Class 12 – Graphs

CSIS 3475 Data Structures and Algorithms

©Michael Hrybyk and others
NOT TO BE REDISTRIBUTED

Graphs

- A road map is a graph
- Definition of a graph:
  - A collection of distinct vertices and distinct edges
- In the map of Figure 29-1
  - The circles are vertices or nodes
  - The lines are called edges
- A subgraph is a portion of a graph that is itself a graph

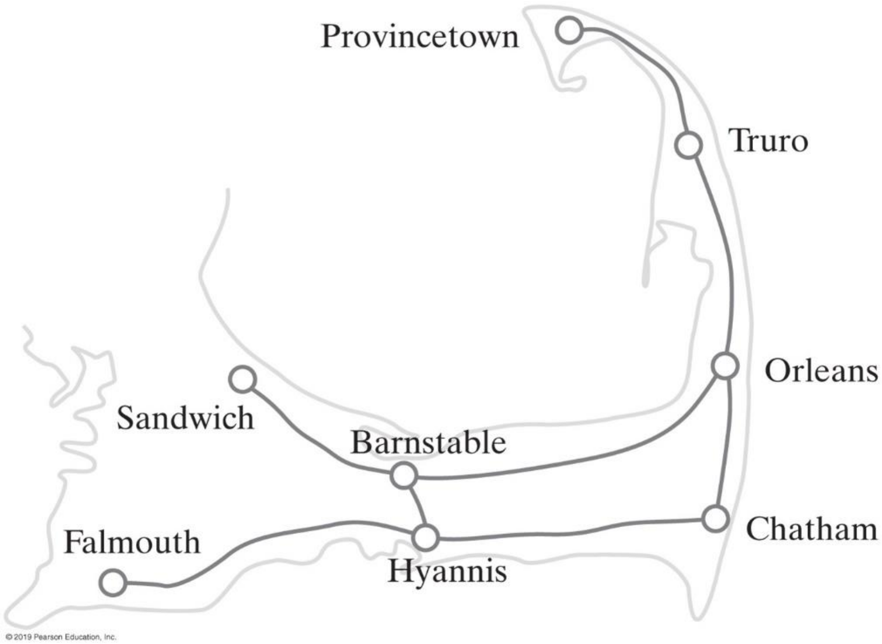


FIGURE 29-1 A portion of a road map

Connected Graphs

- A graph that has a path between every pair of distinct vertices is connected
- A complete graph has an edge between every pair of distinct vertices.
- Undirected graphs can be
  - Connected
  - Complete
  - Disconnected

Graphs

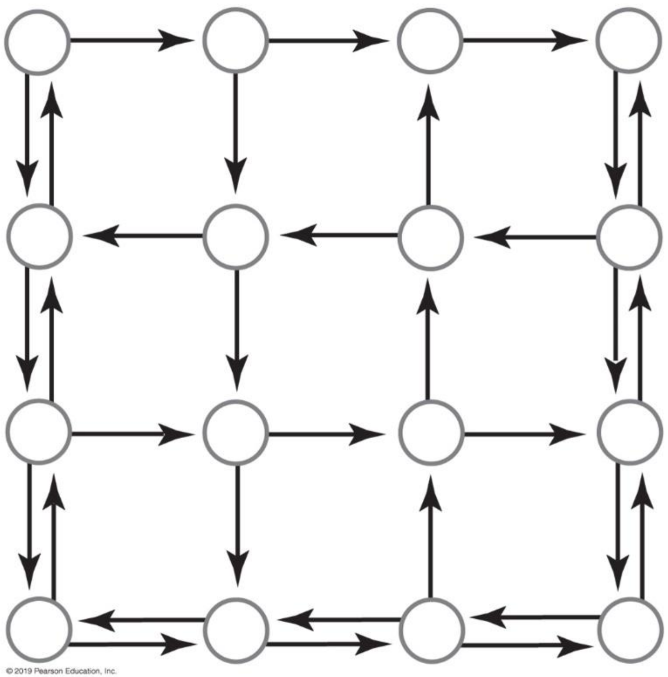
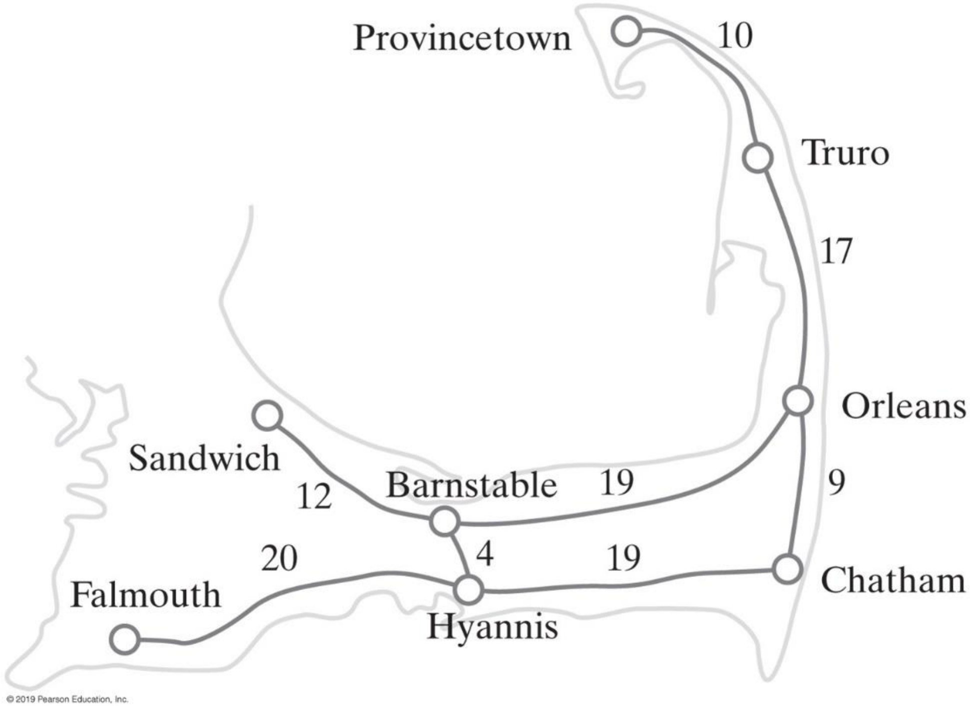


FIGURE 29-2 A directed graph representing a portion of a city's street map



- FIGURE 29-3 A weighted graph

Paths

- A path between two vertices in a graph is a sequence of edges
- A path in a directed graph must consider the direction of the edges
  - Called a directed path
- The length of a path is the number of edges that it comprises.
- A cycle is a path that begins and ends at the same vertex

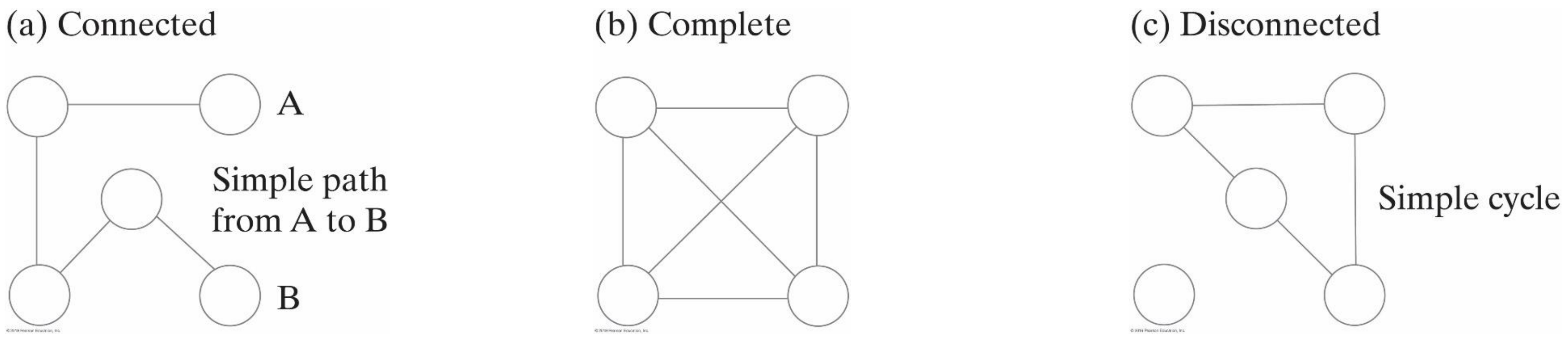
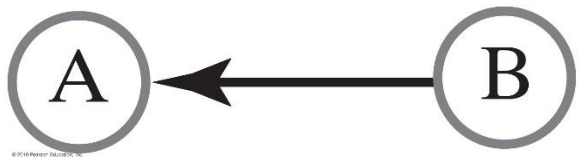


FIGURE 29-4 Undirected graphs

Directed Graphs

- In a directed graph vertex B is adjacent to A, if there is an edge leaving B and coming to A



