DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

Lecture – 13
Graph – Graph Traversal & Shortest Path Algorithm

Instructor: Engr. Nasr Kamal

Department of Computer Science and Information Technology

Graph Algorithms

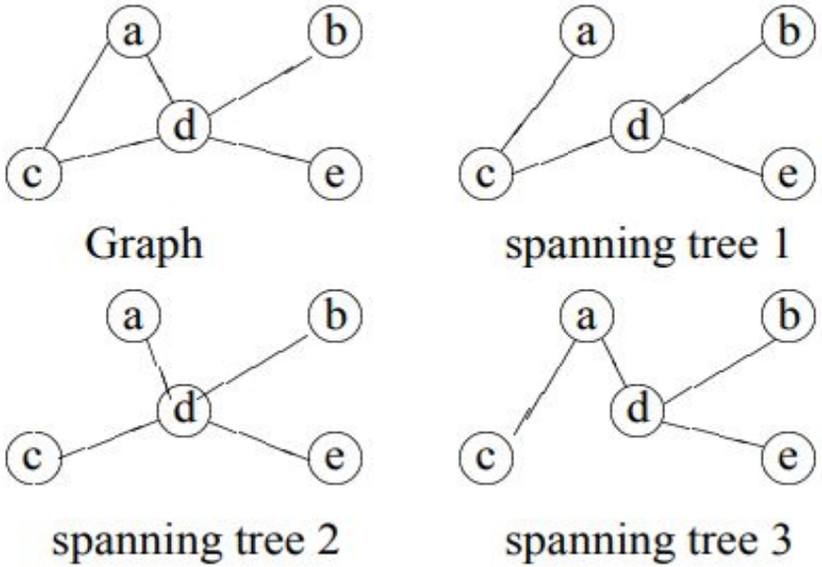
Graph algorithms solve problems like:

- Finding the shortest path between two points.
- Detecting cycles in a graph.
- Finding connected components.
- Determining the minimum cost to connect all vertices.

Spanning Tree

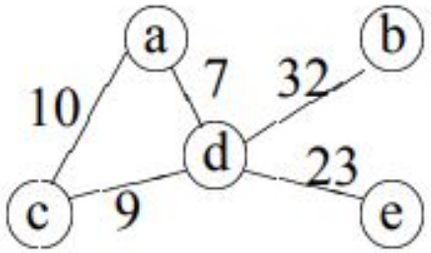
- Spanning Trees: A subgraph of a undirected graph is a spanning tree of if it is a tree and contains every vertex of G

Example:

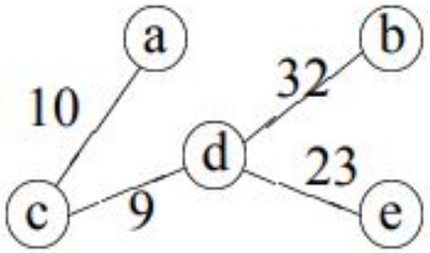


- Weighted Graphs: A weighted graph is a graph, in which each edge has a weight (some real number).
- Weight of a Graph: The sum of the weights of all edges.

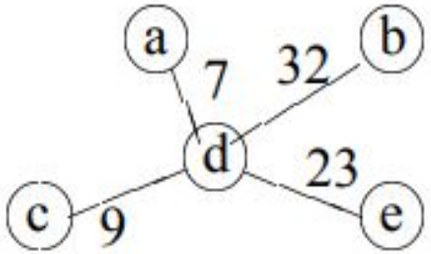
Example:



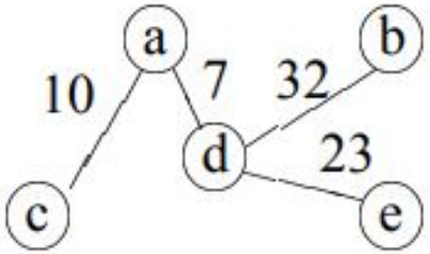
weighted graph



Tree 1. w=74



Tree 2, w=71



Tree 3, w=72

Minimum spanning tree

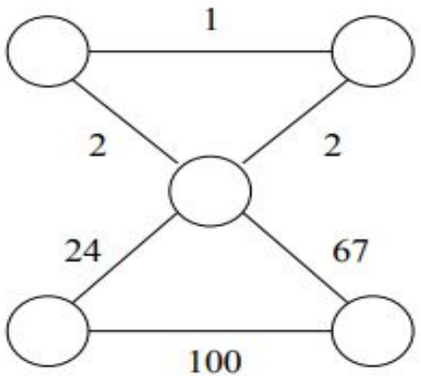
Minimum Spanning Tree

- A **Minimum Spanning Tree (MST)** is a subset of the edges of a connected, weighted graph that connects all the vertices together without forming any cycles, while minimizing the total weight of the edges. It is widely used in network design, such as designing computer networks, transportation routes, and electrical grids.

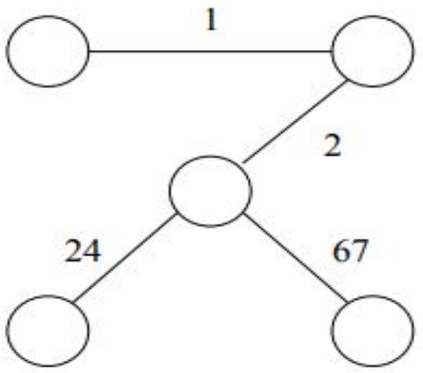
Minimum Spanning Tree

- The minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique (we won't prove this now).

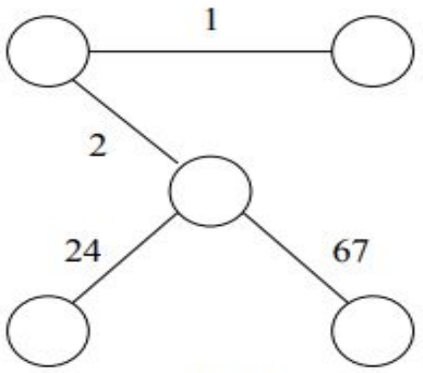
Example:



weighted graph



MST1



MST2

Key Characteristics of MST

Spanning Tree:

- It includes all vertices of the graph.
- It has exactly V−1V-1V−1 edges, where VVV is the number of vertices.

Minimum Weight:

- The sum of the weights of all edges in the MST is the smallest possible among all spanning trees.

No Cycles:

- An MST cannot contain any cycles, as cycles would mean some edges can be removed to reduce the weight further.

.

Example

- **Consider the following weighted graph:**
  - **Vertices: A, B, C, D, E**
  - **Edges:**
  - **A–B (4), A–C (4), B–C (2), B–D (6), C–D (8), C–E (9), D–E (5)**

MST Algorithms

There are two popular algorithms for finding MST: **Kruskal's Algorithm** and **Prim's Algorithm**. Let's use **Kruskal's Algorithm** for this example.

Steps in Kruskal's Algorithm:

1. Sort all edges by weight:
   (B–C: 2), (A–B: 4), (A–C: 4), (D–E: 5), (B–D: 6), (C–D: 8), (C–E: 9)
2. Start adding edges to the MST in order of increasing weight, ensuring no cycles are formed.

Kruskal's Algorithms

1. **Sort edges by weight**:
   - B–C (2), A–B (4), A–C (4), D–E (5), B–D (6), C–D (8), C–E (9).
2. **Initialize MST**: Start with an empty set of edges.
3. **Add edges to the MST (ensuring no cycles)**:
   - Add **B–C (2)** → No cycle.
   - Add **A–B (4)** → No cycle.
   - **Skip A–C (4)** → Adding this edge would form a cycle (A–B–C).
   - Add **D–E (5)** → No cycle.
   - Add **B–D (6)** → No cycle.
4. **Stop** when V–1V–1V–1 edges are added (5–1=45–1 = 45–1=4 edges).

Final MST:

Edges: **B–C (2), A–B (4), D–E (5), B–D (6)**

Total Weight: 2+4+5+6=172 + 4 + 5 + 6 = 172+4+5+6=17

Prim’s Algorithms

Definition

Prim's Algorithm is a **greedy algorithm** that builds the MST by starting with a single vertex and progressively adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.

Steps of Algorithm:

Choose a Starting Vertex: Begin with any vertex in the graph.

Select the Smallest Edge: Add the edge with the smallest weight that connects a vertex in the MST to a vertex outside the MST.

Repeat: Continue adding edges until all vertices are included in the MST.

Step by Step Execution

Vertices: A, B, C, D, E
Edges (with weights):
A–B (4), A–C (4), B–C (2), B–D (6), C–D (8), C–E (9), D–E (5)

1. **Start from Vertex A**.
   - Possible edges: A–B (4), A–C (4).
   - Pick A–B (4).
2. **Include Vertex B**.
   - Possible edges: A–C (4), B–C (2), B–D (6).
   - Pick B–C (2).
3. **Include Vertex C**.
   - Possible edges: A–C (already included), B–D (6), C–D (8), C–E (9).
   - Pick B–D (6).
4. **Include Vertex D**.
   - Possible edges: D–E (5), C–E (9).
   - Pick D–E (5).

Resulting MST:

Edges: A–B (4), B–C (2), B–D (6), D–E (5)

Total Weight: 4+2+6+5=17

Applications of Kruskal's and Prim's Algorithm

Network Design: Building efficient road networks, telecommunications grids, or power lines.

Cluster Analysis: Used in machine learning for clustering datasets.

Approximations: Both algorithms can help in problems requiring cost optimization.


Directed Graph

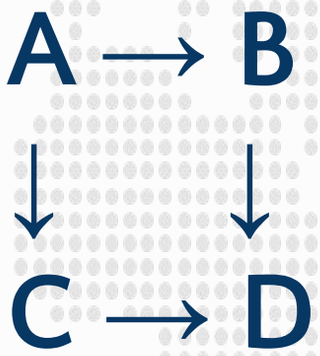
Directed Graph

- A **directed graph** (or **digraph**) is a type of graph in which the edges have a direction.
- In a directed graph, each edge connects an ordered pair of vertices, meaning it goes from one specific vertex to another specific vertex.
- This directionality distinguishes it from an undirected graph, where edges have no direction.

Example

Consider the following directed graph with vertices A, B, C, and D, and edges as follows:

- A -> B
- A -> C
- B -> D
- C -> D



Representation of Directed Graph

1. Adjacency Matrix Representation

The value at position [i][j] is 1 if there is an edge from vertex i to vertex j; otherwise, it is 0.

	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	1
D	0	0	0	0

2. Adjacency list Representation

An **adjacency list** is an array of lists. Each index represents a vertex, and the list at each index contains all vertices that are adjacent to the vertex represented by that index.

- A -> B, C
- B -> D
- C -> D
- D -> (no outgoing edges)

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)
- Outdegree(v)

Outdegree

- All of the arcs going “out” from *v*
- Simple to compute

  Scan through list Adj[v] and count the arcs
- What is the total outdegree? (m=#edges)

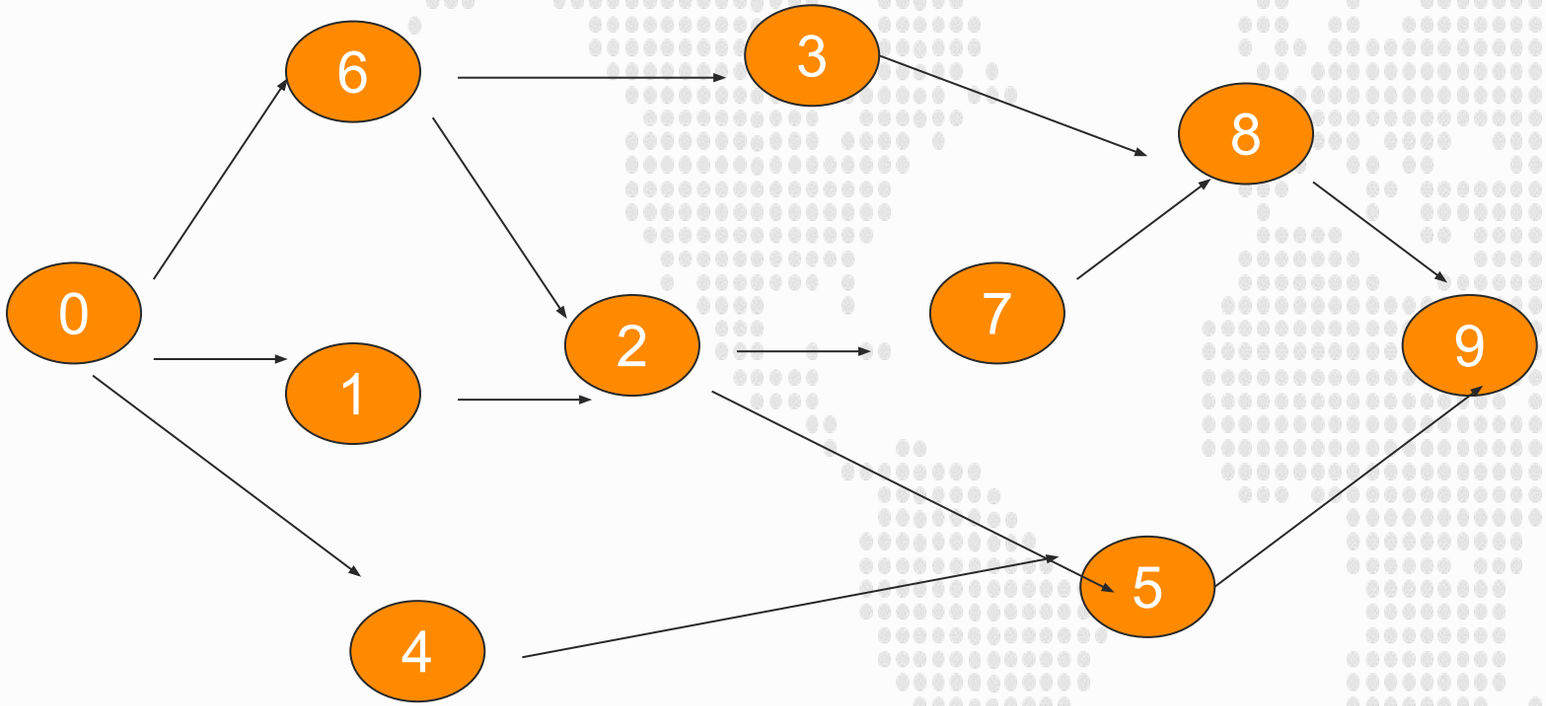
$$\sum_{\text{vertex } v} \text{outdegree}(v) = m$$

Indegree

- All of the arcs *coming "in"* to *v*
  - First, initialize indegree[v]=0 for each vertex v
  - Start by setting indegree[v] = 0 for each vertex v in the graph. This means that initially, we assume there are no edges coming into any vertex.
  - Scan through adj[v] list for each v
  - For each vertex v, look at its **adjacency list** (denoted adj[v]). The adjacency list for v contains all the vertices w that v points to (i.e., all vertices that are directly reachable from v).
  - For every vertex w in the adjacency list of v (i.e., for each vertex w that v points to), increase indegree[w] by 1. This indicates that there is an edge coming into w from v, so we increase the indegree of w accordingly.
  - After scanning through all vertices and their adjacency lists, indegree[v] will contain the count of incoming edges for each vertex v.
- What is the total indegree?

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

Example



Indeg(2)?

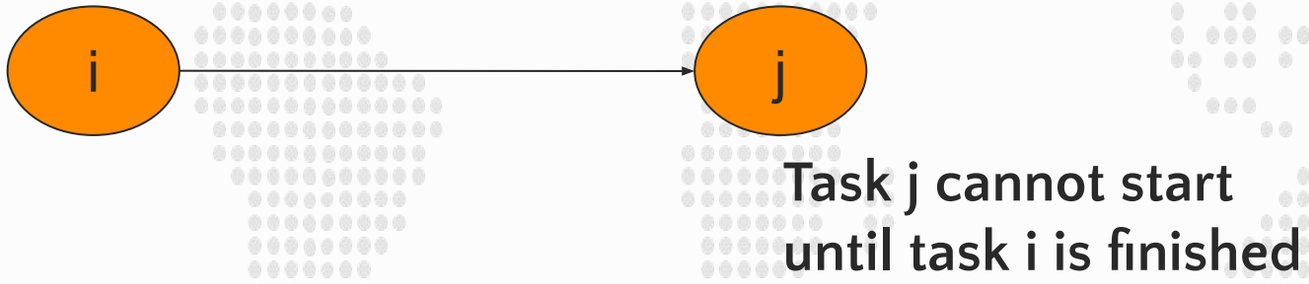
Indeg(8)?

Outdeg(0)?

Num of Edges?

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



Topological Sorting

Definition: Topological Sorting is a linear ordering of vertices in a **directed acyclic graph (DAG)** such that for every directed edge u→v, vertex u comes before vertex v in the ordering.

It is only applicable to DAGs and is widely used in **scheduling tasks, dependency resolution, and compilation processes.**

Properties:

Applicable only to DAGs: A graph must be directed and acyclic (no cycles).

Order of Vertices: It respects the direction of edges—if there is an edge u→v, u appears before v in the order.

Multiple Valid Orders: A DAG can have more than one valid topological ordering.

Steps to Perform Topological Sorting

There are two main methods to find the topological sort of a DAG:

1. **Kahn's Algorithm** (Iterative, using in-degrees)
2. **DFS-based Method** (Recursive, using stack)

Kahn's Algorithm

Key Idea: This algorithm uses the concept of **in-degrees** of vertices (number of incoming edges).

Algorithm:

1. **Calculate in-degree:** For each vertex, compute the number of incoming edges.
2. **Initialize a queue:** Add all vertices with in-degree 0 to a queue.
3. **Process vertices:**
    - Remove a vertex from the queue and add it to the topological order.
    - Decrease the in-degree of all its neighbors by 1.
    - If a neighbor's in-degree becomes 0, add it to the queue.
4. **Repeat until the queue is empty.**
5. **Check for cycles:** If all vertices are processed, the graph is a DAG; otherwise, it contains a cycle.

