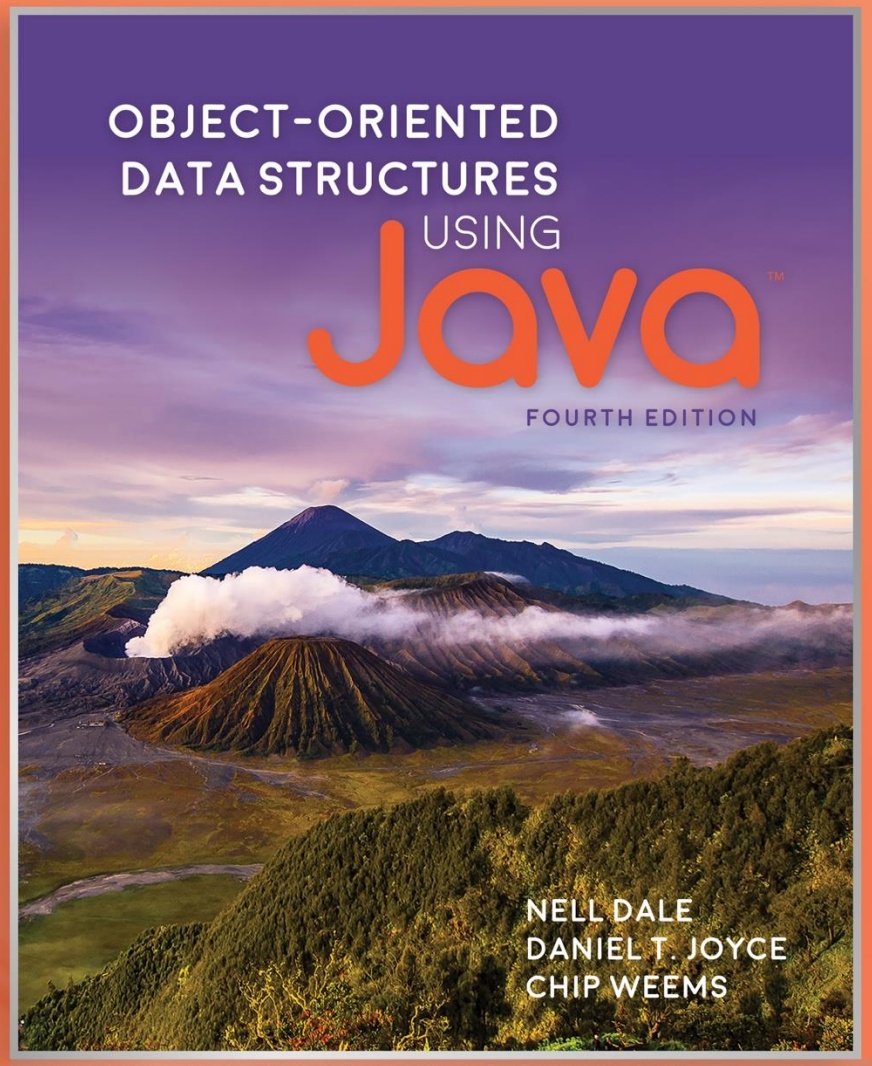
Chapter 10

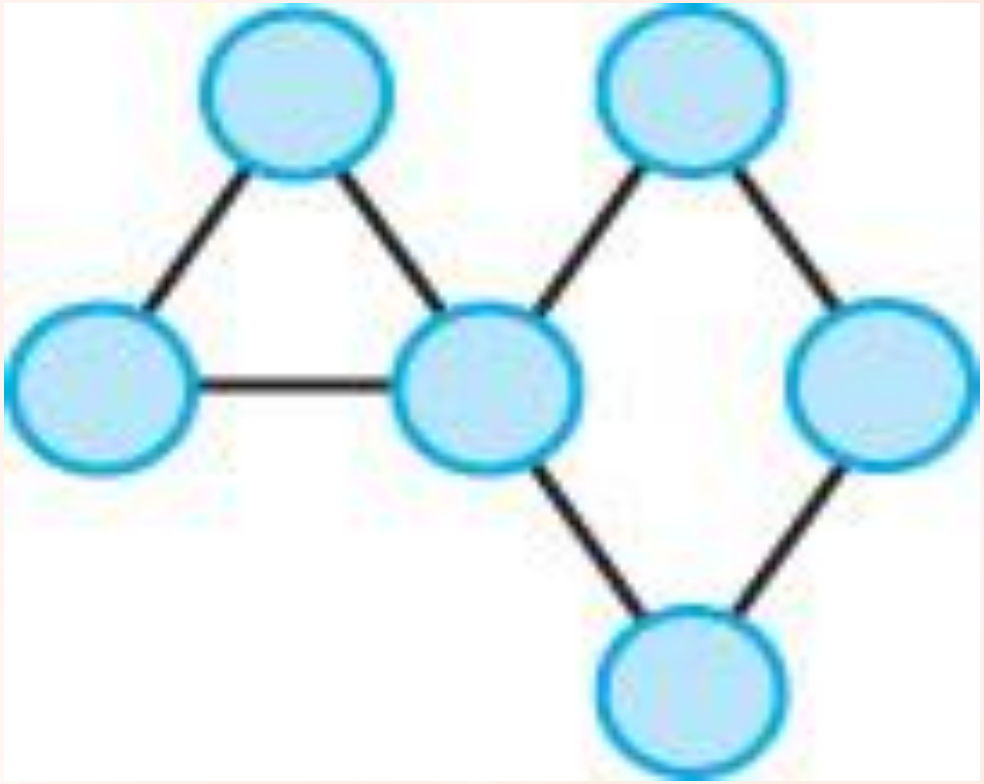
The Graph ADT



Chapter 10: The Graph ADT

10.1 – Introduction to Graphs
10.2 – The Graph Interface
10.3 – Implementations of Graphs
10.4 – Application: Graph Traversals
10.5 – Application: The Single-Source Shortest-Paths Problem

10.1 Introduction to Graphs



Definitions

- **Graph:** A data structure that consists of a set of vertices and a set of edges that relate the vertices to each other
- **Vertex:** A node in a graph
- **Edge (arc):** A pair of vertices representing a connection between the two vertices in a graph
- **Undirected graph:** A graph in which the edges have no direction
- **Directed graph (digraph):** A graph in which each edge is directed from one vertex to another (or the same) vertex

Formally

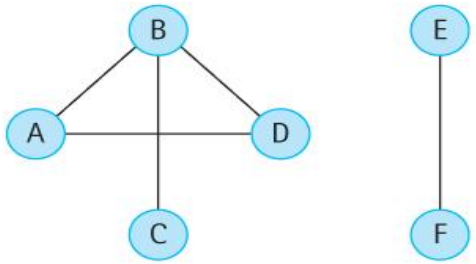
- a graph G is defined as follows:

  G = (V,E)

  where

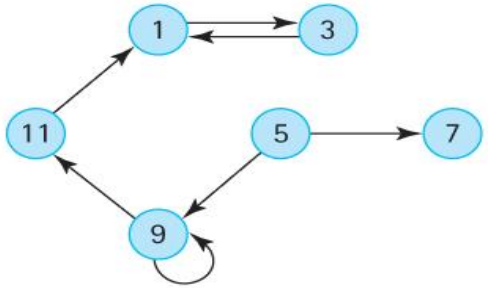
  V(G) is a finite, nonempty set of vertices
  E(G) is a set of edges (written as pairs of vertices)