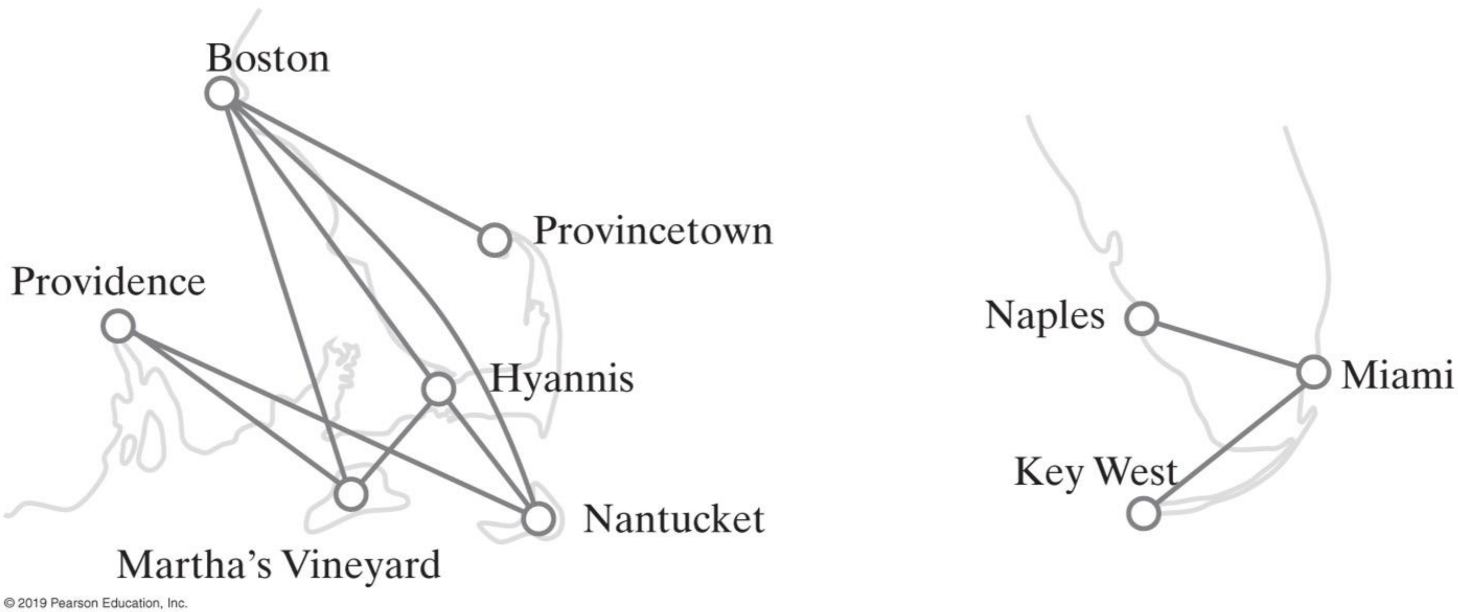
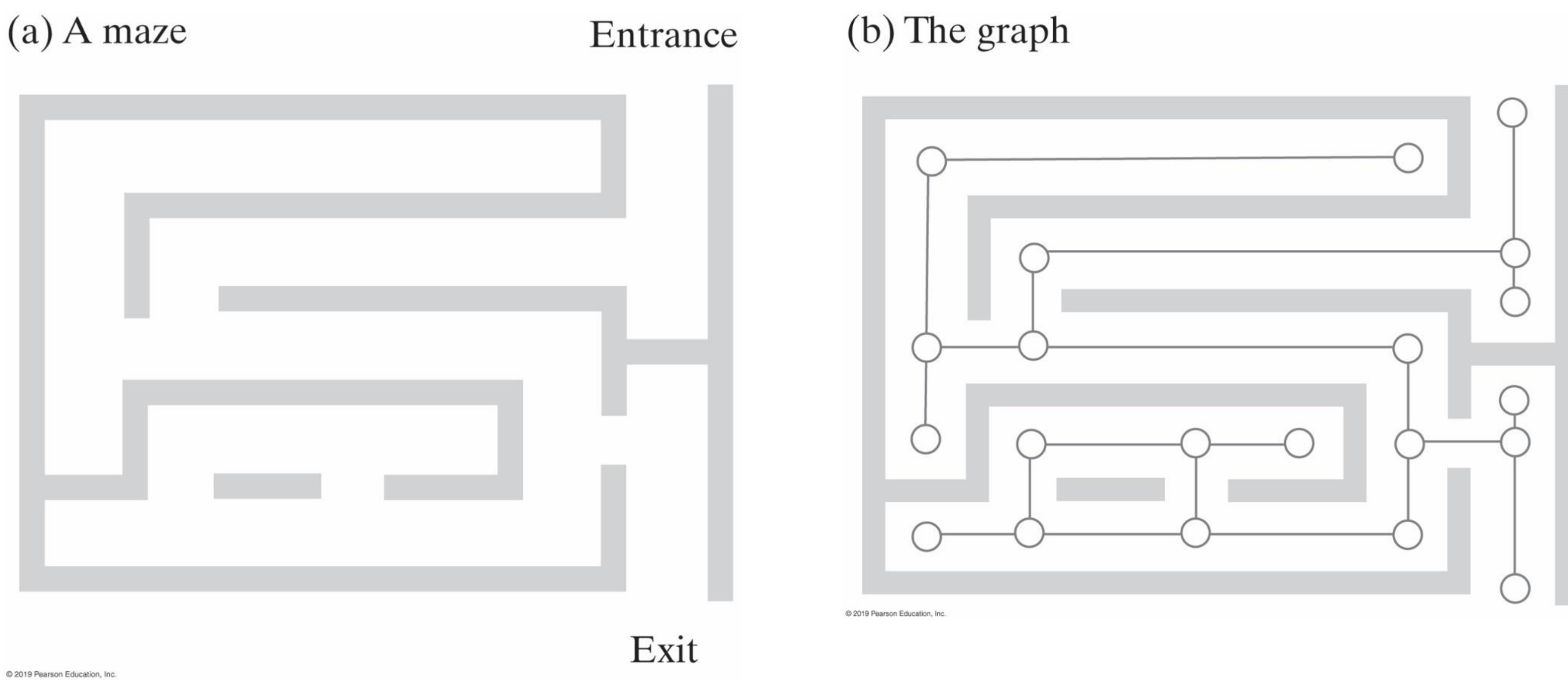


FIGURE 29-6 Airline routes

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

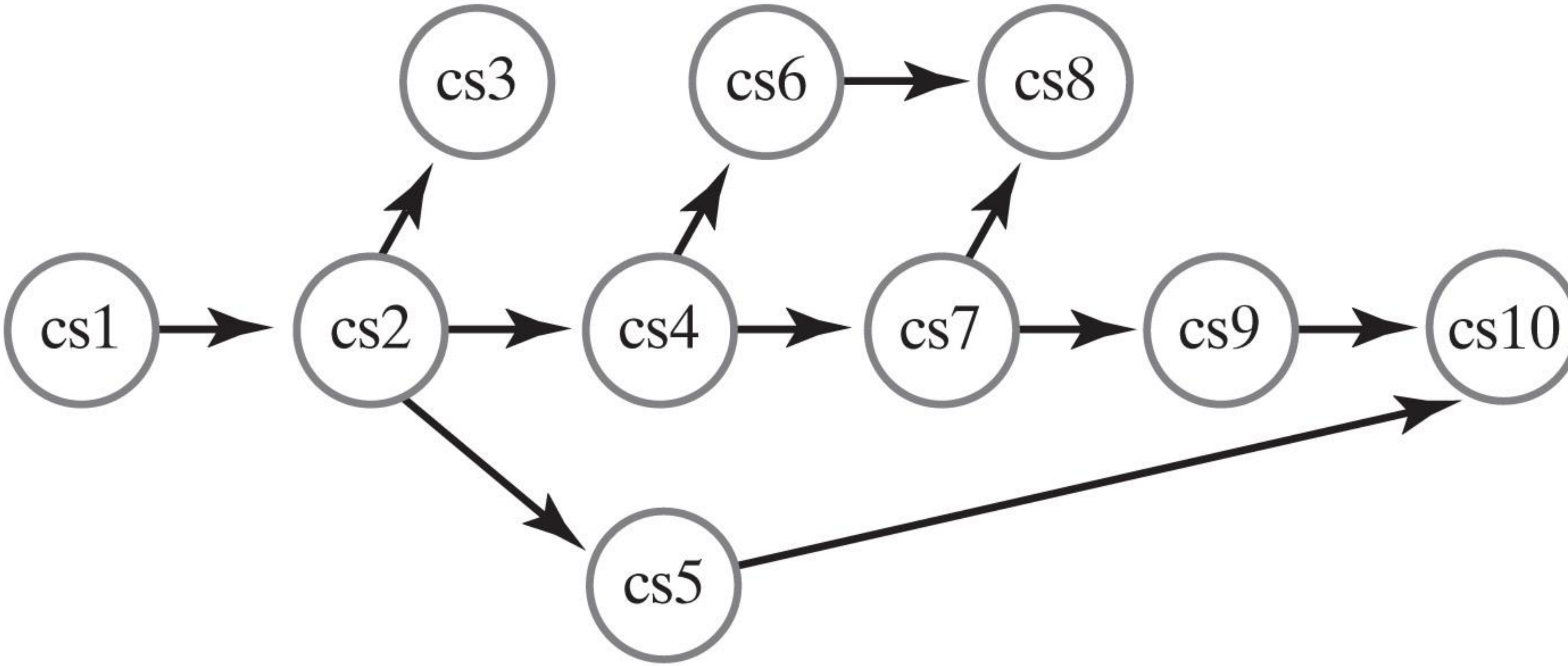
Graphs

- FIGURE 29-7 A maze and its representation as a graph



Directed Graphs

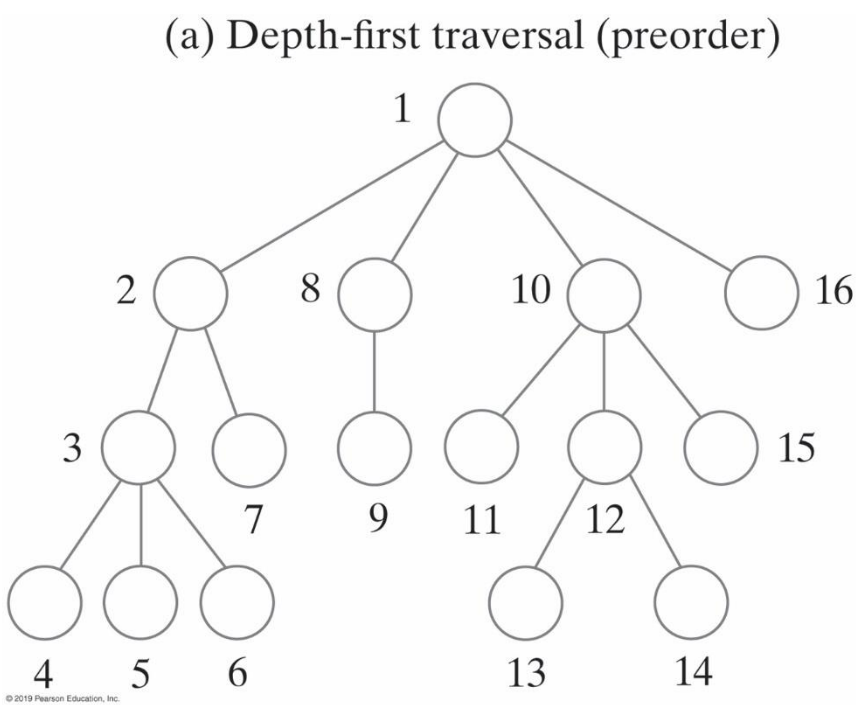
- FIGURE 29-8 The prerequisite structure for a selection of courses as a directed graph without cycles

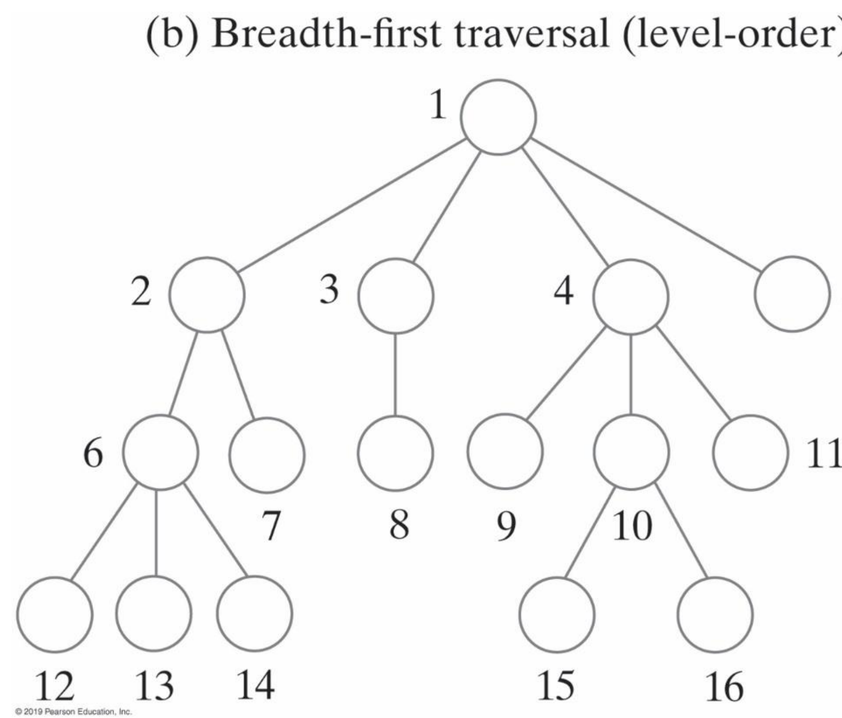


© 2019 Pearson Education, Inc.

