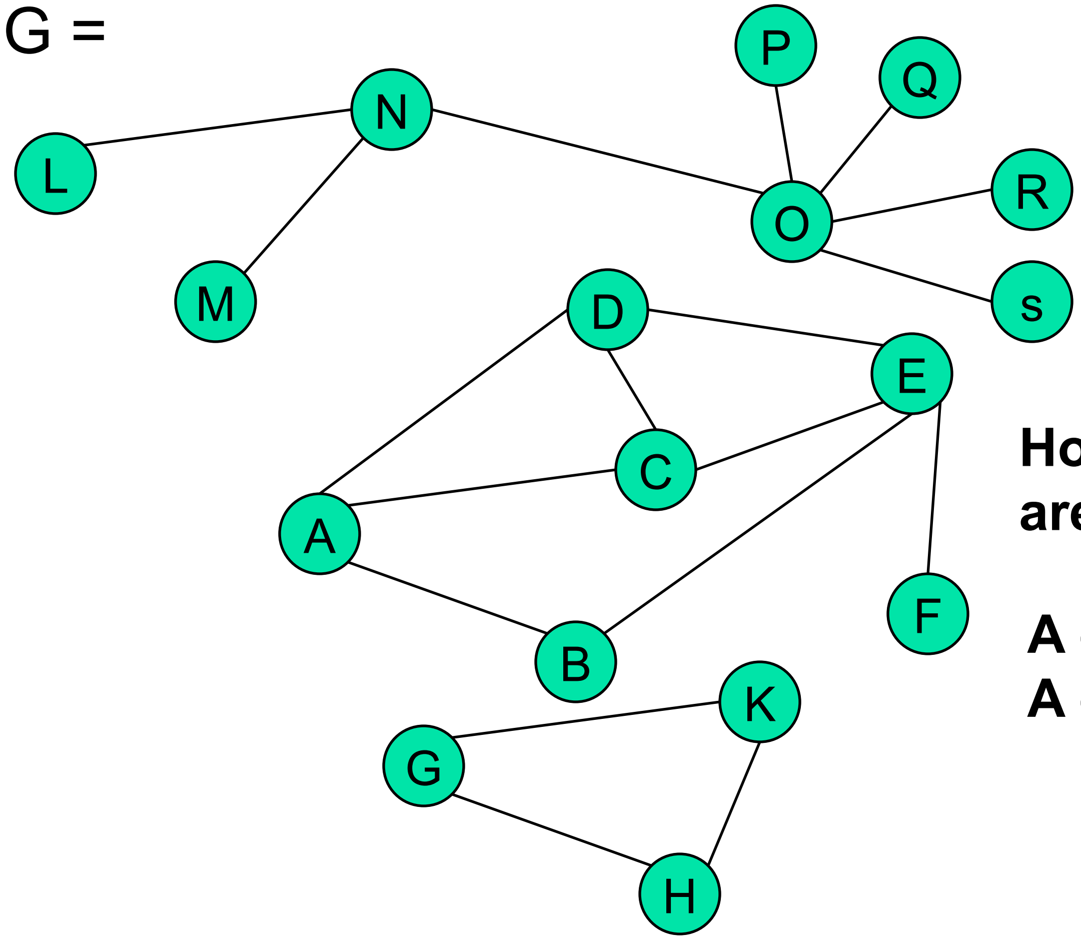
COMP2010
Data Structures and Algorithms

Lecture 17: Connected Components, Directed Graphs, Topological Sort

Department of Computer Science & Technology

United International College

Graph Application: Connectivity

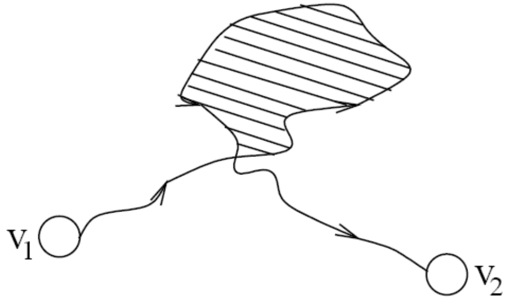


How do we tell if two vertices are connected?

A connected to F?
A connected to L?

Connectivity

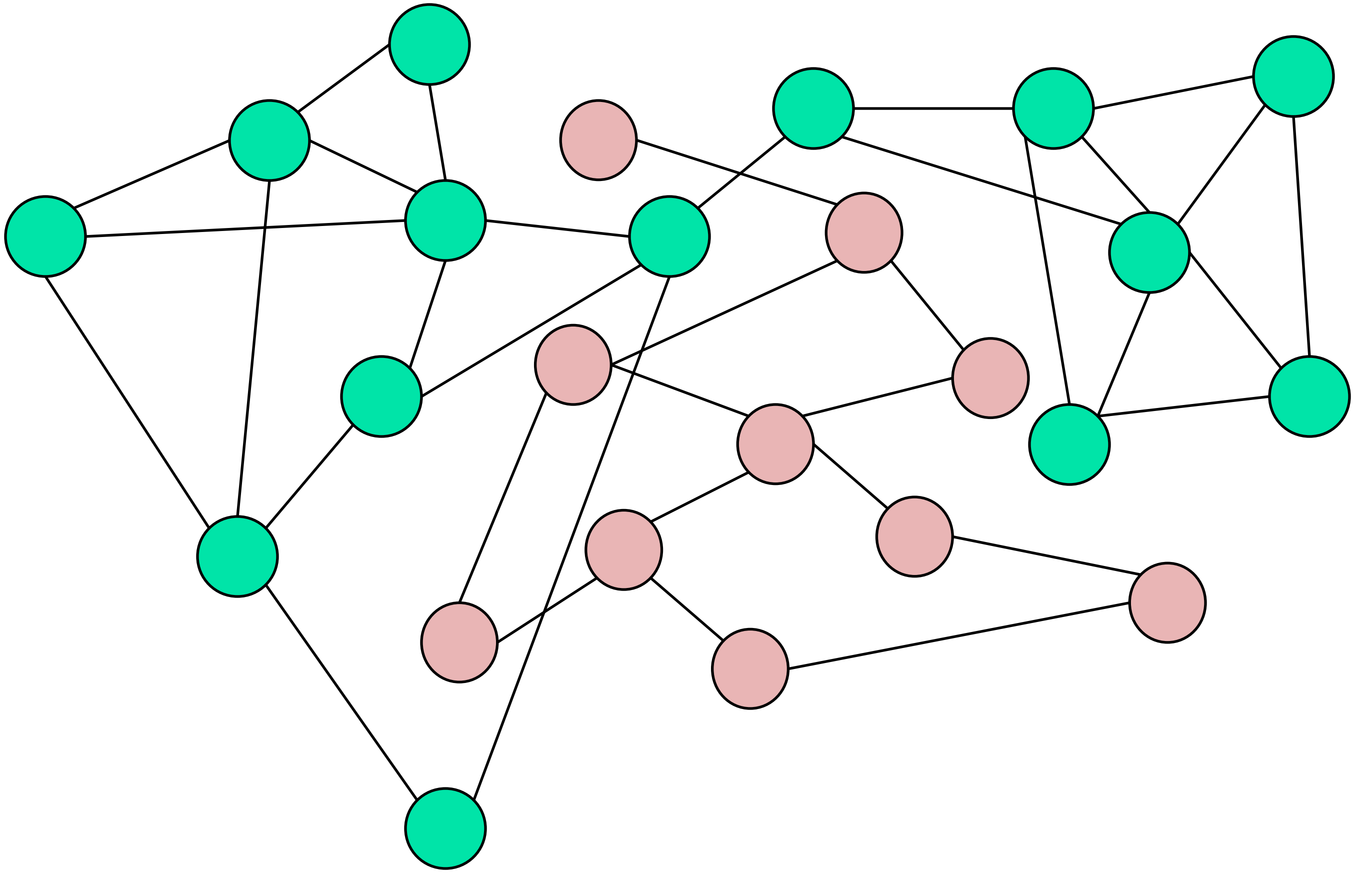
- A graph is *connected* if and only if there exists a path between every pair of distinct vertices.