(a) Graph1 is an undirected graph



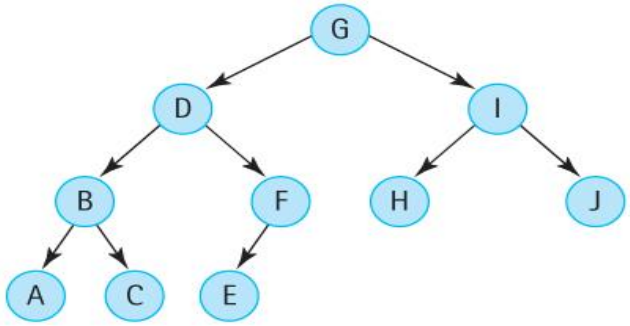
V(Graph1) = {A, B, C, D, E, F}
E(Graph1) = {(A, B), (A, D), (B, C), (B, D), (E, F)}

(b) Graph2 is a directed graph



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

(c) Graph3 is a directed graph

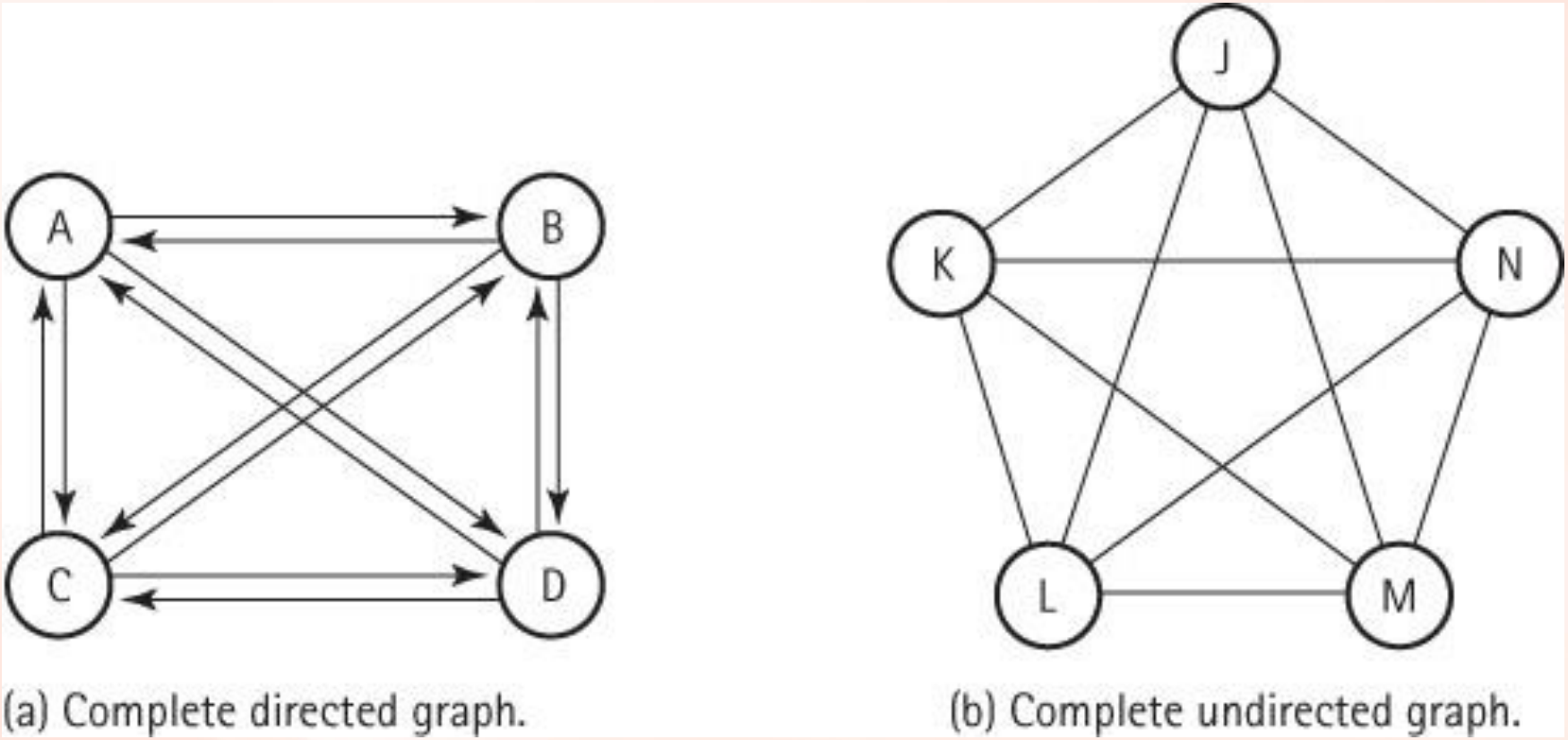


V(Graph3) = {A, B, C, D, E, F, G, H, I, J}
E(Graph3) = {(G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

More Definitions

- **Adjacent vertices:** Two vertices in a graph that are connected by an edge
- **Path:** A sequence of vertices that connects two vertices in a graph
- **Complete graph:** A graph in which every vertex is directly connected to every other vertex
- **Weighted graph:** A graph in which each edge carries a value

