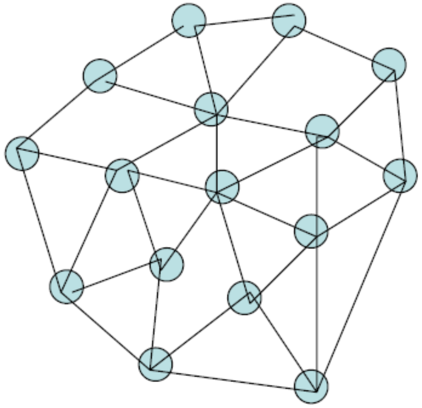
GRAPHS

CSC 212

Graphs

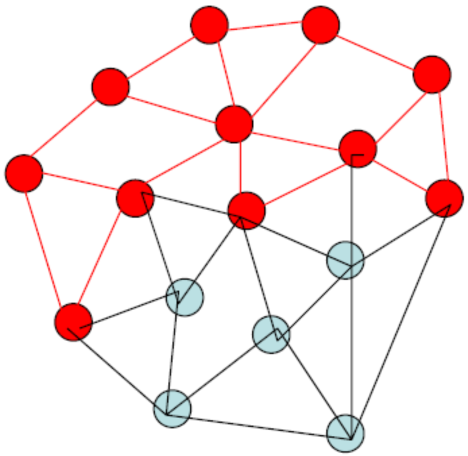
- Many interesting situations can be modeled by a graph.
- is a way of representing connections or relationships between pairs of objects from some set.



- Ex. Mass transportation system, computer network, electrical engineering

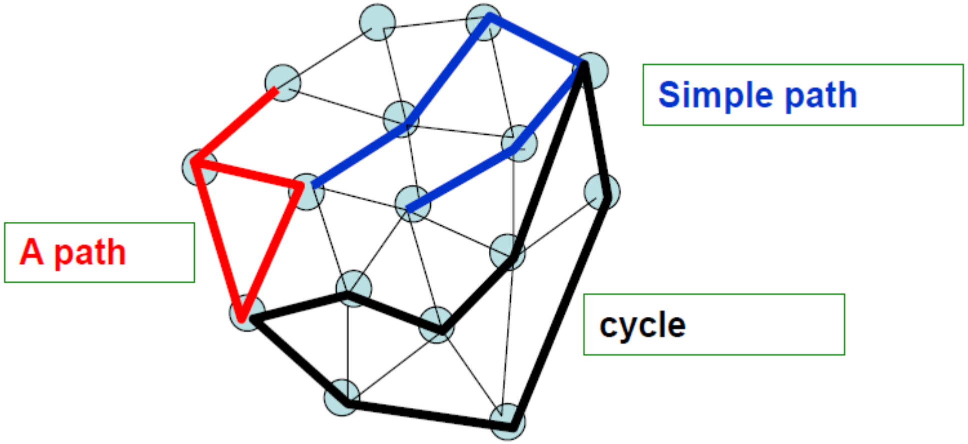
Terminology

- A **graph** consists of a set of vertices and a set of edges.
- A **vertex v** is basic component, which usually contains some information.
- An **edge (v,w)** connects two distinct vertices v and w .
- A **subgraph** is graph which consists of a subset of nodes (vertices) and a subset of edges of a graph.



Terminology

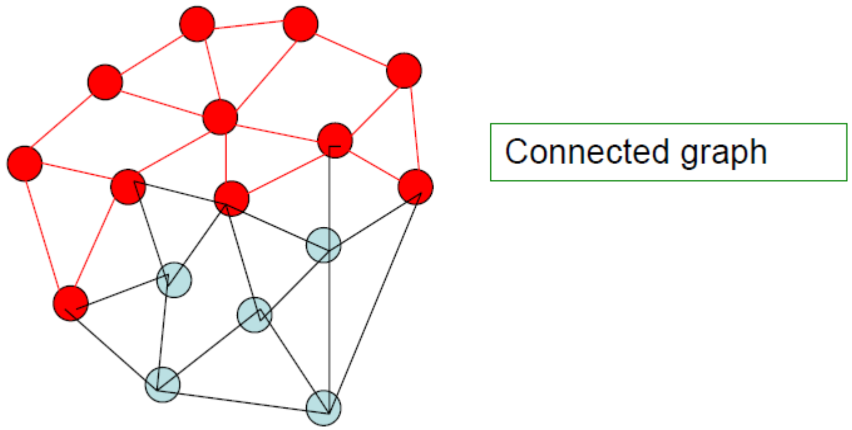
- A **path** is a sequence of nodes such that each successive pair is connected by an edge.
- A path is a **simple path** if each of its nodes occurs once in the sequence.