Idea behind Kahn's Algorithm

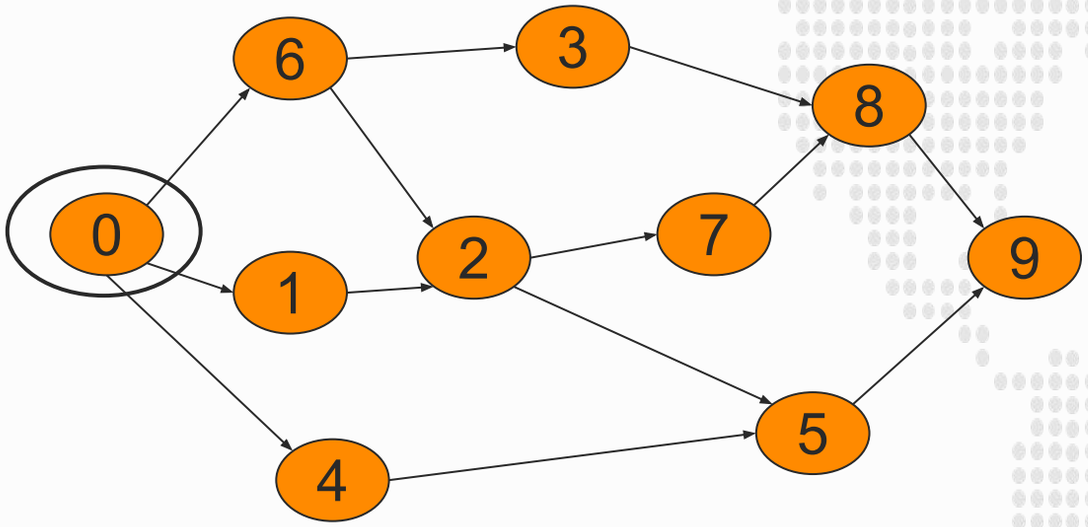
- Starting point must have zero indegree!
- If it doesn't exist, the graph would not be acyclic

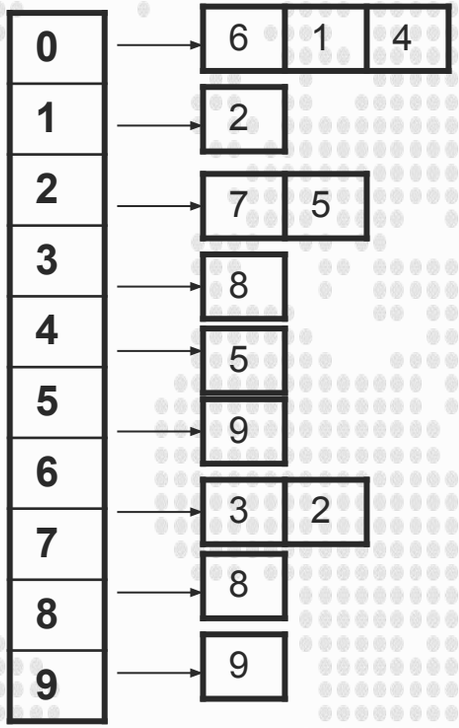
1. A vertex with zero *indegree* is a task that can start right away. So we can output it first in the linear order

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 step 1–2 until no vertex is left.

Example





Indegree

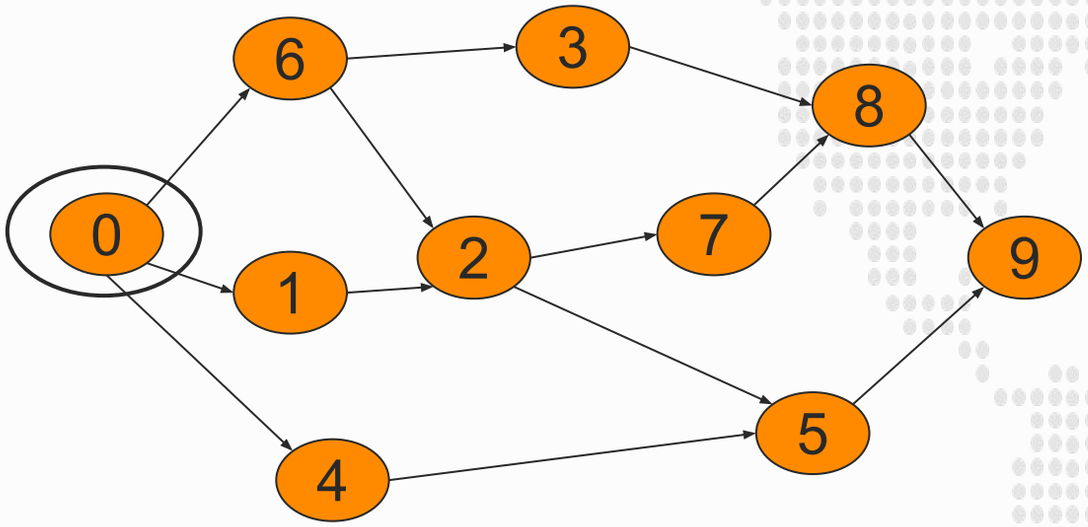
0	0
1	1
2	2
3	1
4	1
5	2
6	1
7	1
8	2
9	2

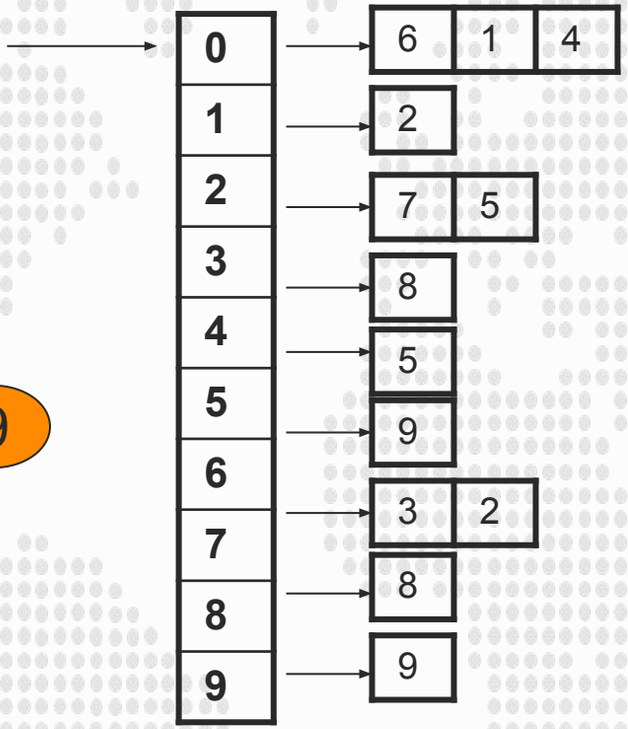
start

Q = { 0 }

OUTPUT: 0

Example





Indegree

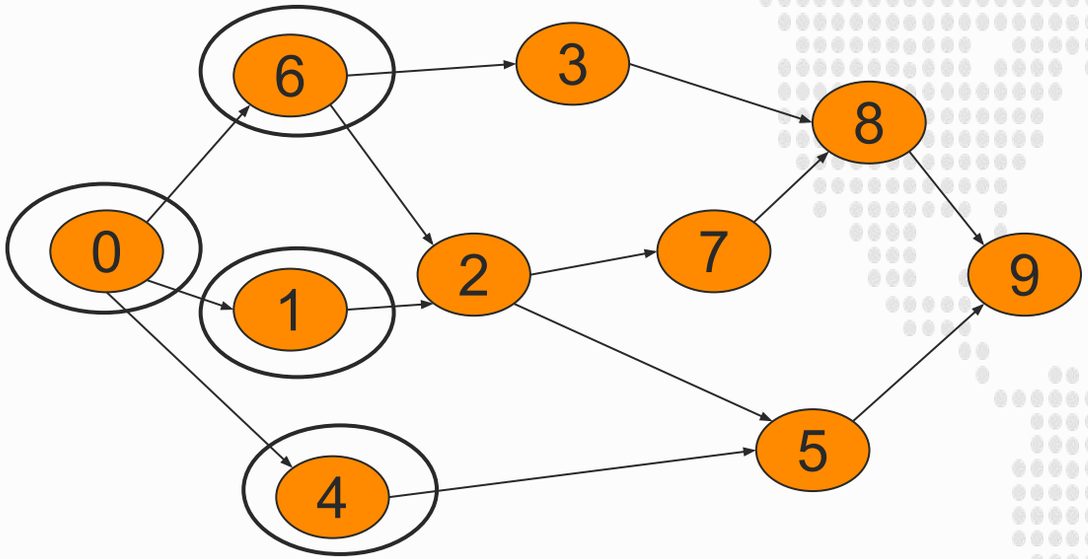
0	0	
1	1	-1
2	2	
3	1	
4	1	-1
5	2	
6	1	-1
7	1	
8	2	
9	2	

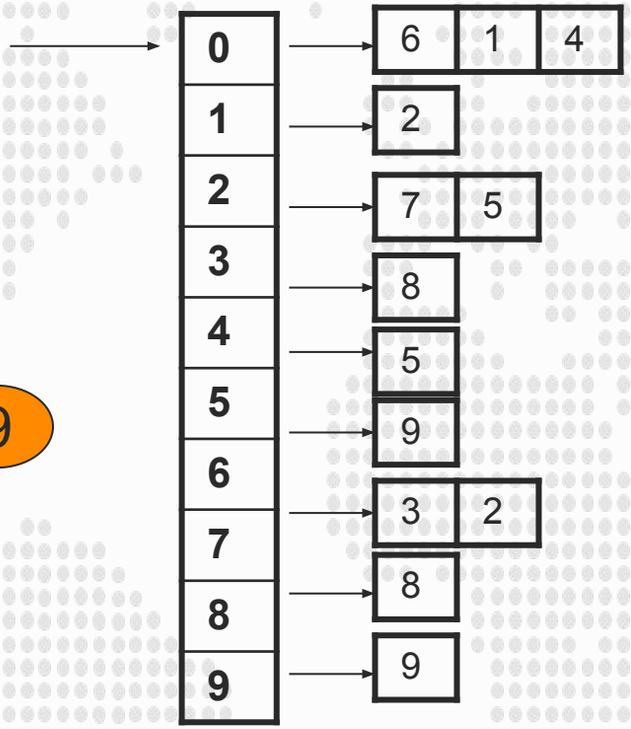
Dequeue 0 Q = { }
-> remove 0's arcs – adjust indegrees of neighbors

Decrement 0's neighbors

OUTPUT:

Example





Indegree

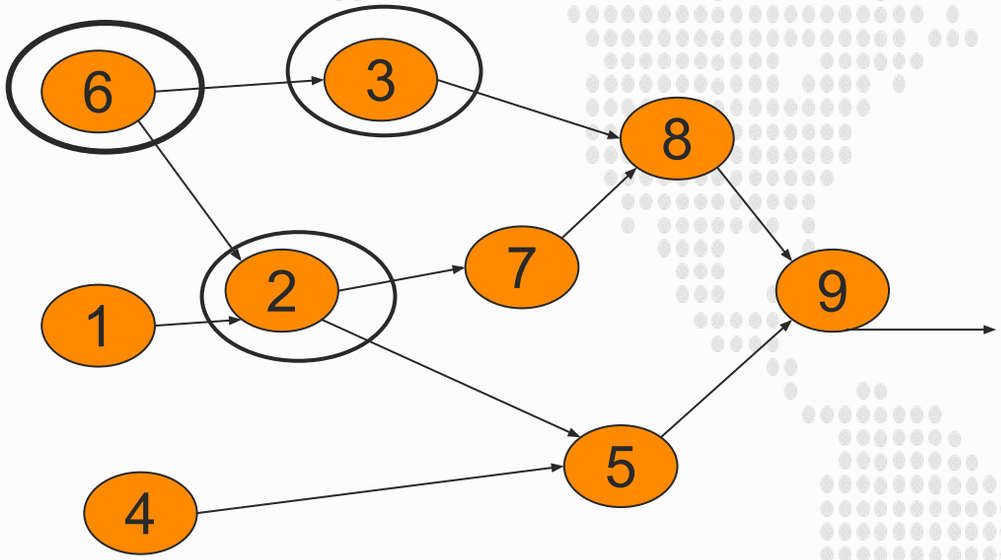
0	0
1	0
2	2
3	1
4	0
5	2
6	0
7	1
8	2
9	2

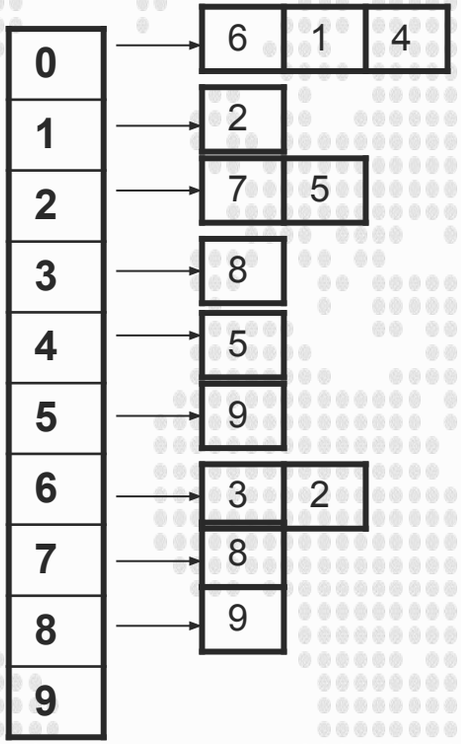
Dequeue 0 Q = { 6, 1, 4 }
Enqueue all starting points

Enqueue all
new start points

OUTPUT: 0

Example





Indegree

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

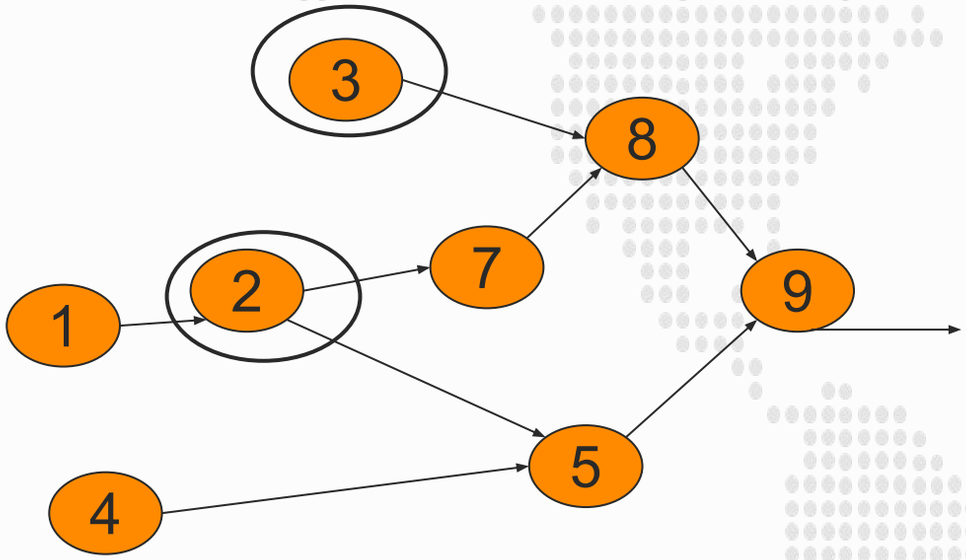
Dequeue 6 Q = { 1, 4 }

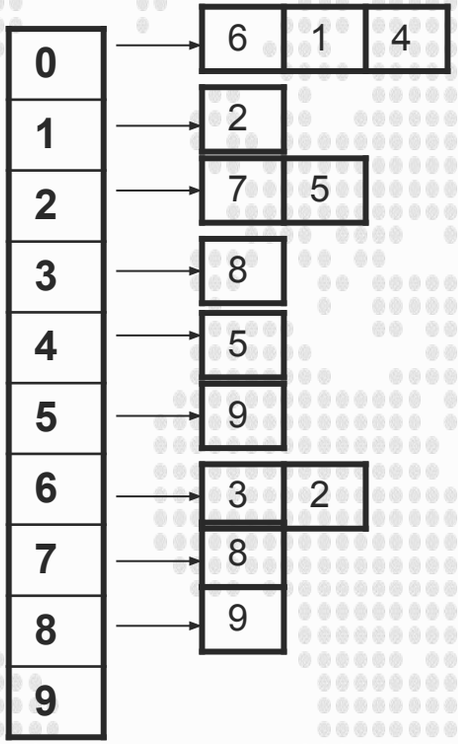
Remove arcs .. Adjust indegrees of neighbors

