Chapter 9
Priority Queues, Heaps, and Graphs

Chapter 8 Homework Solutions

8.

a. Q, K, and M
b. B, D, J, M, P, and N
c. 8
d. 16
e BDJKMNPQRTWY
f BJDNPMKRWYTQ
g QKDBJMPNTRYW

Chapter 8 Homework Solutions

44 a.

nodes	.info	.left	.right
0	Q	1	2
1	L	3	4
2	W	5	NUL
3	F	NUL	NUL
4	M	NUL	6
5	R	NUL	7
6	N	NUL	NUL
7	S	NUL	NUL
8	?	9	?
9	?	NUL	?
free	8		
root	0		

44 b.

nodes	.info	.left	.right
0	Q	1	2
1	L	3	4
2	W	7	NUL
3	F	NUL	8
4	M	NUL	6
5	?	9	?
6	N	NUL	NUL
7	S	NUL	NUL
8	B	NUL	NUL
9	?	NUL	?
free	5		
root	0		

Chapter 8 Homework Solutions

48.

a. e
b. b, d, e
c. b, e

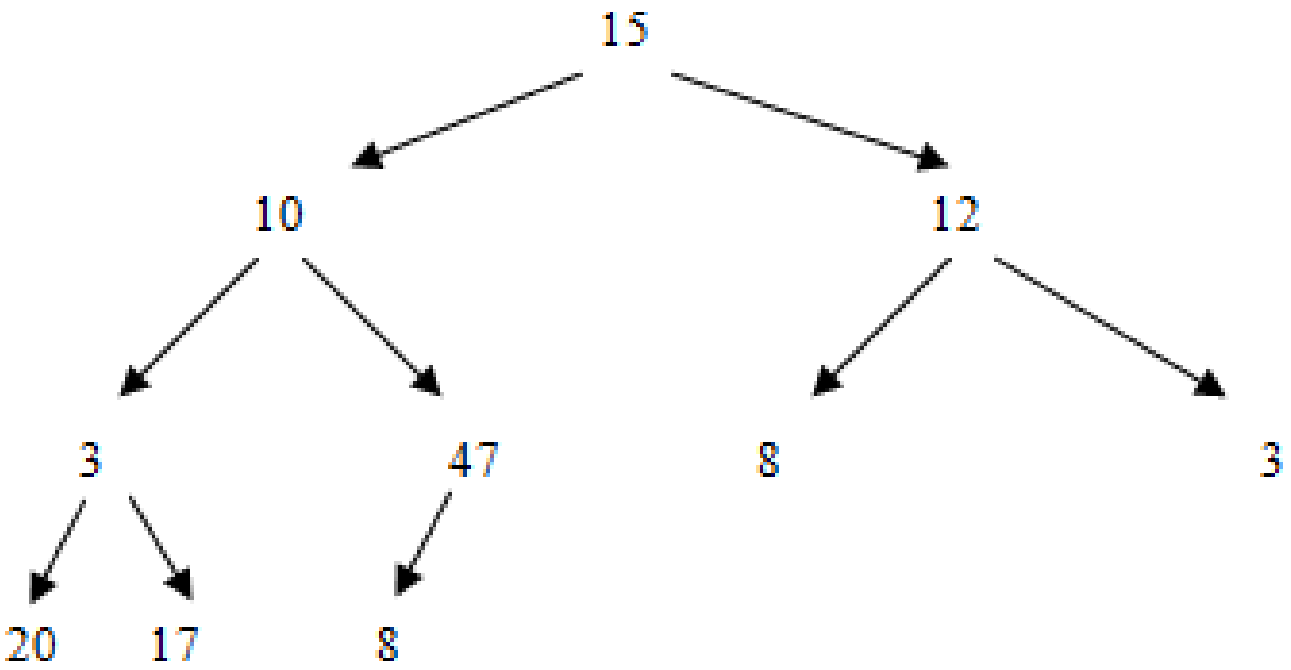
49.

a. Any negative value, for example a -1.
b.

.numElements	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	26	14	38	1	-1	33	50	-1	7	-1	-1	-1	35	44	60	-1

Chapter 8 Homework Solutions

51.



Chapter 9: Priority Queues, Heaps, and Graphs

9.1 – Priority Queues

9.2 – Heaps

9.3 – Introduction to Graphs

9.4 – Formal Specification of a Graph ADT

9.5 – Graph Applications

9.6 – Implementations of Graphs

9.1 Priority Queues

- A priority queue is an abstract data type with an interesting accessing protocol - only the *highest-priority* element can be accessed

- Priority queues are useful for any application that involves processing items by priority

Logical Level

```
//--------------------------------------------------------------------------
// PriQueueInterface.java          by Dale/Joyce/Weems                Chapter 9
//
// Interface for a class that implements a priority queue of
// Comparable Objects.
//--------------------------------------------------------------------------

package ch09.priorityQueues;

public interface PriQueueInterface<T extends Comparable<T>>
{
  // Returns true if this priority queue is empty, false otherwise.
  boolean isEmpty();

  // Returns true if this priority queue is full, false otherwise.
  boolean isFull();

  // Throws PriQOverflowException if this priority queue is full;
  // otherwise, adds element to this priority queue.
  void enqueue(T element);

  // Throws PriQUnderflowException if this priority queue is empty;
  // otherwise, removes element with highest priority from this
  // priority queue and returns it.
  T dequeue();
}
```

Implementation Level

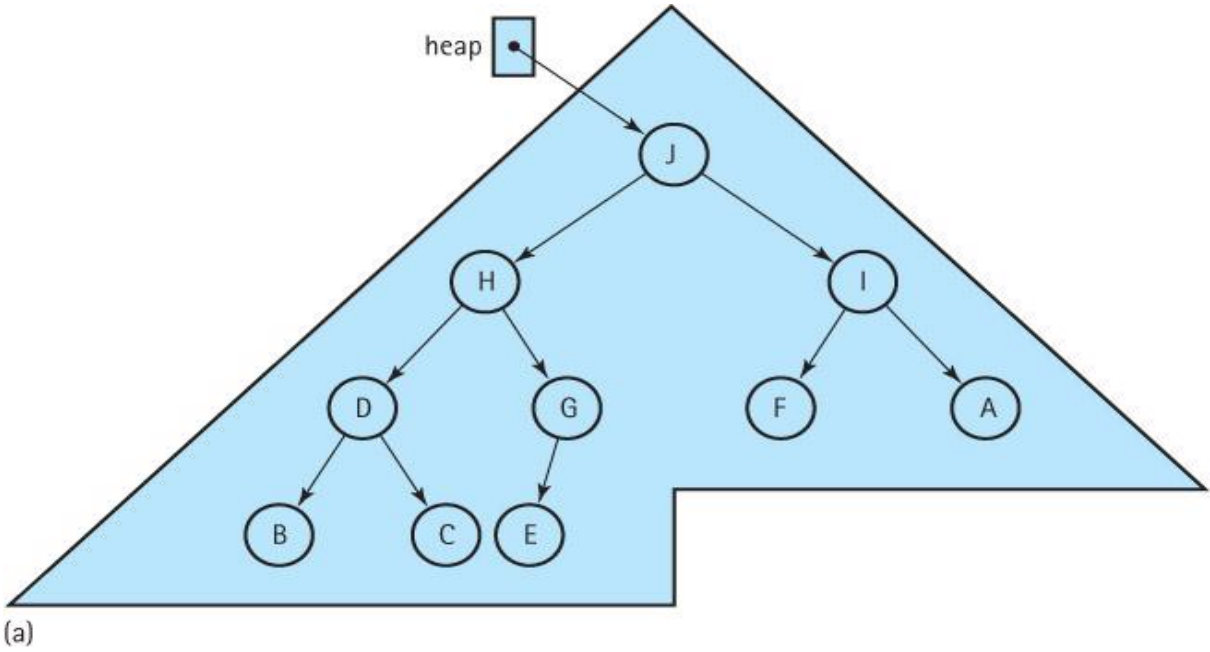
What ADT could we use to implement the priority queue?

- **An Unsorted List**
  - dequeuing would require searching through the entire list
- **An Array-Based Sorted List**
  - Enqueuing is expensive
- **A Reference-Based Sorted List**
  - Enqueuing again is 0(N)
- **A Binary Search Tree**
  - On average, $0(\log_2 N)$ steps for both enqueue and dequeue
- **A Heap**
  - guarantees $0(\log_2 N)$ steps, even in the worst case
  - So… guess what's next?

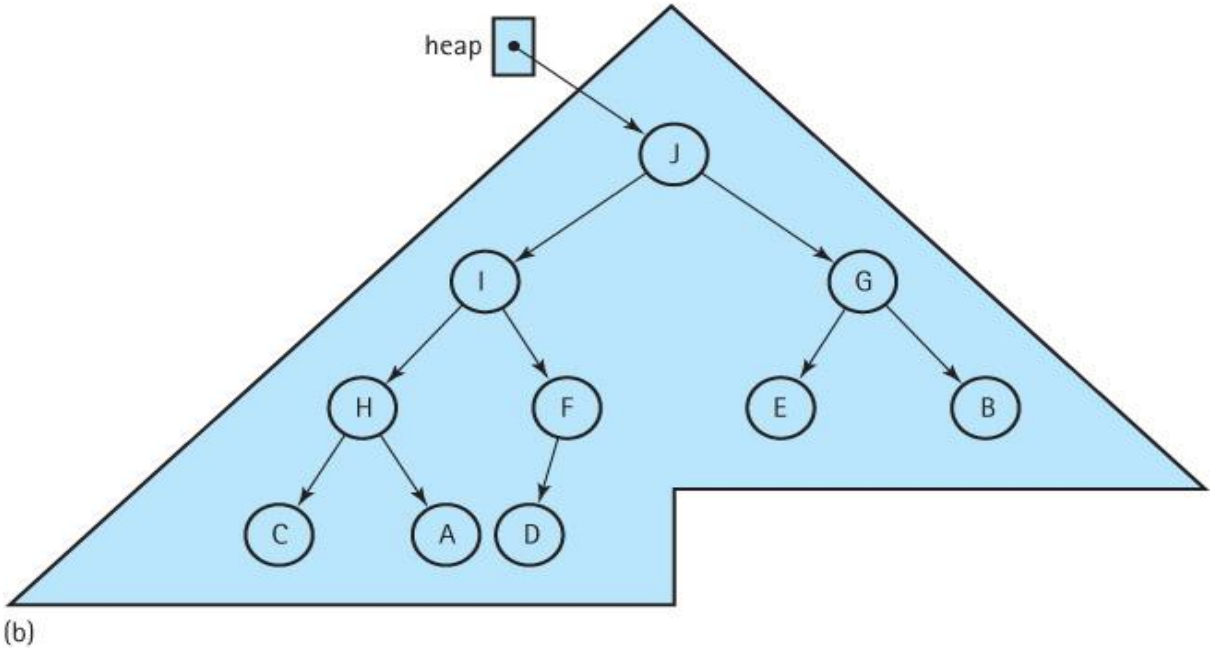
9.2 Heaps

- **Heap**
  - An implementation of a Priority Queue
  - Based on a binary tree satisfying two properties
    - the *shape property*:
      - the tree must be a complete binary tree
    - the *order property:*
      - the value of parent node ≥ value of both of its children

