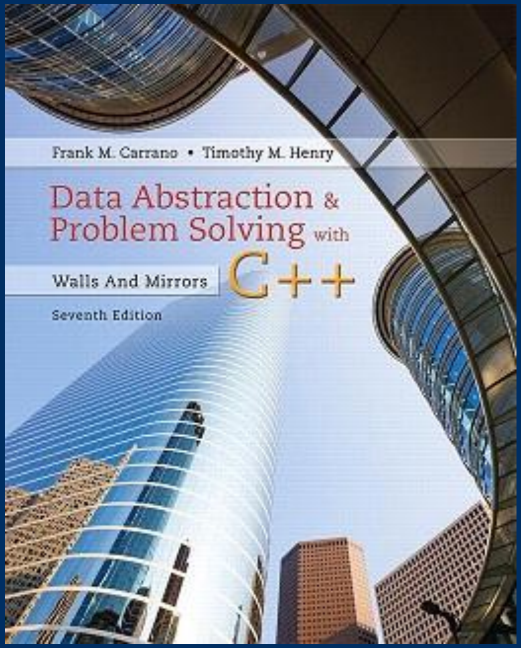
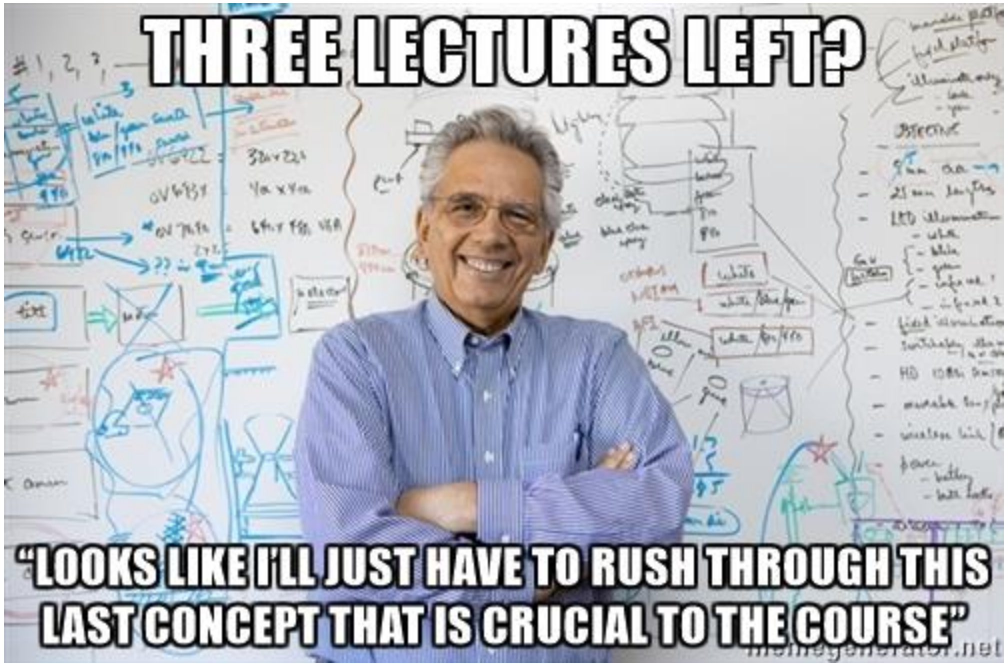
Graphs

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

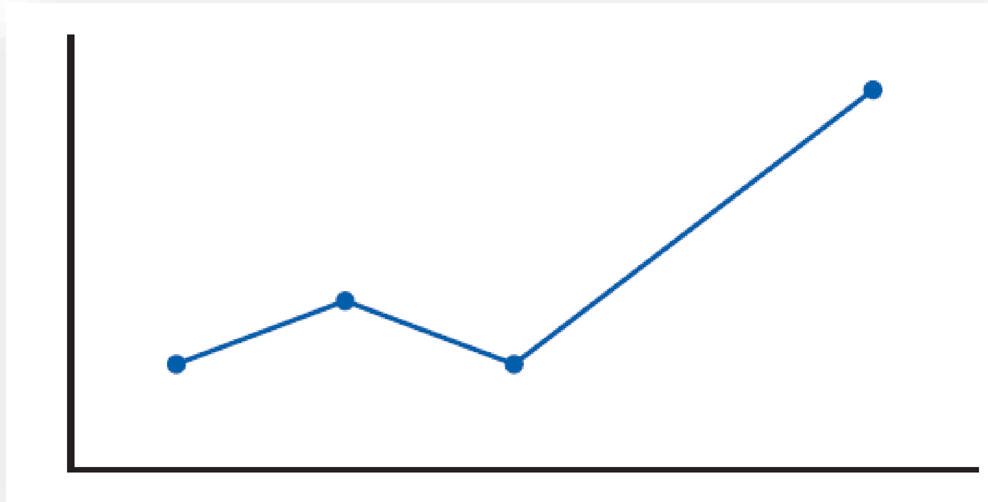

THREE LECTURES LEFT?
"LOOKS LIKE I'LL JUST HAVE TO RUSH THROUGH THIS LAST CONCEPT THAT IS CRUCIAL TO THE COURSE"
memegenerator.net

Reminders

- Assignment 7 is available
  - Due Wednesday, May 7th at 2pm
  - TA
    - Shehryar Khattak,
      Email: *shehryar [at] nevada {dot} unr {dot} edu*,
      Office Hours: Friday, 11:00 am - 1:00 pm at ARF 116
- Assignment 8 is available
  - Due Wednesday, May 16th at 2pm
  - TA
    - Athanasia Katsila,
      Email: *akatsila [at] nevada {dot} unr {dot} edu*,
      Office Hours: Thursdays, 10:30 am - 12:30 pm at SEM 211
- Quiz 11 is due tonight

Terminology

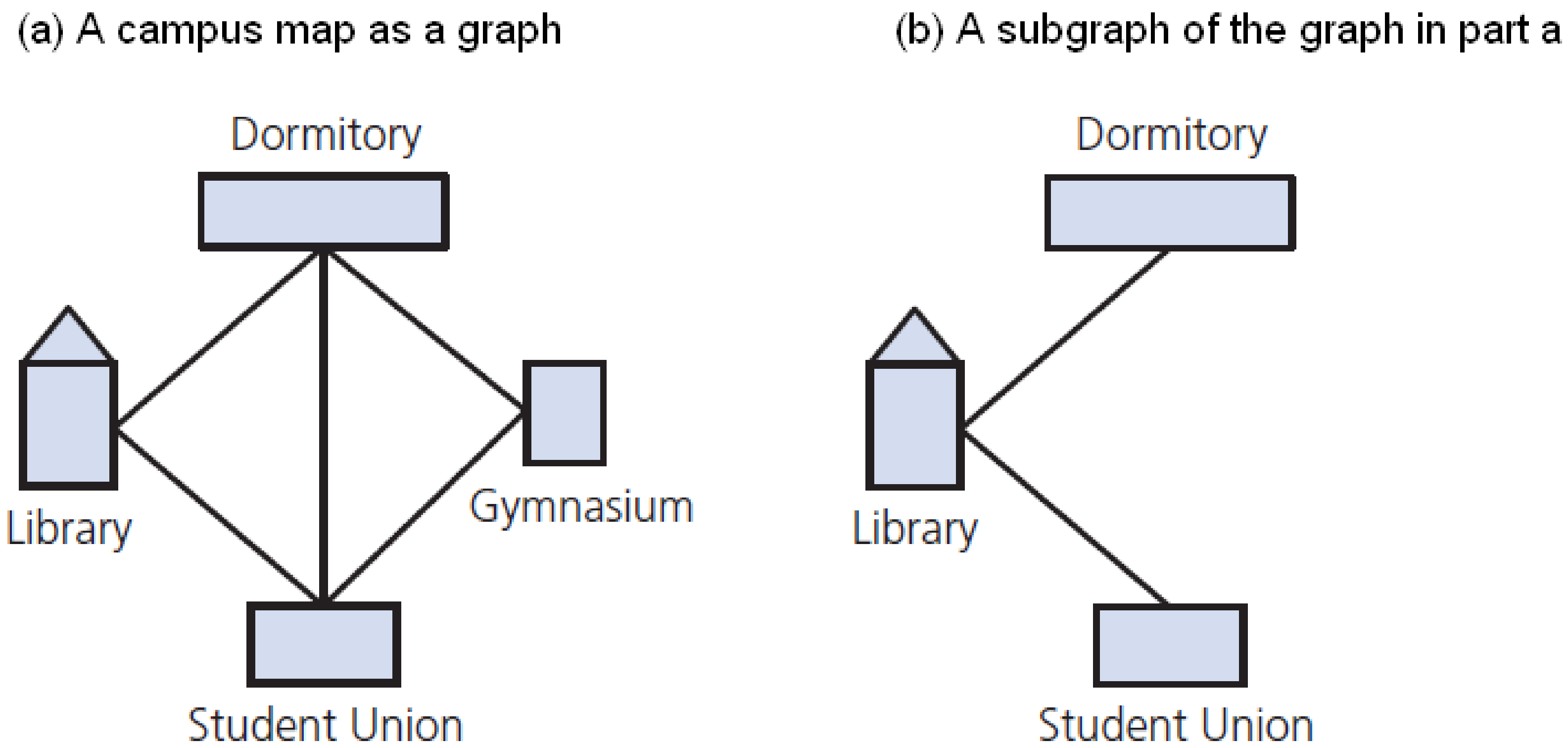
An ordinary line graph



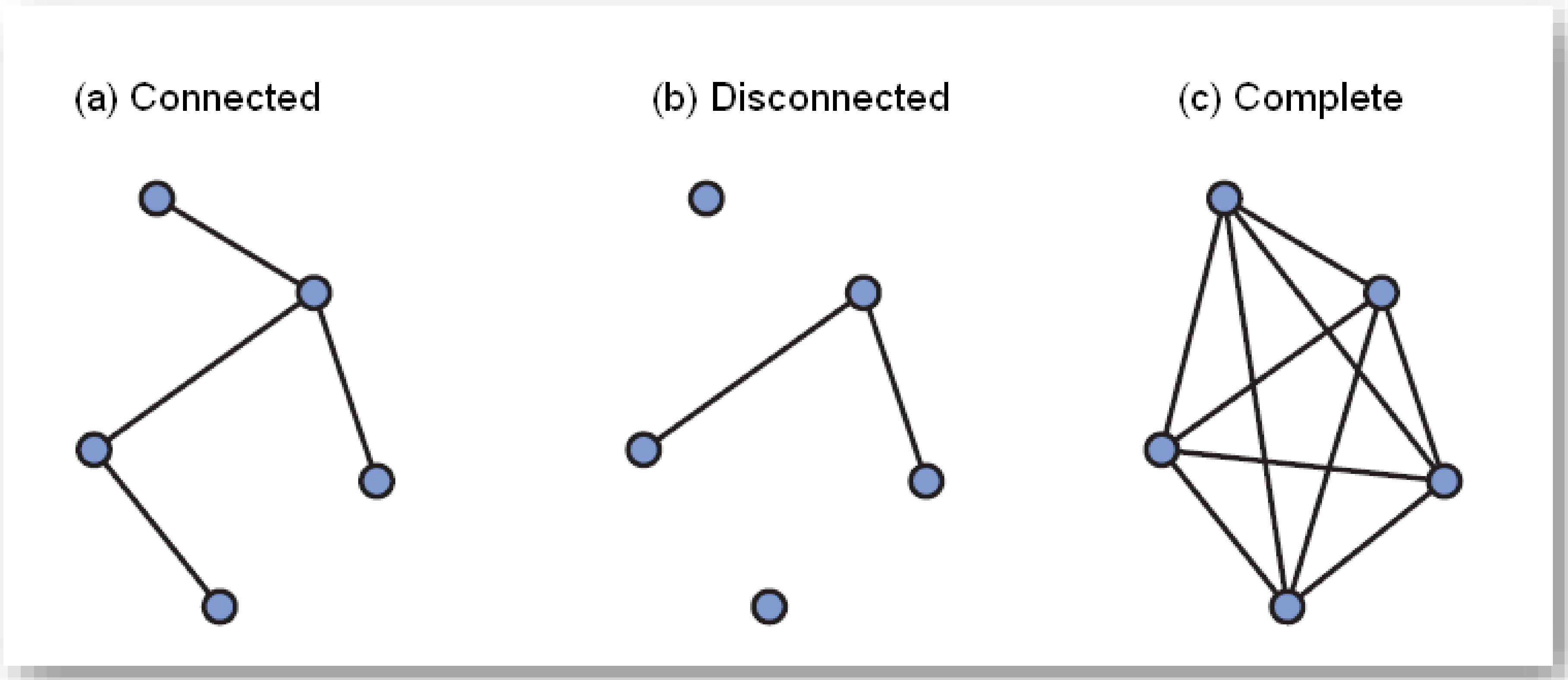
- In the context of this course, graphs represent relations among data items
- G = { V, E}
  – A graph is a set of vertices (nodes) and
  – A set of edges that connect the vertices