Adjust neighbors indegree

OUTPUT: 0 6

Example





Indegree

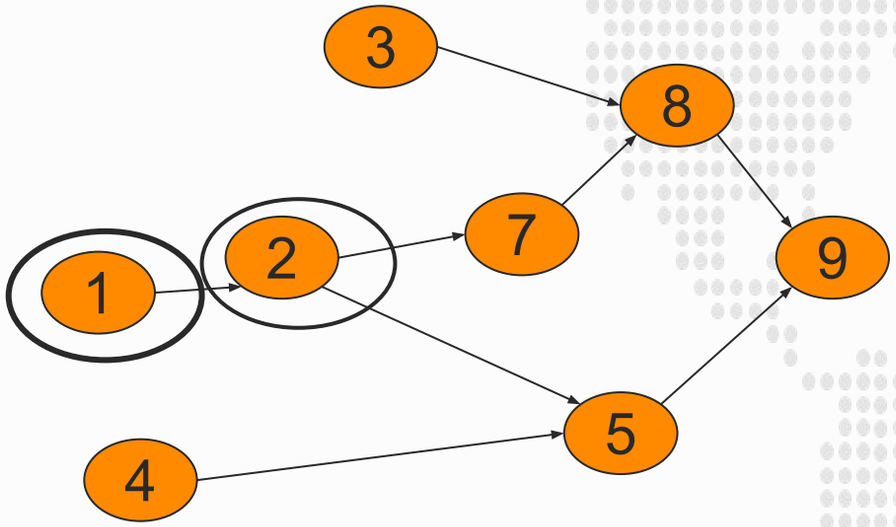
0	0
1	0
2	1
3	0
4	0
5	2
6	0
7	1
8	2
9	2

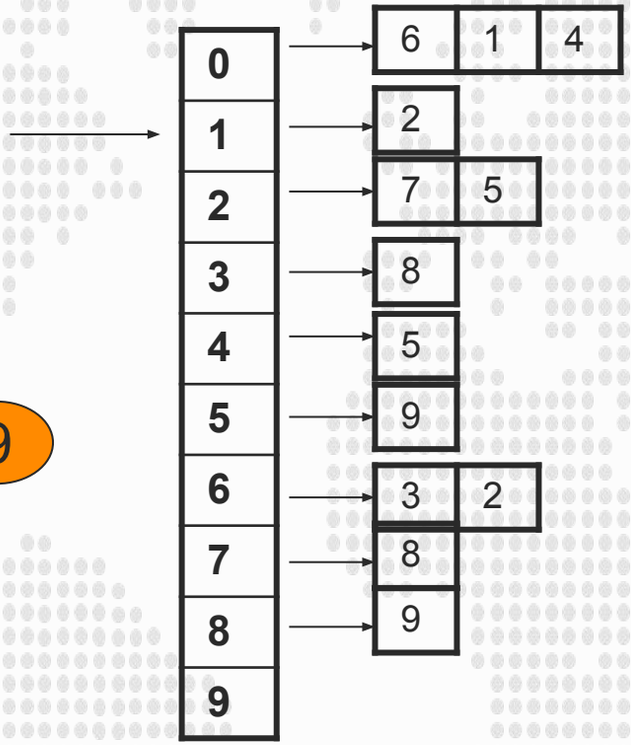
Dequeue 6 Q = { 1, 4, 3 }
Enqueue 3

Enqueue new start

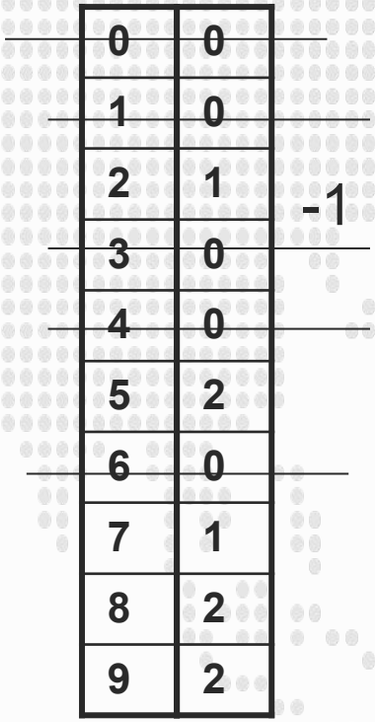
OUTPUT: 0 6

Example





Indegree

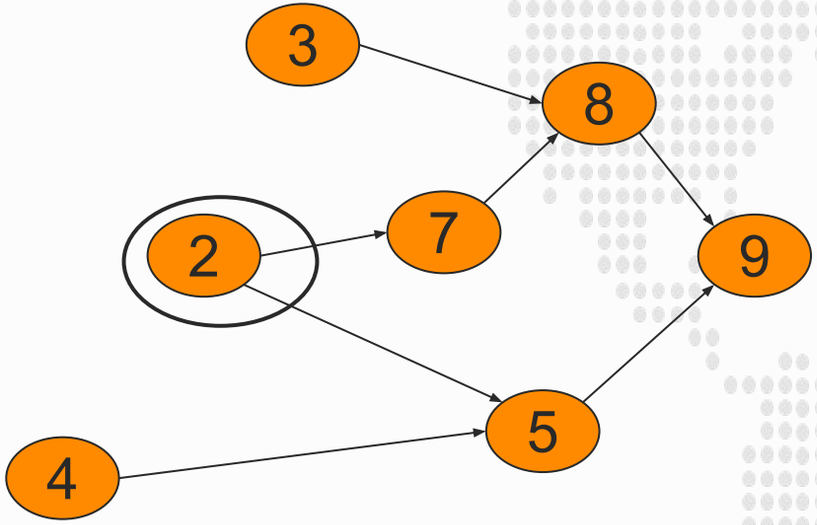


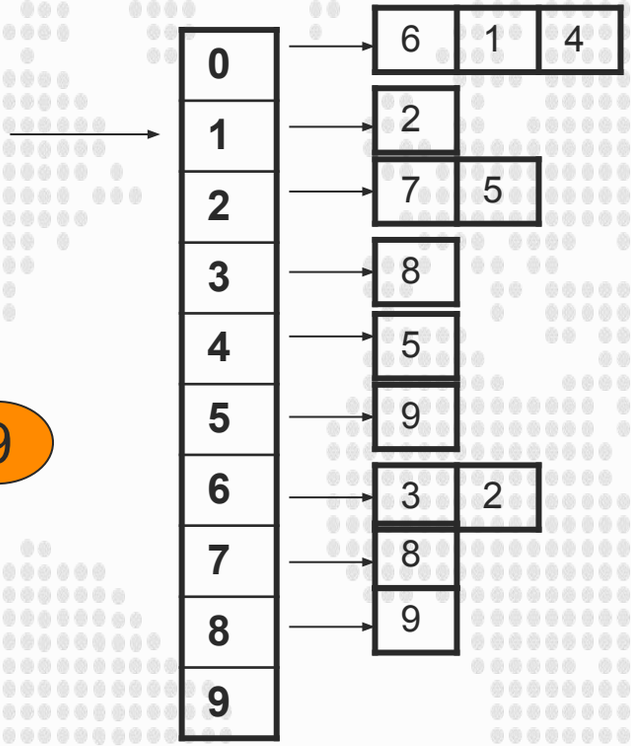
Dequeue 1 Q = { 4, 3 }
Adjust indegrees of neighbors

Adjust neighbors of 1

OUTPUT: 0 6 1

Example





Indegree

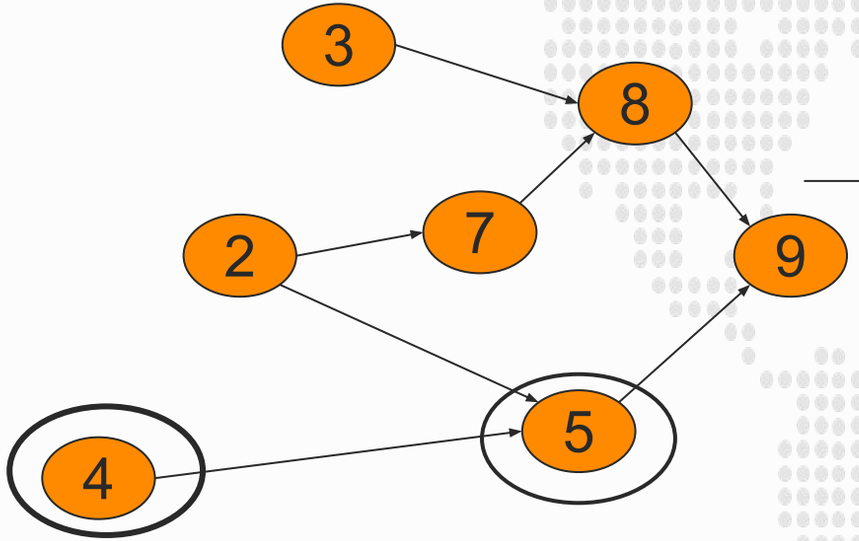
0	0
1	0
2	0
3	0
4	0
5	2
6	0
7	1
8	2
9	2

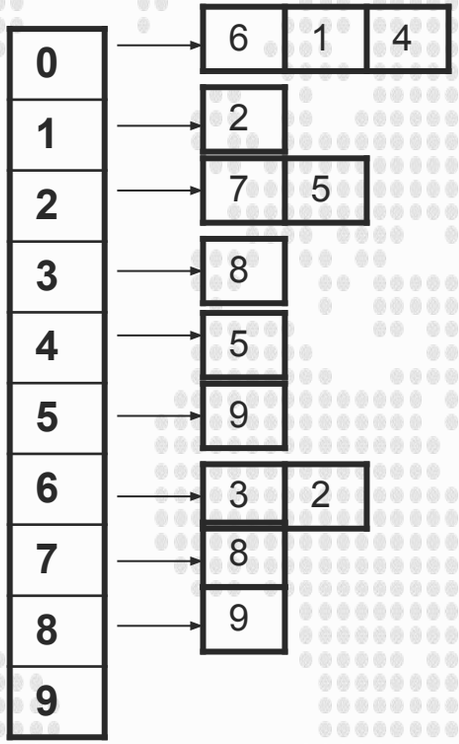
Dequeue 1 Q = { 4, 3, 2 }
Enqueue 2

Enqueue new starting points

OUTPUT: 0 6 1

Example





Indegree

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

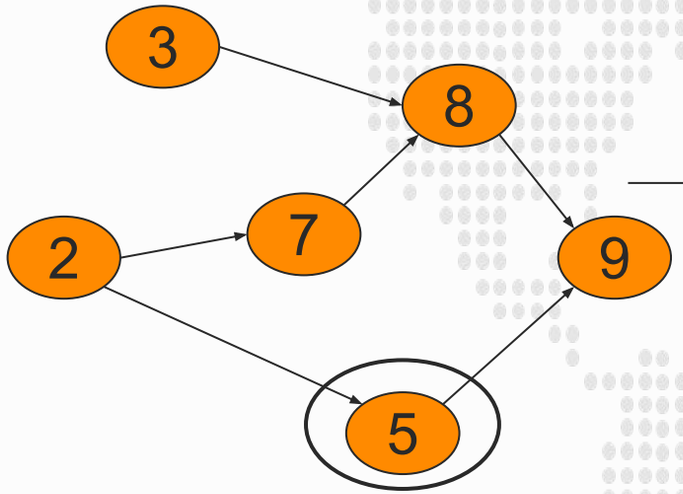
-1

Dequeue 4 Q = { 3, 2 }
Adjust indegrees of neighbors

Adjust 4's neighbors

OUTPUT: 0 6 1 4

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

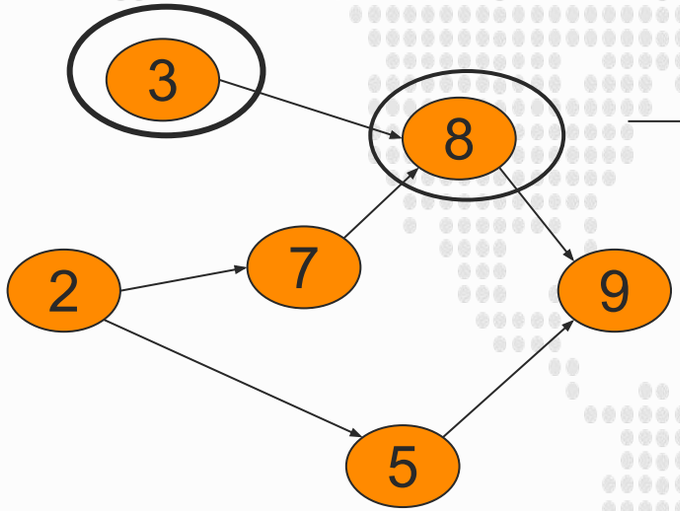
0	0
1	0
2	0
3	0
4	0
5	1
6	0
7	1
8	2
9	2

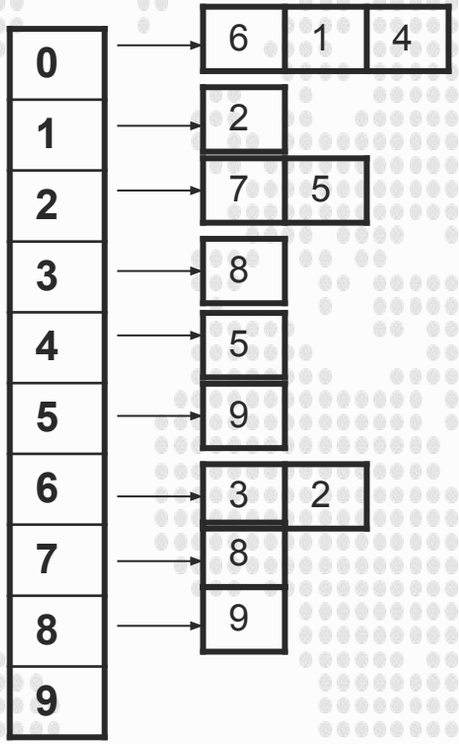
Dequeue 4 Q = { 3, 2 }
No new start points found

NO new start points

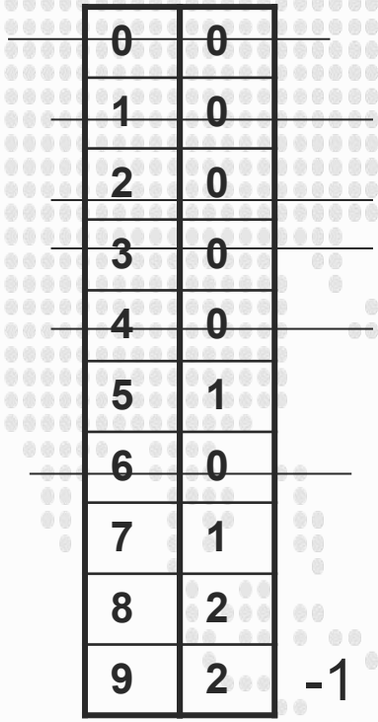
OUTPUT: 0 6 1 4

Example





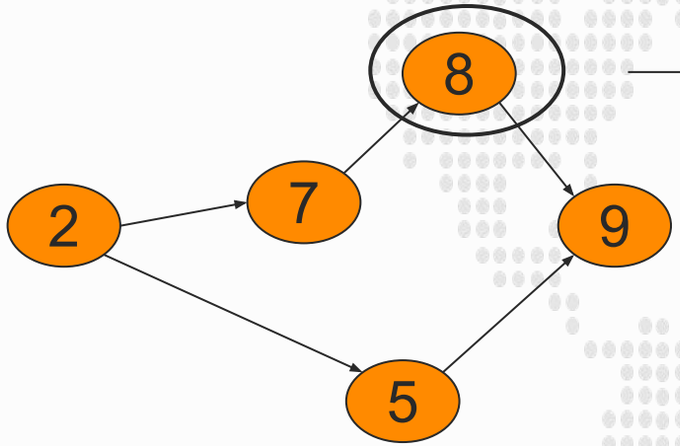
Indegree

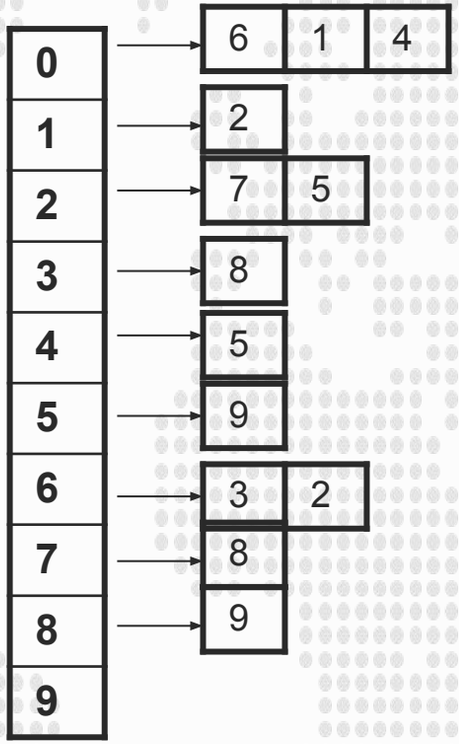


Dequeue 3 Q = { 2 }
Adjust 3's neighbors

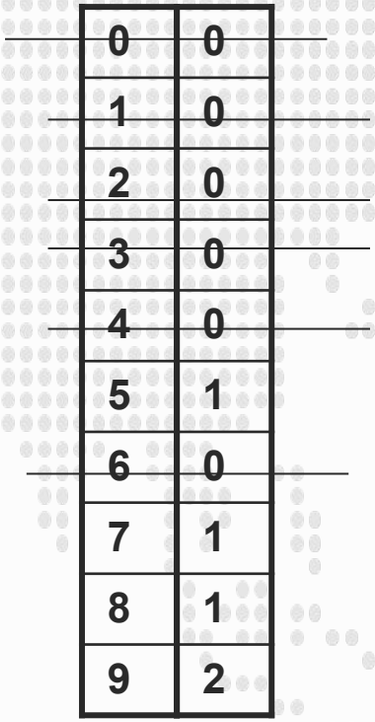
OUTPUT: 0 6 1 4 3

Example





Indegree

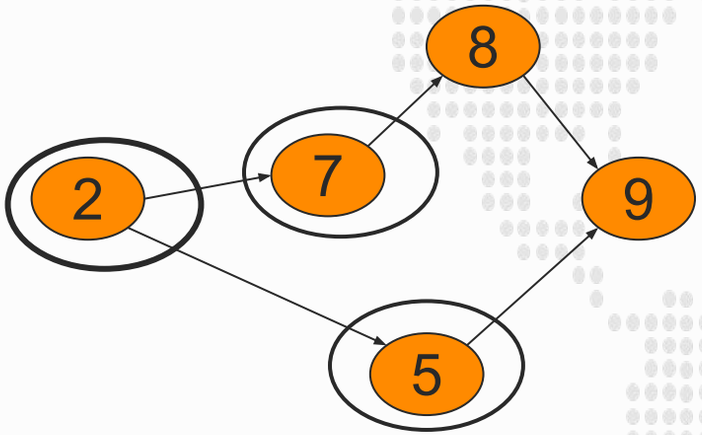


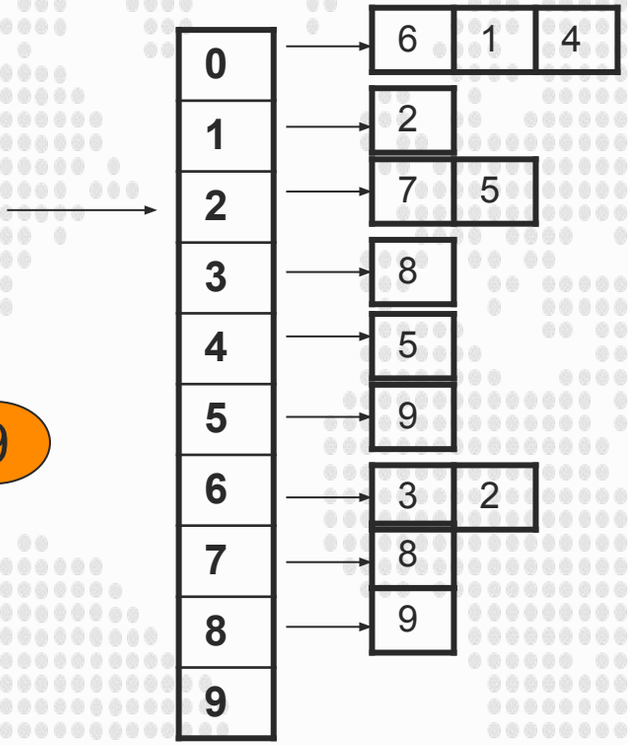
Dequeue 3 Q = { 2 }

No new start points found

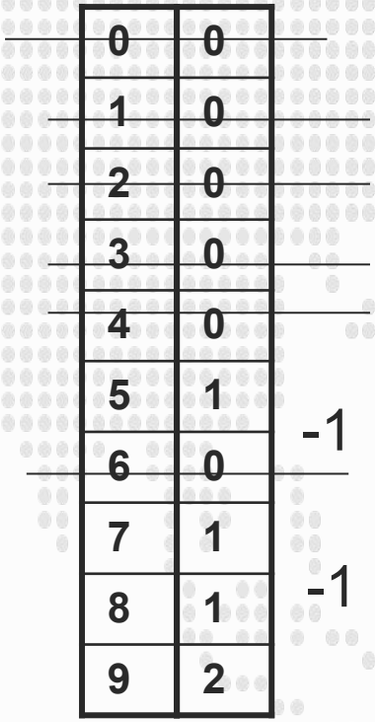
OUTPUT: 0 6 1 4 3

Example





Indegree

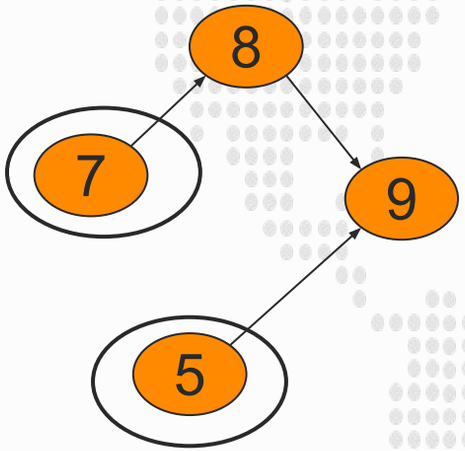


Dequeue 2 Q = { }

Adjust 2's neighbors

OUTPUT: 0 6 1 4 3 2

Example



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

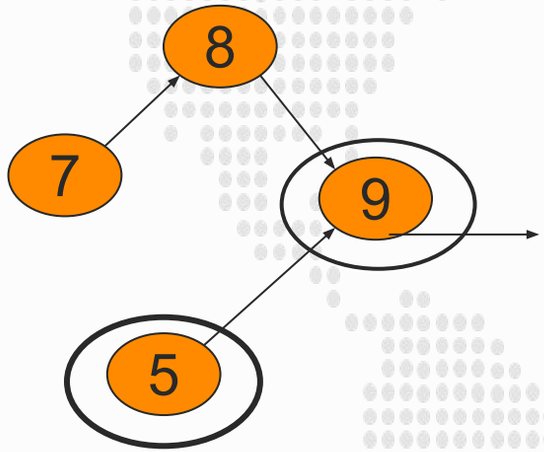
Indegree

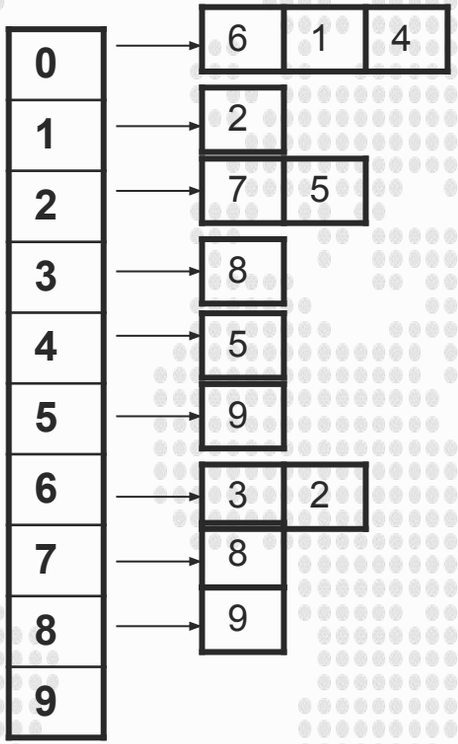
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	2

