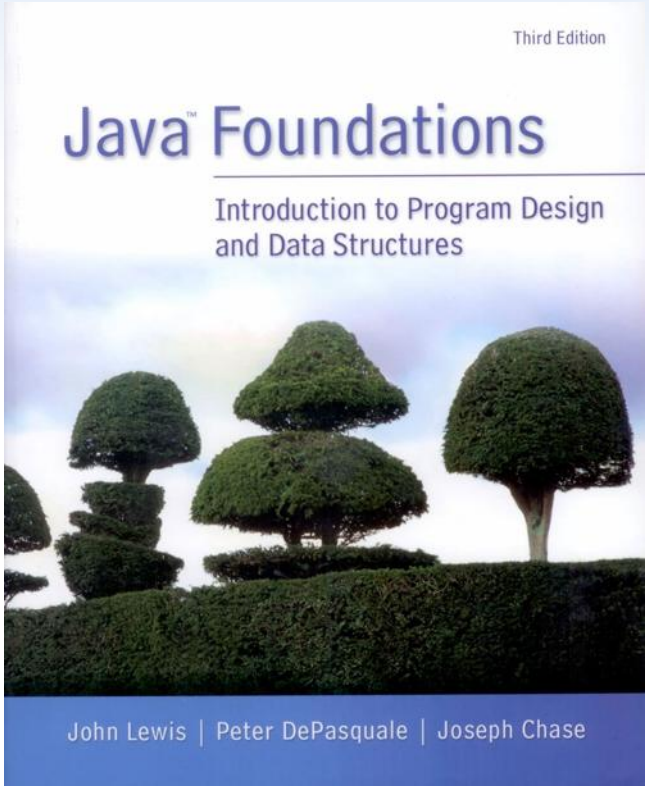


Chapter 24

Graphs

Chapter Scope

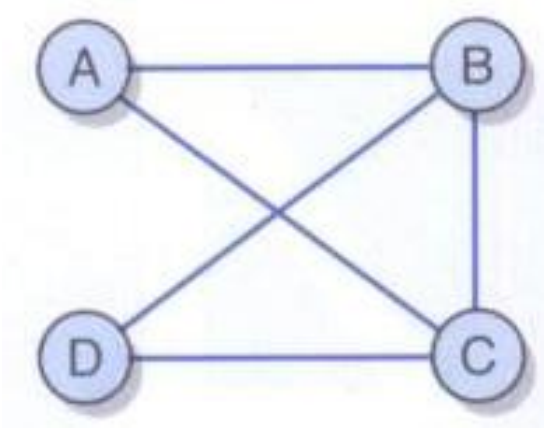
- Directed and undirected graphs
- Weighted graphs (networks)
- Common graph algorithms

Graphs

- Like trees, *graphs* are made up of nodes and the connections between those nodes
- In graph terminology, we refer to the nodes as *vertices* and refer to the connections among them as *edges*
- Vertices are typically referenced by a name or label
- Edges are referenced by a pairing of the vertices (A, B) that they connect
- An *undirected graph* is a graph where the pairings representing the edges are unordered

Undirected Graphs

- An undirected graph:



Undirected Graphs

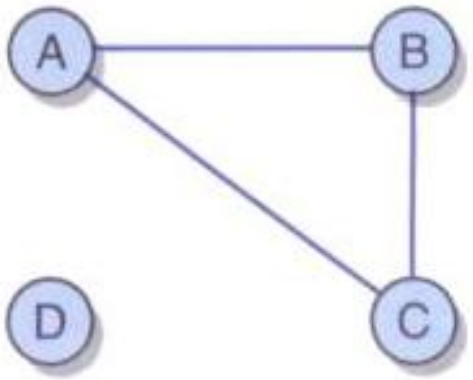
- An edge in an undirected graph can be traversed in either direction
- Two vertices are said to be *adjacent* if there is an edge connecting them
- Adjacent vertices are sometimes referred to as *neighbors*
- An edge of a graph that connects a vertex to itself is called a *self-loop* or *sling*
- An undirected graph is considered *complete* if it has the maximum number of edges connecting vertices

