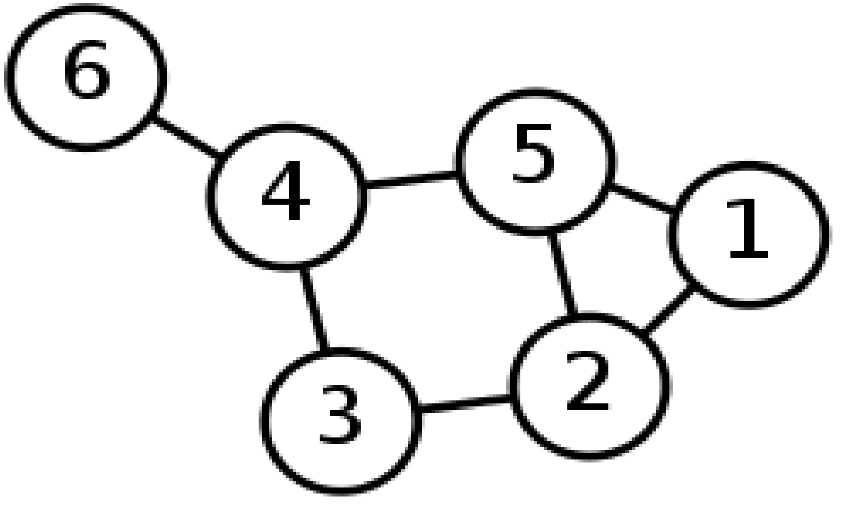
GRAPHS

INTRODUCTION

- A graph G = (V, E) is a set of vertices (or nodes) V and a set of edges E, assumed finite i.e. |V| = n and |E| = m.
- Example



INTRODUCTION

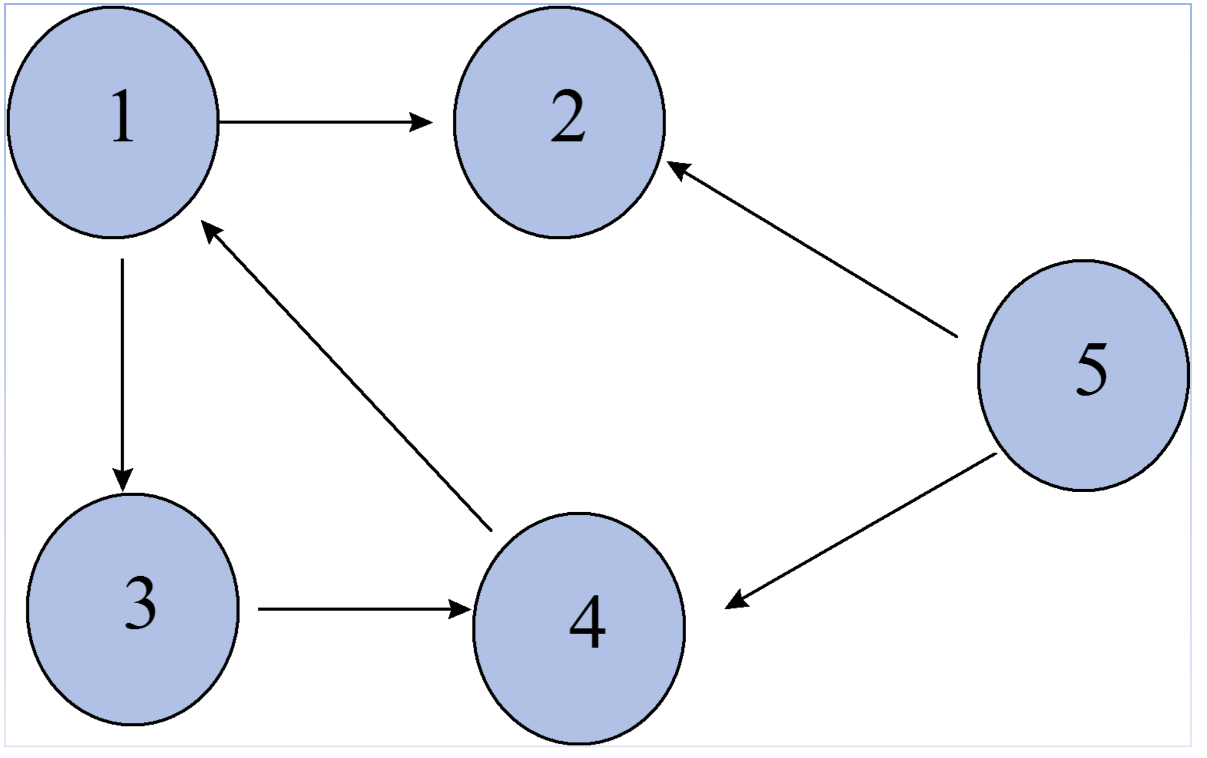
- $V(G) = \{1, 2, 3, 4, 5, 6\}$

- $E(G) = \{(1,2), (1,5), (2,3), (2,5), (3,4), (4,5), (4,6)\}$

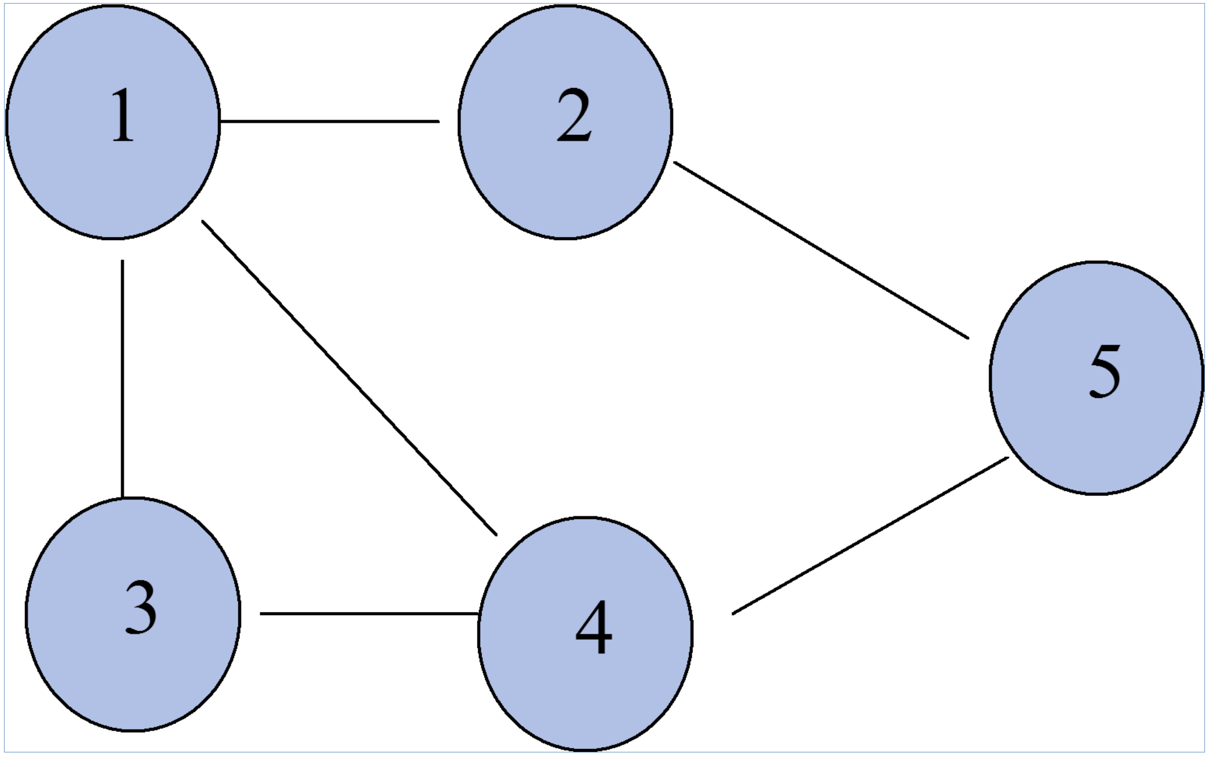
TYPES OF GRAPHS

- Two types of graphs:
  - Directed graphs: G=(V,E) where E is composed of ordered pairs of vertices; i.e. the edges have direction and point from one vertex to another.
  - Undirected graphs: G=(V,E) where E is composed of unordered pairs of vertices; i.e. the edges are bidirectional.

DIRECTED GRAPH



UNDIRECTED GRAPH

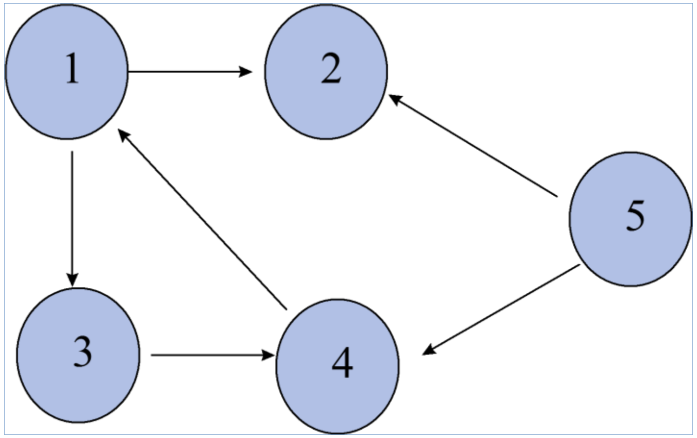


IMPLEMENTING A GRAPH

- Implement a graph in two ways:
  - Adjacency List
  - Adjacency Matrix

ADJACENCY LIST

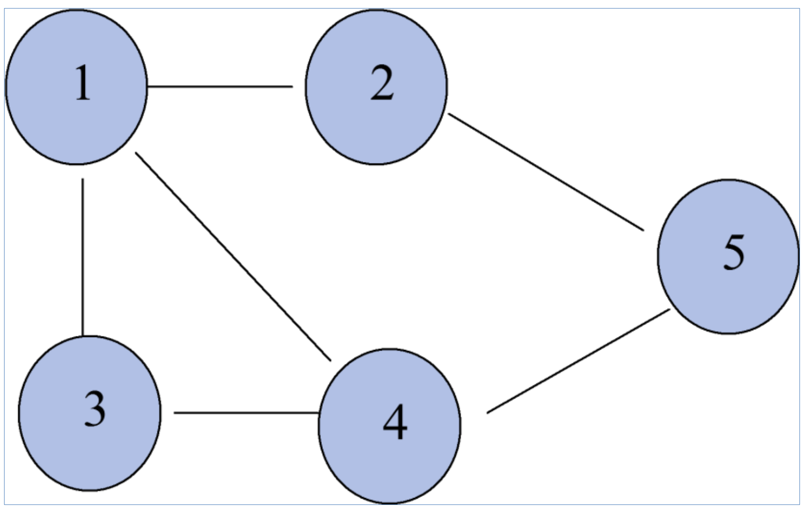
- Directed Graph



1 → 2 → 3
2 →
3 → 4
4 → 1
5 → 2 → 4

ADJACENCY LIST

- Undirected Graph



1 → 2 → 3 → 4
2 → 1 → 5
3 → 4 → 1
4 → 1 → 3 → 5
5 → 2 → 4

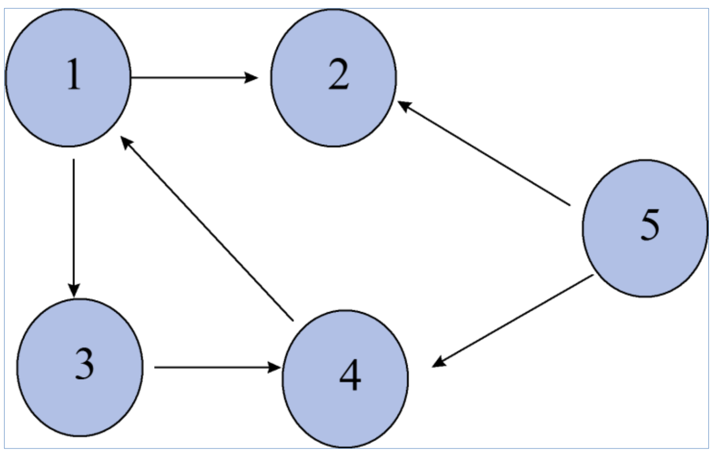
ADJACENCY LIST

- The sum of the lengths of the adjacency lists is $2|E|$ in an undirected graph, and $|E|$ in a directed graph.

- The amount of memory to store the array for the adjacency list is $O(V+E)$.

ADJACENCY MATRIX

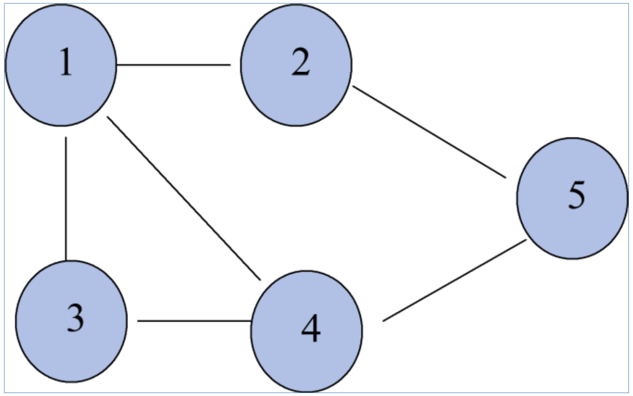
- Directed Graph



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

ADJACENCY MATRIX

- Undirected Graph



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

ADJACENCY MATRIX

- The matrix always uses $\Theta(v^2)$ memory.
- Usually easier to implement and perform lookup than an adjacency list.

ADJACENCY MATRIX VS. LIST?

- Sparse graph: very few edges.
- Dense graph: lots of edges. Up to O(v^2) edges if fully connected.
- The adjacency matrix is a good way to represent a weighted graph. In a weighted graph, the edges have weights associated with them.

PATHS

- Path in a graph G = (V, E) to be a sequence P of nodes V1, V2, ... Vk-1, Vk with the property that each consecutive pair Vi, Vi+1 is joined by an edge in G.

- P is often called a path from V1 to Vk, or a V1-Vk path.

CYCLES

- A cycle is a path that begins and ends on the same vertex.

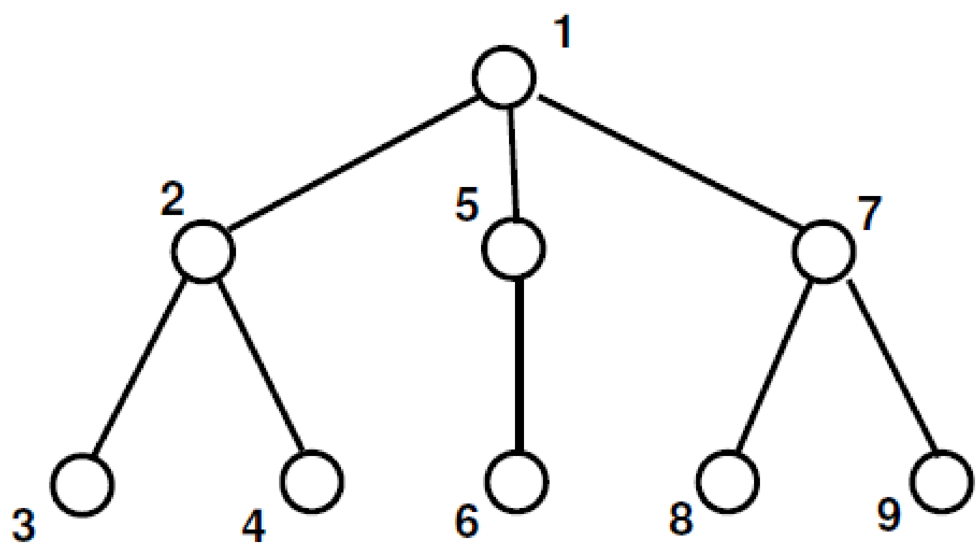
- We call a sequence of nodes a cycle if V1 = Vk. In other words, the sequence “cycles back" to where it began.

- A path is called simple if all its vertices are distinct.

- A path is called a simple cycle if V1, V2, . . . Vk are all distinct, and V1 = Vk.

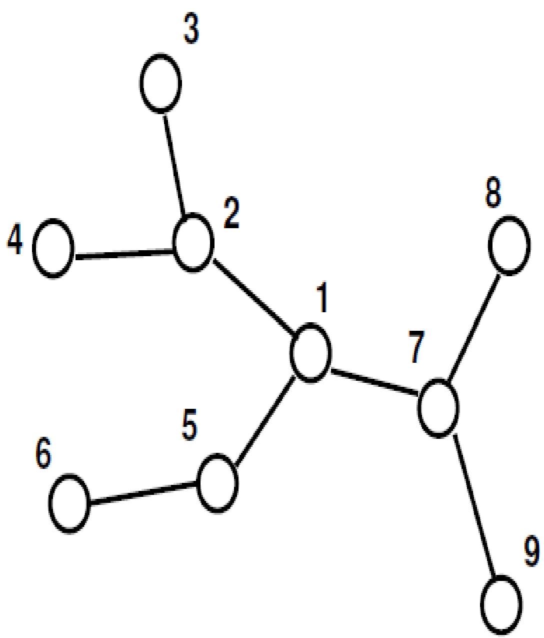
TREES

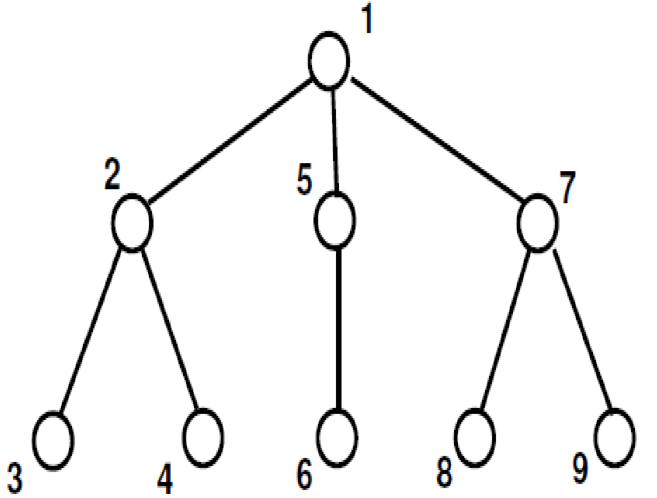
- We say that a graph is connected if for every pair of nodes u and v, there is a path from u to v.
- A tree is a connected graph with no cycles.



TREES

- Same Trees or Different Trees?





APPLICATIONS OF GRAPHS

- Social Network Graphs
- Transportation Networks.
- Web links.
- Robot Planning
- Etc

ALGORITHMS ON GRAPHS

- Searching Graphs
- Detecting Cycles in Graphs
- Shortest Path algorithms

SEARCHING A GRAPH

- Search:
  - The goal is to methodically explore every vertex and every edge; perhaps to do some processing on each.

- For the most part in our algorithms we will assume an adjacency-list representation of the input graph.

GRAPH CONNECTIVITY

- node-to-node connectivity.
- Suppose we are given a graph $G = (V, E)$, and two particular nodes s and t.
- We'd like to find an efficient algorithm that answers the question: Is there a path from s to t in G?