Two Heaps

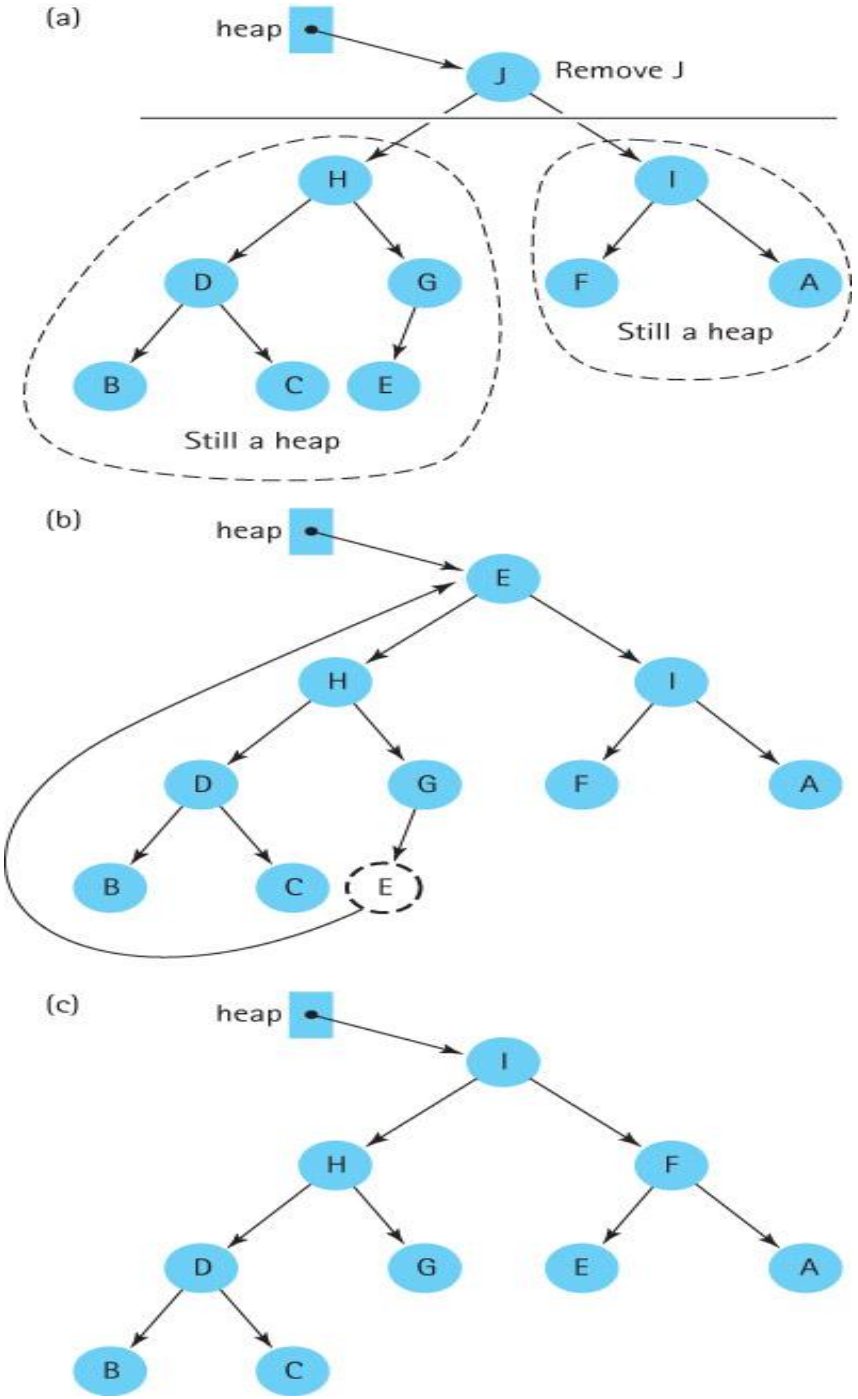


(a)



(b)

The dequeue operation

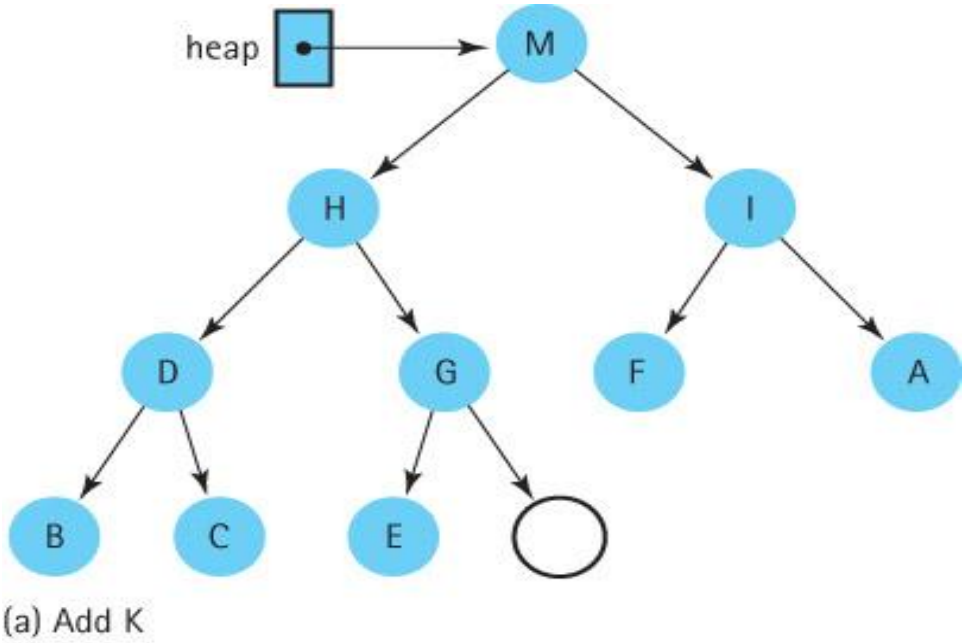


reheapDown (element)
Effect: Adds element to the heap.
Precondition: The root of the tree is empty.

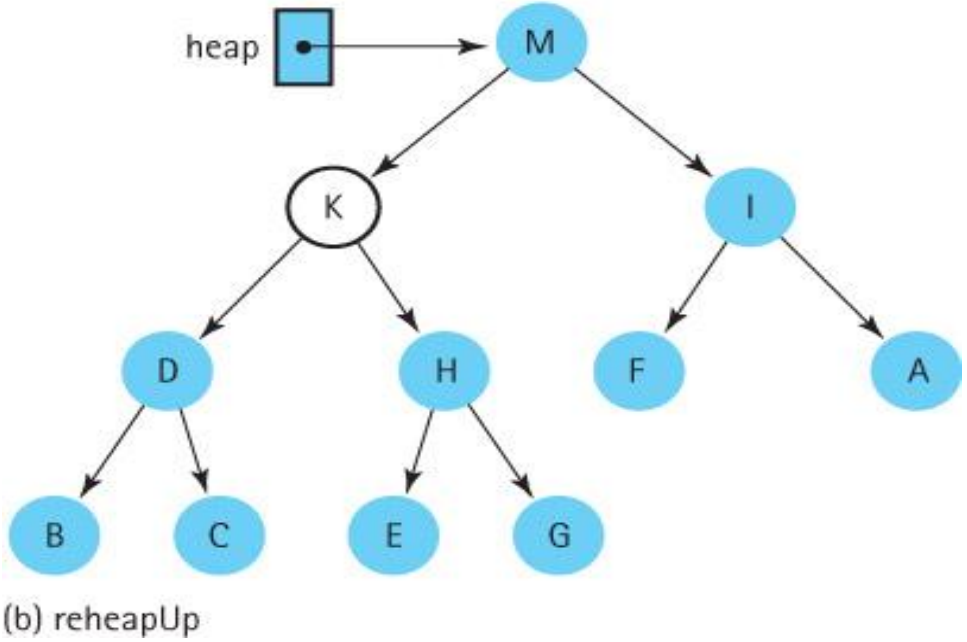
If node 'I' were removed which node would take its place?

C

The enqueue operation



(a) Add K

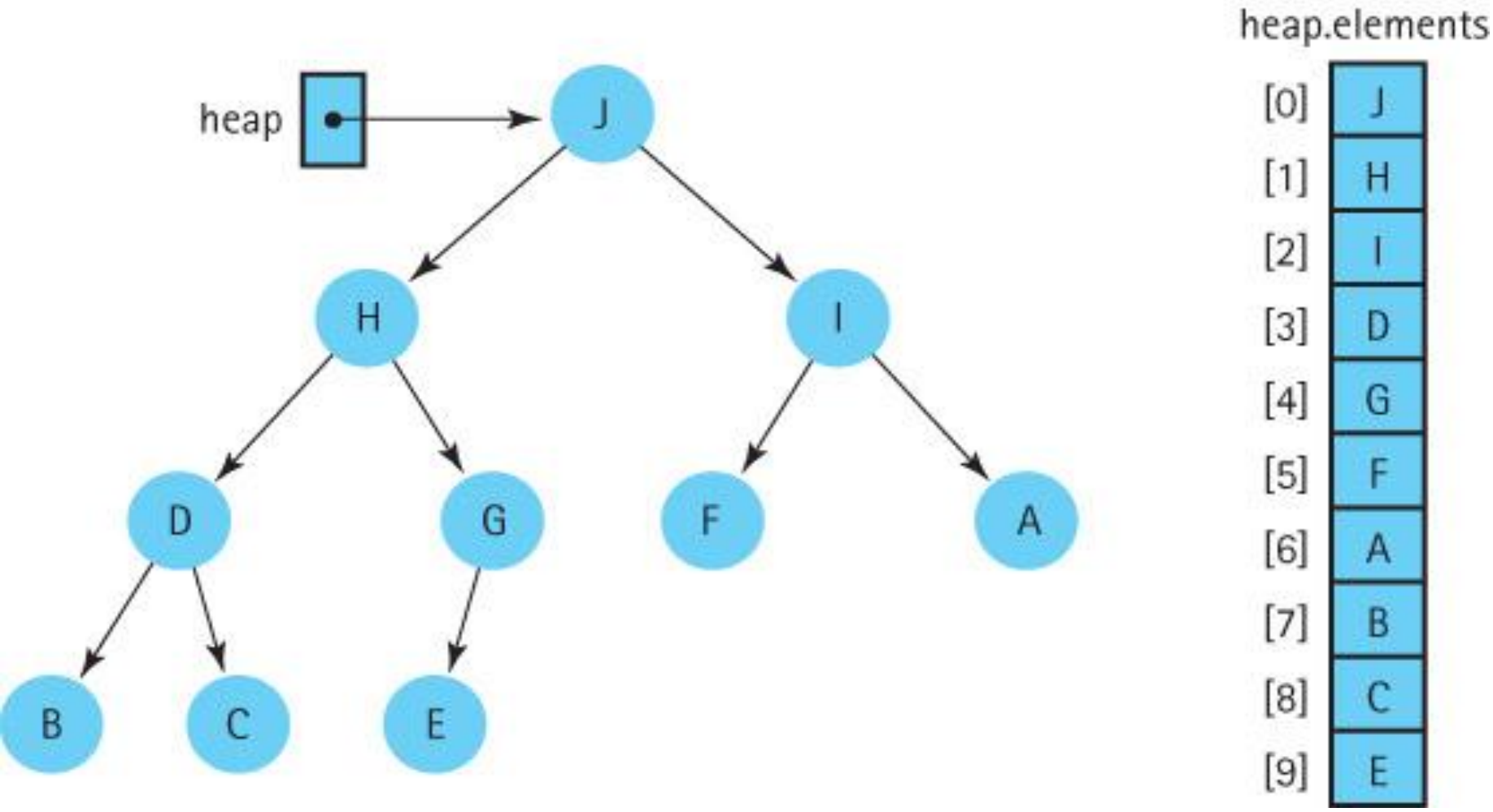


(b) reheapUp

reheapUp (element)
Effect: Adds element to the heap.
Precondition: The last index position of the tree is empty.

What happens if we add node ‘N’?