Graph Traversal

- FIGURE 29-9 The visitation order of two traversals



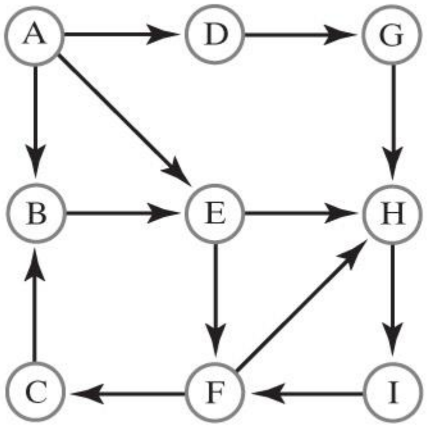


Breadth-First Traversal

- Algorithm that performs a breadth-first traversal of a nonempty graph beginning at a given vertex EXPLAIN THIS

```
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
```

Breadth-First Traversal



frontVertex	nextNeighbor	Visited vertex	vertexQueue (front to back)	traversalOrder
		A	A	A
A			*empty*	
	B	B	B	A B
	D	D	B D	A B D
	E	E	B D E	A B D E
B			D E	
D			E	
	G	G	E G	A B D E G
E			G	
	F	F	G F	A B D E G F
	H	H	G F H	A B D E G F H
G			F H	
F			H	
	C	C	H C	A B D E G F H C
H			C	
	I	I	C I	A B D E G F H C I
C			I	
I			*empty*	

© 2019 Pearson Education, Inc.

