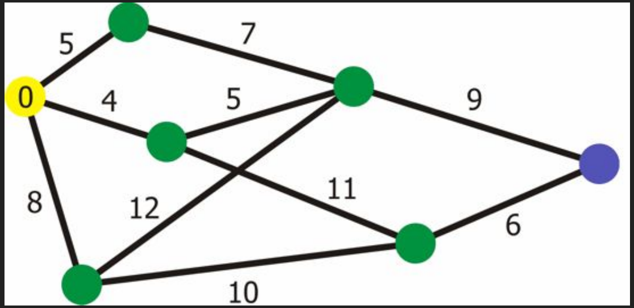
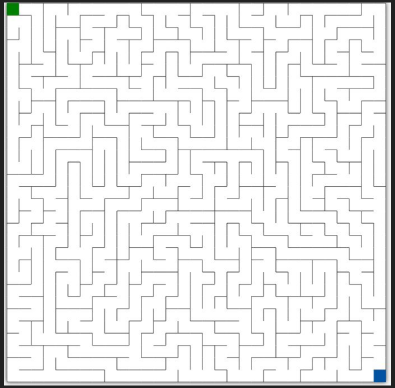
Please do not redistribute these slides without prior written permission



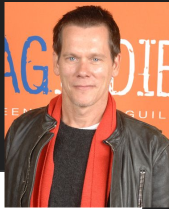


CS 5008/5009

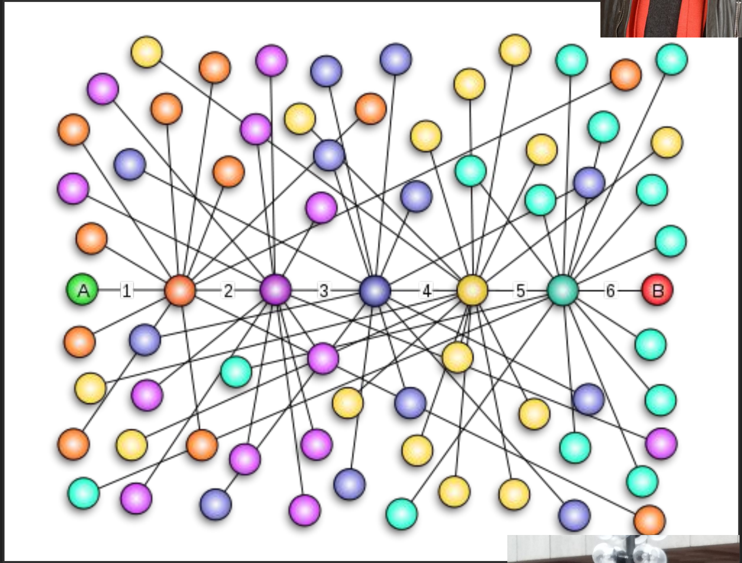
Data Structures, Algorithms, and Their Applications Within Computer Systems

Dr. Mike Shah

Pre-Class Warmup



- "**Six degrees of separation** is the idea that all people are six or fewer social connections away from each other so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps. It was originally set out by Frigyes Karinthy in 1929 and popularized in an eponymous 1990 play written by John Guare. It is sometimes generalized to the average social distance being logarithmic in the size of the population."
  - https://en.wikipedia.org/wiki/Six_degrees_of_separation



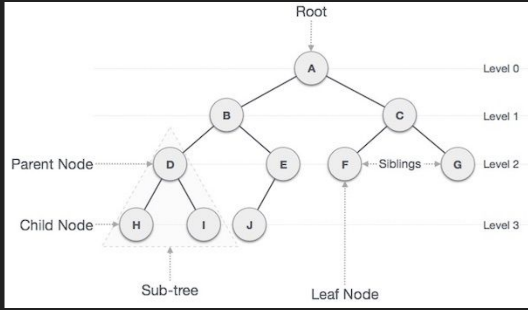
Note to self: Start audio recording of lecture :)
(Someone remind me if I forget!)

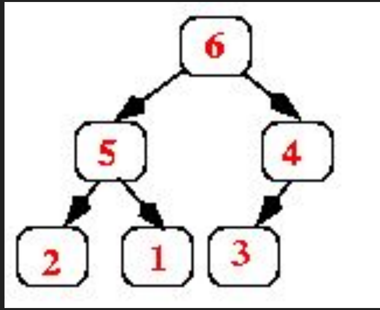
Course Logistics

- HW10 due shortly
    - HW11 will be released by Friday
- Lab 11 is out
- Make sure you are running your code on the servers whenever possible
    - ssh username@login.khoury.neu.edu

Last Time

```
typedef struct TreeNode{
  struct TreeNode* left;
  struct TreeNode* right;
  char data;
}TreeNode_t;
```





- Trees (DAGs, Directed Acyclic graphs) and Binary Search Tree
  - Discussed notation, properties, structure, and DFS
- Binary Heaps
  - Array-based binary tree, often used to implement priority queues and heap sort
- Priority Queues
  - Highest (or lowest) priority item is always at top of tree
  - When we remove item, we rebuild heap (sometimes called rebuild-heap or heapify) to recreate the max-heap or min-heap
  - Removal and insertion takes $O(\log_2 n)$
- Heap Sort
  - Another $O(n\log_2 n)$ sort
  - Remove ‘n items’ and $\log_2 n$ time to rebuild the heap

hw structure discussion

live drawing/coding/discussion

“It’s good to work with things that exist” Erik Demaine

"It's good to work with things that exist" Erik Demaine "Today, as always, we are solving real world problems"

Erik Demaine [wiki]

- Full Professor at MIT
  - http://erikdemaine.org/
- Expert in Computational Geometry
  - Specifically “computational origami”
  - Usually exhibited at MIT Museum of Science
- Offers lots of neat courses on algorithms, some of which are online for viewing!
  - https://en.wikipedia.org/wiki/Erik_Demaine

Today

- Trees (Special instances of Graphs)
- Graphs
- Data Structures supporting Graphs
  - Adjacency Matrix
  - Adjacency List
- Topological Sort
- Breadth-First Search
- Preview of ‘greedy’ algorithms

Trees

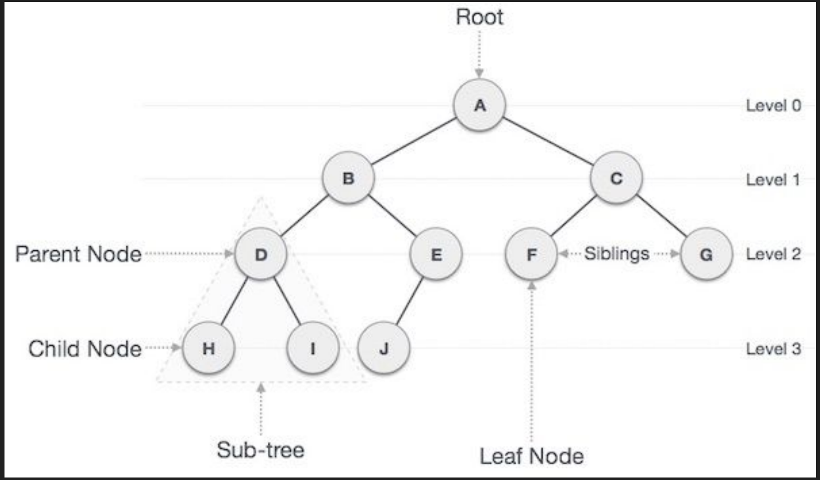
(Again Round 2)

The tree shown is a Baobab tree that I took a picture of in Tanzania in 2019. The indentation (top-left) is from elephants scratching their back frequently on that part of the tree :)

Trees (1/2)

```
typedef struct TreeNode{
  struct TreeNode* left;
  struct TreeNode* right;
  char data;
}TreeNode_t;
```

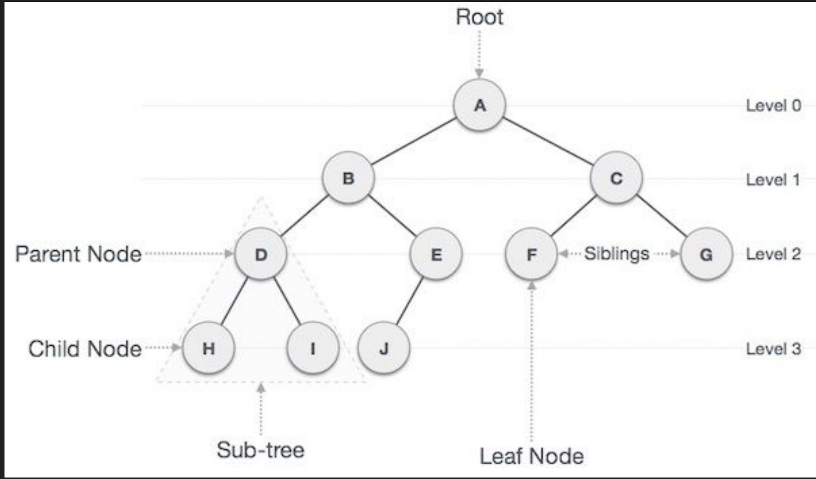
- Previously we discussed trees
- Trees are a data structure used to show a hierarchical relationship
- As a reminder let's review the pieces of a tree
  - Trees have a root at the top that is our 'starting position'
  - Nodes other than the root, have 'parents'
    - e.g. Node 'B''s parent is 'A'
  - Nodes at a lower level are children of the parent
    - e.g. Node 'B' is a child of 'A'



Trees (2/2)

```
typedef struct TreeNode{
  struct TreeNode* left;
  struct TreeNode* right;
  char data;
}TreeNode_t;
```

- A ‘TreeNode’ has ‘brother/sibling nodes’ named ‘left’ and ‘right’ in the case of a binary tree
- Binary Trees, when ordered, are known as Binary Search Trees (BST)
    - (i.e. nodes to the left are always less than their parent, and nodes to the right are always greater than their parent for an ascending ordering)



Other Trees (We have already seen!)

```
typedef struct node{
        int myData;
        struct node* next;
}node_t;
```

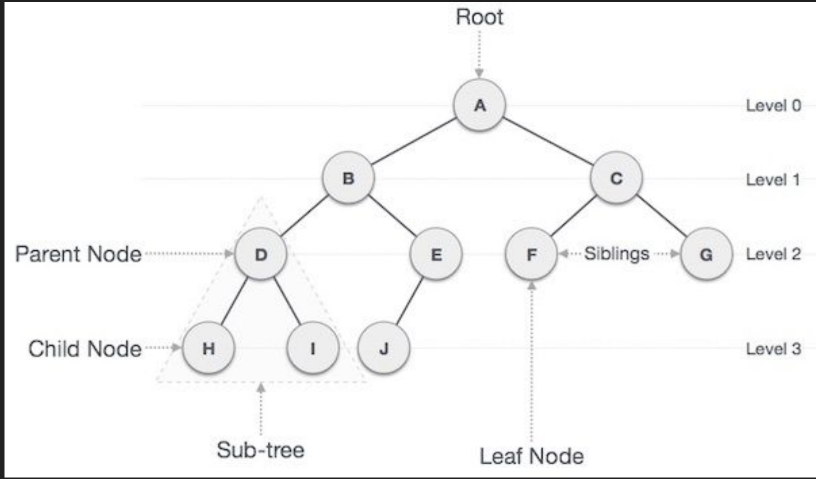
- A singly linked list is also a ‘tree’ data structure as well
- Though, we typically just think of it as a ‘linked data structure’
    - The tree shown on the right we can also say is very unbalanced tree.
        - All of the nodes are going to the ‘right’

Tree Traversals

We do not have built-in ‘for-loops’ for a tree structure, so traversals are our way of ‘iterating’ or ‘searching’ a tree.

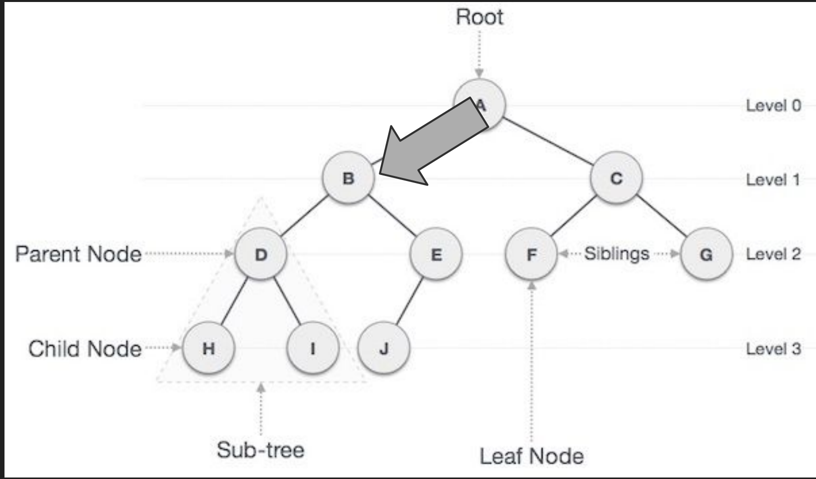
We previously learned DFS (0/16)

- DFS is one way to traverse and search for a node
- We are 'picking' a branch and recurse all the way down
    - Then we return back to the last place we have not visited



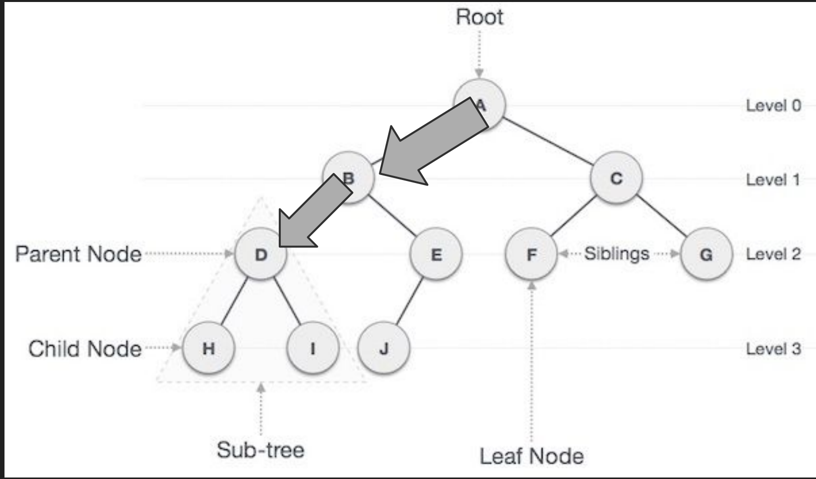
We previously learned DFS (1/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
    - Then we return back to the last place we have not visited



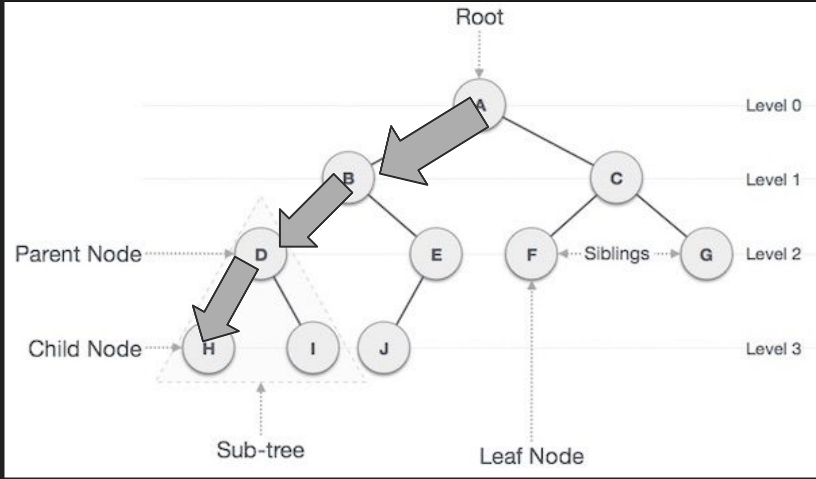
We previously learned DFS (2/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



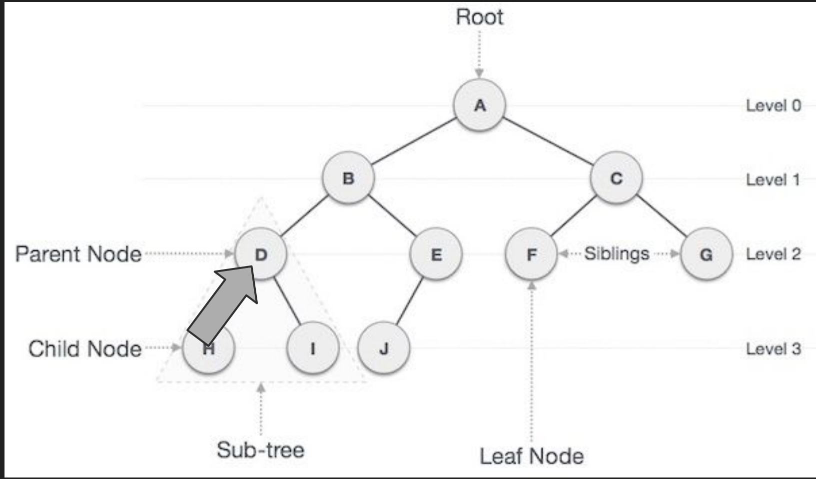
We previously learned DFS (3/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



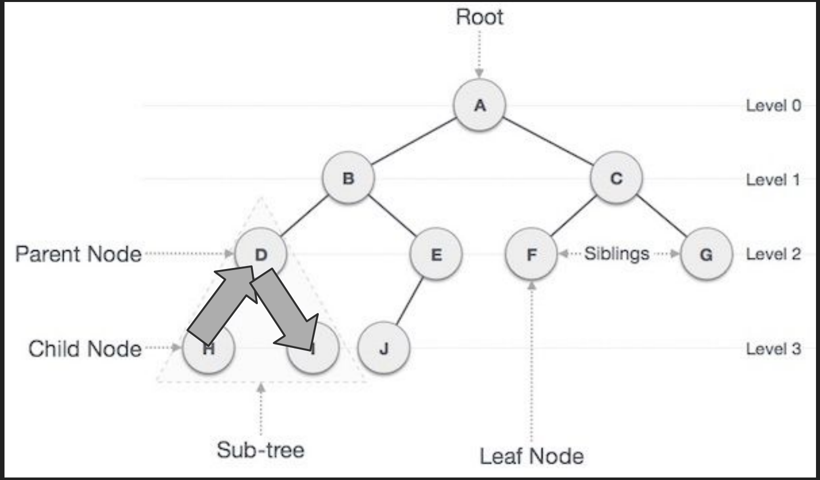
We previously learned DFS (4/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



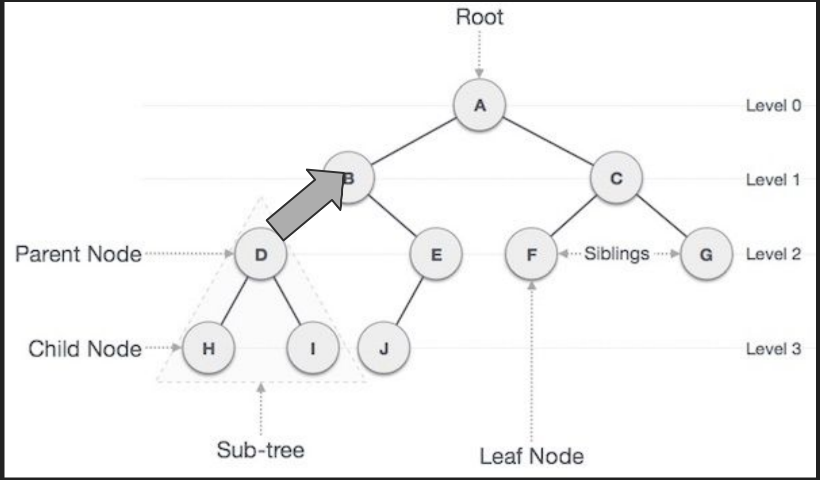
We previously learned DFS (5/16)

- DFS is one way to traverse and search for a node
- We are 'picking' a branch and recurse all the way down
  - Then we return back to the last place we have not visited



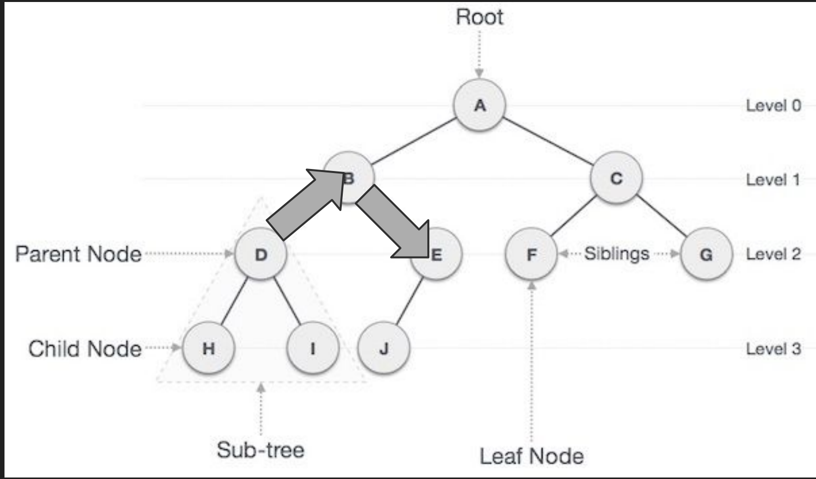
We previously learned DFS (6/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



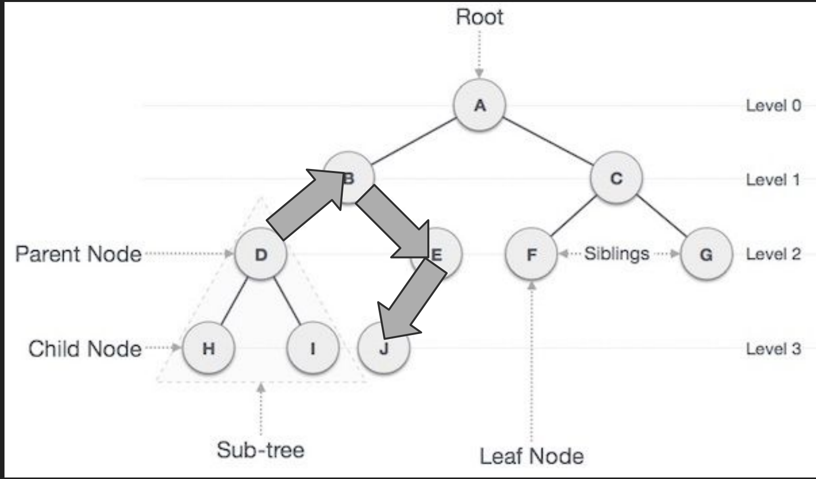
We previously learned DFS (7/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



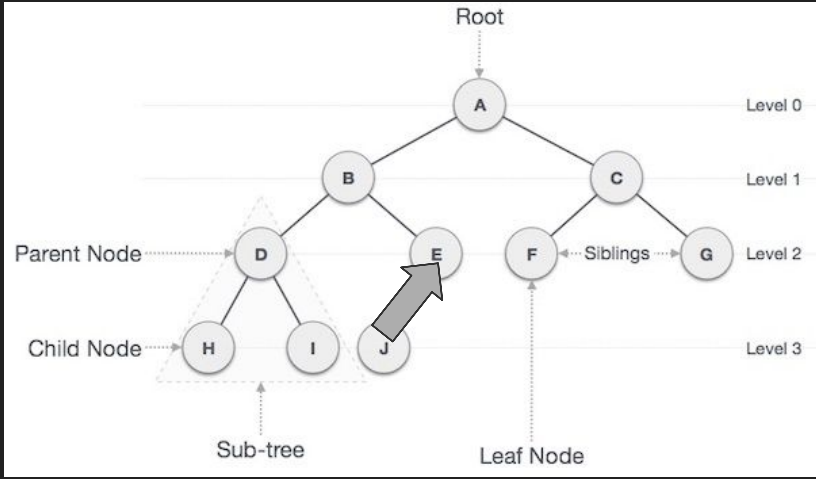
We previously learned DFS (8/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
    - Then we return back to the last place we have not visited



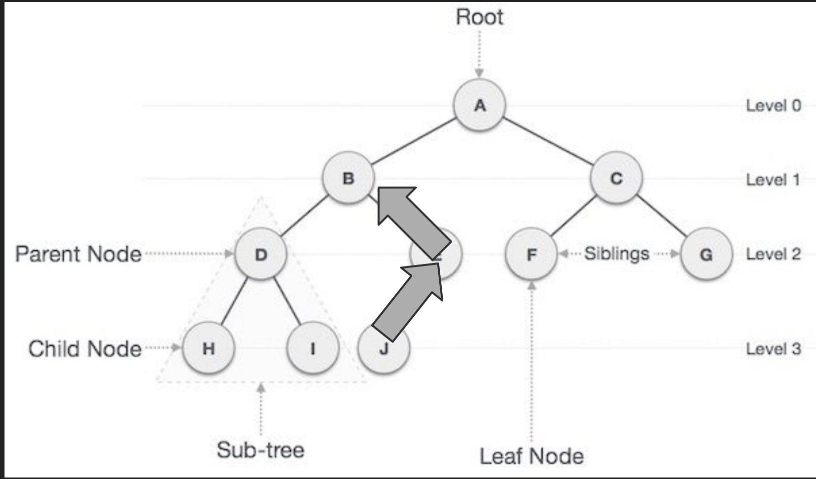
We previously learned DFS (9/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



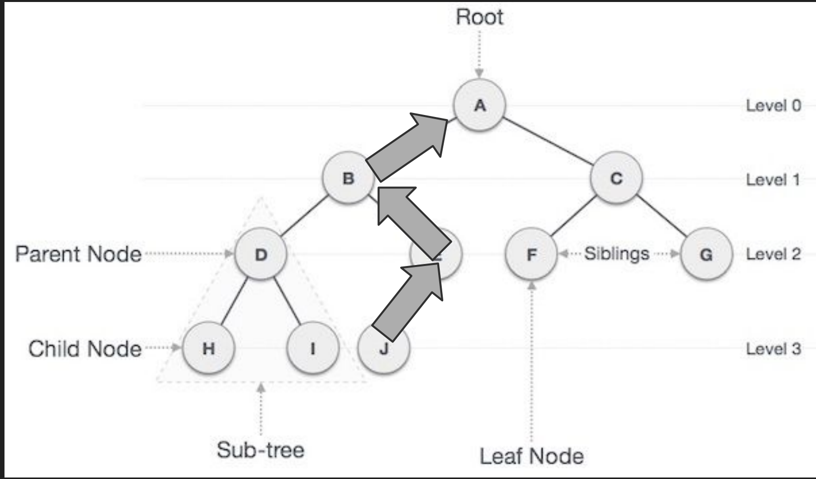
We previously learned DFS (10/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



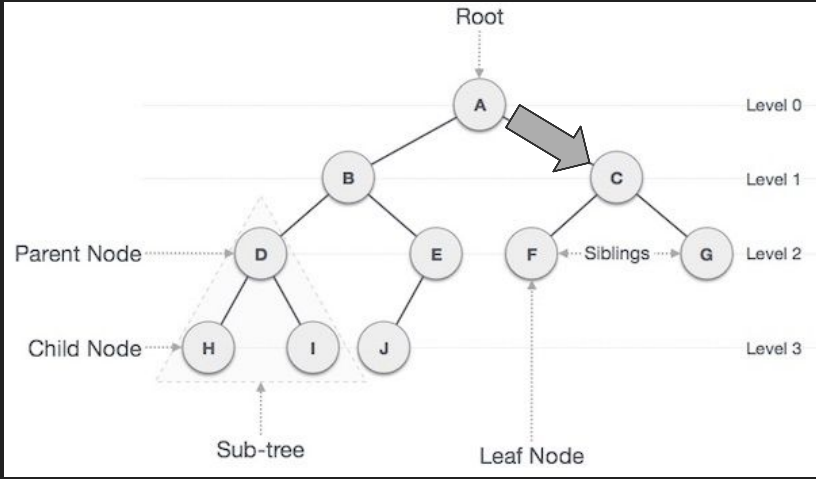
We previously learned DFS (11/16)

- DFS is one way to traverse and search for a node
- We are 'picking' a branch and recurse all the way down
    - Then we return back to the last place we have not visited



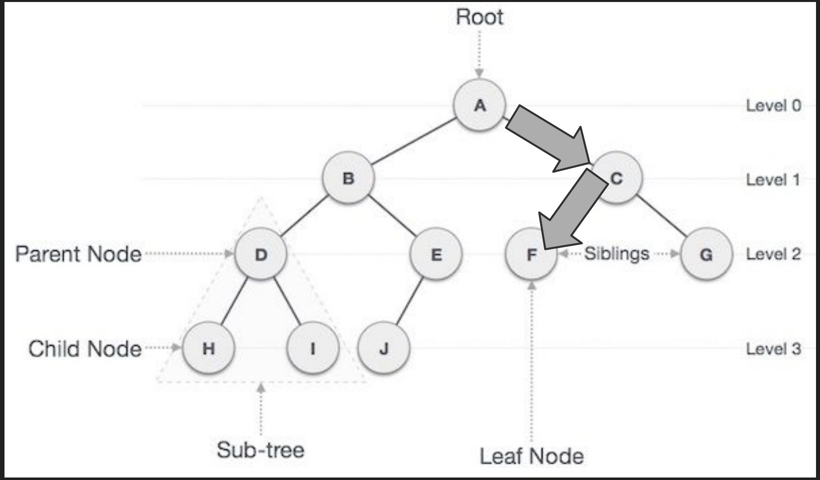
We previously learned DFS (12/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



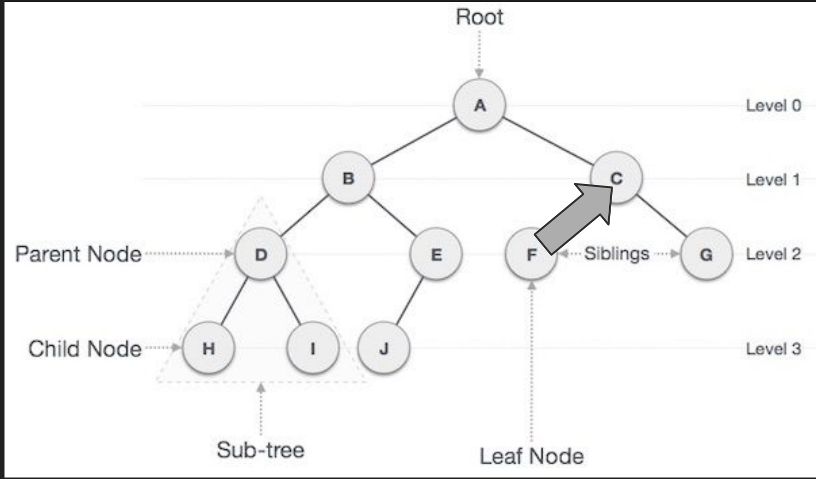
We previously learned DFS (13/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



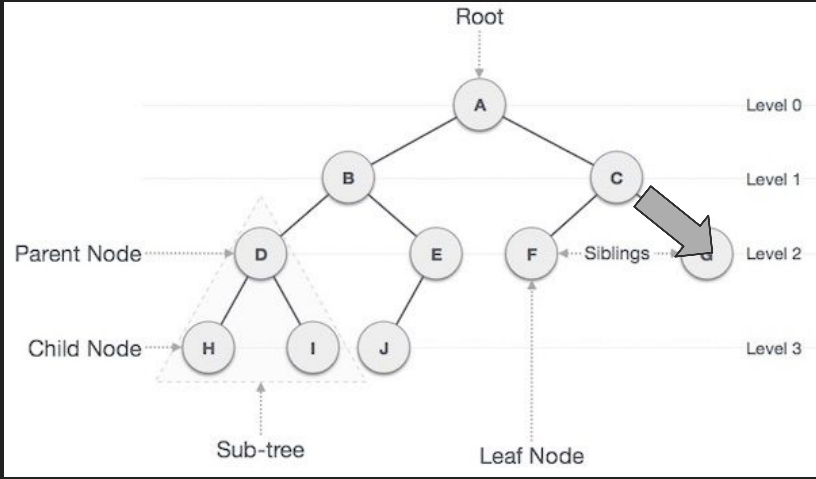
We previously learned DFS (14/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



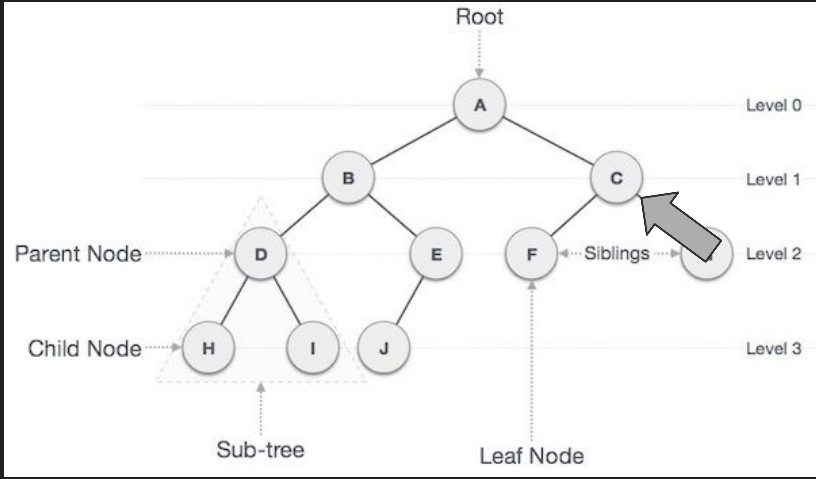
We previously learned DFS (15/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
    - Then we return back to the last place we have not visited



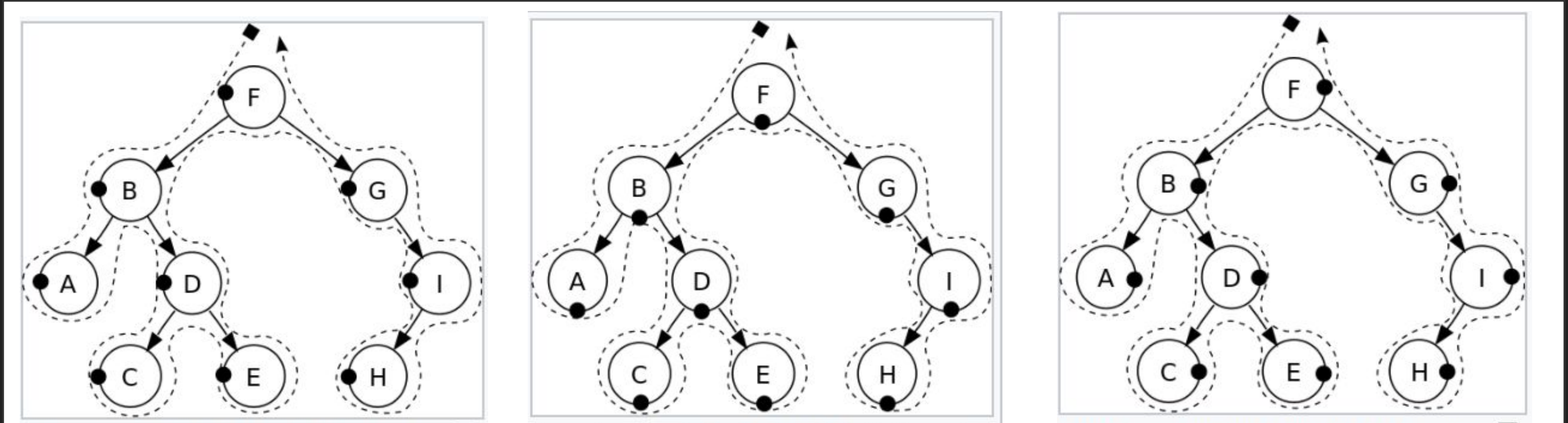
We previously learned DFS (16/16)

- DFS is one way to traverse and search for a node
- We are ‘picking’ a branch and recurse all the way down
  - Then we return back to the last place we have not visited



DFS - Traversals (1/3)

- Depending on how we traverse, there are 3 common traversals (many more exist)
  - pre-order traversal
  - in order traversal
  - post-order traversal



Pre-order: F, B, A, D, C, E, G, I, H.

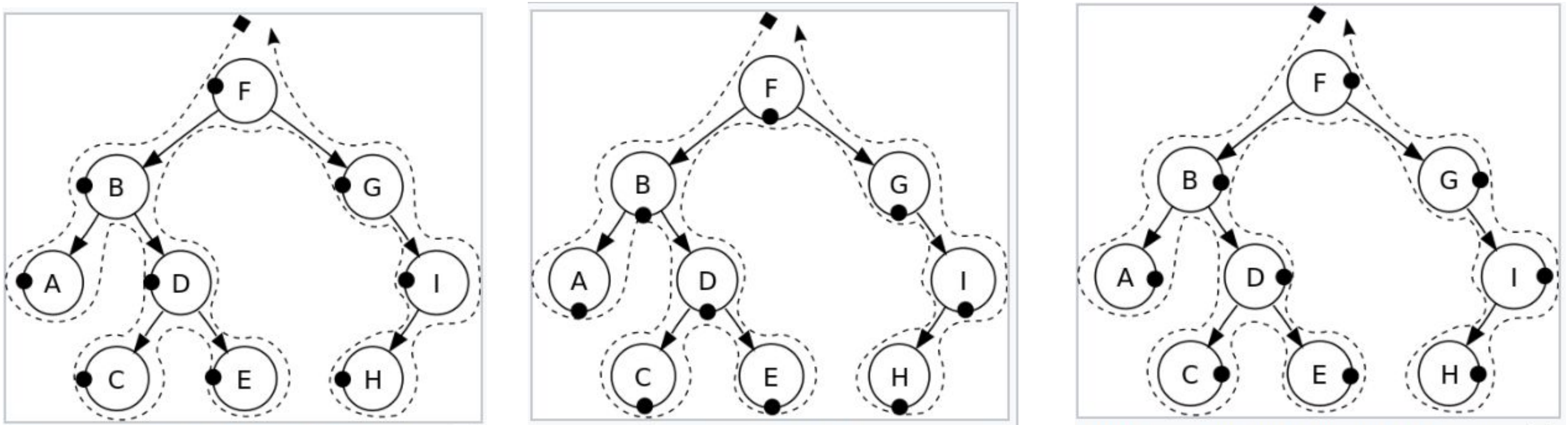
In-order: A, B, C, D, E, F, G, H, I.

Post-order: A, C, E, D, B, H, I, G, F.

DFS - Trav

- Depending o ... sals (many more exist)
  - pre-order t
  - in order tra
  - post-order traversal

(Question to audience): If I wanted to print out the nodes in 'pre-order', when would I print? (i.e. before I recurse, after I recurse left, or after I recurse right?)



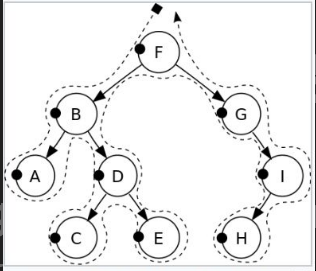
Pre-order: F, B, A, D, C, E, G, I, H.

In-order: A, B, C, D, E, F, G, H, I.

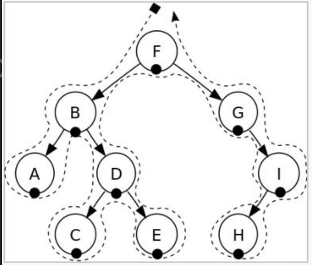
Post-order: A, C, E, D, B, H, I, G, F.

DFS - Tra

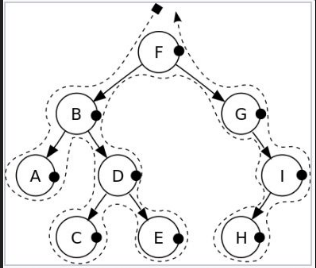
- Depending
  - pre-orde
  - in order
  - post-ord



Pre-order: F, B, A, D, C, E, G, I, H.



In-order: A, B, C, D, E, F, G, H, I.



Post-order: A, C, E, D, B, H, I, G, F.

Pre-order (NLR) [edit]

1. Check if the current node is empty or null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order function.
4. Traverse the right subtree by recursively calling the pre-order function.

In-order (LNR) [edit]

1. Check if the current node is empty or null.
2. Traverse the left subtree by recursively calling the in-order function.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order function.

Post-order (LRN) [edit]

1. Check if the current node is empty or null.
2. Traverse the left subtree by recursively calling the post-order function.
3. Traverse the right subtree by recursively calling the post-order function.
4. Display the data part of the root (or current node).

