Chapter Seven

Graphs

Classification of Data Structures

Linear Data Structure

Arrays

Linked Lists

Stacks

Queues

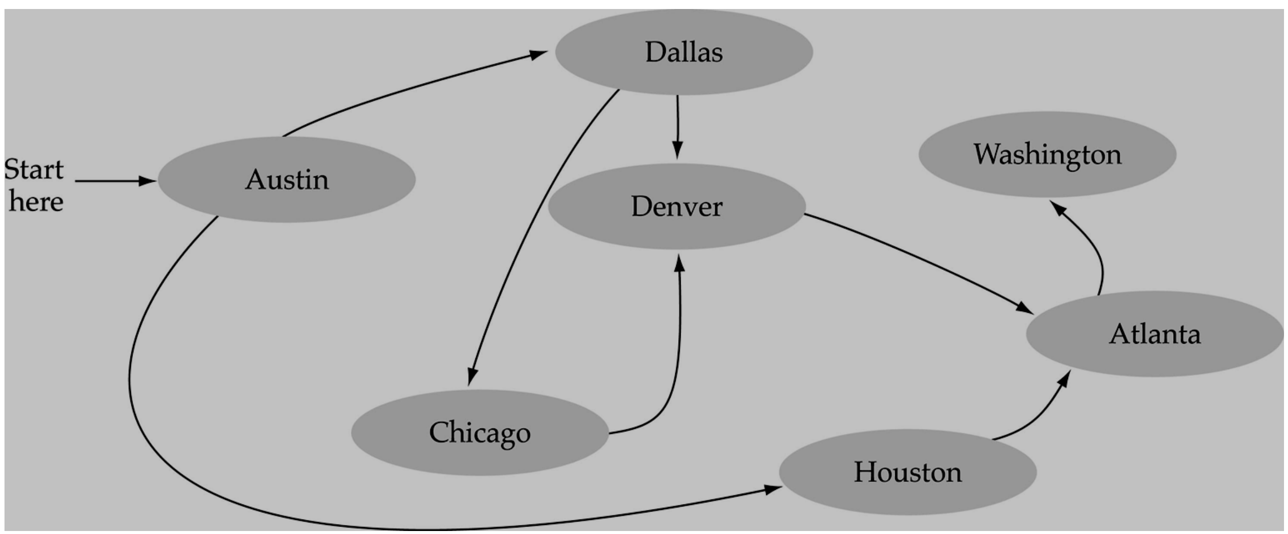
Non-Linear Data Structure

Trees

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

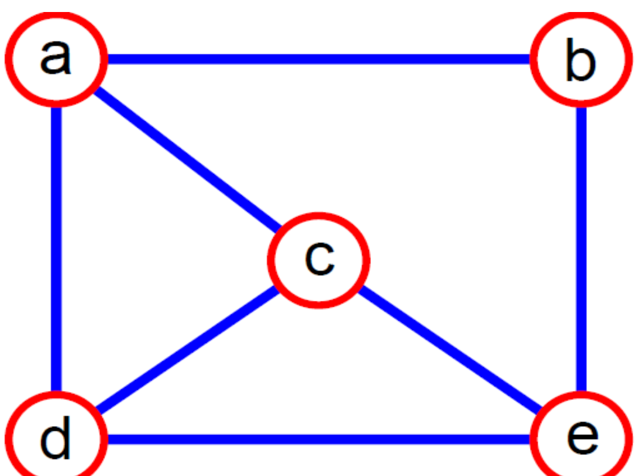
- A graph *G* is defined as follows:

  G=(V,E)

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

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

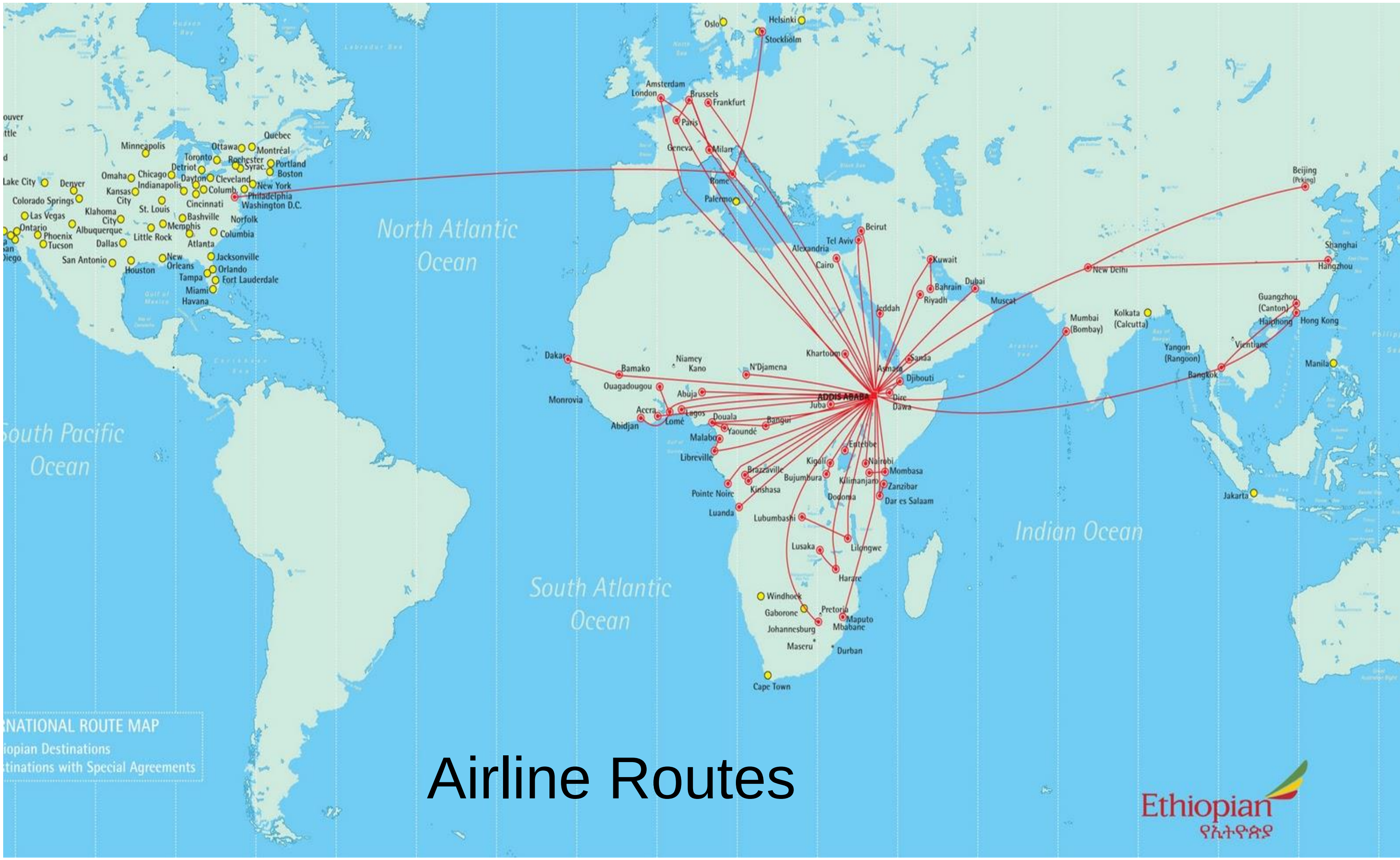
- An edge e = (u,v) is a pair of vertices
- **Example**:



V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

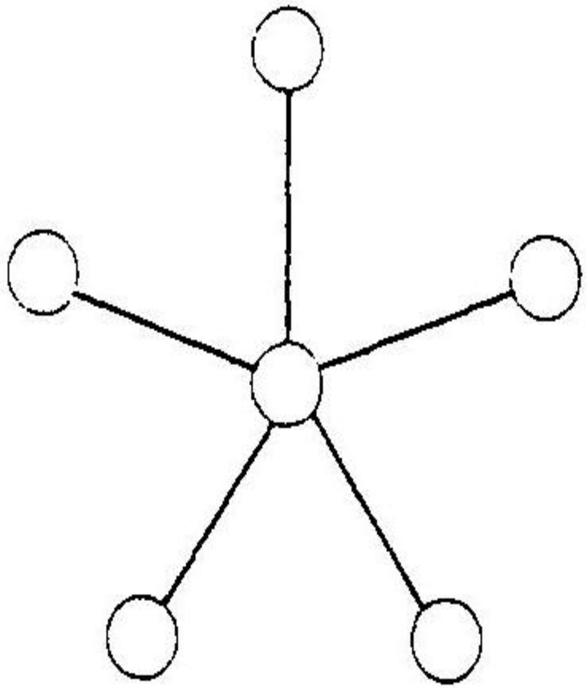
Applications of Graphs



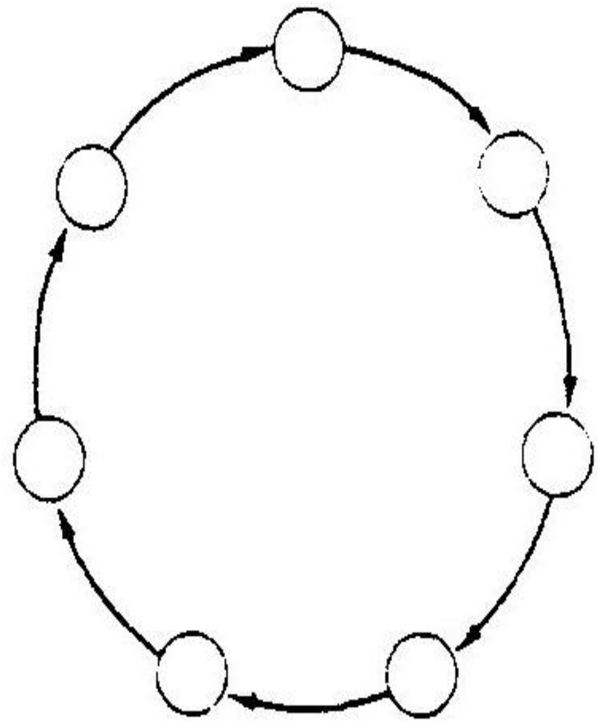
Airline Routes

Applications of Graphs (cont.)

- Computer Networks



(a) A star network

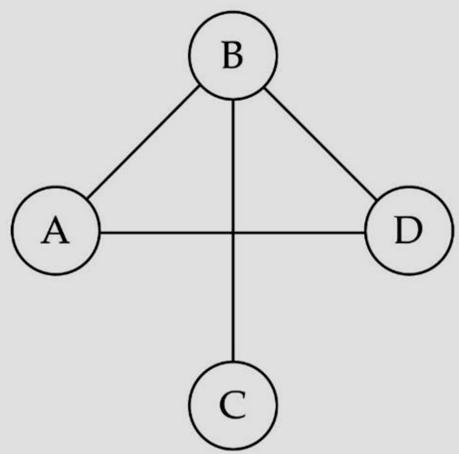


(b) A ring network

Directed vs. Undirected Graphs

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

(a) Graph1 is undirected graph

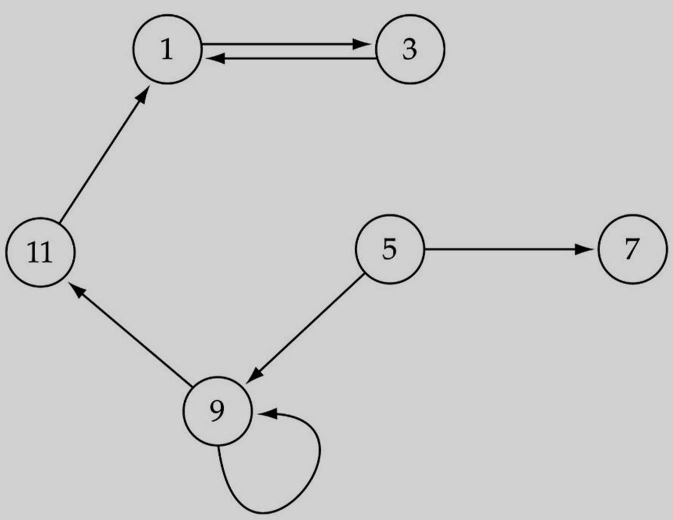


V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

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) }

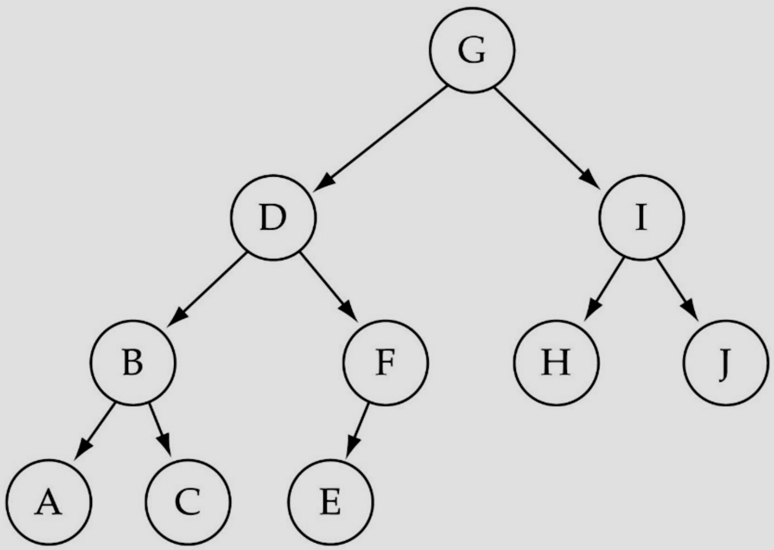
Warning: if the graph is directed, the order of the vertices in each edge is important !!

E is a set of ordered pairs of elements of V.

Trees vs Graphs

- Trees are special cases of graphs!!

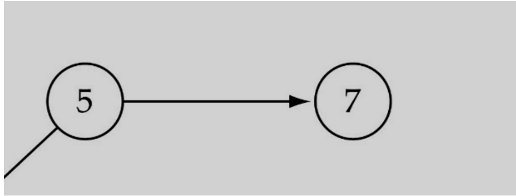
(c) Graph3 is a directed graph.



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) }

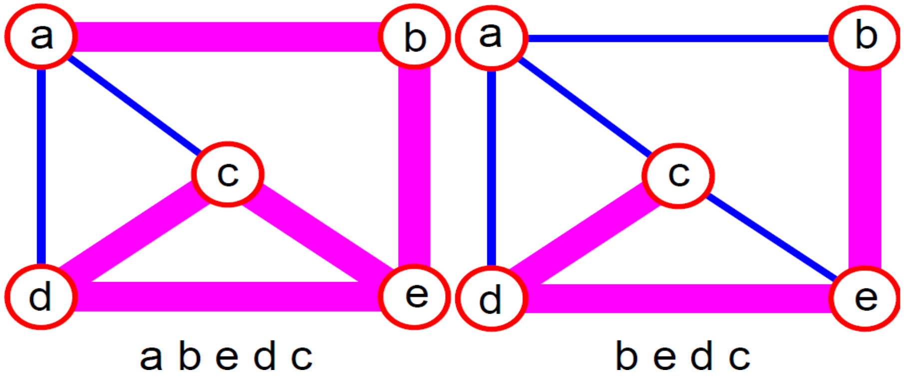
Graph Terminology

- **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.