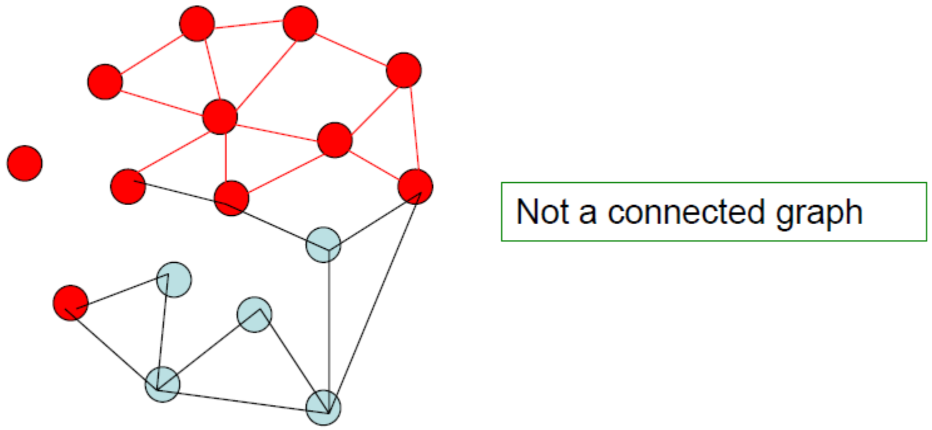


- A **cycle** is path that is a simple path except that the first and last nodes are the same.

Terminology

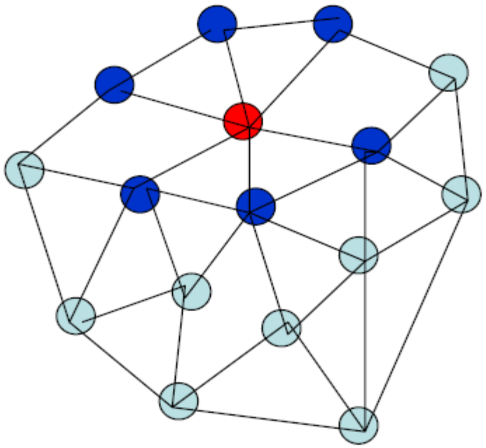
- A graph is a **connected graph** if there is a path between every pair of its nodes.