Depth-First Traversal

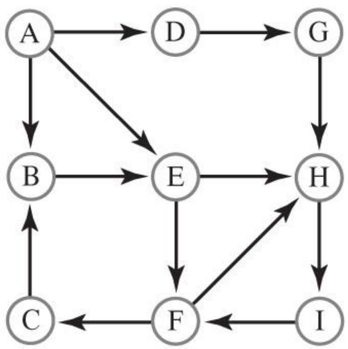
- Algorithm that performs a depth-first traversal of a nonempty graph, beginning at a given vertex

```
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
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

Depth-First Traversal

- FIGURE 29-11 A trace of a depth-first traversal beginning at vertex A of a directed graph



topVertex	nextNeighbor	Visited vertex	vertexStack (top to bottom)	traversalOrder (front to back)
		A	A	A
A			A	
	B	B	B A	A B
B			B A	
	E	E	E B A	A B E
E			E B A	
	F	F	F E B A	A B E F
F			F E B A	
	C	C	C F E B A	A B E F C
C			F E B A	
F			F E B A	
	H	H	H F E B A	A B E F C H
H			H F E B A	
	I	I	I H F E B A	A B E F C H I
I			H F E B A	
H			F E B A	
F			E B A	
E			B A	
B			A	
A			A	
	D	D	D A	A B E F C H I D
D			D A	
	G		G D A	A B E F C H I D G
G			D A	
D			A	
A			*empty*	

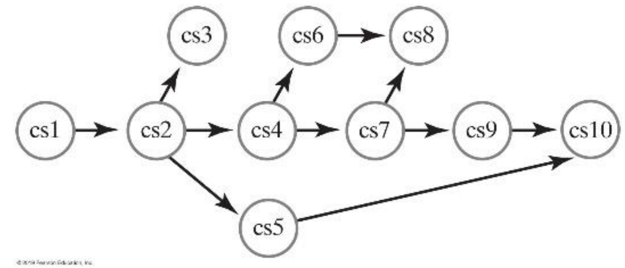
© 2019 Pearson Education, Inc.

Topological Ordering

- FIGURE 29-13 An impossible prerequisite structure for three courses, as a directed graph with a cycle

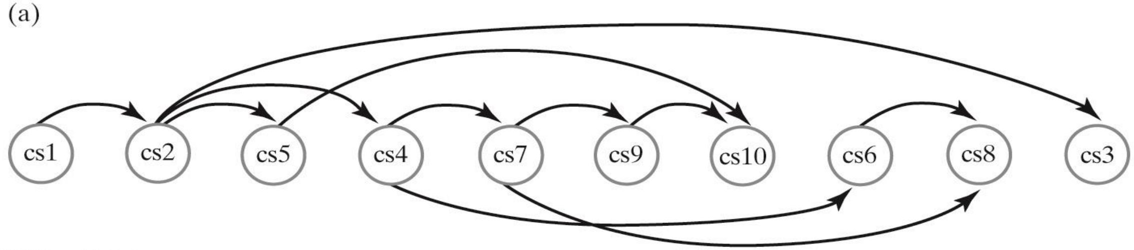
In a topological order of the vertices in a directed graph without cycles, vertex a precedes vertex b whenever a directed edge exists from a to b.

Topological Ordering



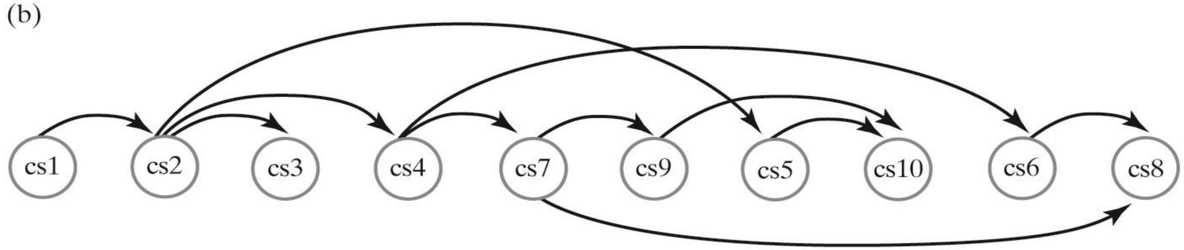
- FIGURE 29-12 Three topological orders for the graph in Figure 29-8

(a)



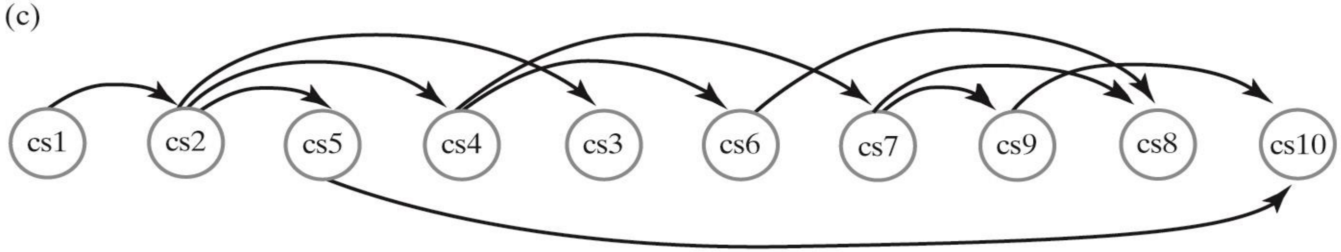
© 2019 Pearson Education, Inc.

(b)



© 2019 Pearson Education, Inc.

(c)



© 2019 Pearson Education, Inc.

Topological Order

- An algorithm that describes a topological sort

Algorithm **getTopologicalOrder()**

vertexStack = *a new stack to hold vertices as they are visited*

numberOfVertices = *number of vertices in the graph*

for (counter = 1 *to* numberOfVertices)

{

nextVertex = *an unvisited vertex whose neighbors, if any, are all visited*

Mark nextVertex *as visited*

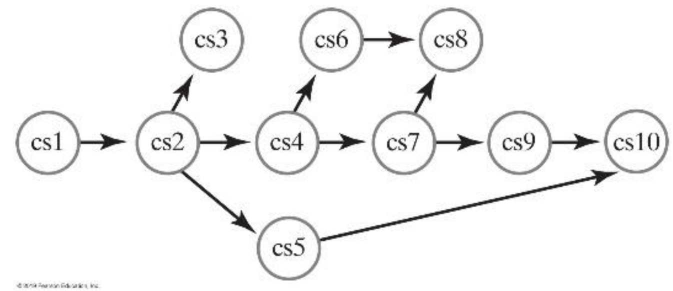
vertexStack.push(nextVertex)

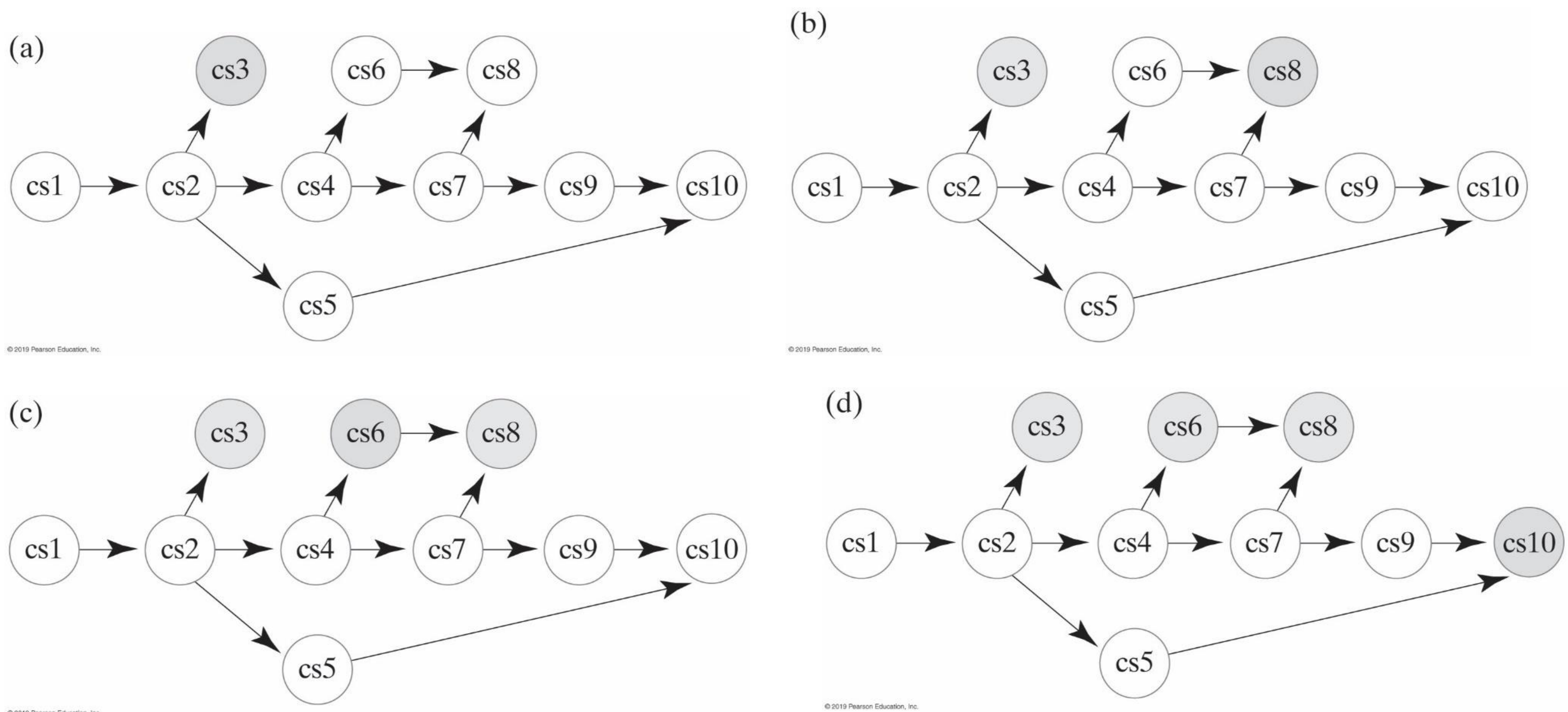
}

return vertexStack

Topological Ordering (Part 1)

- FIGURE 29-14 Finding a topological order for the graph in Figure 29-8

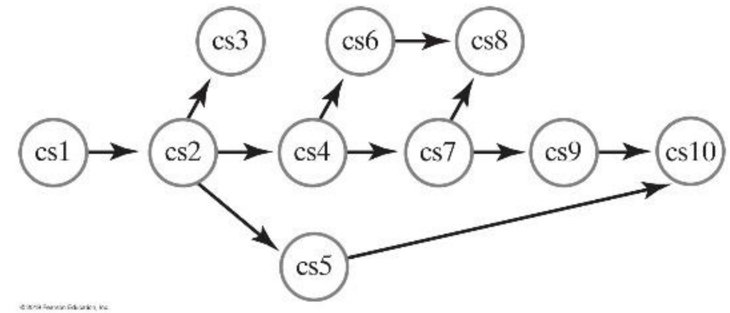


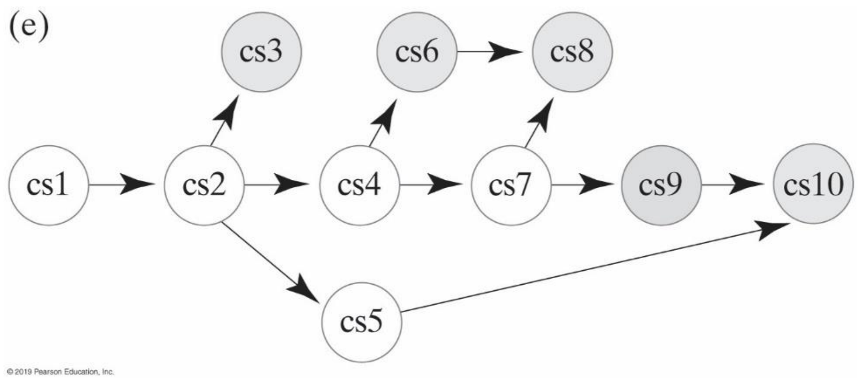


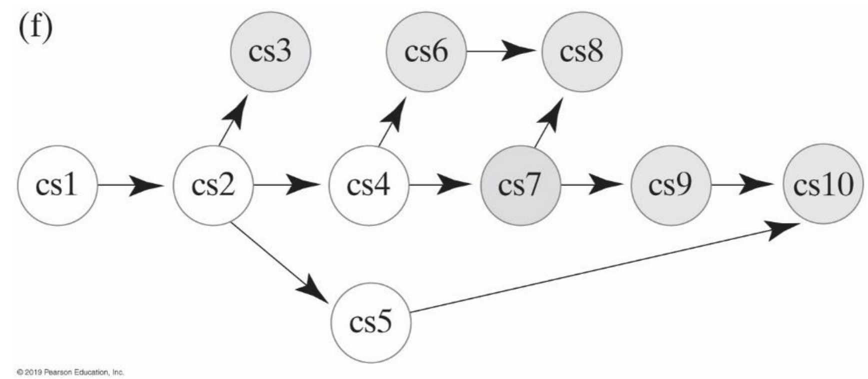
©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

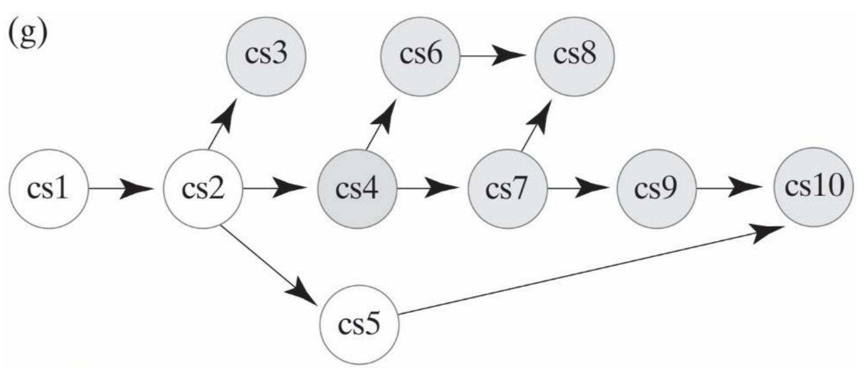
Topological Ordering (Part 2)

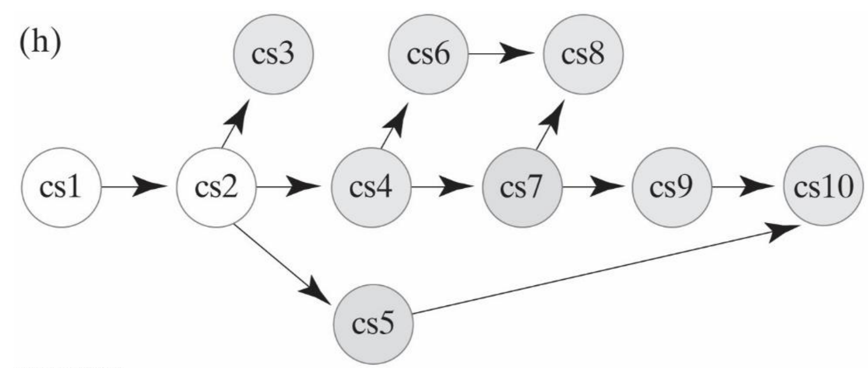
- FIGURE 29-14 Finding a topological order for the graph in Figure 29-8





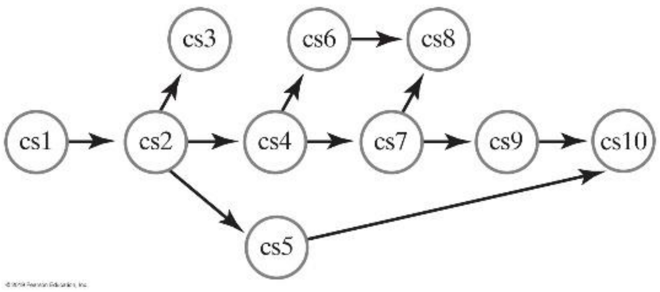


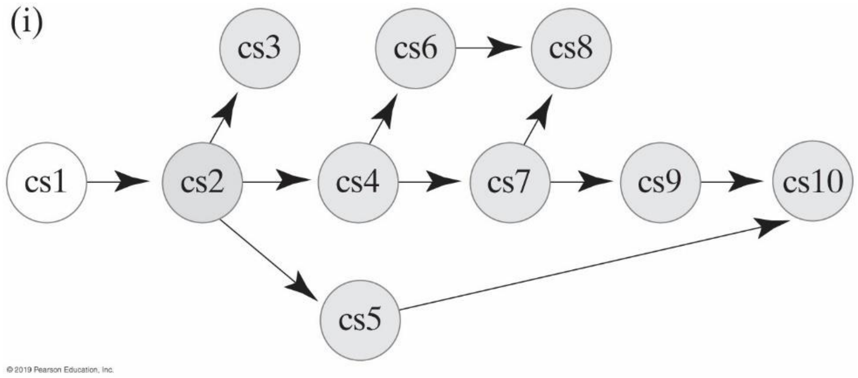


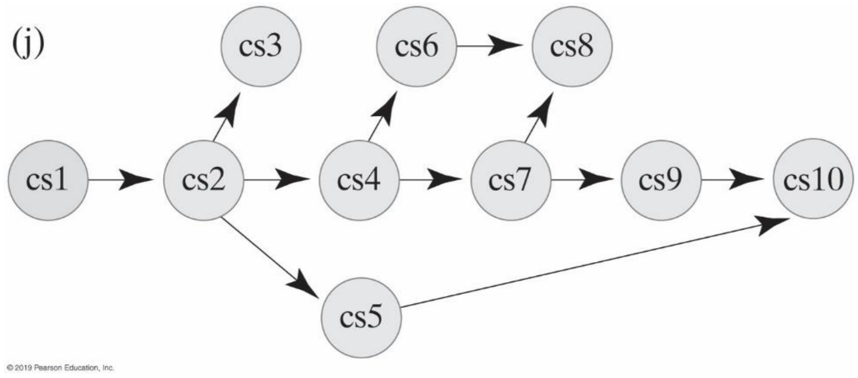


Topological Ordering (Part 3)

- FIGURE 29-14 Finding a topological order for the graph in Figure 29-8



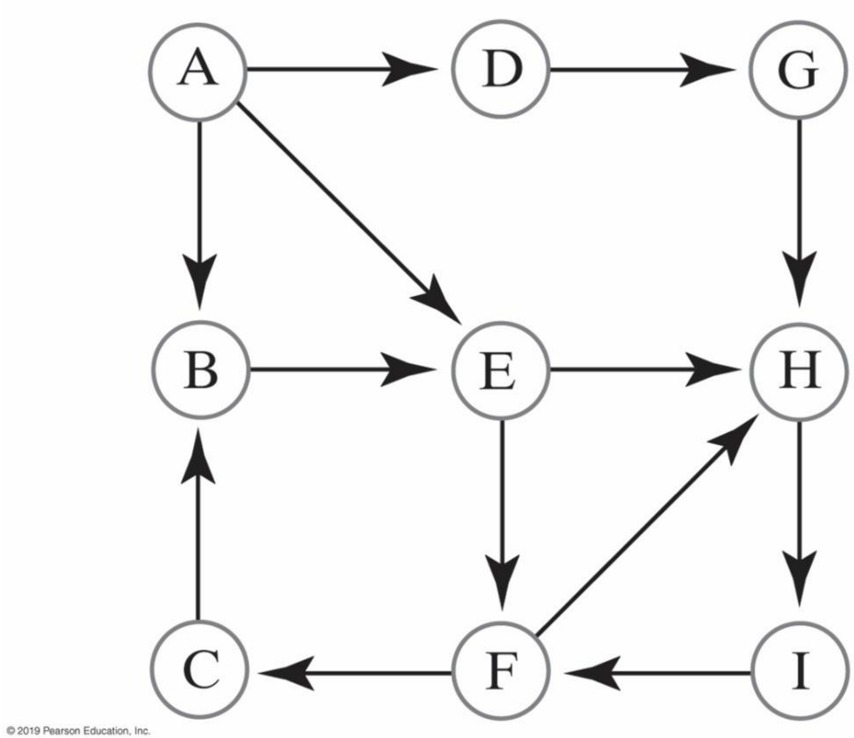




Shortest Path in Unweighted Graph

- FIGURE 29-15 An unweighted graph and the possible paths from vertex A to vertex H

(a) An unweighted graph



(b) Possible paths through the graph

A → B → E → F → H
A → B → E → H
A → D → G → H
A → E → F → H
A → E → H

© 2019 Pearson Education, Inc.

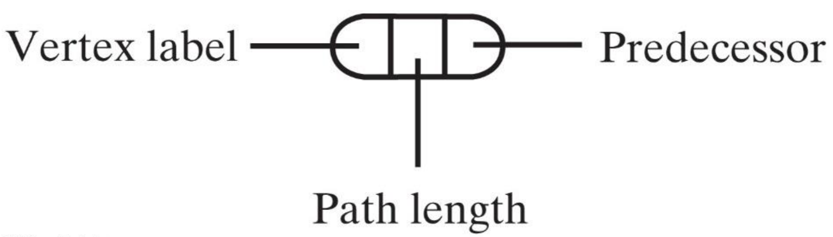
Shortest Path in an Unweighted Graph (Part 1)