- A graph is connected if and only if there exists a simple path between every pair of distinct vertices
  - since every non-simple path contains a cycle, which can be bypassed

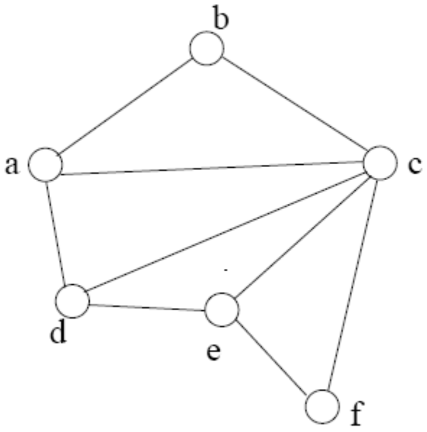
- How to check for connectivity?
  - Run BFS or DFS (using an arbitrary vertex as the source)
  - If all vertices have been visited, the graph is connected.
  - Running time? O(n + m)

Connected Components

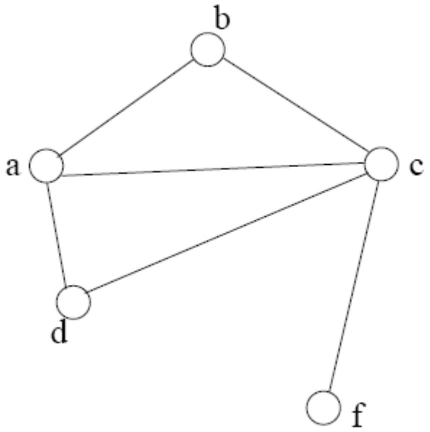


Subgraphs

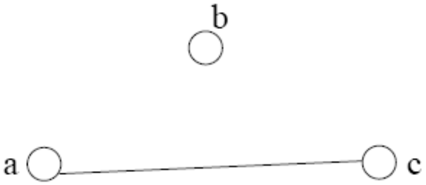
A graph $H(V_H, E_H)$ is a *subgraph* of $G(V_G, E_G)$ if and only if $V_H \subset V_G$ and $E_H \subset E_G$.



graph G



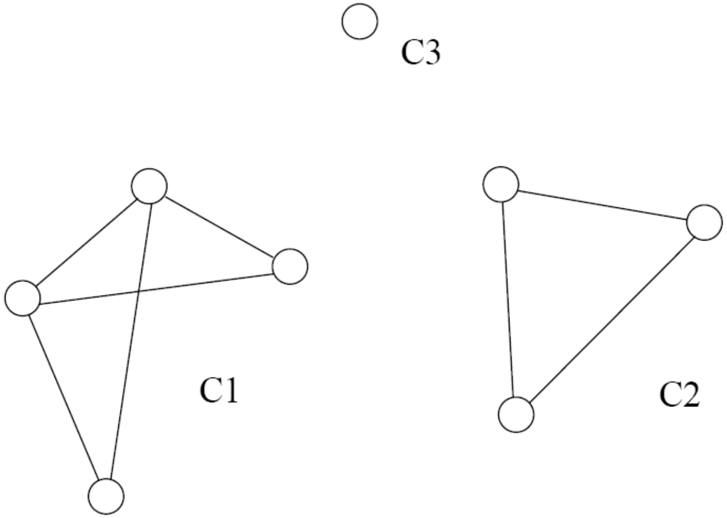
subgraph H$_1$



subgraph H$_2$

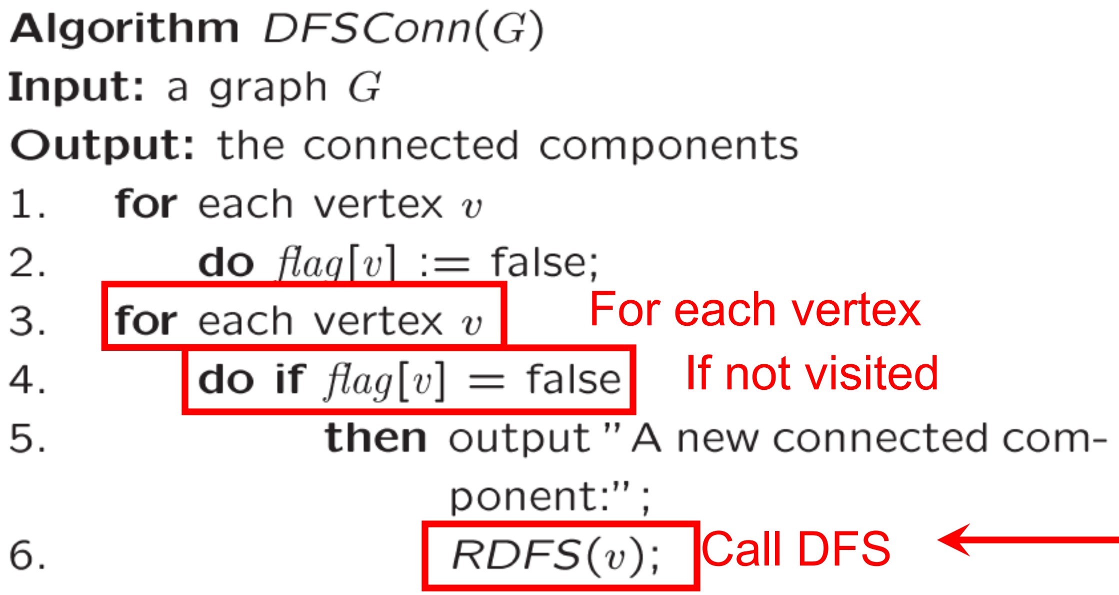
Connected Components

- Formal definition
  - A connected component is a **maximal** connected subgraph of a graph
- The set of connected components is unique for a given graph



3 components: C1, C2, and C3

Finding Connected Components



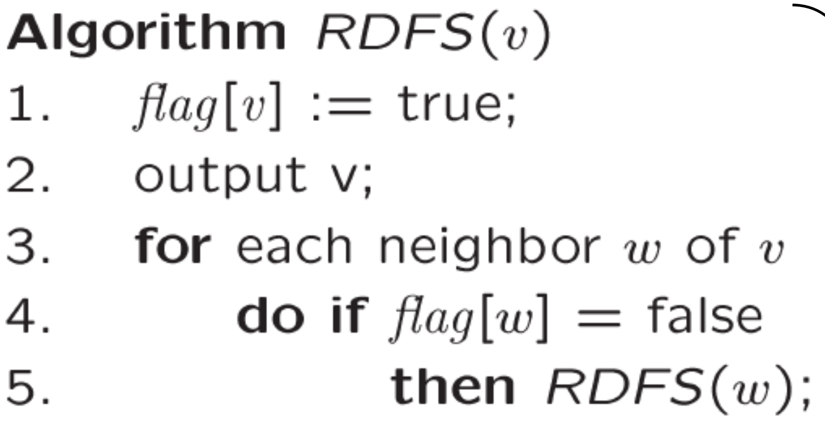
```
Algorithm $DFSConn(G)$
Input: a graph $G$
Output: the connected components
for each vertex $v$
    do $flag[v]$ := false;
for each vertex $v$
    do if $flag[v] = $ false
        then output "A new connected component:";
            $RDFS(v)$;
```

For each vertex

If not visited

Call DFS

This will find all vertices connected to “v” => one connected component



```
Algorithm $RDFS(v)$
$flag[v]$ := true;
output v;
for each neighbor $w$ of $v$
    do if $flag[w] = $ false
        then $RDFS(w)$;
```

Basic DFS algorithm

Time Complexity

- Running time for each *i* connected component

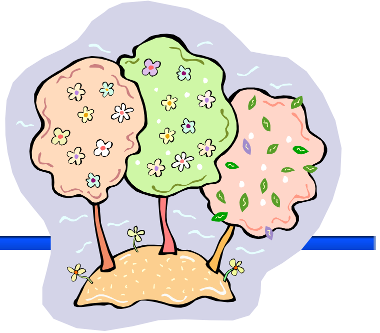
$$O(n_i + m_i)$$

- Running time for the graph G

$$\sum_i O(n_i + m_i) = O(\sum_i n_i + \sum_i m_i) = O(n + m)$$

- Reason: Can two connected components share
  - the same edge?
  - the same vertex?

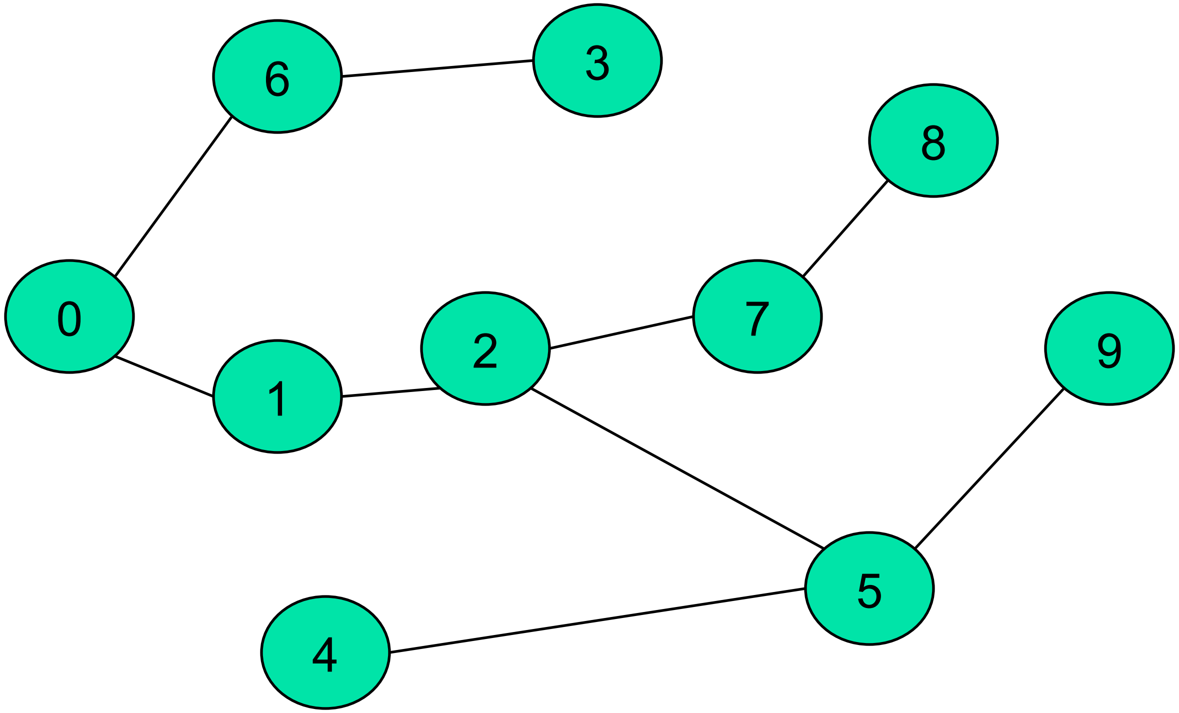
Trees



- Tree arises in many computer science applications
- A graph G is a tree if and only if it is connected and acyclic

  (Acyclic means it does not contain any simple cycles)
- The following statements are equivalent
  - G is a tree
  - G is connected and has exactly n-1 edges

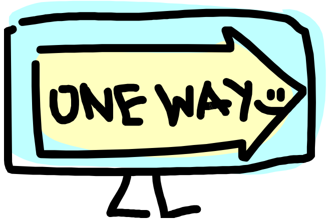
Tree Example



- Is it a graph?
- Does it contain cycles? In other words, is it acyclic?
- How many vertices?
- How many edges?

