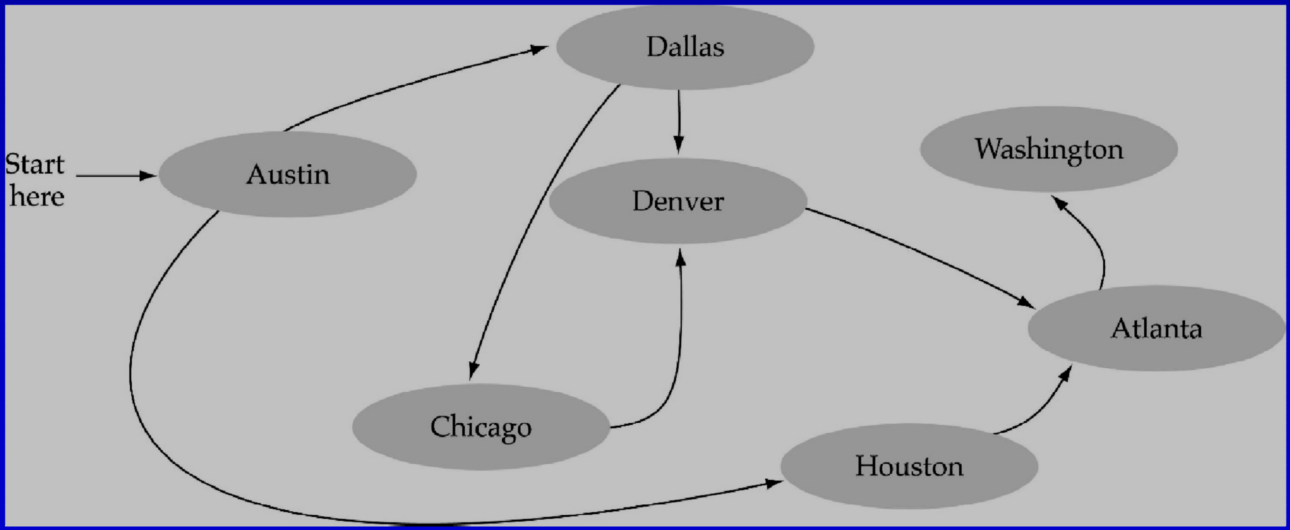
DATA STRUCTURES USING 'C'

Graphs

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph *G* is defined as follows:

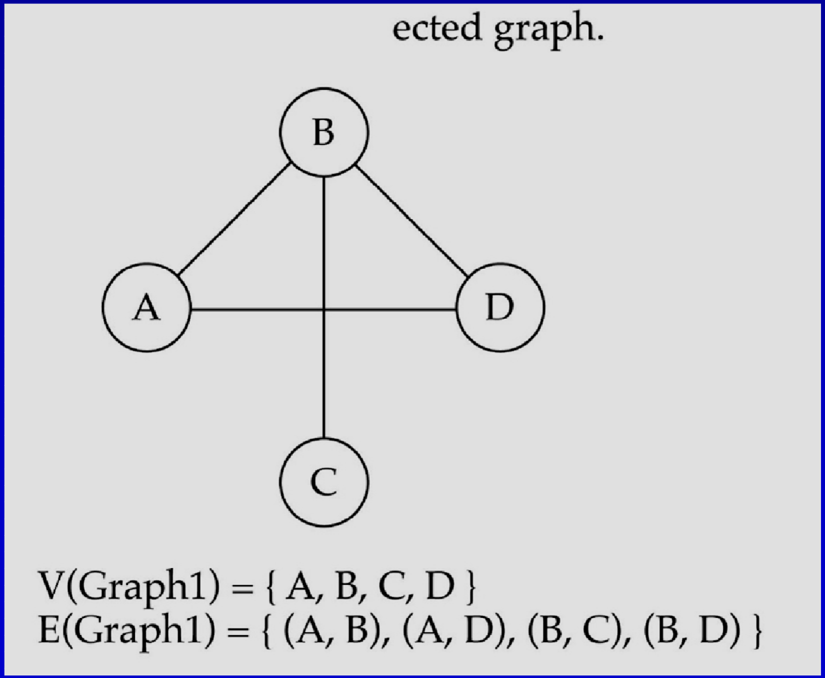
$$G=(V,E)$$

V(G): a finite, nonempty set of vertices

E(G): a set of edges (pairs of vertices)

Directed vs. undirected graphs

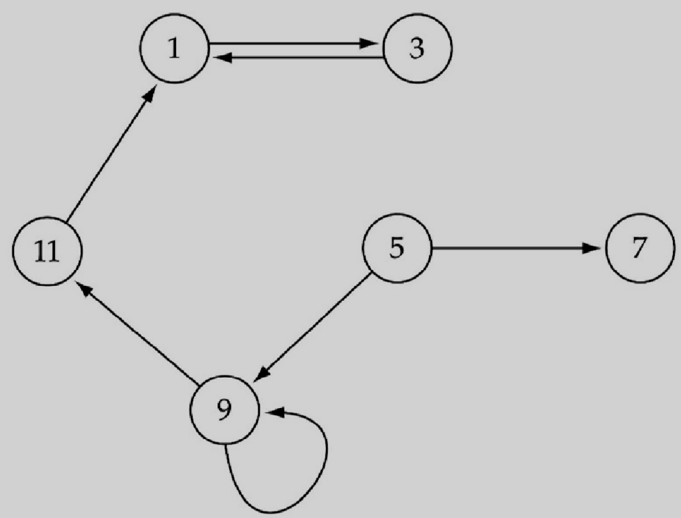
- When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.