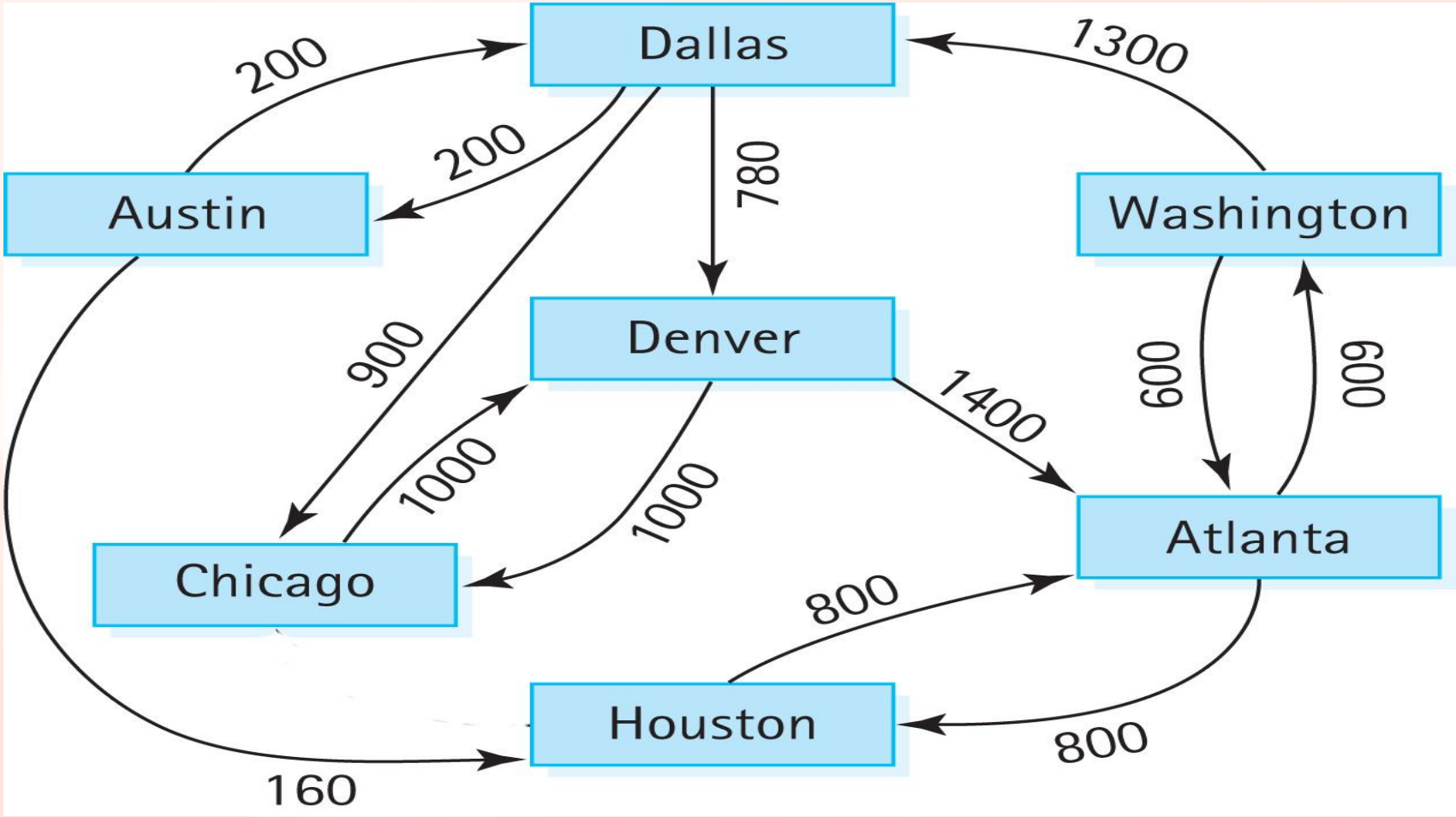
Two complete graphs



(a) Complete directed graph.

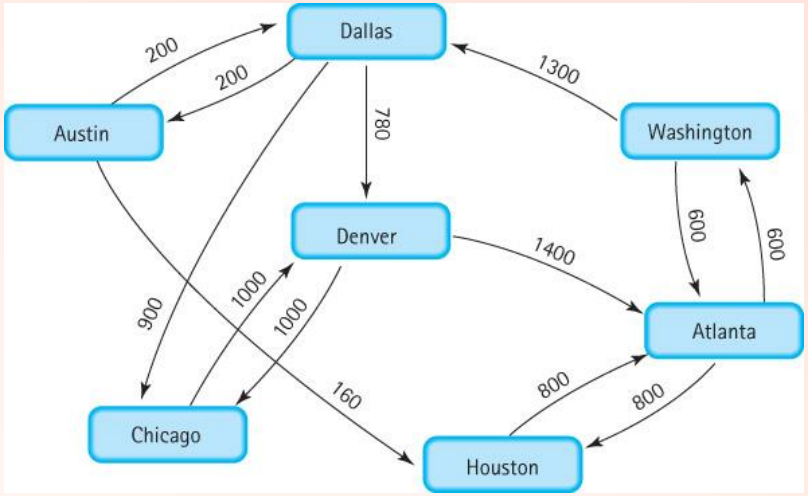
(b) Complete undirected graph.

A weighted graph



10.2 The Graph Interface

- What kind of questions might we ask about a graph?
  - Does a path exist between vertex A and vertex D? Can we fly from Atlanta to Detroit?
  - What is the total weight along this path from A to D? How much does it cost to fly from Atlanta to Detroit? What is the total distance?
  - What is the shortest path from A to D? What is the cheapest way to get from Atlanta to Detroit?
  - If I start at vertex A, where can I go? What cities are accessible if I start in Atlanta?
  - How many connected components are in the graph? What groups of cities are connected to each other?



Graph Operations

- What kind of operations are defined on a graph?
  - We specify and implement a small set of useful graph operations
  - Many other operations on graphs can be defined; we have chosen operations that are useful when building applications to answer typical questions, such as those found on the previous slide

WeightedGraphInterface.java part I

```
//-----------------------------------------------------------------------
// WeightedGraphInterface.java       by Dale/Joyce/Weems            Chapter 10
//
// Interface for classes that implement a directed graph with weighted edges.
// Vertices are objects of class T and can be marked as having been visited.
// Edge weights are integers.
// Equivalence of vertices is determined by the vertices' equals method.
//
// General precondition: except for the addVertex and hasVertex methods,
// any vertex passed as an argument to a method is in this graph.
//-----------------------------------------------------------------------

package ch10.graphs;

import ch04.queues.*;

public interface WeightedGraphInterface<T>
{
  boolean isEmpty();
  // Returns true if this graph is empty; otherwise, returns false.

  boolean isFull();
  // Returns true if this graph is full; otherwise, returns false.
```

WeightedGraphInterface.java part II

```
void addVertex(T vertex);
// Preconditions:   This graph is not full.
//                  vertex is not already in this graph.
//                  vertex is not null.
//
// Adds vertex to this graph.

boolean hasVertex(T vertex);
// Returns true if this graph contains vertex; otherwise, returns false.

void addEdge(T fromVertex, T toVertex, int weight);
// Adds an edge with the specified weight from fromVertex to toVertex.

int weightIs(T fromVertex, T toVertex);
// If edge from fromVertex to toVertex exists, returns the weight of edge;
// otherwise, returns a special "null-edge" value.
```

WeightedGraphInterface.java part III

```
 UnboundedQueueInterface<T> getToVertices(T vertex);
  // Returns a queue of the vertices that vertex is adjacent to.

  void clearMarks();
  // Unmarks all vertices.

  void markVertex(T vertex);
  // Marks vertex.

  boolean isMarked(T vertex);
  // Returns true if vertex is marked; otherwise, returns false.

  T getUnmarked();
  // Returns an unmarked vertex if any exist; otherwise, returns null.
}
```

