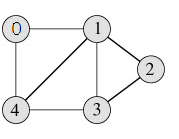
**Graph and its representations**

Graph is a data structure that consists of following two components:  
**1.** A finite set of vertices also called as nodes.  
**2.** A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

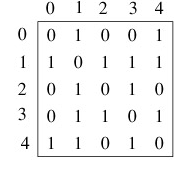
Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Following is an example undirected graph with 5 vertices.  
[](https://www.geeksforgeeks.org/wp-content/uploads/graph_representation12.png)

Following two are the most commonly used representations of graph.  
**1.** Adjacency Matrix  
**2.** Adjacency List  
There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

**Adjacency Matrix:**  
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

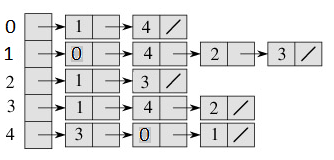
The adjacency matrix for the above example graph is:

[](https://www.geeksforgeeks.org/wp-content/uploads/adjacency_matrix_representation.png)

Adjacency Matrix Representation of the above graph

*Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

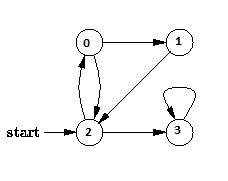
*Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.  
  
**Adjacency List:**  
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

[](https://www.geeksforgeeks.org/wp-content/uploads/adjacency_list_representation.png)

Adjacency List Representation of the above Graph

**Breadth First Traversal or BFS for a Graph**

[Breadth First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](https://www.geeksforgeeks.org/archives/2686)). 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. For simplicity, it is assumed that all vertices are reachable from the starting vertex.  
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. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