```
Algorithm getShortestPath(originVertex, endVertex, path)
done = false
vertexQueue = a new queue to hold vertices as they are visited
Mark originVertex as visited
vertexQueue.enqueue(originVertex)
while (!done && !vertexQueue.isEmpty())
{
    frontVertex = vertexQueue.dequeue()
    while (!done && frontVertex has a neighbor)
    {
        nextNeighbor = next neighbor of frontVertex
        if (nextNeighbor is not visited)
        {
            Mark nextNeighbor as visited
            Set the length of the path to nextNeighbor to 1 + length of path to frontVertex
            Set the predecessor of nextNeighbor to frontVertex
            vertexQueue.enqueue(nextNeighbor)
        }
        if (nextNeighbor equals endVertex)
            done = true
    }
}
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

Shortest Path in an Unweighted Graph (Part 2)

```
// Traversal ends; construct shortest path
pathLength = length of path to endVertex
path.push(endVertex)
vertex = endVertex
while (vertex has a predecessor)
{
	vertex = predecessor of vertex
	path.push(vertex)
}
return pathLength
```

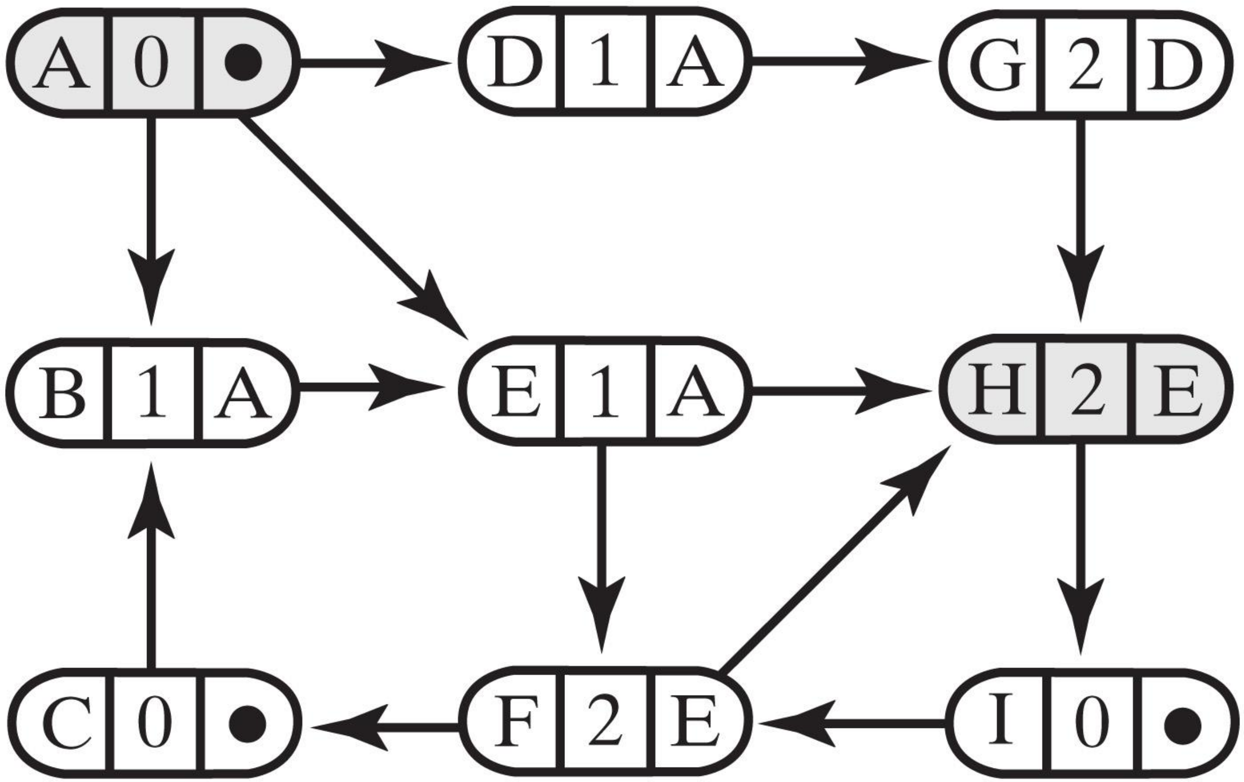


© 2019 Pearson Education, Inc.

FIGURE 29-16 The data in a vertex

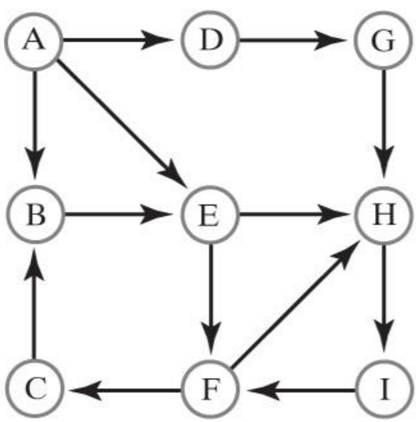
Shortest Path in a Weighted Graph

- FIGURE 29-17 The graph in Figure 29-15a after the shortest-path algorithm has traversed from vertex A to vertex H



Shortest Path in a Weighted Graph

- FIGURE 29-18 A trace of the traversal in the algorithm to find the shortest path from vertex A to vertex H in an unweighted graph



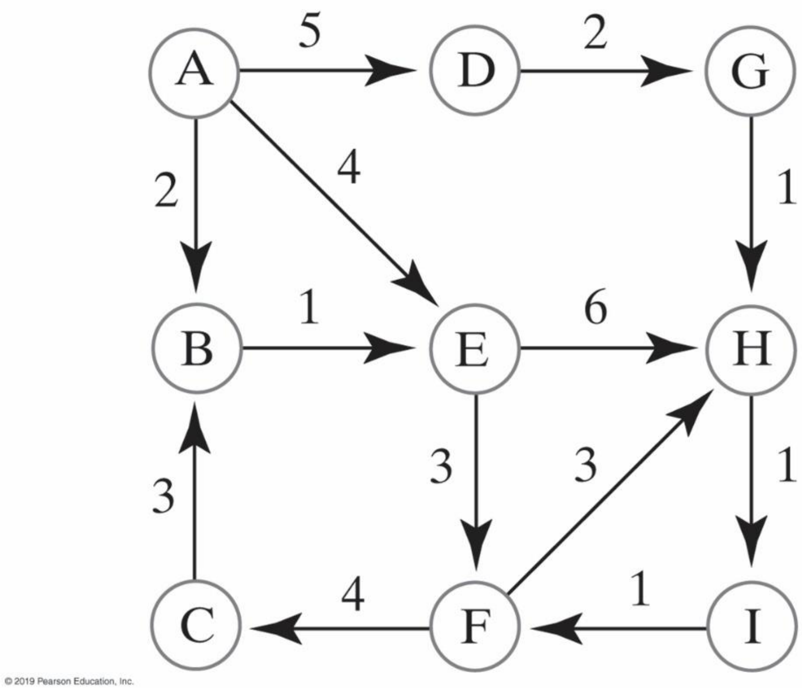
frontVertex	nextNeighbor	Visited vertex	vertexQueue (front to back)
		A	(A 0 •)
(A 0 •)			*empty*
	B	B	(B 1 A)
	D	D	(B 1 A) (D 1 A)
	E	E	(B 1 A) (D 1 A) (E 1 A)
(B 1 A)			(D 1 A) (E 1 A)
(D 1 A)			(E 1 A)
	G	G	(E 1 A) (G 2 D)
(E 1 A)			(G 2 D)
	F	F	(G 2 D) (F 2 E)
	H	H	(G 2 D) (F 2 E) (H 2 E)

© 2019 Pearson Education, Inc.

Shortest Path in a Weighted Graph

- FIGURE 29-19 A weighted graph and the possible paths from vertex A to vertex H

(a) A weighted graph



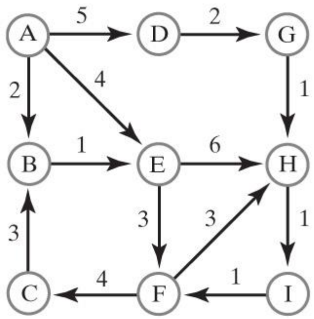
(b) Possible paths and their weights

Path	Weight
A → B → E → F → H	9
A → B → E → H	9
A → D → G → H	8
A → E → F → H	10
A → E → H	10

© 2019 Pearson Education, Inc.

Shortest Path in a Weighted Graph

- FIGURE 29-20 A trace of the traversal in the algorithm to find the cheapest



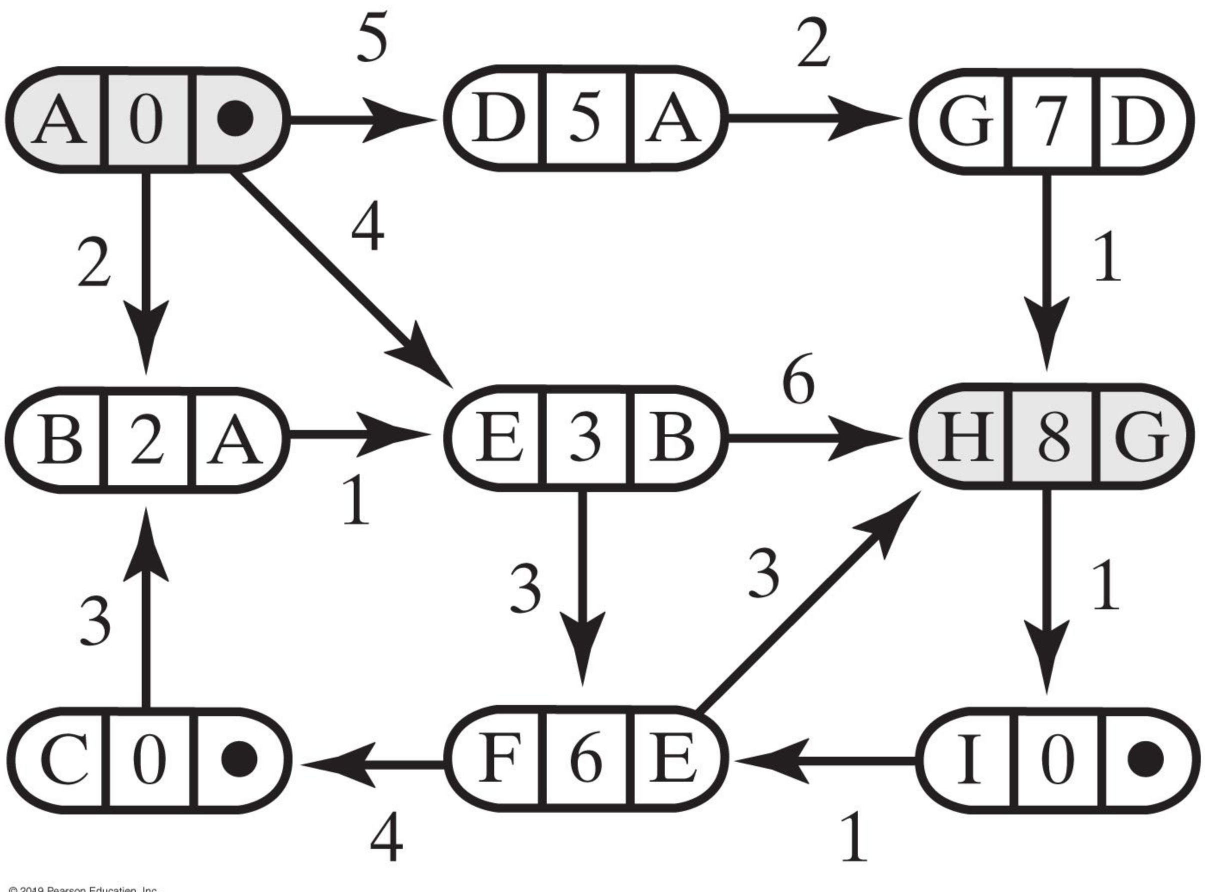
frontVertex	Visited vertex	nextNeighbor	Priority queue (front to back)
			A 0 •
A 0 •	A		*empty*
		B	B 2 A
		D	B 2 A, D 5 A
		E	B 2 A, E 4 A, D 5 A
B 2 A	B		E 4 A, D 5 A
		E	E 3 B, E 4 A, D 5 A
E 3 B	E		E 4 A, D 5 A
		F	E 4 A, D 5 A, F 6 E
		H	E 4 A, D 5 A, F 6 E, H 9 E
			D 5 A, F 6 E, H 9 E
D 5 A	D		F 6 E, H 9 E
		G	F 6 E, G 7 D, H 9 E
F 6 E	F		G 7 D, H 9 E
		H	G 7 D, H 9 E, H 9 F
		C	G 7 D, H 9 E, H 9 F, C 10 F
G 7 D	G		H 9 E, H 9 F, C 10 F
		H	H 8 G, H 9 E, H 9 F, C 10 F
H 8 G	H		H 9 E, H 9 F, C 10 F

© 2019 Pearson Education, Inc.