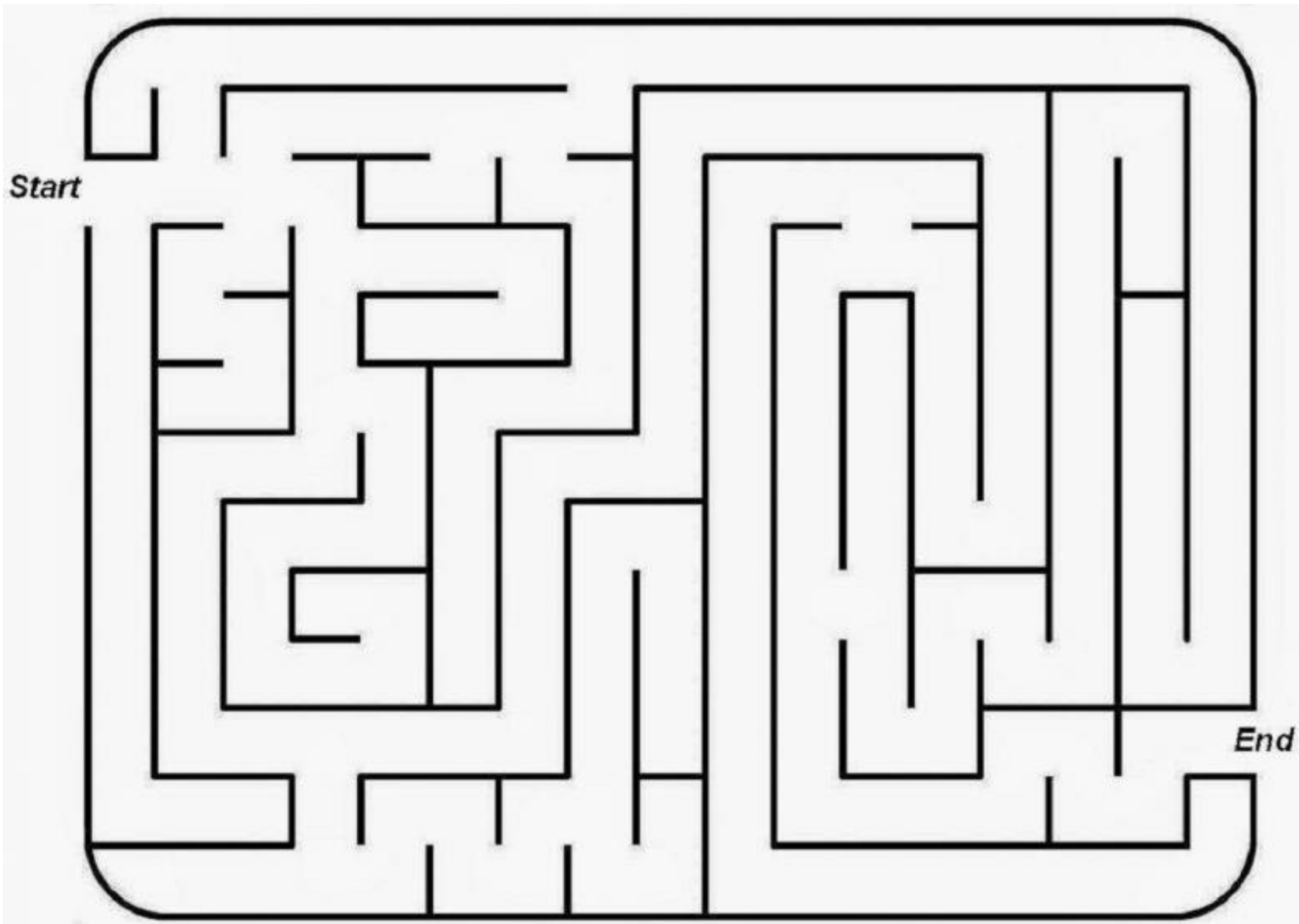
GRAPH CONNECTIVITY

- Easy to find the connectivity for small graphs.

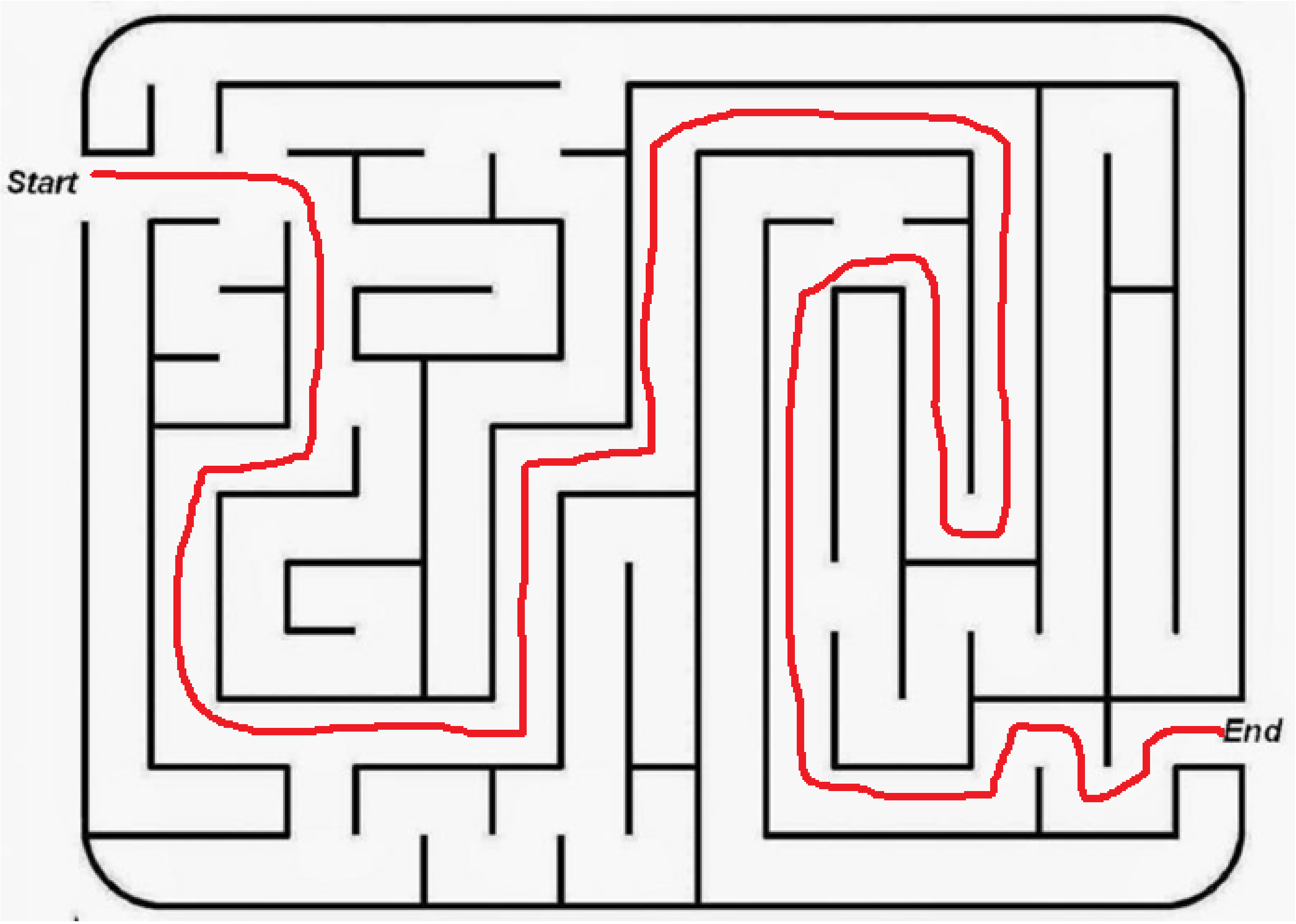
- Tough to find for very large graphs.

- Design an efficient algorithm for finding the connectivity.

GRAPH CONNECTIVITY



GRAPH CONNECTIVITY



GRAPH CONNECTIVITY

- The basic idea in searching for a path is to “explore" the graph G starting from s, maintaining a set R consisting of all nodes that s can reach.
- Initially, we set R = {s}.
- If at any point in time, there is an edge (u, v) where u Є R and v ∉ R, then we claim it is safe to add v to R.

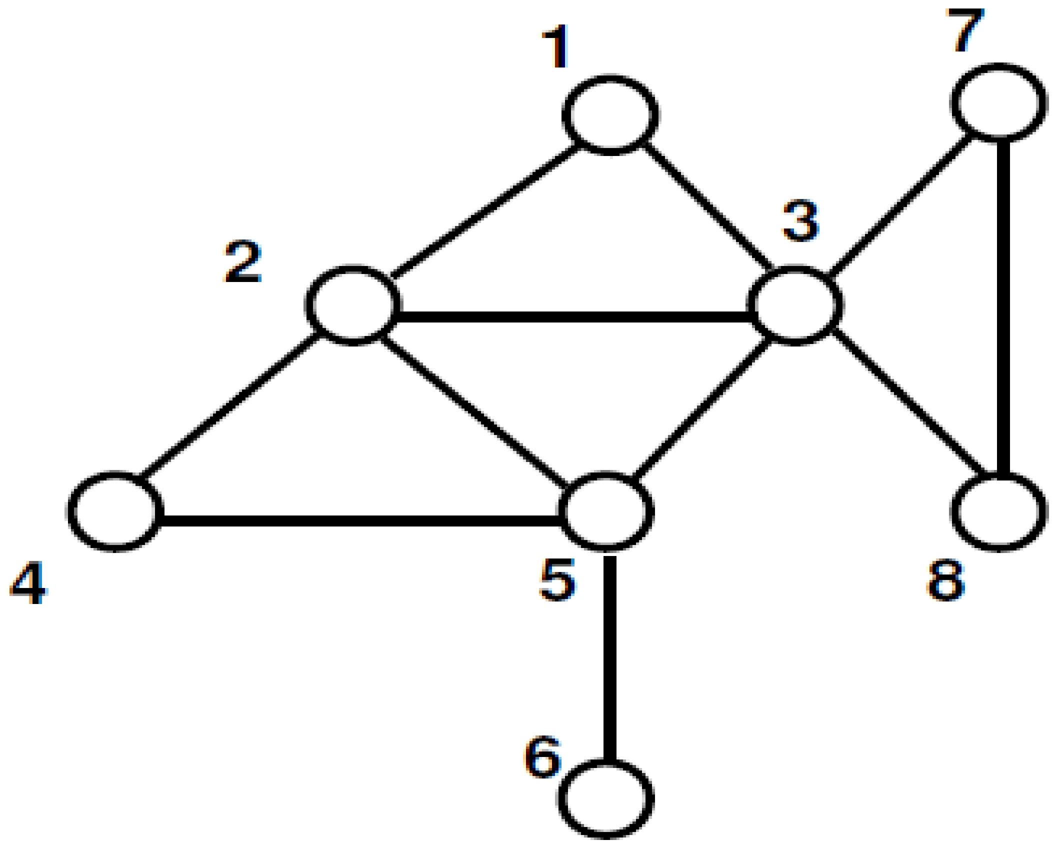
GRAPH CONNECTIVITY

- Indeed, if there is a path P from s to u, then there is a path from s to v obtained by first following P and then following the edge (u,v).
- Suppose we continue this of growing the set R until there are no more edges leading out of R.

GRAPH CONNECTIVITY

R will consist of nodes to which s has a path.
Initially $R = \{s\}$.
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R.
Endwhile

GRAPH CONNECTIVITY



GRAPH TRAVERSALS

- Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
- While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.
- The order in which the vertices are visited are important and may depend upon the algorithm or the problem.

GRAPH TRAVERSALS

- During a traversal, it is important that you track which vertices have been visited.
- The most common way of tracking vertices is to mark them.
- 2 Types
  - Breadth First Search (BFS)
  - Depth First Search (DFS)

BREADTH-FIRST SEARCH

BREADTH-FIRST SEARCH

- BFS is a traversing algorithm where you should start traversing from a source node and traverse the graph layer-wise thus exploring the neighbor nodes.

- Then move towards the next-level neighbor nodes.

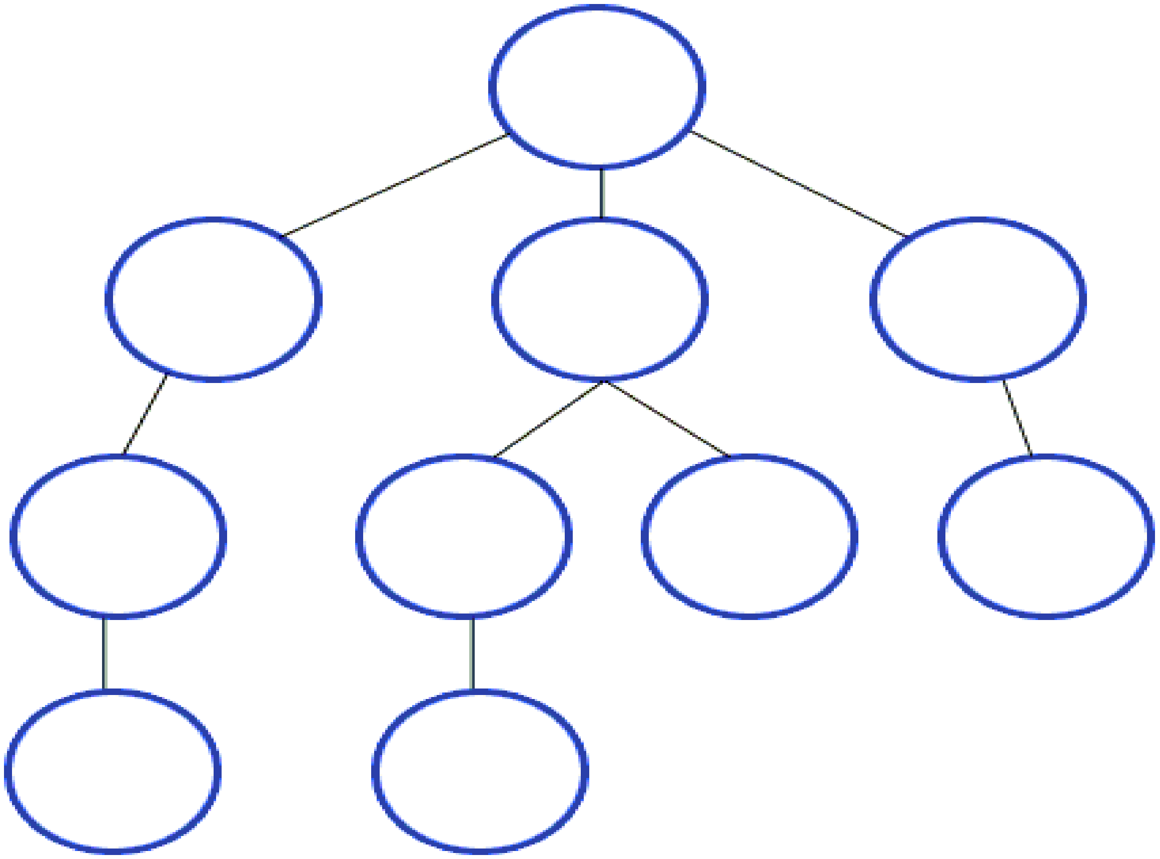
BREADTH-FIRST SEARCH

- As the name BFS suggests, you are required to traverse the graph breadth wise as follows:

1. First move horizontally and visit all the nodes of the current layer

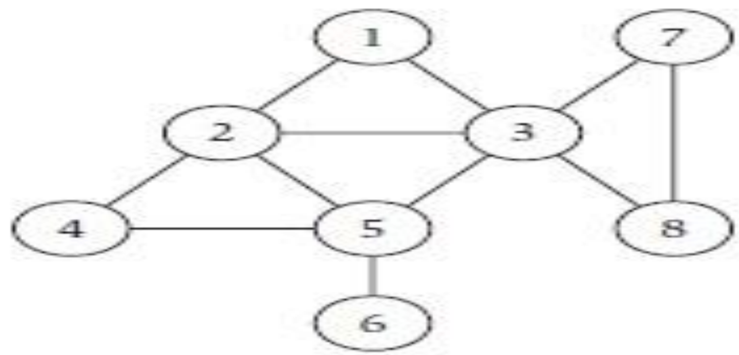
2. Move to the next layer

BREADTH-FIRST SEARCH

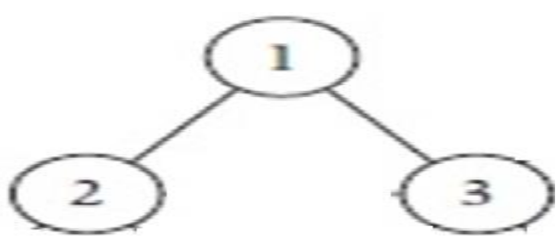


WORKING OF BREADTH-FIRST SEARCH

- In this algorithm, given a graph G= (V, E) and a source vertex S, we explore from S in all possible directions, adding nodes one “layer” at a time.
- **Example 1**

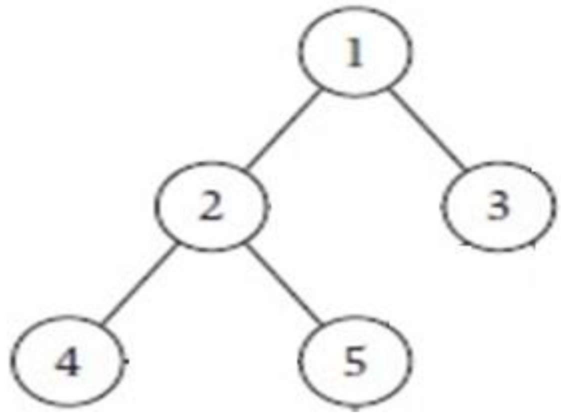


- Consider the source vertex S=1. Add to BFS tree T @ Layer 0.
- We discover nodes {2,3} from node 1 check if {2,3} are in T



Layer L1

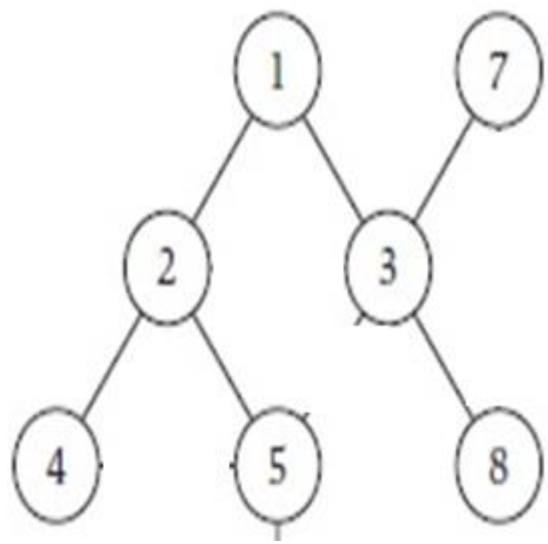
- We now discover nodes {3,4,5} from node 2 in L1
  - 3 is already present in T
  - Add 4,5 to T