Directed Graph

- A graph is directed if direction is assigned to each edge.
- Directed edges are denoted as *arcs*.
  - Arc is an ordered pair (u, v)

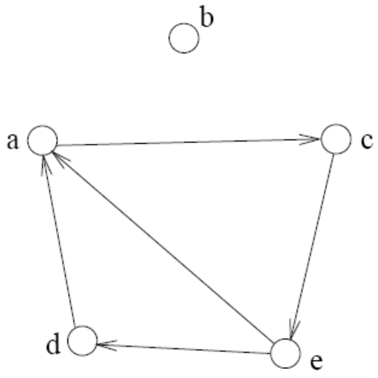


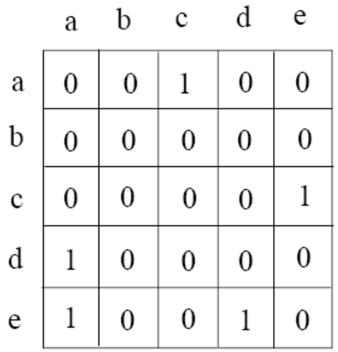
- Recall: for an undirected graph
  - An edge is denoted {u,v}, which actually corresponds to two arcs (u,v) and (v,u)

Representations

- The adjacency matrix and adjacency list can be used

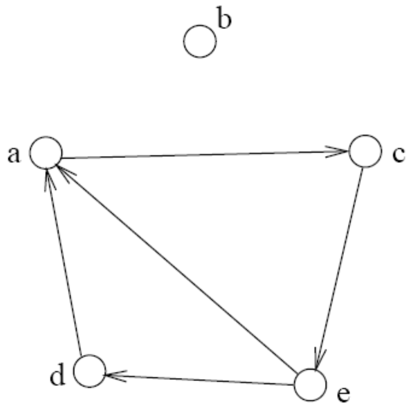
1. *Adjacency Matrix*

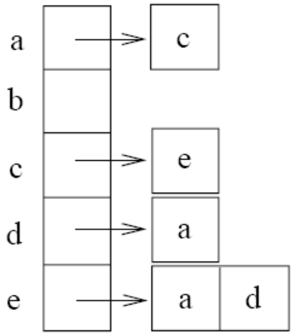




	a	b	c	d	e
a	0	0	1	0	0
b	0	0	0	0	0
c	0	0	0	0	1
d	1	0	0	0	0
e	1	0	0	1	0

2. *Adjacency List*





Directed Acyclic Graph

- A directed path is a sequence of vertices $(v_0, v_1, \ldots, v_k)$
  - Such that (v_i, v_{i+1}) is an *arc*

- *A* directed cycle is a directed path such that the first and last vertices are the same.

- A directed graph is acyclic if it does not contain any directed cycles

Indegree and Outdegree

- Since the edges are directed
  - We can't simply talk about Deg(v)
- Instead, we need to consider the arcs coming "in" and going "out"
  - Thus, we define terms Indegree(v), and Outdegree(v)
- Each arc(u,v) contributes count 1 to the outdegree of u and the indegree of v

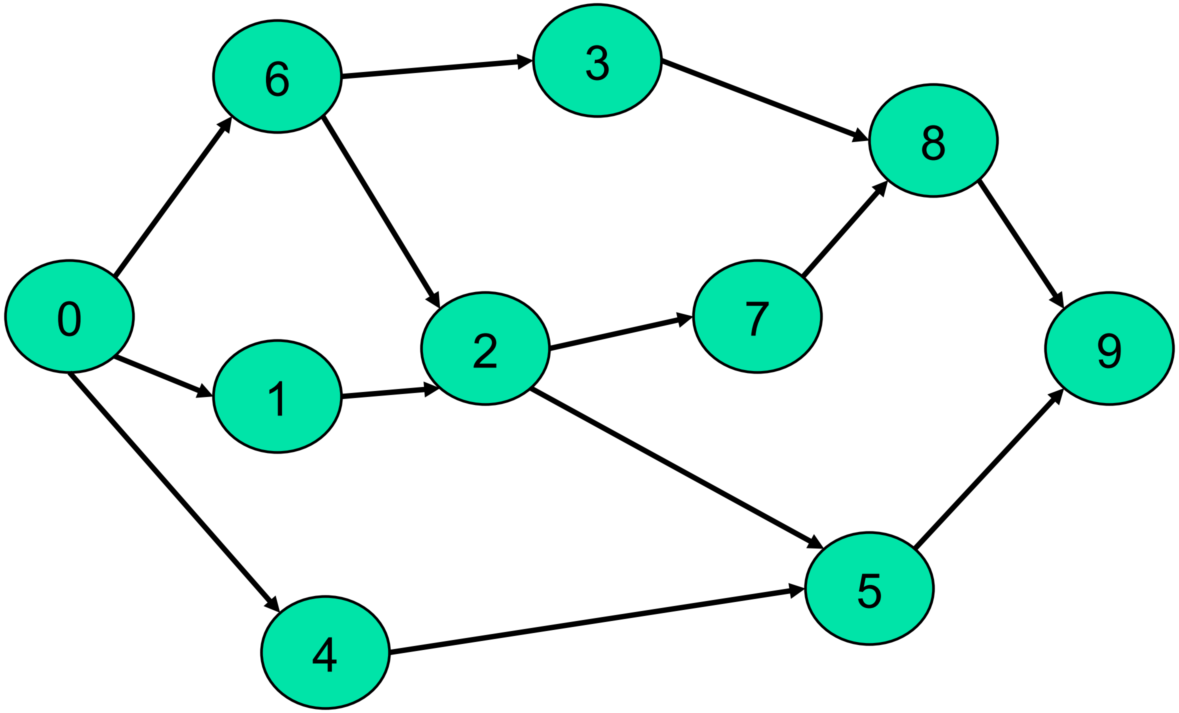
$$\sum_{\text{vertex } v} \text{indegree}(v) = \text{outdegree}(v) = m$$

Calculate Indegree and Outdegree

- Outdegree is simple to compute
  - Scan through list Adj[v] and count the arcs

- Indegree caculation
  - First, initialize indegree[v]=0 for each vertex v
  - Scan through adj[v] list for each v
    - For each vertex w seen, indegree[w]++;
    - Running time: O(n+m)

Example



Indeg(2)?

Indeg(8)?

Outdeg(0)?

Num of Edges?

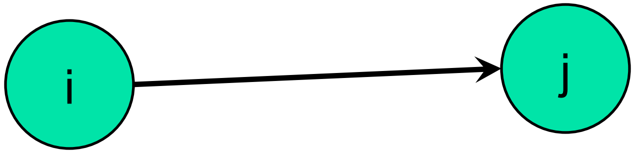
Total OutDeg?

Total Indeg?