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

Shortest Path in a Weighted Graph

- FIGURE 29-21 The graph in Figure 29-19a after finding the cheapest path from vertex A to vertex H



The Shortest Path in a Weighted Graph

```
Algorithm getCheapestPath(originVertex, endVertex, path)
done = false
priorityQueue = a new priority queue
priorityQueue.add(new EntryPQ(originVertex, 0, null))
while (!done && !priorityQueue.isEmpty())
{
	frontEntry = priorityQueue.remove()
	frontVertex = vertex in frontEntry
	if (frontVertex is not visited)
	{
		Mark frontVertex as visited
		Set the cost of the path to frontVertex to the cost recorded in frontEntry
		Set the predecessor of frontVertex to the predecessor recorded in frontEntry
		if (frontVertex equals endVertex)
		done = true
		else
		{
			while (frontVertex has a neighbor)
			{
				nextNeighbor = next neighbor of frontVertex
				weightOfEdgeToNeighbor = weight of edge to nextNeighbor
				if (nextNeighbor is not visited)
				{
					nextCost = weightOfEdgeToNeighbor + cost of path to frontVertex
					priorityQueue.add(new EntryPQ(nextNeighbor, nextCost, frontVertex))
				}
			}
		}
	}
}
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

The Shortest Path in a Weighted Graph (Part 2)

```
// Traversal ends; now construct cheapest path
pathCost = cost of path to endVertex
path.push(endVertex)
vertex = endVertex
while (vertex has a predecessor)
{
        vertex = predecessor of vertex
        path.push(vertex)
}
return pathCost
```

Basic Graph Interface

```java
public interface BasicGraphInterface<T> {
                    /**
                     * Adds a given vertex to this graph.
                     *
                     * @param vertexLabel An object that labels the new vertex and is distinct from
                     *                    the labels of current vertices.
                     * @return True if the vertex is added, or false if not.
                     */
                    public boolean addVertex(T vertexLabel);

                    /**
                     * Adds a weighted edge between two given distinct vertices that are currently
                     * in this graph. The desired edge must not already be in the graph. In a
                     * directed graph, the edge points toward the second vertex given.
                     *
                     * @param begin      An object that labels the origin vertex of the edge.
                     * @param end        An object, distinct from begin, that labels the end vertex
                     *                   of the edge.
                     * @param edgeWeight The real value of the edge's weight.
                     * @return True if the edge is added, or false if not.
                     */
                    public boolean addEdge(T begin, T end, double edgeWeight);

                    /**
                     * Adds an unweighted edge between two given distinct vertices that are
                     * currently in this graph. The desired edge must not already be in the graph.
                     * In a directed graph, the edge points toward the second vertex given.
                     *
                     * @param begin An object that labels the origin vertex of the edge.
                     * @param end   An object, distinct from begin, that labels the end vertex of
                     *              the edge.
                     * @return True if the edge is added, or false if not.
                     */
                    public boolean addEdge(T begin, T end);

                    /**
                     * Sees whether an edge exists between two given vertices.
                     *
                     * @param begin An object that labels the origin vertex of the edge.
                     * @param end   An object that labels the end vertex of the edge.
                     * @return True if an edge exists.
                     */
                    public boolean hasEdge(T begin, T end);

                    /**
                     * Sees whether this graph is empty.
                     *
                     * @return True if the graph is empty.
                     */
                    public boolean isEmpty();

                    /**
                     * Gets the number of vertices in this graph.
                     *
                     * @return The number of vertices in the graph.
                     */
                    public int getNumberOfVertices();

                    /**
                     * Gets the number of edges in this graph.
                     *
                     * @return The number of edges in the graph.
                     */
                    public int getNumberOfEdges();

                    /**
                     * Removes all vertices and edges from this graph resulting in an empty graph.
                     */
                    public void clear();
}
```

REDISTRIBUTED

Java Interfaces for the ADT Graph

- FIGURE 29-22 A portion of the flight map in Figure 29-6

GraphAlgorithmsInterface

```java
public interface GraphAlgorithmsInterface<T> {
		/**
		 * Performs a breadth-first traversal of this graph.
		 *
		 * @param origin An object that labels the origin vertex of the traversal.
		 * @return A queue of labels of the vertices in the traversal, with the label of
		 *         the origin vertex at the queue's front.
		 */
		public QueueInterface<T> getBreadthFirstTraversal(T origin);

		/**
		 * Performs a depth-first traversal of this graph.
		 *
		 * @param origin An object that labels the origin vertex of the traversal.
		 * @return A queue of labels of the vertices in the traversal, with the label of
		 *         the origin vertex at the queue's front.
		 */
		public QueueInterface<T> getDepthFirstTraversal(T origin);

		/**
		 * Performs a topological sort of the vertices in this graph without cycles.
		 *
		 * @return A stack of vertex labels in topological order, beginning with the
		 *         stack's top.
		 */
		public StackInterface<T> getTopologicalOrder();

		/**
		 * Finds the shortest-length path between two given vertices in this graph.
		 *
		 * @param begin An object that labels the path's origin vertex.
		 * @param end   An object that labels the path's destination vertex.
		 * @param path  A stack of labels that is empty initially; at the completion of
		 *              the method, this stack contains the labels of the vertices along
		 *              the shortest path; the label of the origin vertex is at the top,
		 *              and the label of the destination vertex is at the bottom
		 * @return The length of the shortest path.
		 */
		public int getShortestPath(T begin, T end, StackInterface<T> path);

		/**
		 * Finds the least-cost path between two given vertices in this graph.
		 *
		 * @param begin An object that labels the path's origin vertex.
		 * @param end   An object that labels the path's destination vertex.
		 * @param path  A stack of labels that is empty initially; at the completion of
		 *              the method, this stack contains the labels of the vertices along
		 *              the cheapest path; the label of the origin vertex is at the top,
		 *              and the label of the destination vertex is at the bottom
		 * @return The cost of the cheapest path.
		 */
		public double getCheapestPath(T begin, T end, StackInterface<T> path);
}
```

REDISTRIBUTED

GraphInterface

- Combination of basic graph and algorithms interfaces

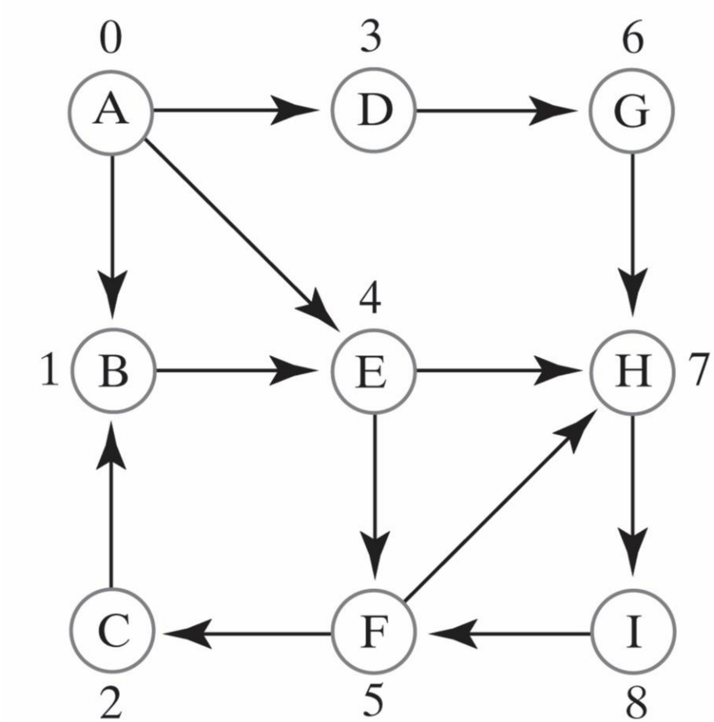
```
public interface GraphInterface<T> extends BasicGraphInterface<T>,GraphAlgorithmsInterface<T> {
}
```

Overview of Two Implementations

- Two common implementations of the ADT graph
  - Array:
    - Typically a two-dimensional array called an *adjacency matrix*
  - List:
    - Referred to as an *adjacency list*.

FIGURE 30-1 An unweighted, directed graph and its adjacency matrix

(a) A graph



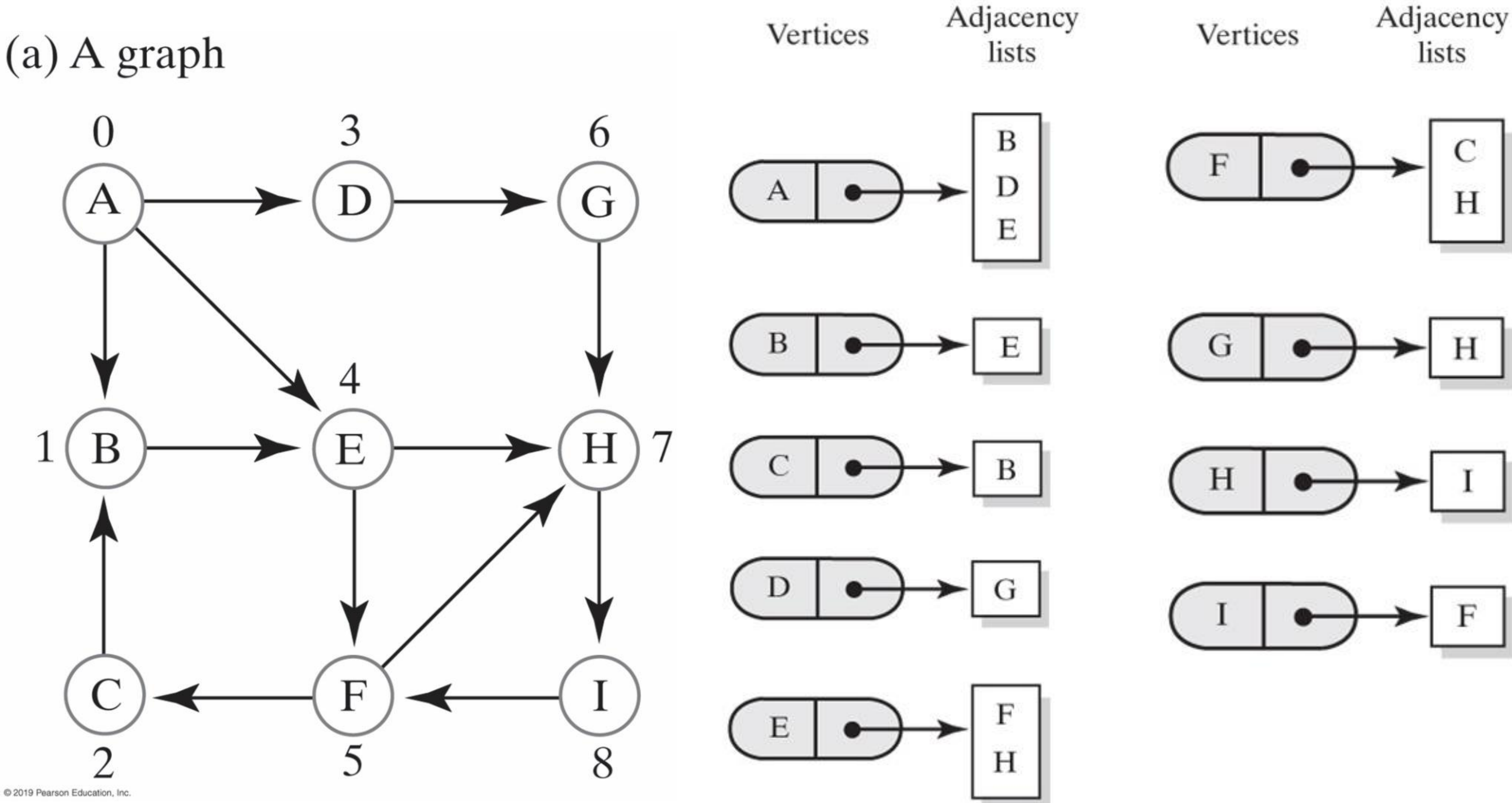
© 2019 Pearson Education, Inc.

(b) The graph's adjacency matrix

	A	B	C	D	E	F	G	H	I	
A		T		T	T					0
B					T					1
C		T								2
D							T			3
E						T		T		4
F			T					T		5
G								T		6
H									T	7
I						T				8
	0	1	2	3	4	5	6	7	8	

© 2019 Pearson Education, Inc.

FIGURE 30-2 Adjacency lists for the directed graph in Figure 30-1a



Vertices and Edges

- Specifying the Class **`Vertex`**
  - Identify vertices
  - Visit vertices
  - Adjacency list
  - Path operations

©Michael Hrybyk and
Pearson Education NOT TO BE
REDISTRIBUTED

Vertex Interface