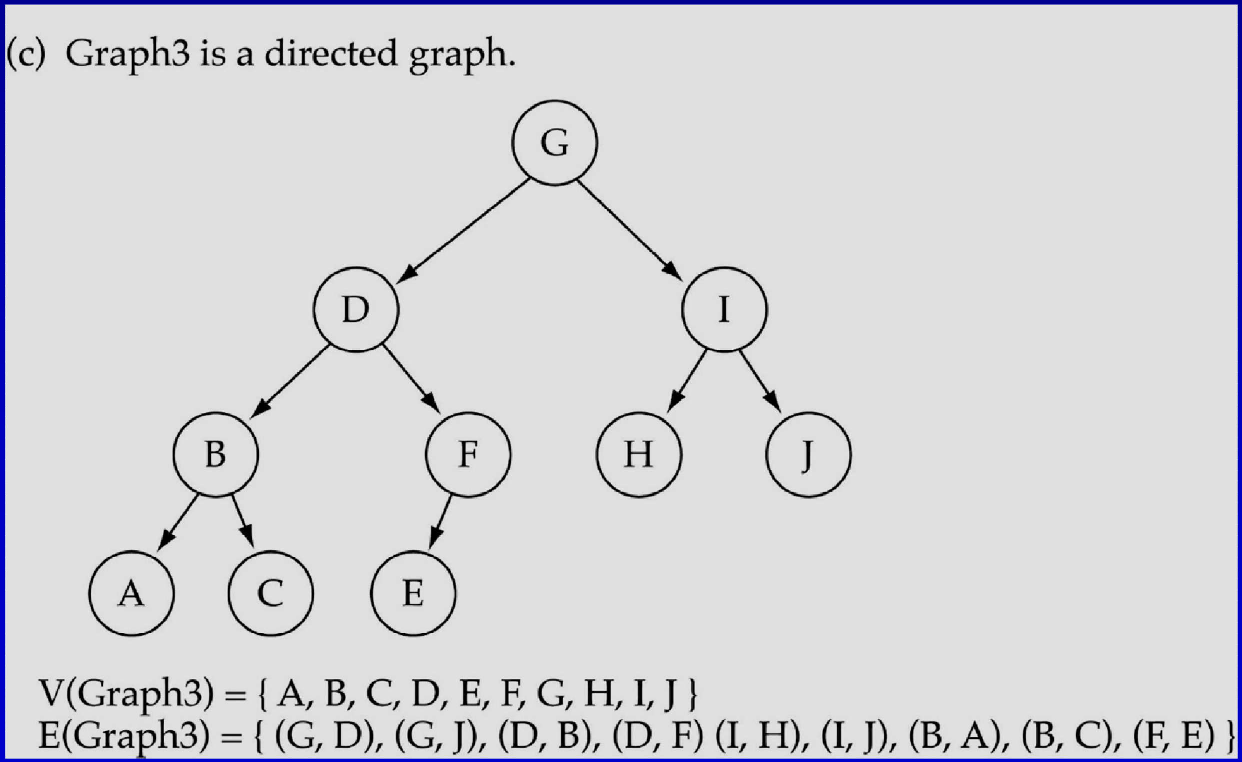


V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7), (9, 9), (11, 1) }

Warning: if the graph is directed, the order of the vertices in each edge is important !!

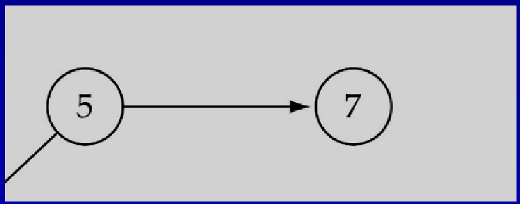
Trees vs graphs

- Trees are special cases of graphs!!



Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



5 is adjacent to 7
7 is adjacent from 5

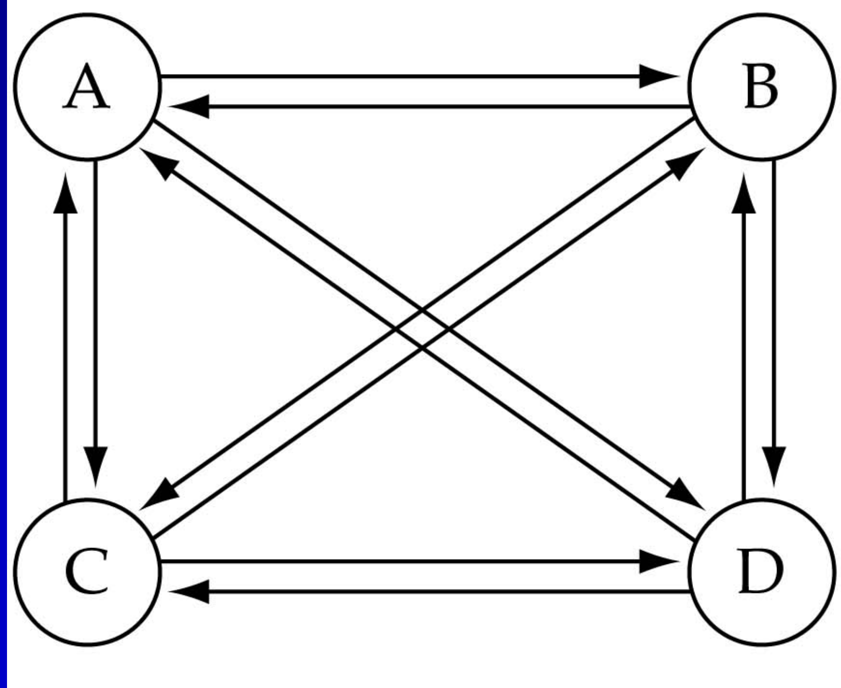
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$O(N^2)$