Sorting Trees

Topological Sort

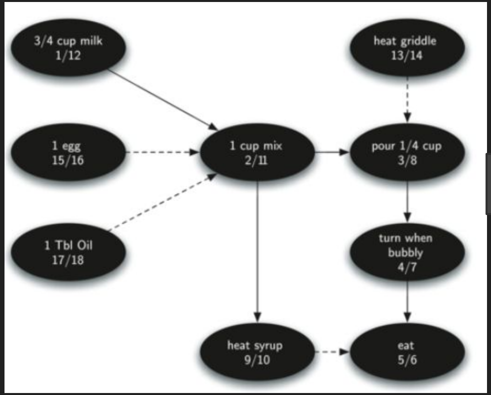
Theme of this class (so far) - Searching and Sorting

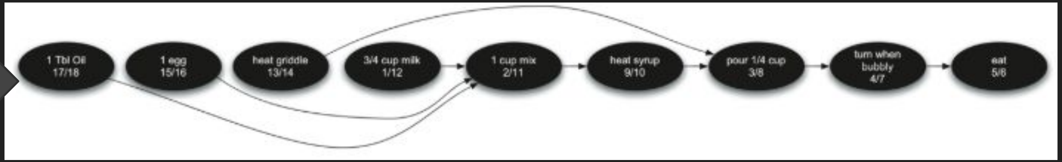
- As we add new data structures, we need some way to search (or traverse) them.
- It is also handy if we can 'sort' them

- We are going to introduce a way to 'sort' or provide an ordering for Directed-Acyclic Graphs (a.k.a. trees!) called Topological Sort
    - Topological sort makes use of our Depth-First Search (DFS)

Back to baking recipes --Topological Sort (1/2)

- Given a Tree (i.e. a Directed Acyclic Graph (DAG)), a linear ordering of vertices can be produced.
  - This is a ‘sorting’ of the tree
  - Why does this matter?
    - It shows us where ‘dependencies’ are in our hierarchy
    - e.g. Baking example below shows an ordering in which you can mix ingredients





Back to baking recipes --Topological Sort (2/2)

a Directed Acyclic Graph (DAG)), a linear ordering of

roduced.

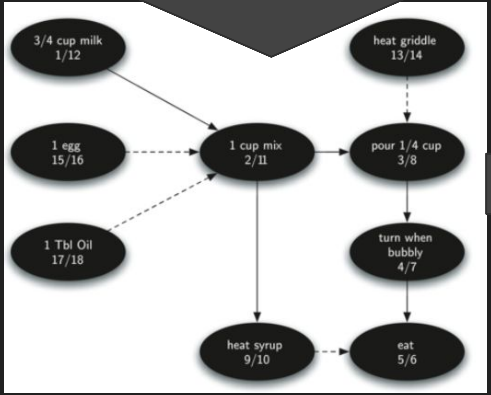
of the tree

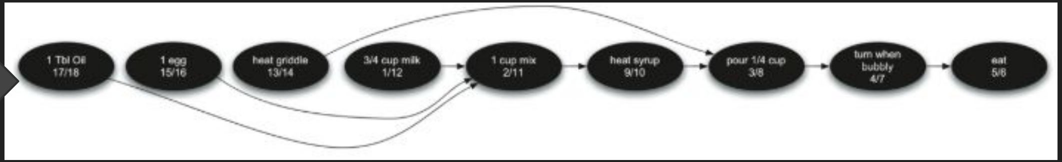
atter?

ependencies’ are in our hierarchy

xample below shows an ordering in which you can mix ingredients

Note that you can have multiple orderings (i.e. egg can come before oil, but all must come before 1 cup mix



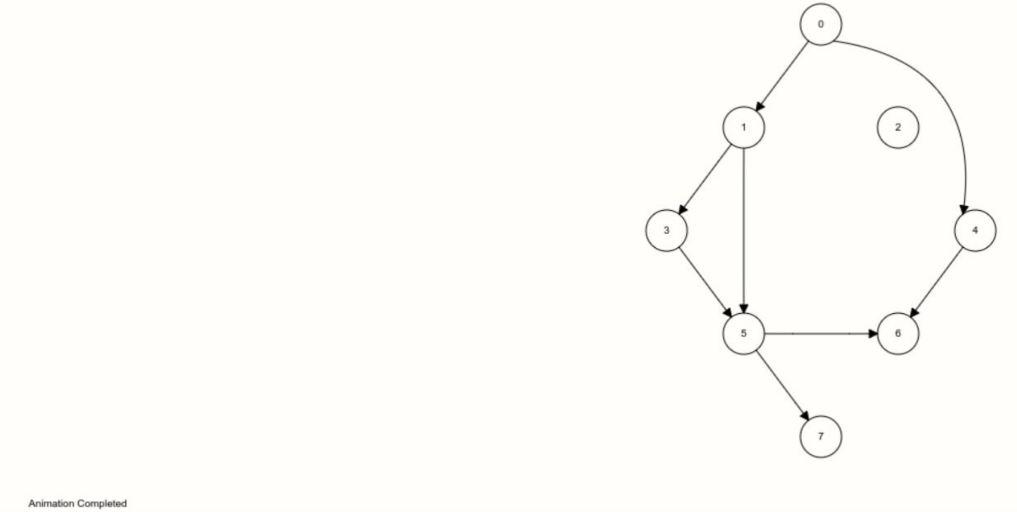


Topological Sort

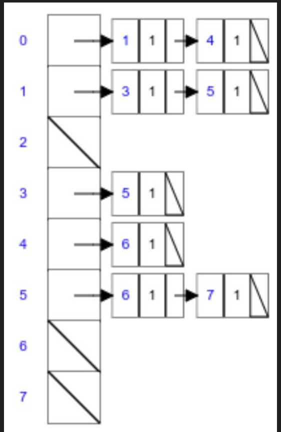
Example(s) useful for ‘Task Scheduling’

Topological Sort Intuition

- The high-level intuition of topological sort is that we are performing a ‘DFS’ from each node
  - Revisit this slide later on and see if you can follow along.
  - (Note: This was a randomly generated graph from: https://www.cs.usfca.edu/~galles/visualization/TopoSortDFS.html)



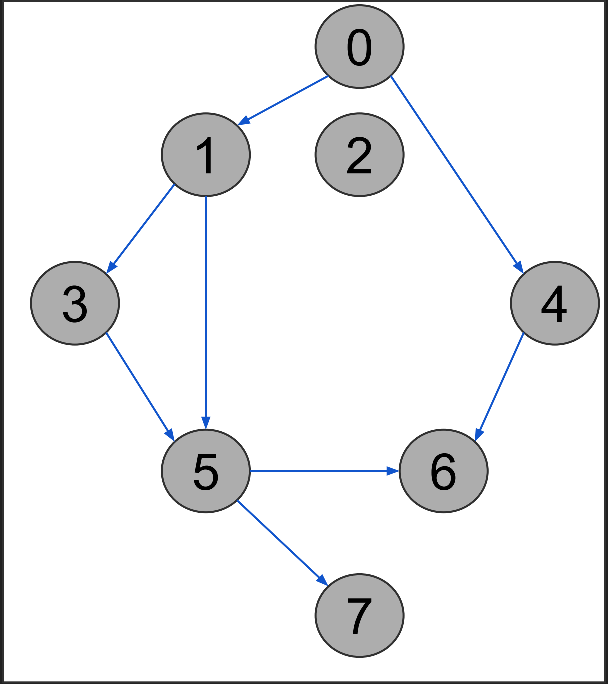
Adjacency List and Matrix representation of the Tree
(We’ll talk about these shortly!)



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

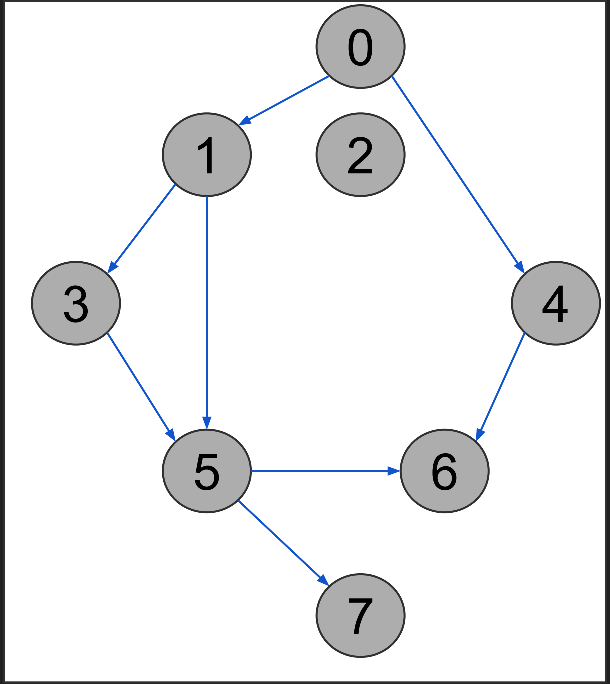
Topological Sort Prompt

- Pretend you have a list of CS classes
  - CS 0 - Seminar
  - CS 1 - Intro to Python
  - CS 2 - Discrete Math
  - CS 3 - Object-Oriented Design
  - CS 4 - Systems
  - etc.
- In order to graduate, you want to figure out *what order can I take the classes*
  - You can only take classes such that you have the prerequisites
  - There may be multiple orders you can take the classes, but you only really care about graduating, and taking that last beloved class-- ****CS 7****



Topological Sort Algorithm

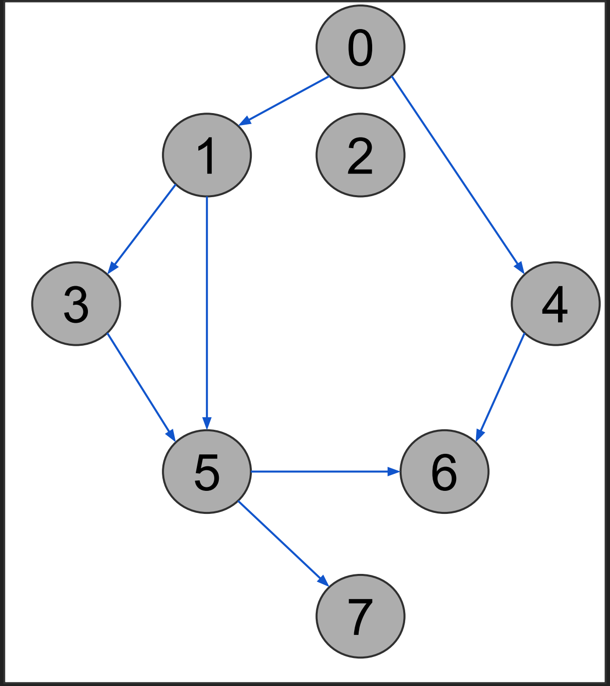
- Pick an unvisited node (typically the root)
  - Perform a Depth First Search (DFS) exploring only unvisited nodes
  - On recursive callback of DFS, add current node to the topological ordering in reverse order
    - (i.e. put that node into a stack, and then when you pop the stack, reverse the ordering)



Topological Sort Algorithm

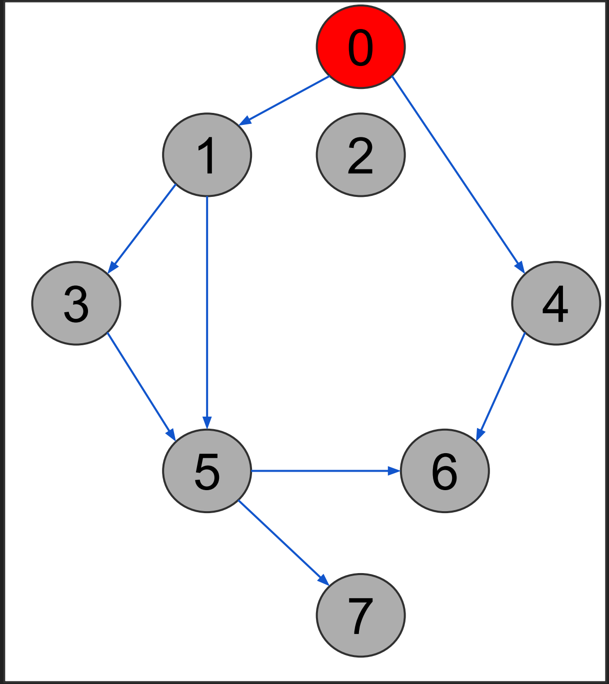
- Pick an unvisited node (typically the root)
  - Perform a Depth First Search (DFS) exploring only unvisited nodes
  - On recursive callback of DFS, add current node to the topological ordering in reverse order
    - (i.e. put that node into a stack, and then when you pop the stack, reverse the ordering)

- **Note:** In upcoming slides, ‘red’ will mark a node as visited, and the blue arrows will indicate our current call stack.



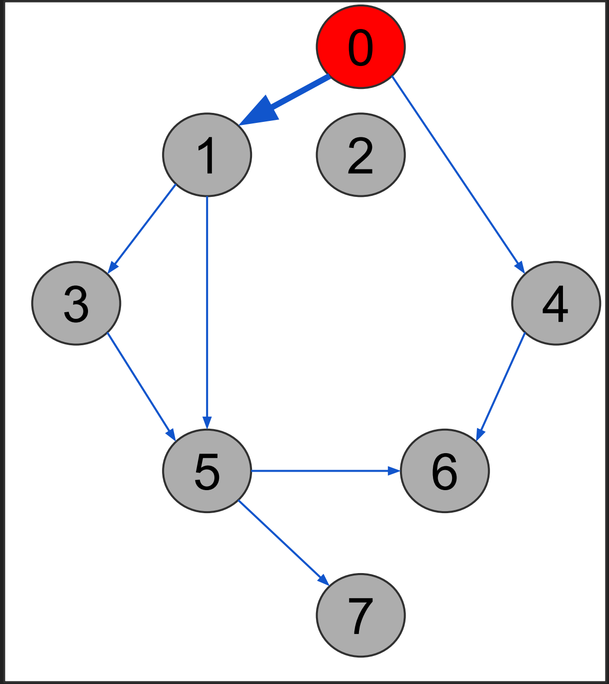
Topological Sort (1/)

- Pick a node (our root 0)



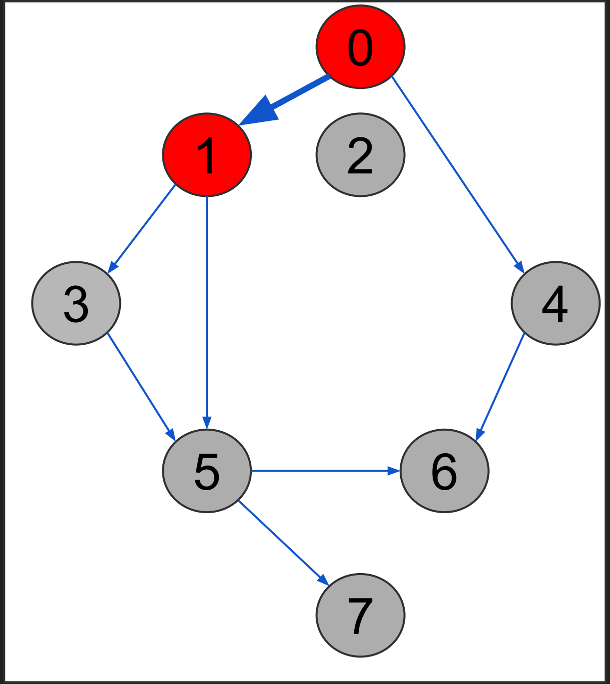
Topological Sort (1/)

- Pick a node (our root 0)
  - Start DFS
    - Pick 1 or 4
      - In this case I choose 1, it doesn't matter, perhaps you pick the smallest of the two nodes as a rule



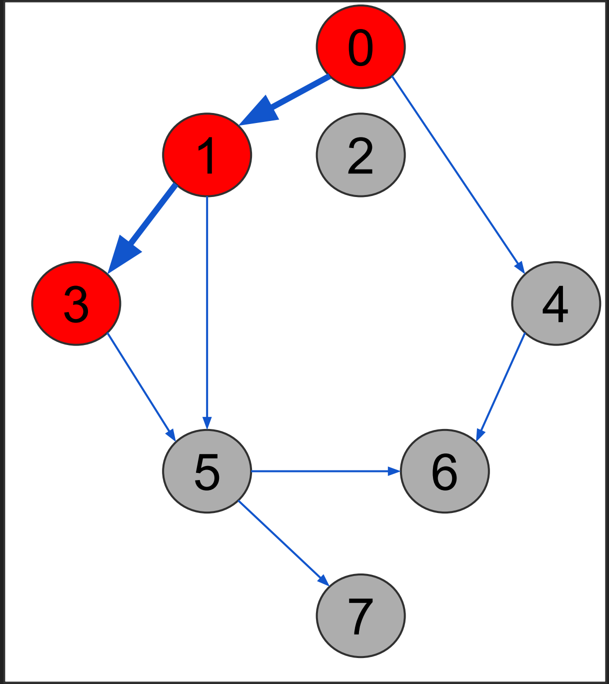
Topological Sort (1/)

- Pick a node (our root 0)
    - Start DFS
        - Pick 1 or 4
            - In this case I choose 1, it doesn't matter, perhaps you pick the smallest of the two nodes as a rule



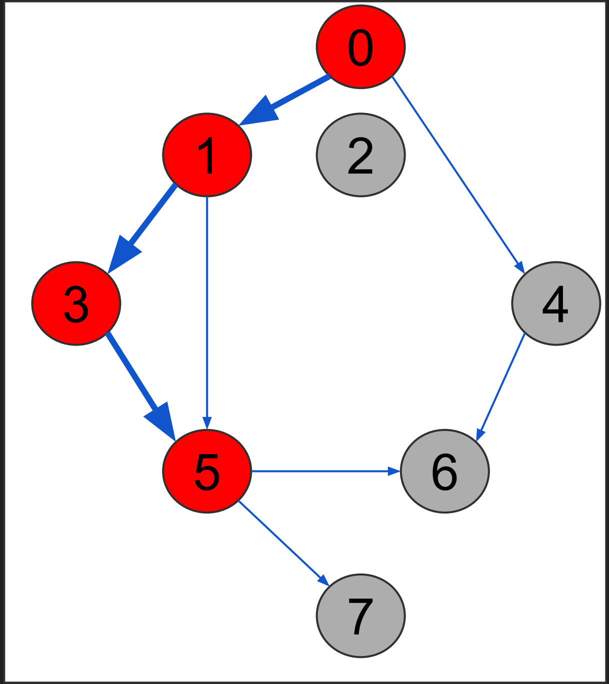
Topological Sort (1/)

- Pick a node (our root 0)
  - Start DFS
    - Pick 1 or 4
      - In this case I choose 1, it doesn't matter, perhaps you pick the smallest of the two nodes as a rule
    - From 1, recurse down
      - Choose 3 or 5 (I'll choose the smaller)



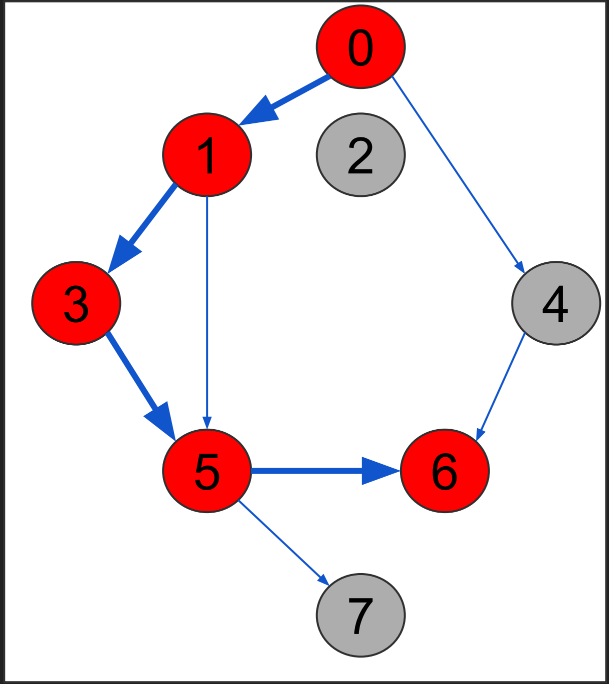
Topological Sort (1/)

- Pick a node (our root 0)
  - Start DFS
    - Pick 1 or 4
      - In this case I choose 1, it doesn't matter, perhaps you pick the smallest of the two nodes as a rule
    - From 1, recurse down
      - Choose 3 or 5 (I'll choose the smaller)
    - From 3, only one way to continue DFS to 5



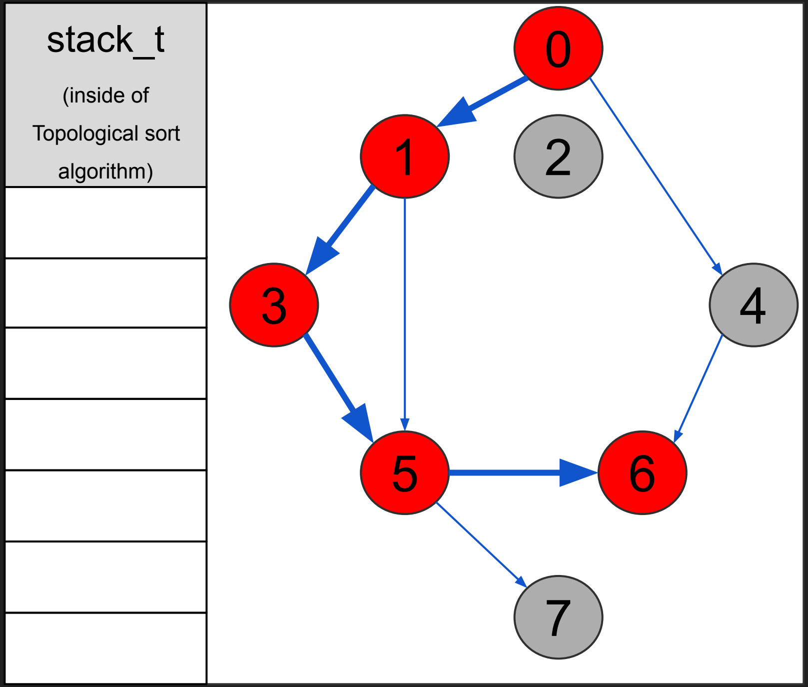
Topological Sort (1/)

- Pick a node (our root 0)
  - Start DFS
    - Pick 1 or 4
      - In this case I choose 1, it doesn't matter, perhaps you pick the smallest of the two nodes as a rule
    - From 1, recurse down
      - Choose 3 or 5 (I'll choose the smaller)
    - From 3, only one way to continue DFS to 5
    - From 5, recurse down to 6 or 7, I'll choose 6



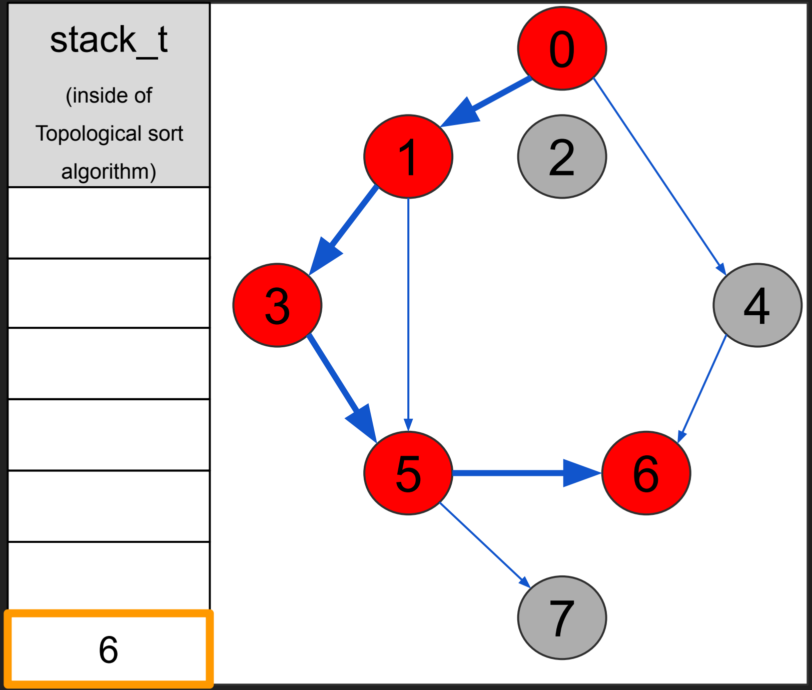
Topological Sort (1/)

- (continued...)
  - From 5, recurse down to 6 or 7, I'll choose 6
  - At this point, nowhere left to recurse--so we put '6' on a stack
    - This stack is stored inside of our topological sort
    - This is what will hold the sorted order for us



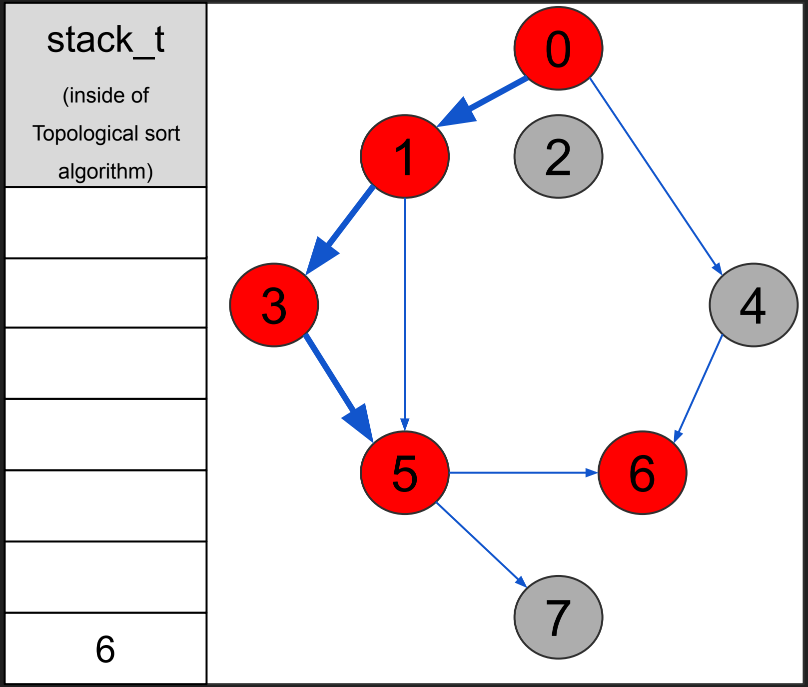
Topological Sort (1/)

- (continued...)
  - From 5, recurse down to 6 or 7, I'll choose 6
  - At this point, nowhere left to recurse--so we put '6' on a stack
    - This stack is stored inside of our topological sort
    - This is what will hold the sorted order for us



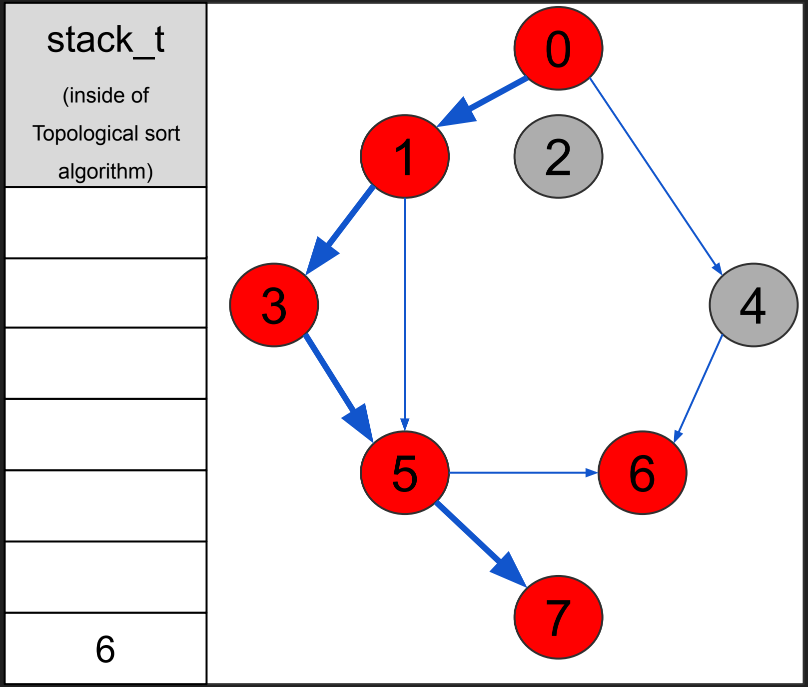
Topological Sort (1/)

- (continued...)
    - From 5, recurse down to 6 or 7, I'll choose 6
    - At this point, nowhere left to recurse--so we put '6' on a stack
        - This stack is stored inside of our topological sort
        - This is what will hold the sorted order for us
- Now we recurse back to 5



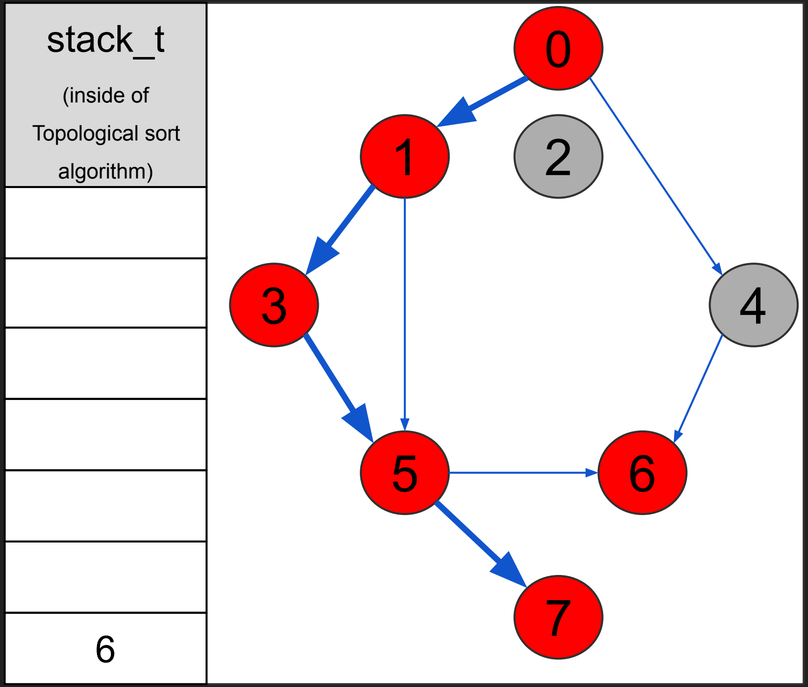
Topological Sort (1/)

- (continued...)
  - From 5, recurse down to 6 or 7, I'll choose 6
  - At this point, nowhere left to recurse--so we put '6' on a stack
    - This stack is stored inside of our topological sort
    - This is what will hold the sorted order for us
- Now we recurse back to 5
  - From '5' we can continue DFS to 7



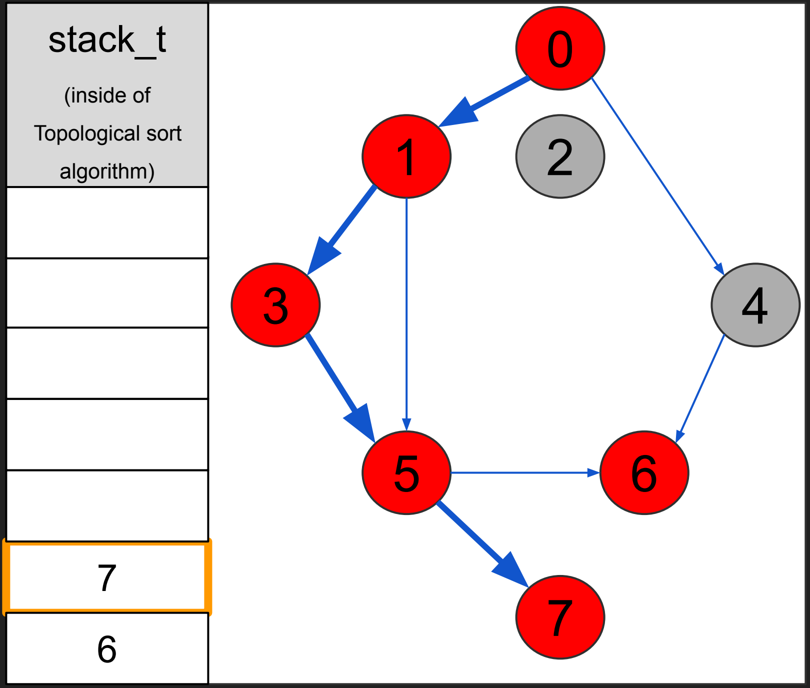
Topological Sort (1/)

- (continued...)
  - From 5, recurse down to 6 or 7, I'll choose 6
  - At this point, nowhere left to recurse--so we put '6' on a stack
    - This stack is stored inside of our topological sort
    - This is what will hold the sorted order for us
- Now we recurse back to 5
  - From '5' we can continue DFS to 7
  - From '7', there is no where to recurse, so we add to our stack



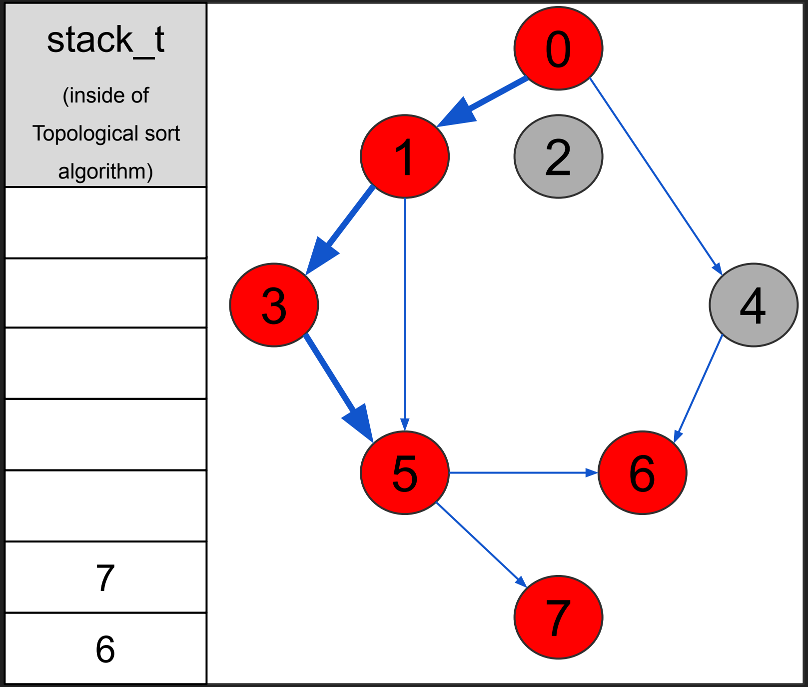
Topological Sort (1/)

- (continued...)
    - From 5, recurse down to 6 or 7, I'll choose 6
    - At this point, nowhere left to recurse--so we put '6' on a stack
        - This stack is stored inside of our topological sort
        - This is what will hold the sorted order for us
- Now we recurse back to 5
    - From '5' we can continue DFS to 7
    - From '7', there is no where to recurse, so we add to our stack



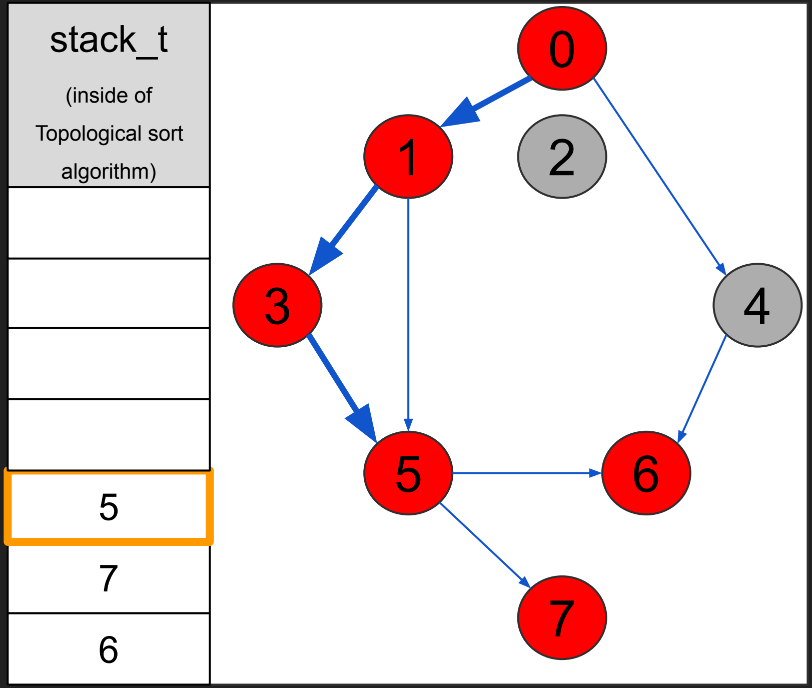
Topological Sort (1/)

- (continued...)
  - Recurse back to 5
    - No where to explore from 5 (i.e. no more nodes to continue our DFS), so we add 5 to the stack



Topological Sort (1/)

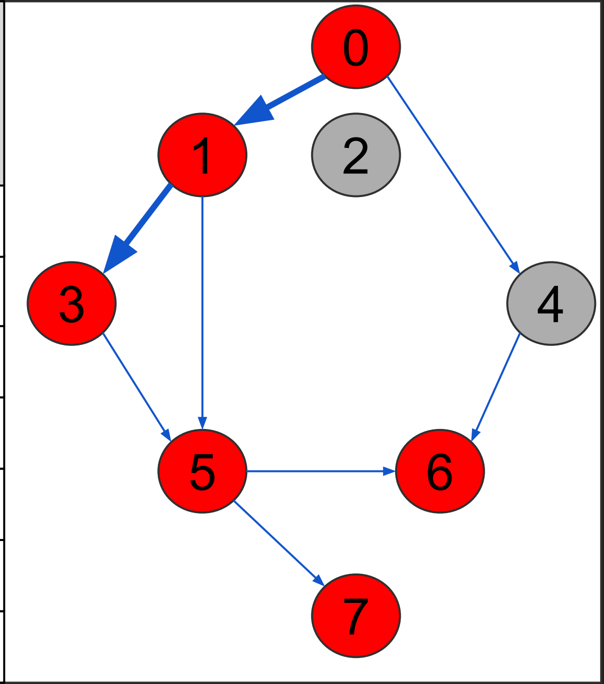
- (continued...)
  - Recurse back to 5
    - No where to explore from 5 (i.e. no more nodes to continue our DFS), so we add 5 to the stack



Topological Sort (1/)

- (continued...)
  - Recurse back to 5
    - No where to explore from 5 (i.e. no more nodes to continue our DFS), so we add 5 to the stack
  - Now we recurse back to where we came from, node 3

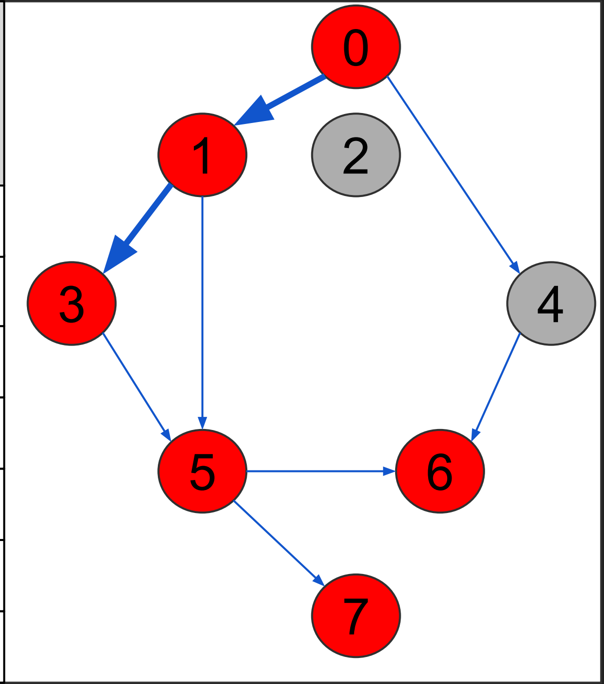
stack_t (inside of Topological sort algorithm)
5
7
6



Topological Sort (1/)