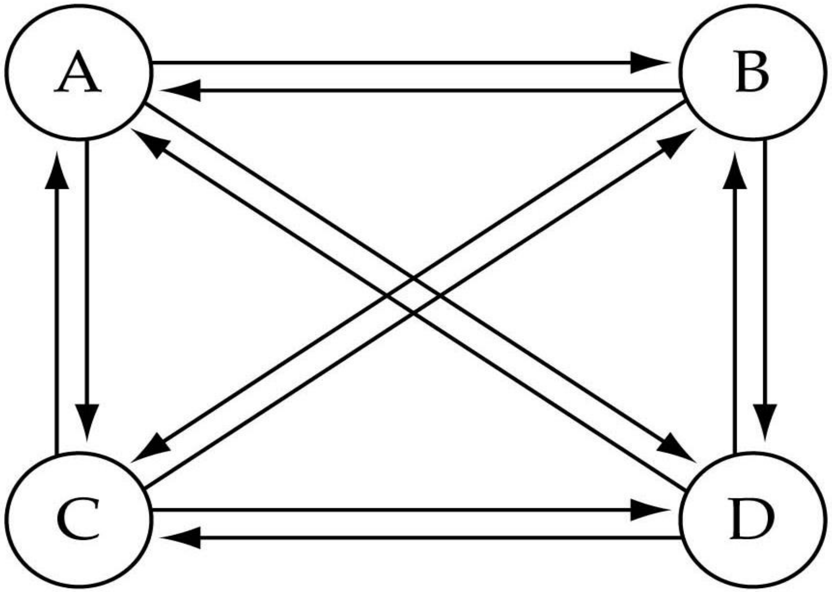


Graph Terminology (cont.)

- **Complete graph:** a graph in which ***every vertex is directly connected*** to ***every other vertex.***
- What is the ***number of edges*** in a complete directed graph with ***N vertices***?

N * (N-1)

$$O(N^2)$$



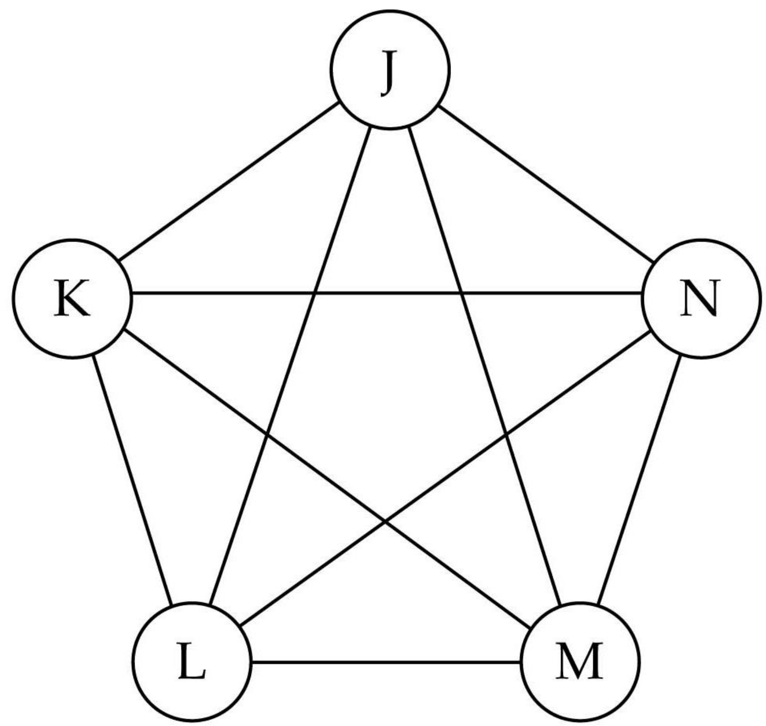
(a) Complete directed graph.

Graph Terminology (cont.)

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

N * (N-1) / 2

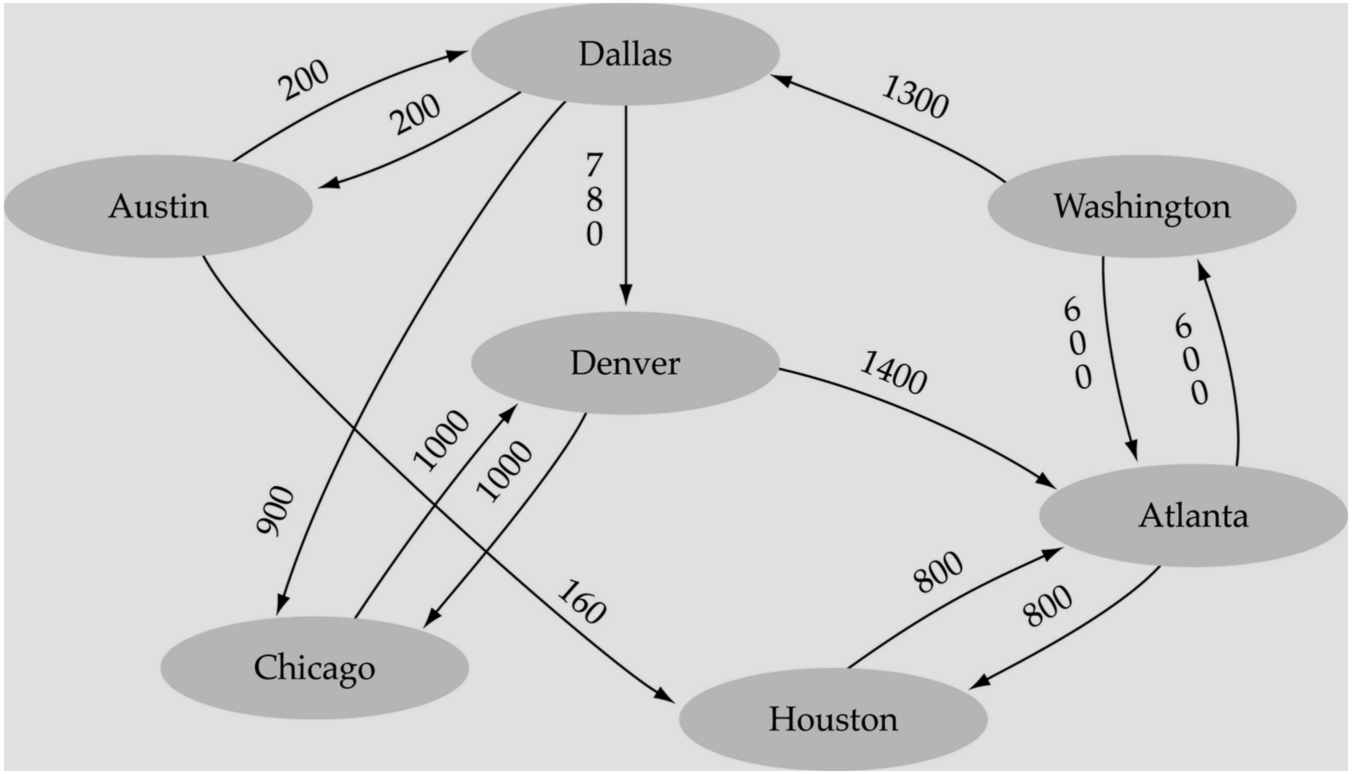
$$O(N^2)$$



(b) Complete undirected graph.

