SE 2205a: Data Structures and Algorithm Design

Unit 7 – Part 1: Graph Fundamentals

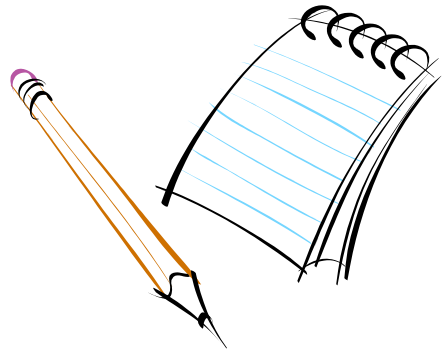
Dr. Quazi M. Rahman, Ph.D, P.Eng, SMIEEE
Office Location: TEB 263
Email: QRAHMAN3@uwo.ca
Phone: 519-661-2111 x81399

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

"Genius is 1% talent and 99% percent hard work..."
~ Albert Einstein

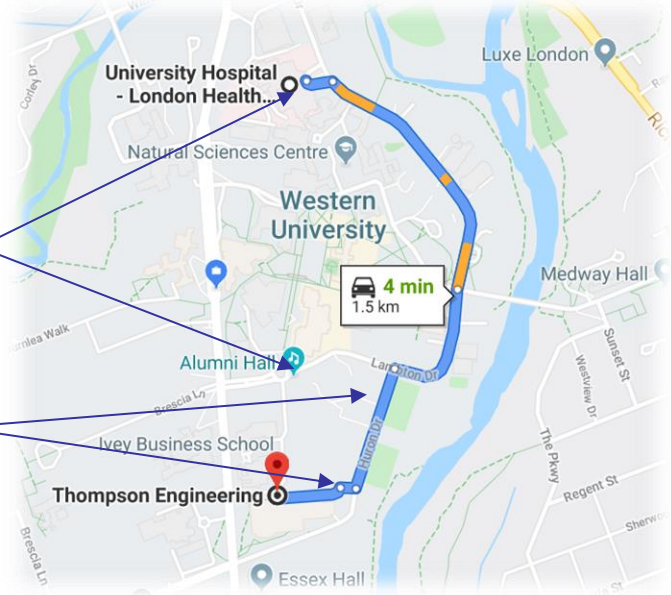
Outline

- Define Graph
- Graph Terminologies
- Graph Implementation
- Graph Traversal

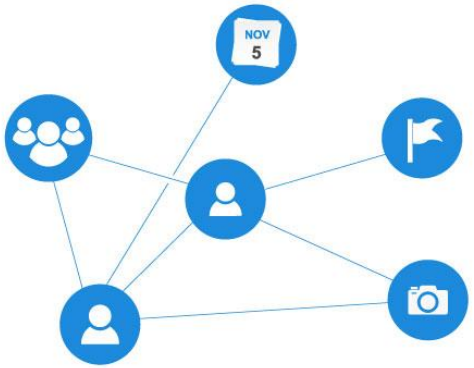


What is a Graph in Data-Structure?

- A Graph is a non-linear data structure consisting of distinct nodes and edges.
  - The nodes are sometimes referred to as vertices, and the edges (discussed in 'Tree') are lines or arcs that connect any two nodes in the graph.
- Graphs in data-Structure include
  - Trees (All trees are graphs)
  - Networks of nodes connected by paths (e.g., city or telephone network, circuit network, computing network, social media network such as Facebook-friends, followers of Instagram, TikTok etc.)
  - A road/air traffic map (e.g, google map, flightradar24 etc.)
- NOT *bar-graphs, pie charts*, etc.



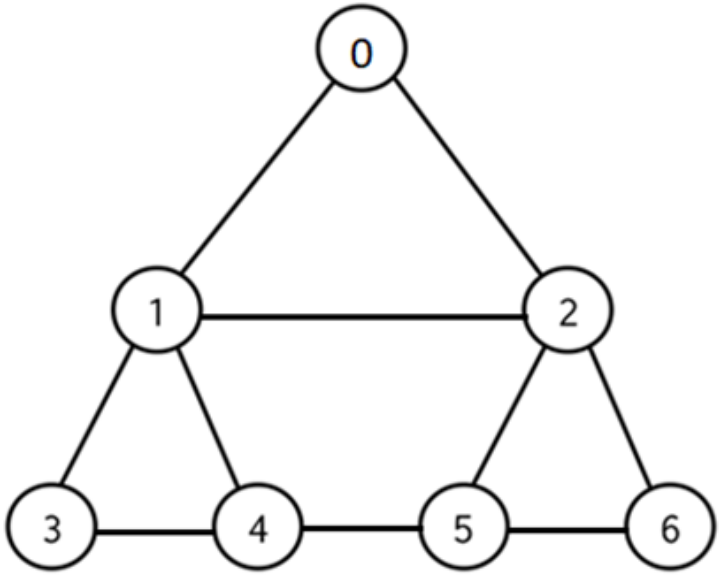
Graph Example: Google map



Graph Example: A Social Media network

What Data Items does a Graph Contain?

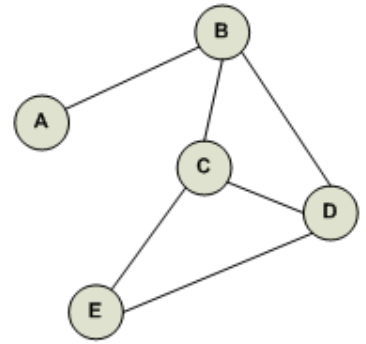
- A Graph (G) data structure contains information on its Vertices (V) and Edges (E) that is, G = {V,E}, where:
  - V is a collection of vertices
    - V = {0, 1, 2, 3, 4, 5, 6} (See the graph)
  - E is a collection of edges as ordered pairs of vertices (v1, v2), represented by
    - E = {(0,1), (0,2), (1,0), (1,2), (1,3), (1,4), (2,0), (2,1), (2,5), (2,6), (3,1), (3,4), (4,1), (4,3), (4,5), (5,2), (5,4), (5,6), (6,2), (6,5)}

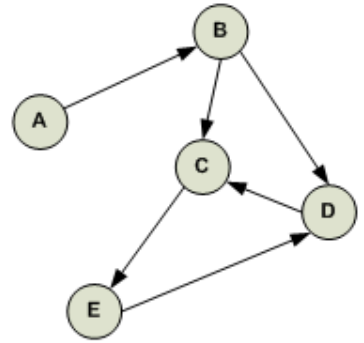


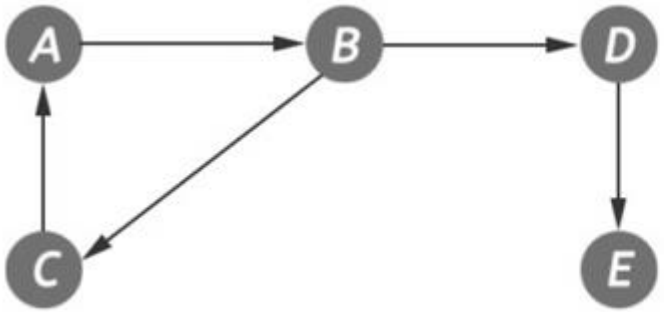
Note: An ordered pair is a pair of numbers in a specific order. E.g, (0, 1) is not equivalent to (1,0)

Graph Terminology: Undirected and Directed Graphs

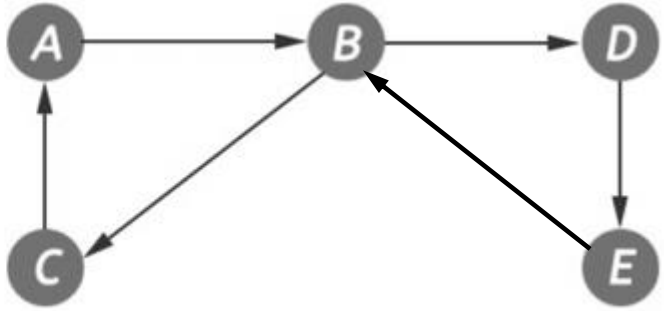
- Undirected graphs: edges do not have direction. The edges indicate a two-way relationship, in that each edge can be traversed in both directions.
- Directed graphs (A.K.A Digraph): edges have direction. The edges indicate a one-way relationship, in that each edge can only be traversed in a single direction.
  - A directed graph is strongly connected if there is a direct or indirect path between any pair of nodes.







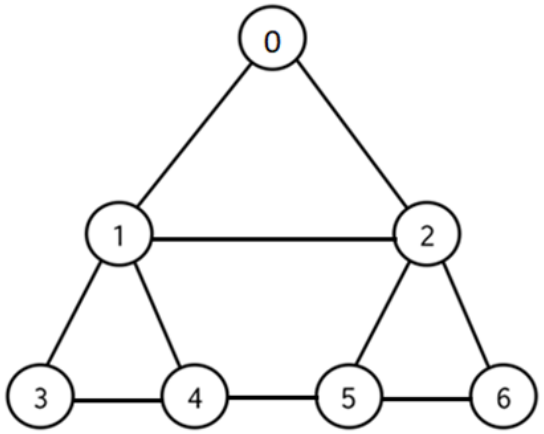
Not Strongly Connected Graph



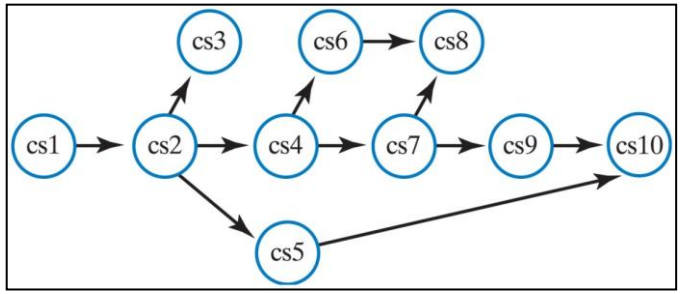
Strongly Connected Graph

Graph Terminology : Path

- Path: A sequence of edges that connects two vertices (source and destination vertices) in a graph. E.g., paths from 0 to 5 in *graph A* are (0, 1, 4, 5), (0, 2, 5) etc.
- A path in a directed graph is know as directed path. In a directed graph the direction of the edges must be considered when visiting any node/vertex.
- A cycle is a path that begins and ends at same vertex.
  - Simple cyclic path does not pass through any vertex more than once (other than the start/stop vertex). E.g., (0, 1, 4, 5, 2, 0) is a simple cyclic path in *graph A*.
  - A graph with no cycles is an acyclic graph
  - Note: A tree is always acyclic, but in a tree, no two parents can have the same child.