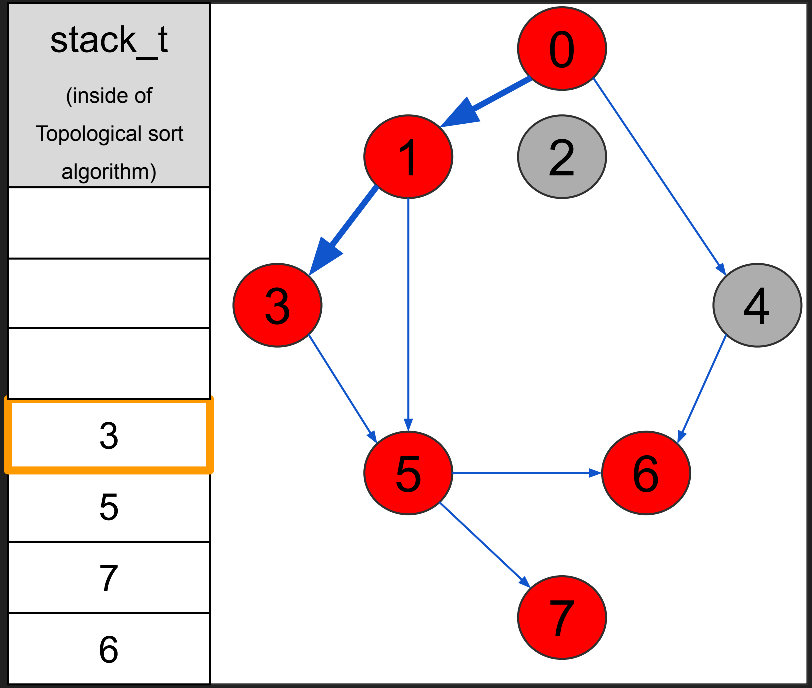
- (continued...)
  - Recurse back to 5
    - No where to explore from 5 (i.e. no more nodes to continue our DFS), so we add 5 to the stack
  - Now we recurse back to where we came from, node 3
    - From 3, no where else to recurse from and continue our DFS
      - So add 3 to stack

stack_t (inside of Topological sort algorithm)
5
7
6



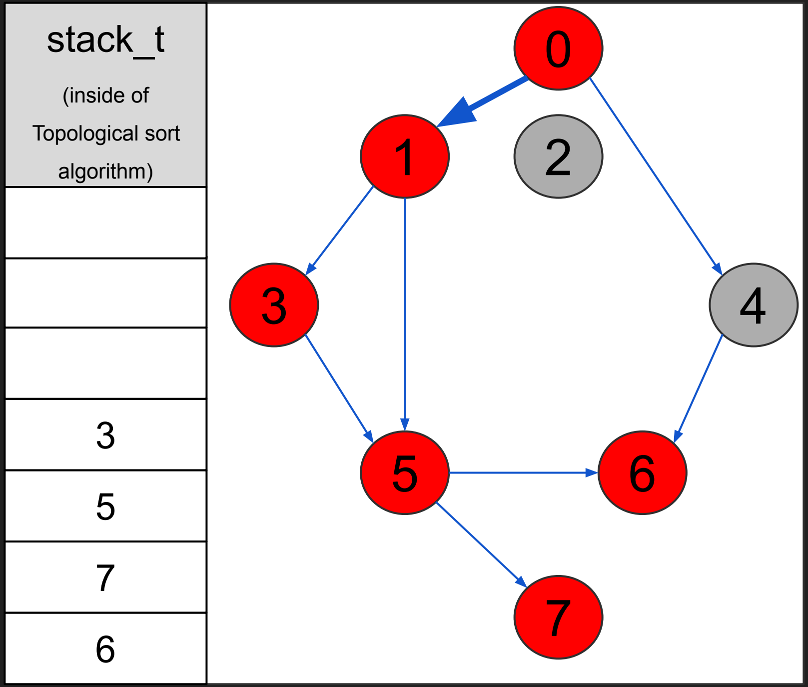
Topological Sort (1/)

- (continued...)
    - Recurse back to 5
        - No where to explore from 5 (i.e. no more nodes to continue our DFS), so we add 5 to the stack
    - Now we recurse back to where we came from, node 3
        - From 3, no where else to recurse from and continue our DFS
            - So add 3 to stack



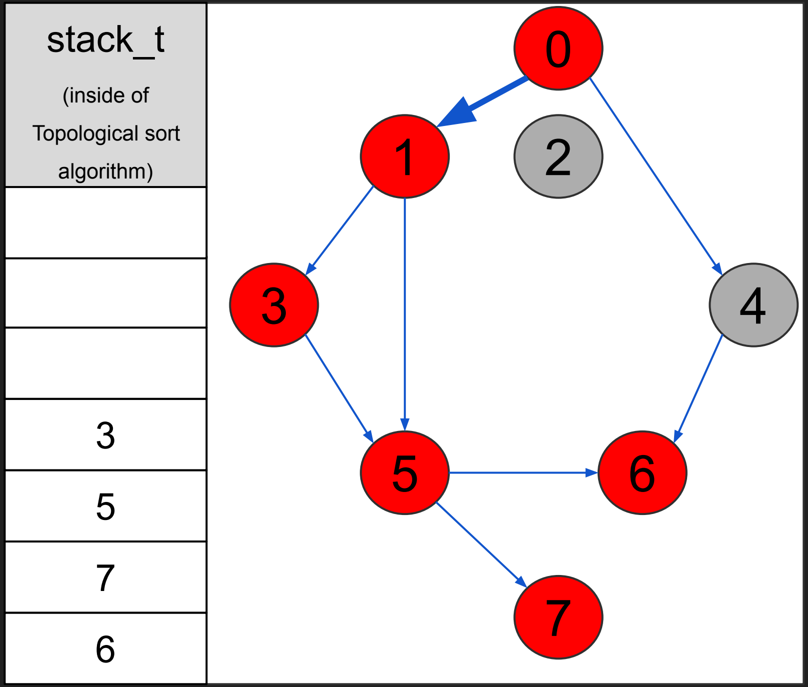
Topological Sort (1/)

- (continued...)
  - Recurse back to 5
    - No where to explore from 5 (i.e. no more nodes to continue our DFS), so we add 5 to the stack
  - Now we recurse back to where we came from, node 3
    - From 3, no where else to recurse from and continue our DFS
      - So add 3 to stack
    - Return to node 1 where we recursed from



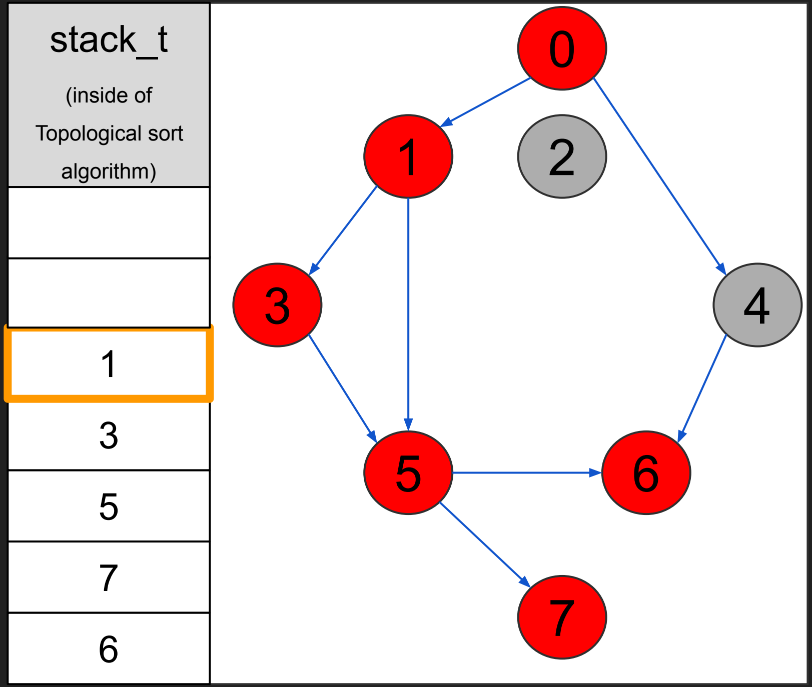
Topological Sort (1/)

- (continued...)
  - No more unexplored nodes for us to continue our DFS from node 1
    - So return back to where we recursed from and add node 1 to stack



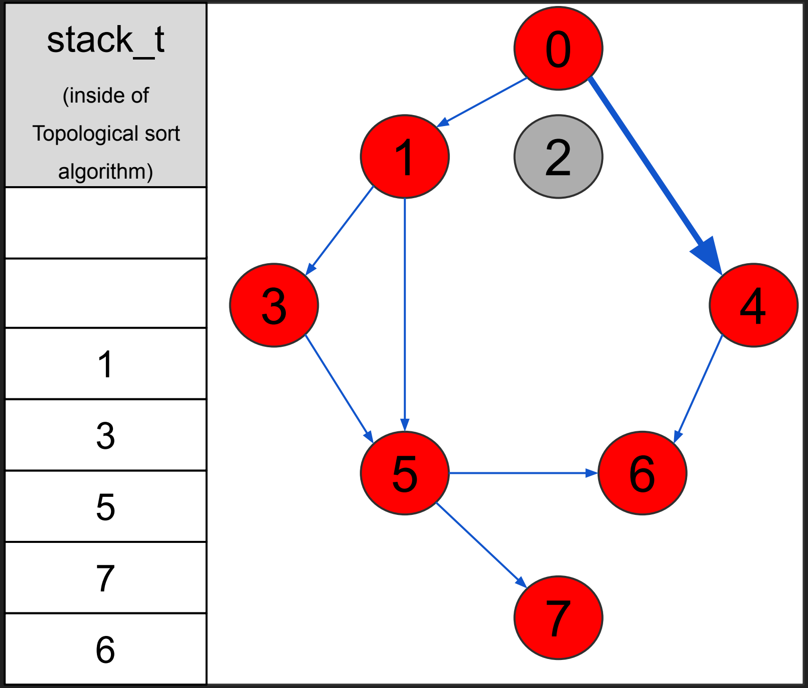
Topological Sort (1/)

- (continued...)
  - No more unexplored nodes for us to continue our DFS from node 1
    - So return back to where we recursed from and add node 1 to stack



Topological Sort (1/)

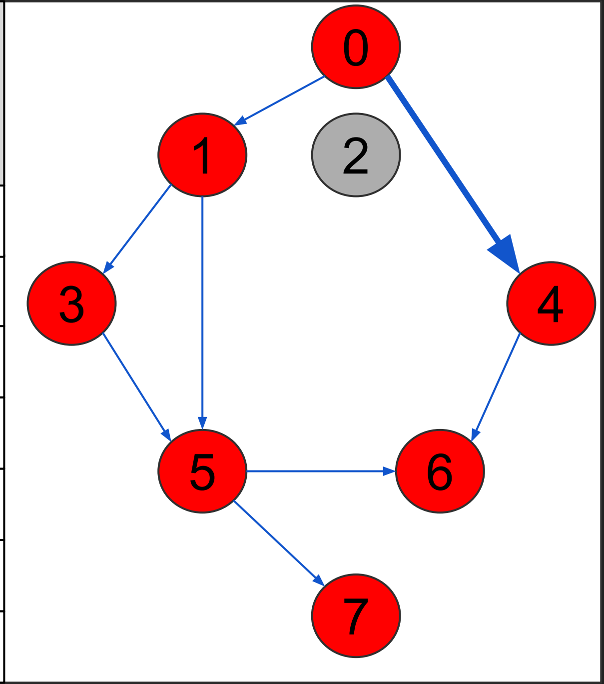
- (continued...)
  - No more unexplored nodes for us to continue our DFS from node 1
    - So return back to where we recursed from and add node 1 to stack
  - From node 0, there is another path to continue DFS to the unexplored node 4.



Topological Sort (1/)

- (continued...)
  - No more unexplored nodes for us to continue our DFS from node 1
    - So return back to where we recursed from and add node 1 to stack
  - From node 0, there is another path to continue DFS to the unexplored node 4.
    - From node 4, no more paths to continue from of unexplored nodes

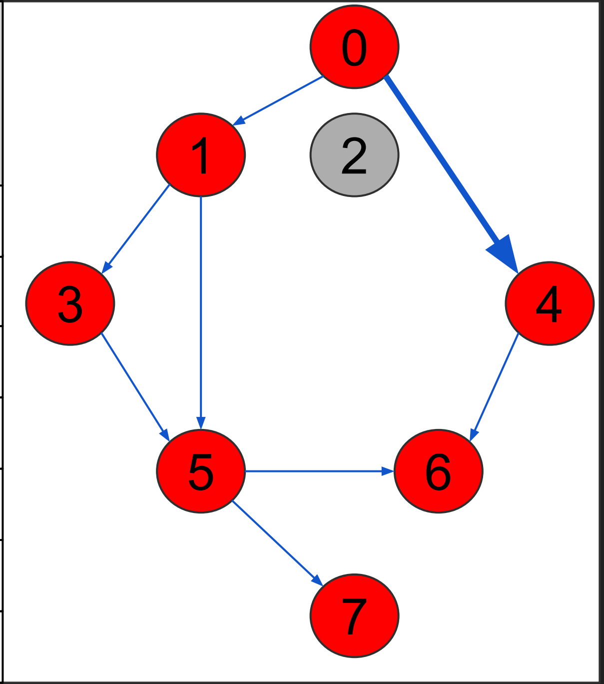
stack_t (inside of Topological sort algorithm)
1
3
5
7
6



Topological Sort (1/)

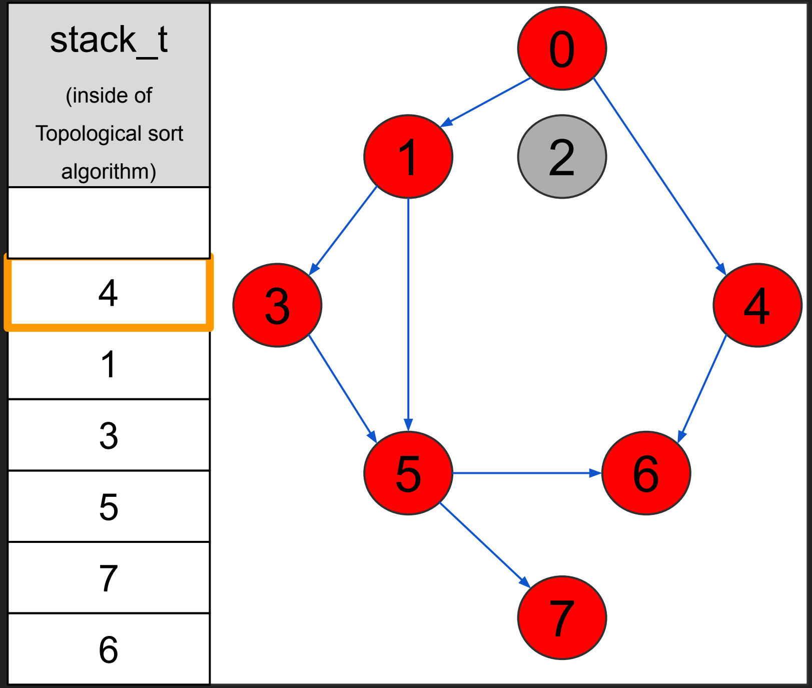
- (continued...)
    - No more unexplored nodes for us to continue our DFS from node 1
        - So return back to where we recursed from and add node 1 to stack
    - From node 0, there is another path to continue DFS to the unexplored node 4.
        - From node 4, no more paths to continue from of unexplored nodes
            - Add 4 to the stack, and return to where we recursed from (node 0)

stack_t (inside of Topological sort algorithm)
1
3
5
7
6



Topological Sort (1/)

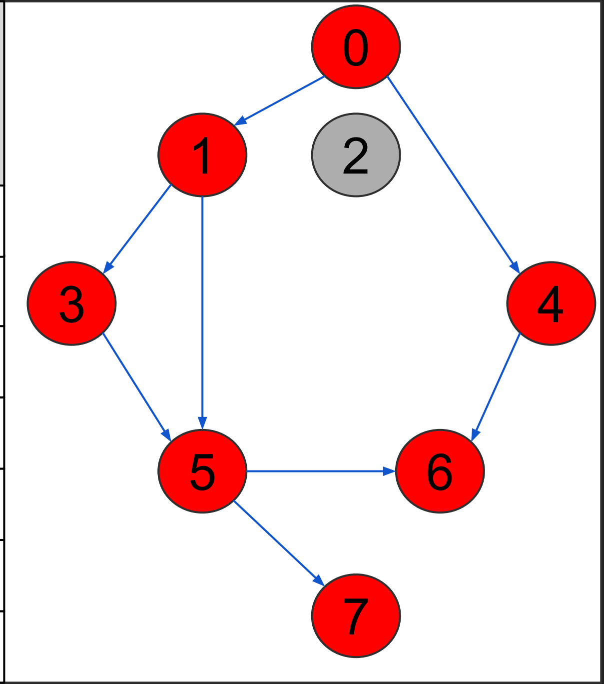
- (continued...)
    - No more unexplored nodes for us to continue our DFS from node 1
        - So return back to where we recursed from and add node 1 to stack
    - From node 0, there is another path to continue DFS to the unexplored node 4.
        - From node 4, no more paths to continue from of unexplored nodes
            - Add 4 to the stack, and return to where we recursed from (node 0)



Topological Sort (1/)

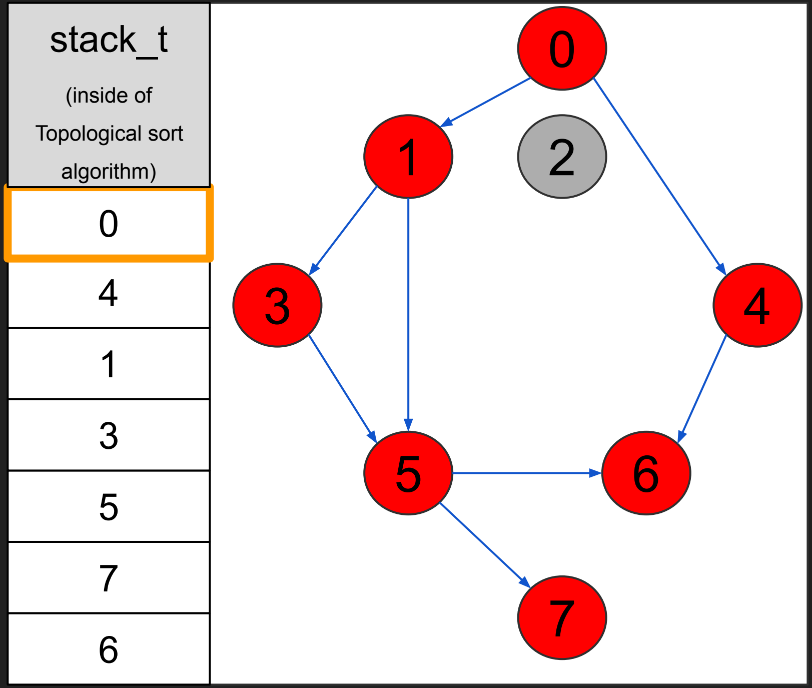
- (continued...)
  - From 0, no more unexplored nodes on any path, so add 0 to our stack

stack_t (inside of Topological sort algorithm)
4
1
3
5
7
6



Topological Sort (1/)

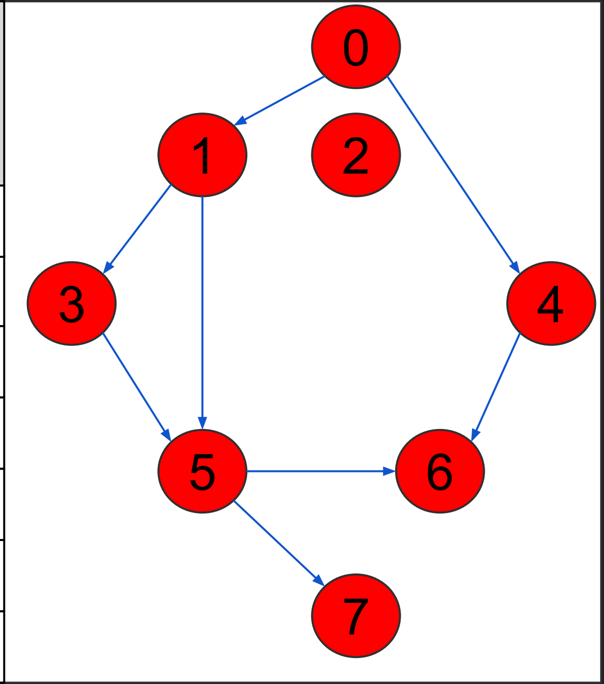
- (continued...)
  - From 0, no more unexplored nodes on any path, so add 0 to our stack



Topological Sort (1/)

- (continued...)
    - From 0, no more unexplored nodes on any path, so add 0 to our stack
    - Are there any more nodes in our ‘Tree’ to explore?
        - We actually have an unconnected tree (i.e. 2 trees), so we perform the same process starting from node ‘2’
        - In this case, it is trivial, and we just add to our stack.

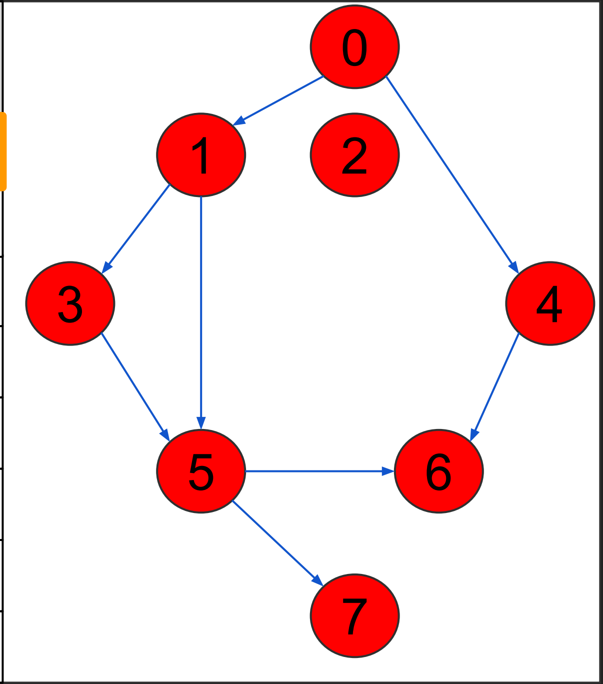
stack_t (inside of Topological sort algorithm)
0
4
1
3
5
7
6



Topological Sort (1/)

- (continued...)
  - From 0, no more unexplored nodes on any path, so add 0 to our stack
  - Are there any more nodes in our ‘Tree’ to explore?
    - We actually have an unconnected tree (i.e. 2 trees), so we perform the same process starting from node ‘2’
    - In this case, it is trivial, and we just add to our stack.

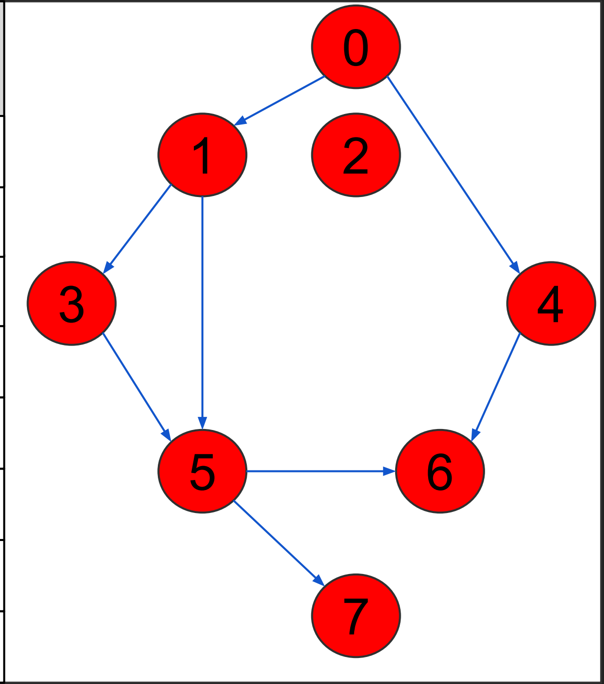
stack_t (inside of
2
0
4
1
3
5
7
6



Topological Sort (1/)

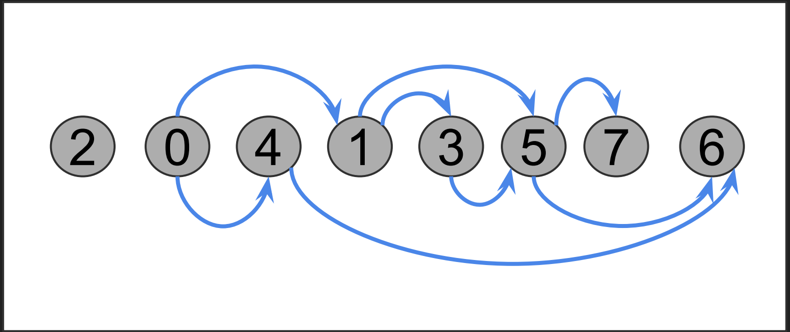
- Here is our topological sorting,
  - We simply pop off the nodes of our stack, and that is an acceptable ordering of courses.
    - 2,0,4,1,3,5,7,6

stack_t (inside of
2
0
4
1
3
5
7
6

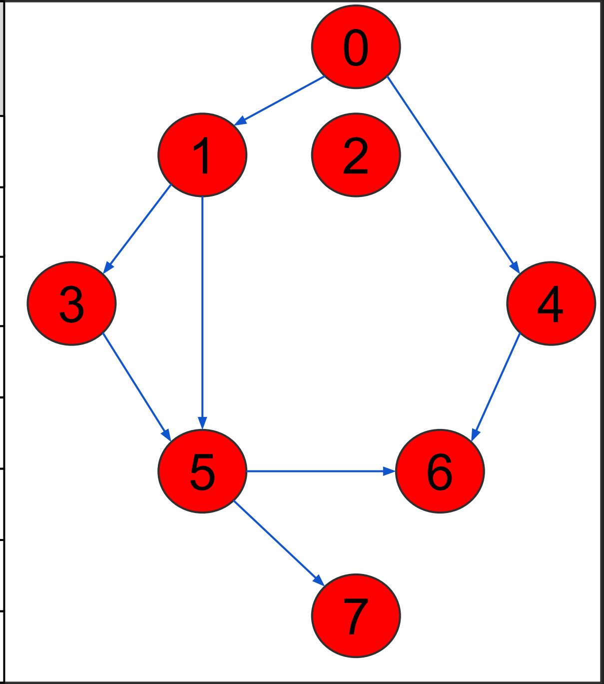


Topological Sort (1/)

- Here is our topological sorting,
  - We simply pop off the nodes of our stack, and that is an acceptable ordering of courses.
    - 2,0,4,1,3,5,7,6
  - Here's another way to visualize the tree below
    - Pick any class (e.g. CS 5) -- Any incoming arrows are prerequisites, and you can follow the links backwards to see all the courses you need to take



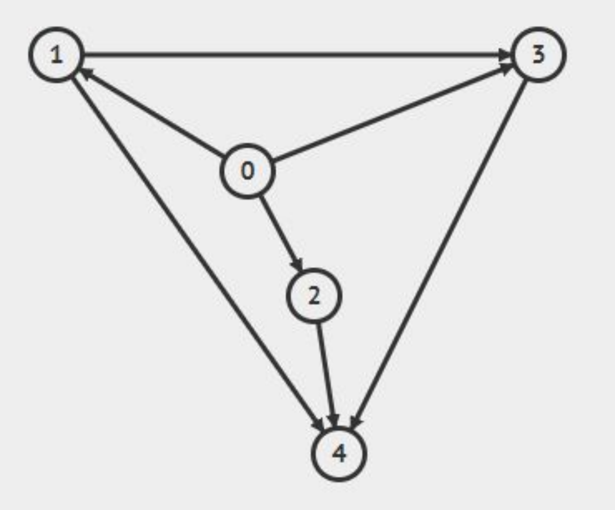
stack_t (inside of
2
0
4
1
3
5
7
6



Topological Sort Example

- Start the DFS(u) from the root.
  - Then visit each neighbor that is unvisited
    - We then start another DFS(v)
  - We continue our DFS(u) recursively
- As we are performing the DFS we are pushing the nodes into a ‘stack’.
  - When we pop the nodes, they come out in reverse order, giving us an ordered, topological sort.
- Run example: https://visualgo.net/en/dfsbfs

```
for each unvisited vertex u
  DFS(u)
    for each neighbor v of u
      if v is unvisited, DFS(v)
    else skip v;
    finish DFS(u), add u to the back of list
reverse list // ch4_01_dfs.cpp/java, ch4, CP3
```



Topological Sort - Complexity Analysis (1/2)

- Question to the audience:
  - Topological sort takes O(??????) time

```
for each unvisited vertex u
  DFS(u)
    for each neighbor v of u
      if v is unvisited, DFS(v)
    else skip v;
    finish DFS(u), add u to the back of list
reverse list // ch4_01_dfs.cpp/java, ch4, CP3
```

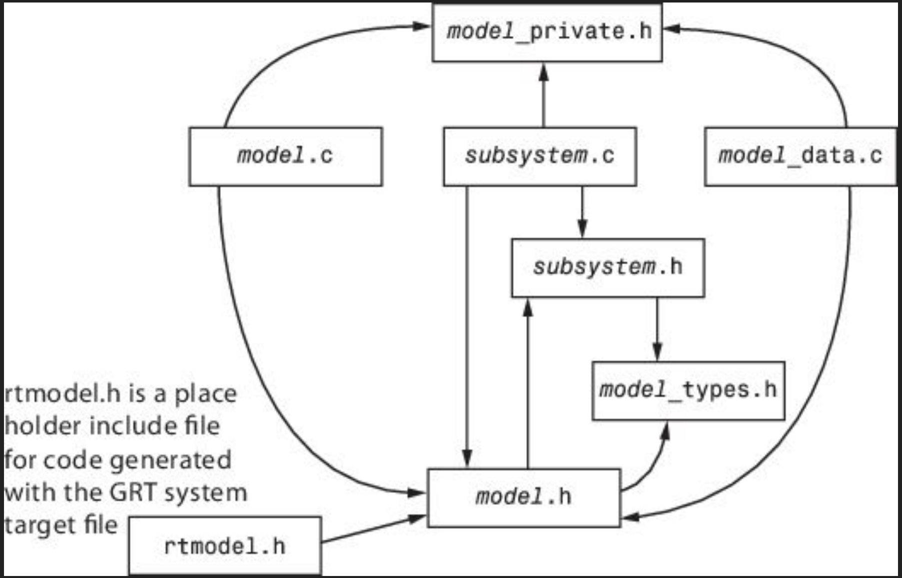
Topological Sort - Complexity Analysis (2/2)

- Question to the audience:
  - Topological sort takes $O(|V|+|E|)$ time
    - Same as DFS actually!
    - The bulk of our work is in the loop
      - We are looping over our edges (neighbor 'v' of u) and performing a DFS on unvisited nodes.

```
for each unvisited vertex u
  DFS(u)
    for each neighbor v of u
      if v is unvisited, DFS(v)
    else skip v;
    finish DFS(u), add u to the back of list
reverse list // ch4_01_dfs.cpp/java, ch4, CP3
```

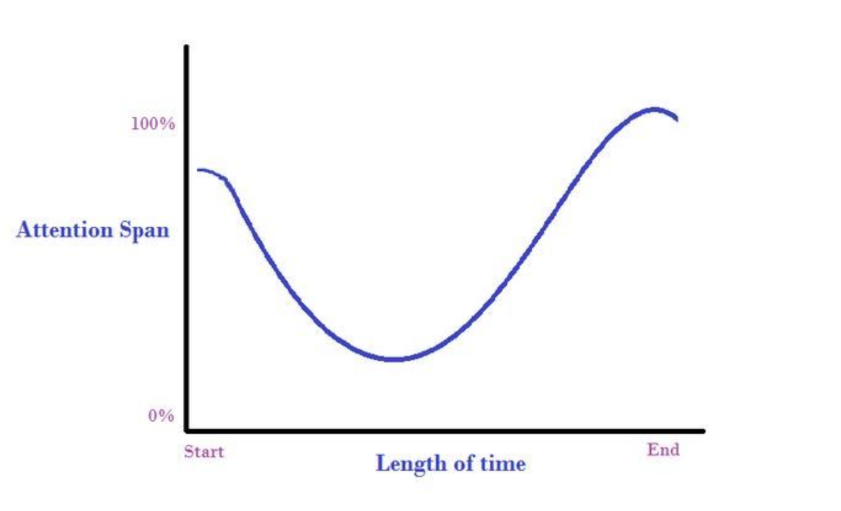
Build Systems and Topological Sort

- When using a ‘Makefile’ and building several .c files and compiling them into .o files, we often have ‘dependencies’ on which are needed before we can link together and build the final executable
- Topological sort can be one way to ‘schedule’ or ‘order’ which files to compile first.



Short 5 minute break

- 3 hours is a long time.
- I will try to never lecture for more than half of that time without some sort of ‘break’ or transition to an in-class activity/lab.
- Use this time to stretch, check your phones, eat/drink something, etc.



Moving on from Trees

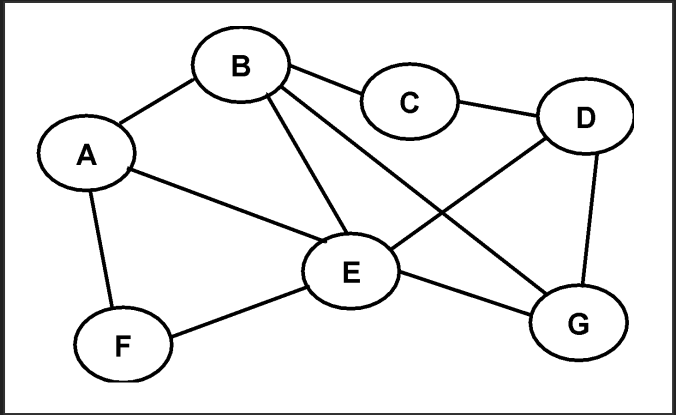
(i.e. “a special instance of a graph”)

Moving to: Introducing Graphs

A new abstract data type made up of nodes (vertices) and edges

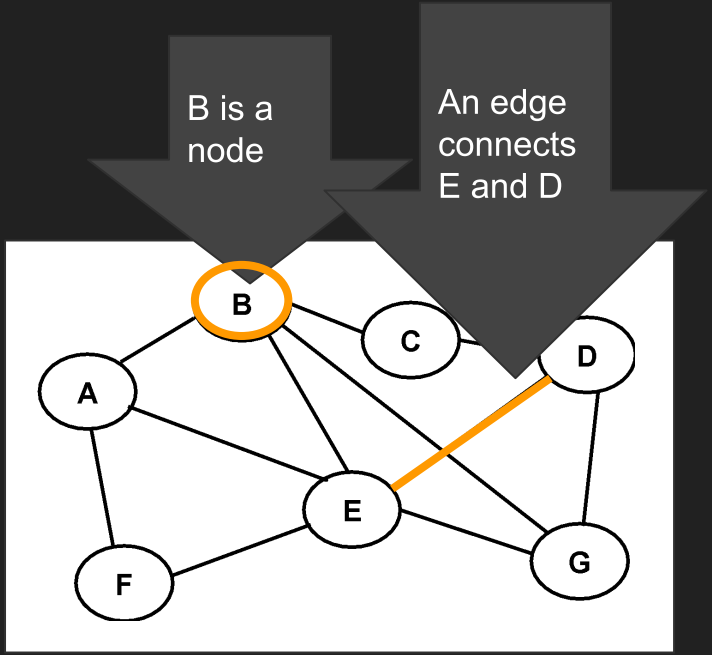
Graphs - Notation (1/3)

- Graphs also consist of ‘nodes’ and ‘edges’
    - Just like trees!
    - (Note: sometimes we refer to a ‘node’ as a ‘vertex’ and I will use them interchangeably)



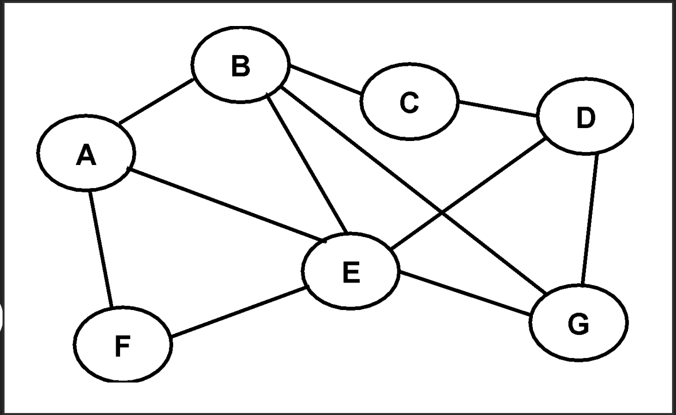
Graphs - Notation (2/3)

- Graphs also consist of ‘nodes’ and ‘edges’
  - Just like trees!
  - (Note: sometimes we refer to a ‘node’ as a ‘vertex’ and I will use them interchangeably)



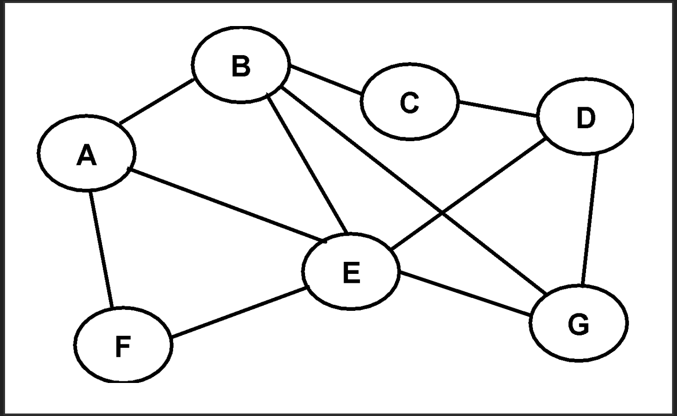
Graphs - Notation (3/3)

- A graph is formally labeled:
  - `G(V,E)`
  - Read as: "Graph G consists of vertices 'V' and edges 'E'"
    - `V` = set of Vertices (a.k.a. set of nodes)
    - `E` = edges between a pair of nodes
    - The size of each parameters is labeled with vertical bars:
      - `n = |V|` (e.g. 7 vertices total)
      - `m = |E|` (e.g. 11 edges total)



Graphs

- A graph is a data structure models some sort of network
  - In our 'binary tree' example our network was *like* a 'family tree' as a analogy
- However, graphs can be modeled to store much more generic relationships



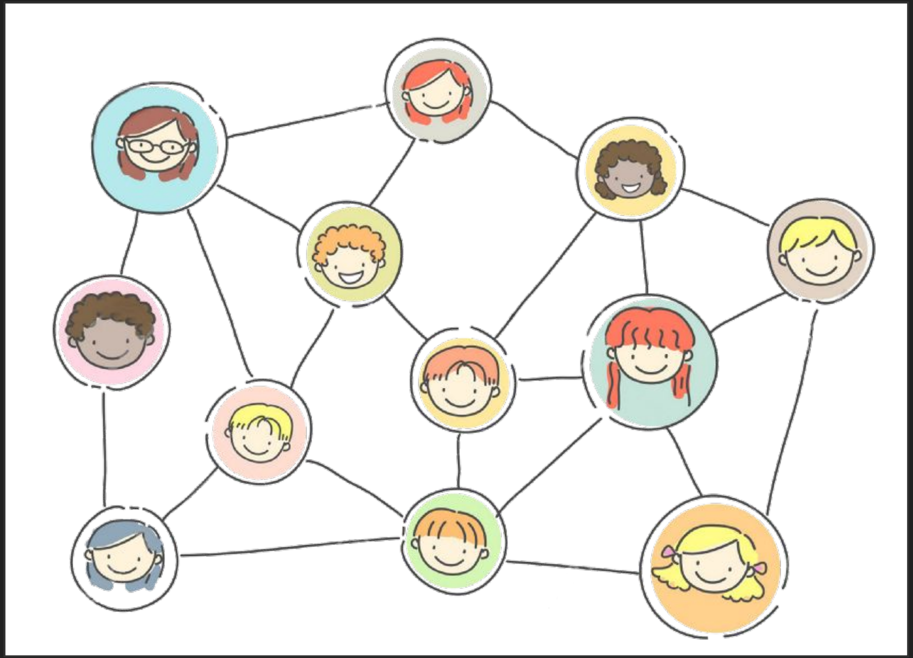
Where are Graphs Used? (1/4)

- Here are some real world examples for how we model relationships in a graphs

Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

Where are Graphs Used? (2/4)

- Social Networks
  - Modeling 'best friends' or some other connection to a person or business for example
- Encoding
  - Nodes: People
  - Edges: Friendship

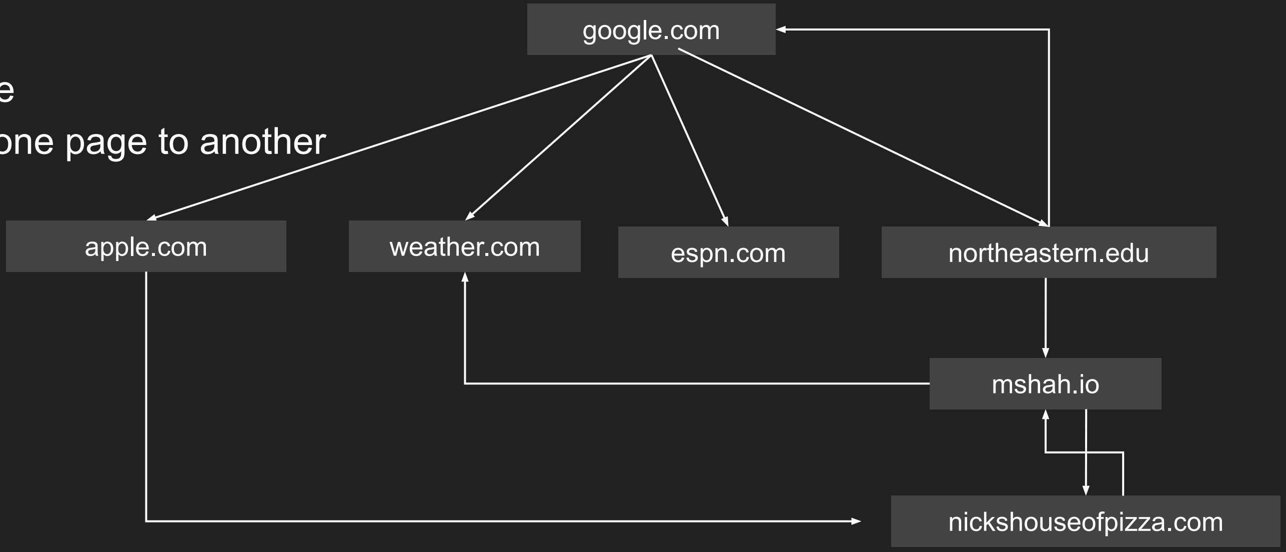


Where are Graphs Used? (3/4)

- Protein Networks
  - Model the physical contact between proteins in our cells.
- Encoding
  - Nodes: Proteins
  - Edges: Represent some interaction between proteins

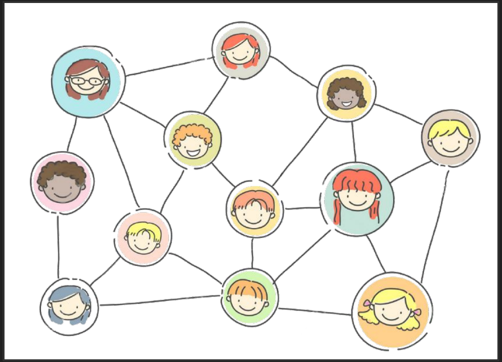
Where are Graphs Used? (4/4)

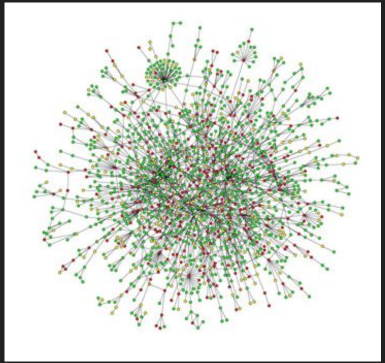
- World Wide Web
  - Collection of resources on the web
- Encoding
  - Node: a web page
  - Edge: a link from one page to another



Takeaway with Graphs

- Take any problem you have in the real world.
    - If you can model the relationships between the entities with nodes and edges, then you can solve it as a graph problem.
        - That means you get access to all of the wonderful graph algorithms we will talk about.