Terminology

- A graph and one of its subgraphs

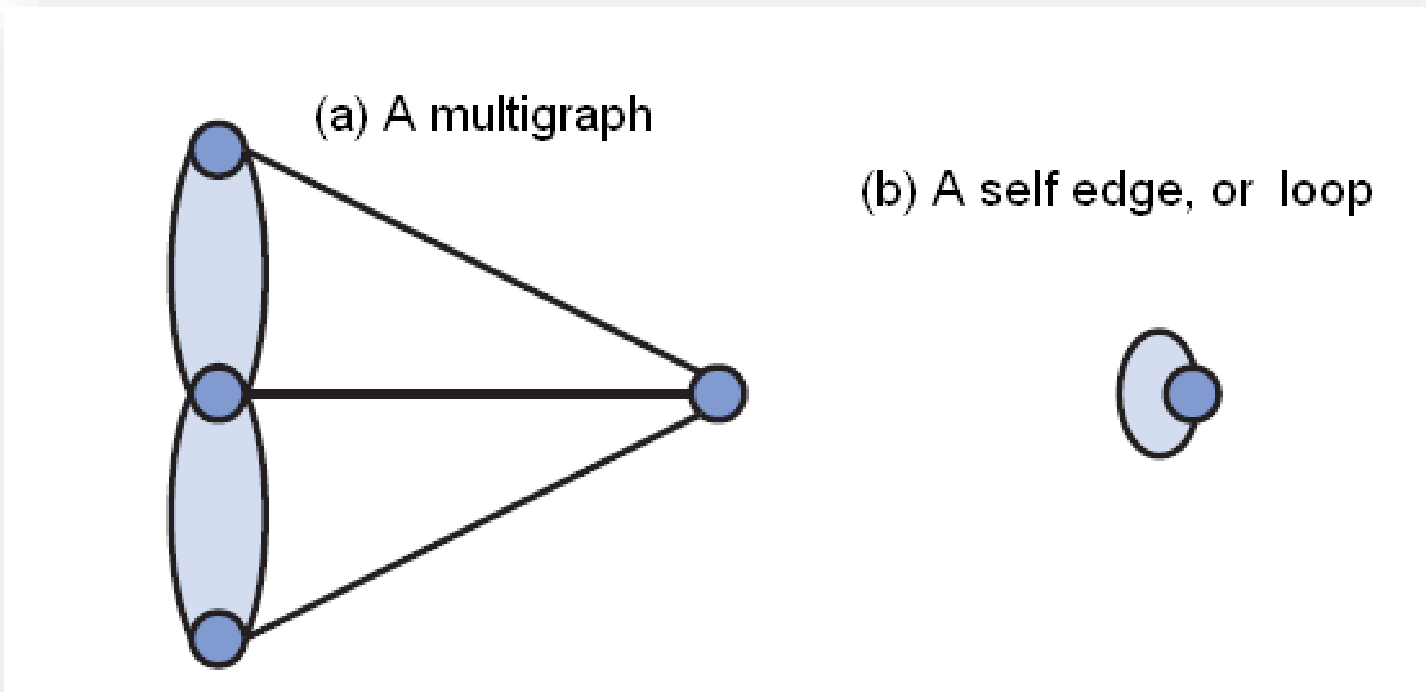


Terminology



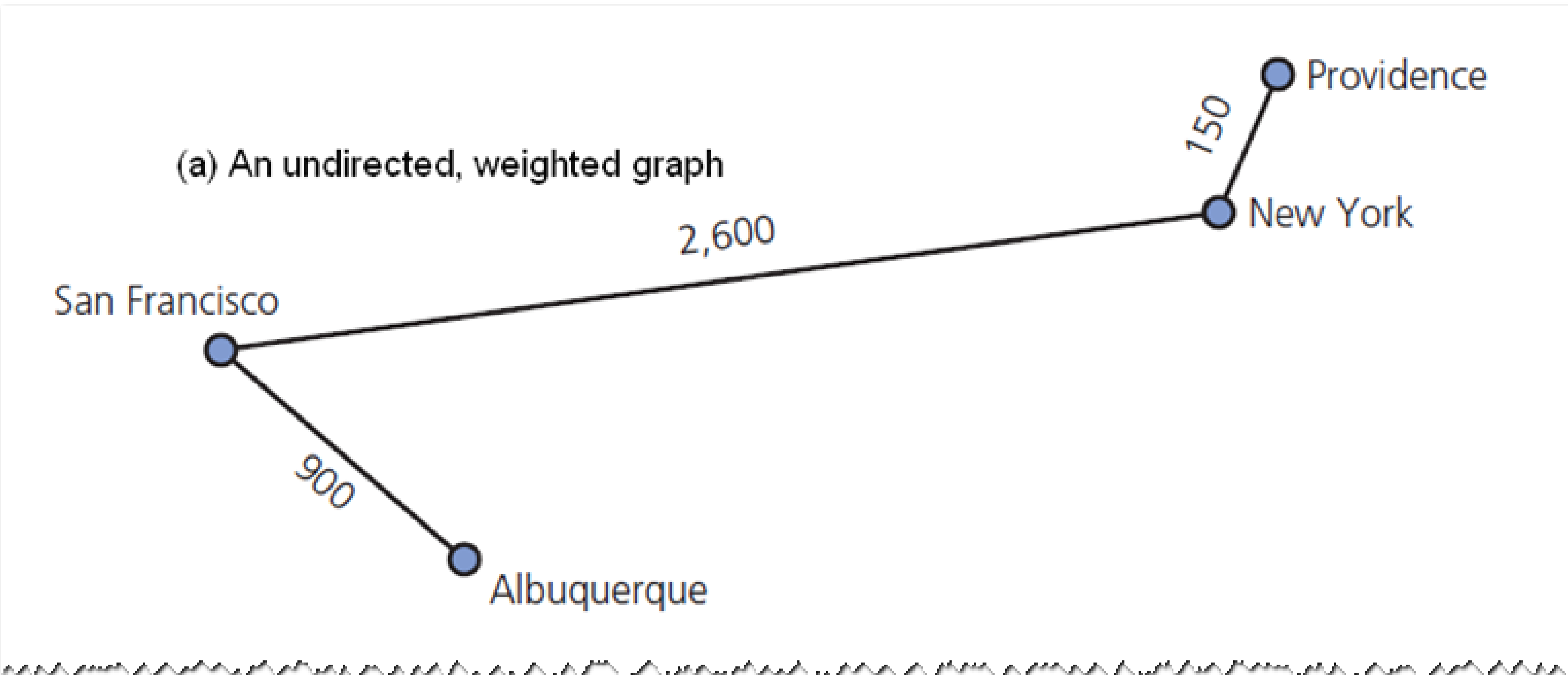
- Examples of graphs that are either connected, disconnected, or complete

Terminology



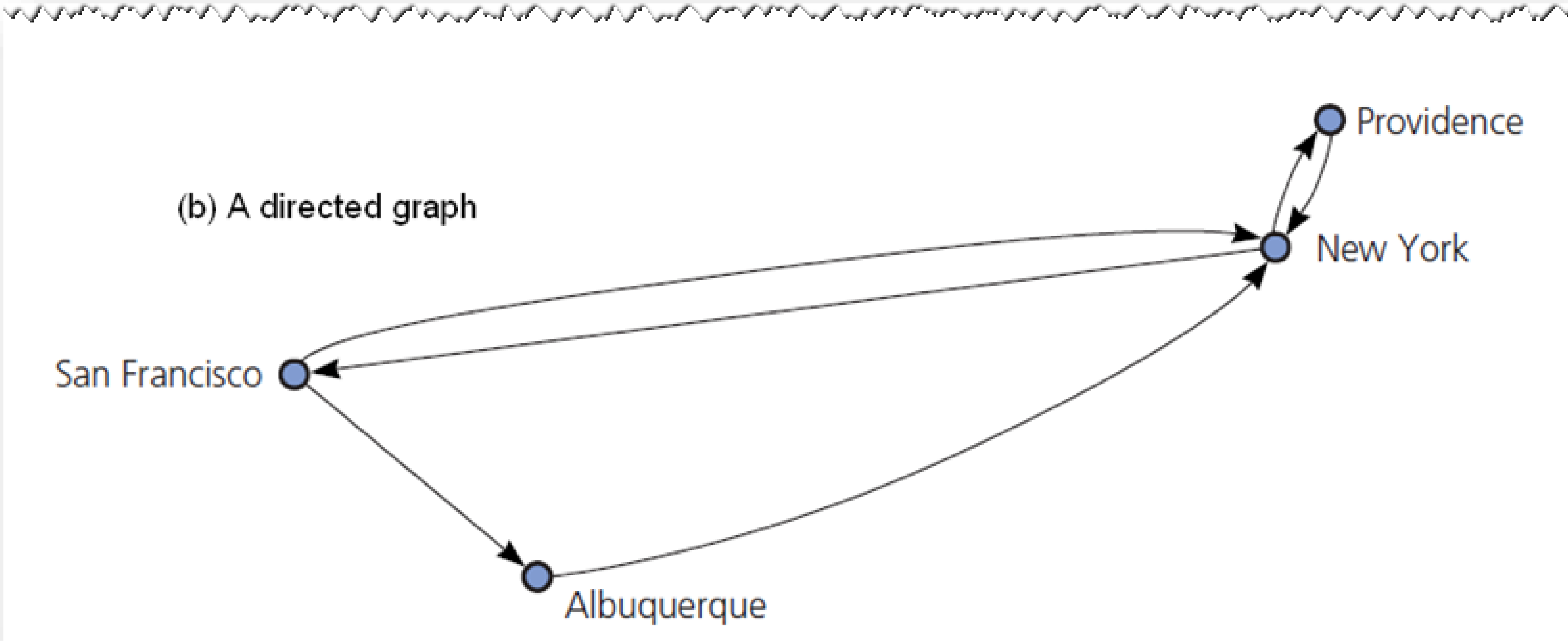
- Graph-like structures that are not graphs

Terminology



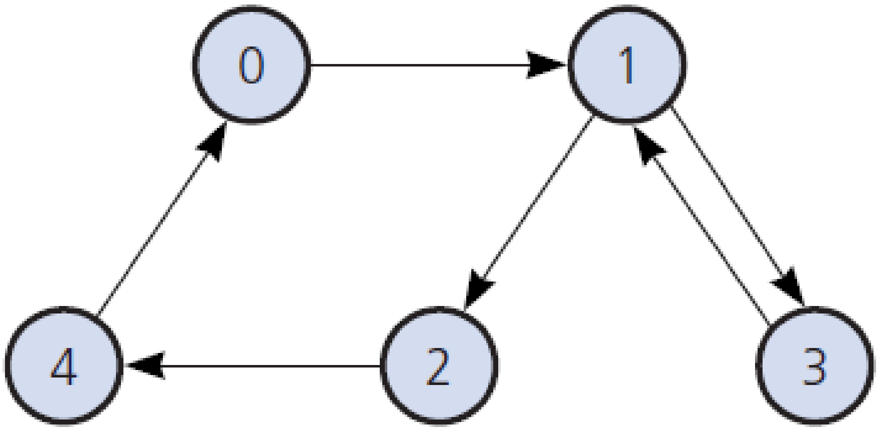
- Examples of graphs

Terminology

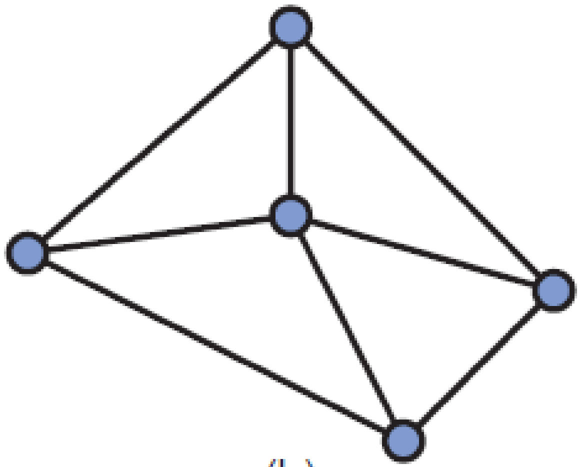


- Examples of graphs

Terminology



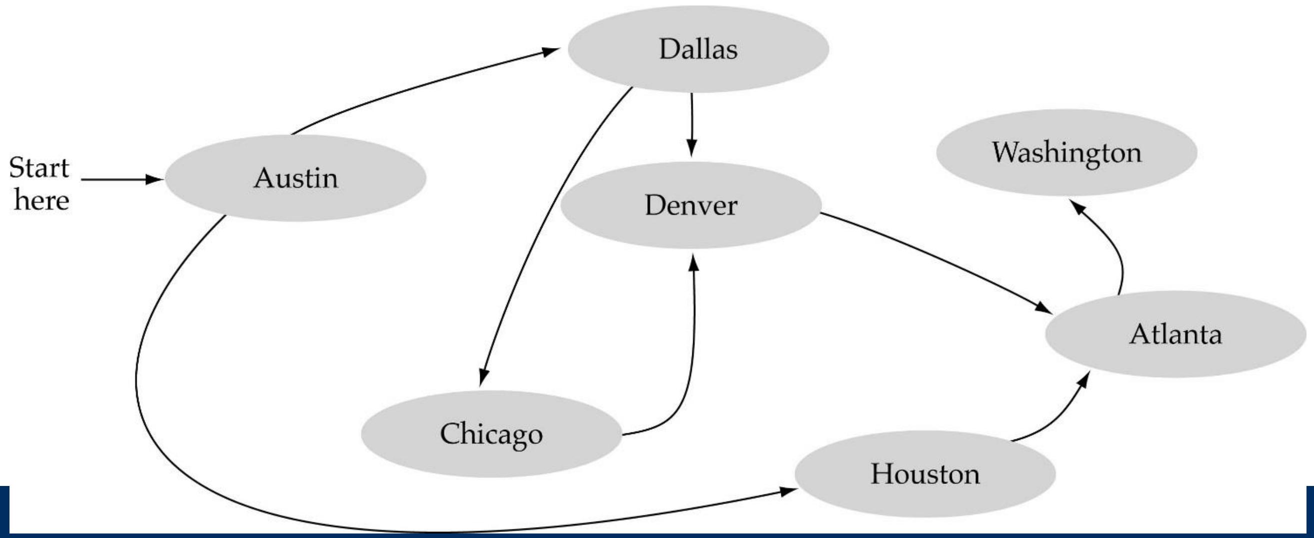
(a)


(b)

- Graphs for Checkpoint Questions 1, 3, 4, 5

What is a graph?

- A data structure that consists of a set of nodes (vertices) and a set of edges between the vertices.
- The set of edges describes relationships among the vertices.



Formally

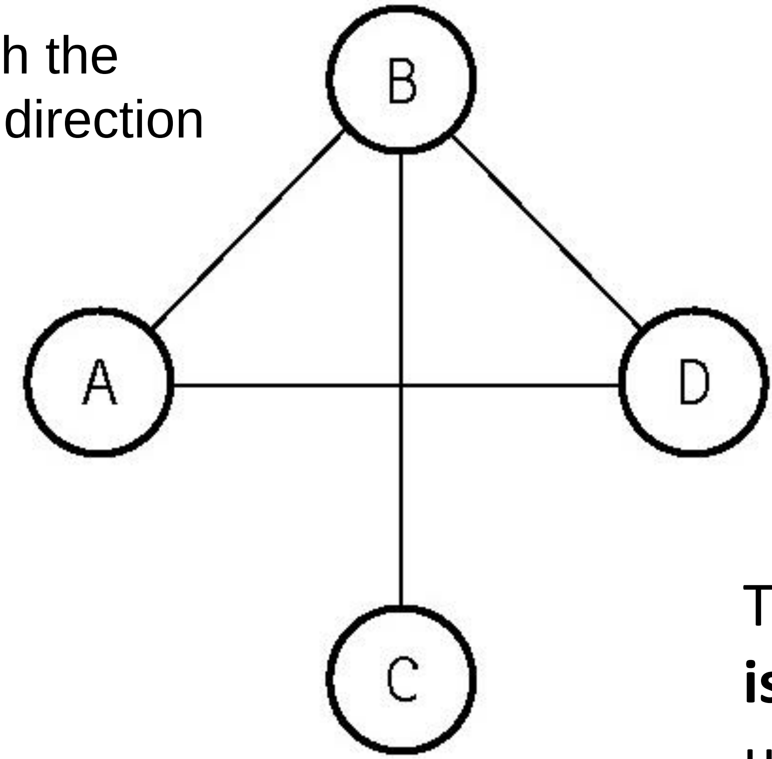
- a graph G is defined as follows:

$$G = (V,E)$$

- where
  - V(G) is a finite, nonempty set of vertices
  - E(G) is a set of edges
    - written as pairs of vertices

An undirected graph

A graph in which the edges have no direction

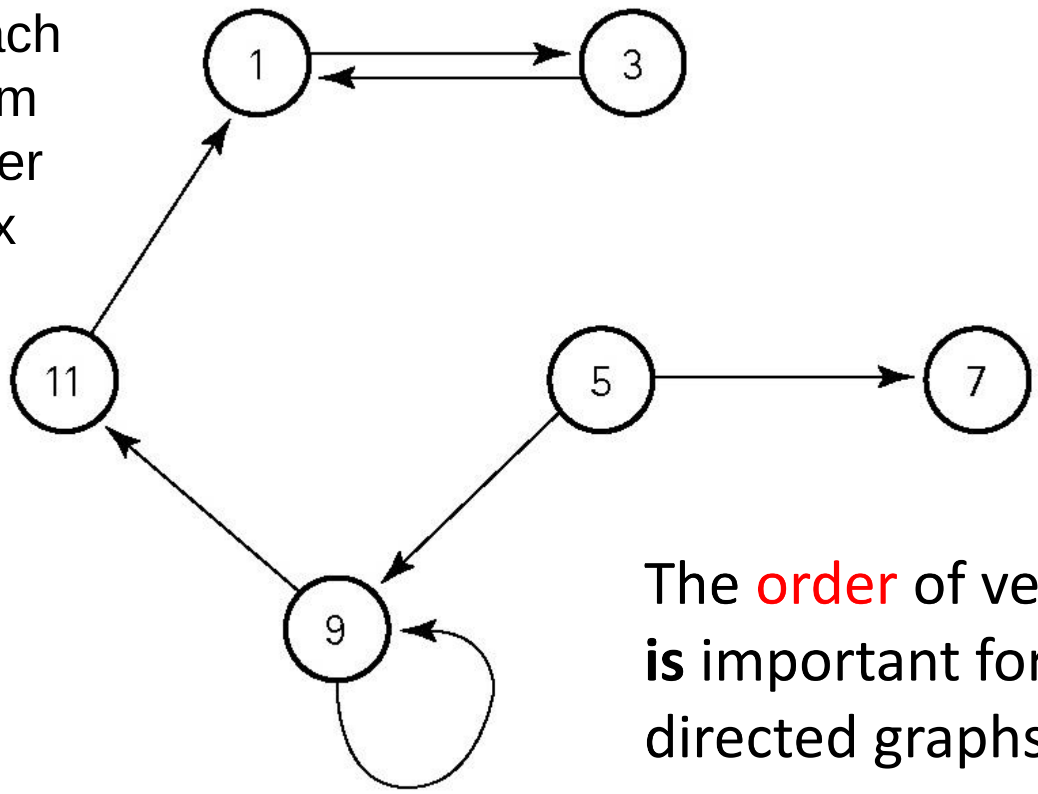


The order of vertices in E **is not** important for undirected graphs!!