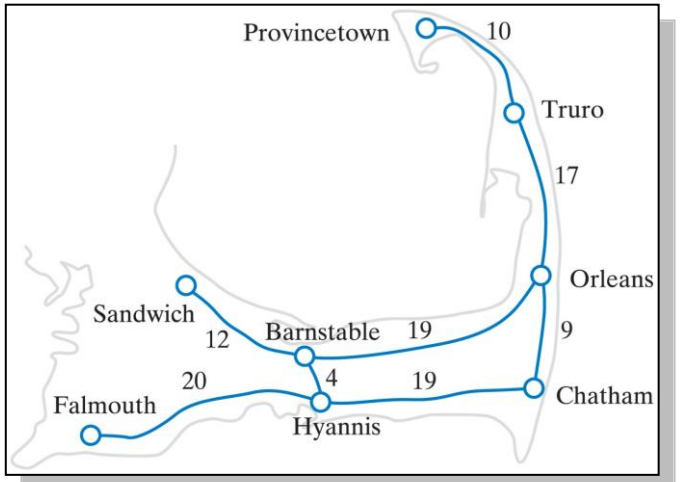
Graph A



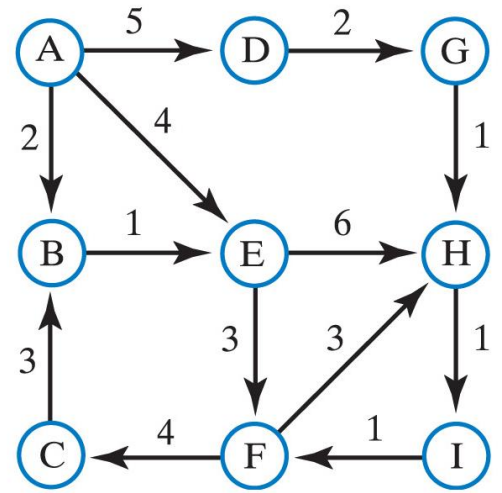
Graph B: An acyclic graph

Graph Terminology: Weighted Graph

- A weighted graph has values (weights) on its edges
- Examples of weights
  - Distance between nodes on a map
  - Driving/flying time between nodes
  - Cost associated to visiting a destination node from a source node
  - Energy required (walking/running) in visiting a destination node from a source node
- The total weight of a path in a weighted graph is the sum of the weights of its edges.
- The shortest path problem refers to finding a path between two nodes such that the total weight is minimum.



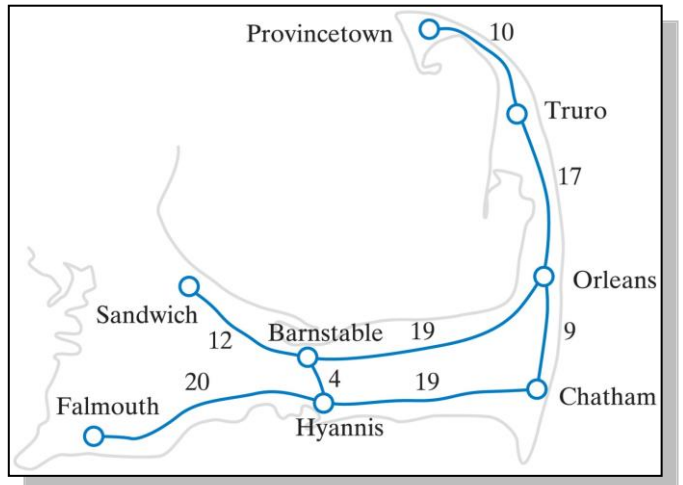
Undirected Weighted Graph



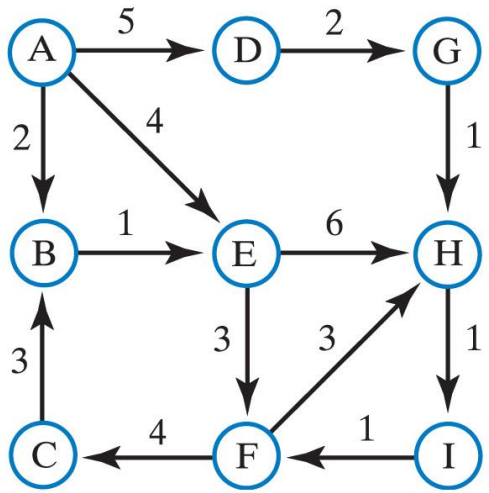
Directed Weighted Graph

Graph Terminology: Adjacent Vertices (AKA Neighbors)

- Undirected Graph:
  - When two vertices, next to each other are joined by an edge these two are known as adjacent vertices or neighbors (*Graph x*: Orleans and Chatham are adjacent, but Orleans and Sandwich are not)
- Directed Graph:
  - In a directed graph, vertex *i* is adjacent to vertex *j*, if a directed edge begins at *j* and ends at *i*.
  - In *Graph y*, Vertex B is adjacent to A (A can visit B), but A is not adjacent to B (B can not visit A). (One way traffic concept)



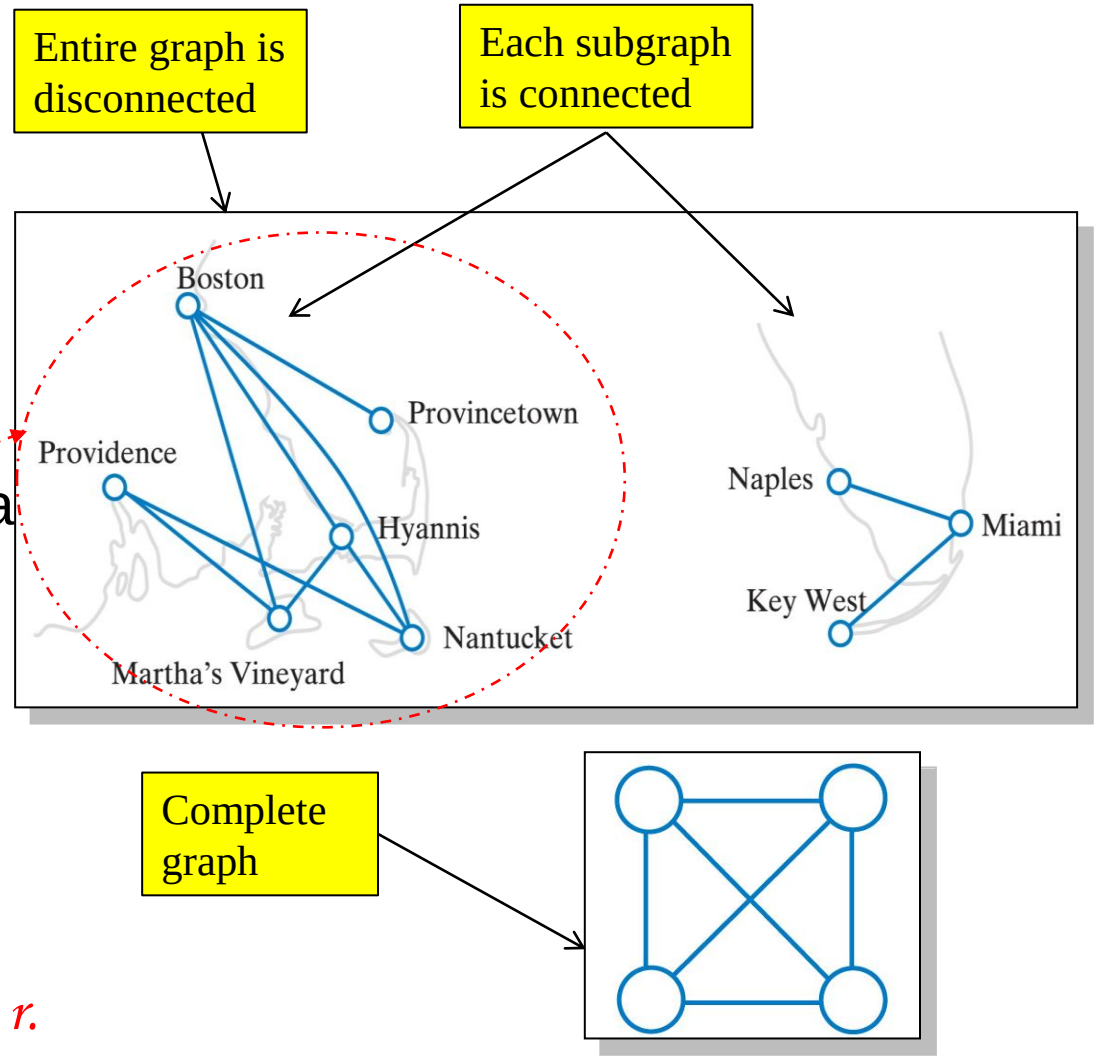
Graph x

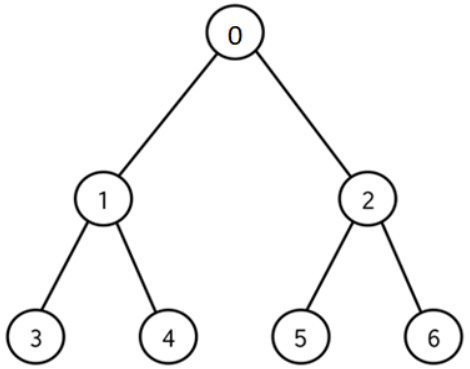


Graph y

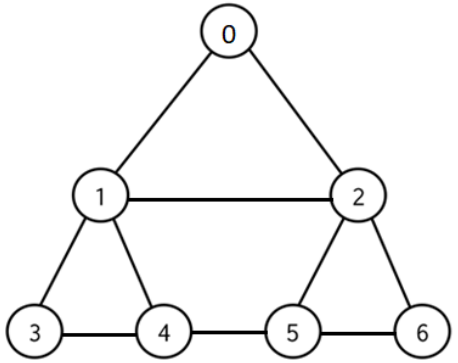
Graph Terminology: Connected, Disconnected and Sub-Graphs, And Spanning Tree