Heap Implementation



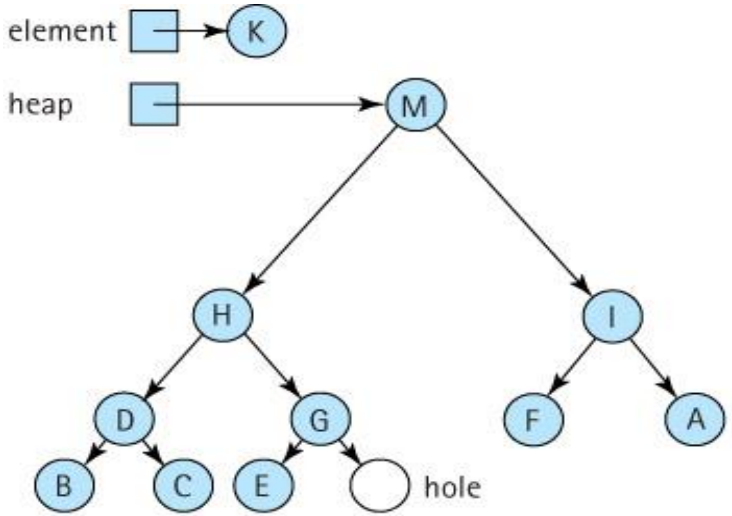
Beginning of `Heap.java`

```
package ch09.priorityQueues;

public class Heap<T extends Comparable<T>> implements PriQueueInterface<T>
{
  private ArrayList<T> elements;  // array that holds priority queue elements
  private int lastIndex;          // index of last element in priority queue
  private int maxIndex;           // index of last position in array

  public Heap(int maxSize)
  {
    elements = new ArrayList<T>(maxSize);
    lastIndex = -1;
    maxIndex = maxSize - 1;
  }

  // Returns true if this priority queue is empty, false otherwise.
  public boolean isEmpty()
    {
    return (lastIndex == -1);
  }

  // Returns true if this priority queue is full, false otherwise.
  public boolean isFull()
  {
    return (lastIndex == maxIndex);
  }
```

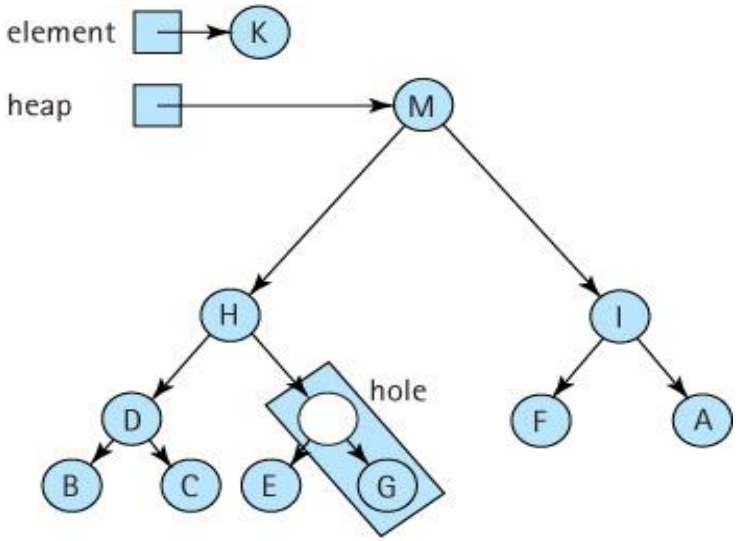
The enqueue method

```
// Throws PriQOverflowException if this priority queue is full;
// otherwise, adds element to this priority queue.
public void enqueue(T element) throws PriQOverflowException
{
  if (lastIndex == maxIndex)
    throw new PriQOverflowException("Priority queue is full");
  else
  {
    lastIndex++;
    elements.add(lastIndex, element);
    reheapUp(element);
  }
}
```

The `reheapUp` algorithm is pictured on the next slide

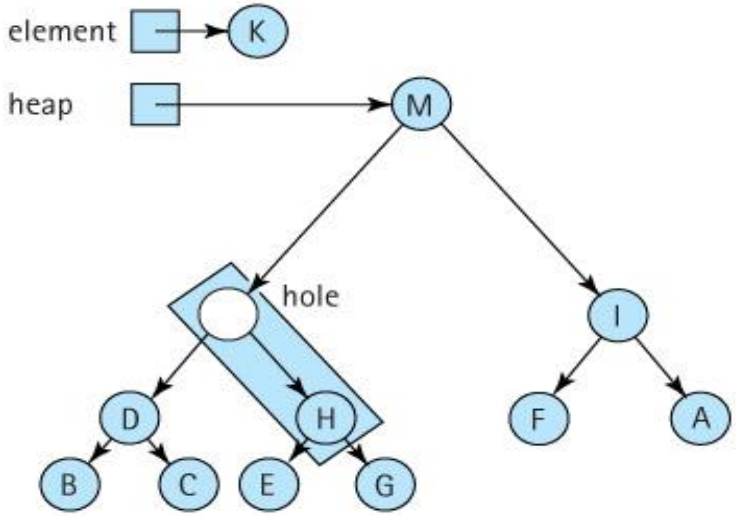


(a) Add K

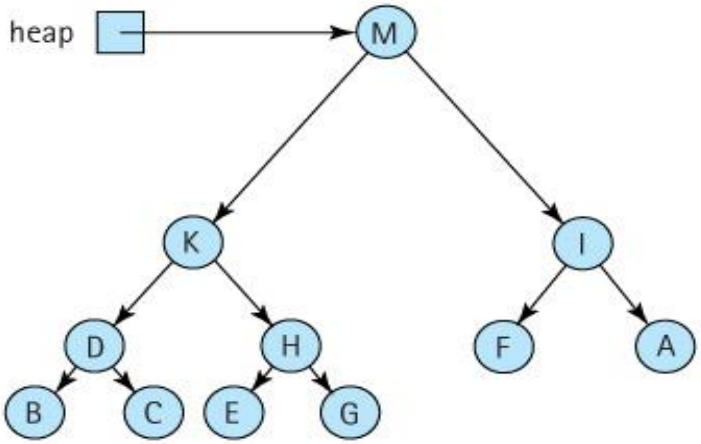


(b) Move hole up

Example: reheapUp



(c) Move hole up



(d) Place element into hole

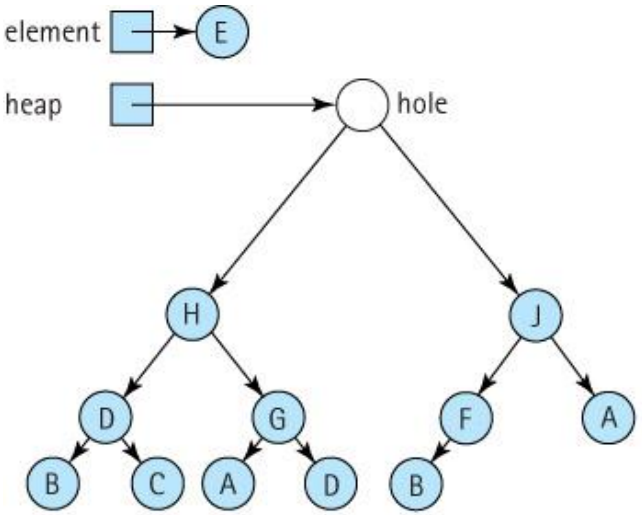
`reheapUp` operation

```
// Current lastIndex position is empty.
// Inserts element into the tree and ensures shape and order properties.
private void reheapUp(T element)
{
  int hole = lastIndex;
  while ((hole > 0)                                    // hole is not root
         &&
      (element.compareTo(elements.get((hole - 1) / 2]) > 0))
                                                       // element > hole's parent
  {
    elements.set(hole,elements.get((hole - 1) / 2));   // move hole's parent down
    hole = (hole - 1) / 2;                             // move hole up
  }
  elements.set(hole, element);                    // place element into final hole
}
```

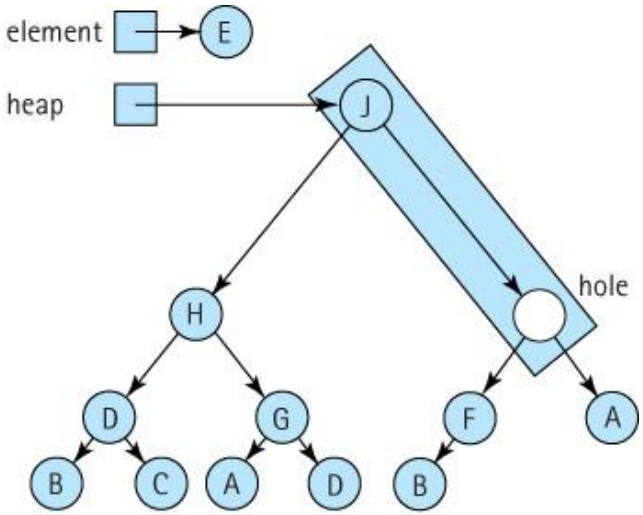
The `dequeue` method

```
// Throws PriQUnderflowException if this priority queue is empty;
// otherwise, removes element with highest priority from this
// priority queue and returns it.
public T dequeue() throws PriQUnderflowException
{
  T hold;      // element to be dequeued and returned
  T toMove;    // element to move down heap
  if (lastIndex == -1)
    throw new PriQUnderflowException("Priority queue is empty");
  else
  {
    hold = elements.get(0);              // remember element to be returned
    toMove = elements.remove(lastIndex); // element to reheap down
    lastIndex--;                         // decrease priority queue size
    if (lastIndex != -1)
       reheapDown(toMove);               // restore heap properties
    return hold;                         // return largest element
  }
}
```

The `reheapDown` algorithm is pictured on the next slide

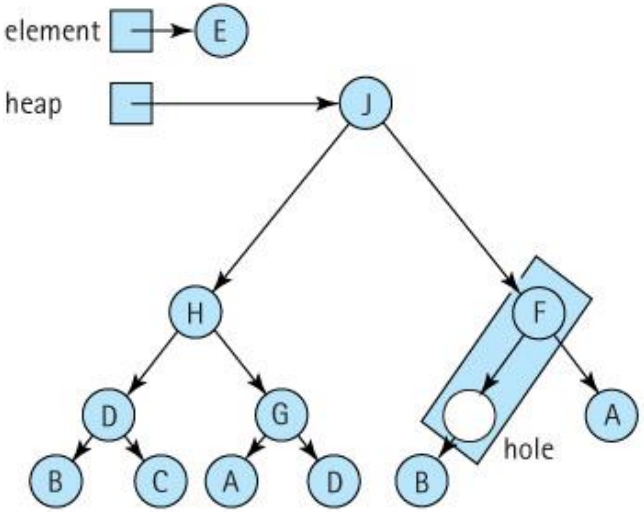


(a) reheapDown (E);