V(Graph1) = {A, B, C, D}
E(Graph1) = {(A, B), (A, D), (B, C), (B, D)}

A directed graph

A graph in which each edge is directed from one vertex to another (or the same) vertex

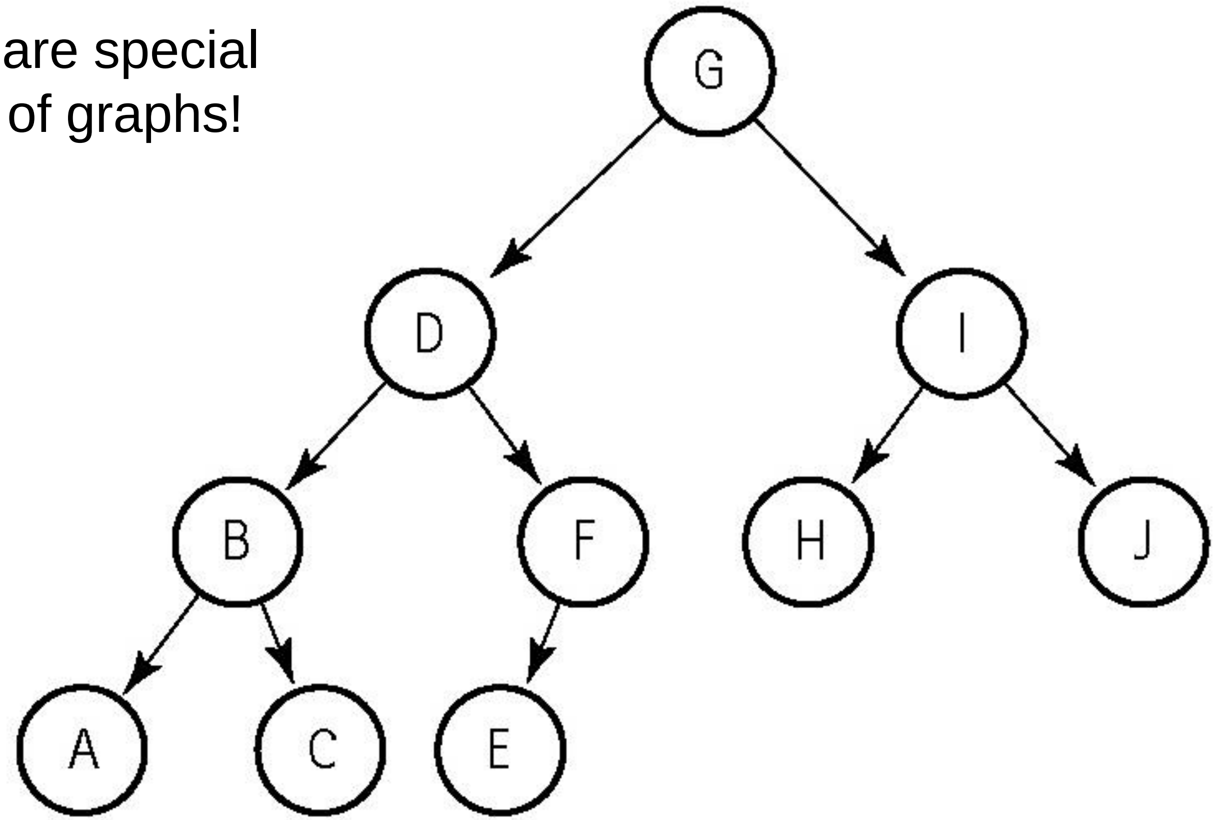


$V(Graph2) = \{1, 3, 5, 7, 9, 11\}$
$E(Graph2) = \{(1, 3), (3, 1), (5, 7), (5, 9), (9, 11), (9, 9), (11, 1)\}$

The order of vertices in E **is** important for directed graphs!!

A directed graph

Trees are special cases of graphs!



V(Graph3) = {A, B, C, D, E, F, G, H, I, J}
E(Graph3) = {(G, D), (G, J), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

Graphs as ADTs

ADT graph operations

- Test if empty
- Get number of vertices, edges in a graph
- See if edge exists between two given vertices
- Add vertex to graph whose vertices have distinct, different values from new vertex
- Add/remove edge between two given vertices
- Remove vertex, edges to other vertices
- Retrieve vertex that contains given value

Graphs as ADTs

- A C++ interface for undirected, connected graphs

```
/** An interface for the ADT undirected, connected graph.
 @file GraphInterface.h */
#ifndef GRAPH_INTERFACE_
#define GRAPH_INTERFACE_

template<class LabelType>
class GraphInterface
{
public:
   /** Gets the number of vertices in this graph.
    @return  The number of vertices in the graph. */
   virtual int getNumVertices() const = 0;

   /** Gets the number of edges in this graph.
    @return  The number of edges in the graph. */
   virtual int getNumEdges() const = 0;
```

Graphs as ADTs

```cpp

   /** Creates an undirected edge in this graph between two vertices
       that have the given labels. If such vertices do not exist, creates
       them and adds them to the graph before creating the edge.
    @param start  A label for the first vertex.
    @param end  A label for the second vertex.
    @param edgeWeight  The integer weight of the edge.
    @return  True if the edge is created, or false otherwise. */
   virtual bool add(LabelType start, LabelType end, int edgeWeight) = 0;

   /** Removes an edge from this graph. If a vertex is left with no other edges,
      it is removed from the graph since this is a connected graph.
    @param start  A label for the vertex at the beginning of the edge.
    @param end  A label for the vertex at the end of the edge.
    @return  True if the edge is removed, or false otherwise. */
   virtual bool remove(LabelType start, LabelType end) = 0;

   /** Gets the weight of an edge in this graph.
    @return  The weight of the specified edge.
       If no such edge exists, returns a negative integer. */
   virtual int getEdgeWeight(LabelType start, LabelType end) const = 0;

```

- A C++ interface for undirected, connected graphs

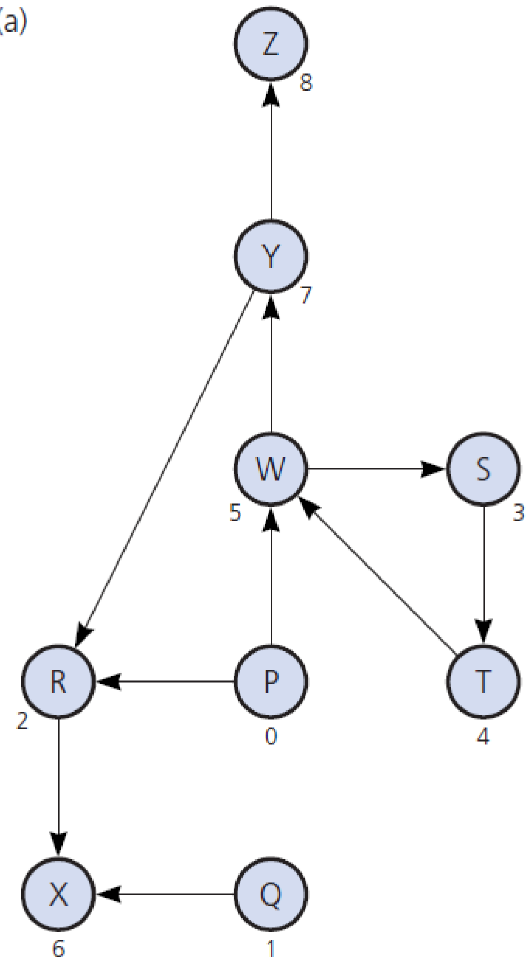
Graphs as ADTs

```
    /** Performs a depth-first search of this graph beginning at the given
        vertex and calls a given function once for each vertex visited.
     @param start  A label for the beginning vertex.
     @param visit  A client-defined function that performs an operation on
        or with each visited vertex. */
    virtual void depthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;

    /** Performs a breadth-first search of this graph beginning at the given
        vertex and calls a given function once for each vertex visited.
     @param start  A label for the beginning vertex.
     @param visit  A client-defined function that performs an operation on
        or with each visited vertex. */
    virtual void breadthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;

    /** Destroys this graph and frees its assigned memory. */
    virtual ~GraphInterface() {   }
}; // end GraphInterface
#endif
```

- A C++ interface for undirected, connected graphs

Implementing Graphs

(a)



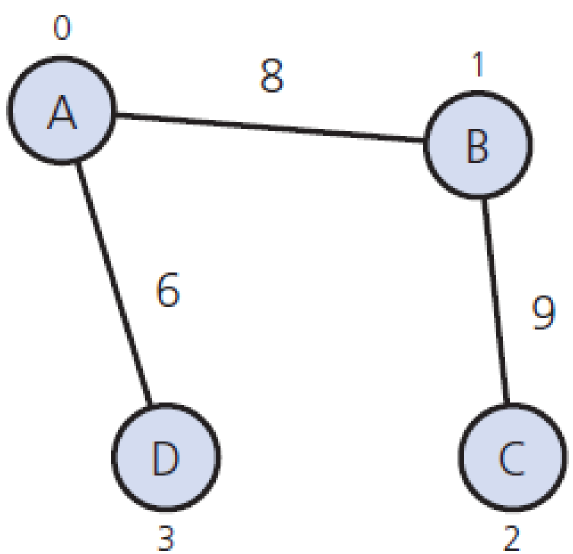
(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

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

Implementing Graphs

(a)

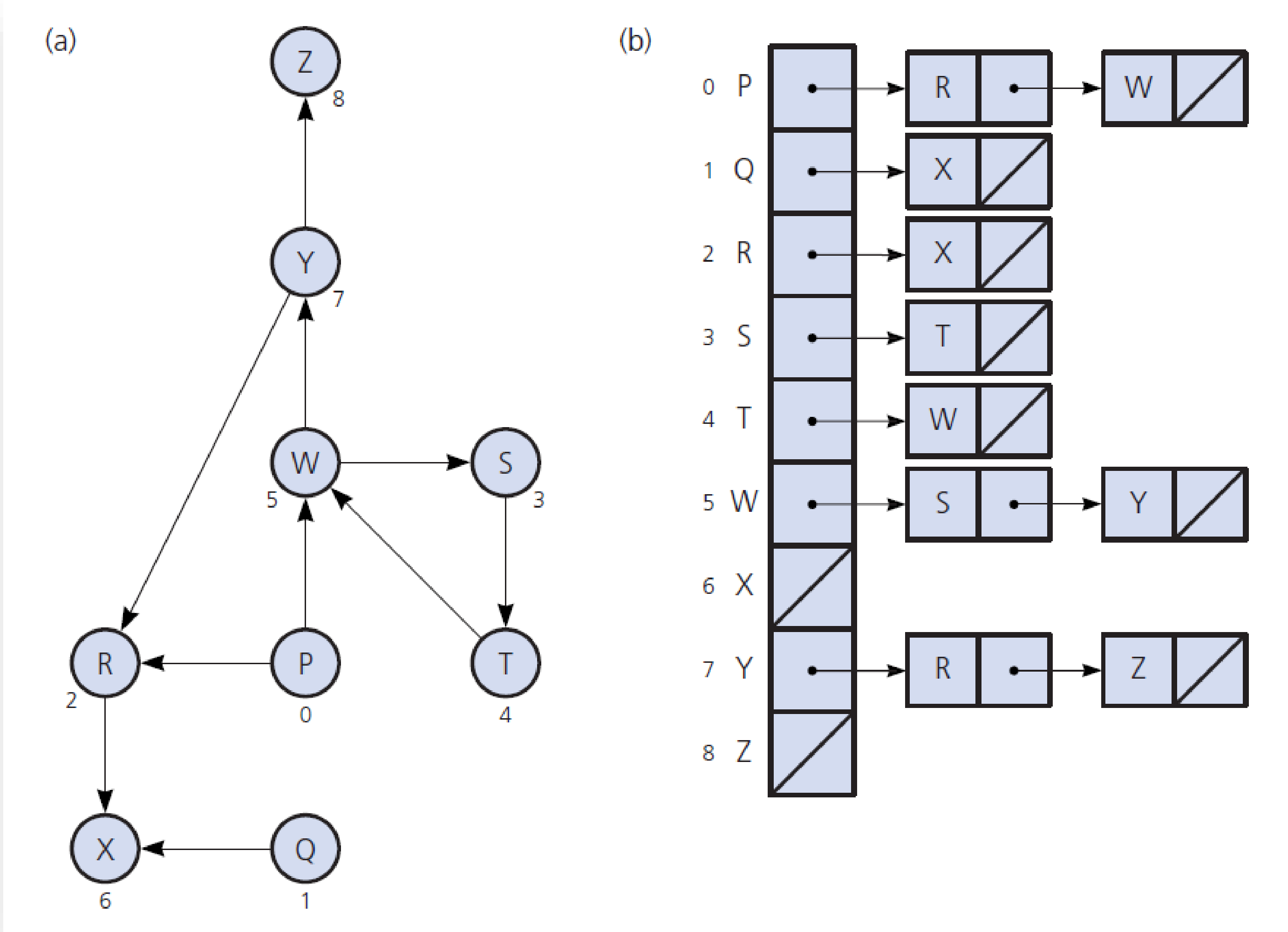


(b)