- A connected graph
  - Has a path (sequences of edges) between every pair of distinct vertices
- A complete graph
  - Has an edge between every pair of distinct vertices
- A disconnected graph - Not connected
- A subgraph is a portion of a graph that itself is a graph
- A spanning tree of graph G is a connected subgraph of G, and this subgraph is a tree that contains all vertices of G (*in-class example*).
  Application: Computer network protocol etc.
- *Note: A directed graph G contains a directed spanning tree rooted at r if and only if all vertices in G are reachable from r.*





Tree vs. Graph



Tree	Graph
A Tree is a hierarchical model.	Graph is a network model.
A tree is a connected graph without cycles.	Graphs can have cycles.
All trees are graph.	All graphs are not trees.
Trees have root-node, and the nodes have parent-child relationship.	Graphs have no concept of either root node or parent-child relationship between the nodes (vertices).
Tree can be traversed using Depth-first traversal method (pre, post or in-order) or Breadth-Frist traversal (level order) method.	Graph can be traversed using Depth-first traversal method (mainly preorder) or Breadth-Frist traversal (level order) method.
“Visit a node for a tree” means “process the node's data”.	“Visit a node (vertex) for a graph” means “mark the node (vertex) as visited”

Graph Representation

- Graph is generally represented using one of the following data structures:
  - Adjacency matrix or
  - Adjacency list

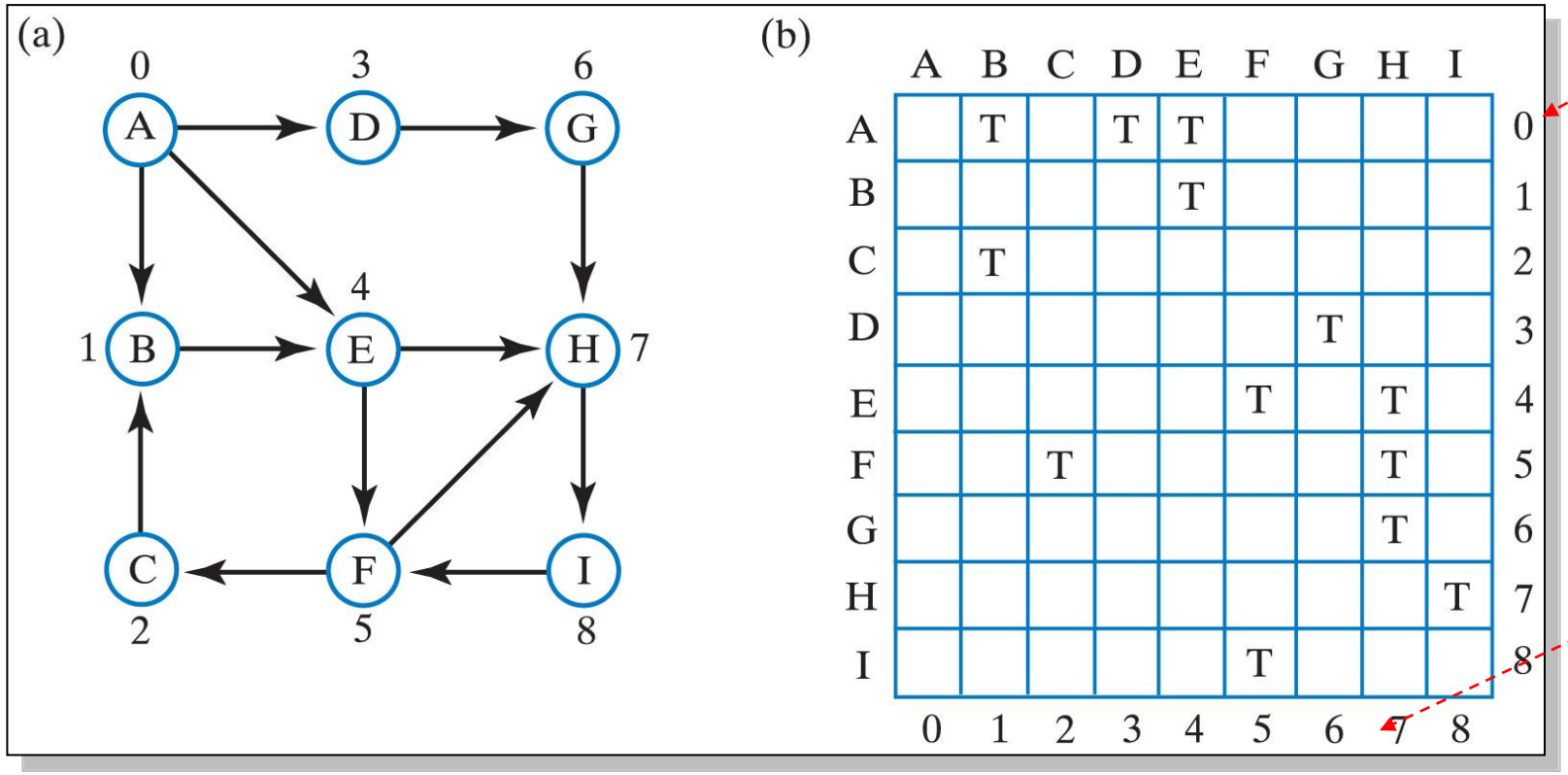
- These two ideas are discussed next….

The Adjacency Matrix

- An *n* x *n* matrix for a graph with *n* vertices, is known as the adjacency matrix that has the following properties:
  - Each row-column pair corresponds to a vertex in the graph
  - Element a_{ij} indicates whether an edge exists between vertex *i* and vertex *j*
- Adjacency matrix uses fixed number of spaces
  - Depends on number of vertices
  - Does not depend on number of edges
- Elements of the matrix contain
  - Boolean (T or F) data or integer (1 or 0) data for un-weighted graph (It depends on the implementation).
  - Edge weights for weighted graph when edges exist, otherwise, infinity (no edge)
- Adjacency Matrix is a good choice for a dense graph (graph with many edges).

The Adjacency Matrix for an unweighted and directed graph with T/F (F is not put on the Matrix for clarity)

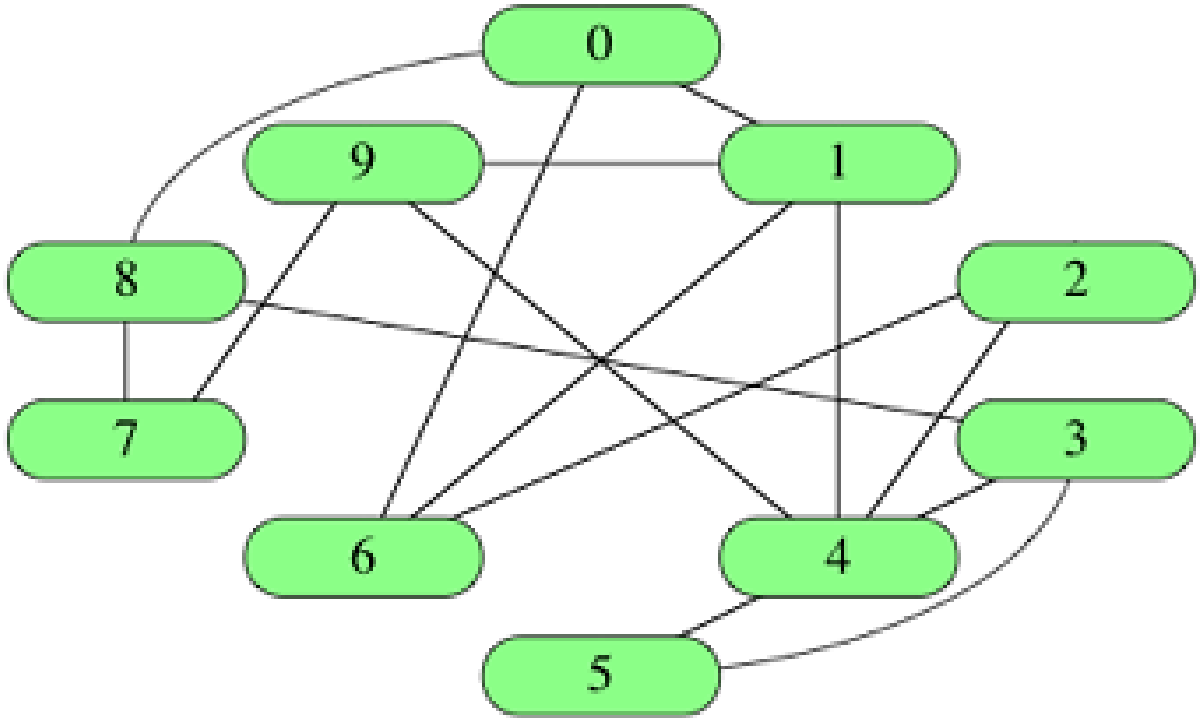
Homework: Create a spanning tree from the graph given in (a), starting from vertex B, and visit alphabetically. Repeat the task for the adjacency matrix in (b). Starting from the vertex A and visit the other vertices Alphabetically. For both cases use breadth first and depth first traversal.



(a) A directed un-weighted graph and (b) its adjacency matrix.

