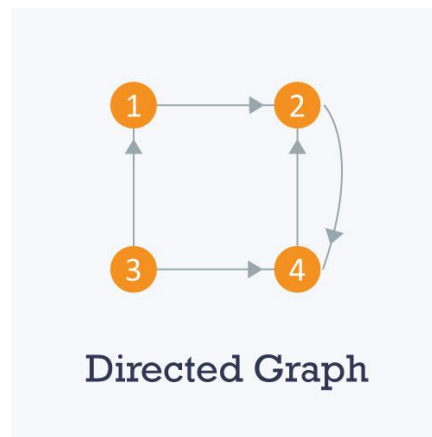
- **Root node:** The root node is the ancestor of all other nodes in a graph. It does not have any ancestor. Each graph consists of exactly one root node. Generally, you must start traversing a graph from the root node.
- **Leaf nodes:** In a graph, leaf nodes represent the nodes that do not have any successors. These nodes only have ancestor nodes. They can have any number of incoming edges but they will not have any outgoing edges.

Types of graphs

- **Undirected:** An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction

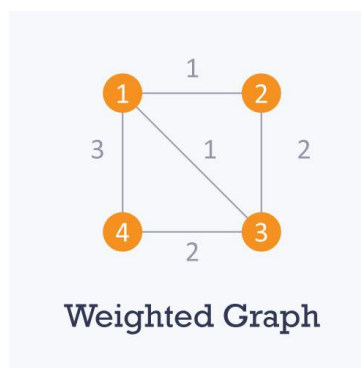


- Directed: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



- Weighted: In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:
 - 1 -> 2 -> 3
 - 1 -> 3
 - 1 -> 4 -> 3

Therefore the total cost of each path will be as follows: - The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units

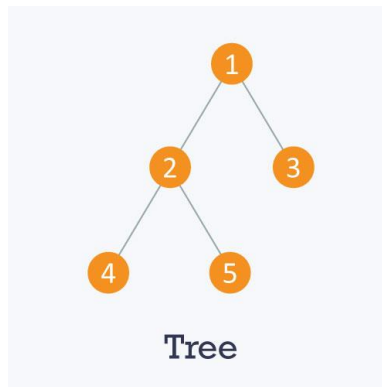


- Cyclic: A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

A **tree** is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has $N - 1$ edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

Note: A root node has no parent.

A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



Graph representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

Adjacency matrix

An adjacency matrix is a $V \times V$ binary matrix A . Element $A_{i,j}$ is 1 if there is an edge from vertex i to vertex j else $A_{i,j}$ is 0.

Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

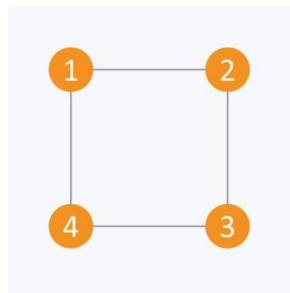
The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in $A_{i,j}$, the weight or cost of the edge will be stored.

In an undirected graph, if $A_{i,j} = 1$, then $A_{j,i} = 1$. In a directed graph, if $A_{i,j} = 1$, then $A_{j,i}$ may or may not be 1.

Adjacency matrix provides **constant time access** ($O(1)$) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is $O(V^2)$.

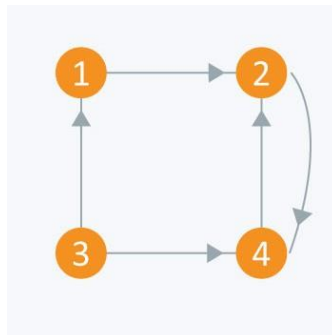
The adjacency matrix of the following graph is:

i/j : 1 2 3 4
1 : 0 1 0 1
2 : 1 0 1 0
3 : 0 1 0 1
4 : 1 0 1 0



The adjacency matrix of the following graph is:

i/j: 1 2 3 4
1 : 0 1 0 0
2 : 0 0 0 1
3 : 1 0 0 1
4 : 0 1 0 0



Consider the directed graph given above. Let's create this graph using an adjacency matrix and then show all the edges that exist in the graph.