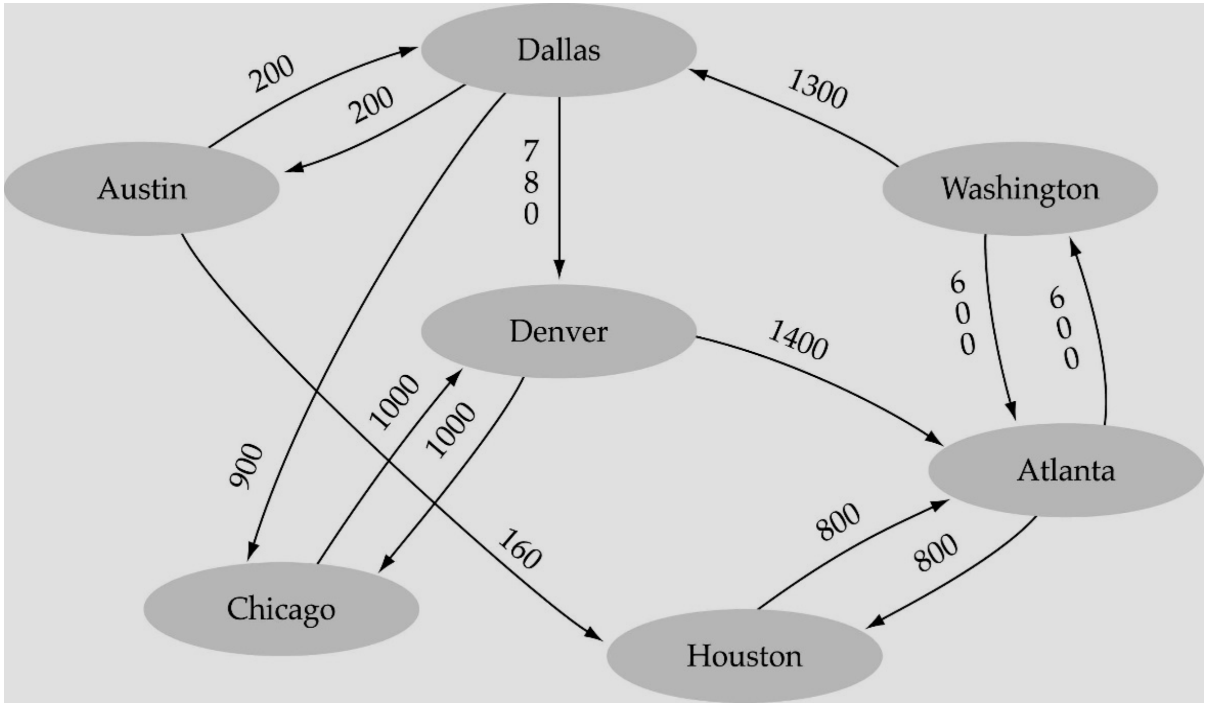
Graph Terminology (cont.)

- **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

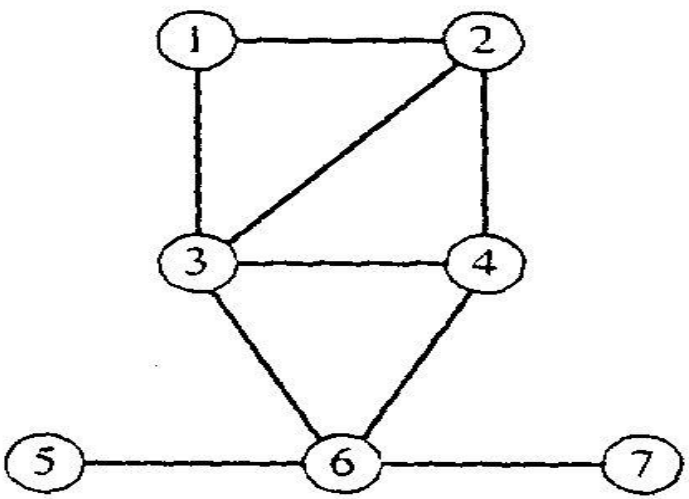
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•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

(Array positions marked '•' are undefined)

Adjacency Matrix Representation

- Let $G = (V,E)$, $n = |V|$, $m = |E|$, $V = \{v_1, v_2, \ldots, v_n\}$
- G can be represented by an $n \times n$ matrix.

Implemented using 2D array



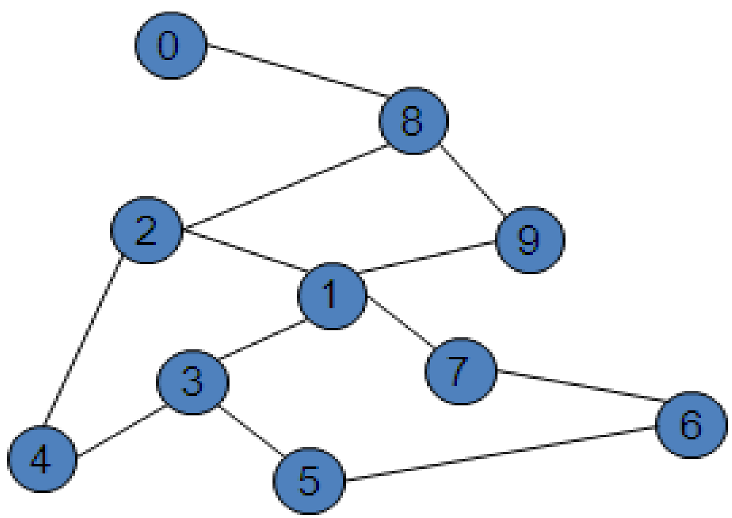
(a) An undirected graph

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

(b) Its adjacency matrix

Adjacency Matrix Representation (cont.)