```java
/**
 * Gets this vertex's label.
 *
 * @return The object that labels the vertex.
 */
public T getLabel();

/** Marks this vertex as visited. */
public void visit();

/** Removes this vertex's visited mark. */
public void unvisit();

/**
 * Sees whether the vertex is marked as visited.
 *
 * @return True if the vertex is visited.
 */
public boolean isVisited();

/**
 * Connects this vertex and a given vertex with a weighted edge. The two
 * vertices cannot be the same, and must not already have this edge between
 * them. In a directed graph, the edge points toward the given vertex.
 *
 * @param endVertex  A vertex in the graph that ends the edge.
 * @param edgeWeight A real-valued edge weight, if any.
 * @return True if the edge is added, or false if not.
 */
public boolean connect(VertexInterface<T> endVertex, double edgeWeight);

/**
 * Connects this vertex and a given vertex with an unweighted edge. The two
 * vertices cannot be the same, and must not already have this edge between
 * them. In a directed graph, the edge points toward the given vertex.
 *
 * @param endVertex A vertex in the graph that ends the edge.
 * @return True if the edge is added, or false if not.
 */
public boolean connect(VertexInterface<T> endVertex);

/**
 * Creates an iterator of this vertex's neighbors by following all edges that
 * begin at this vertex.
 *
 * @return An iterator of the neighboring vertices of this vertex.
 */
public Iterator<VertexInterface<T>> getNeighborIterator();

/**
 * Creates an iterator of the weights of the edges to this vertex's neighbors.
 *
 * @return An iterator of edge weights for edges to neighbors of this vertex.
 */
public Iterator<Double> getWeightIterator();
```

Vertex Interface

```java
/**
 * Sees whether this vertex has at least one neighbor.
 *
 * @return True if the vertex has a neighbor.
 */
public boolean hasNeighbor();

/**
 * Gets an unvisited neighbor, if any, of this vertex.
 *
 * @return Either a vertex that is an unvisited neighbor or null if no such
 *         neighbor exists.
 */
public VertexInterface<T> getUnvisitedNeighbor();

/**
 * Records the previous vertex on a path to this vertex.
 *
 * @param predecessor The vertex previous to this one along a path.
 */
public void setPredecessor(VertexInterface<T> predecessor);

/**
 * Gets the recorded predecessor of this vertex.
 *
 * @return Either this vertex's predecessor or null if no predecessor was
 *         recorded.
 */
public VertexInterface<T> getPredecessor();

/**
 * Sees whether a predecessor was recorded for this vertex.
 *
 * @return True if a predecessor was recorded.
 */
public boolean hasPredecessor();

/**
 * Records the cost of a path to this vertex.
 *
 * @param newCost The cost of the path.
 */
public void setCost(double newCost);

/**
 * Gets the recorded cost of the path to this vertex.
 *
 * @return The cost of the path.
 */
public double getCost();
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

Edge class

```java
/**
 * Edge in a graph. An edge connects two vertices. We will
 * just keep the vertex at the end.
 *
 * Weight is also kept.
 */
protected class Edge implements Comparable<Edge> {
	private VertexInterface<T> vertex; // Vertex at end of edge
	private double weight;

	protected Edge(VertexInterface<T> endVertex, double edgeWeight) {
		vertex = endVertex;
		weight = edgeWeight;
	} // end constructor

	protected Edge(VertexInterface<T> endVertex) {
		vertex = endVertex;
		weight = 0;
	} // end constructor

	protected VertexInterface<T> getEndVertex() {
		return vertex;
	} // end getEndVertex

	protected double getWeight() {
		return weight;
	} // end getWeight

	@Override
	public int compareTo(Edge other) {

		// so we can reuse previous ArrayList and others

		throw new java.lang.UnsupportedOperationException("Comparison not supported.");
	}
}
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

Vertex class

- Implement with adjacency list (linked list)
- Could use ArrayList as well

```
class Vertex<T> implements VertexInterface<T> {
          private T label;
          private ListWithIteratorInterface<Edge> edgeList; // Edges to neighbors
          private boolean visited; // True if visited
          private VertexInterface<T> previousVertex; // On path to this vertex
          private double cost; // Of path to this vertex

          public Vertex(T vertexLabel) {
                    label = vertexLabel;
                    edgeList = new LinkedListWithIterator<>();
                    visited = false;
                    previousVertex = null;
                    cost = 0;
          }
```

Vertex class basic operations

```
public T getLabel() {
        return label;
} // end getLabel

public boolean hasPredecessor() {
        return previousVertex != null;
} // end hasPredecessor

public void setPredecessor(VertexInterface<T> predecessor) {
        previousVertex = predecessor;
} // end setPredecessor

public VertexInterface<T> getPredecessor() {
        return previousVertex;
} // end getPredecessor

public void visit() {
        visited = true;
} // end visit

public void unvisit() {
        visited = false;
} // end unvisit

public boolean isVisited() {
        return visited;
} // end isVisited

public double getCost() {
        return cost;
} // end getCost

public void setCost(double newCost) {
        cost = newCost;
} // end setCost

public String toString() {
        return label.toString();
} // end toString
```

Vertex class – connect()

- connects this object to another vertex
- make sure the other object is a different one!
  - Need to look at all of the neighbors as well.

```
    public boolean connect(VertexInterface<T> endVertex, double edgeWeight) {
        boolean result = false;

        if (!this.equals(endVertex)) { // Vertices are distinct
            Iterator<VertexInterface<T>> neighbors = getNeighborIterator();
            boolean duplicateEdge = false;

            while (!duplicateEdge && neighbors.hasNext()) {
                VertexInterface<T> nextNeighbor = neighbors.next();
                if (endVertex.equals(nextNeighbor))
                    duplicateEdge = true;
            } // end while

            if (!duplicateEdge) {
                edgeList.add(new Edge(endVertex, edgeWeight));
                result = true;
            } // end if
        } // end if

        return result;
    } // end connect

    public boolean connect(VertexInterface<T> endVertex) {
        return connect(endVertex, 0);
    }
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

Vertex class – Neighbor iterator

```
private class NeighborIterator implements Iterator<VertexInterface<T>> {
      private Iterator<Edge> edges;

      private NeighborIterator() {
            edges = edgeList.iterator();
      } // end default constructor

      public boolean hasNext() {
            return edges.hasNext();
      } // end hasNext

      public VertexInterface<T> next() {
            VertexInterface<T> nextNeighbor = null;

            if (edges.hasNext()) {
                  Edge edgeToNextNeighbor = edges.next();
                  nextNeighbor = edgeToNextNeighbor.getEndVertex();
            } else
                  throw new NoSuchElementException();

            return nextNeighbor;
      } // end next

      public void remove() {
            throw new UnsupportedOperationException();
      } // end remove
}

public Iterator<VertexInterface<T>> getNeighborIterator() {
      return new NeighborIterator();
}
```

Vertex class – Weight Iterator

- needed because Edge is not publicly accessible

```
    private class WeightIterator implements Iterator<Double> {
        private Iterator<Edge> edges;

        private WeightIterator() {
            edges = edgeList.iterator();
        } // end default constructor

        public boolean hasNext() {
            return edges.hasNext();
        } // end hasNext

        public Double next() {
            Double edgeWeight = 0.0;
            if (edges.hasNext()) {
                Edge edgeToNextNeighbor = edges.next();
                edgeWeight = edgeToNextNeighbor.getWeight();
            } else
                throw new NoSuchElementException();

            return edgeWeight;
        } // end next

        public void remove() {
            throw new UnsupportedOperationException();
        } // end remove
    } // end WeightIterator

    public Iterator<Double> getWeightIterator() {
        return new WeightIterator();
    }
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

Vertex class – Neighbor methods

- getUnvisitedNeighbor() is useful for topological ordering and shortest path

```
public boolean hasNeighbor() {
      return !edgeList.isEmpty();
} // end hasNeighbor

public VertexInterface<T> getUnvisitedNeighbor() {
      VertexInterface<T> result = null;