Input file

4 // nodes
5 //edges

```
1 2 //showing edge from node 1 to node 2
2 4 //showing edge from node 2 to node 4
3 1 //showing edge from node 3 to node 1
3 4 //showing edge from node 3 to node 4
4 2 //showing edge from node 4 to node 2
```

Adjacency Matrix:

```
# include < stdio.h >
void main()
{
    int ch;
    while(1)
    {
        printf("\n A Program to represent a Graph by using an ");
        printf("Adjacency Matrix method \n ");
        printf("\n 1. Directed Graph ");
        printf("\n 2. Un-Directed Graph ");
        printf("\n 3. Exit ");
        printf("\n\n Select a proper option : ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1 : dir_graph();
                    break;
            case 2 : undir_graph();
                    break;
            case 3 : exit(0);
        } // switch
    }
} // main

int dir_graph()
{
    int adj_mat[50][50];
    int n;
    int in_deg, out_deg, i, j;
```

```

printf("\n How Many Vertices ? : ");
scanf("%d", &n);

read_graph (adj_mat, n);
printf("\n Vertex \t In_Degree \t Out_Degree \t Total_Degree ");
for (i = 1; i <= n ; i++ )
{
    in_deg = out_deg = 0;
    for ( j = 1 ; j <= n ; j++ )
    {
        if ( adj_mat[j][i] == 1 )
            in_deg++;
    }
    for ( j = 1 ; j <= n ; j++ )
        if (adj_mat[i][j] == 1 )
            out_deg++;
    printf("\n\n
%5d\t\t\t%d\t\t%d\t\t%d\n\n",i,in_deg,out_deg,in_deg+out_deg);
} // for
return;
} // dir_graph

```

```

int undir_graph()
{
    int adj_mat[50][50];
    int deg, i, j, n;

    printf("\n How Many Vertices ? : ");
    scanf("%d", &n);

    read_graph(adj_mat, n);
    printf("\n Vertex \t Degree ");

    for ( i = 1 ; i <= n ; i++ )
    {

```

```

        deg = 0;
        for ( j = 1 ; j <= n ; j++ )
            if ( adj_mat[i][j] == 1 )
                deg++;
        printf("\n\n %5d \t\t %d\n\n", i, deg);
    } // for
    return;
} // undir_graph

int read_graph ( int adj_mat[50][50], int n )
{
    int i, j;
    int reply;

    for ( i = 1 ; i <= n ; i++ )
    {
        for ( j = 1 ; j <= n ; j++ )
        {
            if ( i == j )
            {
                adj_mat[i][j] = 0;
                continue;
            } // if
            printf("\n Vertices %d & %d are Adjacent ? (Y/N) :\n enter 1 for adjacent,2 for non
adjacent nodes",i,j);
            scanf("%d", &reply);
            if ( reply == 1 )
                adj_mat[i][j] = 1;
            else
                adj_mat[i][j] = 0;
        } // for
    } // for
    return;
}

```

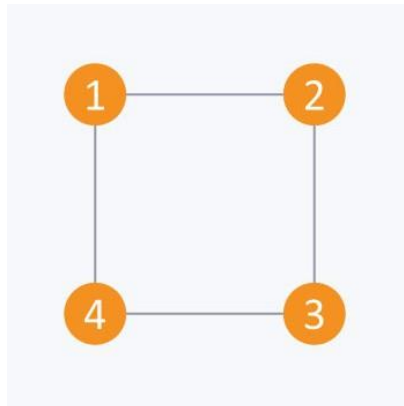
Adjacency list

The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array A_i is a list, which contains all the vertices that are adjacent to vertex i .

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list A_i then vertex i will be in list A_j .

The space complexity of adjacency list is $O(V + E)$ because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.

Note: A sparse matrix is a matrix in which most of the elements are zero, whereas a dense matrix is a matrix in which most of the elements are non-zero.