Terminology

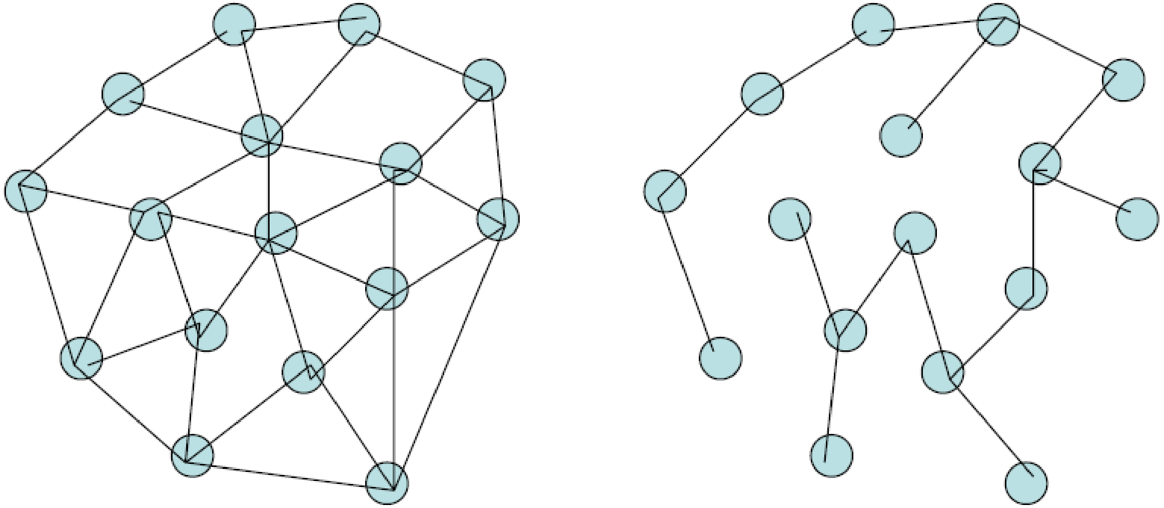
- Two nodes are **adjacent nodes** if there is an edge that connects them.



- **Neighbors** of a node are all nodes that are adjacent to it.
- A Tree is the special case of a graph that
  - (i) is connected
  - (ii) has no cycle

Terminology

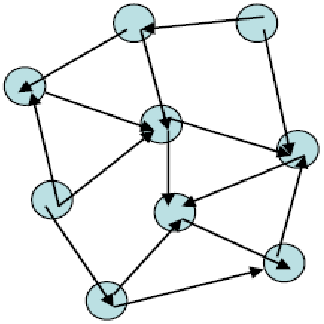
- If a connected graph has n nodes and n-1 edges, then it is a tree. This tree is called **Spanning Tree**.



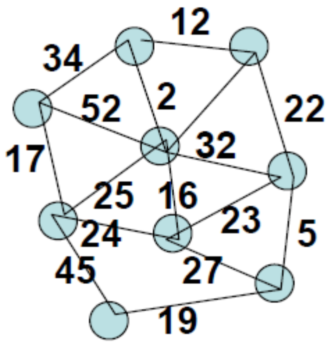
- An oriented tree is a tree in which one node has been designated the root node.

Terminology

- A **directed graph** or digraph is a graph in which each edge has an associated direction.



- A **weighted graph** is a graph in which each edge has an associated value.



Specification of Graph

- **Elements**:

  A graph consists of nodes and edges

- **Structure**:

  An edge is a one-to-one relationship between a pair of distinct nodes. A pair of nodes can be connected by at most one edge, but any node can be connected to any collection of other nodes.

- **Domain**:

  The number of nodes (vertices) in a graph is bounded.

Specification of Graph

Operations:

InsertNode (Type e)

Requires: G is not full.

Results: If G does not contain an element whose key value is e.key then e is inserted in G and inserted is true, otherwise inserted is false.

InsertEdge (Key k1, k2)

Requires: G is not full and k1!=k2.

Results: If G contains two nodes whose key values are k1 and k2 then G contains an edge connecting those nodes. If the two nodes were connected by an edge before operation InsertEdge then inserted is false; otherwise inserted is true.

Specification of Graph

- DeleteNode (Key k)
- Results: G does not contain an element whose key value is k. if G contained a node before this operation with key value k then deleted is true and no edge that connected this node to an other node is in G; otherwise deleted is false.
- DeleteEdge (Key k1, k2)
- Results: G does not contain an edge that connects nodes whose key values are k1 and k2. If G contained such an edge before this operation then deleted is true; otherwise deleted is false.

Specification of Graph

- Update (T e)

  Results: If G contained a node with key value e.key then the element in the node is e and updated is true. Otherwise updated is false.

- Retrieve (key k, T e)

  Results: If G contains a node whose key value is k before this operation then e is that element and retrieved is true; otherwise retrieved is false.

- Full ()

  Results: If G is full then Full returns true; otherwise Full returns false.

