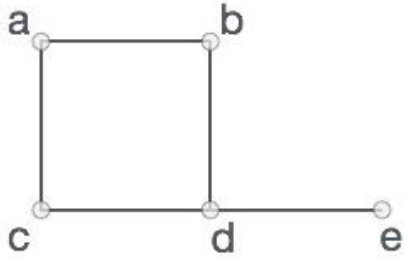
Graphs - Background

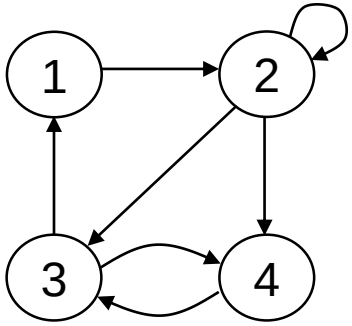
Graphs = a set of nodes (vertices) with edges (links) between them.

Notations:

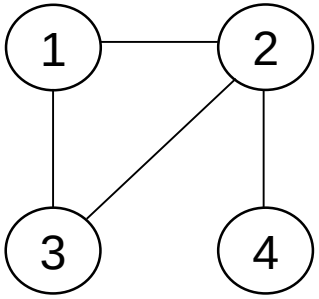
- G = (V, E) - graph
- V = set of vertices $|V| = n$
- E = set of edges $|E| = m$



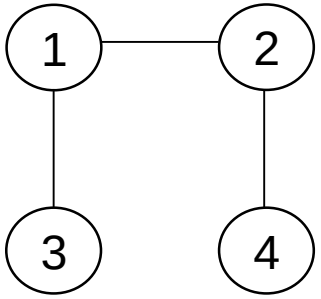
In the above graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}



Directed graph



Undirected graph



Acyclic graph

Graph Data Structure

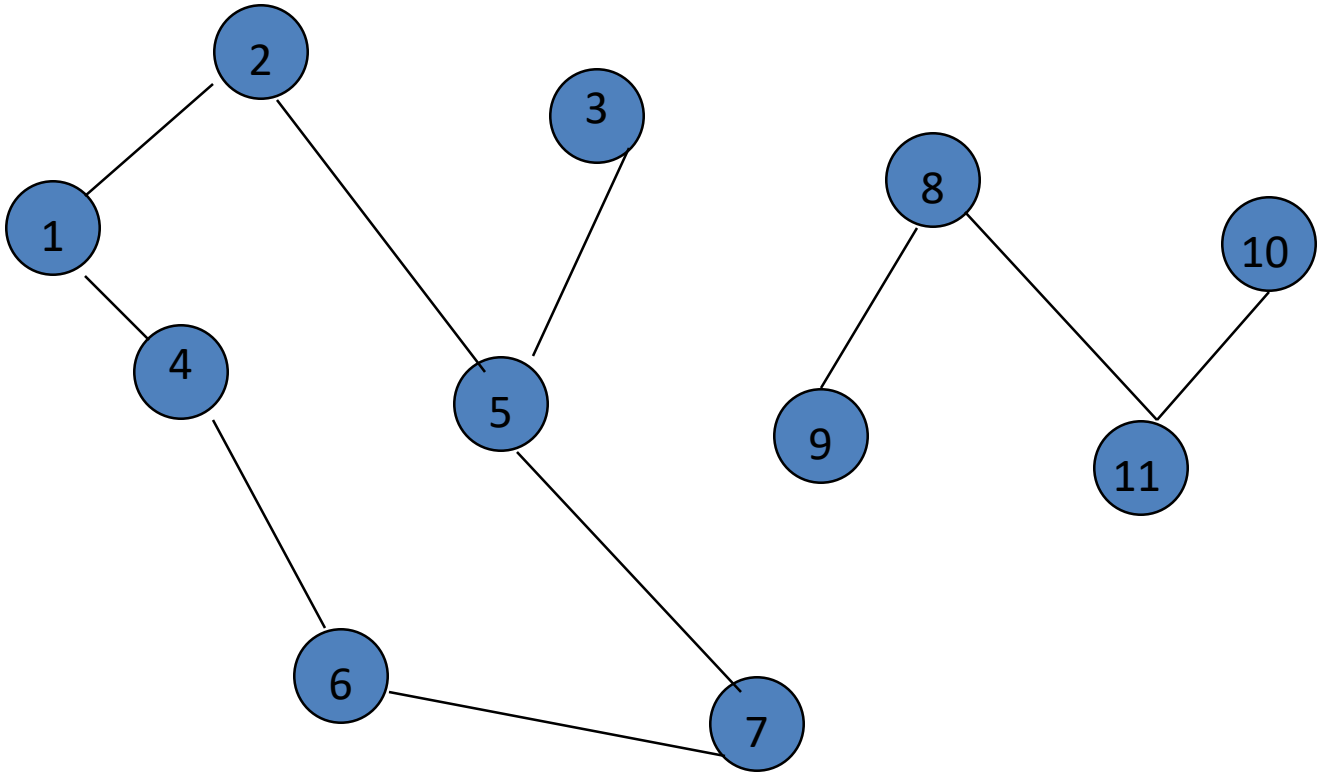
- Mathematical graphs can be represented in data-structure.
- We can represent a graph using an array of vertices and a two dimensional array of edges.
- Some important terms −
  - **Vertex** − Each node of the graph is represented as a vertex.
  - **Edge** − Edge represents a path between two vertices or a line between two vertices.
  - **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In example given earlier, b is adjacent to a, d, c is adjacent to a, b and so on.
  - **Path** − Path represents a sequence of edges between two vertices. In example given previous, abde represents a path from a to e.

Basic Operations

Following are basic primary operations of a Graph which are following.

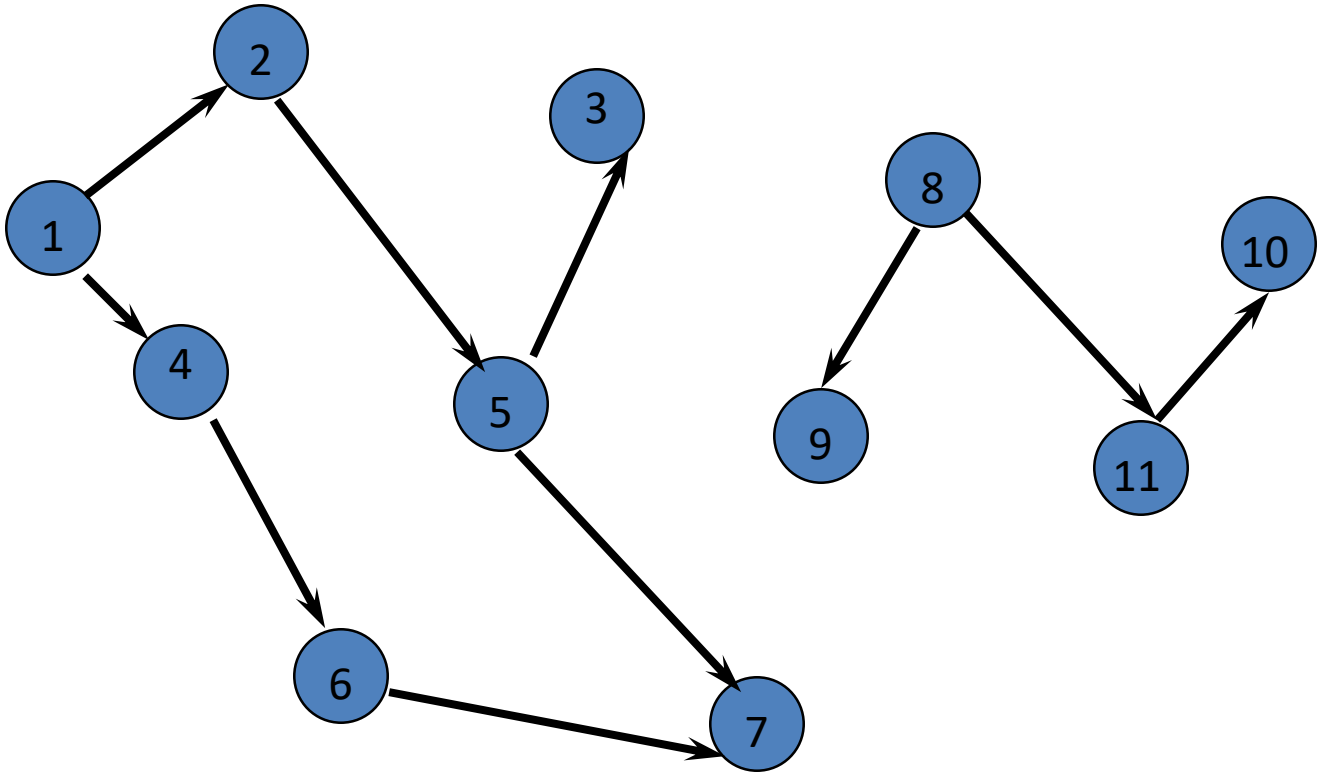
- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

Undirected Graph



In the above graph,
V = {1,2,3,4,5,6,7,8,9,10,11}
E = {12, 14, 25, 35, 46, 57, 67, 89, 811, 1011}

Directed Graph (Digraph)



Applications

It has applications in diverse fields of science and engineering:

- **Computer Science** – Graph theory is used for the study of various algorithms.
- **Computer Network** – The relationships among interconnected computers in the network follows the principles of graph theory.
- **The graphic representation of** world wide web (www)
- **Resource allocation graph** for processes that are active in the system.
- **The graphic representation of a map**
- **Scene graphs:** The contents of a visual scene are also managed by using graph data structure.
- The concepts of graph theory is used extensively in designing **circuit connections**. Some examples for topologies are star, bridge, series, and parallel topologies.

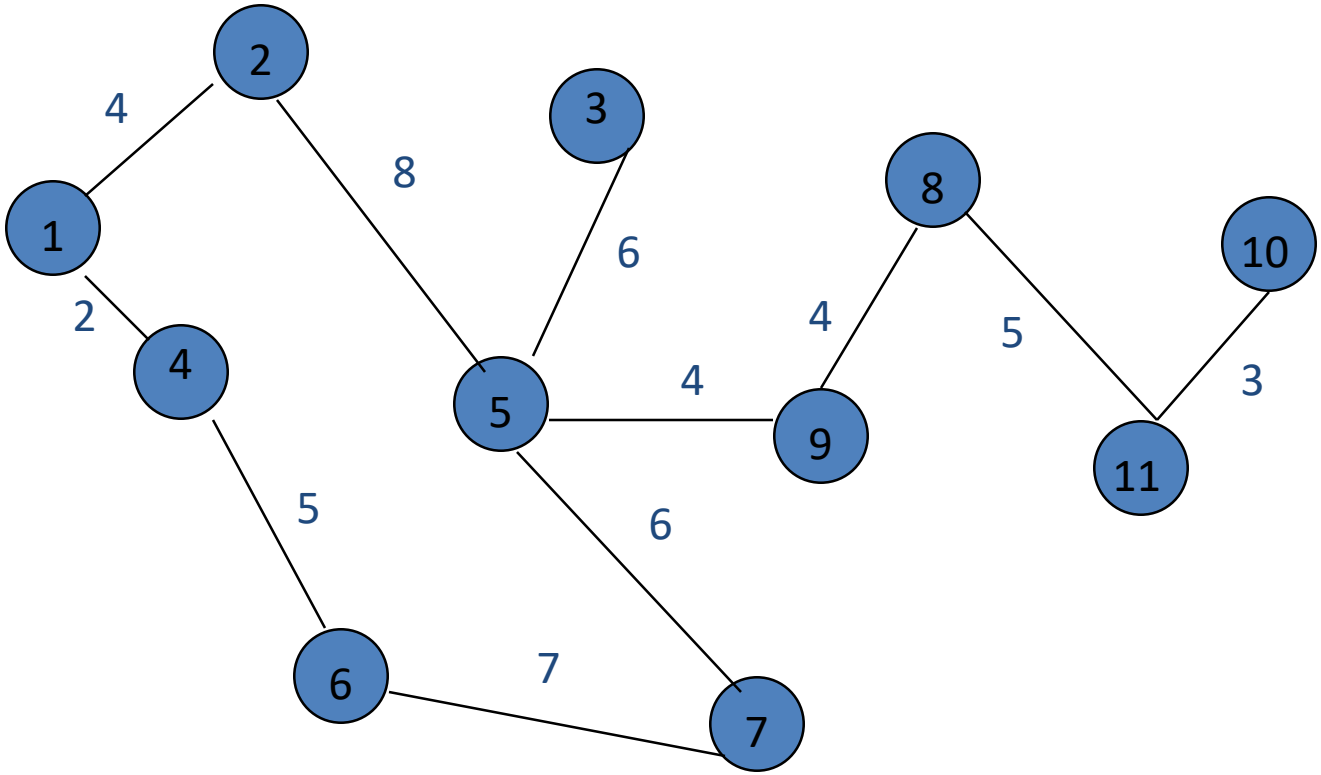
Continuation……

- **Science** – The molecular structure and chemical structure of a substance, the DNA structure of an organism, etc., are represented by graphs.

- **Linguistics** – The parsing tree of a language and grammar of a language uses graphs.

- **General** – Routes between the cities can be represented using graphs. Depicting hierarchical ordered information such as family tree can be used as a special type of graph called tree.

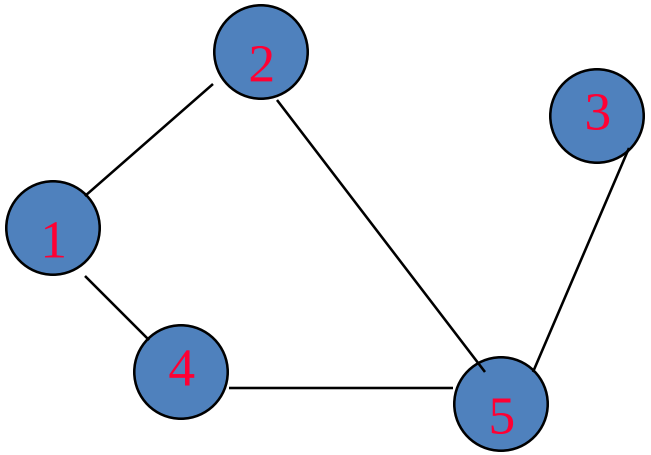
Driving Distance/Time Map



- Vertex = city, edge weight = driving distance/time.

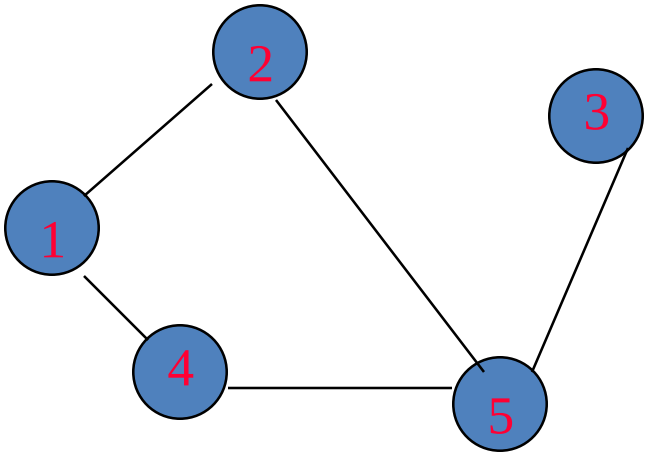
Undirected Graph Representation:

- 0/1.. n x n matrix, where n = # of vertices
- A(x,y) = 1 iff (x,y) is an edge, otherwise A(x,y)=0



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

Adjacency Matrix Properties



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

- Diagonal entries are zero.
- Adjacency matrix of an undirected graph is symmetric.
  - $A(i,j) = A(j,i)$ for all i and j.

Implementation of Graph by Two-dimensional Array

```
#include<iostream>

using namespace std;

#define CONNECTED 1
#define DISCONNECTED 0

int main()
{
   int nodes, edges, x,y;

   cout<<"How many nodes? "<<endl;
   cin>>nodes;

   //Declere the matrix for graph
   int Graph[nodes+1][nodes+1];
```

Continuation……

```
// initializing matrix by zero using no. of nodes
   int i,j;
   for( i=1;i<=nodes; i++)
      for( j=1; j<=nodes; j++)
      {
         Graph[i][j]= DISCONNECTED;
      }

   cout<<"How many edges? ";
   cin>>edges;

   // take connected edge info from user and initialize graph
   for(i=1; i<=edges; i++)
   {
      cout<<"Enter node x and y for edge "<<i<<endl;
      cin>>x>>y;
      Graph[x][y]=Graph[y][x]= CONNECTED;
   }
```

Continuation….

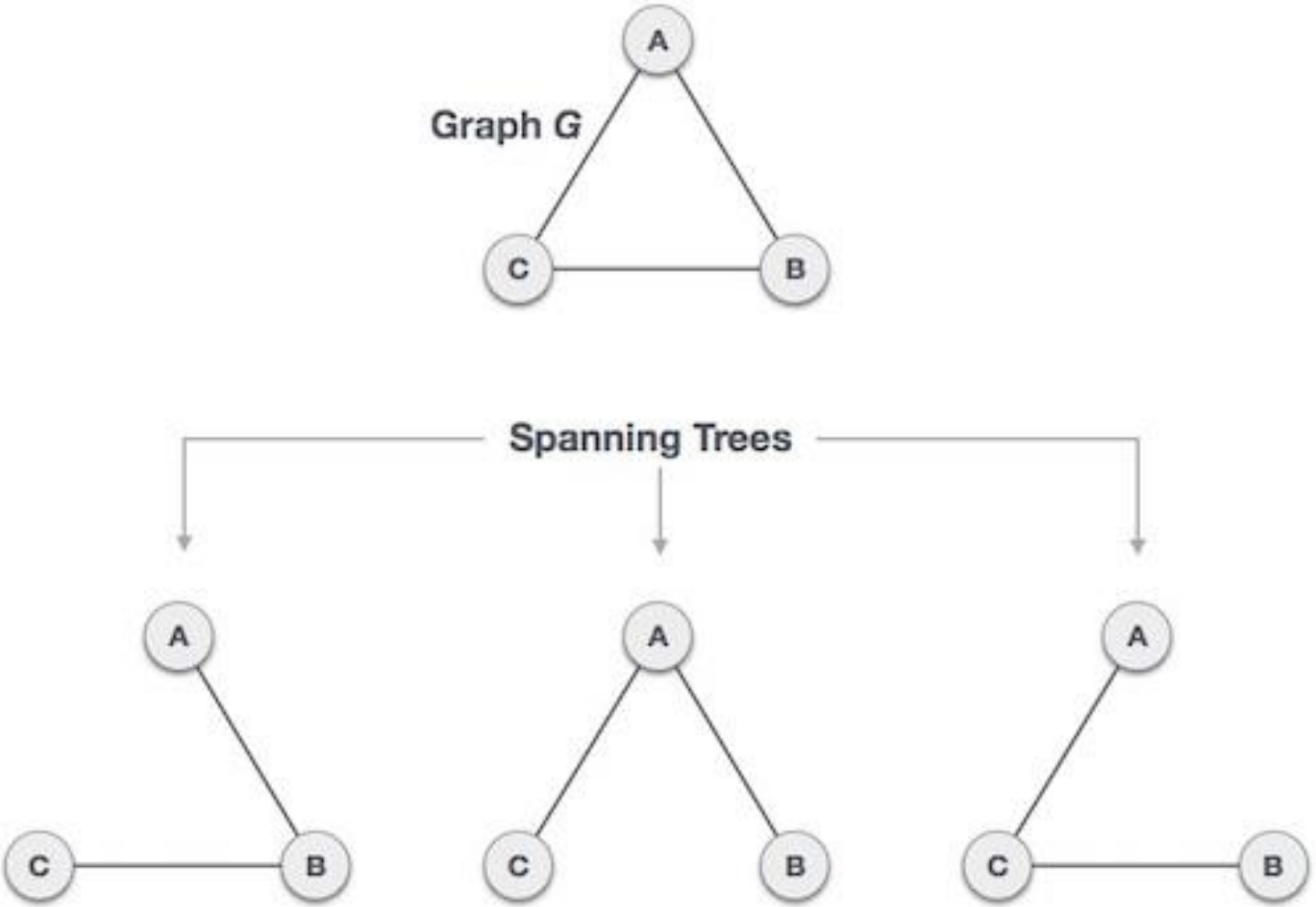
```
//Print the adjacent matrix
   cout<<"Adjacent Matrix: "<<endl;
   for( i=1;i<=nodes; i++)
   {
      for( j=1; j<=nodes; j++)
      {
         cout<<Graph[i][j]<<" ";
      }
      cout<<endl;
   }
return 0;

}
```

Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.

- Hence, a spanning tree does not have cycles and it can not be disconnected.

- Subgraph is a tree.
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.



A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is number of nodes. In addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General properties of spanning tree

Below are few properties of spanning tree of given connected graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have same number of edges and vertices.
- Spanning tree does not have any cycle (loops)
- Removing one edge from spanning tree will make the graph disconnected i.e. spanning tree is minimally connected.
- Adding one edge to a spanning tree will create a circuit or loop i.e. spanning tree is maximally acyclic.

