Sets & Maps

Sets

A set is a collection of unordered elements without any duplicates.

Properties:

1.Unlike lists and stacks, the elements present in a set do not follow any particular order. They are randomly present in the set.

2.The elements are not repeated in a given set.

Methods	Description
add(ele)	Adds an element to the set
clear()	Removes all elements from the set
contains(ele)	Returns true if a specified element is in the set
isEmpty()	Returns true if the set is empty
remove(ele)	Removes a specific element from the set
size()	Returns the number of elements in the set

Usage: A set can be used to determine if a node is a neighbour of another node i.e. if you store all of node A's neighbours as a set, you can use the contains() method to quickly find out if a node named X is a neighbour of node A.

Sets & Maps

Maps

Maps represent a collection type that provides connection or mapping between the elements of a source set (domain) and a target set (range). A **Map cannot contain duplicate keys and each key can map to at most one value.(Keys are unique but values can be repeated)**

Methods in Map Interface:

1.public Object put(Object key, Object value): This method is used to insert an entry in this map.
2.public void putAll(Map map): This method is used to insert the specified map in this map.
3.public Object remove(Object key): This method is used to delete an entry for the specified key.
4.public Object get(Object key):This method is used to return the value for the specified key.
5.public boolean containsKey(Object key): This method is used to search the specified key from this map.
6.public Set keySet(): This method is used to return the Set view containing all the keys.
7.public Set entrySet(): This method is used to return the Set view containing all the keys and values.

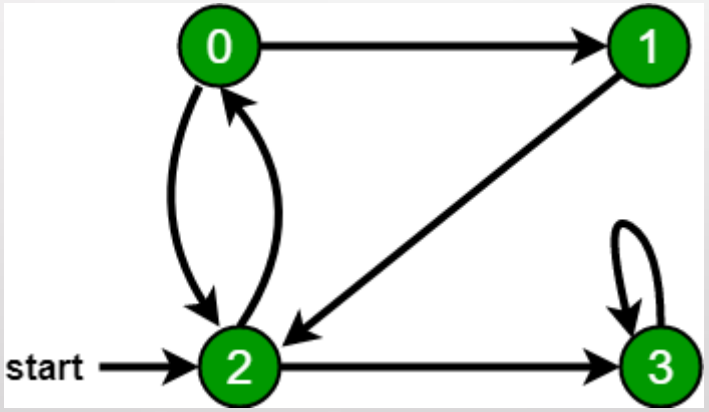
Implementation of Graphs

Adjacency List Implementation using Set & Map

Breadth First Search (BFS)

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. It will become a non-terminating process.

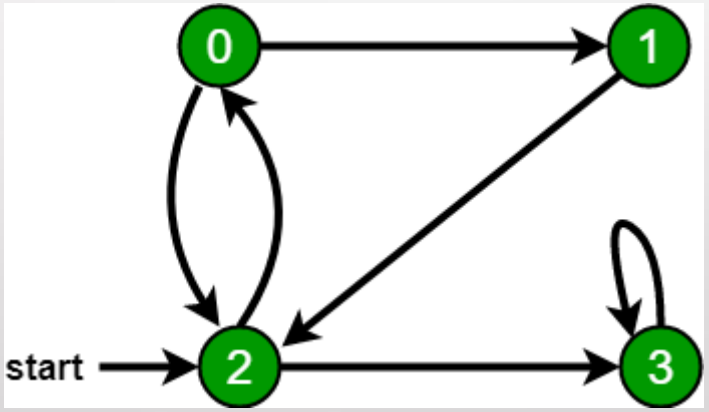


Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Breadth First Search (BFS)

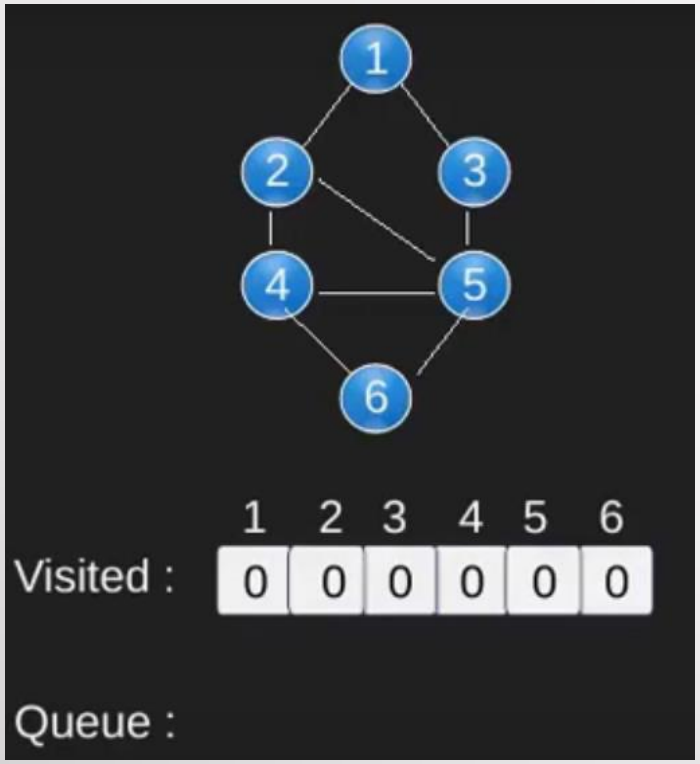
Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.