Representation of a Graph

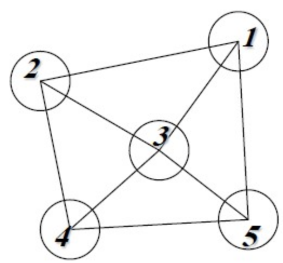
There are two approaches to representing graphs

- (i) Adjacency Matrix
- (ii) Adjacency List

Adjacency Matrix

- A two dimensional array whose components are of type Boolean and whose index values correspond to the nodes.

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



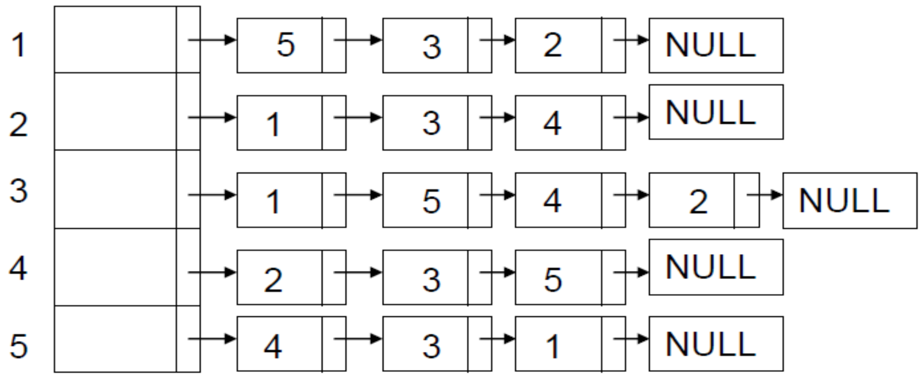
If A is the adjacency matrix corresponding to a graph G, then A_{ij} corresponds to the edge (i, j), and is true if there is an edge between the vertices i, and j.

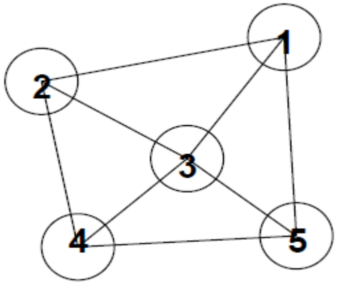
- This representation is very simple, but the space requirement is $O(n^2)$ if the number of vertices is n.

- This representation is suitable only if the graph is dense with the number of edges i.e. $O(n^2)$, n being the number of vertices. But in most of the applications this is not true.

Adjacency list

- It is an array where each cell corresponds to a vertex of a graph and stores the header of a list of all adjacent vertices.
  - Ex. The following is an adjacency list corresponding to the graph on the right.





- This is an ideal representation if the graph is not dense.

- In this case the space requirement is O(e + n) where e is the number of edges and n is the number of vertices.

Traversal of a Graph

- Process each node of the graph only once

- There are two methods for graph traversal

  (i) Breadth First Search

  (ii) Depth First Search

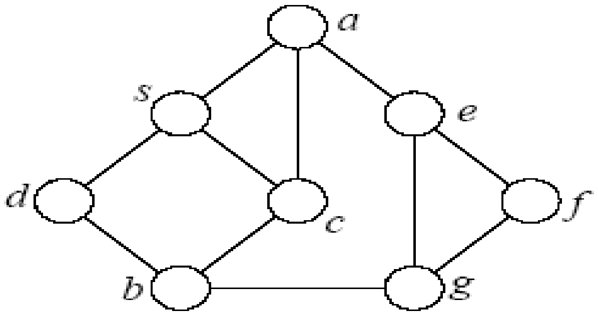
Breadth First Search (BFS)