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

OUTPUT: 0 6 1 4 3 2

Example





Indegree

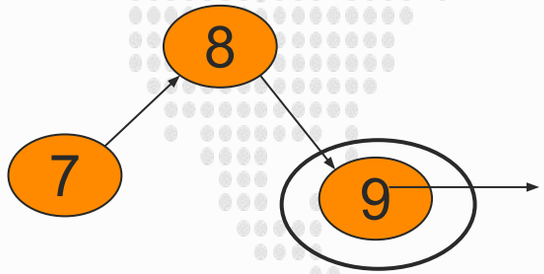
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	2

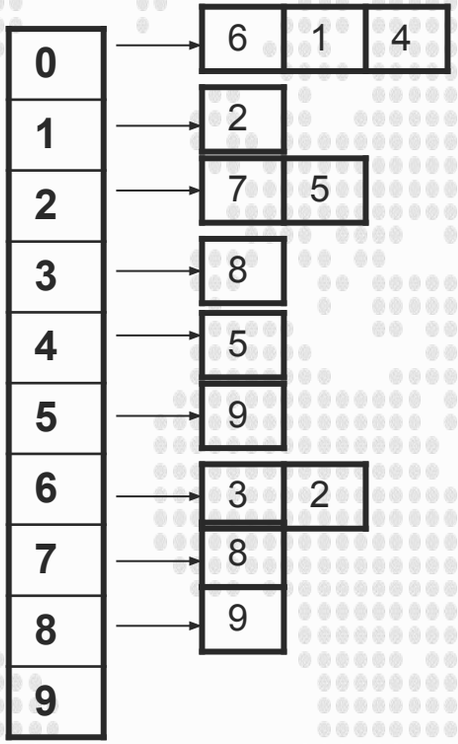
-1

Dequeue 5 Q = { 7 }
Adjust neighbors

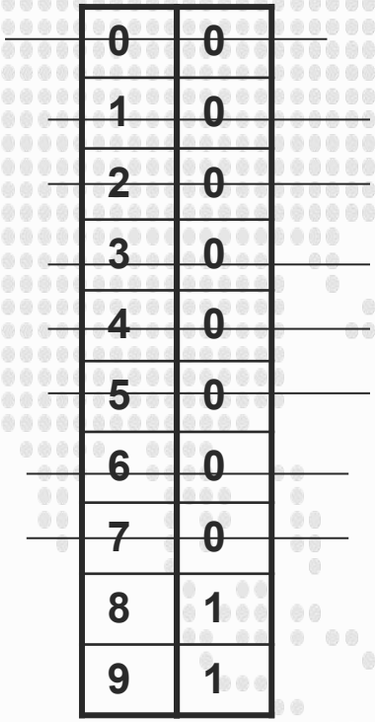
OUTPUT: 0 6 1 4 3 2 5

Example





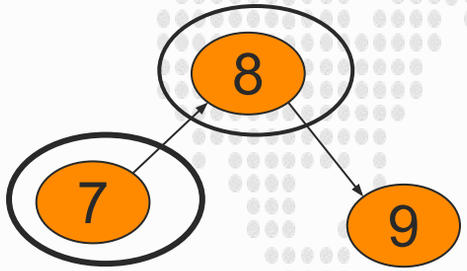
Indegree

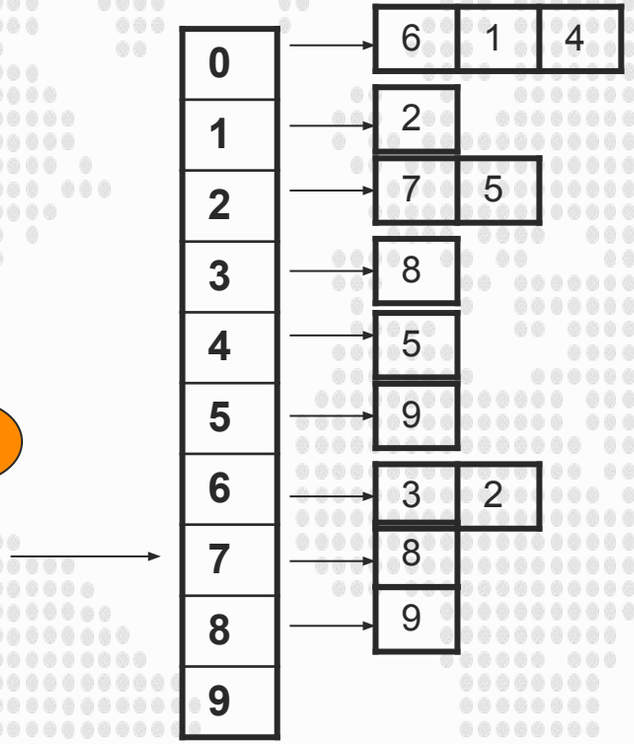


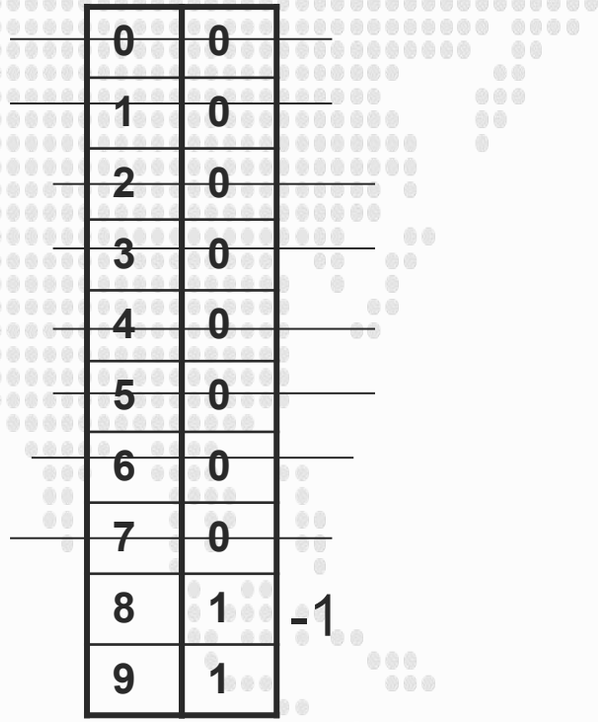
Dequeue 5 Q = { 7 }
No new starts

OUTPUT: 0 6 1 4 3 2 5

Example



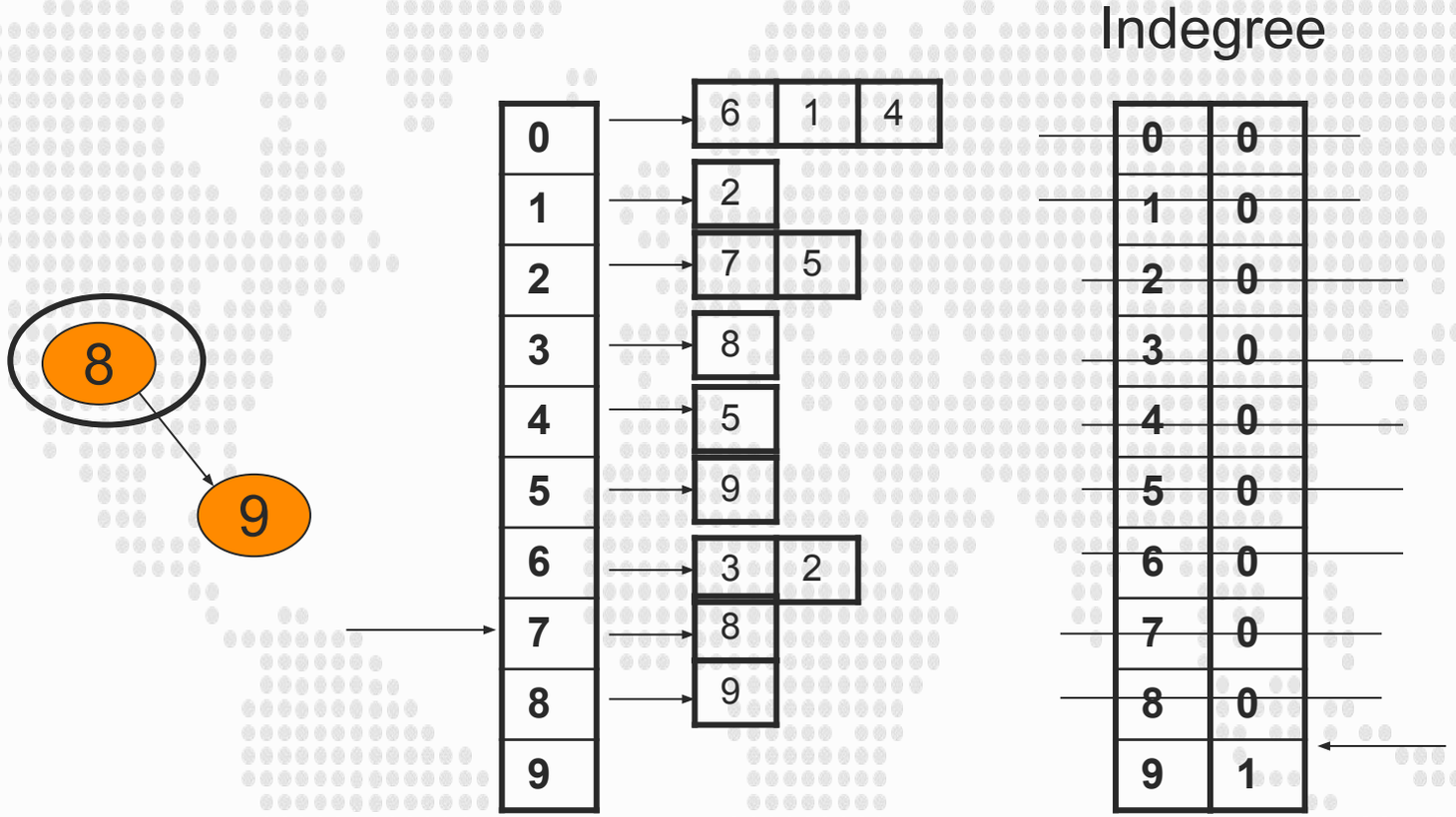




Dequeue 7 Q = { }
Adjust neighbors

OUTPUT: 0 6 1 4 3 2 5 7

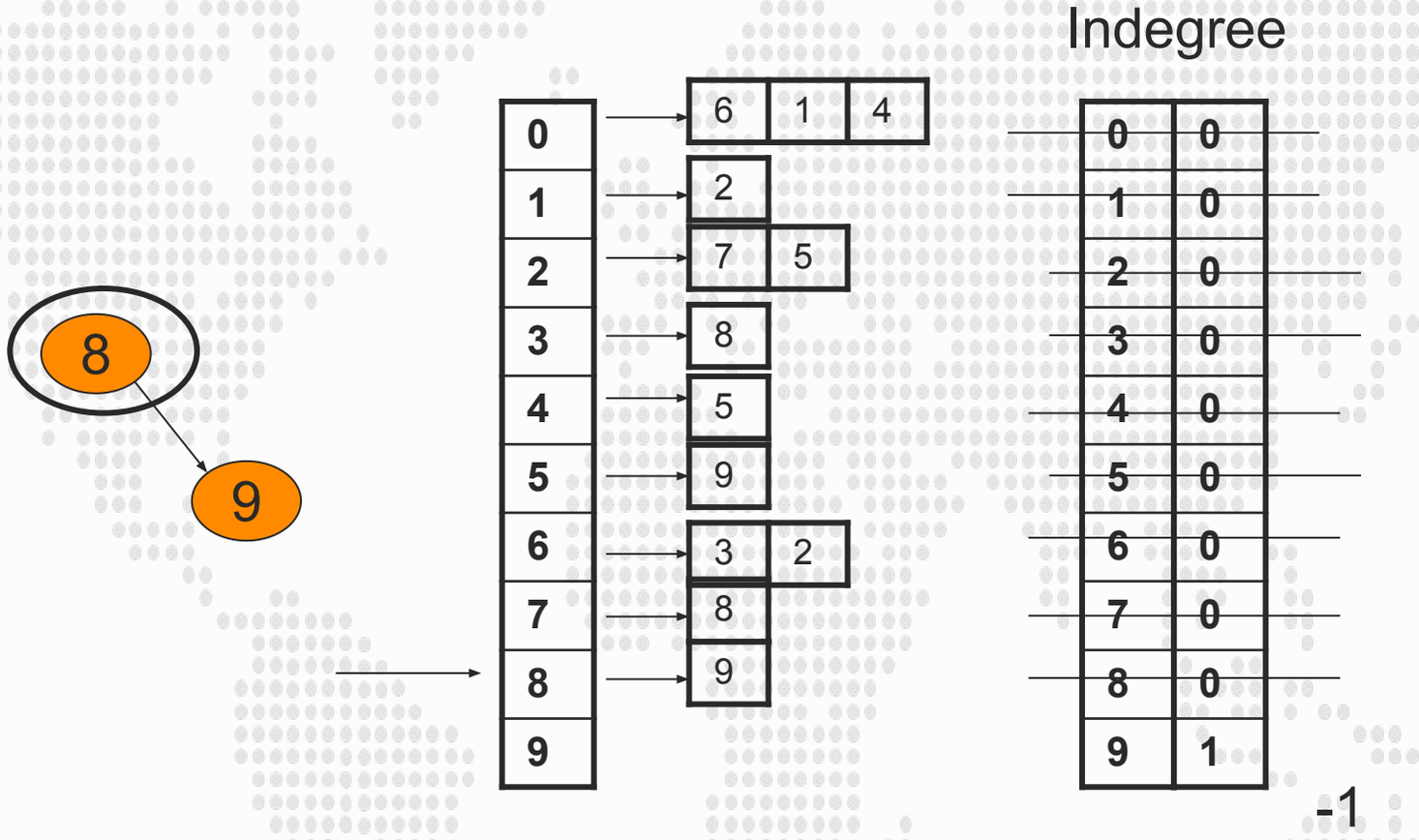
Example



Dequeue 7 Q = { 8 }
Enqueue 8

OUTPUT: 0 6 1 4 3 2 5 7

Example

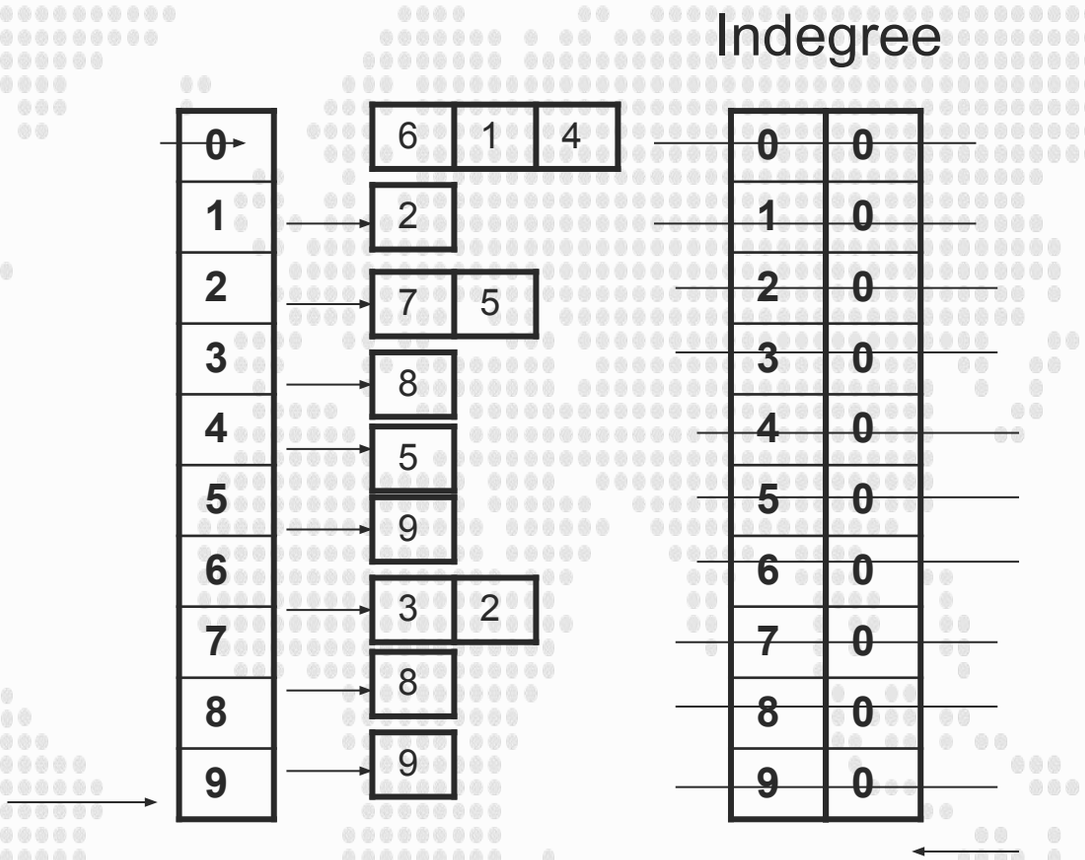


Dequeue 8 Q = { }

Adjust indegrees of neighbors

OUTPUT: 0 6 1 4 3 2 5 7 8

Example

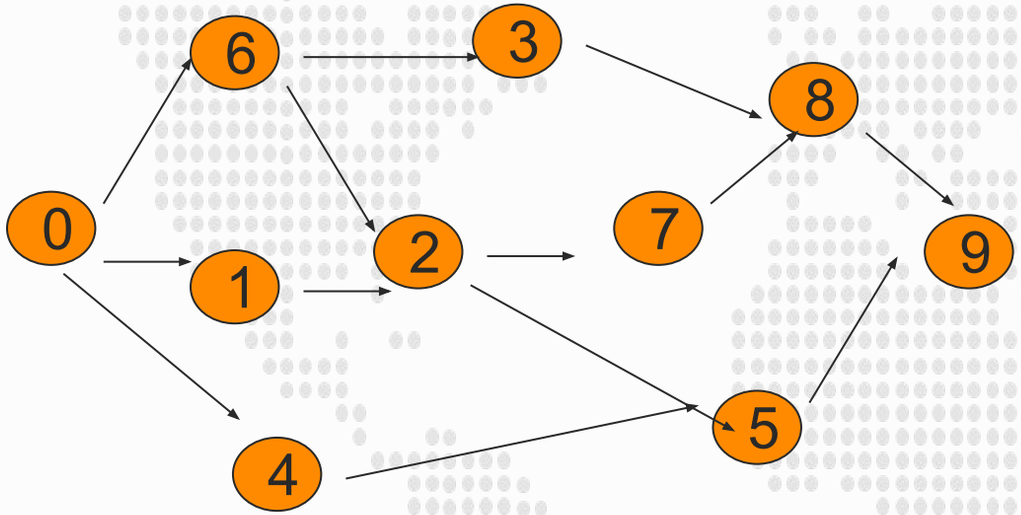


9

Dequeue 8 Q = { 9 }
    Enqueue 9
Dequeue 9 Q = { }
    STOP – no neighbors

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

Example



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

Khan's Algorithm Steps

1. **In-degree Calculation**:
   - Count how many edges are directed towards each vertex. This is stored in the inDegree array.
2. **Initialize Queue**:
   - Add all vertices with in-degree 0 to a queue, as these can be the starting points for the topological order.