Layer 2

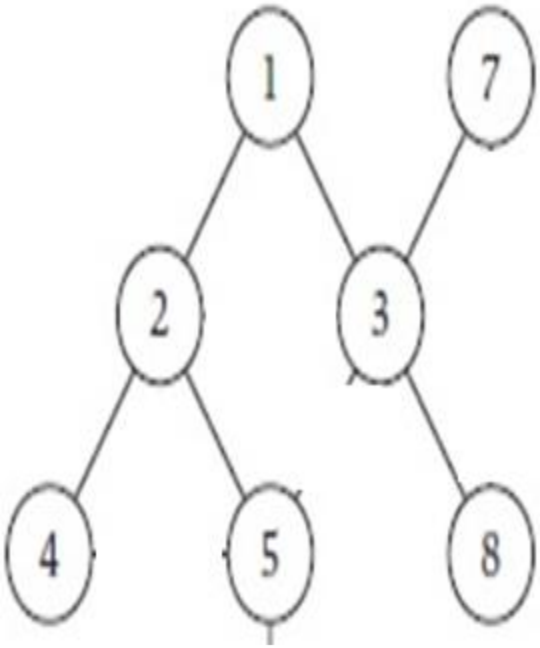
- We now discover the nodes {2,5,7,8} from the node 3 in L1.
  - 2 is already present in T
  - 5 is already present in T
  - Add 7,8 to T

L2



L2

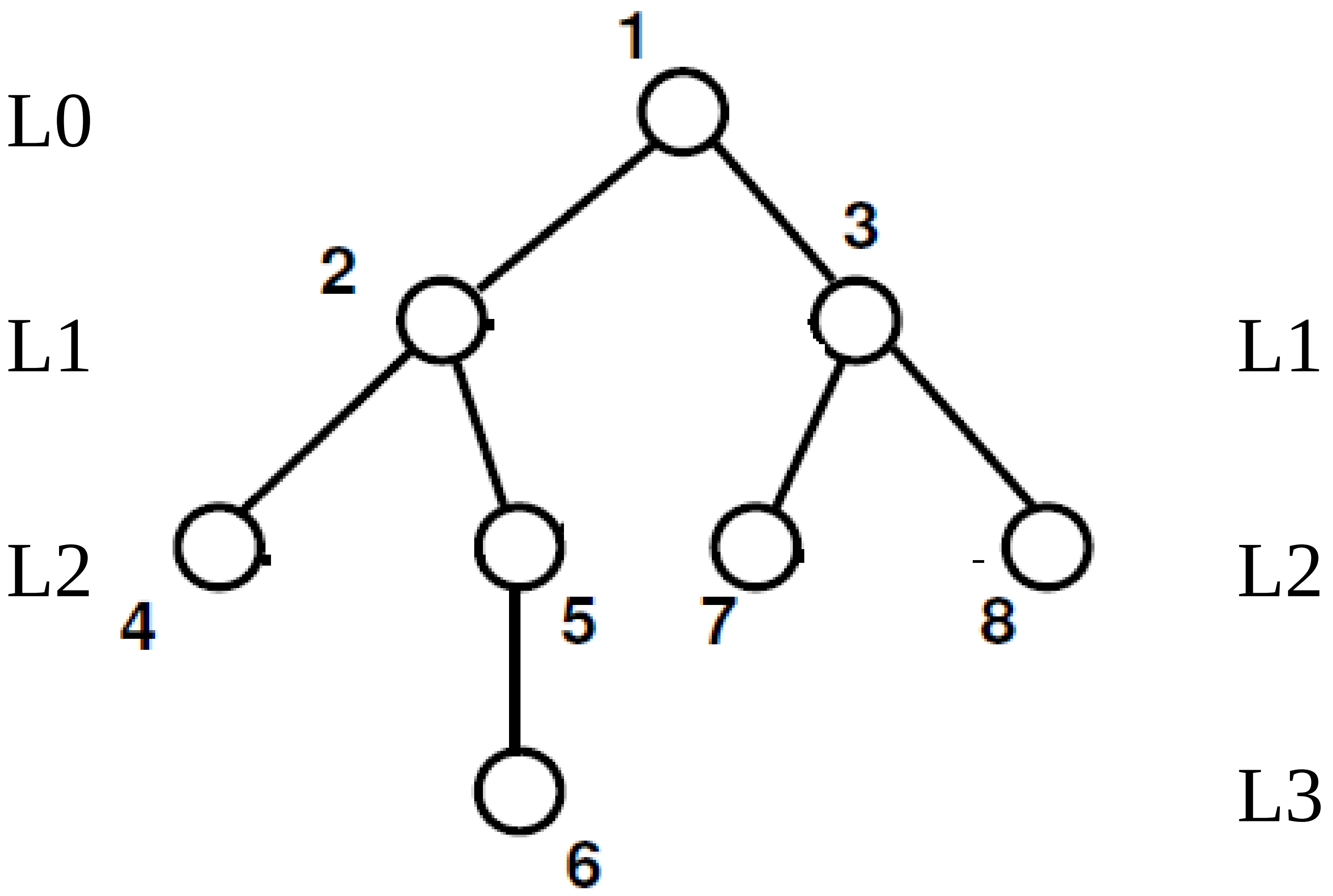
- Now we consider each node in L2
- From 4 we discover node 5.
  - 5 is already there in T.



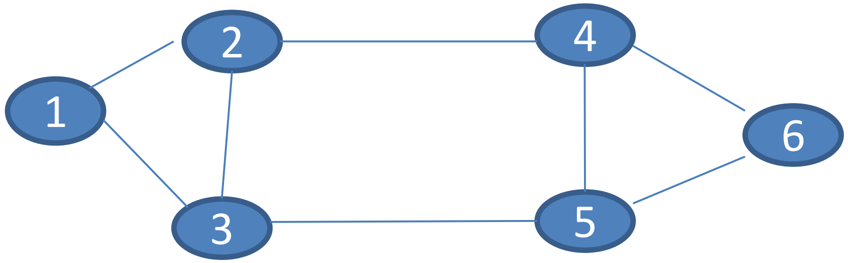
- From 5 we discover nodes {4,6}
  - 4 is already there in T
  - Add 6 to T

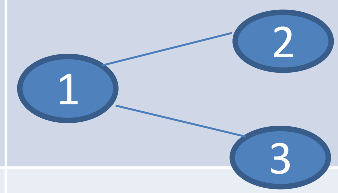
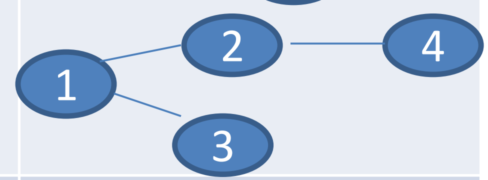
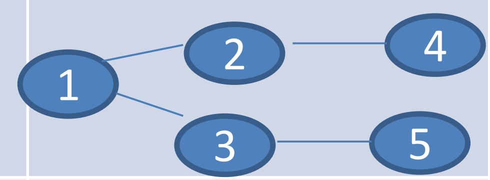
- From 7 we discover {3,8} and 8 we discover {3,7} which are already in T.

- For new node 6 added in L3, there are no more further nodes to be explored.
- The full BFS tree is as depicted below.



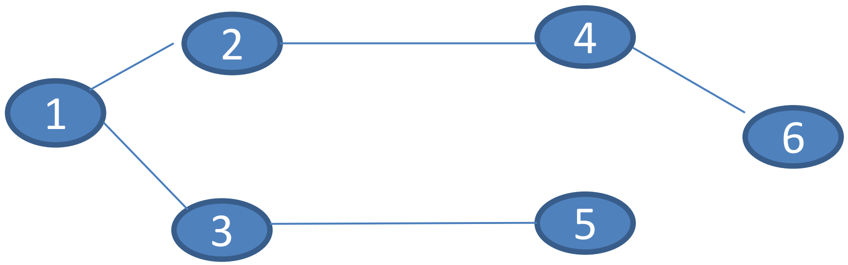
APPLYING THE BFS ALGORITHM



Discovered						List	Edge	Tree
T	F	F	F	F	F	L[0] = 1 I=0		Ø
T	T	F	F	F	F	L[1] =2 I=1	U=1, V=2	
T	T	T	F	F	F	L[1] =2, 3 I=1	U=1, V=3	
T	T	T	T	F	F	L[2]= 4 I=2	U=2, V=1, V=3, V=4	
T	T	T	T	T	F	L[2] = 5 I=2	U=3, V=1, V=2, V=5	
T	T	T	T	T	T	L[3] = 6 I=3	U=4, V=2, V=5, V=6	

Now L[i] is empty. So the BFS Tree.

APPLYING THE BFS ALGORITHM



BREADTH-FIRST SEARCH

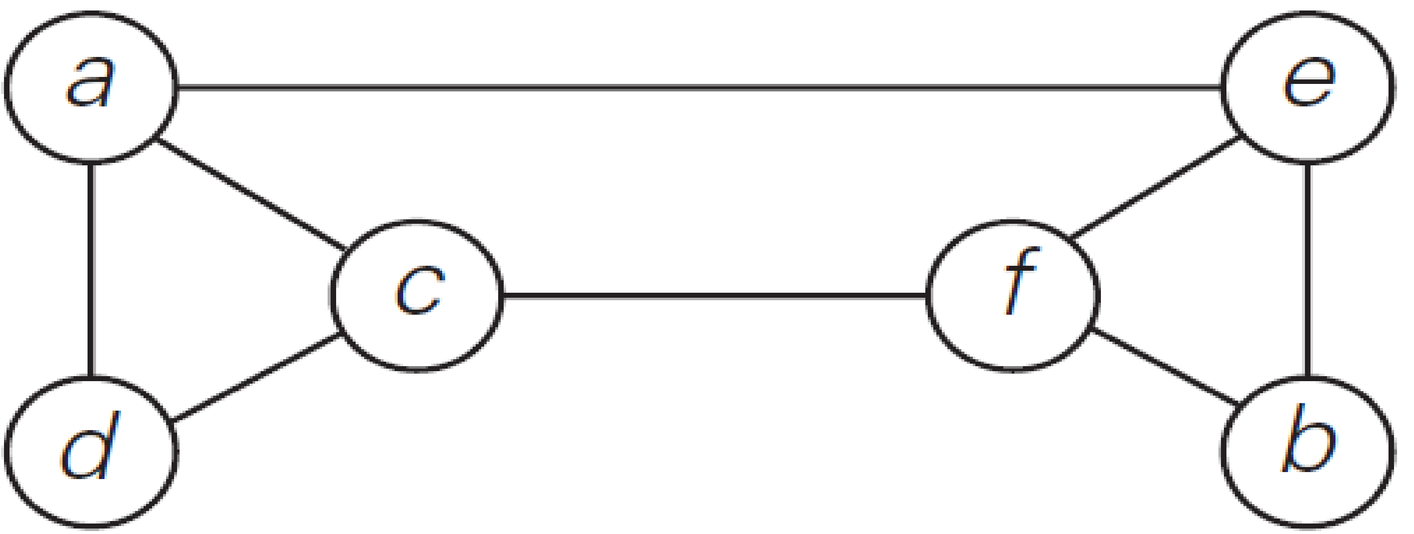
BFS(s):
  Mark s as "Visited".
  Initialize $R = \{s\}$.
  Define layer $L_0 = \{s\}$.
  While L_i is not empty
    For each node $u \in L_i$
      Consider each edge (u, v) incident to v
      If v is not marked "Visited" then
        Mark v "Visited"
        Add v to the set R and to layer L_{i+1}
      Endif
    Endfor
  Endwhile

BREADTH-FIRST SEARCH

- If we store each layer Li as a queue, then inserting nodes into layers and subsequently accessing them takes constant time per node.
- Furthermore, if we represent G using an adjacency list, then we spend constant time per edge over the course of the whole algorithm, since we consider each edge e at most once from each end.
- Thus, the overall time spent by the algorithm is O(m + n).

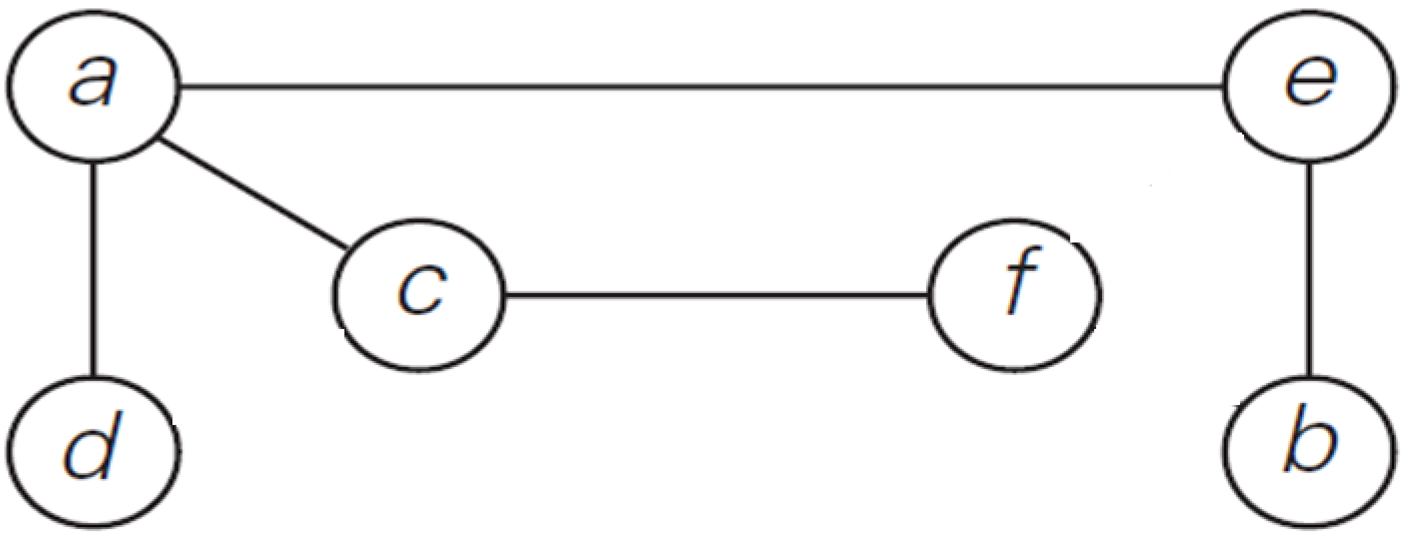
BREADTH-FIRST SEARCH

- Find the BFS Traversal for the following graph.



BREADTH-FIRST SEARCH

- BFS Tree



BFS CLAIMS

1. For each $j \geq 1$, layer Lj produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.

BFS CLAIMS

2. Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers Li and Lj respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.

BFS CLAIMS

Proof:

- Suppose by way of contradiction that i and j differed by more than 1; in particular, suppose $i < j - 1$.
- Now consider the point in the BFS algorithm when the edges incident to x were being examined.

BFS CLAIMS

- Since x belongs to layer Li, the only nodes discovered from x belong to layers Li+1 and earlier.
- If y is a neighbor of x, then it should have been discovered by this point at the latest and hence should belong to layer Li+1 or earlier.

BFS CLAIMS

3. The implementation of the BFS algorithm runs in time O(m + n) (i.e., linear in the input size), if the graph is given by the adjacency list representation.

Proof:

- We need to observe that the For loop processing a node u can take less than O(n) time if u has only a few neighbors.

BFS CLAIMS

- As before, let n_u denote the degree of node u, the number of edges incident to u.

- Now, the time spent in the For loop considering edges incident to node u is $O(n_u)$, so the total over all nodes is $O(\sum_{u \in V} n_u)$.

BFS CLAIMS

- Recall that $\sum_{u \in V} n_u = 2m$, and so the total time spent considering edges over the whole algorithm is O(m).

- We need O(n) additional time to set up lists and manage the array Discovered.

- So the total time spent is O(m + n) as claimed.

TESTING BIPARTITENESS: AN APPLICATION OF BFS

- **The Problem**
- Triangle is not bipartite, since we can color one node red, another one blue, and then we can’t do anything with the third node.
- If a graph G simply contains an odd cycle, then we can apply an argument; thus
- ***If a graph G is bipartite, then it cannot contain an odd cycle.***

TESTING BIPARTITENESS: AN APPLICATION OF BFS

- **Designing the Algorithm**
- We can implement this on top of BFS, by simply taking the implementation of BFS and adding an extra array Color over the nodes.
- Whenever we get to a step in BFS where we are adding a node v to a list L[i + 1], we assign
- Color[v]= red if i + 1 is an even number
- Color[v]= blue if i + 1 is an odd number.

TESTING BIPARTITENESS: AN APPLICATION OF BFS

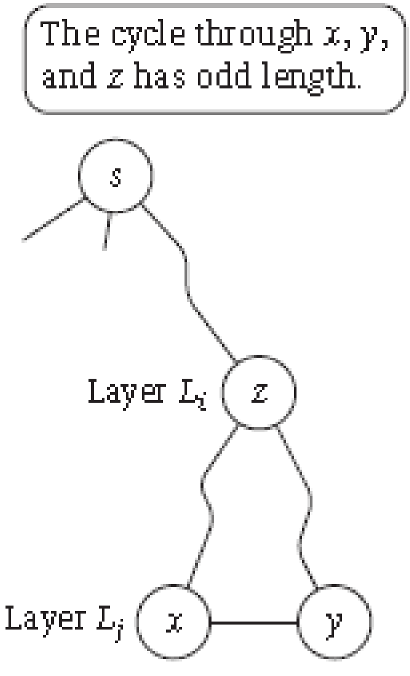
- **Designing the Algorithm**
- At the end of this procedure, we simply scan all the edges and determine whether there is any edge for which both ends received the same color.
- Thus, the total running time for the coloring algorithm is $O(m + n)$, just as it is for BFS.

TESTING BIPARTITENESS: AN APPLICATION OF BFS

- **Analyzing the Algorithm**
- Let G be a connected graph, and let L1, L2, . . . be the layers produced by BFS starting at node s. Then exactly one of the following two things must hold.
- (i) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.

TESTING BIPARTITENESS: AN APPLICATION OF BFS

- **Analyzing the Algorithm**
- (ii) There is an edge of G joining two nodes of the same layer. In this case, G contains an odd-length cycle, and so it cannot be bipartite.



TESTING BIPARTITENESS: AN APPLICATION OF BFS

- **Proof:**
- First consider case (i), where we suppose that there is no edge joining two nodes of the same layer.
- We know that every edge of G joins nodes either in the same layer or in adjacent layers.
  - Our assumption for case (i) is precisely that the first of these two alternatives never happens, so this means that every edge joins two nodes in adjacent layers.
  - But our coloring procedure gives nodes in adjacent layers the opposite colors, and so every edge has ends with opposite colors.
  - Thus this coloring establishes that G is bipartite.