Undirected Graphs

- A *path* is a sequence of edges that connects two vertices in a graph
- The *length* of a path in is the number of edges in the path (or the number of vertices minus 1)
- An undirected graph is considered *connected* if for any two vertices in the graph there is a path between them

Undirected Graphs

- An example of an undirected graph that is not connected:



Cycles

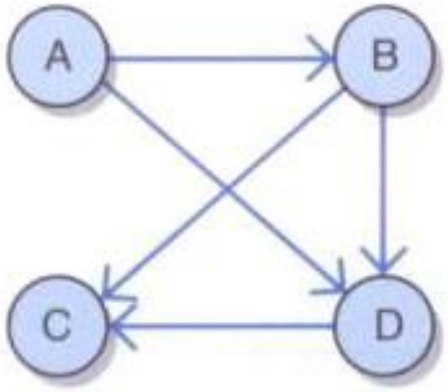
- A *cycle* is a path in which the first and last vertices are the same and none of the edges are repeated
- A graph that has no cycles is called *acyclic*

Directed Graphs

- A *directed graph*, sometimes referred to as a *digraph*, is a graph where the edges are ordered pairs of vertices
- This means that the edges (A, B) and (B, A) are separate, directional edges in a directed graph

Directed Graphs

- A directed graph with
  - vertices A, B, C, D
  - edges (A, B), (A, D), (B, C), (B, D) and (D, C)

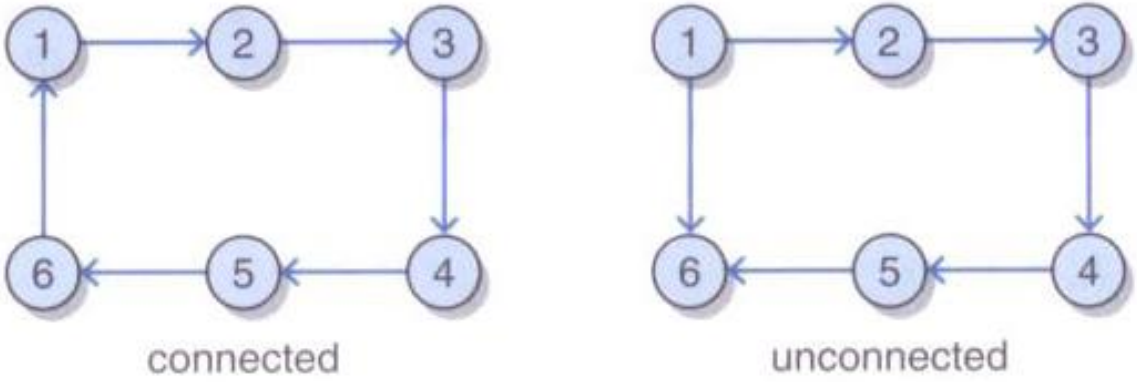


Directed Graphs

- Previous definitions change slightly for directed graphs
  - a path in a direct graph is a sequence of directed edges that connects two vertices in a graph
  - a directed graph is connected if for any two vertices in the graph there is a path between them
  - if a directed graph has no cycles, it is possible to arrange the vertices such that vertex A precedes vertex B if an edge exists from A to B

Directed Graphs

- A connected directed graph and an unconnected directed graph:

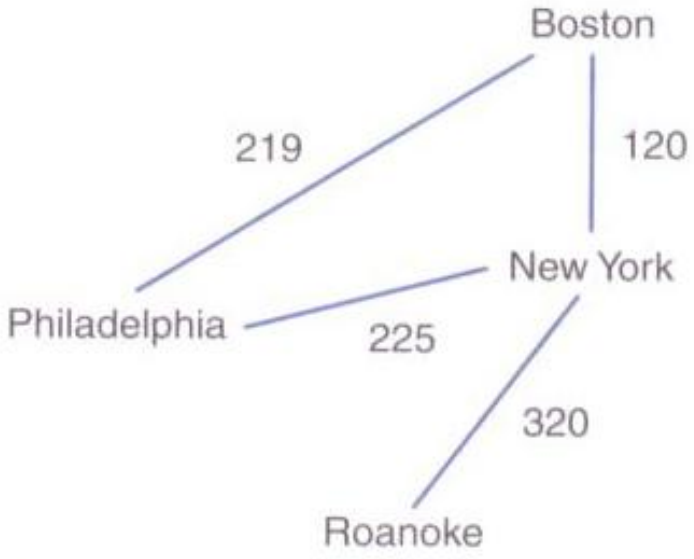


Weighted Graphs

- A *weighted graph*, sometimes called a *network*, is a graph with weights (or costs) associated with each edge
- The weight of a path in a weighted graph is the sum of the weights of the edges in the path
- Weighted graphs may be either undirected or directed
- For weighted graphs, we represent each edge with a triple including the starting vertex, ending vertex, and the weight (Boston, New York, 120)

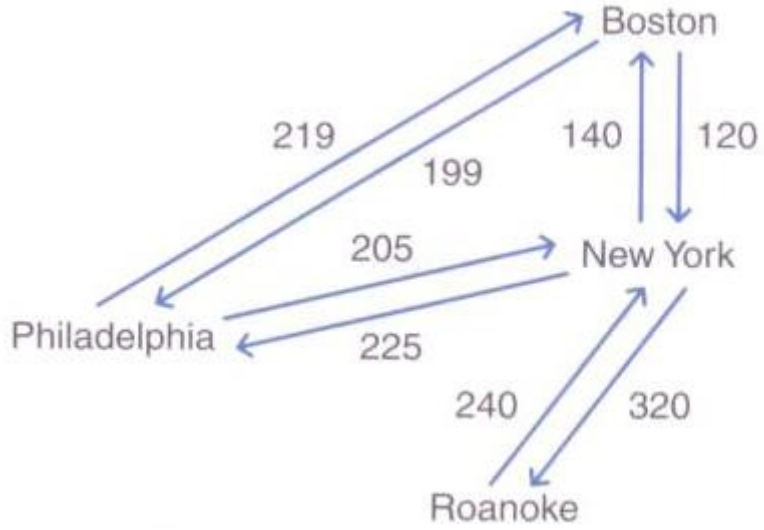
Weighted Graphs

- We could use an undirected network to represent flights between cities
- The weights are the cost



Weighted Graphs

- A directed version of the graph could show different costs depending on the direction



Common Graph Algorithms

- There are a number of a common algorithms that may apply to undirected, directed, and/or weighted graphs
- These include
  - various traversal algorithms
  - algorithms for finding the shortest path
  - algorithms for finding the least costly path in a network
  - algorithms for answering simple questions (such as connectivity)

Graph Traversals

- There are two main types of graph traversal algorithms
  - *breadth-first:* behaves much like a level-order traversal of a tree
  - *depth-first:* behaves much like the preorder traversal of a tree
- One difference: there is no root node present in a graph
- Graph traversals may start at any vertex in the graph