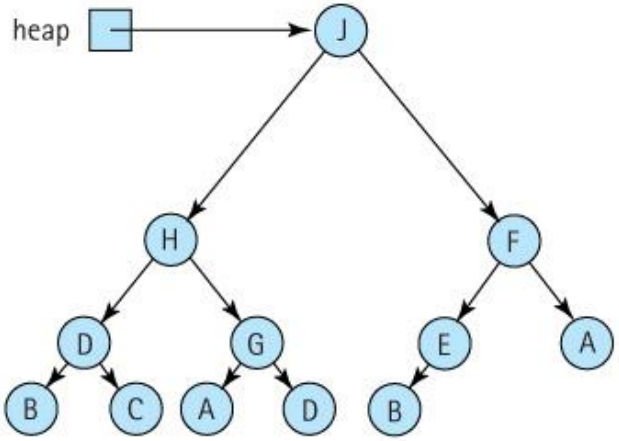


(b) Move hole down

Example: reheapDown



(c) Move hole down



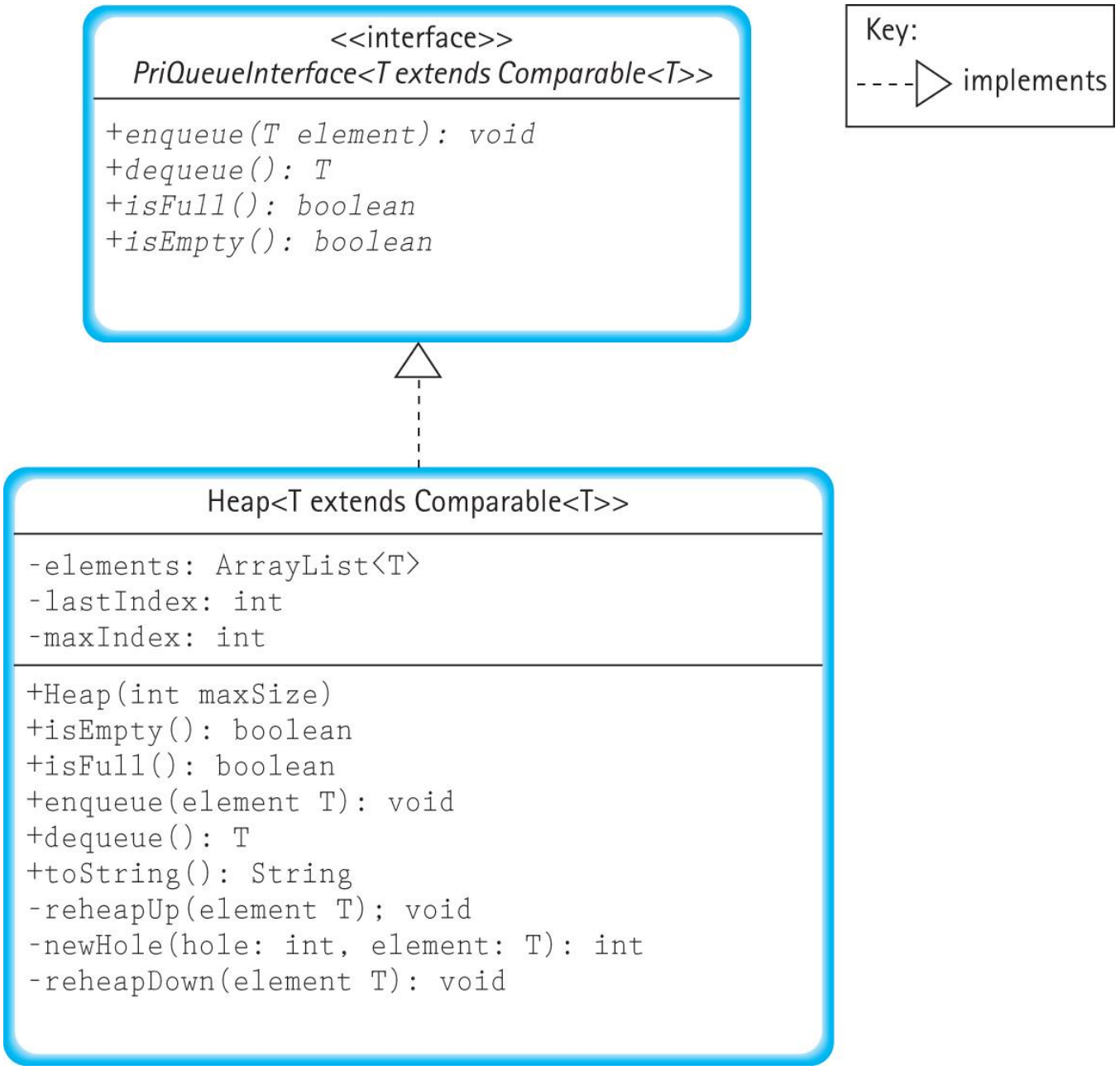
(d) Fill in final hole

`reheapDown` operation

```
// Current root position is "empty";
// Inserts element into the tree and ensures shape and order properties.
private void reheapDown(T element)
{
  int hole = 0;                              // current index of hole
  int newhole;                               // index where hole should move to
  newhole = newHole(hole, element);          // find next hole
  while (newhole != hole)
  {
    elements.set(hole, elements.get(newhole));  // move element up
    hole = newhole;                             // move hole down
    newhole = newHole(hole, element);           // find next hole
  }
  elements.set(hole, element);             // fill in the final hole
}
```

The `newHole` method

```
// If either child of hole is larger than element return the index
// of the larger child; otherwise return the index of hole.
private int newHole(int hole, T element)
{
  int left = (hole * 2) + 1;
  int right = (hole * 2) + 2;
  if (left > lastIndex)
    // hole has no children
    return hole;
  else
  if (left == lastIndex)
    // hole has left child only
    if (element.compareTo(elements.get(left)) < 0)
      // element < left child
      return left;
    else
      // element >= left child
      return hole;
  else
  // hole has two children
  if (elements.get(left).compareTo(elements.get(right)) < 0)
    // left child < right child
    if (elements.get(right).compareTo(element) <= 0)
      // right child <= element
      return hole;
    else
      // element < right child
      return right;
  else
  // left child >= right child
  if (elements.get(left).compareTo(element) <= 0)
    // left child <= element
    return hole;
  else
    // element < left child
    return left;
}
```


<<interface>>
PriQueueInterface<T extends Comparable<T>>
+enqueue(T element): void
+dequeue(): T
+isFull(): boolean
+isEmpty(): boolean
Key:
implements
Heap<T extends Comparable<T>>
-elements: ArrayList<T>
-lastIndex: int
-maxIndex: int
+Heap(int maxSize)
+isEmpty(): boolean
+isFull(): boolean
+enqueue(element T): void
+dequeue(): T
+toString(): String
-reheapUp(element T); void
-newHole(hole: int, element: T): int
-reheapDown(element T): void

Heaps Versus Other Representations of Priority Queues

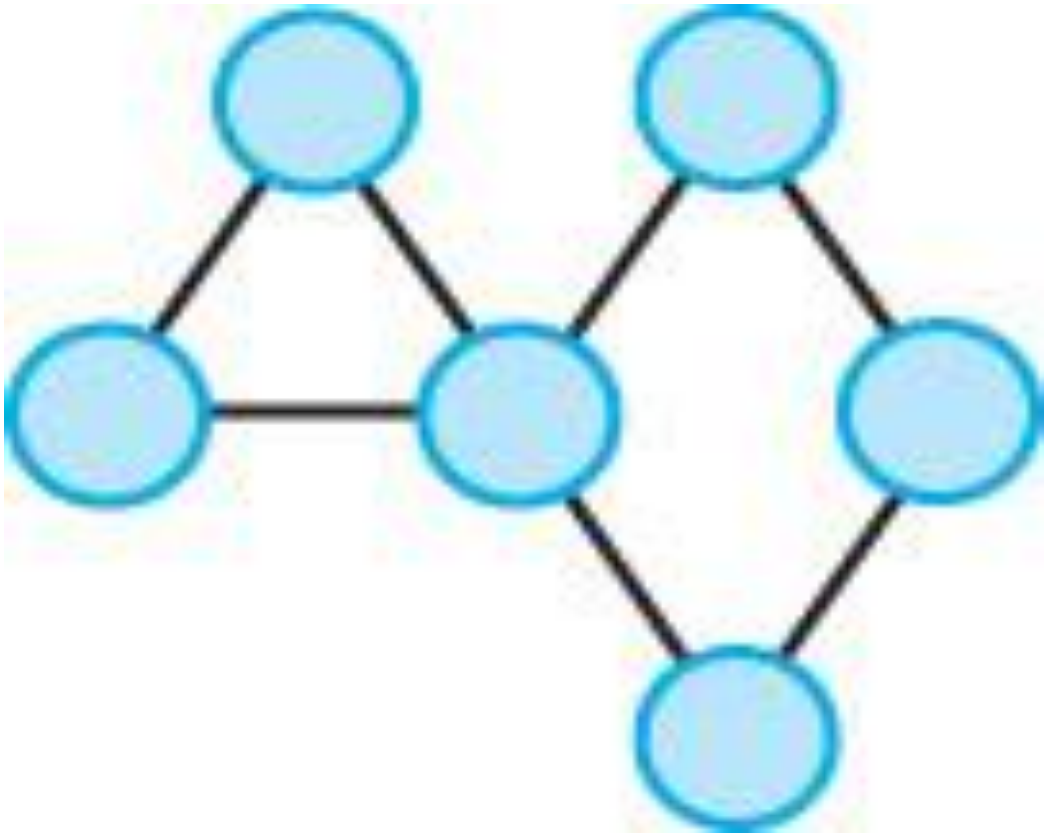
	enqueue	dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$

Lab

- Answer some chapter Questions together
- 6, 7, 8, 10, 11

Break

9.3 Introduction to Graphs



Definitions

- **Graph:** A data structure that consists of a set of nodes and a set of edges that relate the nodes to each other
- **Vertex:** A node in a graph
- **Edge (arc):** A pair of vertices representing a connection between two nodes 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:

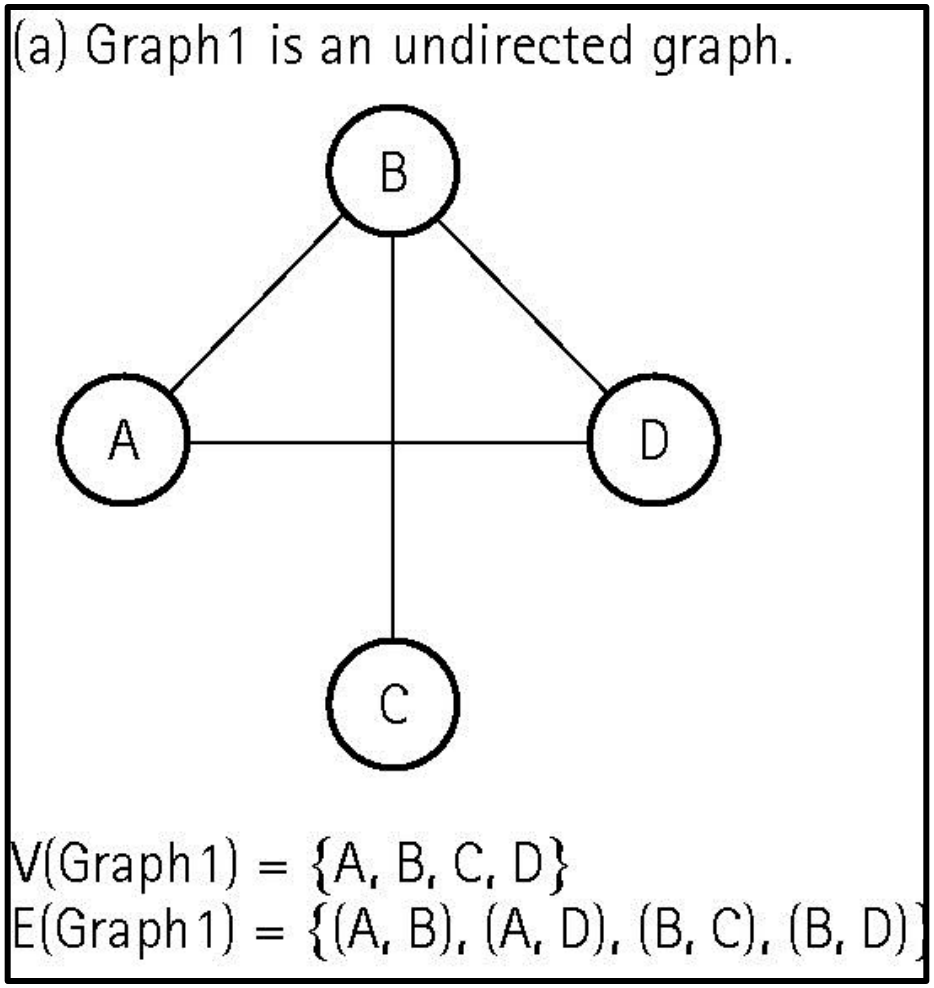
  G = (V,E)

  where

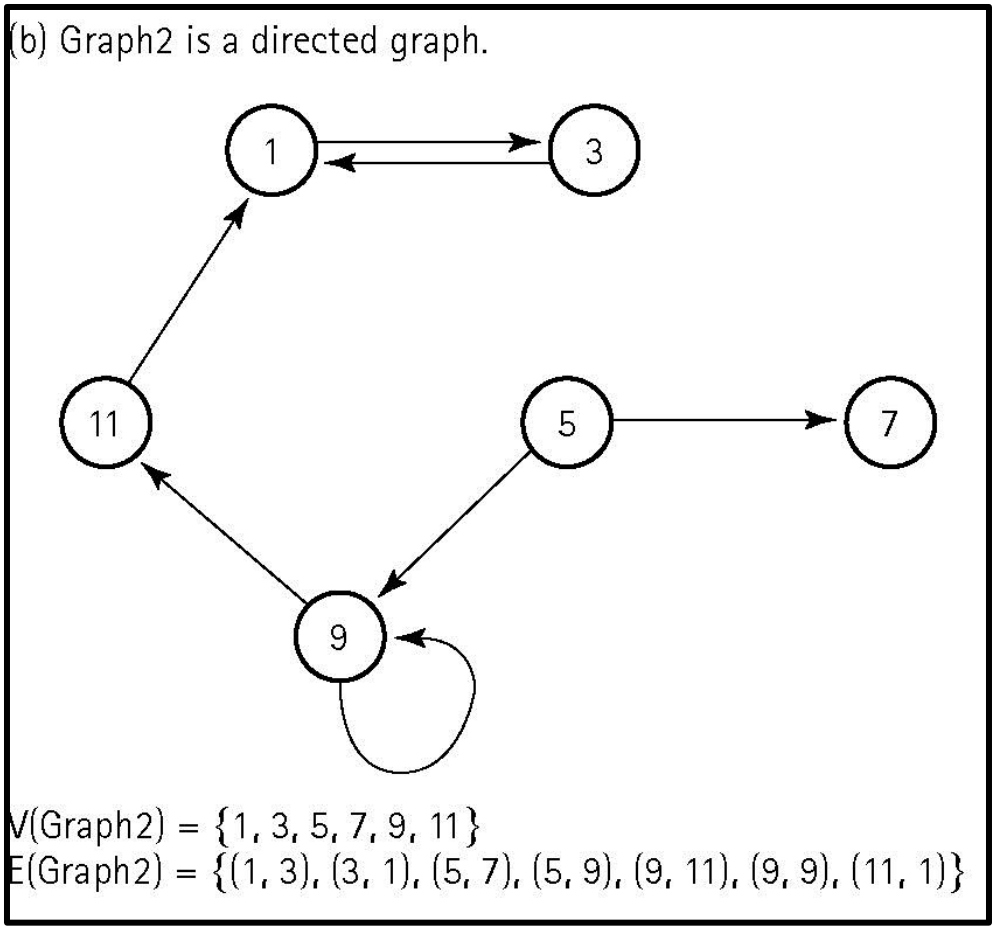
  V(G) is a finite, nonempty set of vertices

