# CS2020 – Data Structures and Algorithms Accelerated

## Lecture 14 – How to Explore Your Graph

stevenhalim@gmail.com

National University of Singapore
School of Computing

# Outline

- What are we going to learn in this lecture?
  - Review
    - Graph DS (esp Adjacency List)
    - Binary Tree Traversal
    - Stack/Queue DS
  - Two Graph Traversal Algorithms
    - Breadth First Search (BFS)
    - Depth First Search (DFS)
  - Some Applications
    - Reachability Test
    - Finding Connected Components
    - Topological Sort

# Review – Graph DS

- Last Tuesday, we have covered AdjMatrix & AdjList
- We will use AdjList for most cases
- Vector < Vector < ii > > AdjList;
  - Why use ii?
    - We need to store pair of information for each edge: (neighbor number, weight)
  - Why use Vector of ii?
    - For Vector's **auto-resize feature** ☺: If you have **k** neighbors of a vertex, just add **k** times to an initially empty Vector of ii of this vertex.
    - You can replace this with Java List if you want to…
  - Why use Vector of Vector of ii?
    - For Vector's **indexing feature** ☺: if we want to enumerate neighbors of vertex u, use **AdjList.get(u)** to access the correct List (Vector) of ii

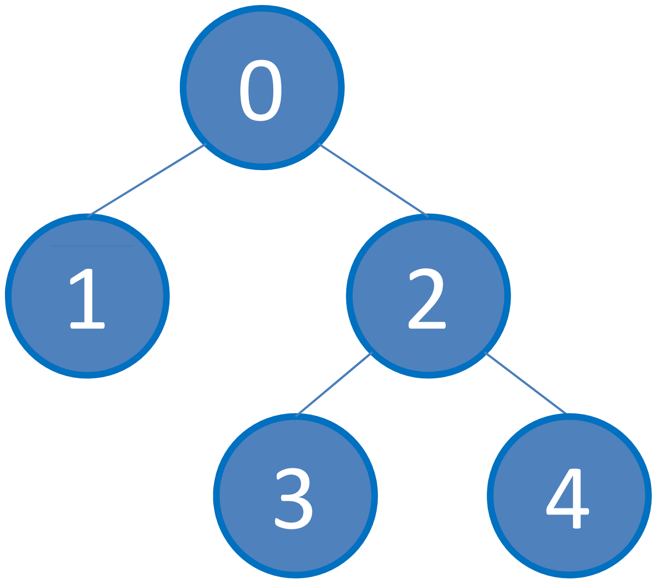
# Review – **Binary Tree** Traversal

- In a binary tree, there are three standard traversal:
  - Preorder
  - **Inorder**
  - Postorder
  - (I skip "level order" – it looks like BFS)

Inorder discussed in Lect5

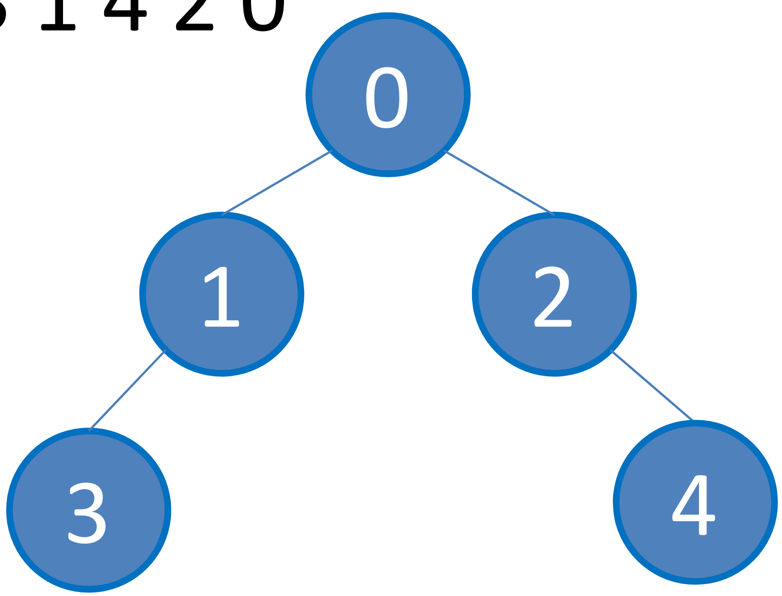
| | | |
|---|---|---|
| `pre(u)`<br>`  visit(u);`<br>`  pre(u->left);`<br>`  pre(u->right);` | `in(u)`<br>`  in(u->left);`<br>`  visit(u);`<br>`  in(u->right);` | `post(u)`<br>`  post(u->left);`<br>`  post(u->right);`<br>`  visit(u);` |

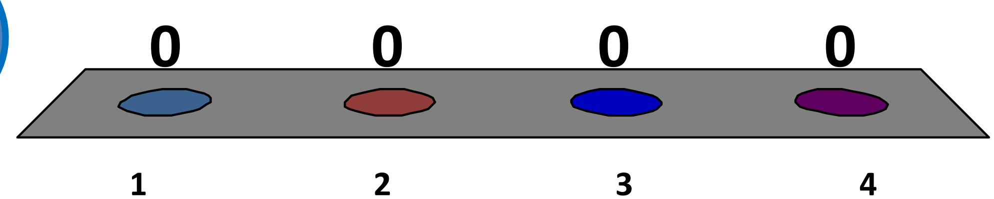
- We start binary tree traversal from:
  - pre(root)/in(root)/post(root)
    - pre = 0, 1, 2, 3, 4
    - in = 1, 0, 3, 2, 4
    - post = 1, 3, 4, 2, 0



# Quick Test, what is the **Post**Order Traversal of this Binary Tree?

1. 0 1 2 3 4
2. 0 1 3 2 4
3. 3 1 0 2 4
4. 3 1 4 2 0

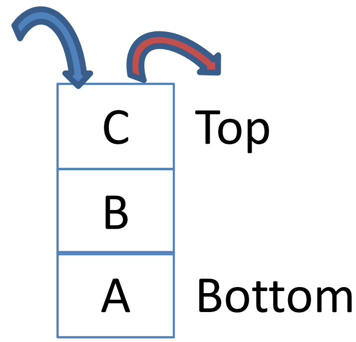


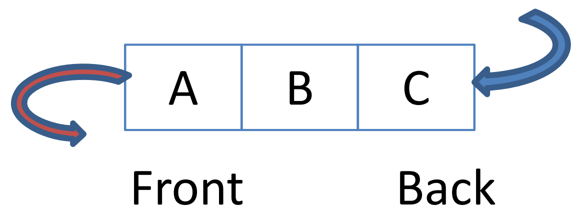

0 of 54

# Review – Stack/Queue DS

- Stack
  - Last In First Out (LIFO)
  - Demo: Java Stack




- Queue
  - First In First Out (FIFO)
  - Demo: Java Queue




- We do not have to create our own Stack/Queue
  - Use Java Collections framework!
  - See StackQueueDemo.java

# Traversing a Graph (1)

- Two ingredients are needed for a **traversal**
  1. The start
  2. The movement
- Defining the start ("source")
  - In tree, we *normally* start from root
    - Note: not all tree are rooted though,
      in that case, we have to select one vertex as the "source",
      as in general graph below
  - In general graph, we do not have the notion of root
    - Instead, we start from a distinguished vertex
    - We call this vertex as the **"source" s**

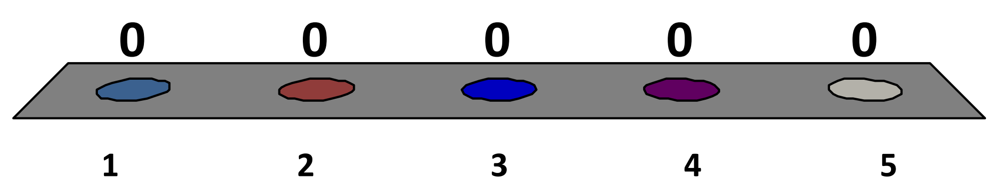
# Traversing a Graph (2)

- Defining the movement:
  - In (binary) tree, we only have (at most) two choices:
    - Go to the left **subtree** or to the right **subtree**
  - In general graph, we can have more choices:
    - If **vertex u** and **vertex v** are adjacent/connected with edge (u, v);
      and we are now in **vertex u**;
      then we can also go to **vertex v** by traversing that edge (u, v)
  - In (binary) tree, there is **no cycle**
  - In general graph, we **may have (trivial/non trivial) cycles**
    - We need a way to avoid revisiting u → v → u → u → … indefinitely
- Solution: BFS and DFS ☺

# More Detailed Survey of BFS

## What is your level of understanding as of now?

1. I have not heard about BFS, tell me please ☺
2. I have heard about BFS, but not the details :O
3. I know the theoretical details about BFS but have not implement/code it even once ☹
4. I know and have implemented BFS, but I prefer ‘simpler’ DFS
5. I know and have implemented BFS and I know that it is useful for solving SSSP on unweighted graph (if you say ‘what is this’?, do not select this option)




**0 of 54**

# Breadth First Search (BFS)

- Key ideas:
  - Start from **s**; If a vertex **v** is reachable from **s**, then all neighbors of **v** will also be reachable from **s** (recursive definition)
  - BFS visits vertices of G in *breadth-first* manner (when viewed from source vertex s)
    - How to maintain such order?
      - Queue **Q**, initially, it contains only **s**
    - How to differentiate visited vs not visited vertices (to avoid cycle)?
      - 1D array/Vector **visited** of size V, **visited[v] = 0** initially, and **visited[v] = 1** when **v** is visited
    - How to memorize the path?
      - 1D array/Vector **p** of size V, **p[v]** denotes the **p**redecessor (or **p**arent) of **v**