10.3 Implementations of Graphs

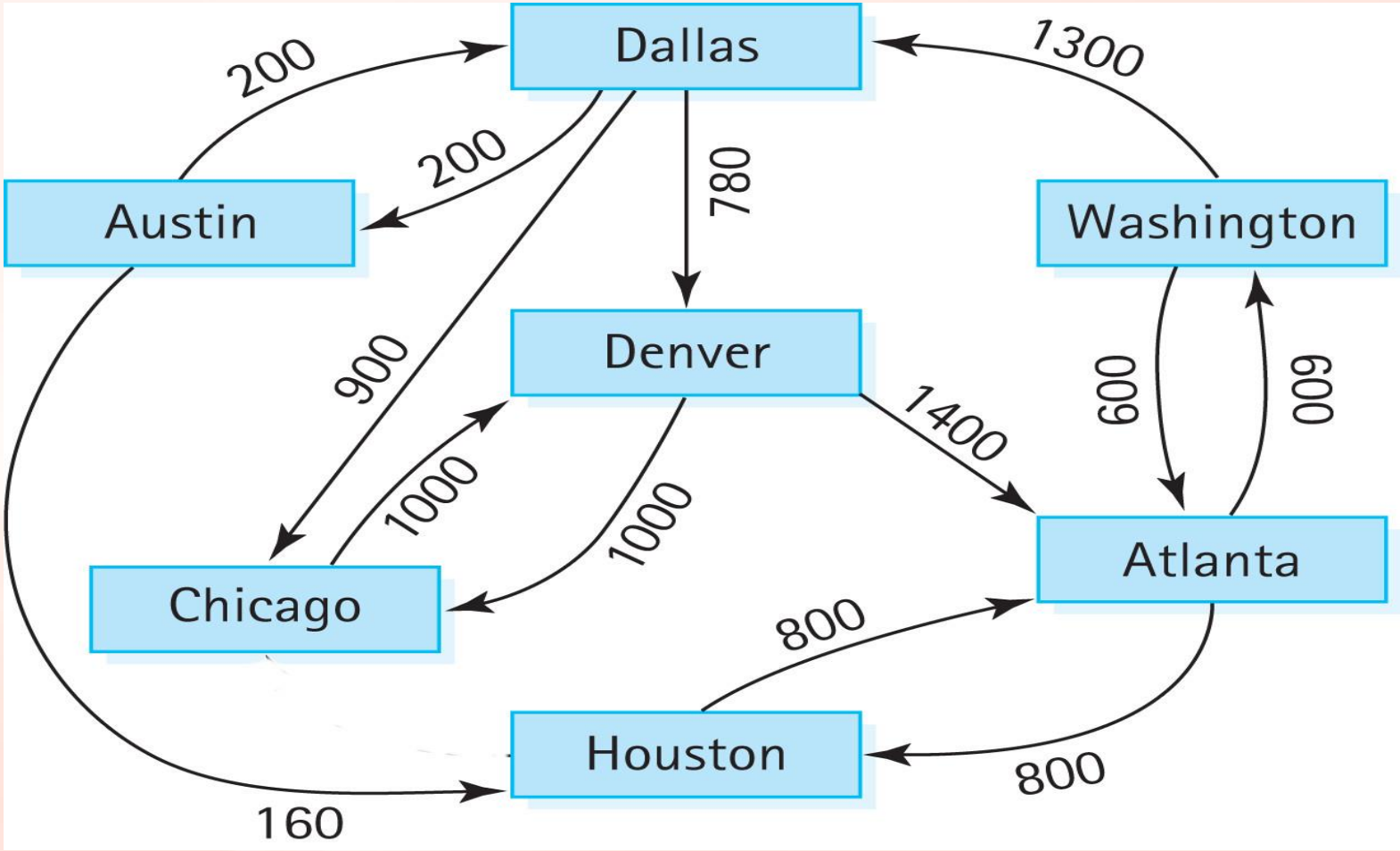
- In this section we introduce two graph implementation approaches
  - an array based approach
  - a linked approach

Array-Based Implementation

- **Adjacency matrix** For a graph with N nodes, an N by N table that shows the existence (and weights) of all edges in the graph

- With this approach a graph consists of
  - an integer variable `numVertices`
  - a one-dimensional array `vertices`
  - a two-dimensional array `edges` (the adjacency matrix)

A repeat of the abstract model



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)

WeightedGraph.java
instance variables

```
package ch10.graphs;

import ch04.queues.*;

public class WeightedGraph<T> implements WeightedGraphInterface<T>
{
  public static final int NULL_EDGE = 0;
  private static final int DEFCAP = 50;  // default capacity
  private int numVertices;
  private int maxVertices;
  private T[] vertices;
  private int[][] edges;
  private boolean[] marks;  // marks[i] is mark for vertices[i]

    . . .
```

WeightedGraph.java
Constructors

```
public WeightedGraph()
  // Instantiates a graph with capacity DEFCAP vertices.
  {
    numVertices = 0;
    maxVertices = DEFCAP;
    vertices = (T[ ]) new Object[DEFCAP];
    marks = new boolean[DEFCAP];
    edges = new int[DEFCAP][DEFCAP];
  }

  public WeightedGraph(int maxV)
  // Instantiates a graph with capacity maxV.
  {
    numVertices = 0;
    maxVertices = maxV;
    vertices = (T[ ]) new Object[maxV];
    marks = new boolean[maxV];
    edges = new int[maxV][maxV];
  }
...
```

Adding a vertex

```
public void addVertex(T vertex)
// Preconditions:   This graph is not full.
//                  Vertex is not already in this graph.
//                  Vertex is not null.
//
// Adds vertex to this graph.
{
  vertices[numVertices] = vertex;
  for (int index = 0; index < numVertices; index++)
  {
    edges[numVertices][index] = NULL_EDGE;
    edges[index][numVertices] = NULL_EDGE;
  }
  numVertices++;
}
```

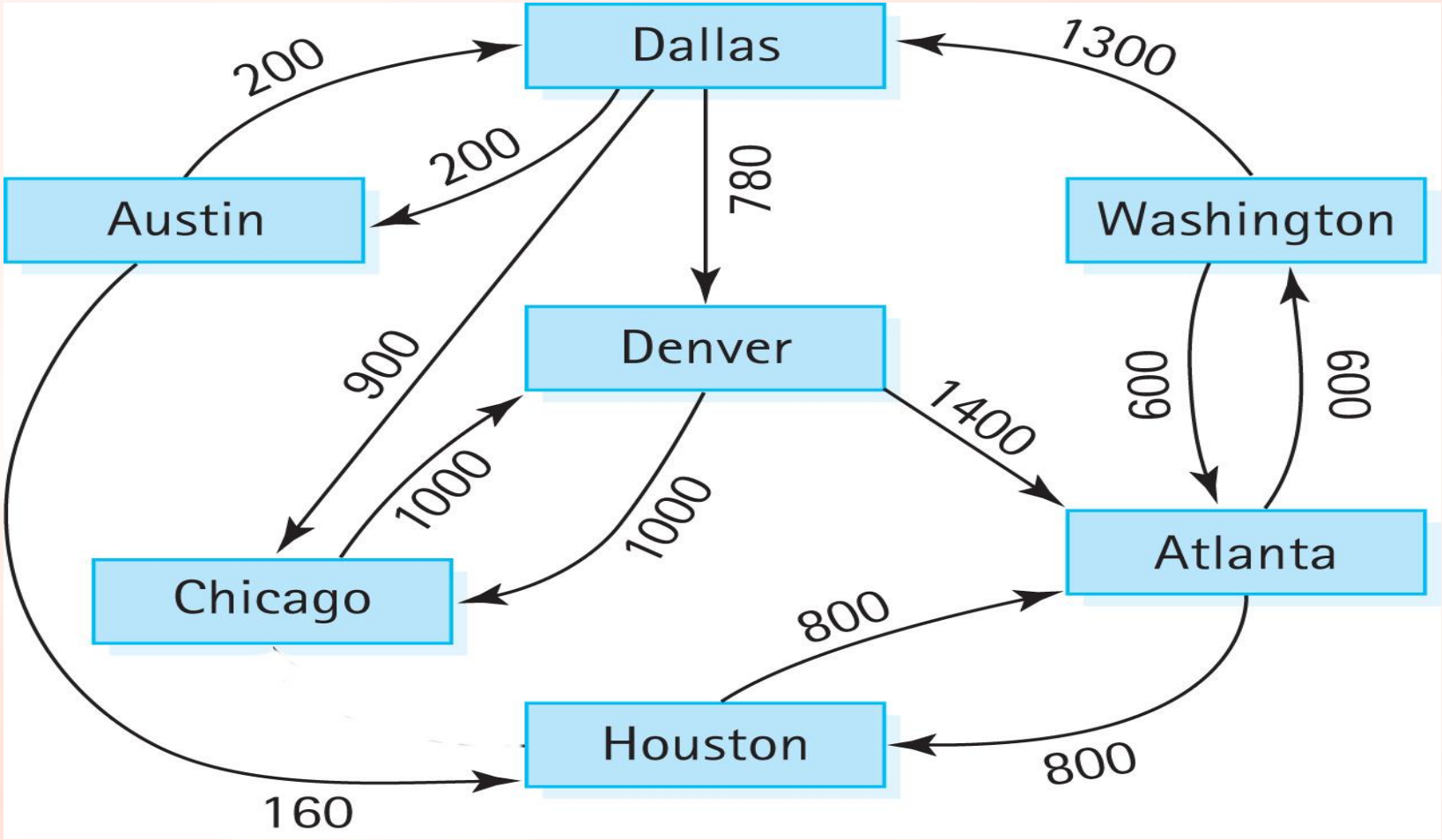
Textbook also includes code for `indexIs`, `addEdge`, `weightIs`, and `getToVertices`.