      Iterator<VertexInterface<T>> neighbors = getNeighborIterator();
      while (neighbors.hasNext() && (result == null)) {
            VertexInterface<T> nextNeighbor = neighbors.next();
            if (!nextNeighbor.isVisited())
                  result = nextNeighbor;
      } // end while

      return result;
}
```

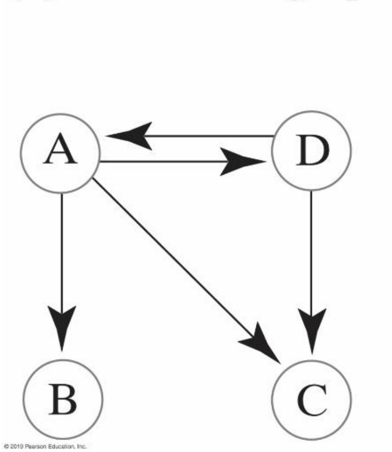
Vertex class – determining equality

- Note use of getClass() to make sure they are the same
- Simply checks the label (used as identifier)

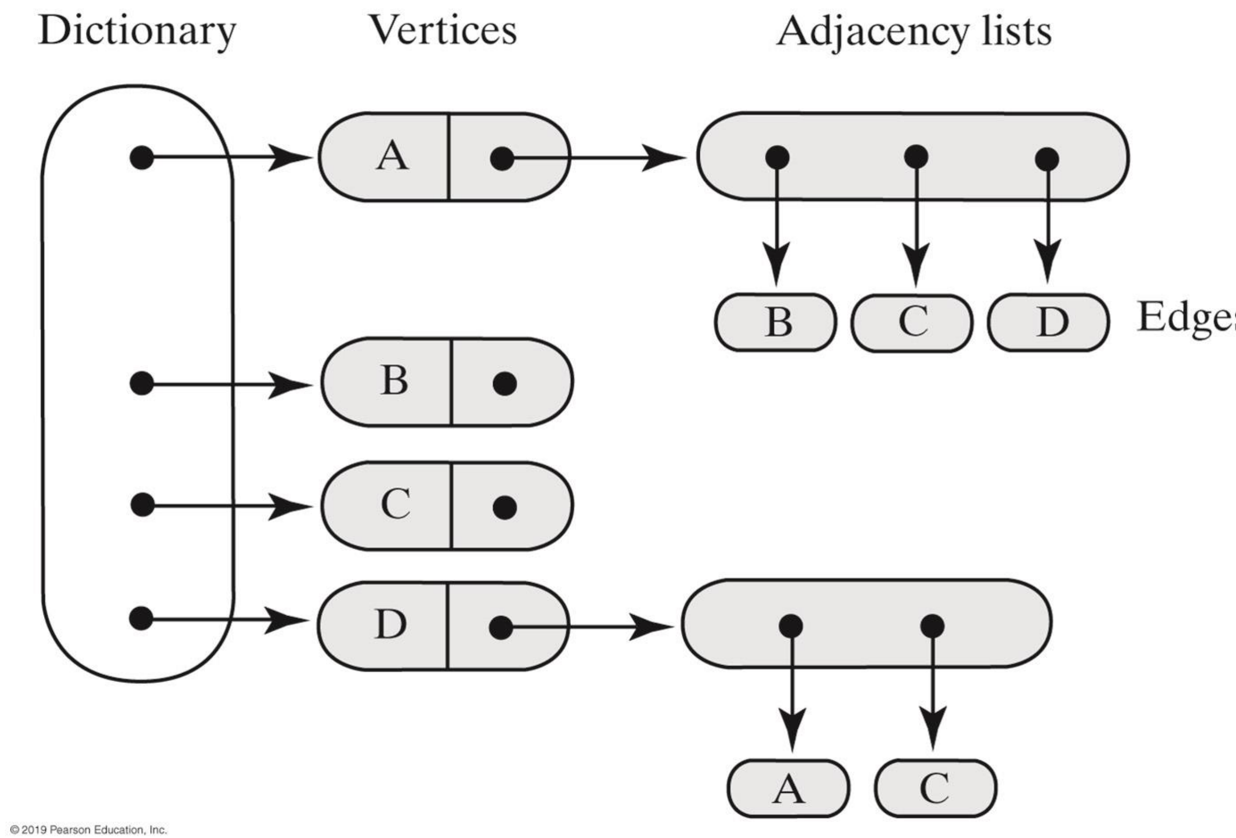
```
public boolean equals(Object other) {
    boolean result;

    if ((other == null) || (getClass() != other.getClass()))
        result = false;
    else {
        // The cast is safe within this else clause
        @SuppressWarnings("unchecked")
        Vertex<T> otherVertex = (Vertex<T>) other;
        result = label.equals(otherVertex.label);
    } // end if

    return result;
}
```

Directed graph as a dictionary of vertices

(a) A directed graph



(b) An implementation of the graph using a dictionary



DirectedGraph implementation

- Use of linked dictionary, but could use others as well
  - ArrayDictionary, HashMapDictionary, …
- Need to added vertices and edges with other methods
- Dictionary key will be the label, and vertex the value.

```
public class DirectedGraph<T> implements GraphInterface<T> {
   private DictionaryInterface<T, VertexInterface<T>> vertices;
   private int edgeCount;

   public DirectedGraph() {
      vertices = new UnsortedLinkedDictionary<>();
      edgeCount = 0;
   } // end default constructor
```

DirectedGraph – addVertex()

- Add a vertex with a label.
- Could be a string, or any other identifying object

```
public boolean addVertex(T vertexLabel) {
    VertexInterface<T> addOutcome = vertices.add(vertexLabel, new Vertex<>(vertexLabel));
    return addOutcome == null; // Was addition to dictionary successful?
}
```

DirectedGraph – adding edges

- Take two labels and see if they exist
- If so, connect the begin to the end, which creates the edge

```
public boolean addEdge(T begin, T end, double edgeWeight) {
      boolean result = false;
      VertexInterface<T> beginVertex = vertices.getValue(begin);
      VertexInterface<T> endVertex = vertices.getValue(end);
      if ((beginVertex != null) && (endVertex != null))
            result = beginVertex.connect(endVertex, edgeWeight);
      if (result)
            edgeCount++;
      return result;
} // end addEdge

public boolean addEdge(T begin, T end) {
      return addEdge(begin, end, 0);
}
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

DirectedGraph – hasEdge()

- Similar to addEdge(), in that we check to see if begin and end exist
- If so see if begin has a neighbor that is the same as end

```
public boolean hasEdge(T begin, T end) {
    boolean found = false;
    VertexInterface<T> beginVertex = vertices.getValue(begin);
    VertexInterface<T> endVertex = vertices.getValue(end);
    if ((beginVertex != null) && (endVertex != null)) {
        Iterator<VertexInterface<T>> neighbors = beginVertex.getNeighborIterator();
        while (!found && neighbors.hasNext()) {
            VertexInterface<T> nextNeighbor = neighbors.next();
            if (endVertex.equals(nextNeighbor))
                found = true;
        } // end while
    } // end if

    return found;
}
```

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

DirectedGraph – other methods

```
public boolean isEmpty() {
        return vertices.isEmpty();
} // end isEmpty

public void clear() {
        vertices.clear();
        edgeCount = 0;
} // end clear

public int getNumberOfVertices() {
        return vertices.getSize();
} // end getNumberOfVertices

public int getNumberOfEdges() {
        return edgeCount;
}
```

DirectedGraph – resetVertices()

- make each vertex unvisited, and cost set to zero

```java
protected void resetVertices() {
    Iterator<VertexInterface<T>> vertexIterator = vertices.getValueIterator();
    while (vertexIterator.hasNext()) {
        VertexInterface<T> nextVertex = vertexIterator.next();
        nextVertex.unvisit();
        nextVertex.setCost(0);
        nextVertex.setPredecessor(null);
    } // end while
}
```

Efficiency of Basic Graph Operations

- The performance of basic operations of the ADT graph when implemented by using adjacency lists

`addVertex`	$O(n)$
`addEdge`	$O(n)$
`hasEdge`	$O(n)$
`isEmpty`	$O(1)$
`getNumberOfVertices`	$O(1)$
`getNumberOfEdges`	$O(1)$
`Clear`	$O(1)$

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

DirectedGraph - BreadthFirstTraversal

- Note use of linked list queue, but any queue will do
- Queue of vertices
- Dequeue, then visit neighbors, enqueue each
- Record traversal order in separate queue

```
public QueueInterface<T> getBreadthFirstTraversal(T origin) {
    resetVertices();
    QueueInterface<T> traversalOrder = new LinkedQueue<>();
    QueueInterface<VertexInterface<T>> vertexQueue = new LinkedQueue<>();

    VertexInterface<T> originVertex = vertices.getValue(origin);
    originVertex.visit();
    traversalOrder.enqueue(origin); // Enqueue vertex label
    vertexQueue.enqueue(originVertex); // Enqueue vertex

    while (!vertexQueue.isEmpty()) {
        VertexInterface<T> frontVertex = vertexQueue.dequeue();
        Iterator<VertexInterface<T>> neighbors = frontVertex.getNeighborIterator();

        while (neighbors.hasNext()) {
            VertexInterface<T> nextNeighbor = neighbors.next();
            if (!nextNeighbor.isVisited()) {
                nextNeighbor.visit();
                traversalOrder.enqueue(nextNeighbor.getLabel());
                vertexQueue.enqueue(nextNeighbor);
            } // end if
        } // end while
    } // end while
    return traversalOrder;
}
```

Directed Graph – shortest path

- Use cost as hop count

```
public int getShortestPath(T begin, T end, StackInterface<T> path) {
     resetVertices();
     boolean done = false;
     QueueInterface<VertexInterface<T>> vertexQueue = new LinkedQueue<>();
     VertexInterface<T> originVertex = vertices.getValue(begin);
     VertexInterface<T> endVertex = vertices.getValue(end);
     originVertex.visit();

     vertexQueue.enqueue(originVertex);
     while (!done && !vertexQueue.isEmpty()) {
          VertexInterface<T> frontVertex = vertexQueue.dequeue();
          Iterator<VertexInterface<T>> neighbors = frontVertex.getNeighborIterator();
          while (!done && neighbors.hasNext()) {
               VertexInterface<T> nextNeighbor = neighbors.next();
               if (!nextNeighbor.isVisited()) {
                    nextNeighbor.visit();
                    nextNeighbor.setCost(1 + frontVertex.getCost());
                    nextNeighbor.setPredecessor(frontVertex);
                    vertexQueue.enqueue(nextNeighbor);
               } // end if

               if (nextNeighbor.equals(endVertex))
                    done = true;
          } // end while
     } // end while

     // Traversal ends; construct shortest path
     int pathLength = (int) endVertex.getCost();
     path.push(endVertex.getLabel());

     VertexInterface<T> vertex = endVertex;
     while (vertex.hasPredecessor()) {
          vertex = vertex.getPredecessor();
          path.push(vertex.getLabel());
     } // end while

     return pathLength;
}
```

DirectedGraph – topological order

- Look for an unvisited terminal vertex through ALL vertices
- If found, push onto stack

```
public StackInterface<T> getTopologicalOrder() {
        resetVertices();

        // for each vertex in the graph, see if any of its
        // neighbors are terminal (unvisited and ONLY visited neighbors)
        // if so, push it onto the stack

        StackInterface<T> vertexStack = new LinkedStack<>();
        int numberOfVertices = getNumberOfVertices();
        for (int counter = 0; counter < numberOfVertices; counter++) {
                VertexInterface<T> nextVertex = findTerminal();
                nextVertex.visit();
                vertexStack.push(nextVertex.getLabel());
        } // end for

        return vertexStack;
} // end getTopologicalOrder

/**
 * Find the first vertex that is a terminal. That is, it has ONLY
 * visited neighbors.
 * @return
 */
protected VertexInterface<T> findTerminal() {
        boolean found = false;
        VertexInterface<T> result = null;

        // search through all of the vertices in the graph.

        Iterator<VertexInterface<T>> vertexIterator = vertices.getValueIterator();

        while (!found && vertexIterator.hasNext()) {
                VertexInterface<T> nextVertex = vertexIterator.next();

                // If nextVertex is unvisited AND has only visited neighbors)
                // then we have found our terminal vertex

                if (!nextVertex.isVisited()) {
                        if (nextVertex.getUnvisitedNeighbor() == null) {
                                found = true;
                                result = nextVertex;
                        } // end if
                } // end if
        } // end while

        return result;
}
```