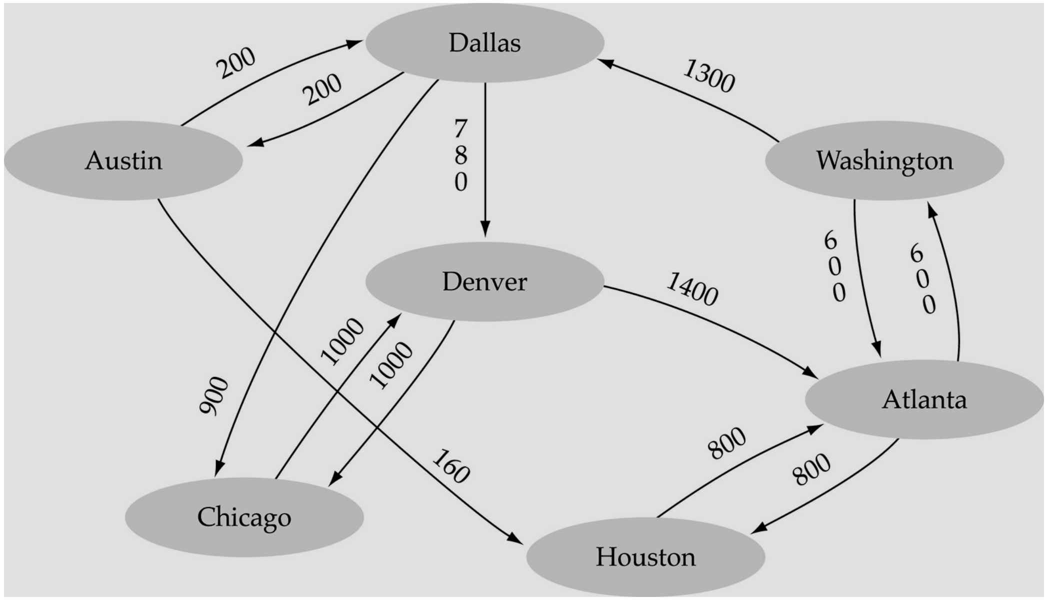
Adjacency Matrix Example

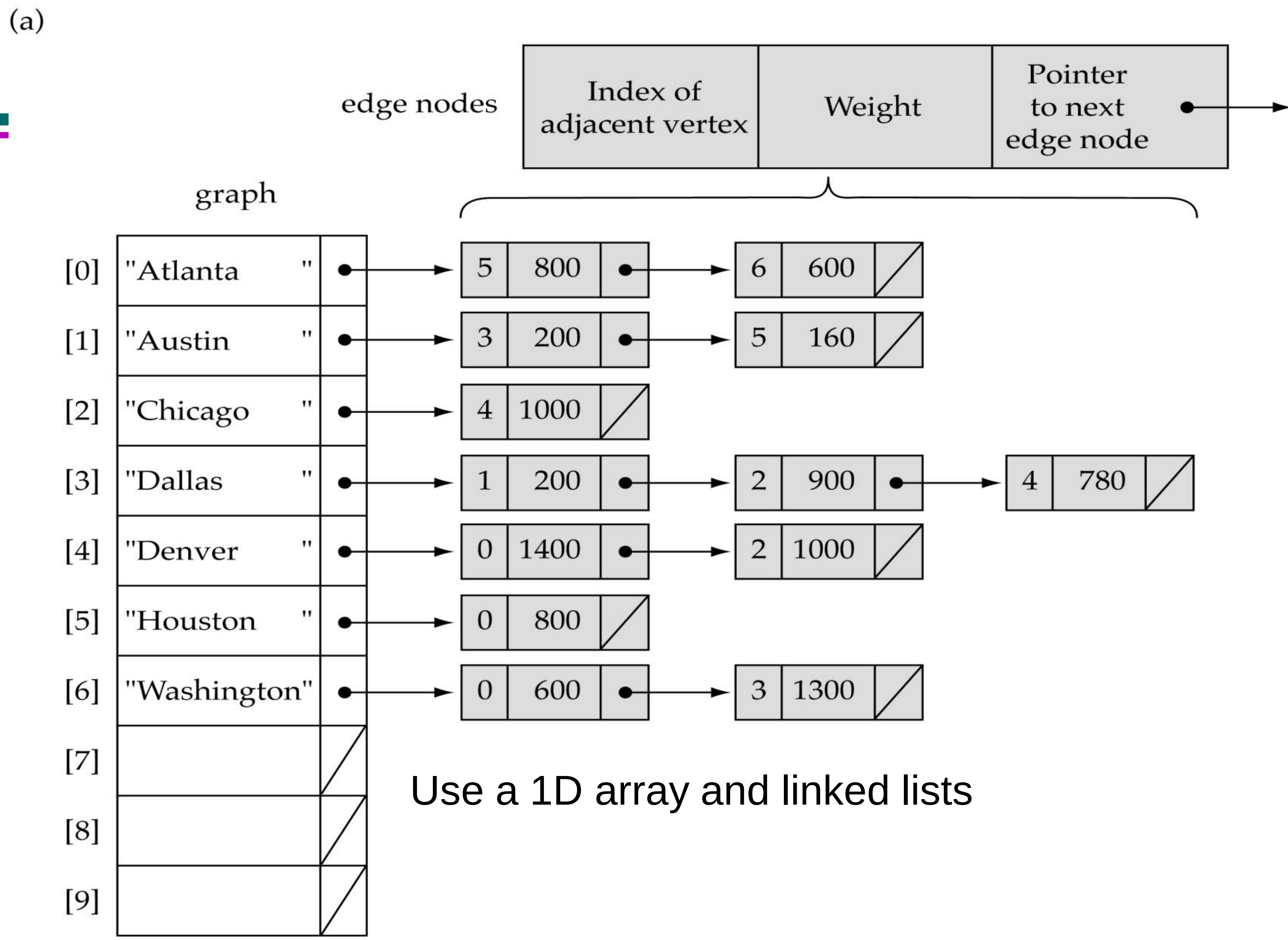


	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	1	0	0	0	1	0	1
2	0	1	0	0	1	0	0	0	1	0
3	0	1	0	0	1	1	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0
5	0	0	0	1	0	0	1	0	0	0
6	0	0	0	0	0	1	0	1	0	0
7	0	1	0	0	0	0	1	0	0	0
8	1	0	1	0	0	0	0	0	0	1
9	0	1	0	0	0	0	0	0	1	0

Graph Implementation (cont.)

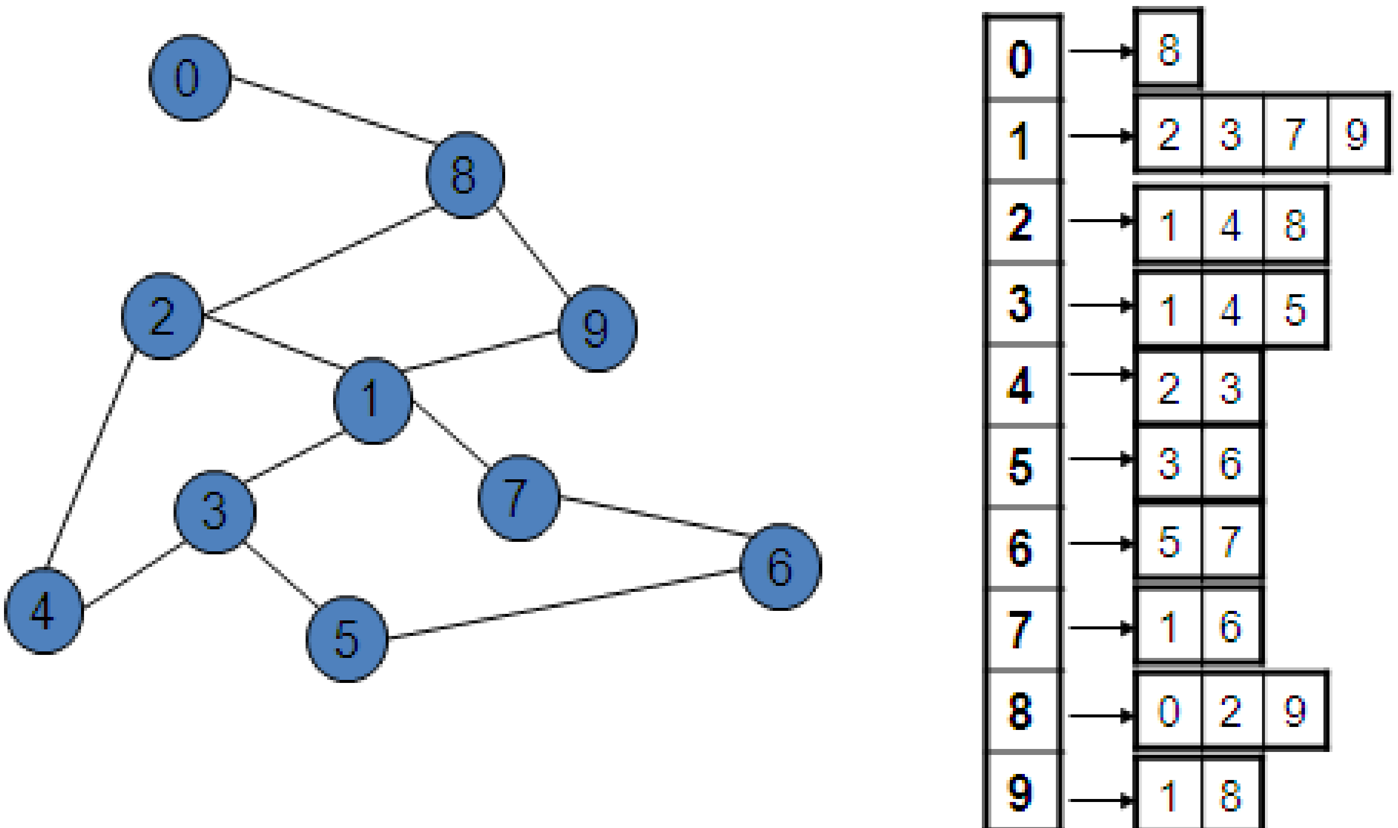
- **Linked-list Implementation**
  - A 1D array is used to represent the vertices.
  - A list is used for ***each vertex v*** which contains the ***vertices*** which are adjacent from *v* (adjacency list).