Coding the remaining methods is left as an exercise.

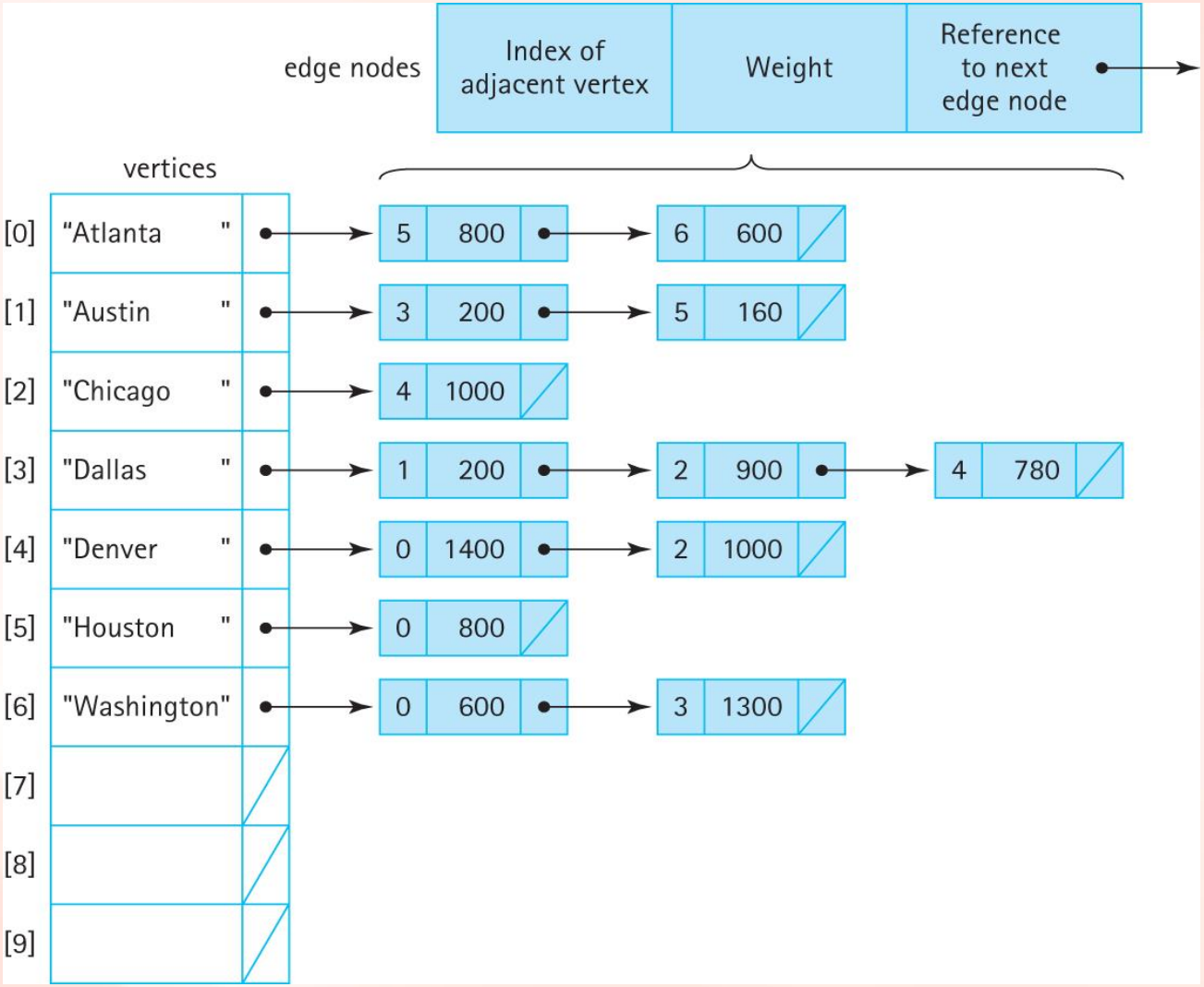
Linked Implementation

- **Adjacency list** A linked list that identifies all the vertices to which a particular vertex is connected; each vertex has its own adjacency list
- We look at two alternate approaches:
  - use an array of vertices that each contain a reference to a linked list of nodes
  - use a linked list of vertices that each contain a reference to a linked list of nodes