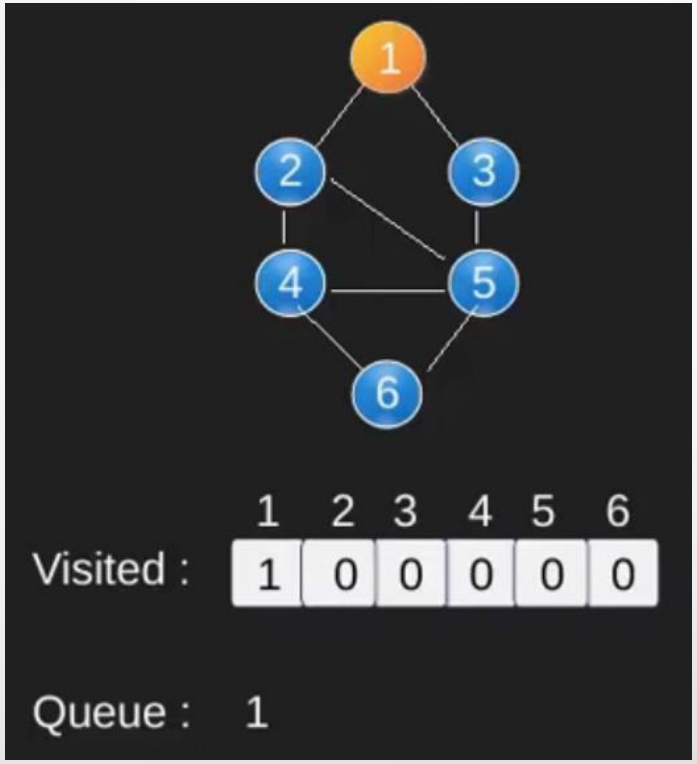
To avoid processing a node more than once, we use a boolean visited array in case of List Representation or a HashSet in case of Set Representation.

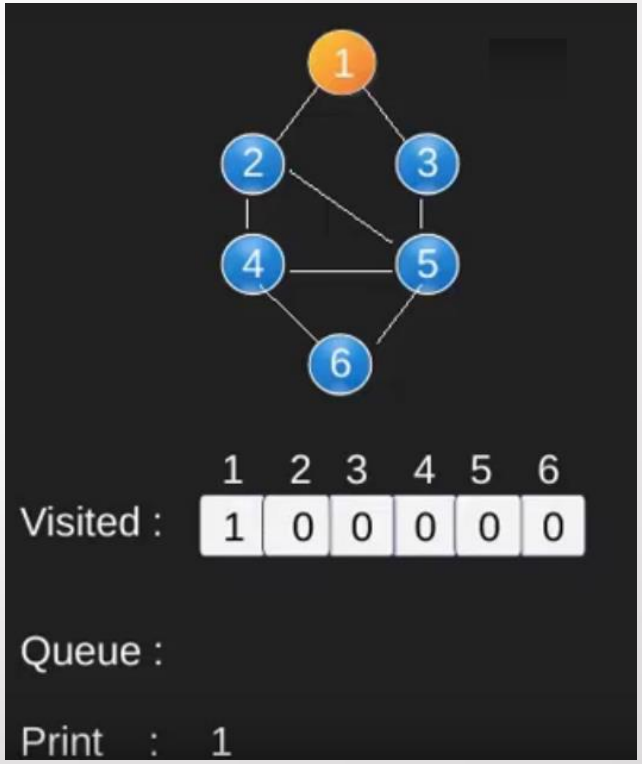


Breadth First Search (BFS)