Breadth-First Traversal

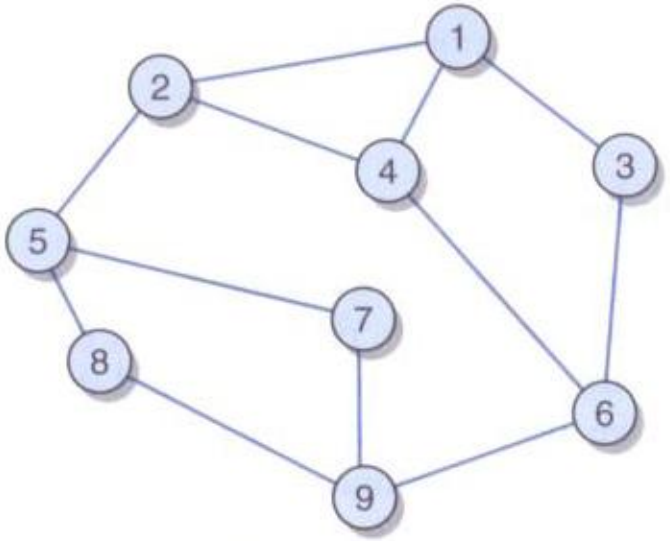
- We can construct a breadth-first traversal for a graph using a queue and an iterator
- We will use the queue to manage the traversal and the iterator to build the result
- Breadth-first traversal algorithm overview
  - enqueue the starting vertex, and mark it as “visited”
  - loop until queue is empty
    - dequeue first vertex and add it to iterator
    - enqueue each unmarked vertex adjacent to the dequeued vertex
    - mark each vertex enqueued as “visited”

```
/**
 * Returns an iterator that performs a breadth first
 * traversal starting at the given index.
 *
 * @param startIndex the index from which to begin the traversal
 * @return an iterator that performs a breadth first traversal
 */
public Iterator<T> iteratorBFS(int startIndex)
{
    Integer x;
    QueueADT<Integer> traversalQueue = new LinkedQueue<Integer>();
    UnorderedListADT<T> resultList = new ArrayUnorderedList<T>();

    if (!indexIsValid(startIndex))
        return resultList.iterator();

    boolean[] visited = new boolean[numVertices];
    for (int i = 0; i < numVertices; i++)
        visited[i] = false;

    traversalQueue.enqueue(new Integer(startIndex));
    visited[startIndex] = true;
```

```
    while (!traversalQueue.isEmpty())
    {
        x = traversalQueue.dequeue();
        resultList.addToRear(vertices[x.intValue()]);

        //  Find all vertices adjacent to x that have not been visited
        //  and queue them up

        for (int i = 0; i < numVertices; i++)
        {
            if (adjMatrix[x.intValue()][i] && !visited[i])
            {
                traversalQueue.enqueue(new Integer(i));
                visited[i] = true;
            }
        }
    }
    return new GraphIterator(resultList.iterator());
}
```

Graph Traversals

- A traversal example:



Depth-First Traversal

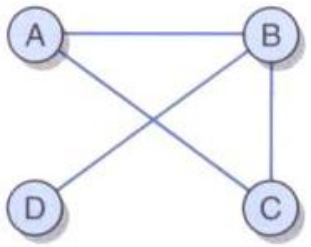
- Use nearly the same approach as the breadth-first traversal, but replace the queue with a stack
- Additionally, do not mark a vertex as visited until it has been added to the iterator

```
/**
 * Returns an iterator that performs a depth first traversal
 * starting at the given index.
 *
 * @param startIndex the index from which to begin the traversal
 * @return an iterator that performs a depth first traversal
 */
public Iterator<T> iteratorDFS(int startIndex)
{
    Integer x;
    boolean found;
    StackADT<Integer> traversalStack = new LinkedStack<Integer>();
    UnorderedListADT<T> resultList = new ArrayUnorderedList<T>();
    boolean[] visited = new boolean[numVertices];

    if (!indexIsValid(startIndex))
        return resultList.iterator();

    for (int i = 0; i < numVertices; i++)
        visited[i] = false;

    traversalStack.push(new Integer(startIndex));
    resultList.addToRear(vertices[startIndex]);
    visited[startIndex] = true;
```

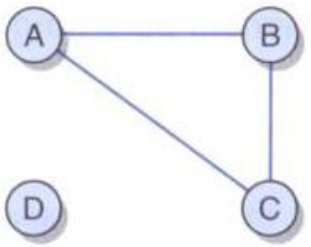
```
        while (!traversalStack.isEmpty())
        {
            x = traversalStack.peek();
            found = false;

            //  Find a vertex adjacent to x that has not been visited
            //  and push it on the stack
            for (int i = 0; (i < numVertices) && !found; i++)
            {
                if (adjMatrix[x.intValue()][i] && !visited[i])
                {
                    traversalStack.push(new Integer(i));
                    resultList.addToRear(vertices[i]);
                    visited[i] = true;
                    found = true;
                }
            }
            if (!found && !traversalStack.isEmpty())
                traversalStack.pop();
        }
        return new GraphIterator(resultList.iterator());
    }
```