		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

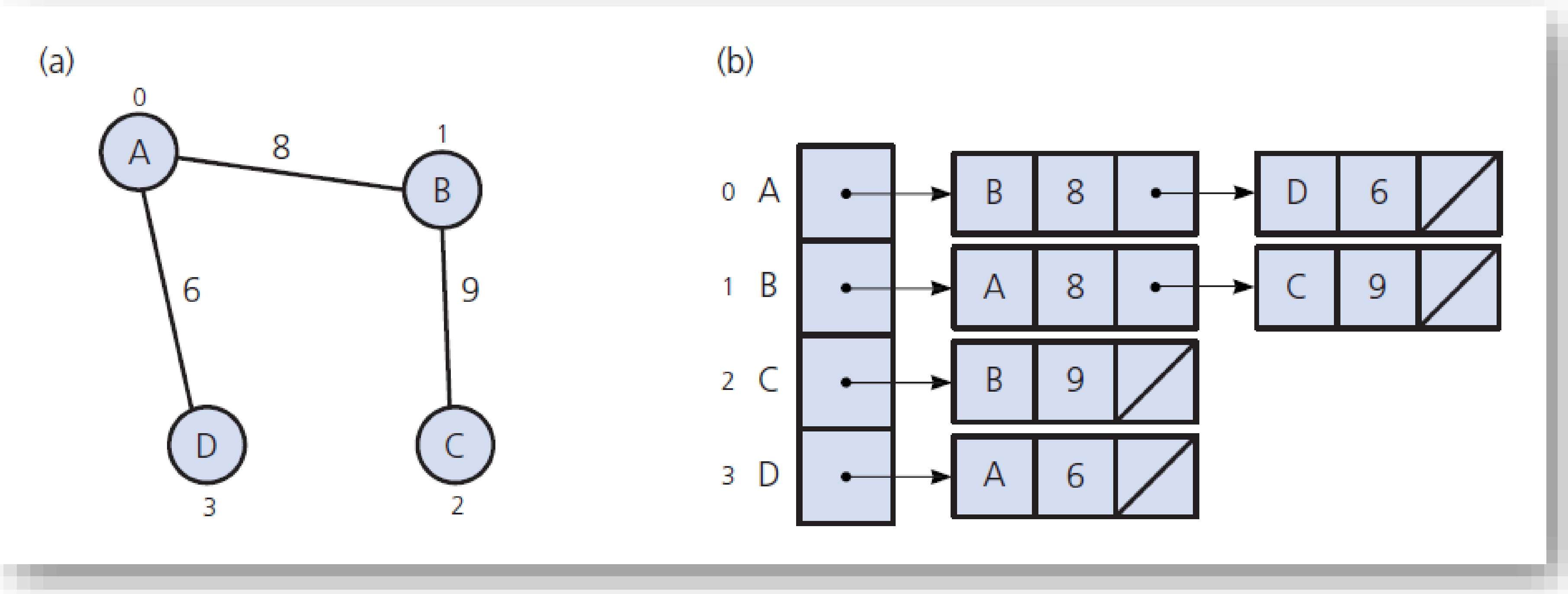
- (a) A weighted undirected graph and (b) its adjacency matrix

Implementing Graphs



- (a) A directed graph and (b) its adjacency list

Implementing Graphs



- (a) A weighted undirected graph and (b) its adjacency list

Implementing Graphs

- **Adjacency list**
  - Often requires less space than adjacency matrix
  - Supports finding vertices adjacent to given vertex
- **Adjacency matrix**
  - Supports process of seeing if there exists an edge from vertex *i* to vertex *j*

Adjacency Matrix for Flight Connections

graph
.num Vertices 7
.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

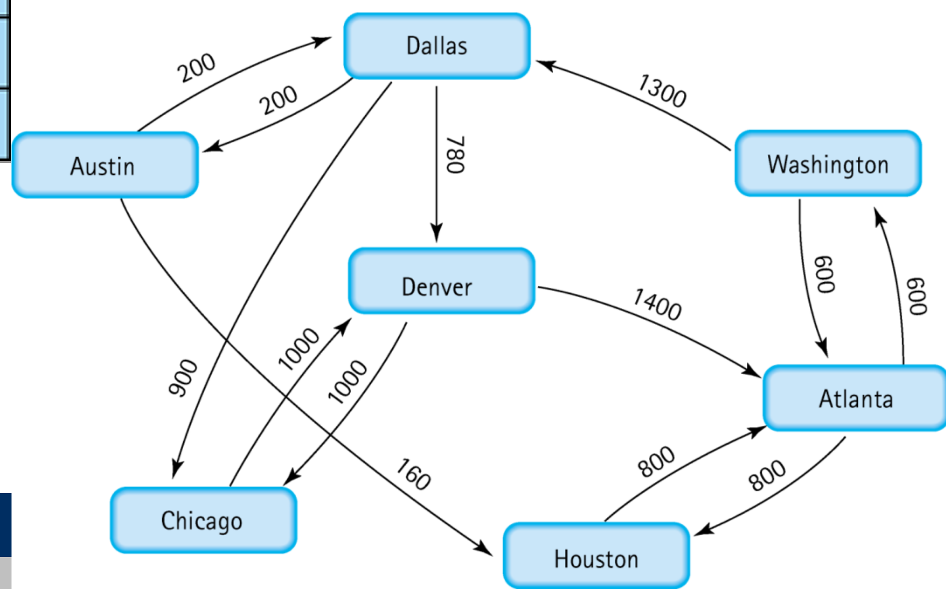
.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

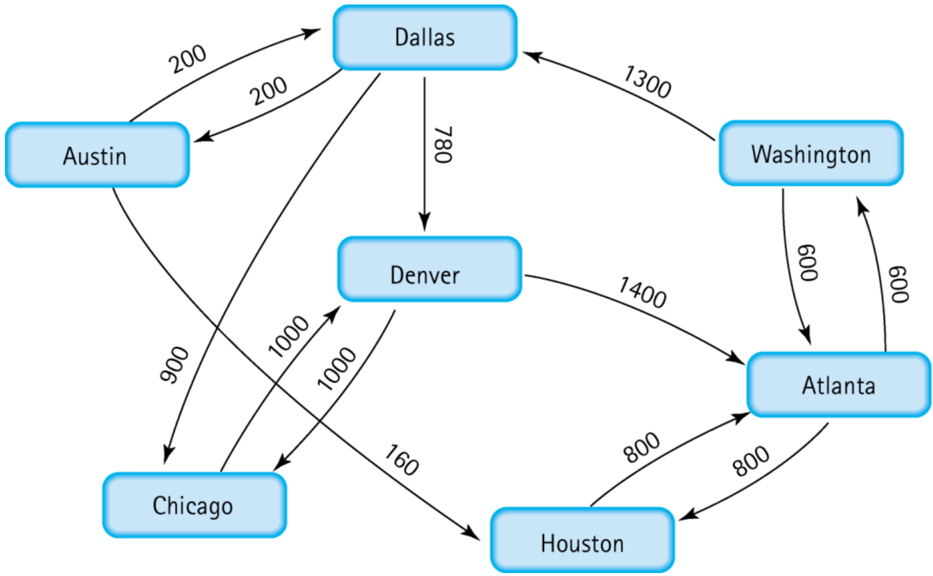
(Array positions marked '•' are undefined)

from node x ?

to node x ?

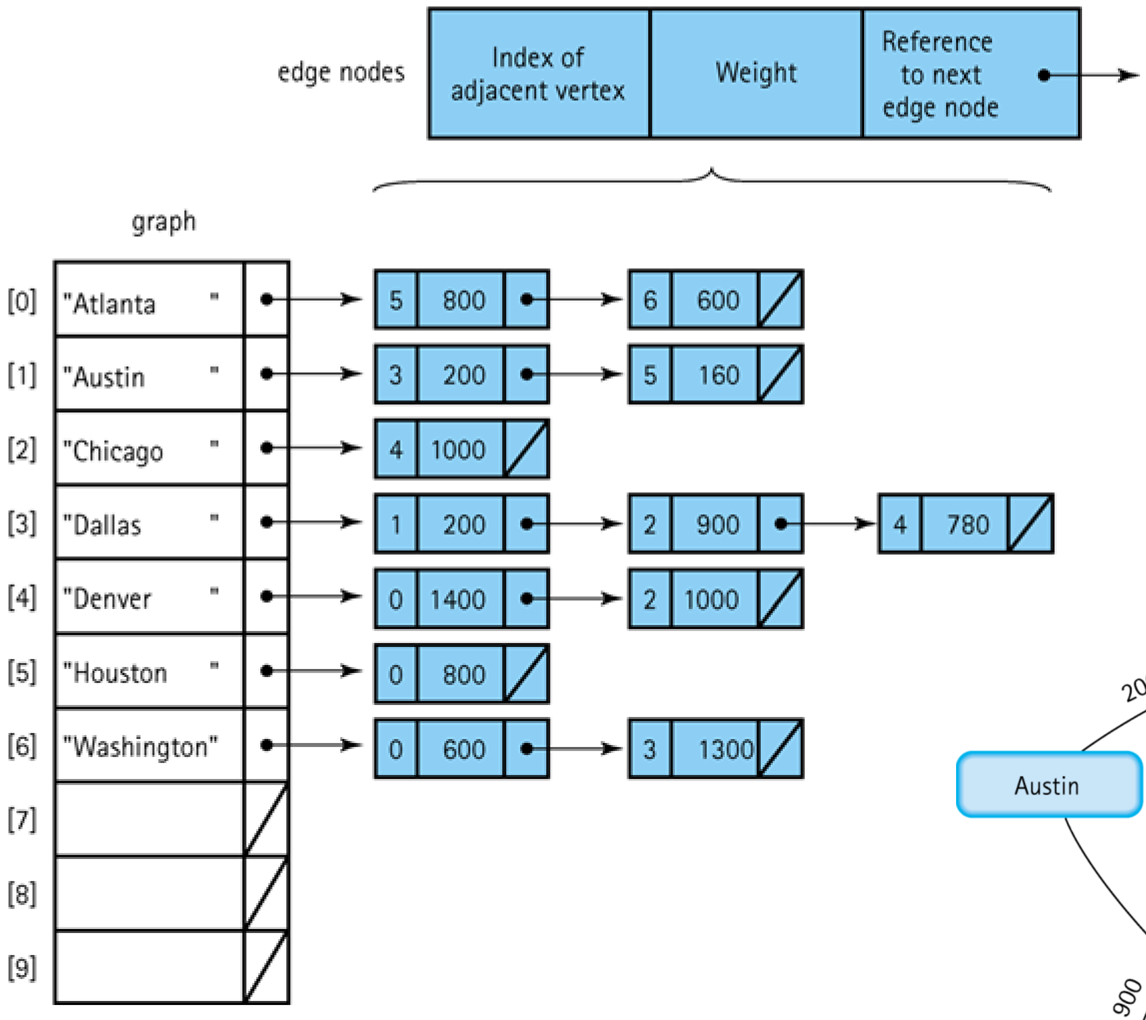


Array-Based Implementation (cont.)



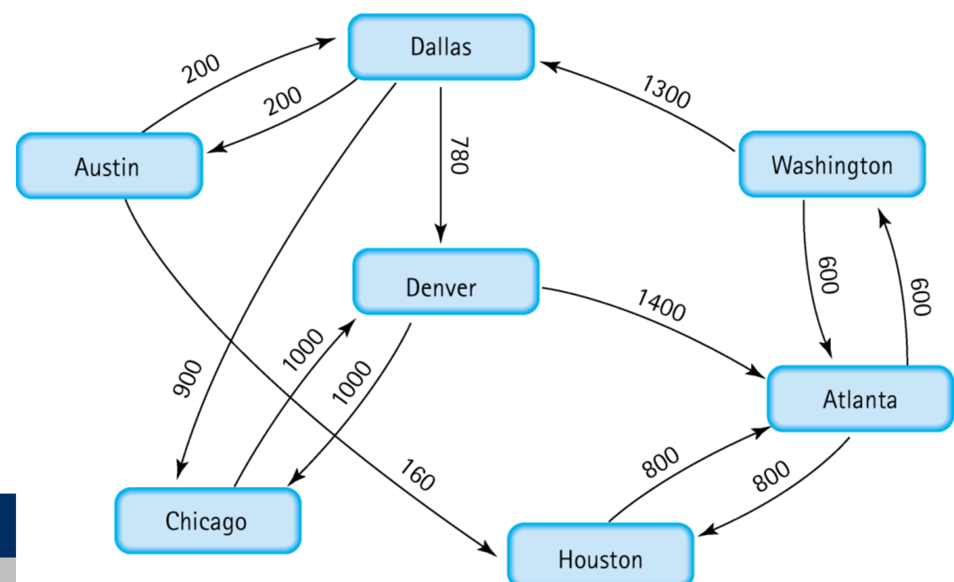
- Memory required
  - $O(V+V^2)=O(V^2)$
- Preferred when
  - The graph is **dense:** $E = O(V^2)$
- Advantage
  - Can quickly determine if there is an edge between two vertices
- Disadvantage
  - Consumes significant memory for sparse large graphs

Adjacency List Representation of Graphs



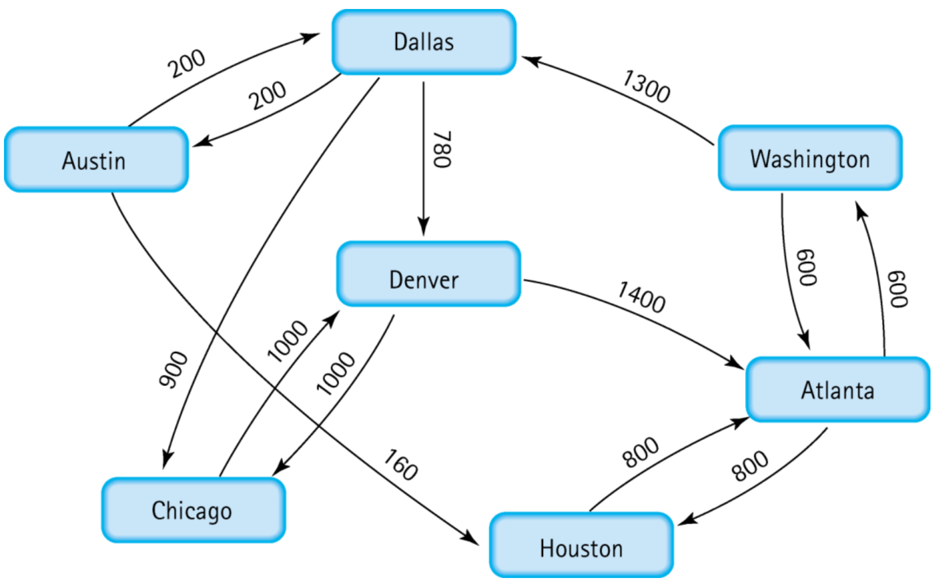
from node x ?

to node x ?



Link-List-based Implementation (cont.)