An Example

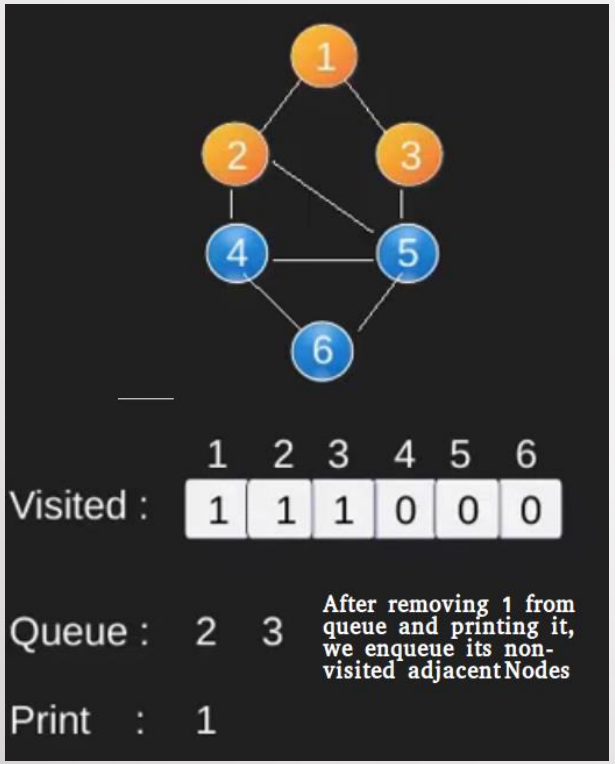


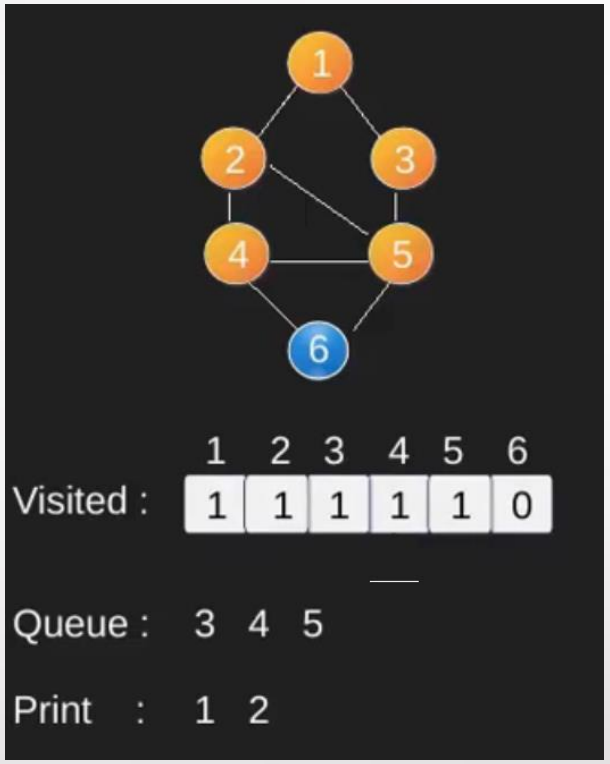


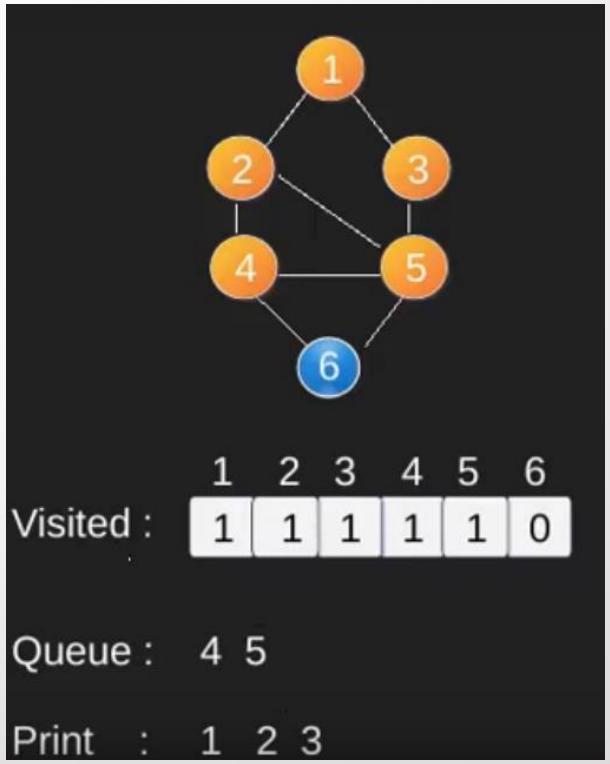


Breadth First Search (BFS)

An Example (Continued)