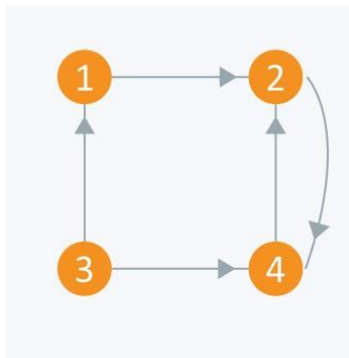
Consider the same undirected graph from an adjacency matrix. The adjacency list of the graph is as follows:

$A_1 \rightarrow 2 \rightarrow 4$

$A_2 \rightarrow 1 \rightarrow 3$

$A_3 \rightarrow 2 \rightarrow 4$

A4 → 1 → 3



Consider the same directed graph from an adjacency matrix. The adjacency list of the graph is as follows:

A1 → 2

A2 → 4

A3 → 1 → 4

A4 → 2

Adjacency List Representation:

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

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

```
typedef struct node
```

```
{
```

```
    int vertex;
```

```
    struct node *next;
```

```
}N;
```

```
void main()
```

```
{
```

```
    int ch;
```

```
while(1)
```

```
{
```

```
    printf("\n A Program to represent a Graph by using an Adjacency List \n ");
```

```
    printf("\n 1. Directed Graph ");
```

```
    printf("\n 2. Un-Directed Graph ");
```

```
    printf("\n 3. Exit ");
```

```
    printf("\n\n Select a proper option : ");
```

```
    scanf("%d", &ch);
```

```
    switch(ch)
```

```

    {
        case 1 : dir_graph();
            break;
        case 2 : undir_graph();
            break;
        case 3 : exit(0);
    }
}
}
int dir_graph()
{
    N *adj_list[10], *p;
    int n;
    int in_deg, out_deg, i, j;
    printf("\n How Many Vertices ? : ");
    scanf("%d", &n);
    for( i = 1 ; i <= n ; i++ )
        adj_list[i] = NULL;
    read_graph (adj_list, n);
    printf("\n Vertex \t In_Degree \t Out_Degree \t Total_Degree ");
    for ( i = 1; i <= n ; i++ )
    {
        in_deg = out_deg = 0;
        p = adj_list[i];
        while( p != NULL )
        {
            out_deg++;
            p = p -> next;
        }
        for ( j = 1 ; j <= n ; j++ )
        {
            p = adj_list[j];
            while( p != NULL )
            {
                if ( p -> vertex == i )
                    in_deg++;
                p = p -> next;
            }
        }
    }
}

```

```

        printf("\n\n %5d\t\t\t%d\t\t%d\t\t%d\n\n", i, in_deg, out_deg, in_deg +
out_deg);
    }
    return;
}
int undir_graph()
{
    N *adj_list[10], *p;
    int deg, i, j, n;
    printf("\n How Many Vertices ? : ");
    scanf("%d", &n);
    for ( i = 1 ; i <= n ; i++ )
        adj_list[i] = NULL;
    read_graph(adj_list, n);
    printf("\n Vertex \t Degree ");
    for ( i = 1 ; i <= n ; i++ )
    {
        deg = 0;
        p = adj_list[i];
        while( p != NULL )
        {
            deg++;
            p = p -> next;
        }
        printf("\n\n %5d \t\t %d\n\n", i, deg);
    }
    return;
}
int read_graph ( N *adj_list[10], int n )
{
    int i, j;
    int reply;
    N *p, *c;
    for ( i = 1 ; i <= n ; i++ )
    {
        for ( j = 1 ; j <= n ; j++ )
        {
            if ( i == j )
                continue;