Mathematical properties of spanning tree

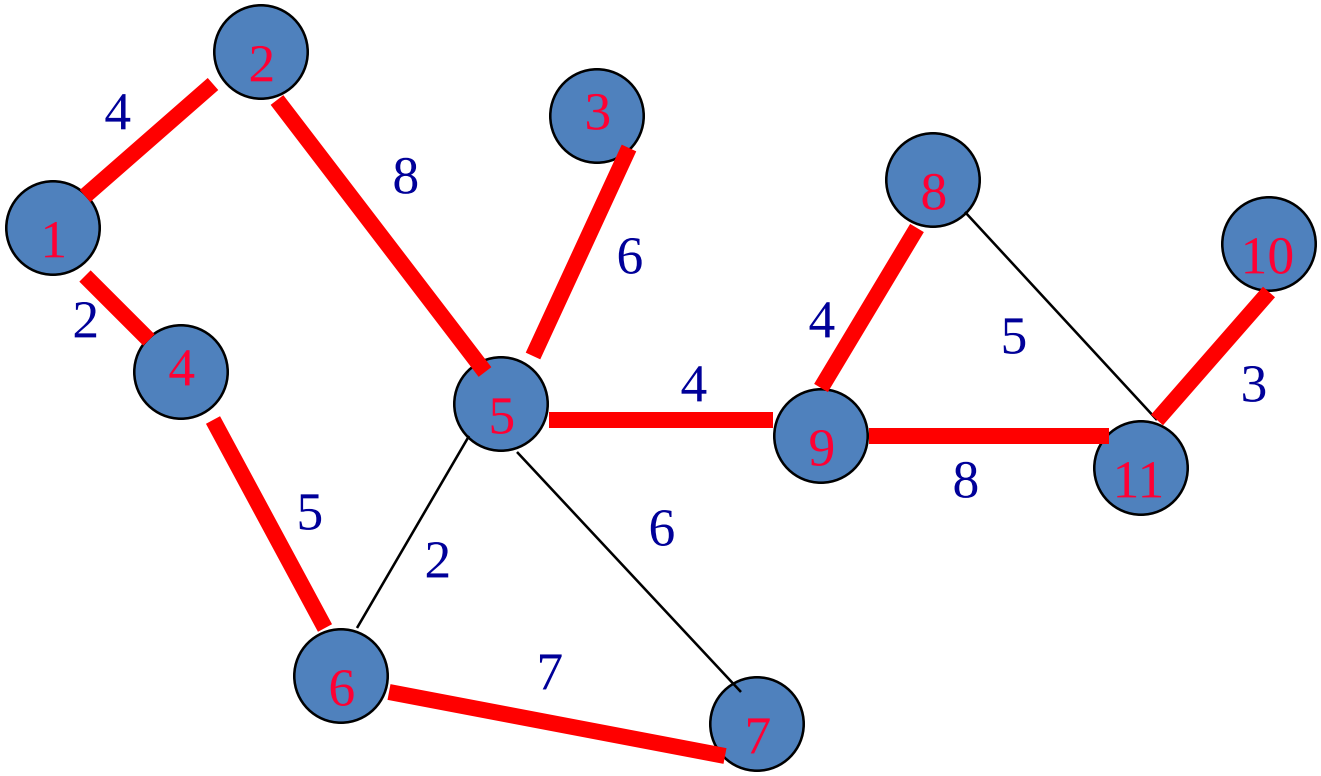
- Spanning tree has n-1 edges, where n is number of nodes (vertices)
- From a complete graph, by removing maximum e-n+1 edges, we can construct a spanning tree.
- A complete graph can have maximum nn-2 number of spanning trees.

Application of Spanning Tree

Spanning tree is basically used to find minimum paths to connect all nodes in a graph. Common application of spanning trees are −

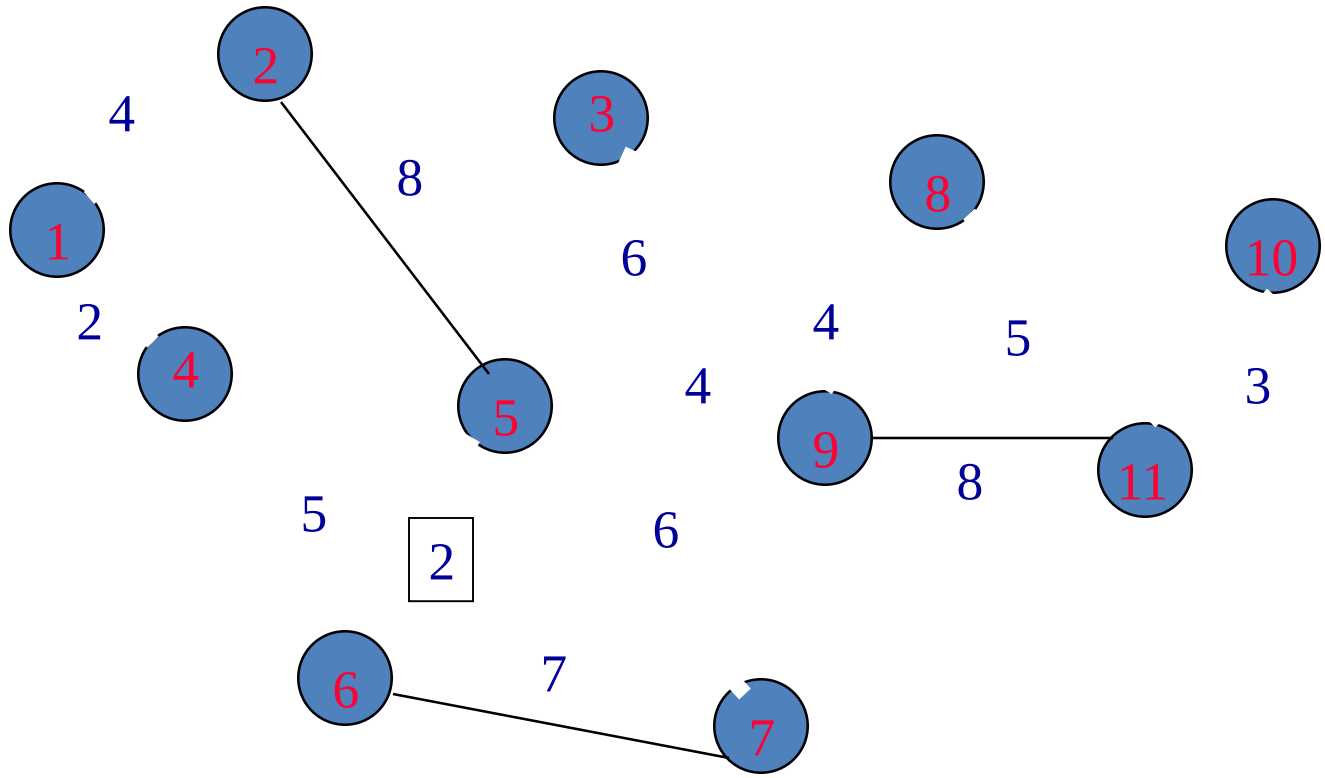
- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

A Spanning Tree



Spanning tree cost = 51.

Minimum Cost Spanning Tree



Spanning tree cost = 41.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms −

- Kruskal's Algorithm
- Prim's Algorithm

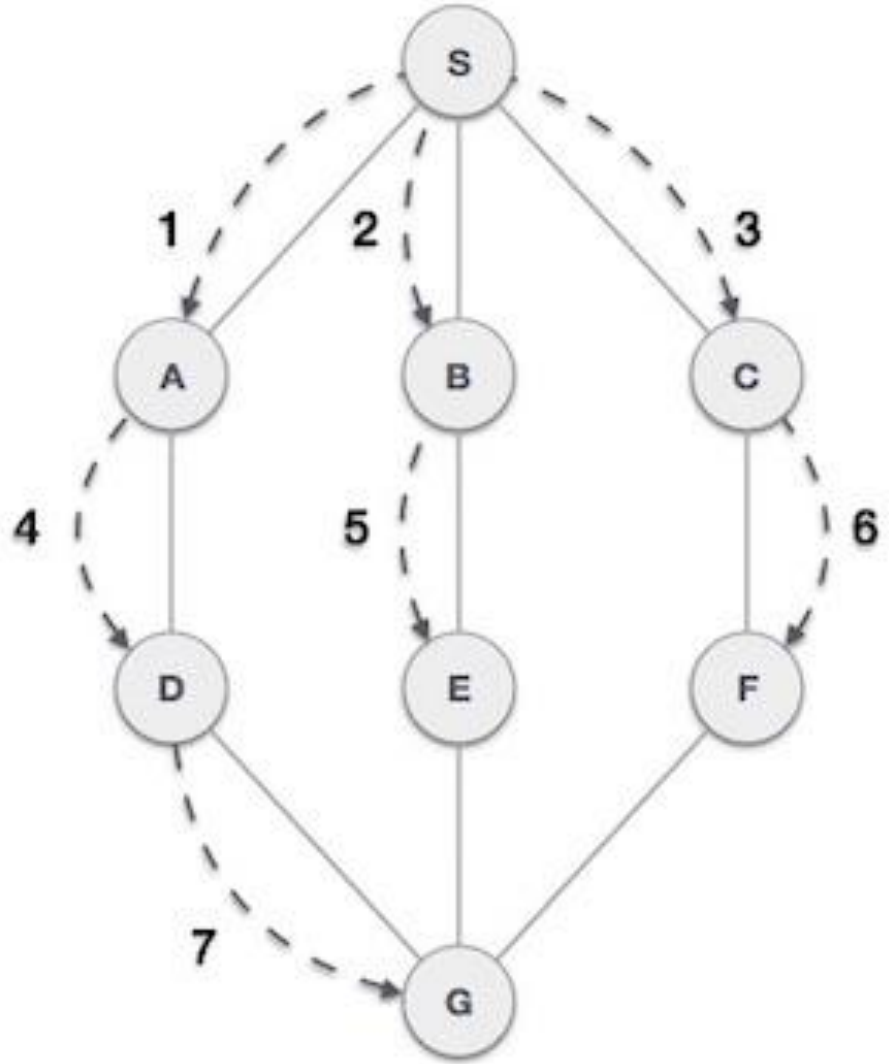
Both are greedy algorithms.

Graph Search Methods

- Many graph problems solved using a search method.
  - Path from one vertex to another.
  - Is the graph connected?
  - Find a spanning tree.
  - Etc.
- Commonly used search methods:
  - Breadth-first search (BFS).
  - Depth-first search (DFS).

Breadth First Traversal (BFS)

Breadth First Search algorithm(BFS) traverses a graph in a breadth wards 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 example given above, BFS algorithm traverses from A to B to C to D first then to E and F lastly to G.

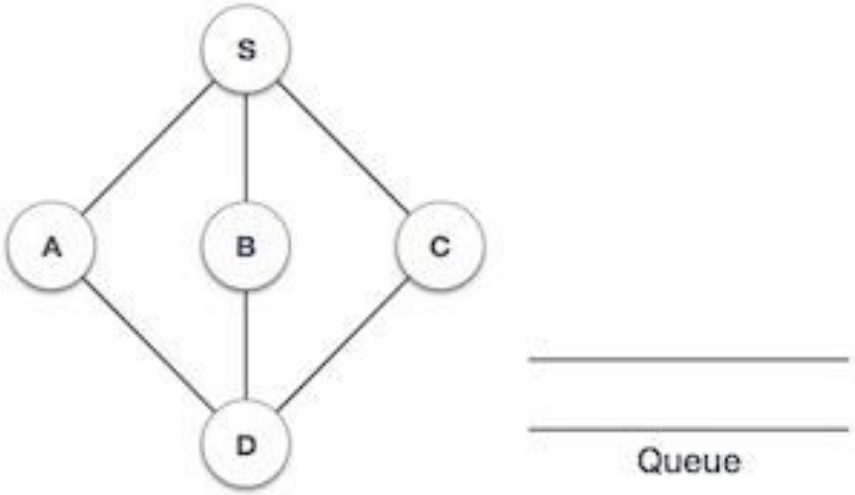
BFS Algorithm

- **Rule 1** − Visit adjacent unvisited node. Mark the node and Insert it in a queue.

- **Rule 2** − If no adjacent node found unvisited, remove the first node from the queue.

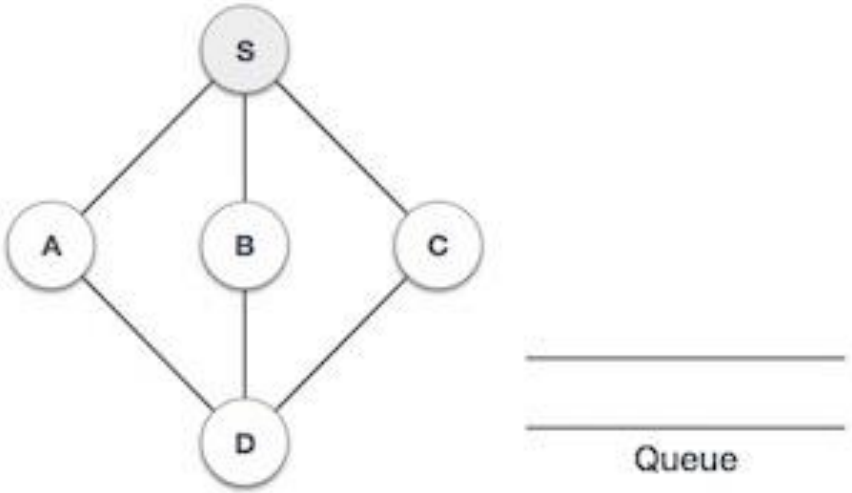
- **Rule 3** − Repeat Rule 1 and Rule 2 until queue is empty.

1.



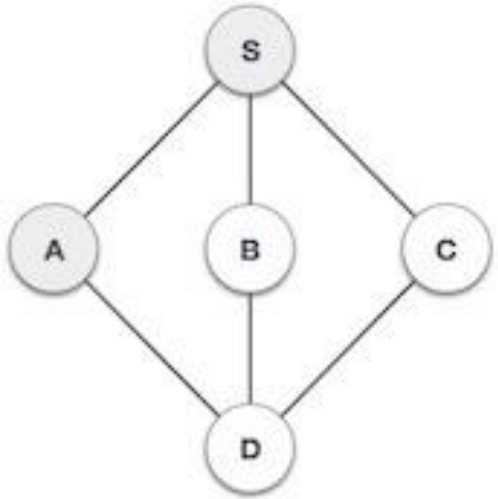
Initialize the queue.

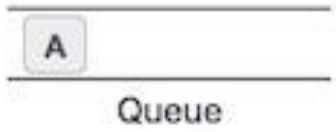
2.



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

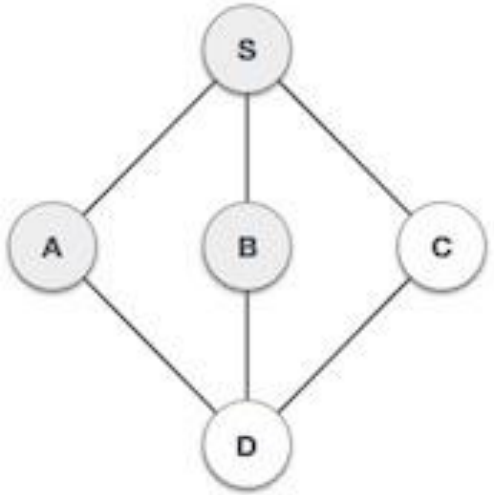
3.

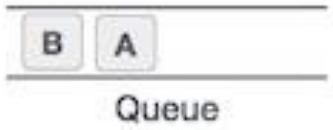




We then see unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A** mark it visited and enqueue it.

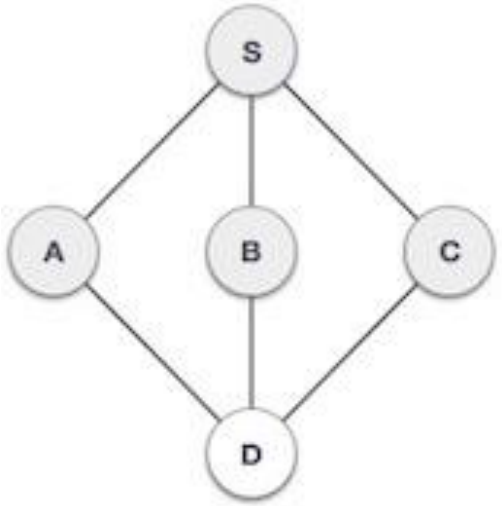
4.

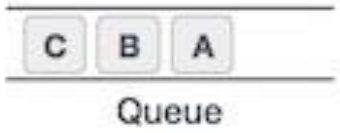




Next unvisited adjacent node from **S** is **B**. We mark it visited and enqueue it.

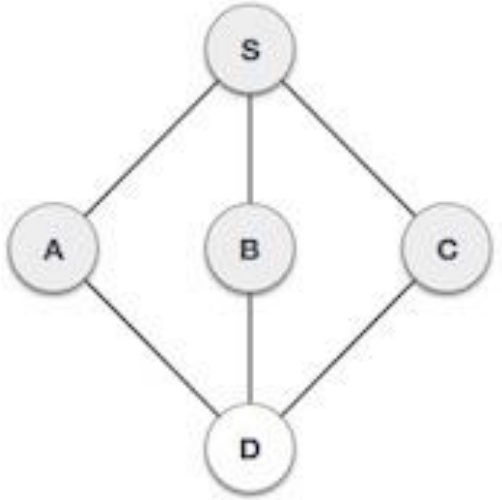
5.

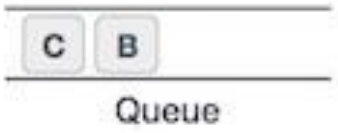




Next unvisited adjacent node from **S** is **C**. We mark it visited and enqueue it.

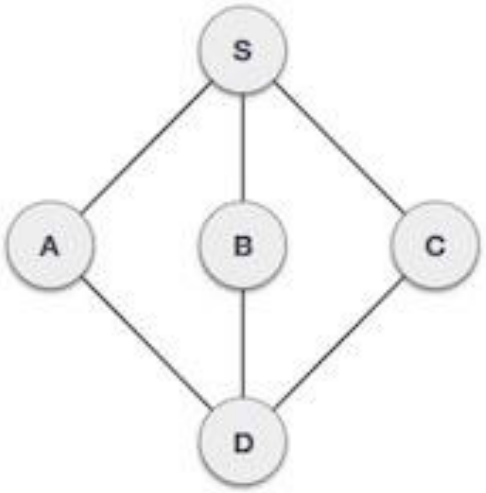
6.

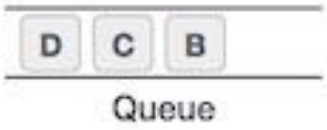




Now **S** is left with no unvisited adjacent nodes. So we dequeue and find **A**.

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.