Directed Graphs Usage

- Directed graphs are often used to represent order-dependent tasks
  - That is we cannot start a task before another task finishes

- We can model this task dependent constraint using *arcs*
- An *arc (i,j)* means *task j* cannot start until *task i* is finished

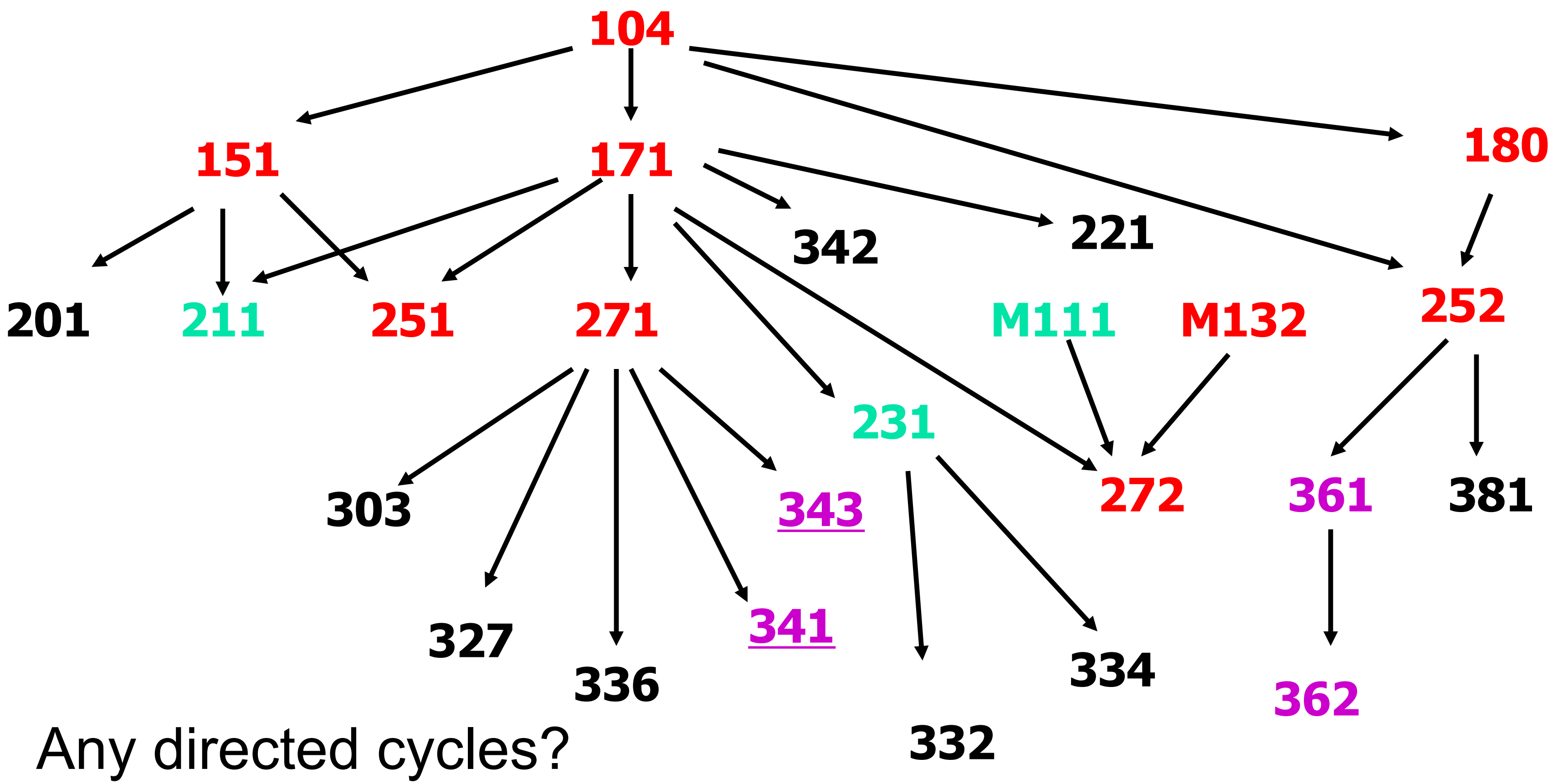


Task j cannot start
until task i is finished

- Clearly, for the system not to hang, the graph must be acyclic

University Example

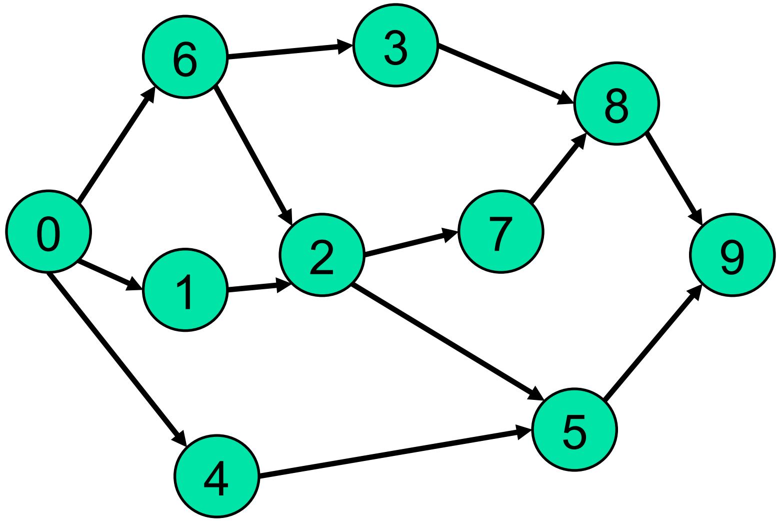
- CS departments course structure



Any directed cycles?
How many indeg(171)?
How many outdeg(171)?

Topological Sort

- Topological sort is an algorithm for a directed acyclic graph
- Linearly order the vertices so that the linear order respects the ordering relations implied by the arcs



For example:

0, 1, 2, 5, 9
0, 4, 5, 9
0, 6, 3, 7 ?

Topological Sort Algorithm

- Observations
  - Starting point must have zero indegree.
  - If it doesn’t exist, the graph would not be acyclic.
- Algorithm

1. A vertex with zero *indegree* is a task that can start right away. So we can output it first in the linear order.
2. If a vertex *i* is output, then its outgoing arcs *(i, j)* are no longer useful, since tasks *j* does not need to wait for *i* anymore- so remove all *i*’s outgoing arcs.
3. With vertex *i* removed, the new graph is still a directed acyclic graph. So, repeat steps 1-2 until no vertex is left.

Topological Sort

Algorithm *TSort*(G)

Input: a directed acyclic graph G

Output: a topological ordering of vertices

```
initialize $Q$ to be an empty queue;
for each vertex $v$
      do if $indegree(v) = 0$
            then $enqueue(Q, v)$;
while $Q$ is non-empty
      do $v := dequeue(Q)$;
            output $v$;
            for each arc $(v, w)$
                  do $indegree(w) = indegree(w) - 1$;
                  if $indegree(w) = 0$
                     then $enqueue(w)$;
```

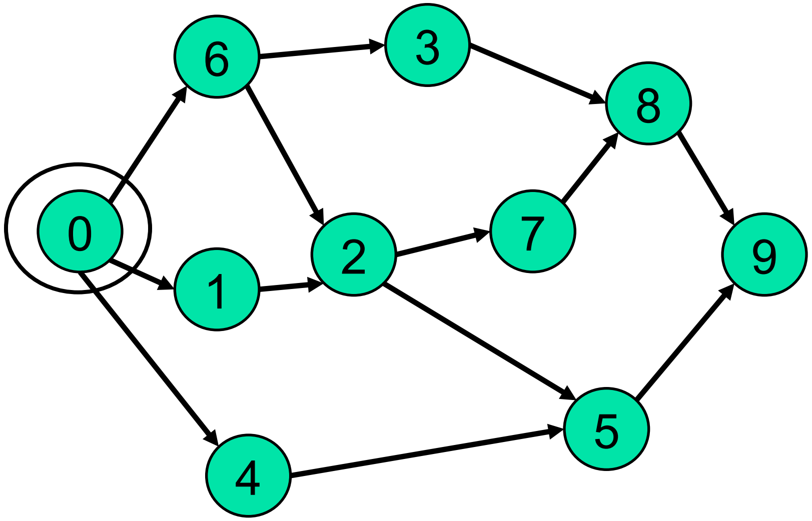
Find all starting points

Reduce indegree(w)

Place new start vertices on the Q

The running time is $O(n + m)$.

Example



0 → 6 1 4
1 → 2
2 → 7 5
3 → 8
4 → 5
5 → 9
6 → 3 2
7 → 8
8 → 9
9

Indegree

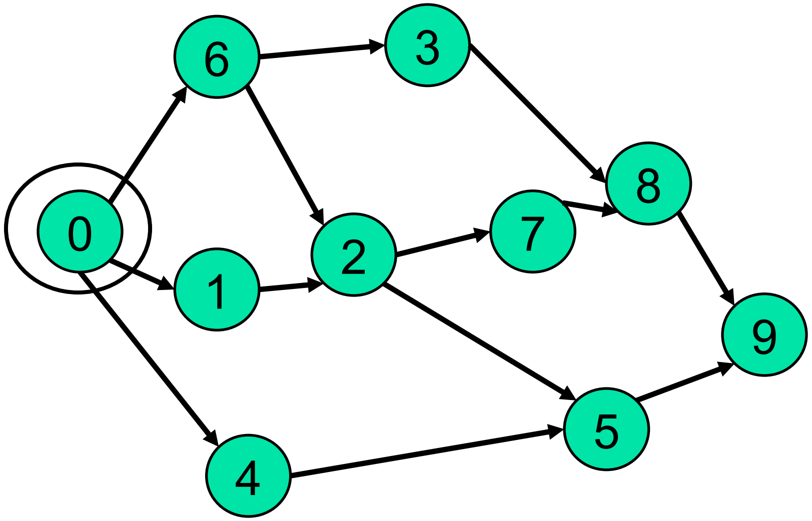
0	**0**
1	**1**
2	**2**
3	**1**
4	**1**
5	**2**
6	**1**
7	**1**
8	**2**
9	**2**