Note: Although a directed edge exists from vertex A to B, the converse is not true. Therefore, a_{10} is false even though a_{01} is true. For an undirected graph adjacency matrix is symmetric ($a_{01} = a_{10} = True$)

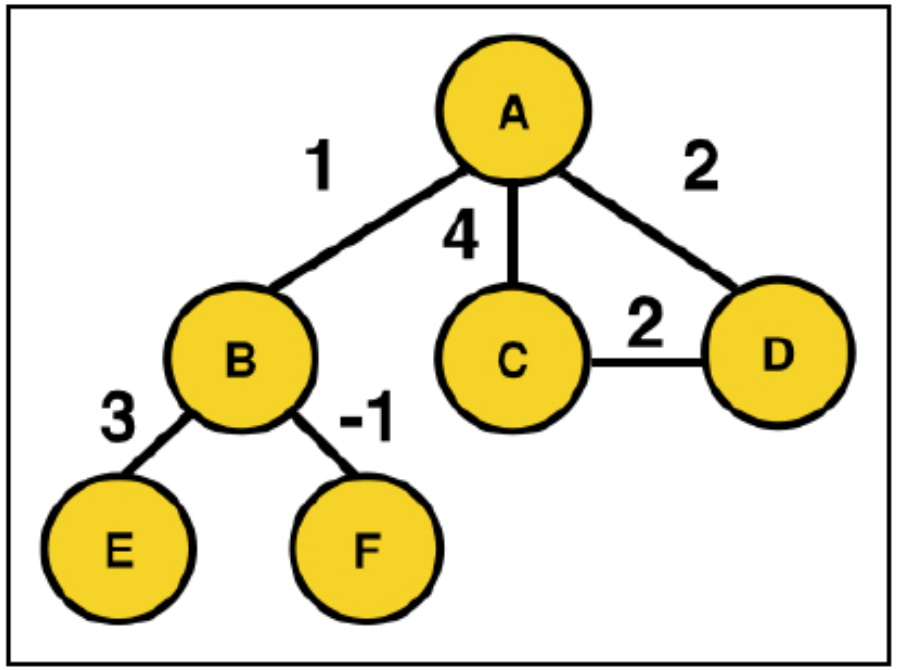
The Adjacency Matrix for an unweighted and Undirected Graph with 1 and 0



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

The Adjacency Matrix for a weighted and Undirected Graph

Weighted Undirected Graph *G*

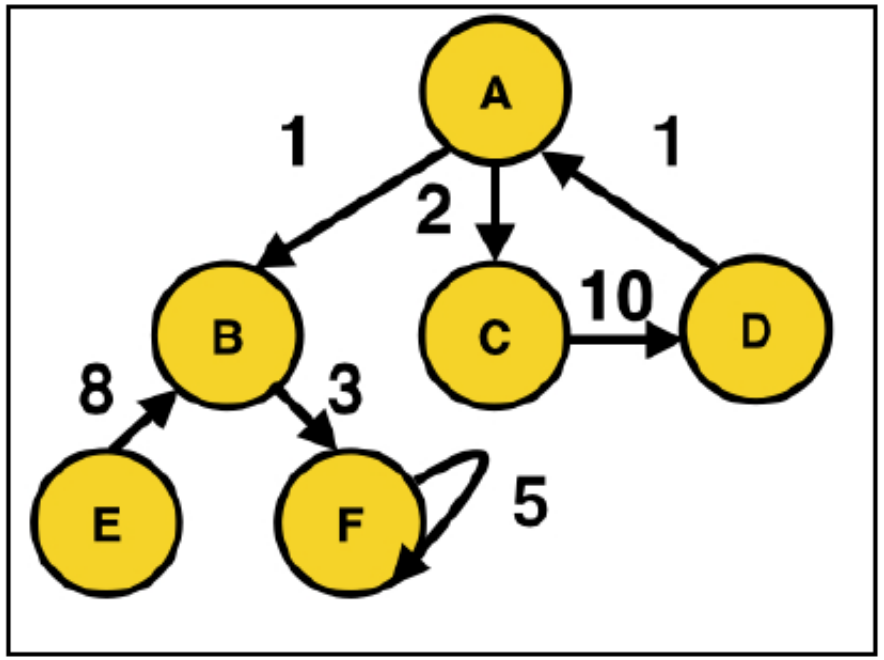


Adjacency Matrix for *G*

	A	B	C	D	E	F
A	0	1	4	2	0	0
B	1	0	0	0	3	-1
C	4	0	0	2	0	0
D	2	0	2	0	0	0
E	0	3	0	0	0	0
F	0	-1	0	0	0	0

The Adjacency Matrix for a weighted and directed Graph

Weighted Directed Graph *G*



Adjacency Matrix for *G*

	A	B	C	D	E	F
A	0	1	2	0	0	0
B	0	0	0	0	0	3
C	0	0	0	10	0	0
D	1	0	0	0	0	0
E	0	8	0	0	0	0
F	0	0	0	0	0	5

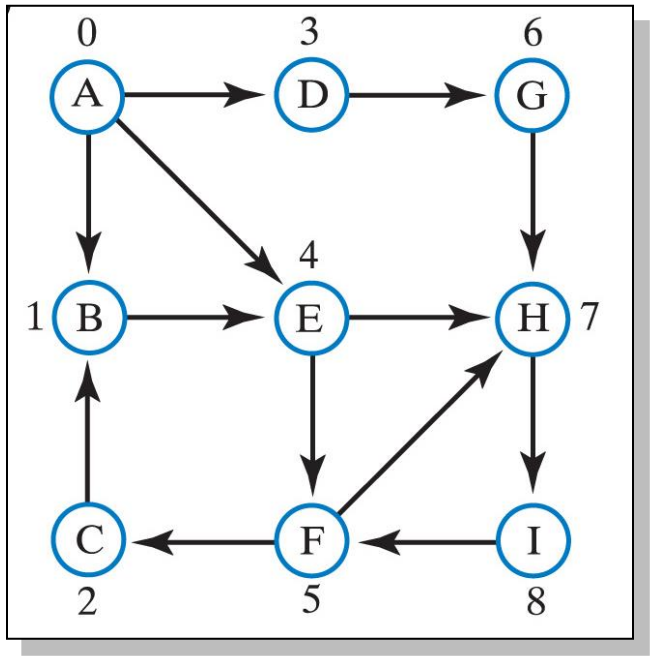
Worst-Case Time Complexity: Graph with Adjacency Matrix

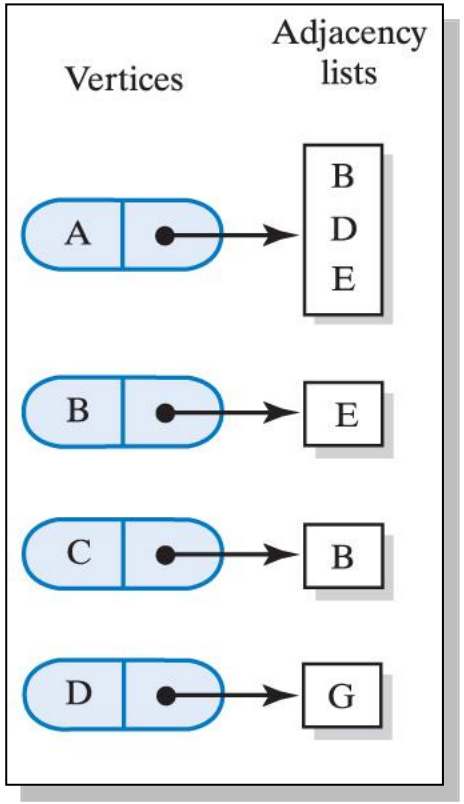
- Presence of an edge between two vertices can be known immediately [O(1)]
- All neighbors of a vertex can be found by scanning entire row for that vertex [O(n)]
- Add/remove a vertex: need to change the size of the matrix, copy everything from the old to the new one by traversing every row [$O(n^2)$]
- Add/remove an edge [O(1)]

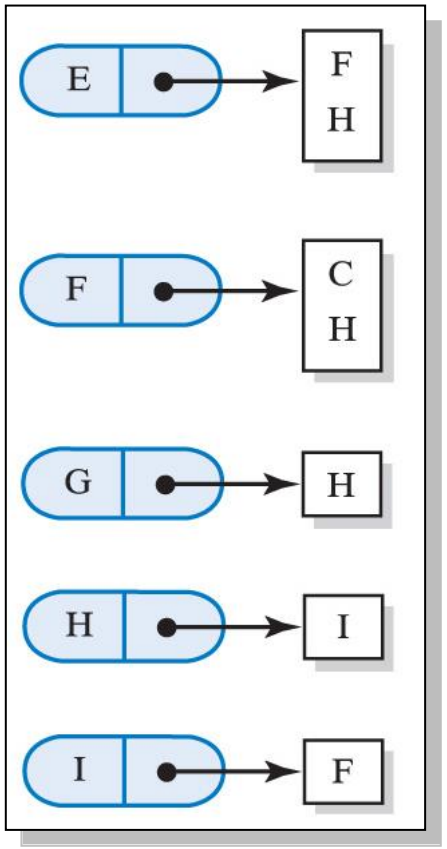
The Adjacency List

- It is a linked list that represents only edges that originate from the vertex (see the next slide)
- Space not wasted for edges that do not exist
- Uses less memory compared to adjacency matrix
- Preferred over adjacency matrix when working with sparse graph (graph with relatively few edges).
- For weighted graph, each list will provide the info on each of the neighbors along with the corresponding weight.