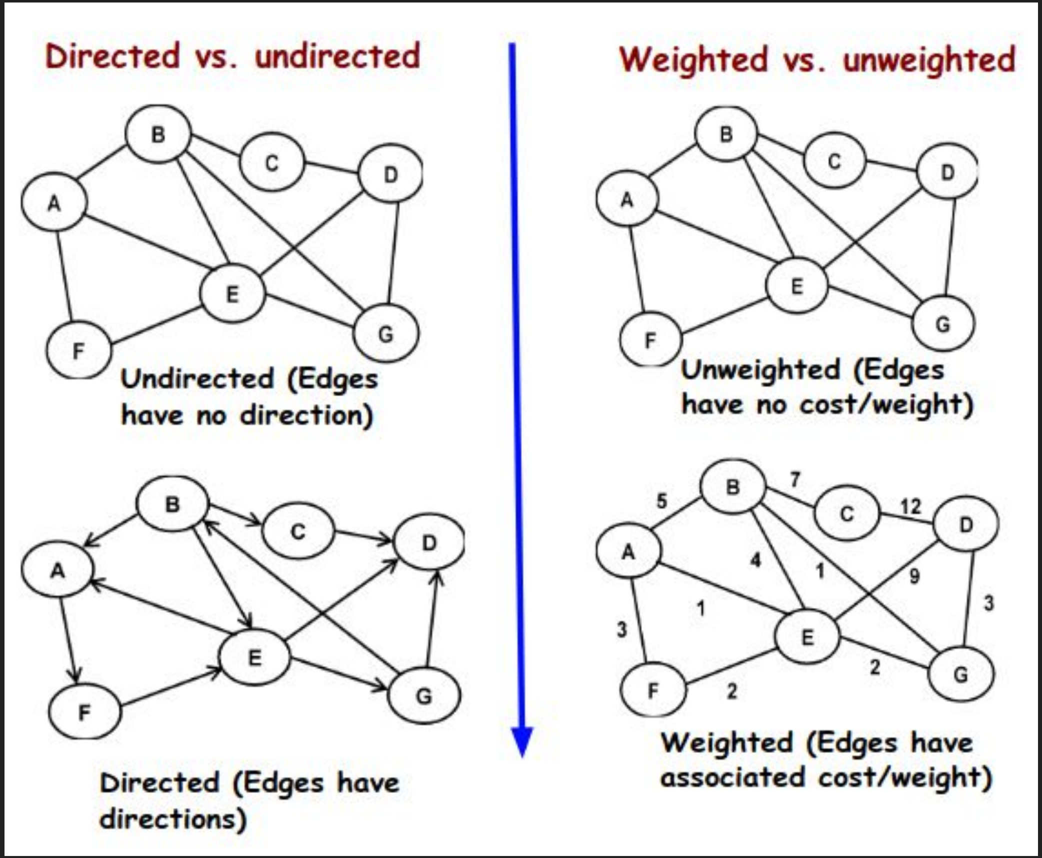




More Graph Notation

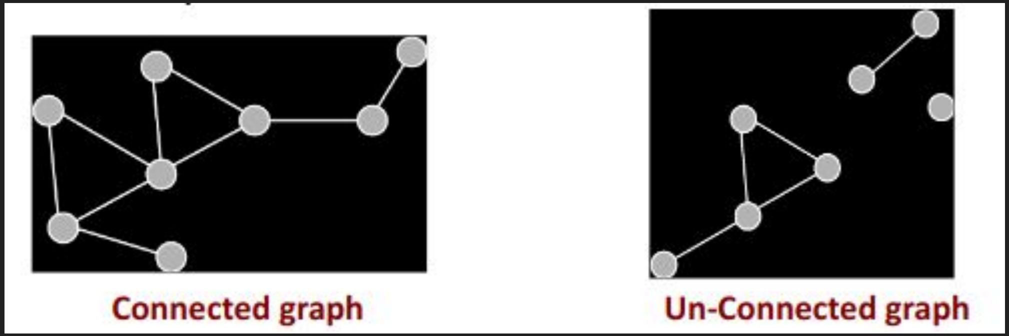
Different Types Graphs

- There are several ways to characterize graphs depending on what we are trying to model
- The image on the right shows ‘directed vs undirected’ and ‘weighted vs unweighted’ graphs
    - (If it is helpful, you can think of unweighted graphs as each edge having the same weight)



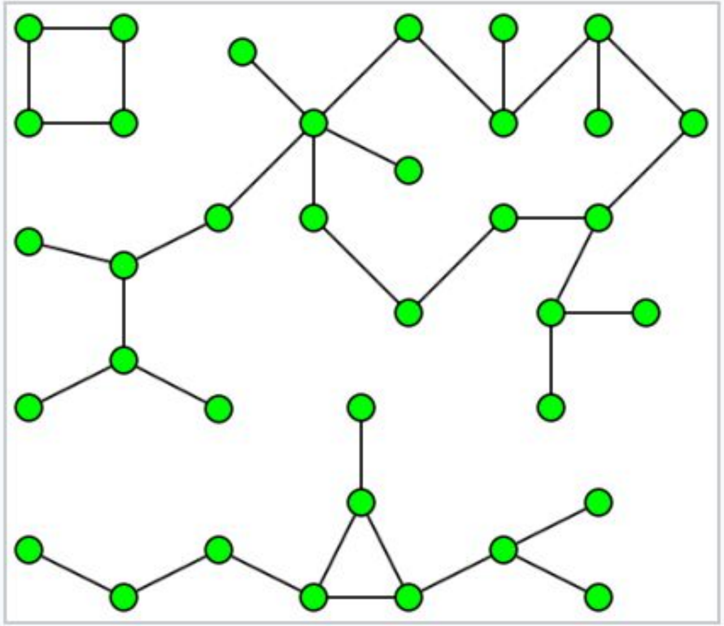
Connected vs Unconnected Graph

- A graph which has a *path* from every pair of nodes is known as a connected graph (see left).
  - That is, a path exists, and every node is reachable.
- In this course we will primarily work with ‘connected graphs’
  - (See the unweighted connected graph to the left)
  - A real world example however:
    - Your personal ‘social network’ and mine maybe ‘unconnected’ (see right un-connected graph)



Connected Components [wiki]

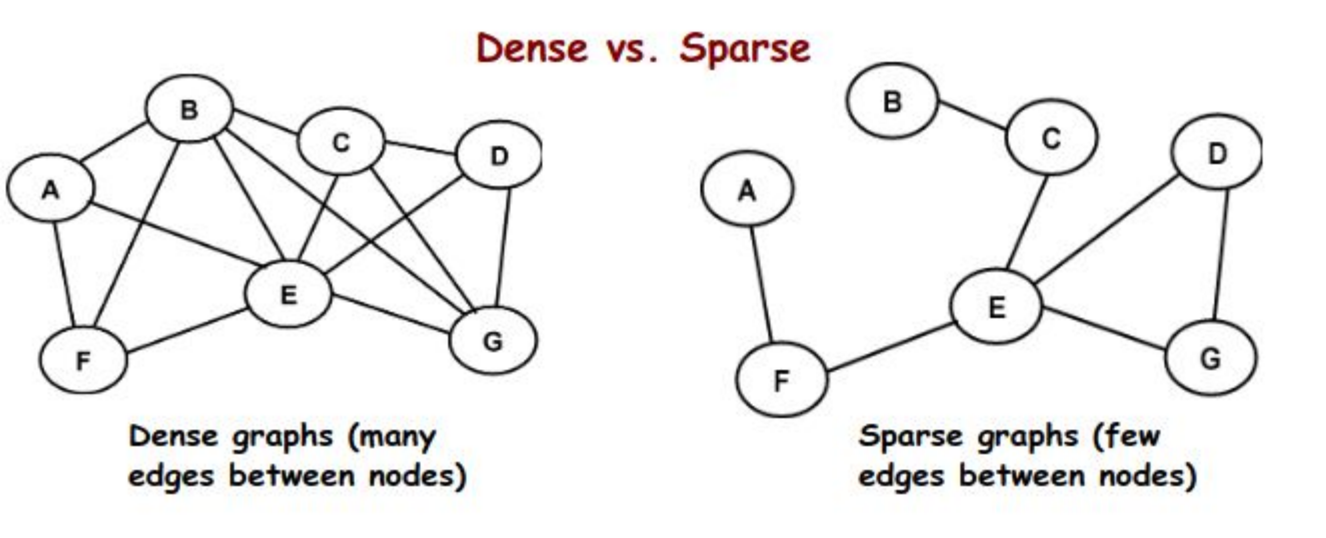
- Here is an ‘unconnected’ graph.
- There are three individual connected components however.
- A connected component, is simply a node which has some *path* to every other another node.



A graph with three connected components.

Graphs Density

- Depending on how ‘connected’ nodes are in a graph, we may also describe them as ‘dense’ or ‘sparse’’
  - See [Dense graph]()



Data Structures to Support Graphs

How to represent graph structure in code

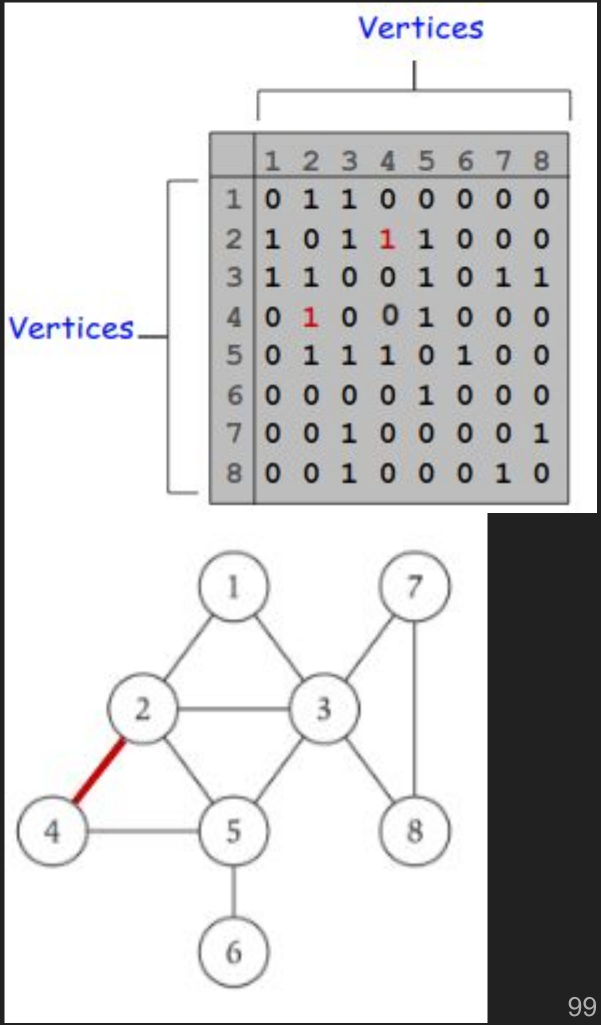
Two Key Data Structures Supporting Graphs

1. Adjacency Matrix
    - An array based data structure
2. Adjacency List
    - A linked list based data structure

- Both representations we can use for directed, undirected, or weighted, and unweighted graphs
    - There are some trade-offs however!

#1 Adjacency Matrix (1/2)

- An adjacency Matrix is a ‘V’ by ‘V’ matrix
    - i.e. There are 8 vertices (or nodes) in the graph to the right
- Thus, I need `O(|V|*|V|)` space to represent the graph
- For an unweighted graph
    - A node is connected if `A[i,j] = 1`
    - A node is not connected if `A[i,j] = 0`
- For a positively weighted graph
    - A node is connected if `A[i,j] > 0`
    - A node is not connected if `A[i,j] = 0`

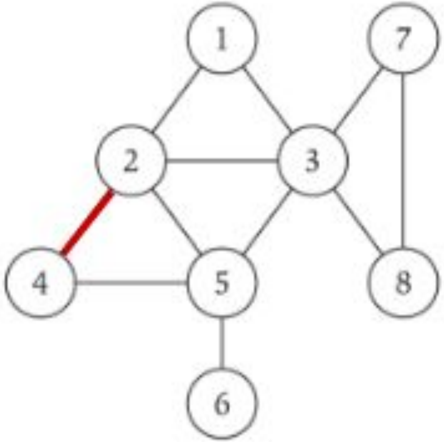


#1 Adjacency Matrix (2/2)

- Examples
    - Node 2 is connected to node 4
    - Node 4 is connected to node 2
- Typically the ‘rows’ represent the node, and the columns are what we are connecting to
- Note: In an undirected graph, there is symmetry, but that may not be true in a directed graph
    - e.g. In our undirected graph (which you can think of like a two-way road):
        - 2 is connected to 4, and 4 is connected to 2.

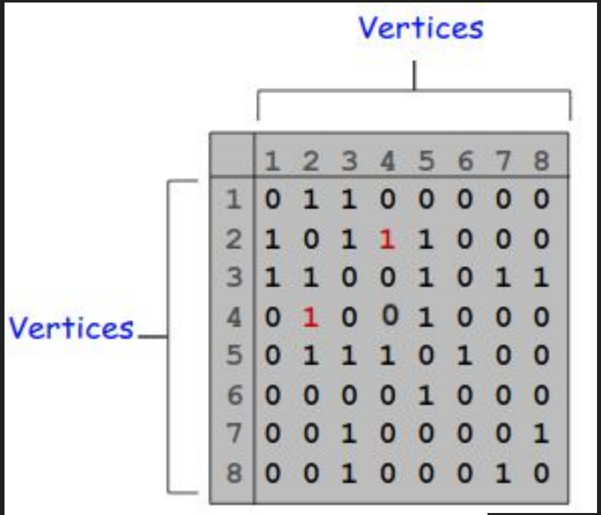
Vertices (columns) / Vertices (rows)

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

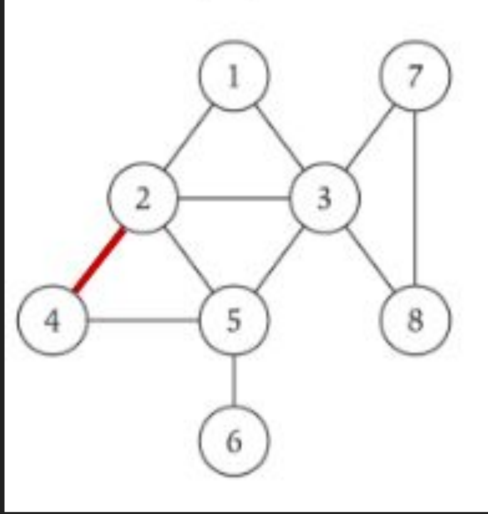


#1 Adjacency Matrix - Degree (1/4)

- The ‘degree’ of a node is how many other nodes a node is connected to in this undirected graph
    - (next slide for example)

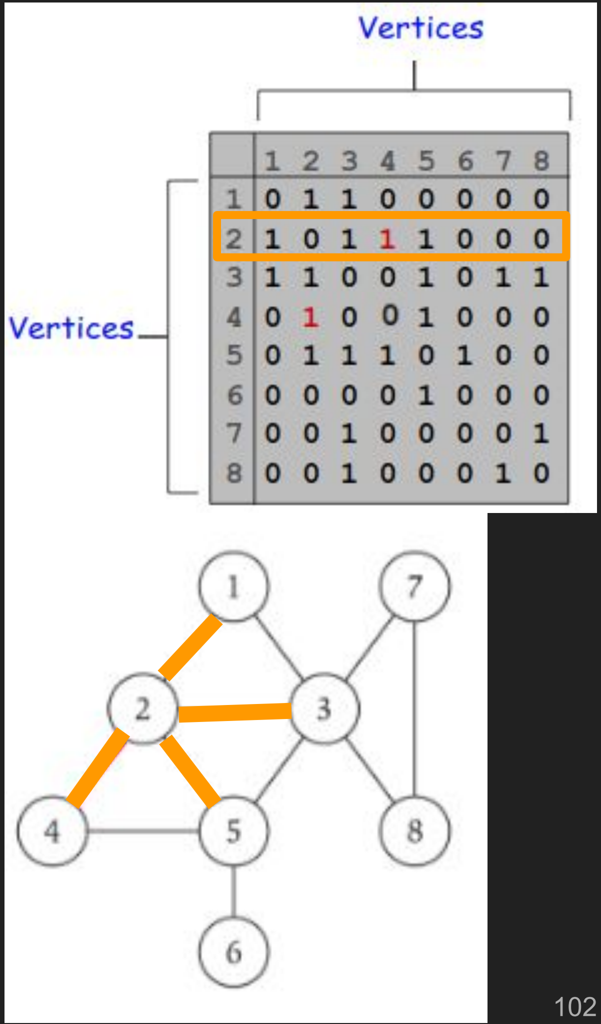


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0



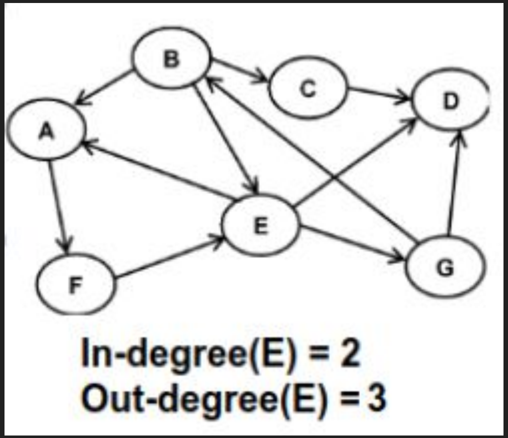
#1 Adjacency Matrix - Degree (2/4)

- The ‘degree’ of a node is how many other nodes a node is connected to in this undirected graph
  - 2 for example is connected to 1,3,4,and 5
  - 2 thus has a degree=4
  - (Note: One trick is that by summing the non-zero values in node ‘2’s row, I see the degree of that node!)



#1 Adjacency Matrix - Degree (3/4)

- The ‘degree’ of a node is how many other nodes one is connected to in this undirected graph
  - 2 for example is connected to 1,3,4,and 5
  - 2 thus has a degree=4
  - One trick is that by summing the non-zero values in node ‘2’s row, I see the degree of that node!
- Note: If our graph is directed, the in-degree is incoming edges, and out-degree is edges leaving
  - Looking at ‘E’ I see:
    - 2 arrows ‘in’
    - 3 arrows ‘out’



#1 Adjacency Matrix - Degree (4/4)

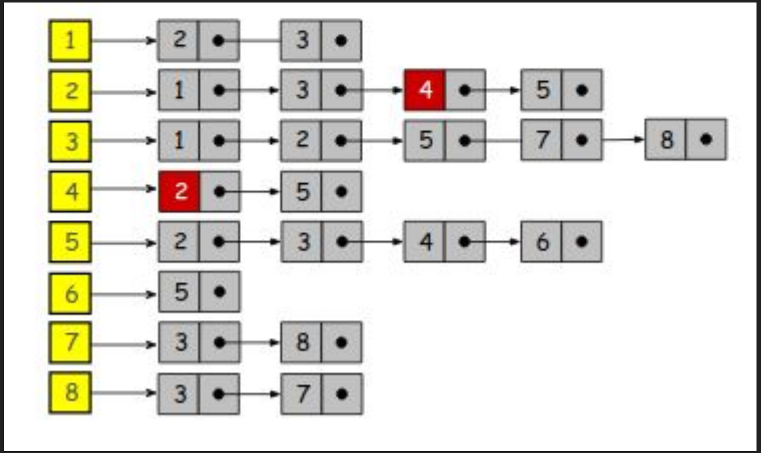
- The ‘degree’ of a node is how many other nodes one is conne
  - 2 for exar
  - 2 thus has
  - One trick ‘2’s row, l
- Note: If our incoming ed
  - Looking a
    - 2 arrows ‘in’
    - 3 arrows ‘out’

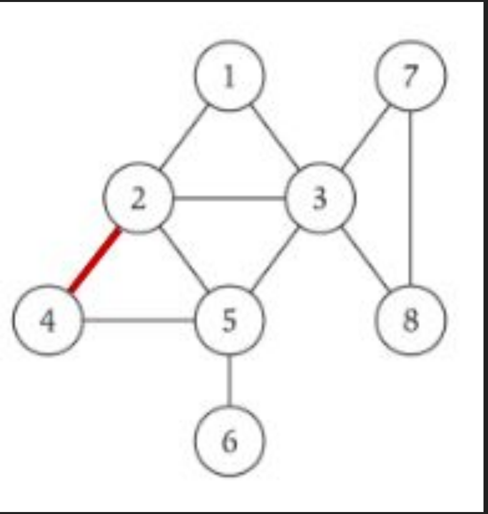
An Adjacency Matrix seems great if we know exactly how many nodes we have (as do all array-based structures)

What if we do not? (next slide!)

#2 Adjacency List (1/5)

- An adjacency list is an array of lists
  - (i.e. each index in our array a pointer to a linked list)
  - (This is exactly like our chained hashmap!)
  - (Note: It could also be a linked list of linked lists--same idea, but the key is each entry in our list has a linked list from it indicating the edges of that node))

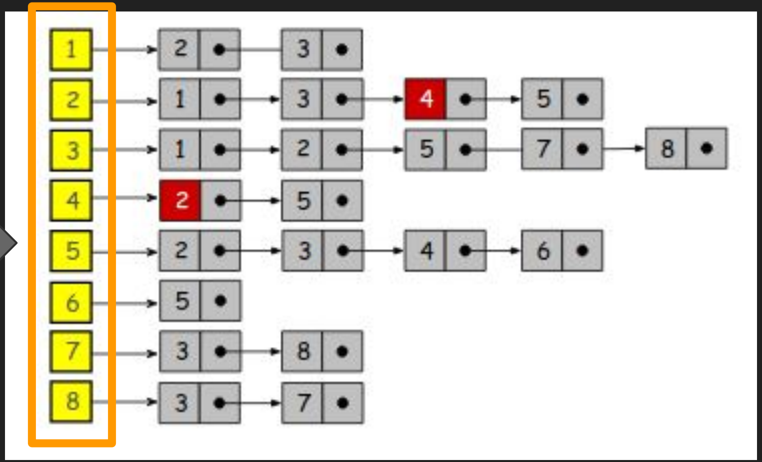


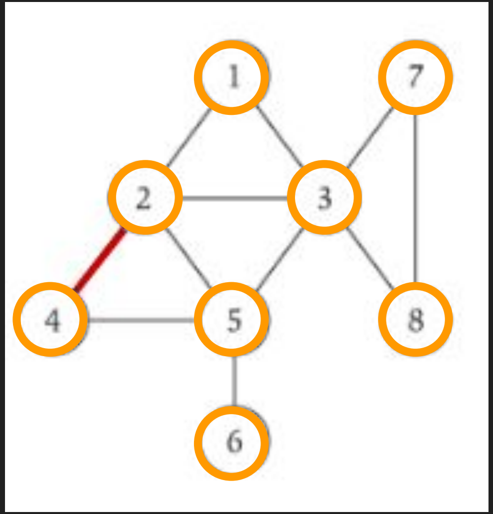


#2 Adjacency List (2/5)

We have our ‘8’ nodes labeled 1,2,3,4,5,6,7,8

- An adjacency lis
  - (i.e. each index in our array a pointer to a link
    list)

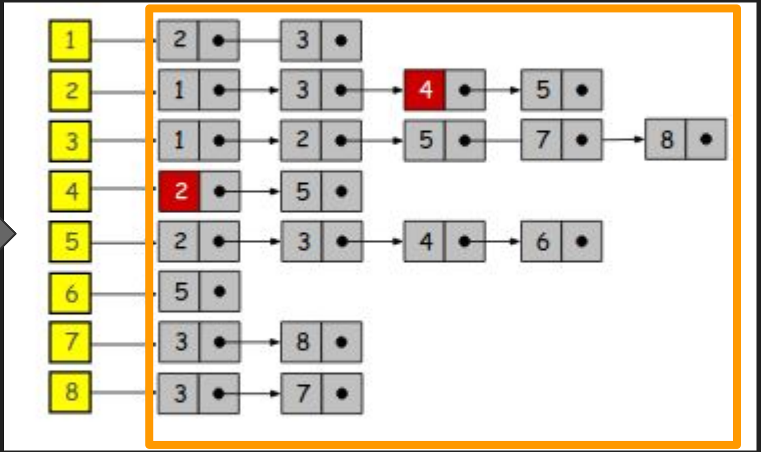


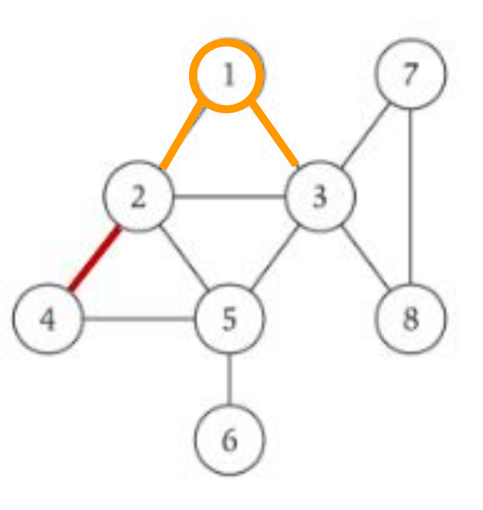


#2 Adjacency List (3/5)

- An adjacency lis
  - (i.e. each index in our array a pointer to a link
    list)

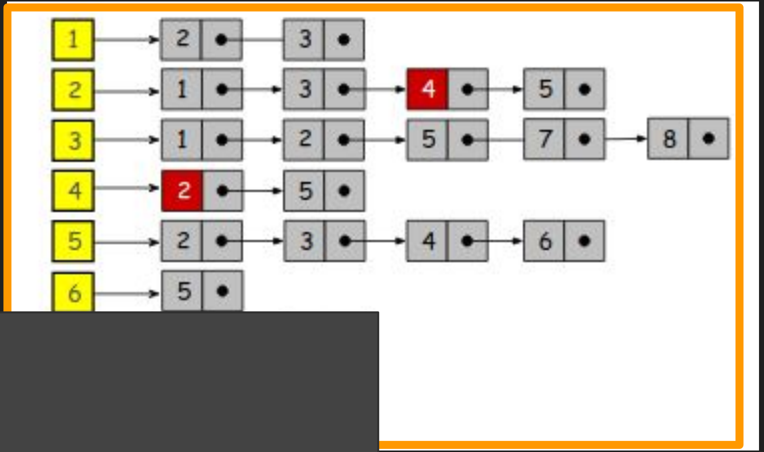
We have all of the edges
e.g. Node ‘1’ has an edge to 2 and 3





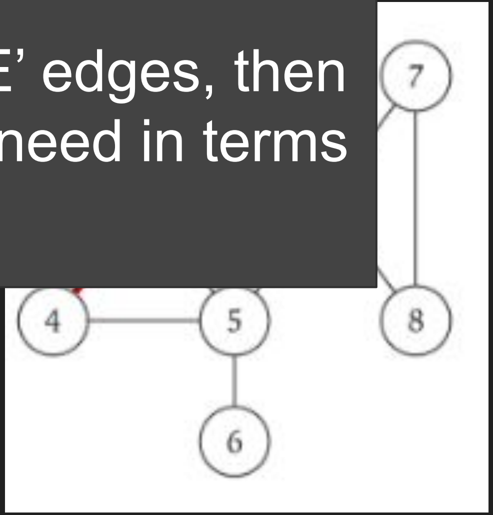
#2 Adjacency List (4/5)

- An adjacency list is an array of lists
  - (i.e. each
    list)



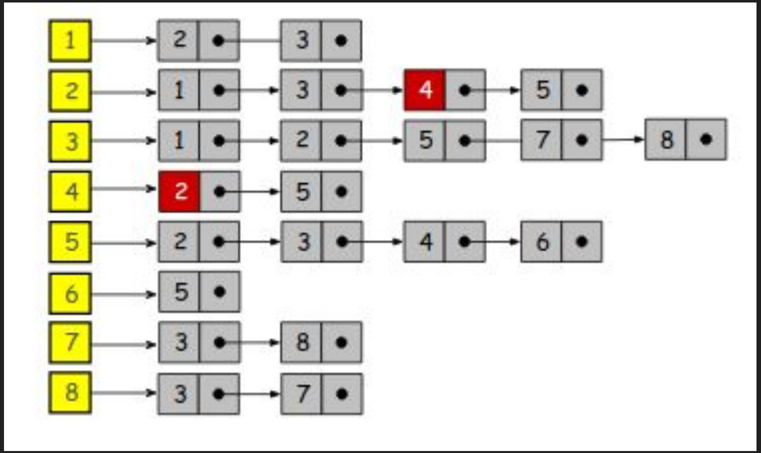
Question to audience:

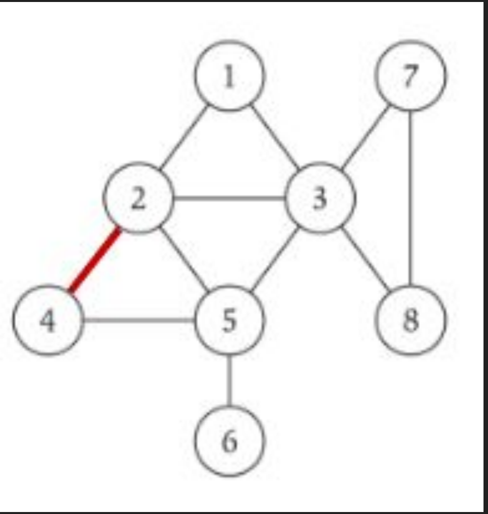
If we have ‘V’ nodes and ‘E’ edges, then how much memory do we need in terms of Big-O?



#2 Adjacency List (5/5)

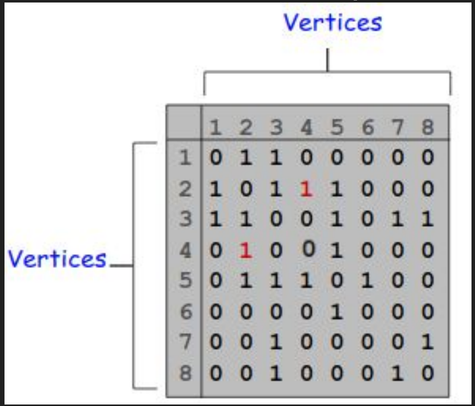
- An adjacency list is an array of lists
  - (i.e. each index in our array a pointer to a linked list)
- Answer:
  - We only need `O(|E|+|V|)` space to represent our graph
  - The number of edges that are connected, and the number of nodes.

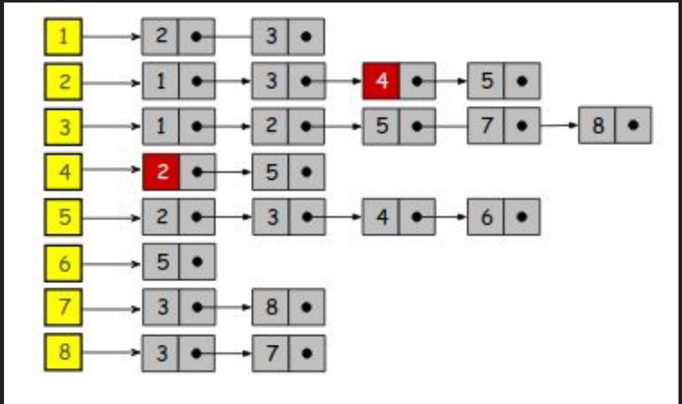




#1 Adjacency Matrix vs #2 Adjacency List (1/2)

- With an adjacency matrix we have random access
  - i.e. we can can check if A[i,j]==1 and find out if we are connected in O(1)
  - But we are not very space efficient because we have to model every relationship
- With an adjacency list we do not have random access
  - To find who we are connected to (‘neighbors’ or ‘adjacent nodes’) we need to iterate through our linked list to see if there is a connection in our graph
  - However, we are more space efficienct

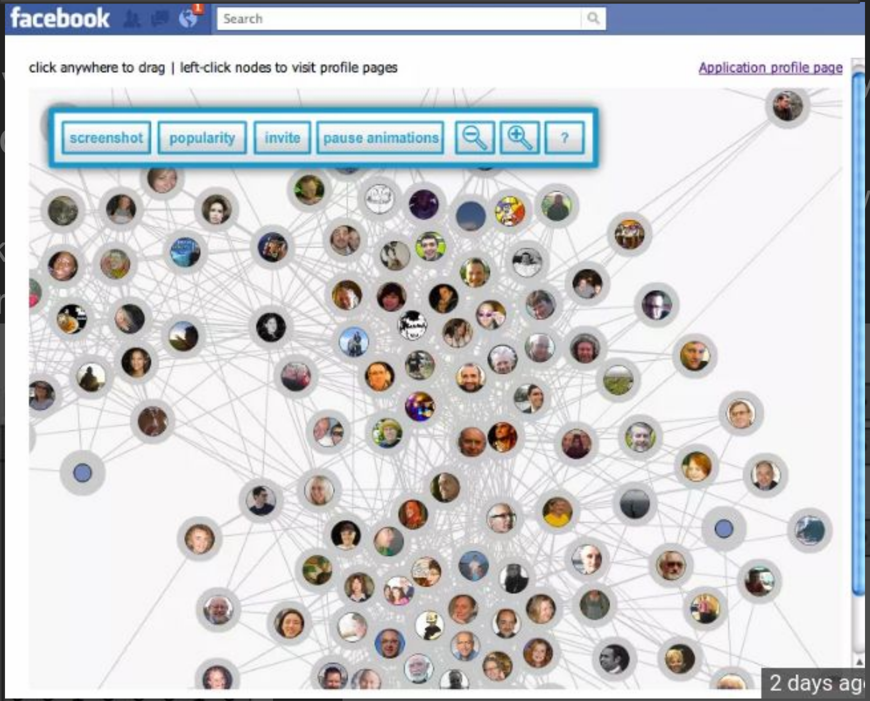


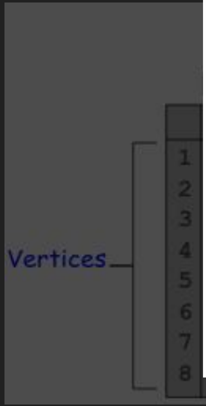


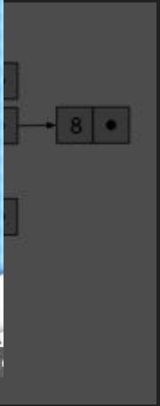
#1 Adjacency Matrix vs #2 Adjacency List (2/2)

- With an adjacen
  - i.e. we can can … O(1)
  - But we are not … y relationship
- With an adjacenc
  - To find who we … ve need to iterate through our link
  - However, we ar

These tradeoffs matter at scale!





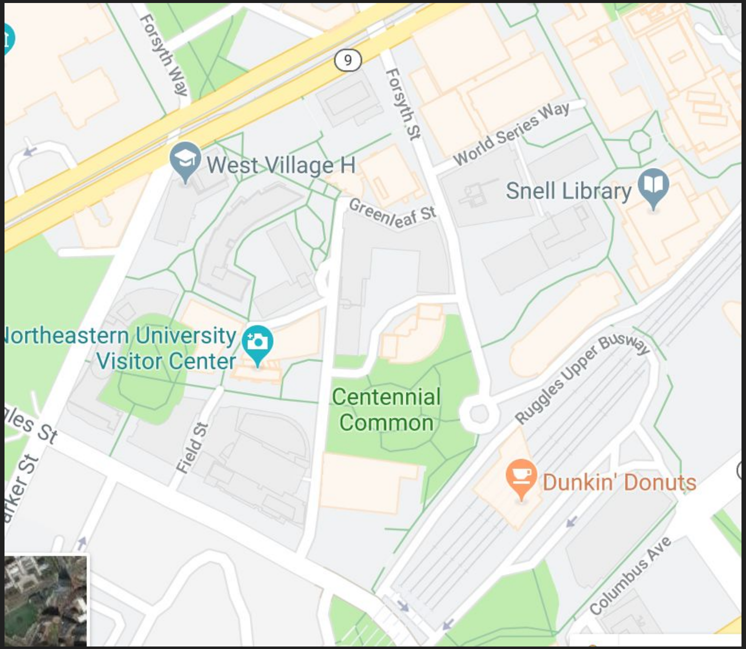


Searching Graphs

Breadth-First Search, Dijkstra’s

Motivating Problem

- You are hired to work on the Google maps team
- They want you to help search graphs
  - (i.e. a map has roads as edges and locations as nodes)
- They want you to find routes that find the shortest amount of ‘transfers’ and ‘shortest path’ to different destinations.
- We will learn some tools to do so!



Find a Job - Google Careers
https://careers.google.com/jobs/
Search, find and apply to **job** opportunities at **Google**. Bring your insight, imagination and healthy ...
Find your next **job** at **Google**. What do you want to do? Next.

Breadth-First Search

We have done ‘depth’ first search, now a new strategy!

Breadth-First Search (1/2)

- Motivating Idea: Search those nodes closest to you
  - e.g. If you need to borrow something, ask your friends.
  - If they don't know, then you ask friends of friends,
  - Then ask "friends of friends of friends"
  - etc.