Use a 1D array and linked lists

Adjacency List Example



Each list *A[i]* stores the ids of the vertices adjacent to vertex *i.*

Adjacency Matrix vs Adjacency List Representation

- **Adjacency Matrix**
  - Good for dense graphs (more edges):$|E| \sim O(|V|^2)$.
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$.
  - Connectivity between two vertices can be tested quickly.
- **Adjacency List**
  - Good for sparse graphs(few edges) -- $|E| \sim O(|V|)$.
  - Memory requirements: $O(|V| + |E|) = O(|V|)$.
  - Vertices adjacent to another vertex can be found quickly.

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***.
  - ***Search deeper*** in the graph, when ever possible.
- Given an input graph G = (V, E) and a source ***vertex S***, from where the ***searching starts***.
  - First we visit the ***starting node***.
  - Then we travel through ***each node*** along a path, which begins at S.
  - That is we ***visit a neighbor vertex of S*** and ***again a neighbor of a neighbor of S***, and so on.
- DFS can be implemented efficiently using a ***stack.***

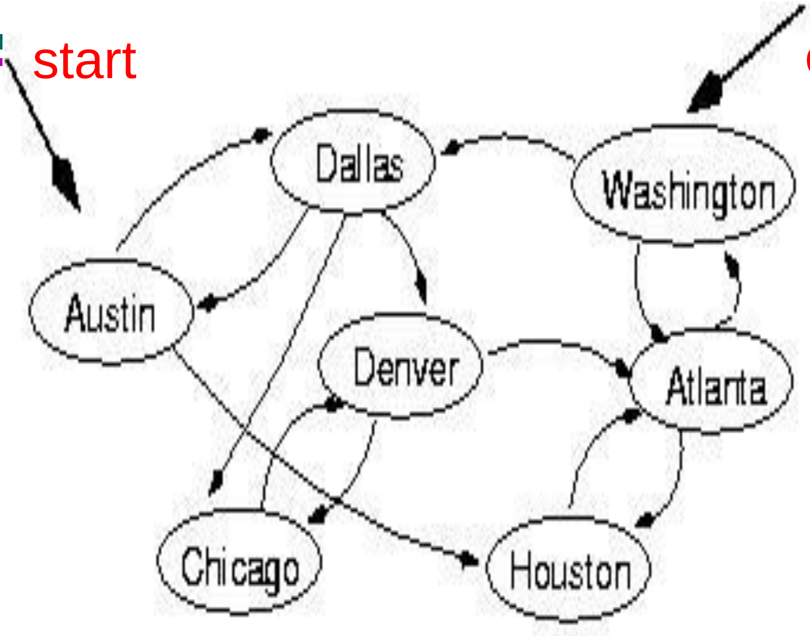
Algorithm

1. Input the vertices and edges of the graph G = (V, E).
2. Input the source vertex and assign it to the variable S.
3. ***Push the source vertex*** to the stack.
4. Repeat the steps 5 and 6 until the stack is empty & ***destination is found***.
5. ***Pop the top element*** of the stack and display it.
6. Push the vertices which is ***neighbor*** to just popped element, if it is not in the stack displayed (i.e; not visited).
7. Exit.