Khan’s Algorithm Steps

3. BFS and Update In-degree:

- Process each vertex from the queue:
  - Add it to the topological order.
  - Reduce the in-degree of its neighbors by 1.
  - If a neighbor's in-degree becomes 0, enqueue it.

4. Cycle Detection:

- If the number of vertices in the topological order is less than the total number of vertices, the graph contains a cycle.

5. Result:

- Return the topological order if no cycles exist; otherwise, indicate a cycle.

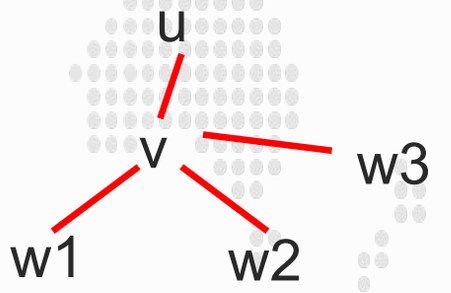
Depth-First Search (DFS)

- DFS is another popular graph search strategy
  - Idea is similar to pre-order traversal (visit children first)

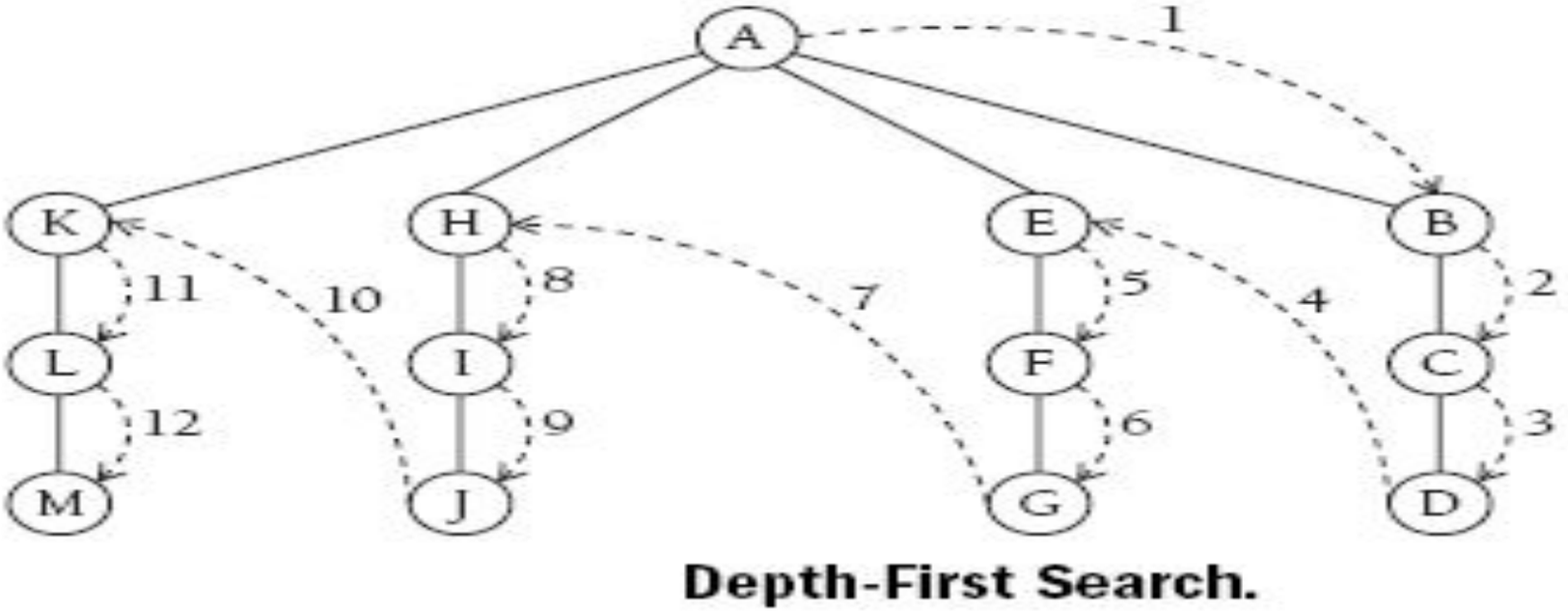
- DFS can provide certain information about the graph that BFS cannot
  - It can tell whether we have encountered a cycle or not

DFS Algorithm

- DFS will continue to visit neighbors in a recursive pattern
  - Whenever we visit v from u, we recursively visit all unvisited neighbors of v. Then we backtrack (return) to u.
  - Note: it is possible that w2 was unvisited when we recursively visit w1, but became visited by the time we return from the recursive call.



DFS

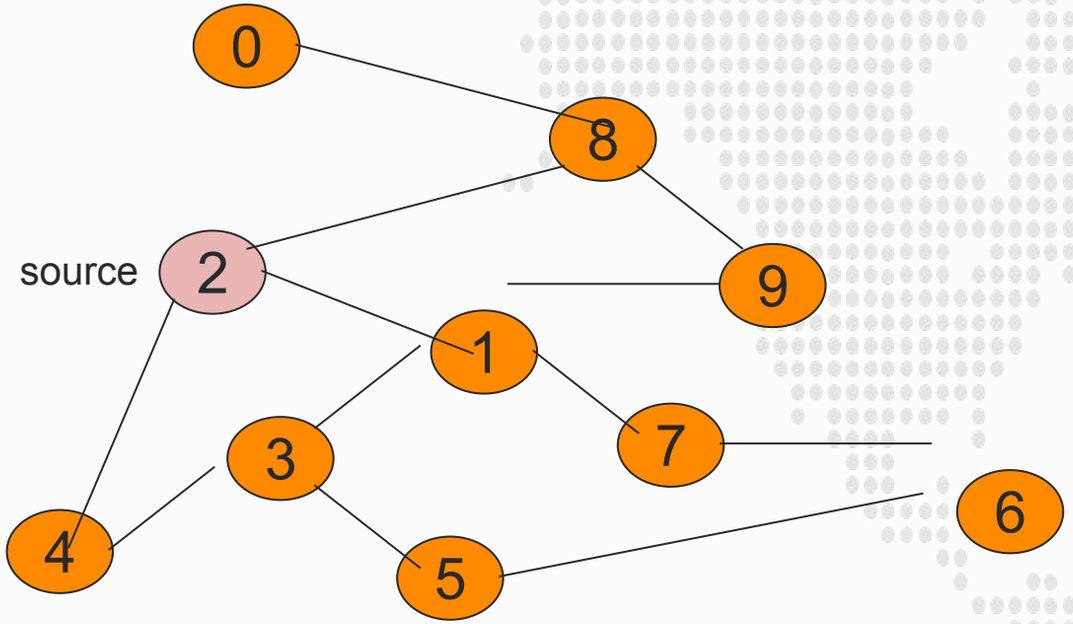


Depth-First Search.

DFS Algorithm

```
DFS(graph, start, visited):
    Mark start as visited
    For each neighbor of start:
        If neighbor is not visited:
            Call DFS(graph, neighbor, visited)
```

Example



Adjacency List

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

Visited Table (T/F)

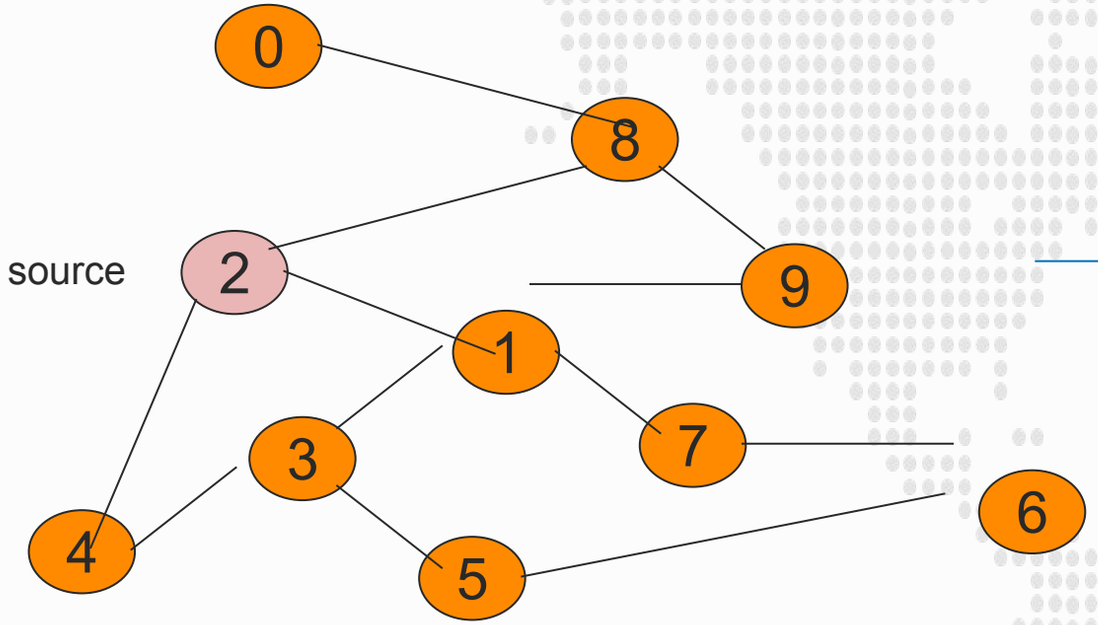
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Pred

Initialize visited table (all False)

Initialize Pred to -1

Example



Adjacency List

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

Visited Table (T/F)

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

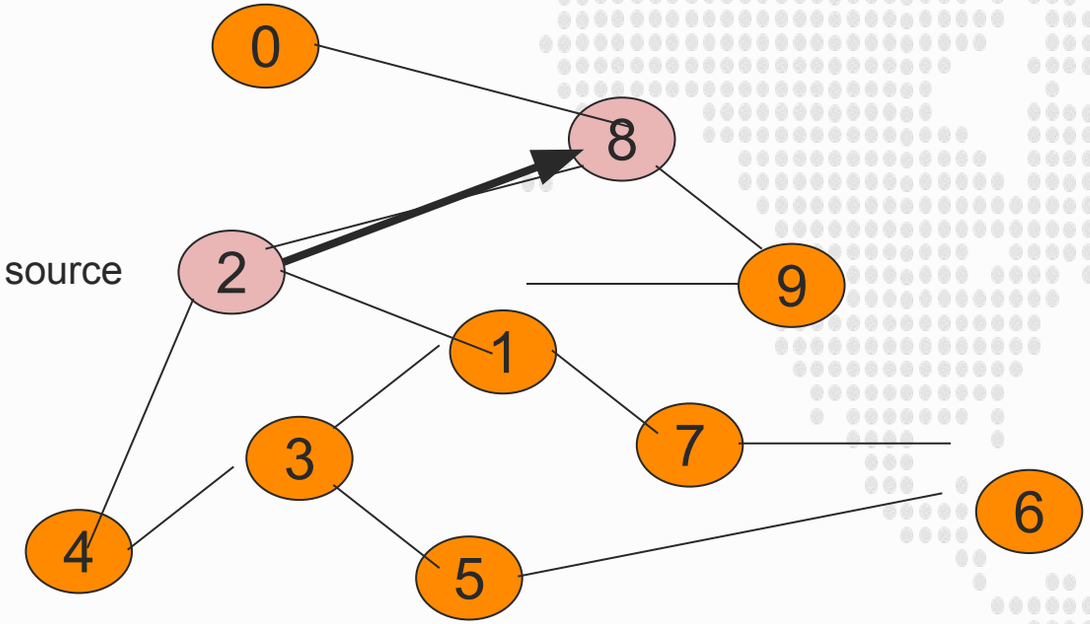
Pred

RDFS(2)

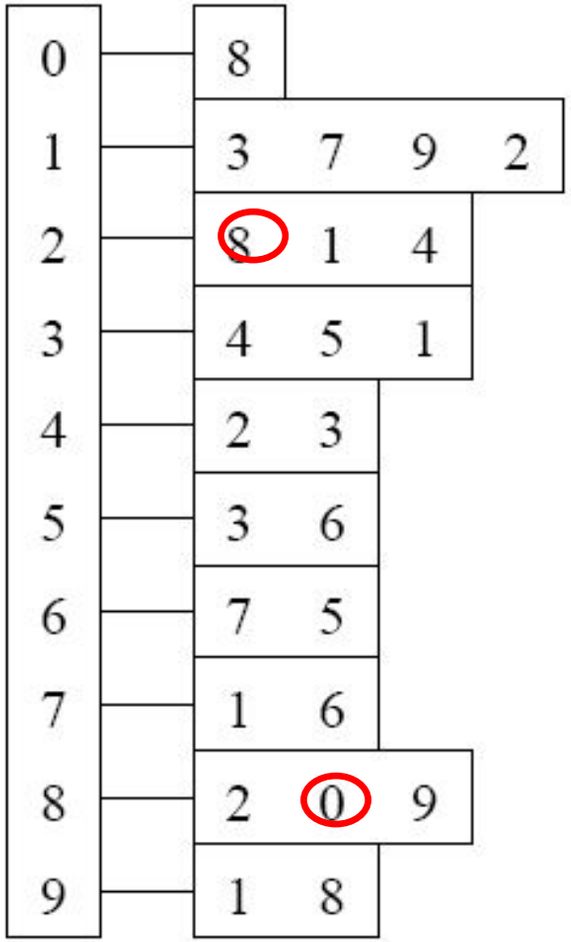
Now visit RDFS(8)

Mark 2 as visited

Example



Adjacency List



Visited Table (T/F)

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

Pred

Recursive calls

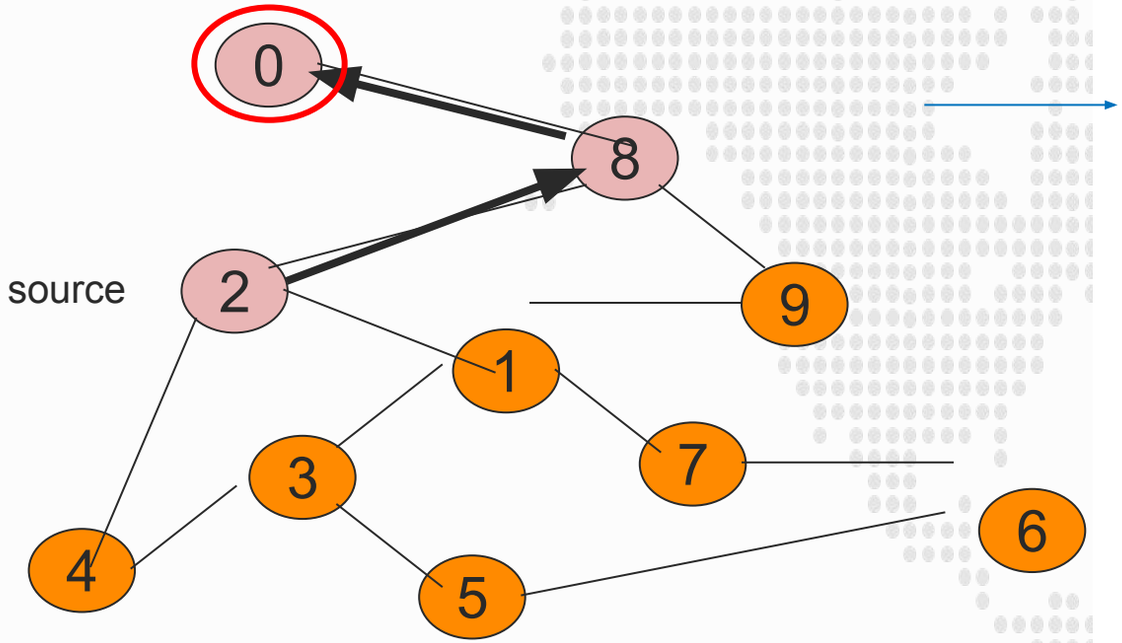
RDFS(2)

Mark 8 as visited

RDFS(8)

mark Pred[8]

Example



Adjacency List

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

Visited Table (T/F)

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

Pred

Recursive calls

RDFS(2)

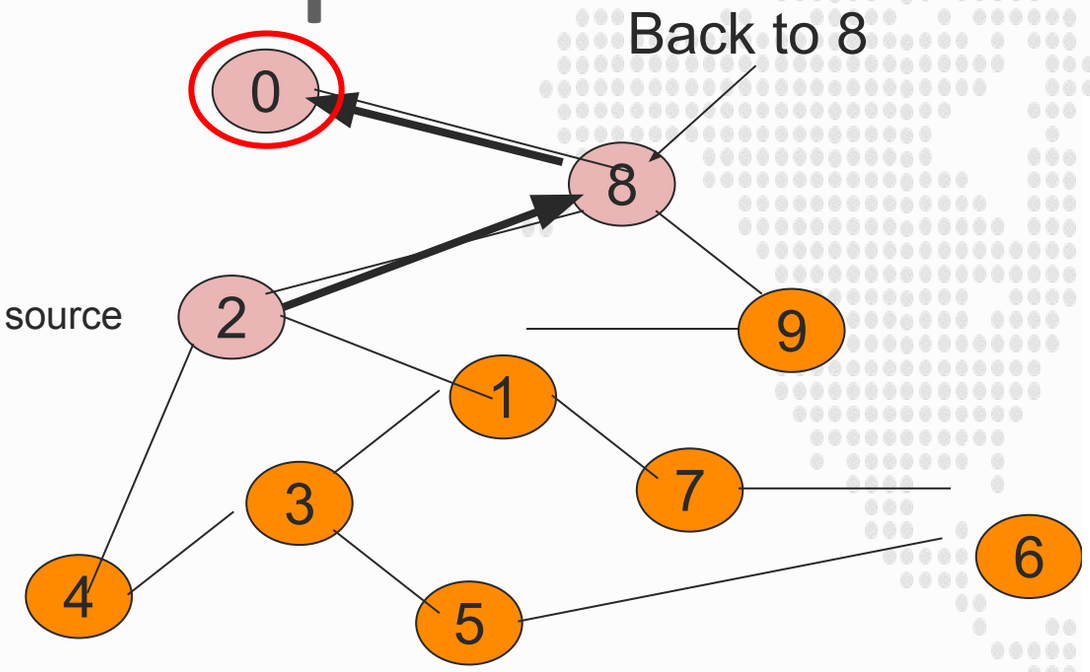
RDFS(8)