- Memory required
  - $O(V + E)$
    - $O(V)$ for sparse graphs since $E=O(V)$
    - $O(V^2)$ for dense graphs since $E=O(V^2)$
- Preferred when
  - for **sparse** graphs: $E = O(V)$
- Disadvantage
  - No quick way to determine the vertices adjacent to a given vertex



- Advantage
  - Can quickly determine the vertices adjacent **from** a given vertex

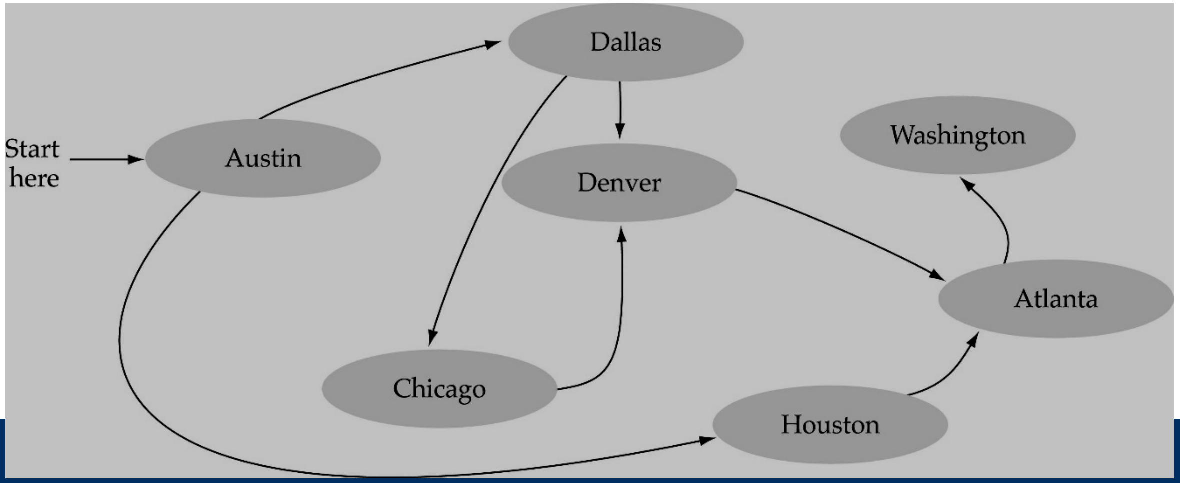
Graph Traversals

Graph Traversals

- Visits all vertices it can reach
- Visits all vertices if and only if the graph is connected
- Connected component
  - Subset of vertices visited during a traversal that begins at a given vertex

Graph searching

- *Problem*: find if there is a path between two vertices of the graph
  - e.g., Austin and Washington
- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)



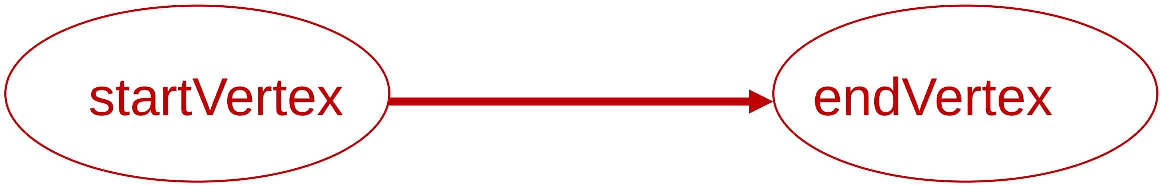
Depth-First Search

- DFS traversal
  - Goes as far as possible from a vertex before backing up
- Recursive transversal algorithm

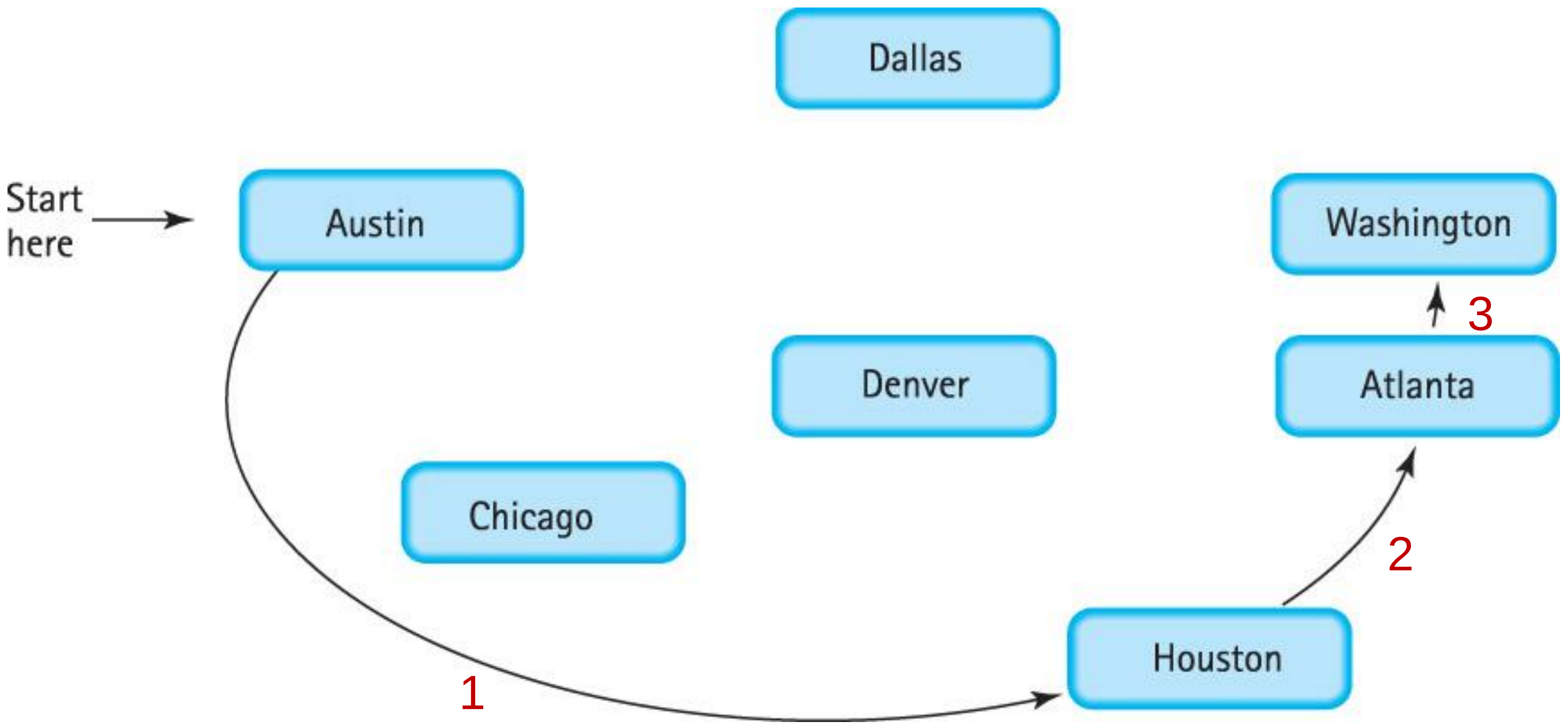
```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.
dfs(v: Vertex)
 { Mark v as visited
   for (each unvisited vertex u adjacent to v)
      dfs(u)
  }
```

Depth-First-Search (DFS)

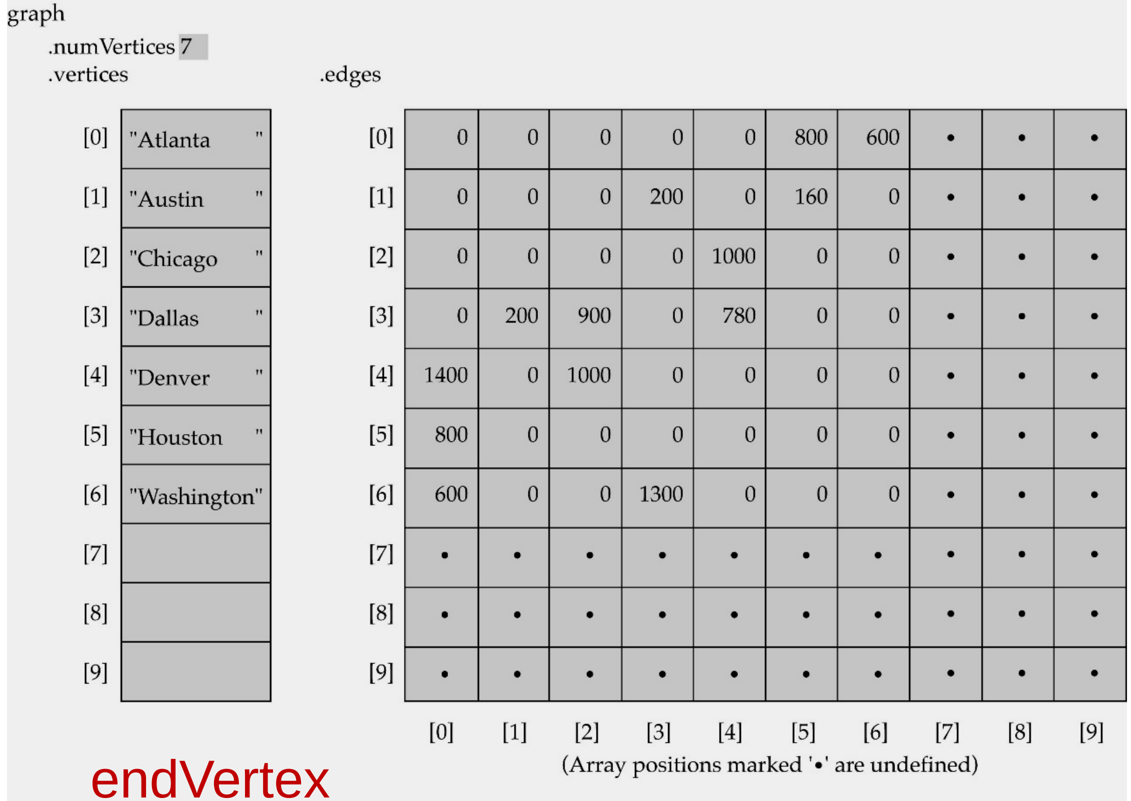
- Main idea:
  - Travel as far as you can down a path
  - Back up *<u>as little as possible</u>* when you reach a "dead end"
    - i.e., next vertex has been "marked" or there is no next vertex



Depth First Search: Follow Down



DFS uses Stack !



graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

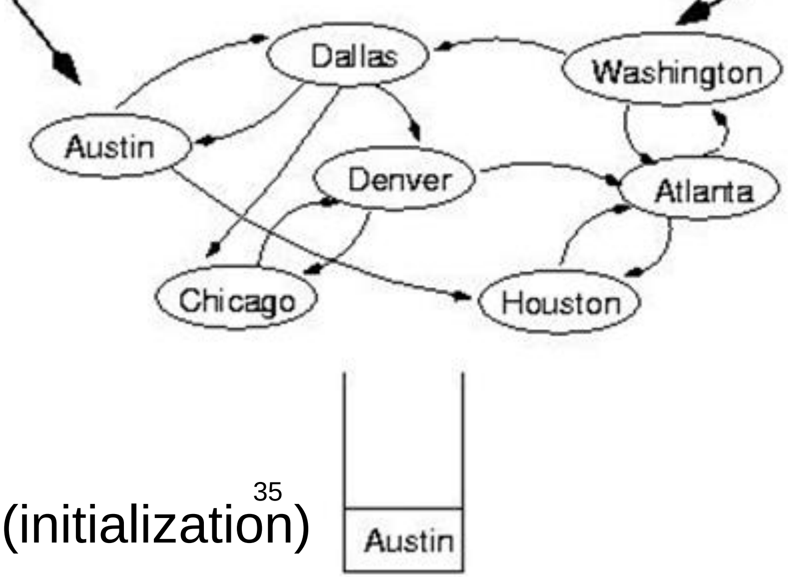
.edges

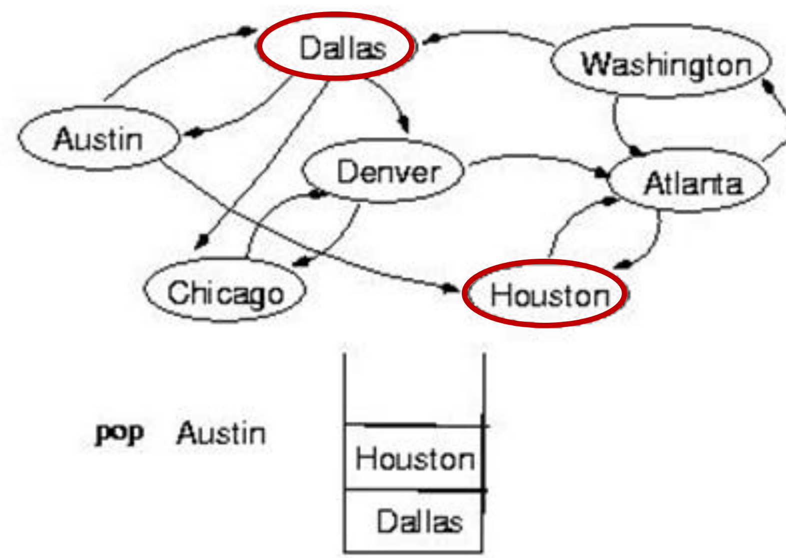
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

(Array positions marked '•' are undefined)

endVertex

startVertex





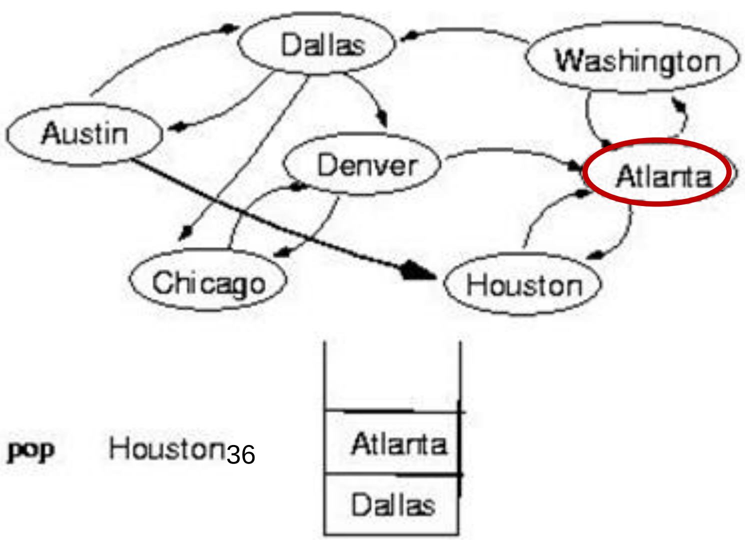
(initialization)

graph
.numVertices 7
.vertices .edges

	.vertices
[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

(Array positions marked '•' are undefined)



graph

.numVertices 7

.vertices

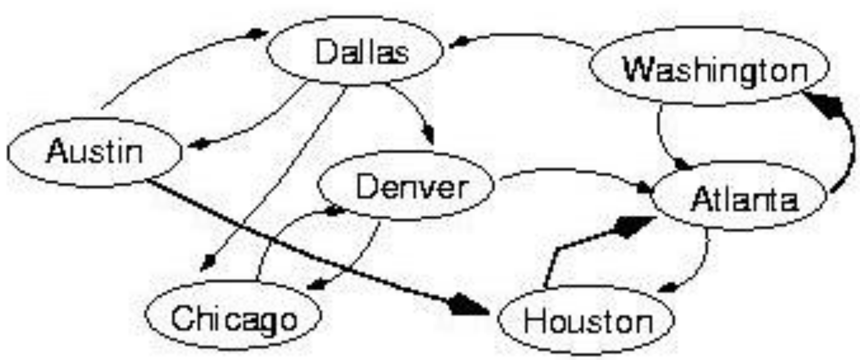
[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

(Array positions marked '•' are undefined)

endVertex



pop Washington

Depth-First Search

- Iterative algorithm, using a stack

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Iterative version.
dfs(v: Vertex)

   s= a new empty stack

   // Push v onto the stack and mark it
   s.push(v)
   Mark v as visited

   // Loop invariant: there is a path from vertex v at the
   // bottom of the stack s to the vertex at the top of s
   while (!s.isEmpty())
```

Depth-First Search

- Iterative algorithm, using a stack, ctd.