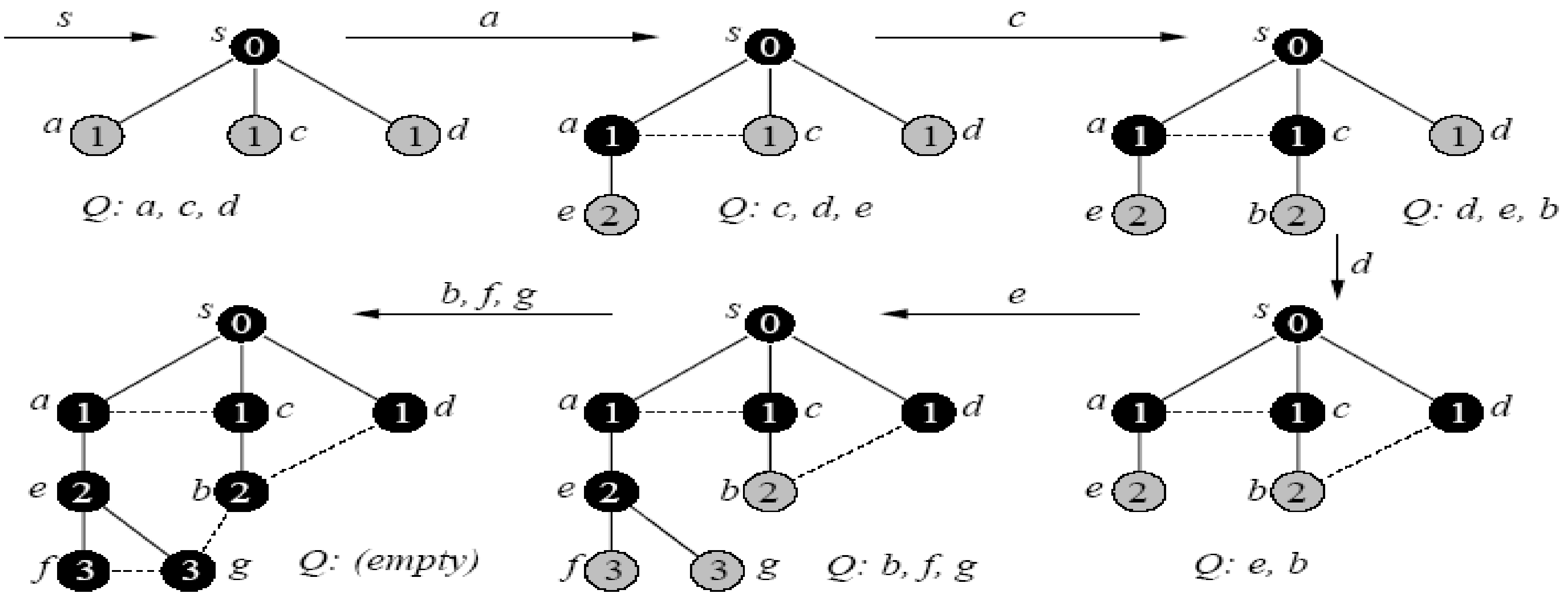
- **Breadth First Search (BFS)** Start at some source node s and visit all its neighbors, then visit the neighbors of each neighbor of s and so on.
- **ALgorithm**

1. Assign the status of waiting to all nodes of a graph.
2. Start with source node s, and put it in queue
3. Dequeue one node from the queue, assign it the status of processed, and assign its neighbors the status of ready and put them in the queue.
4. Repeat Step 3 until the queue is empty.

Breadth First Search (BFS)

Example
Consider a graph with
nodes a, b, c, d, e, f, g, s





Breadth First Search (BFS)