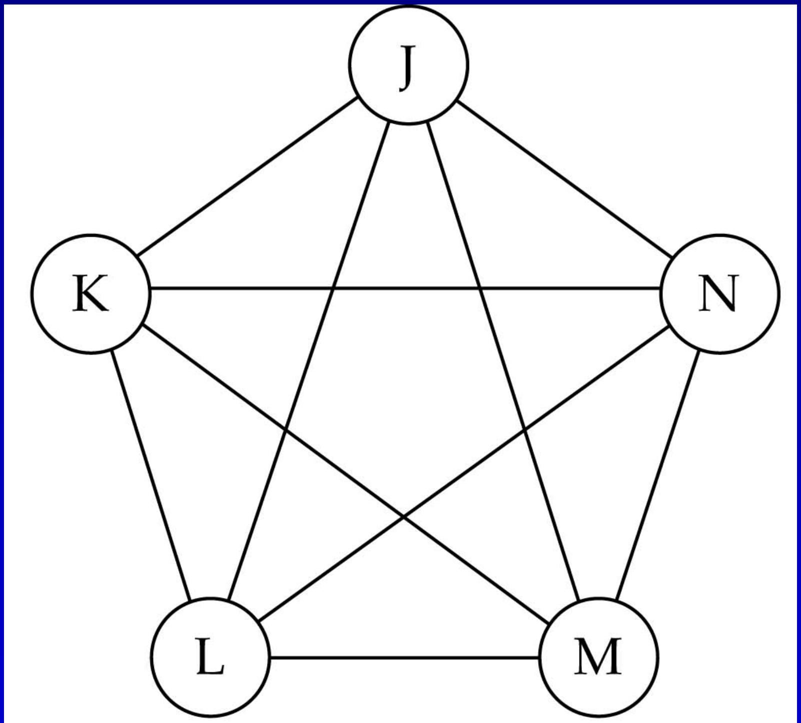
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

*N * (N-1) / 2*

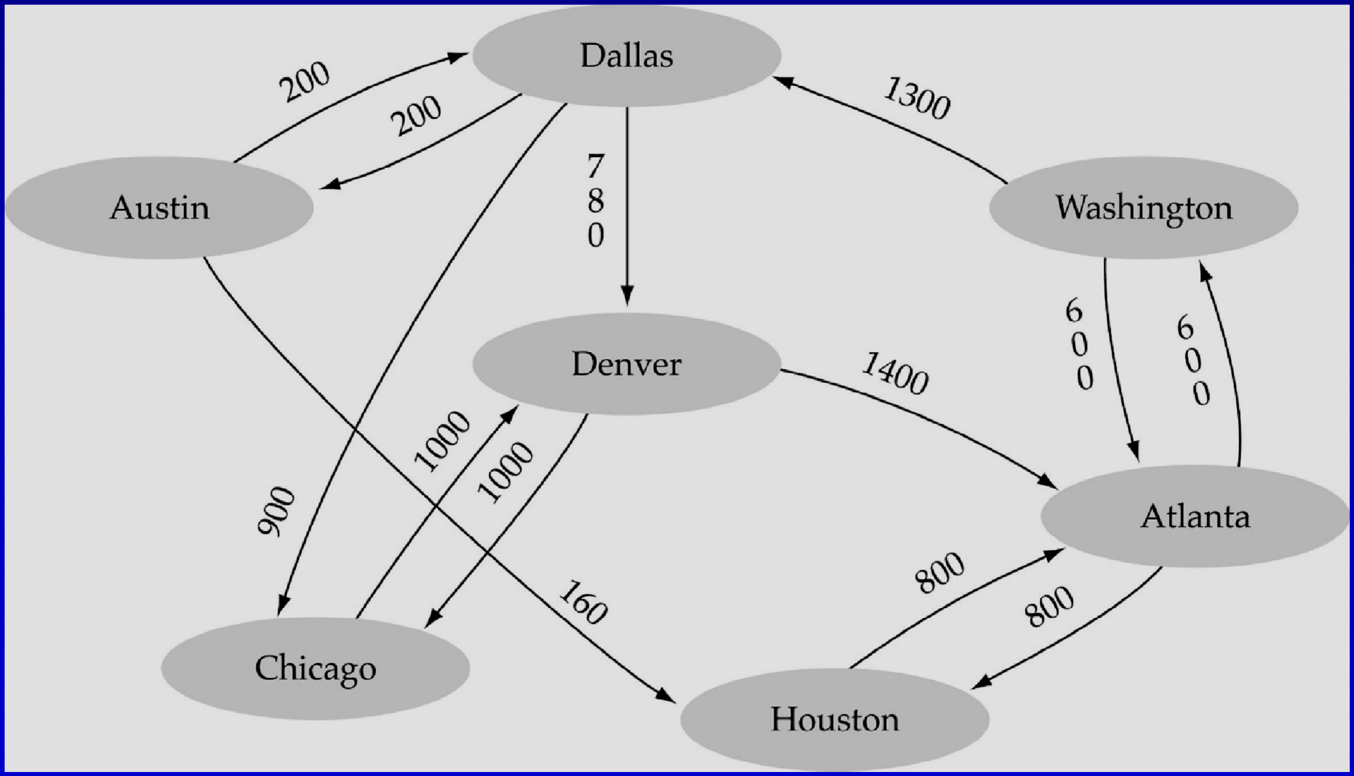
$O(N^2)$



(b) Complete undirected graph.