```
{
   if (no unvisited vertices are adjacent to the vertex on the top of the stack)
      s.pop()   // Backtrack

   else
   {
      Select an unvisited vertex u adjacent to the vertex on the top of the stack
      s.push(u)
      Mark u as visited
   }
}
```

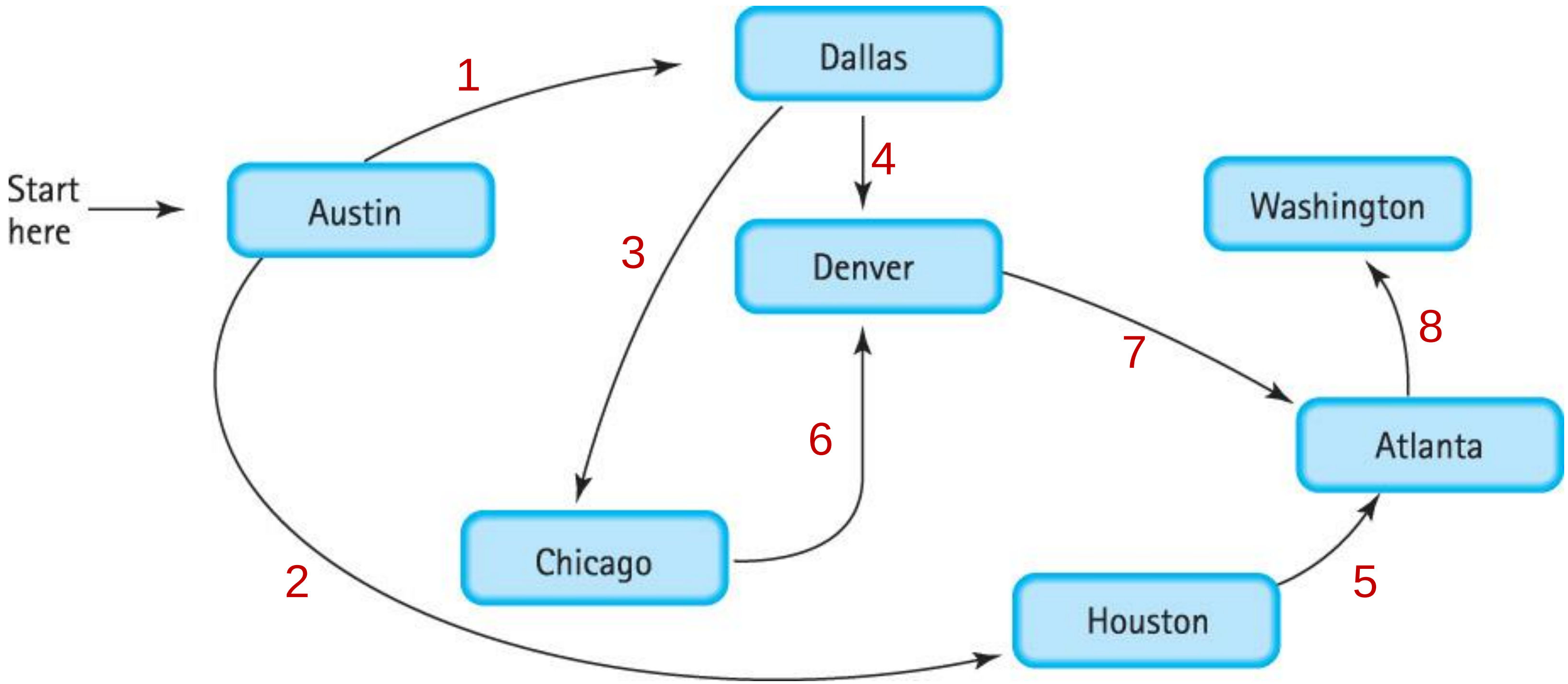
Breadth-First Search

- BFS traversal
  - Visits all vertices adjacent to a vertex before going forward
- BFS is a first visited, first explored strategy
  - Contrast DFS as last visited, first explored

Breadth-First-Searching (BFS)

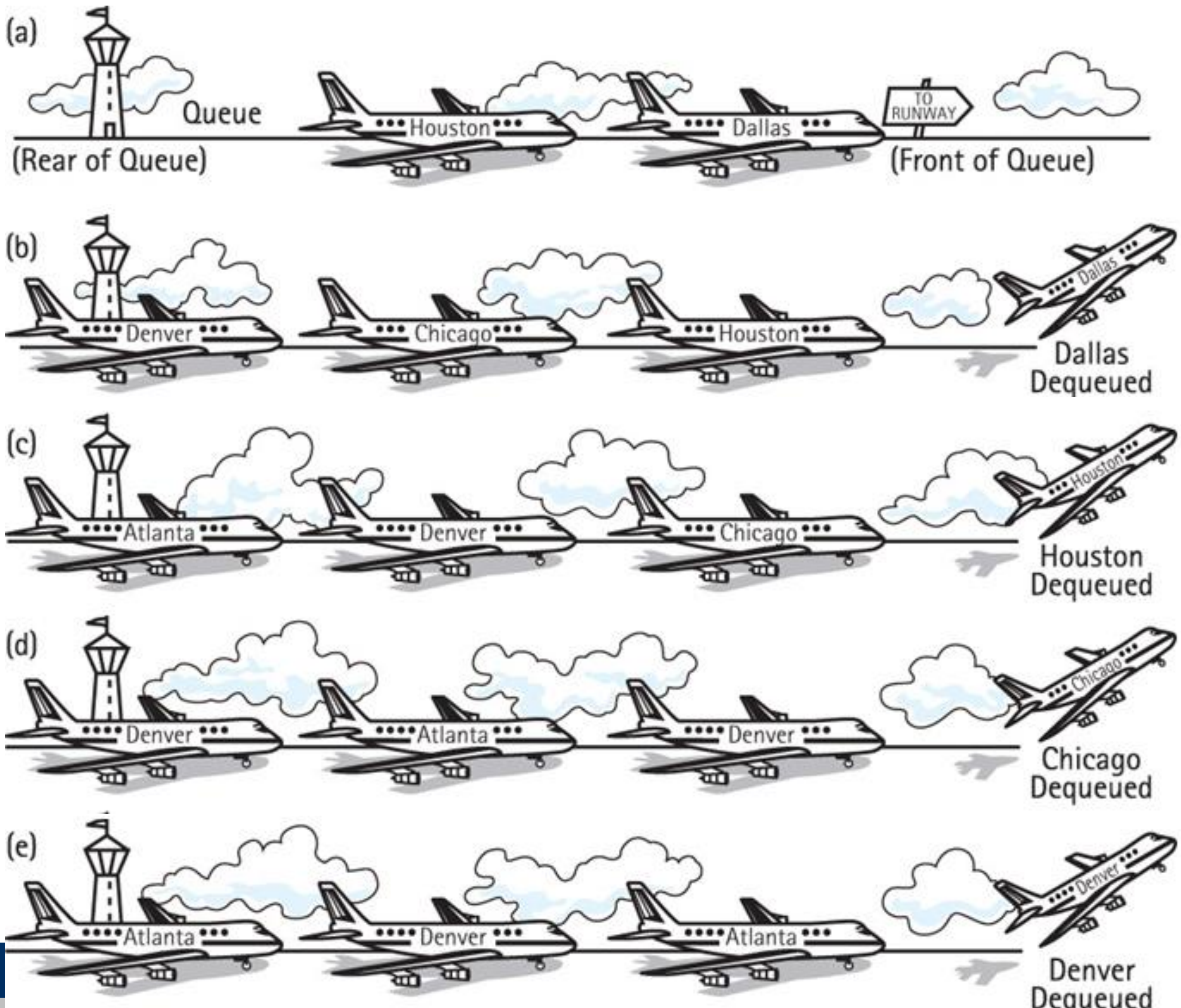
- Main idea:
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end“
    - i.e., next vertex has been "marked" or there is no next vertex

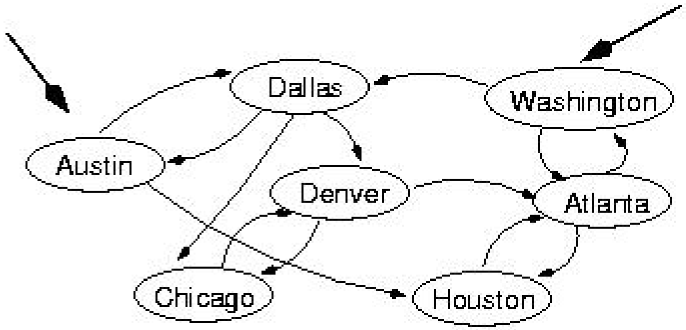
Breadth First: Follow Across



BFS uses Queue !

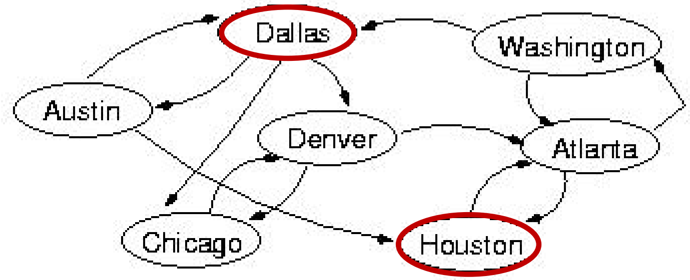
Breadth First Uses Queue





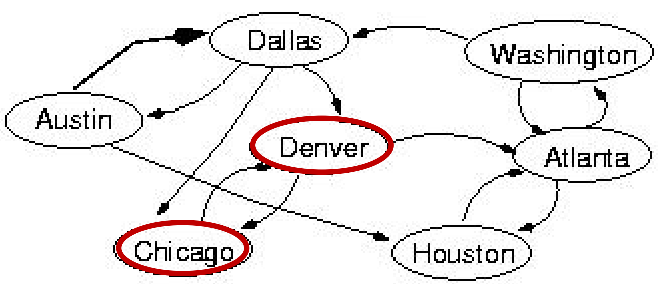
			Austin

(initialization)



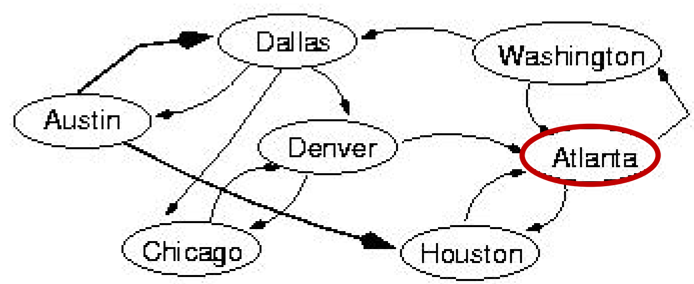
dequeue Austin

		Dallas	Houston



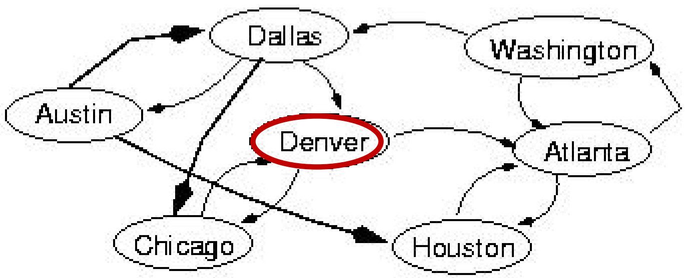
dequeue Dallas

	Houston	Chicago	Denver



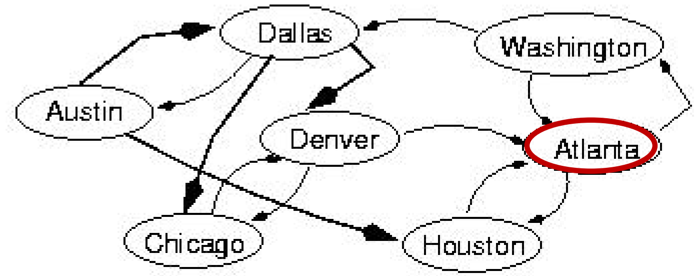
dequeue Houston

	Chicago	Denver	Atlanta



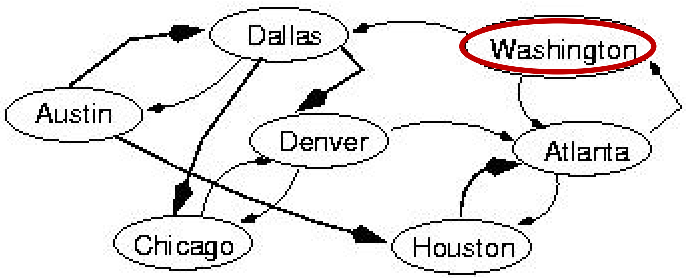
dequeue Chicago

	Denver	Atlanta	Denver



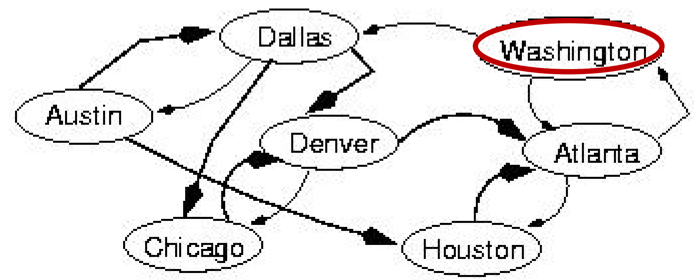
dequeue Denver

	Atlanta	Denver	Atlanta



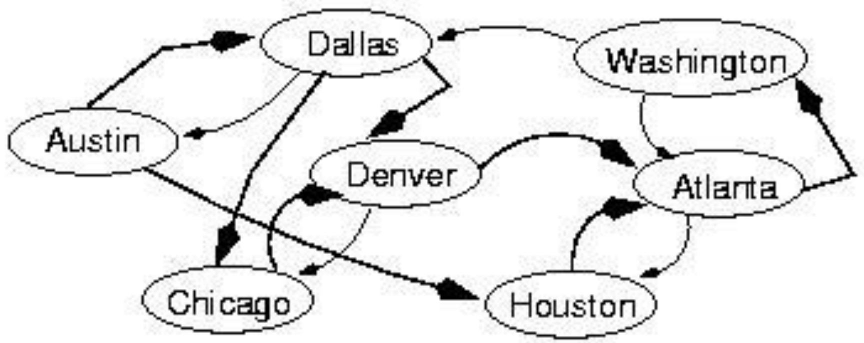
dequeue Atlanta

	Denver	Atlanta	Washington

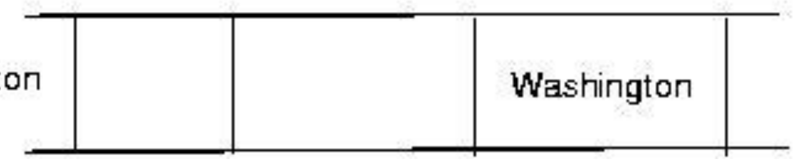


dequeue Denver, Atlanta

	Washington	Washington



dequeue Washington



Breadth-First Search

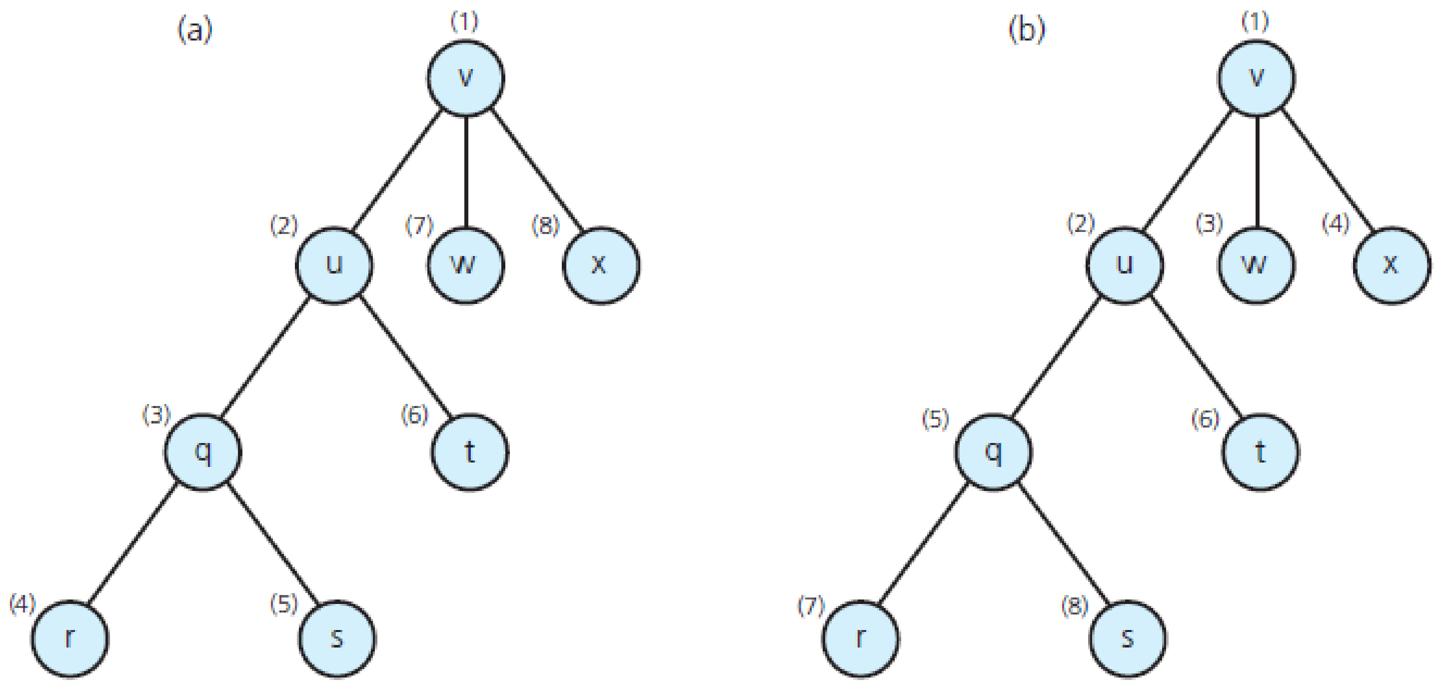
- Visits all vertices adjacent to vertex before going forward
- Breadth-first search uses a queue

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.
bfs(v: Vertex)

   q = a new empty queue

   // Add v to queue and mark it
   q.enqueue(v)
   Mark v as visited

   while (!q.isEmpty())
```

```
while (!q.isEmpty())
{
   q.dequeue(w)

   // Loop invariant: there is a path from vertex w to every vertex in the
   for (each unvisited vertex u adjacent to w)
   {
      Mark u as visited
      q.enqueue(u)
   }
}
```