TESTING BIPARTITENESS: AN APPLICATION OF BFS

- Now suppose we are in case (ii); why must G contain an odd cycle?
- We are told that G contains an edge joining two nodes of the same layer.
- Suppose this is the edge e = (x, y), with x, y ∈ Lj.
- Also, for notational reasons, recall that L0 (“layer 0”) is the set consisting of just s.
- Now consider the BFS tree T produced by our algorithm, and let z be the node whose layer number is as large as possible, subject to the condition that z is an ancestor of both x and y in T;

TESTING BIPARTITENESS: AN APPLICATION OF BFS

- Now suppose we are in case (ii); why must G contain an odd cycle?
- Suppose z ∈ Li, where i < j.
- We consider the cycle C defined by following the z-x path in T, then the edge e and then y-z path in T.
- The length of this cycle is $(j - i) + 1 + (j - i)$, adding the length of its three parts separately; this is equal to $2(j - i) + 1$, which is an odd number.

DEPTH-FIRST SEARCH

DEPTH-FIRST SEARCH

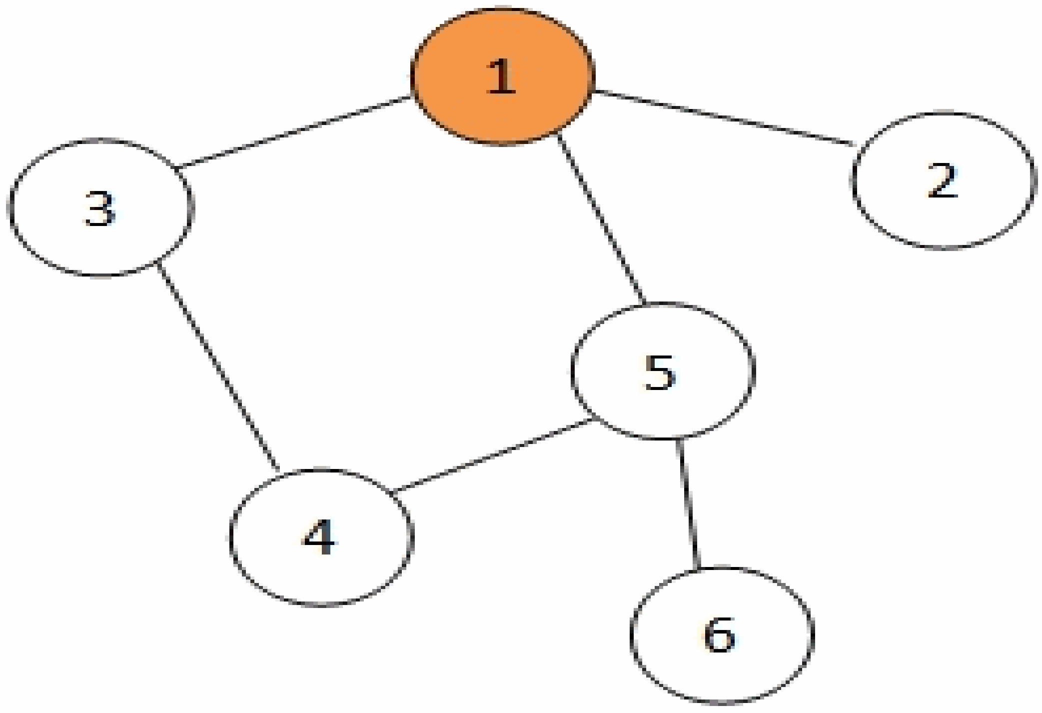
- Another method to find the nodes reachable from S.
- Start from S and try the first edge leading out of it, to a node v.
- Follow the first edge leading out of v, and continue in this way until you reached a “dead end“.

DEPTH-FIRST SEARCH

- Then back-track till you got to a node with an unvisited neighbor, and resume from there.

- Called Depth-First Search (DFS) as it explores G by going as deeply as possible, and only retreating when necessary.

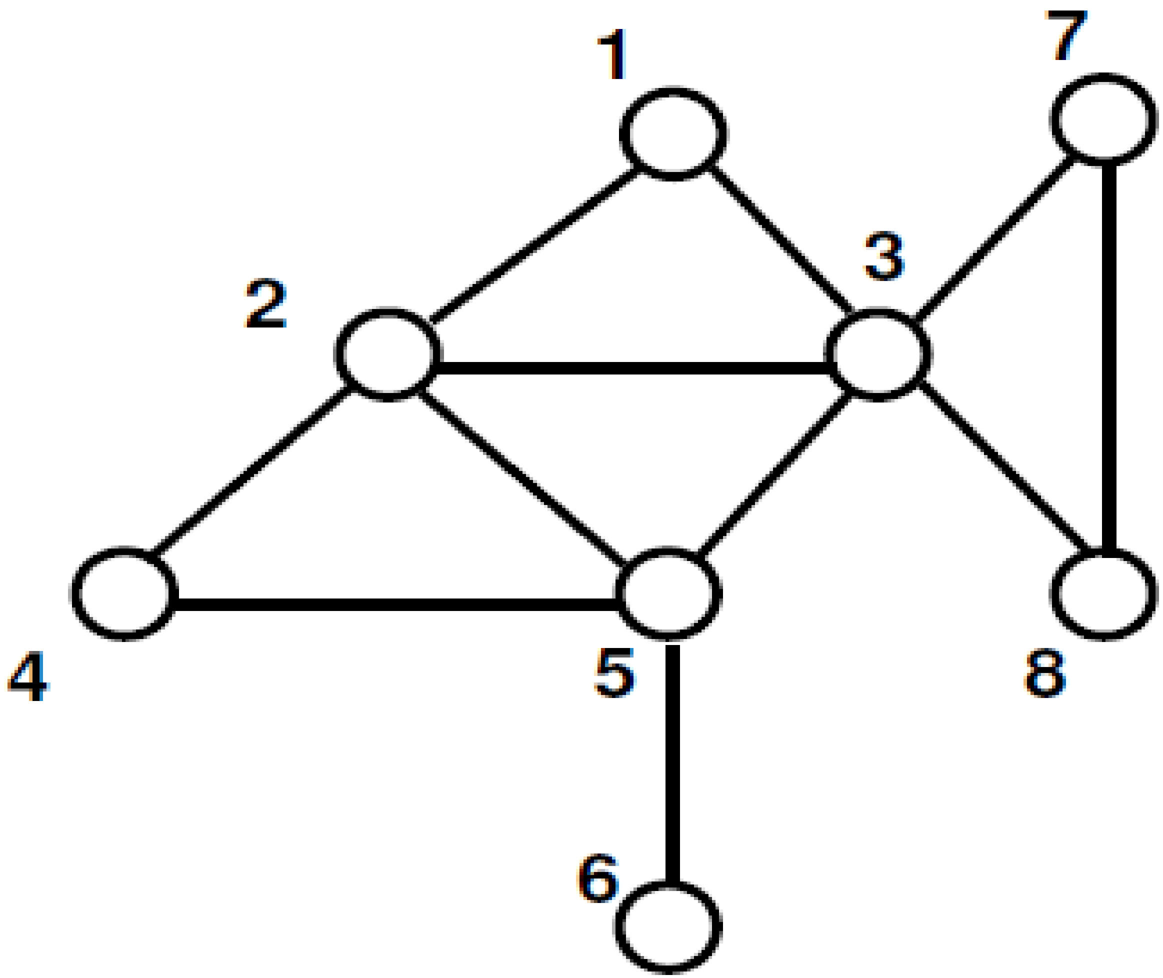
DEPTH-FIRST SEARCH



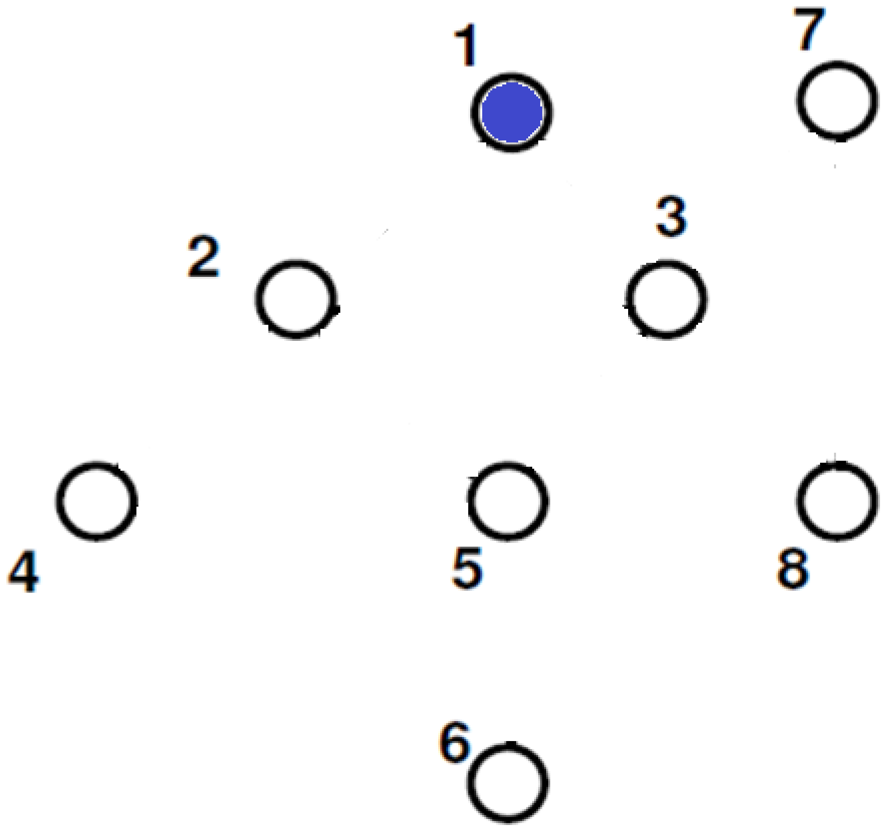
DEPTH-FIRST SEARCH

```
DFS(u):
  Mark u as "Visited" and add u to R.
  For each edge (u,v) incident to u
    If v is not marked "Visited" then
      Add v to R.
      Recursively invoke DFS(v).
    Endif
  Endfor
```