The Adjacency List for an unweighted and directed Graph

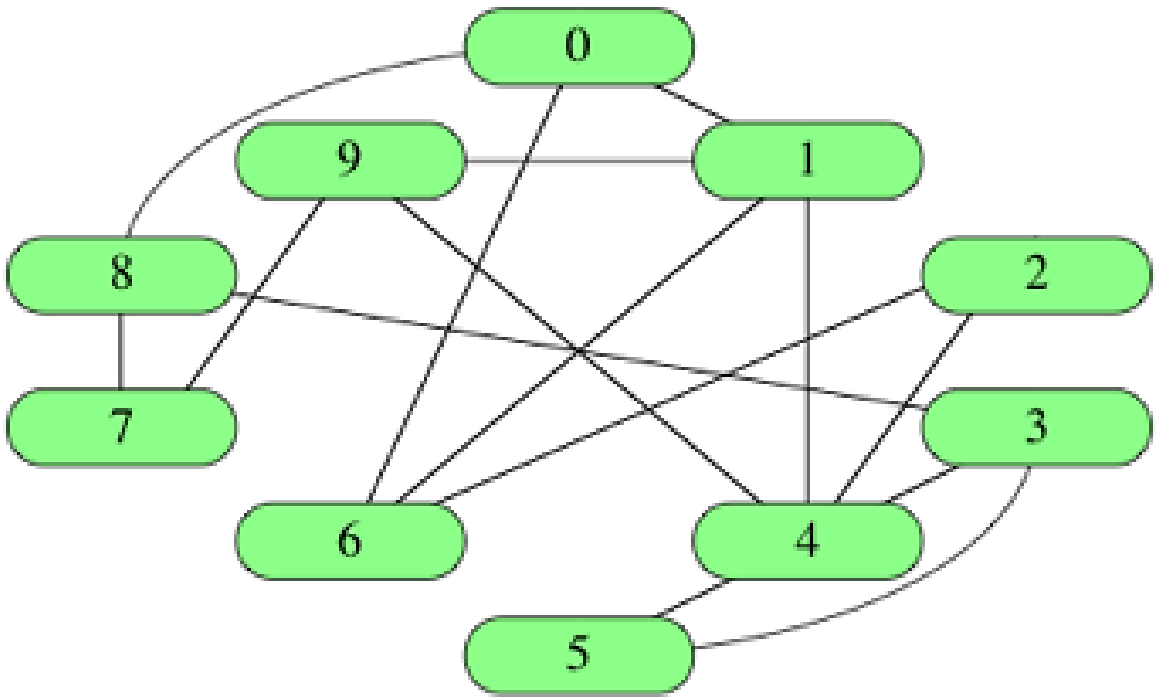


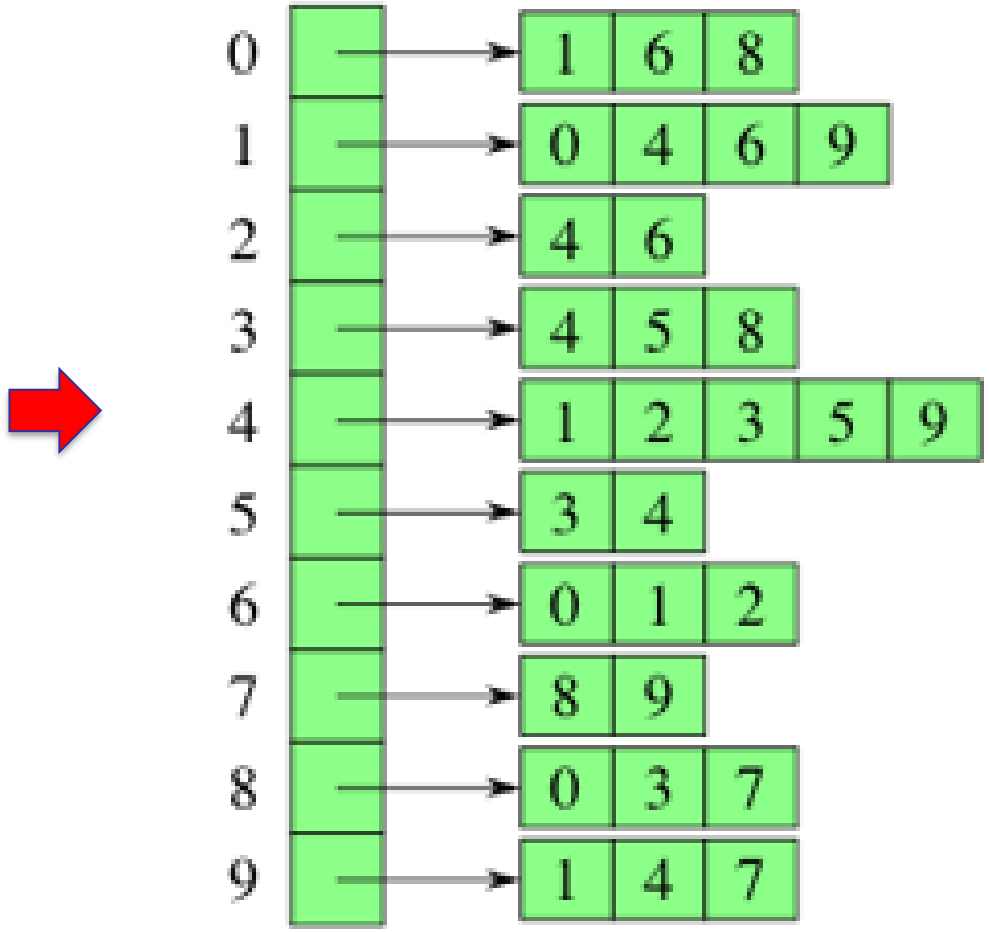




Adjacency lists for the directed un-weighted graph

The Adjacency List for an unweighted and undirected Graph

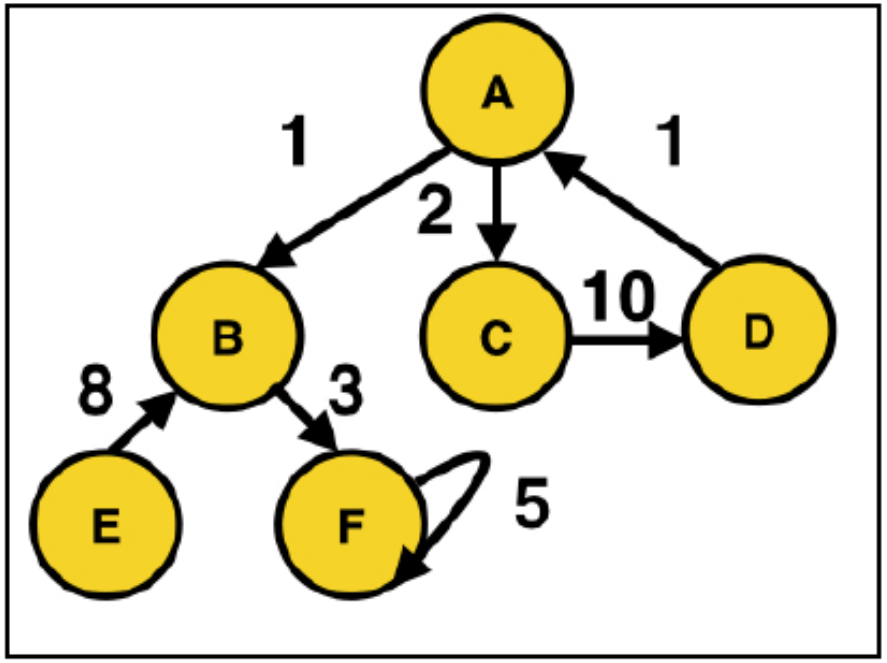




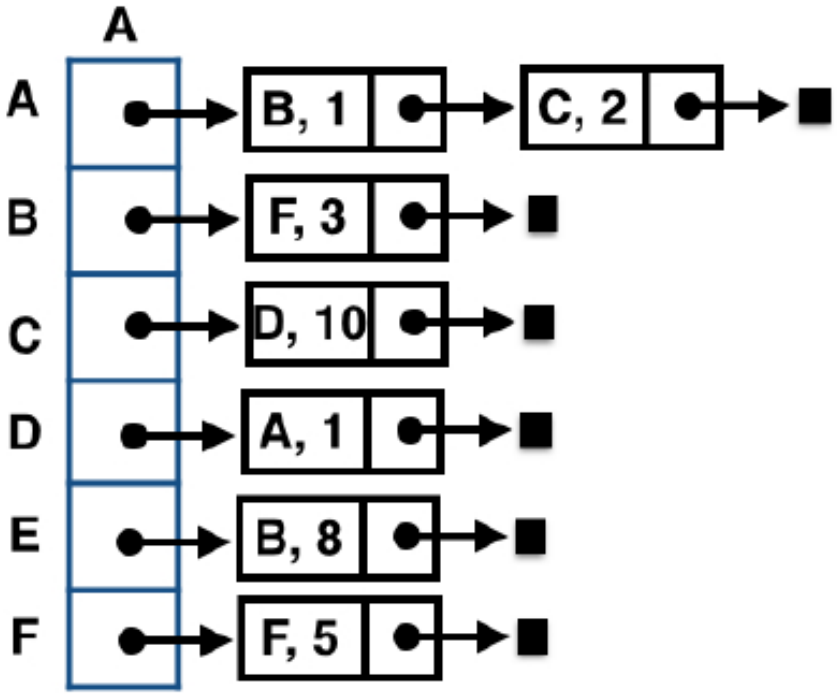
© khanacademy.org

Adjacency List for a weighted and directed Graph

Weighted Directed Graph *G*



Adjacency List for *G*

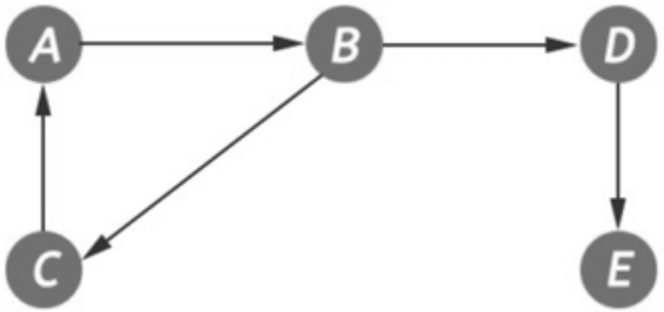


Worst-Case Time Complexity: Graph with Adjacency List

- Presence of an edge between two vertices: traverse a list for a particular vertex [O(n), in worst case]
- All neighbors of a vertex found by scanning entire list for that vertex [O(n) in the worst case; better than adjacency matrix on average]
- Adding a vertex to a graph: O(n) [since the vertices need to be stored in a Linked list. Note: If both head and tail nodes are implemented then it becomes O(1)]
- Adding/Removing an edge requires adding two vertices after retrieving them: O(n)
- Removing a vertex: O(n)

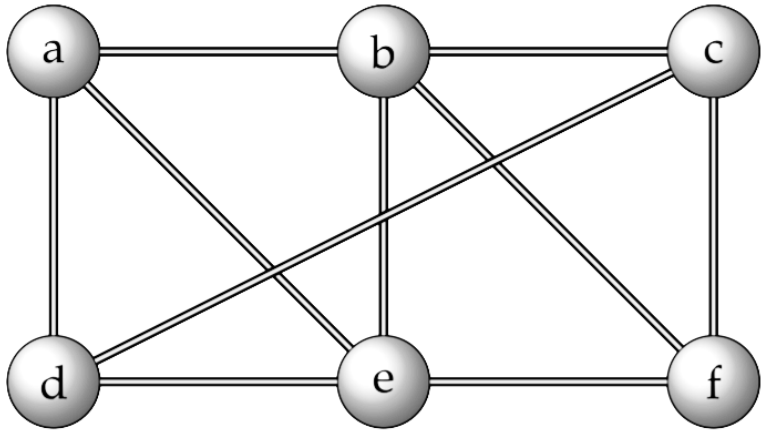
In-Class Discussion

Find the adjacency matrix and the adjacency list of the following graph.



In-Class Discussion

Find the adjacency matrix of the following graph and investigate if it is symmetric.