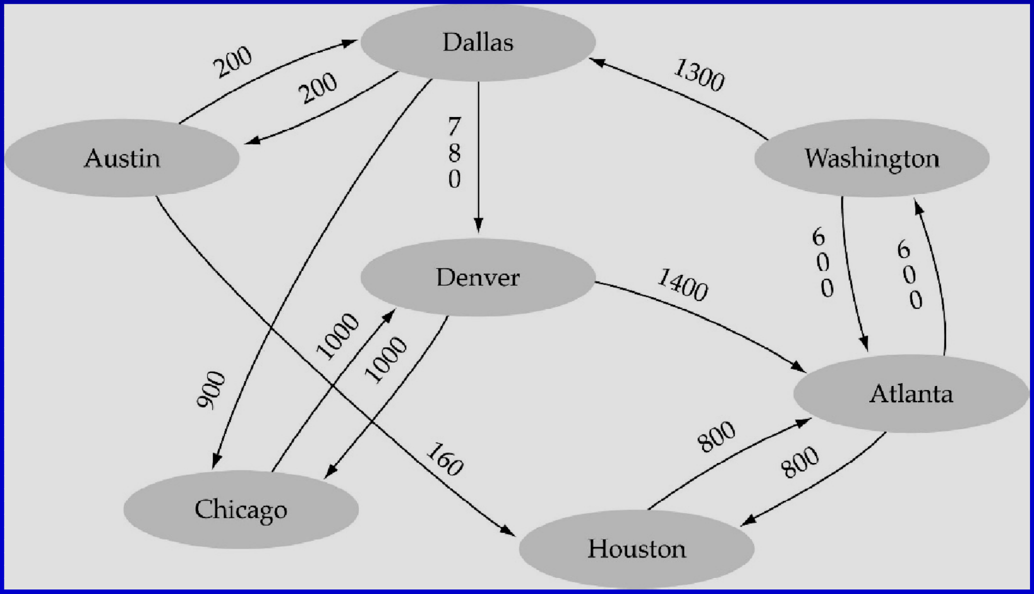
Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



Graph implementation

- Array-based implementation
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges



Array-based implementation

graph

.numVertices 7

.vertices

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

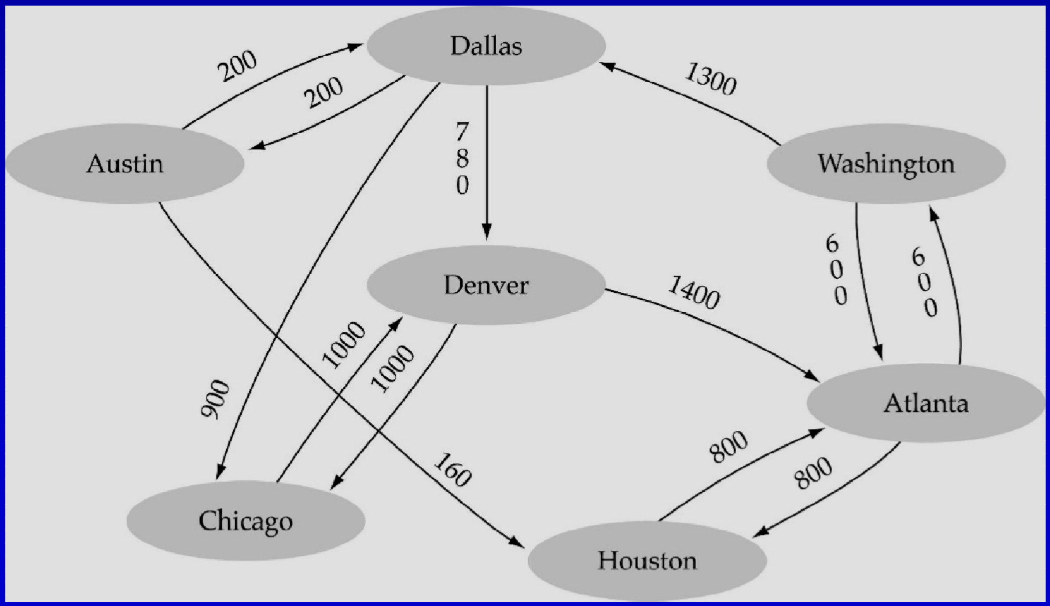
.edges

	[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]	•	•	•	•	•	•	•	•	•	•

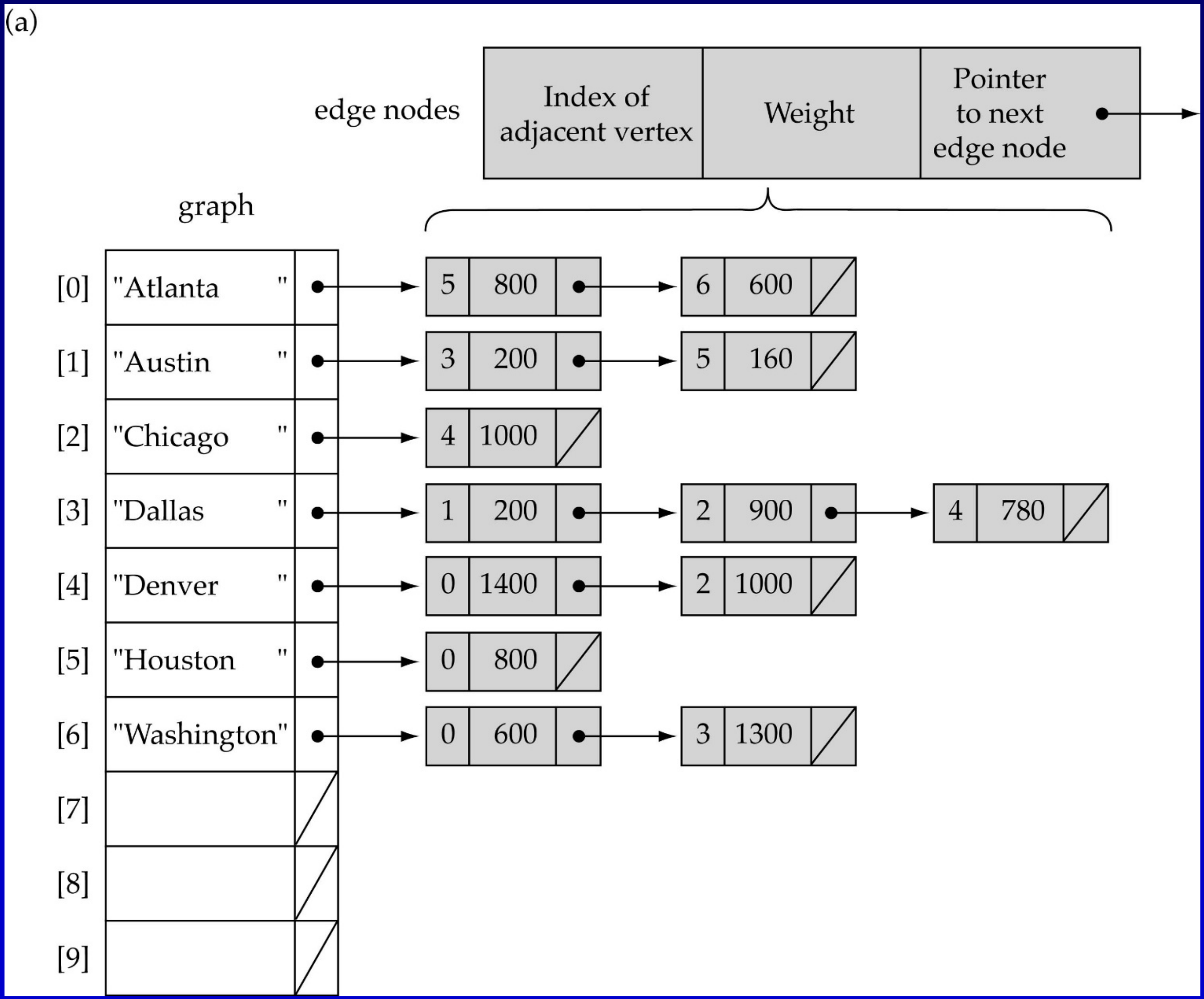
(Array positions marked '•' are undefined)