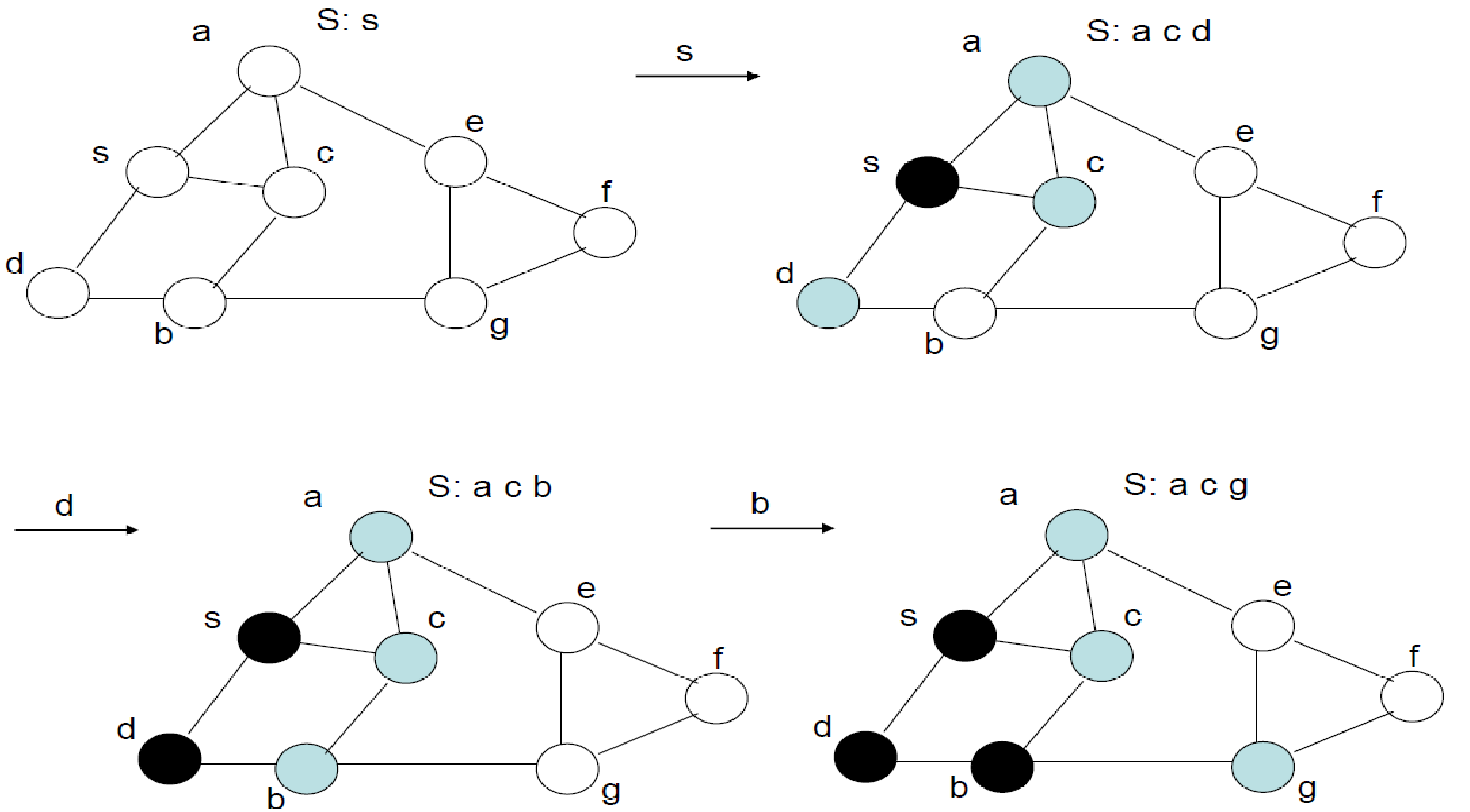
Pseudocode for BFS

```
BFS(G, s)
{
    for each node u of G
        set status[u] = waiting;
    status[s] = ready;
    Q = {s};
    while( !Q.IsEmpty)
    {
        u = Q.dequeue();
        for each v in Neigh[u]
            {
            if(status[v] = waiting)
            {
            staus[v] = ready;
            Q.enqueue(v);
            }
            }
        status[u] = processed;
    }
}
```