```

```

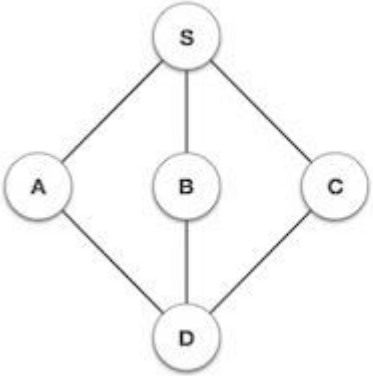

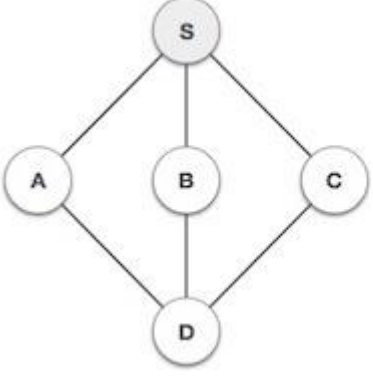

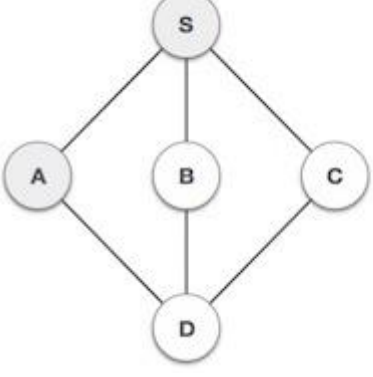
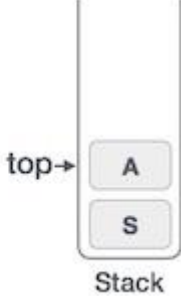
printf("\n Vertices %d & %d are Adjacent ? (Y/N) : \nenter 1 for adjacent
nodes, 2 for non adjacent nodes", i, j);
scanf("%d", &reply);
if ( reply == 1 )
{
    c = (N*)malloc(sizeof(N));
    c -> vertex = j;
    c -> next = NULL;
    if ( adj_list[i] == NULL )
        adj_list[i] = c;
    else
    {
        p = adj_list[i];
        while ( p -> next != NULL )
            p = p -> next;
        p -> next = c;
    }
}
}
}
return;
}

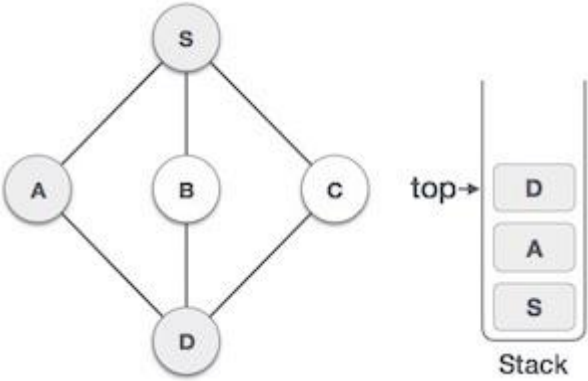
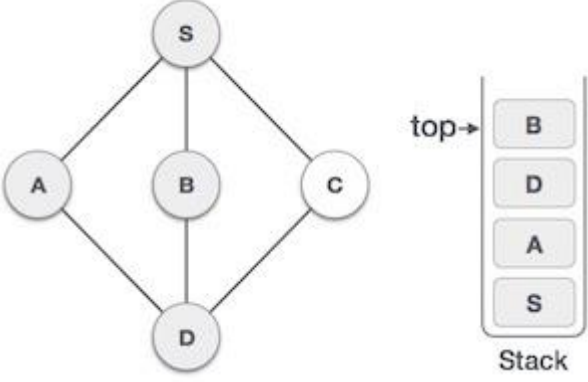
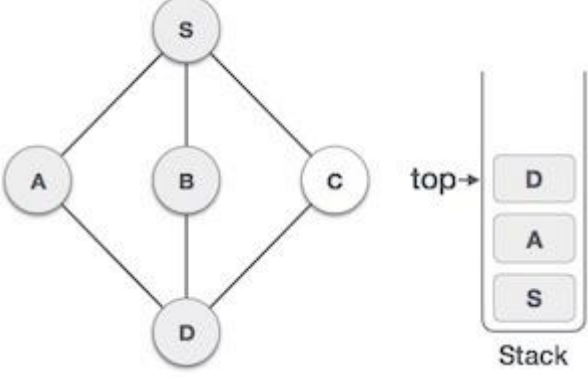
```