Graph implementation (cont.)

- <u>Linked-list implementation</u>
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)



Linked-list implementation



Graph searching

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

Depth-First-Search (DFS)

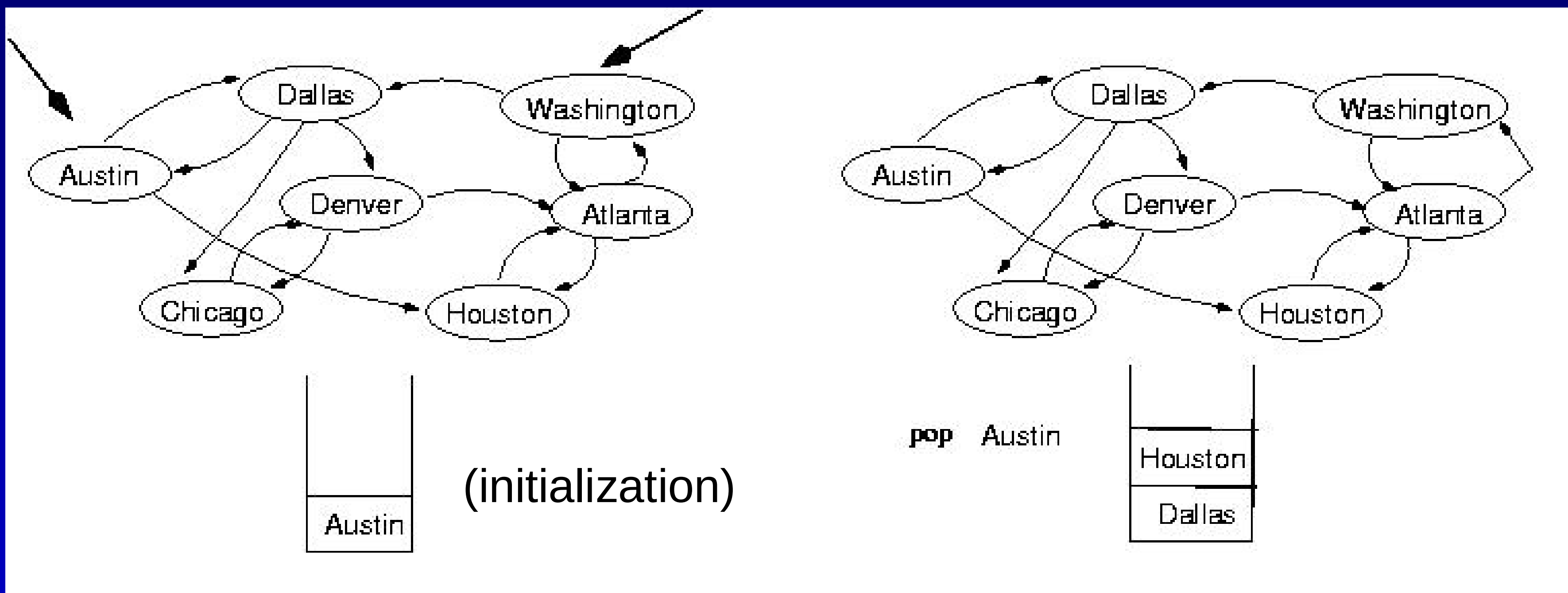
- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a *stack*