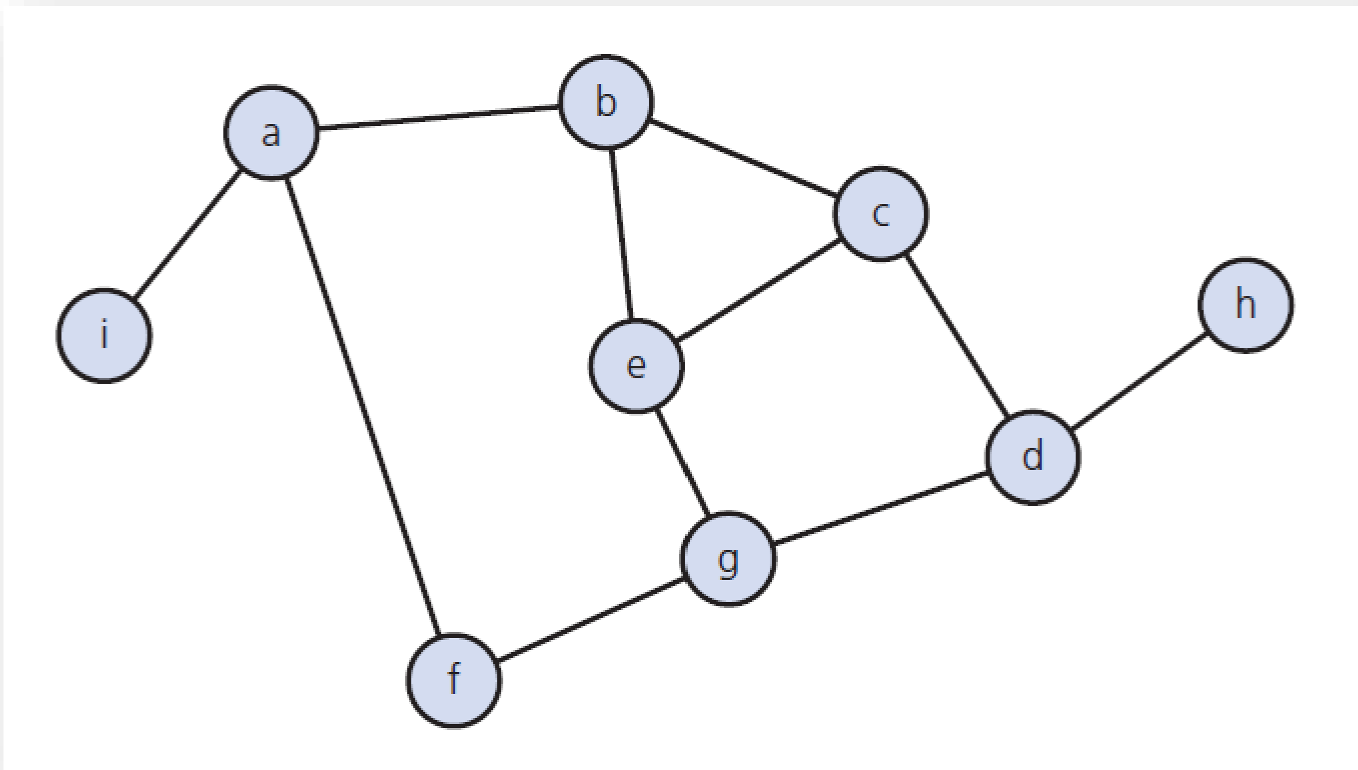
Depth-First Search vs Breadth-First Search

- Visitation order for (a) a depth-first search; (b) a breadth-first search



Depth-First Search vs Breadth-First Search

- A connected graph with cycles



Depth-First Search

- The results of a depth-first traversal, beginning at vertex a, of the graph

Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Breadth-First Search

- The results of a breadth-first traversal, beginning at vertex a, of the graph

Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

Applications of Graphs

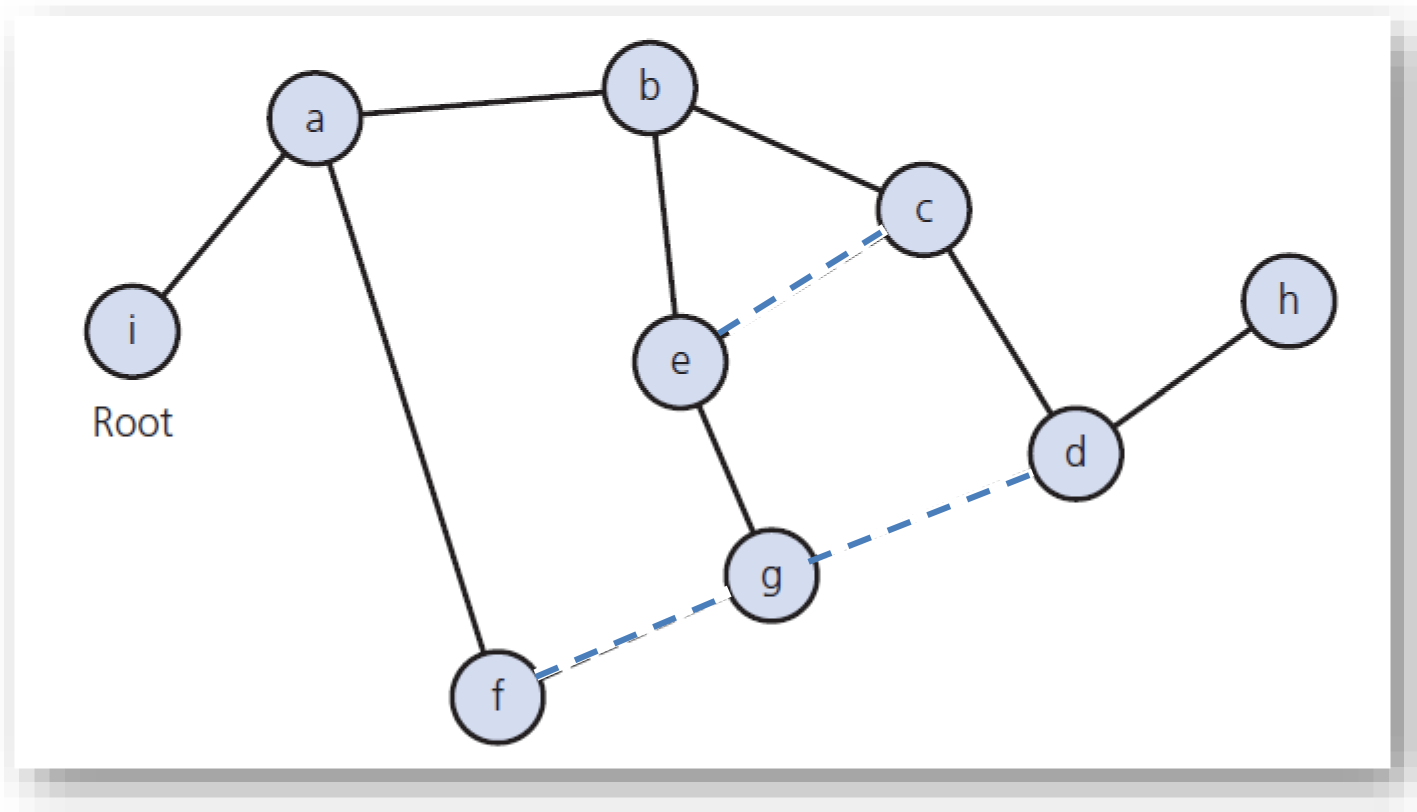
- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits
- Some Difficult Problems

Spanning Trees

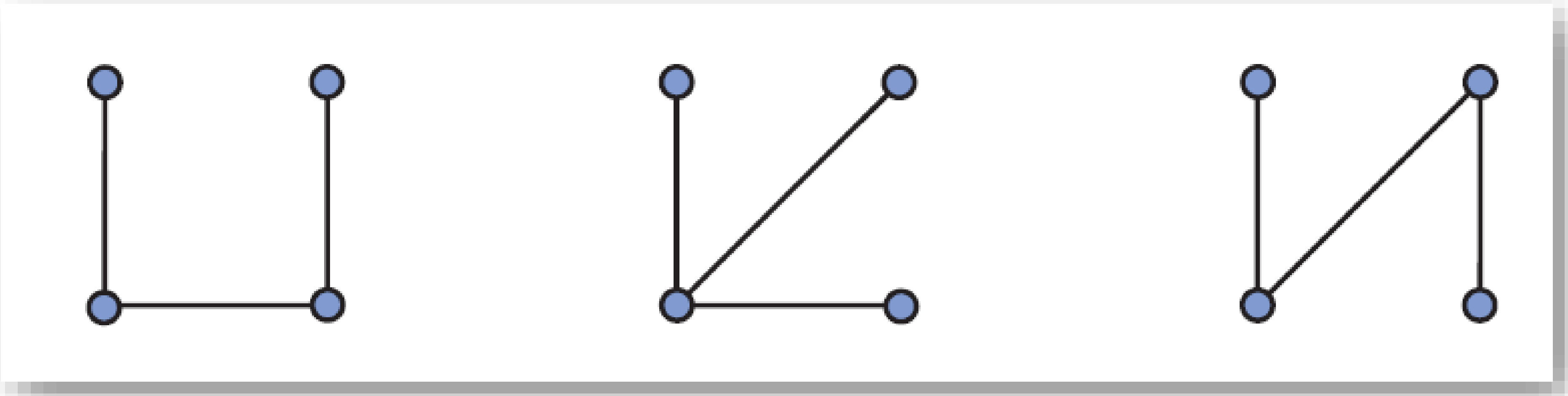
- A tree is an undirected connected graph without cycles
- Detecting a cycle in an undirected graph
  - Connected undirected graph with n vertices must have at least $n - 1$ edges
  - If it has exactly $n - 1$ edges, it cannot contain a cycle
  - With more than $n - 1$ edges, must contain at least one cycle

Spanning Trees

- A spanning tree for the graph

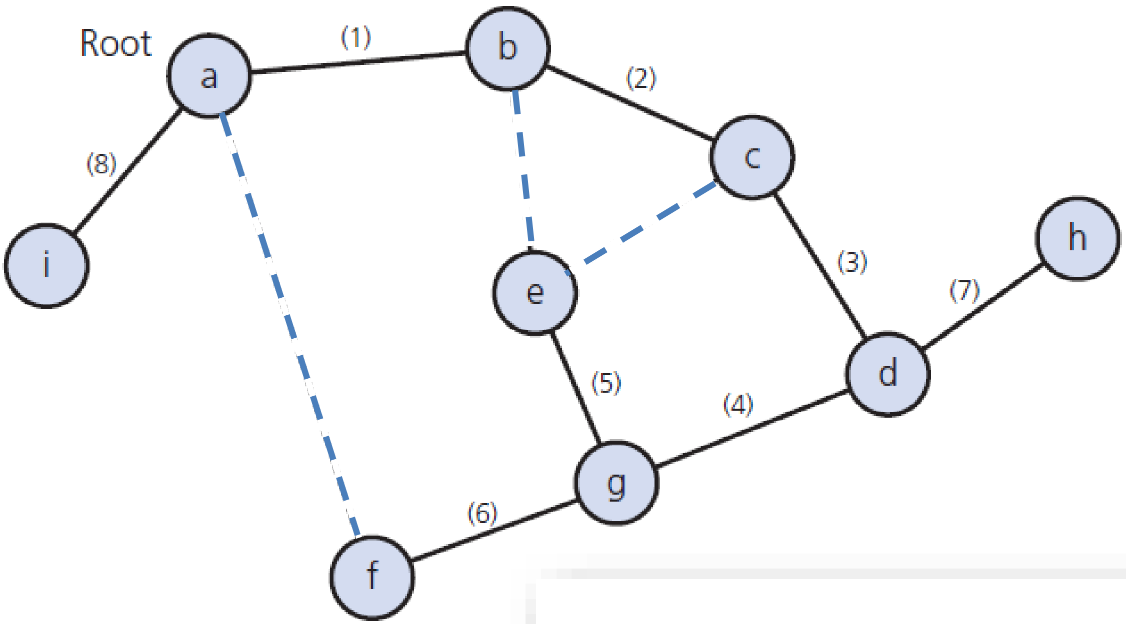


Spanning Trees



- Connected graphs that each have four vertices and three edges

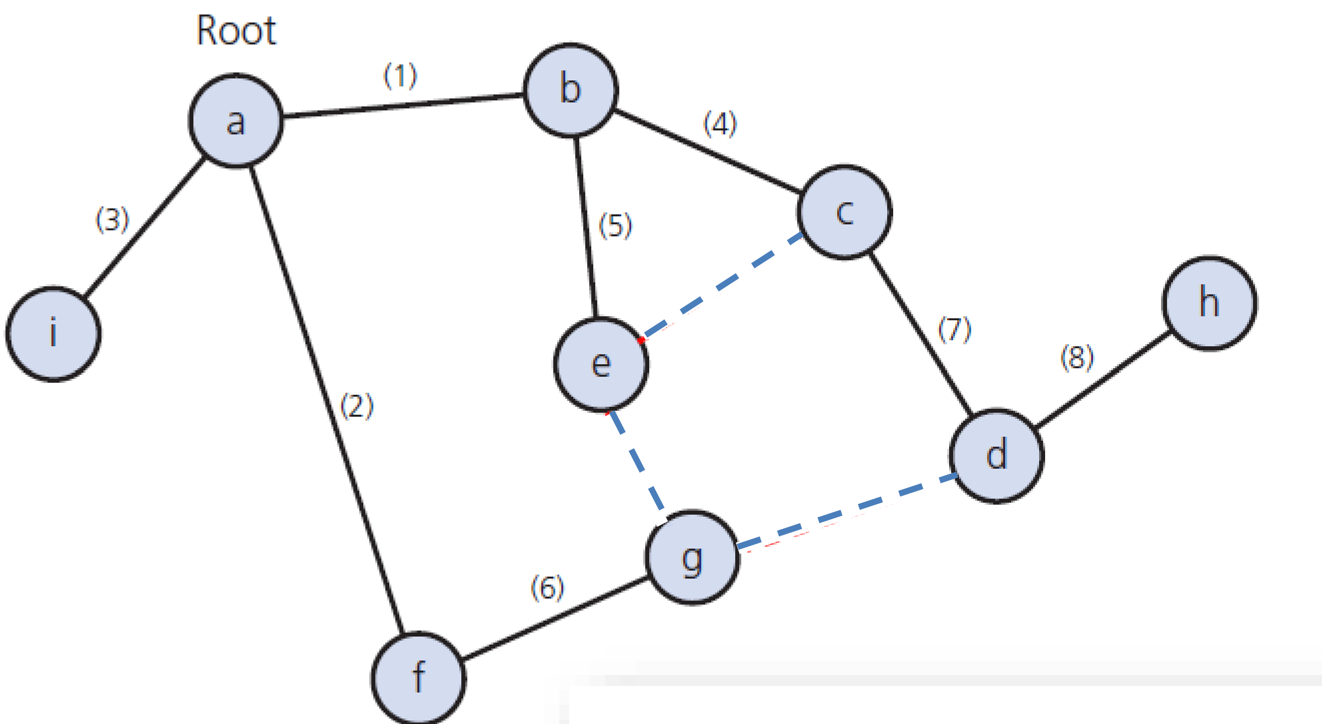
Spanning Trees



The DFS spanning tree algorithm visits vertices in this order: a, b, c, d, g, e, f, h, i. Numbers indicate the order in which the algorithm marks edges.