DEPTH-FIRST SEARCH



DEPTH-FIRST SEARCH

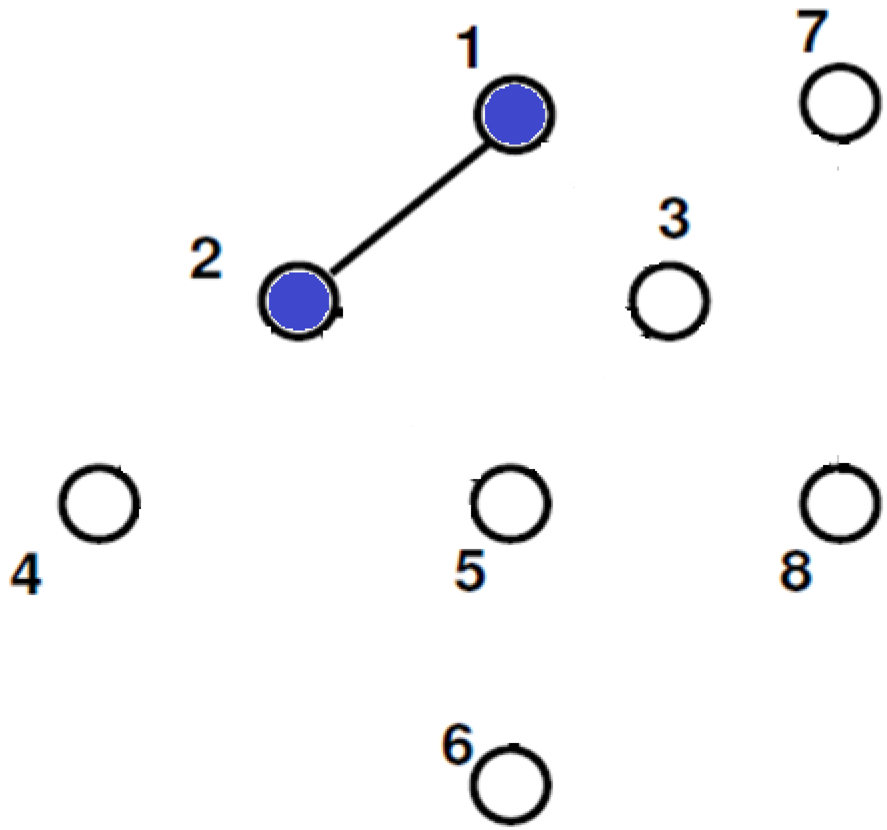


STACK

1

OUTPUT: 1

DEPTH-FIRST SEARCH

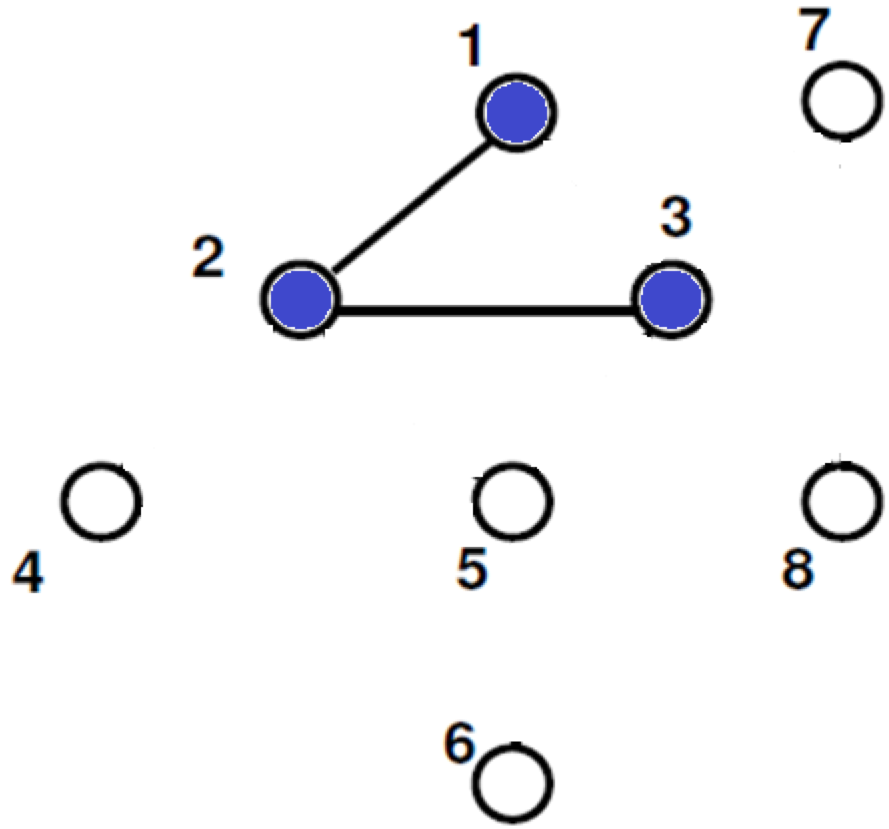


STACK

2
1

OUTPUT: 1 2

DEPTH-FIRST SEARCH

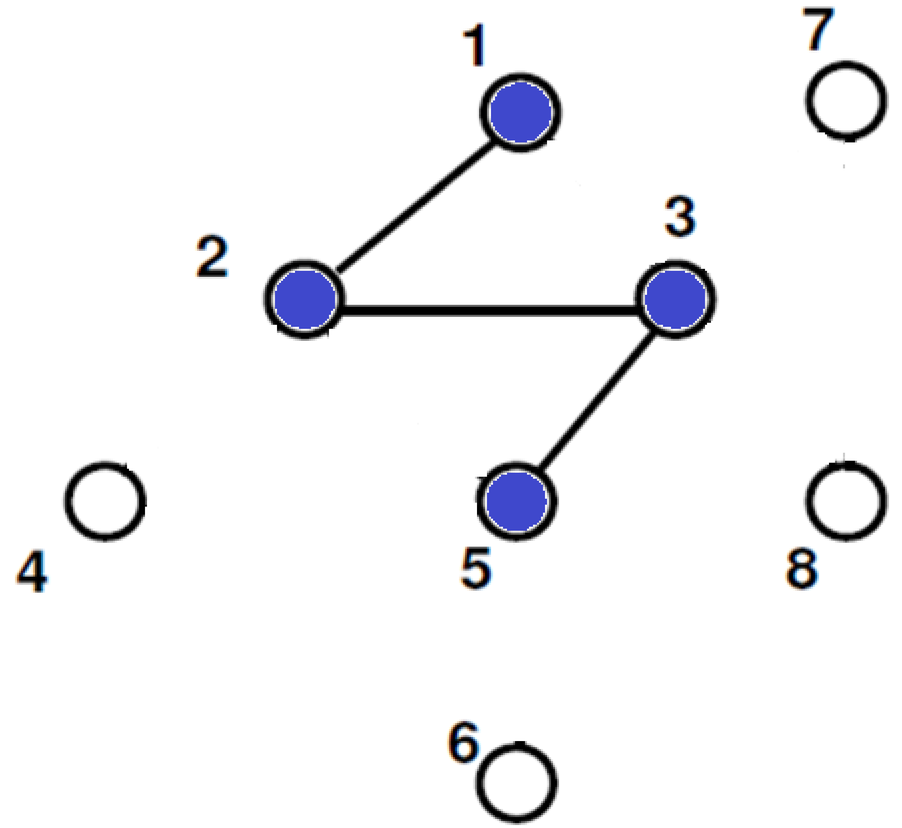


STACK

3
2
1

OUTPUT: 1 2 3

DEPTH-FIRST SEARCH

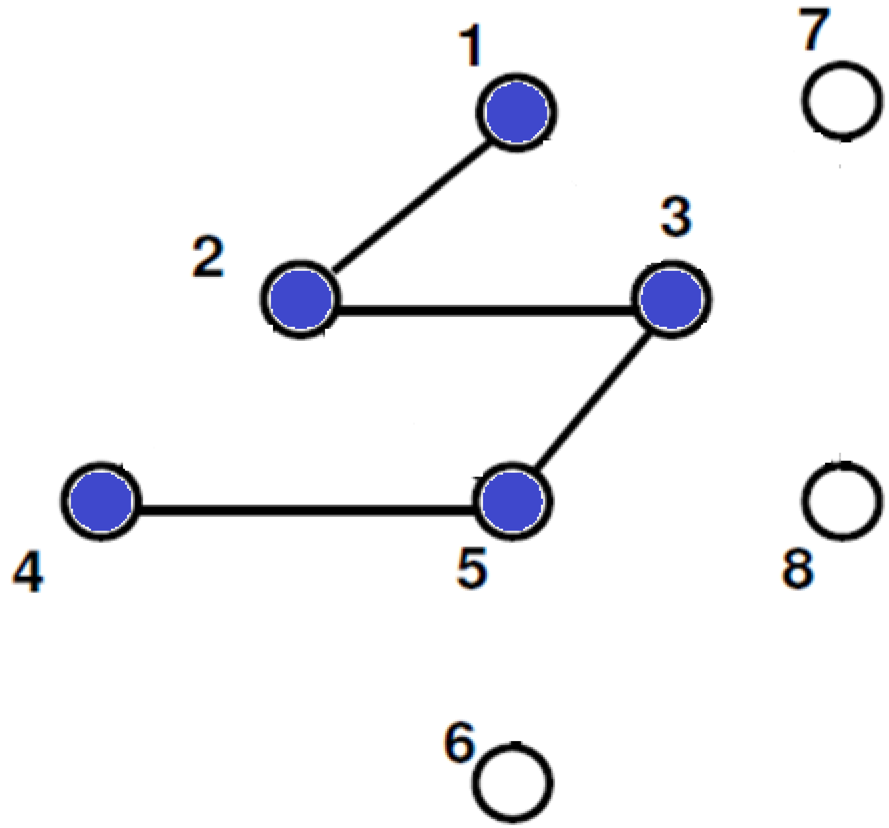


STACK

5
3
2
1

OUTPUT: 1 2 3 5

DEPTH-FIRST SEARCH

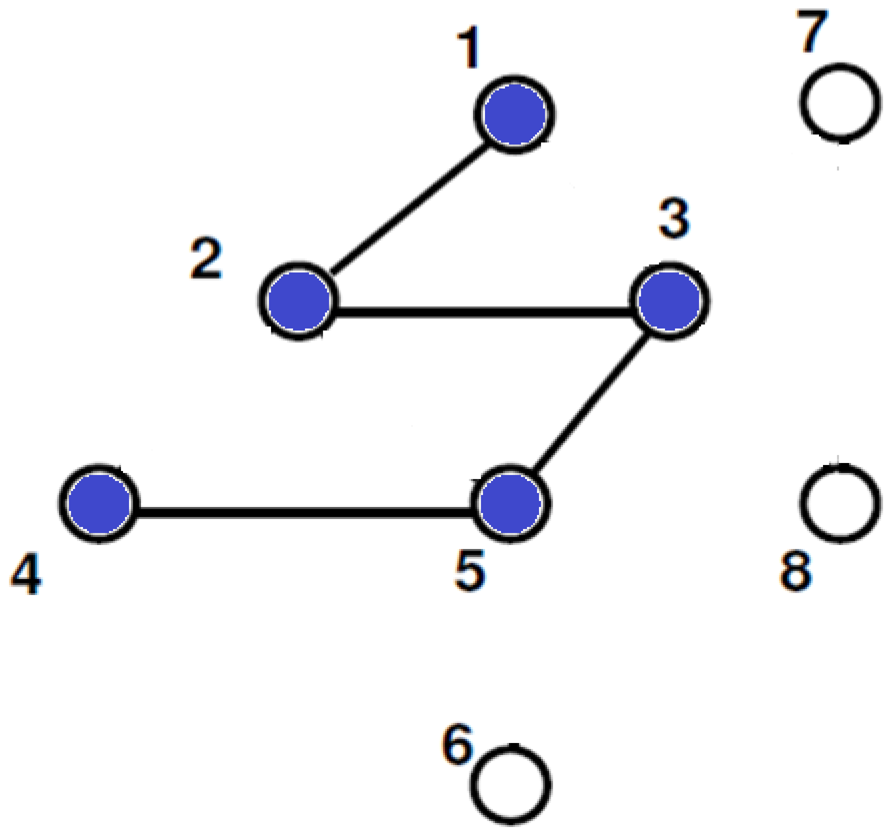


STACK

4
5
3
2
1

OUTPUT: 1 2 3 5 4

DEPTH-FIRST SEARCH



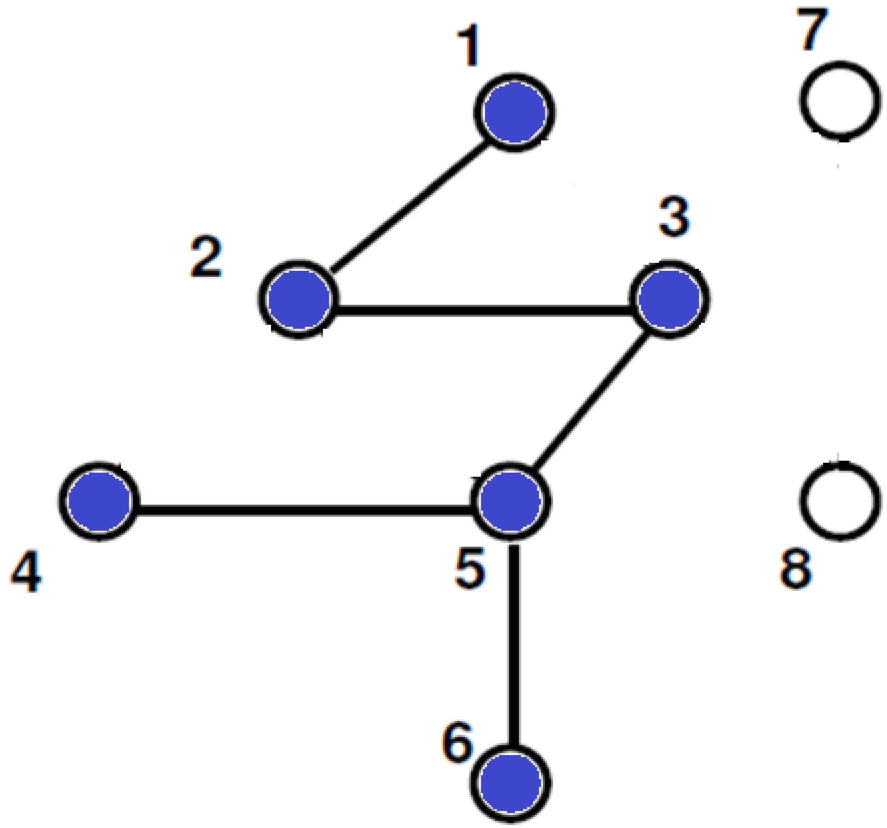
STACK

5
3
2
1

POP 4

OUTPUT: 1 2 3 5 4

DEPTH-FIRST SEARCH

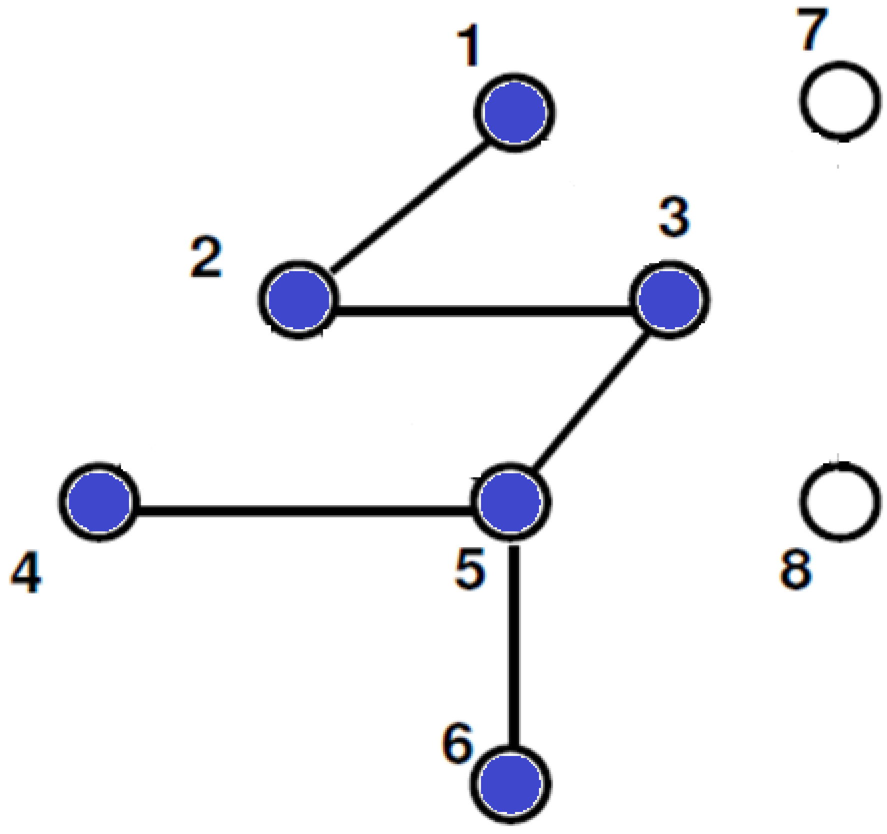


STACK

6
5
3
2
1

OUTPUT: 1 2 3 5 4 6

DEPTH-FIRST SEARCH



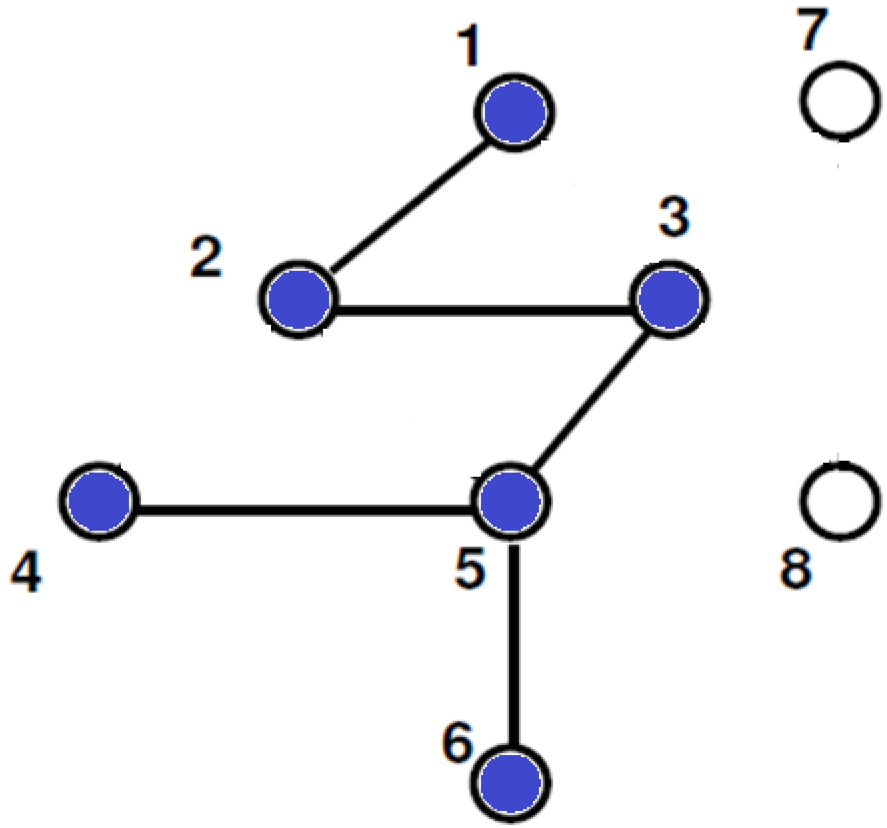
STACK

5
3
2
1

POP 6

OUTPUT: 1 2 3 5 4 6

DEPTH-FIRST SEARCH



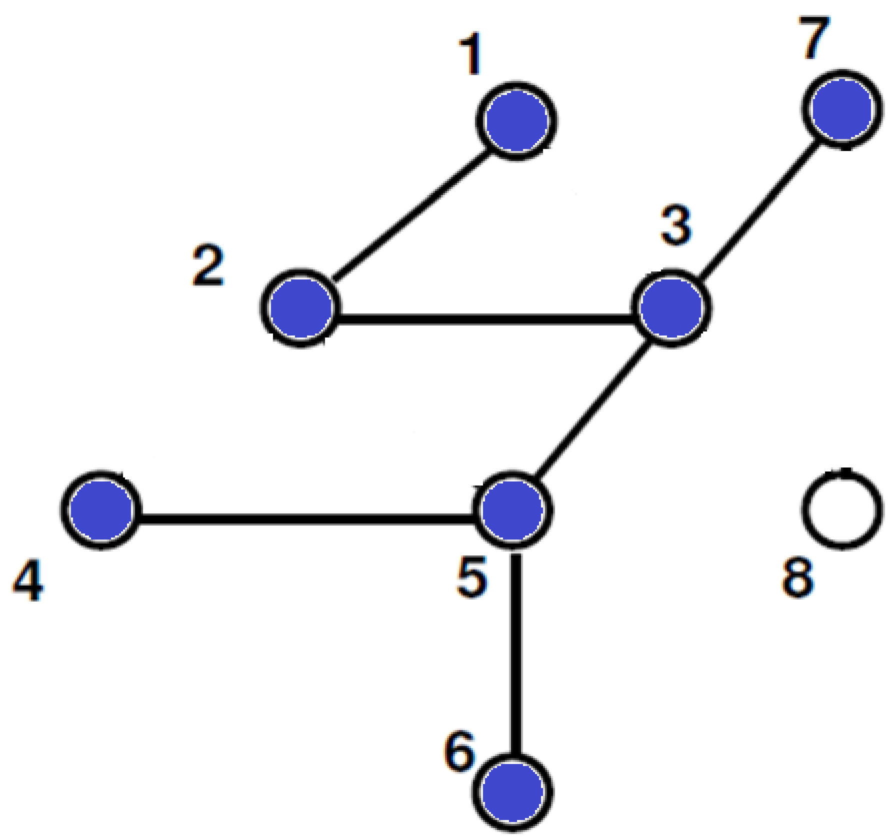
STACK

3
2
1

POP 5

OUTPUT: 1 2 3 5 4 6

DEPTH-FIRST SEARCH

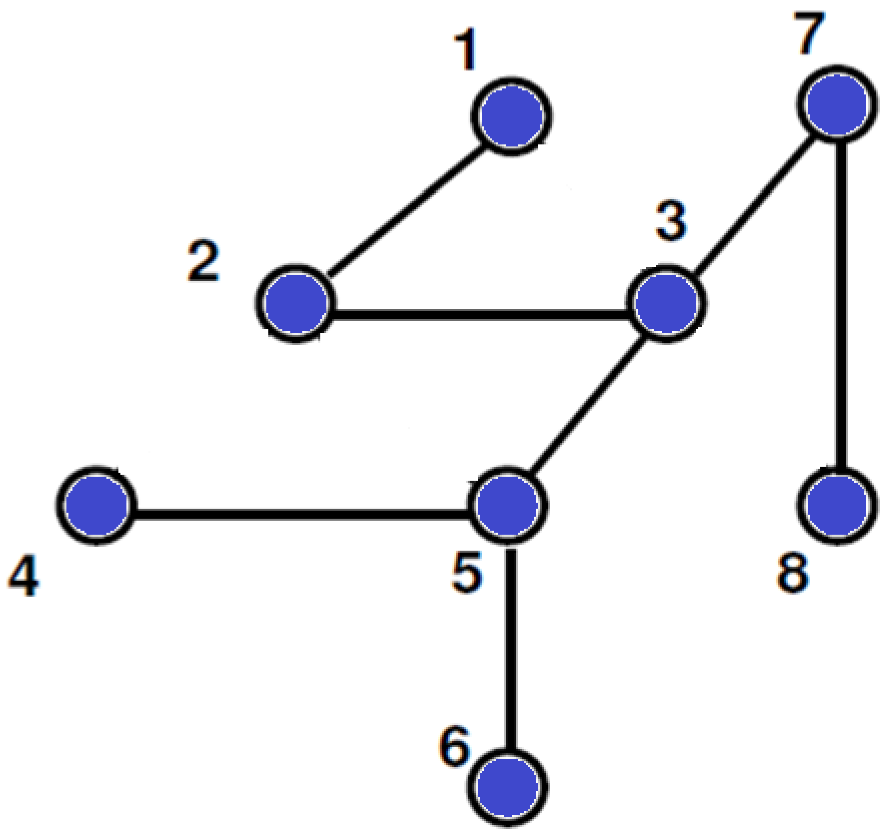


STACK

7
3
2
1

OUTPUT: 1 2 3 5 4 6 7

DEPTH-FIRST SEARCH

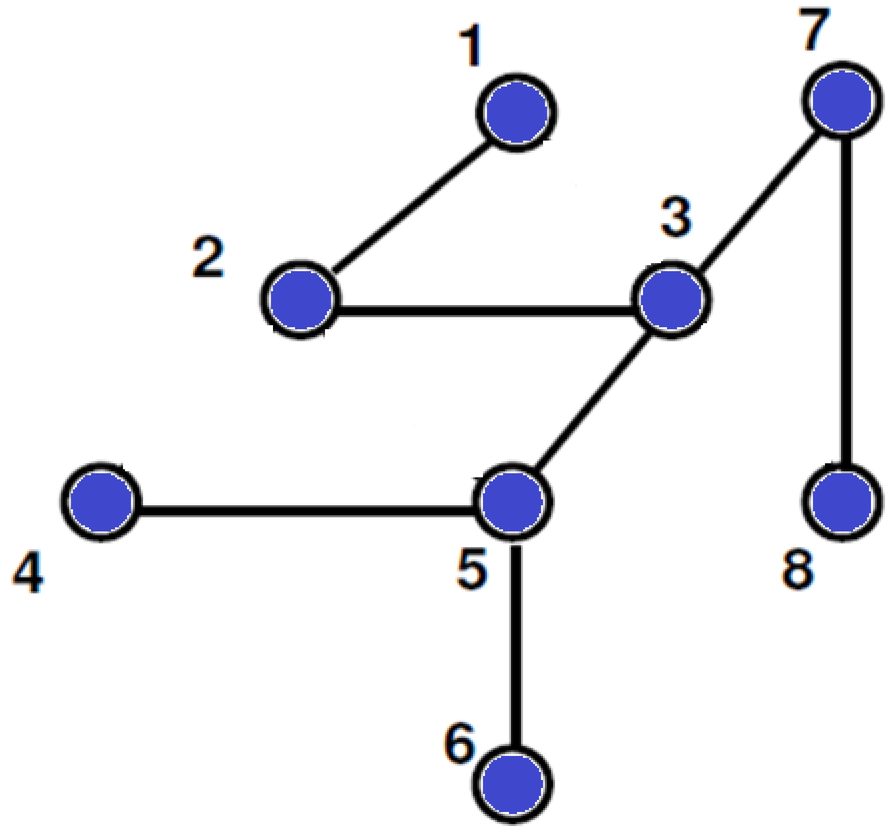


STACK

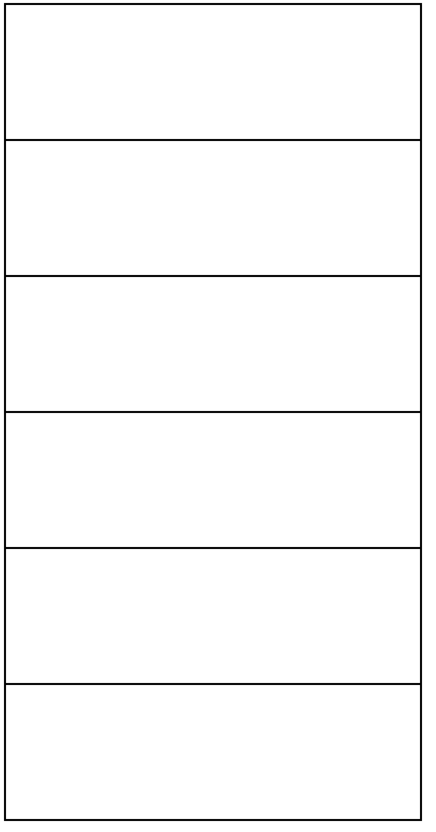
8
7
3
2
1

OUTPUT: 1 2 3 5 4 6 7 8

DEPTH-FIRST SEARCH

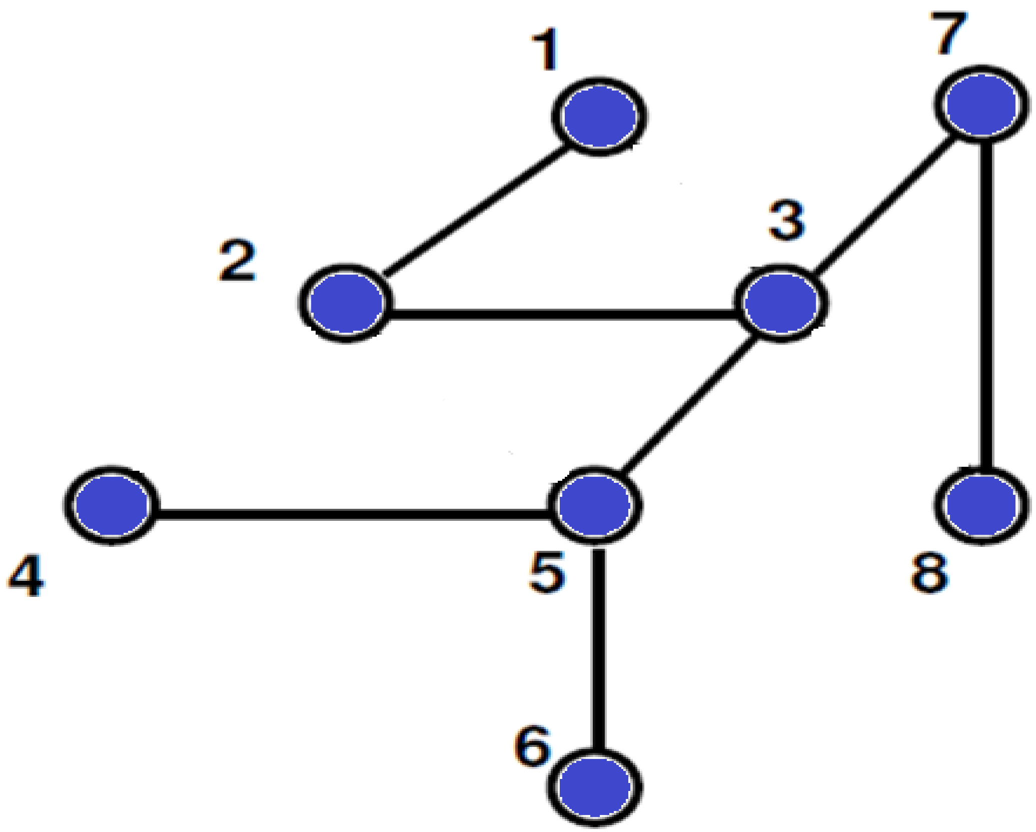


STACK EMPTY



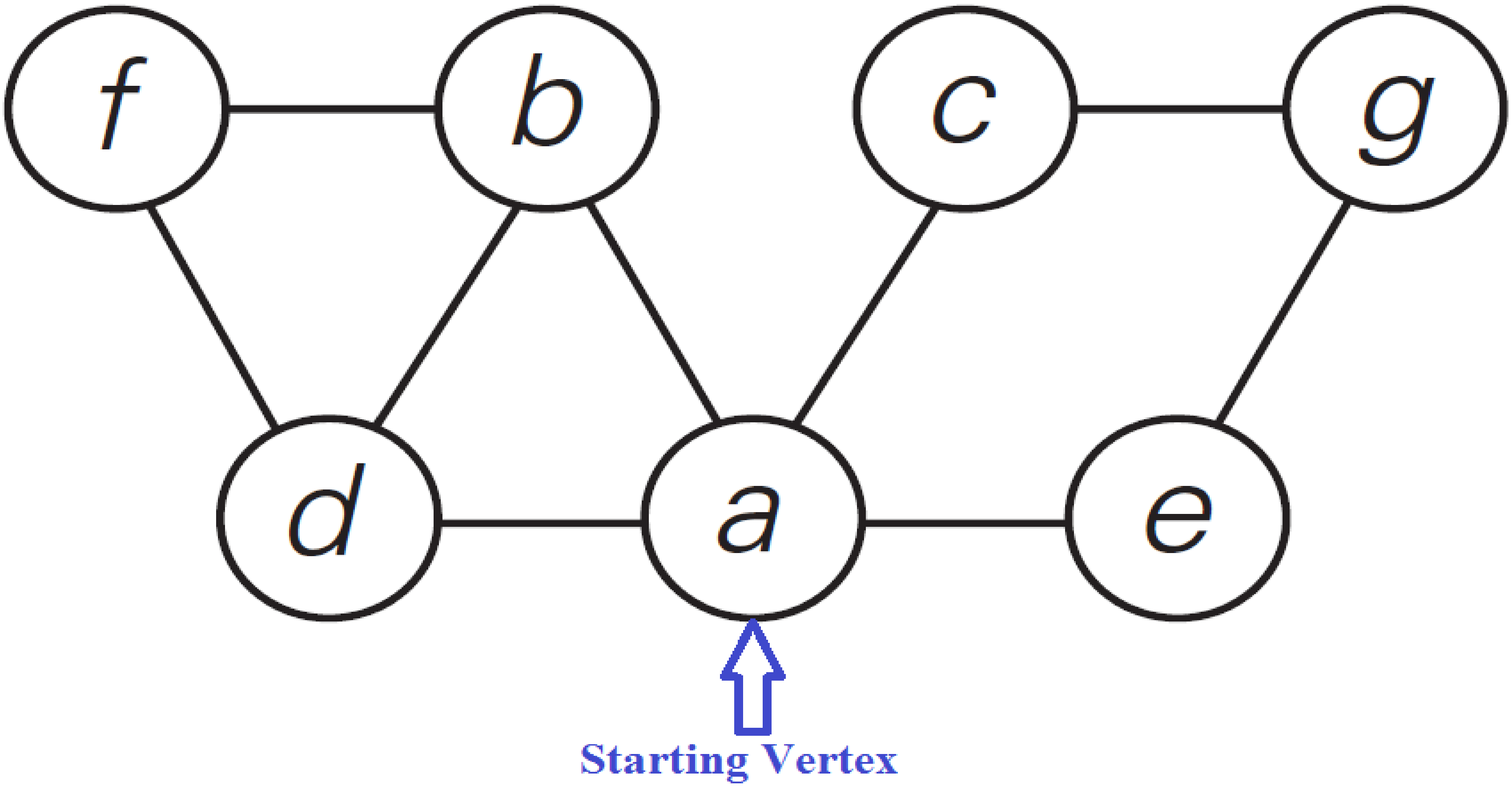
OUTPUT: 1 2 3 5 4 6 7 8

DEPTH-FIRST SEARCH

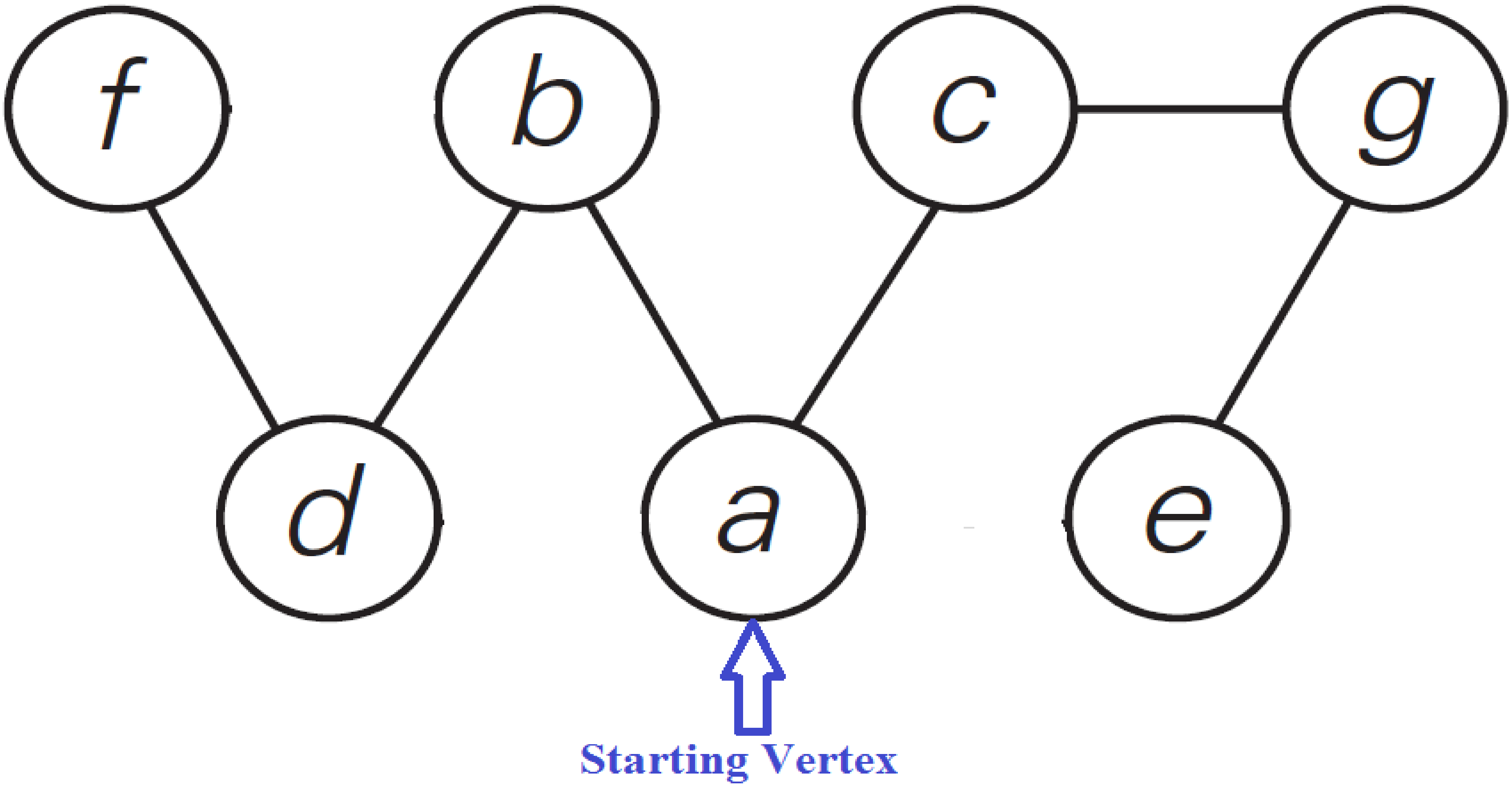


OUTPUT: 1 2 3 5 4 6 7 8

DEPTH-FIRST SEARCH



DEPTH-FIRST SEARCH



APPLICATIONS OF GRAPH TRAVERSALS