The implementation of BFS algorithm in C++

```
#include<iostream>
using namespace std;

#define unvisited -1
#define marked 0
#define visited 1
#define CONNECTED 1
#define DISCONNECTED 0
#define SIZE 100

int graph[SIZE+1][SIZE+1], label[SIZE+1];

int queue[SIZE], head, tail;

int nodes, edges;
```

```
void initialize_queue()
{
	head = tail = 0;
}

void enqueue(int node)
{
	queue[tail++] = node;
}

int dequeue()
{
	return queue[head++];
}

bool queue_empty()
{
	return head == tail;
}

void BFS(int source, int nodes);
```

The implementation of BFS algorithm in C++ (continuation...)

```
int main()
{
	int x, y, source;
	cout<<"No. of nodes: ";
	cin>>nodes;
	for(int i=1; i<=nodes; i++)
		for(int j=1; j<=nodes; j++)
			graph[i][j] = DISCONNECTED;

	cout<<"No. of edges: ";
	cin>>edges;
	for(int i=1; i<=edges; i++)
	{
		cout<<"Enter vertes x and y for edge: "<< i;
		cin>>x >>y;
		graph[x][y] = graph[y][x] = CONNECTED;
	}
	initialize_queue();   // initialize queue before using
	cout<<"Give source:" ;  cin>>source;
	BFS(source, nodes); // call bfs for graph
		return 0;
}
```

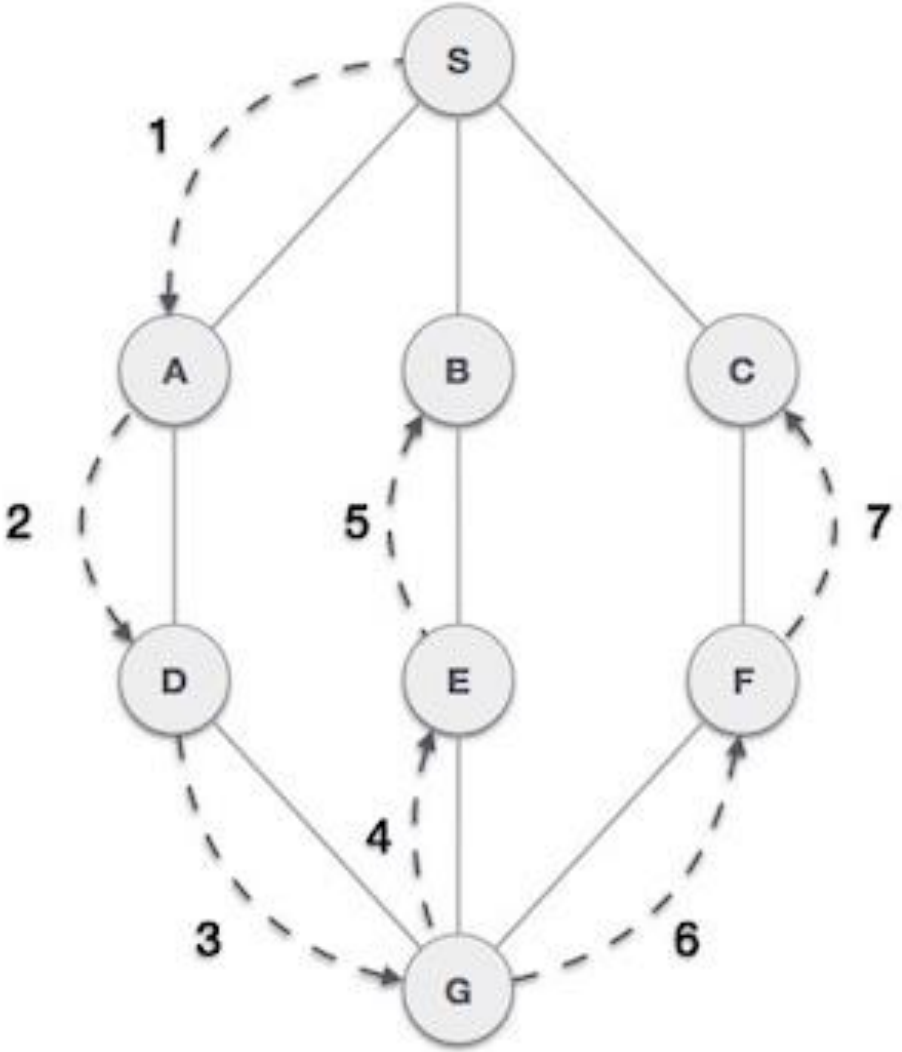
// initializing graph with 0 using no. of nodes

// take edge info from user and initialize graph

```
void BFS(int source, int nodes)
{
    for(int i=1; i<=nodes; i++)
        label[i] = unvisited;          // initially mark all nodes as undiscovered
    enqueue(source);
    label[source] = marked;            // inserting source into the queue and modifying relevant values
    while(!queue_empty())  // repeat until queue is empty
    {
        int vn = dequeue();
        cout<< vn;<< " ";   // print nodes in visiting sequence i.e. spanning tree
        for(int v=1; v<=nodes; v++)    // check all nodes for neighbour
        {
            if(graph[vn][v] == CONNECTED)     // check connected node
            {
                if(label[v] == unvisited)   // check if node is undiscovered
                {
                    label[v] = marked;   // mark node as discovered
                    enqueue(v);                  // insert neighbour into the queue}
            }
        }
        label[vn] = visited;  // mark node as finished
    }
    cout<<endl;
}
```

Depth First Traversal (DFS)

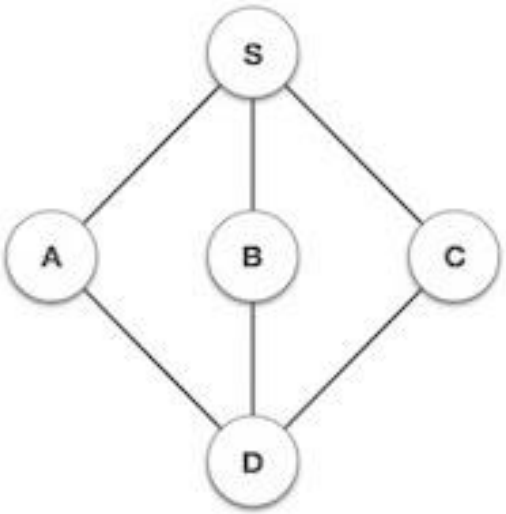
Depth First Search algorithm(DFS) traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.



DFS Algorithm

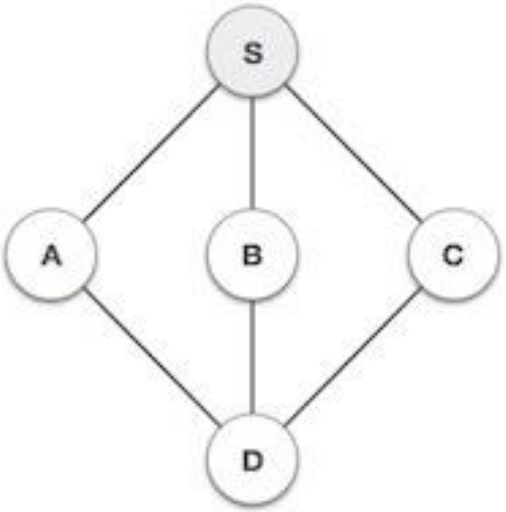
- **Rule 1 – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.**
- **Rule 2 – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)**
- **Rule 3 – Repeat Rule 1 and Rule 2 until stack is empty.**

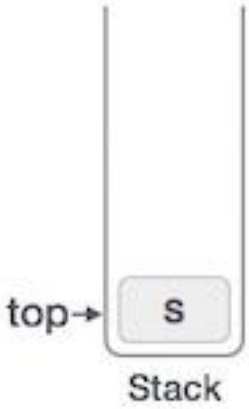
1.



Initialize the stack

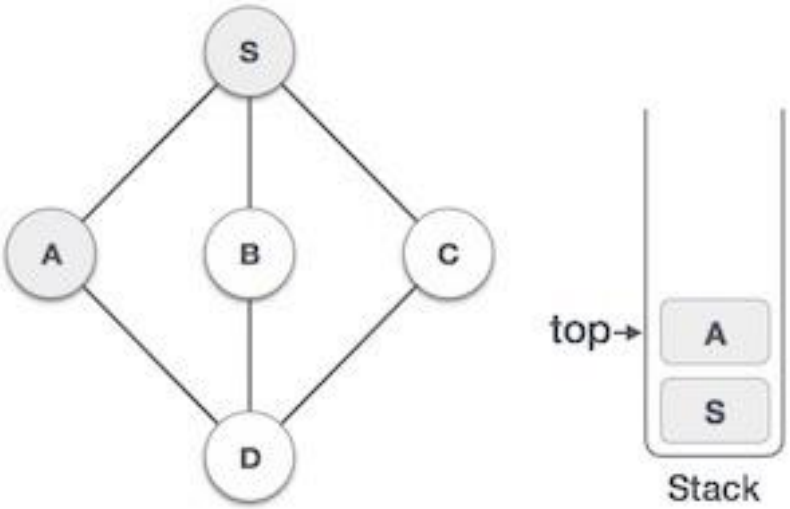
2.





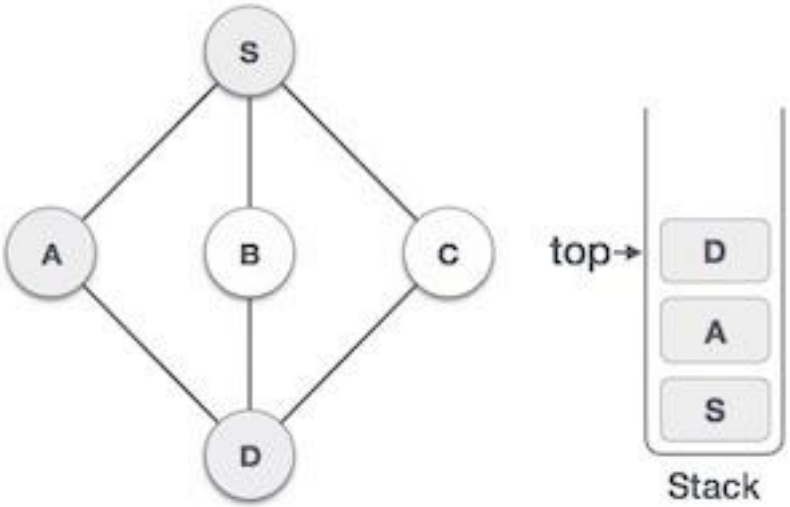
Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.

3.



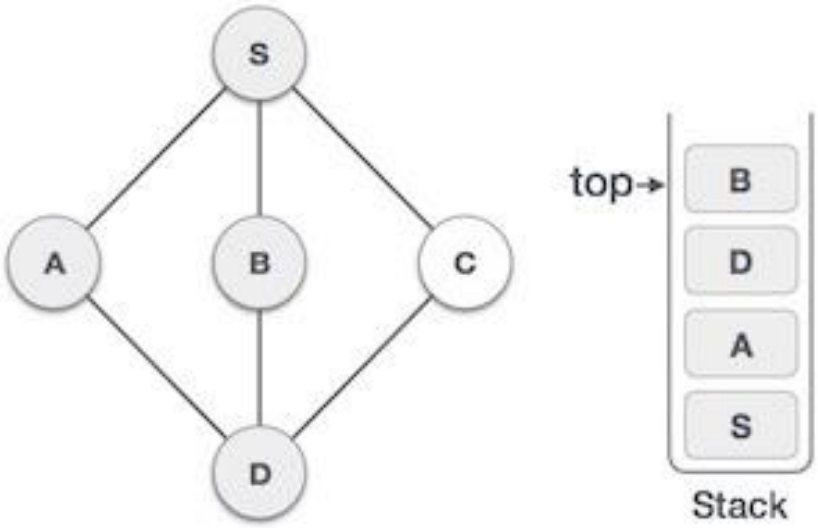
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A.
Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4.



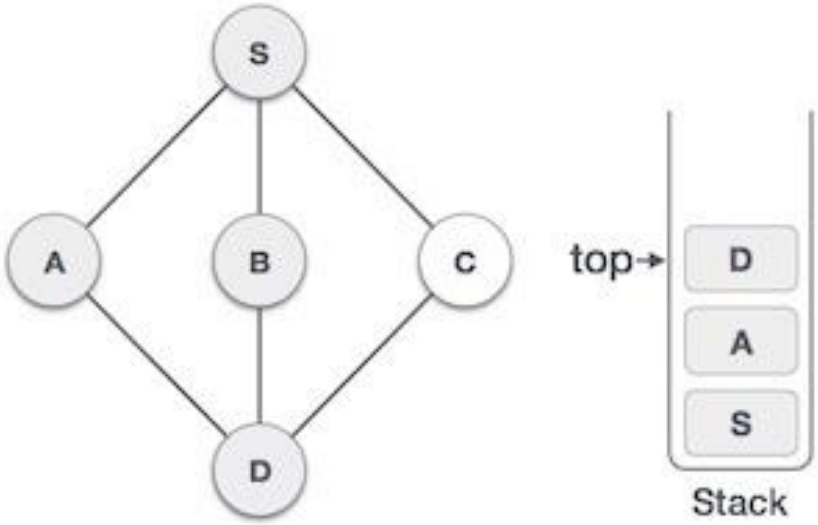
Visit **D** and mark it visited and put onto the stack. Here we have **B** and **C** nodes which are adjacent to **D** and both are unvisited. But we shall again choose in alphabetical order.

5.



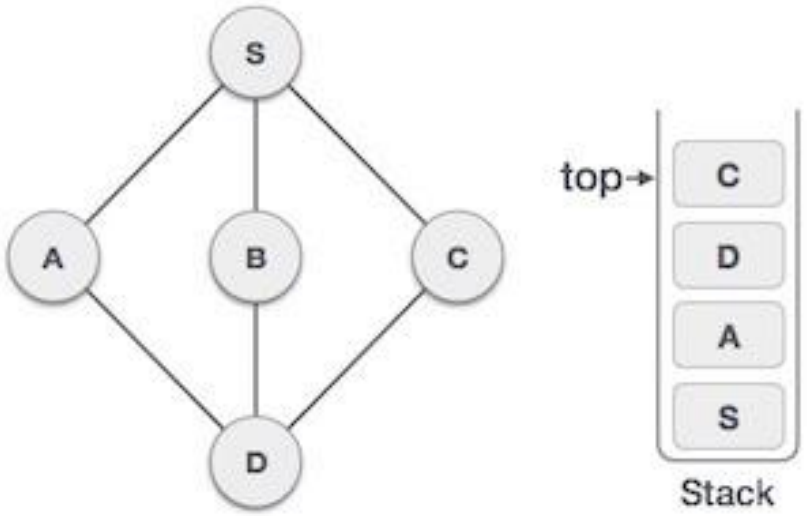
We choose **B**, mark it visited and put onto stack. Here **B** does not have any unvisited adjacent node. So we pop **B** from the stack.

6.



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.

7.



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

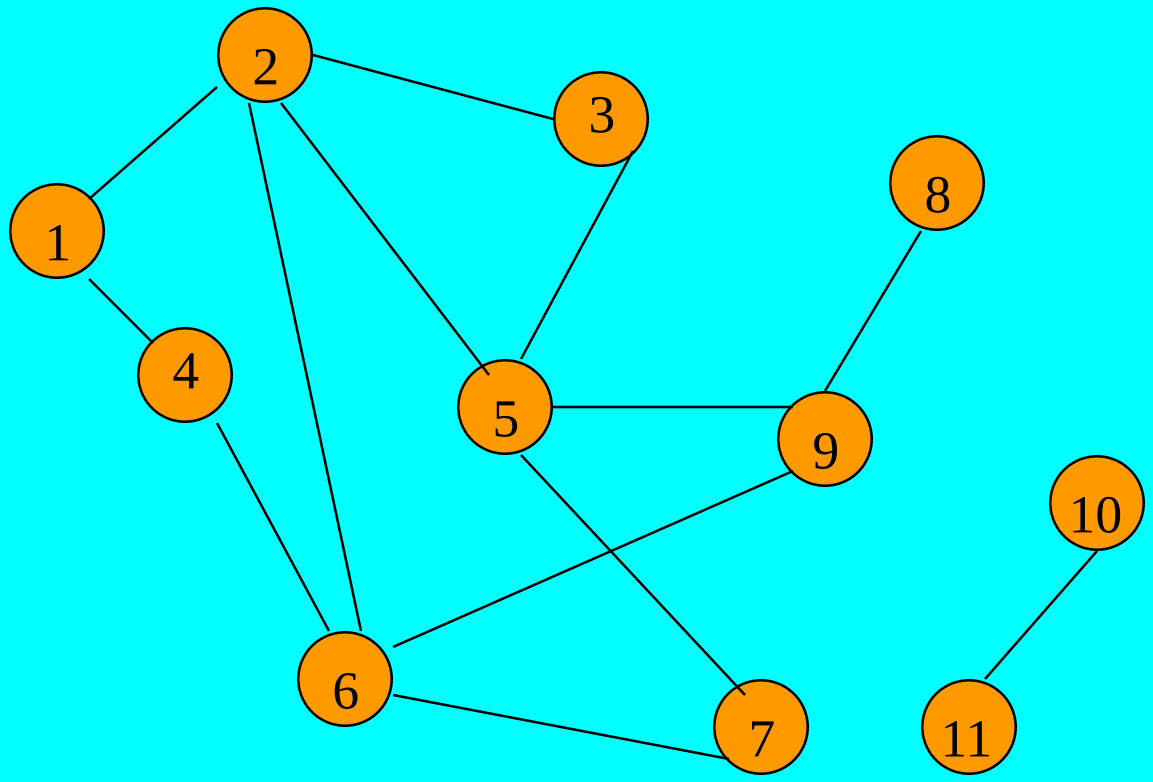
As C does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

More Example for BFS & DFS

Breadth-First Search (BFS)

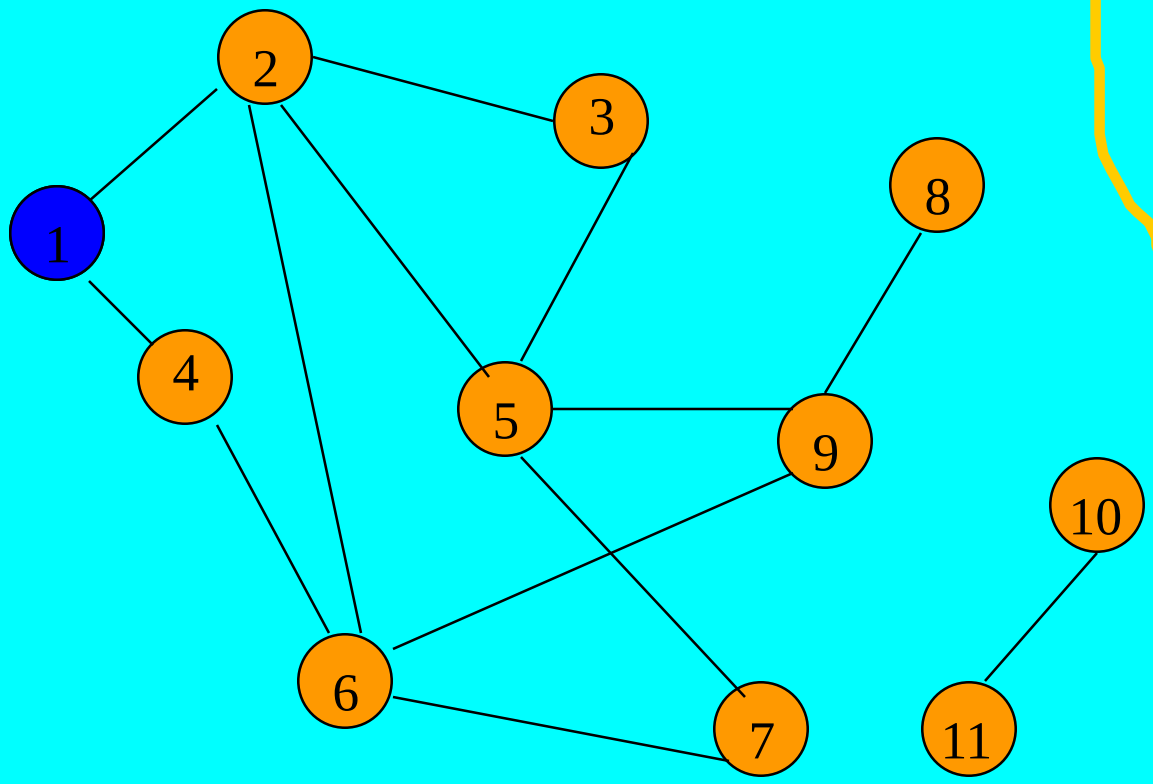
- Visit
  - start vertex and put into a FIFO queue.
- Repeatedly
  - remove a vertex from the queue,
  - visit its unvisited adjacent vertices,
  - put newly visited vertices into the queue.

Breadth-First Search Example



Start search at vertex 1.

Breadth-First Search Example

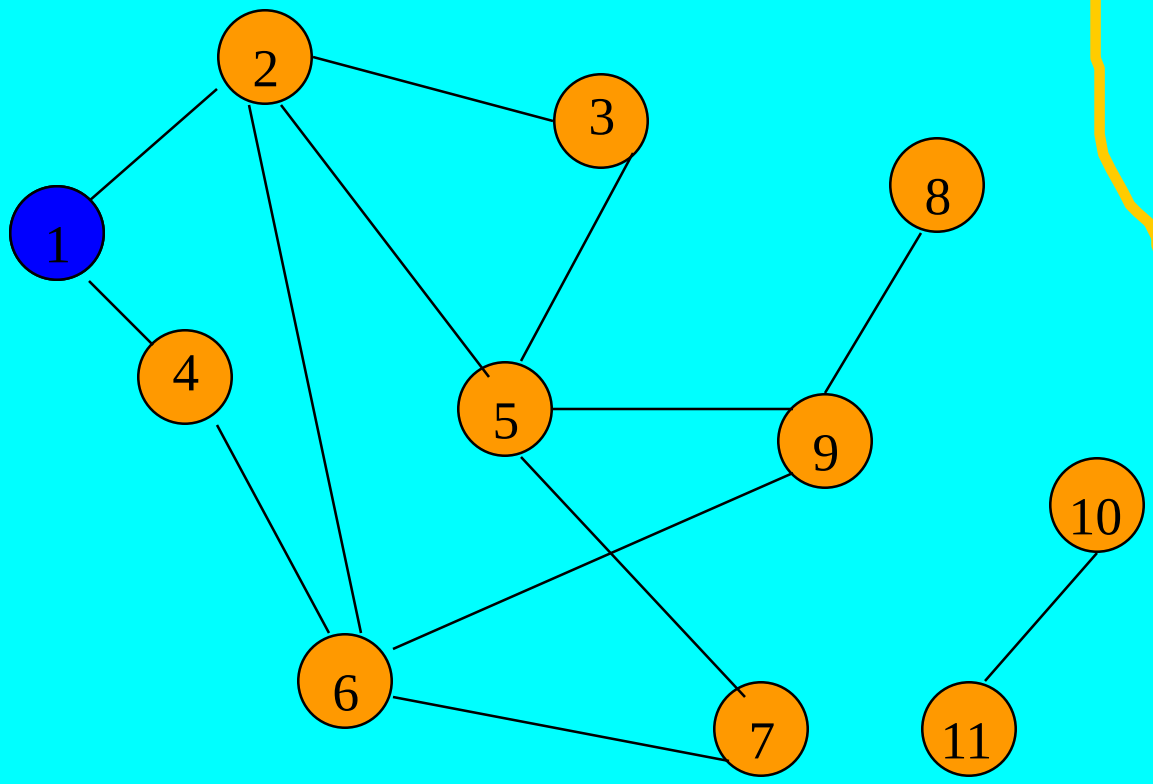


FIFO Queue

1

Visit/mark/label start vertex and put in a FIFO queue.