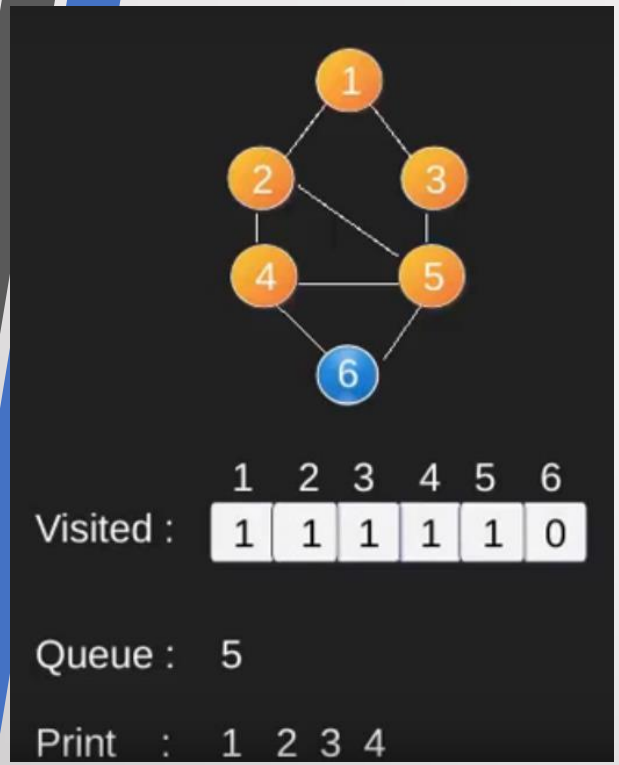


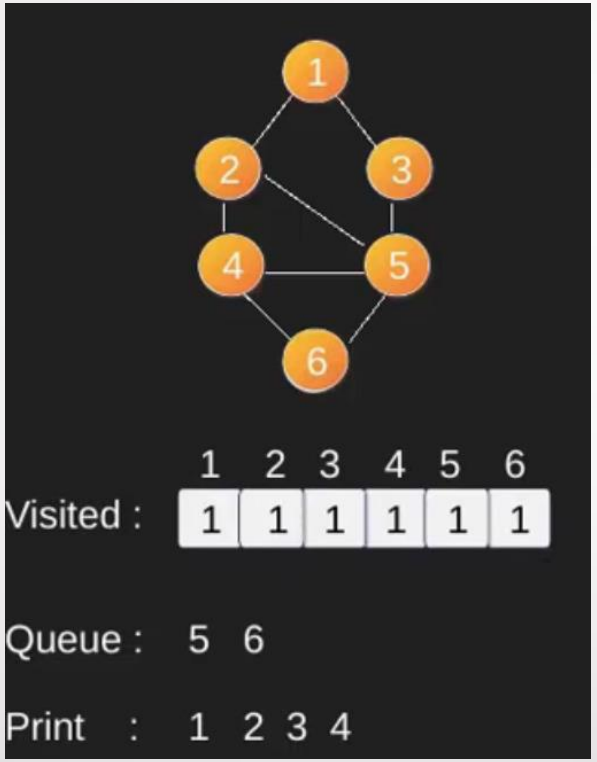


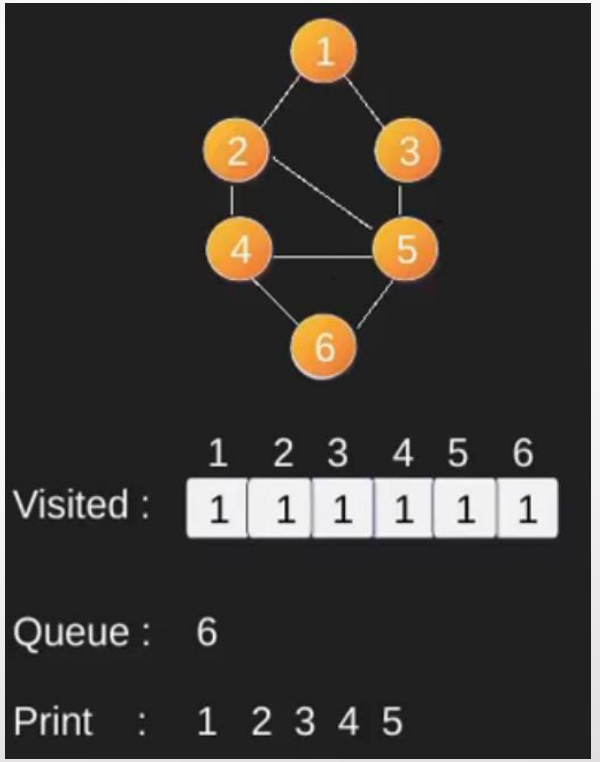


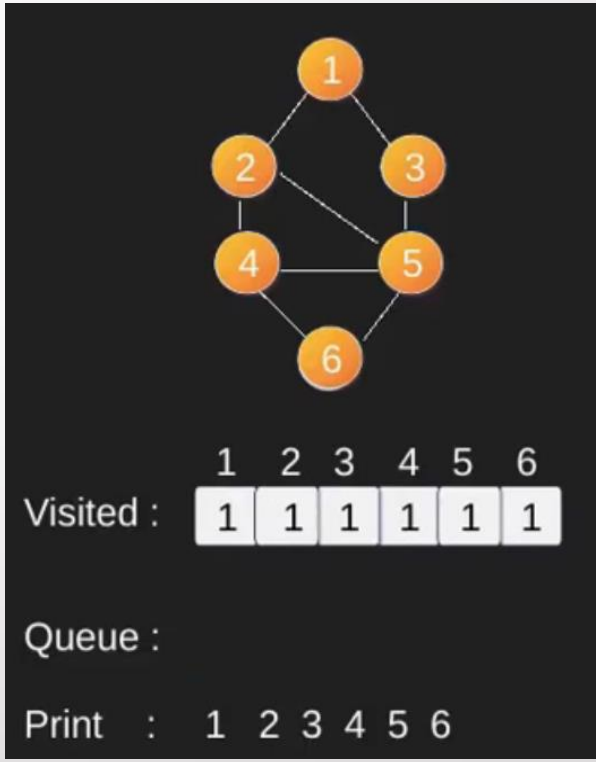
Breadth First Search (BFS)