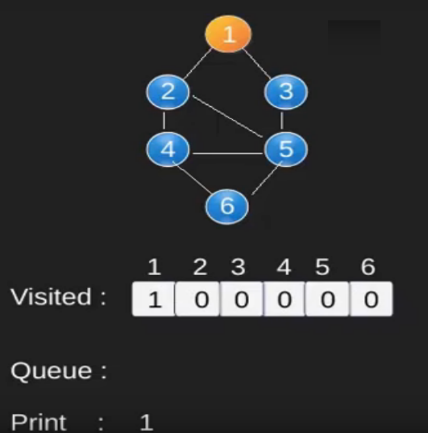
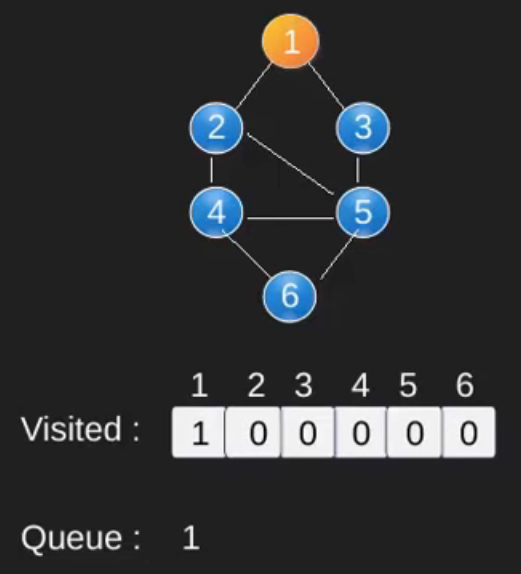
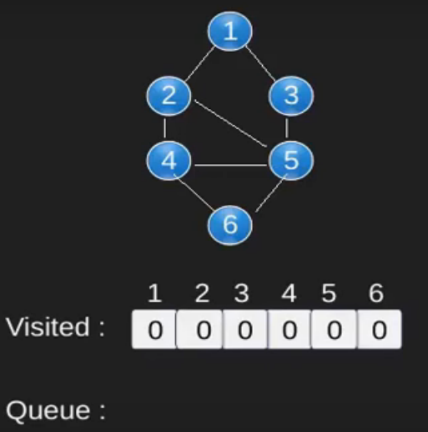
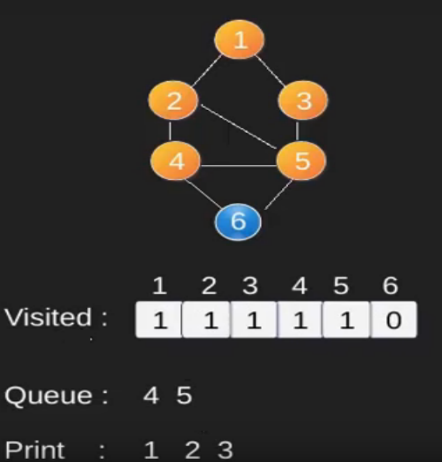
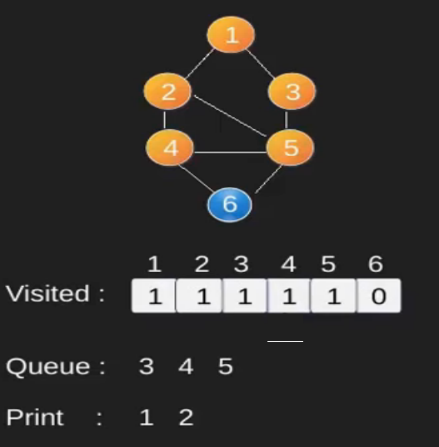
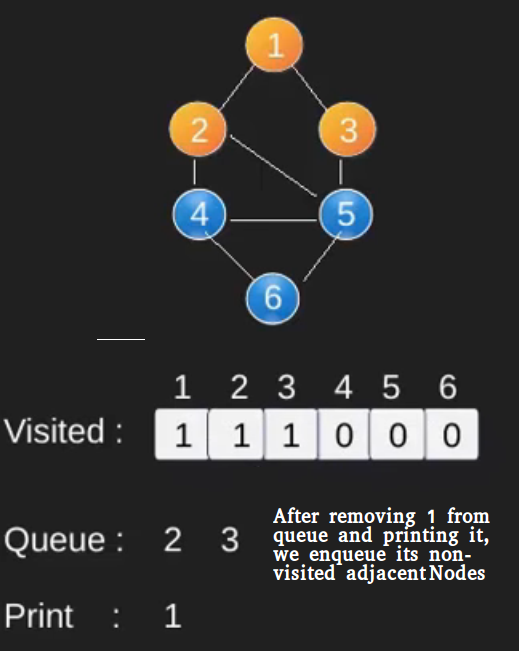
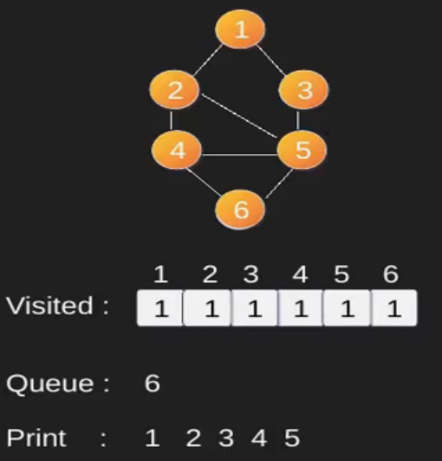
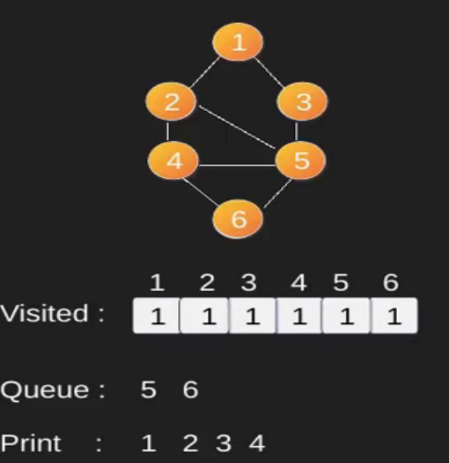
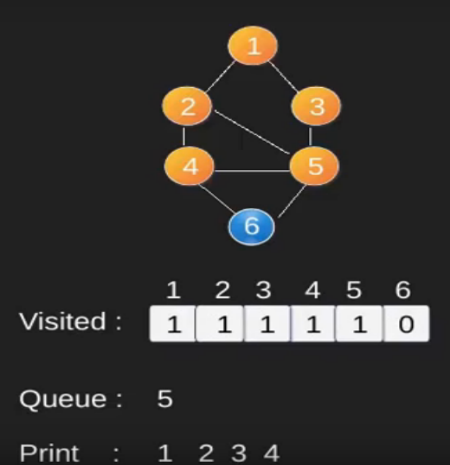
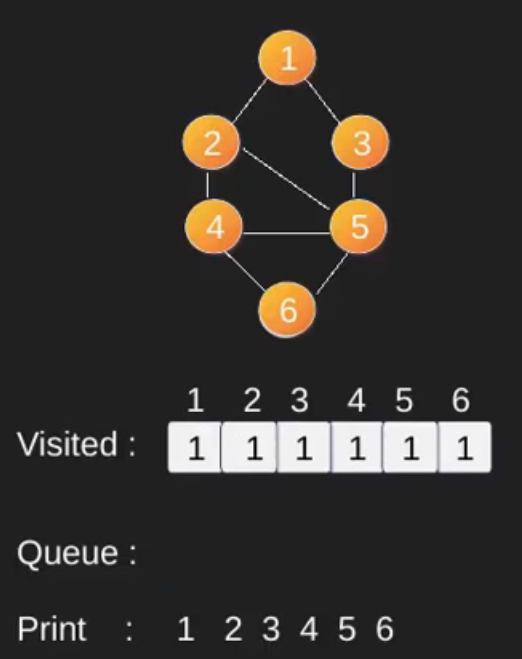
[](https://www.geeksforgeeks.org/wp-content/uploads/BFS.jpg)