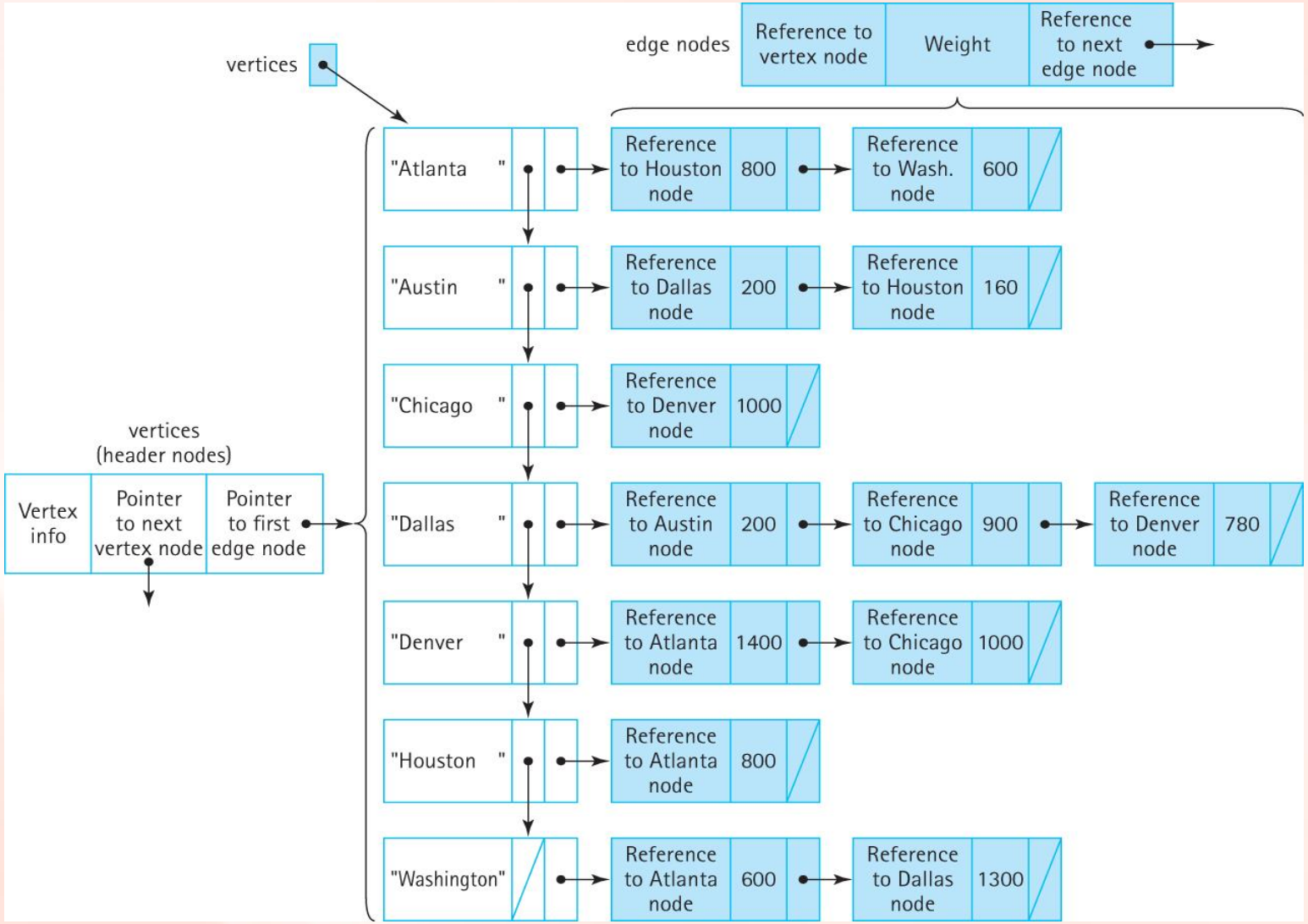
A repeat of the abstract model



The first link-based implementation



The second link-based implementation



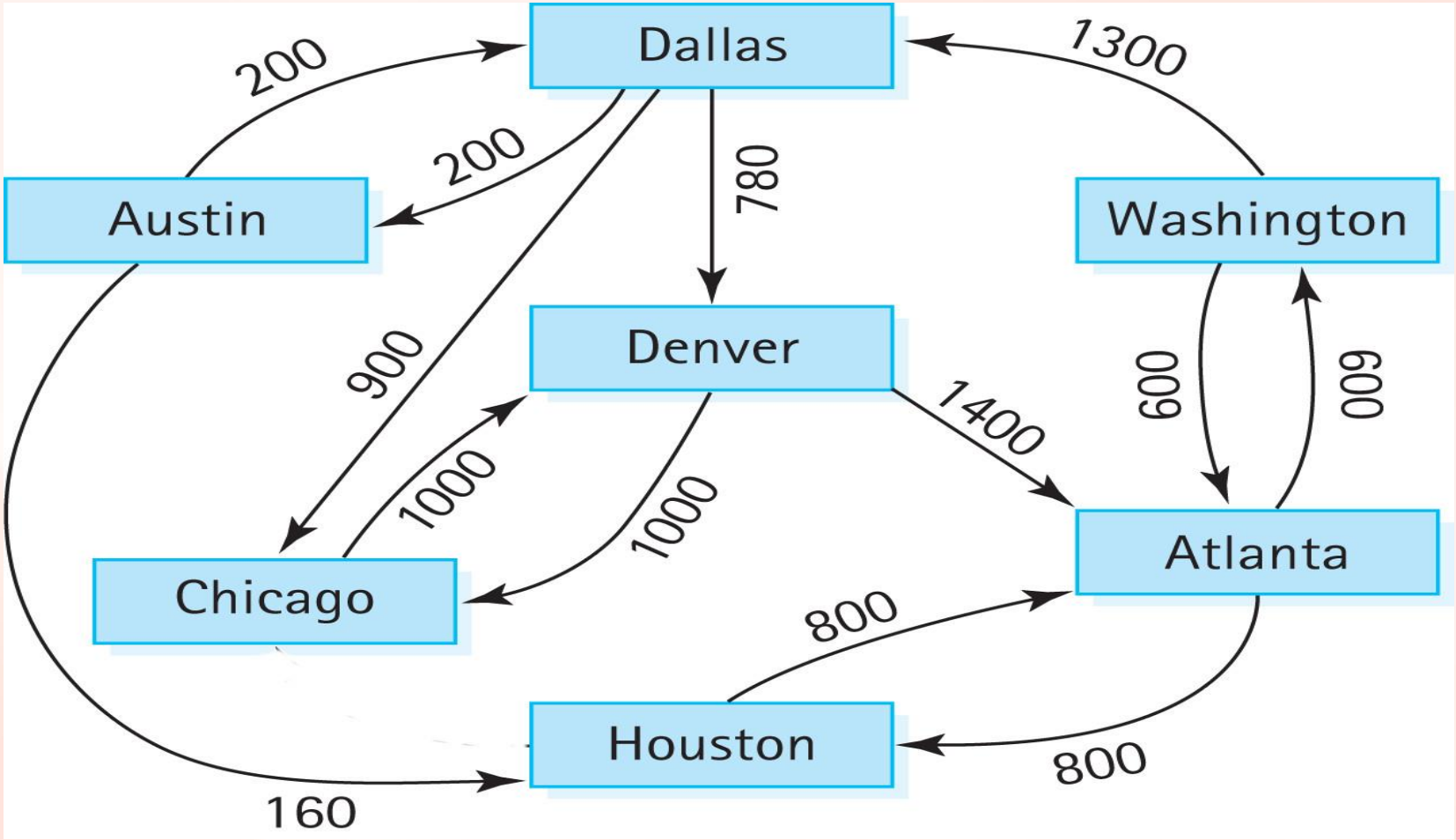
10.4 Application: Graph Traversals

- Our graph specification does not include traversal operations.
- We treat traversal as a graph application rather than an innate operation.
- The basic operations given in our specification allow us to implement different traversals *independent* of how the graph itself is actually implemented.
- The application `UseGraph` in the `ch10.apps` package contains the code for all the algorithms presented in Sections 10.4 and 10.5