RDFS(0) -> no unvisited neighbors, return to call RDFS(8)

Mark 0 as visited

Mark Pred[0]

Example



Adjacency List

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

Visited Table (T/F)

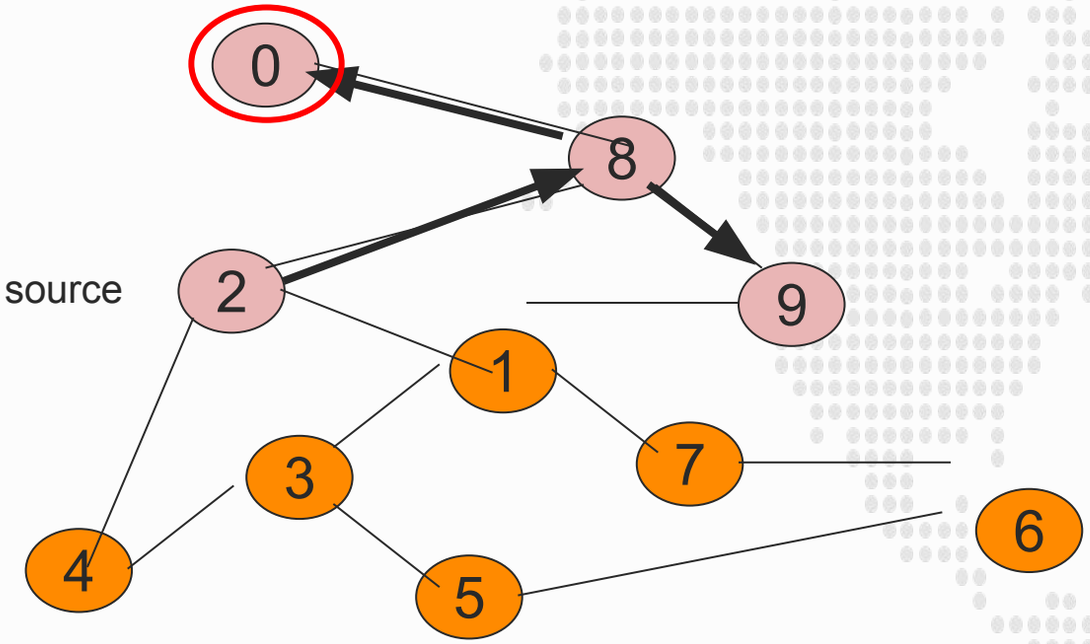
0	T	8
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	T	2
9	F	-

Pred

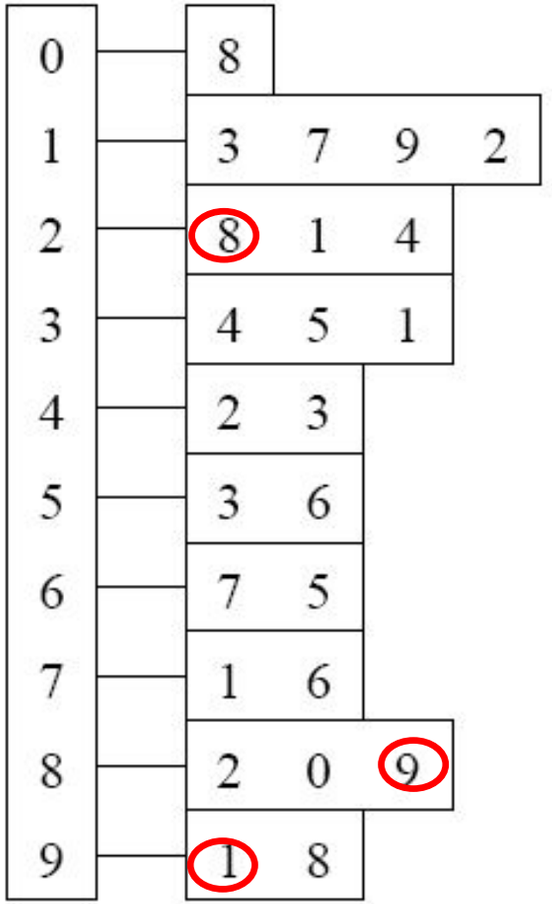
Recursive calls

RDFS(2)
RDFS(8)
Now visit 9 -> RDFS(9)

Example



Adjacency List



Visited Table (T/F)

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

Pred
8
-
-
-
-
-
-
-
2
8

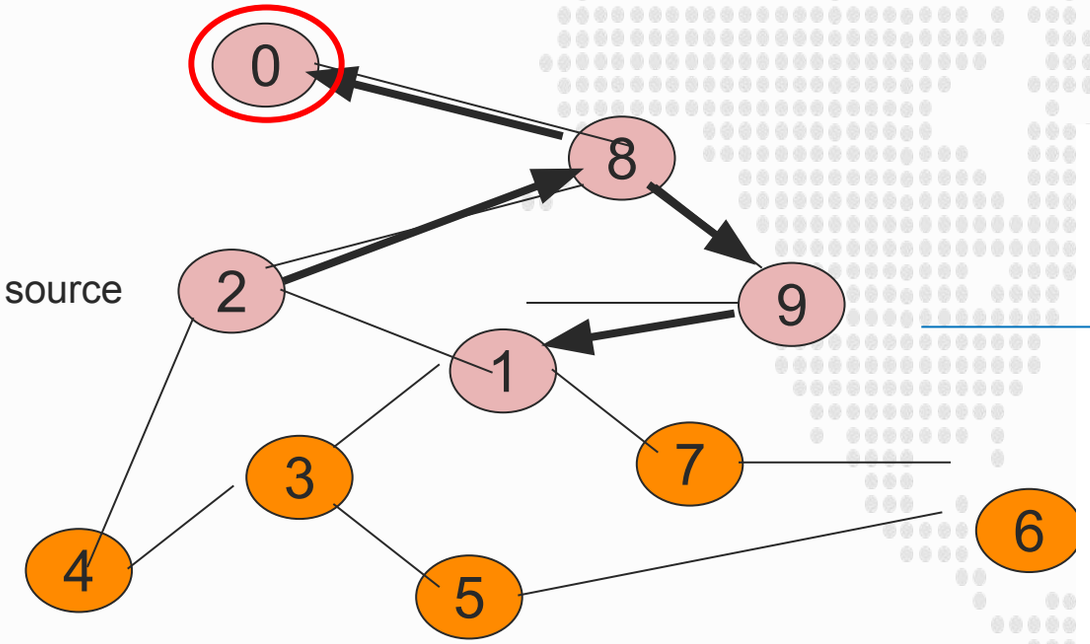
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
-> visit 1, RDFS(1)

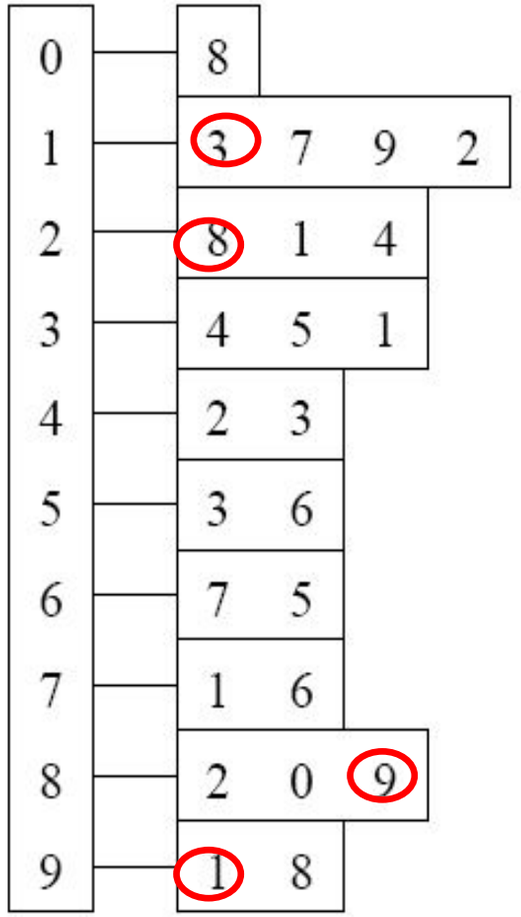
Mark 9 as visited

Mark Pred[9]

Example



Adjacency List



Visited Table (T/F)

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

8
9
-
-
-
-
-
-
2
8

Pred

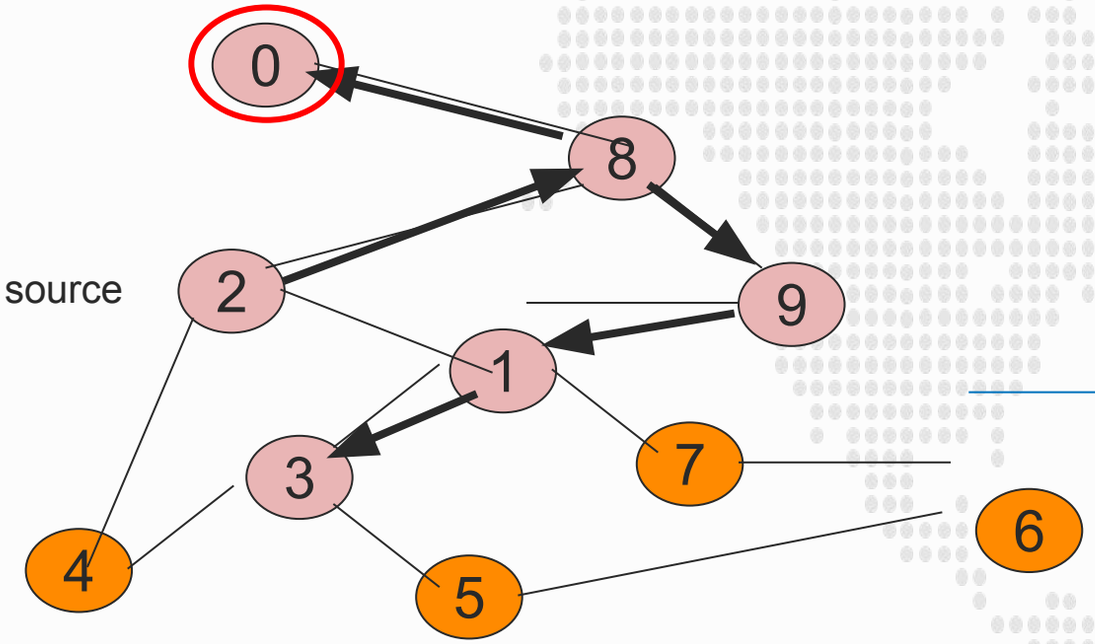
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
visit RDFS(3)

Mark 1 as visited

Mark Pred[1]

Example



Adjacency List

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

Visited Table (T/F)

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

Pred
8
9
-
1
-
-
-
-
2
8

Recursive calls

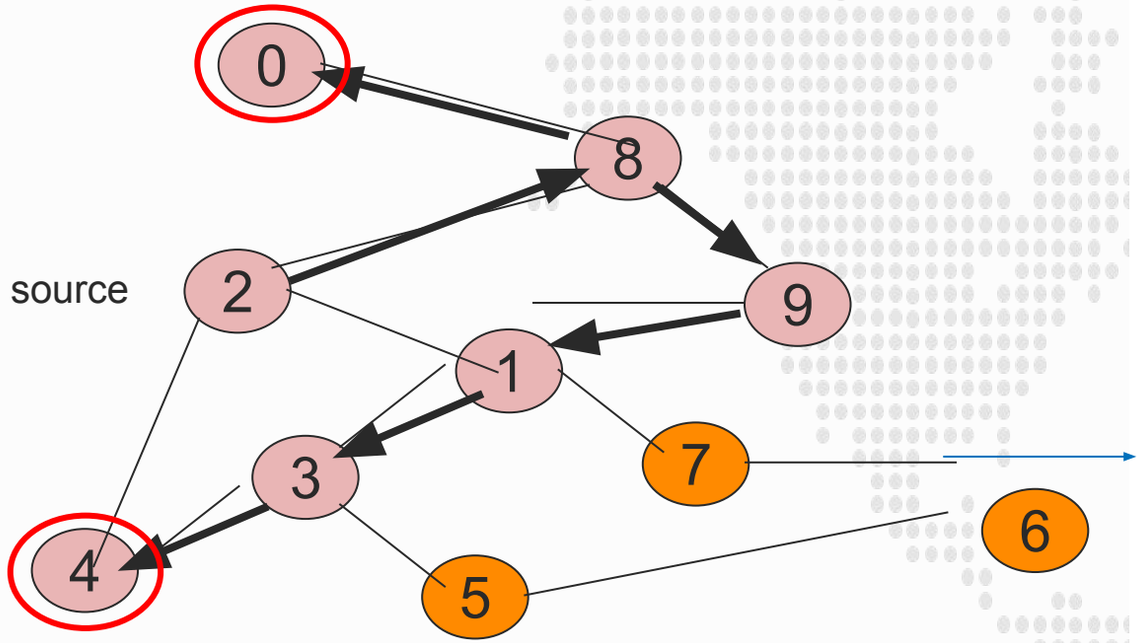
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

visit RDFS(4)

Mark 3 as visited

Mark Pred[3]

Example



Adjacency List

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

Visited Table (T/F)

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

Pred
8
9
-
1
3
-
-
-
2
8

Recursive calls

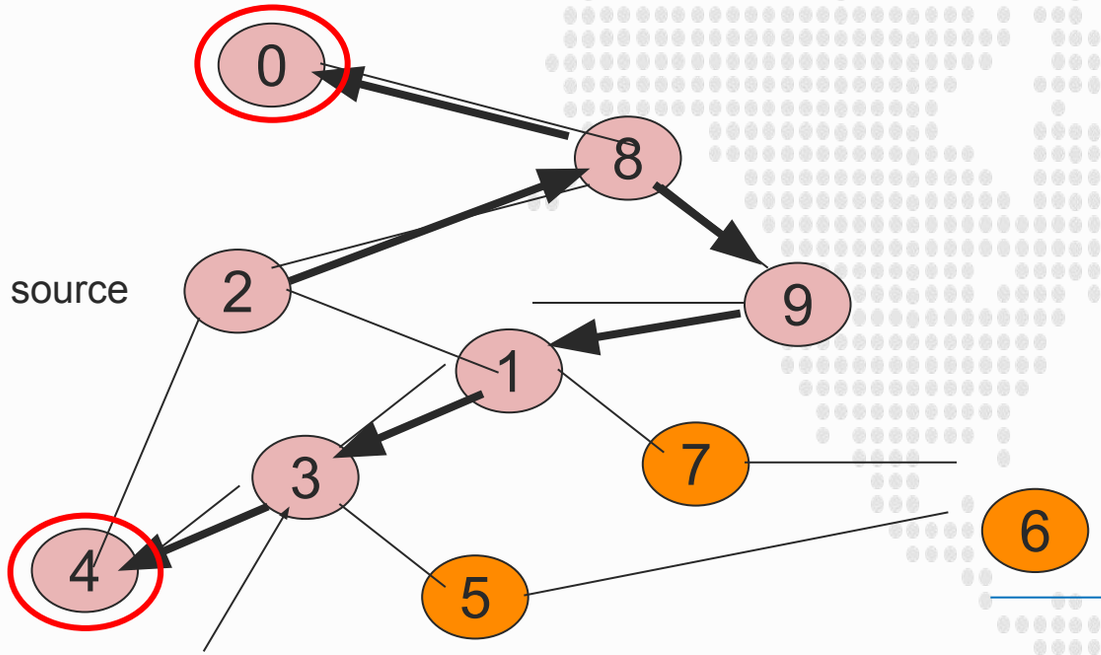
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

Mark 4 as visited

Mark Pred[4]

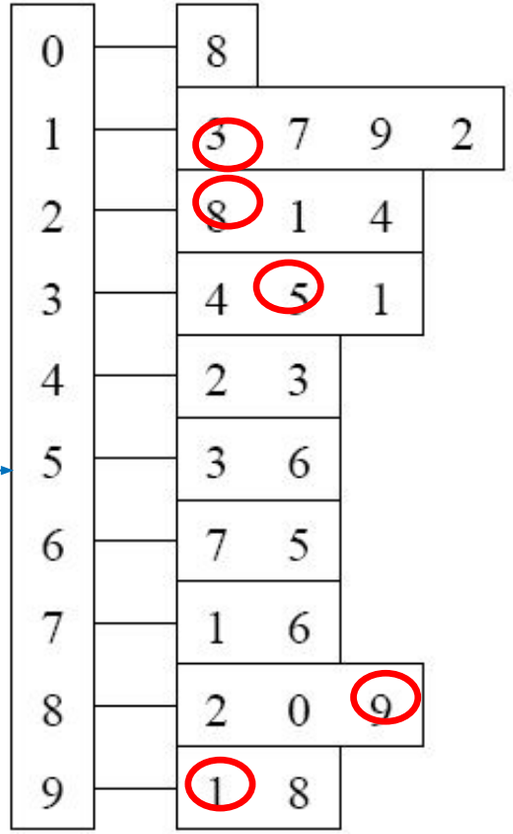
RDFS(4) □ STOP all of 4's neighbors have been visited

Example



Back to 3

Adjacency List



Visited Table (T/F)

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

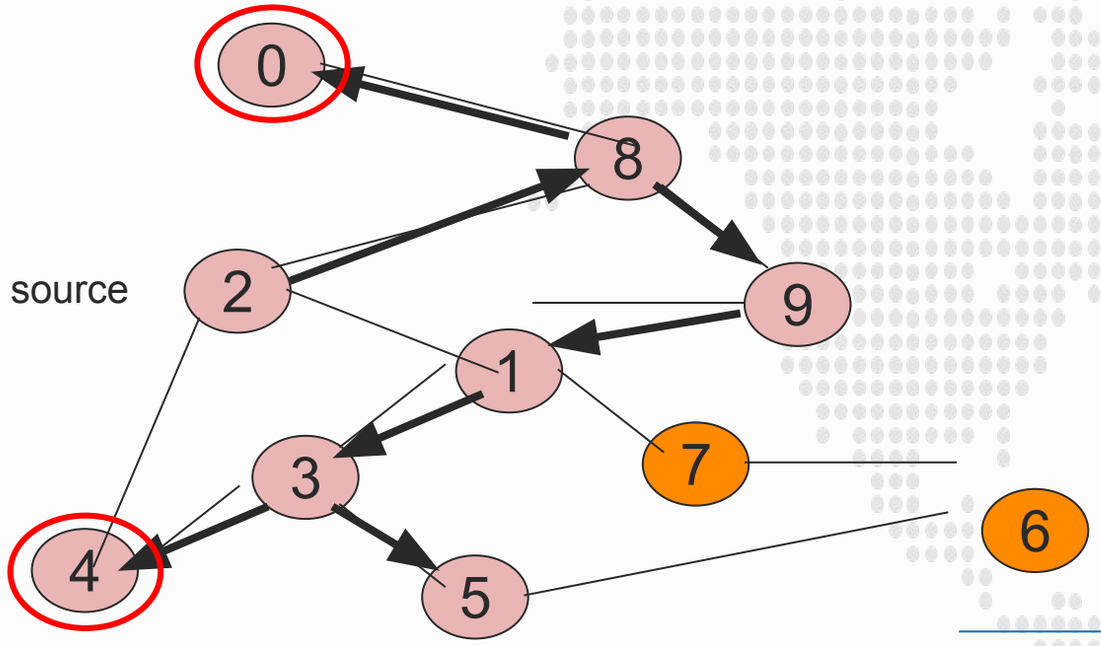
Pred
8
9
-
1
3
-
-
-
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

visit 5 -> RDFS(5)