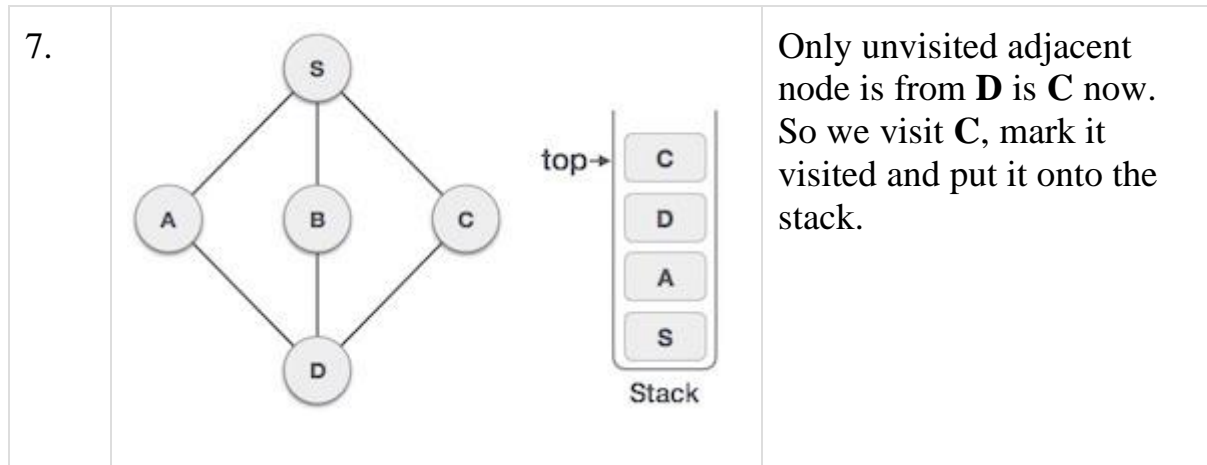
DFS:

Depth First Search algorithm(DFS) 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.

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 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.	 <p>The graph shows a diamond-shaped structure with nodes S at the top, A, B, and C in the middle row, and D at the bottom. Edges connect S to A, B, and C; A to D; B to D; and C to D. To the right, a stack is shown with nodes D, A, and S from top to bottom. An arrow labeled 'top' points to the D node.</p>	<p>Visit D and mark it visited and put onto the stack. Here we have B and C nodes which are adjacent to D and both are unvisited. But we shall again choose in alphabetical order.</p>
5.	 <p>The graph is identical to the previous step. The stack now contains nodes B, D, A, and S from top to bottom. An arrow labeled 'top' points to the B node.</p>	<p>We choose B, mark it visited and put onto stack. Here B does not have any unvisited adjacent node. So we pop B from the stack.</p>
6.	 <p>The graph is identical to the previous step. The stack now contains nodes D, A, and S from top to bottom. An arrow labeled 'top' points to the D node.</p>	<p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find D to be on the top of stack.</p>



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

DFS ALGORITHM:

$n \leftarrow$ number of nodes

Initialize visited[] to false (0)

for($i=0; i < n; i++$)

visited[i] = 0;

void DFS(vertex i) [DFS starting from i]

{

visited[i]=1;

for each w adjacent to i

if(!visited[w])

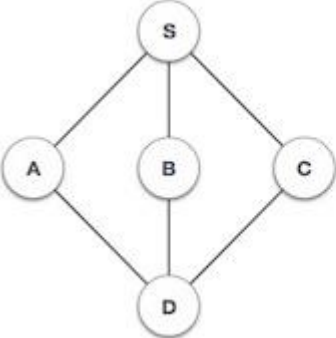
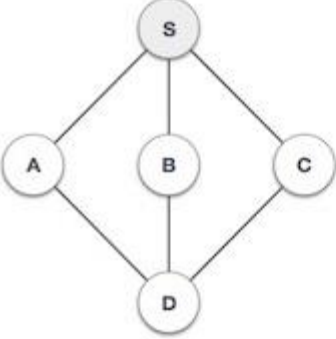
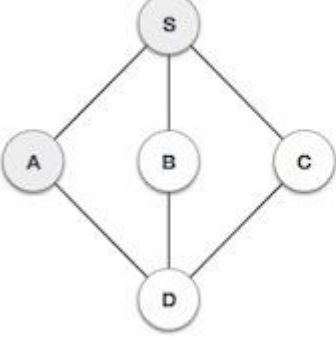
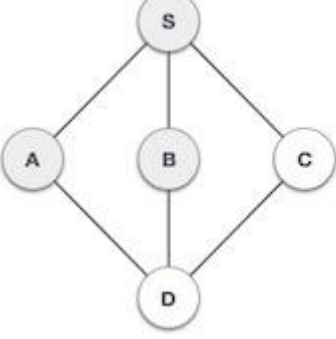
DFS(w);