- Connected graphs that each have four vertices and three edges

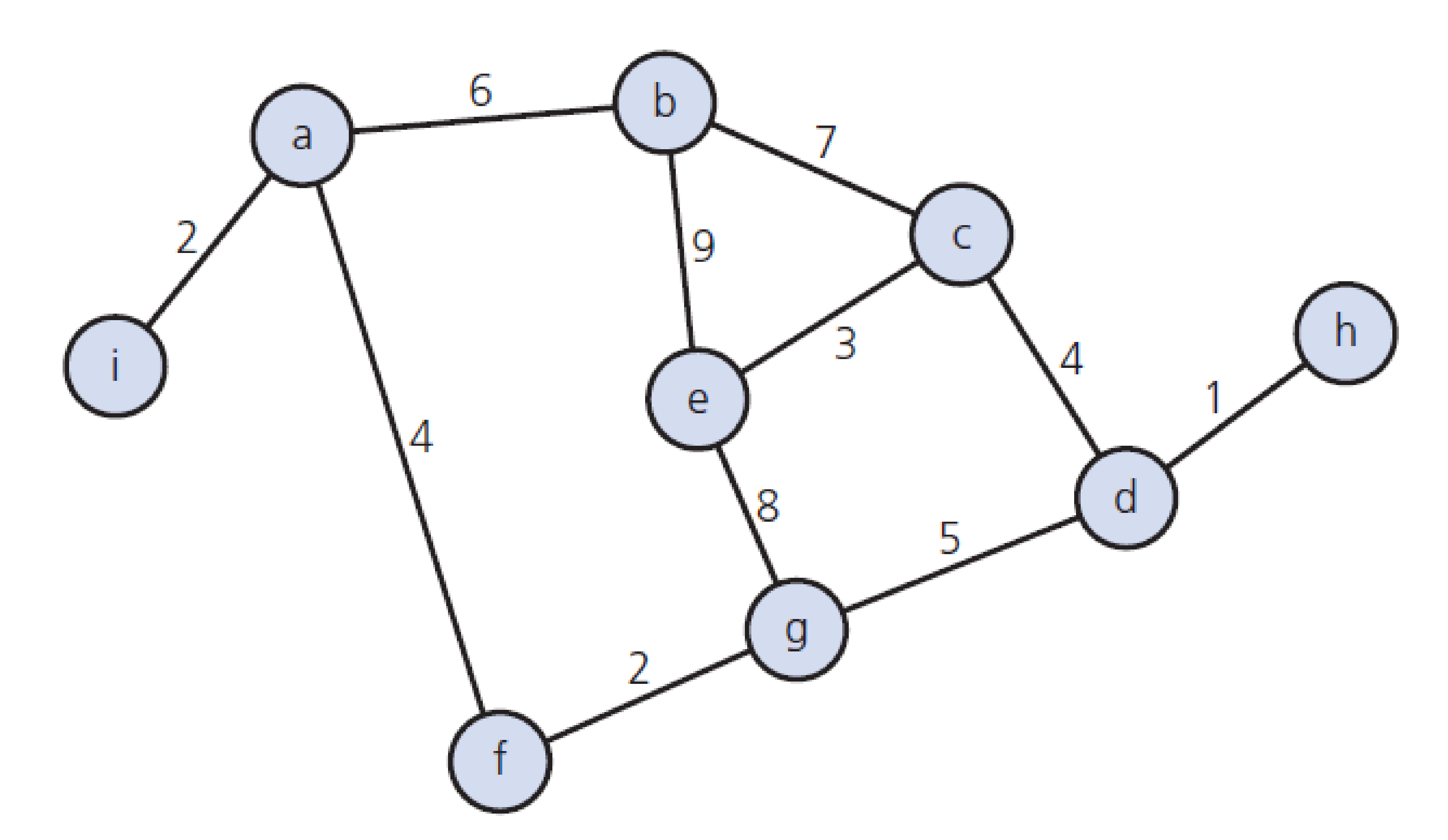
Spanning Trees



The BFS spanning tree algorithm visits vertices in this order: a, b, f, i, c, e, g, d, h. Numbers indicate the order in which the algorithm marks edges.

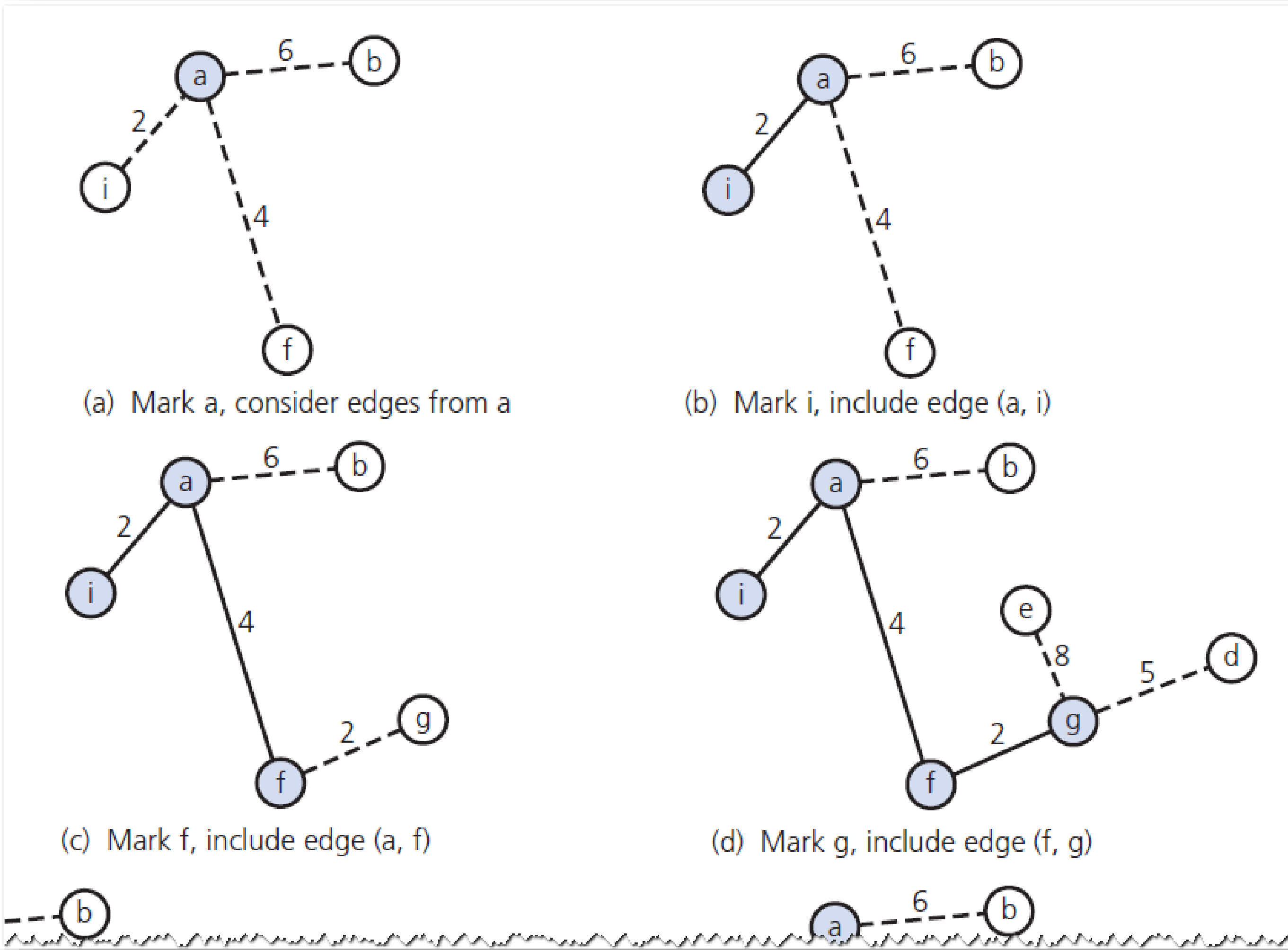
- The BFS spanning tree rooted at vertex *a* for the graph

Minimum Spanning Trees



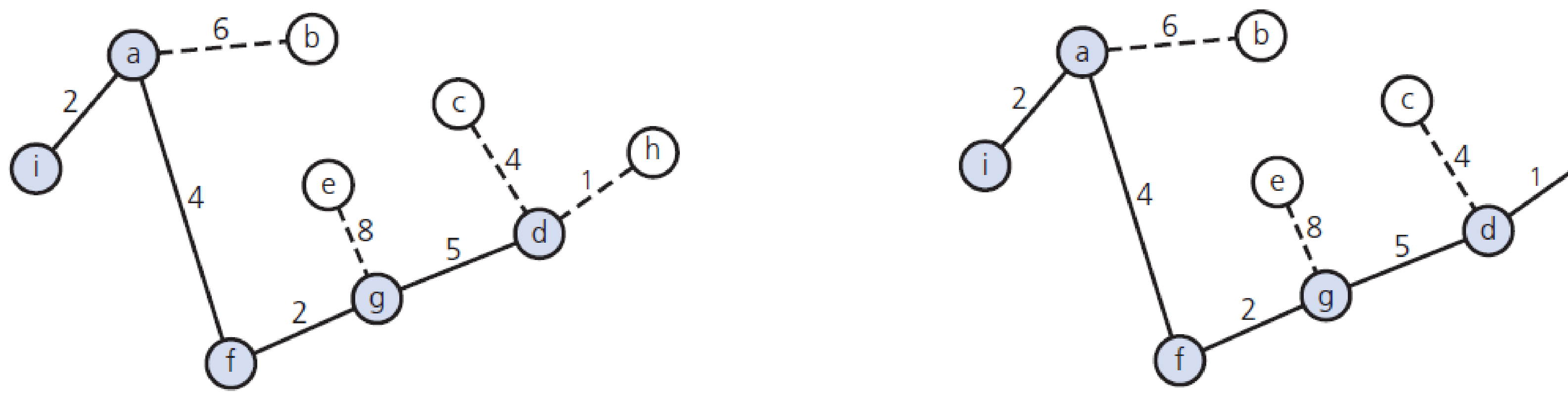
- A weighted, connected, undirected graph

Minimum Spanning Trees



- A trace of primsAlgorithm for the graph beginning at vertex a

Minimum Spanning Trees

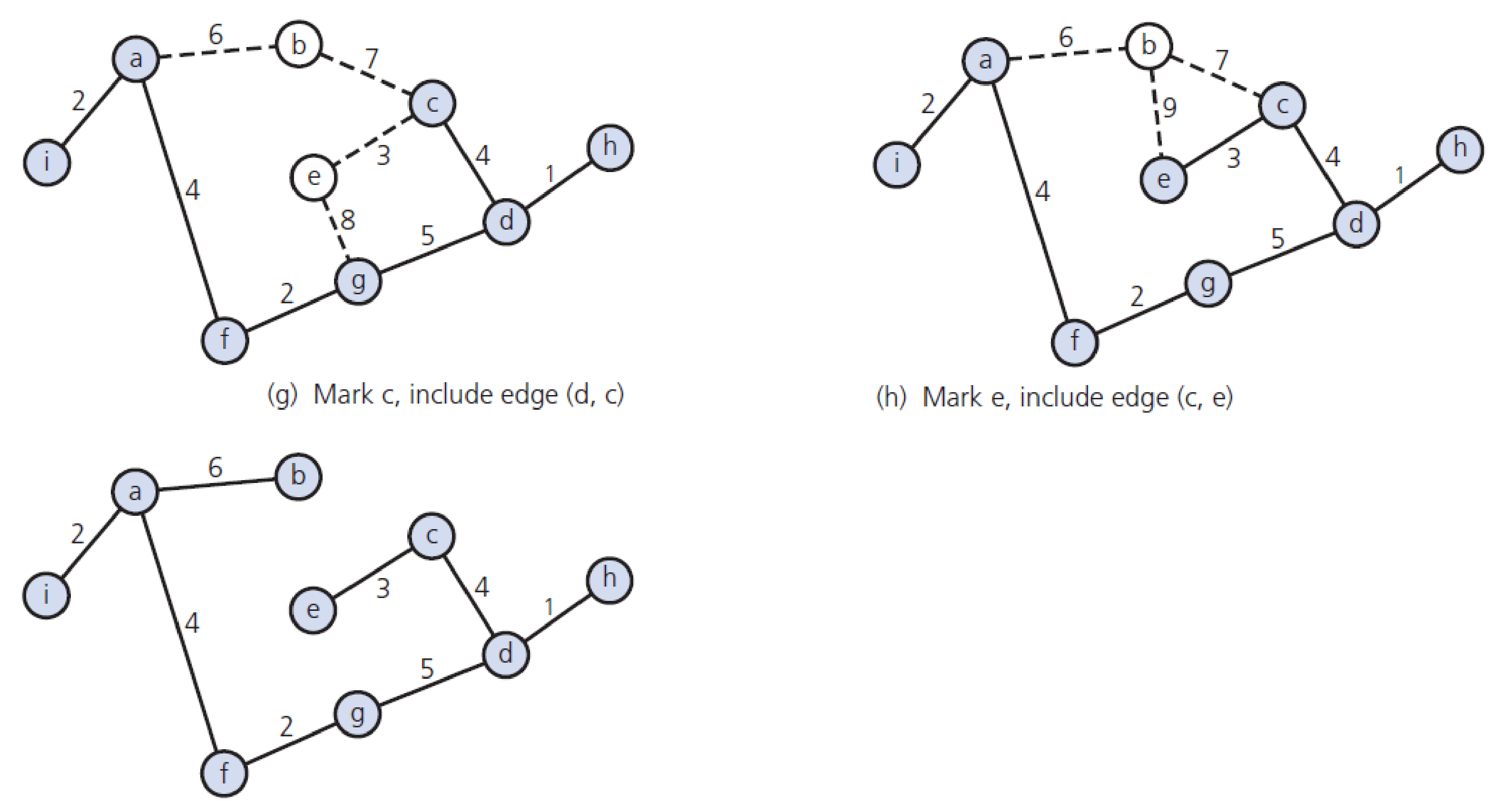


(e) Mark d, include edge (g, d)

(f) Mark h, include edge (d, h)

- A trace of primsAlgorithm for the graph beginning at vertex a

Minimum Spanning Trees



(e) Mark d, include edge (g, d)

(f) Mark h, include edge (d, h)

(g) Mark c, include edge (d, c)

(h) Mark e, include edge (c, e)

(i) Mark b, include edge (a, b)

- A trace of primsAlgorithm for the graph beginning at vertex a