1. Testing Whether a Graph is Bipartite

2. Finding Cut-Points in a Graph

APPLICATIONS OF GRAPH TRAVERSALS

- Finding Cut-Points in a Graph

- A Connected graph G = (V, E), we say that u Є V is a cut-point if deleting u disconnects G.
- In other words, if G – {u} is not connected.
- We can think of the cut-points as the " weak points" of G, the destruction of a single cut-point separates the graph into multiple pieces.

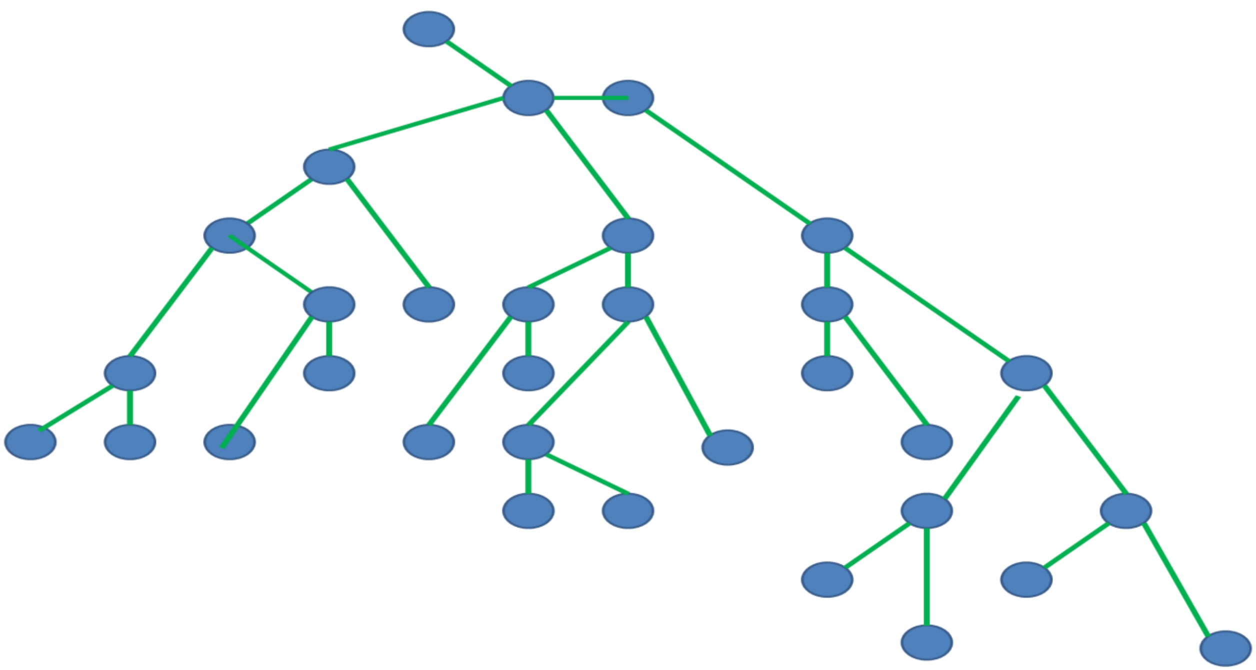
DIRECTED ACYCLIC GRAPHS

AND

TOPOLOGICAL ORDERING

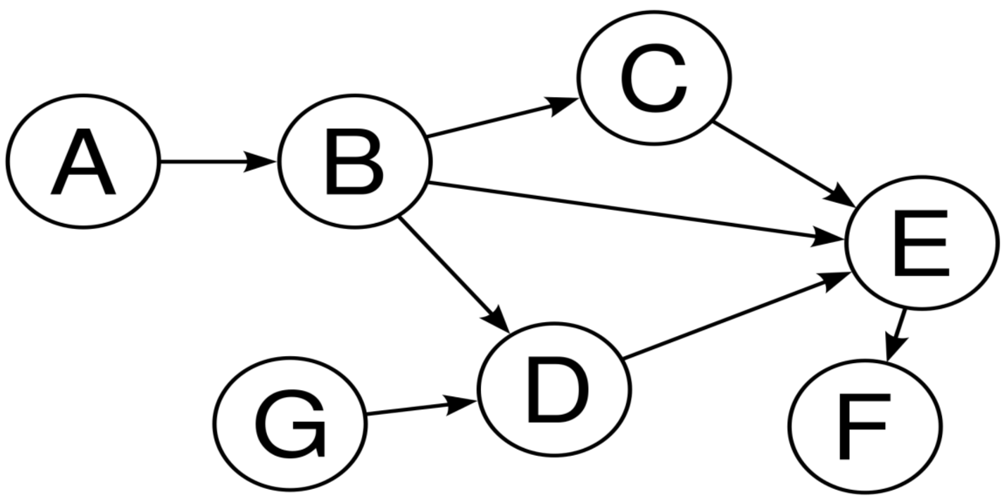
DIRECTED ACYCLIC GRAPHS AND TOPOLOGICAL ORDERING

- If an undirected graph has no cycles, then it has an extremely simple structure.
- Each of its connected components is a tree.



DIRECTED ACYCLIC GRAPHS AND TOPOLOGICAL ORDERING

- But it is possible for a directed graph to have no (directed) cycles and still have a very rich structure.
- If a directed graph has no cycles, we call it as a Directed Acyclic Graph, or a DAG for short.

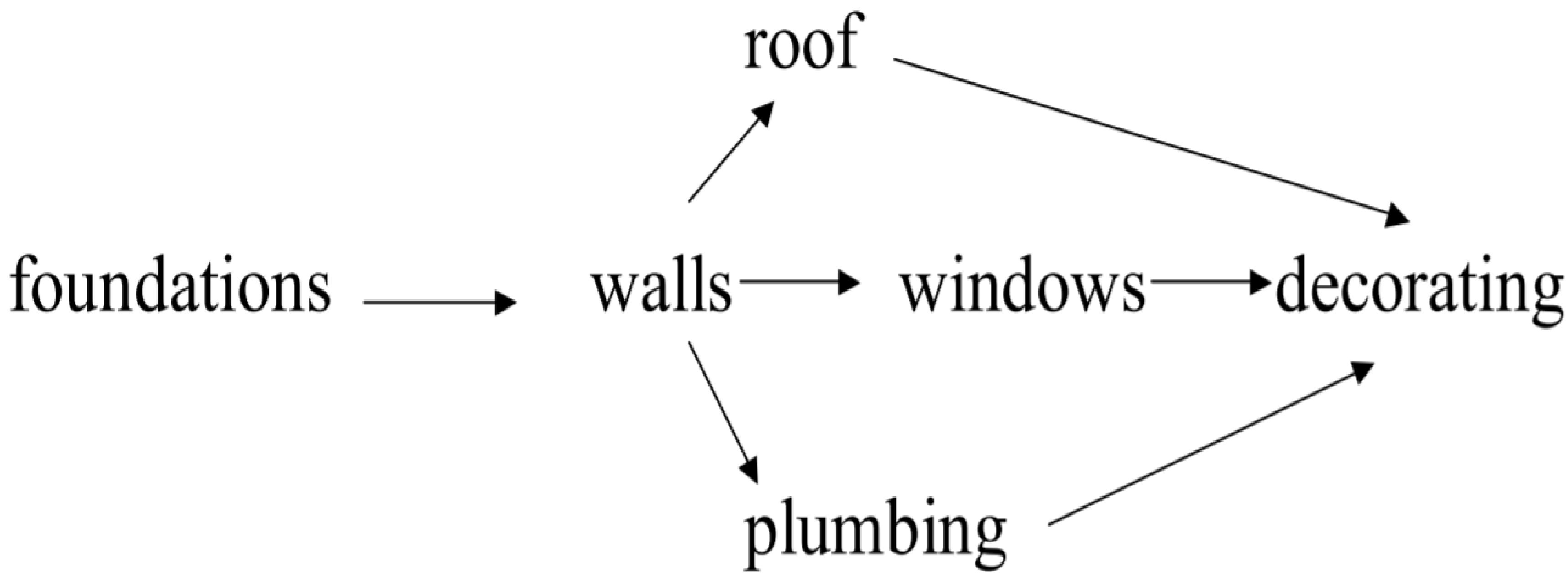


EXAMPLE

decorating walls plumbing

foundations windows roof

EXAMPLE



Possible sequence:

Foundations-Walls-Roof-Windows-Plumbing-Decorating

THE PROBLEM

- DAGs can be used to encode precedence relations or dependencies in a natural way.
- Suppose we have a set of tasks labeled {1, 2, ….. , n} that need to be performed.
- There are dependencies among them stipulating, for certain pairs i and j, that i must be performed before j.

THE PROBLEM

- We can represent such an interdependent set of tasks by introducing a node for each task, and a directed edge (i, j) whenever i must be done before j.

- If the precedence relation is to be at all meaningful, the resulting graph G must be a DAG.

TOPOLOGICAL ORDERING

SOURCE – REMOVAL ALGORITHM

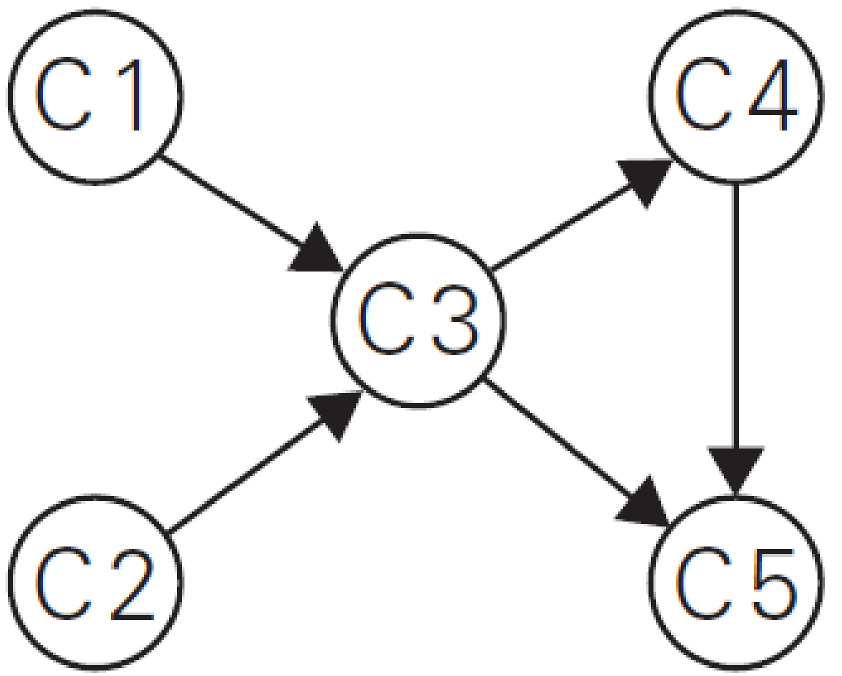
To compute a topological ordering of G:

Find a node v with no in-coming edges and order it first

Delete v from G.

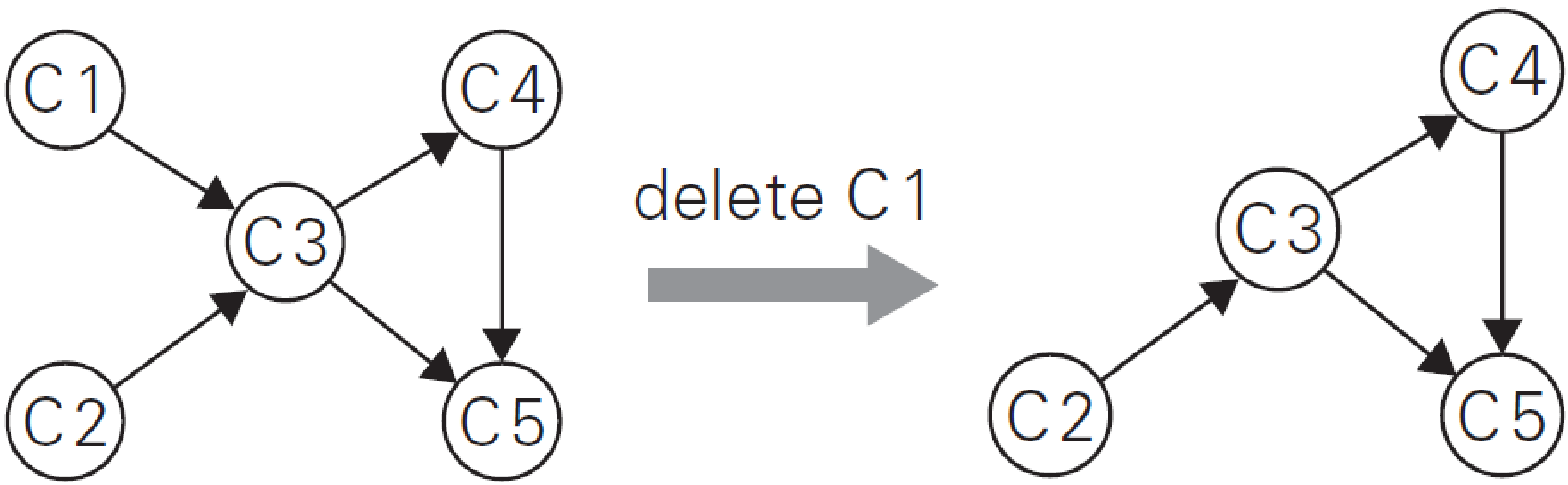
Recursively compute a topological ordering of $G - \{v\}$
and append this order after v

TOPOLOGICAL ORDERING

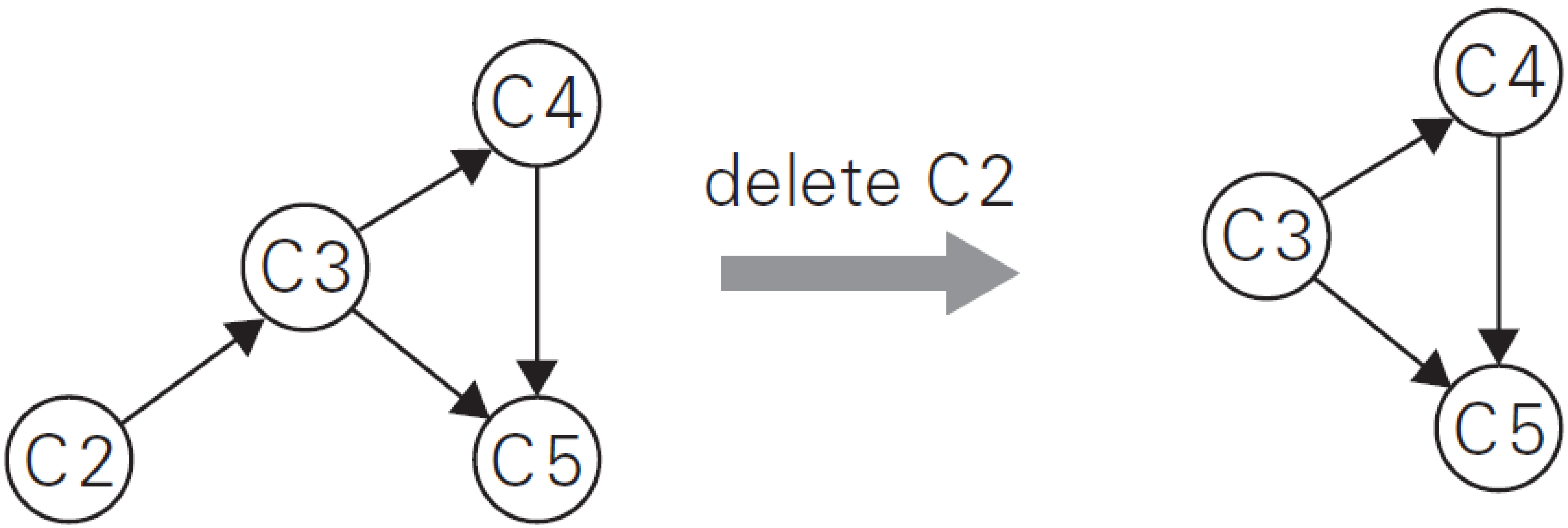


TOPOLOGICAL ORDERING

Method 1: Source – Removal Algorithm



TOPOLOGICAL ORDERING



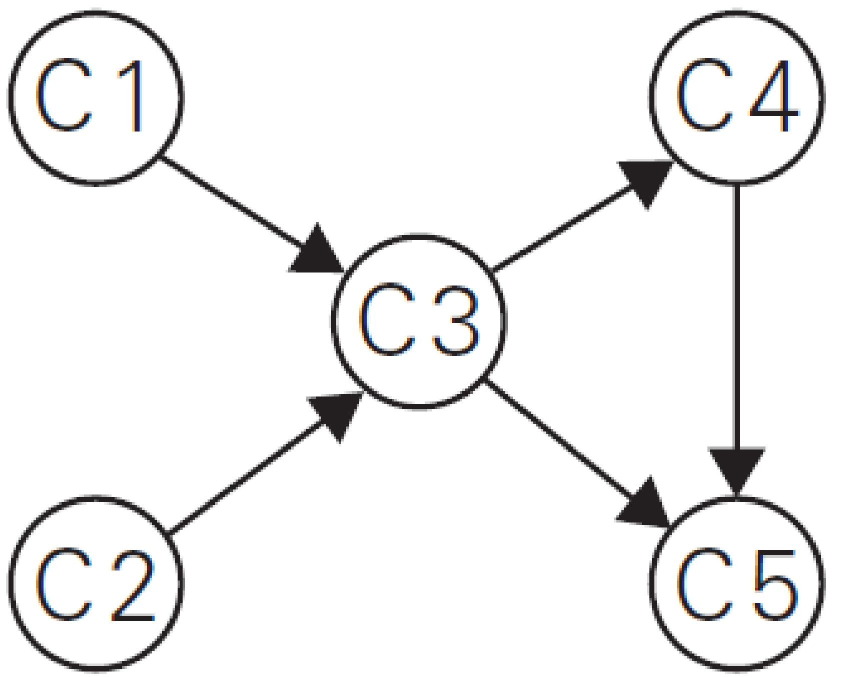
TOPOLOGICAL ORDERING

TOPOLOGICAL ORDERING

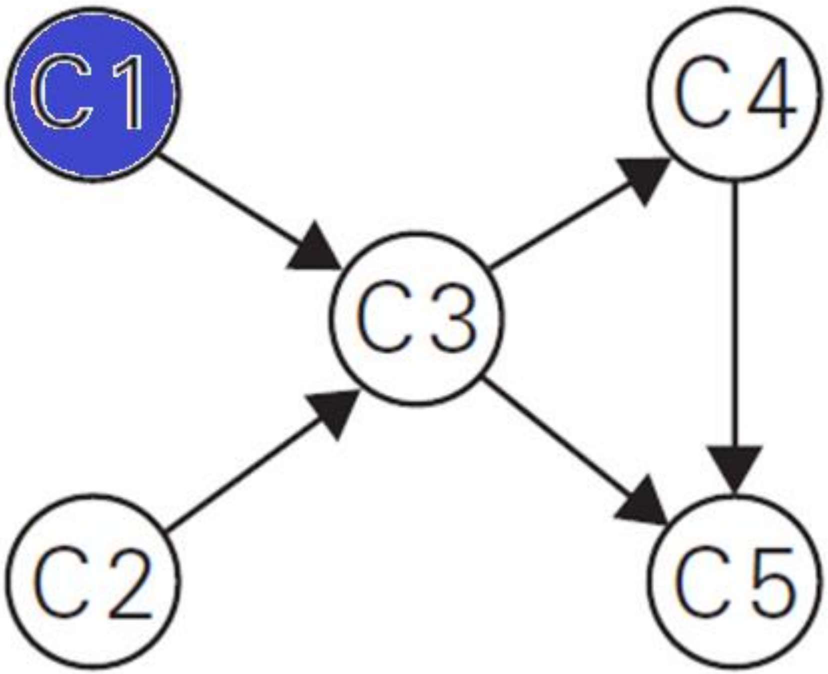
The solution obtained is C1, C2, C3, C4, C5