Breadth-First Search Example

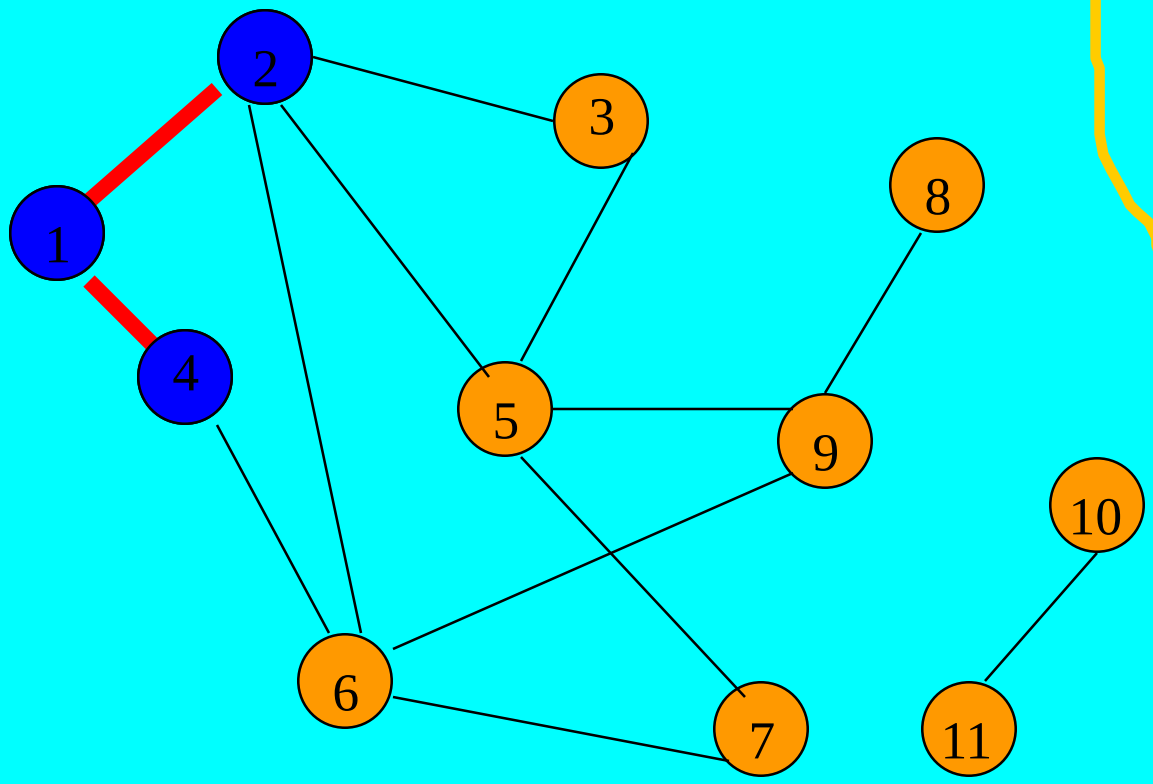


FIFO Queue

1

Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

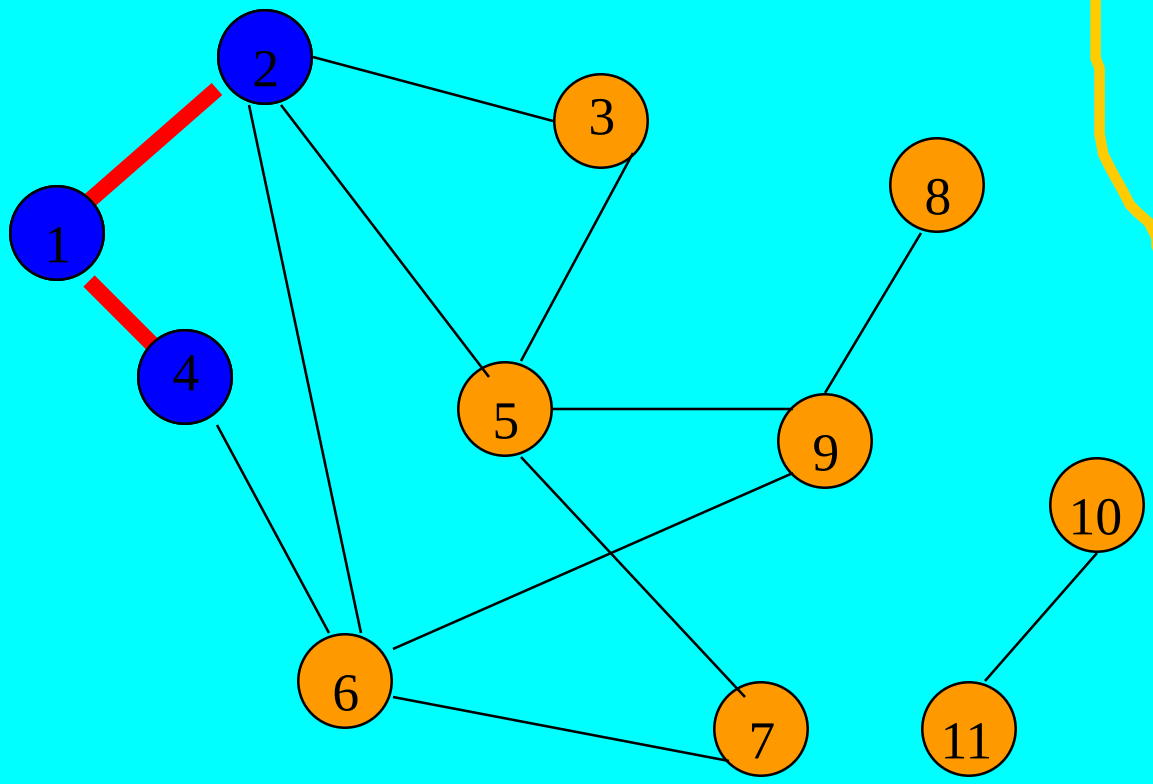


FIFO Queue

2 4

Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

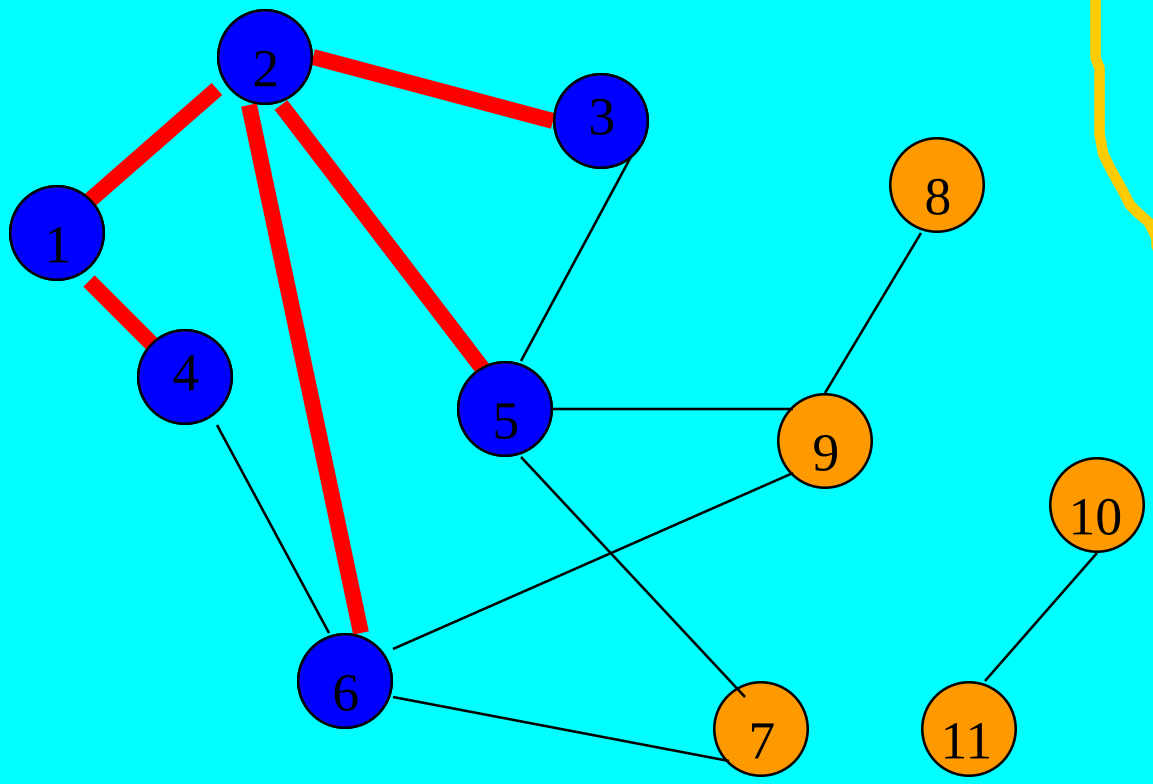


FIFO Queue

2 4

Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

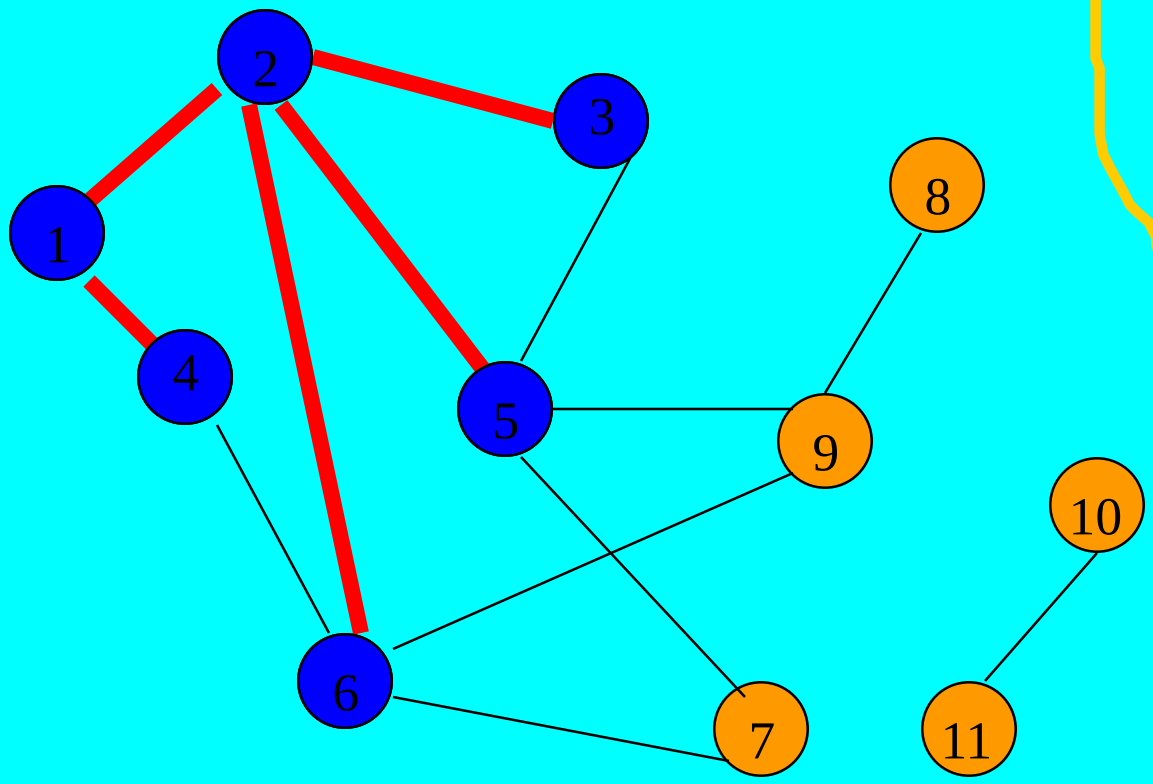


FIFO Queue

4 5 3 6

Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

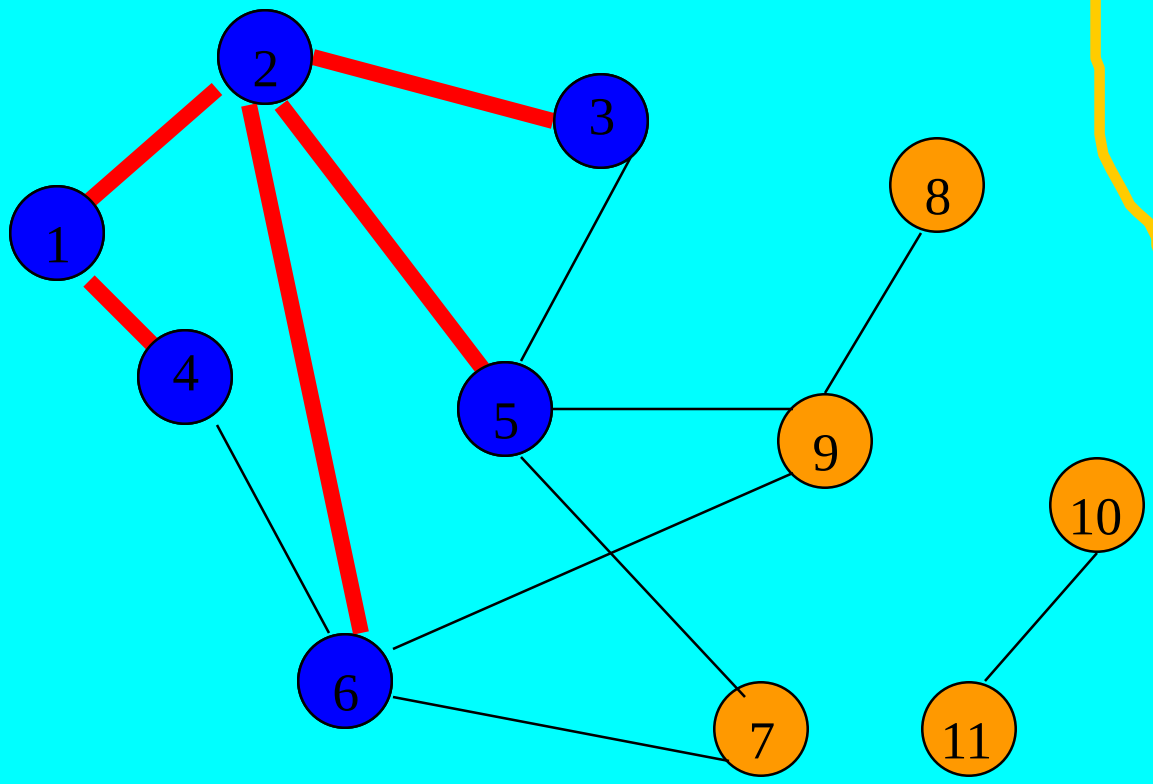


FIFO Queue

4 5 3 6

Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

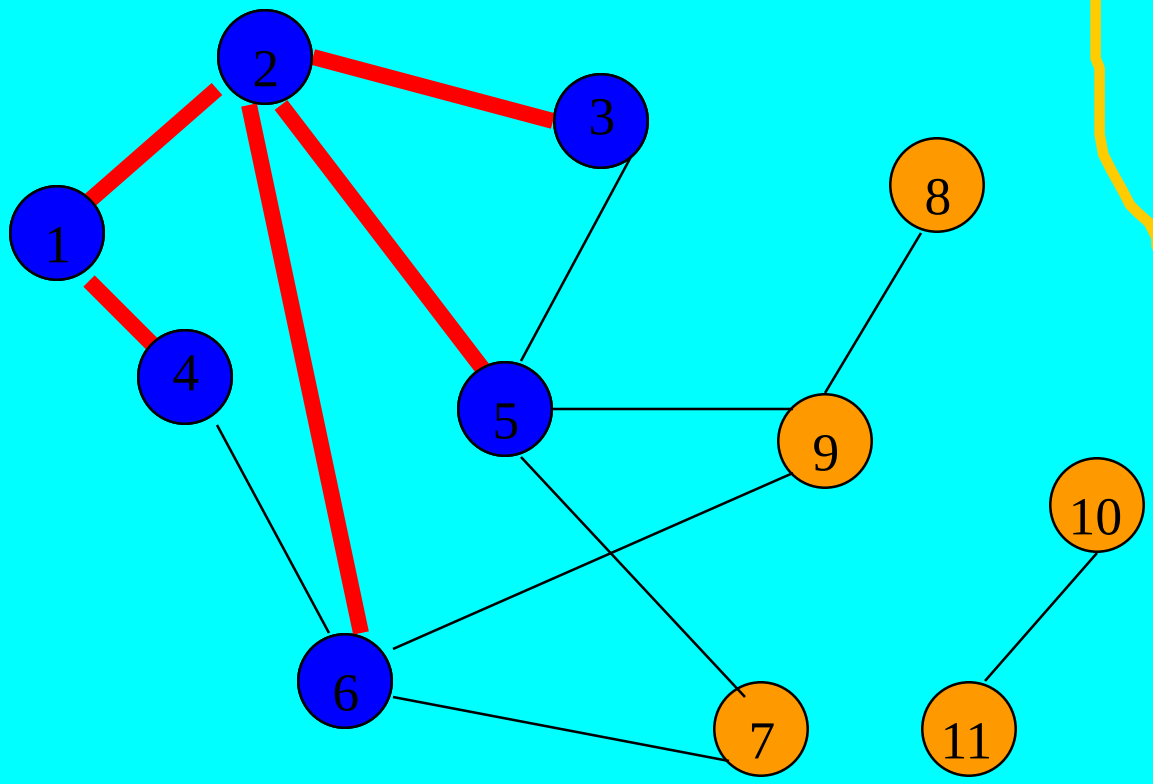


FIFO Queue

5 3 6

Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

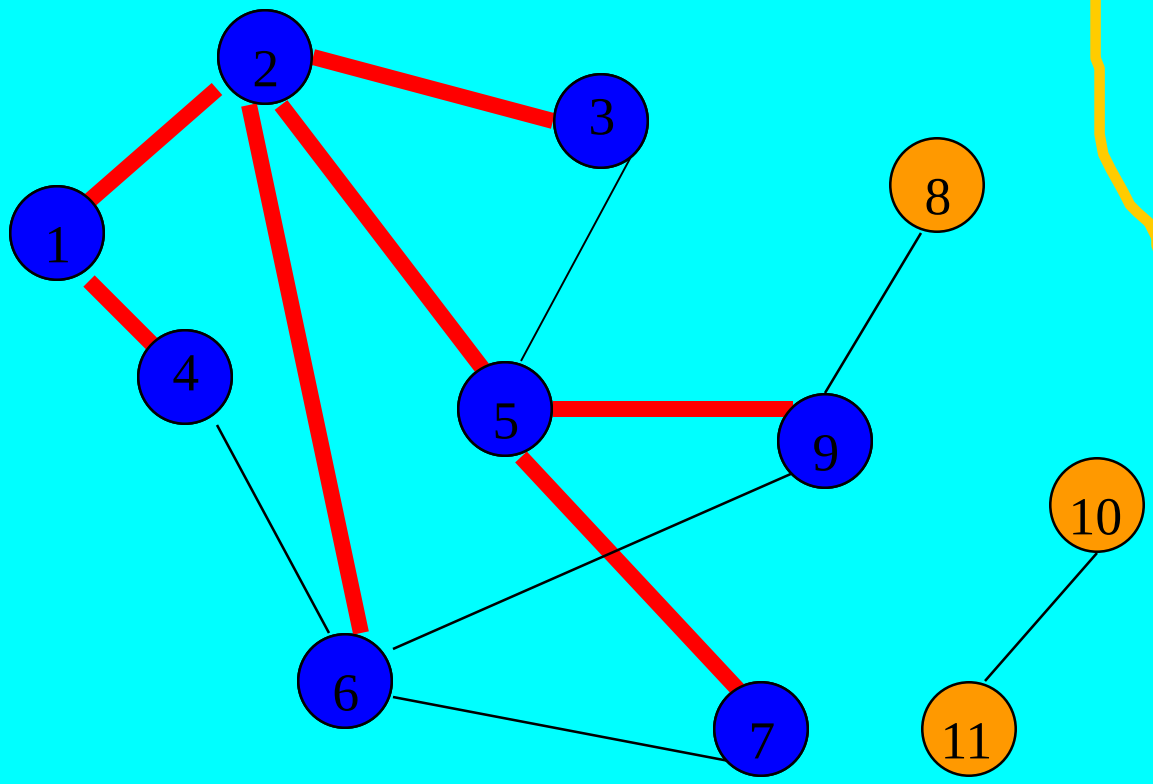


FIFO Queue

5 3 6

Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

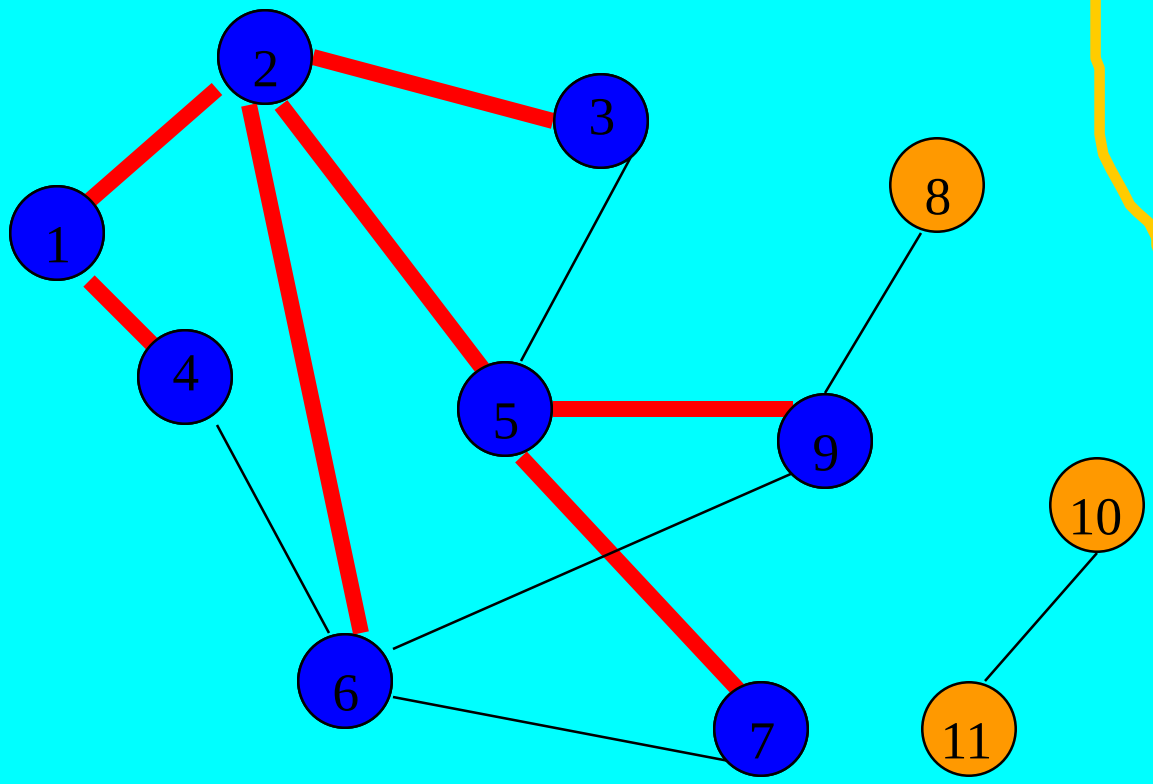


FIFO Queue

3 6 9 7

Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

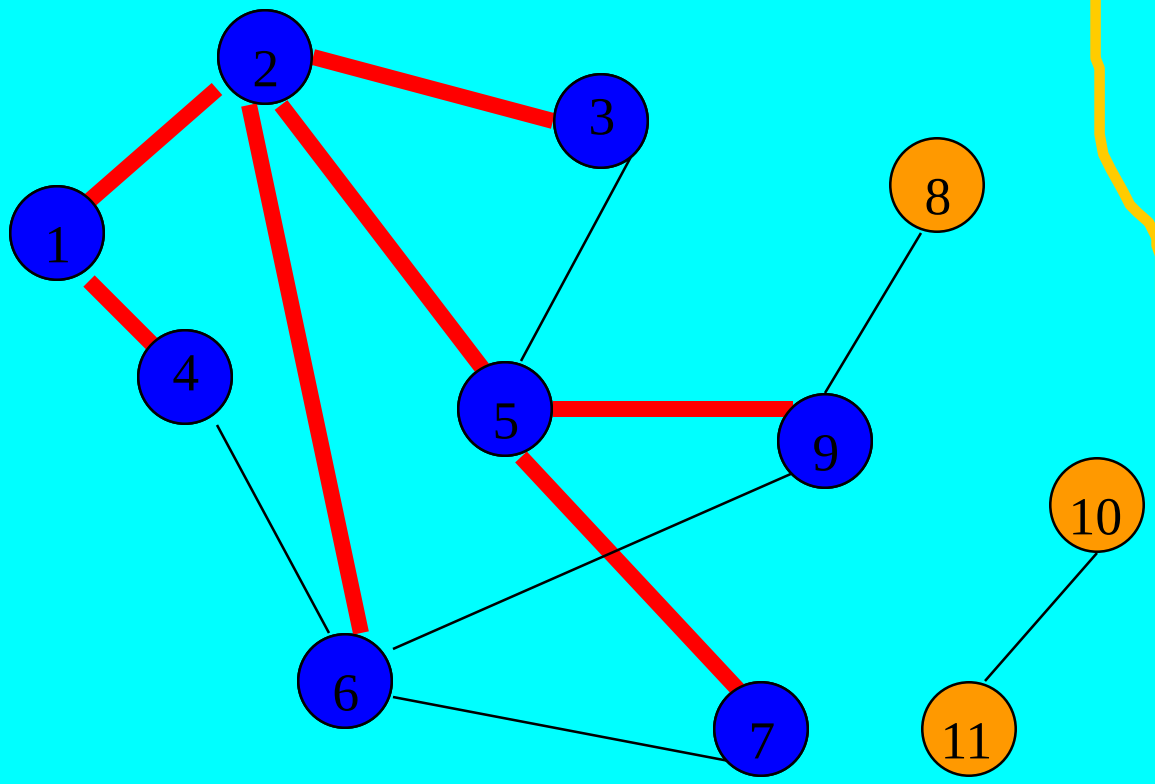


FIFO Queue

3 6 9 7

Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

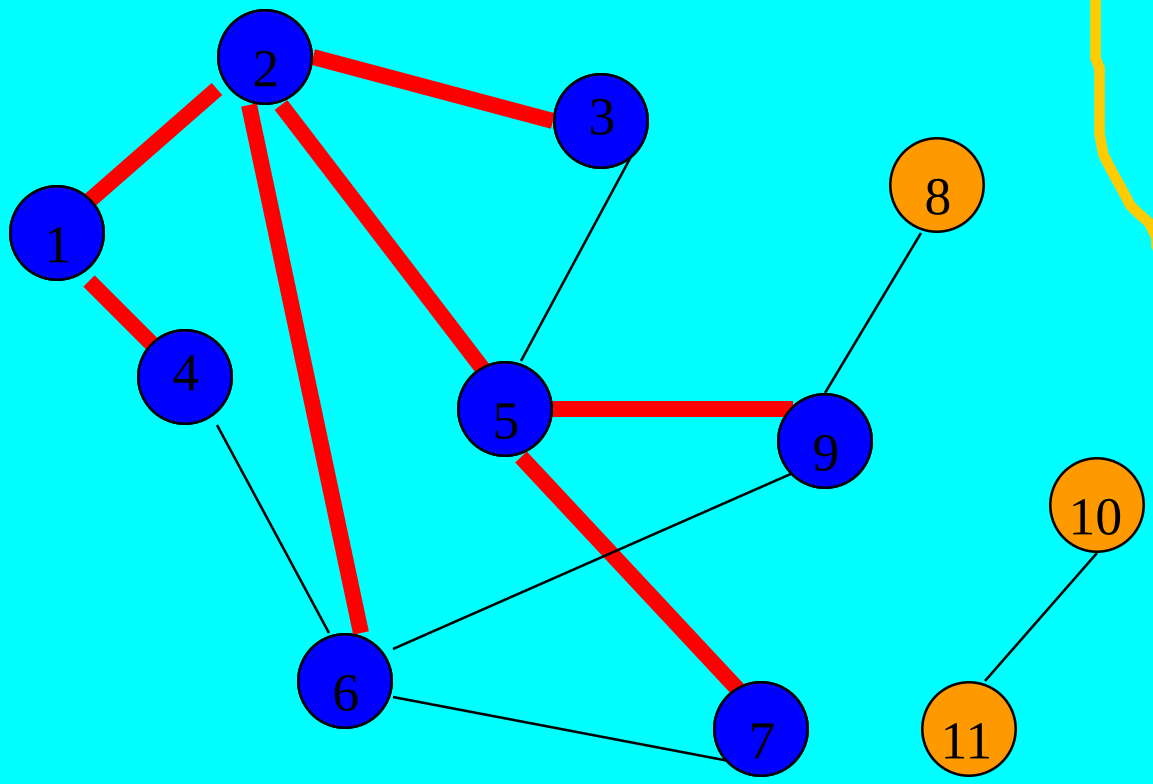


FIFO Queue

6 9 7

Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

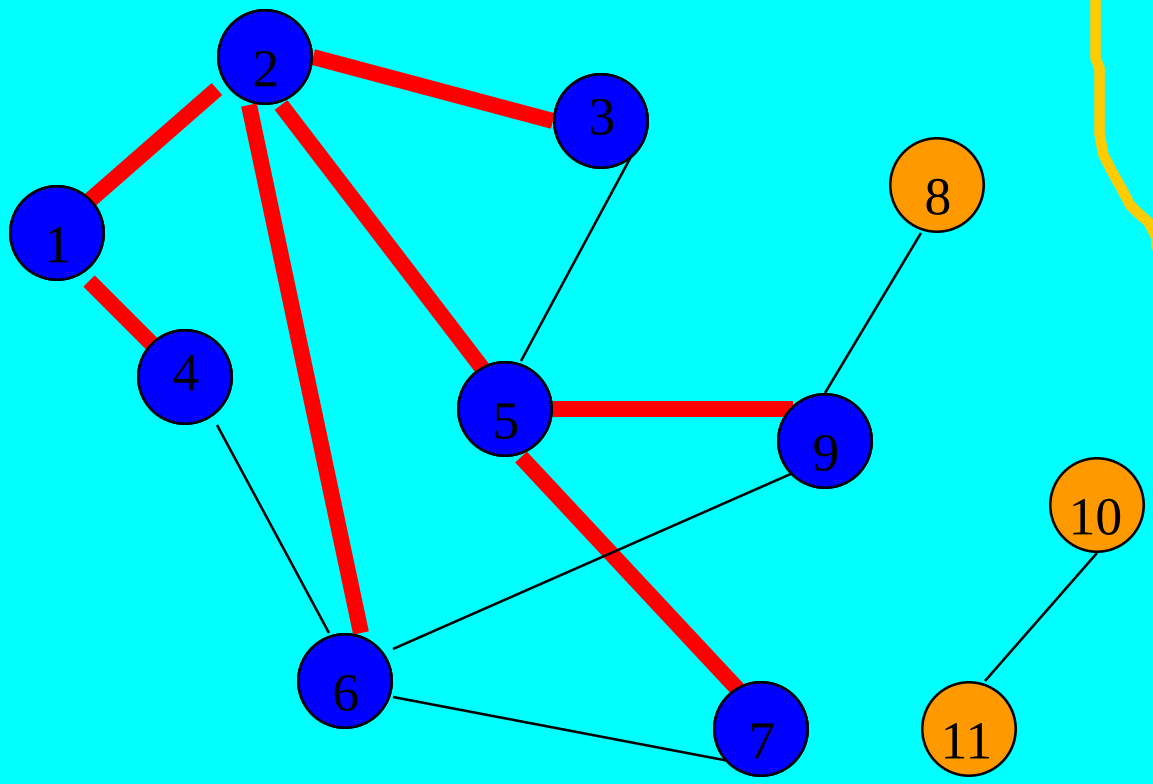


FIFO Queue

6 9 7

Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

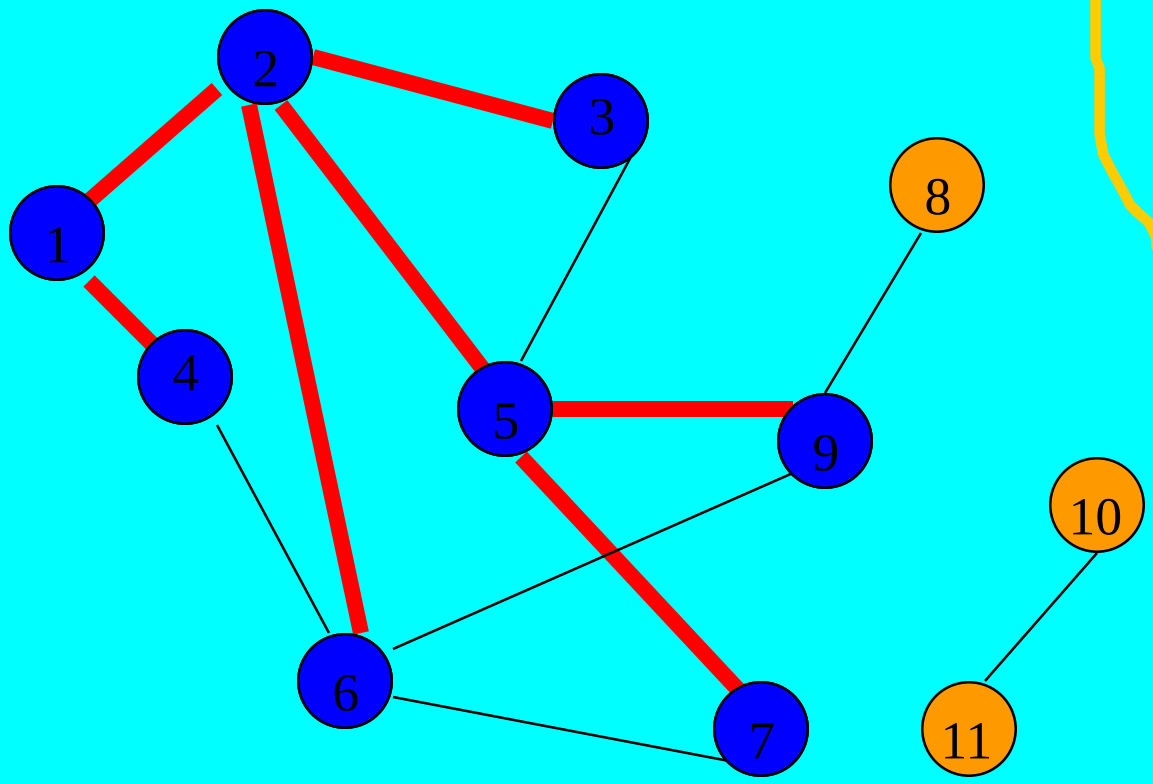


FIFO Queue

9 7

Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

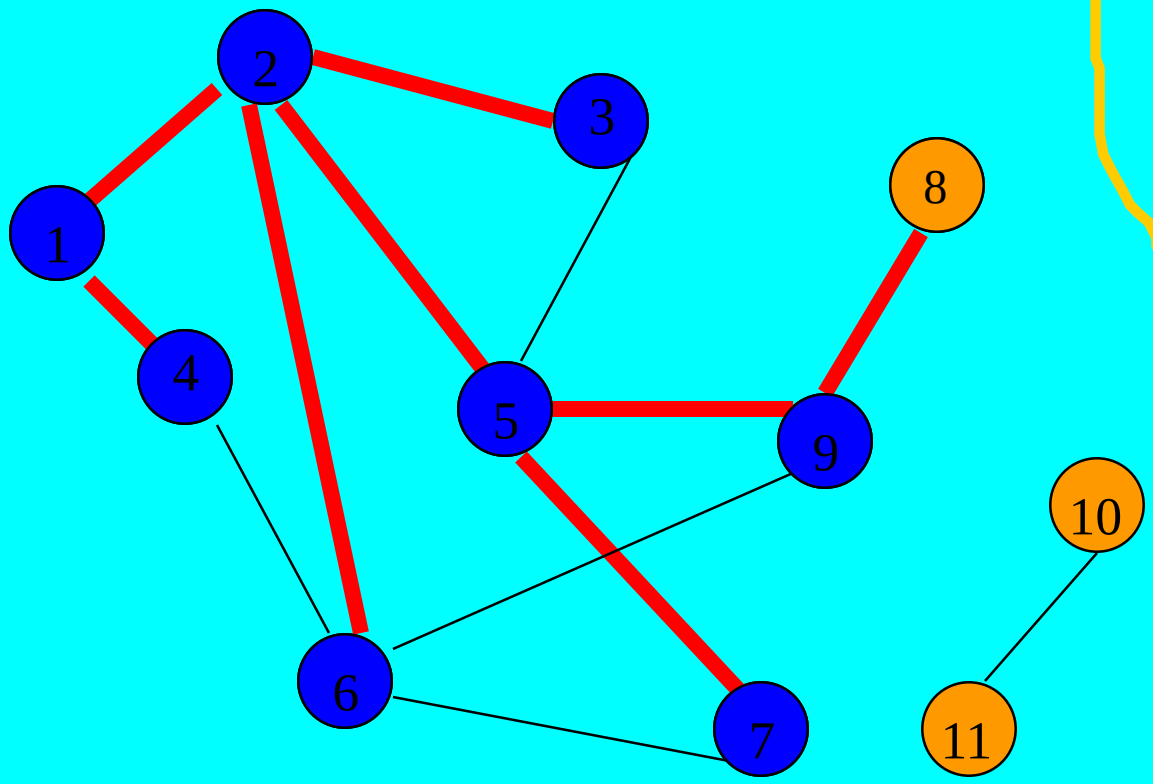


FIFO Queue

9 7

Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

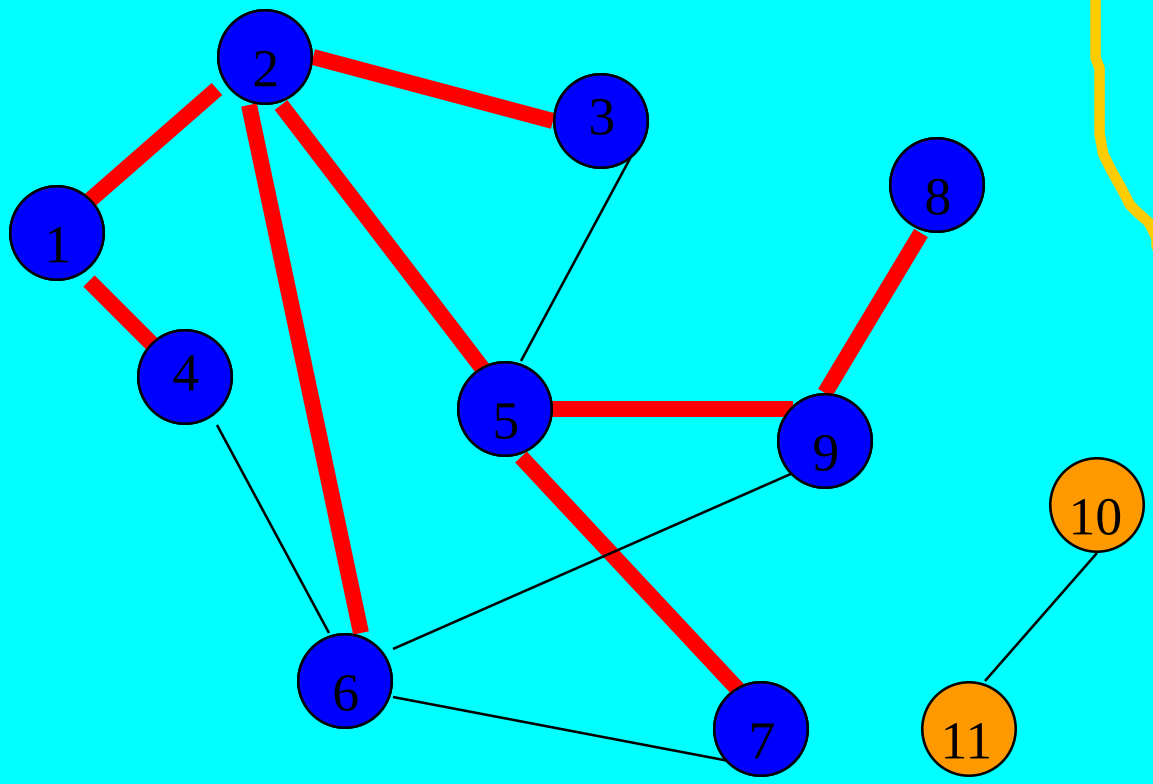


FIFO Queue

7

Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

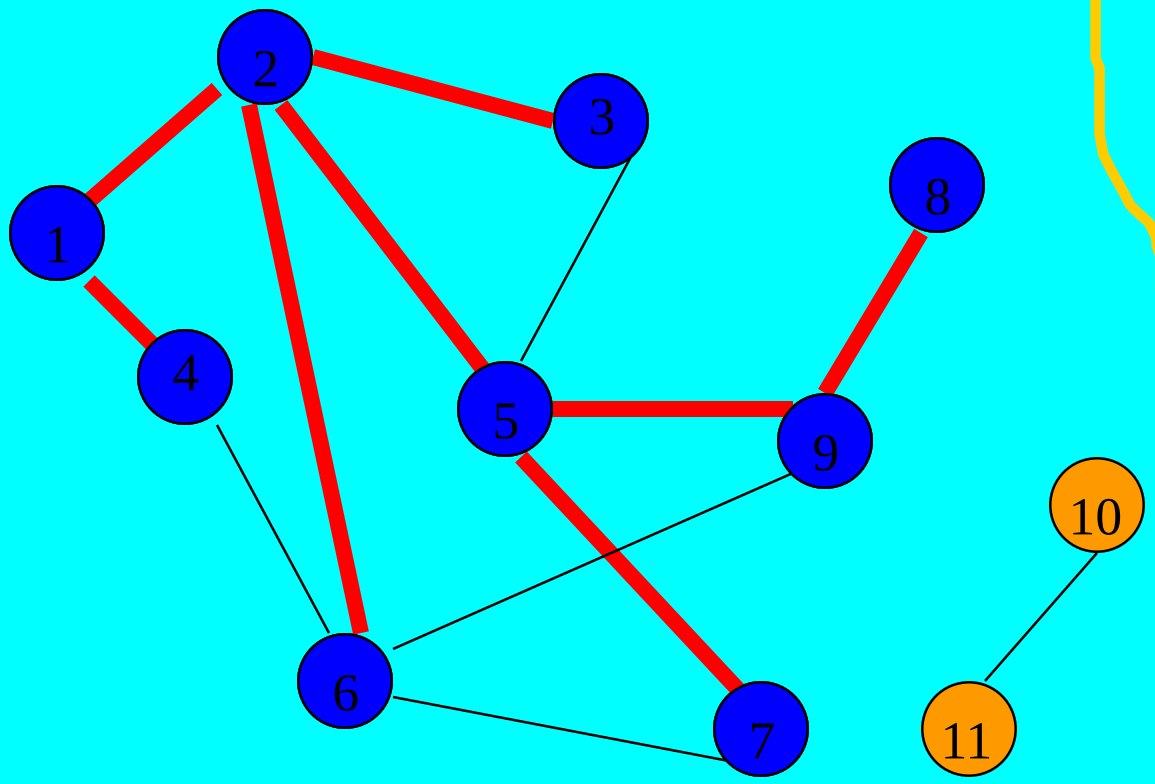


FIFO Queue

7 8

Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



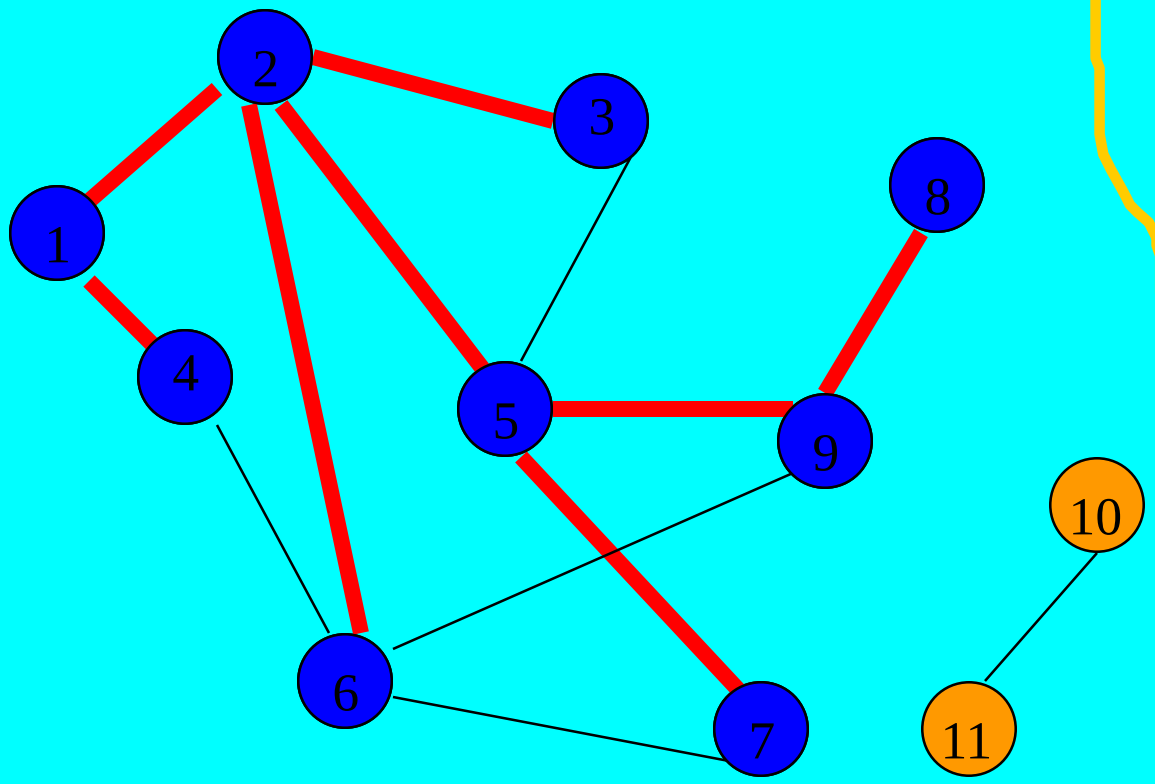
FIFO Queue

8

Remove 7 from Q; visit adjacent unvisited vertices;

put in Q.

Breadth-First Search Example

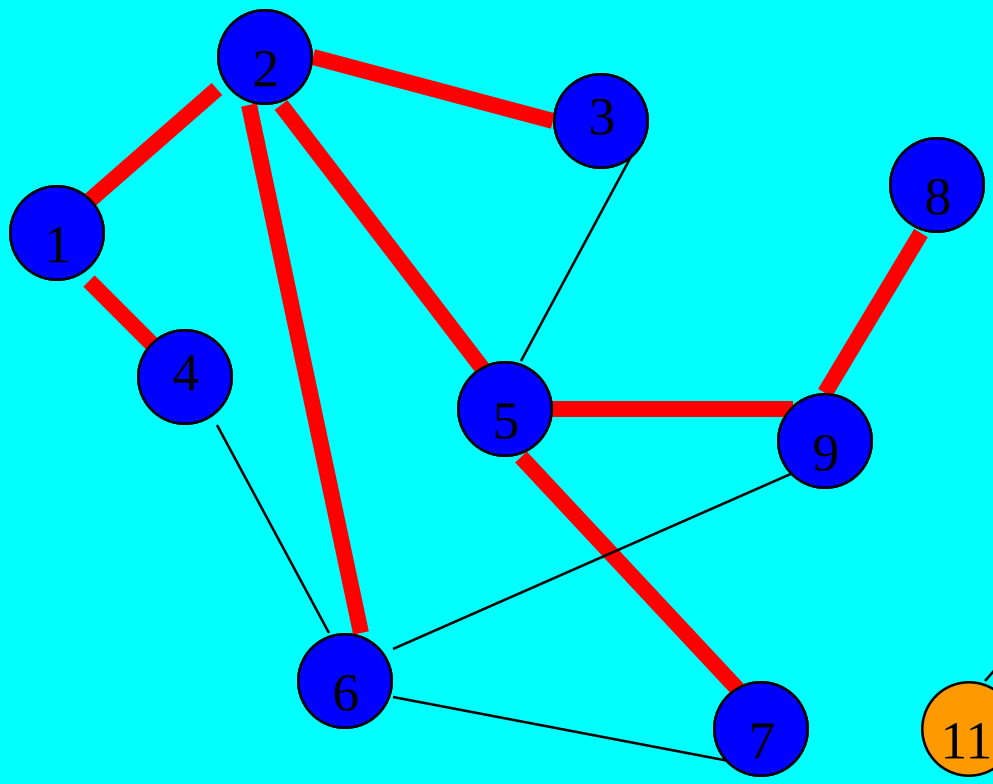


FIFO Queue

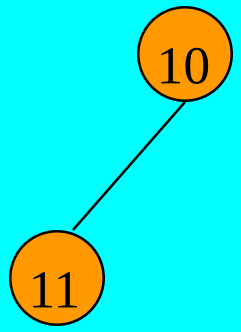
8

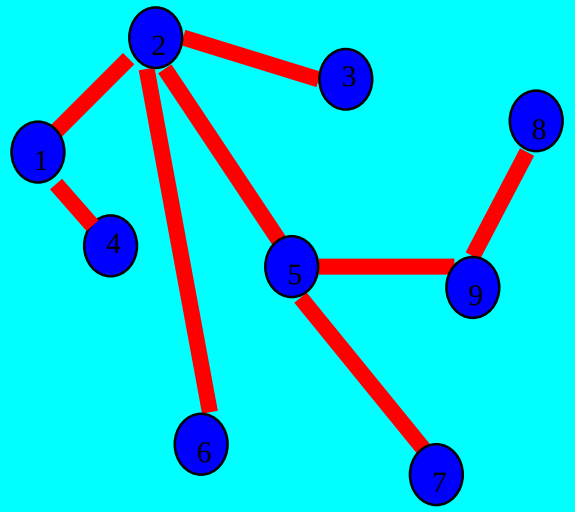
Remove 8 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