Graph Traversal

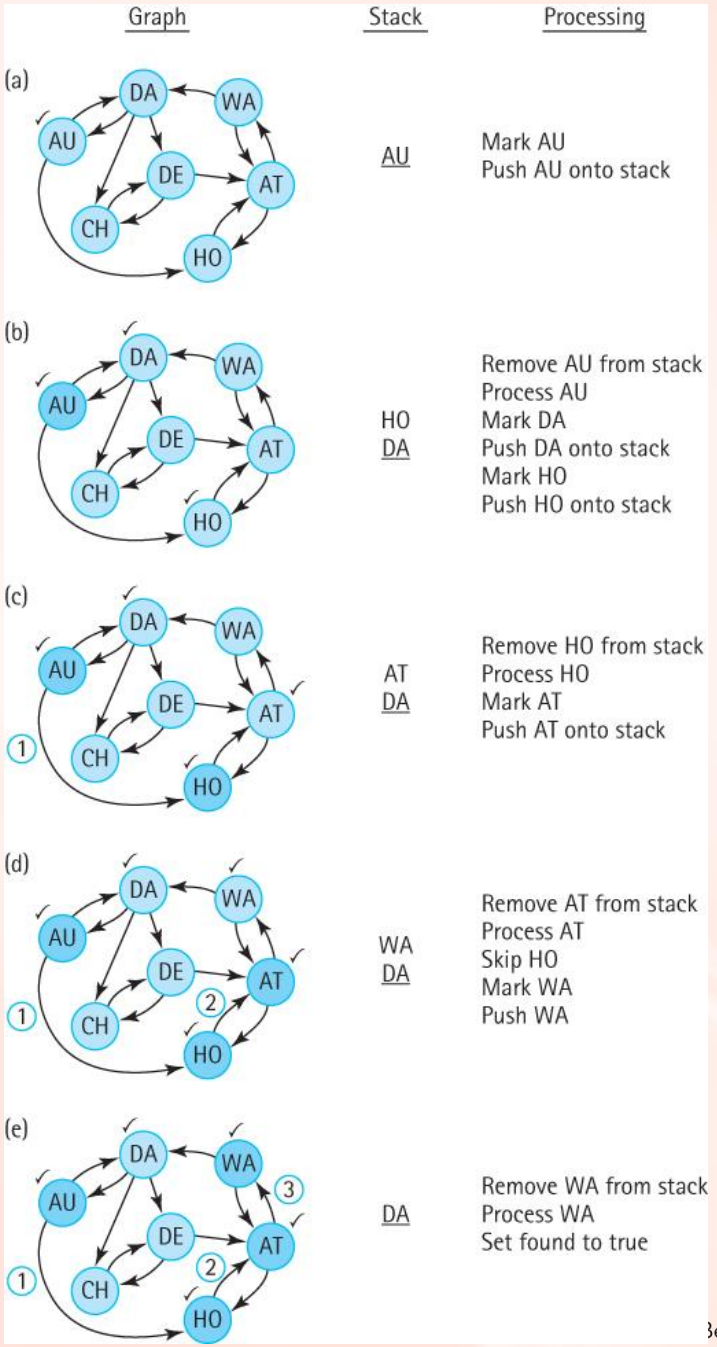
- As we did for general trees, we look at two types of graph traversal:
  - The strategy of going as far as we can and then backtracking is called a *depth-first* strategy.
  - The strategy of fanning out "level by level" is called a *breadth-first* strategy.
- We discuss algorithms for employing both strategies within the context of determining if two cities are connected in our airline example.

Can we get from Austin to Washington?



Algorithm: IsPathDF (startVertex, endVertex): returns boolean

```
Set found to false
Clear all marks
Mark the startVertex
Push the startVertex onto the stack
do
  Set current vertex = stack.top()
  stack.pop()
  if current vertex equals endVertex
     Set found to true
  else
    for each adjacent vertex
       if adjacent vertex is not marked
          Mark the adjacent vertex and
          Push it onto the stack
while !stack.isEmpty() AND !found
return found
```



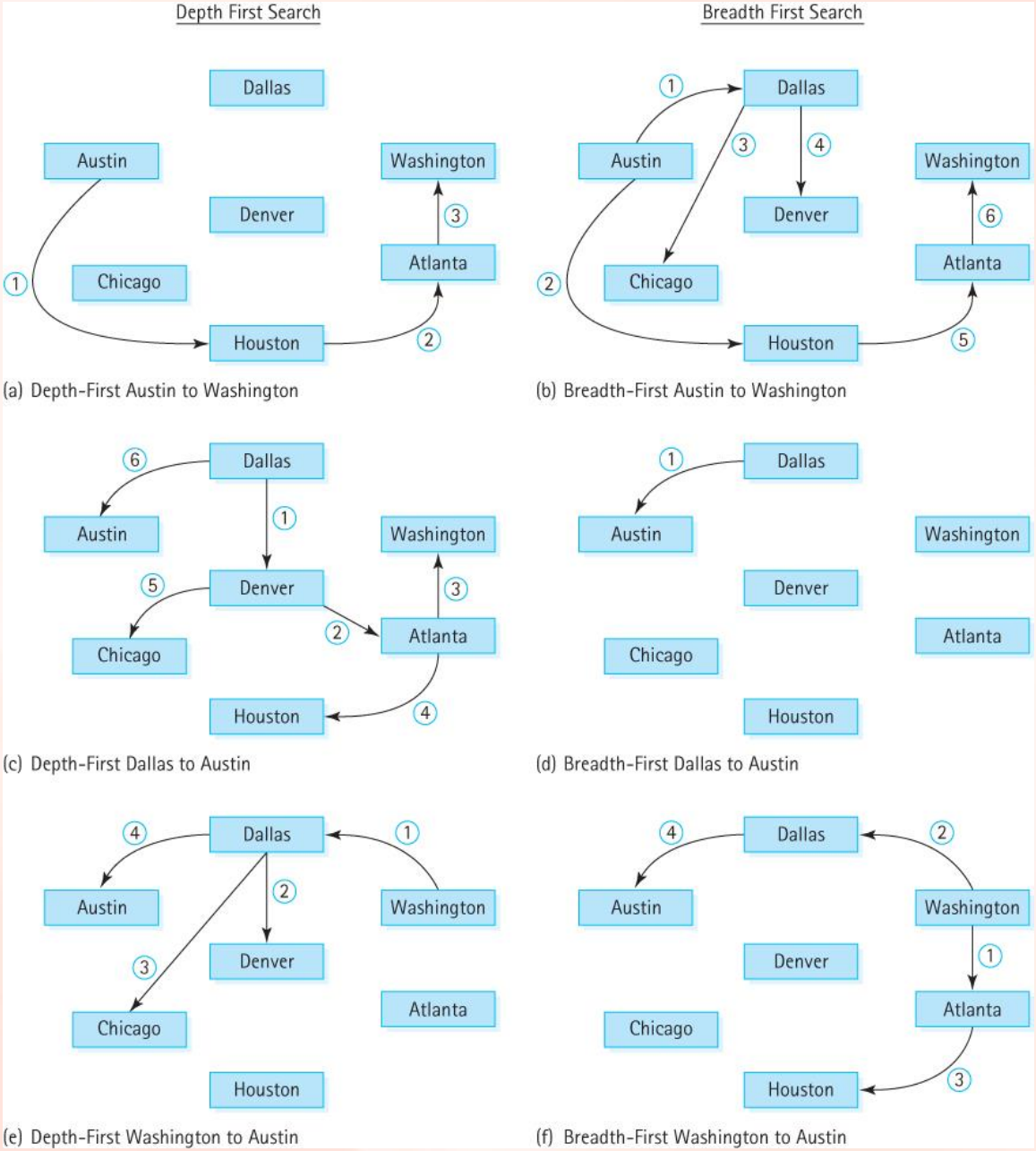

```
Set found to false
Clear all marks
Mark the startVertex
Push the startVertex onto the stack
do
  Set current vertex = stack.top()
  stack.pop()
  if current vertex equals endVertex
     Set found to true
  else
    for each adjacent vertex
       if adjacent vertex is not marked
          Mark the adjacent vertex and
          Push it onto the stack
while !stack.isEmpty() AND !found
return found
```