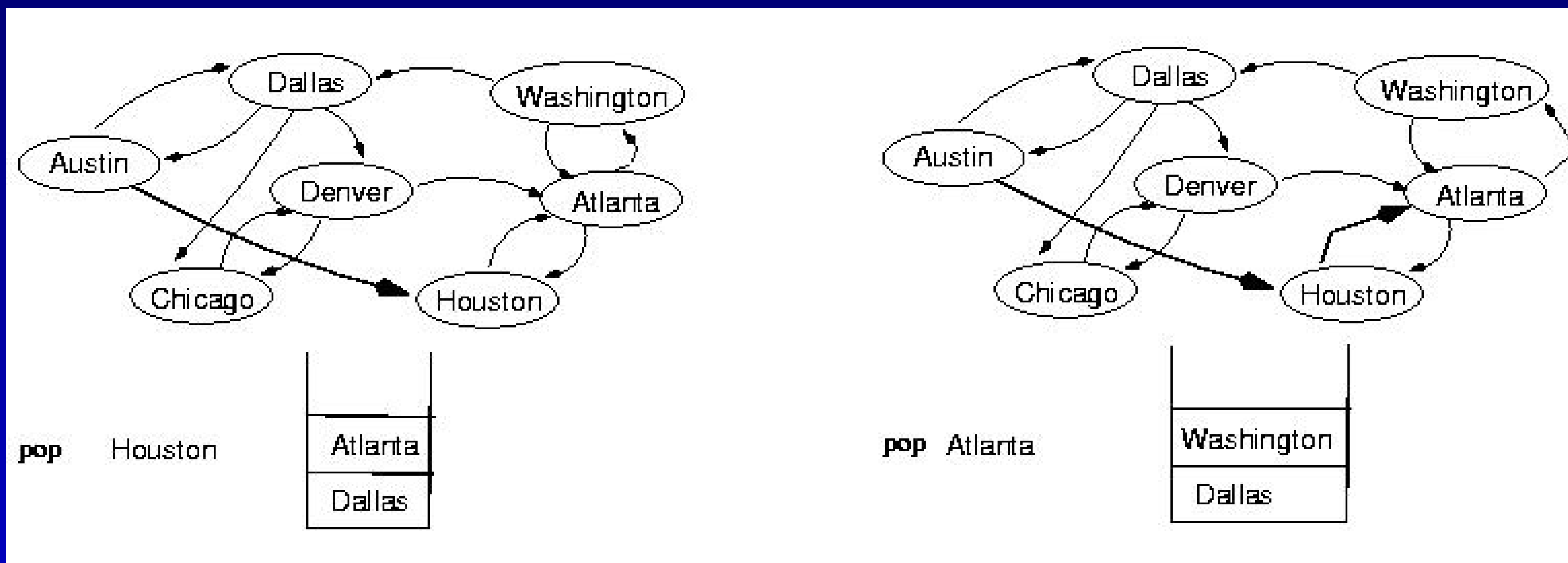
Depth-First-Search (DFS) *(cont.)*

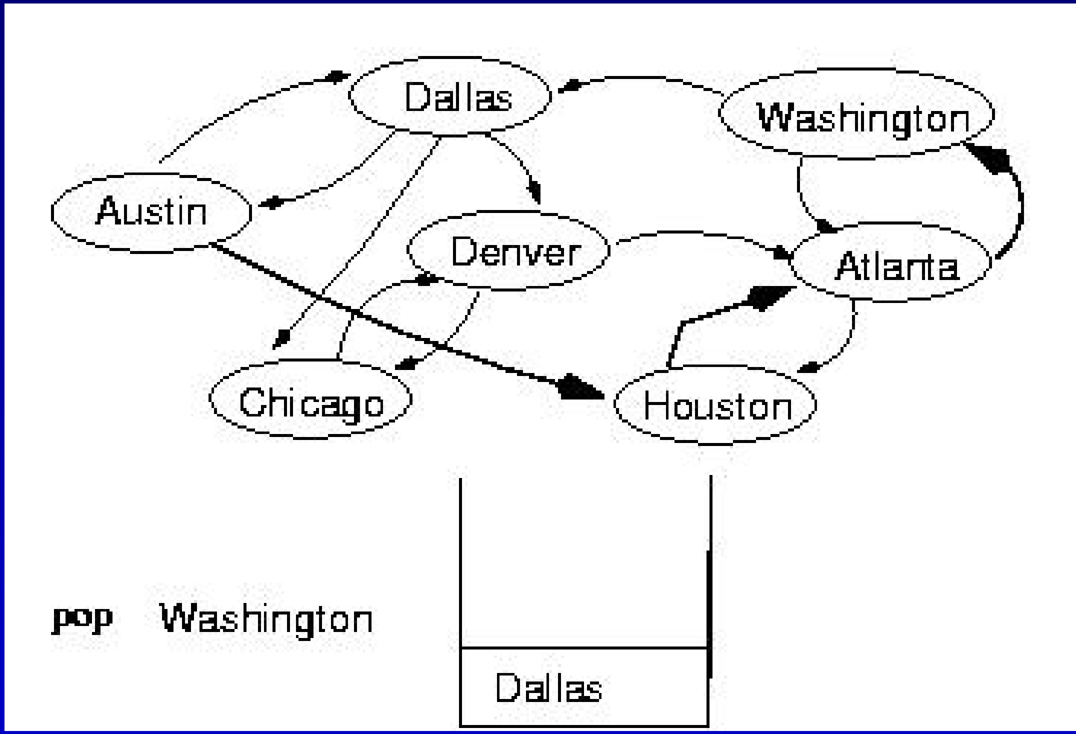
```
Set found to false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Push all adjacent vertices onto stack
WHILE !stack.IsEmpty() AND !found

IF(!found)
  Write "Path does not exist"
```

start

end


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
Austin
(initialization)
Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Austin
Houston
Dallas


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Houston
Atlanta
Dallas
Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Atlanta
Washington
Dallas


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Washington
Dallas

Breadth-First-Searching (BFS)

- What is the idea behind BFS?
  – 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-Searching (BFS) (cont.)