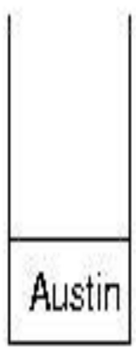
Algorithm

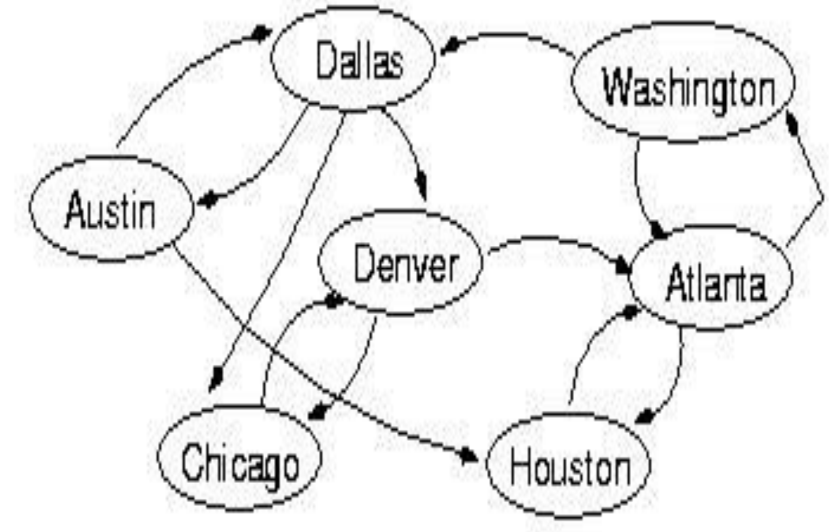
```
Set found to false
Push(startVertex)
DO
  Pop(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Push all adjacent vertices onto stack
WHILE !IsEmpty() AND !found

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

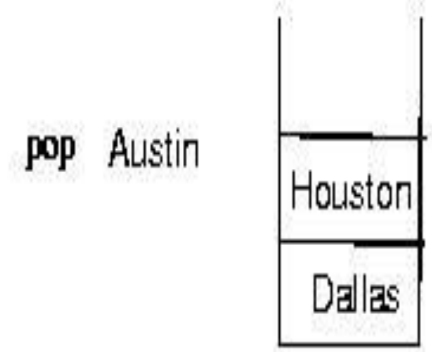
Example: Is there a path from Austin to Washington?

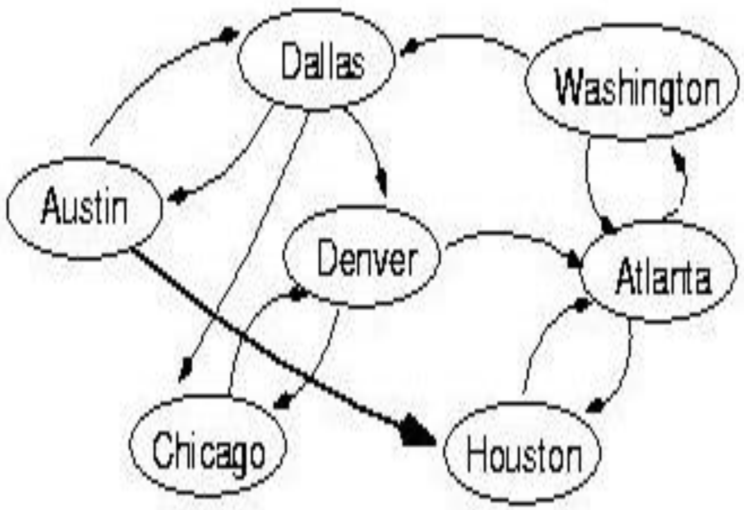


(initialization)

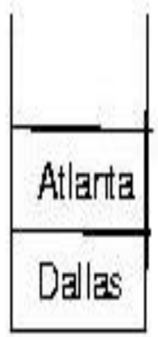


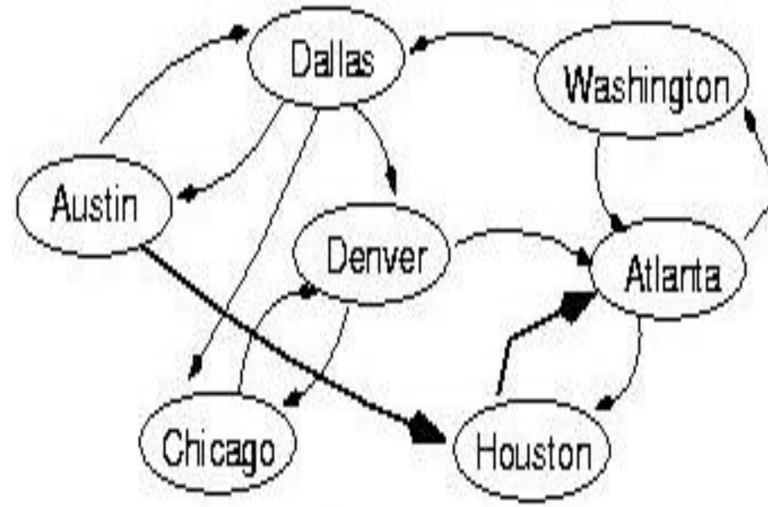




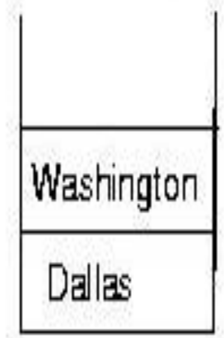


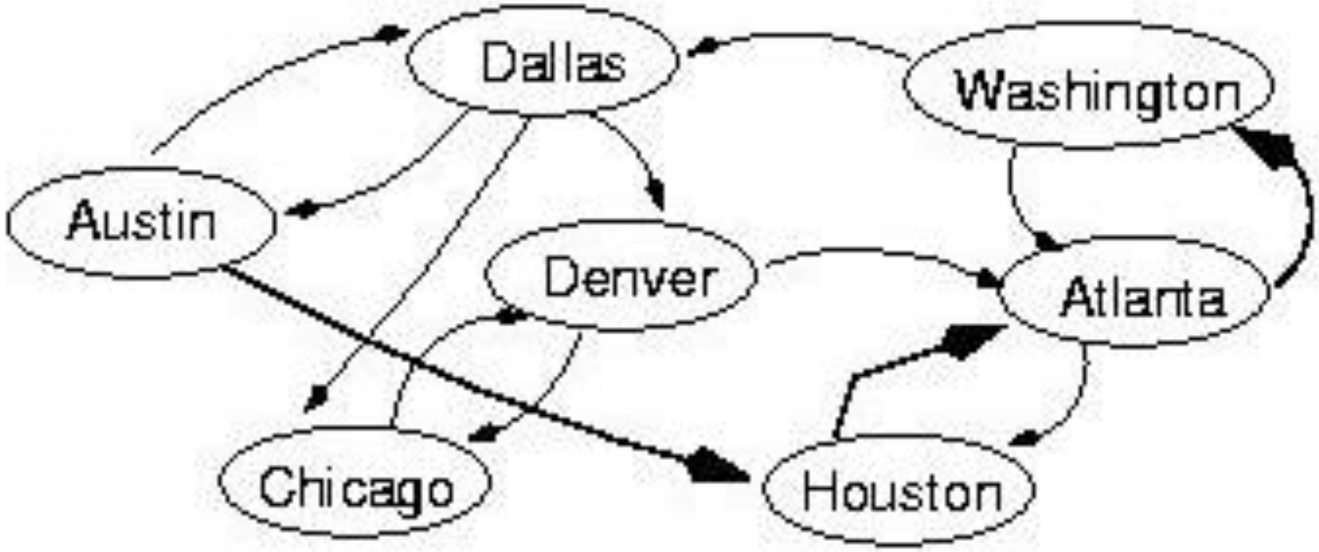
pop Houston





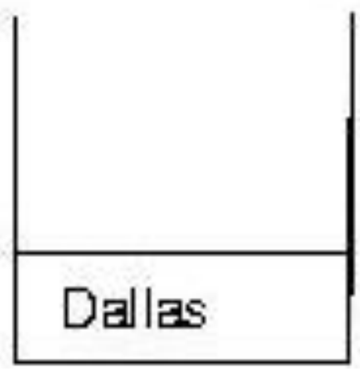
pop Atlanta





Found!

pop Washington



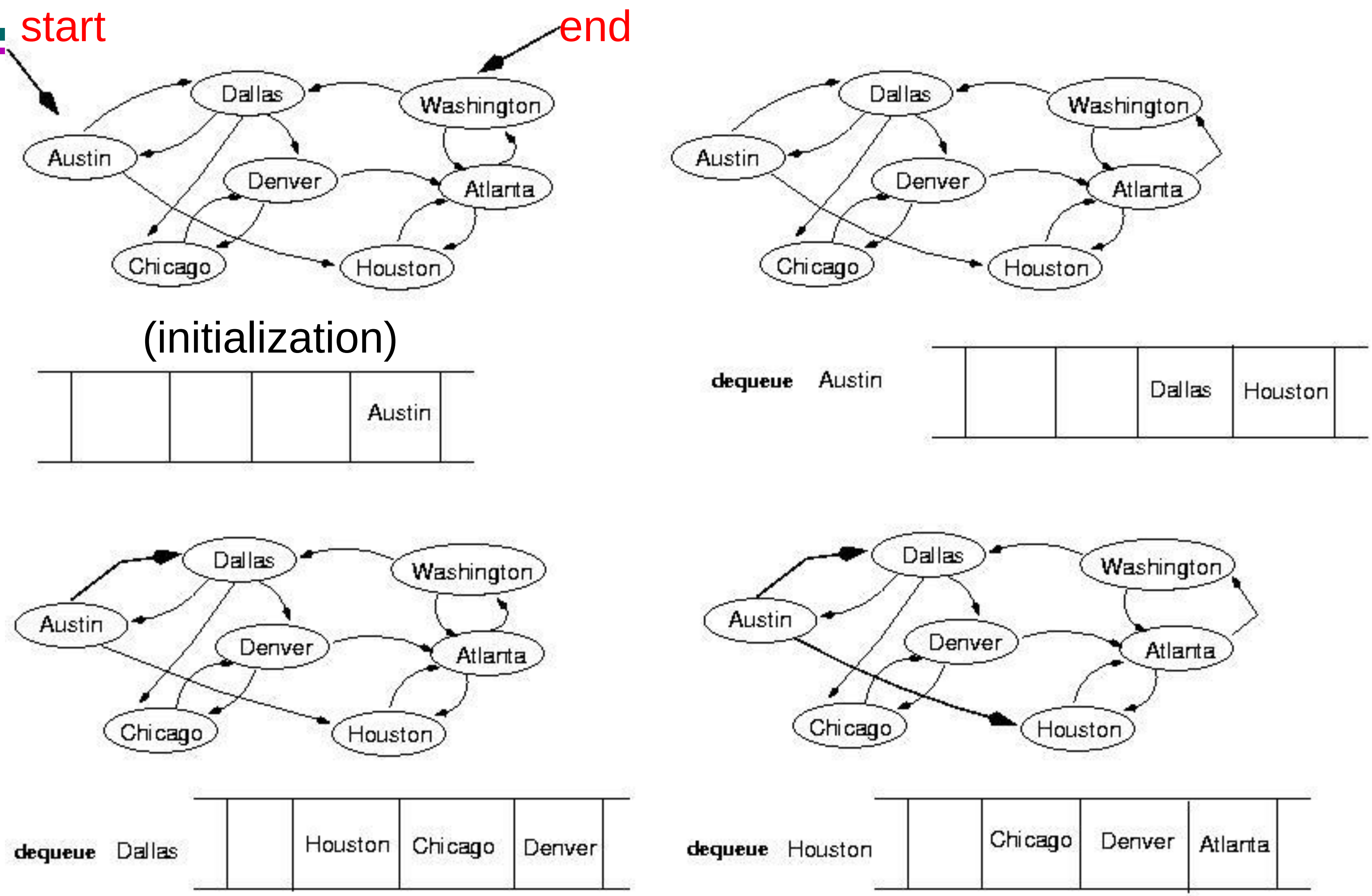
Breadth-First Search (BFS)

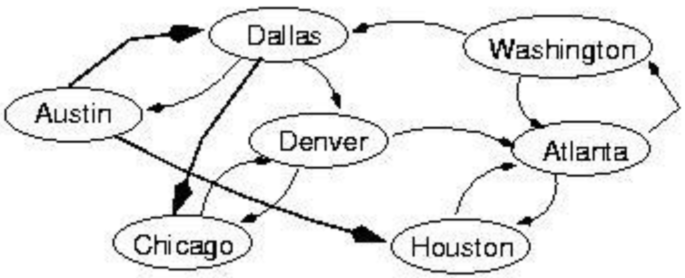
- What is the idea behind BFS?
  - Look at all possible paths at ***the same depth*** before you go at a deeper level.
  - ***Explore every vertex*** that is reachable from source vertex, S.
  - Examine the entire vertices neighbor to S.
  - Then ***traverse all the neighbors*** of the neighbors of S and so on.
  - A queue is used to keep track of the progress of traversing the neighbor nodes.
  - BFS can be implemented efficiently using a ***queue.***

Algorithm