Connectivity

- A graph is connected if and only if for each vertex *v* in a graph containing *n* vertices, the size of the result of a breadth-first traversal at *v* is *n*



Starting Vertex	Breadth-First Traversal
A	A, B, C, D
B	B, A, D, C
C	C, B, A, D
D	D, B, A, C



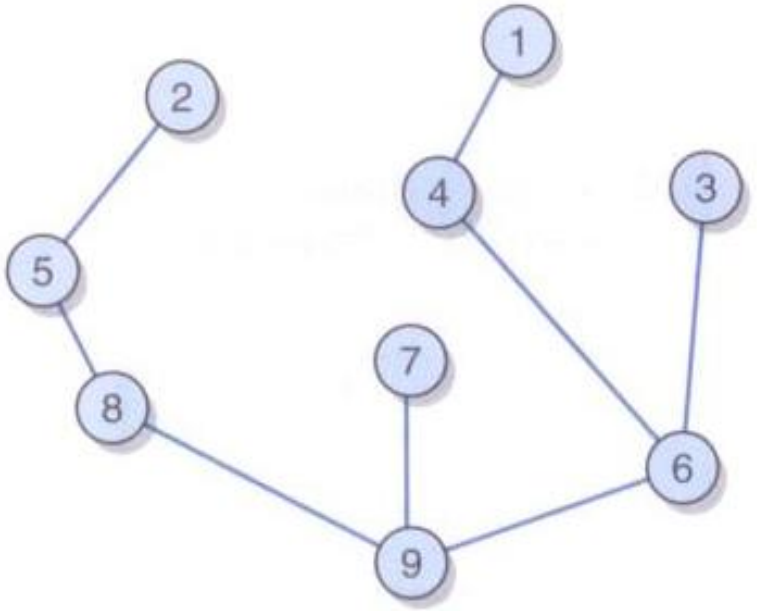
Starting Vertex	Breadth-First Traversal
A	A, B, C
B	B, A, C
C	C, B, A
D	D

Spanning Trees

- A *spanning tree* is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges
- Since trees are also graphs, for some graphs, the graph itself will be a spanning tree, and thus the only spanning tree

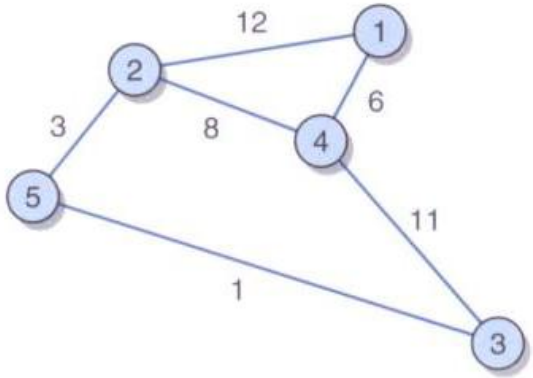
Spanning Tree

- An example of a spanning tree:

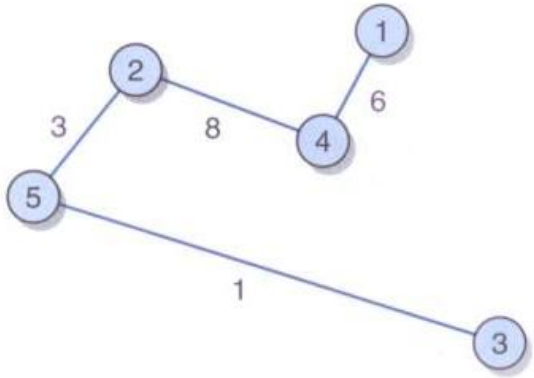


Minimum Spanning Tree

- A *minimum spanning tree* (MST) is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph



Network



Minimum Spanning Tree

Minimum Spanning Tree

- Minimal Spanning Tree algorithm overview
  - pick an arbitrary starting vertex and add it to our MST
  - add all of the edges that include our starting vertex to a minheap ordered by weight
  - remove the minimum edge from the minheap and add the edge and the new vertex to our MST
  - add to the minheap all of the edges that include this new vertex and whose other vertex is not already in the MST
  - continue until the minheap is empty or all vertices are in the MST

Shortest Path

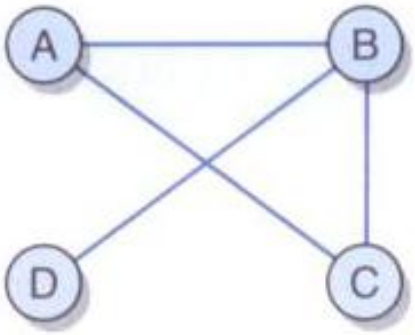
- There are two possibilities for determining the "shortest" path in a graph
  - determine the literal shortest path between a starting vertex and a target vertex (the least number of edges between the two vertices)
    - simplest approach – a variation of the breadth-first traversal algorithm
  - look for the cheapest path in a weighted graph
    - use a minheap or a priority queue storing vertex, weight pairs

Implementing Graphs

- Strategies for implementing graphs:
- *Adjacency lists*
  - use a set of graph nodes which contain a linked list storing the edges within each node
  - for weighted graphs, each edge would be stored as a triple including the weight
- *Adjacency matrices*
  - use a two dimensional array
  - each position of the array represents an intersection between two vertices in the graph
  - each intersection is represented by a boolean value indicating whether or not the two vertices are connected

Adjacency Matrices

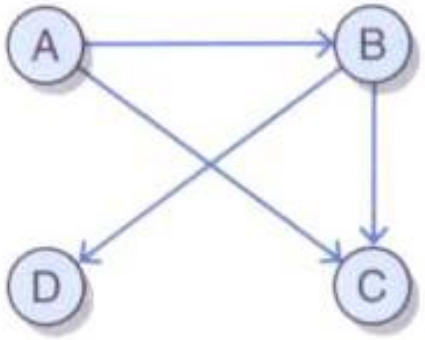
- An undirected graph and it's adjacency matrix:



	A	B	C	D
A	F	T	T	F
B	T	F	T	T
C	T	T	F	F
D	F	T	F	F

Adjacency Matrices

- A directed graph and its adjacency matrix:



	A	B	C	D
A	F	T	T	F
B	F	F	T	T
C	F	F	F	F
D	F	F	F	F

```
package jsjf;

import java.util.Iterator;

/**
 * GraphADT defines the interface to a graph data structure.
 *
 * @author Java Foundations
 * @version 4.0
 */
public interface GraphADT<T>
{
    /**
     * Adds a vertex to this graph, associating object with vertex.
     *
     * @param vertex the vertex to be added to this graph
     */
    public void addVertex(T vertex);

    /**
     * Removes a single vertex with the given value from this graph.
     *
     * @param vertex the vertex to be removed from this graph
     */
    public void removeVertex(T vertex);
```

```
/**
 * Inserts an edge between two vertices of this graph.
 *
 * @param vertex1 the first vertex
 * @param vertex2 the second vertex
 */
public void addEdge(T vertex1, T vertex2);

/**
 * Removes an edge between two vertices of this graph.
 *
 * @param vertex1 the first vertex
 * @param vertex2 the second vertex
 */
public void removeEdge(T vertex1, T vertex2);

/**
 * Returns a breadth first iterator starting with the given vertex.
 *
 * @param startVertex the starting vertex
 * @return a breadth first iterator beginning at the given vertex
 */
public Iterator iteratorBFS(T startVertex);
```

```
/**
 * Returns a depth first iterator starting with the given vertex.
 *
 * @param startVertex the starting vertex
 * @return a depth first iterator starting at the given vertex
 */
public Iterator iteratorDFS(T startVertex);

/**
 * Returns an iterator that contains the shortest path between
 * the two vertices.
 *
 * @param startVertex the starting vertex
 * @param targetVertex the ending vertex
 * @return an iterator that contains the shortest path
 *         between the two vertices
 */
public Iterator iteratorShortestPath(T startVertex, T targetVertex);

/**
 * Returns true if this graph is empty, false otherwise.
 *
 * @return true if this graph is empty
 */
public boolean isEmpty();
```

```
    /**
     * Returns true if this graph is connected, false otherwise.
     *
     * @return true if this graph is connected
     */
    public boolean isConnected();

    /**
     * Returns the number of vertices in this graph.
     *
     * @return the integer number of vertices in this graph
     */
    public int size();

    /**
     * Returns a string representation of the adjacency matrix.
     *
     * @return a string representation of the adjacency matrix
     */
    public String toString();
}
```

```
package jsjf;

import java.util.Iterator;

/**
 * NetworkADT defines the interface to a network.
 *
 * @author Java Foundations
 * @version 4.0
 */
public interface NetworkADT<T> extends GraphADT<T>
{
    /**
     * Inserts an edge between two vertices of this graph.
     *
     * @param vertex1 the first vertex
     * @param vertex2 the second vertex
     * @param weight the weight
     */
    public void addEdge(T vertex1, T vertex2, double weight);

    /**
     * Returns the weight of the shortest path in this network.
     *
     * @param vertex1 the first vertex
     * @param vertex2 the second vertex
     * @return the weight of the shortest path in this network
     */
    public double shortestPathWeight(T vertex1, T vertex2);
}
```

```
package jsjf;

import jsjf.exceptions.*;
import java.util.*;

/**
 * Graph represents an adjacency matrix implementation of a graph.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class Graph<T> implements GraphADT<T>
{
    protected final int DEFAULT_CAPACITY = 5;
    protected int numVertices;    // number of vertices in the graph
    protected boolean[][] adjMatrix;    // adjacency matrix
    protected T[] vertices;    // values of vertices
    protected int modCount;

    /**
     * Creates an empty graph.
     */
    public Graph()
    {
        numVertices = 0;
        this.adjMatrix = new boolean[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
        this.vertices = (T[])(new Object[DEFAULT_CAPACITY]);
    }
```

```
/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param vertex1  the first vertex
 * @param vertex2  the second vertex
 */
public void addEdge(T vertex1, T vertex2)
{
    addEdge(getIndex(vertex1), getIndex(vertex2));
}

/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param index1  the first index
 * @param index2  the second index
 */
public void addEdge(int index1, int index2)
{
    if (indexIsValid(index1) && indexIsValid(index2))
    {
        adjMatrix[index1][index2] = true;
        adjMatrix[index2][index1] = true;
        modCount++;
    }
}
```

```
/**
 * Adds a vertex to the graph, expanding the capacity of the graph
 * if necessary.  It also associates an object with the vertex.
 *
 * @param vertex  the vertex to add to the graph
 */
public void addVertex(T vertex)
{
    if ((numVertices + 1) == adjMatrix.length)
        expandCapacity();

    vertices[numVertices] = vertex;
    for (int i = 0; i < numVertices; i++)
    {
        adjMatrix[numVertices][i] = false;
        adjMatrix[i][numVertices] = false;
    }
    numVertices++;
    modCount++;
}
```

```
/**
 * Creates new arrays to store the contents of the graph with
 * twice the capacity.
 */
protected void expandCapacity()
{
    T[] largerVertices = (T[])(new Object[vertices.length*2]);
    boolean[][] largerAdjMatrix =
            new boolean[vertices.length*2][vertices.length*2];

    for (int i = 0; i < numVertices; i++)
    {
        for (int j = 0; j < numVertices; j++)
        {
            largerAdjMatrix[i][j] = adjMatrix[i][j];
        }
        largerVertices[i] = vertices[i];
    }

    vertices = largerVertices;
    adjMatrix = largerAdjMatrix;
}
```