  E(G) is a set of edges (written as pairs of vertices)

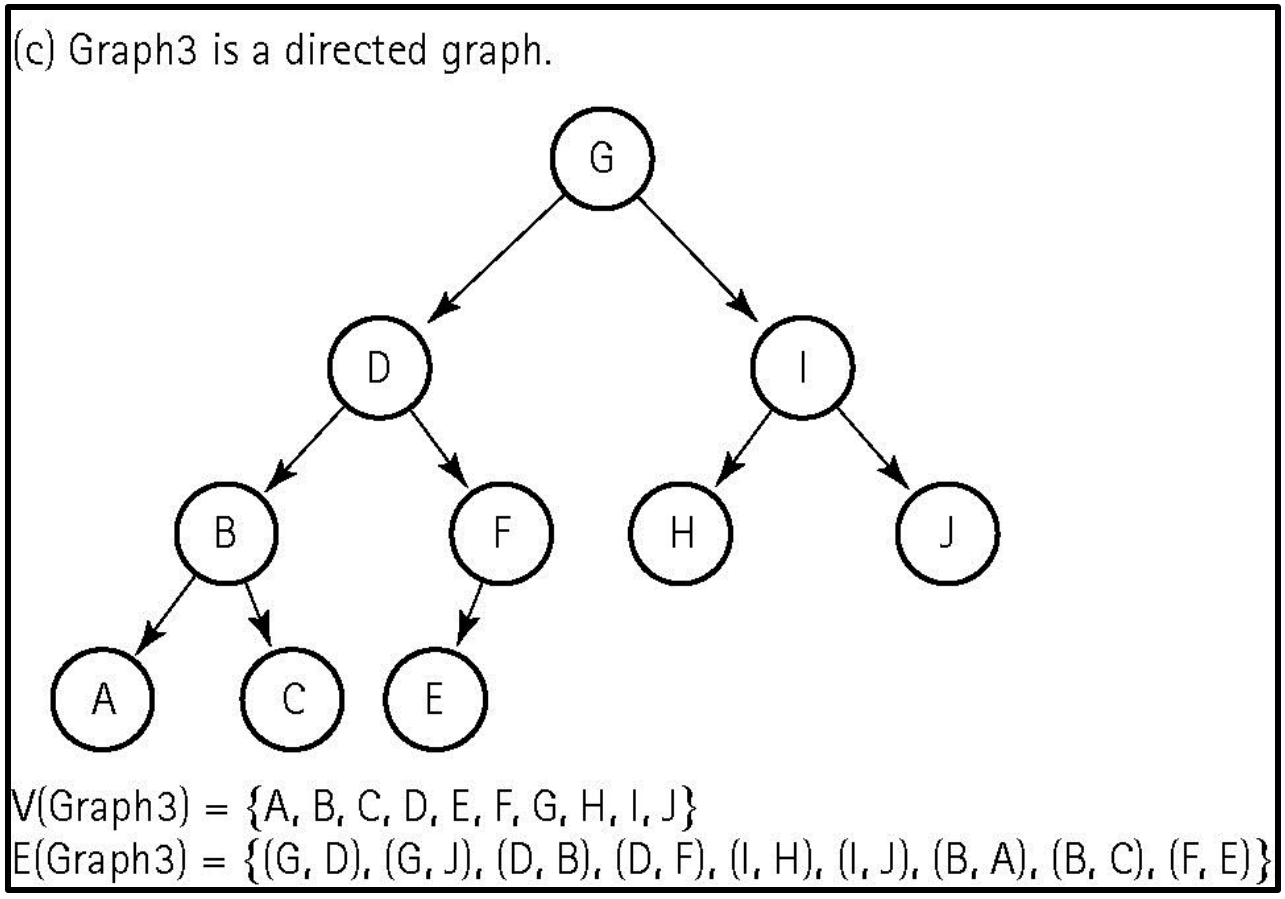
An undirected graph



A directed graph



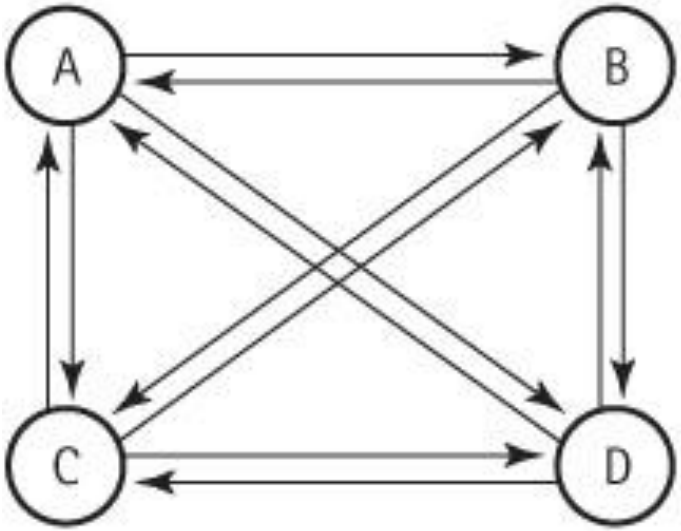
Another directed graph



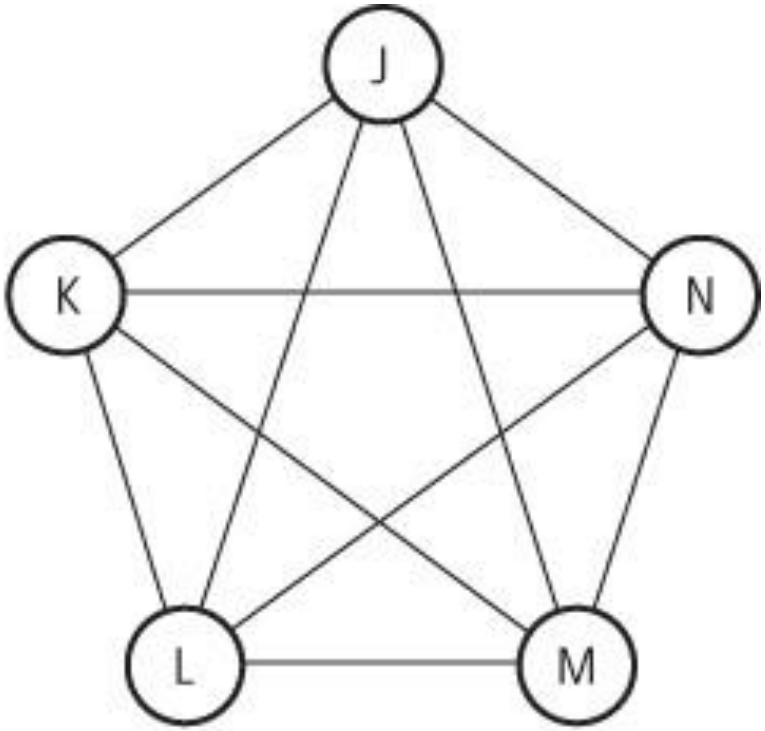
More Definitions

- **Adjacent vertices:** Two vertices in a graph that are connected by an edge
- **Path:** A sequence of vertices that connects two nodes 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

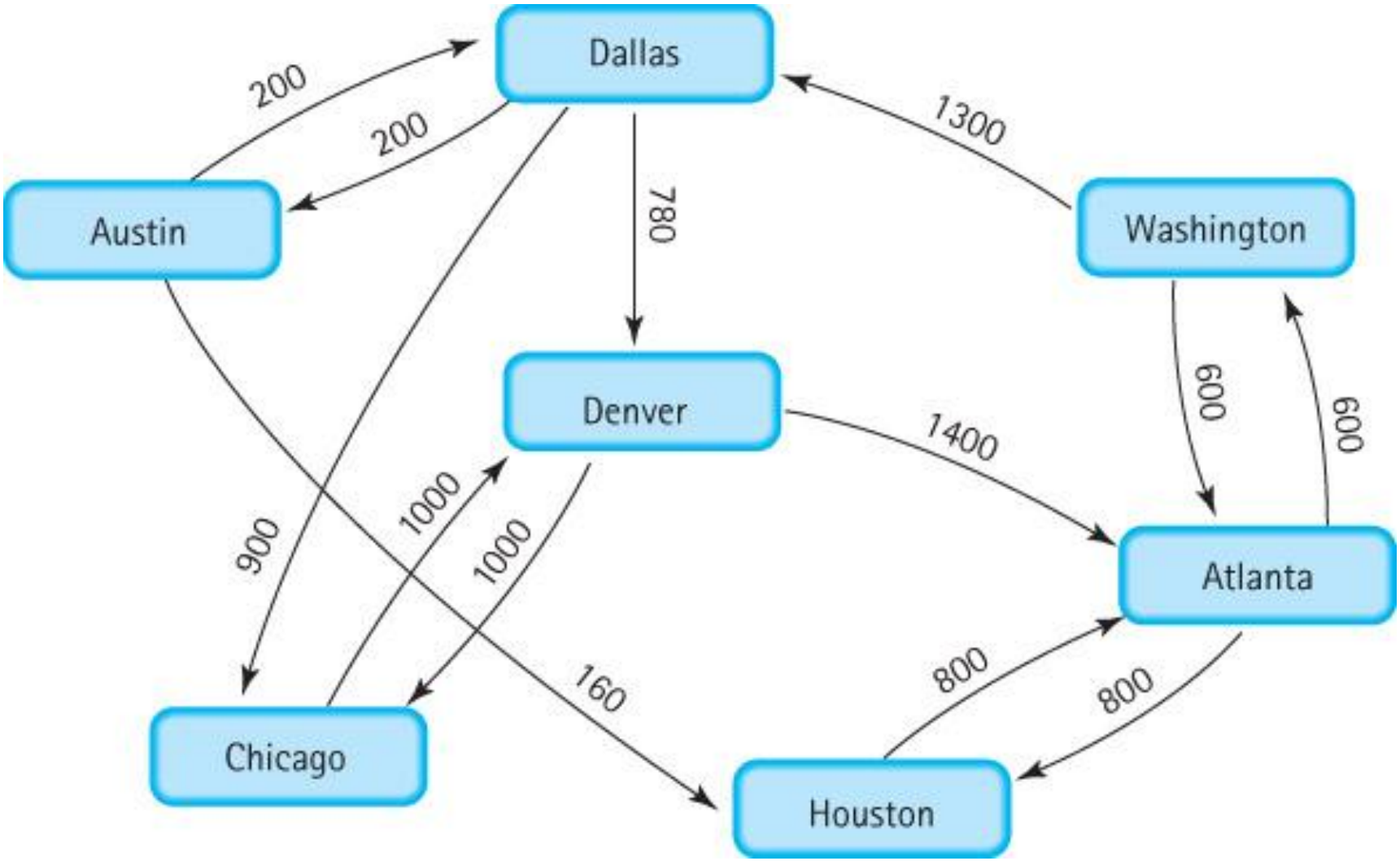


(a) Complete directed graph.



(b) Complete undirected graph.