Example



Adjacency List

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

Visited Table (T/F)

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

Pred
8
9
-
1
3
3
-
-
2
8

Recursive calls

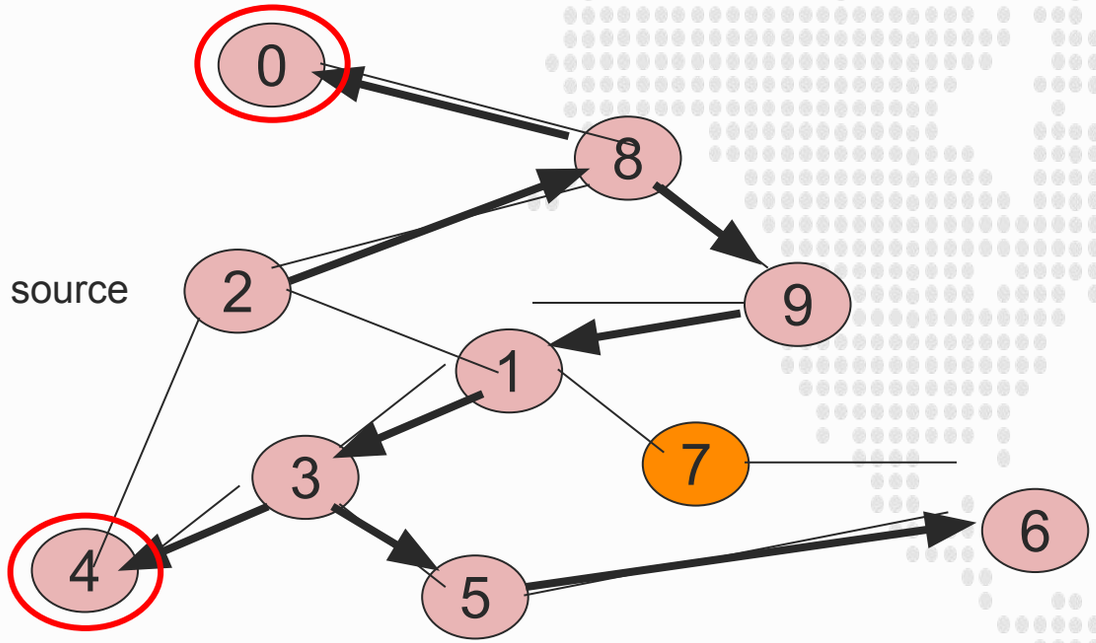
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

RDFS(5)
3 is already visited, so visit 6 -> RDFS(6)

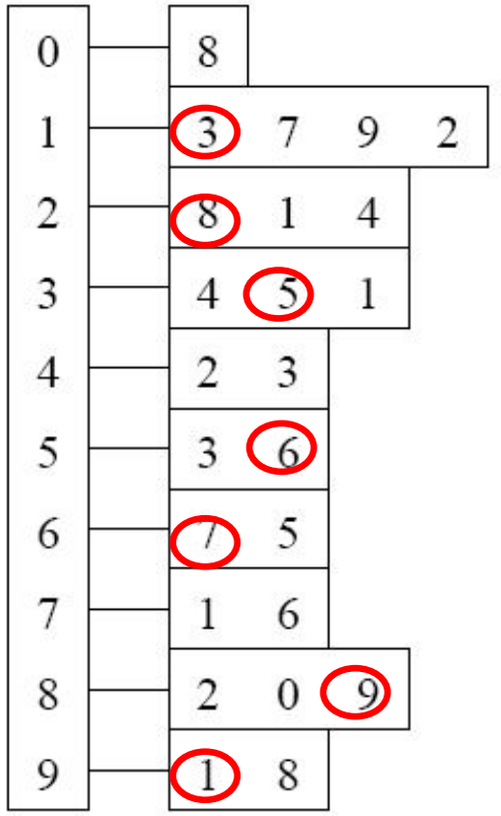
Mark 5 as visited

Mark Pred[5]

Example



Adjacency List



Visited Table (T/F)

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

Pred
8
9
-
1
3
3
5
-
2
8

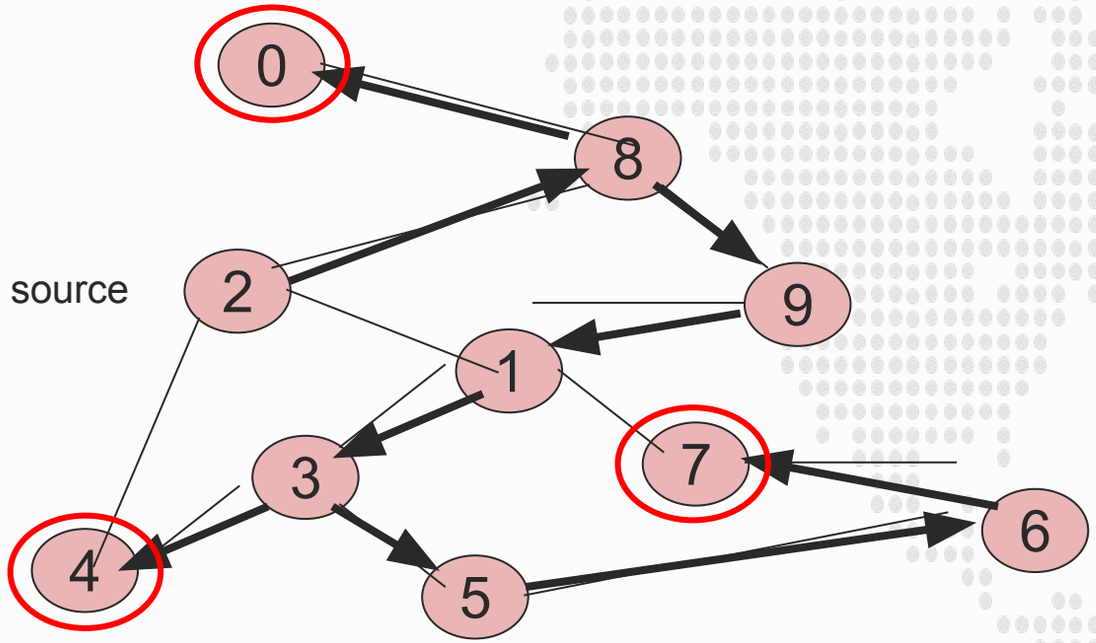
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
visit 7 -> RDFS(7)

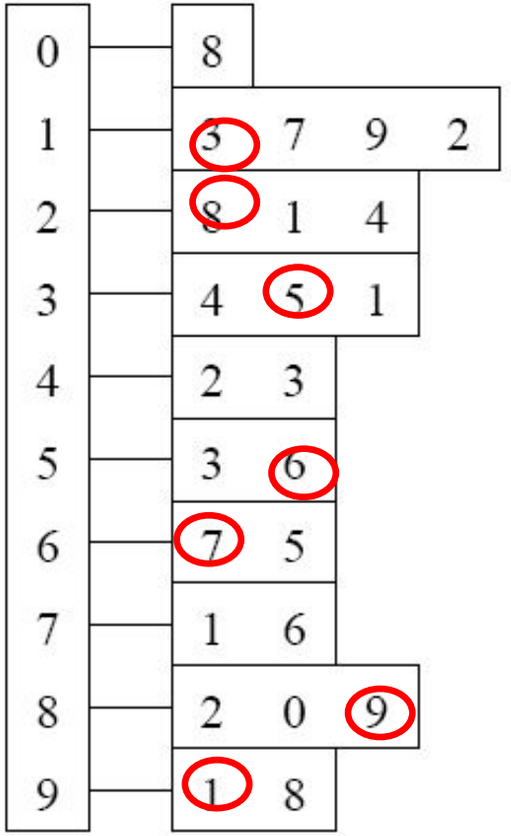
Mark 6 as visited

Mark Pred[6]

Example



Adjacency List



Visited Table (T/F)

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

Pred
8
9
-
1
3
3
5
6
2
8

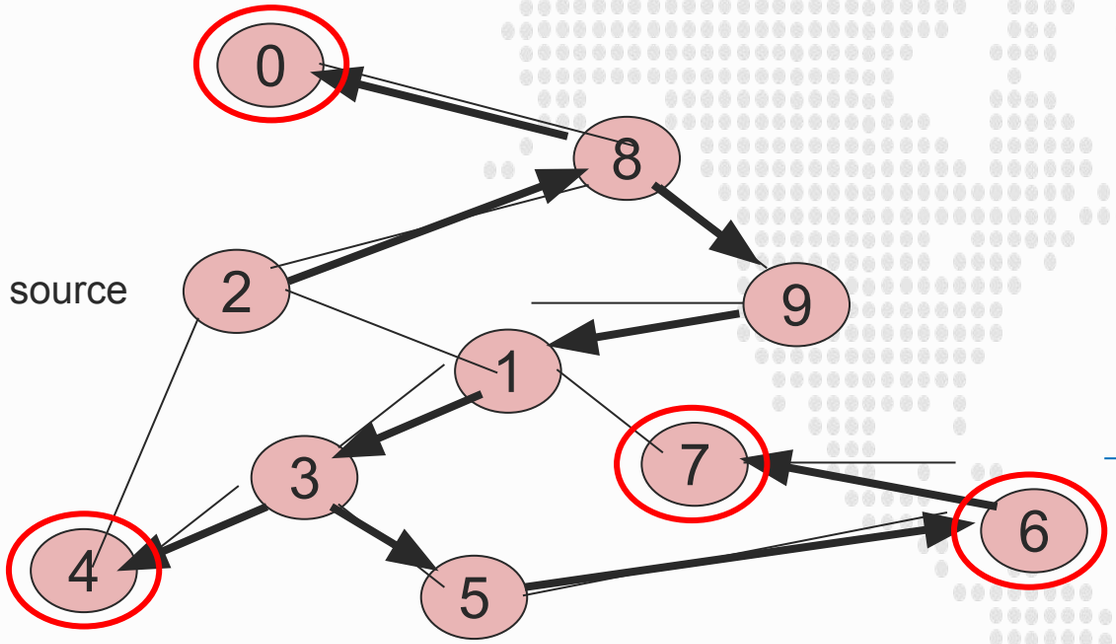
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
RDFS(7) -> Stop no more unvisited neighbors

Recursive calls

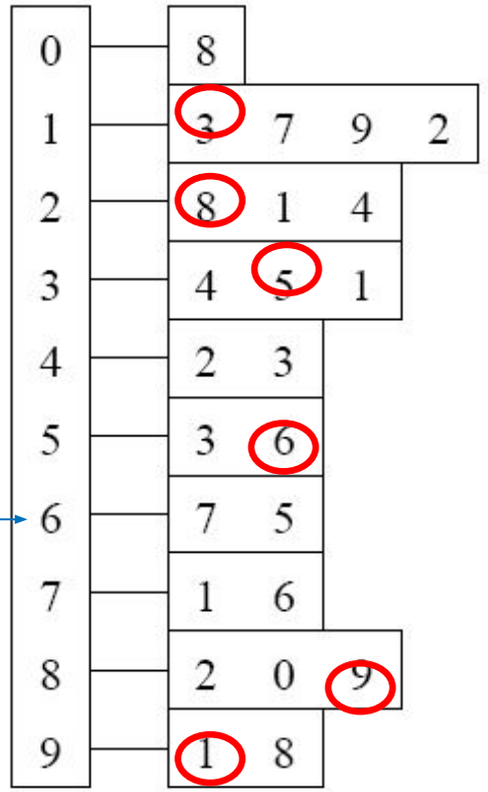
Mark 7 as visited

Mark Pred[7]

Example



Adjacency List



Visited Table (T/F)

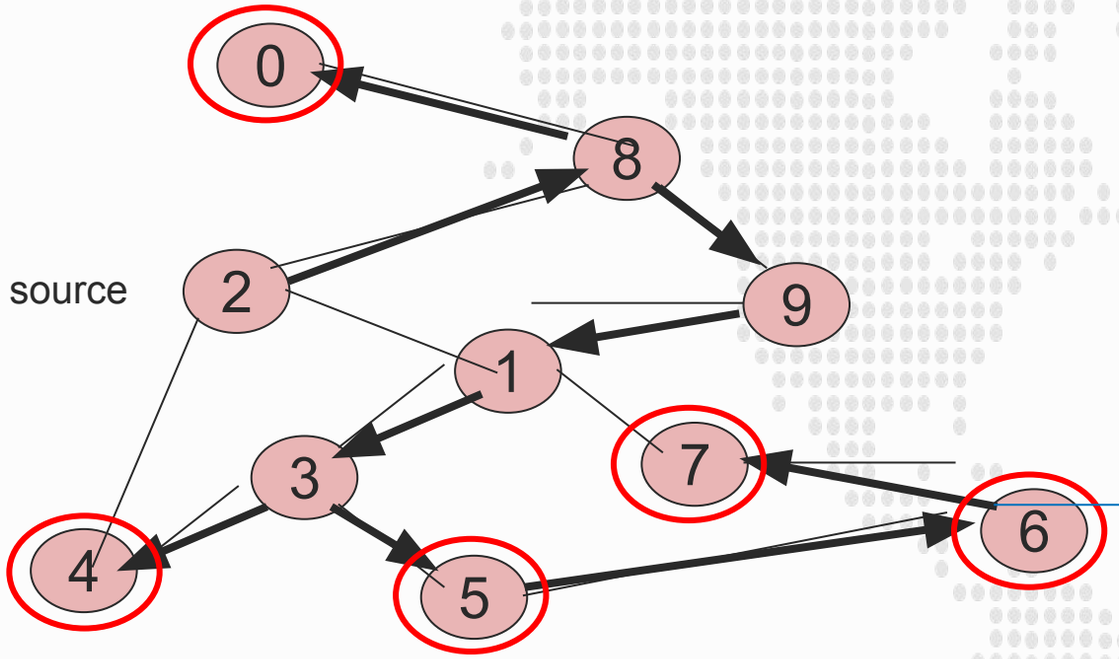
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) -> Stop

Example



Adjacency List

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

Visited Table (T/F)

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

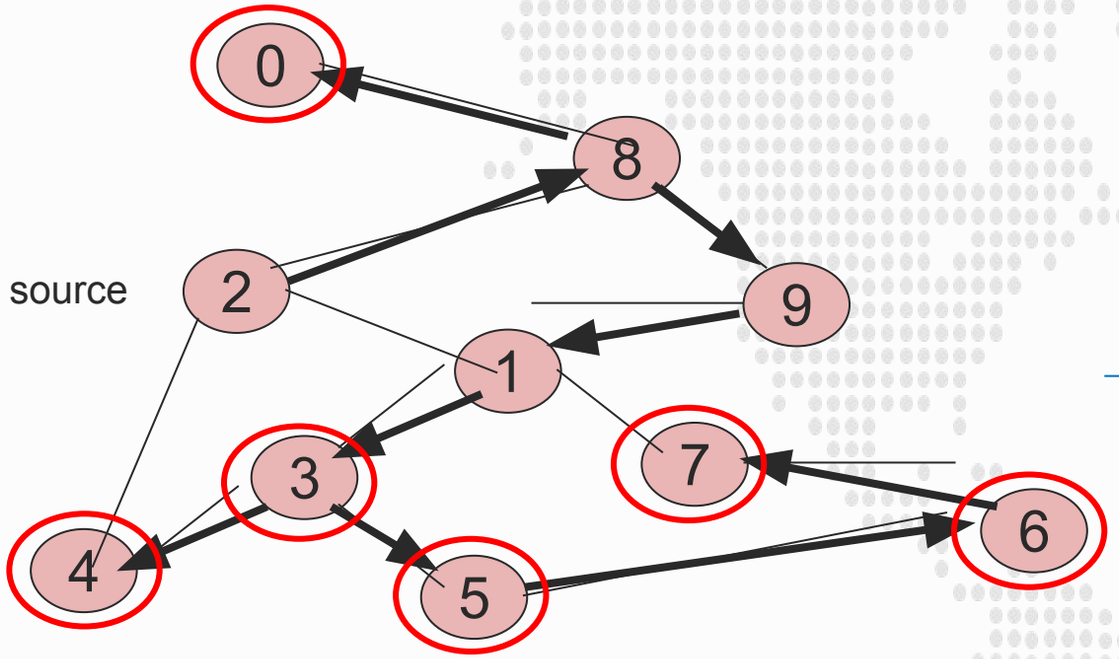
Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

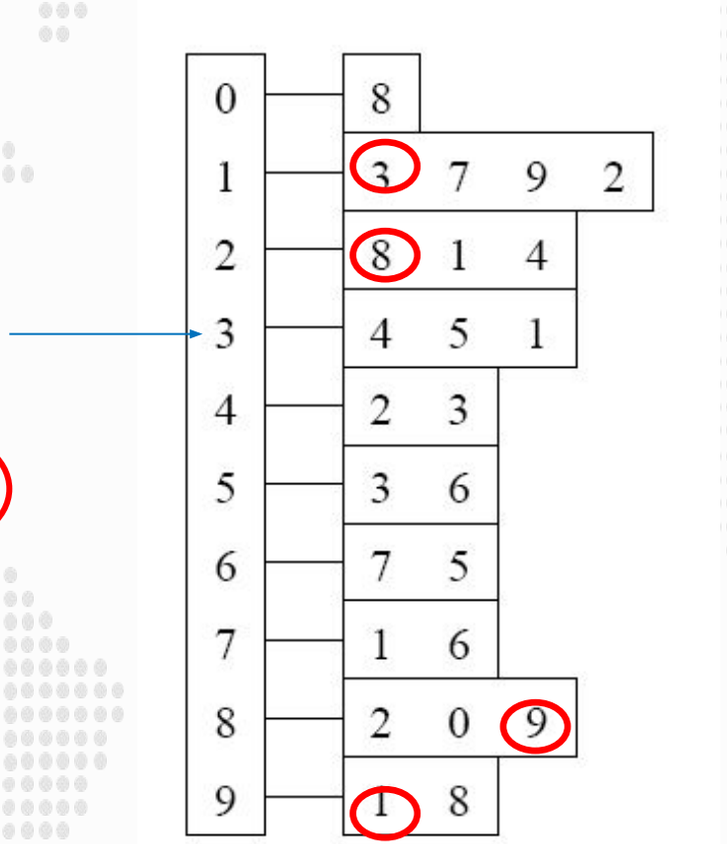
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