FIFO Queue



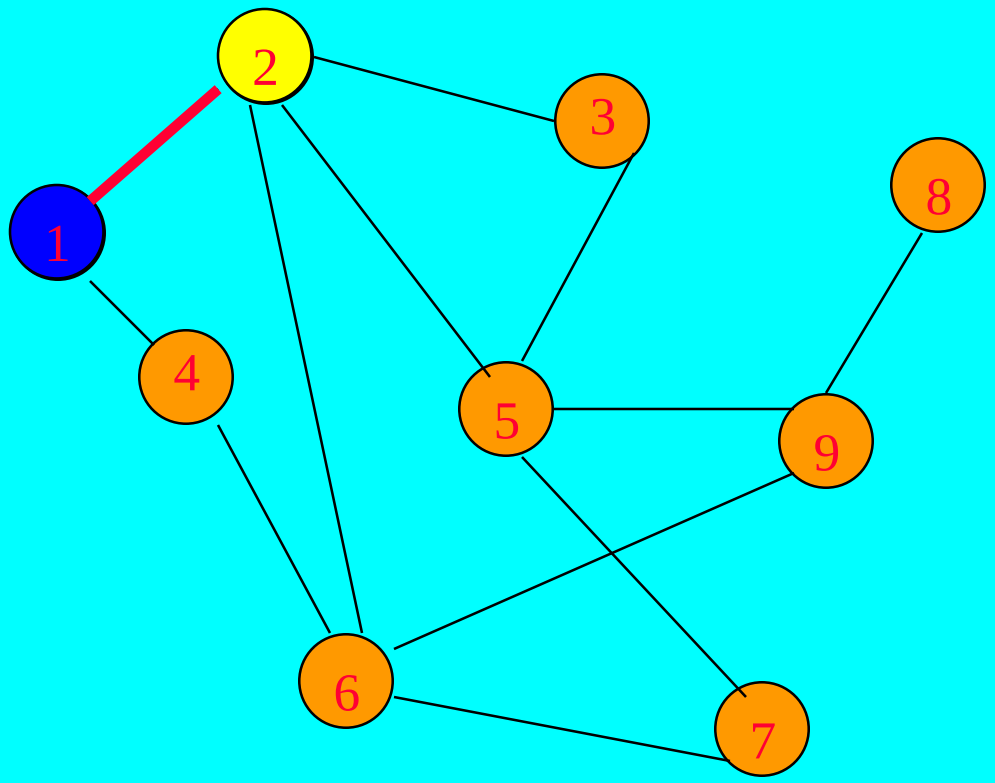


Queue is empty. Search terminates.

Depth-First Search

```
depthFirstSearch(v)
{
   Label vertex v as reached.
   for (each unreached vertex u
                                adjacenct from v)
      depthFirstSearch(u);
}
```

Depth-First Search Example



1

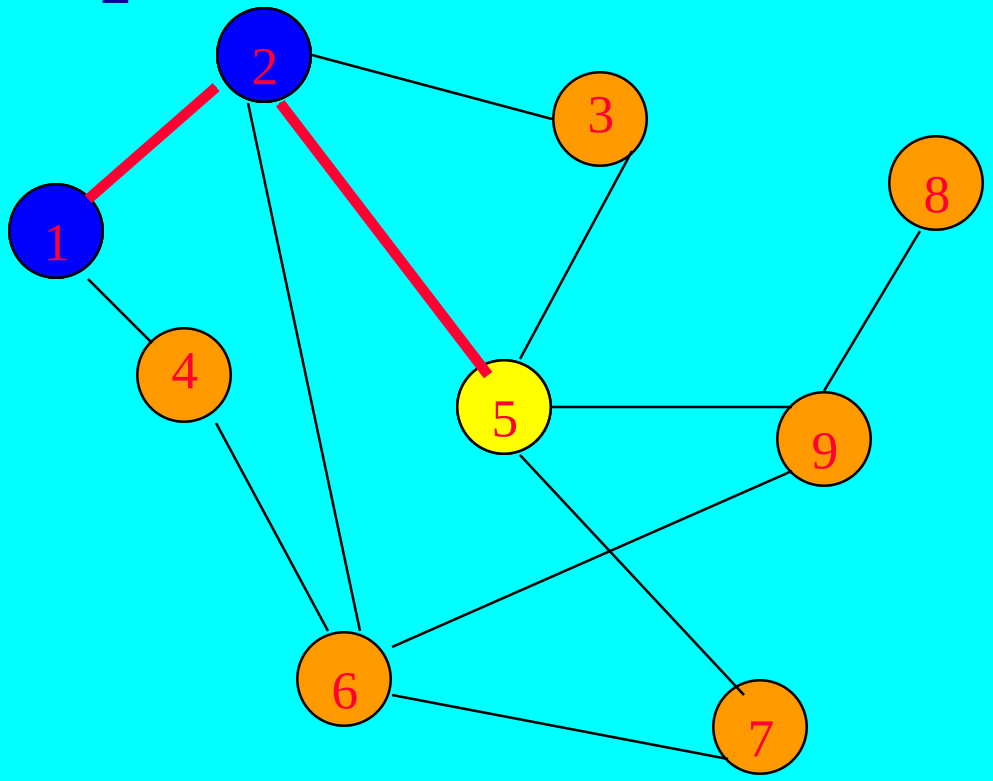
stack

Start search at vertex 1.

Label vertex 1 and do a depth first search from either 2 or 4.

Suppose that vertex 2 is selected.

Depth-First Search Example



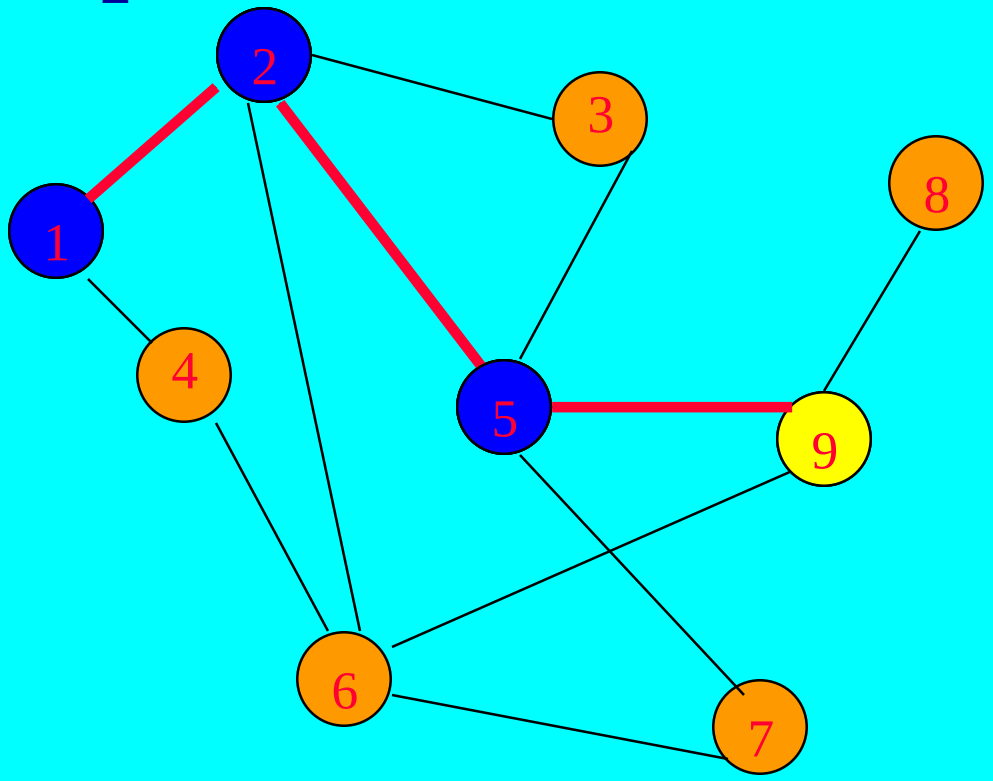
2 1

stack

Label vertex 2 and do a depth first search from either 3, 5, or 6.

Suppose that vertex 5 is selectd.

Depth-First Search Example



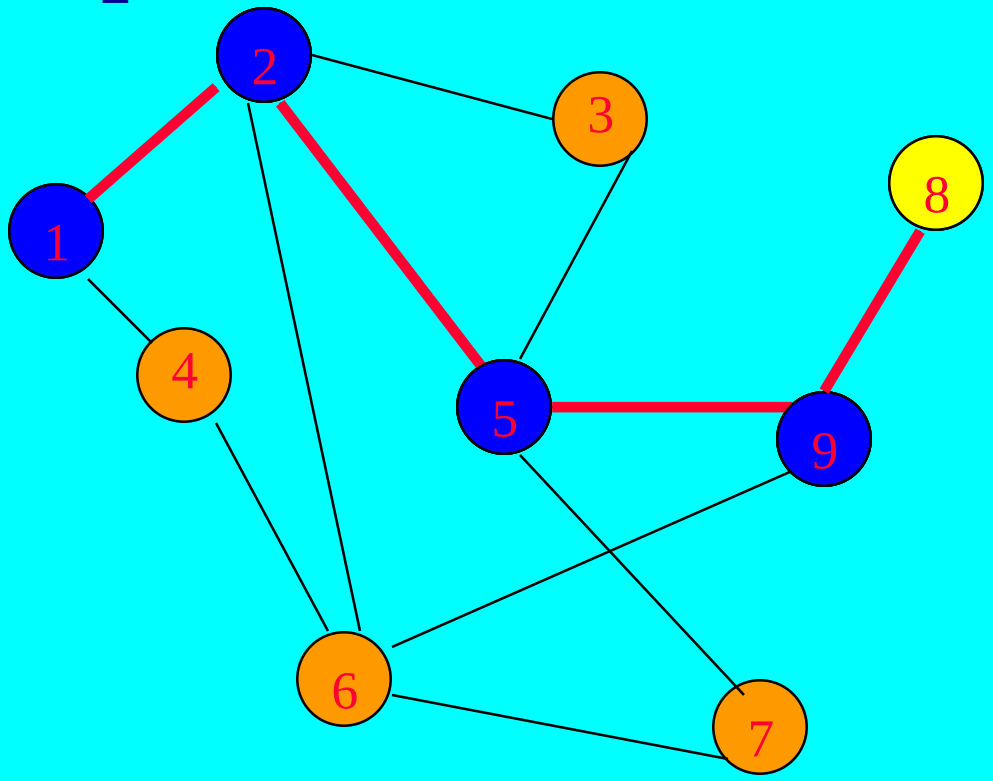
5 2 1

stack

Label vertex 5 and do a depth first search from either 3, 7, or 9.

Suppose that vertex 9 is selected.

Depth-First Search Example



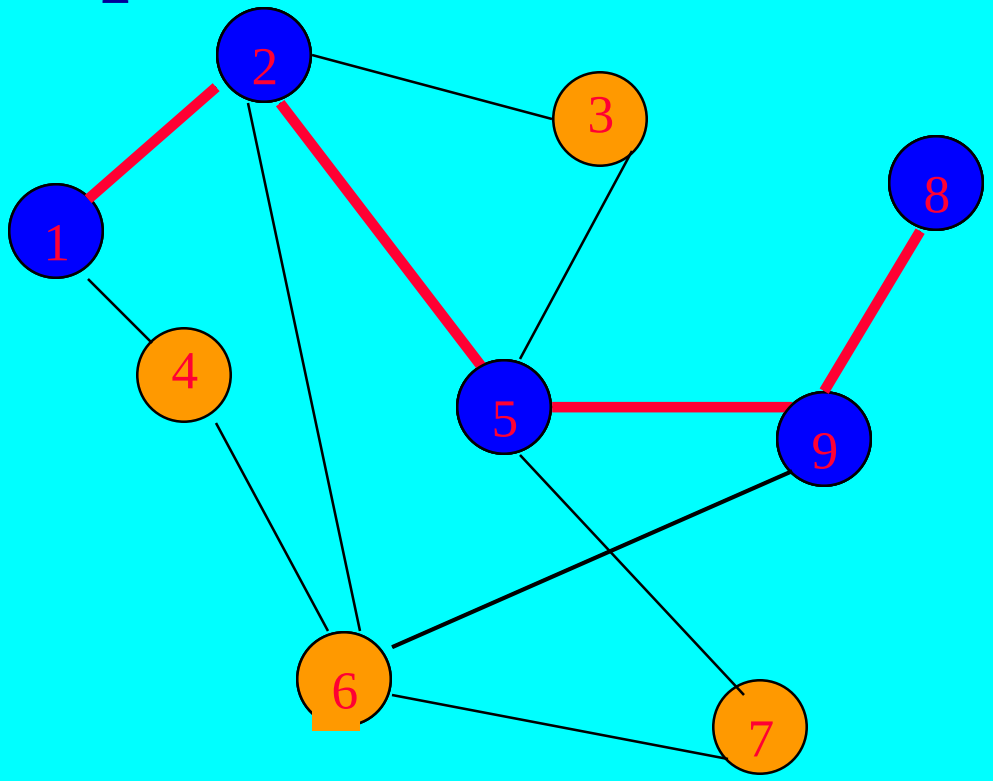
9 5 2 1

stack

Label vertex 9 and do a depth first search from either 6 or 8.