|  |
| --- |
| // Program to print BFS traversal from a given  // source vertex. BFS(int s) traverses vertices  // reachable from s.  #include<iostream>  #include <list>    using namespace std;    // This class represents a directed graph using  // adjacency list representation  class Graph  {      int V;    // No. of vertices        // Pointer to an array containing adjacency      // lists      list<int> \*adj;  public:      Graph(int V);  // Constructor        // function to add an edge to graph      void addEdge(int v, int w);        // prints BFS traversal from a given source s      void BFS(int s);  };    Graph::Graph(int V)  {      this->V = V;      adj = new list<int>[V];  }    void Graph::addEdge(int v, int w)  {      adj[v].push\_back(w); // Add w to v’s list.  }    void Graph::BFS(int s)  {      // Mark all the vertices as not visited      bool \*visited = new bool[V];      for(int i = 0; i < V; i++)          visited[i] = false;        // Create a queue for BFS      list<int> queue;        // Mark the current node as visited and enqueue it      visited[s] = true;      queue.push\_back(s);        // 'i' will be used to get all adjacent      // vertices of a vertex      list<int>::iterator i;        while(!queue.empty())      {          // Dequeue a vertex from queue and print it          s = queue.front();          cout << s << " ";          queue.pop\_front();            // Get all adjacent vertices of the dequeued          // vertex s. If a adjacent has not been visited,          // then mark it visited and enqueue it          for (i = adj[s].begin(); i != adj[s].end(); ++i)          {              if (!visited[\*i])              {                  visited[\*i] = true;                  queue.push\_back(\*i);              }          }      }  }    // Driver program to test methods of graph class  int main()  {      // Create a graph given in the above diagram      Graph g(4);      g.addEdge(0, 1);      g.addEdge(0, 2);      g.addEdge(1, 2);      g.addEdge(2, 0);      g.addEdge(2, 3);      g.addEdge(3, 3);        cout << "Following is Breadth First Traversal "           << "(starting from vertex 2) \n";      g.BFS(2);        return 0;  } |

Output:

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

**Illustration :**   
  
  
  


Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

**Applications of Breadth First Traversal**

**1) Shortest Path and Minimum Spanning Tree for unweighted graph** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

**2) Peer to Peer Networks.** In Peer to Peer Networks like [BitTorrent](https://www.geeksforgeeks.org/how-bittorrent-works/), Breadth First Search is used to find all neighbor nodes.

**3) Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

**4) Social Networking Websites:** In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

**5) GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.

**6) Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

**7) In Garbage Collection:** Breadth First Search is used in copying garbage collection using [Cheney’s algorithm](http://en.wikipedia.org/wiki/Cheney%27s_algorithm). Refer [this](https://lambda.uta.edu/cse5317/notes/node48.html) and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

**8) Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

**9)** [**Ford–Fulkerson algorithm**](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).

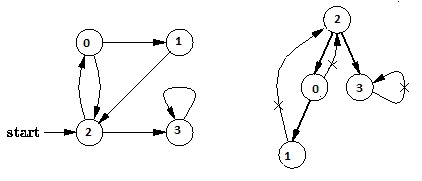
**10)** [**To test if a graph is Bipartite**](https://www.geeksforgeeks.org/bipartite-graph/) We can either use Breadth First or Depth First Traversal.

**11) Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

**12)**  [**Cycle detection in undirected graph:**](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/) In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles 0->2->0, 0->1->2->0 and 3->3, so your function must return true.

Depth First Traversal can be used to detect cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a [back edge](http://en.wikipedia.org/wiki/Depth-first_search#Output_of_a_depth-first_search) present in the graph. A back edge is an edge that is from a node to itself (selfloop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.

[](https://www.geeksforgeeks.org/wp-content/uploads/DFS.jpg)

For a disconnected graph, we get the DFS forrest as output. To detect cycle, we can check for cycle in individual trees by checking back edges.To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is back edge. We have used recStack[] array to keep track of vertices in the recursion stack.

|  |
| --- |
| // A C++ Program to detect cycle in a graph  #include<iostream>  #include <list>  #include <limits.h>    using namespace std;    class Graph  {      int V;    // No. of vertices      list<int> \*adj;    // Pointer to an array containing adjacency lists      bool isCyclicUtil(int v, bool visited[], bool \*rs);  // used by isCyclic()  public:      Graph(int V);   // Constructor      void addEdge(int v, int w);   // to add an edge to graph      bool isCyclic();    // returns true if there is a cycle in this graph  };    Graph::Graph(int V)  {      this->V = V;      adj = new list<int>[V];  }    void Graph::addEdge(int v, int w)  {      adj[v].push\_back(w); // Add w to v’s list.  }    // This function is a variation of DFSUytil() in <https://www.geeksforgeeks.org/archives/18212>  bool Graph::isCyclicUtil(int v, bool visited[], bool \*recStack)  {      if(visited[v] == false)      {          // Mark the current node as visited and part of recursion stack          visited[v] = true;          recStack[v] = true;            // Recur for all the vertices adjacent to this vertex          list<int>::iterator i;          for(i = adj[v].begin(); i != adj[v].end(); ++i)          {              if ( !visited[\*i] && isCyclicUtil(\*i, visited, recStack) )                  return true;              else if (recStack[\*i])                  return true;          }        }      recStack[v] = false;  // remove the vertex from recursion stack      return false;  }    // Returns true if the graph contains a cycle, else false.  // This function is a variation of DFS() in <https://www.geeksforgeeks.org/archives/18212>  bool Graph::isCyclic()  {      // Mark all the vertices as not visited and not part of recursion      // stack      bool \*visited = new bool[V];      bool \*recStack = new bool[V];      for(int i = 0; i < V; i++)      {          visited[i] = false;          recStack[i] = false;      }        // Call the recursive helper function to detect cycle in different      // DFS trees      for(int i = 0; i < V; i++)          if (isCyclicUtil(i, visited, recStack))              return true;        return false;  }    int main()  {      // Create a graph given in the above diagram      Graph g(4);      g.addEdge(0, 1);      g.addEdge(0, 2);      g.addEdge(1, 2);      g.addEdge(2, 0);      g.addEdge(2, 3);      g.addEdge(3, 3);        if(g.isCyclic())          cout << "Graph contains cycle";      else          cout << "Graph doesn't contain cycle";      return 0;  } |