# BFS Pseudo Code

```
for all v in V
  visited[v] ← 0
  p[v] ← -1
Q ← {s} // start from s
visited[s] ← 1
```

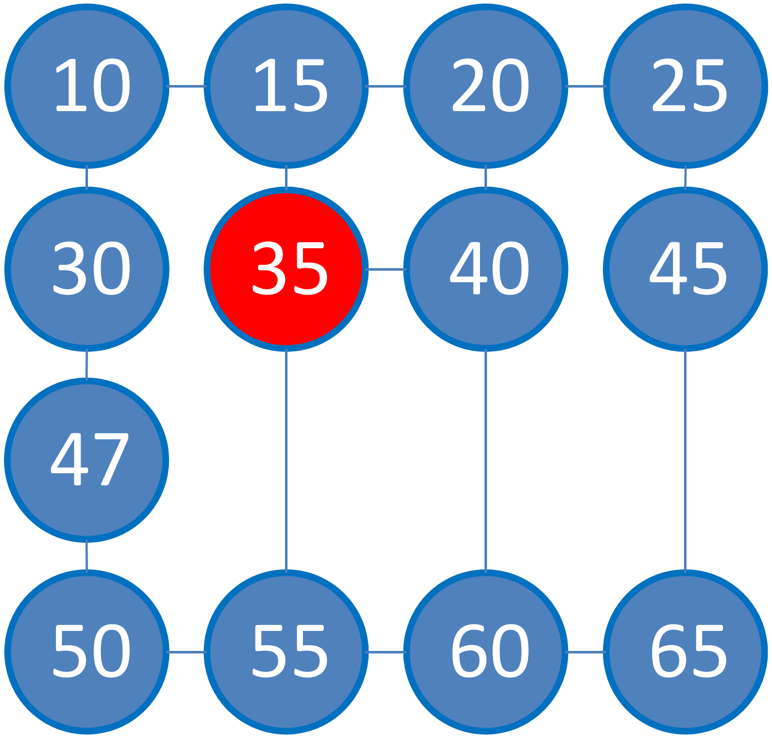
Initialization phase

```
while Q is not empty
  u ← Q.dequeue()
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences BFS
      visited[v] ← true // visitation sequence
      p[v] ← u
      Q.enqueue(v)
```

Main loop

```
// we can then use information stored in visited/p
```

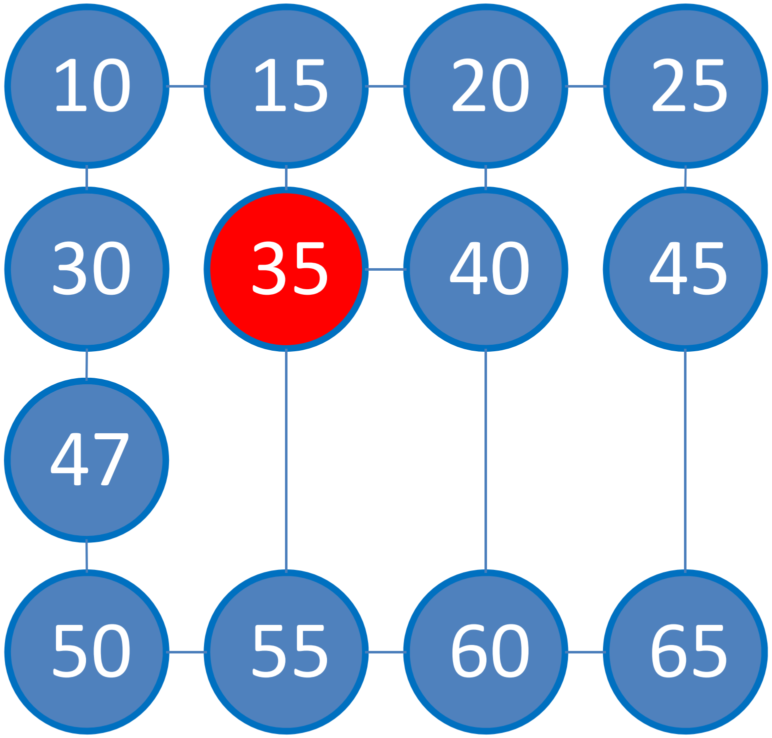
# Example (1)




Q = {35}
Q = {15, 40, 55}

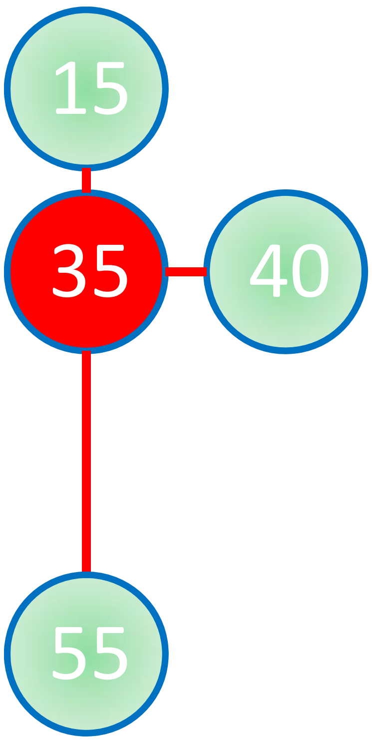
Neighbors are listed in increasing order