Suppose that vertex 8 is selected.

Depth-First Search Example



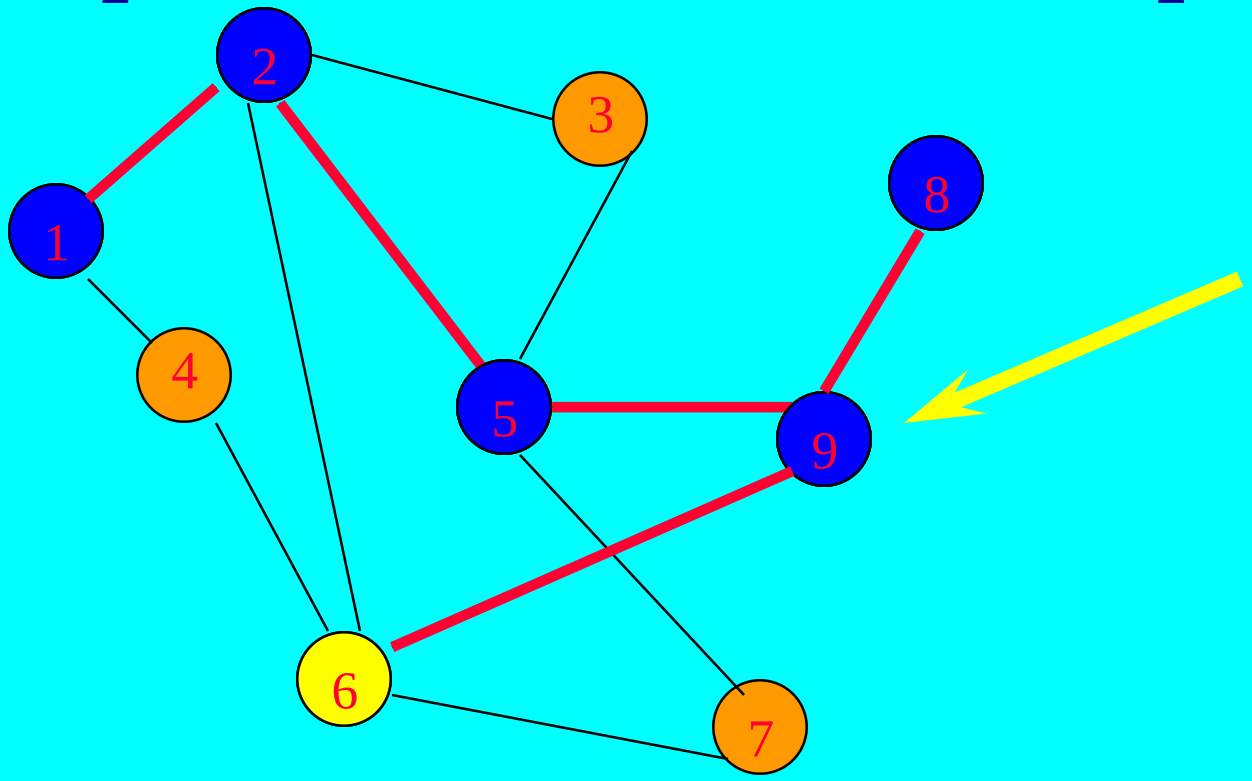
8 9 5 2 1

stack

Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6).

Depth-First Search Example



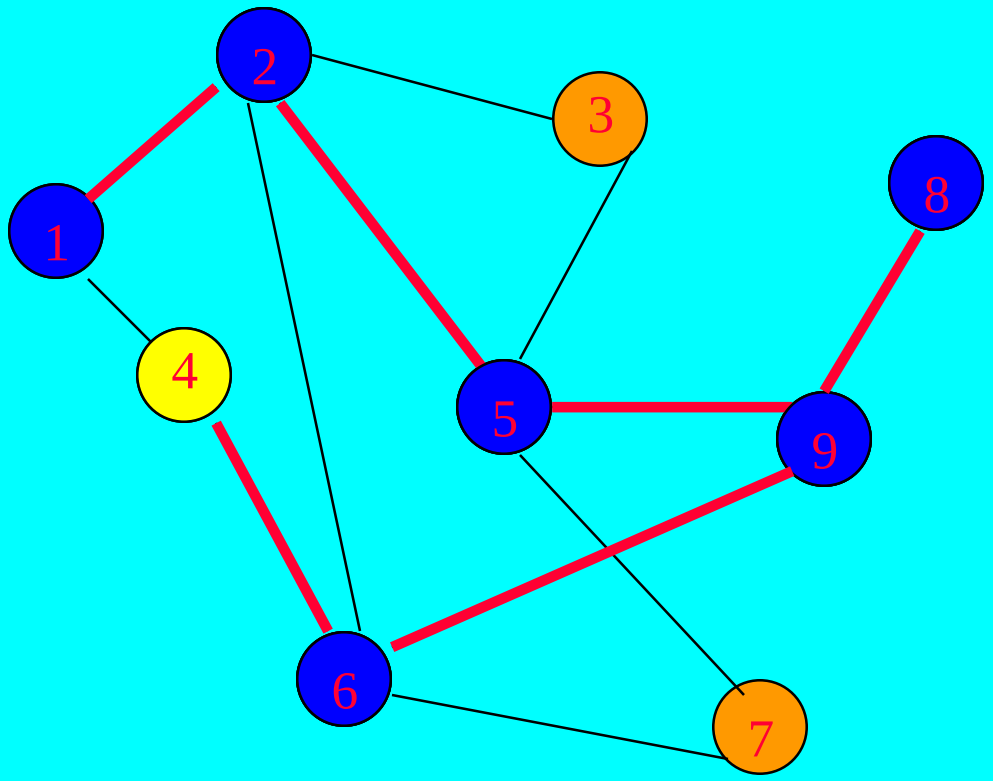
9 5 2 1

stack

Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6).

Depth-First Search Example



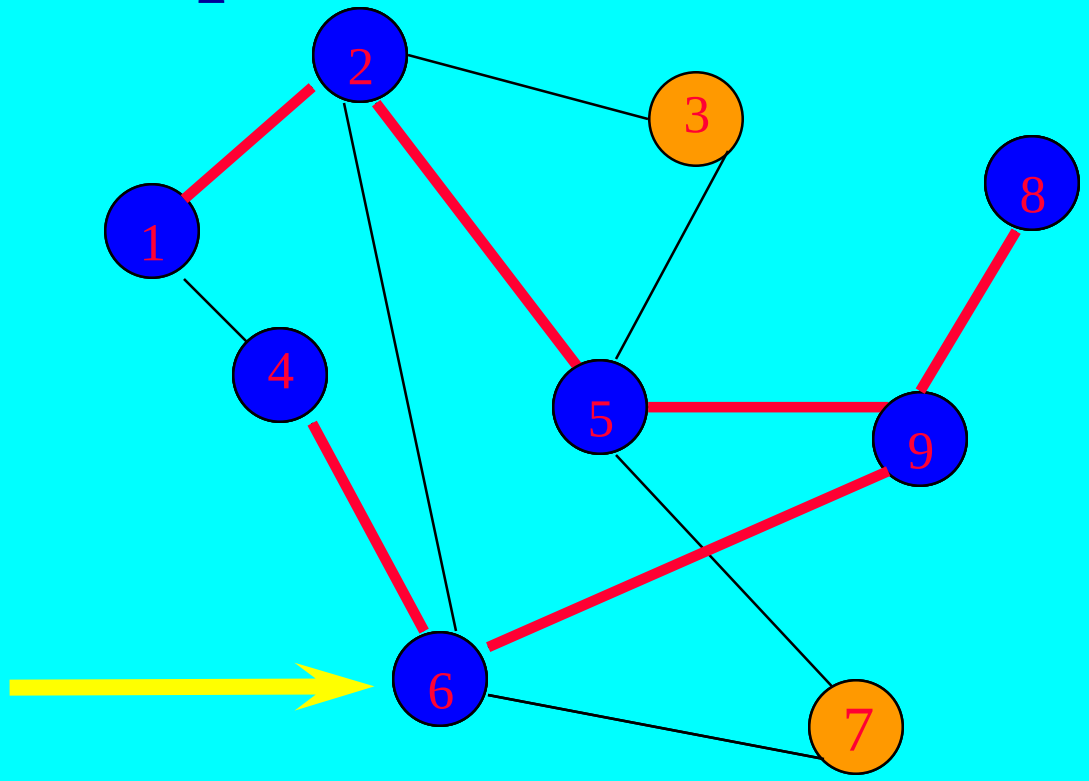
6 9 5 2 1

stack

Label vertex 6 and do a depth first search from either 4 or 7.

Suppose that vertex 4 is selected.

Depth-First Search Example



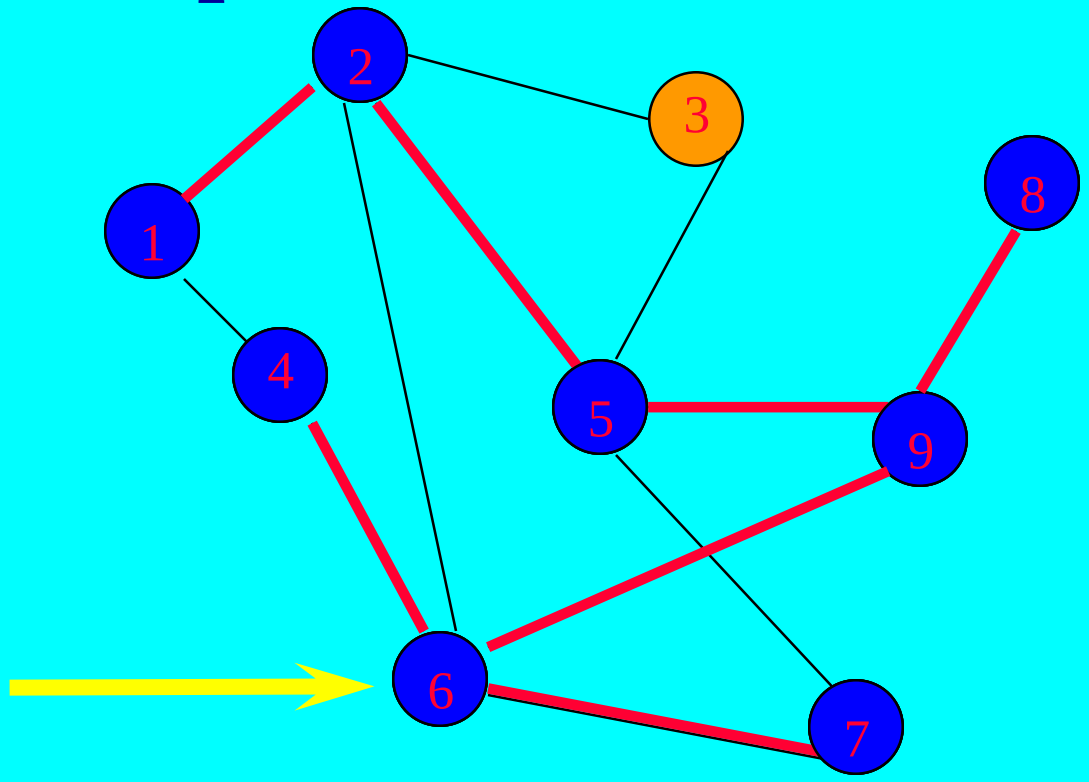
4 6 9 5 2 1

stack

Label vertex 4 and return to 6.

From vertex 6 do a dfs(7).

Depth-First Search Example

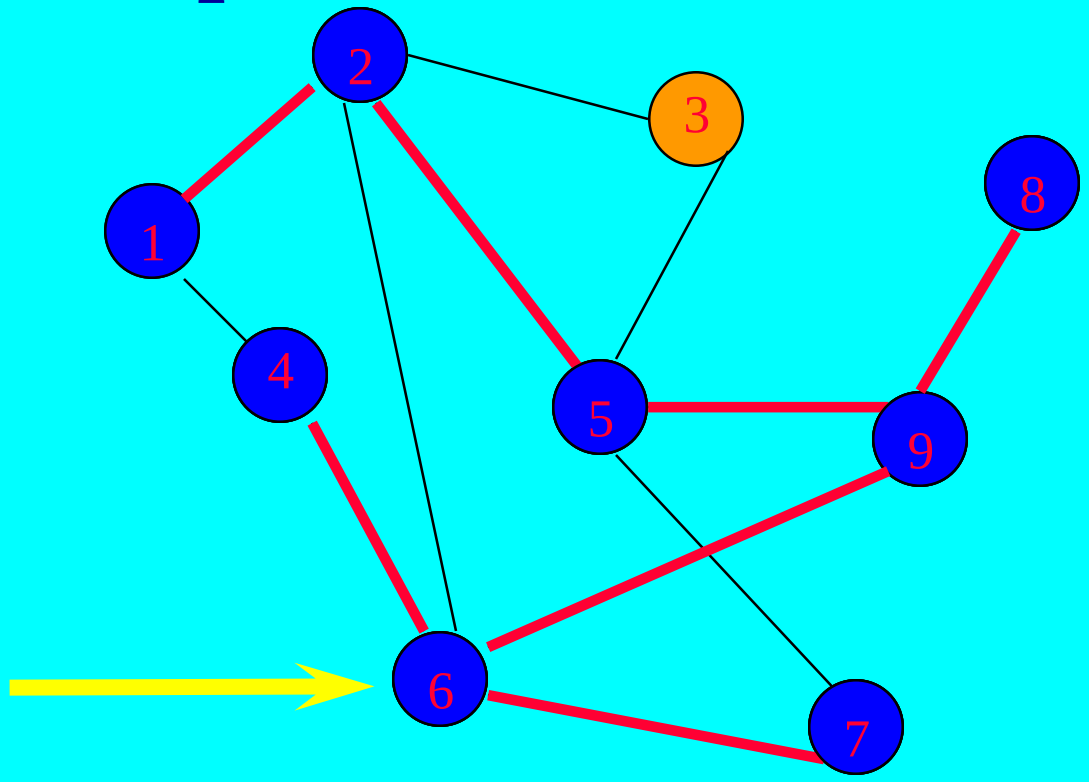


7 6 9 5 2 1

stack

Label vertex 7 and return to 6.

Depth-First Search Example



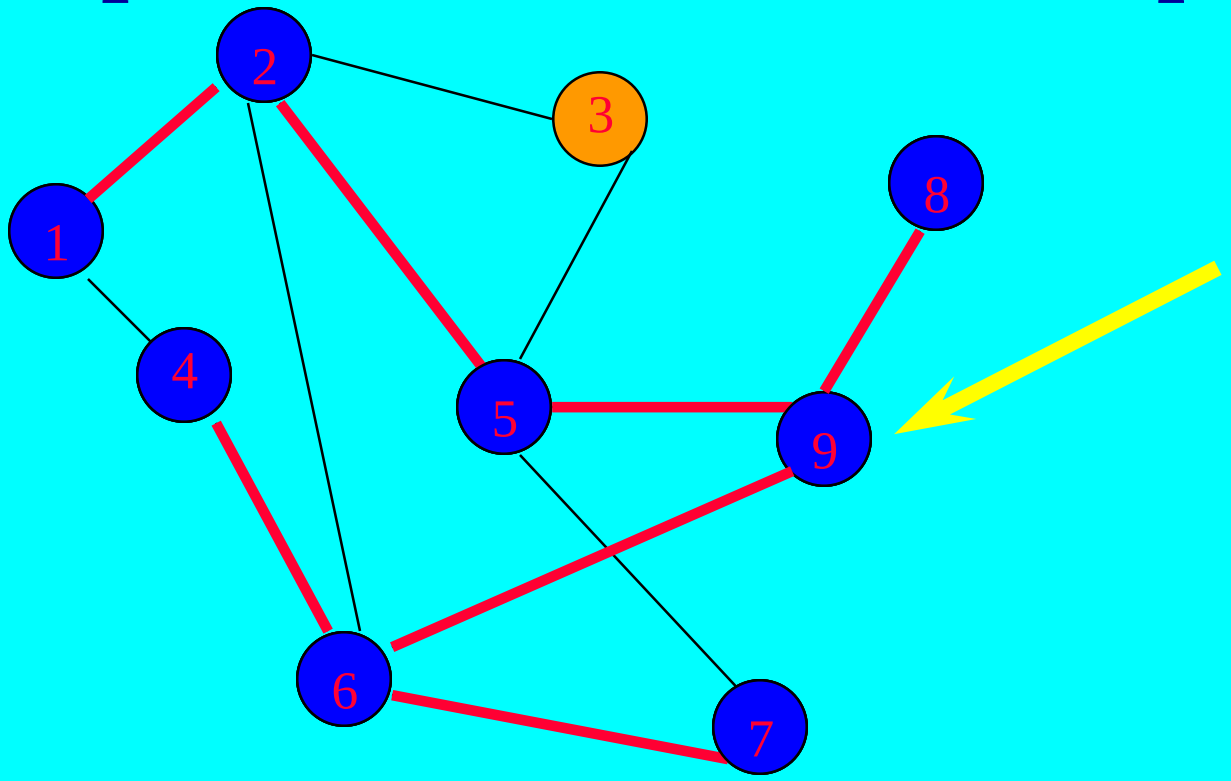
6 9 5 2 1

stack

Label vertex 7 and return to 6.

Return to 9.

Depth-First Search Example



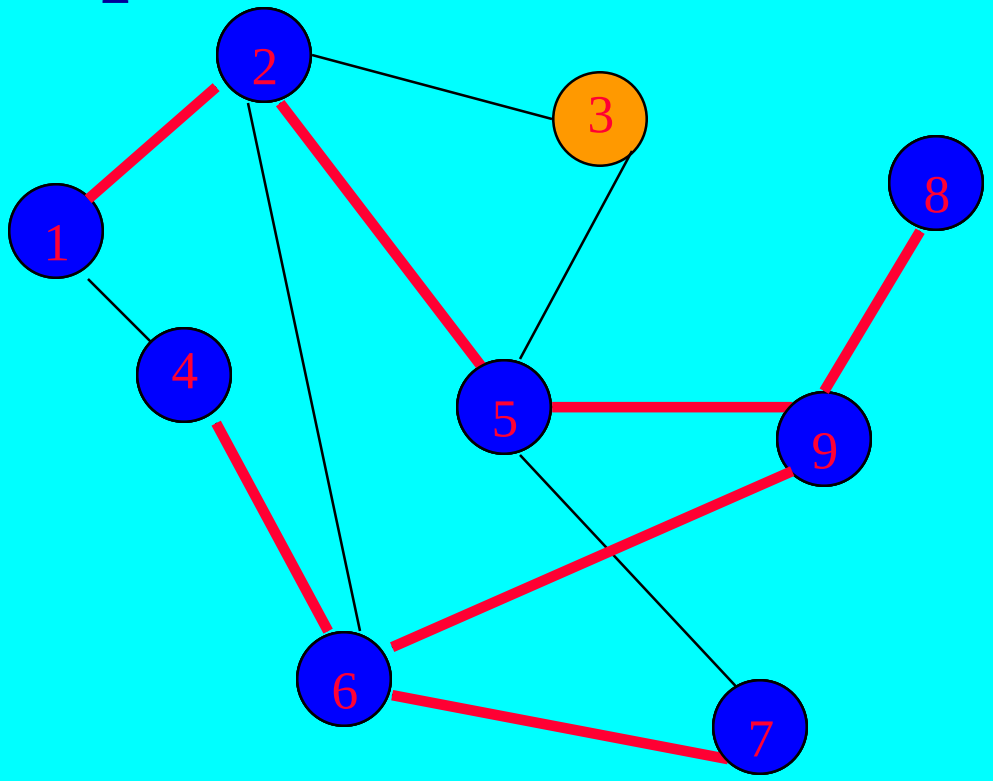
9 5 2 1

stack

Label vertex 7 and return to 6.

Return to 9.