An Example (Continued)









Implementation of BFS

BFS Implementation using Visited Array

```
void BFS(int s)
{
    System.out.println("-BFS-");
    boolean visited[]=new boolean[v];
    Queue<Integer> queue=new LinkedList<Integer>();
    visited[s]=true;
    queue.add(s);
    while(queue.size()!=0)
    {
        s=queue.poll();
        System.out.print(s+" ");
        for(Integer n:adjListArray[s])
        {
            if(!visited[n])
                {
                    visited[n]=true;
                    queue.add(n);
                }
        }
    }
    System.out.println("\n-----");
}
```

Implementation of BFS

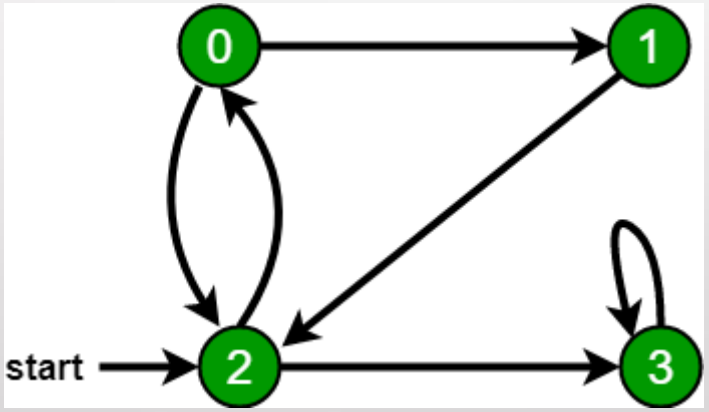
BFS Implementation using Visited Set

```
void BFS(int s)
{
    System.out.println("-BFS-");
    HashSet<Integer> visited=new HashSet<>();
    Queue<Integer> queue=new LinkedList<>();
    visited.add(s);
    queue.add(s);
    while(!queue.isEmpty())
    {
        s=queue.poll();
        System.out.print(s+" ");
        for(Integer n:g.get(s))
        {
            if(!visited.contains(n))
            {
                visited.add(n);
                queue.add(n);
            }
        }
    }
}
```

Depth First Search (DFS)

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.

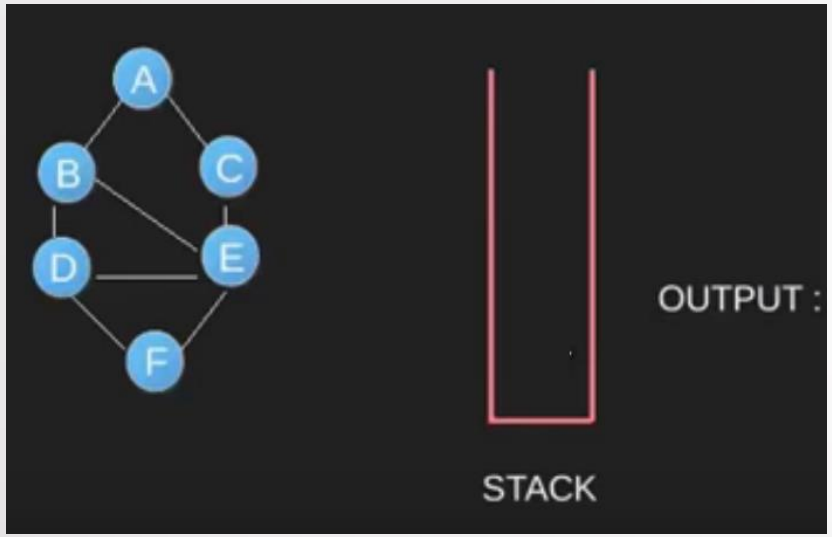
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. It will become a non-terminating process.