# Example (2)

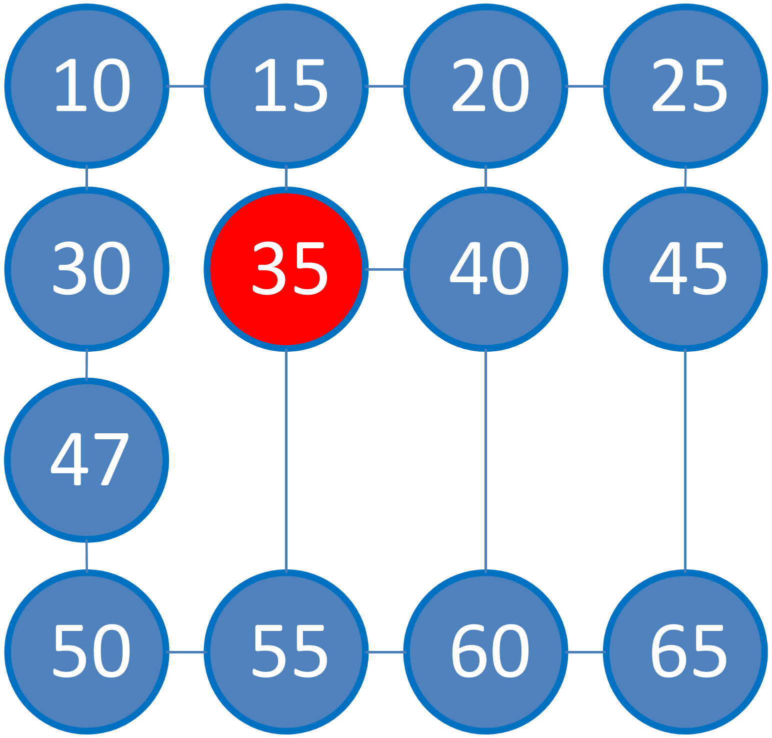


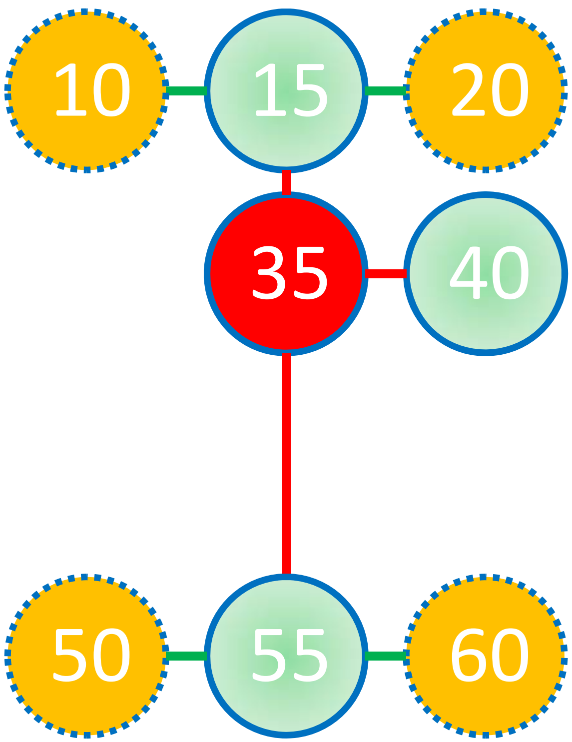

Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}

Neighbors are listed in increasing order



# Example (3)

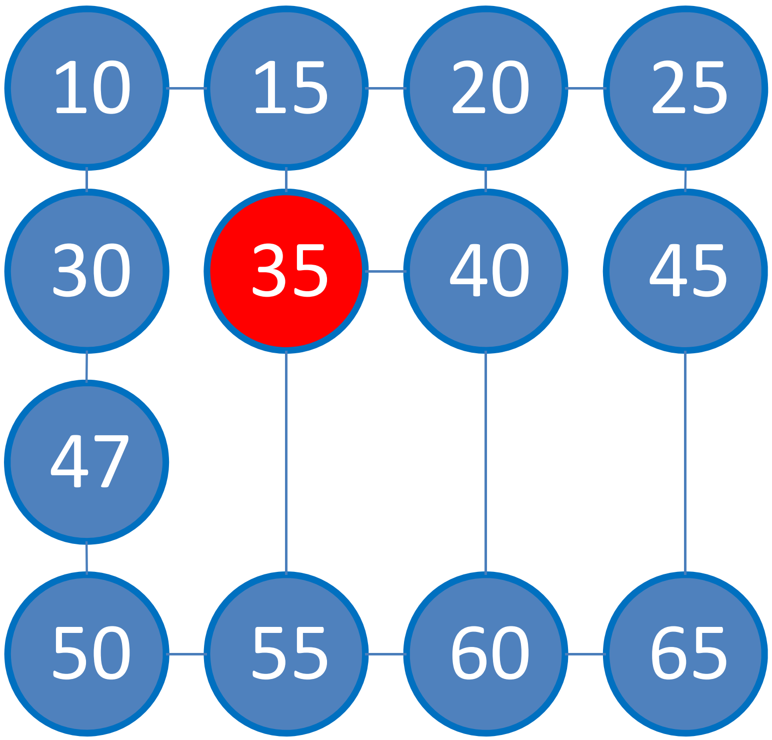


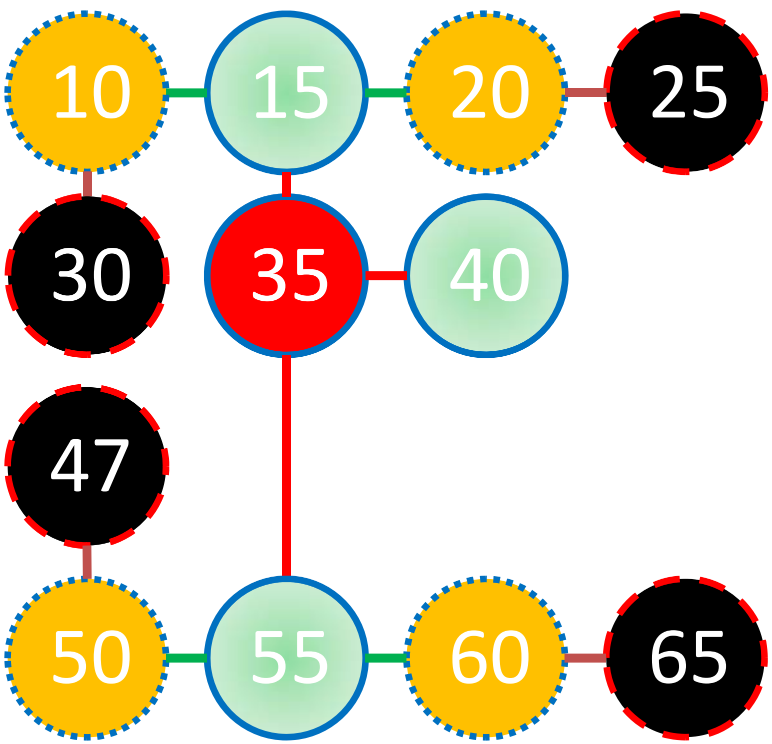




Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}
Q = {20, 60, 50, **30**}
Q = {60, 50, 30, **25**}
Q = {50, 30, 25, **65**}
Q = {30, 25, 65, **47**}

Neighbors are listed in increasing order

# Example (4)

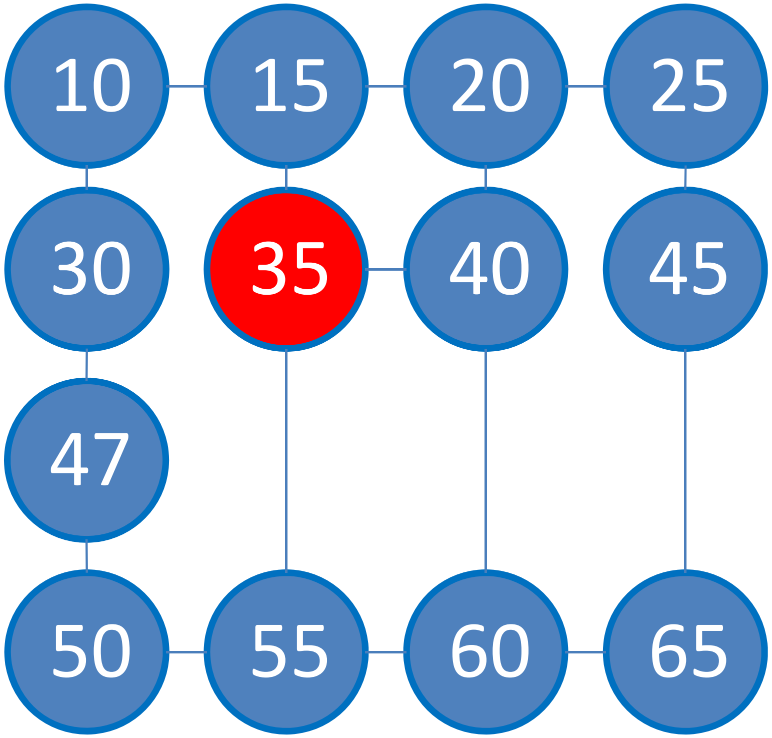



Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}
Q = {20, 60, 50, **30**}
Q = {60, 50, 30, **25**}
Q = {50, 30, 25, **65**}
Q = {30, 25, 65, **47**}
Q = {25, 65, 47}
Q = {65, 47, **45**}
Q = {47, 45}
Q = {45}

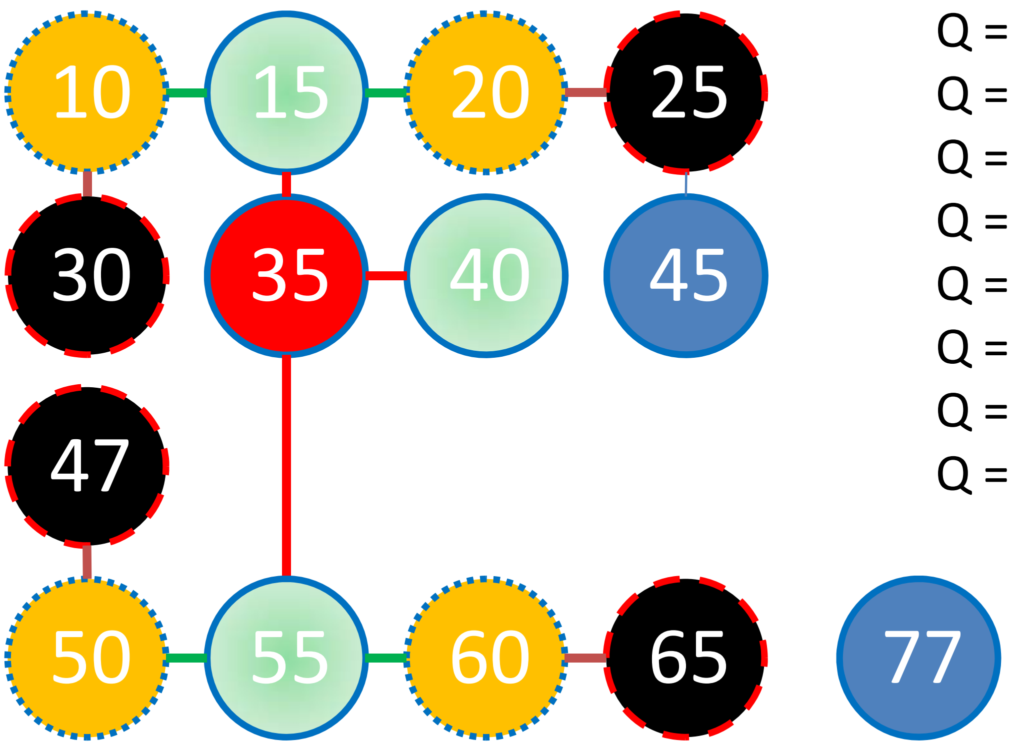
Neighbors are listed in increasing order

# Example (5)




Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}
Q = {20, 60, 50, **30**}
Q = {60, 50, 30, **25**}
Q = {50, 30, 25, **65**}
Q = {30, 25, 65, **47**}
Q = {25, 65, 47}
Q = {65, 47, **45**}
Q = {47, 45}
Q = {45}
Q = {}

Neighbors are listed in increasing order




To think about:

What if we have another vertex "77" that is not connected with any other vertex?
Any consequences?

# BFS Analysis

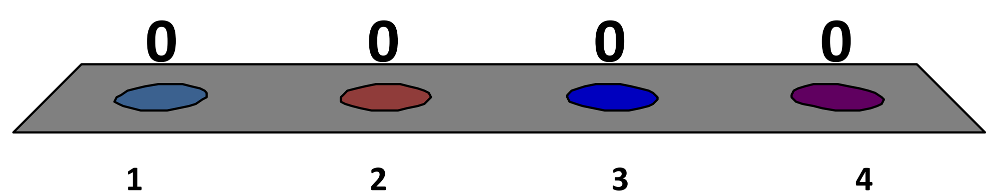
```
for all v in V
  visited[v] ← 0
  p[v] ← -1
Q ← {s} // start from s
visited[s] ← 1

while Q is not empty
  u ← Q.dequeue()
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences BFS
      visited[v] ← true // visitation sequence
      p[v] ← u
      Q.enqueue(v)

// we can then use information stored in visited/p
```

Time Complexity: O(V + E)

- Each vertex is only in the queue once ~ O(V)
- Every time a vertex is dequeued, all its k neighbors are scanned; After all vertices are dequeued, all E edges are examined ~ O(E)
  → assuming that we use **Adjacency List**!
- Overall: O(V + E)

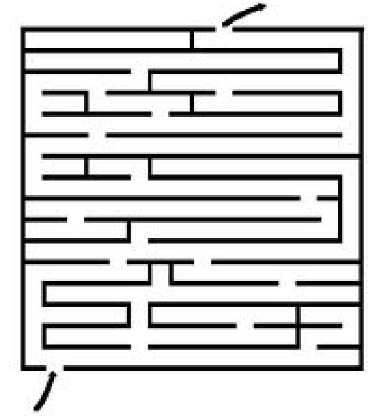
# More Detailed Survey of DFS
## What is your level of understanding as of now?