```
Set found to false
enqueue(startVertex)
DO
  dequeue(vertex)
  IF vertex == endVertex
     Set found to true
  ELSE
     Enqueue all adjacent vertices onto queue
WHILE !IsEmpty() AND !found
  IF(!found)
     Write "Path does not exist"
```

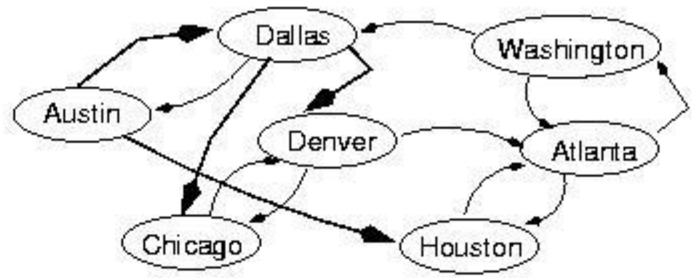
Example: Is there a path from Austin to Washington?





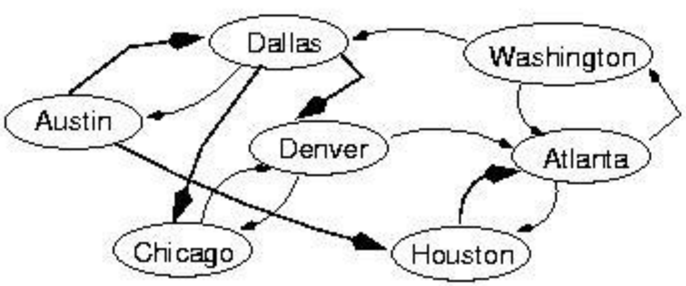
dequeue Chicago

		Denver	Atlanta	Denver



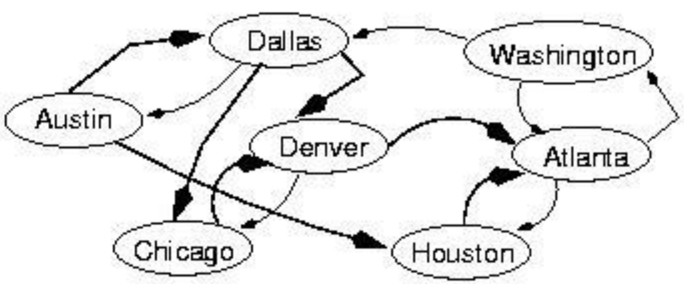
dequeue Denver

		Atlanta	Denver	Atlanta



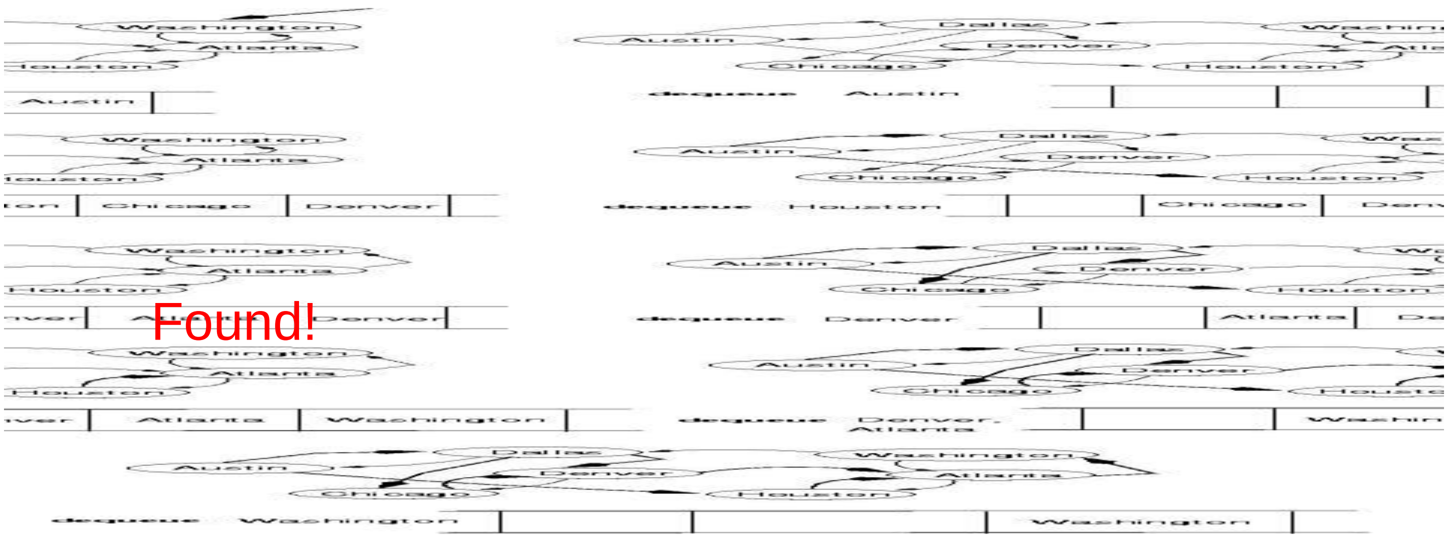
dequeue Atlanta

		Denver	Atlanta	Washington



dequeue Denver,
next: Atlanta

		Washington	Washington


Found!

Reading Assignment

- Minimum Spanning Tree
  - Kruskal’s Algorithm
  - Prim‘s Algorithm