Breadth first search – use queue

IsPathBF (startVertex, endVertex): returns boolean

```
Set found to false
Clear all marks
Mark the startVertex
Enqueue the startVertex into the queue
do
    Set current vertex = queue.dequeue()
    if current vertex equals endVertex
        Set found to true
    else
    for each adjacent vertex
        if adjacent vertex is not marked
            Mark the adjacent vertex and
            Enqueue it into the queue
while !queue.isEmpty() AND !found
return found
```

Examples of search paths



10.5 Application: The Single-Source Shortest-Paths Problem

- An algorithm that displays the shortest path from a designated starting city to *every other* city in the graph
- In our example graph if the starting point is Washington we should get

```
Last Vertex   Destination   Distance
-----------------------------------
Washington    Washington    0
Washington    Atlanta       600
Washington    Dallas        1300
Atlanta       Houston       1400
Dallas        Austin        1500
Dallas        Denver        2080
Dallas        Chicago       2200
```

An erroneous approach

shortestPaths(graph, startVertex)

```
graph.ClearMarks( )
Create flight(startVertex, startVertex, 0)
pq.enqueue(flight)   // pq is a priority queue
do
    flight = pq.dequeue( )
    if flight.getToVertex() is not marked
        Mark flight.getToVertex()
        Write flight.getFromVertex, flight.getToVertex, flight.getDistance
        flight.setFromVertex(flight.getToVertex())
        Set minDistance to flight.getDistance()
        Get queue vertexQueue of vertices adjacent from flight.getFromVertex()
        while more vertices in vertexQueue
            Get next vertex from vertexQueue
            if vertex not marked
                flight.setToVertex(vertex)
                flight.setDistance(minDistance + graph.weightIs(flight.getFromVertex(), vertex))
                pq.enqueue(flight)
while !pq.isEmpty( )
```

Notes

- The algorithm for the shortest-path traversal is similar to those we used for the depth-first and breadth-first searches, but there are three major differences:
  - We use a priority queue rather than a FIFO queue or stack
  - We stop only when there are no more cities to process; there is no destination
  - It is incorrect if we use a reference-based priority queue improperly!

The Incorrect Part of the Algorithm

```
while more vertices in vertexQueue
   Get next vertex from vertexQueue
   if vertex not marked
      flight.setToVertex(vertex)
      flight.setDistance(minDistance + graph.weightIs(flight.getFromVertex(), vertex))
      pq.enqueue(flight)
```

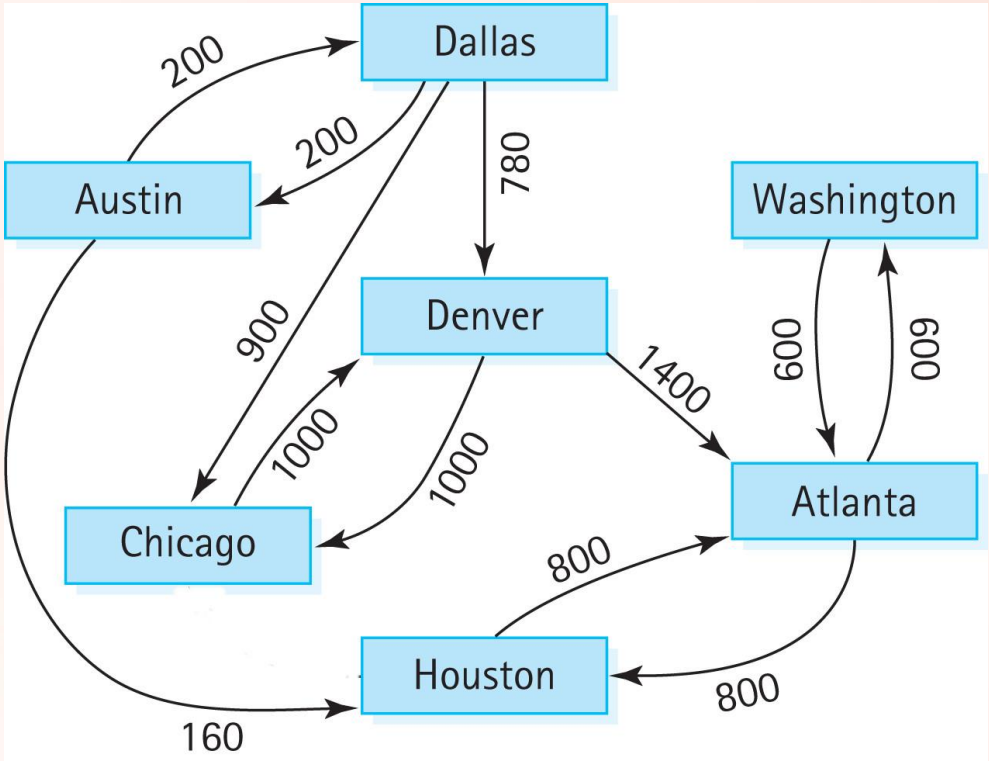
- This part of the algorithm walks through the queue of vertices adjacent to the current vertex, and enqueues `Flight` objects onto the priority queue `pq` based on the information.
- The flight variable is actually a reference to a `Flight` object.
- Suppose the queue of adjacent vertices has information in it related to the cities Atlanta and Houston.
- The first time through this loop we insert information related to Atlanta in `flight` and enqueue it in `pq`.
- But the next time through the loop we make changes to the `Flight` object referenced by `flight`. We are inadvertently reaching into the priority queue and changing one of its entries.

Correcting the Algorithm

```
while more vertices in vertexQueue
   Get next vertex from vertexQueue
   if vertex not marked
      Set newDistance to minDistance + graph.weightIs(flight.getFromVertex(), vertex)
      Create newFlight(flight.getFromVertex(), vertex, newDistance)
      pq.enqueue(newFlight)
```

The application `UseGraph` in the `ch10.apps` package contains the code for all the algorithms presented in Sections 10.4 and 10.5

Unreachable Vertices



With this new graph we cannot fly from Washington to Austin, Chicago, Dallas, or Denver

To print unreachable vertices

- Append the following to the `shortestPaths` method:

```
System.out.println("The unreachable vertices are:");
vertex = graph.getUnmarked();
while (vertex != null)
{
  System.out.println(vertex);
  graph.markVertex(vertex);
  vertex = graph.getUnmarked();
}
```