1. I have not heard about DFS, tell me please ☺
2. I have heard about DFS, but not the details :O
3. I know the theoretical details about DFS but have not implement/code it even once ☹
4. I know and have implemented DFS and I also know that DFS is useful for finding articulation points, bridges, SCC (if you say ‘what are these’?, do not select this option)




**0 of 54**

# Depth First Search (DFS)



- Key ideas:
  - Start from **s;** If a vertex **v** is reachable from **s**, then all neighbors of **v** will also be reachable from **s** (recursive definition)
  - DFS visits vertices of G in *depth-first* manner (when viewed from source vertex s)
    - How to maintain such order?
      - Stack **S**, but we will simply use recursion (implicit stack)
    - How to differentiate visited vs not visited vertices (to avoid cycle)?
      - 1D array/Vector **visited** of size V,
        **visited[v] = 0** initially, and **visited[v] = 1** when **v** is visited
    - How to memorize the path?
      - 1D array/Vector **p** of size V,
        **p[v]** denotes the **p**redecessor (or **p**arent) of **v**

# DFS Pseudo Code

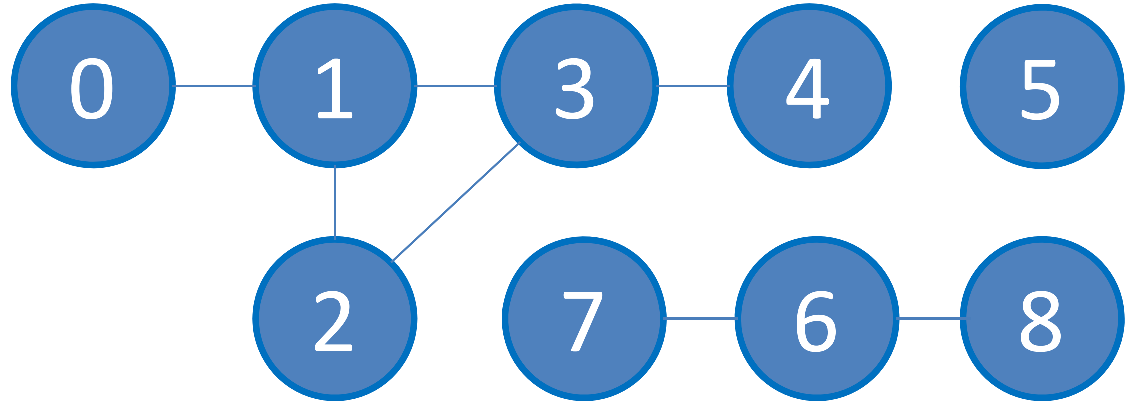
```
DFSrec(u)
  visited[v] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)
```

Recursive phase

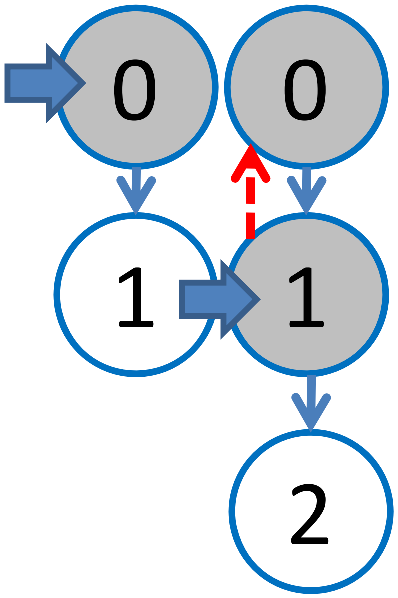
```
// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1
DFSrec(s) // start the
recursive call from s
```

Initialization phase, same as with BFS

# Example (1)

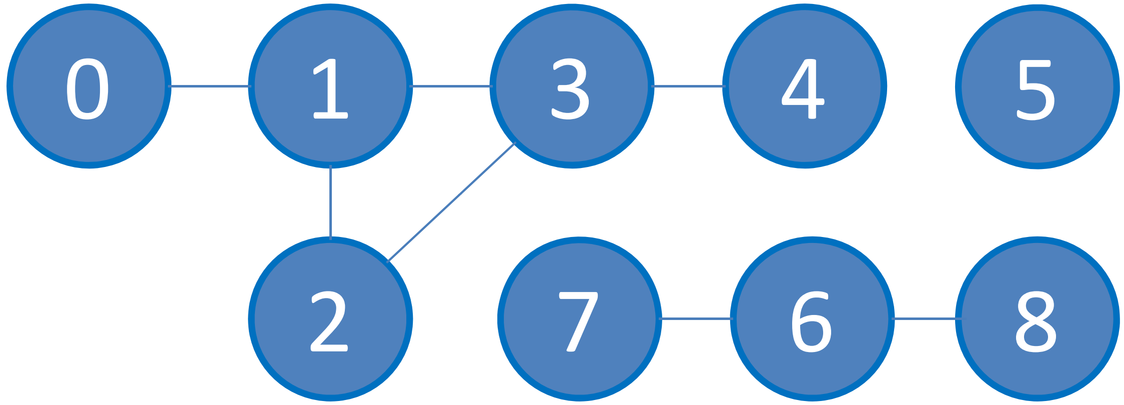



Assume that we start from source s = 0,
neighbors are listed in ascending order

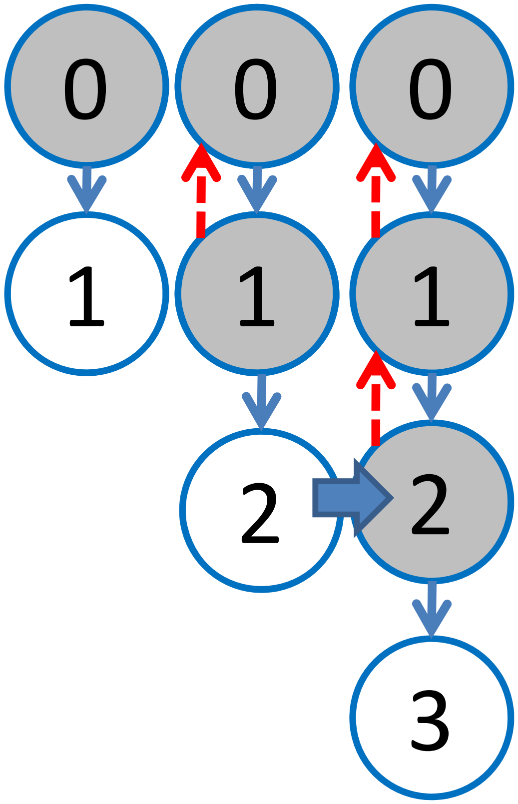



At vertex 1, we cannot go back to vertex 0 as it has been “flagged”;
but we can continue (more depth) to vertex 2 **or vertex 3;**
assume for this case we visit vertex 2 first (ascending order)