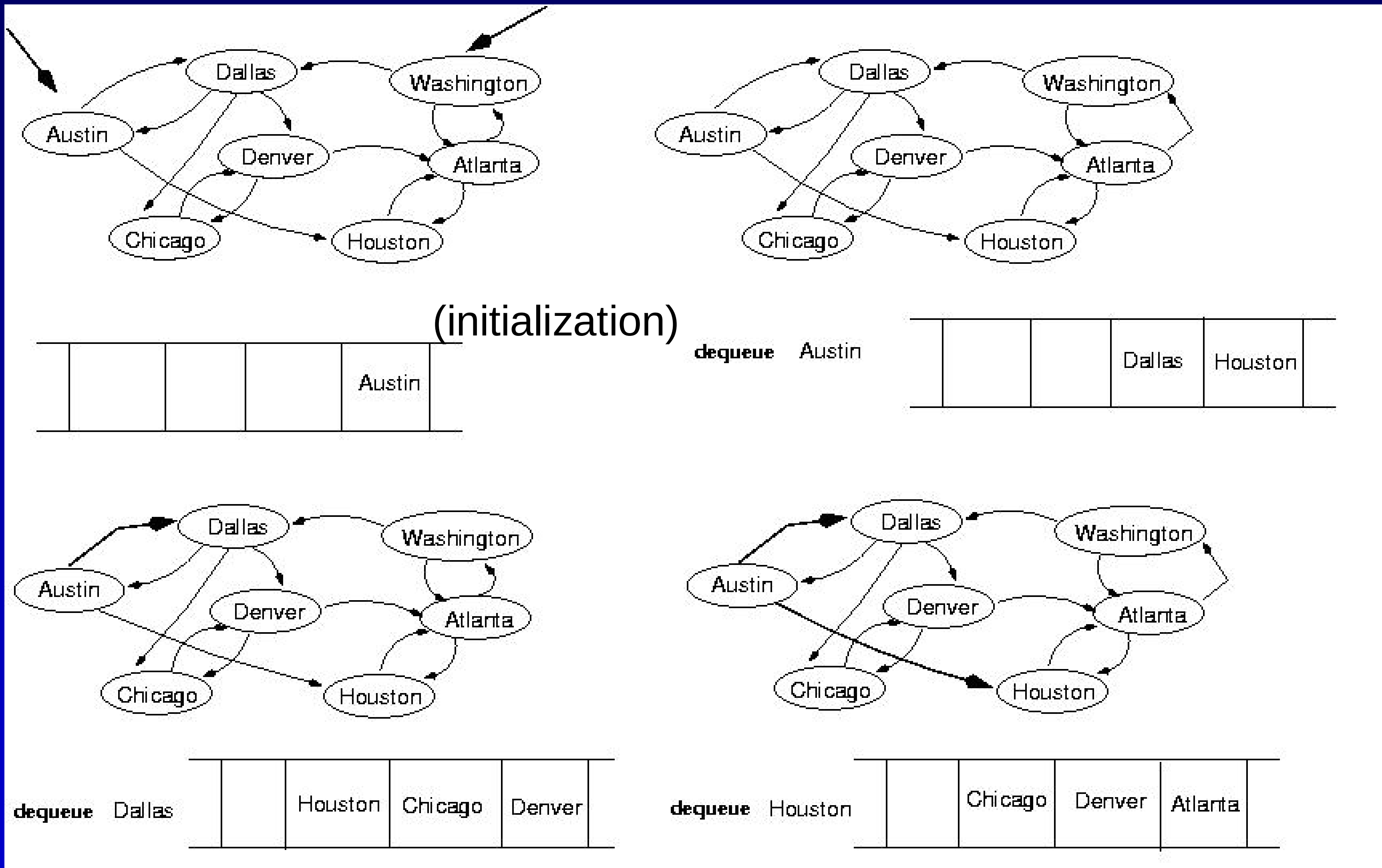
- BFS can be implemented efficiently using a *queue*

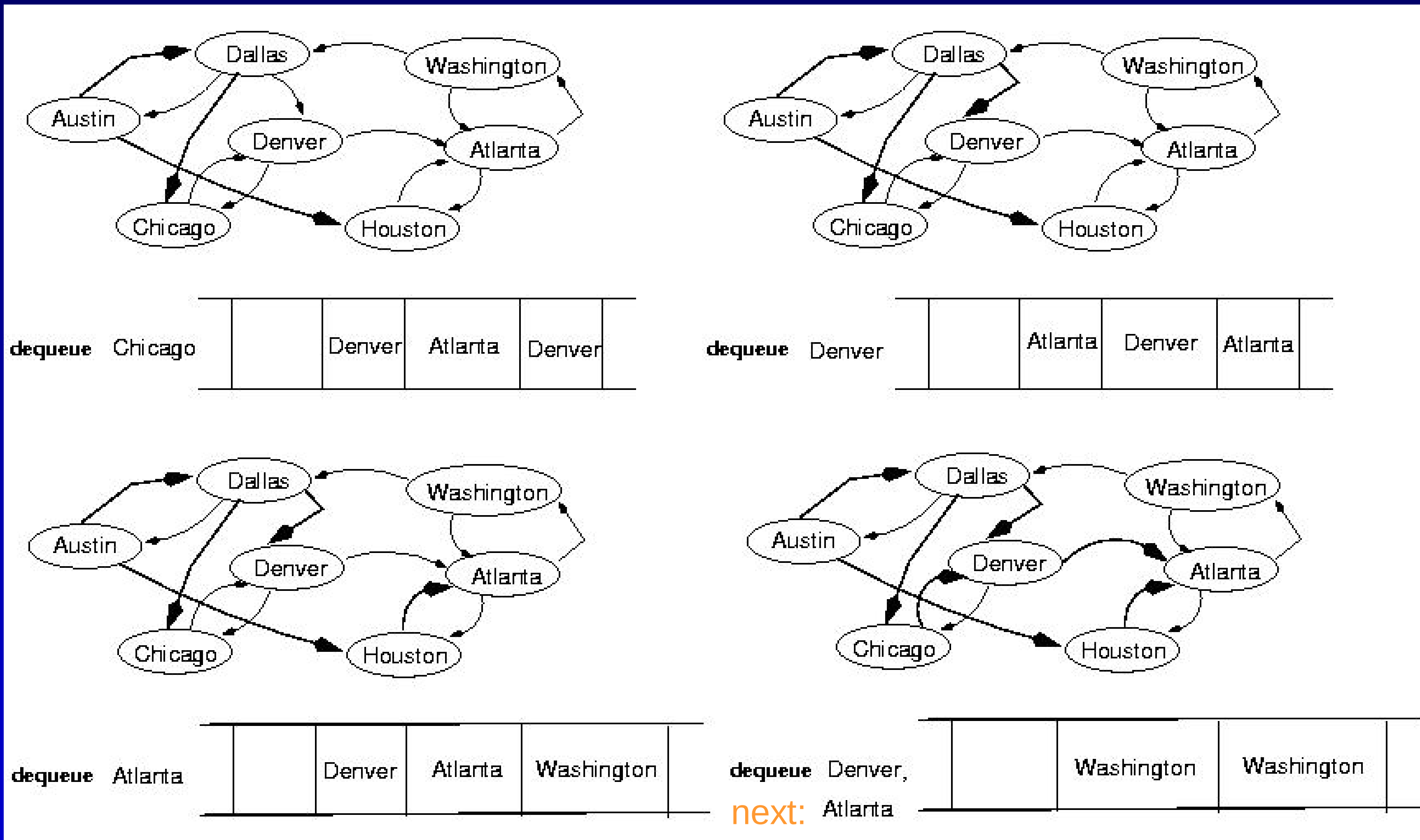
```
Set found to false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
```