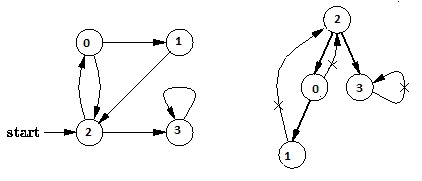
Time Complexity of this method is same as time complexity of [DFS traversal](https://www.geeksforgeeks.org/archives/18212) which is O(V+E)

**Depth First Traversal or DFS for a Graph**

[2.3](https://www.geeksforgeeks.org/easy/)

[Depth First Traversal (or Search)](http://en.wikipedia.org/wiki/Depth-first_search) for a graph is similar to [Depth First Traversal of a tree](https://www.geeksforgeeks.org/archives/618). 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.

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. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.

[](https://www.geeksforgeeks.org/wp-content/uploads/DFS.jpg)

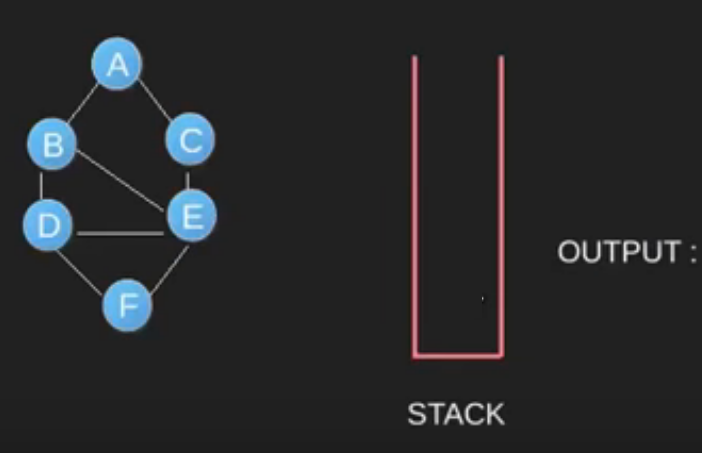
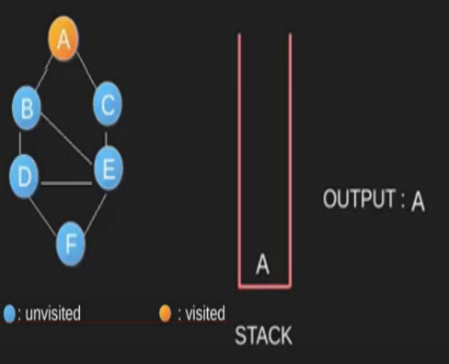
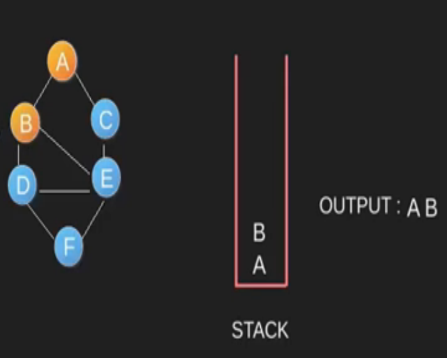
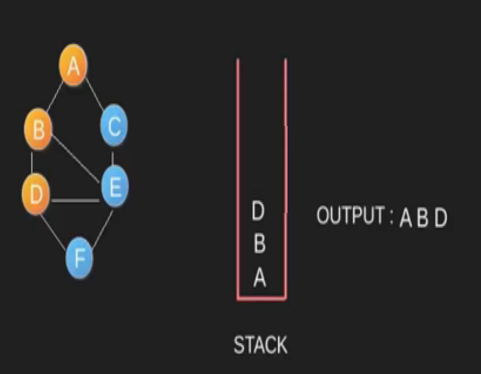
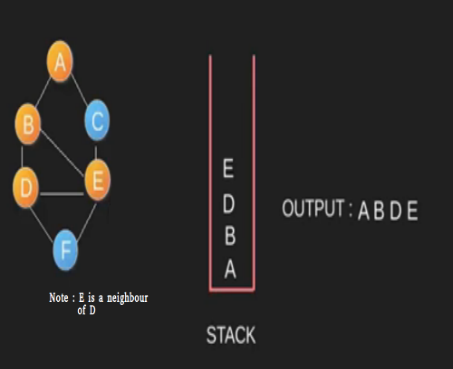
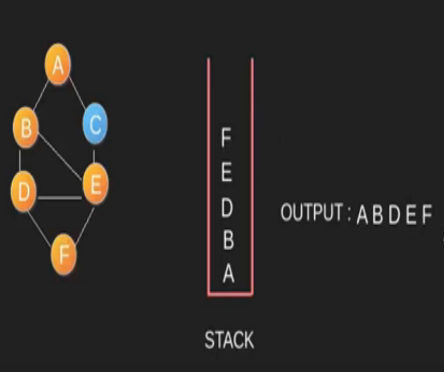
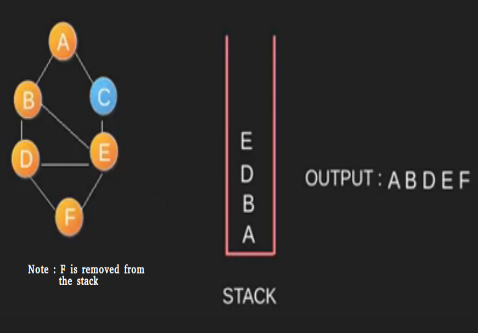
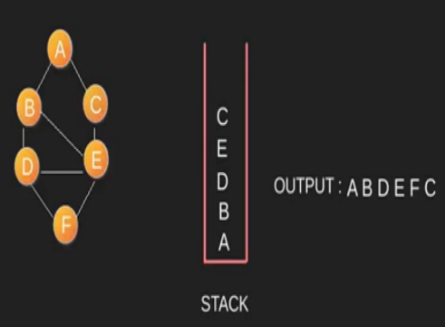
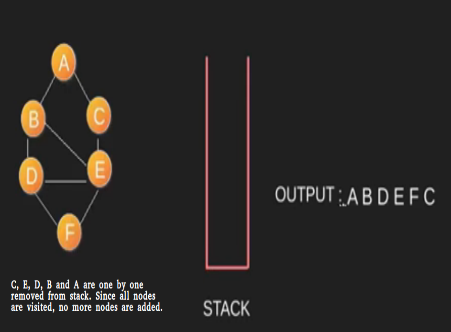
See [this post](https://www.geeksforgeeks.org/archives/11644) for all applications of Depth First Traversal.  
Following are implementations of simple Depth First Traversal. The C++ implementation uses [adjacency list representation](http://en.wikipedia.org/wiki/Adjacency_list) of graphs. [STL](http://en.wikipedia.org/wiki/Standard_Template_Library)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) is used to store lists of adjacent nodes.

|  |
| --- |