# Example (2)

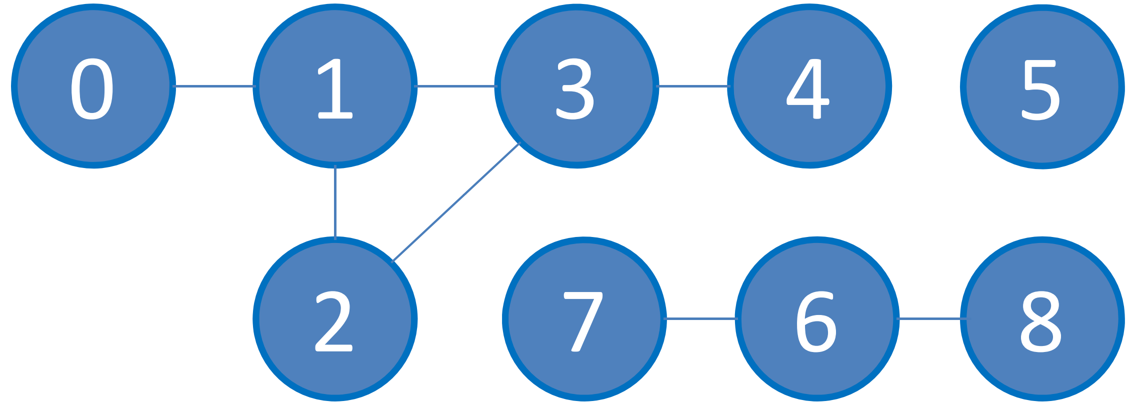



Assume that we start from source s = 0,
neighbors are listed in ascending order

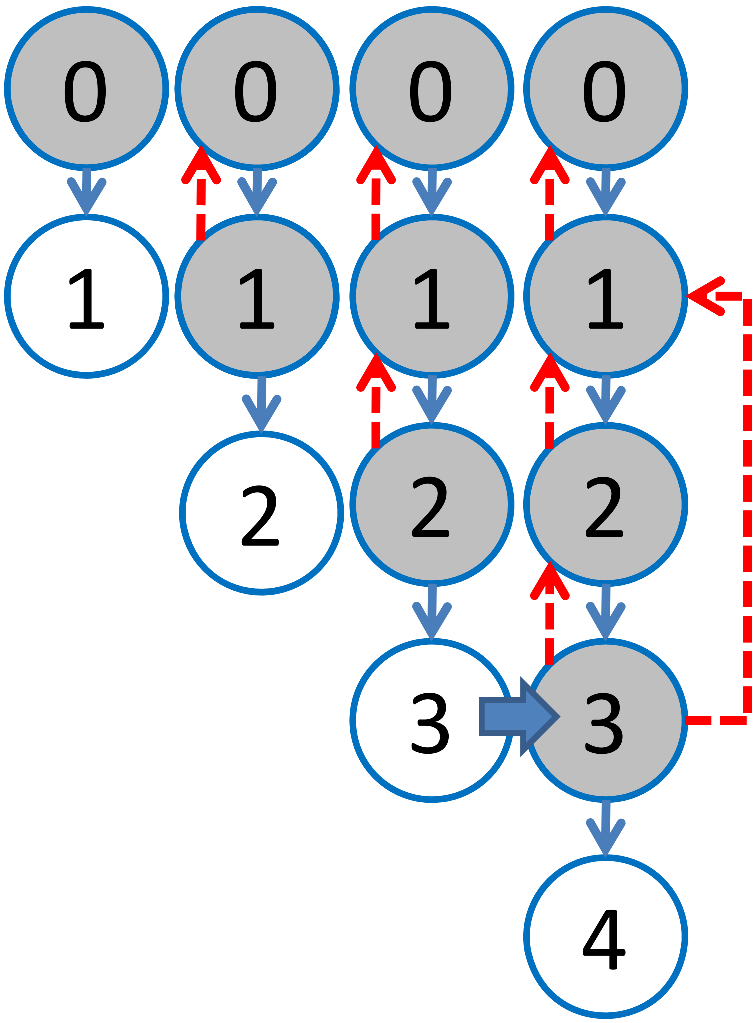



At vertex 2, we cannot go back to vertex 1 as it has been “flagged”;
But we can continue (more depth) to vertex 3

# Example (3)

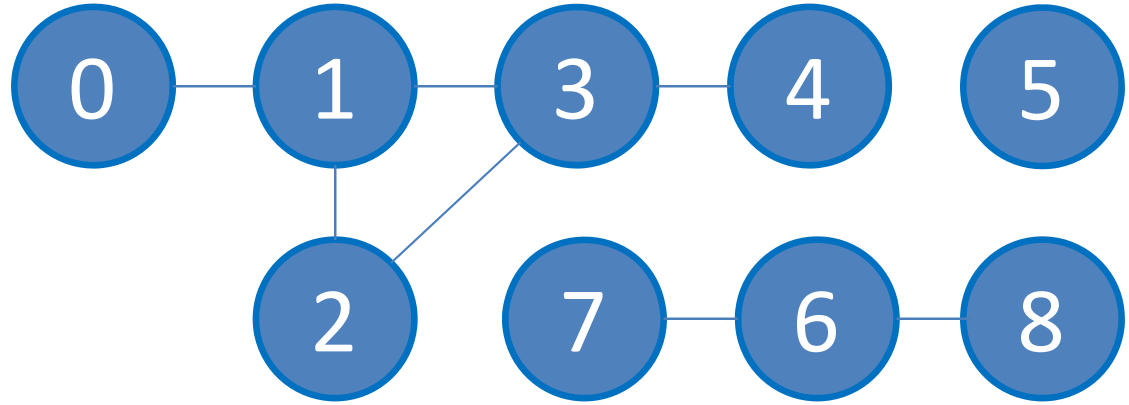



Assume that we start from source s = 0,
neighbors are listed in ascending order

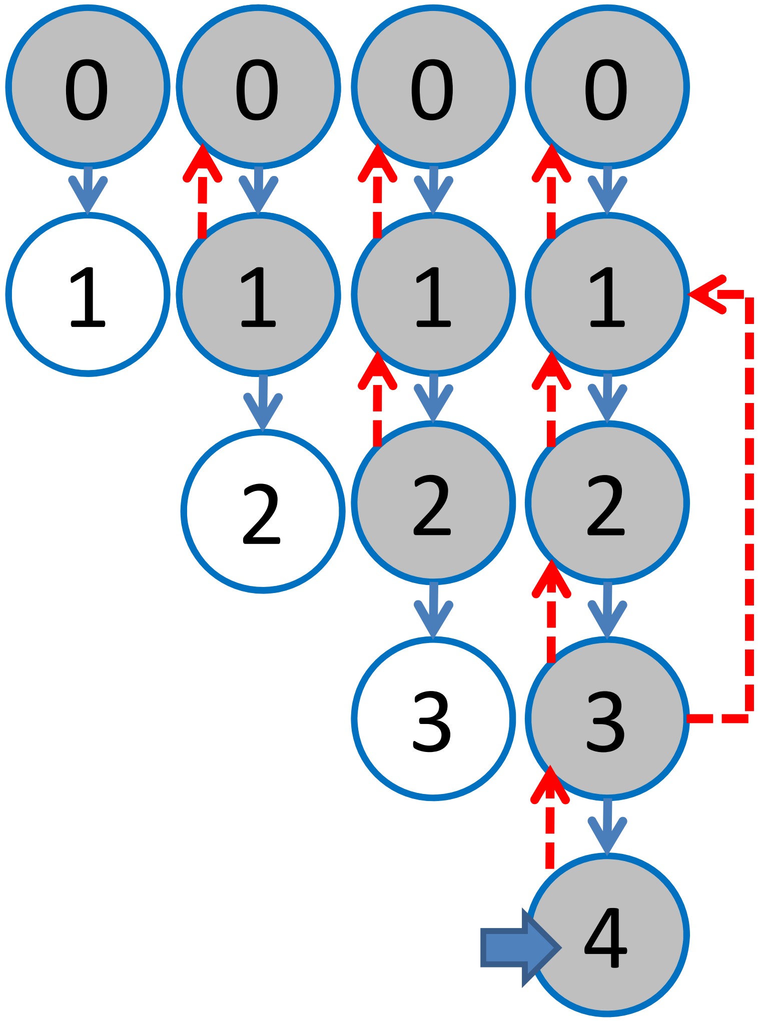



At vertex 3, we cannot go back to vertex 1 or to vertex 2 as both have been “flagged”;
But we can continue (more depth) to vertex 4