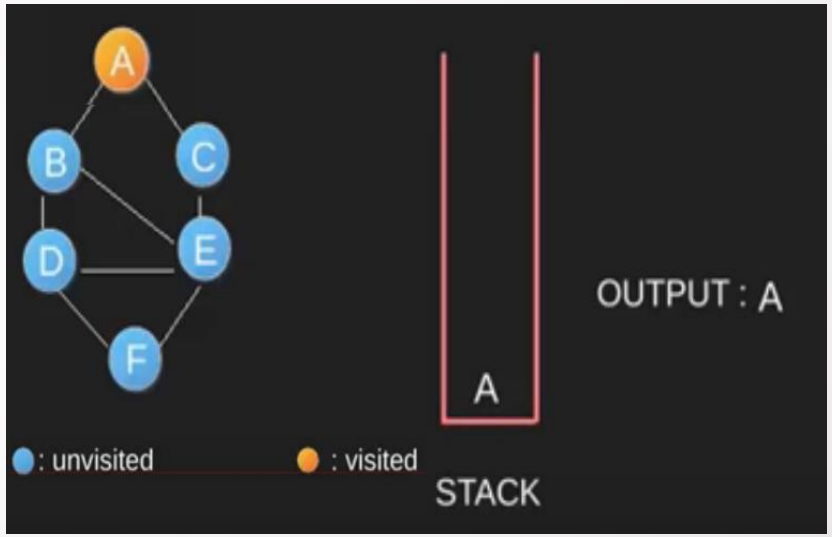


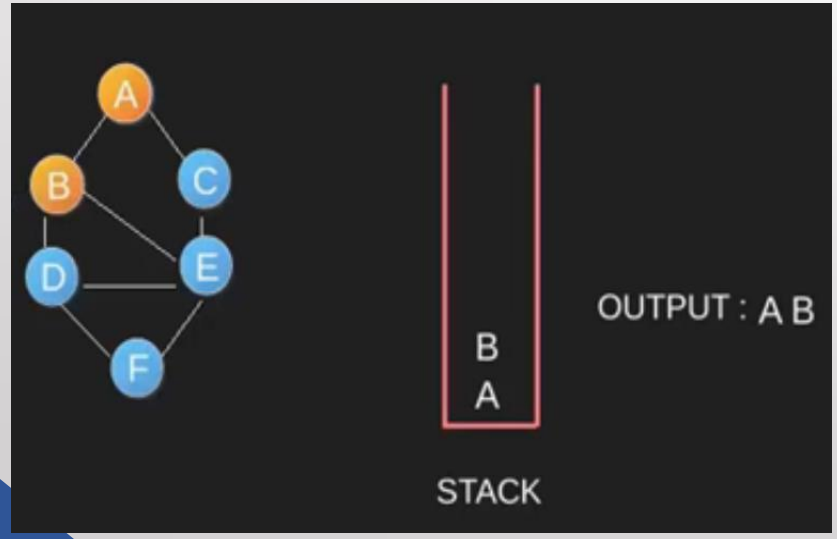
Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

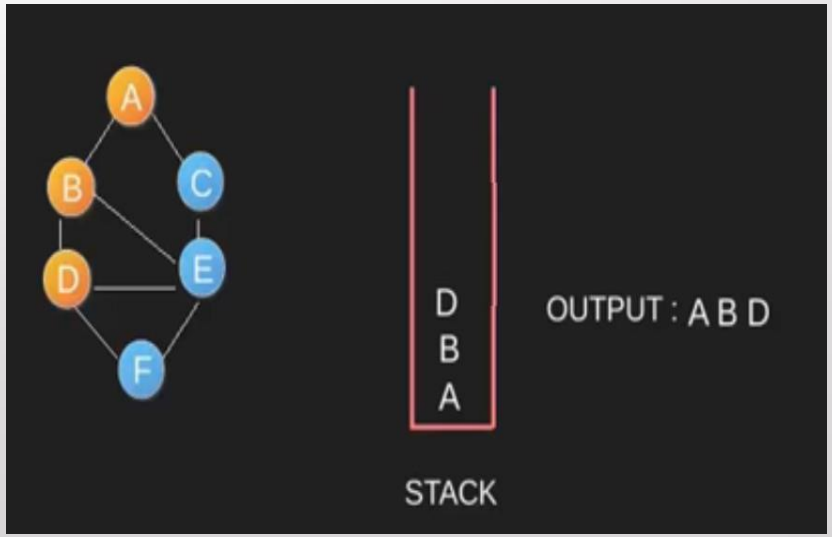
Depth First Search (DFS)