| // C++ program to print DFS traversal from  // a given vertex in a  given graph  #include<iostream>  #include<list>  using namespace std;    // Graph class represents a directed graph  // using adjacency list representation  class Graph  {      int V;    // No. of vertices        // Pointer to an array containing      // adjacency lists      list<int> \*adj;        // A recursive function used by DFS      void DFSUtil(int v, bool visited[]);  public:      Graph(int V);   // Constructor        // function to add an edge to graph      void addEdge(int v, int w);        // DFS traversal of the vertices      // reachable from v      void DFS(int v);  };    Graph::Graph(int V)  {      this->V = V;      adj = new list<int>[V];  }    void Graph::addEdge(int v, int w)  {      adj[v].push\_back(w); // Add w to v’s list.  }    void Graph::DFSUtil(int v, bool visited[])  {      // Mark the current node as visited and      // print it      visited[v] = true;      cout << v << " ";        // Recur for all the vertices adjacent      // to this vertex      list<int>::iterator i;      for (i = adj[v].begin(); i != adj[v].end(); ++i)          if (!visited[\*i])              DFSUtil(\*i, visited);  }    // DFS traversal of the vertices reachable from v.  // It uses recursive DFSUtil()  void Graph::DFS(int v)  {      // Mark all the vertices as not visited      bool \*visited = new bool[V];      for (int i = 0; i < V; i++)          visited[i] = false;        // Call the recursive helper function      // to print DFS traversal      DFSUtil(v, visited);  }    int main()  {      // Create a graph given in the above diagram      Graph g(4);      g.addEdge(0, 1);      g.addEdge(0, 2);      g.addEdge(1, 2);      g.addEdge(2, 0);      g.addEdge(2, 3);      g.addEdge(3, 3);        cout << "Following is Depth First Traversal"              " (starting from vertex 2) \n";      g.DFS(2);        return 0;  } |

Output:

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

**Illustration for an Undirected Graph :**   
[](https://www.geeksforgeeks.org/wp-content/uploads/dfs1.png)[](https://www.geeksforgeeks.org/wp-content/uploads/dfs2.png)[](https://www.geeksforgeeks.org/wp-content/uploads/dfs3.png)  
[](https://www.geeksforgeeks.org/wp-content/uploads/dfs4.png)[](https://www.geeksforgeeks.org/wp-content/uploads/dfs5.png)[](https://www.geeksforgeeks.org/wp-content/uploads/DFS6.png)  
[](https://www.geeksforgeeks.org/wp-content/uploads/DFS7.png)[](https://www.geeksforgeeks.org/wp-content/uploads/DFS8.png)[](https://www.geeksforgeeks.org/wp-content/uploads/DFS9.png)  
  
Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

**Applications of Depth First Search**.

**1)** For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

**2) Detecting cycle in a graph**   
A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See [this](http://people.csail.mit.edu/thies/6.046-web/recitation9.txt) for details)

**3) Path Finding**  
We can specialize the DFS algorithm to find a path between two given vertices u and z.  
i) Call DFS(G, u) with u as the start vertex.  
ii) Use a stack S to keep track of the path between the start vertex and the current vertex.  
iii) As soon as destination vertex z is encountered, return the path as the  
contents of the stack

**4)** [**Topological Sorting**](https://www.geeksforgeeks.org/topological-sorting/)  
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

**5) To test if a graph is** [**bipartite**](http://en.wikipedia.org/wiki/Bipartite_graph)  
We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn’t link two vertices of the same color. The first vertex in any connected component can be red or black! See [this](http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LEC/LECTUR16/NODE16.HTM) for details.

**6) Finding** [**Strongly Connected Components**](http://en.wikipedia.org/wiki/Strongly_connected_component) **of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See [this](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/strongComponent.htm) for DFS based algo for finding Strongly Connected Components) **7) Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

**Topological Sorting**

[](https://www.geeksforgeeks.org/wp-content/uploads/graph.png)Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

***Topological Sorting vs Depth First Traversal (DFS)***:  
In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting

***Algorithm to find Topological Sorting:***  
We recommend to first see implementation of DFS [here](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/). We can modify [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) to find Topological Sorting of a graph. In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

|  |
| --- |
| // A C++ program to print topological sorting of a DAG  #include<iostream>  #include <list>  #include <stack>  using namespace std;    // Class to represent a graph  class Graph  {      int V;    // No. of vertices'        // Pointer to an array containing adjacency listsList      list<int> \*adj;        // A function used by topologicalSort      void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);  public:      Graph(int V);   // Constructor         // function to add an edge to graph      void addEdge(int v, int w);        // prints a Topological Sort of the complete graph      void topologicalSort();  };    Graph::Graph(int V)  {      this->V = V;      adj = new list<int>[V];  }    void Graph::addEdge(int v, int w)  {      adj[v].push\_back(w); // Add w to v’s list.  }    // A recursive function used by topologicalSort  void Graph::topologicalSortUtil(int v, bool visited[],                                  stack<int> &Stack)  {      // Mark the current node as visited.      visited[v] = true;        // Recur for all the vertices adjacent to this vertex      list<int>::iterator i;      for (i = adj[v].begin(); i != adj[v].end(); ++i)          if (!visited[\*i])              topologicalSortUtil(\*i, visited, Stack);        // Push current vertex to stack which stores result      Stack.push(v);  }    // The function to do Topological Sort. It uses recursive  // topologicalSortUtil()  void Graph::topologicalSort()  {      stack<int> Stack;        // Mark all the vertices as not visited      bool \*visited = new bool[V];      for (int i = 0; i < V; i++)          visited[i] = false;        // Call the recursive helper function to store Topological      // Sort starting from all vertices one by one      for (int i = 0; i < V; i++)        if (visited[i] == false)          topologicalSortUtil(i, visited, Stack);        // Print contents of stack      while (Stack.empty() == false)      {          cout << Stack.top() << " ";          Stack.pop();      }  }    // Driver program to test above functions  int main()  {      // Create a graph given in the above diagram      Graph g(6);      g.addEdge(5, 2);      g.addEdge(5, 0);      g.addEdge(4, 0);      g.addEdge(4, 1);      g.addEdge(2, 3);      g.addEdge(3, 1);        cout << "Following is a Topological Sort of the given graph n";      g.topologicalSort();        return 0;  } |

Output:

Following is a Topological Sort of the given graph

5 4 2 3 1 0

**Time Complexity:** The above algorithm is simply DFS with an extra stack. So time complexity is same as DFS which is O(V+E).

**Applications:**  
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers.

**Topological sorting using DFS and BFS**

[topological sorting](https://en.wikipedia.org/wiki/Topological_sorting) can be solved using DFS and BFS in asymptotical time complexity *O*(*V*+*E*)

The problem is int[] findOrder(int numCourses, int[][] prerequisites). For (the official) example, one answer to 4, [[1,0],[2,0],[3,1],[3,2]] is [0, 1, 2, 3]. We should pay attention that we need to construct the adjacency list based on the provided edges. Moreover, we need to rememeber to **check the cycle**.