# Example (4)

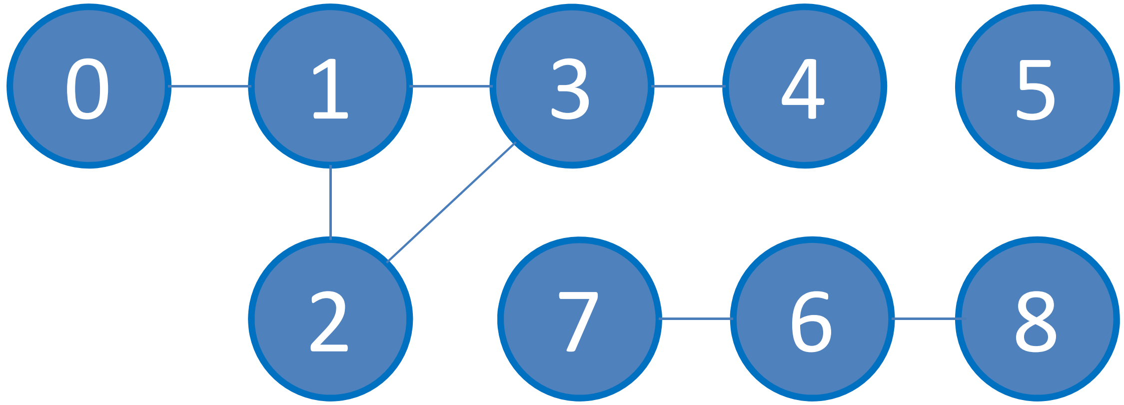



Assume that we start from source s = 0, neighbors are listed in ascending order

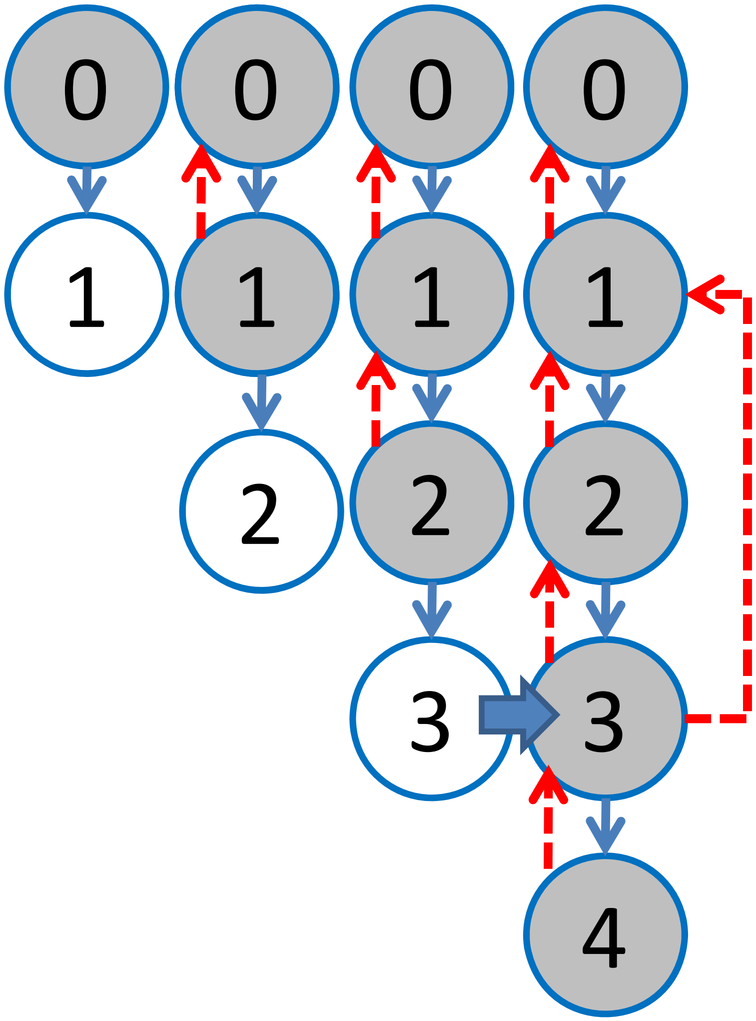



At vertex 4, we cannot go back to vertex 3 as it has been “flagged”;
All neighbors of vertex 4 have been explored, we now “backtrack” to previous vertex

# Example (5)

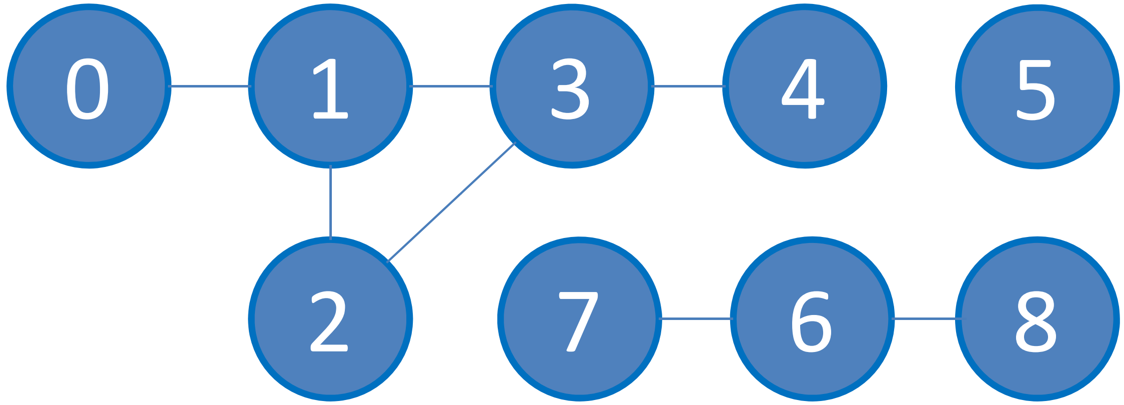



Assume that we start from source s = 0, neighbors are listed in ascending order

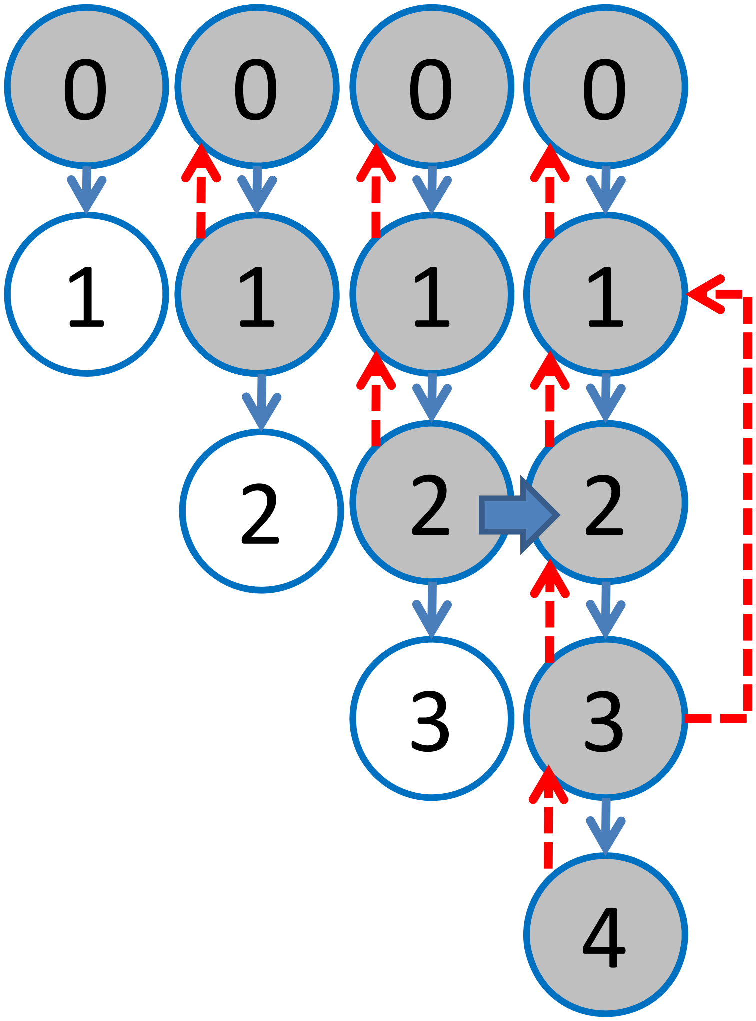



Back at vertex 3, all 3 neighbors have now been visited, we backtrack again

# Example (6)

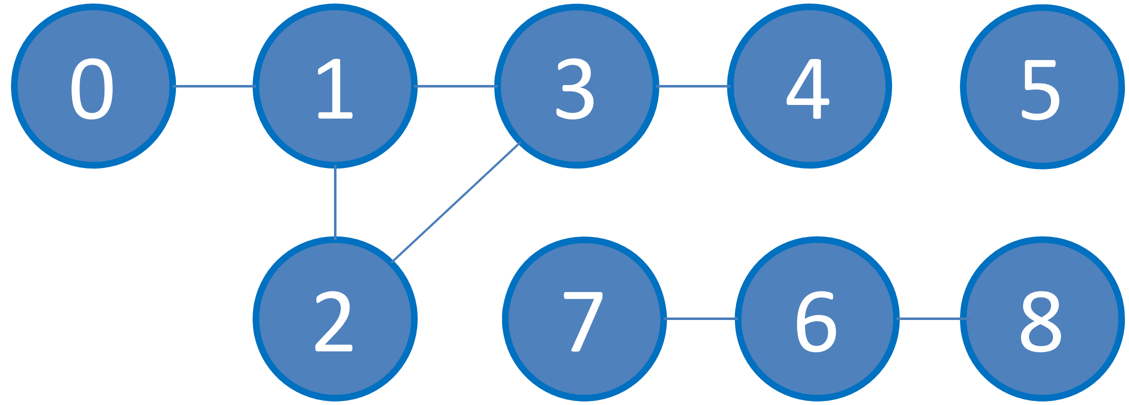



Assume that we start from source s = 0, neighbors are listed in ascending order

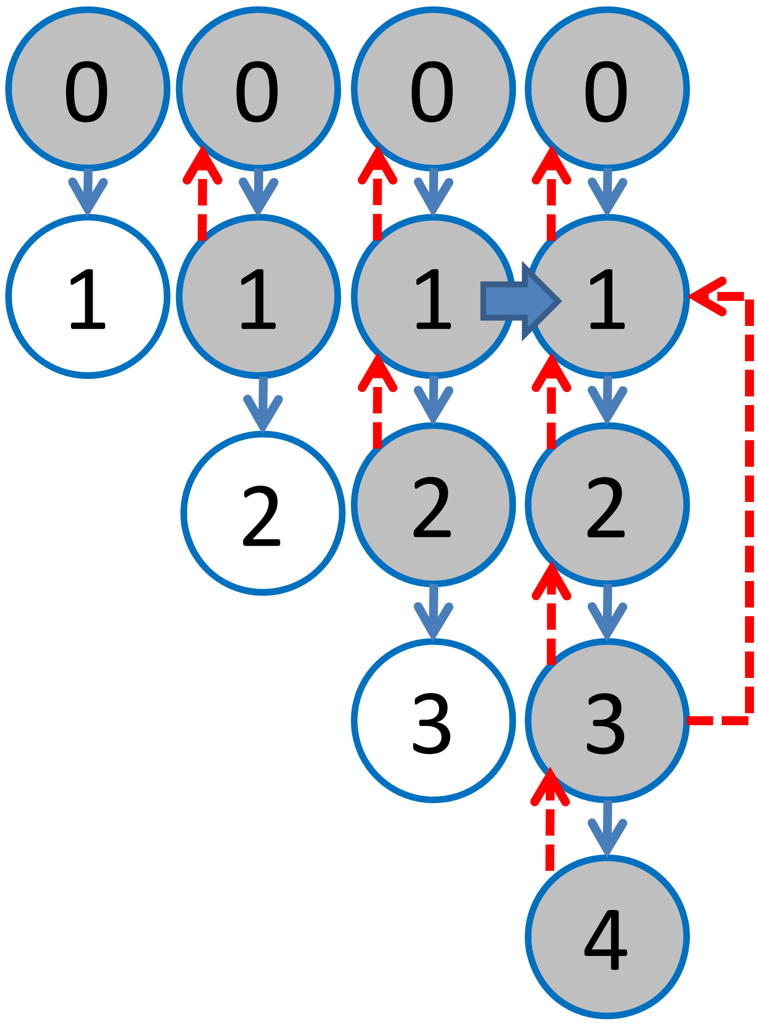



Back at vertex 2, all 2 neighbors have now been visited, we backtrack again

# Example (7)




Assume that we start from source s = 0, neighbors are listed in ascending order




Back at vertex 1, all 3 neighbors have now been visited, we backtrack again to starting vertex 0, DONE