An Example



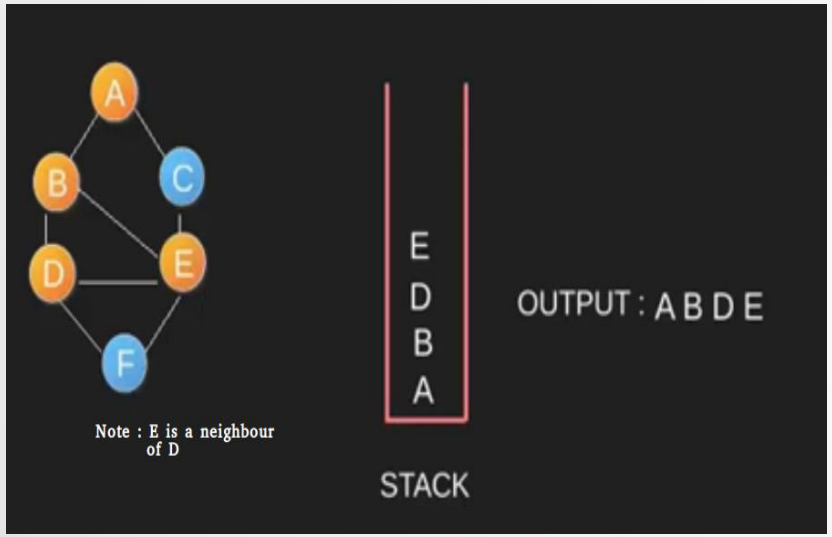


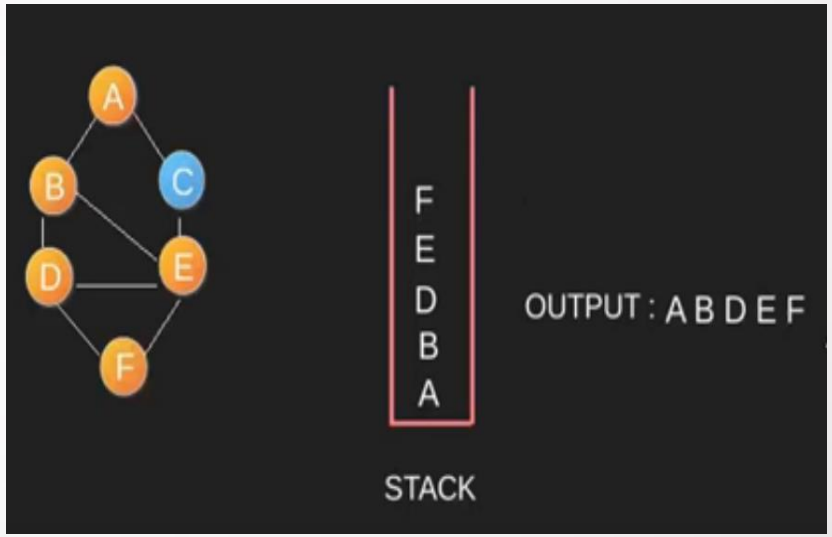


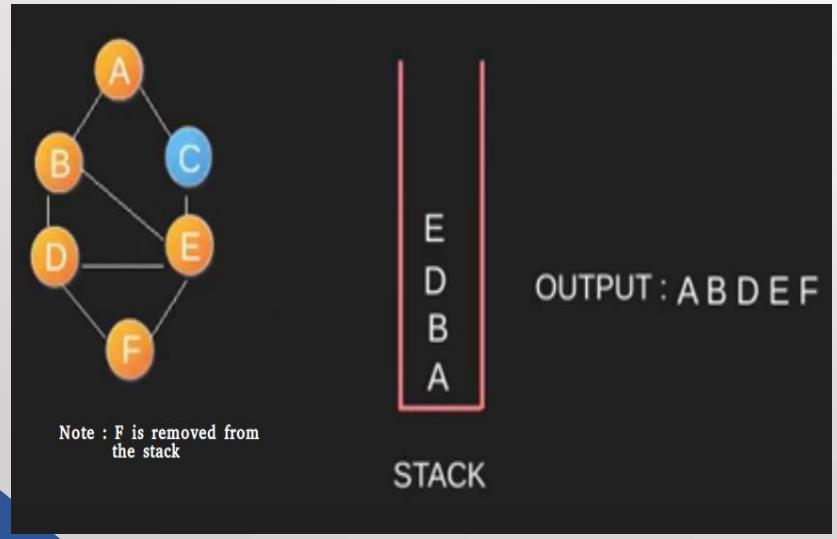


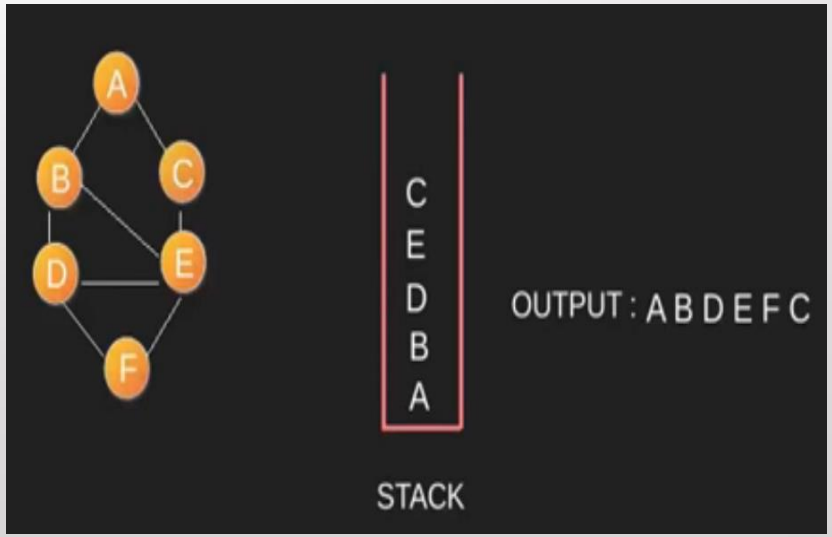
Depth First Search (DFS)

An Example (Continued)