Note: $Matrix\ A\ is\ symmetric\ if\ a_{ij} = a_{ji}, for\ all\ i\ and\ j.$

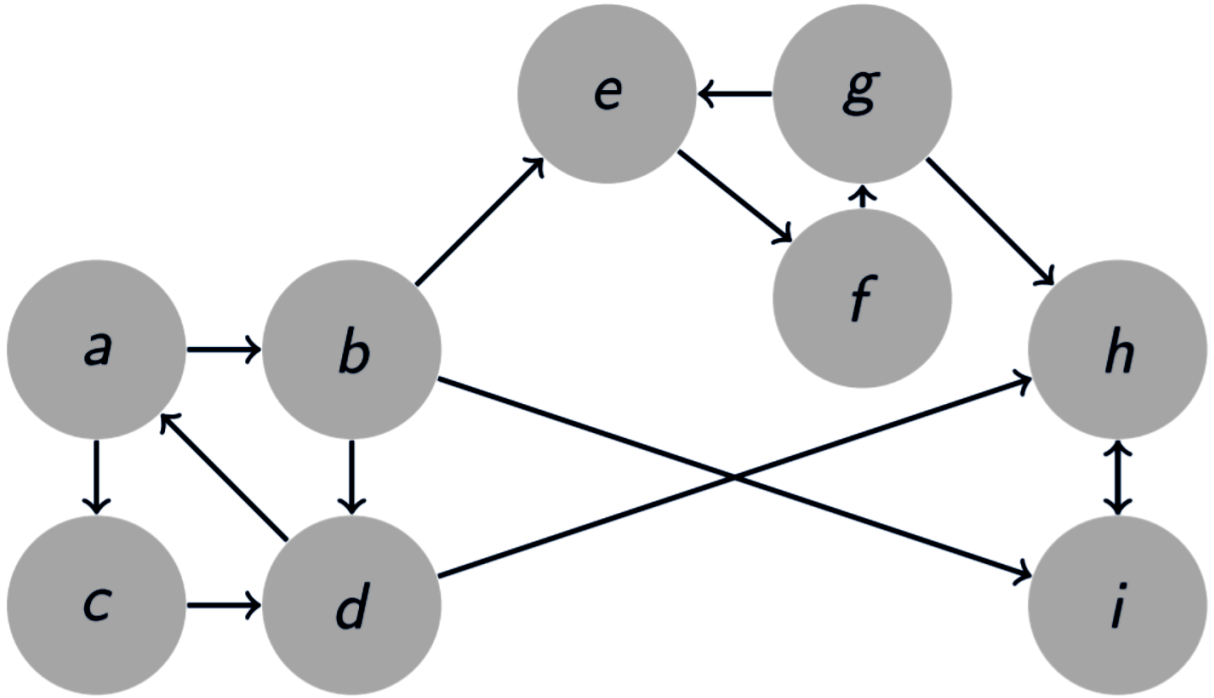
a_{ij} denotes the entry in the ith row and jth column of A.

Note on Matrix

- In an undirected graph A, if there is an edge between vi and vj, then both entries A[i][j] and A[j][i] are set to 1, otherwise they are set to 0. Therefore, A is a symmetric matrix.
- In a directed graph B, if there is a directed edge between vi and vj, then only B[i][j] is set to 1, otherwise it is set to 0. Therefore, B is not necessarily symmetric.
- If the edges have weights, then weights are listed explicitly in the corresponding entry instead of 1

In-Class Discussion

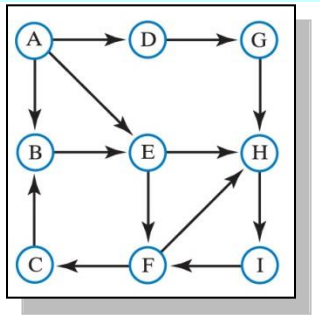
Is the following graph strongly connected?



Graph Traversal

- Graph traversal is the process of visiting each vertex (node) in a graph for either checking or updating data.
- Generally, there are two graph traversal Algorithms:
  - Breadth first Traversal (BFT); also known as Breadth first Search (BFS)
  - Depth first Traversal (DFT); also known as Depth first Search (DFS)
- Both traversals result in a spanning tree.

Breadth-First Traversal: High Level Description

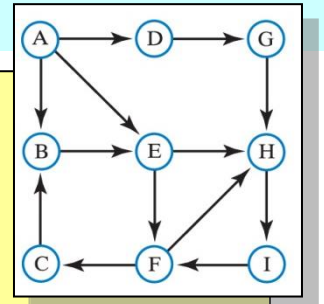


- A breadth-first traversal
  - Step 1: visit any vertex (v) of choice
  - Step 2: visit each of the v's neighbors
  - Step 3: then backtrack and start visiting the neighbors of the 'first neighbor of v' (if not visited). Continue this process till all the vertices are visited.

Note: The order in which the neighbors are visited is not specified and can depend on the graph's implementation.

- Examine the Pseudo Code (algorithm) for breadth-first traversal of a nonempty graph beginning at a given vertex

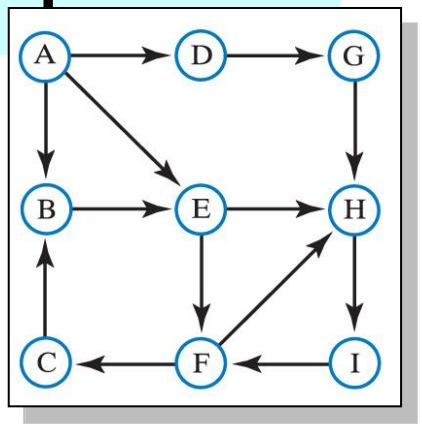
Breadth-First Traversal: The Pseudo-Code




```
Algorithm getBreadthFirstTraversal (originVertex)
    traversalOrder = a new queue for the resulting traversal order
    vertexQueue = a new queue to hold vertices as they are visited
    Mark originVertex as visited
    traversalOrder.enqueue (originVertex)
    vertexQueue.enqueue (originVertex)
    while (!vertexQueue.isEmpty ())
    {
        frontVertex = vertexQueue.dequeue ()
        while (frontVertex has a neighbor)
        {
            nextNeighbor = next neighbor of frontVertex
            if (nextNeighbor is not visited)
            {
                Mark nextNeighbor as visited
                traversalOrder.enqueue (nextNeighbor)
                vertexQueue.enqueue (nextNeighbor)
            }
        }
    }
    return traversalOrder
```

frontVertex	nextNeighbor	Visited vertex	vertexQueue (front to back)	traversalOrder (front to back)
		A	A	A
A			*empty*	
	B	B	B	AB
	D	D	BD	ABD
	E	E	BDE	ABDE
B			DE	
D			E	
	G	G	EG	ABDEG
E			G	
	F	F	GF	ABDEGF
	H	H	GFH	ABDEGFH
G			FH	
F			H	
	C	C	HC	ABDEGFHC
H			C	
	I	I	CI	ABDEGFHCI
C			I	
I			*empty*	

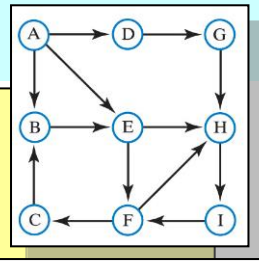
Depth-First Traversal : High Level Description



- Step 1: Visits any vertex (v), then
  - A neighbor (w) of v,
  - A neighbor (x) of w,
  - Following this approach, continue visiting the unvisited ones as far as possible from the original vertex
- Step 2: Then back up by one vertex
  - Consider the next neighbor of that vertex, if any, and then follow Step 1 for the unvisited ones.
- Step 3: Go back to Step 2 till all the vertices are visited.

  Note: The order in which the neighbors are visited is not specified and can depend on the graph's implementation.
- View algorithm (next slide) for depth-first traversal

Depth-First Traversal: The Pseudo-Code




```
Algorithm getDepthFirstTraversal (originVertex)
    traversalOrder = a new queue for the resulting traversal order
    vertexStack = a new stack to hold vertices as they are visited
    Mark originVertex as visited
    traversalOrder.enqueue (originVertex)
    vertexStack.push (originVertex)
     while (!vertexStack.isEmpty ())
    {
        topVertex = vertexStack.peek ()
        if (topVertex has an unvisited neighbor)
        {
           nextNeighbor = next unvisited neighbor of topVertex
           Mark nextNeighbor as visited
           traversalOrder.enqueue (nextNeighbor)
           vertexStack.push (nextNeighbor)
        }
        else // all neighbors are visited
            vertexStack.pop ()
    }
    return traversalOrder
```

topVertex	nextNeighbor	Visited vertex	vertexStack (top to bottom)	traversalOrder (front to back)
		A	A	A
A			A	
	B	B	BA	AB
B			BA	
	E	E	EBA	ABE
E			EBA	
	F	F	FEBA	ABEF
F			FEBA	
	C	C	CFEBA	ABEFC
C			FEBA	
F			FEBA	
	H	H	HFEBA	ABEFCH
H			HFEBA	
	I	I	IHFEBA	ABEFCHI
I			HFEBA	
H			FEBA	
F			EBA	
E			BA	
B			A	
A			A	
	D	D	DA	ABEFCHID
D			DA	
	G		GDA	ABEFCHIDG
G			DA	
D			A	
A			*empty*	ABEFCHIDG

In-Class Discussion

Which of the following statements is NOT correct?

1) Weighted graphs are always undirected.

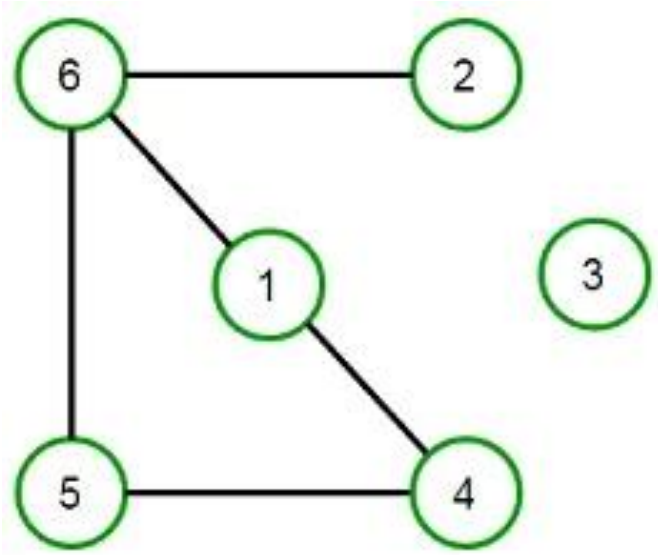
2) A tree is a type of graph.

3) Not all graphs are trees.

4) A tree is a connected graph without cycles, and in a tree, no two parent can have the same child.