A weighted graph



9.4 Formal Specification of a Graph ADT

- 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; operations have been chosen that are useful in the graph applications described later in the chapter

WeightedGraphInterface.java part I

```
//-----------------------------------------------------------------------
// WeightedGraphInterface.java       by Dale/Joyce/Weems              Chapter 9
//
// Interface for a class that implements a directed graph with weighted edges.
// Vertices are objects of class T and can be marked as having been visited.
// Edge weights are integers.
//
// General precondition: except for the addVertex and hasVertex methods,
// any vertex passed as an argument to a method is in this graph.
//-----------------------------------------------------------------------

package ch09.graphs;

import ch05.queues.*;

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

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

WeightedGraphInterface.java part II

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

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

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

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

WeightedGraphInterface.java part III

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

  // Sets marks for all vertices to false.
  void clearMarks();

  // Sets mark for vertex to true.
  void markVertex(T vertex);

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

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

9.5 Implementations of Graphs

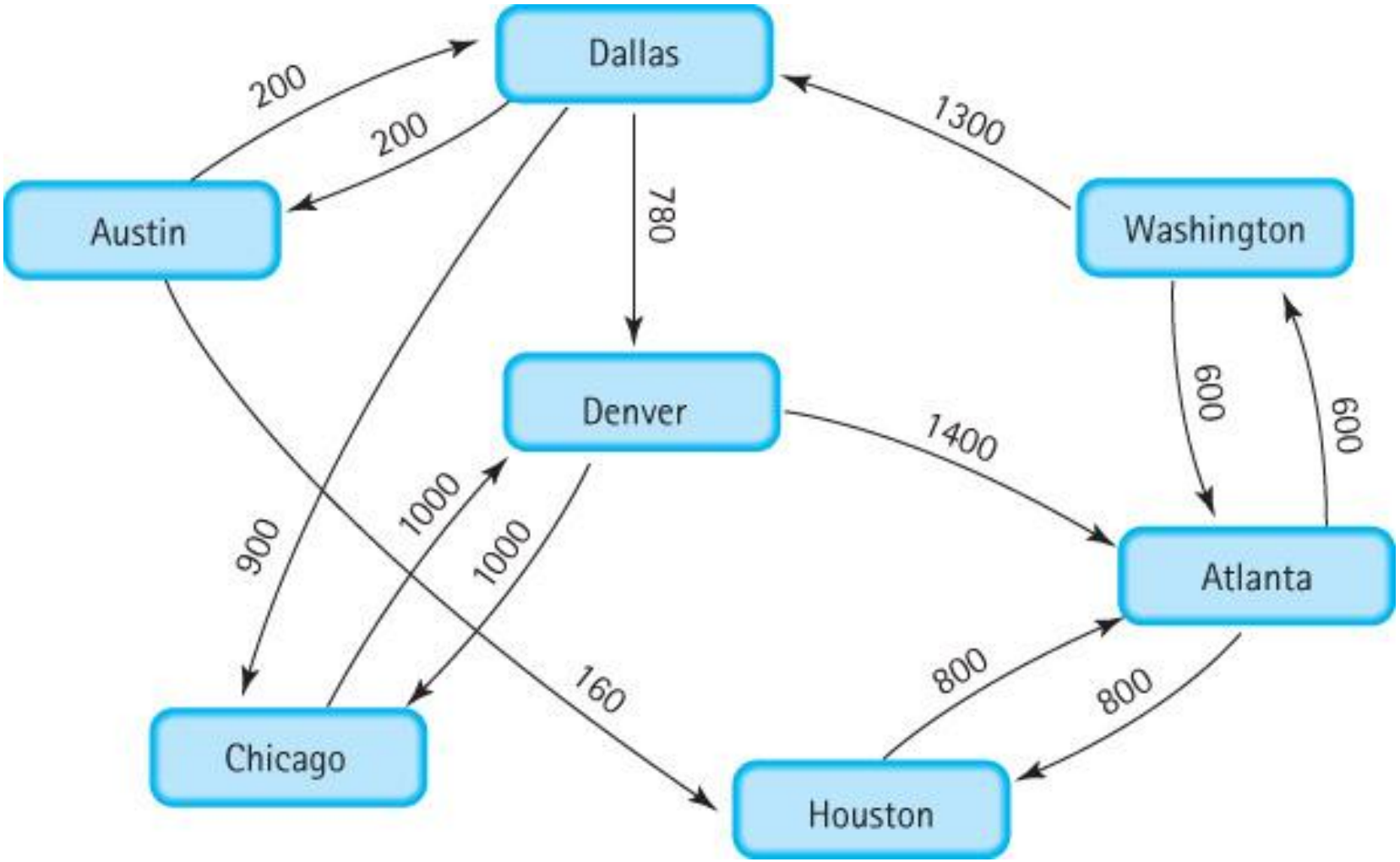
– array based approach
– linked based approach

Array-Based Implementation

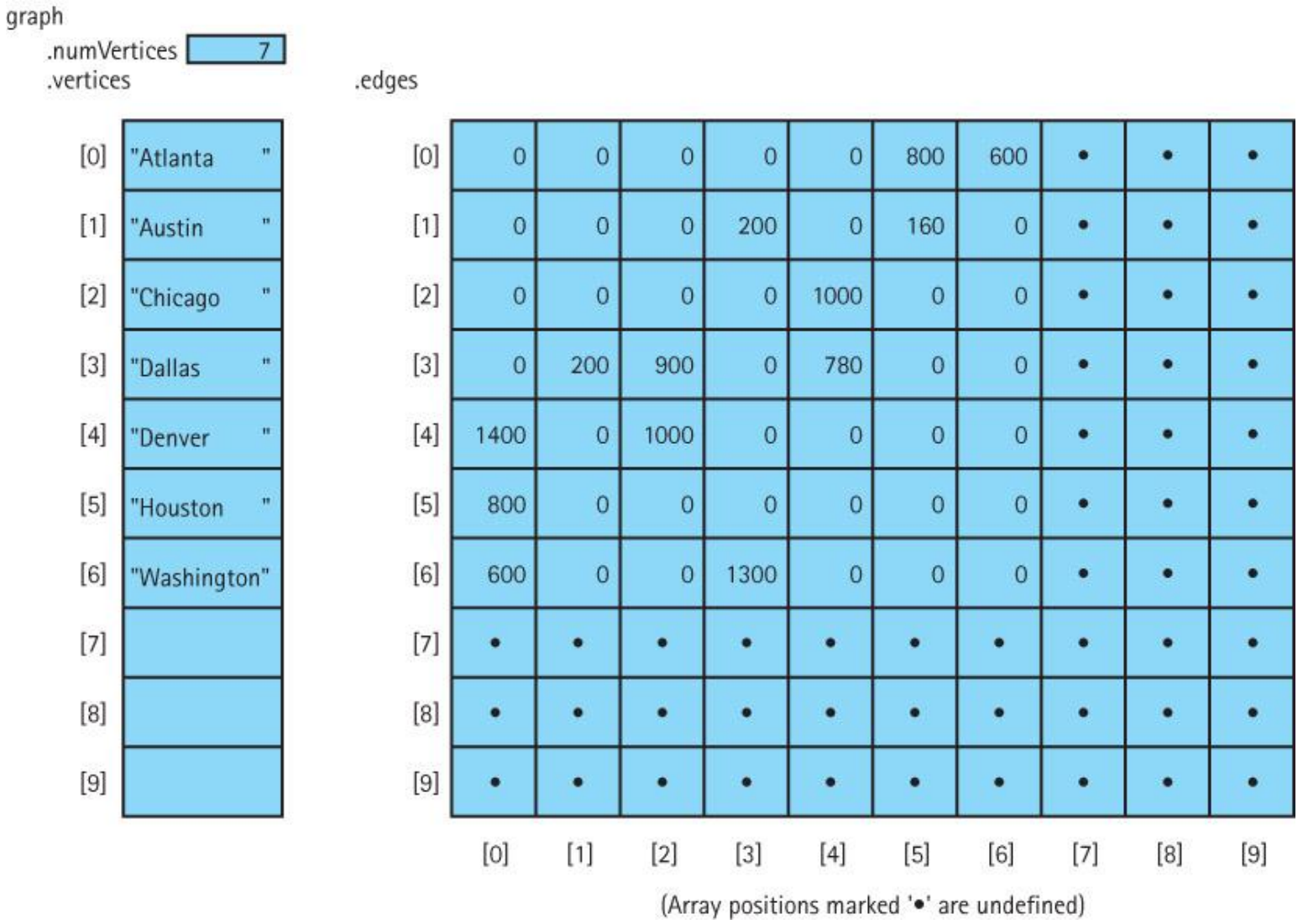
- **Adjacency matrix**
  - For a graph with N nodes, an N by N table that shows the existence (and weights) of all edges in the graph

- The 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



The array-based implementation



WeightedGraph.java
instance variables

```
package ch09.graphs;

import ch05.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

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

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

WeightedGraph.java
adding a vertex

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

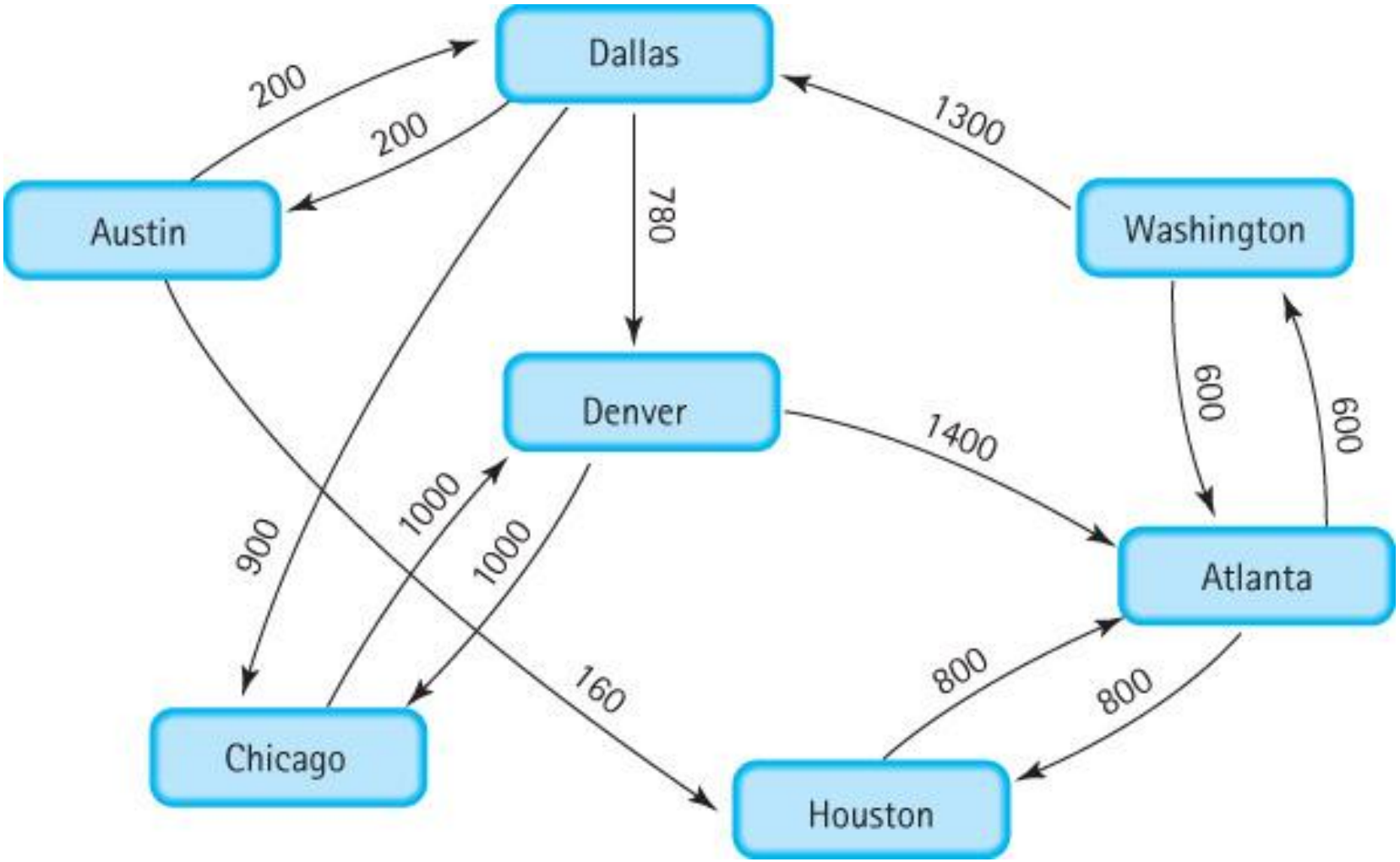
Coding the remaining methods is left as homework.