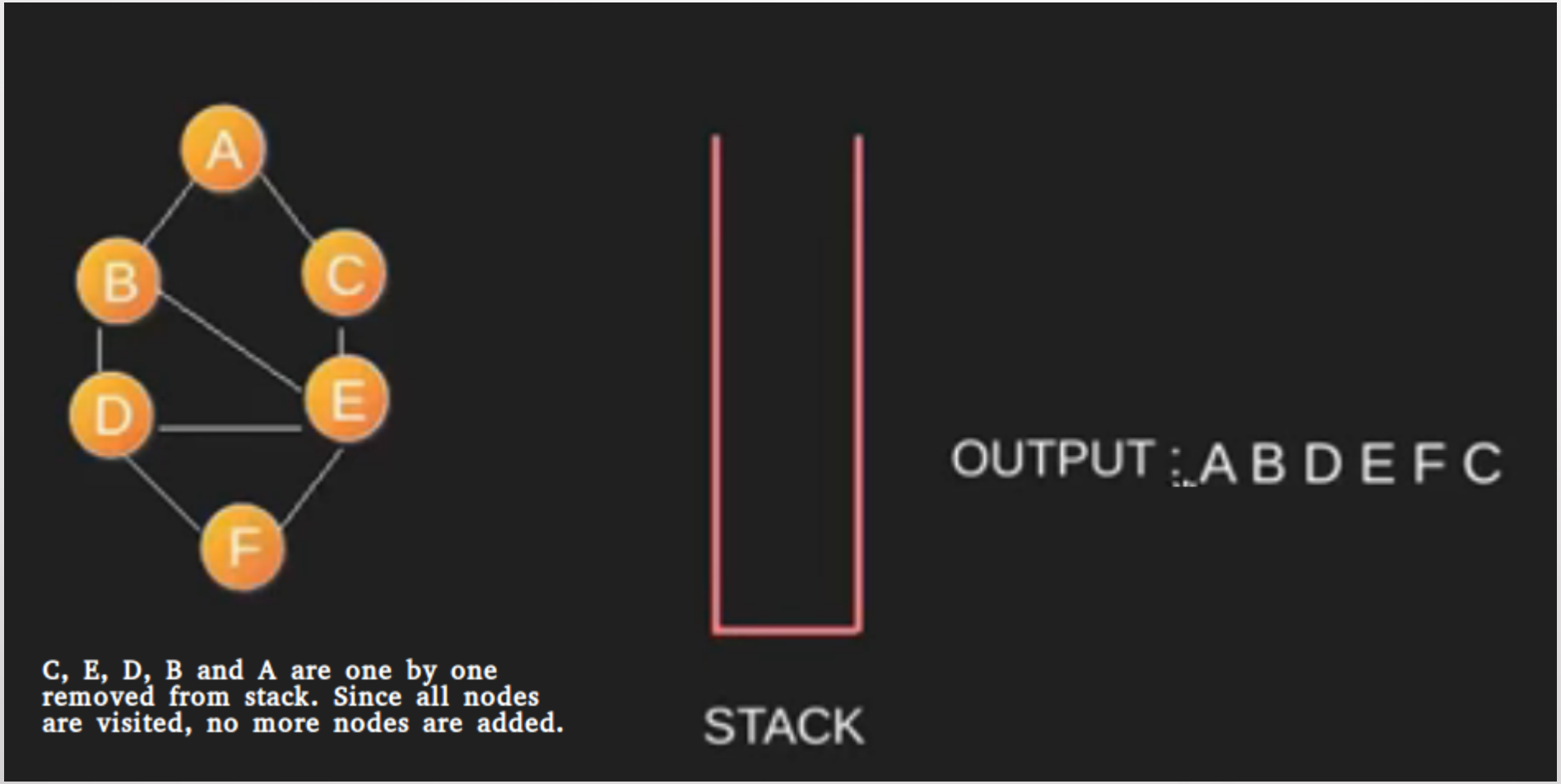






Depth First Search (DFS)

An Example (Continued)



Implementation of DFS

DFS Implementation using Visited Array

```
static void DFS(Graph graph,int s)
{   boolean visited[]=new boolean[graph.v];
    System.out.println("-DFS-");
    DFSUtil(graph,visited,0);
}
static void DFSUtil(Graph graph, boolean visited[],int s)
{
    visited[s]=true;
    System.out.print(s+" ");
    Iterator<Integer> it=graph.adlistArray[s].iterator();
    while(it.hasNext())
    {
        int n=it.next();
        if(!visited[n])
        {
            DFSUtil(graph,visited,n);
        }
    }
}
```

Implementation of DFS

DFS Implementation using Visited Set

```
void DFS()
{
    HashSet<Integer> visited= new HashSet<>();
    System.out.println("--DFS--");
    DFSUtil(visited,0);
    System.out.println();
}
void DFSUtil(HashSet<Integer> visited,int s)
{
    visited.add(s);
    System.out.print(s+" ");
    for(Integer n:g.get(s))
    {
        if(!visited.contains(n))
        {
            DFSUtil(visited,n);
        }
    }
}
```