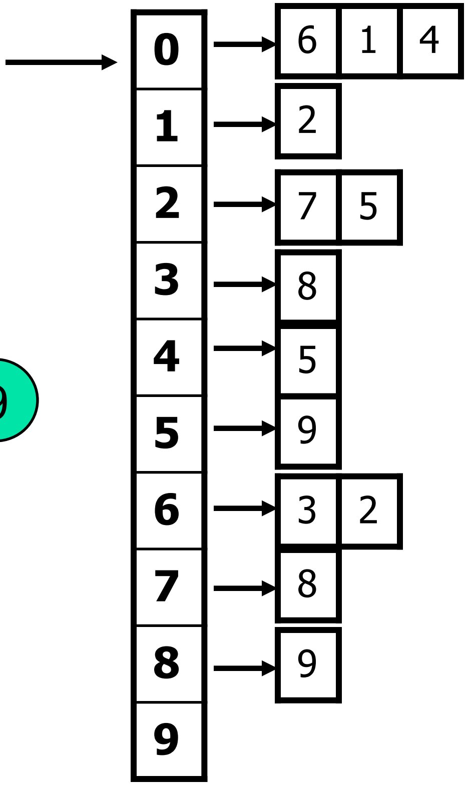
← start

Q = { 0 }

OUTPUT: 0

Example Cont'd





0	6, 1, 4
1	2
2	7, 5
3	8
4	5
5	9
6	3, 2
7	8
8	9
9	

Indegree

0	0	
1	1	-1
2	2	
3	1	
4	1	-1
5	2	
6	1	-1
7	1	
8	2	
9	2	

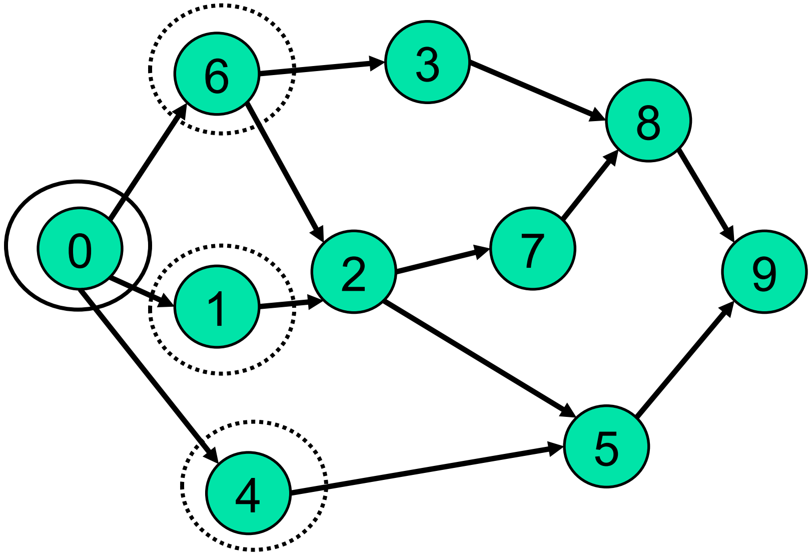
Dequeue 0 Q = { }

-> remove 0's arcs – adjust indegrees of neighbors (6, 1, 4)

OUTPUT: 0

Decrement 0's neighbors, which are 6, 1 and 4.

Example Cont’d



0	→ 6, 1, 4
1	→ 2
2	→ 7, 5
3	→ 8
4	→ 5
5	→ 9
6	→ 3, 2
7	→ 8
8	→ 9
9	

Indegree

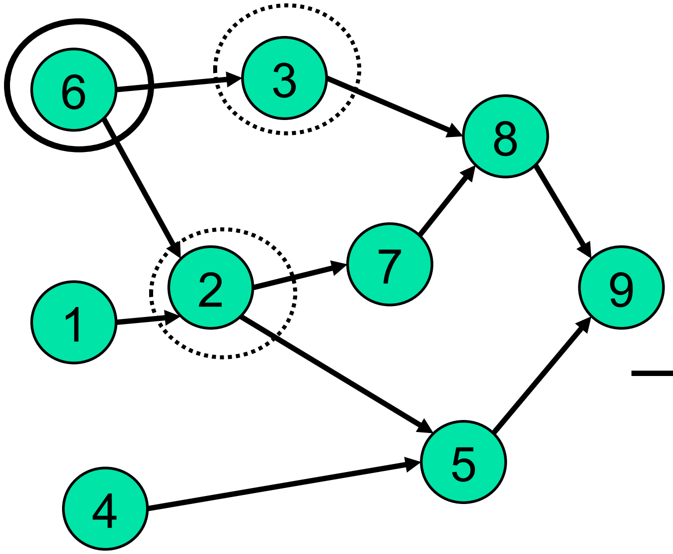
0	**0**	
1	**0**	←
2	**2**	
3	**1**	
4	**0**	←
5	**2**	
6	**0**	←
7	**1**	
8	**2**	
9	**2**	

Q = { 6, 1, 4 }
Enqueue all starting points

OUTPUT: 0

Enqueue all
new starting points

Example Cont'd



0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
8	9		
9			

Indegree

0	**0**	
1	**0**	
2	**2**	-1
3	**1**	-1
4	**0**	
5	**2**	
6	**0**	
7	**1**	
8	**2**	
9	**2**	

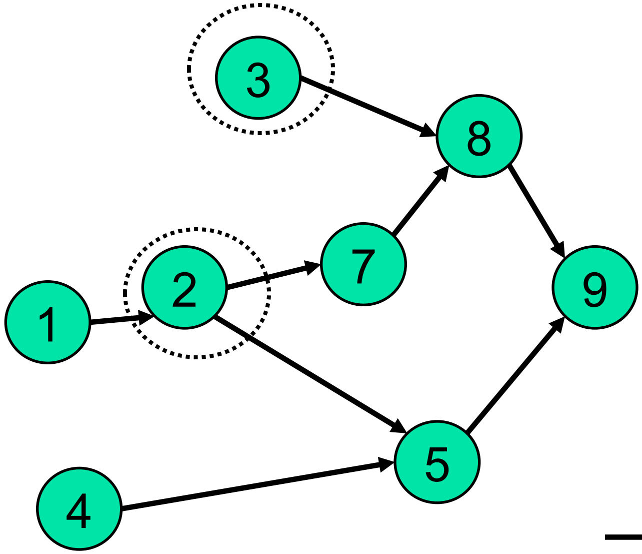
Dequeue 6 Q = { 1, 4 }

Remove arcs .. Adjust indegrees of neighbors, 3 and 2

OUTPUT: 0 6

Adjust indegrees of neighbors, which are 3 and 2

Example Cont'd



0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

Indegree

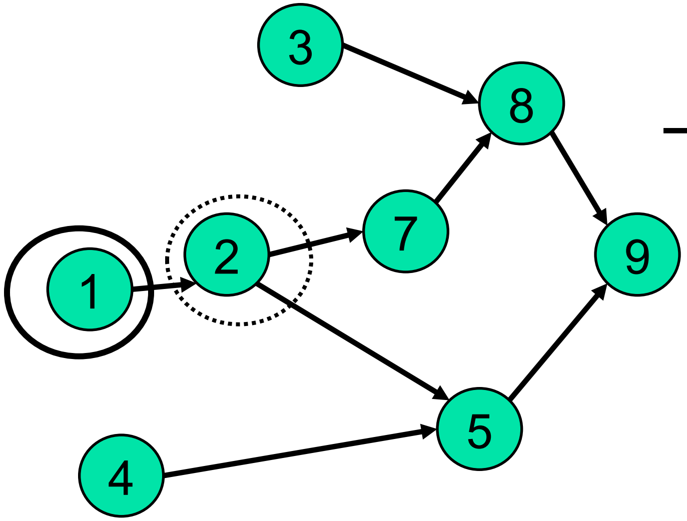
0	**0**
1	**0**
2	**1**
3	**0**
4	**0**
5	**2**
6	**0**
7	**1**
8	**2**
9	**2**

Q = { 1, 4, 3 }
Enqueue 3

OUTPUT: 0 6

Enqueue new starting point, which is 3 alone.

Example Cont’d



0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
8	9		
9			

Indegree

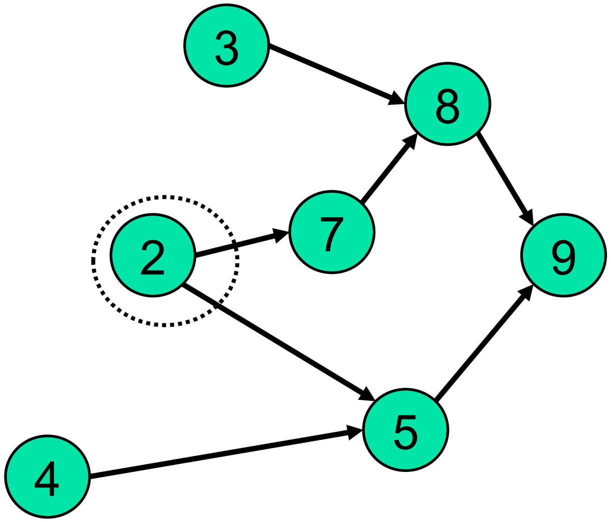
0	**0**	
1	**0**	
2	**1**	-1
3	**0**	
4	**0**	
5	**2**	
6	**0**	
7	**1**	
8	**2**	
9	**2**	

Dequeue 1 Q = { 4, 3 }
Adjust indegree of neighbor

OUTPUT: 0 6 1

Adjust indegree of neighbor, which is 2 alone.