<<interface>>
WeightedGraphInterface<T>
+isFull(): boolean
+isEmpty(): boolean
+addVertex(vertex: T): void
+hasVertex(vertex: T): boolean
+addEdge(fromVertex: T, toVertex: T, weight: int): void
+weightIs(fromVertex: T, toVertex: T): int
+getToVertices(vertex: T): UnboundedQueueInterface<T>
+clearMarks(): void
+markVertex(vertex: T): void
+isMarked(vertex: T): boolean
+getUnmarked(): T
Key:
implements
WeightedGraph<T>
+NULL_EDGE = 0
-DEFCAP = 50
-numVertices: int
-maxVertices: int
-vertices: T[]
-edges: int[][]
-marks: boolean[]
+WeightedGraph()
+WeightedGraph(int maxV)
+isFull(): boolean
+isEmpty(): boolean
+addVertex(vertex: T): void
+hasVertex(vertex: T): boolean
+addEdge(fromVertex: T, toVertex: T, weight: int): void
+weightIs(fromVertex: T, toVertex: T): int
+getToVertices(vertex: T): UnboundedQueueInterface<T>
+clearMarks():void
+markVertex(vertex: T): void
+isMarked(vertex: T): boolean
+getUnmarked(): T
-indexIs(vertex: T): int

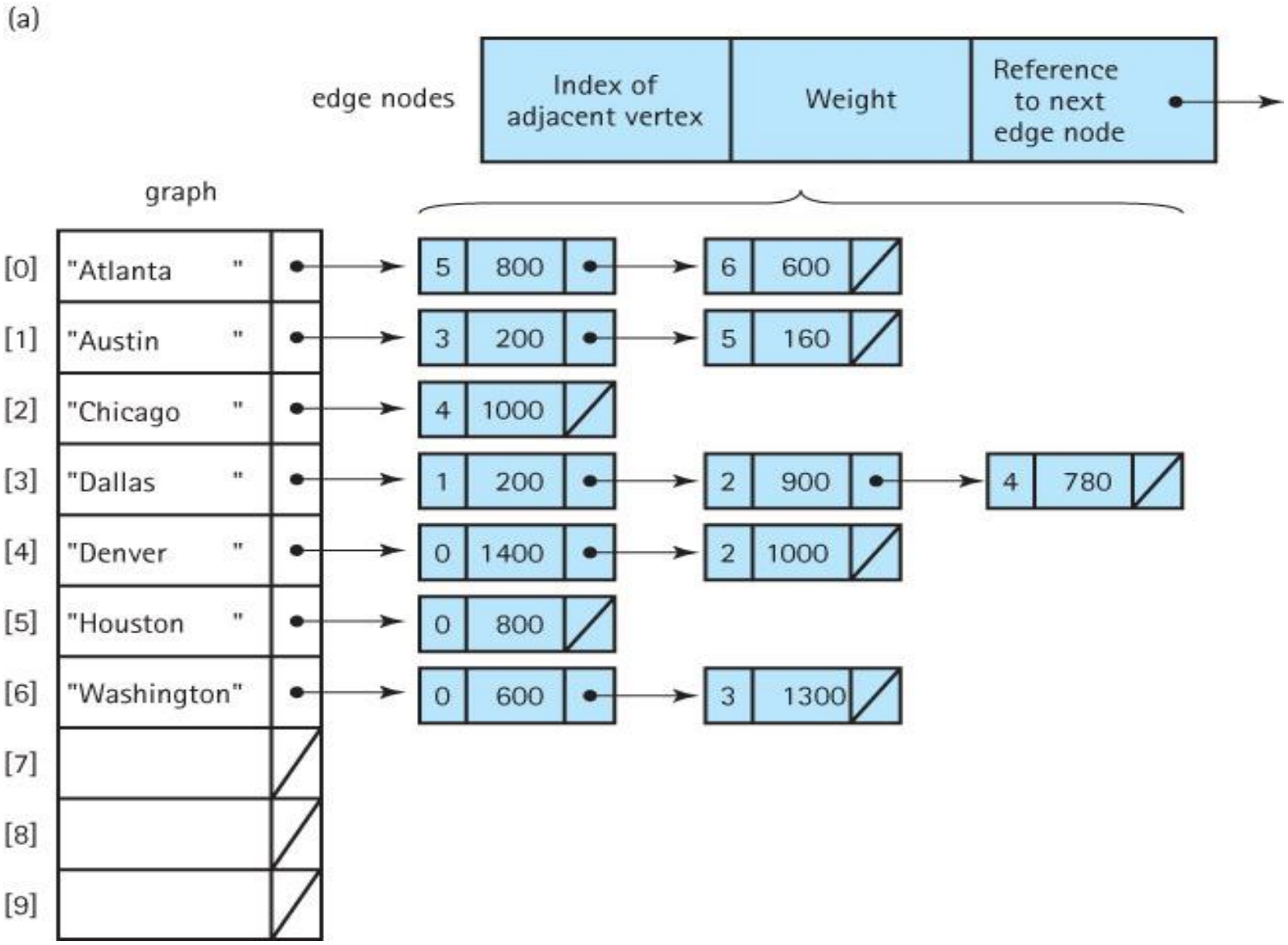
Link-Based Implementation

- **Adjacency list**
  - A linked list that identifies all the vertices to which a particular vertex is connected.
  - Each vertex has its own adjacency list
- Two alternate approaches:
  - An array of vertices that each contain a reference to a linked list of nodes
  - A linked list of vertices that each contain a reference to a linked list of nodes

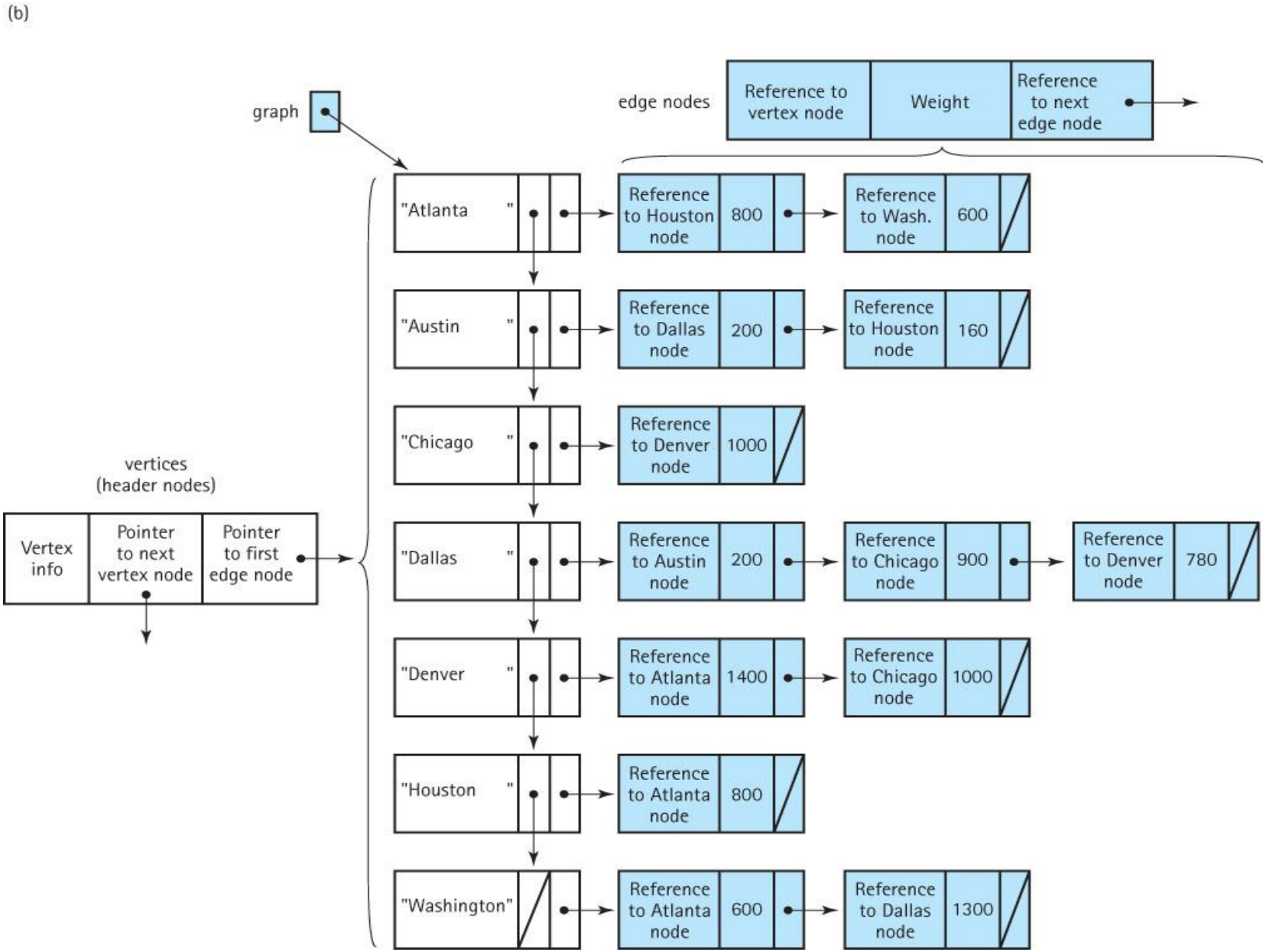
A repeat of the abstract model



The first link-based implementation



The second link-based implementation



9.6 Graph Applications

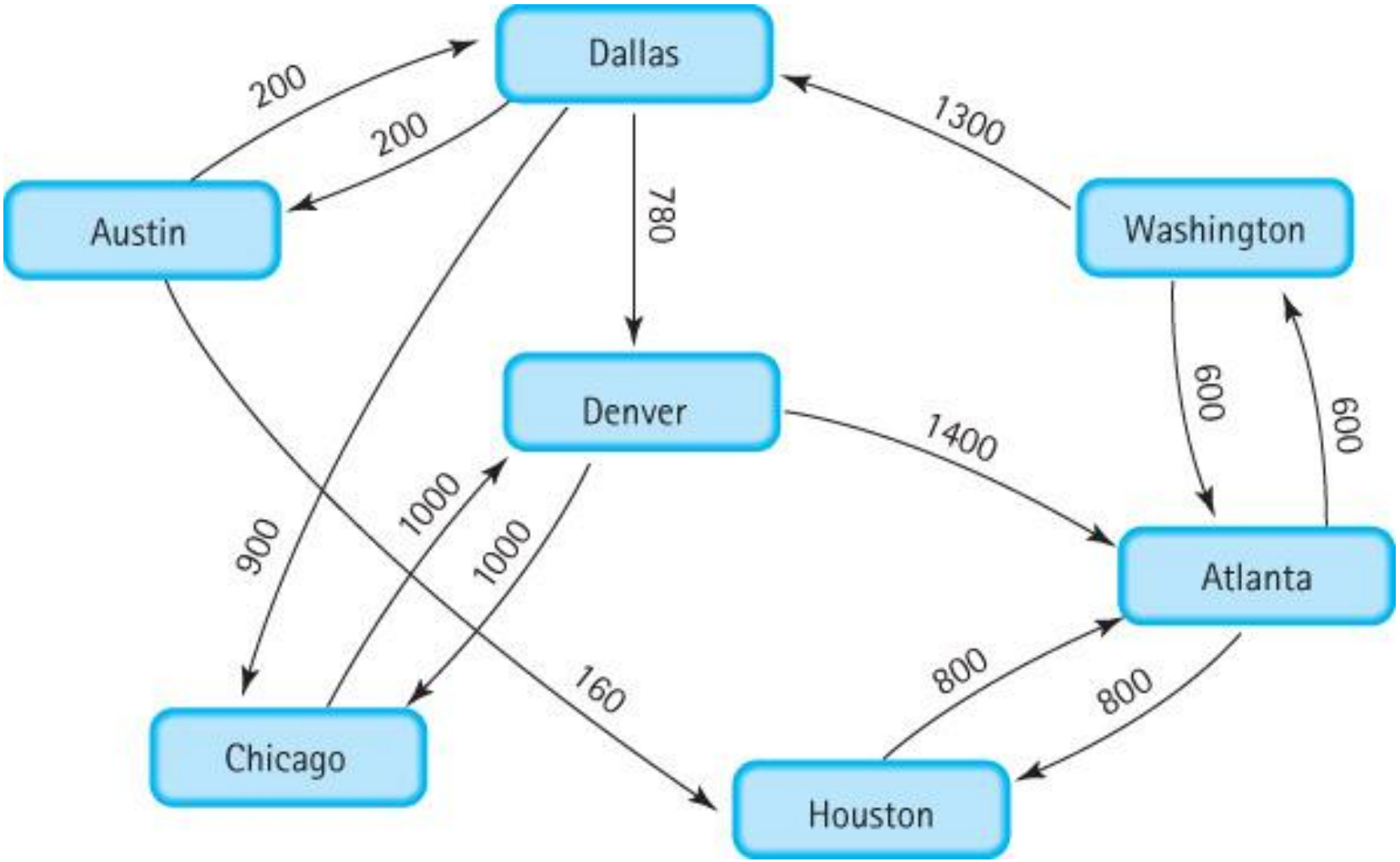
- Our graph specification does not include traversal operations.
- We treat traversal as a graph application/algorithm rather than an innate operation.
- The basic operations given in our specification allow us to implement different traversals *independent* of how the graph itself is actually implemented.