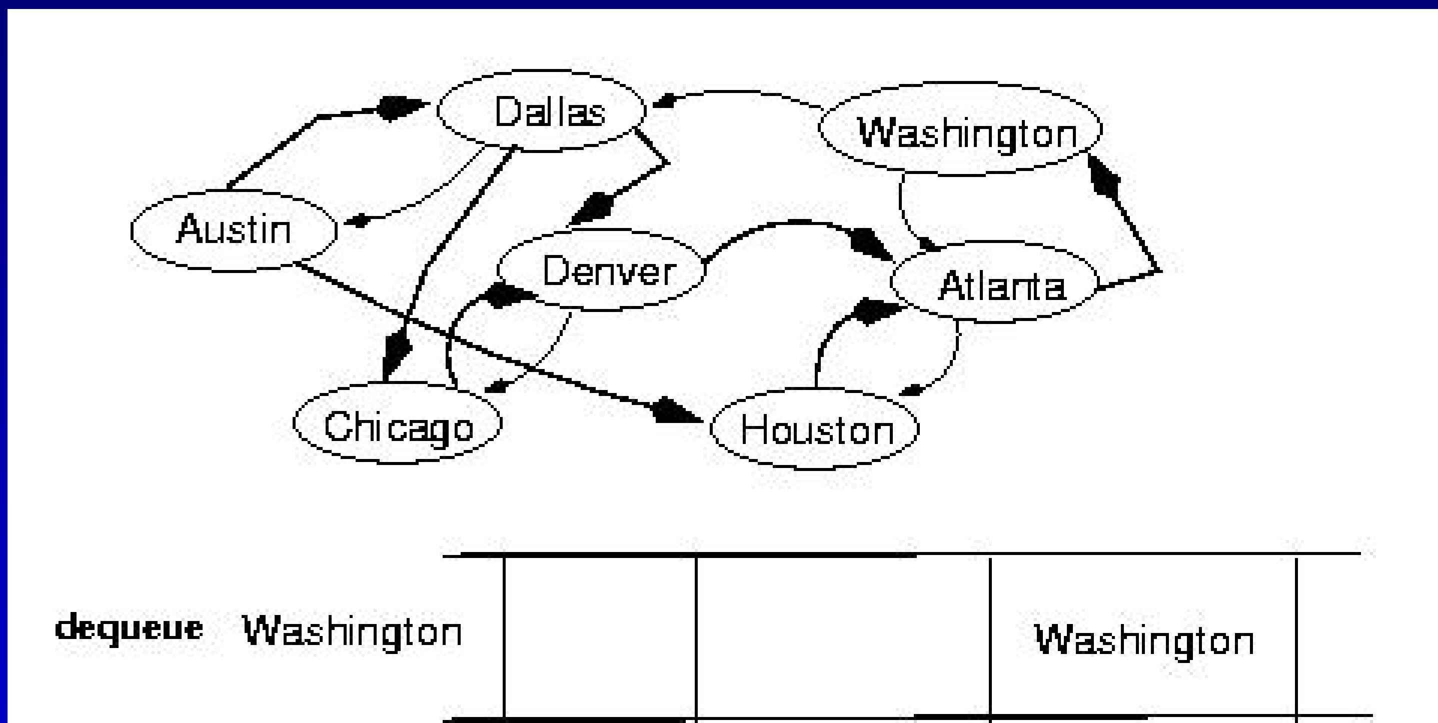
```
IF(!found)
  Write "Path does not exist"
```

- Should we mark a vertex when it is enqueued or when it is dequeued ?

start end


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
(initialization)
Austin
dequeue Austin
Dallas
Houston
dequeue Dallas
Houston
Chicago
Denver
dequeue Houston
Chicago
Denver
Atlanta


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
dequeue Chicago
Denver
Atlanta
Denver
dequeue Denver
Atlanta
Denver
Atlanta
dequeue Atlanta
Denver
Atlanta
Washington
dequeue Denver,
next: Atlanta
Washington
Washington


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
dequeue Washington
Washington

```
 else {
   if(!graph.IsMarked(vertex)) {
     graph.MarkVertex(vertex);
     graph.GetToVertices(vertex, vertexQ);

     while(!vertxQ.IsEmpty()) {
       vertexQ.Dequeue(item);
       if(!graph.IsMarked(item))
         queue.Enqueue(item);
     }
   }
  }
 } while (!queue.IsEmpty() && !found);

 if(!found)
   cout << "Path not found" << endl;
}
```

Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- *Shortest path*: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
  - Austin->Houston->Atlanta->Washington: 1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles

Single-source shortest-path problem (cont.)

- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is weightless or all the weights are the same

(mark vertices before Enqueue)