I really prefer BFS way. It’s really easy to remember: always add the vertices with indegree 0 to the queue. After poping out a vertex from the queue, decrease the indegrees of its neighbors.

On the other hand, DFS tries to reach out the last vertex by going deep, and add the last vertex into the stack since it is the last one after sorting. In the meanwhile we need to record whether the vertex is in current DFS path (int[] inPath) and whether the vertex has been visited / sorted (int[] visited) for every vertex.

The BFS / breadth-first search / Kahn’s algorithm way:

public class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

Map<Integer, Set<Integer>> g = new HashMap<>();

int[] inDegrees = new int[numCourses];

buildGraph(numCourses, prerequisites, g, inDegrees);

int[] res = new int[numCourses];

int sortedLen = 0;

Queue<Integer> q = new ArrayDeque<>();

for (int i = 0; i < numCourses; ++i) {

if (inDegrees[i] == 0) {

q.add(i);

}

}

while (q.size() > 0) {

int curr = q.remove();

for (int neigh: g.get(curr)) {

inDegrees[neigh] -= 1;

if (inDegrees[neigh] == 0) {

q.add(neigh);

}

}

res[sortedLen] = curr;

sortedLen += 1;

}

if (sortedLen != numCourses) {

return new int[]{};

}

return res;

}

private void buildGraph(int numCourses, int[][] prerequisites,

Map<Integer, Set<Integer>> g, int[] inDegrees) {

for (int i = 0; i < numCourses; ++i) {

g.put(i, new HashSet<Integer>());

}

for (int[] pre: prerequisites) {

if (g.get(pre[1]).add(pre[0])) {

inDegrees[pre[0]] += 1;

}

}

}

}

The DFS / depth-first search way:

public class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

Map<Integer, Set<Integer>> g = buildGraph(numCourses, prerequisites);

boolean[] inPath = new boolean[numCourses];

boolean[] visited = new boolean[numCourses];

Deque<Integer> stack = new ArrayDeque<>();

for (int i = 0; i < numCourses; ++i) {

if (!visited[i] && !hasOrder(g, stack, inPath, visited, i)) {

return new int[]{};

}

}

int[] res = new int[numCourses];

for (int i = 0; i < numCourses; ++i) {

res[i] = stack.removeFirst();

}

return res;

}

private boolean hasOrder(Map<Integer, Set<Integer>> g,

Deque<Integer> stack, boolean[] inPath, boolean[] visited, int i) {

if (visited[i]) {

return true;

}

visited[i] = true;

inPath[i] = true;

for (int neigh: g.get(i)) {

if (inPath[neigh] || !hasOrder(g, stack, inPath, visited, neigh)) {

return false;

}

}

inPath[i] = false;

stack.addFirst(i);

return true;

}

private Map<Integer, Set<Integer>> buildGraph(int numCourses, int[][] prerequisites) {

Map<Integer, Set<Integer>> g = new HashMap<>();

for (int i = 0; i < numCourses; ++i) {

g.put(i, new HashSet<Integer>());

}

for (int[] dep: prerequisites) {

g.get(dep[1]).add(dep[0]);

}

//System.out.println(g);

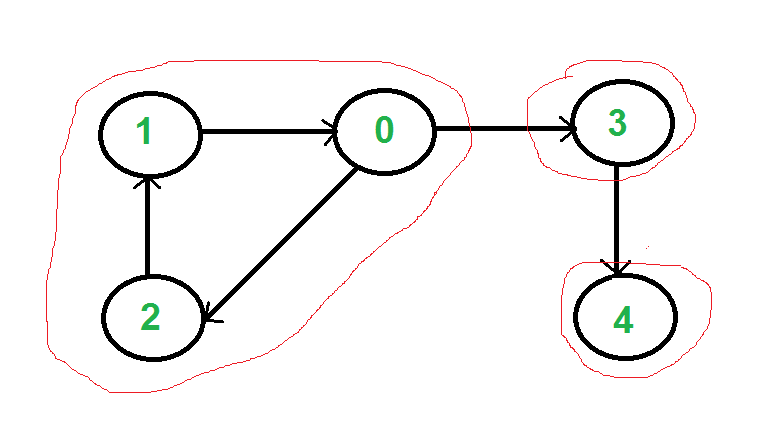
return g;

}

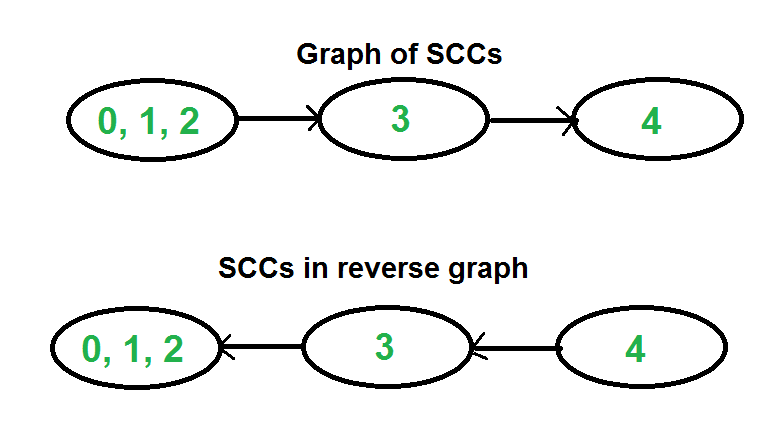
}

**Strongly Connected Components**

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.