Depth-First Search Example



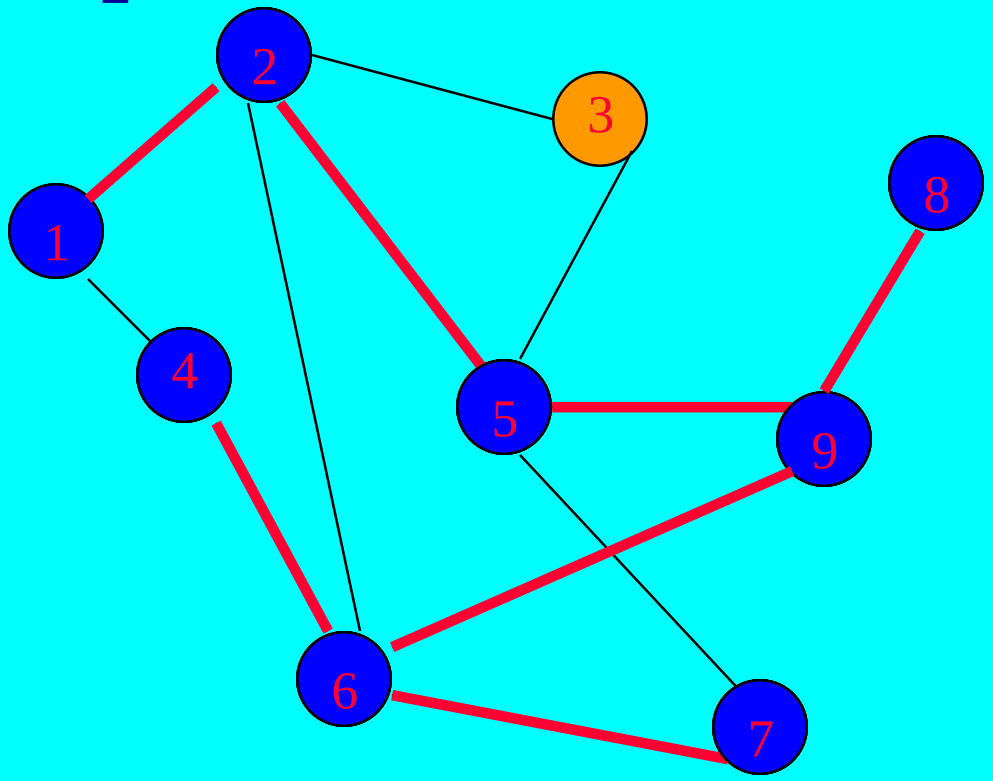
5 2 1

stack

Label vertex 7 and return to 6.

Return to 9.

Depth-First Search Example

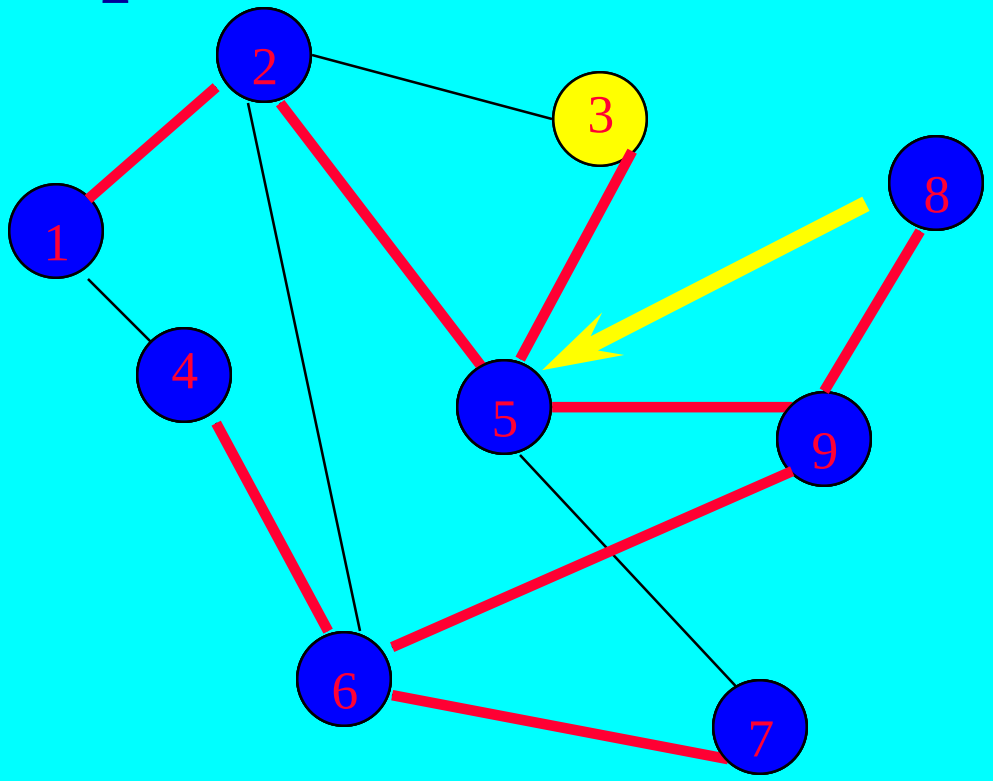


5 2 1

stack

Return to 5.

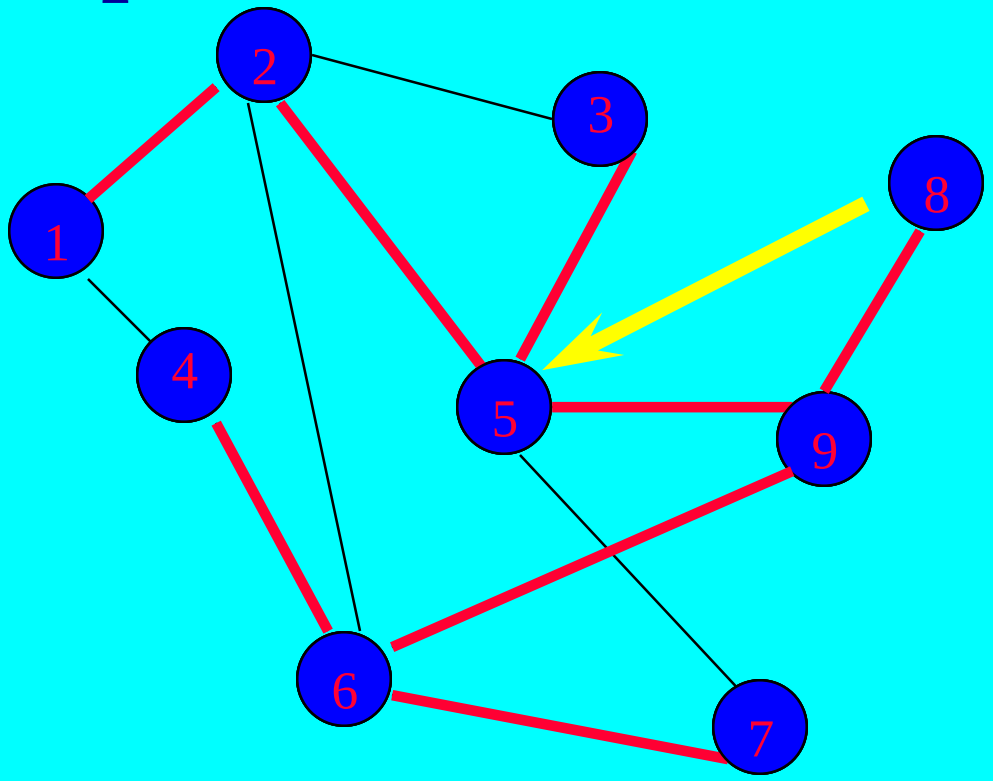
Depth-First Search Example



5 2 1

Do a dfs(3).

Depth-First Search Example

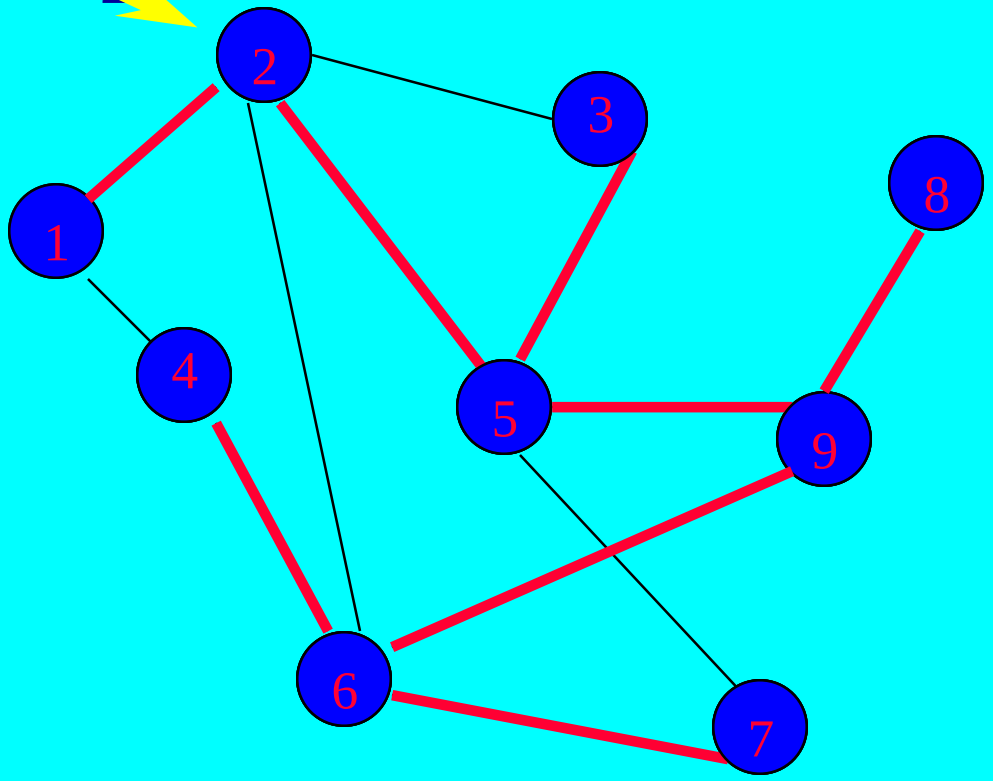


3 5 2 1

stack

Label 3 and return to 5.

Depth-First Search Example

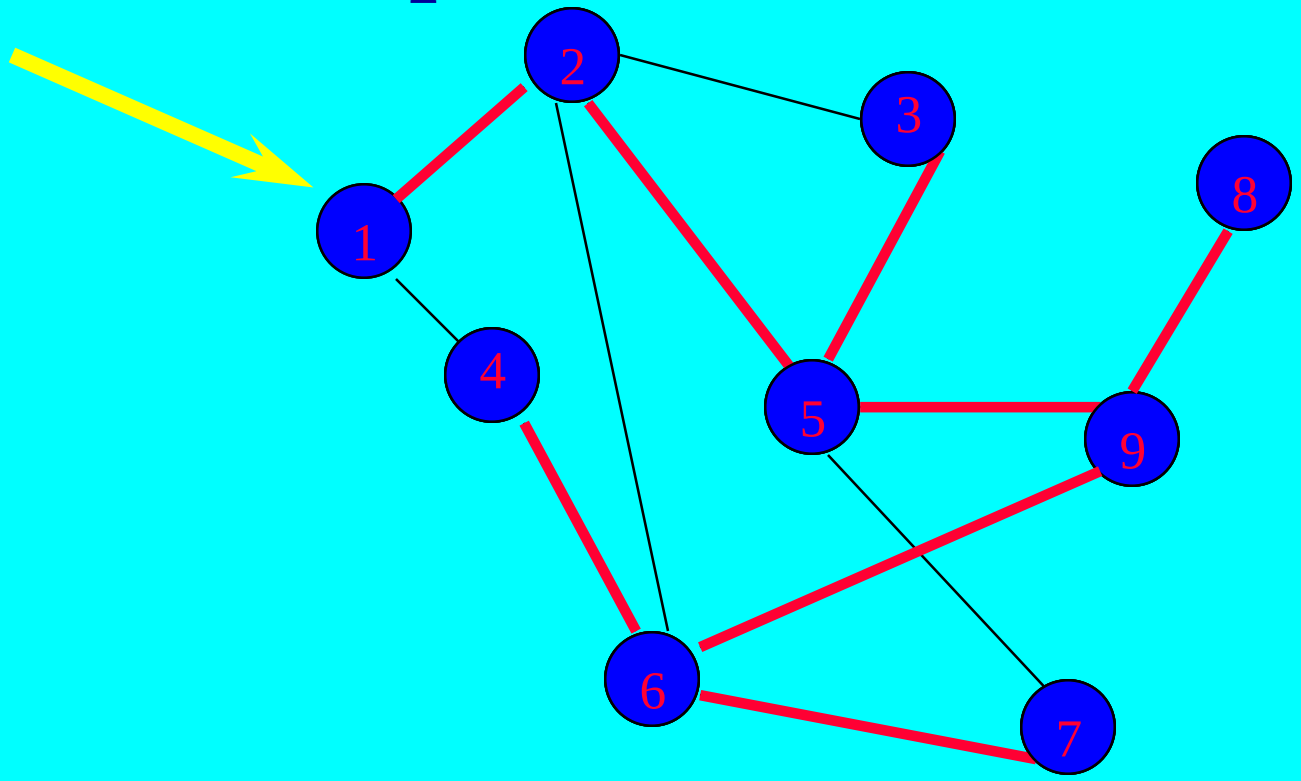


2 1

stack

Return to 2.

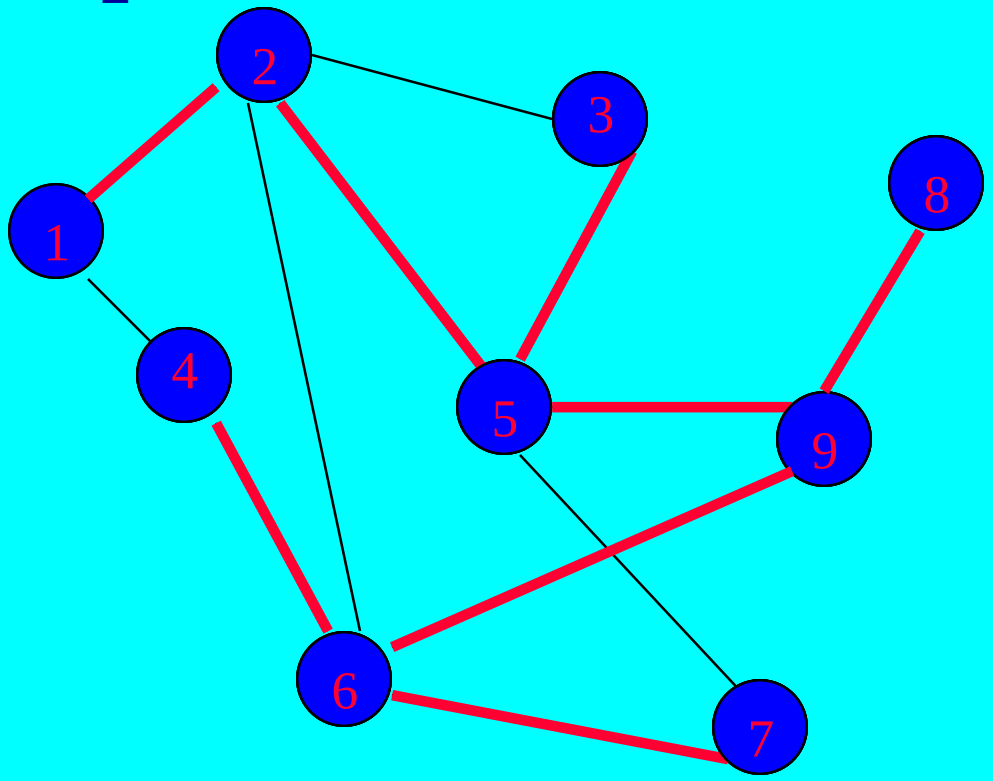
Depth-First Search Example



1
stack

Return to 1

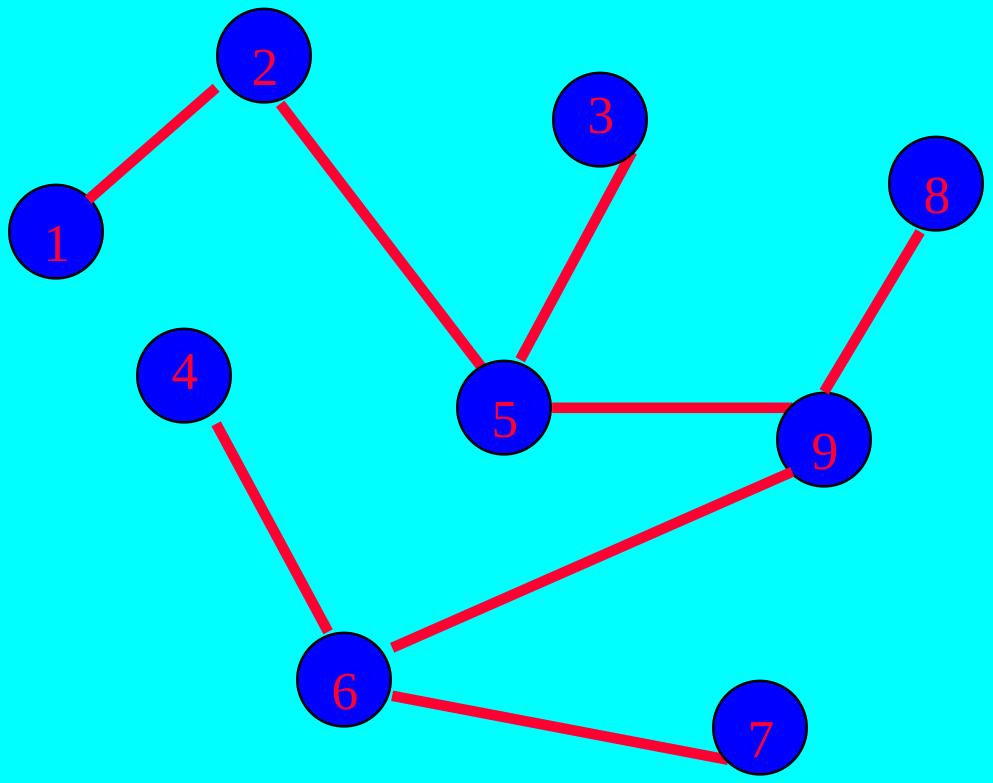
Depth-First Search Example

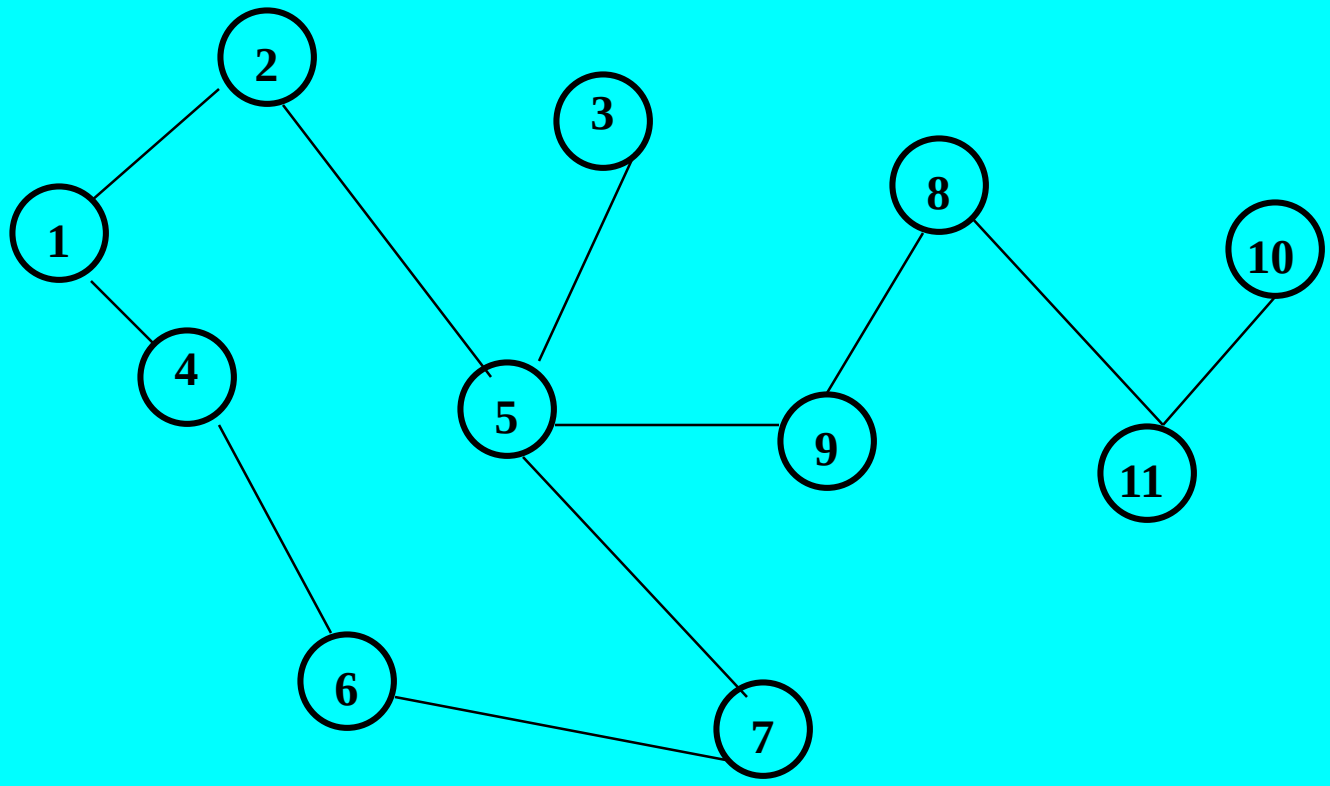


stack

Return to invoking method.

OUT PUT:




2
3
1
8
10
4
5
9
11
6
7