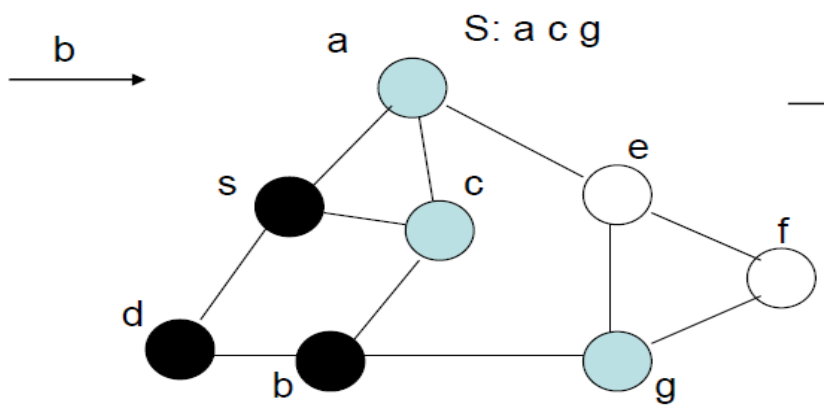
Depth First Search (DFS)

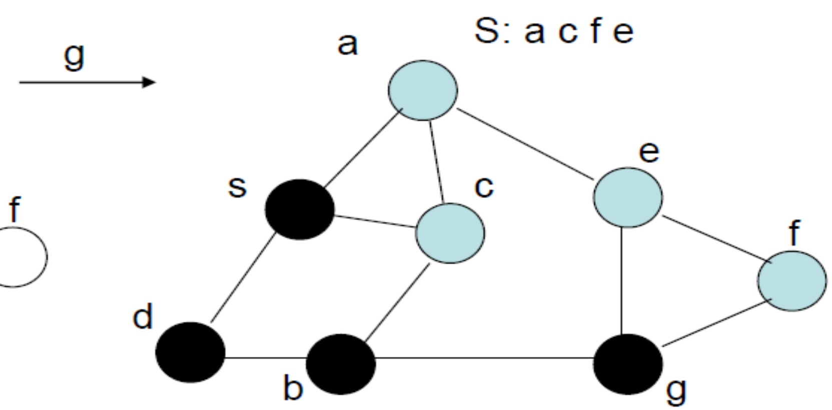
- DFS visits one node and then visits one of its neighbors and puts the rest of its neighbors into a stack and so on.
- Algorithm
  1. Assign the status of **waiting** to all nodes of a graph.
  2. For each node of the graph, follow the following steps,
     a) Change the status of the node to be **ready** and put in a stack
     b) Repeat the following steps until the stack is empty
     c) Get a node from the stack, **process** it, change its status to **processed** and change the status from waiting to ready of its neighbors and put them in the stack
     d) Go to step b

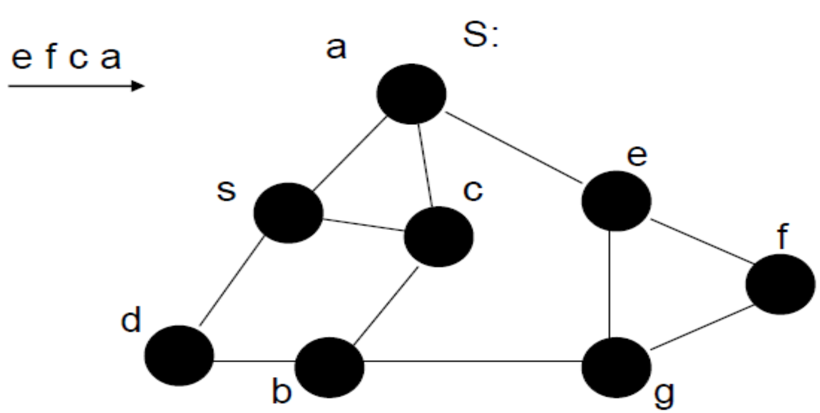
Depth First Search (DFS)



Depth First Search (DFS)







Depth First Search (DFS)

Pseudocode for DFS

```
DFS(G)
{
for each node u of G
       status[u] = waiting;
for each node u of G
  if(status[u] == waiting)
       Visit(u)
}
Visit(u)
{
stack S;
status[u] = ready;
S.push(u)
while( S.IsEmpty)
 {
       v = S.pop();
       for each w in Neigh[v]
       {
         if(status[w] = waiting)
       {
          staus[w] = ready;
          S.push(w);
        }
       }
       status[v] = processed;
}
}
```