TOPOLOGICAL ORDERING

Method 2: Using DFS Traversal



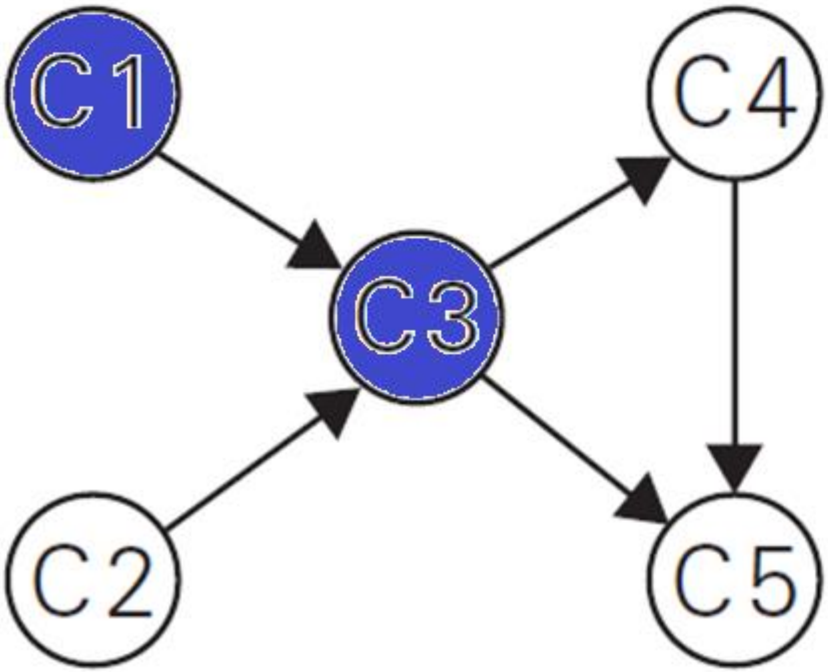
TOPOLOGICAL ORDERING



STACK

C1

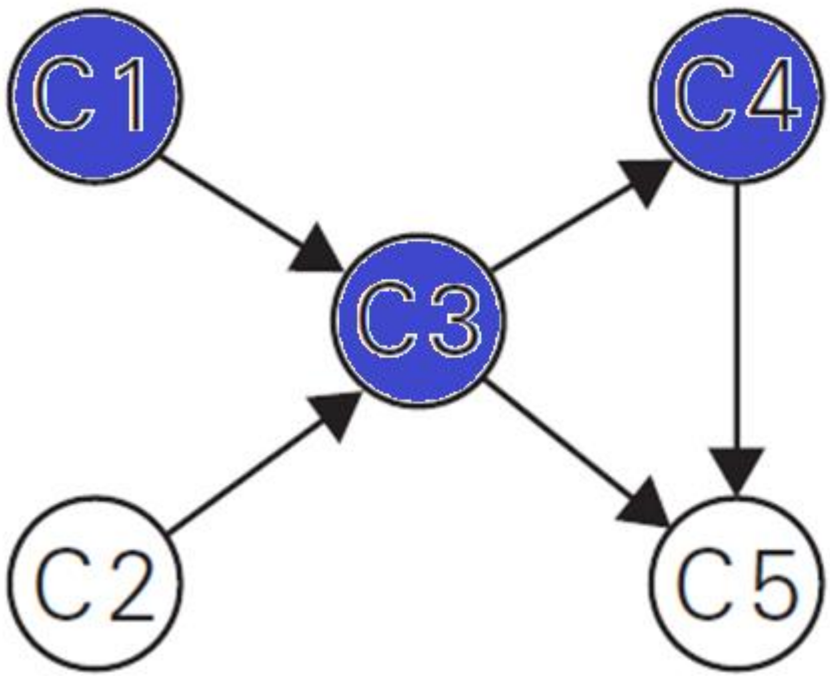
TOPOLOGICAL ORDERING



STACK

C3
C1

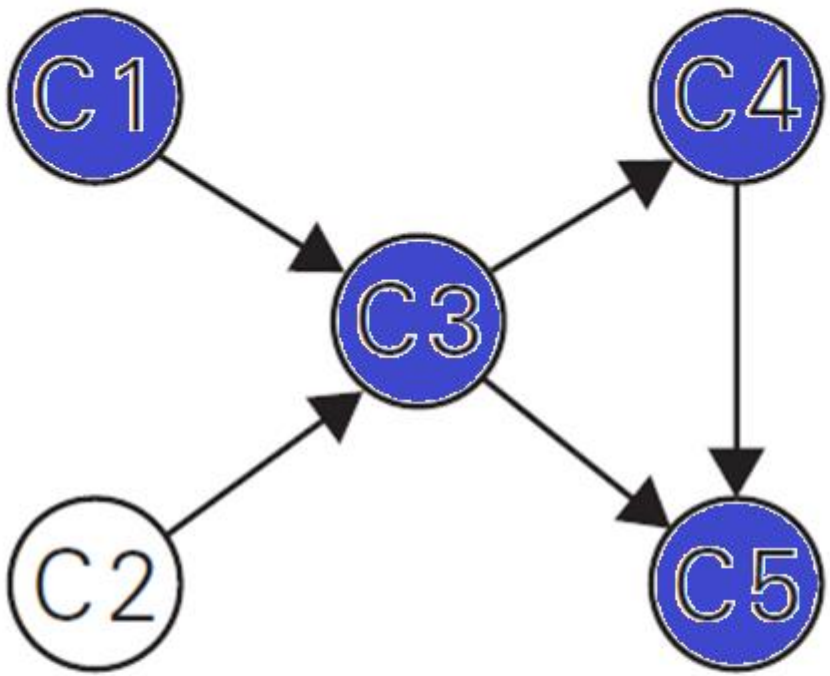
TOPOLOGICAL ORDERING



STACK

C4
C3
C1

TOPOLOGICAL ORDERING



STACK

C5
C4
C3
C1

TOPOLOGICAL ORDERING

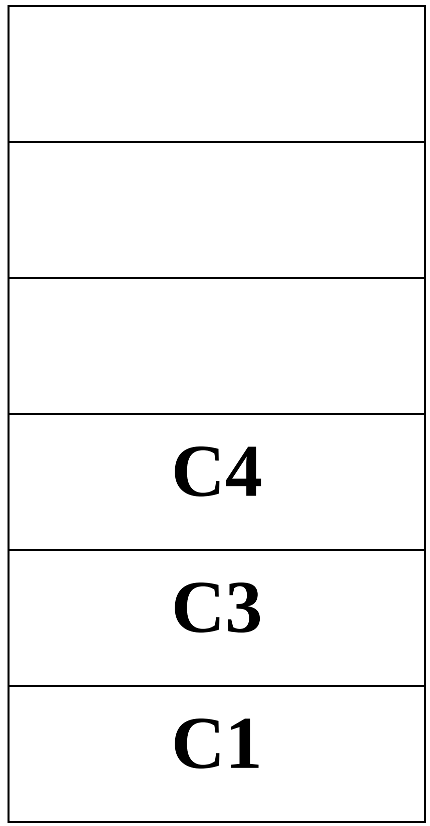
STACK

C5
C4
C3
C1

POP C5

TOPOLOGICAL ORDERING

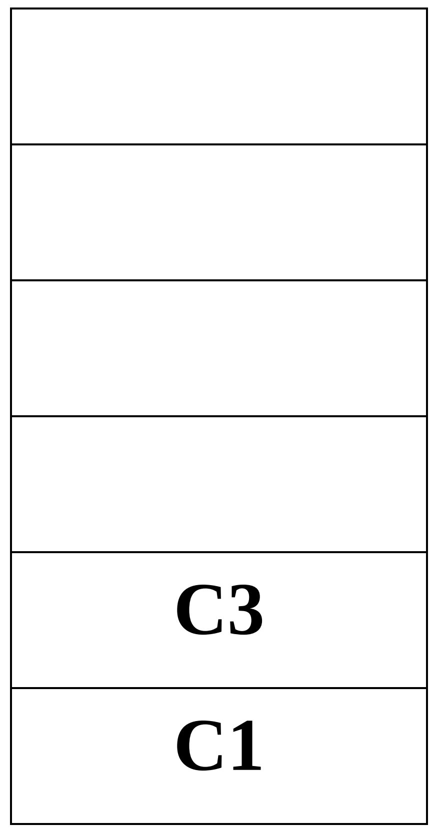
STACK



POP C4

TOPOLOGICAL ORDERING

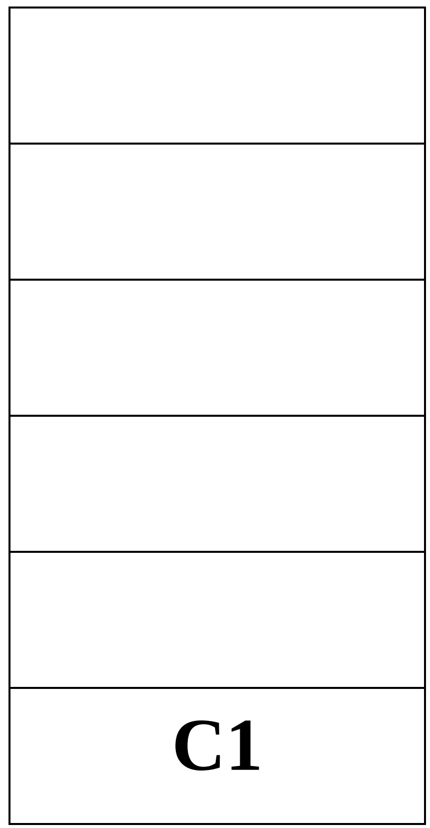
STACK



POP C3

TOPOLOGICAL ORDERING

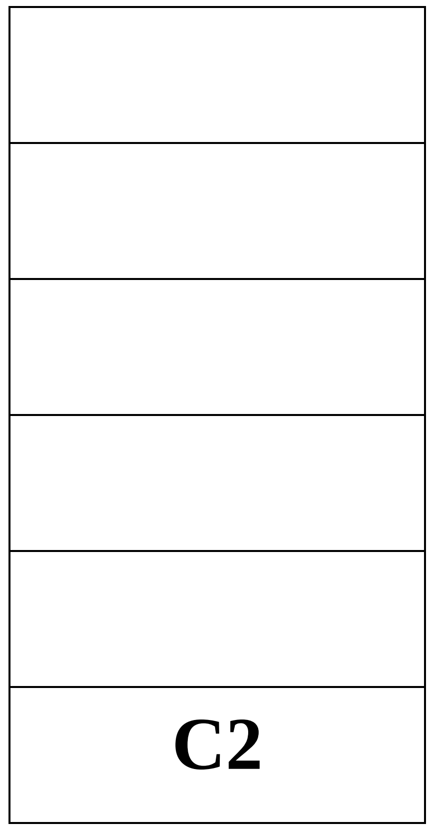
STACK



POP C1

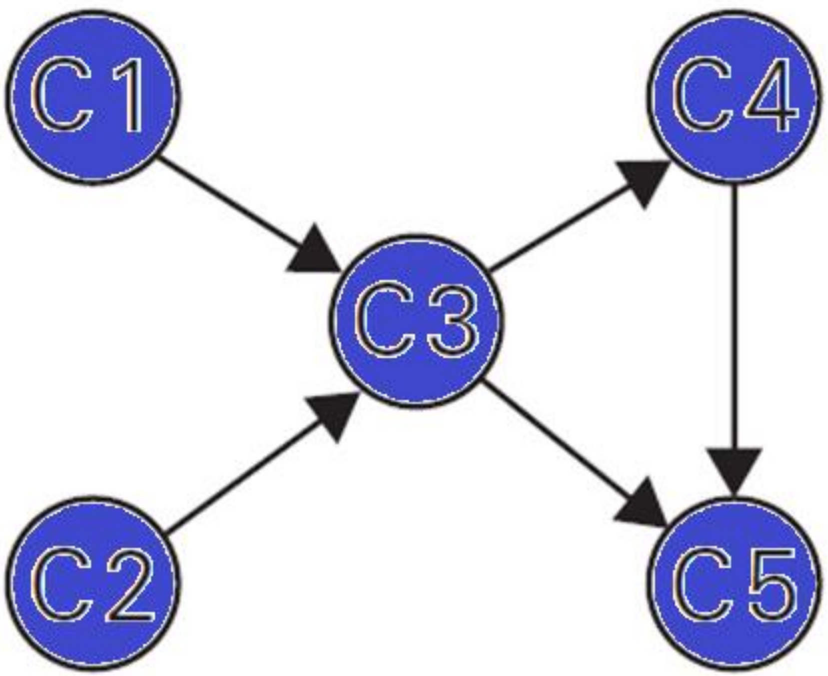
TOPOLOGICAL ORDERING

STACK



PUSH C2

TOPOLOGICAL ORDERING



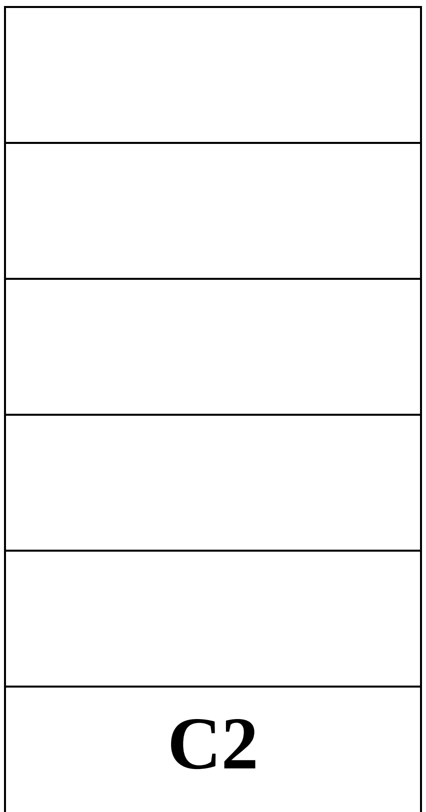
STACK

C2

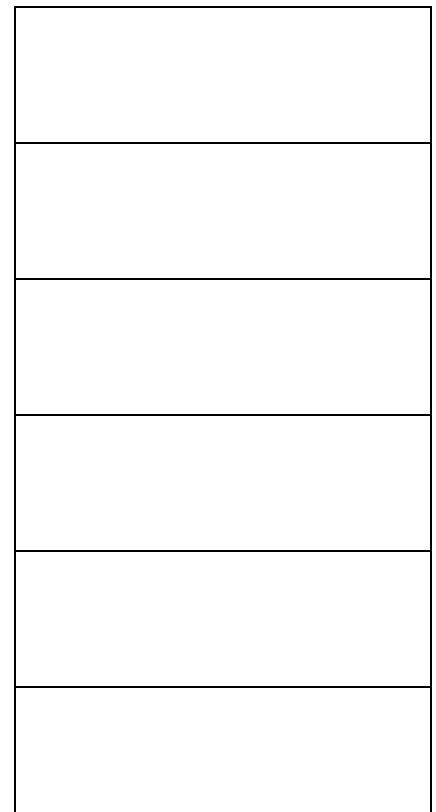
TOPOLOGICAL ORDERING

STACK

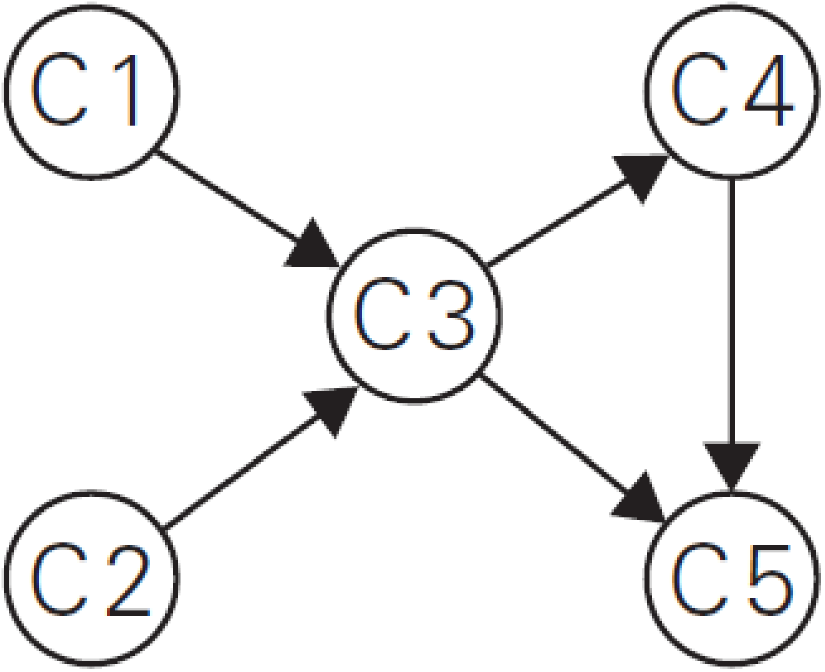
EMPTY STACK



POP C2



TOPOLOGICAL ORDERING



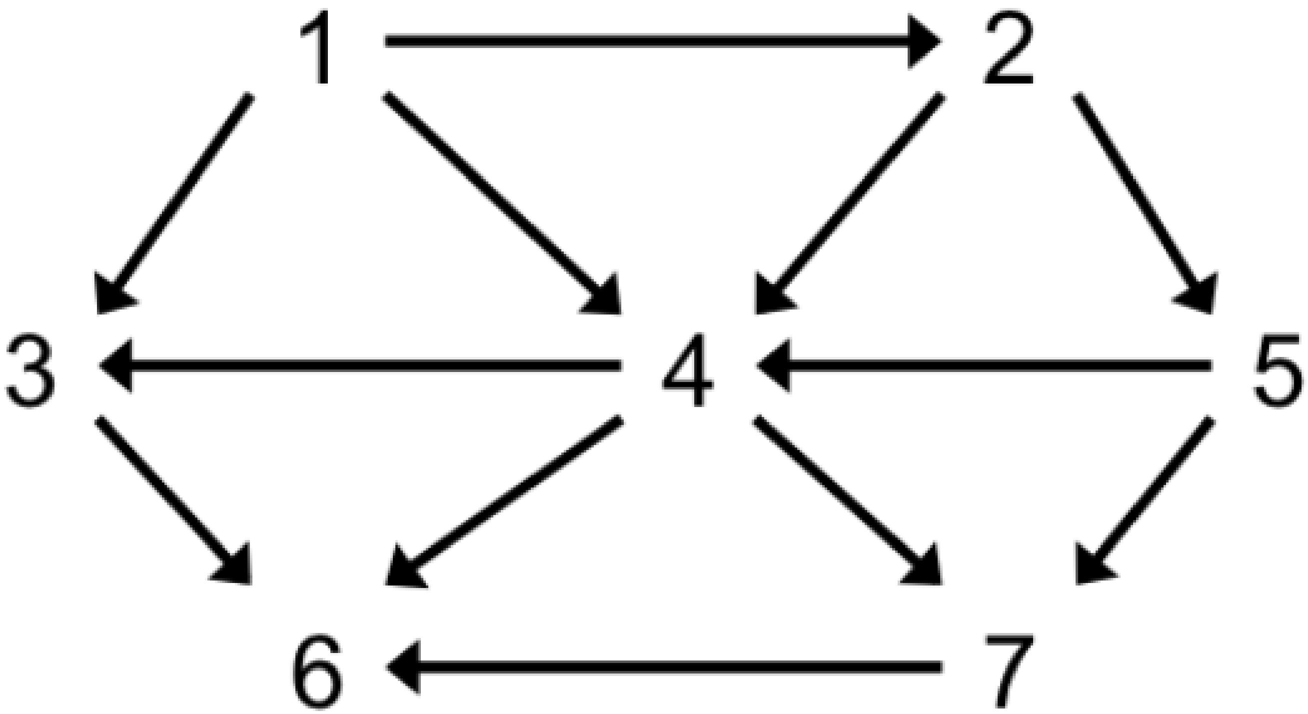
POPPING OFF ORDER

C5
C4
C3
C1
C2

Topologically Sorted List:
C2 → C1 → C3 → C4 → C5

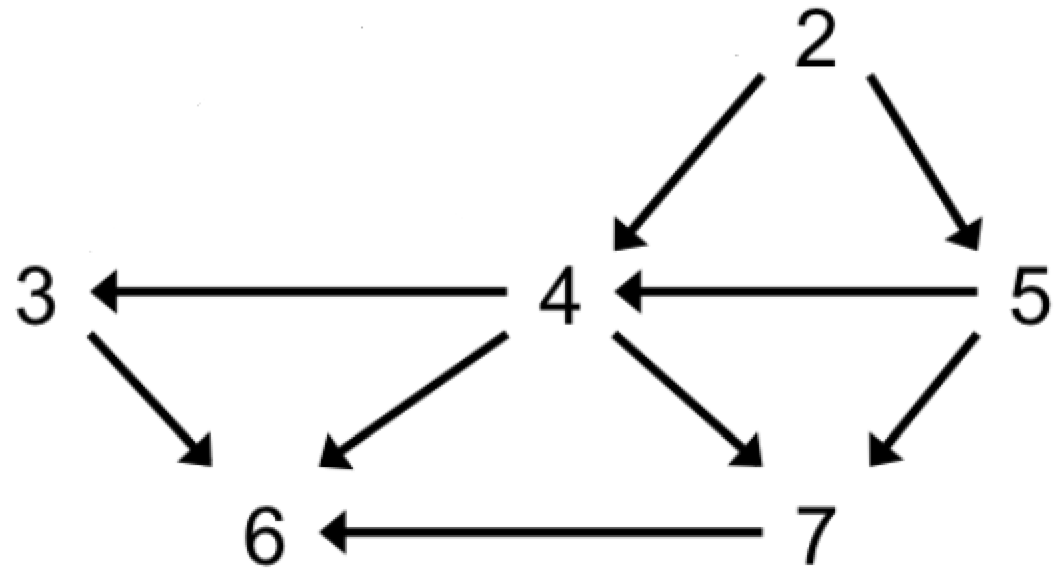
TOPOLOGICAL ORDERING

Method 1: Using Source - Removal



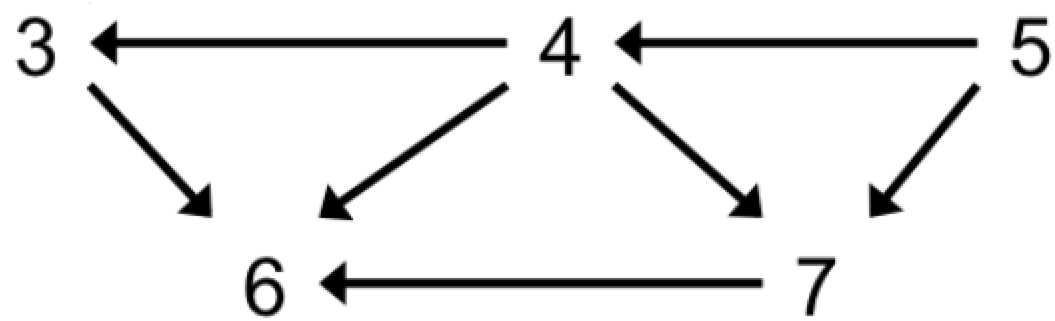
TOPOLOGICAL ORDERING

Delete Vertex 1



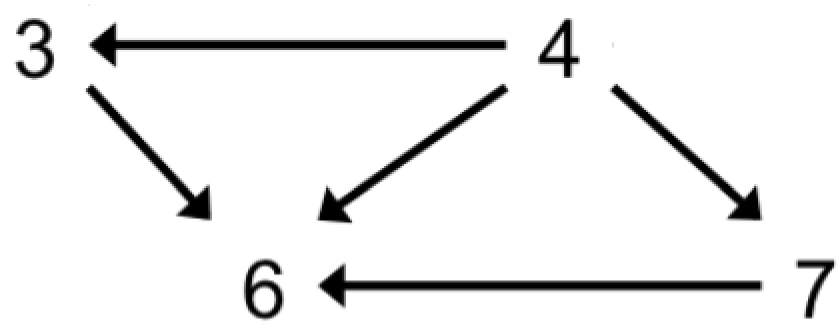
TOPOLOGICAL ORDERING

Delete Vertex 2



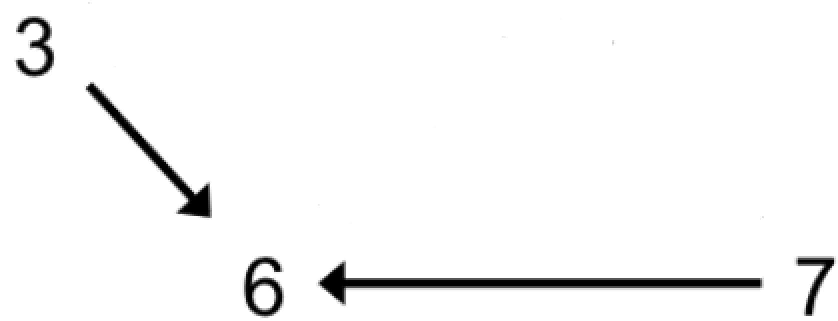
TOPOLOGICAL ORDERING

Delete Vertex 5



TOPOLOGICAL ORDERING

Delete Vertex 4



TOPOLOGICAL ORDERING

Delete Vertex 3

TOPOLOGICAL ORDERING

Delete Vertex 7

Delete Vertex 6

Solution: 1 2 5 4 3 7 6

ANALYSIS OF TOPOLOGICAL ORDERING

ANALYSIS

- If G has a topological ordering, then G is a DAG.

Proof

- Suppose, by way of contradiction, that G has a topological ordering v1, v2, . . . , vn, and also has a cycle C.
- Let vi be the lowest-indexed node on C, and let vj be the node on C just before vi—thus (vj , vi) is an edge.
- But by our choice of i, we have j > i, which contradicts the assumption that v1, v2, . . . , vn was a topological ordering.

ANALYSIS

- If G is a DAG, then G has a topological ordering.

Proof

- Since G is a DAG, there is a node v with no incoming edges.

- We place v first in the topological ordering; this is safe, since all edges out of v will point forward.

ANALYSIS

- Now $G-\{v\}$ is a DAG, since deleting v cannot create any cycles that weren't there previously.

- Also, $G-\{v\}$ has n-1 nodes, so we can apply the induction hypothesis to obtain a topological ordering of $G-\{v\}$.

- We append the nodes of $G-\{v\}$ in this order after v; this is an ordering of G in which all edges point forward, and hence it is a topological ordering.

THANK YOU