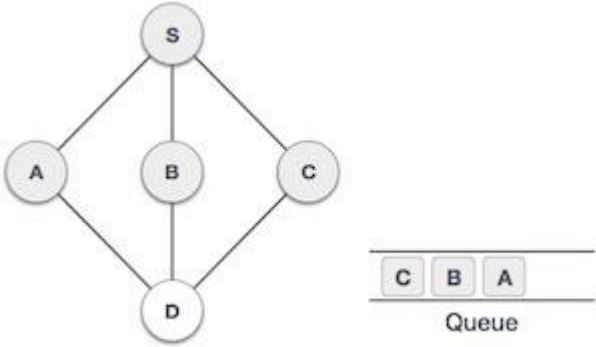
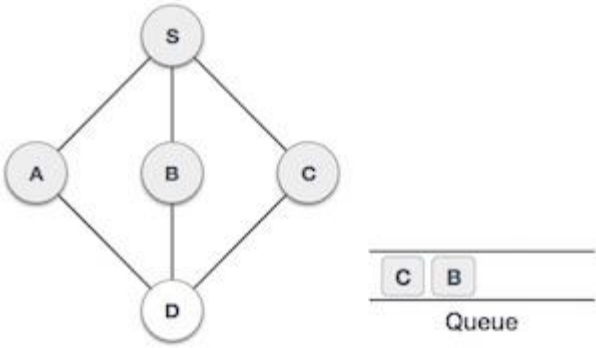
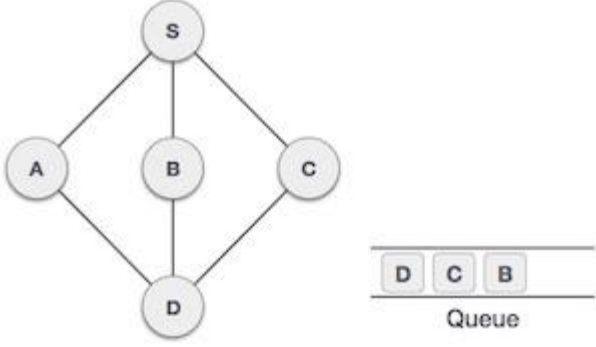
}

BFS:

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

Step	Traversal	Description

1.	 <hr/> <hr/> <p style="text-align: center;">Queue</p>	Initialize the queue.
2.	 <hr/> <hr/> <p style="text-align: center;">Queue</p>	We start from visiting S (starting node), and mark it visited.
3.	 <hr/> <div style="border: 1px solid black; display: inline-block; padding: 2px;">A</div> <hr/> <p style="text-align: center;">Queue</p>	We then see unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A mark it visited and enqueue it.
4.	 <hr/> <div style="border: 1px solid black; display: inline-block; padding: 2px;">B</div> <div style="border: 1px solid black; display: inline-block; padding: 2px; margin-left: 5px;">A</div> <hr/> <p style="text-align: center;">Queue</p>	Next unvisited adjacent node from S is B . We mark it visited and enqueue it.

5.		Next unvisited adjacent node from S is C . We mark it 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 visited and enqueue it.

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

DIFFERENCE BETWEEN DFS AND BFS:

BFS	DFS
BFS Stands for “ Breadth First Search ”.	DFS stands for “ Depth First Search ”.
BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.	DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
Breadth First Search can be done with the help of queue i.e. FIFO implementation. This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.	Depth First Search can be done with the help of Stack i.e. LIFO implementations. This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off.
BFS is slower than DFS. BFS requires more memory compare to DFS.	DFS is more faster than BFS. DFS require less memory compare to BFS.
Applications of BFS <ul style="list-style-type: none"> > To find Shortest path > Single Source & All pairs shortest paths > In Spanning tree > In Connectivity 	Applications of DFS <ul style="list-style-type: none"> > Useful in Cycle detection > In Connectivity testing > Finding a path between V and W in the graph. > useful in finding spanning trees & forest.

BFS is useful in finding shortest path. BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.

DFS is not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (**backtrack**) to add branches also as long as possible.



A, B, C, D, E, F

Example :

Example :



A, B, D, C, E, F