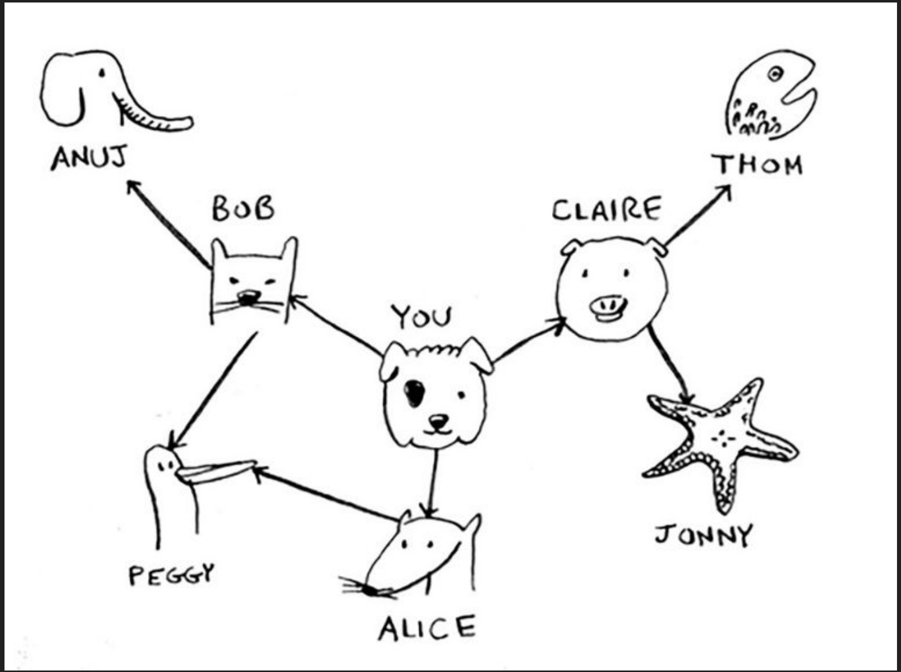


Illustration from Grokking Algorithms

Breadth-First Search (2/2)

- So you can see this ordering that appears
  - You ask Alice, Bob, and Claire,
    - Then since you asked Alice first (in your 1st degree), alice went and asked Peggy.
    - And you proceed to ask Anuj, Thom, and Johnny

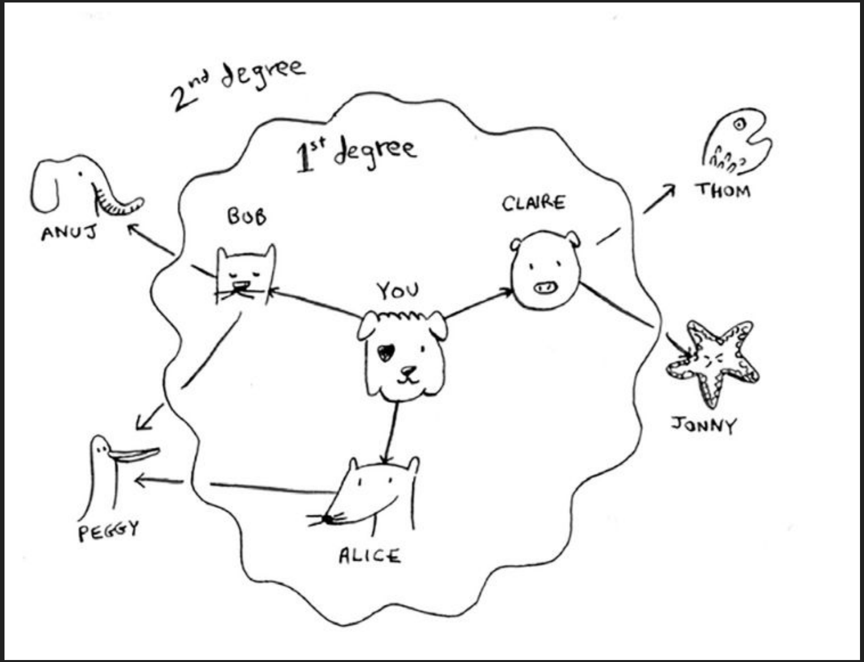
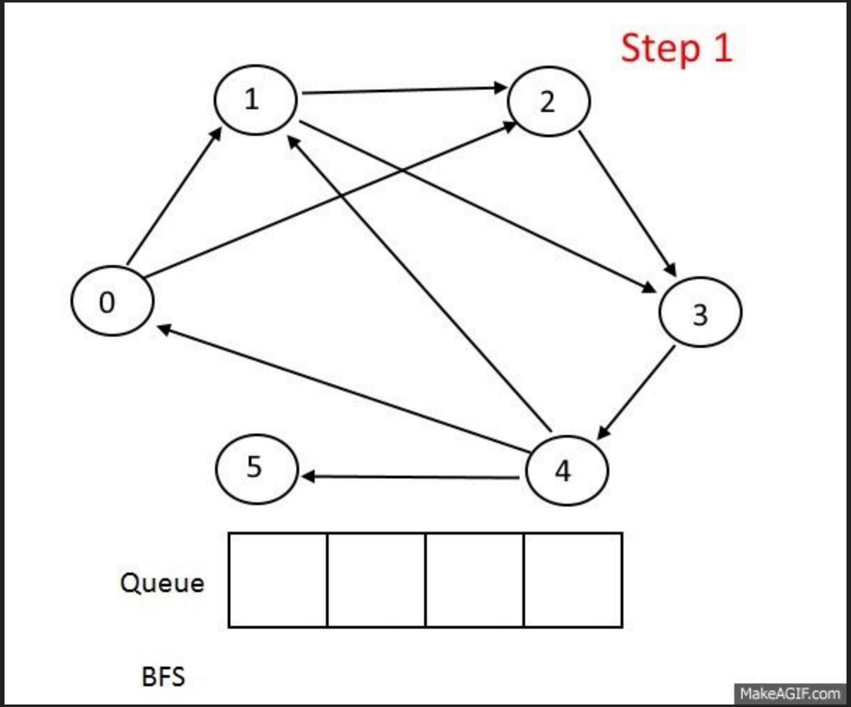


Illustration from Grokking Algorithms

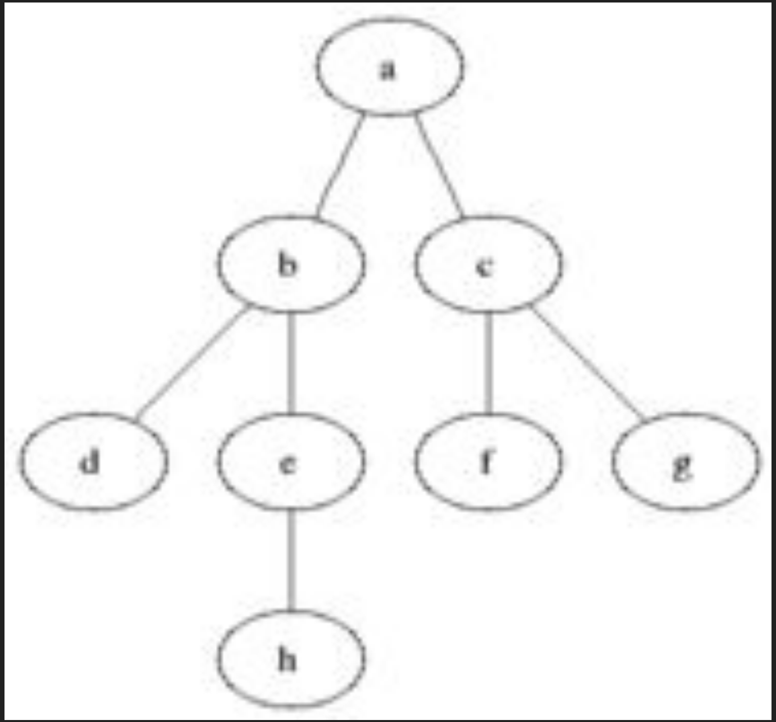
BFS Visual (1/2)

- Here is a sample of running a BFS on a graph
    - Observe a ‘queue’ data structure will help us perform the BFS.
- I’m going to advance us to a ‘tree’ (also a graph) visual which is a little easier to follow--next slide!
    - (But use this slide as a nice reference for graphs, both work)



BFS Visual (2/2)

- A breadth first search of a tree searches nodes one level at a time in the tree.
- Sometimes this is called ‘level-order traversal’
    - White are unvisited nodes
    - gray are ‘queued up’ to be visited
    - black are visited nodes



BFS Example

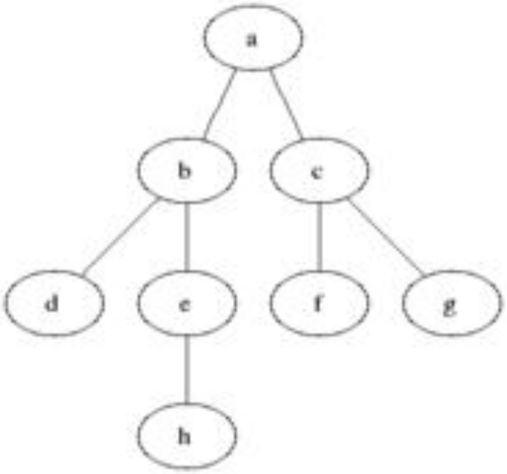
Pseudocode and walkthrough

BFS Pseudo-code (1/3)

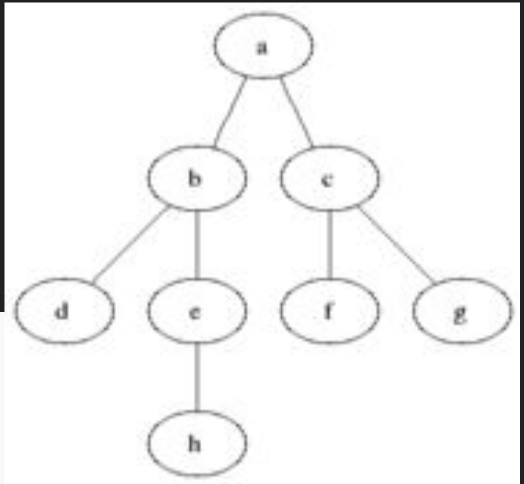
Key of Algorithm

- We need a queue

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            u ← G.adjacentVertex(t,e)
            if u is not marked:
                mark u
                enqueue u onto Q
    return none
```



BFS Pseudo-code (2/3)

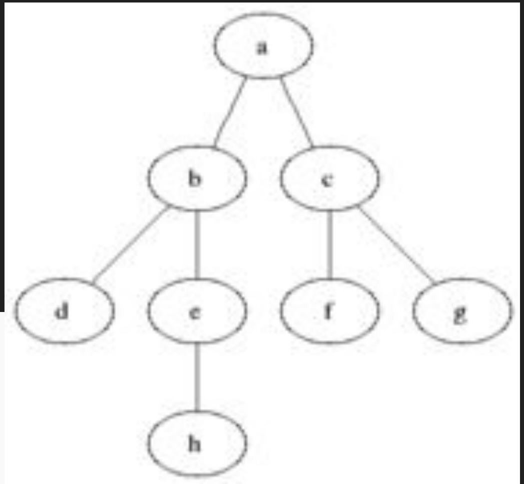


Key of Algorithm

- We need a queue
- We need a ‘field’ in our node_t to mark as ‘visited’ or not

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            u ← G.adjacentVertex(t,e)
            if u is not marked:
                mark u
                enqueue u onto Q
    return none
```

BFS Pseudo-code (3/3)



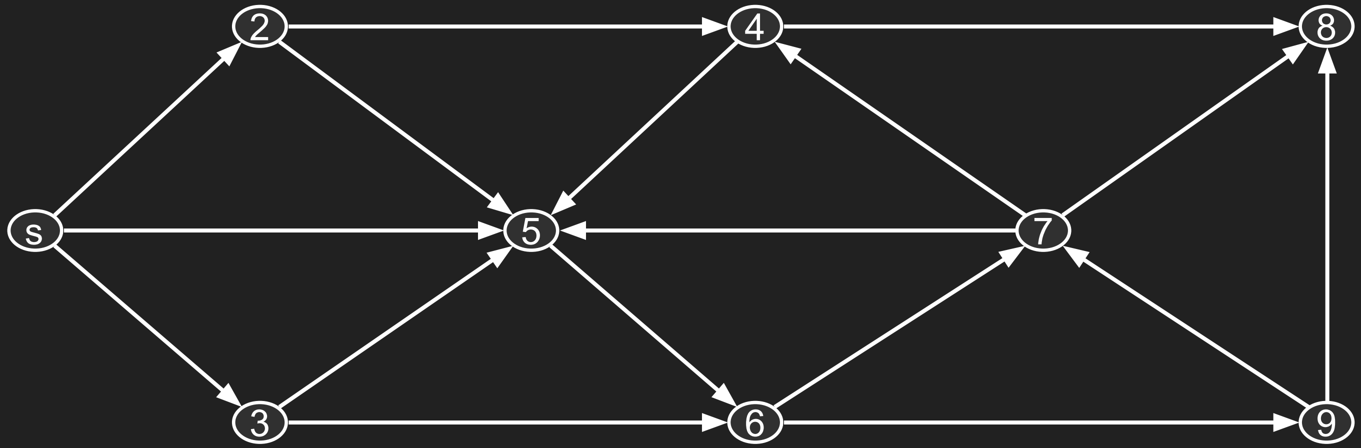
<u>Key of Algorithm</u>

- We need a queue
- We need a ‘field’ in our node_t to mark as ‘visited’ or not
- We need to know which vertices are adjacent
    - (i.e. our neighbors or nodes we are connected to by an edge)