Example Cont’d



0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
8	9		
9			

Indegree

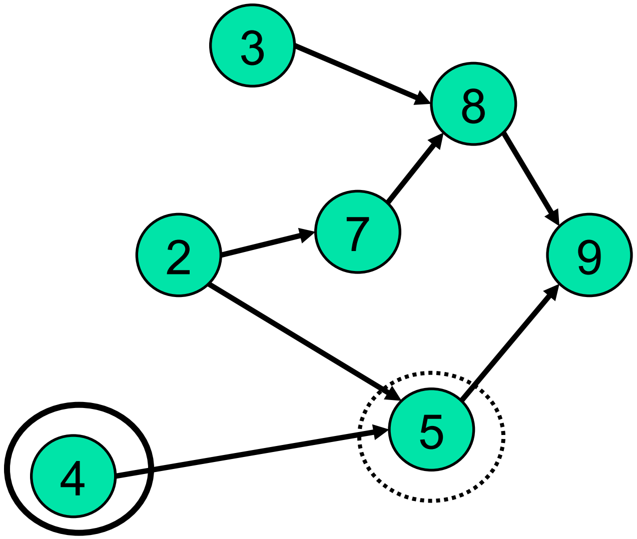
0	**0**
1	**0**
2	**0**
3	**0**
4	**0**
5	**2**
6	**0**
7	**1**
8	**2**
9	**2**

Dequeue 1 Q = { 4, 3, 2 }
Enqueue 2

OUTPUT: 0 6 1

Enqueue new starting point, which is 2 alone.

Example Cont’d



0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
8	9		
9			

Indegree

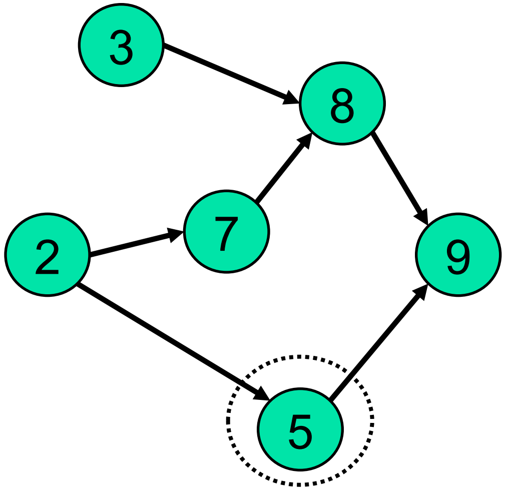
0	**0**	
1	**0**	
2	**0**	
3	**0**	
4	**0**	
5	**2**	-1
6	**0**	
7	**1**	
8	**2**	
9	**2**	

Dequeue 4 Q = { 3, 2 }
Adjust indegree of neighbor

OUTPUT: 0 6 1 4

Adjust 4’s neighbor, which is 5

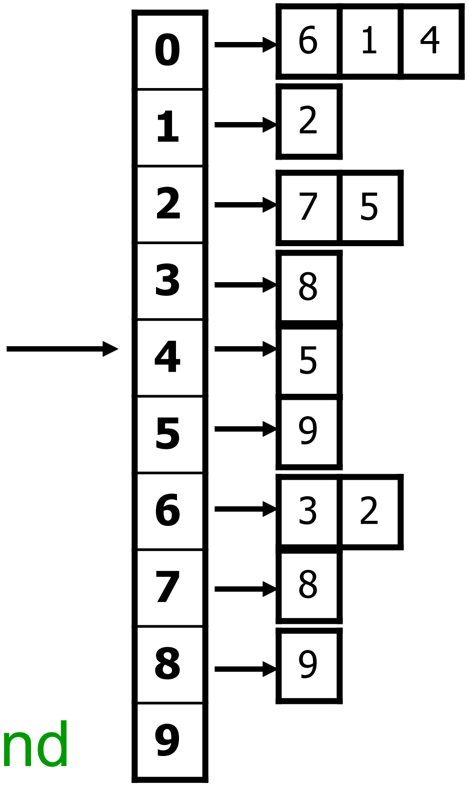
Example Cont’d



Dequeue 4 Q = { 3, 2 }

No new starting point found

OUTPUT: 0 6 1 4



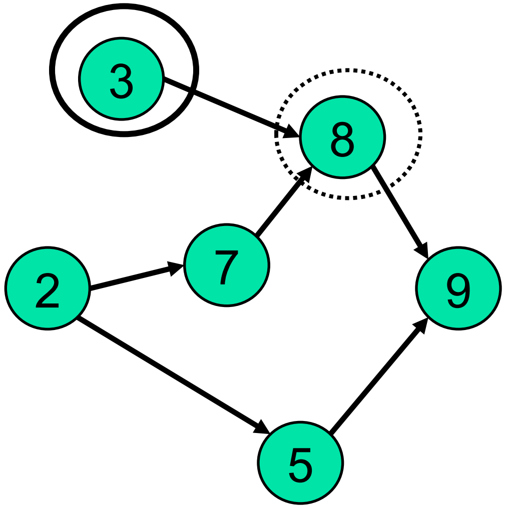
0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

Indegree

0	0
1	0
2	0
3	0
4	0
5	1
6	0
7	1
8	2
9	2

NO new starting point

Example Cont'd



0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
8	9		
9			

Indegree

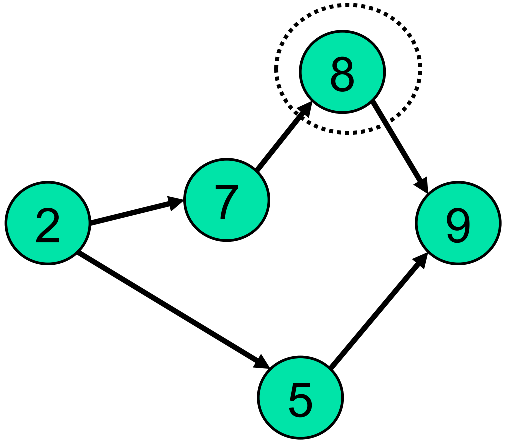
0	**0**	
1	**0**	
2	**0**	
3	**0**	
4	**0**	
5	**1**	
6	**0**	
7	**1**	
8	**2**	
9	**2**	-1

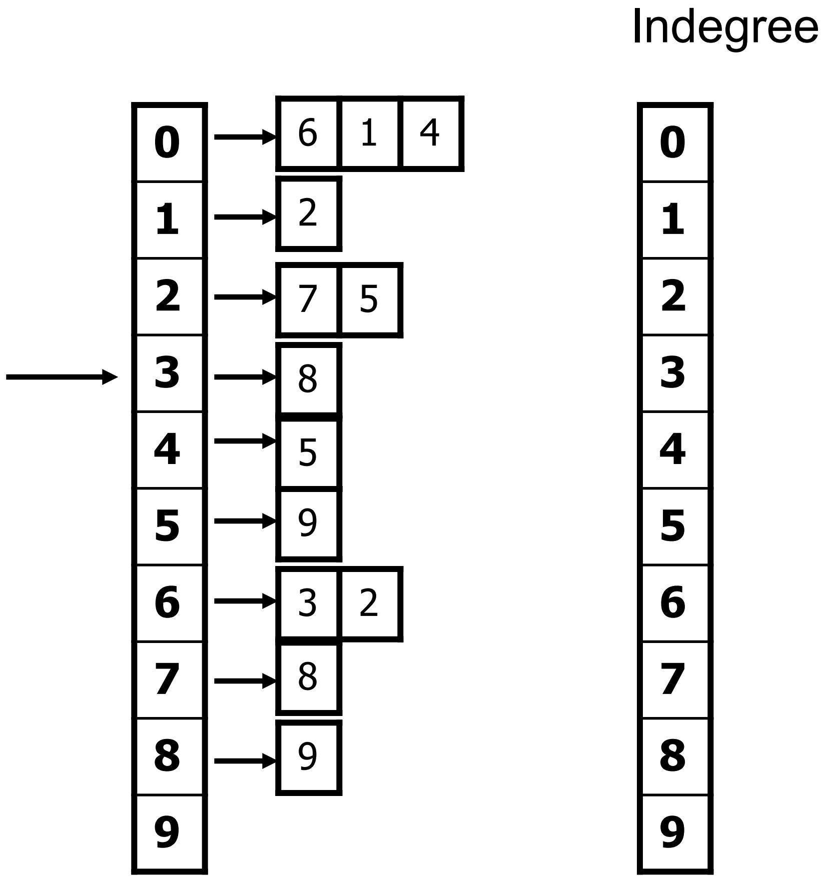
Dequeue 3 Q = { 2 }

Adjust 3's neighbor, which is 8 alone.