[](https://www.geeksforgeeks.org/wp-content/uploads/SCC.png)

We can find all strongly connected components in O(V+E) time using [Kosaraju’s algorithm](http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm). Following is detailed Kosaraju’s algorithm.  
**1)** Create an empty stack ‘S’ and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.  
**2)** Reverse directions of all arcs to obtain the transpose graph.  
**3)** One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘v’. Take v as source and do DFS (call [DFSUtil(v)](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

**How does this work?**  
The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC), will always be greater than finish time of vertices in the other SCC (See [this](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/strongComponent.htm) for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn’t guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.  
In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.  
[](https://www.geeksforgeeks.org/wp-content/uploads/SCCGraph2.png)

**Time Complexity:** The above algorithm calls DFS, fins reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simple traverse all adjacency lists.

The above algorithm is asymptotically best algorithm, but there are other algorithms like [Tarjan’s algorithm](http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm) and [path-based](http://en.wikipedia.org/wiki/Path-based_strong_component_algorithm) which have same time complexity but find SCCs using single DFS. The Tarjan’s algorithm is discussed in the following post.

**Applications:**  
SCC algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph.  
In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.

|  |
| --- |
| // C++ Implementation of Kosaraju's algorithm to print all SCCs  class Graph  {      int V;    // No. of vertices      list<int> \*adj;    // An array of adjacency lists        // Fills Stack with vertices (in increasing order of finishing      // times). The top element of stack has the maximum finishing      // time      void fillOrder(int v, bool visited[], stack<int> &Stack);        // A recursive function to print DFS starting from v      void DFSUtil(int v, bool visited[]);  public:      Graph(int V);      void addEdge(int v, int w);        // The main function that finds and prints strongly connected      // components      void printSCCs();        // Function that returns reverse (or transpose) of this graph      Graph getTranspose();  };    Graph::Graph(int V)  {      this->V = V;      adj = new list<int>[V];  }    // A recursive function to print DFS starting from v  void Graph::DFSUtil(int v, bool visited[])  {      // Mark the current node as visited and print it      visited[v] = true;      cout << v << " ";        // Recur for all the vertices adjacent to this vertex      list<int>::iterator i;      for (i = adj[v].begin(); i != adj[v].end(); ++i)          if (!visited[\*i])              DFSUtil(\*i, visited);  }    Graph Graph::getTranspose()  {      Graph g(V);      for (int v = 0; v < V; v++)      {          // Recur for all the vertices adjacent to this vertex          list<int>::iterator i;          for(i = adj[v].begin(); i != adj[v].end(); ++i)          {              g.adj[\*i].push\_back(v);          }      }      return g;  }    void Graph::addEdge(int v, int w)  {      adj[v].push\_back(w); // Add w to v’s list.  }    void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)  {      // Mark the current node as visited and print it      visited[v] = true;        // Recur for all the vertices adjacent to this vertex      list<int>::iterator i;      for(i = adj[v].begin(); i != adj[v].end(); ++i)          if(!visited[\*i])              fillOrder(\*i, visited, Stack);        // All vertices reachable from v are processed by now, push v      Stack.push(v);  }    // The main function that finds and prints all strongly connected  // components  void Graph::printSCCs()  {      stack<int> Stack;        // Mark all the vertices as not visited (For first DFS)      bool \*visited = new bool[V];      for(int i = 0; i < V; i++)          visited[i] = false;        // Fill vertices in stack according to their finishing times      for(int i = 0; i < V; i++)          if(visited[i] == false)              fillOrder(i, visited, Stack);        // Create a reversed graph      Graph gr = getTranspose();        // Mark all the vertices as not visited (For second DFS)      for(int i = 0; i < V; i++)          visited[i] = false;        // Now process all vertices in order defined by Stack      while (Stack.empty() == false)      {          // Pop a vertex from stack          int v = Stack.top();          Stack.pop();            // Print Strongly connected component of the popped vertex          if (visited[v] == false)          {              gr.DFSUtil(v, visited);              cout << endl;          }      }  }    // Driver program to test above functions  int main()  {      // Create a graph given in the above diagram      Graph g(5);      g.addEdge(1, 0);      g.addEdge(0, 2);      g.addEdge(2, 1);      g.addEdge(0, 3);      g.addEdge(3, 4);        cout << "Following are strongly connected components in "              "given graph \n";      g.printSCCs();      return 0;  } |