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            u ← G.adjacentVertex(t,e)
            if u is not marked:
                mark u
                enqueue u onto Q
    return none
```

Breadth First Search on a Graph Example

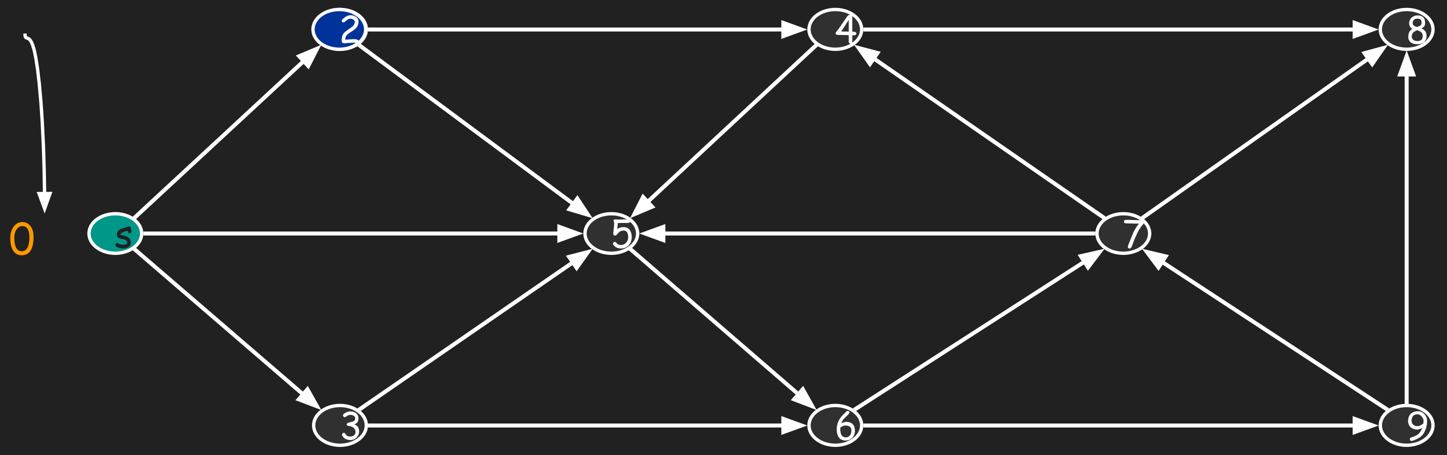
- **Visit the nodes one-level at a time**
- **Requires a queue (First-in-first-out)**



Start from 's' and push into a queue everything reachable, 2, 3, and 5

Breadth First Search

Shortest path from s



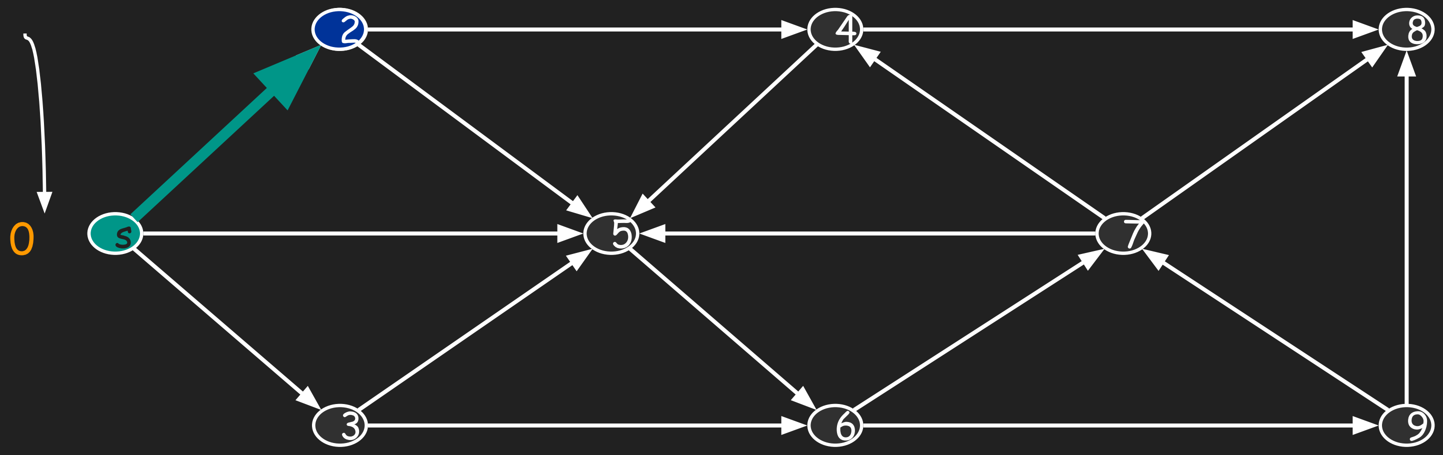
Undiscovered
Discovered
Top of queue
Finished

Queue: s

add 2

Breadth First Search

Shortest path from s

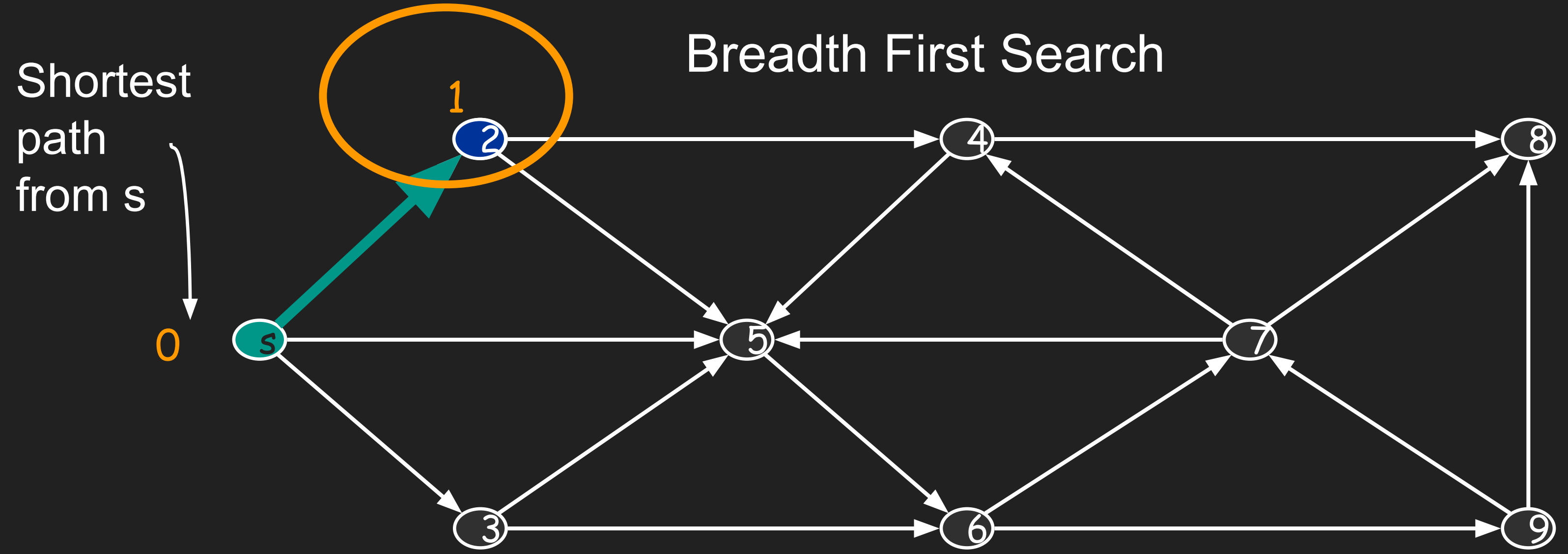


Undiscovered
Discovered
Top of queue
Finished

Queue: s

I am also going to do some bookkeeping for what 'level' or how deep we are in our tree to measure distance.

Breadth First Search

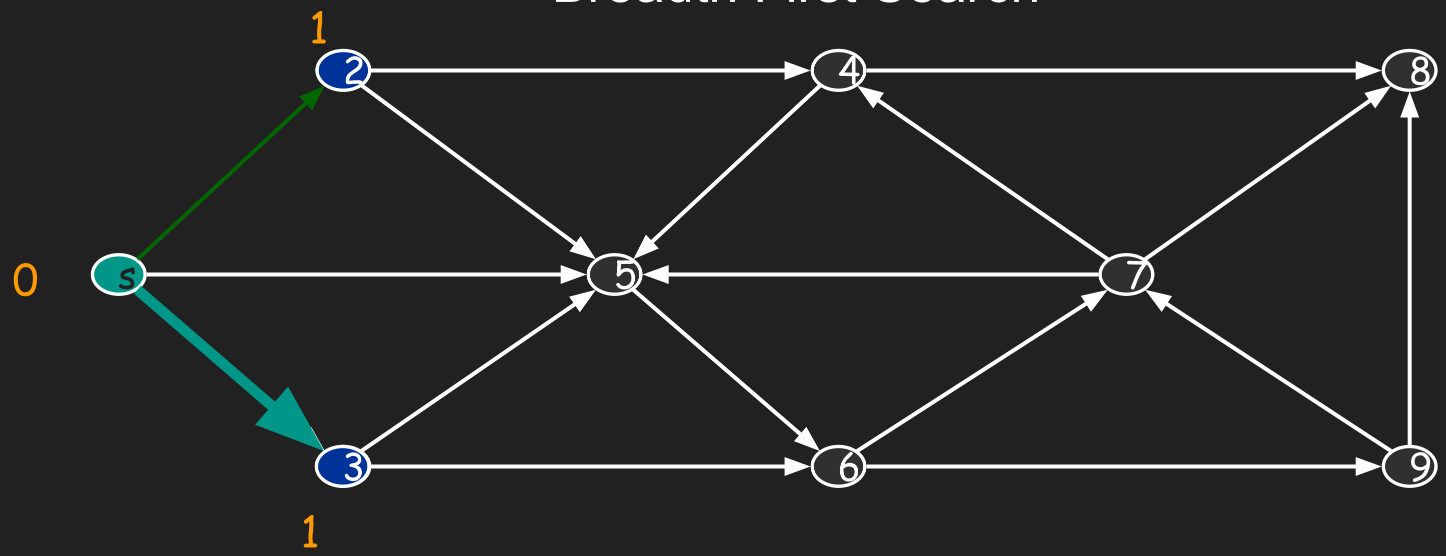


Undiscovered
Discovered
Top of queue
Finished

Queue: s

Add 3

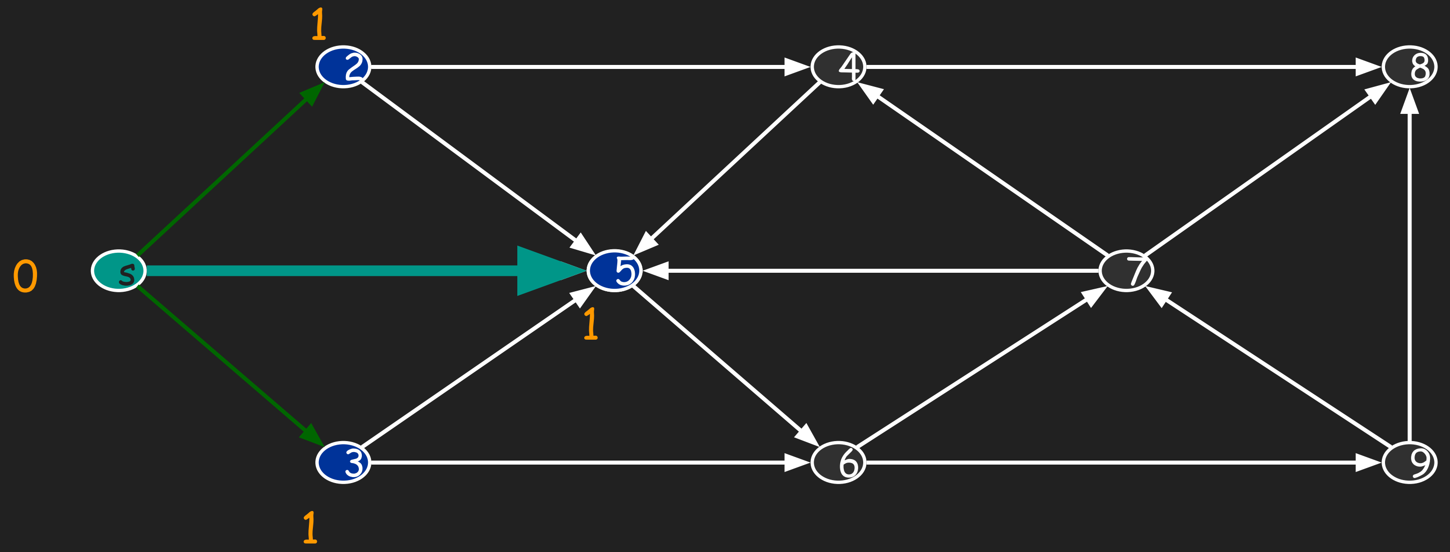
Breadth First Search



Queue: s 2

Add 5

Breadth First Search

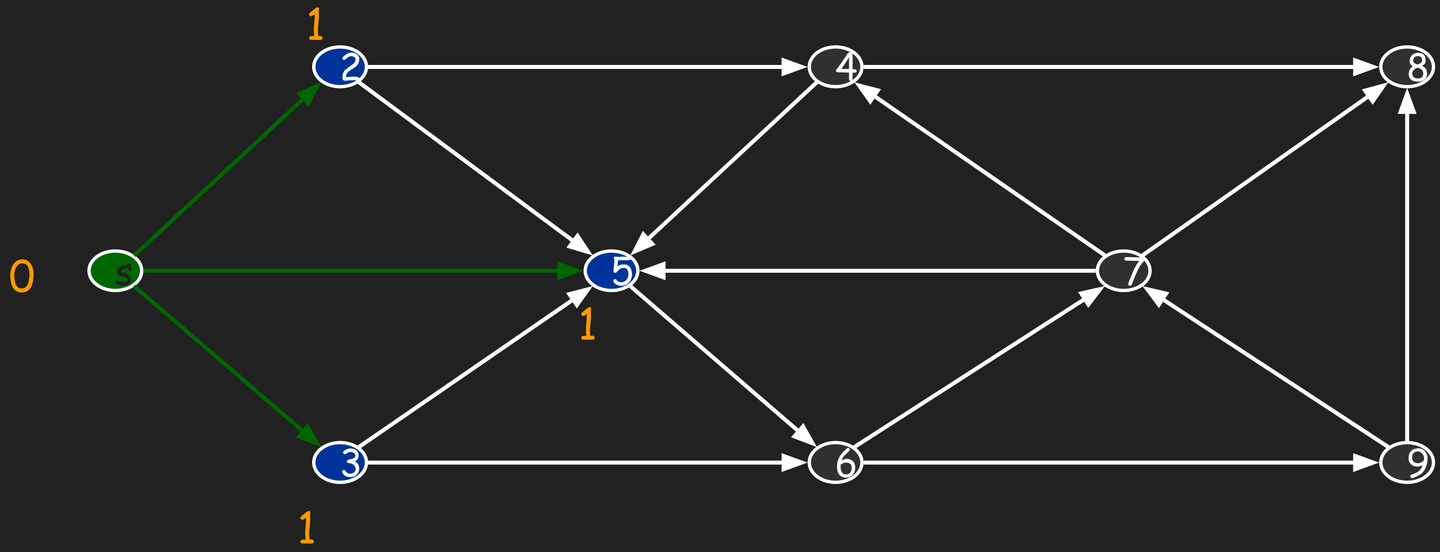


Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3

Alright, we have finished ‘s’. Our queue is not empty, so we start from ‘2’

Breadth First Search

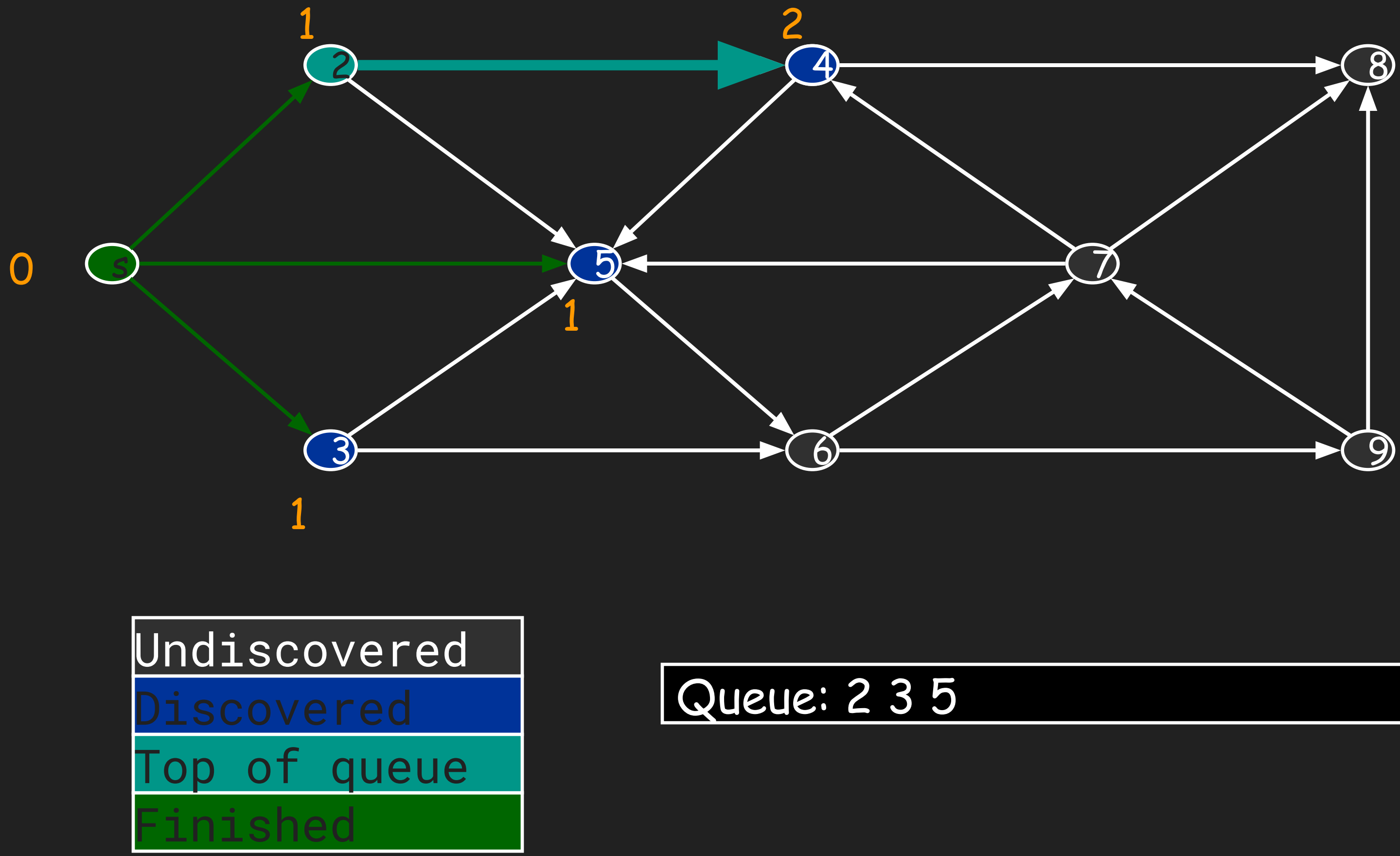


Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

Now, again from the front of our ‘q’, start exploring

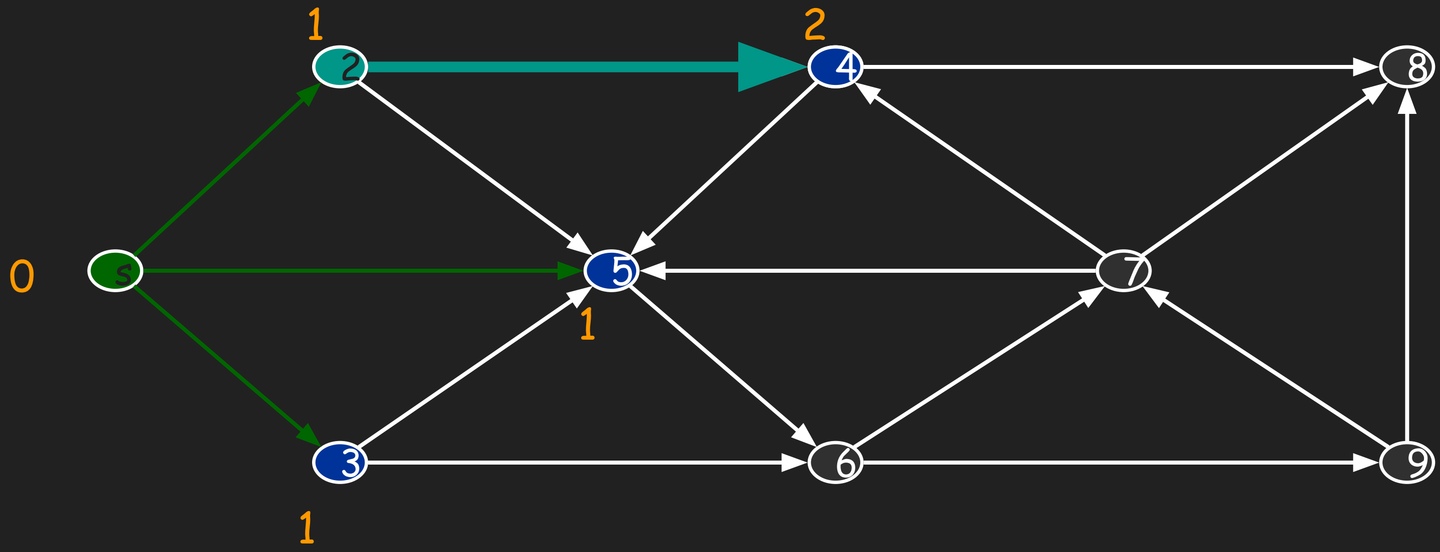
Breadth First Search



Queue: 2 3 5

Let's add 4

Breadth First Search

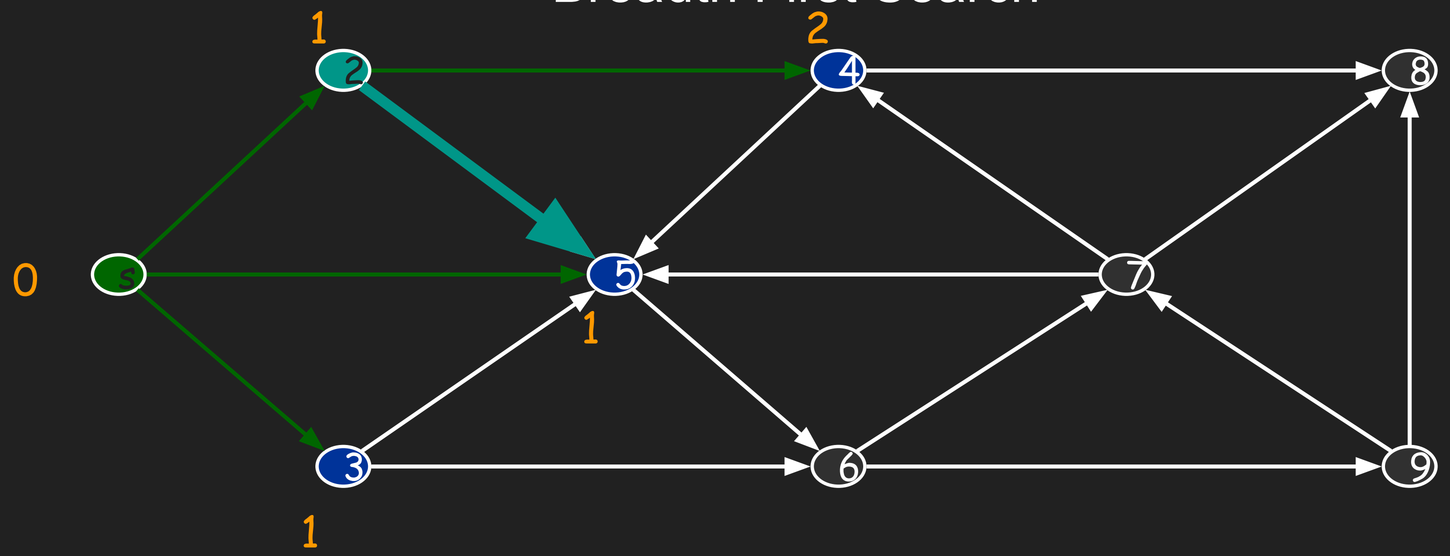


Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

Next neighbor is ‘5’ -- should we add it?

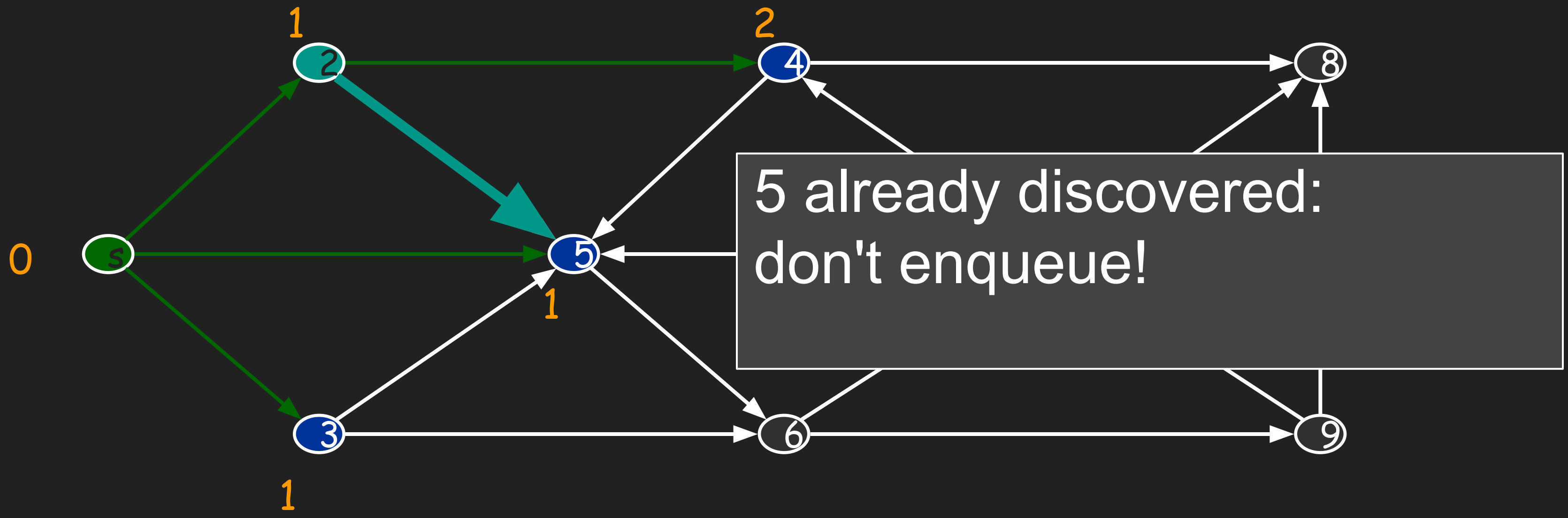
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

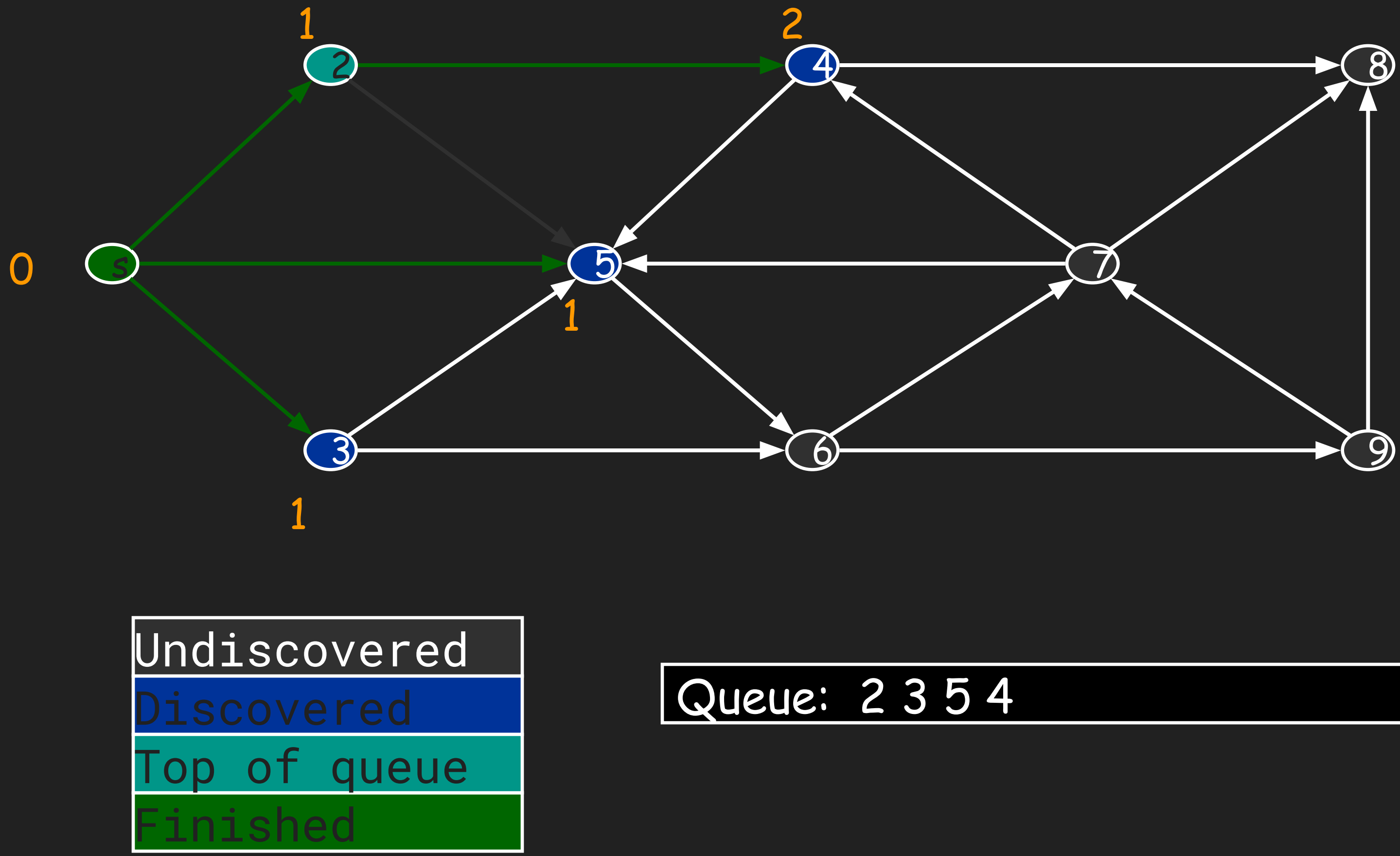
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

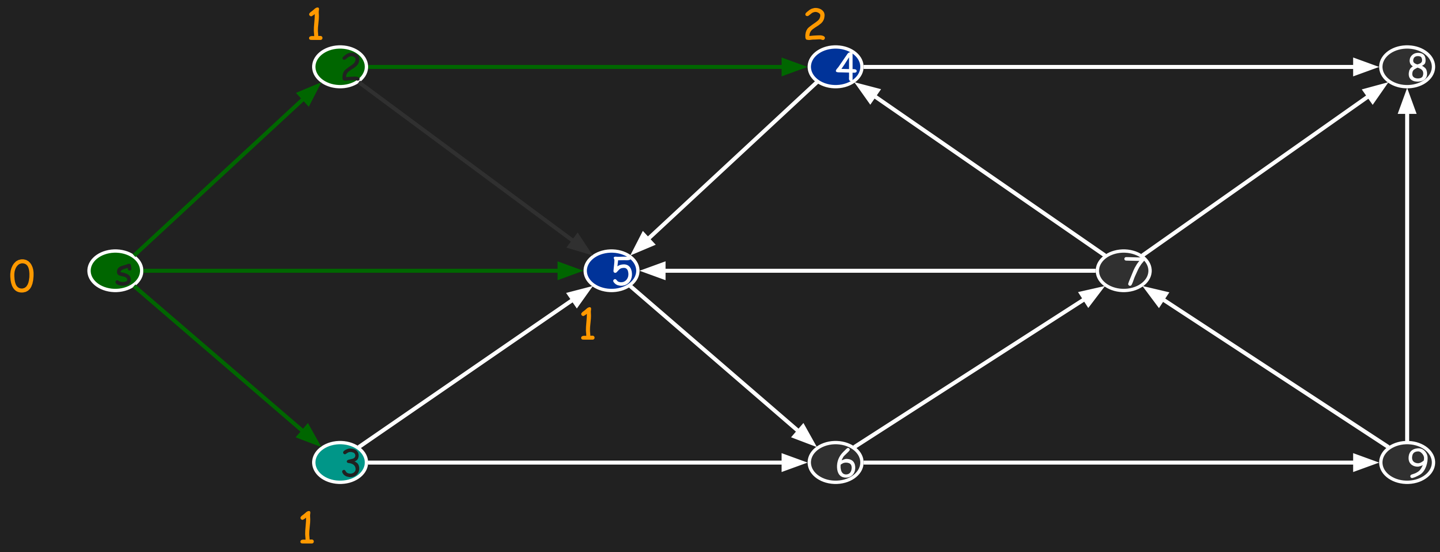
Breadth First Search



Queue: 2 3 5 4

Repeat, and start dequeuing from top of our queue (this time 3’s neighbors)

Breadth First Search

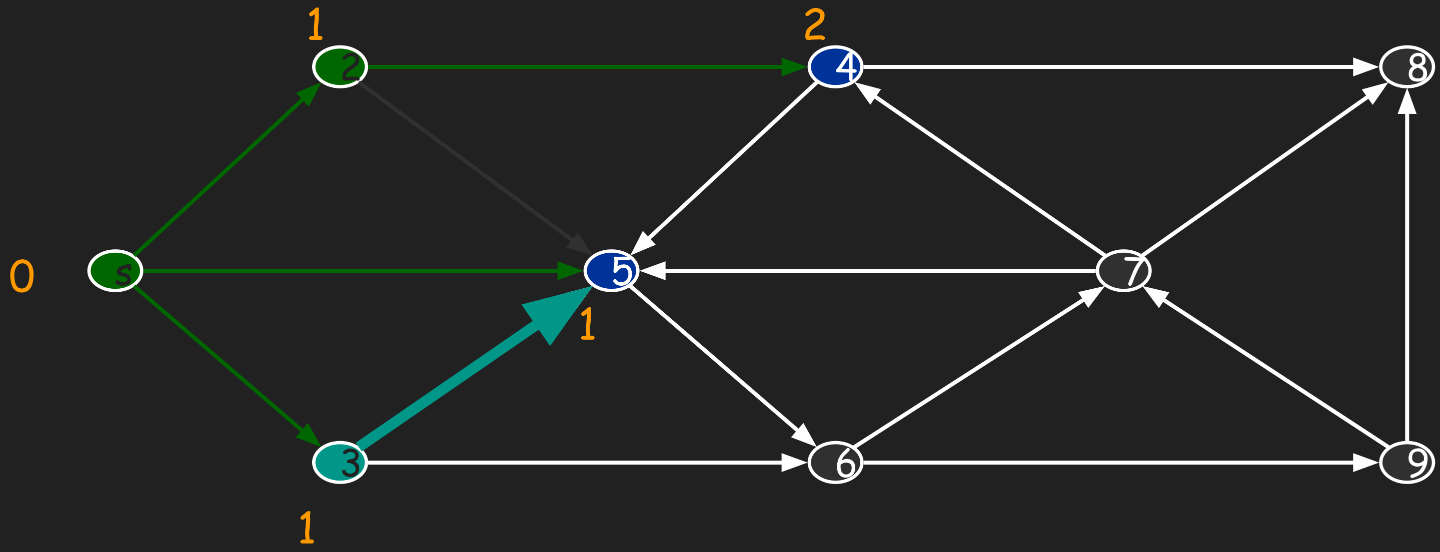


Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4

‘5’ again has already been discovered

Breadth First Search

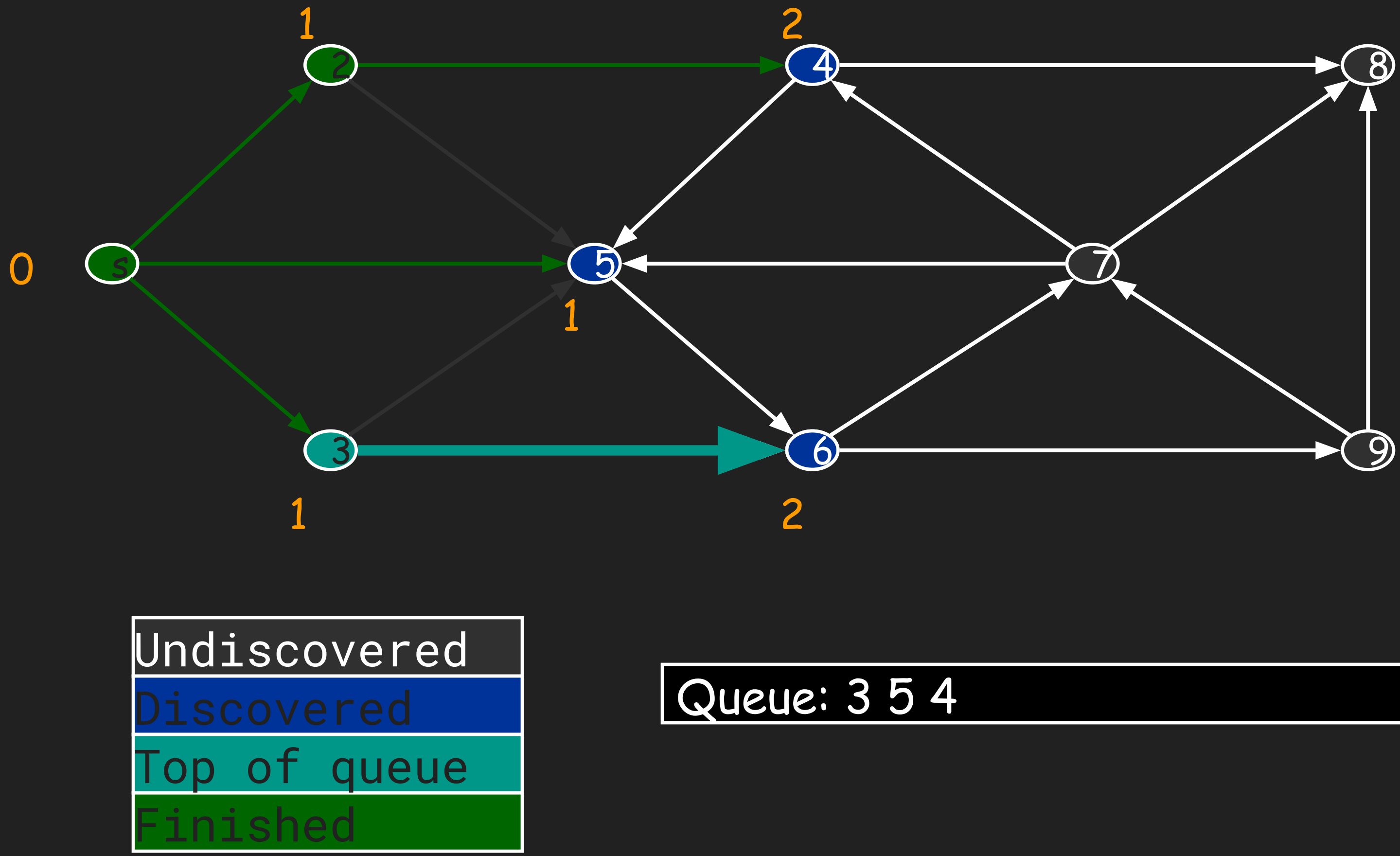


Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4

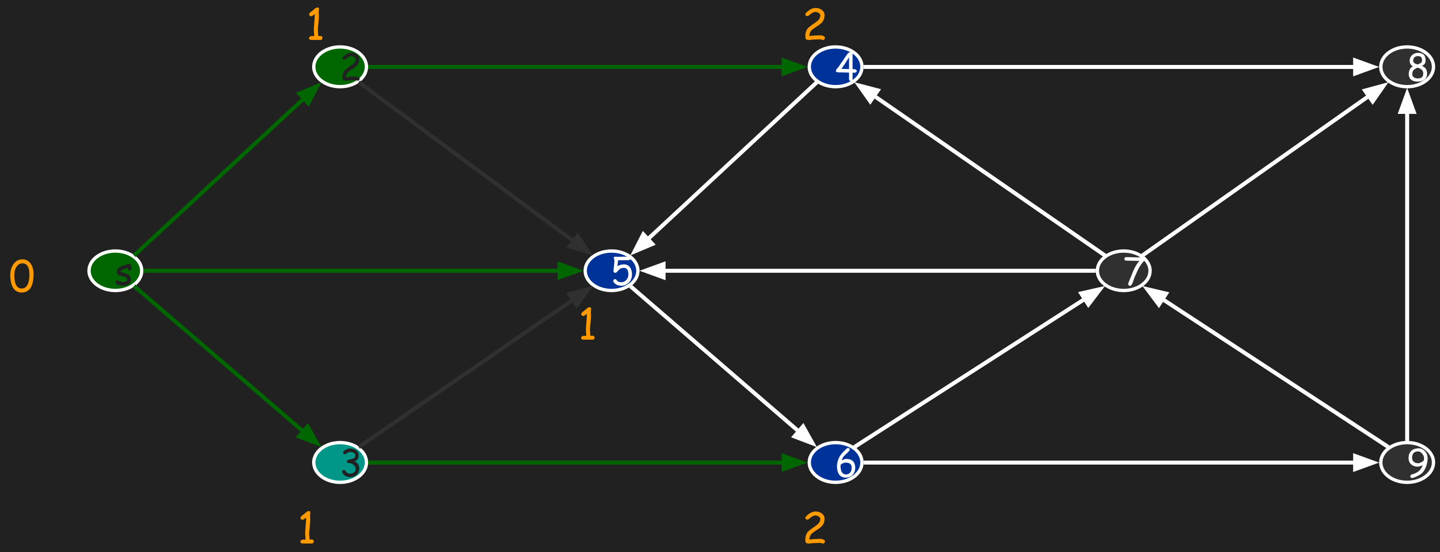
6 has not been ‘visited’ so enqueue

Breadth First Search



Queue: 3 5 4

Breadth First Search

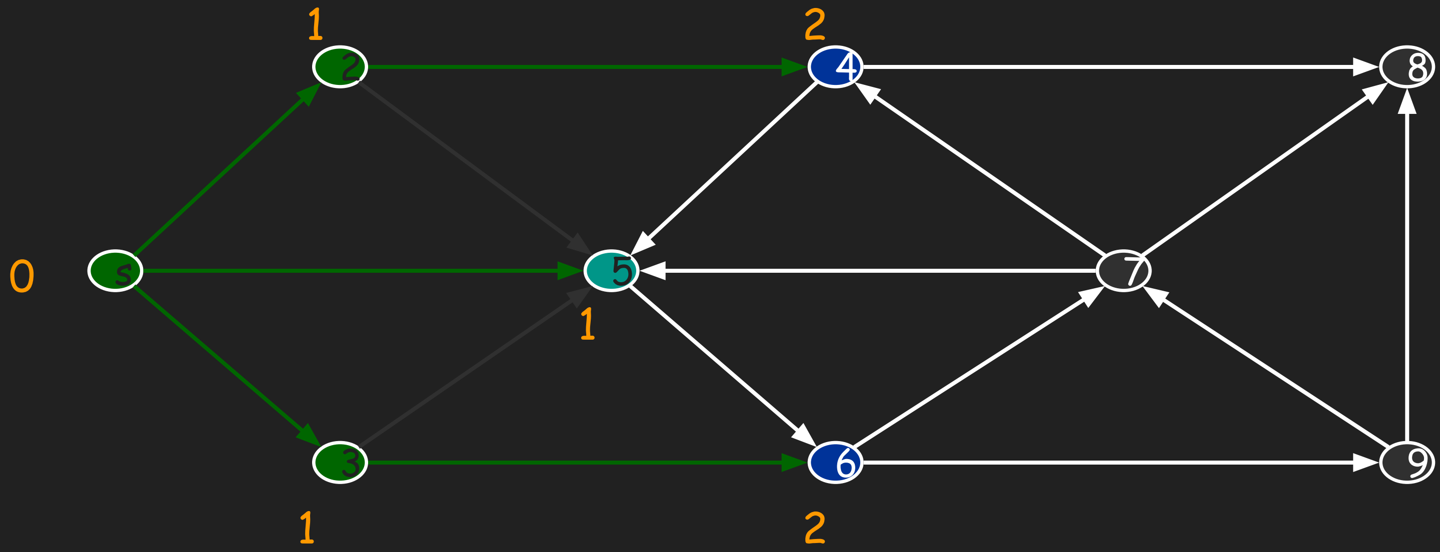


Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4 6

5 is now top of queue, so find anything not discovered

Breadth First Search

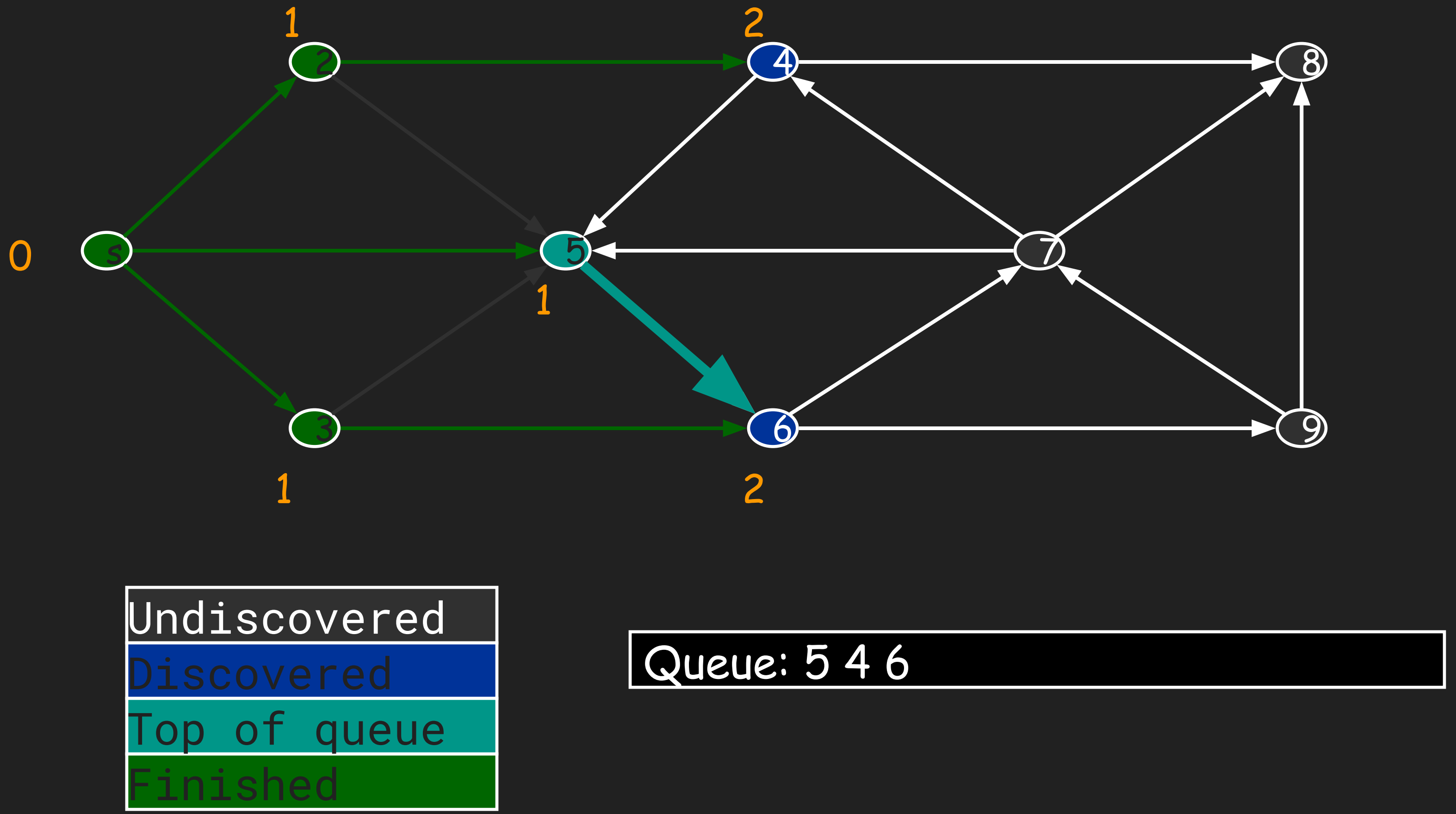


Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6

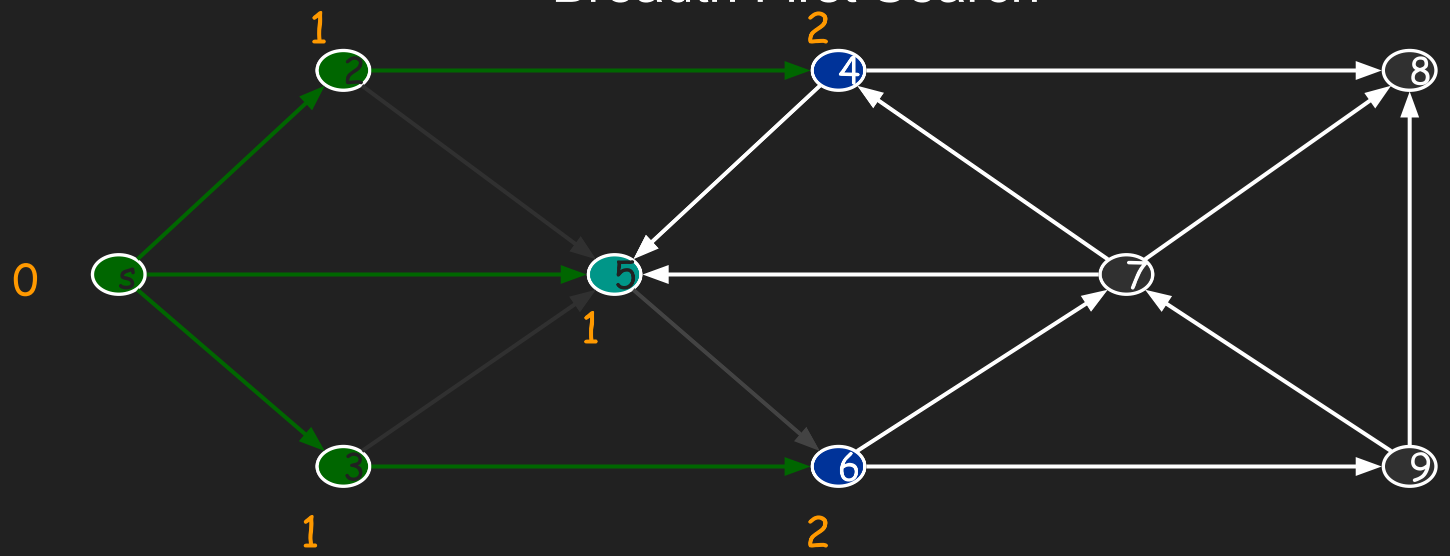
‘6’ already discovered and in the queue--no work to be done.

Breadth First Search



‘5’ has no outgoing edges that have not been already discovered

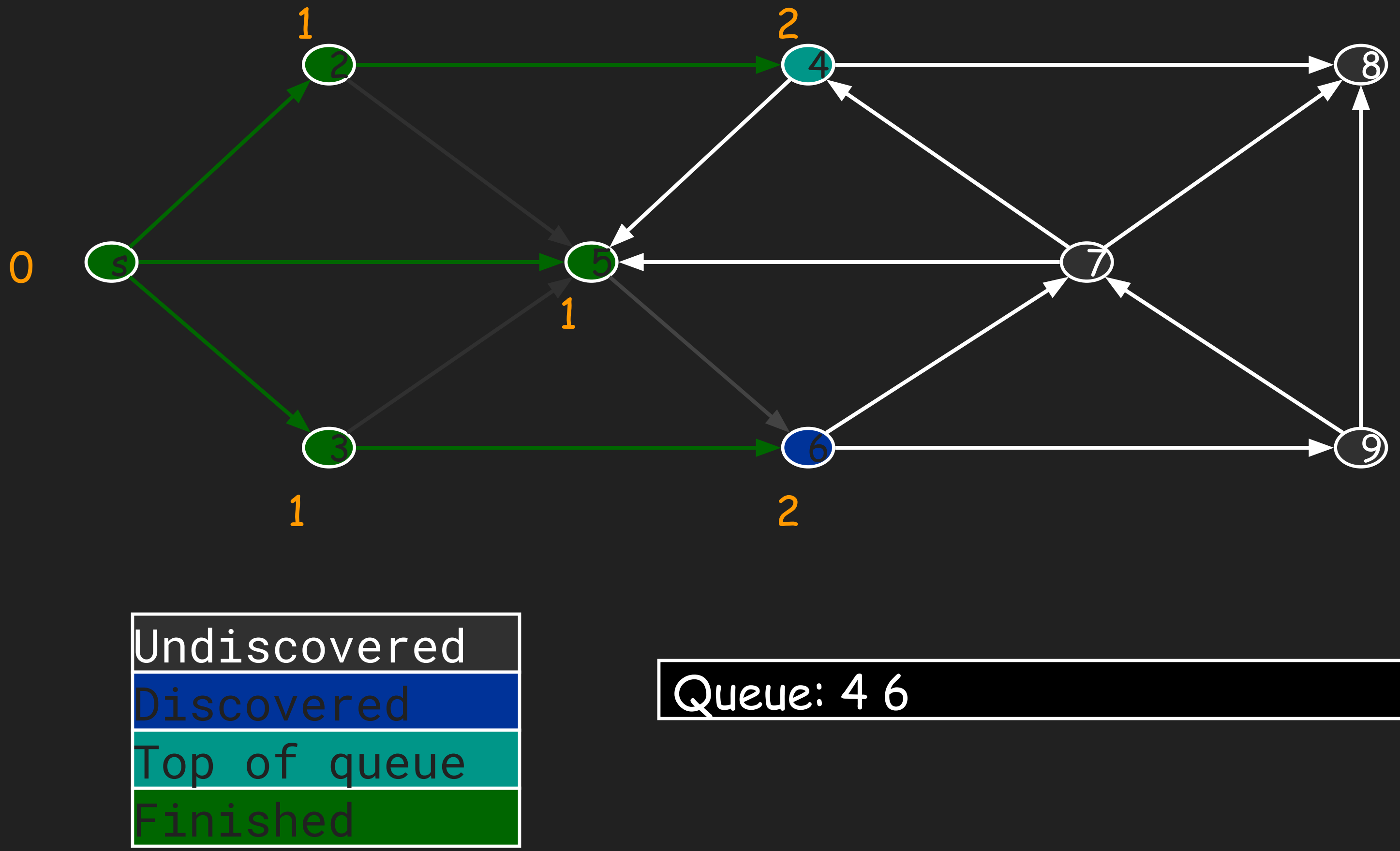
Breadth First Search



Queue: 5 4 6

Again from 4, look for undiscovered nodes

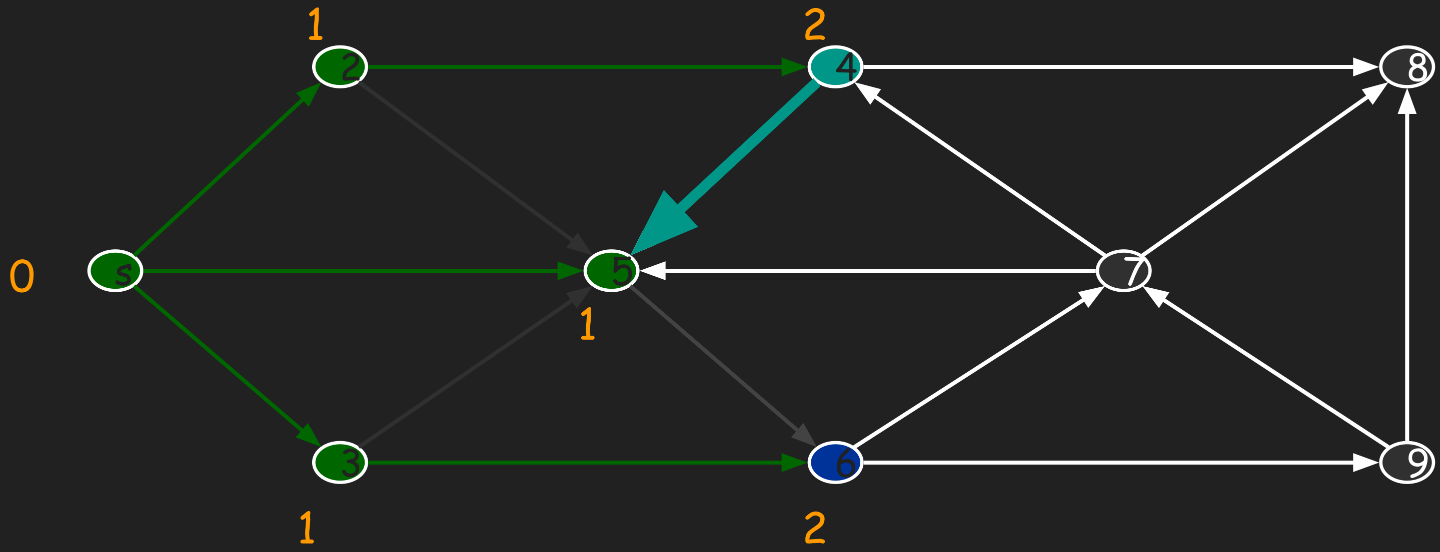
Breadth First Search



Queue: 4 6

‘5’ is already finished

Breadth First Search

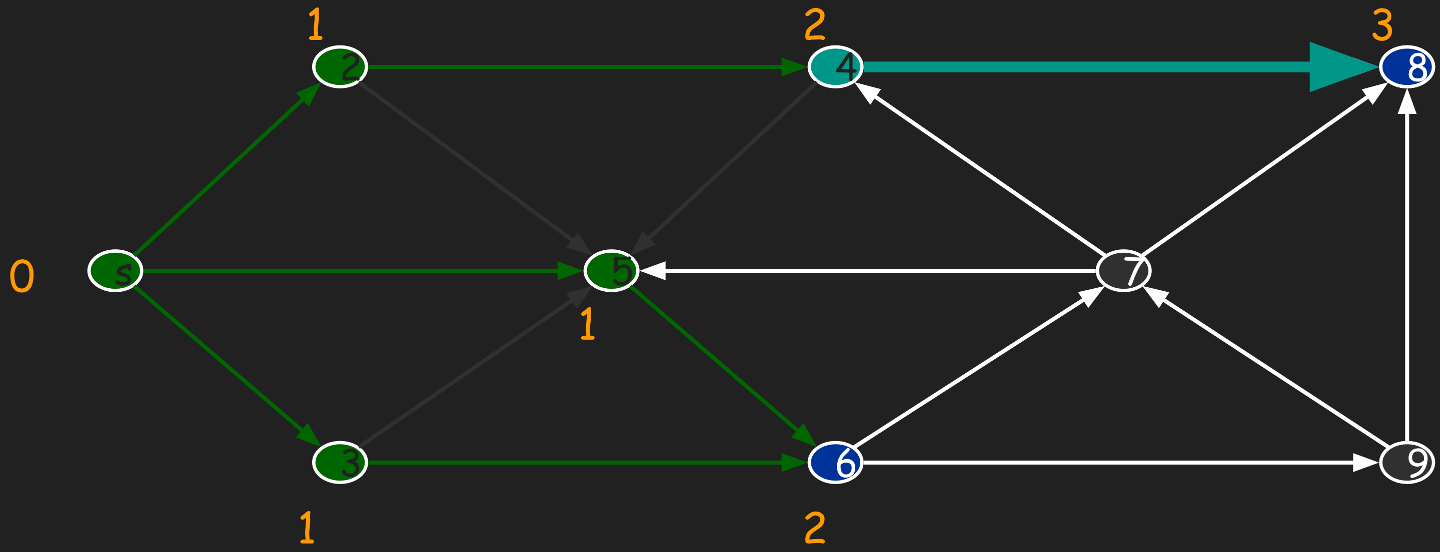


Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

Aha, found 8!

Breadth First Search

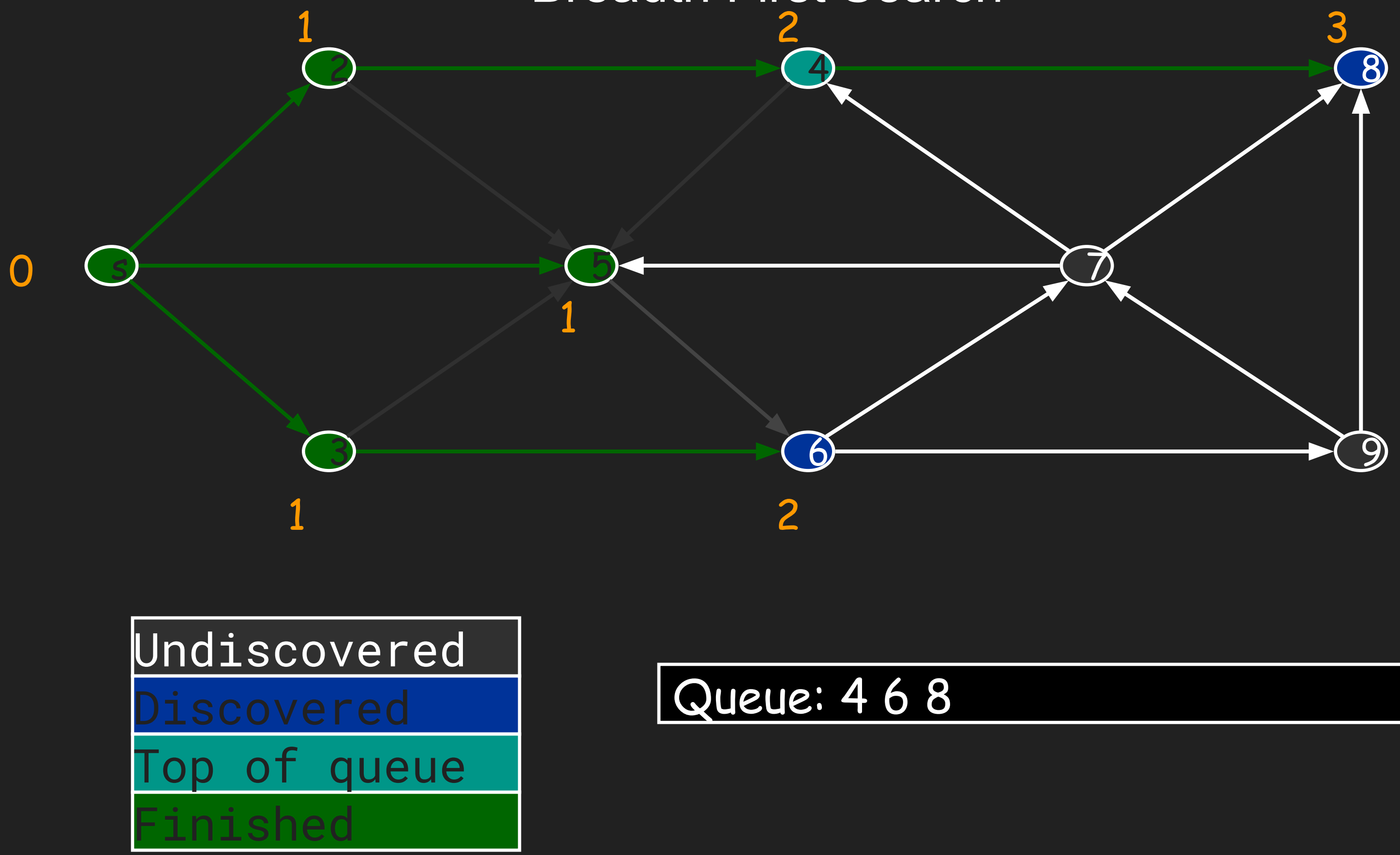


Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

8 pushed to back of queue

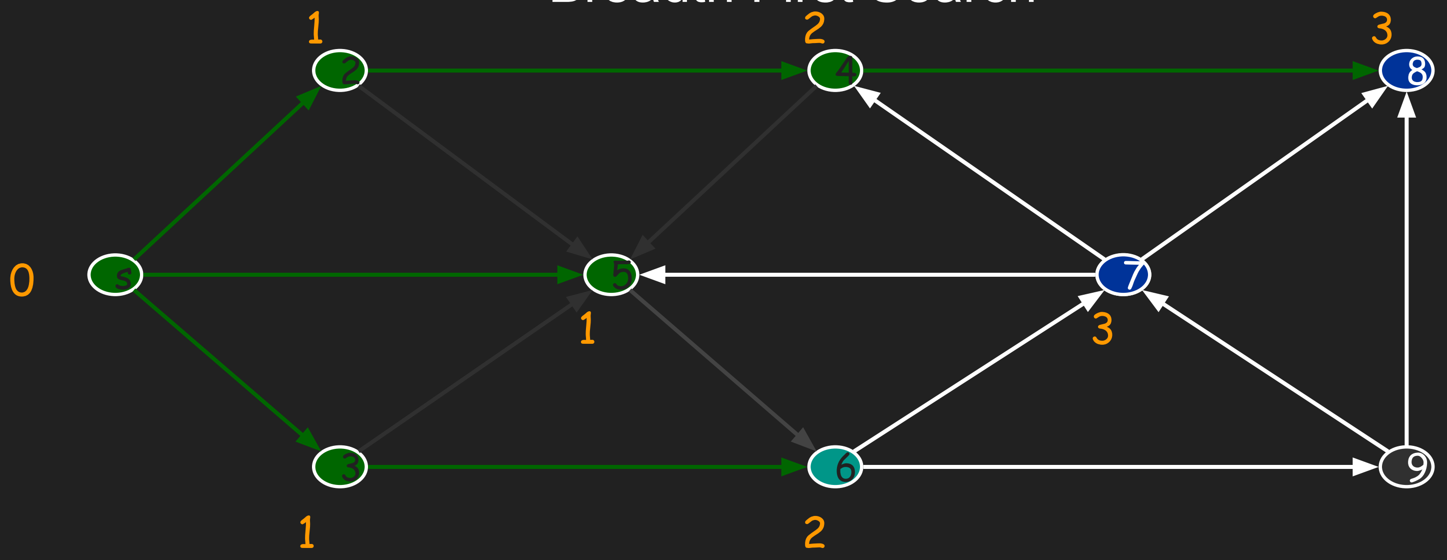
Breadth First Search



Queue: 4 6 8

Start from top of queue, ‘6’

Breadth First Search

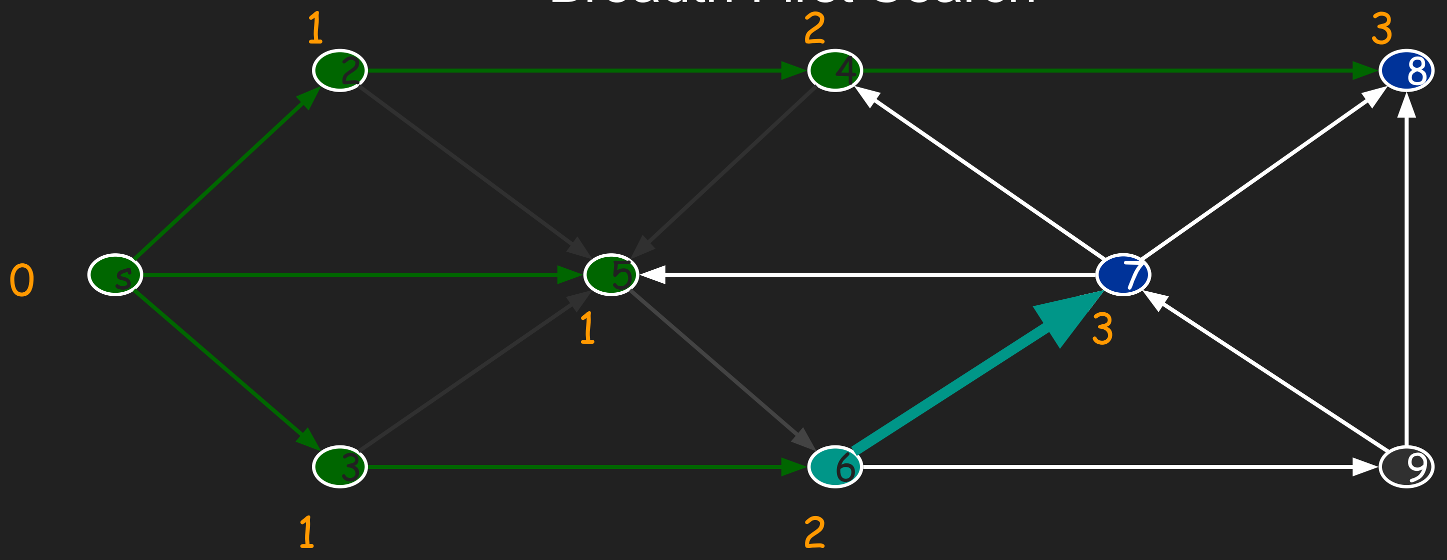


Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8

we then find ‘7’ to enqueue

Breadth First Search

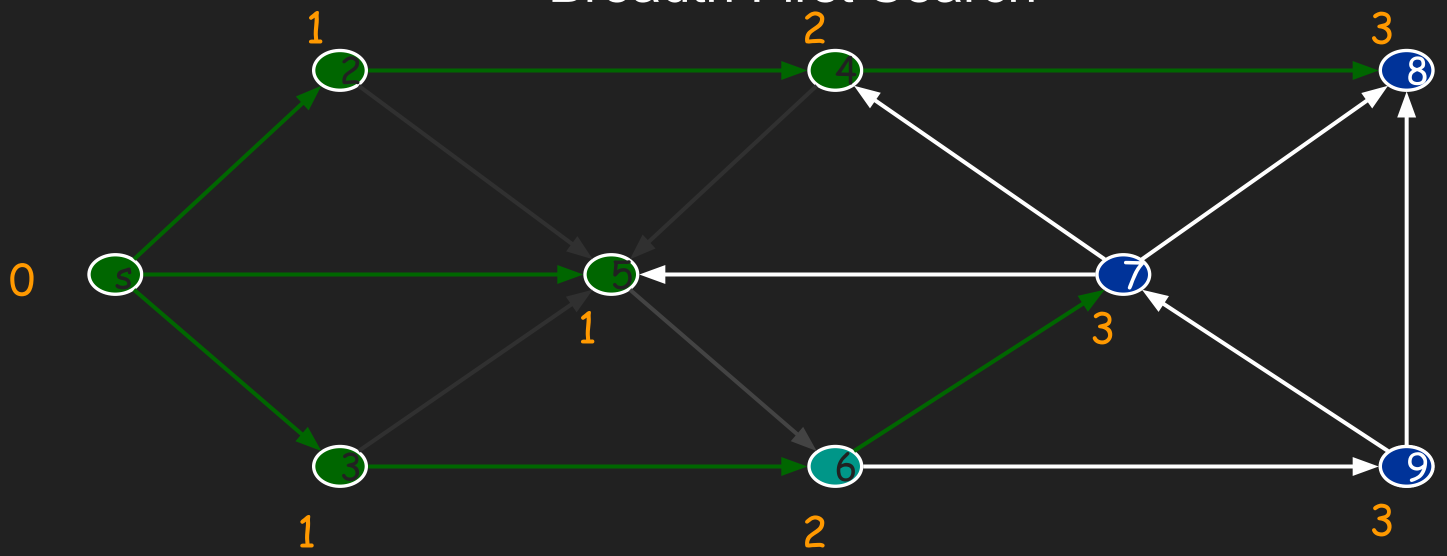


Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8

Added ‘7’

Breadth First Search

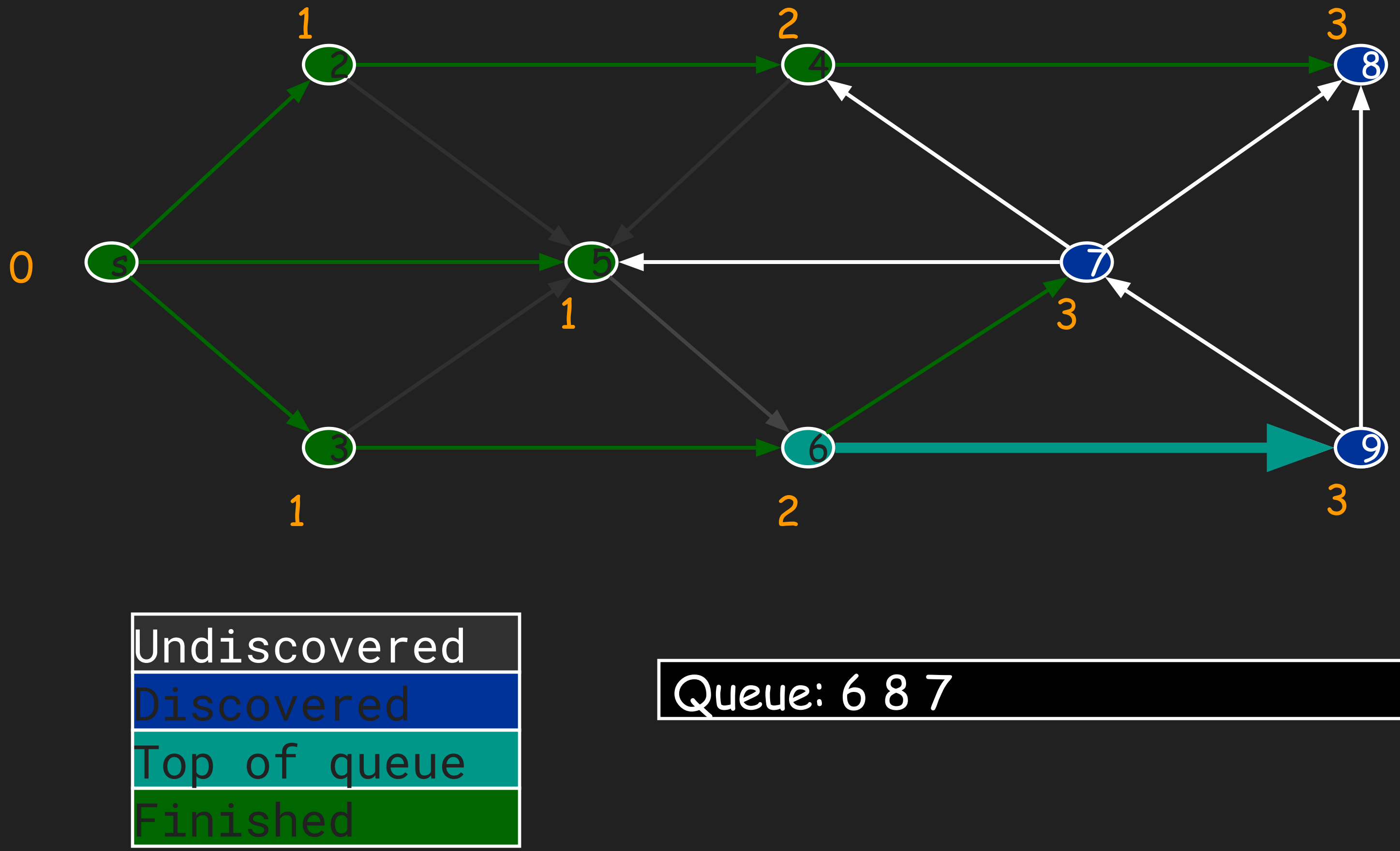


Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7

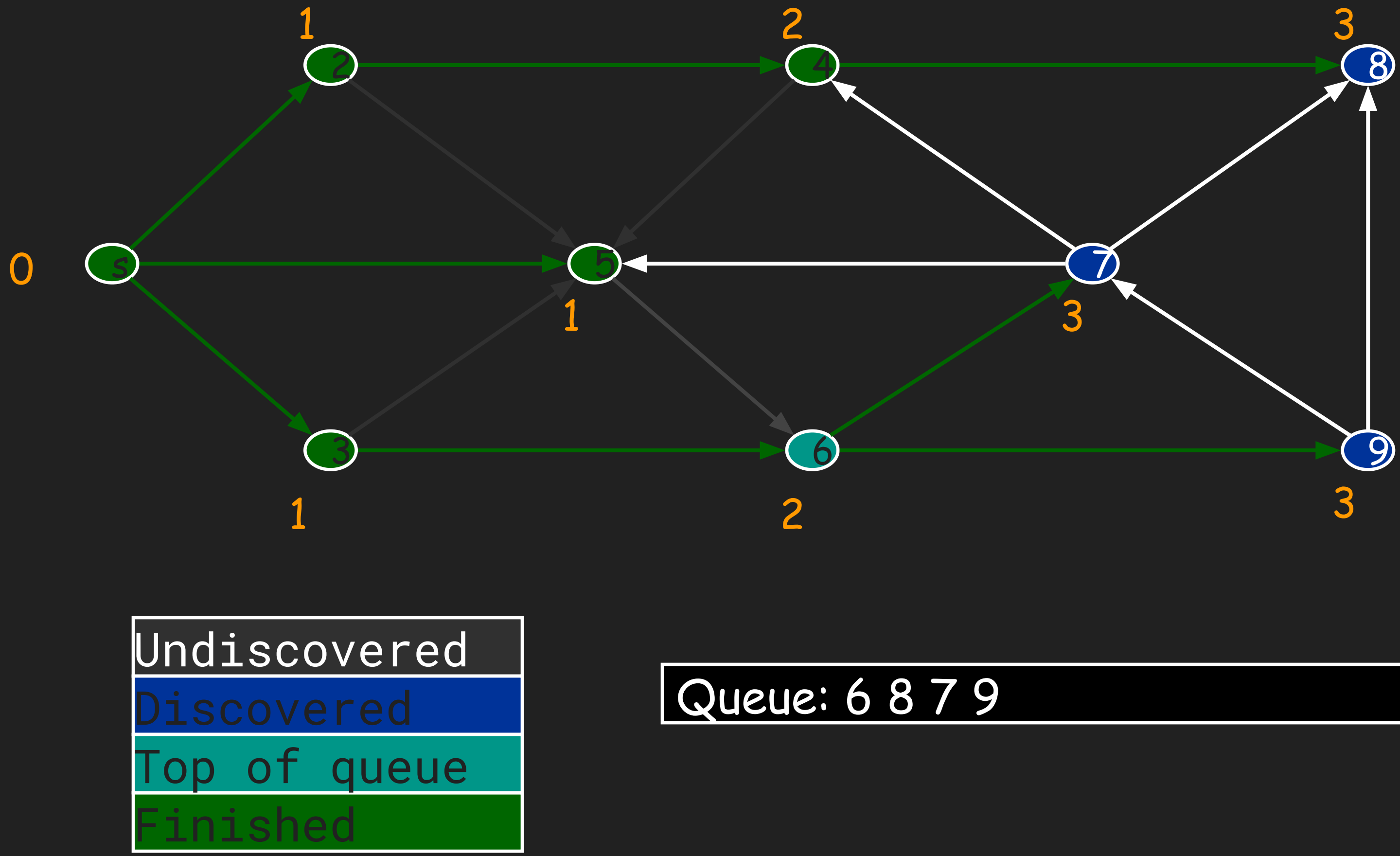
And also 9

Breadth First Search



Queue: 6 8 7

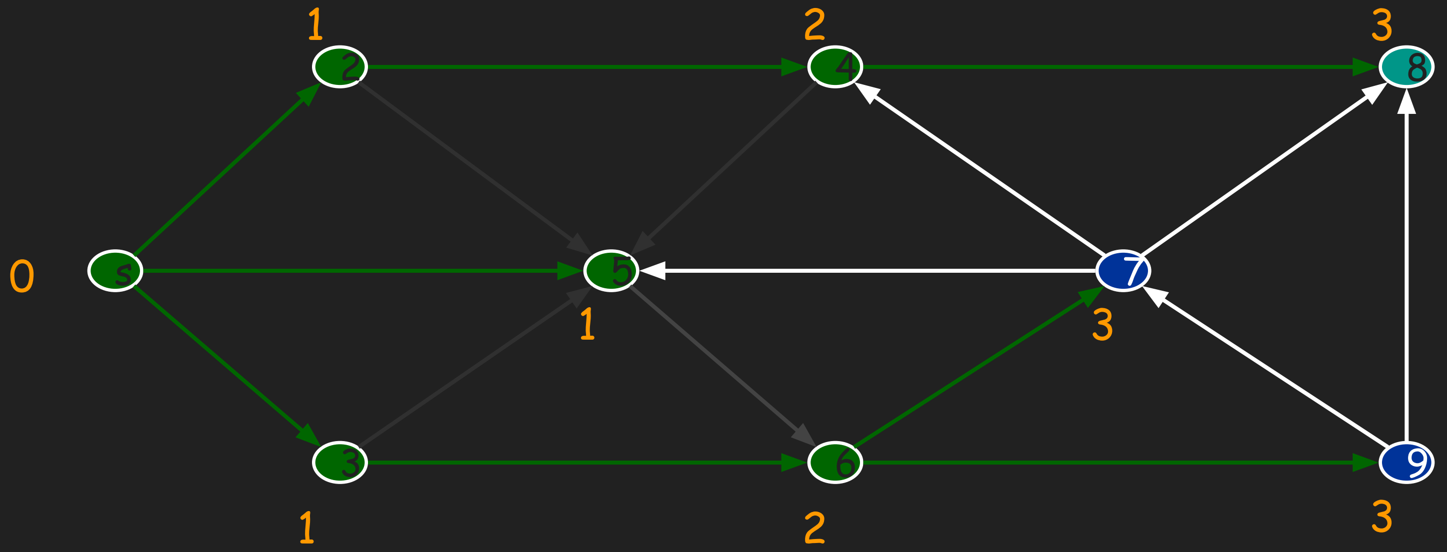
Breadth First Search



Queue: 6 8 7 9

top of queue is ‘6’, no undiscovered neighbors

Breadth First Search

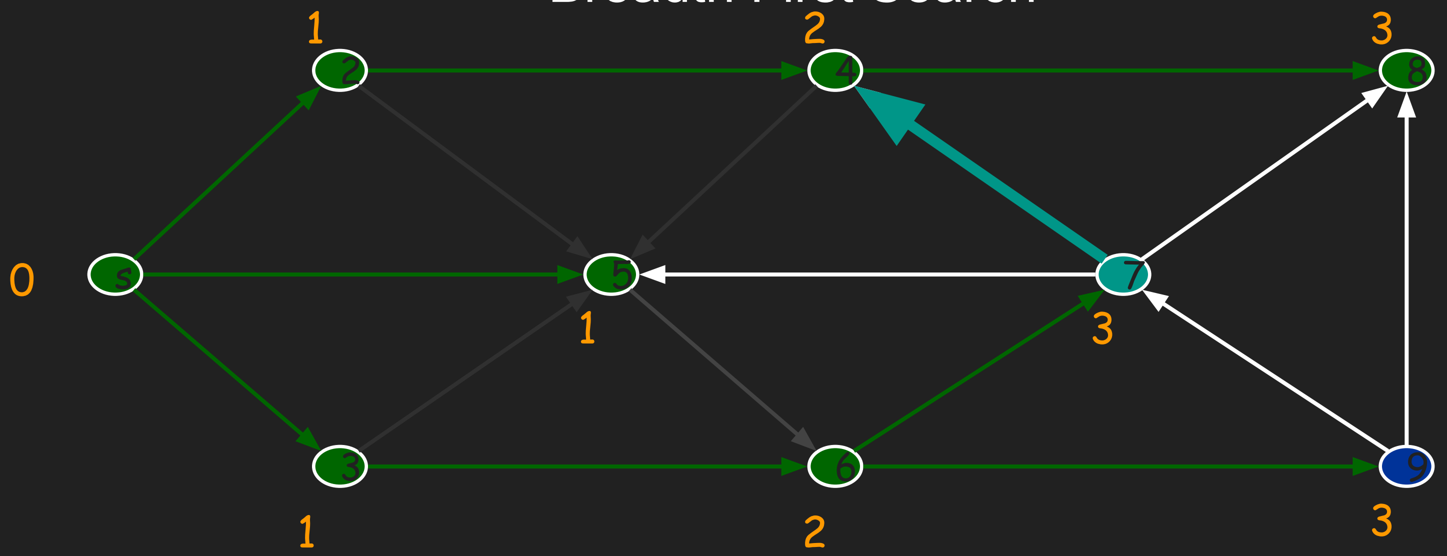


Undiscovered
Discovered
Top of queue
Finished

Queue: 8 7 9

‘7’ also has no more nodes to discovere (watch it search outgoing edges)

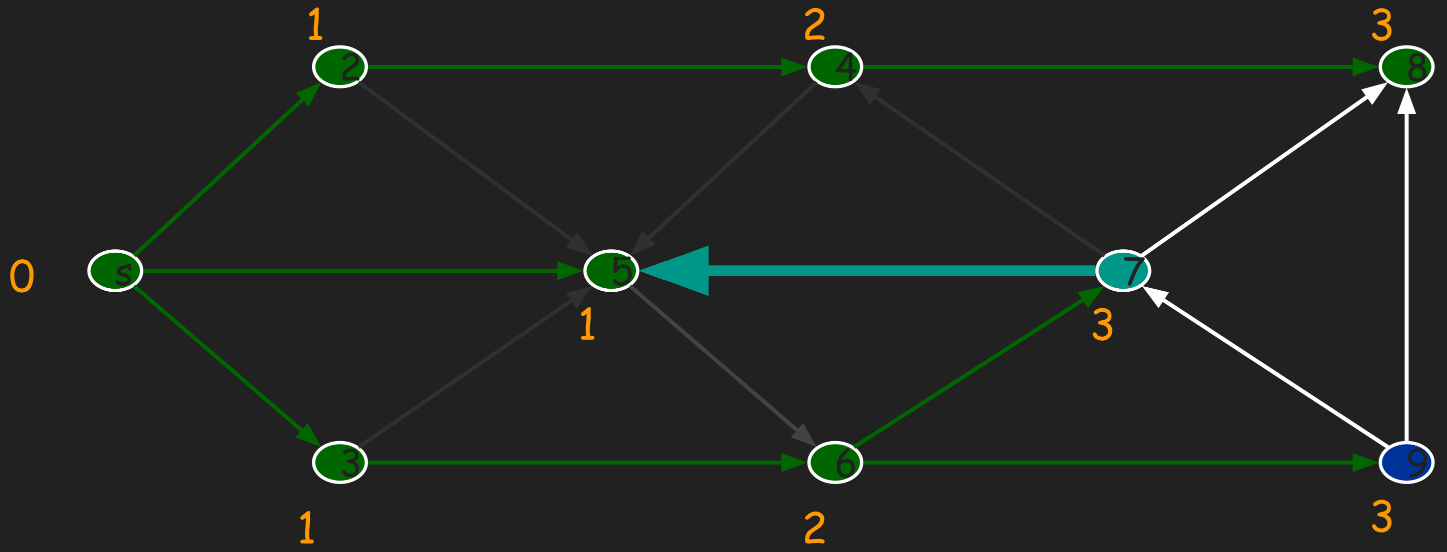
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

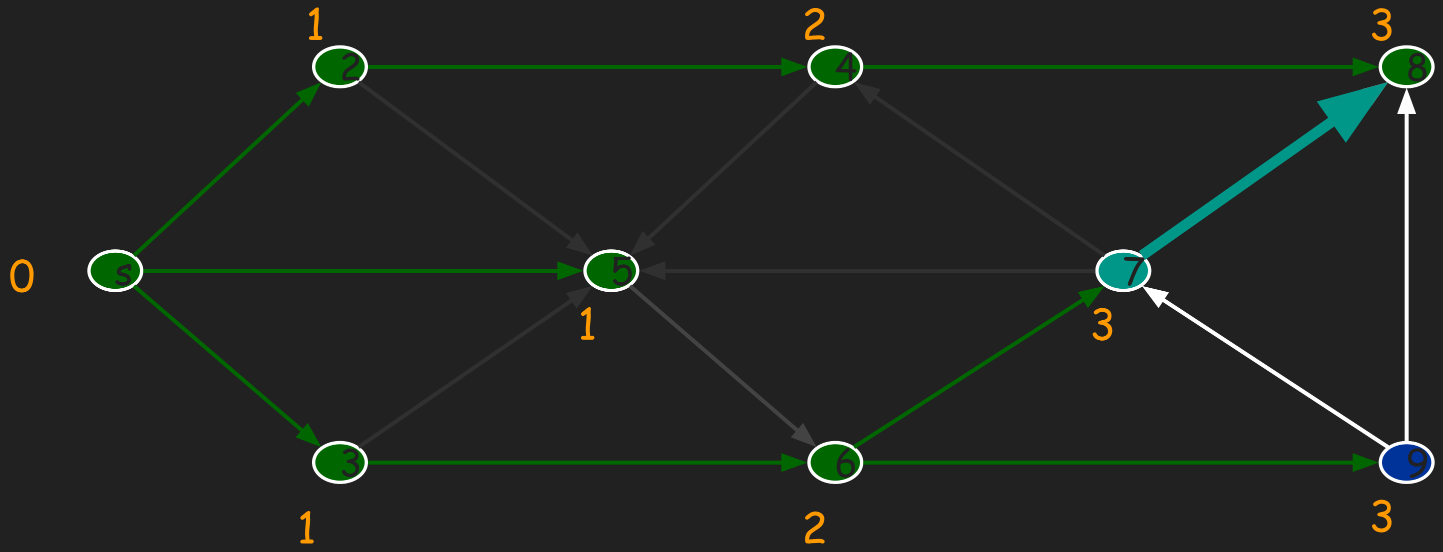
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

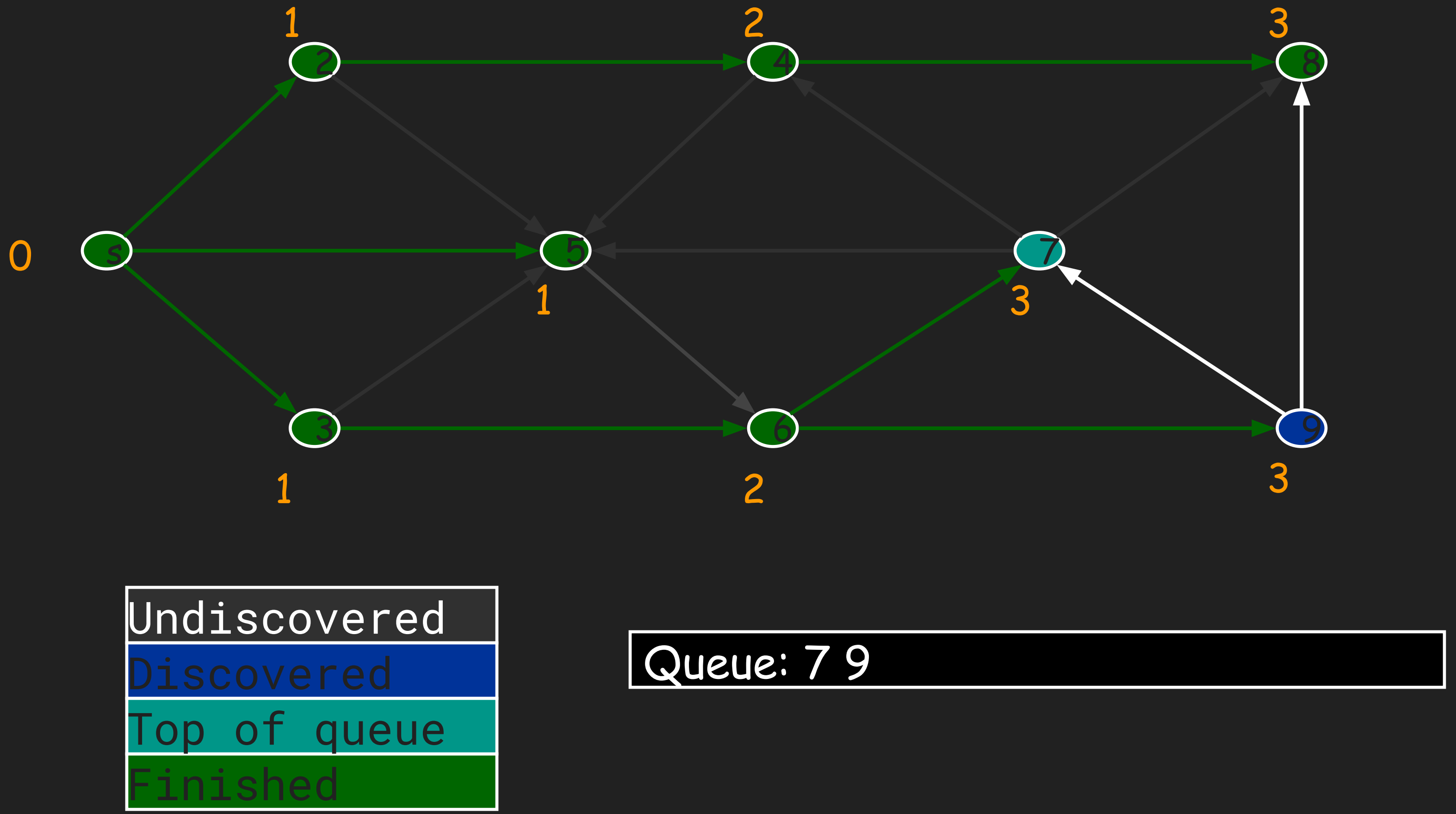
Breadth First Search



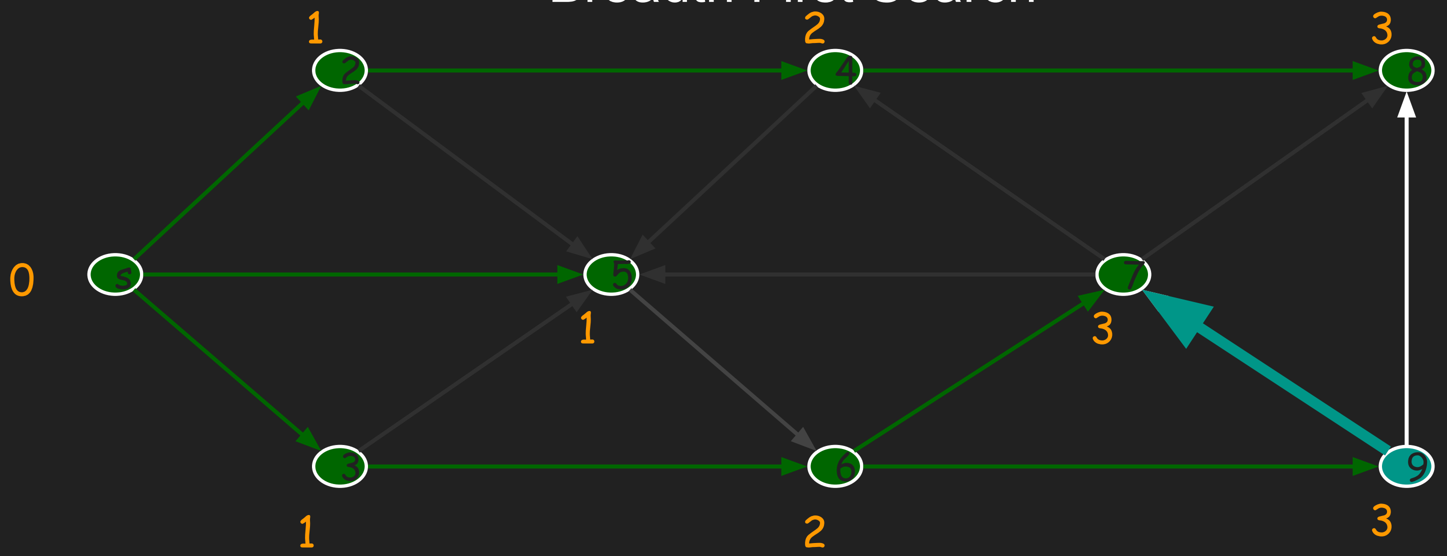
Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

Breadth First Search



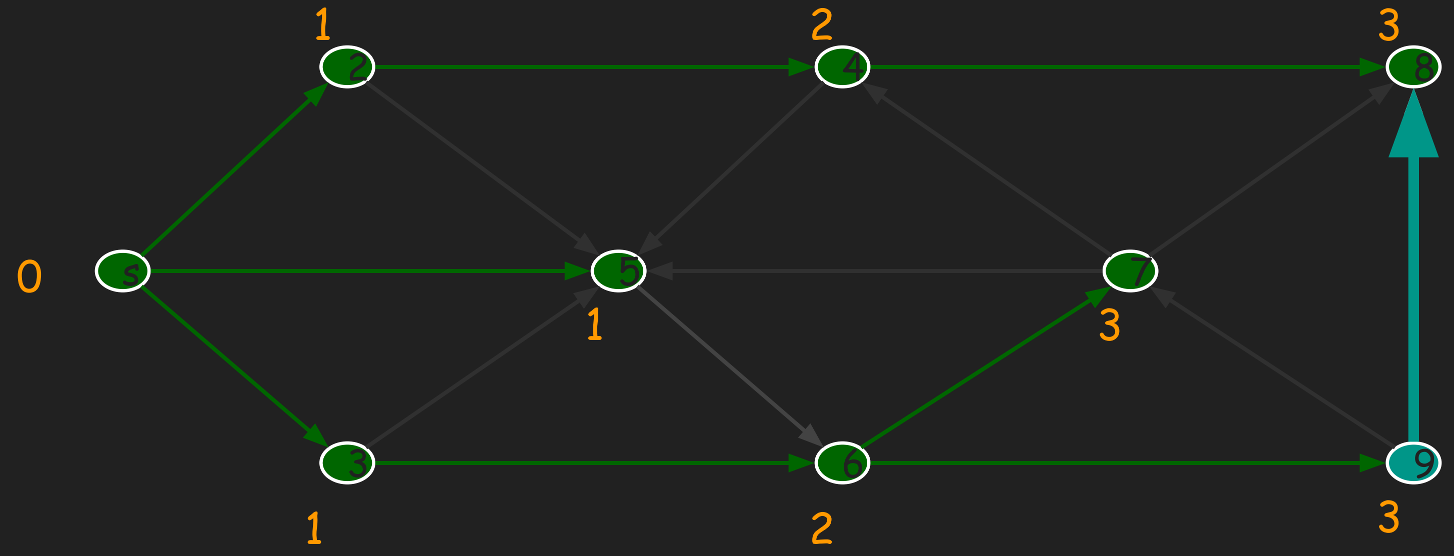
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

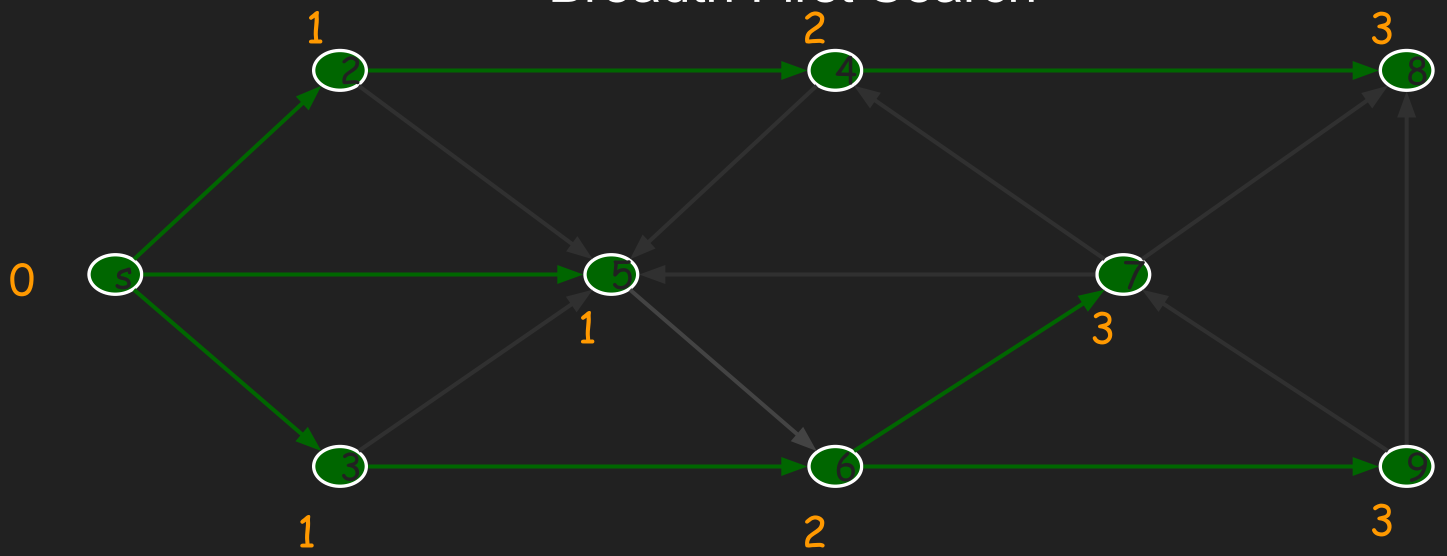
Breadth First Search

Undiscovered
Discovered
Top of queue
Finished

Queue: 9

No more nodes to be found, so we are done. Note that we have logged the 'distance' to each node to give us the minimum number of steps to each node from our start 's'

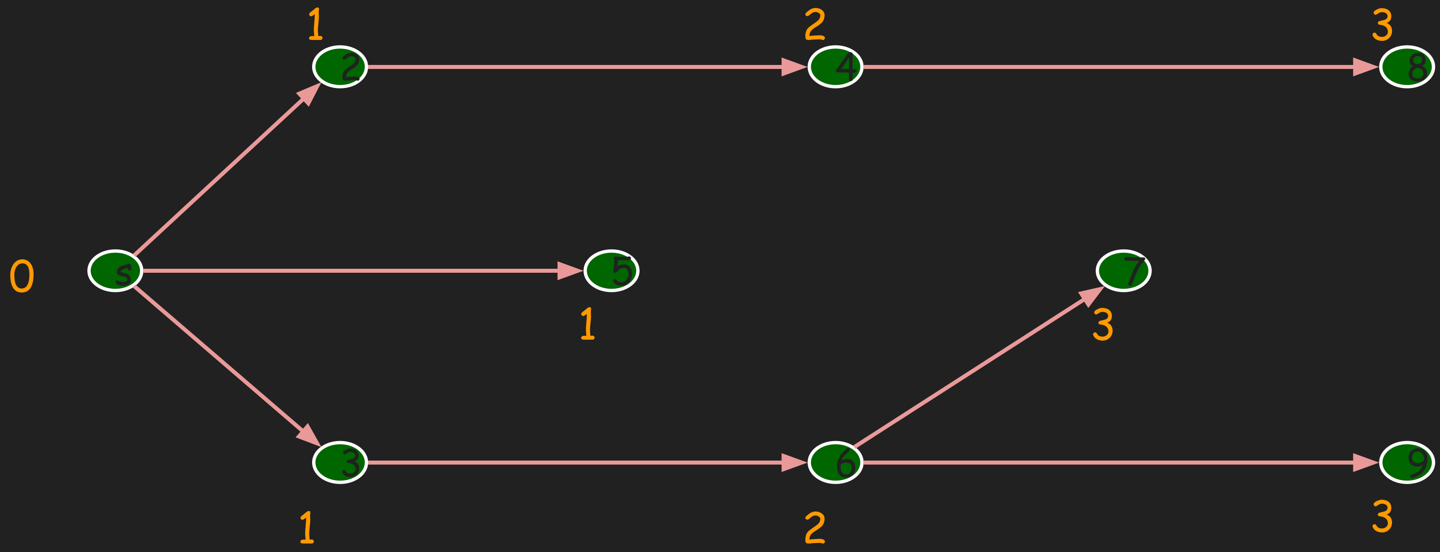
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

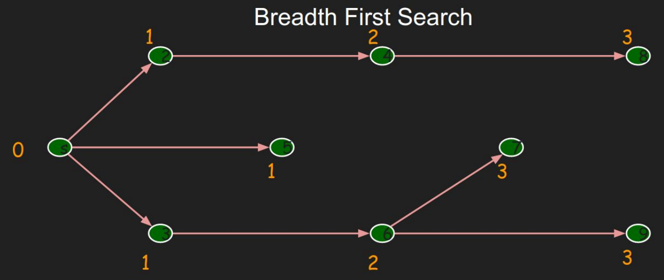
Queue:

Breadth First Search

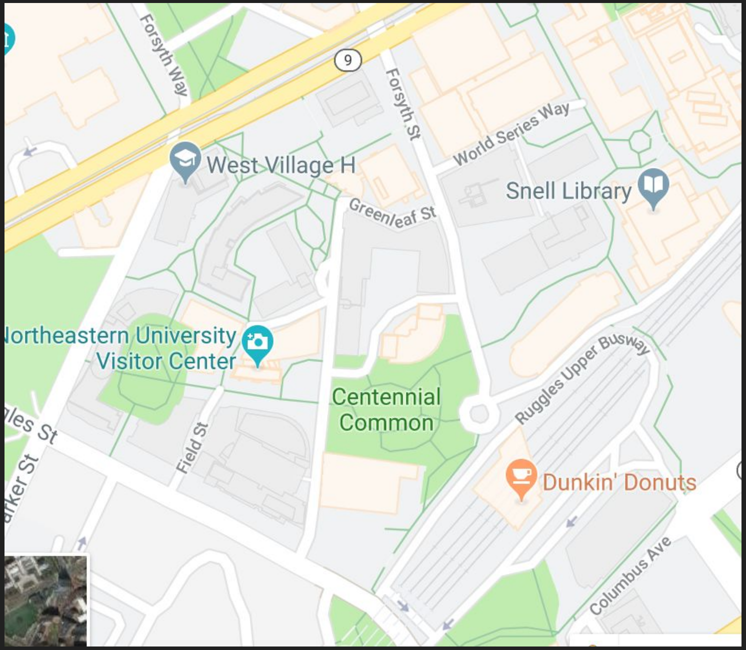


- Starting from s, we visited all (reachable) nodes
- Note: BFS forms a tree rooted at s (BFS Tree)
- For each node x reachable from s → we created a shortest path from s to x

So we have found a shortest path!



- Sort of -- to be continued!
  - We assumed each edge is weighted ‘1’ by default
    - So if every road is a mile in distance we have solved the problem
    - Or perhaps we have found the ‘minimum number of transfers’ to different sources.
  - We’ll keep working on this problem as we move forward--to provide a shortest path solution on a weighted graph



Shortest Path using BFS of Unweighted Graph (1/2)

- Question to the audience: Given a graph, how would I find the minimum number of edges from a given source to destination?
  - i.e. How could I modify BFS to keep track of this for us?

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            u ← G.adjacentVertex(t,e)
            if u is not marked:
                mark u
                enqueue u onto Q
    return none
```