OUTPUT: 0 6 1 4 3

Example Cont’d




Indegree

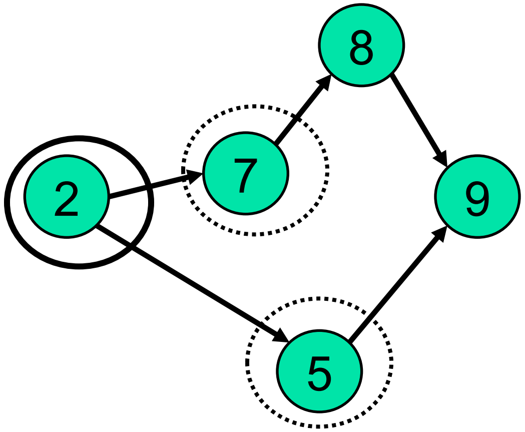
0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

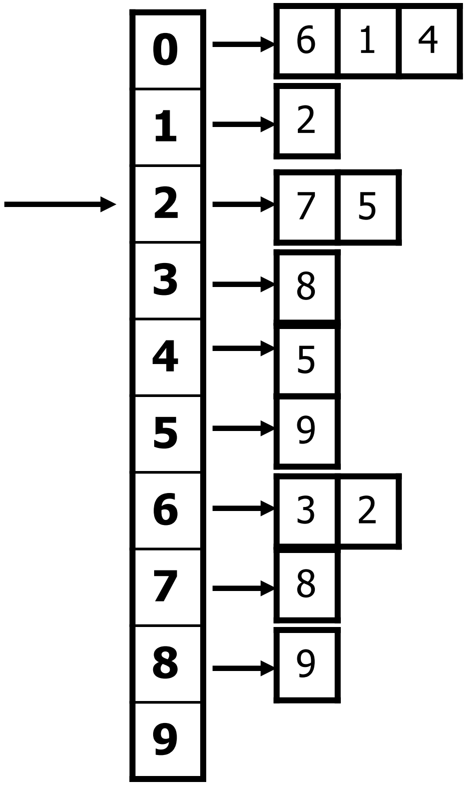
0
1
2
3
4
5
6
7
8
9

Dequeue 3 Q = { 2 }
No new starting point found

OUTPUT: 0 6 1 4 3

Example Cont’d





0 → 6 1 4

1 → 2

2 → 7 5

3 → 8

4 → 5

5 → 9

6 → 3 2

7 → 8

8 → 9

9

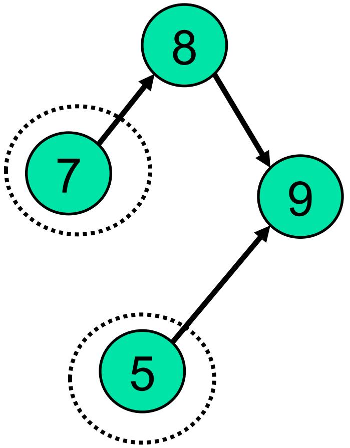
Indegree

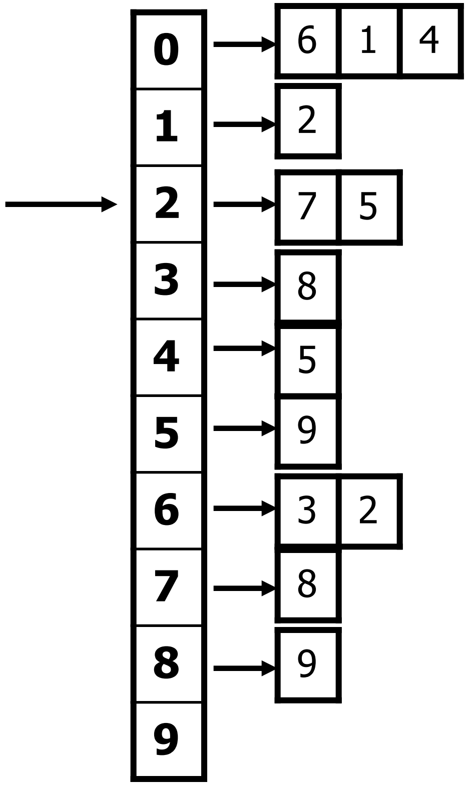
0	0	
1	0	
2	0	
3	0	
4	0	
5	1	-1
6	0	
7	1	-1
8	1	
9	2	

Dequeue 2 Q = { }
Adjust 2’s neighbors, which are 7 and 5.

OUTPUT: 0 6 1 4 3 2

Example Cont'd





0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

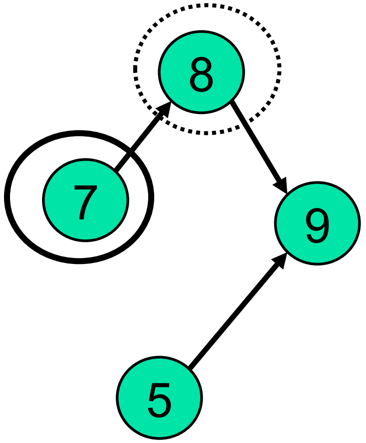
Indegree

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	2

Dequeue 2 Q = { 7, 5 }
Enqueue 7, 5

OUTPUT: 0 6 1 4 3 2

Example Cont'd



0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
→ 8			
9			

Indegree

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	2

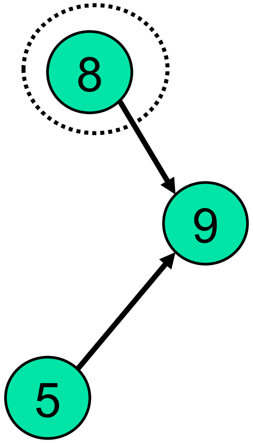
-1

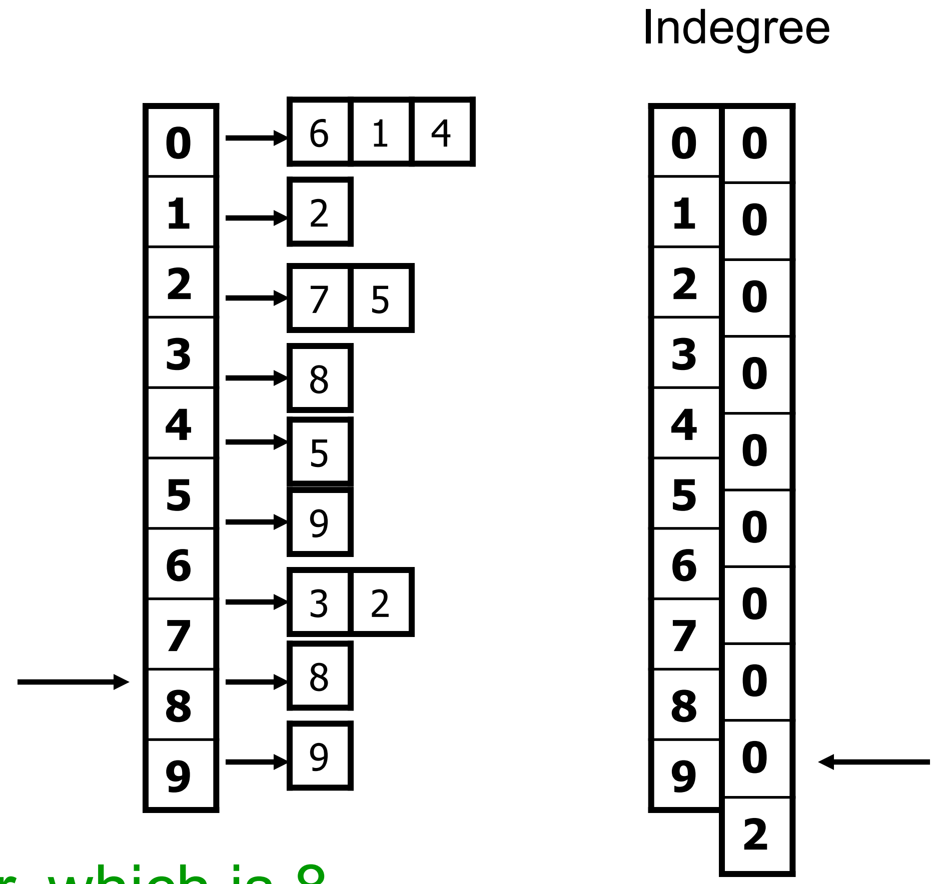
Dequeue 7 Q = { 5 }

Adjust indegree of neighbor, which is 8.