# DFS Analysis

```
DFSrec(u)
  visited[v] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)

// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1
DFSrec(s) // start the
recursive call from s
```

Time Complexity: O(V + E)

- Each vertex is only visited once O(V), then it is flagged to avoid cycle
- Every time a vertex is visited, all its k neighbors are scanned; Thus after all V vertices are visited, we have examined all E edges ~ O(E) → assuming that we use **Adjacency List**!
- Overall: O(V + E)

# Path Reconstruction Algorithm (1)

```
// iterative version (will produce reversed output)
Output "(Reversed) Path:"
i ← t // start from end of path: suppose vertex t
while i != s
  Output i
  i ← p[i] // go back to predecessor of i
Output s



// try it on this array p, t = 4
// p = {-1, 0, 1, 2, 3, -1, -1, -1}
```

# Path Reconstruction Algorithm (2)

```
void backtrack(u)
  if (u == -1) // recall: predecessor of s is -1
    stop
  backtrack(p[u]) // go back to predecessor of u
  Output u // recursion will reverse the order

// in main method
// recursive version (normal path)
Output "Path:"
backtrack(t); // start from end of path (vertex t)
// try it on this array p, t = 4
// p = {-1, 0, 1, 2, 3, -1, -1, -1}
```

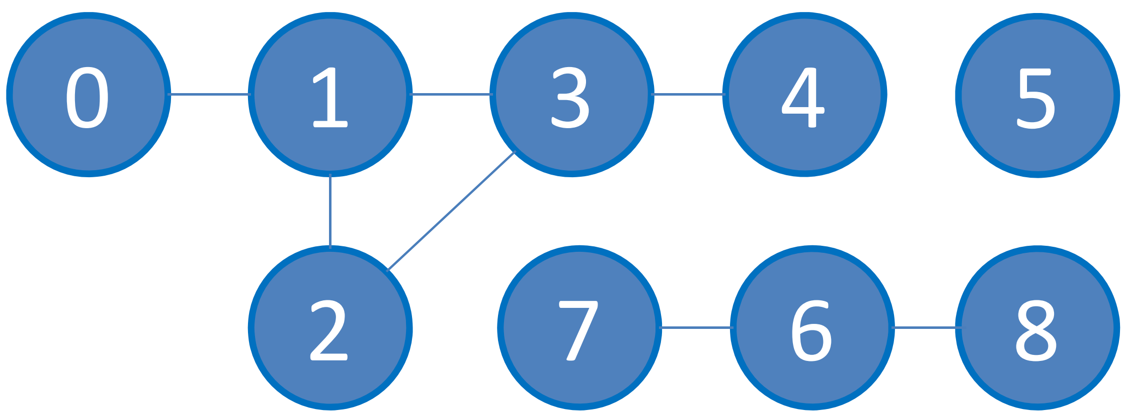
Hm… I prefer not to use recursion but I still want the correct path (from source to target), can I do that?

1. No, I have no choice but to use recursion to get the correct path
2. Possible, use this technique ________________


0 of 54

# Quick Challenge

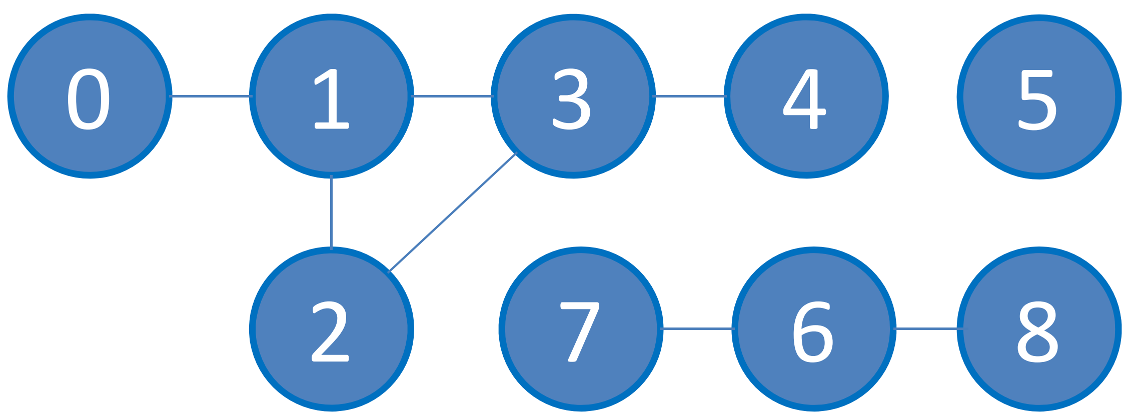
- Run BFS and then DFS from various source in the graph below



# What can we do with BFS/DFS? (1)

- Several stuffs, let's see ***some of them***:
  - Reachability test
    - Test whether vertex v is reachable from vertex u?
    - Start BFS/DFS from **s = u**
    - If **visited[v] = 1** after BFS/DFS terminates, then **v** is *reachable* from **u**; otherwise, **v** is *not reachable* from **u**

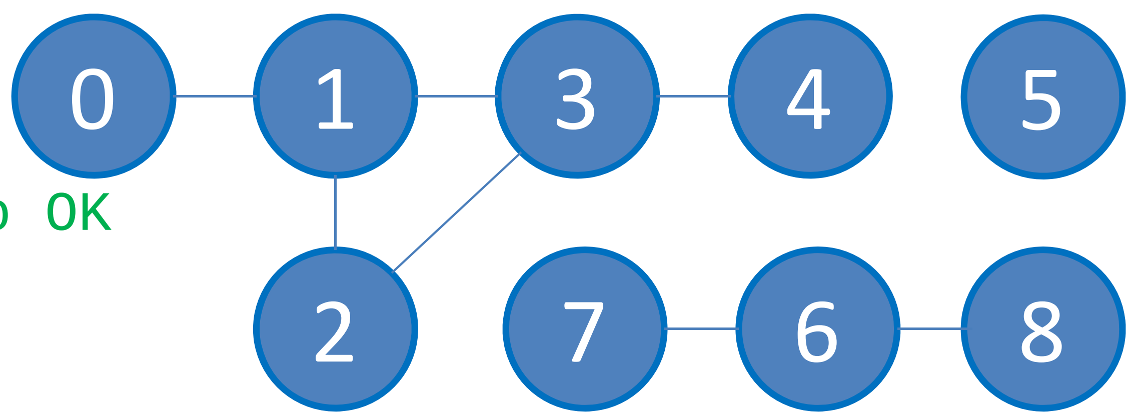
```
BFS(u) // DFS(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



# What can we do with BFS/DFS? (2)

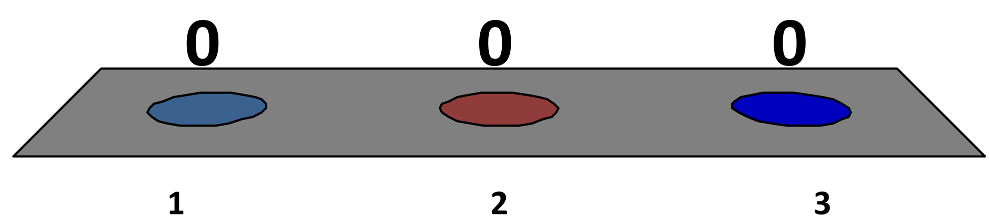
- Identifying component(s)
  - Component is sub graph in which any 2 vertices are connected to each other by paths, and is connected to no additional vertices
  - Identify/label/count components in graph G
  - Solution:

```
CC ← 0
for all v in V
  visited[v] ← 0
for all v in V // O(V)?
  if visited[v] == 0
    DFSrec(v)
    // O(V + E)?
    // PS: BFS is also OK
```



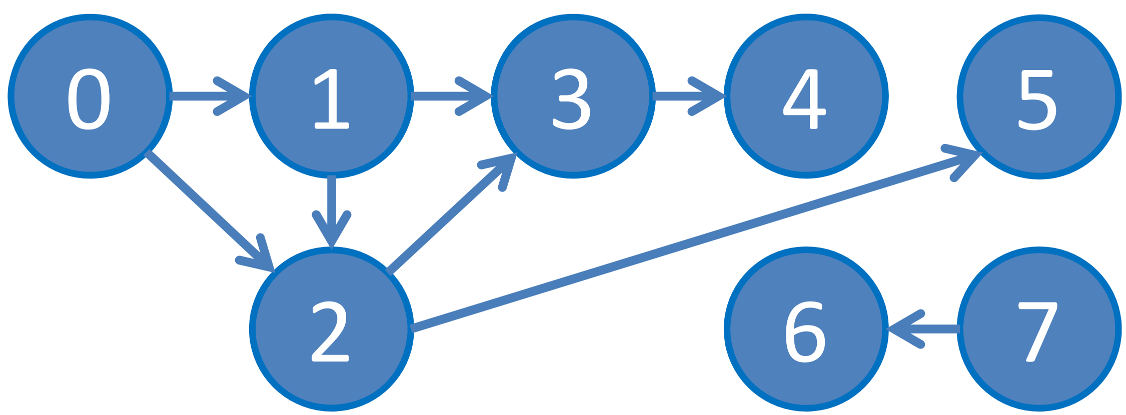
# What is the time complexity for "counting connected component"?

1. Hm… you can call O(V+E) DFS/BFS up to V times… I think it is O(V*(V + E)) = O(V^2 + VE)
2. I think it is O(V + E)...
3. Maybe some other time complexity, it is O(_______)



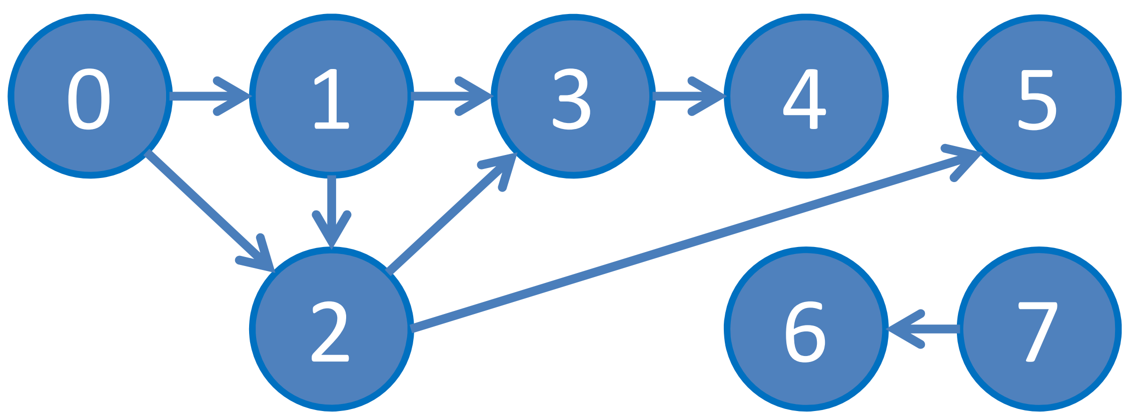
# What can we do with BFS/DFS? (3)

- Topological Sort
  - Topological sort of a DAG is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges
  - Every DAG has one *or more* topological sorts
  - One of the main purpose of finding topological sort: for Dynamic Programming (DP) on DAG (will be discussed few weeks later…)



# What can we do with BFS/DFS? (4)

- Topological Sort
  - If the graph is a DAG, then simply running **DFS** on it (and at the same time record the vertices in "post-order" manner) will give us one valid topological order
  - See pseudo code in the next slide



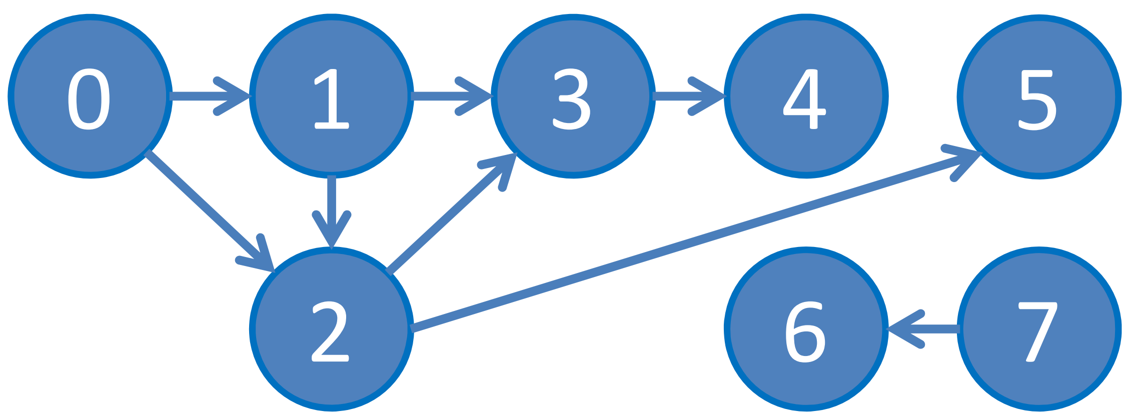
# DFS for TopoSort – Pseudo Code

```
topoVisit(u)
  for all v adjacent to u
    if visited[v] == 0
      topoVisit(v)
  append u to the back of toposort // "post-order"

// in the main method
for all v in V
  visited[v] ← 0
clear toposort
for all s in V
  if visited[s] == 0
    topoVisit(s)
reverse toposort and Output it
```

**toposort** is a kind of List (Vector)

# What can we do with BFS/DFS? (5)

- Topological Sort
  - Suppose we have visited all neighbors of 0 recursively with DFS
  - toposort list = [list of vertices reachable from 0] - vertex 0
    - Suppose we have visited all neighbors of 1 recursively with DFS
    - toposort list = [[list of vertices reachable from 1] - vertex 1] - vertex 0
    - and so on…
  - We will eventually have = [4, 3, 5, 2, 1, 0, 6, 7]
  - Reversing it, we will have = [7, 6, 0, 1, 2, 5, 3, 4]



# What is the given graph is not a DAG?

1. There will be no topological order and modified DFS (topoVisit) **will** be able to tell
2. There will be no topological order and modified DFS (topoVisit) **will NOT** be able to tell

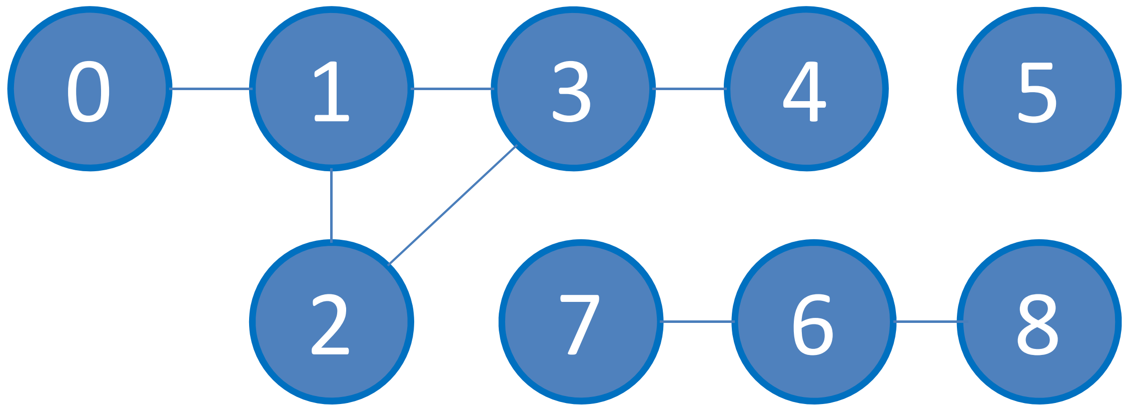

0 of 54

# Trade-Off

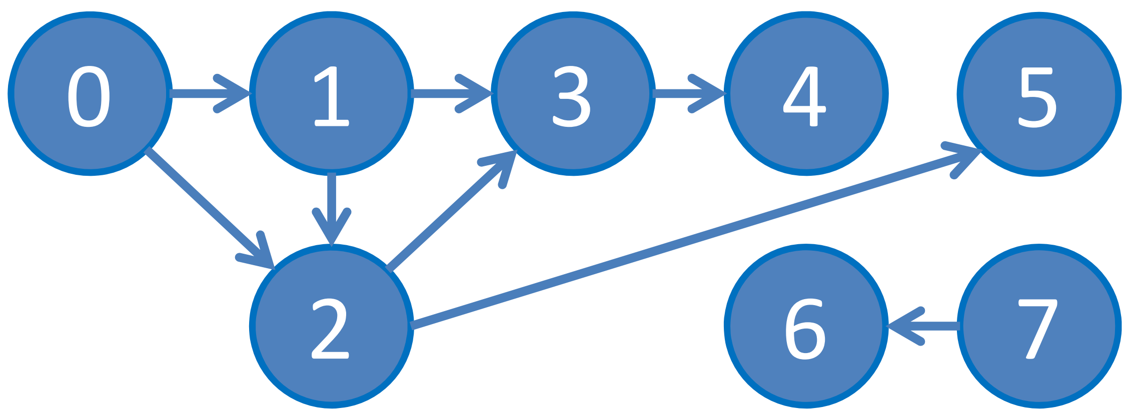
- O(V + E) DFS
  - Pro:
    - Slightly easier? to code (this one depends)
    - Use less memory
  - Cons:
    - Cannot solve SSSP on unweighted graphs (this will be discussed soon and will be “right before” PS7 due ☺)
- O(V + E) BFS
  - Pro:
    - Can solve SSSP on unweighted graphs (this will be discussed soon and will be “right before” PS7 due ☺)
  - Cons:
    - Slightly longer? to code (this one depends)
    - Use more memory (especially for the queue)

# Java Implementation

- Let's see Java implementation of BFS/DFS algorithms and their applications as discussed in this lecture
  - See the updated GraphDemo2.java
  - undirected.txt




  - dag.txt



# Summary

- Graph Traversal Algorithms: Start + Movement
- Breadth-First Search: uses queue, breadth-first
- Depth-First Search: uses stack/recursion, depth-first
- Both BFS/DFS uses “flag” technique to avoid cycling
- Both BFS/DFS generates BFS/DFS “Spanning Tree”
  - Path reconstruction algorithm has been shown
- Some applications: Reachability, CC, Toposort