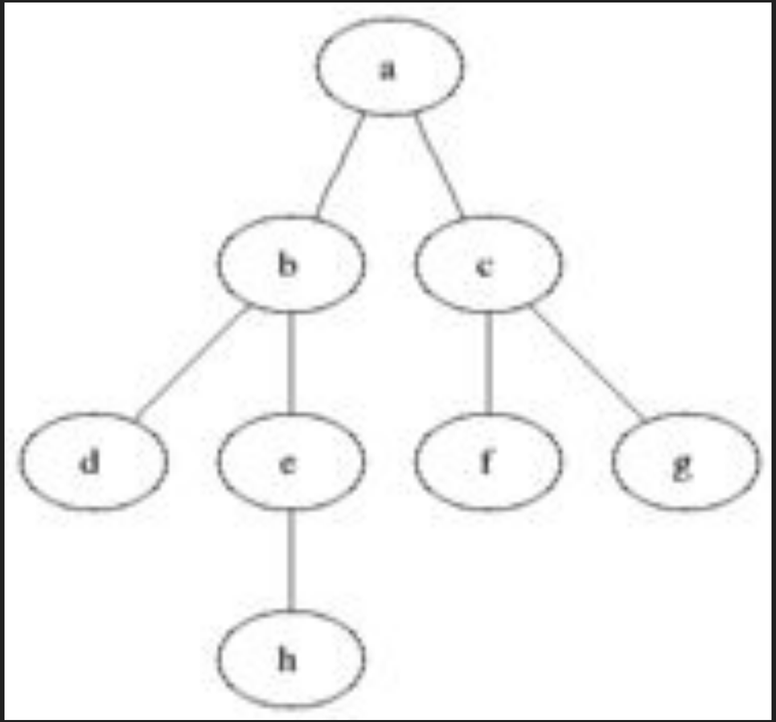
Shortest Path using BFS of Unweighted Graph (2/2)

- Question to the audience: Given a graph, how would I find the minimum number of edges from a given source to destination?
  - i.e. How could I modify BFS to keep track of this for us?
  - Answer: Have an additional field for ‘length’ that keeps track of shortest path from each parent.
    - (i.e. how many steps we have traveled)