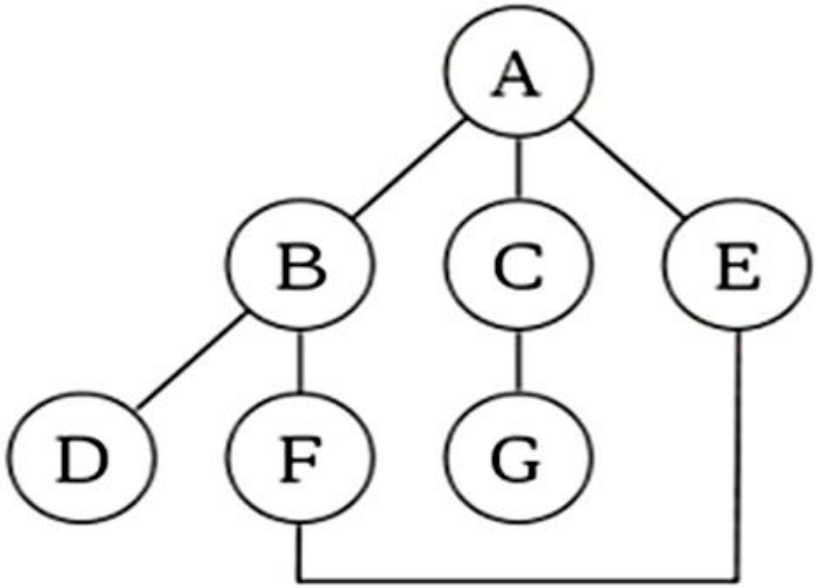
In-Class Discussion

Define the following graph as an ordered pair of Edges E with the aid of the vertices.



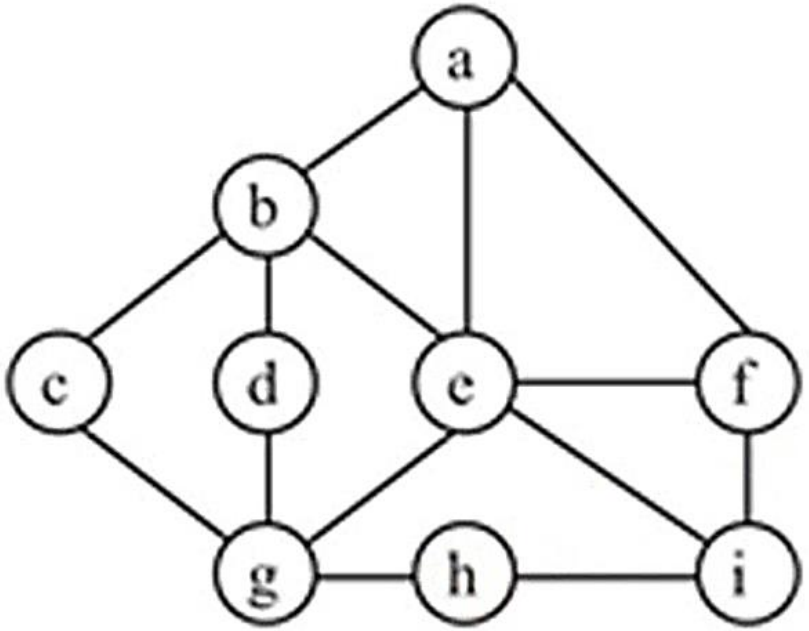
In-Class Discussion

Perform a breath first search on the graph below by considering that the vertex ‘A’ would be visited first, and the visitation will be performed in alphabetical order.



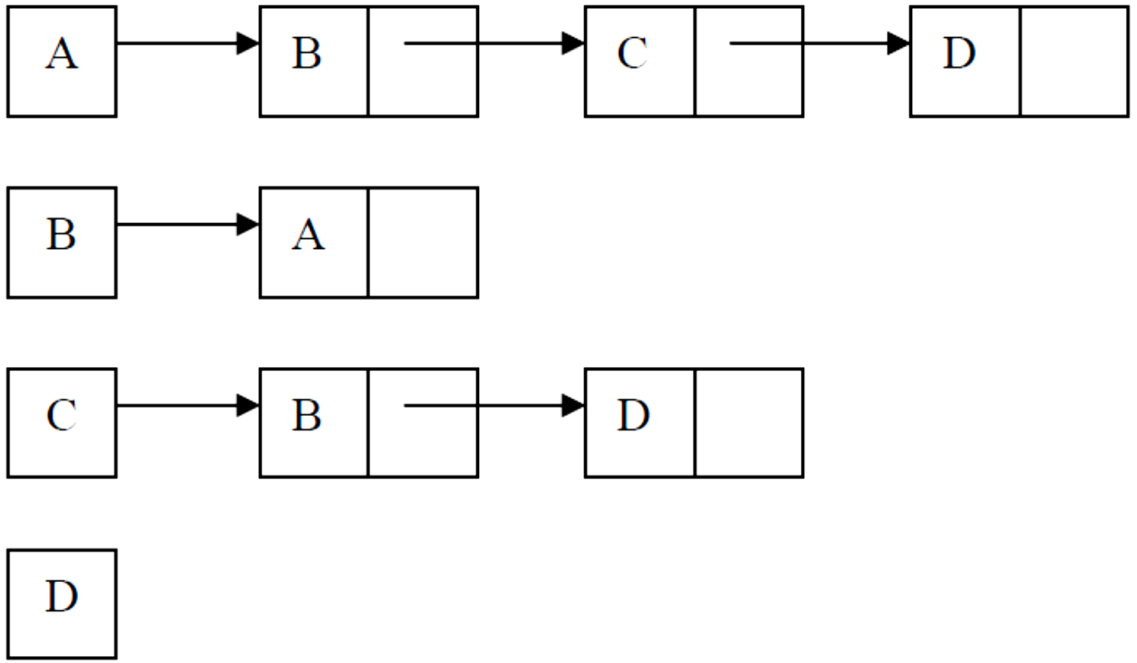
In-Class Discussion

Perform a depth first search on the graph below by considering that the vertex ‘a’ would be visited first, and the visitation will be performed in alphabetical order.



In-Class Discussion

Draw the picture of a directed graph that has the following adjacency list representation.



End of this Unit

Appendix: Implementing Graphs

- This section is not included in any assessment on this course.
- This section has been added so that you've a guideline to follow when you are implementing graph, which you will do down the road.

Representing Graphs in Programming

- Representing a graph in a program is to store its Edges and Vertices.
- The array and list data structures are used to represent a graph in a program.
- Representing Vertices:
  - Arrays
  - List
- Representing Edges
  - Edge Array
  - Edge Objects
  - Adjacency Matrices
  - Adjacency Lists

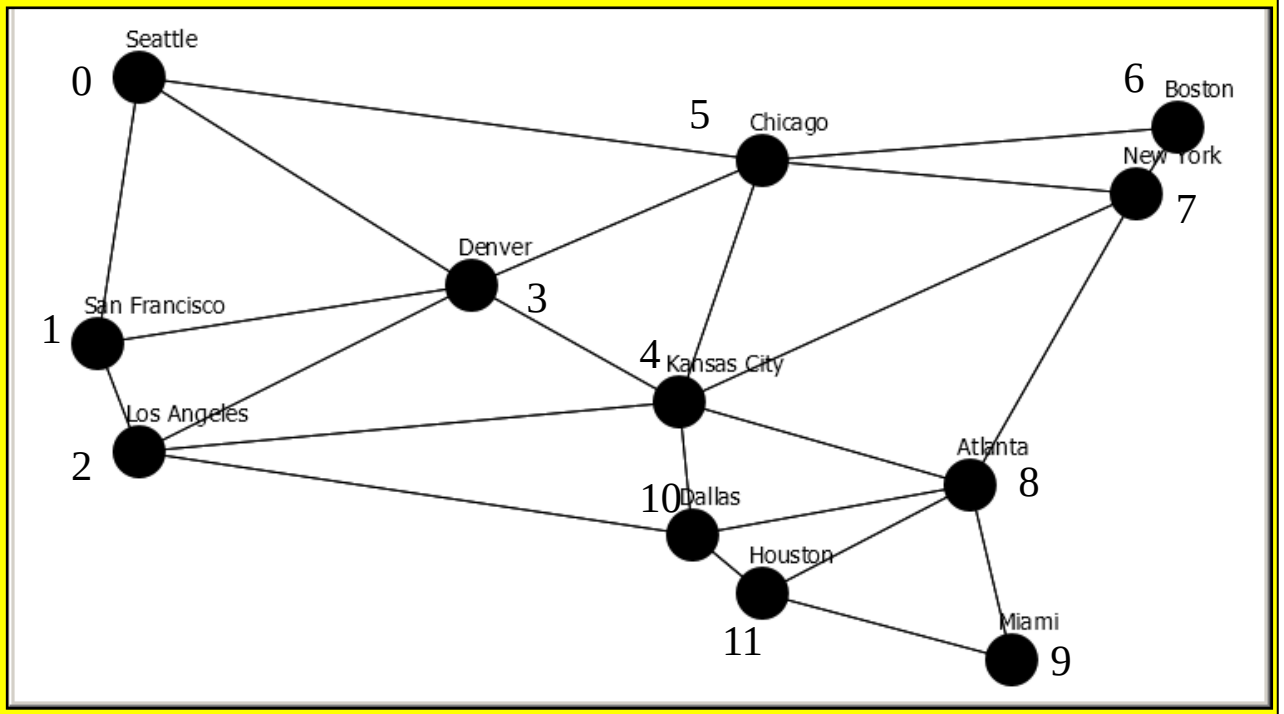
Representing Vertices in Java

- The vertices can be stored in an array or list. E.g.
  - String[] verticesInArray = {“Seattle“, “San Francisco“, … };
  - List<String> verexList;
- The vertices can be objects of any type. E.g.,
  - Student[] studentVertices = {student0, student1, … };