OUTPUT: 0 6 1 4 3 2 7

Example Cont’d





0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

Indegree

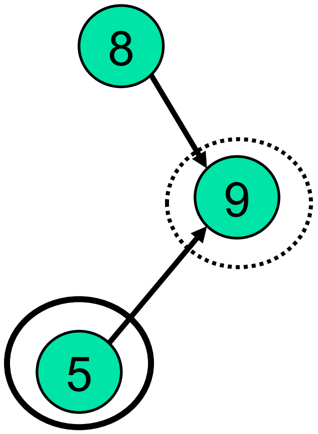
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	2

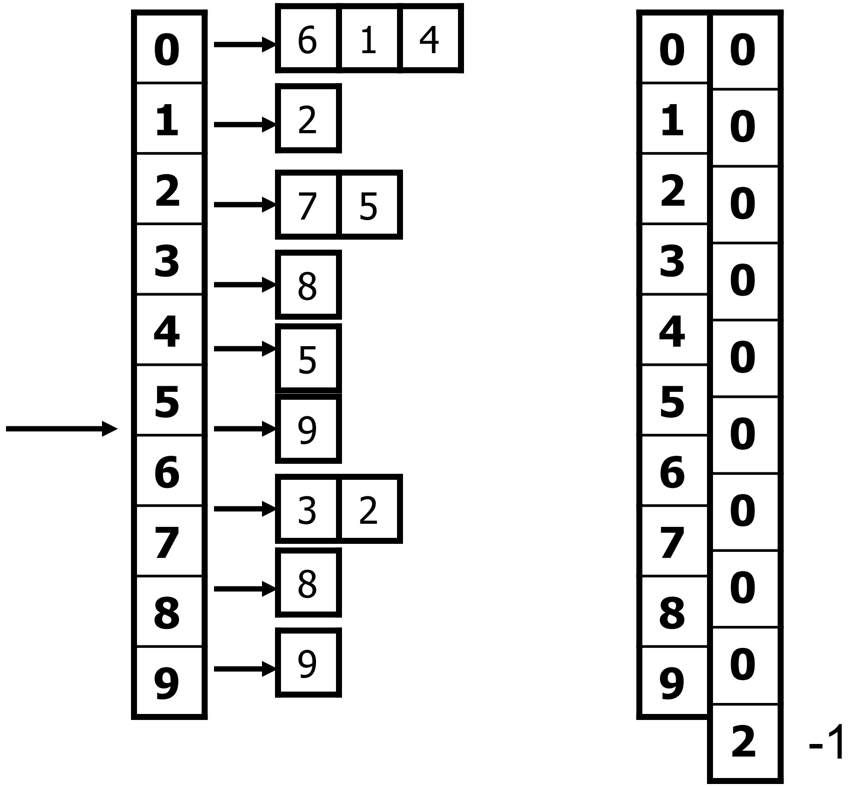
Dequeue 7 Q = { 5, 8 }

Adjust indegree of neighbor, which is 8.

OUTPUT: 0 6 1 4 3 2 7

Example Cont’d





0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

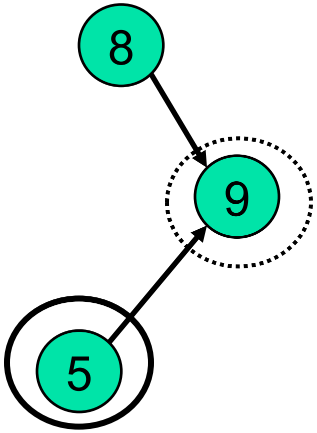
Indegree

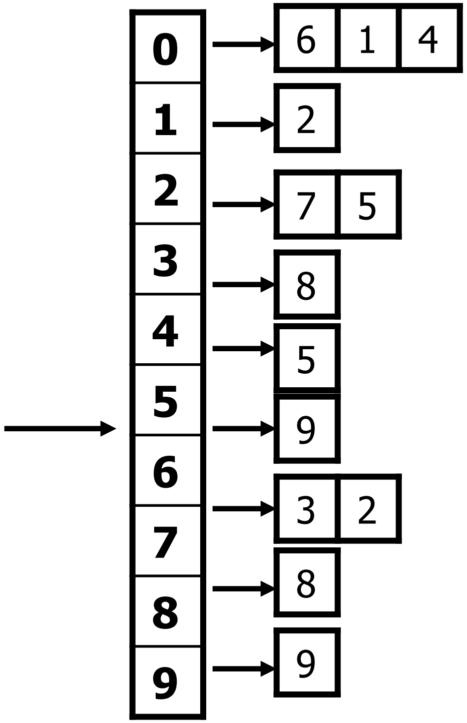
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
7	0	
8	0	
9	2	-1

Dequeue 5 Q = { 8 }
Adjust indegree of neighbor, which is 9.

OUTPUT: 0 6 1 4 3 2 7 5

Example Cont’d





0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
8	9		
9			

Indegree

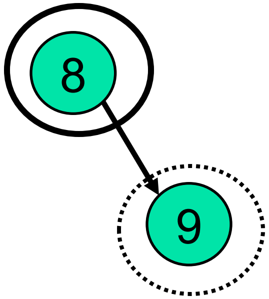
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	1

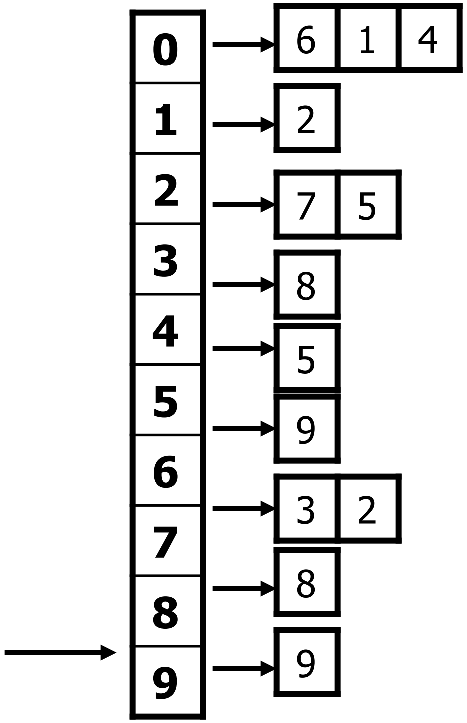
Dequeue 5 Q = { 8 }

No new starting point found

OUTPUT: 0 6 1 4 3 2 7 5

Example Cont'd





0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

Indegree

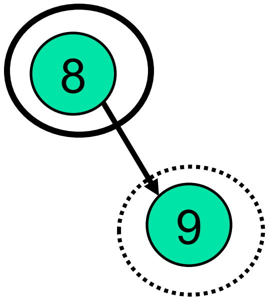
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
7	0	
8	0	
9	1	-1

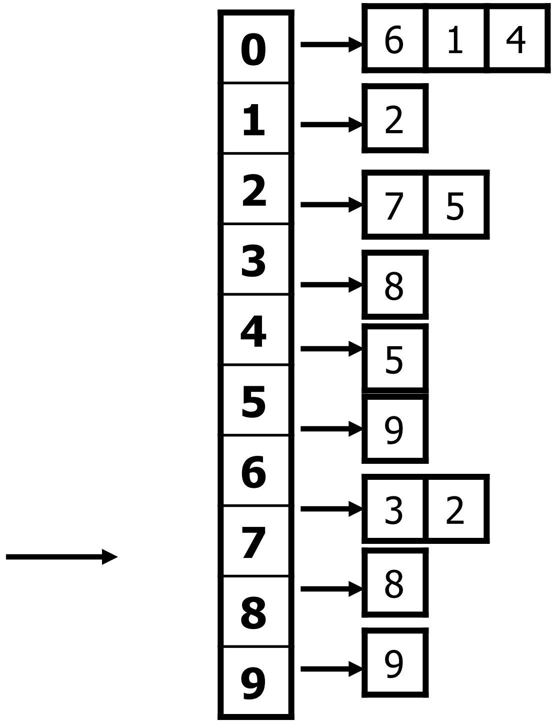
Dequeue 8 Q = { }

Adjust indegree of neighbor, which is 9.

OUTPUT: 0 6 1 4 3 2 7 5 8

Example Cont’d





0	6	1	4
1	2		
2	7	5	
3	8		
4	5		
5	9		
6	3	2	
7	8		
8	9		
9	9		

Indegree

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Dequeue 8 Q = { 9 }
Enqueue 9.

OUTPUT: 0 6 1 4 3 2 7 5 8

Example Cont’d

Dequeue 9 Q = { }

STOP – no neighbors

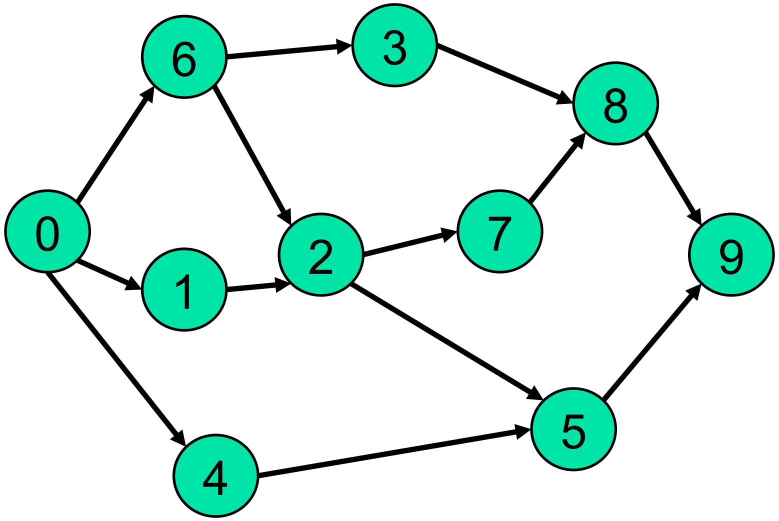
0	6 1 4
1	2
2	7 5
3	8
4	5
5	9
6	3 2
7	8
8	9
9	

Indegree

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

OUTPUT: 0 6 1 4 3 2 7 5 8 9

Example Cont'd



OUTPUT: 0 6 1 4 3 2 7 5 8 9

Is output topologically correct?

Topological Sort: Complexity

- We never visited a vertex more than one time.
- For each vertex,
  - we had to examine all outgoing edges,
  - it took time proportional to outdegree(v) + 1.
- Since it is summed over all vertices, the running time is O(n + m) if there are n vertices and m arcs.