CS2010 – Data Structures and Algorithms II

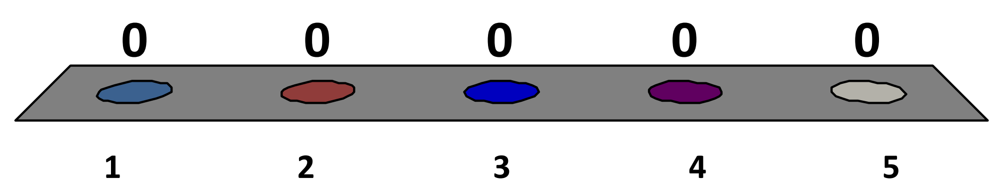
Lecture 05 – Graph Basics (Revisited*)

stevenhalim@gmail.com

NUS National University of Singapore
School of Computing

PS2 (Already open for 5 days), I…

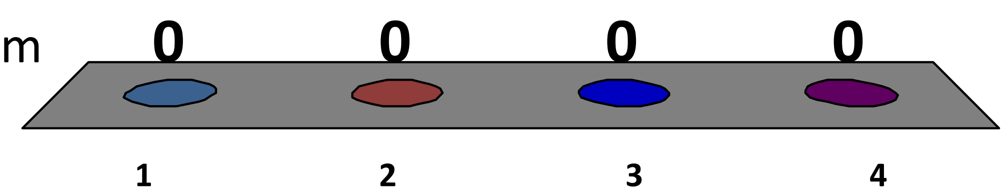
1. Have solved PS2 Subtask 3
   + the R-option requirement
2. Have tried up to Subtask 3
3. Have tried up to Subtask 2
4. Have tried up to Subtask 1
5. Have not read it :O…



0 of 120

User Behavior in Game System

1. I will stop at 3+3 ~ 6 hours (recommended self-study + assignment time for this 4MC module), and grab whatever points (lower subtasks) that I manage to solve at that point
2. No matter what I must keep improving my solution until deadline to get as many points as possible, even if the harder Subtasks look incredibly difficult...
3. Depends on my workload that week but mostly option 2 above
4. I will wait for my friend(s) to complete it first and then I will ask him/her/them

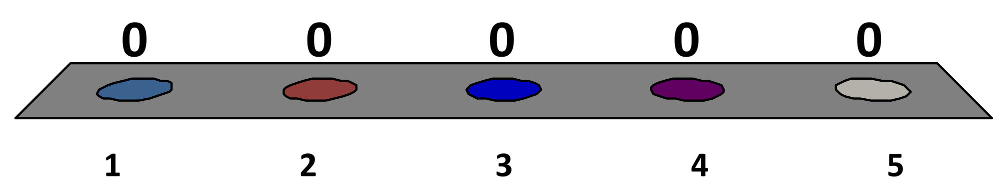


0 of 120

Average Time Spent for (PS1+PS2)/2

1. < 3 hours (:O)
2. < 6 hours
   (recommended time)
3. < 10 hours
4. < 15 hours
5. I don't know… probably
   more than 15 hours?

0 of 120



Game System versus Sit-in Lab

1. I prefer this **game system** where I can code at my own pace and *get points that I deserve*
2. I prefer **sit-in lab** and get each assignment done in < 2 hours, regardless whether I crack under time pressure or not...

0 of 120

Quiz 1 (Details to be updated…)

- Wait until Week05 to know the details

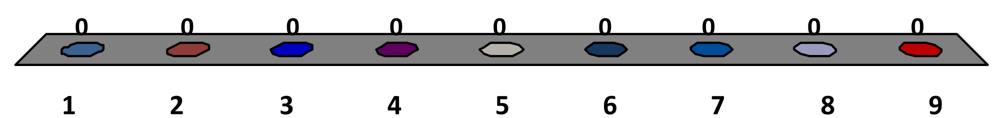
Outline

- What are you going to learn in this lecture?
  - Motivation on why you should learn graph
    - ***Very quick review*** on graph terminologies (from CS1231)
  - Two graph data structures (CP2.5 Section 2.4.1)
    - Adjacency Matrix
    - Adjacency List
    - Some applications
  - Two algorithms to traverse a graph (CP2.5 Section 4.2.1/4.2.2)
    - Depth First Search (DFS)
    - Breadth First Search (BFS)
    - Some applications
  - PS: We will likely only discuss the latter part of Lecture 05 during the early part of Week06

Select graph terminologies that you **already know**… (can select up to 9/clicker)

1. Adjacency Matrix/List
2. Edge List
3. Traversal: DFS/BFS
4. Topological Sort
5. MST/Prim’s
6. MST/Kruskal’s
7. SSSP/Bellman Ford’s
8. SSSP/Dijkstra’s
9. APSP/Floyd Warshall’s

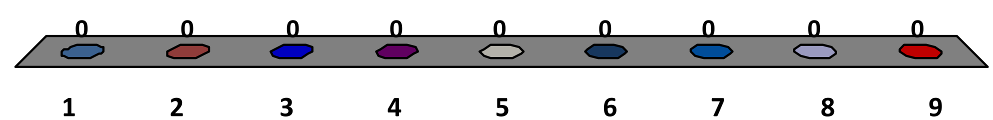
0 of 120



Select DS/algorithms that you have already **implement**… (can select up to 9/clicker)

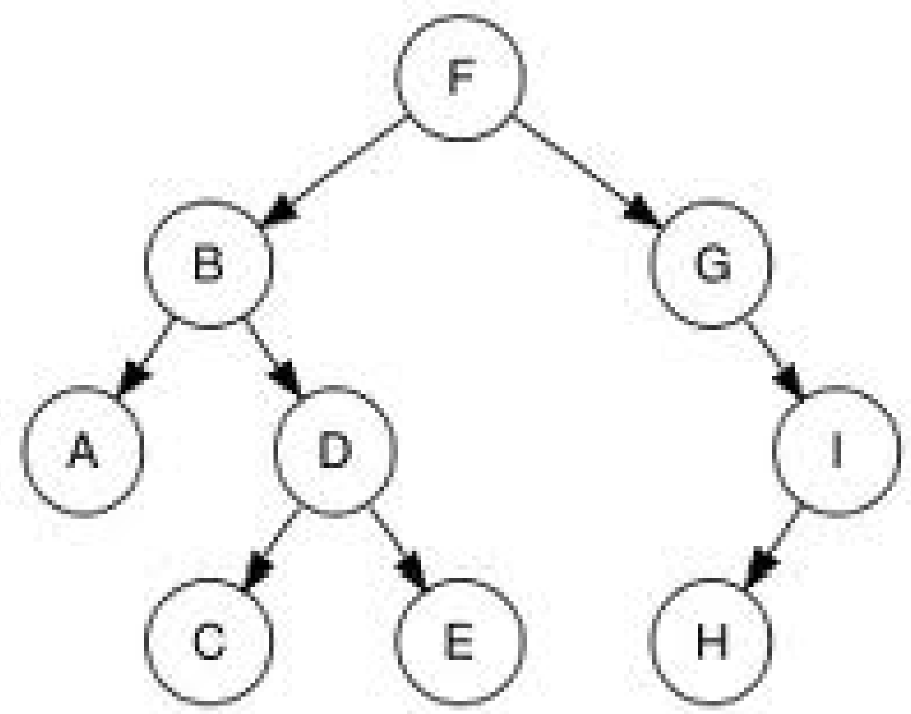
1. Adjacency Matrix/List
2. Edge List
3. Traversal: DFS/BFS
4. Topological Sort
5. MST/Prim's
6. MST/Kruskal's
7. SSSP/Bellman Ford's
8. SSSP/Dijkstra's
9. APSP/Floyd Warshall's

0 of 120



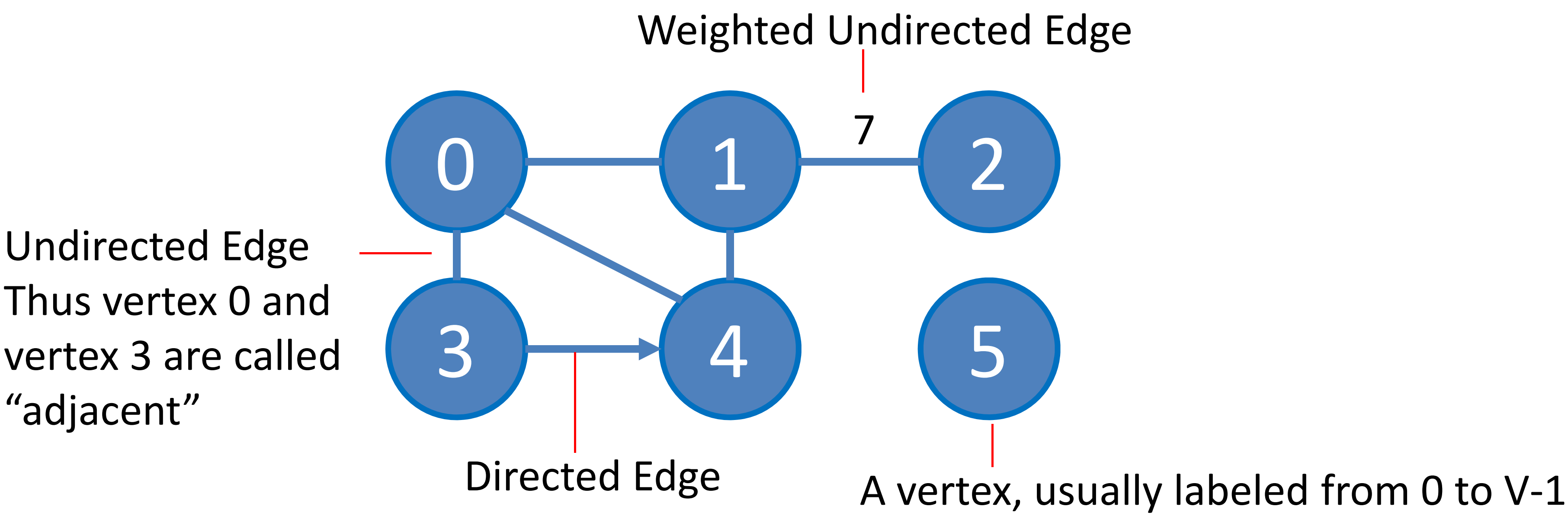
Graph Terminologies (1)

- Extension from what you already know: *(Binary) Tree*
  - Vertex/Node
  - Edge
  - Direction (of Edge)
  - Weight
- But in general graph, there is no notion of:
  - Root
  - Parent/Child
  - Ancestor/Descendant



Graph is…

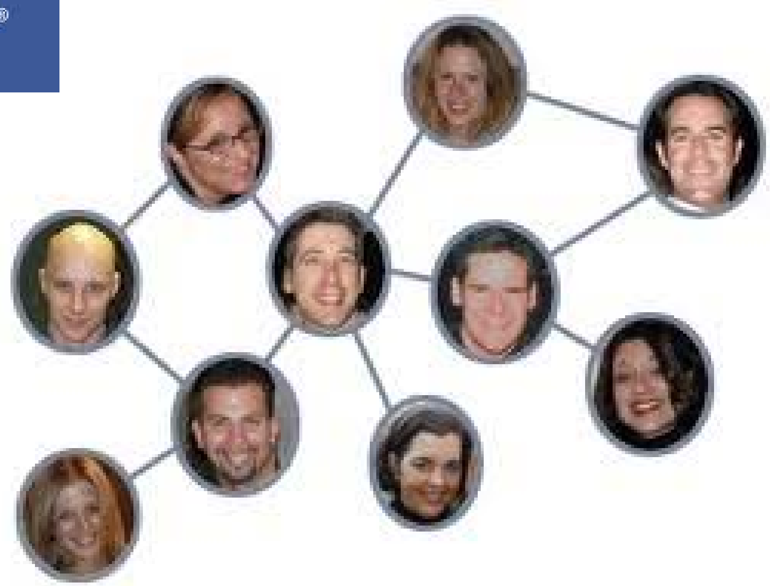
- (Simple) graph is a set of vertices where some $[0 \,..\, {}_N C_2]$ pairs of the vertices are connected by edges
  - We will ignore "multi graph" where there can be more than one edge between a pair of vertices



Social Network

facebook

twitter



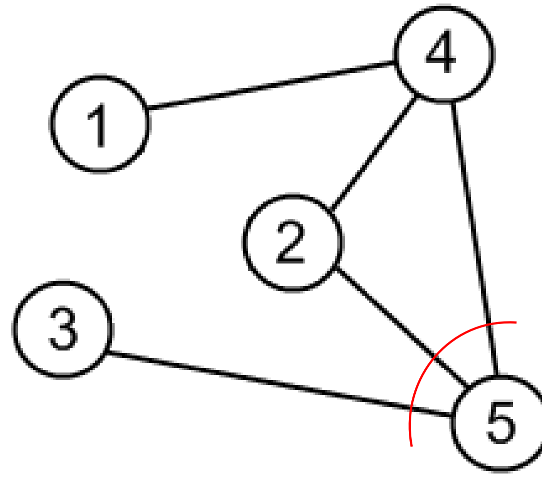
LinkedIn

friendster

This one is different now :O

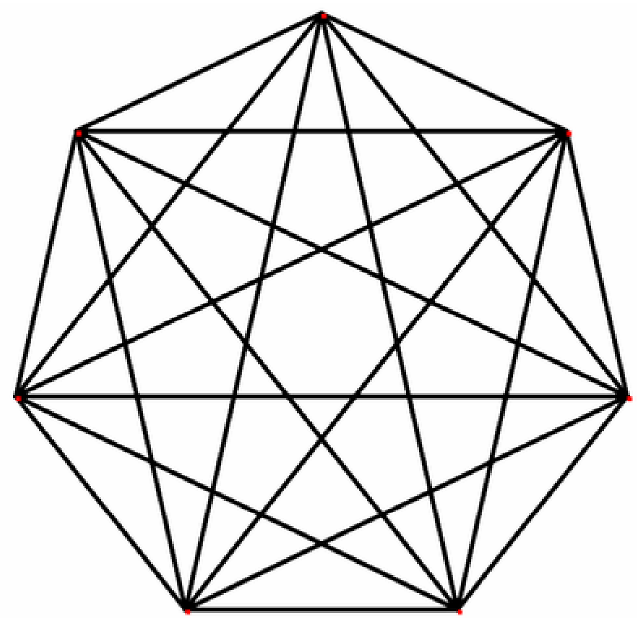
Graph Terminologies (2)

- More terminologies (simple graph):
  - Sparse/Dense
    - Sparse = not so many edges
    - Dense = many edges
    - No guideline for “how many”
  - Complete Graph
    - Simple graph with N vertices and ${}_{N}C_2$ edges
  - In/Out Degree of a vertex
    - Number of in/out edges from a vertex



Sparse Graph

In/out degree of vertex 5 = 3



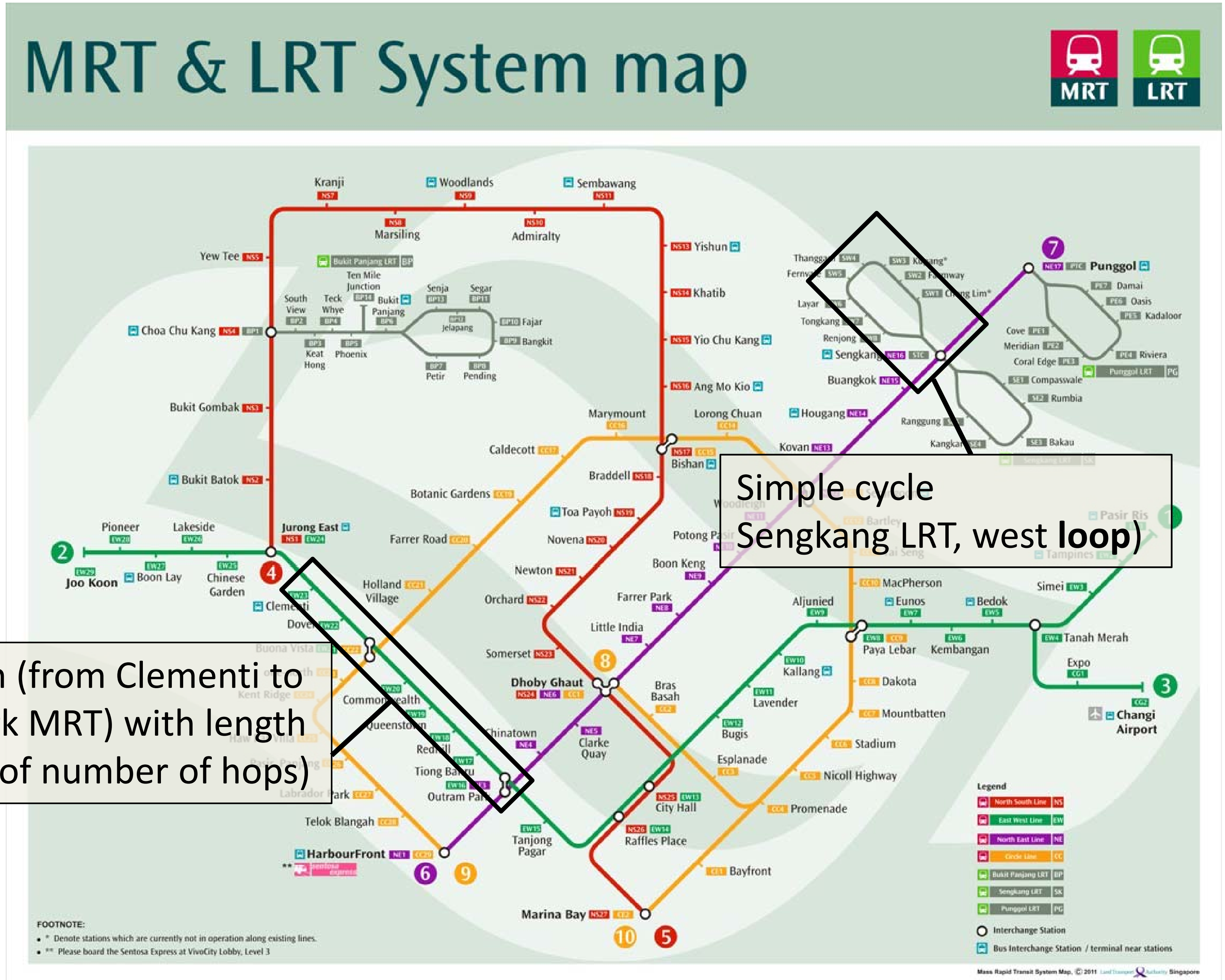
Dense Graph

Complete Graph
7 vertices,
${}_{7}C_2$ = 21 edges

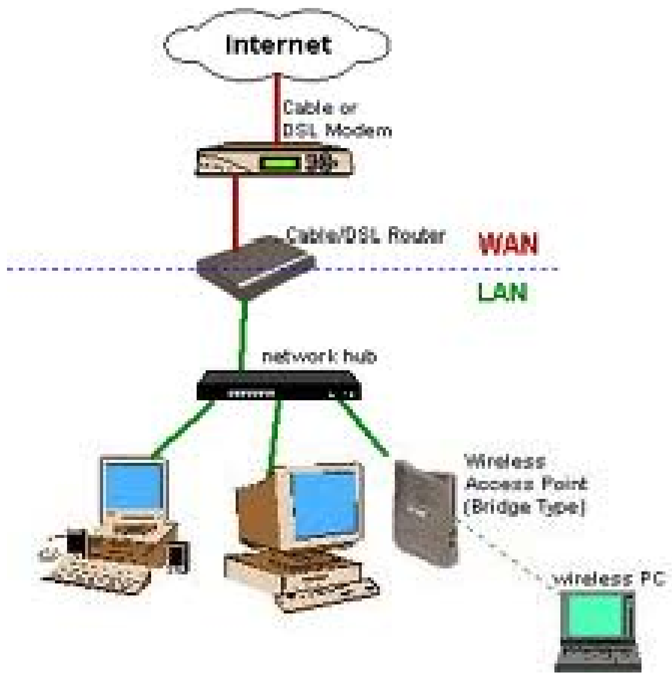
Graph Terminologies (3)

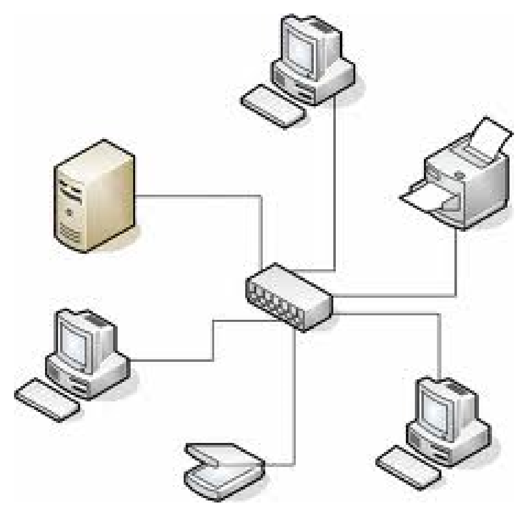
- Yet more terminologies (example in the next slide):
  - (Simple) Path
    - Sequence of vertices adjacent to each other
    - Simple = no repeated vertex
  - Path Length/Cost
    - In unweighted graph, usually number of edges in the path
    - In weighted graph, usually sum of edge weight in the path
  - (Simple) Cycle
    - Path that starts and ends with the same vertex
    - With no repeated vertex except start/end

Transportation Network



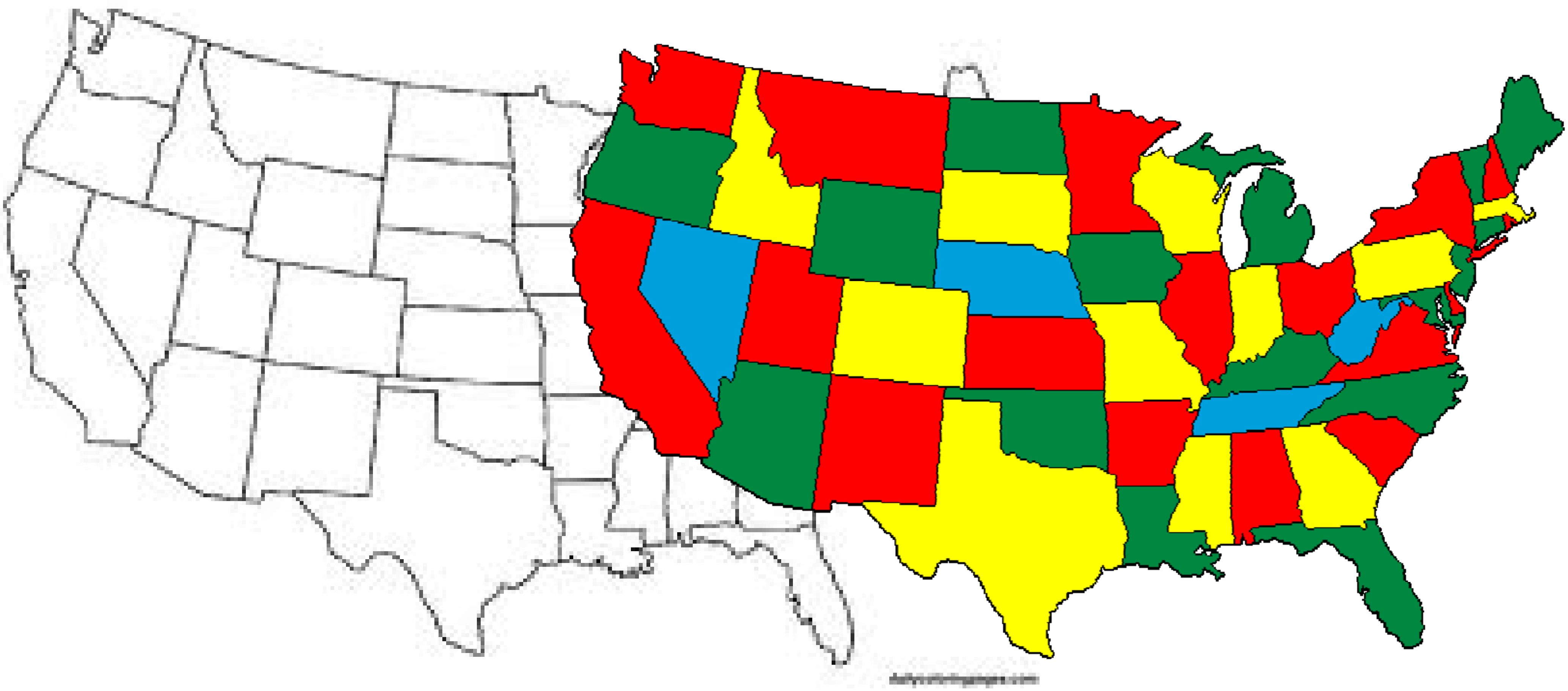
Internet / Computer Networks





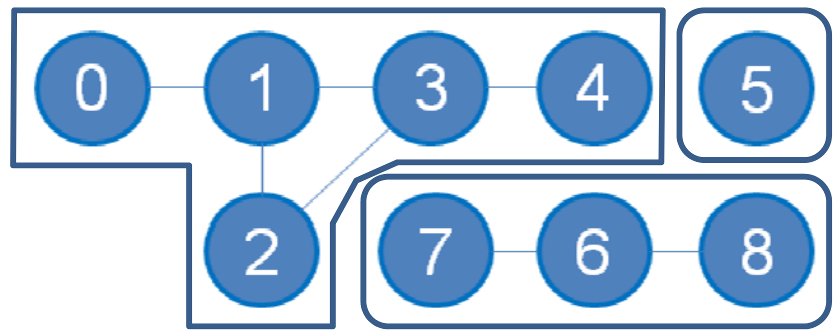
Communication Network

Optimization



Graph Terminologies (4)

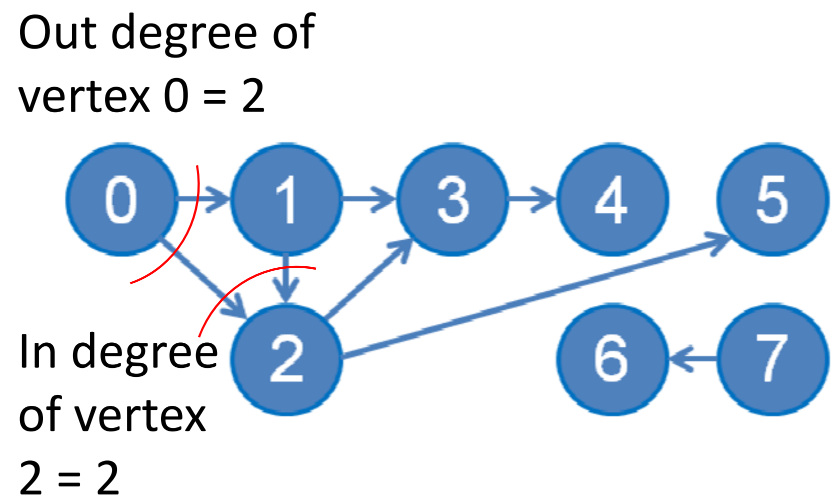
- Yet More Terminologies:
  - Component
    - A group of vertices in undirected graph that can visit each other via some path
  - Connected graph
    - Graph with only 1 component
  - Reachable/Unreachable Vertex
    - See example
  - Sub Graph
    - Subset of vertices (and their edges) of the original graph

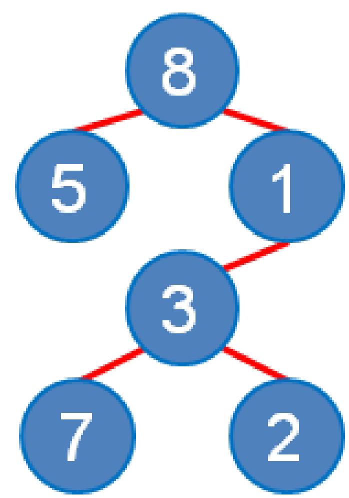


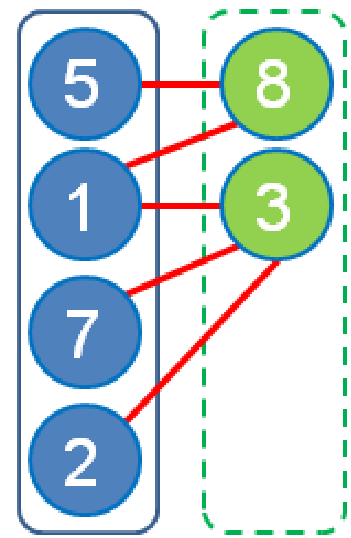
- There are 3 components in this graph
- Disconnected graph
(since it has > 1 component)
- Vertices 1-2-3-4 are reachable from vertex 0
- Vertices 5, 6-7-8 are unreachable from vertex 0
- {7-6-8} is a sub graph of this graph

Graph Terminologies (5)

- Yet More Terminologies:
  - Directed Acyclic Graph (DAG)
    - Directed graph that has no cycle
  - Tree (bottom left)
    - Connected graph, E = V - 1, one unique path between any pair of vertices
  - Bipartite Graph (bottom right)
    - If we can partition the vertices into two sets so that there is no edge between members of the same set







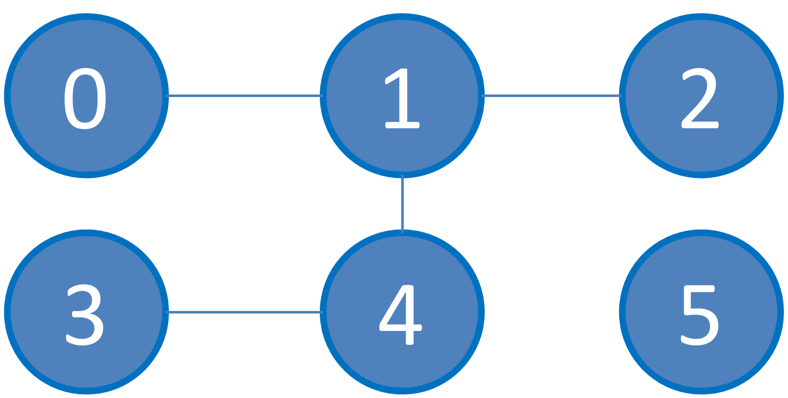
Next, we will discuss two Graph DS
(One more will be discussed during MST lecture)
Followed with some basic applications
Reference: CP2.5 Section 2.4.1

GRAPH DATA STRUCTURES

Storing Graph Information

Can we just store vertex information *only*?

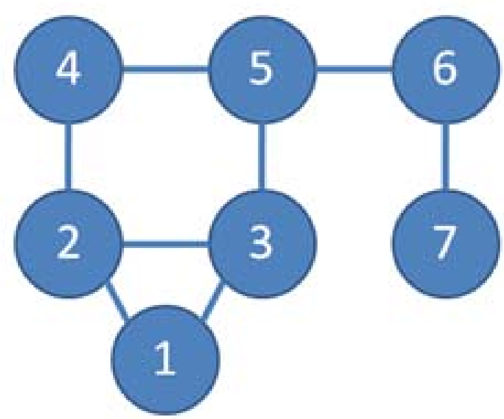
1. YES, vertex information is enough to rebuild the graph
2. NO, we also need to store edge/connectivity information too!



0 of 120

Adjacency Matrix

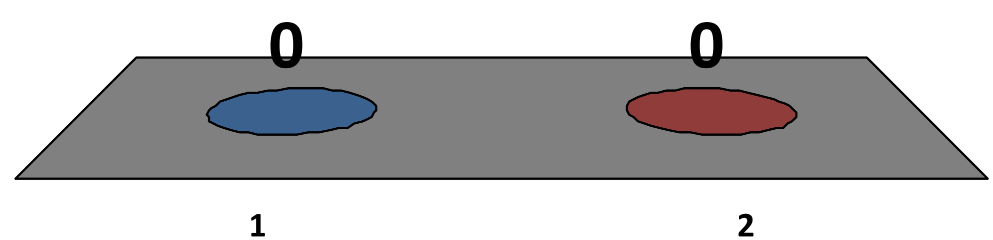
- Format: a 2D array **AdjMatrix** (see an example below)
- Cell **AdjMatrix**[i][j] contains value 1 if there exist an edge i->j in G, otherwise **AdjMatrix**[i][j] contains 0
  - For weighted graph, **AdjMatrix**[i][j] contains the weight of edge i->j, not just binary values {1, 0}.
- **Space Complexity**: $O(V^2)$
  - V is |V| = number of vertices in G



	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

To think about: If I have a graph with V = 100000 vertices, can I use **Adjacency Matrix**?

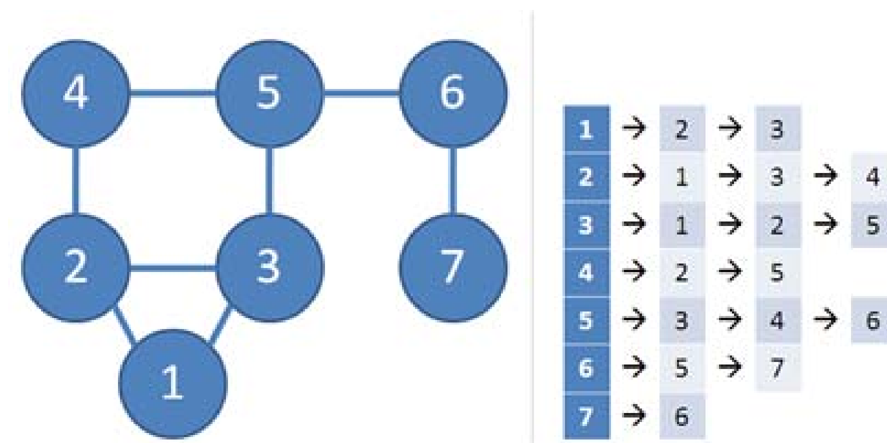
1. Yes, what is the problem?
2. No, because ________________



0 of 120

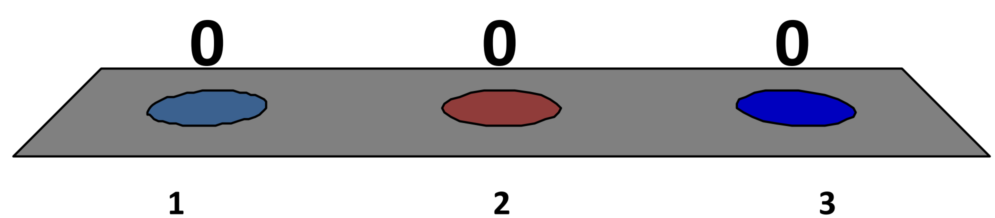
Adjacency List

- Format: array **AdjList** of V lists, one for each vertex in V (see an example below)
- For each vertex i, **AdjList[i]** stores list of i's neighbors
  - For weighted graph, stores **pairs (neighbor, weight)**
    - Note that for unweighted graph, we can also use the same strategy as the weighted version: (neighbor, weight = 0 or 1)
- **Space Complexity**: O(V + E)
  - E is |E| = number of edges in G, $E = O(V^2)$
  - V + E ~= max(V, E)



To think about: If I have a graph with V = 100000 vertices, can I use **Adjacency List**?

1. Yes, or course
2. No, because ________________
3. Depends, because ________________



0 of 120

Java Implementation (1)

- Adjacency Matrix
  - Simple built-in 2D array

```
int i, V = NUM_V; // NUM_V has been set before
int[][] AdjMatrix = new int[V][V];
```

- Adjacency List
  - Use Java Collections framework

```
Vector < Vector < IntegerPair > > AdjList =
  new Vector < Vector < IntegerPair > >();
// IntegerPair is a simple integer pair class
// to store pair info, see the next slide
```

- PS: This is *my* implementation, there are other ways

Java Implementation (2)

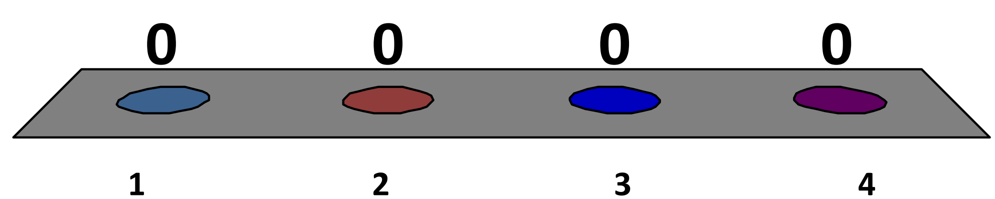
```
class IntegerPair implements Comparable {
  Integer _first, _second;
  public IntegerPair(Integer f, Integer s) {
    _first = f;
    _second = s;
  }
  public int compareTo(Object o) {
    if (!this.first().equals(((IntegerPair)o).first()))
      return this.first() - ((IntegerPair)o).first();
    else
      return this.second() - ((IntegerPair)o).second();
  }
  Integer first() { return _first; }
  Integer second() { return _second; }
}
```

Java Implementation (3)

- We will use AdjList for most graph problems in CS2010
- Vector < Vector < IntegerPair > > AdjList;
  - Why do we use IntegerPair?
    - We need to store pair of information for each edge: (neighbor number, weight)
  - Why do we use Vector of IntegerPair?
    - For Vector's **auto-resize feature** ☺: If you have **k** neighbors of a vertex, just add **k** times to an initially empty Vector of IntegerPair of this vertex
    - You can replace this with Java List or ArrayList if you want to…
  - Why do we use Vector of Vector of IntegerPair?
    - For Vector's **indexing feature** ☺: if we want to enumerate neighbors of vertex u, use **AdjList.get(u)** to access the correct Vector of IntegerPair

Trade Off (1), after knowing Adjacency Matrix and List, which one should be our **default choice**?

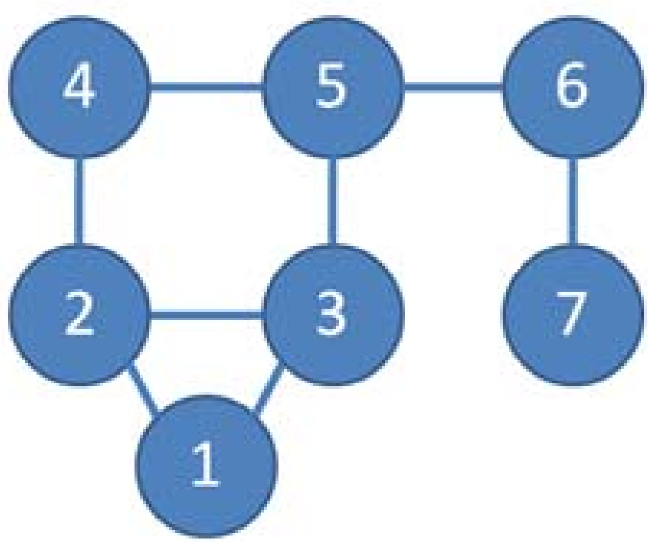
1. Adjacency Matrix
2. Adjacency List
3. Use BOTH at the same time
4. Depends, because ____________



0 of 120

So, what can we do so far? (1)

- With just graph DS, not much that we can do…
- But here are some:
  - Counting V (the number of vertices)
    - Very trivial for both AdjMatrix and AdjList: **V = number of rows!**
    - Sometimes this number is stored in separate variable so that we do not have to re-compute every time



A

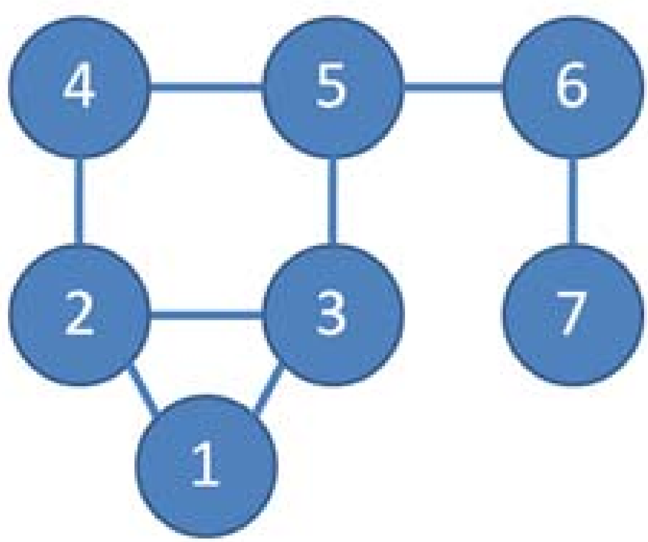
	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1	→ 2	→ 3	
2	→ 1	→ 3	→ 4
3	→ 1	→ 2	→ 5
4	→ 2	→ 5	
5	→ 3	→ 4	→ 6
6	→ 5	→ 7	
7	→ 6		

So, what can we do so far? (2)

- Enumerating neighbors of a vertex v
  - O(V) for AdjMatrix: **scan AdjMatrix[v][j], for all j in [0 .. V-1]**
  - O(k) for AdjList, **scan AdjList[v]**
    - Where k is the number of neighbors of vertex v
    - This is an important difference between Adjacency Matrix and Adjacency List and it affects the performance of many graph algorithms. Remember this!



A

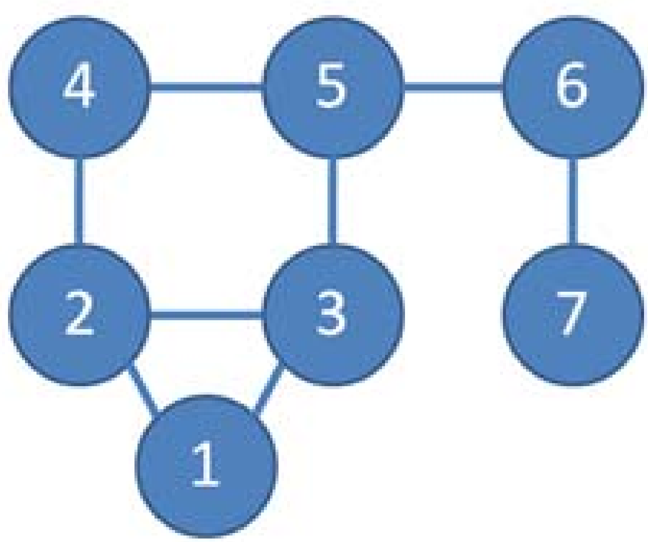
	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1 → 2 → 3
2 → 1 → 3 → 4
3 → 1 → 2 → 5
4 → 2 → 5
5 → 3 → 4 → 6
6 → 5 → 7
7 → 6

So, what can we do so far? (3)

– Counting E (the number of edges)
  - O(V^2) for AdjMatrix: **count non zero entries in AdjMatrix**
  - O(V + E) for AdjList: **sum the length of all V lists**
  - Sometimes this number is stored in separate variable so that we do not have to re-compute every time



A

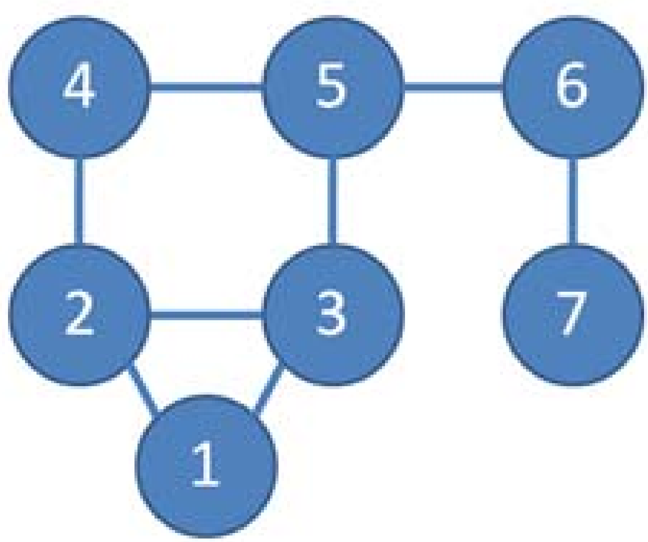
	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1	→ 2	→ 3	
2	→ 1	→ 3	→ 4
3	→ 1	→ 2	→ 5
4	→ 2	→ 5	
5	→ 3	→ 4	→ 6
6	→ 5	→ 7	
7	→ 6		

So, what can we do so far? (4)

- Checking the existence of edge(u, v)
  - O(1) for AdjMatrix: **see if AdjMatrix[u][v] is non zero**
  - O(k) for AdjList: **see if AdjList[u] contains v**
- There are few others, but let's reserve them for our PSes or even for test questions ☺



A

	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1	→ 2	→ 3	
2	→ 1	→ 3	→ 4
3	→ 1	→ 2	→ 5
4	→ 2	→ 5	
5	→ 3	→ 4	→ 6
6	→ 5	→ 7	
7	→ 6		

Trade-Off (2)

- Adjacency Matrix:
  - Pro:
    - Existence of edge i-j can be found in O(1)
    - Good for dense graph/ Floyd Warshall's (Lecture 11)*
  - Cons:
    - O(V) to enumerate neighbors of a vertex
    - $O(V^2)$ space
- Adjacency List:
  - Pro:
    - O(k) to enumerate k neighbors of a vertex
    - Good for sparse graph/ Dijkstra's (Lecture 08)*/ DFS/BFS, O(V + E) space
  - Cons:
    - O(k) to check the existence of edge i-j
    - A little bit overhead in maintaining the list (for sparse graph)

5 Minutes Break

- Meanwhile, you can play with graph DS visualization:

  www.comp.nus.edu.sg/~stevenha/visualization/representation.html

Depth First Search (DFS)

Breadth First Search (BFS)

Some Basic Applications

Reference: CP2.5 Section 4.2.1 + 4.2.2

GRAPH TRAVERSAL ALGORITHMS

Review – **Binary Tree** Traversal

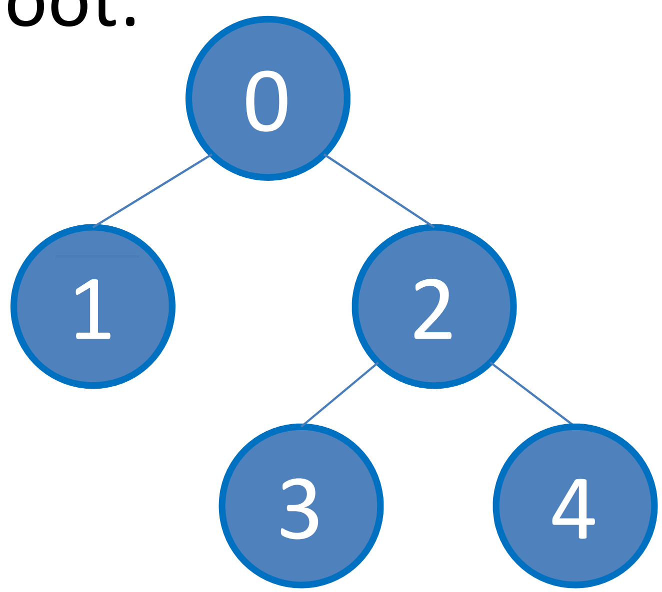
- In a binary tree, there are three standard traversal:
  - Preorder
  - **Inorder**
  - Postorder
  - (Note: “level order” is just BFS which we will see next)

```
pre(u)
  visit(u);
  pre(u->left);
  pre(u->right);
```

```
in(u)
 in(u->left);
 visit(u);
 in(u->right);
```

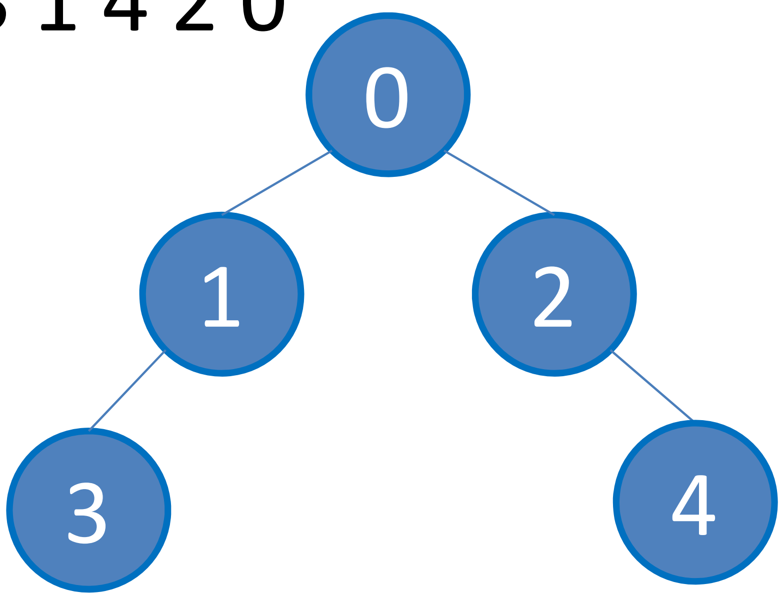
```
post(u)
 post(u->left);
 post(u->right);
 visit(u);
```

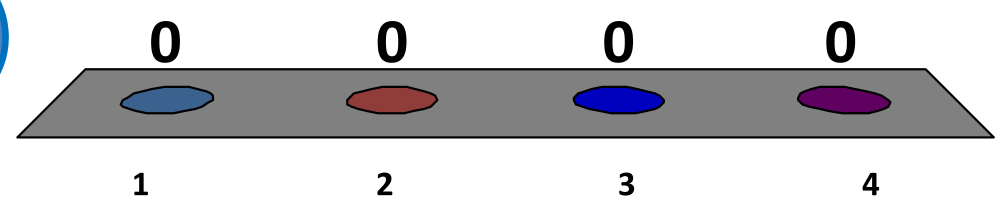
- We start binary tree traversal from root:
  - pre(root)/in(root)/post(root)
    - pre = 0, 1, 2, 3, 4
    - in = 1, 0, 3, 2, 4
    - post = 1, 3, 4, 2, 0



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 120

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**

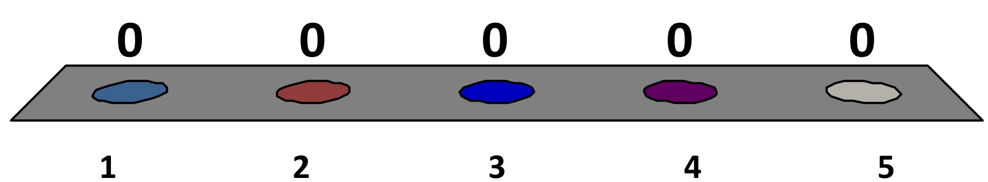
Traversing a Graph (2)

- 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 120

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
```

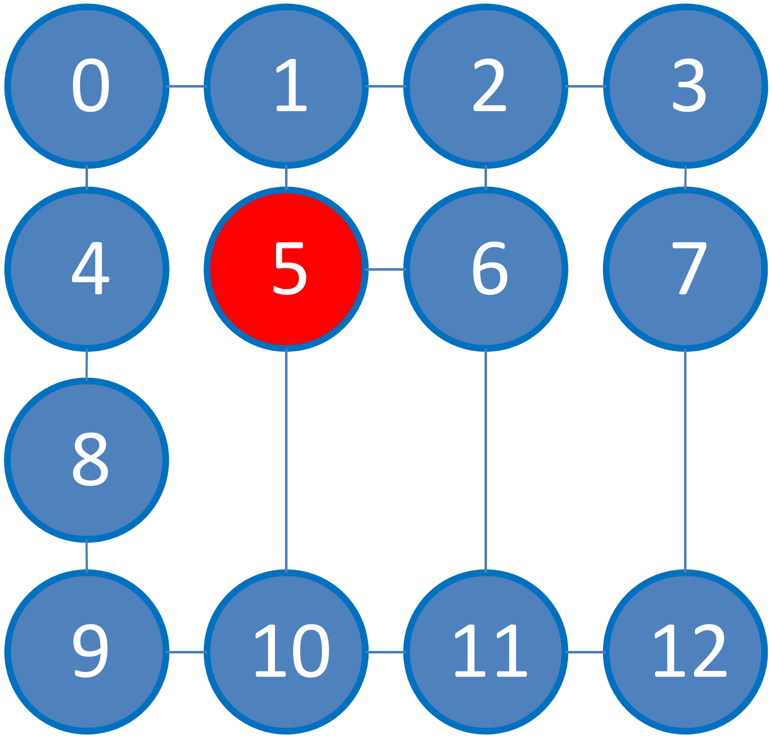
Initialization phase

```
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
```

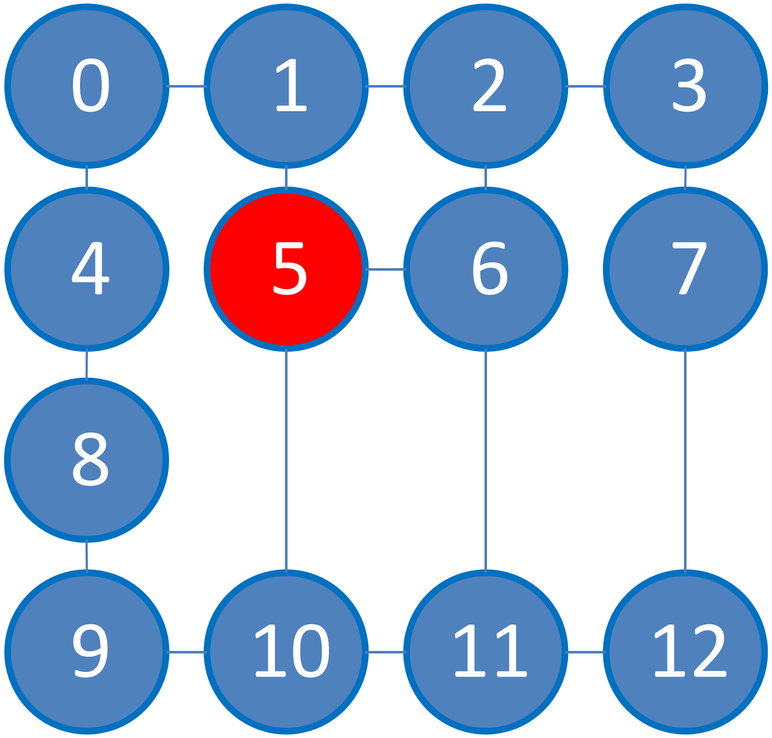
Example (1)



Q = {5}

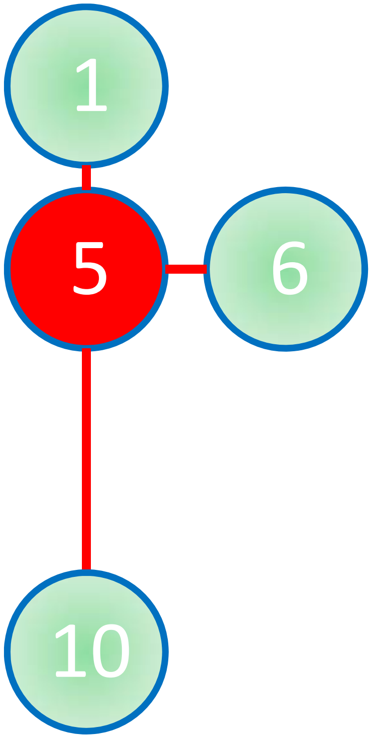
Neighbors are listed in increasing order

Example (2)

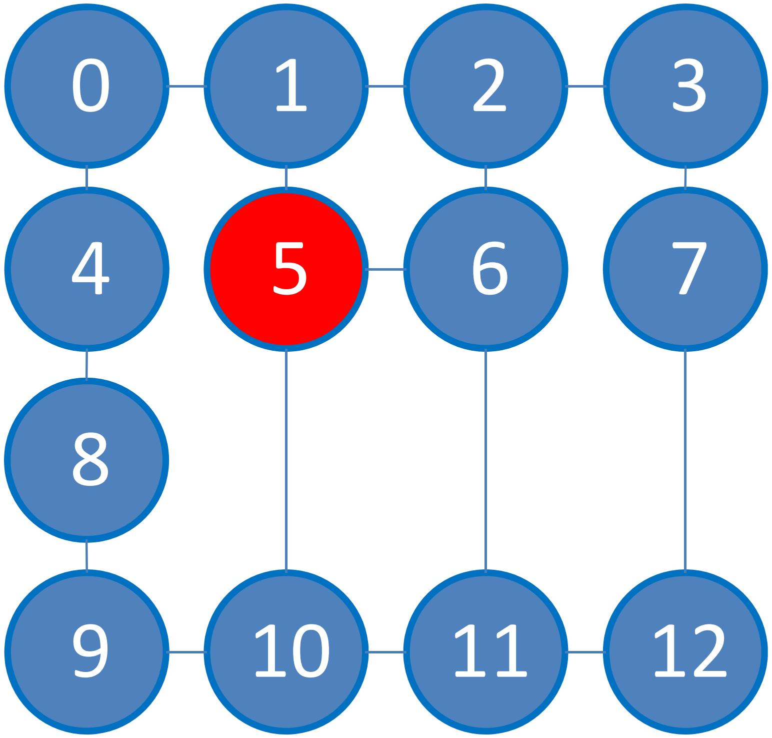


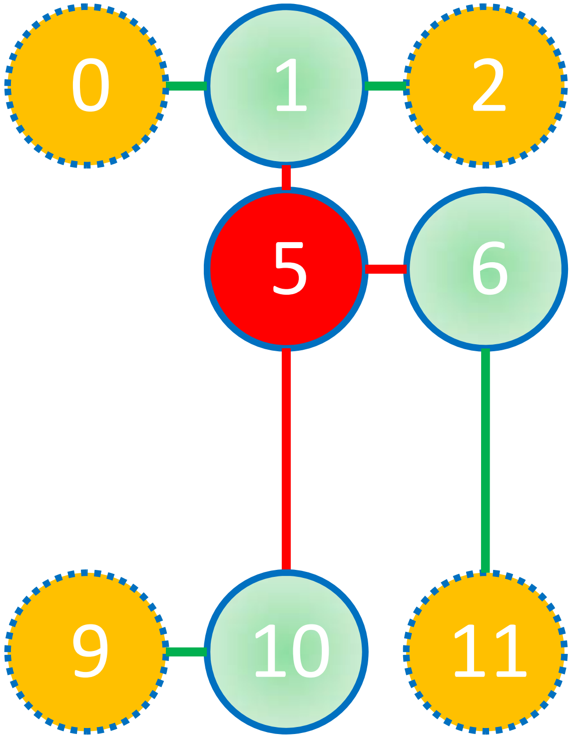
Q = {5}
Q = {1, 6, 10}

Neighbors are listed in increasing order



Example (3)

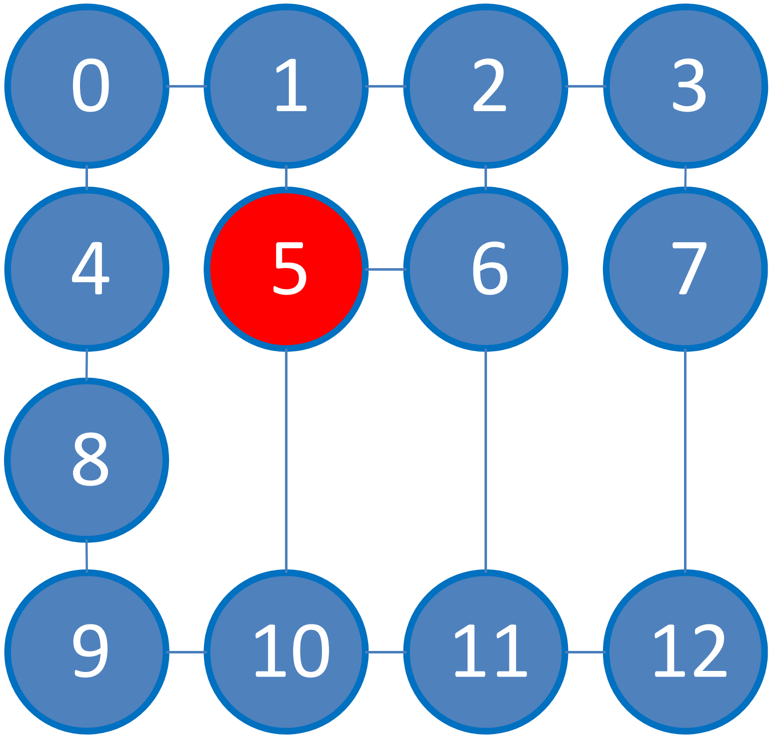


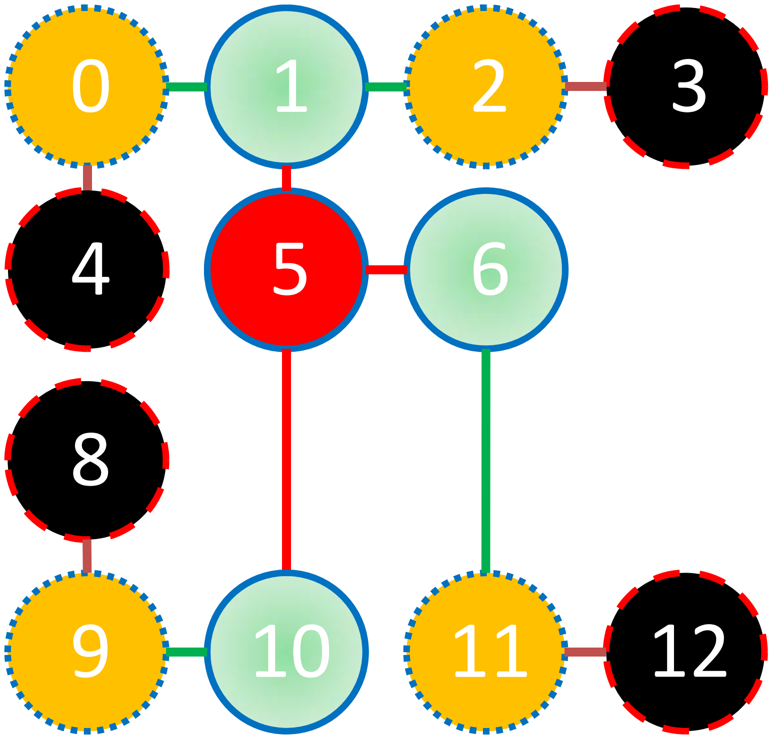


Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}

Neighbors are listed in increasing order

Example (4)

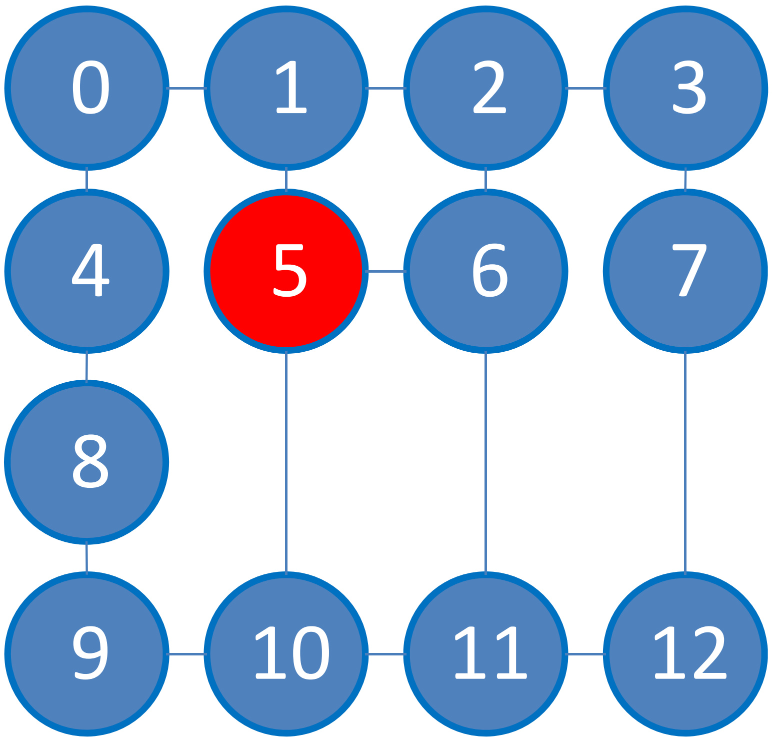


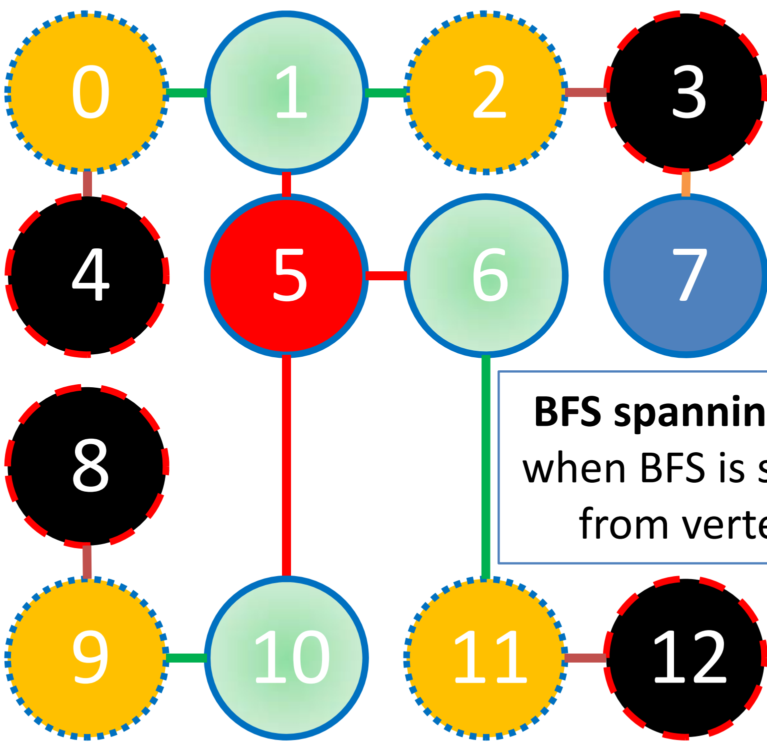


Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}
Q = {2, 11, 9, **4**}
Q = {11, 9, 4, **3**}
Q = {9, 4, 3, **12**}
Q = {4, 3, 12, **8**}

Neighbors are listed in increasing order

Example (5)



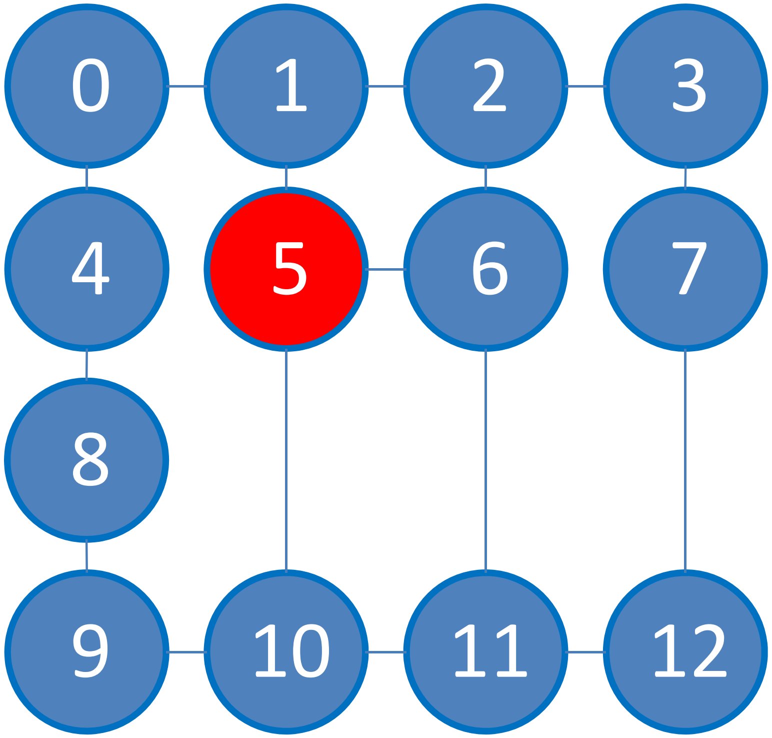


BFS spanning tree when BFS is started from vertex 5

Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}
Q = {2, 11, 9, **4**}
Q = {11, 9, 4, **3**}
Q = {9, 4, 3, **12**}
Q = {4, 3, 12, **8**}
Q = {3, 12, 8}
Q = {12, 8, **7**}
Q = {8, 7}
Q = {7}
Q = {}

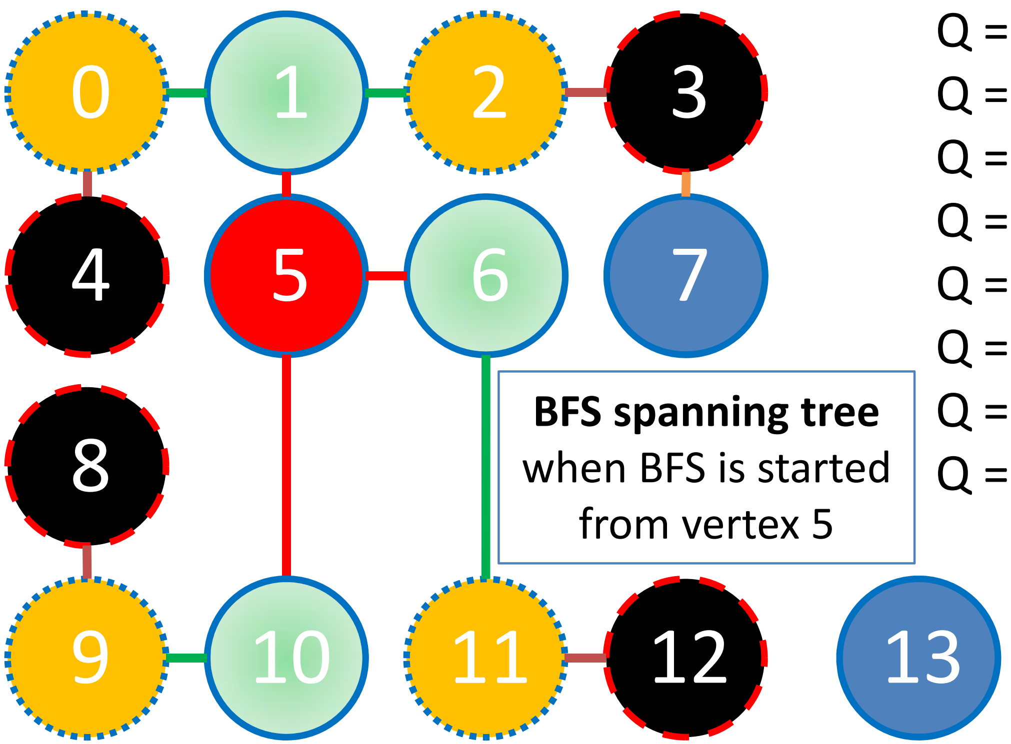
Neighbors are listed in increasing order

Example (6)



Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}
Q = {2, 11, 9, **4**}
Q = {11, 9, 4, **3**}
Q = {9, 4, 3, **12**}
Q = {4, 3, 12, **8**}
Q = {3, 12, 8}
Q = {12, 8, **7**}
Q = {8, 7}
Q = {7}
Q = {}

Neighbors are listed in increasing order



To think about:

What if we have another vertex "13" 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
```

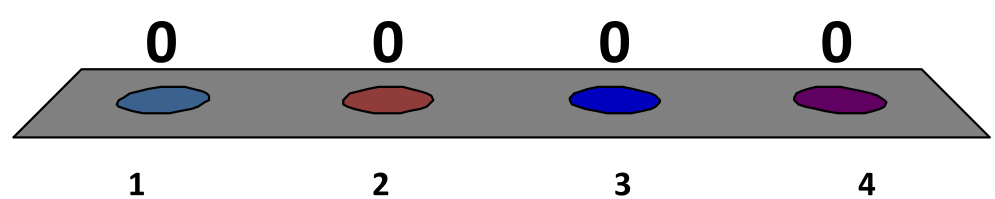
Time Complexity: $O(V + E)$

- 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

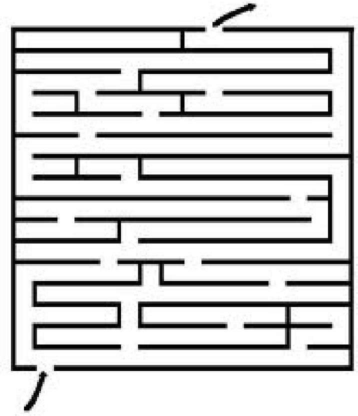
What is your level of understanding as of now?

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 120

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 (an 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

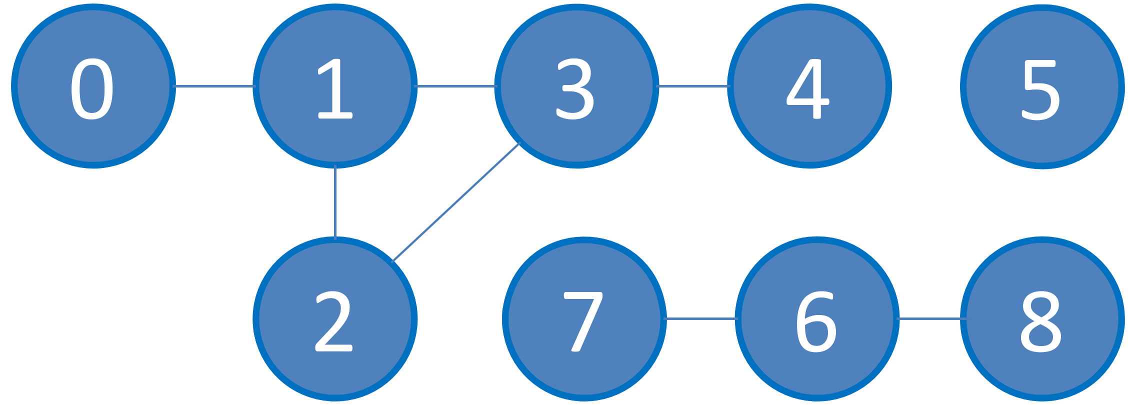
```
DFSrec(u)
  visited[u] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)
```

Recursive phase

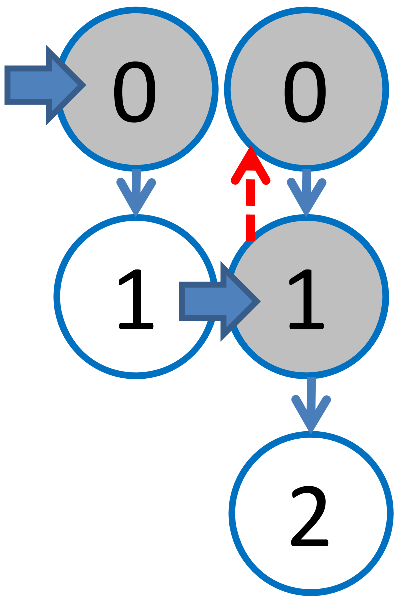
```
// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1
DFSrec(s) // start the
recursive call from s
```

Initialization phase, same as with BFS

Example (1)

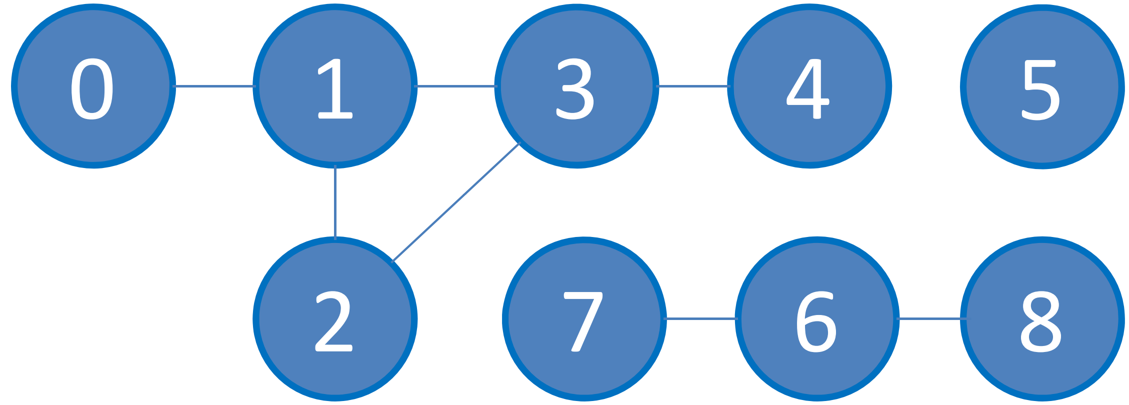


Assume that we start from source s = 0,
neighbors are listed in ascending order

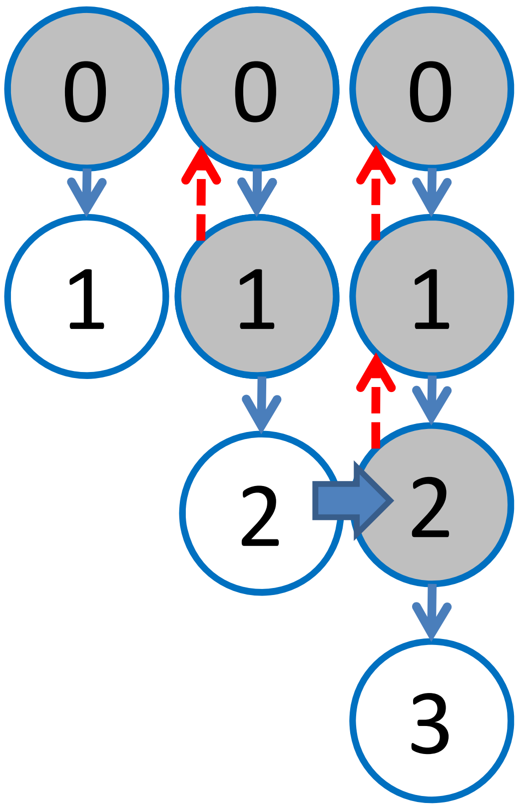


At vertex 1, we cannot go back to vertex 0 as it has been “flagged”;
but we can continue (more depth) to vertex 2 **or vertex 3;**
assume for this case we visit vertex 2 first (ascending order)

Example (2)

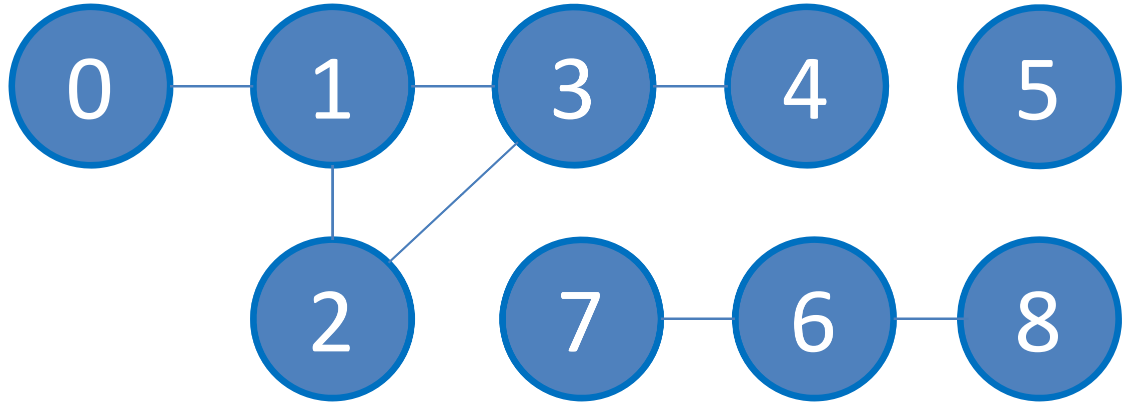


Assume that we start from source s = 0,
neighbors are listed in ascending order

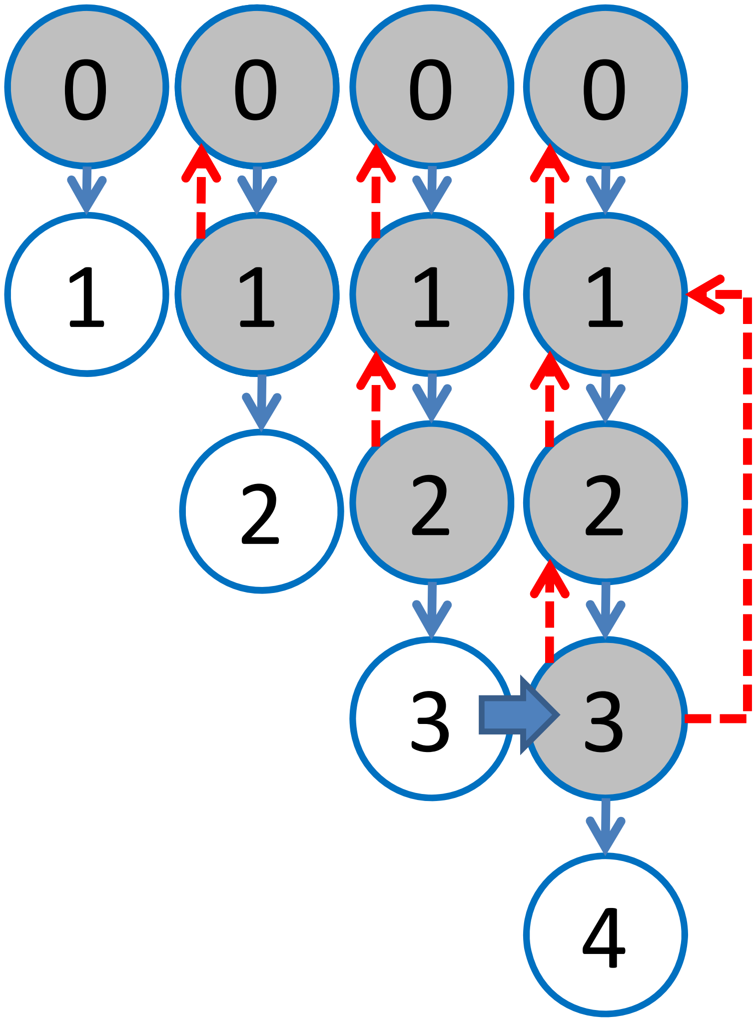


At vertex 2, we cannot go back to vertex 1 as it has been "flagged";
But we can continue (more depth) to vertex 3

Example (3)

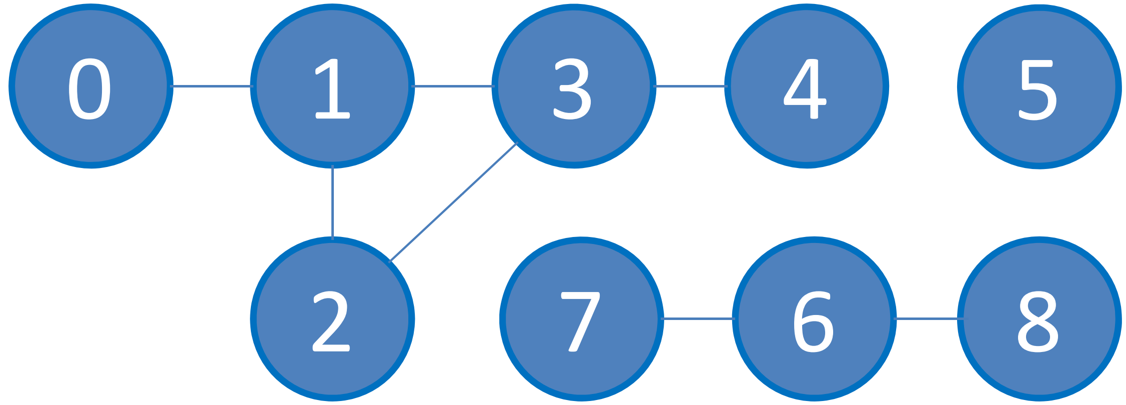


Assume that we start from source s = 0, neighbors are listed in ascending order

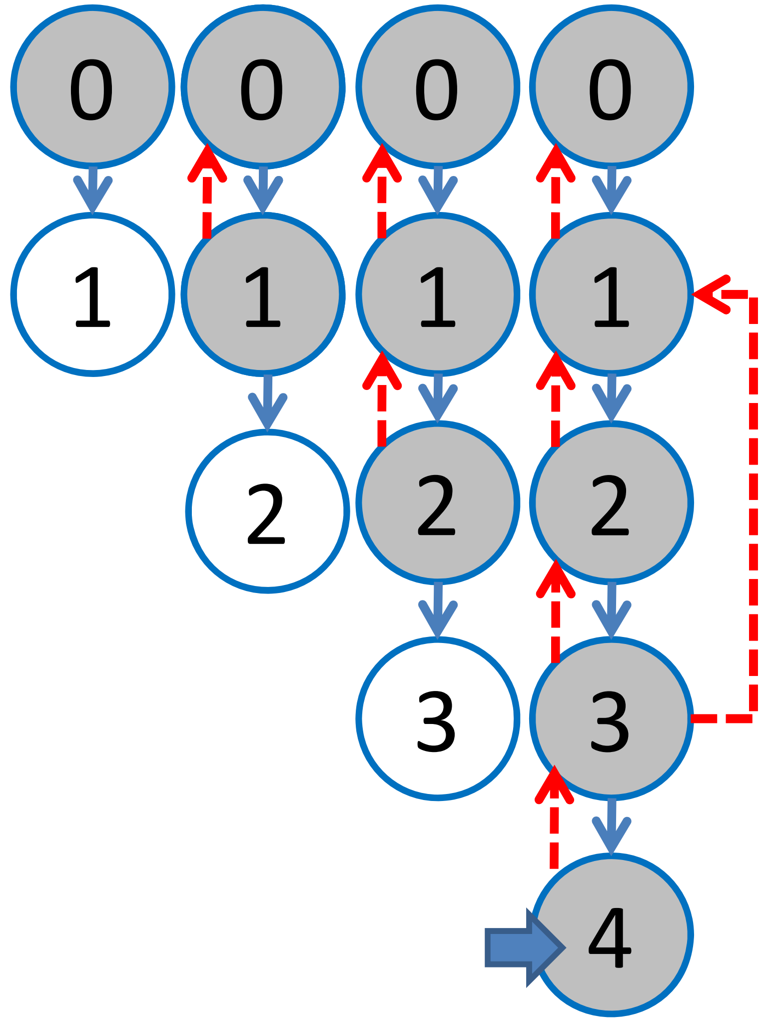


At vertex 3, we cannot go back to vertex 1 or to vertex 2 as both have been “flagged”;
But we can continue (more depth) to vertex 4

Example (4)

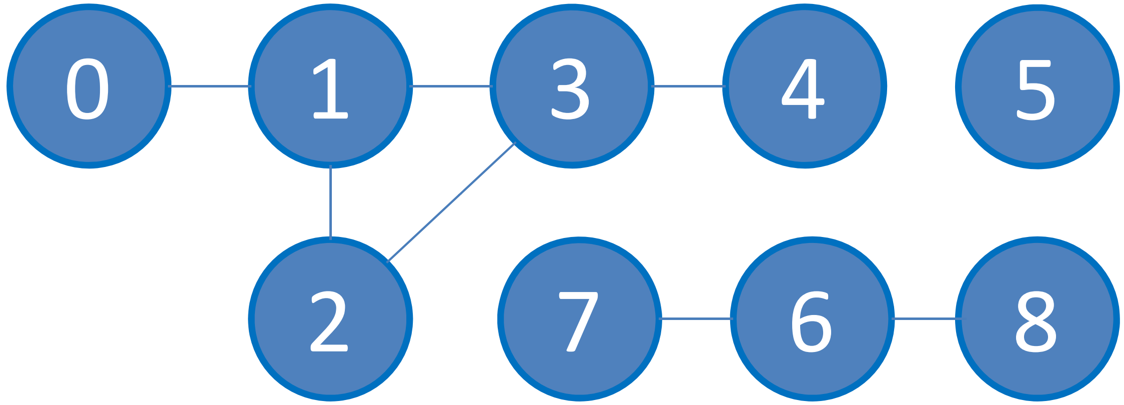


Assume that we start from source s = 0, neighbors are listed in ascending order

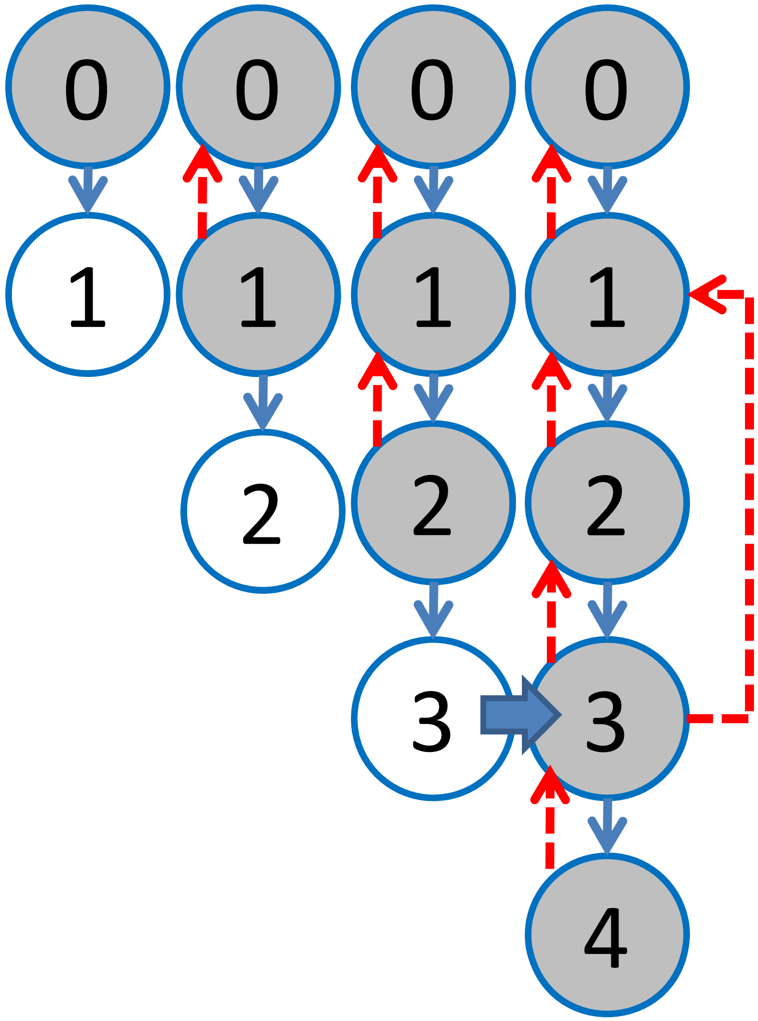


At vertex 4, we cannot go back to vertex 3 as it has been “flagged”;
All neighbors of vertex 4 have been explored, we now “backtrack” to previous vertex

Example (5)

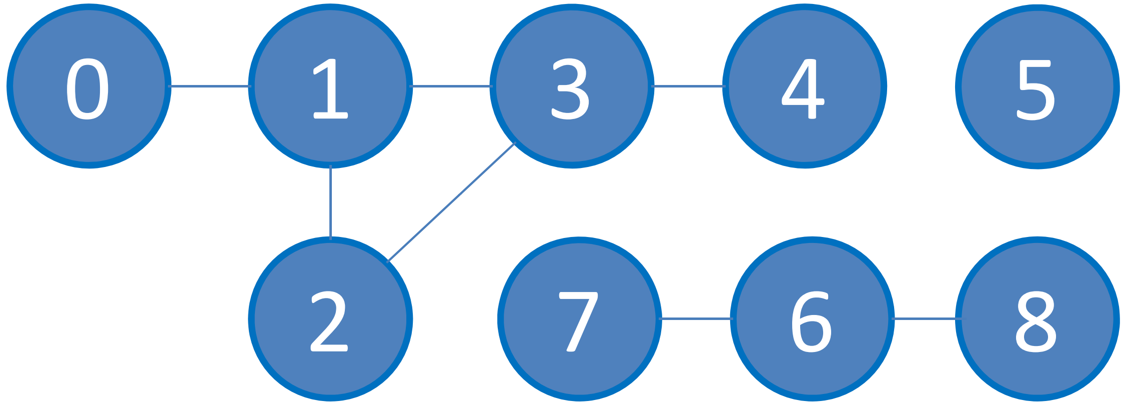


Assume that we start from source s = 0, neighbors are listed in ascending order

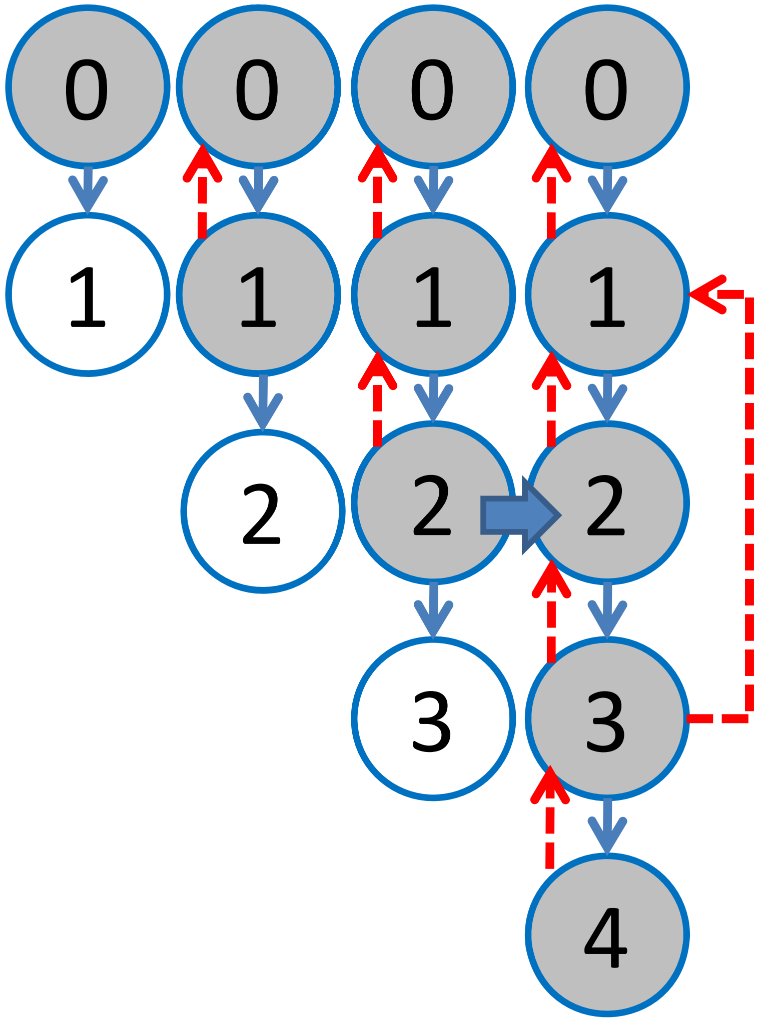


Back at vertex 3, all 3 neighbors have now been visited, we backtrack again

Example (6)

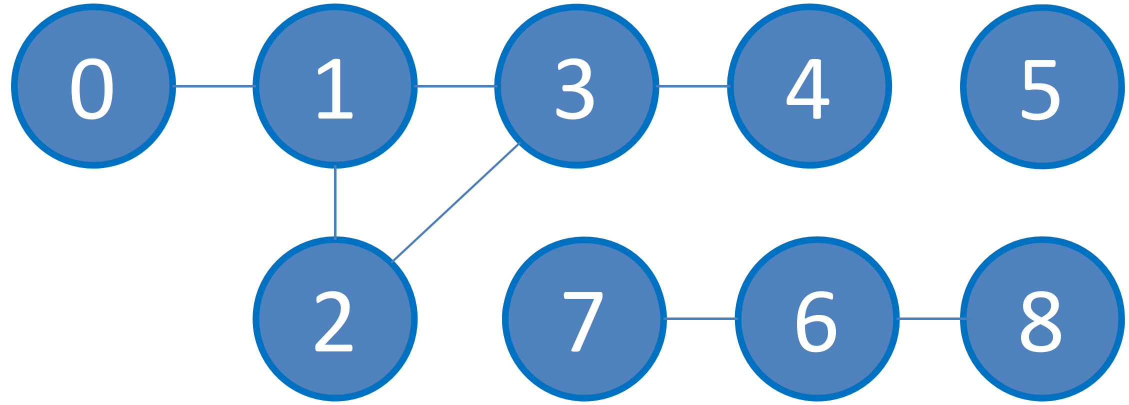


Assume that we start from source s = 0, neighbors are listed in ascending order

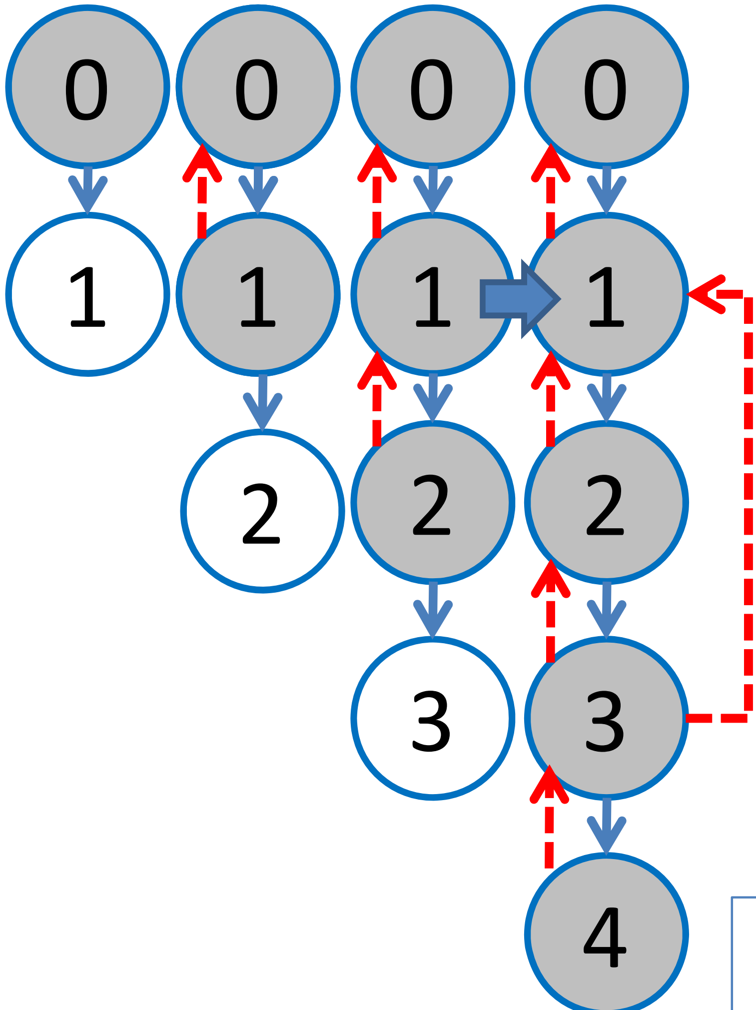


Back at vertex 2, all 2 neighbors have now been visited, we backtrack again

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

The blue (solid) arrows form the **DFS spanning tree** of the **component/sub graph** of the original graph when DFS is started from vertex 0

DFS Analysis

```
DFSrec(u)
  visited[u] ← 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 like this reverses 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}
```

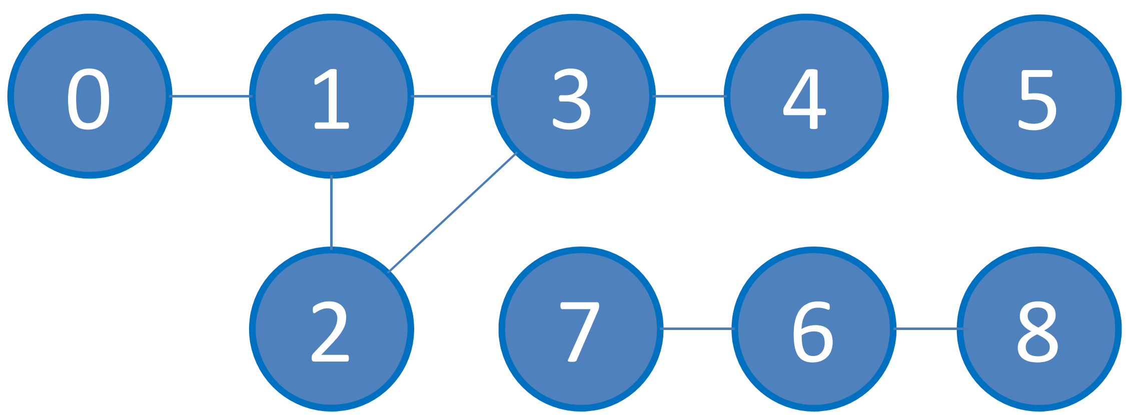
DFS/BFS Visualization

- Will be helpful for those who learn visually:
  www.comp.nus.edu.sg/~stevenha/visualization/dfsbfs.html

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**

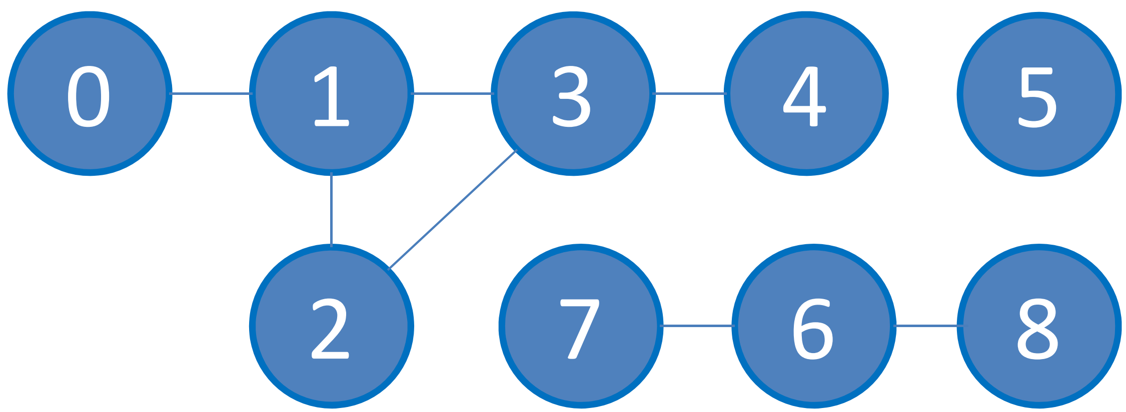
```
BFS(u) // DFSrec(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



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

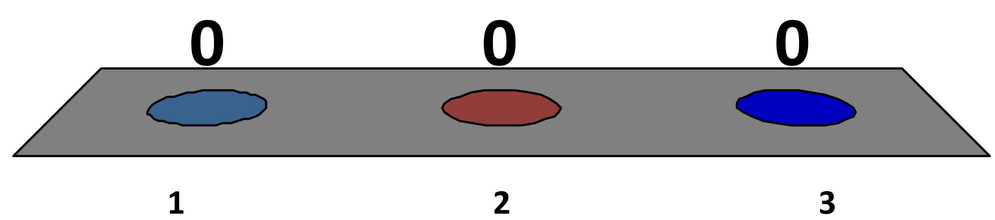
- Identifying component(s)
  - Component is sub graph in which any 2 vertices are connected to each other by paths, and is connected to no additional vertices
  - Identify/label/count components in graph G
  - Solution:

```
CC ← 0
for all v in V
  visited[v] ← 0
for all v in V // O(V)?
  if visited[v] == 0
    CC ← CC + 1
    DFSrec(v)//O(V+E)?
    // PS: BFS from v
    // 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(______)

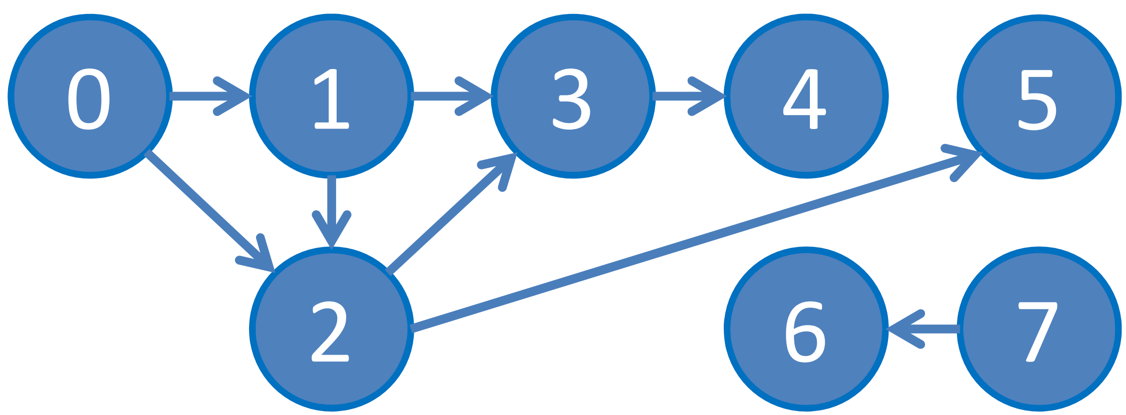


0 of 120

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

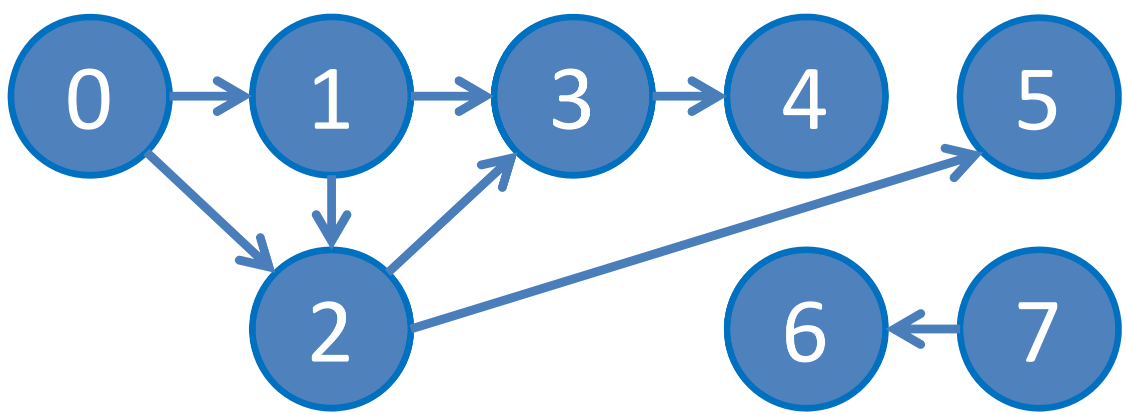
– Topological Sort
  - Topological sort of a DAG is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges
  - Every DAG has one *or more* topological sorts
  - One of the main purpose of finding topological sort: for Dynamic Programming (DP) on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

Reminder to myself:
slow down here
(last year's survey result)



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
    - "Post-order" = process vertex u after all children of u have been visited
  - See pseudo code in the next slide



DFS for TopoSort – Pseudo Code

Simply look at the codes in <u>red/underlined</u>

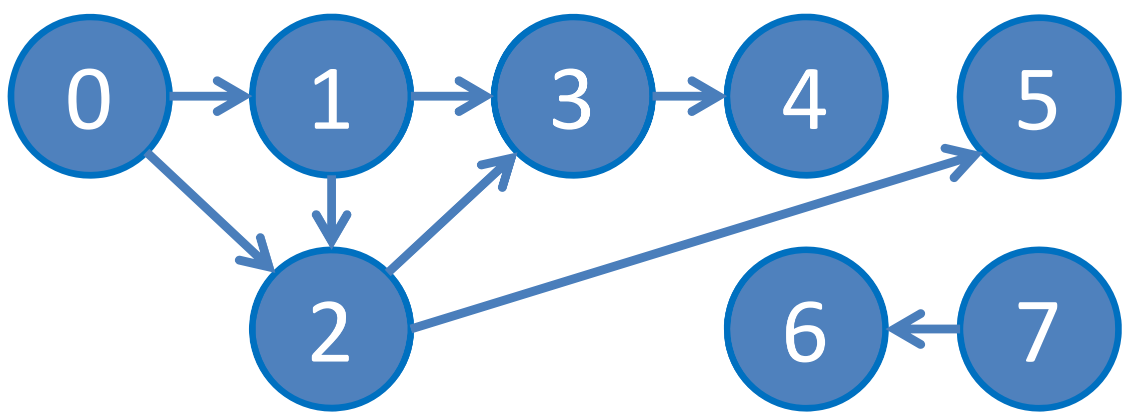
```
DFSrec(u)
  visited[u] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)
  append u to the back of toposort // "post-order"

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



Trade-Off

- O(V + E) DFS
  - Pro:
    - Slightly easier? to code (this one depends)
    - Use less memory
    - Has some extra features (not in CS2010 syllabus and useful for your PS3)
  - Cons:
    - Cannot solve SSSP on unweighted graphs
- O(V + E) BFS
  - Pro:
    - Can solve SSSP on unweighted graphs
  - Cons:
    - Slightly longer? to code (this one depends)
    - Use more memory (especially for the queue)

Summary

- In this lecture we looked at:
  - Graph terminologies + why we have to learn graph
  - How to store graph information in computer memory
    - Some applications with just graph data structure
  - 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”
    - Some applications: Reachability, CC, Toposort