RDFS(5) -> Stop

Example



Adjacency List



Visited Table (T/F)

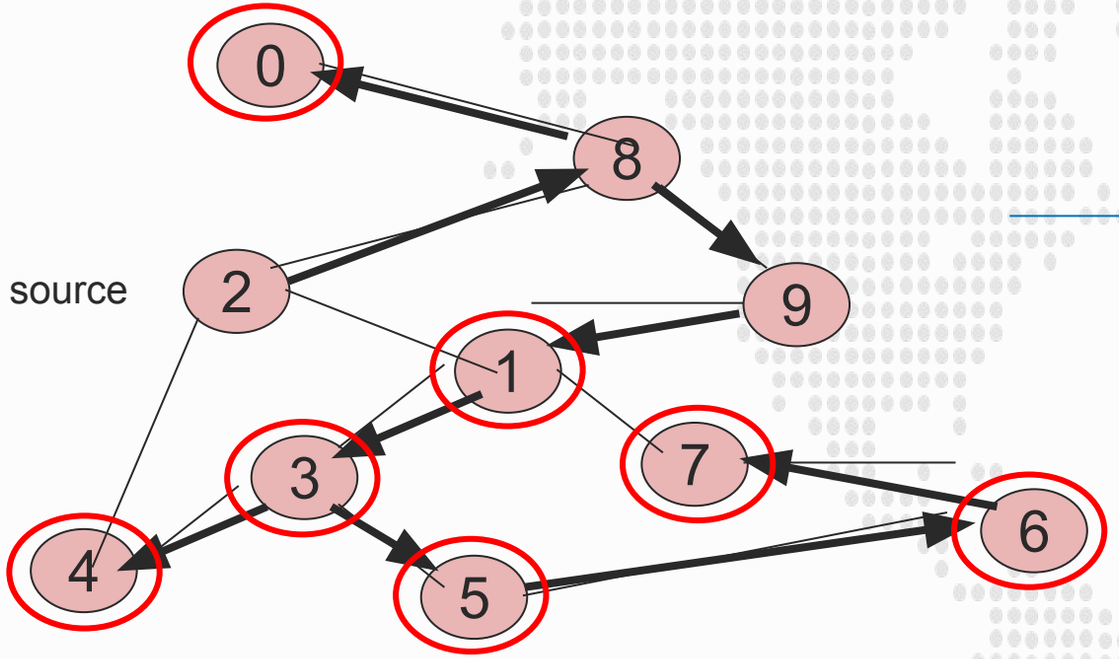
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3) -> Stop

Example



Adjacency List

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

Visited Table (T/F)

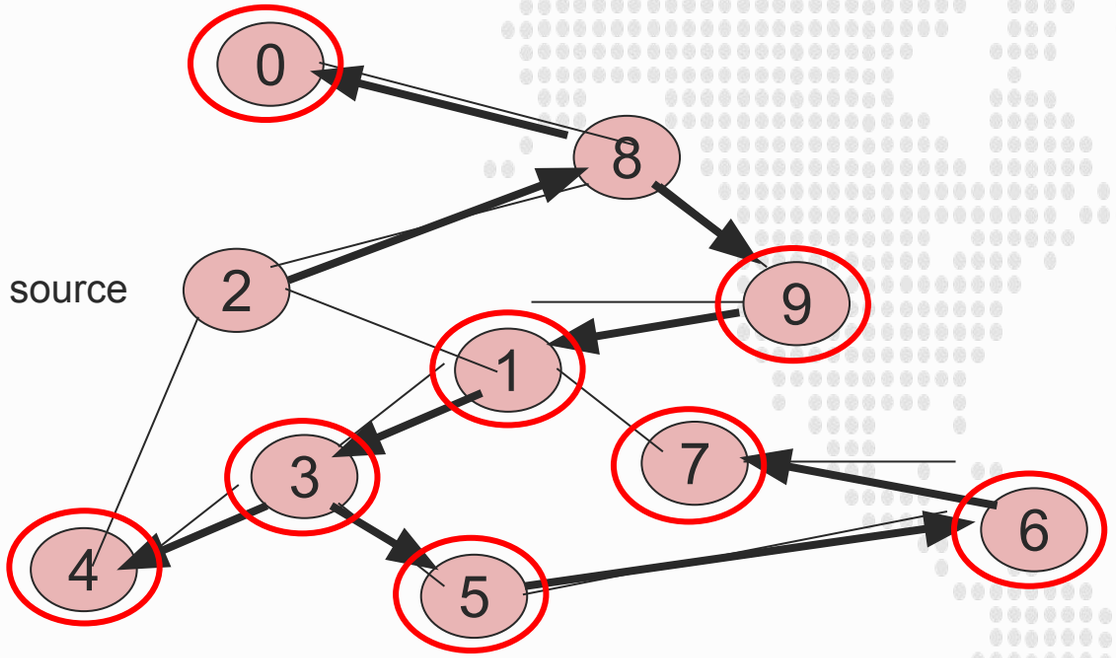
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1) -> Stop

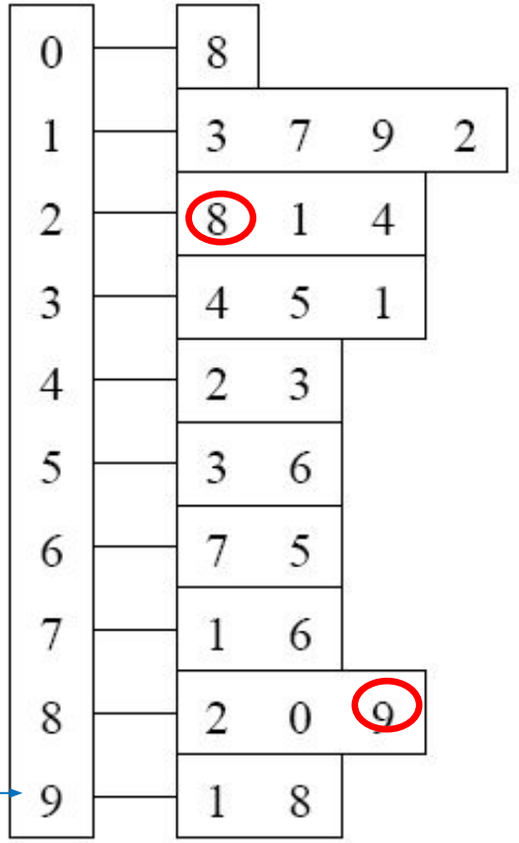
Example



RDFS(2)
RDFS(8)
RDFS(9) -> Stop

Recursive calls

Adjacency List



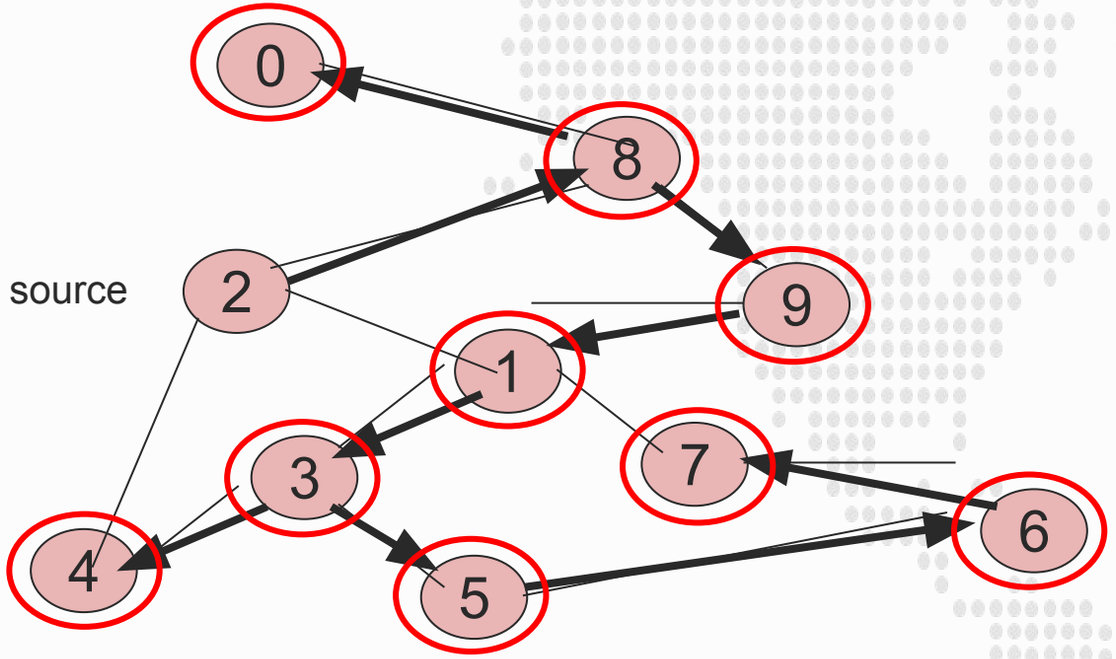
Visited Table (T/F)

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

Pred

8
9
-
1
3
3
5
6
2
8

Example



RDFS(2)

RDFS(8) -> Stop

Recursive calls

Adjacency List

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

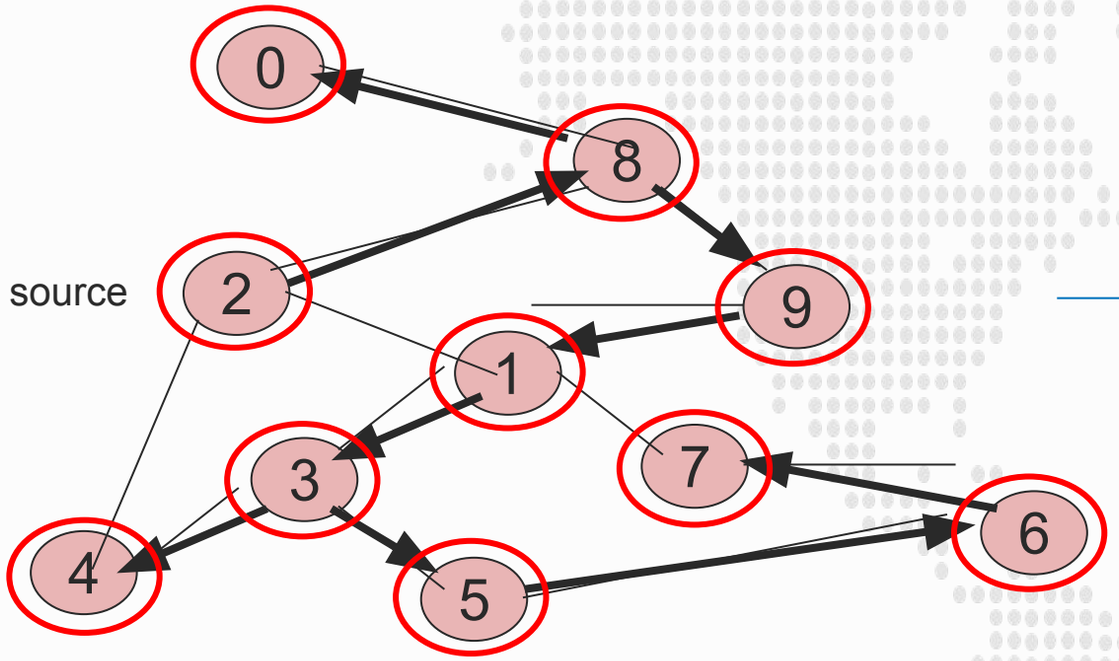
Visited Table (T/F)

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

8
9
-
1
3
3
5
6
2
8

Pred

Example



RDFS(2) -> Stop

Recursive calls

Adjacency List

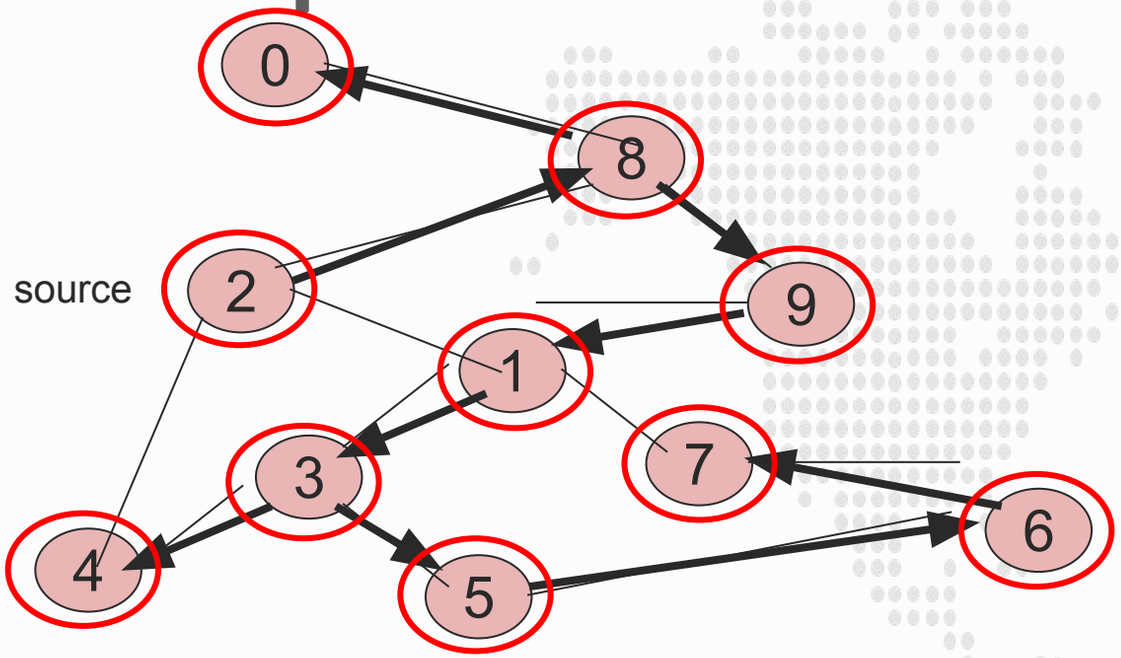
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

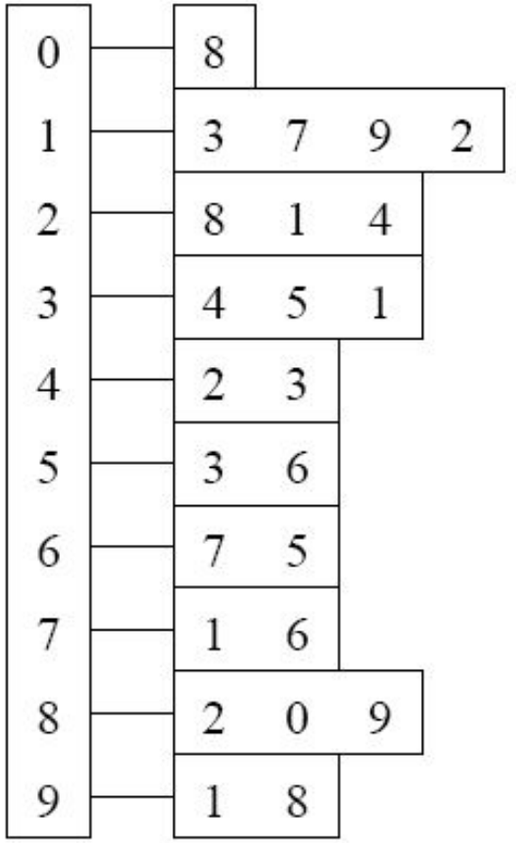
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Example



Adjacency List



Visited Table (T/F)

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

8
9
-
1
3
3
5
6
2
8

Pred

Check our paths, does DFS find valid paths? Yes.

```
Algorithm Path(w)
1.    if pred[w] ≠ −1
2.       then
3.             Path(pred[w]);
4.    output w
```

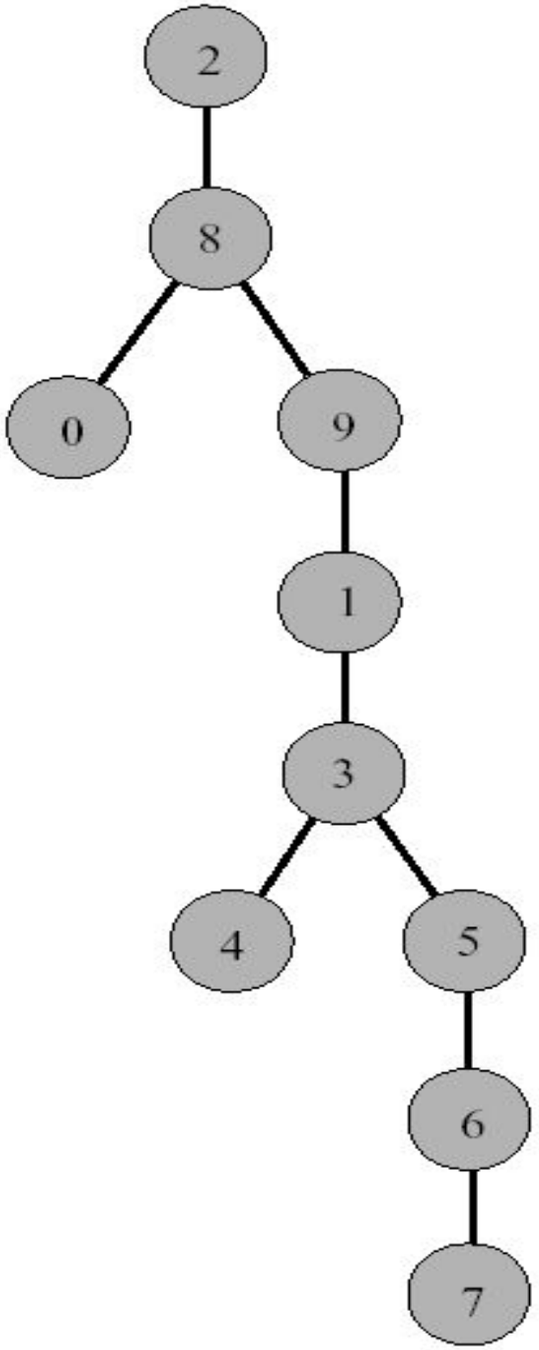
Time Complexity of DFS

(Using adjacency list)

- We never visited a vertex more than once
- We had to examine all edges of the vertices
  - We know $\Sigma_{\text{vertex } v} \text{degree}(v) = 2m$ where m is the number of edges
- So, the running time of DFS is proportional to the number of edges and number of vertices (same as BFS)
  - O(n + m)

DFS Tree

Resulting DFS-tree.
Notice it is much “deeper” than the BFS tree.



Captures the structure of the recursive calls
- when we visit a neighbor w of v, we add w as child of v
- whenever DFS returns from a vertex v, we climb up in the tree from v to its parent

Topological Sort: Complexity

- We never visited a vertex more than one time
- For each vertex, we had to examine all outgoing edges
  - $\Sigma\ outdegree(v) = m$
  - This is summed over all vertices, not per vertex
- So, our running time is exactly
  - O(n + m)

Comparison of Methods

Method	Time Complexity	Space Complexity	Key Features
Kahn's	O(V+E)	O(V)	iterative, In-degree
DFS-Based	O(V+E)	O(V)	Recursive, Stack-Based