Graph Traversal

- We look at two types of graph traversal:
  - *depth-first*
    - Going down a branch to its deepest point and moving up
  - *breadth-first*
    - Visit each vertex in a tree
      - visit each vertex on level 0 (the root),
      - then each vertex on level 1,
      - then each vertex on level 2, and so on.

- We discuss algorithms for employing both strategies within the context of determining if two cities are connected in our airline example.

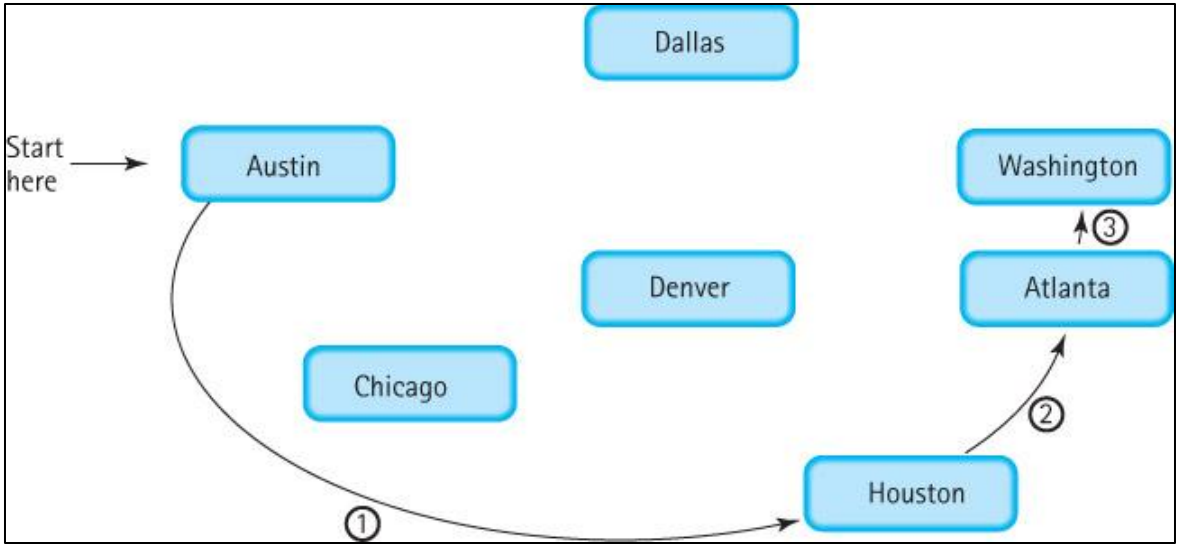
Can we get from Austin to Washington?



Depth first search

IsPath (startVertex, endVertex): returns boolean

```
Set found to false
graph.clearMarks()
stack.push(startVertex)
do
   vertex = stack.top( )
   stack.pop( )
   if vertex = endVertex
      Set found to true
   else
      if vertex is not marked
         Mark vertex
         Push all adjacent vertices onto stack
while !stack.isEmpty( ) AND !found
return found
```



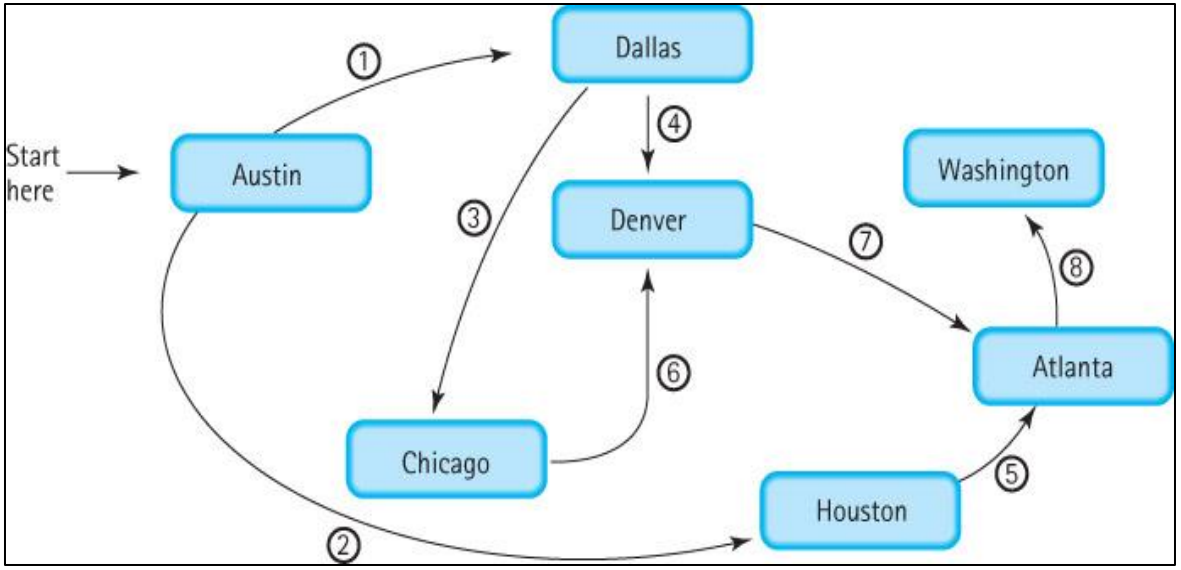
```
private static boolean isPath(WeightedGraphInterface<String> graph,
                              String startVertex, String endVertex    )
{
  UnboundedStackInterface<String> stack = new LinkedStack<String> ();
  UnboundedQueueInterface<String> vertexQueue = new LinkedUnbndQueue<String> ();

  boolean found = false;
  String vertex;
  String item;
  graph.clearMarks();
  stack.push(startVertex);
  do
  {
    vertex = stack.top();
    stack.pop();
    if (vertex == endVertex)
       found = true;
    else
    {
      if (!graph.isMarked(vertex))
      {
        graph.markVertex(vertex);
        vertexQueue = graph.getToVertices(vertex);

        while (!vertexQueue.isEmpty())
        {
          item = vertexQueue.dequeue();
          if (!graph.isMarked(item))
            stack.push(item);
        }
      }
    }
  } while (!stack.isEmpty() && !found);
  return found;
}
```

`isPath` method
Depth-first approach

Breadth first search – use queue

IsPath (startVertex, endVertex): returns boolean

Set found to false
graph.clearMarks()
queue.enqueue(startVertex)




```
do
   vertex = queue.dequeue( )
   if vertex = endVertex
      Set found to true
   else
      if vertex is not marked
         Mark vertex
         Enqueue all adjacent vertices onto queue
while !queue.isEmpty( ) AND !found

return found
```

```
private static boolean isPath2(WeightedGraphInterface<String>  graph,
                               String startVertex, String endVertex)
{
  QueueInterface<String> queue = new LinkedQueue<String>();
  QueueInterface<String> vertexQueue = new LinkedQueue<String>();
  boolean found = false;
  String vertex;
  String element;

  graph.clearMarks();
  queue.enqueue(startVertex);
  do
  {
    vertex = queue.dequeue();
    if (vertex == endVertex)
       found = true;
    else
    {
      if (!graph.isMarked(vertex))
      {
        graph.markVertex(vertex);
        vertexQueue = graph.getToVertices(vertex);

        while (!vertexQueue.isEmpty())
        {
          element = vertexQueue.dequeue();
          if (!graph.isMarked(element))
            queue.enqueue(element);
         }
      }
    }
  } while (!queue.isEmpty() && !found);

  return found;
}
```

`isPath` method
Breadth-first approach

The Single-Source Shortest-Paths Algorithm

- 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)
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)
```

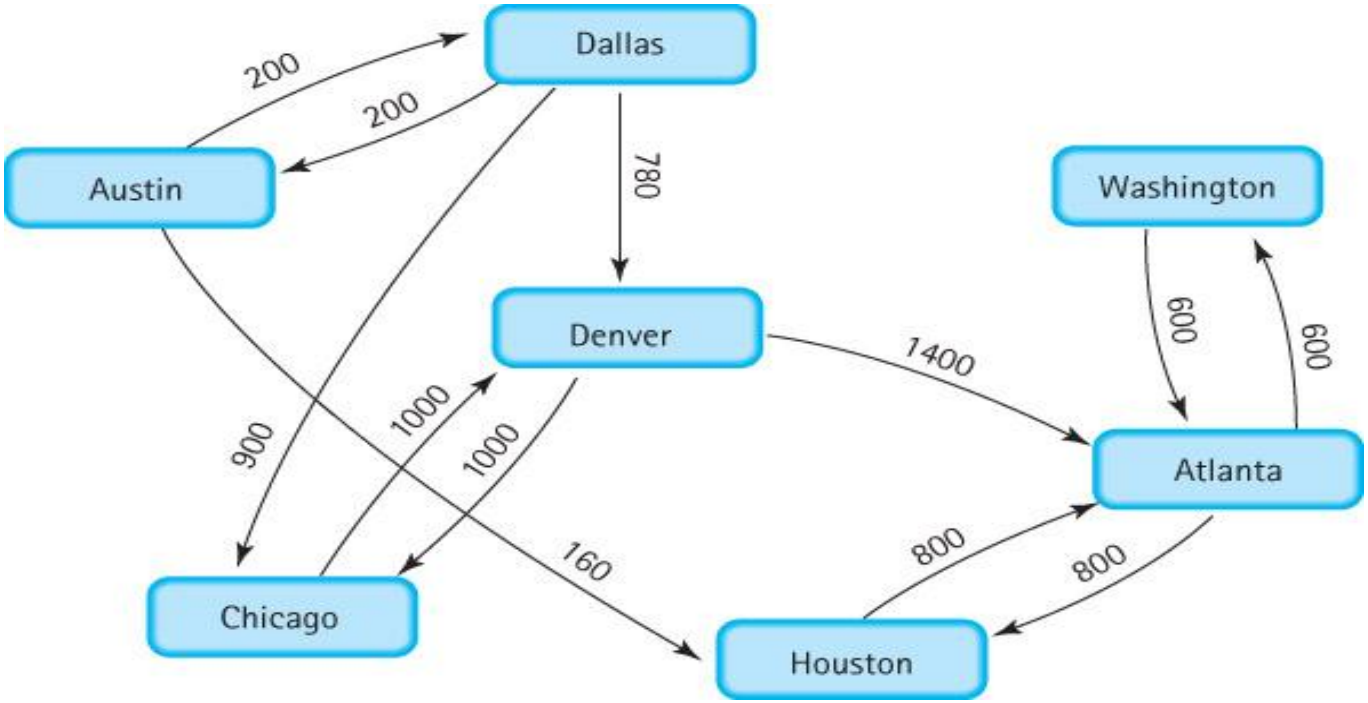
This part of the algorithm walks through the queue of vertices adjacent to the current vertex, and enqueues `Flights` objects onto the priority queue `pq` based on the information. The flight variable is actually a reference to a `Flights` object. Suppose the queue of adjacent vertices has information 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 `Flights` object referenced by flight. We update it to contain information about Houston. And we again enqueue it in `pq`. So now `pq` loses the information it had about Atlanta.

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 source code for the `shortestPaths` method is too long to fit on a slide. It can be found on pages 657-658 of the textbook and also with the other textbook code in the `UseGraph.java` application.

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();
}
```

Homework / Lab

In WeightedGraph.java code the following:

- isEmpty
- isFull
- hasVertex
- indexIs
- clearMarks
- markVertex

Final Project/Exam

- Compare and benchmark 5 sorting algorithms using ADTs. Compare all 7 for up to 20 points extra credit.
  - **You must use HeapSort (Heaps), QuickSort (Recursive), MergeSort (Array)**
  - StoogeSort (recursive), BubbleSort, InsertionSort, SelectionSort
  - Or your choice of others from class or on Wikipedia.
  - Comparing an implementation using an array vs. linked lists might be interesting
  - I will also accept Binary search vs sequential search analysis (swaps = 0)
- You will run comparisons as performed previously in class during lab and in homework assignments.
- I will provide 3 files for sort testing and you must also generate random sets of numbers of different sizes (10, 100, 1000, 10000, 100000). You will also perform 3 runs with each set of randomly generated numbers. Report
  - Results in tabular form, preferably a spreadsheet (Excel)
  - Time, # of swaps, # of comparisons for each run and the average of similar runs for each sorting method
  - BigO notation for swaps and comparisons
  - Report in a common word processor format such as .doc or .docx
- Your analysis. Which method would you use and why?