Representing Edges

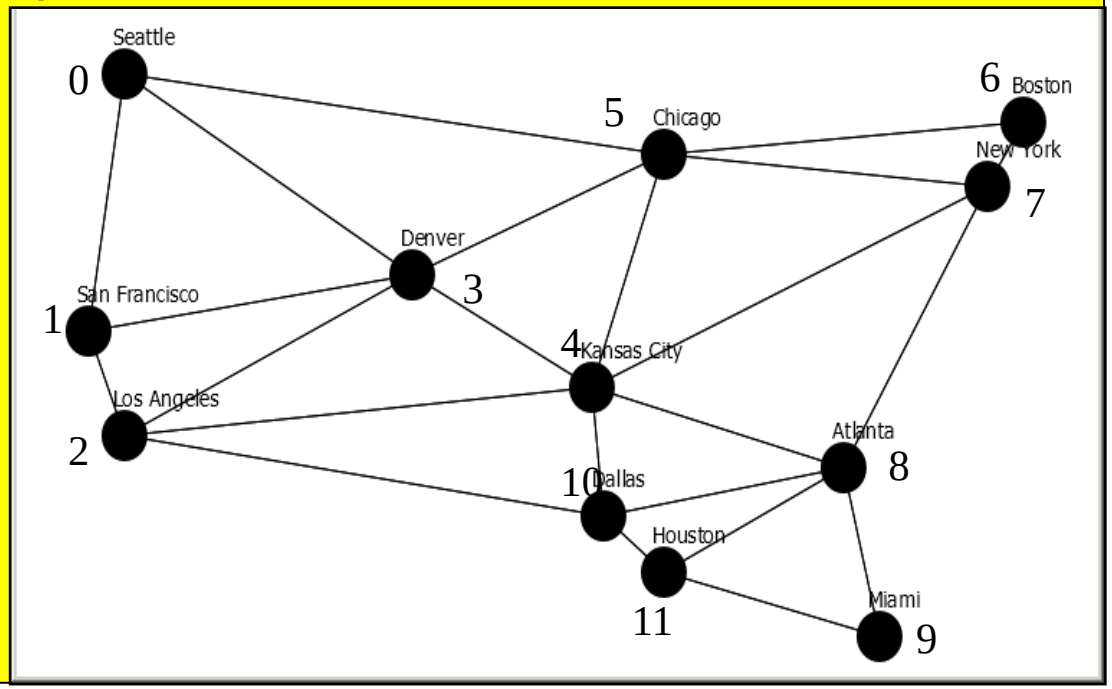
- Edge Array
  - int[][] edges = {{0, 1}, {0, 3} {0, 5}, {1, 0}, {1, 2}, … };
- Edge Object

```
public class Edge {
   int u, v;
   public Edge(int u, int v) {
   this.u = u;
   this.v = v;
   }
}
//in driver..
List<Edge> list = new ArrayList<>();
list.add(new Edge(0, 1)); list.add(new Edge(0, 3)); …
```



Representing Edges: Adjacency Matrix

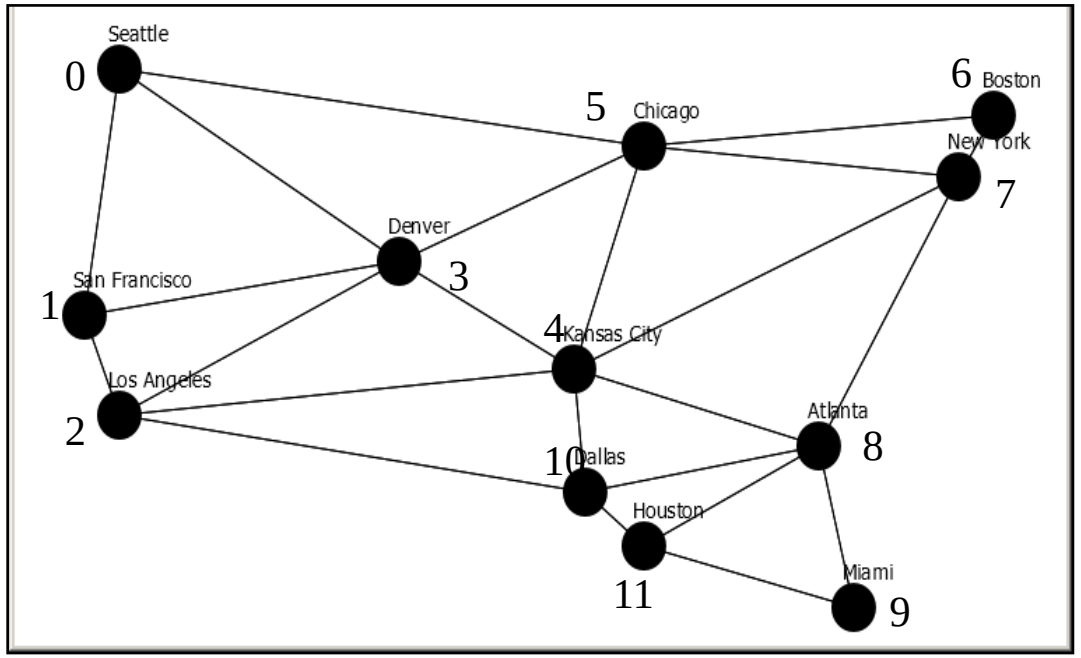
```
int[][] adjacencyMatrix = {
  {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
  {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
  {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
  {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
  {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
  {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
  {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
  {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
  {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
  {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```



Representing Edges: Adjacency Vertex List

List<Integer>[] neighbors = new List[12];

Seattle	neighbors[0]	1	3	5			
San Francisco	neighbors[1]	0	2	3			
Los Angeles	neighbors[2]	1	3	4	10		
Denver	neighbors[3]	0	1	2	4	5	
Kansas City	neighbors[4]	2	3	5	7	8	10
Chicago	neighbors[5]	0	3	4	6	7	
Boston	neighbors[6]	5	7				
New York	neighbors[7]	4	5	6	8		
Atlanta	neighbors[8]	4	7	9	10	11	
Miami	neighbors[9]	8	11				
Dallas	neighbors[10]	2	4	8	11		
Houston	neighbors[11]	8	9	10			

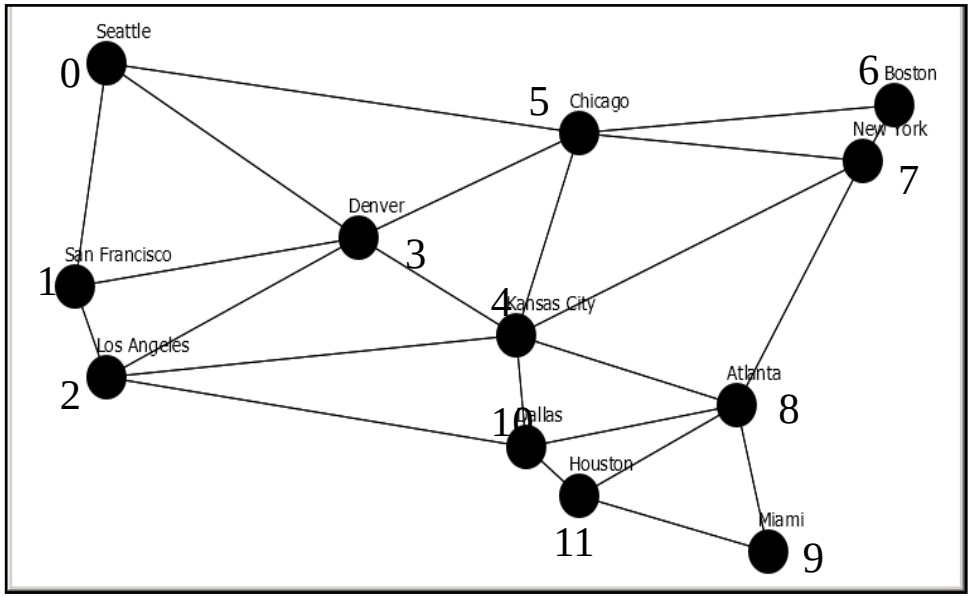


List<List<Integer>> neighbors = new ArrayList<>();

Representing Edges: Adjacency Edge List

List<Edge>[] neighbors = new List[12];

Seattle	neighbors[0]	Edge(0, 1)	Edge(0, 3)	Edge(0, 5)			
San Francisco	neighbors[1]	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
Los Angeles	neighbors[2]	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
Denver	neighbors[3]	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
Kansas City	neighbors[4]	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
Chicago	neighbors[5]	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
Boston	neighbors[6]	Edge(6, 5)	Edge(6, 7)				
New York	neighbors[7]	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
Atlanta	neighbors[8]	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
Miami	neighbors[9]	Edge(9, 8)	Edge(9, 11)				
Dallas	neighbors[10]	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
Houston	neighbors[11]	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			



Representing Adjacency Edge List Using ArrayList

```
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
...
...
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```

Modeling Graphs

«interface» *Graph<V>*	
+getSize(): int	Returns the number of vertices in the graph.
+getVertices(): List<V>	Returns the vertices in the graph.
+getVertex(index: int): V	Returns the vertex object for the specified vertex index.
+getIndex(v: V): int	Returns the index for the specified vertex.
+getNeighbors(index: int): List<Integer>	Returns the neighbors of vertex with the specified index.
+getDegree(index: int): int	Returns the degree for a specified vertex index.
+printEdges(): void	Prints the edges.
+clear(): void	Clears the graph.
+addVertex(v: V): boolean	Returns true if v is added to the graph. Returns false if v is already in the graph.
+addEdge(u: int, v: int): boolean	Adds an edge from u to v to the graph throws IllegalArgumentException if u or v is invalid. Returns true if the edge is added and false if (u, v) is already in the graph.
+addEdge(e: Edge): boolean	Adds an edge into the adjacency edge list.
+remove(v: V): boolean	Removes a vertex from the graph.
+remove(u: int, v: int): boolean	Removes an edge from the graph.
+dfs(v: int): UnWeightedGraph<V>.SearchTree	Obtains a depth-first search tree starting from v.
+bfs(v: int): UnWeightedGraph<V>.SearchTree	Obtains a breadth-first search tree starting from v.

The generic type V is the type for vertices.

UnweightedGraph<V>	
#vertices: List<V>	Vertices in the graph.
#neighbors: List<List<Edge>>	Neighbors for each vertex in the graph.
+UnweightedGraph()	Constructs an empty graph.
+UnweightedGraph(vertices: V[], edges: int[][])	Constructs a graph with the specified edges and vertices stored in arrays.
+UnweightedGraph(vertices: List<V>, edges: List<Edge>)	Constructs a graph with the specified edges and vertices stored in lists.
+UnweightedGraph(edges: int[][], numberOfVertices: int)	Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)	Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….

Graph {Check the code in the Code Folder V2}

UnweightedGraph {Check the code in the Code Folder V2}

TestGraph {Check the code in the Code Folder V2}