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            u ← G.adjacentVertex(t,e)
            if u is not marked:
                mark u
                enqueue u onto Q
    return none
```

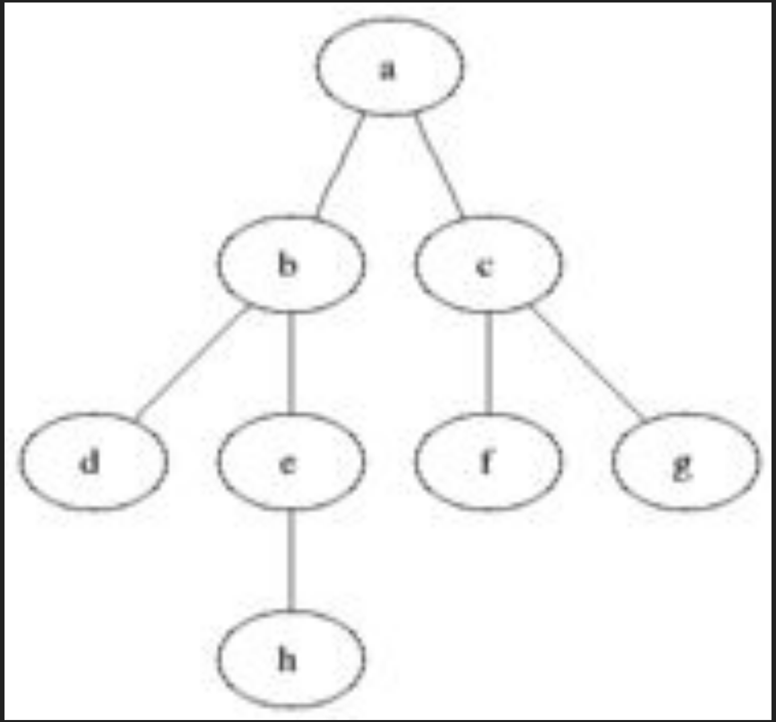
BFS Complexity Analysis (1/2)

- (Question to audience) What do you think the complexity is of a Breadth First Search(BFS)?



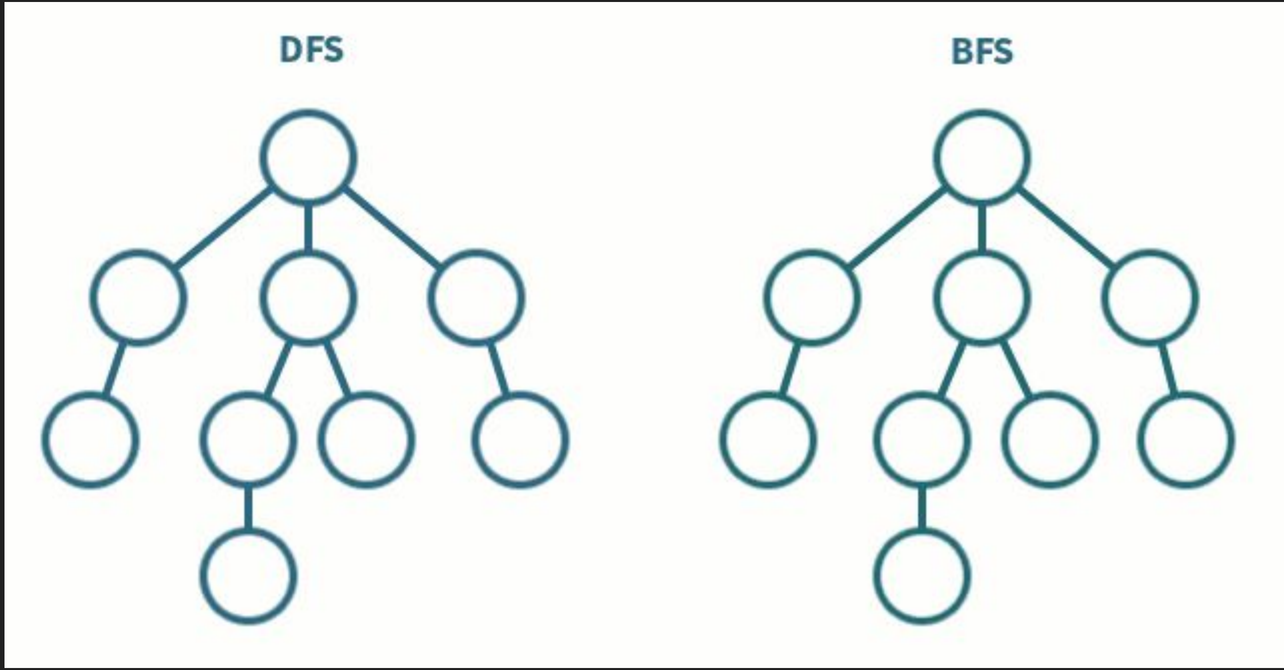
BFS Complexity Analysis (2/2)

- (Question to audience) What do you think the complexity is of a BFS search?
- Answer:
  - Well each node will only get queued one time (`O(V)`)
  - For each node we visit all of its edges (`O(E)`)
  - Makes a total of `O(|V|+|E|)`
  - `(Again, I'm thinking of things in V+E because that's the number of 'steps' or my input size of the data structure representation--usually we were used to working with 'n')`



BFS vs DFS Visual

- Nice comparison showing how ‘DFS’ searches the deepest level in a tree, vs ‘BFS’ which searches all nodes within a level.

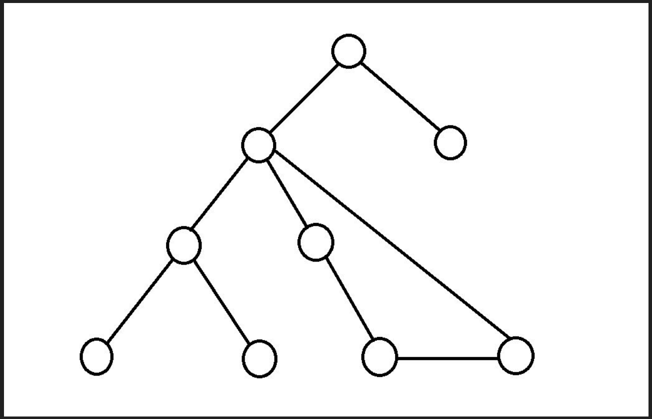


Cycle Detection in Graphs

Cycle Detection in Graphs (1/4)

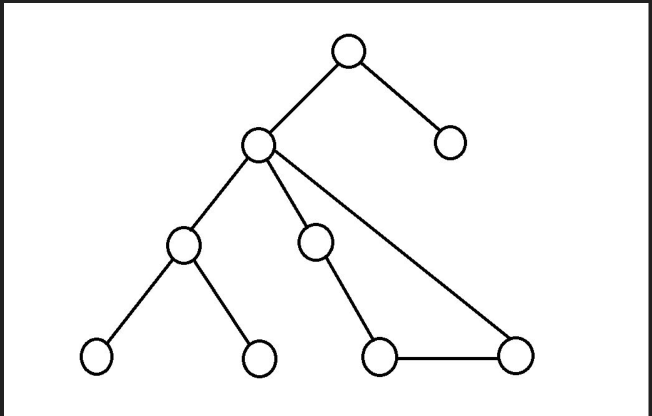
- Let's say I am trying to compute the shortest number of edges in a graph
- (Question to Audience): What happens if I have a cycle in the graph?
  - (i.e. Can I accurately use one of these algorithms?)

Cycle Detection in Graphs (2/4)



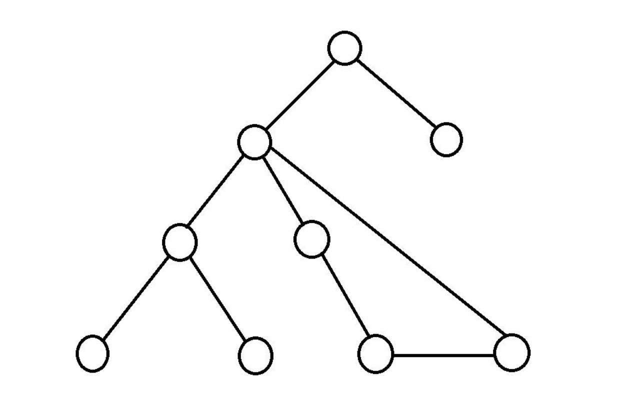
- Let's say I am trying to compute the shortest number of edges in a graph
- (Question to Audience): What happens if I have a cycle in the graph?
  - (i.e. Can I accurately use one of these algorithms?)
  - Since we can repeat the cycle forever in a weighted graph (with say negative weights) we cannot use certain algorithms to find a shortest path.

Cycle Detection in Graphs (3/4)



- (Question to Audience): How would I otherwise detect a cycle?
  - Could you use either BFS or DFS to find it?

Cycle Detection in Graphs (4/4)



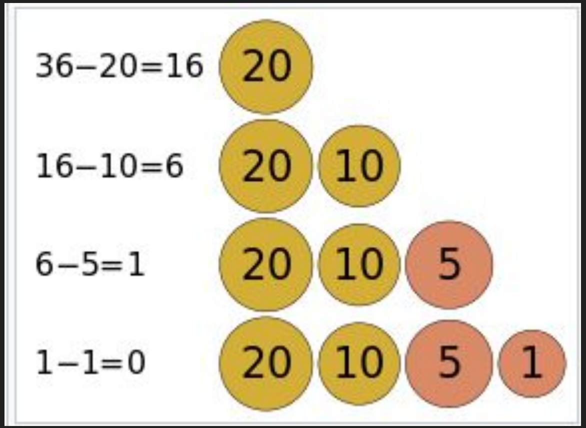
- (Question to Audience): How would I otherwise detect a cycle?
  - Could you use either BFS or DFS to find it?
  - Using DFS, we can detect if we revisit a node that we have already visited. If that is so, then we have a cycle!
    - (Typically DFS would be faster, but you could also use a BFS and see if the node you started from also gets revisited at any point)

Greedy Algorithms

Quick Introduction

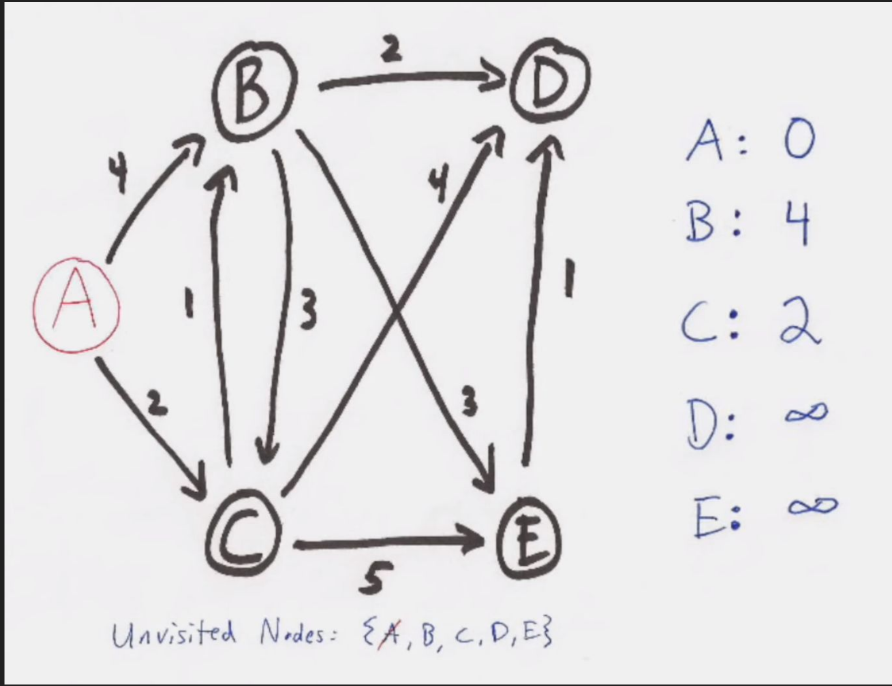
Greedy Algorithm Fundamentals

- A greedy algorithm is another problem solving strategy
  - We previously have learned about 'divide and conquer' as one algorithmic strategy
- Greedy algorithms work by making the 'optimal decision' first
  - An example to the right of returning the fewest amounts of coins to a user is a greedy algorithm
    - (in increments of 20, 10, 5, or 1 cent coins)
- Greedy algorithms are not always optimal, but sometimes provide us a *good enough* solution to our problem.



A Preview of Dijkstra’s Algorithm

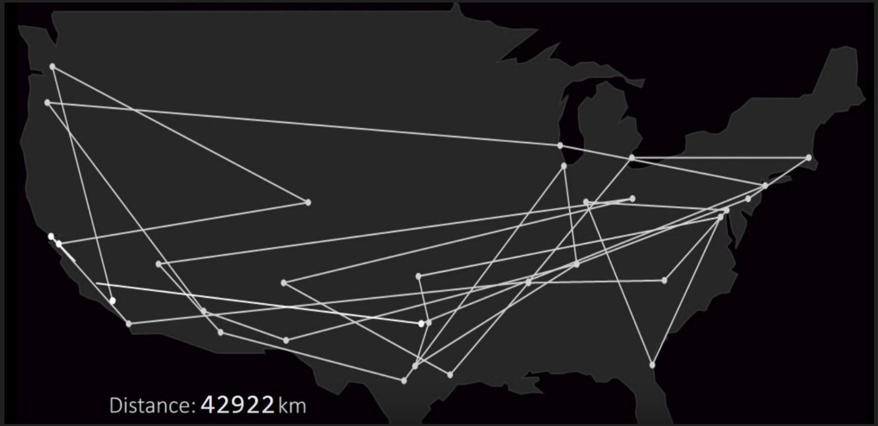
- Sometimes in order to find the ‘shortest path’ we can use a greedy algorithm.
- A preview of the algorithm is given in this video
    - We will investigate it further
- https://www.youtube.com/watch?v=_lHSawdgXpI



Computer Systems Feed

YOUR DAILY FEED

- (An article/image/video/thought injected in each class!)
- Traveling Salesman Problem (often abbreviated ‘TSP’) https://www.youtube.com/watch?v=SC5CX8drAtU
    - “Given a list of cities, what is the shortest possible route that visits each city”
    - https://en.wikipedia.org/wiki/Travelling_salesman_problem



Algorithm, Data Structure, and Proof Toolboxes

For this course, I want you to be able to see how each data structure and algorithm is different.

- For data structures learn how each restriction on how we organize our data causes tradeoffs
- For algorithms, think about the higher level technique

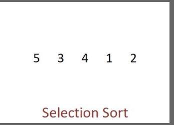
Algorithm Toolbox: Searches and Sorts

Comparison Sorts

Bubble Sort - $O(n^2)$

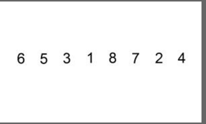
Swap adjacent elements and ‘bubble’ up element

Selection Sort - $O(n^2)$



Search for minimum element and place in ordered position amongst unordered elements

Insertion Sort - $O(n^2)$



Select each element and place in its sorted position amongst all elements that have been previously placed

Heap Sort- $O(n*\log_2(n))$

Divide and Conquer Sorts

Merge Sort- $O(n*\log_2(n))$

Randomized Algorithms

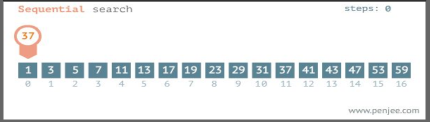
QuickSort- $\Theta(n*\log_2(n))$ (“theta”)

$O(n^2)$ - in the worst case

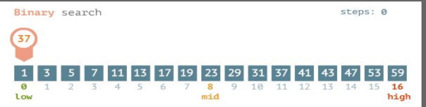
Searches

Linear Search - $O(n)$

Search and compare each element one at a time



Binary Search - $\log_2(n)$



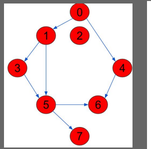
Search sorted data from midpoint, eliminate values less than or greater than ‘element’ you are search for each step until we match the mid

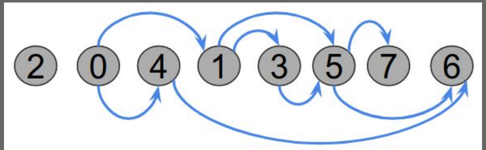
Algorithm Toolbox: Trees and Graphs

Tree Sorts

Topological Sort

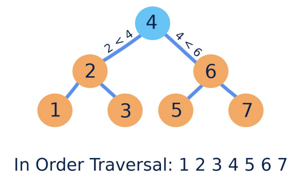
Generates a possible linear ordering of nodes from a Tree by performing a DFS on each unvisited node. O(|V|+|E|)





Tree and/or Graph Searches

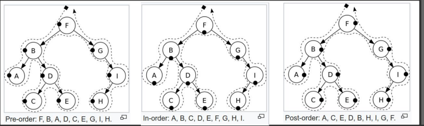
Binary Search Tree



Data structure containing a left and right child
$\Theta(\log_2(n))$ for search, insertion, and deletion

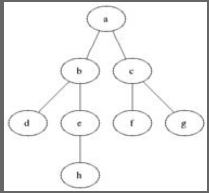
Depth-First Search (DFS)

Traverse graph (or tree) as far as possible in a direction, storing nodes in a stack. O(|V|+|E|)



Breadth-first search (BFS)

Traverse graph (or tree) as widely as possible (level-order traversal) storing each nodes neighbors in a queue.
O(|V|+|E|)



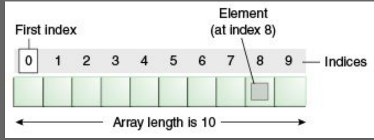
And many more! https://en.wikipedia.org/wiki/List_of_data_structures

Data Structure Toolbox: Fundamental Structures

Associative Containers

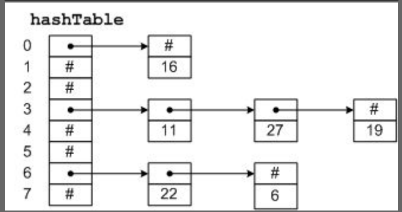
Arrays -

A contiguous block of memory, random access O(1)



Hashmap (chained implementation) -

Associative Data Structure with key/value pairs and a 'hash function'



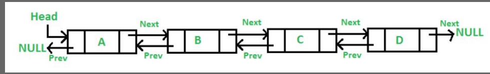
Sequence Containers

Linked Lists -

A 'chain' of nodes, can traverse in one direction

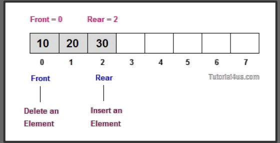
Doubly Linked Lists -

A 'chain' of nodes, can traverse in both directions



Queues -

A First in, First out data structure (FIFO)

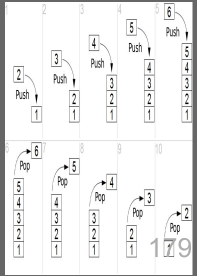


Priority Queues -

Uses a min-heap or max-heap and promotes min or max element to front of queue.

Stacks -

A Last in, last out data structure (LIFO)



Data Structure Toolbox: Tree and/or Graphs

Graph and Tree Data Structures

Adjacency Matrix -

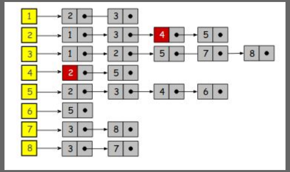
Vertices (columns) / Vertices (rows)

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

2D array holding edge weights (or 0 if unconnected)

Space complexity of O(|V|*|V|)

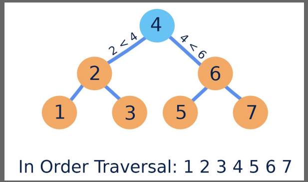
Adjacency List -



An array of lists or list of lists where each list indicates connectivity

Space complexity of O(|V|+|E|)

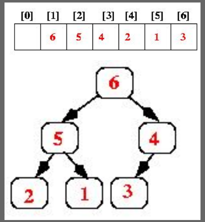
Binary Tree



Data structure containing a left and right child
$\Theta(\log_2(n))$ for search, insertion, and deletion

Heaps

Binary Heaps



Array-based structure to hold a complete binary tree

Proof Toolbox

- Our tools so far!

Notation

```
∀n - "for all"
| - "such that"
n ∈ ℤ - "n is an element of the integers"
```

Proof Techniques

- Proof by Case
  - Enumerate or test all possible inputs
- Proof by Induction
  - Show that two cases hold
- Proof by Invariant
  - Step through 4 steps of algorithm
- Big-O Analysis
  - Prove run-time complexity

- Recurrence
  - Can be solved with Subsitution Method
- Recurrence Tree
  - "A Visual Proof" (Somewhat informal)
- Master Theorem
  - Proven by definition
- Substitution Method
  - (Works for any recurrence)

Building Blocks

Definition - Something given, we can assume is true

e.g. let x = 7

Proposition - A true or false statement

e.g.
1+7 = 7 FALSE
2+7 = 9 TRUE

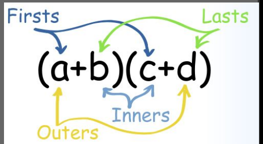
Predicate - A proposition whose truth depends on its input. It is a function that returns true or false.

"P(n) ::= "n is a perfect square"
P(4) thus is true, because 4 is a perfect square
P(3) is false, because it is not a perfect square.

Math Toolbox

Mathematics

Multiplication



$(a+b)(c+d) = ac + ad + bc + bd$

Logs

Logs -

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Usually we work in log base 2, i.e. $\log_2(n)$. The change of base formula is given below.

In this course we think about logs as ‘halving’ the number of our sub-problems (or search space).

Notation

Pi Production Notation

$$n! = \prod_{i=1}^{n} i.$$

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

Big-O: O(n) - “Worse Case Analysis or upper bound”
Big-Theta: Θ - “Average Case Analysis”
Big-Omega Ω - “Best Case or lower bound”

Factorial (!) $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Summation (“sigma”)

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$$

i = index of summation

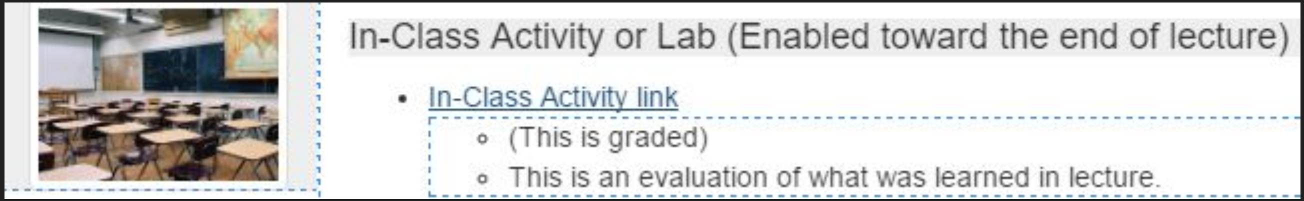
n = upper limit of summation

This lecture in summary

- We have looked at the Graph Data Structure
- Trees are a special instance of graphs (Linked Lists as well)
- We can sort some instances of graphs (topological sort)
- We can explore graphs using DFS, BFS, and Dijkstra’s to reveal properties of certain graphs
    - (i.e. Dijkstra’s to find a shortest path of a weighted graph)

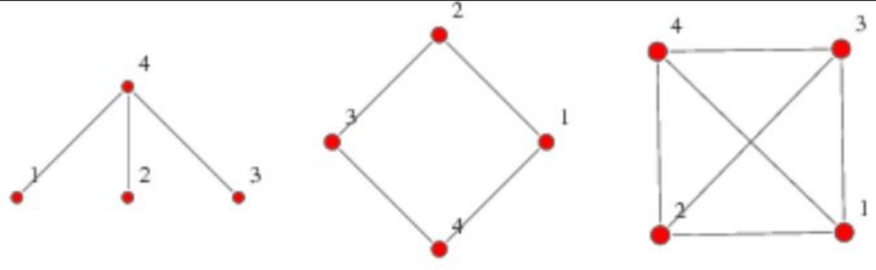
In-Class Activity

1. Complete the in-class activity from the schedule
    a. (Do this during class, not before :))
2. Please take 2-5 minutes to do so
3. These make up a total of 5% of your grade
    a. We will review the answers shortly



For today's lab

- You will be working on two exercises
  - 1. Navigating an adjacency matrix
  - 2. Exploring topological sort



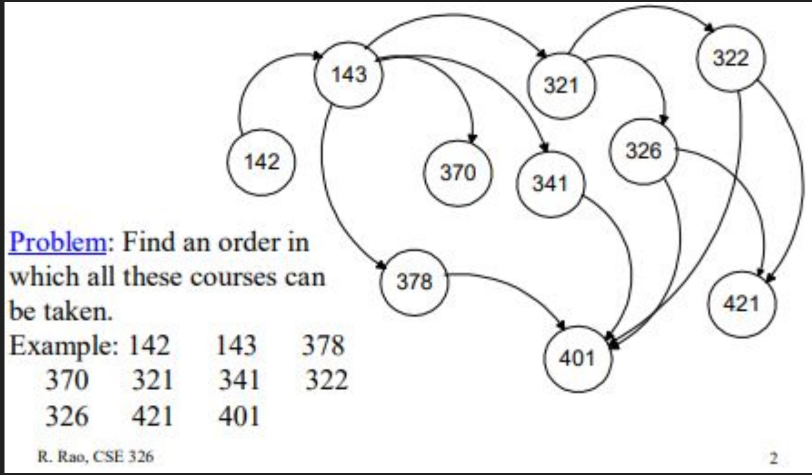
$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Lab Time!

Old Examples

Topological Sort - Real world use cases (1/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another, like in baking)
  - Finding an order of required classes to take before graduation

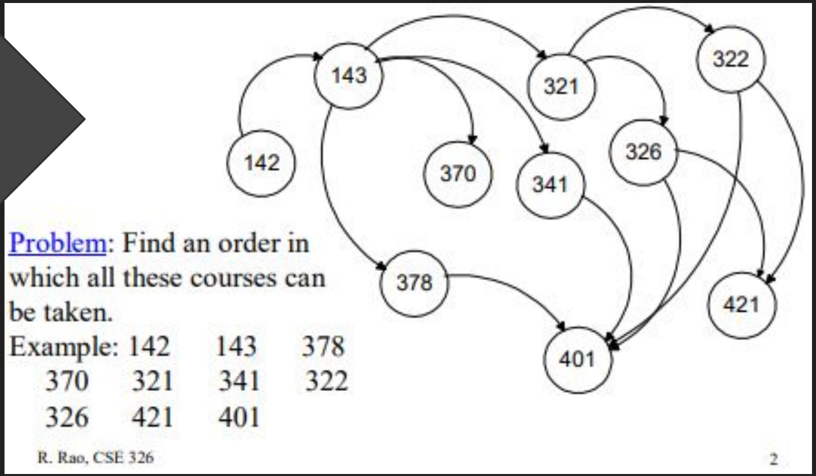


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (2/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you ‘mark’ it off.

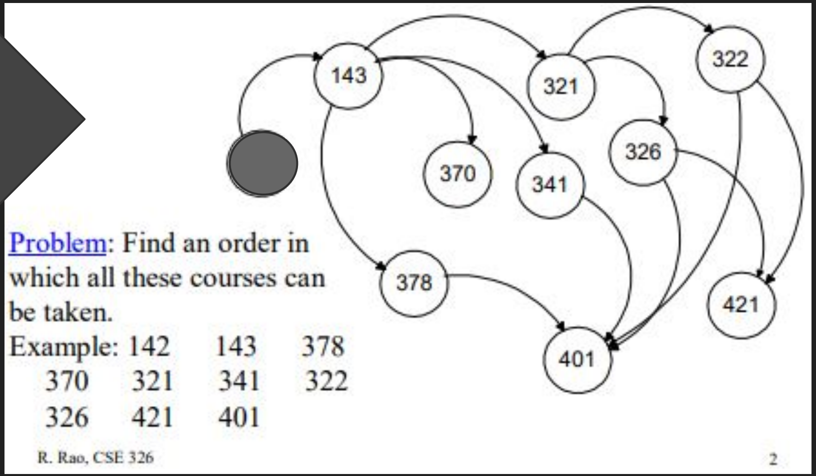


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (3/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you ‘mark’ it off.

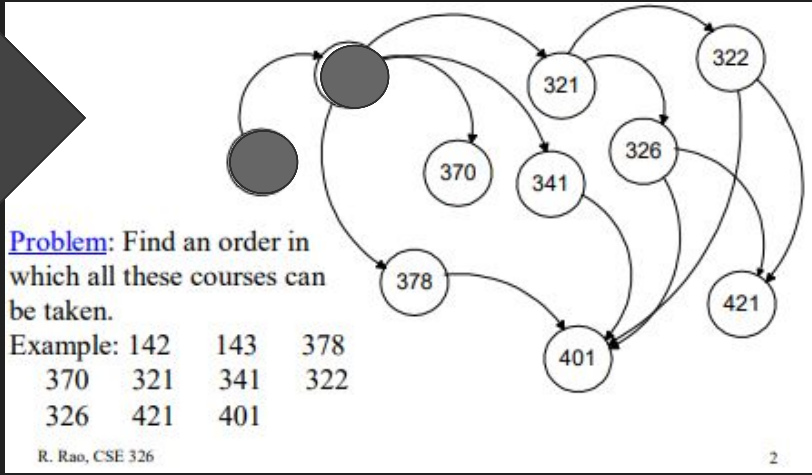


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (4/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you 'mark' it off.

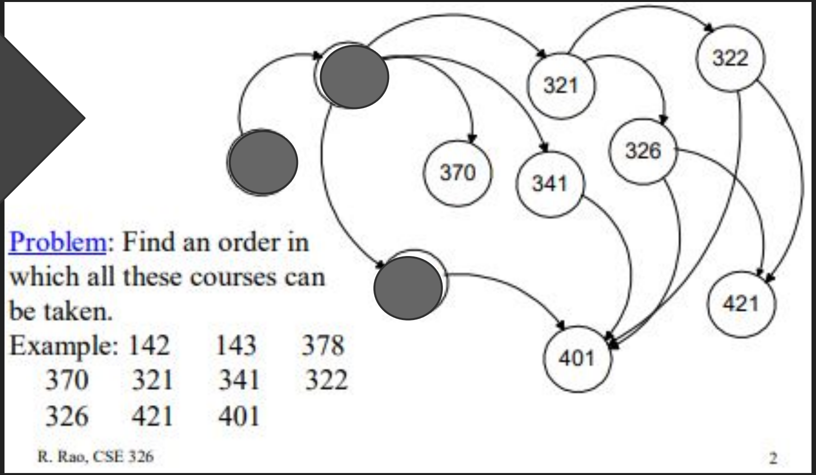


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (5/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you ‘mark’ it off.

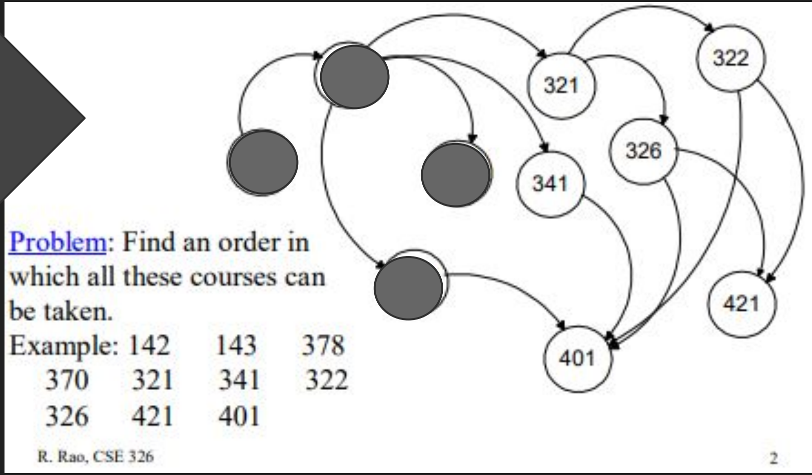


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (6/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you ‘mark’ it off.

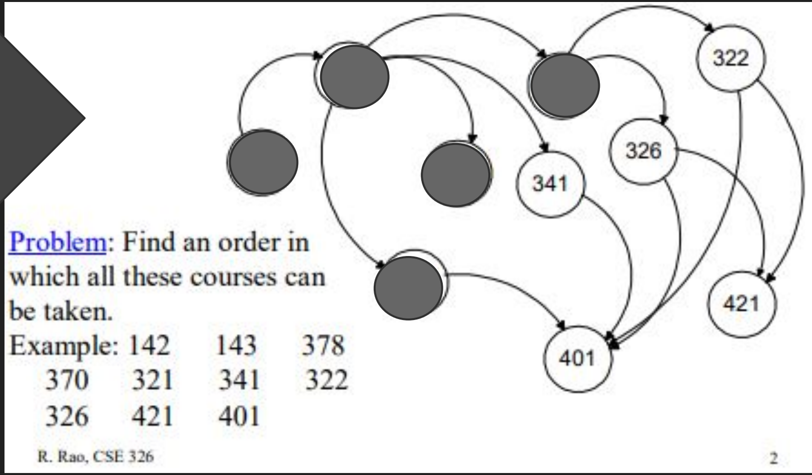


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (7/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you 'mark' it off.

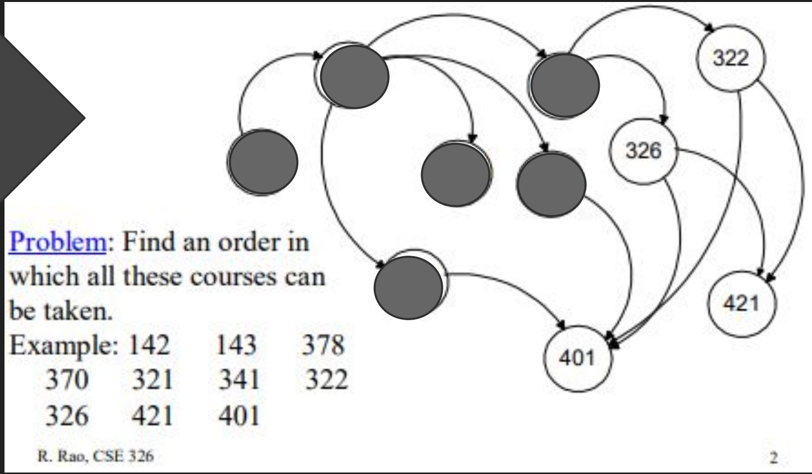


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (8/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you 'mark' it off.

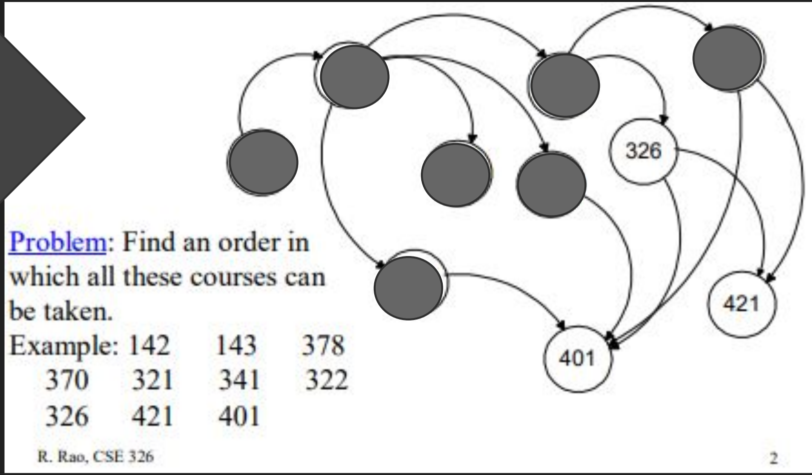


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (9/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you 'mark' it off.

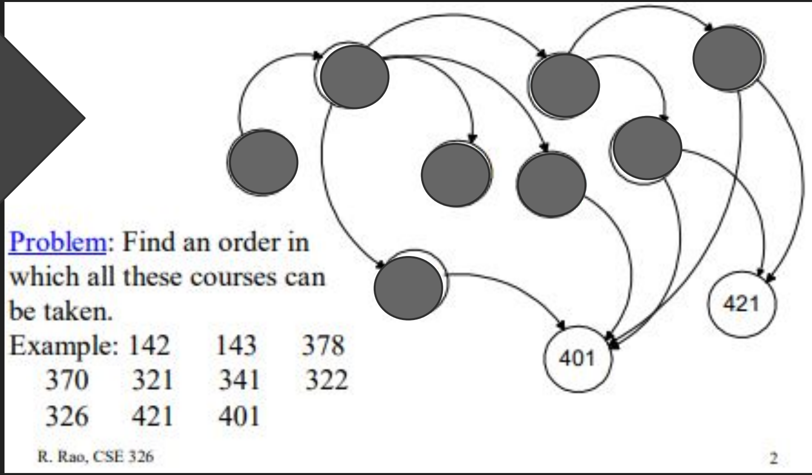


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (10/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you ‘mark’ it off.

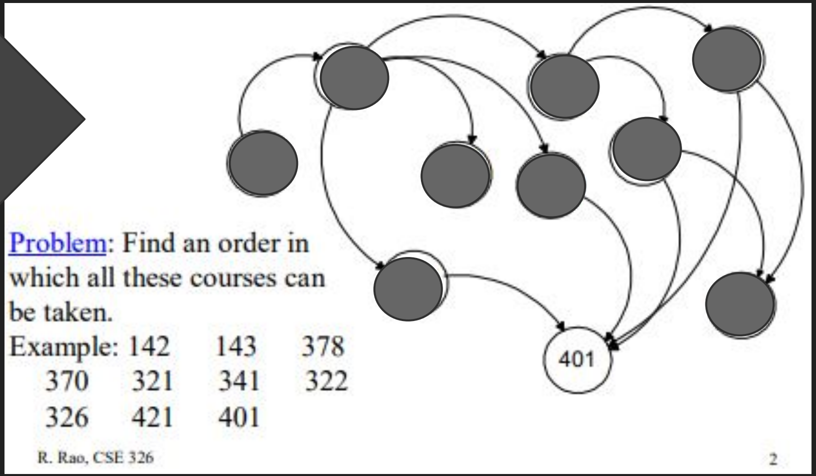


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (11/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you ‘mark’ it off.

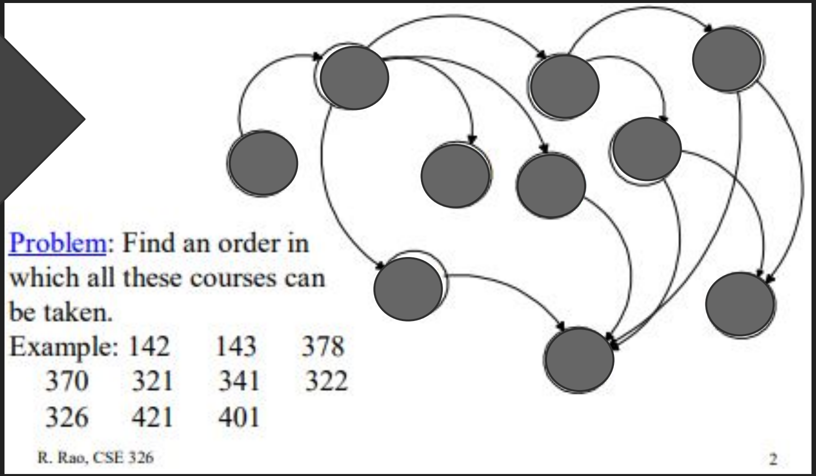


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (12/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you ‘mark’ it off.

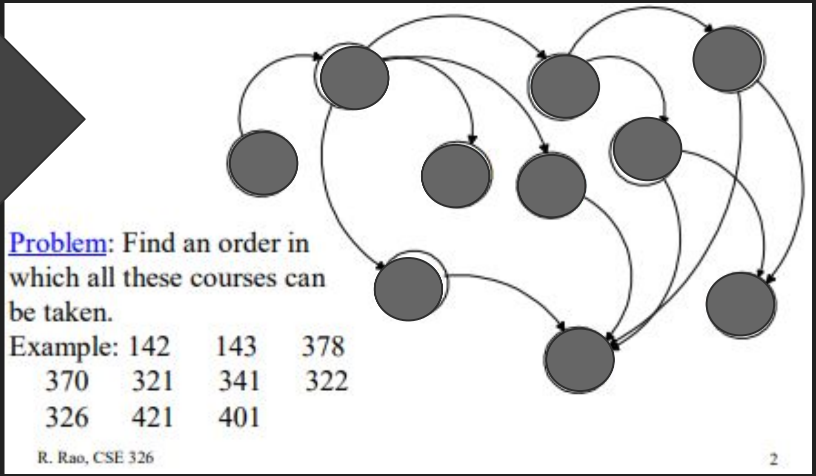


Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

Topological Sort - Real world use cases (13/13)

- Topological sort is used for things like
  - Task scheduling (i.e. which task must precede another)
  - Finding an order of required classes to take before graduation

As you finish each class, you 'mark' it off.



Example from:
https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf