SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT

Number 224, March 2006

Implementation of Graphs Using **java.util**

Part One: Preliminary Concepts

Nicholas J. De Lillo

PACE UNIVERSITY
IVAN G. SEIDENBERG
SCHOOL OF COMPUTER SCIENCE
AND INFORMATION SYSTEMS

Nicholas J. De Lillo is Professor of Mathematics and Computer Science at Manhattan College where he has taught courses in computer science, computer engineering, and software engineering at both the undergraduate and graduate levels for over thirty years. In addition, Professor De Lillo regularly teaches courses in the masters program in computer science here at Pace. He is also on the Editorial Board of *Technical Reports*.

Professor De Lillo is the author of numerous research papers and textbooks in mathematics and computer science. The texts include Advanced Calculus with Applications (1982); Computability with Pascal, co-authored with John S. Mallozzi (1984); A First Course in Computer Science with Ada (1993); Data Structures with C++, co-authored with John S. Mallozzi (1997); Object-Oriented Design in C++ Using the Standard Template Library (2002); and Object-Oriented Design in Java Using **java.util** (2004).

Professor De Lillo holds a B.S. in mathematics from Manhattan College, an M.A. in mathematics from Fordham University, and the Ph.D. in mathematics from New York University, where he was a student of Martin Davis's.

IMPLEMENTATION OF GRAPHS USING java.util
Part One: Preliminary Concepts

Nicholas J. De Lillo
Department of Mathematics and Computer Science
Manhattan College
Riverdale, New York 10471

Abstract

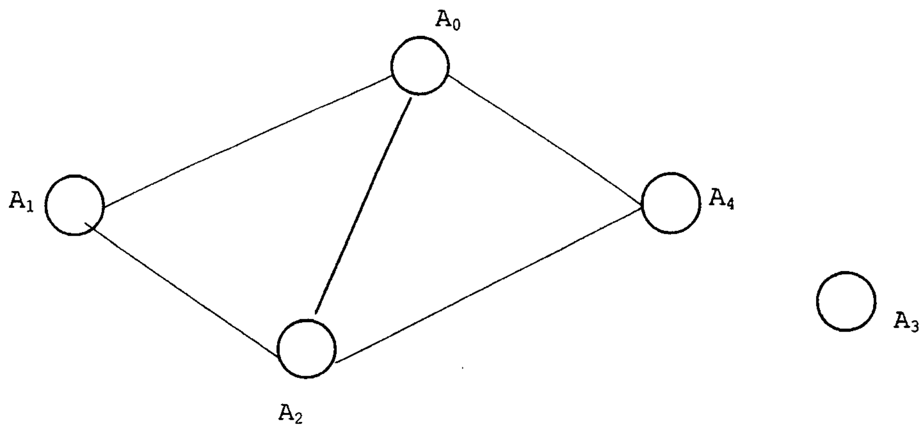
This paper presents an implementation in Java of two abstract data types for graphs: directed and undirected graphs. We view the concrete class of directed graphs as the parent class with undirected graphs as an immediate subclass. Each is designed using facilities available from Java 5.0. This paper describes the use of adjacency lists, using the predefined `List` interface and the `LinkedList` implementation class from `java.util`.

1. Introduction: Basic Ideas Concerning Graphs.

We may introduce the concept of trees as a specific type of nonlinear data structure. We may also observe that certain forms of trees, such as red-black trees and heaps, are very useful in implementing some of the predefined utilities of `java.util`, and in solving fundamental sorting problems in an efficient manner. In general, the nodes appearing on any nonempty tree share a common characteristic: either the node is the unique root node or it has exactly one parent.

There are other kinds of useful nonlinear data structures, many of which we encounter every day. Examples are road maps between cities, the modeling of a local area network, the electrical wiring and plumbing networks of a home or office building, and the description of the routes between cities flown by a commercial airline.

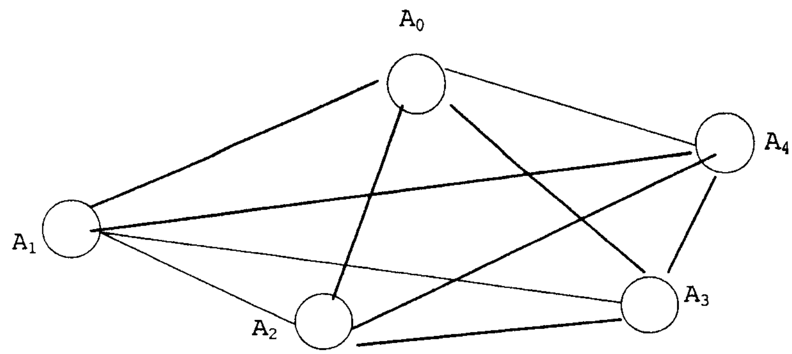
We can define a *graph* as a nonlinear structure in which there is no set root node, and for which a node can be the child of more than one parent. We will use graphs to represent a finite collection of data values expressed as *points* (or *vertices*), some of which are joined by *edges*. The edges represent relationships existing between pairs of these vertices. Indeed, suppose G names a specific graph. Then G is defined by two sets: a nonempty set V of *vertices* and a set E (possibly empty) of *edges* whose members consist of certain pairs of vertices. The order of appearance of the vertices in an edge might be significant: if so, the graph is called a *directed graph* or simply a *digraph*. In addition, edges with the same vertex at each end are permitted. Since any graph is characterized by V and E, we use the notation $G = (V,E)$. Figure 1 is a simple illustration of an (undirected) graph. Here, $V = \{A_0, A_1, A_2, A_3, A_4\}$ and $E = \{\{A_0,A_1\}, \{A_0,A_2\}, \{A_0,A_4\}, \{A_1,A_2\}, \{A_2,A_4\}\}$, where, for example, the (undirected) edge with vertices A_0 and A_1 is denoted by $\{A_0, A_1\}$. Note also that G has no edge with vertex A_3.



(Figure 1)

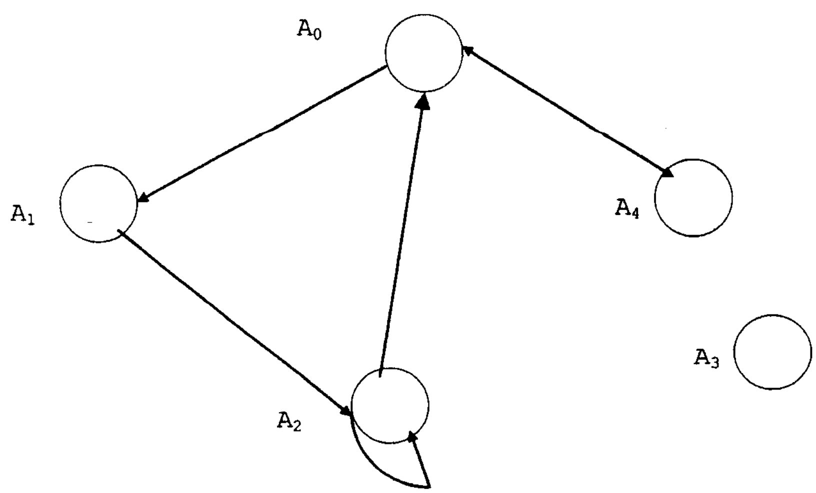
A *path* in any graph G between vertices X and Y is a finite sequence of undirected edges of G: $\{X_0, X_1\}, \{X_1, X_2\}, \ldots, \{X_{k-2}, X_{k-1}\}$, where $X_0 = X$ and $X_{k-1} = Y$. For the graph described in (Figure 1), there is a path between A_0 and A_4 given by $\{A_0, A_1\}, \{A_1, A_2\}, \{A_2, A_0\}, \{A_0, A_4\}$. We define a path to be *simple* if the path never passes through any vertex more than once. The last example of a path given is not simple, but the path $\{A_1, A_2\}, \{A_2, A_4\}, \{A_4, A_0\}$ is simple. Further, a *cycle* is a path that begins and ends at the same vertex and otherwise passes through no vertex more than once. The path $\{A_0, A_1\}, \{A_1, A_2\}, \{A_2, A_0\}$ in Figure 1 is an example of a cycle. A graph G is *connected* if a path exists between every pair of distinct vertices of G. The graph in Figure 1 is not connected because, for example, there is no path joining A_3 to any other vertex. However, if A_3 were omitted, the resulting graph would be connected.

A graph is *complete* if there is an edge connecting each distinct pair of vertices. The graph described in Figure 2 is an example of a complete graph.



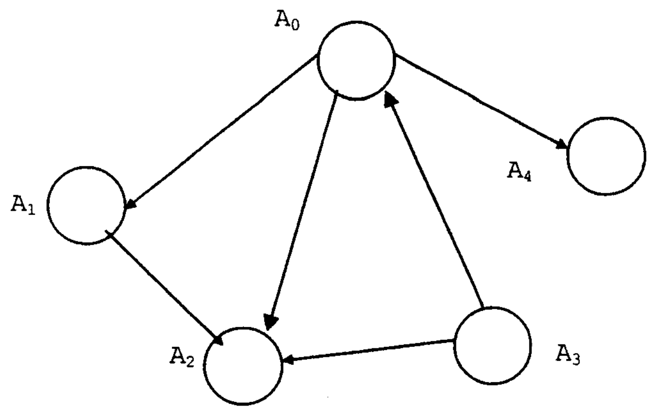
(Figure 2)

A graph is *directed* if each of its edges has a *direction*: that is, there is a well-defined initial vertex and a terminal vertex. Thus, each edge in a directed graph determines a flow from its initial to its terminal vertex. We may represent a directed graph $G = (V, E)$, where V is the set of vertices defined exactly as above and E is the set of *directed edges* represented as ordered pairs (A_i, A_j), where A_i and A_j are in V. The ordering of the pair is used to indicate that the initial vertex of the pair (A_i, A_j) is given by the left member A_i and the terminal vertex is given by the right member A_j. The graph in Figure 3 is an example of a digraph, where the arrows describe the direction of flow in each edge:



(Figure 3)

Here, $V = \{A_0, A_1, A_2, A_3, A_4\}$ and $E = \{(A_0, A_1), (A_0, A_4), (A_1, A_2), (A_2, A_0), (A_2, A_2), (A_4, A_0)\}$. We define a *cycle* in a digraph as a path $(X_0, X_1), (X_1, X_2), \ldots, (X_{k-2}, X_{k-1})$ in which $X_0 = X_{k-1}$. A cycle in a digraph is *simple* if X_0 and X_{k-1} represent the only pair of common vertices. A digraph is *acyclic* if it contains no cycles, and is commonly referred to as a *DAG*. The graph in Figure 4 is an example of a DAG.



(Figure 4)

Is this graph connected? It would be if we can find a directed path between any two of its vertices. For example, let us consider the vertex pair A_0, A_1. While there is a directed path (in fact, a directed edge) beginning at A_0 and terminating at A_1, there is no directed path beginning at A_1 and terminating at A_0. Consequently, this DAG is not connected.

2. Design Issues for Graph ADTs.

We are interested in answering several key questions regarding graphs:

- How do we define a graph ADT?
- How can we implement graphs using classes in Java?
- How are graphs *traversed*? That is, how do we move around a graph from one vertex to another? How do we implement these traversals?
- Are there efficient algorithms for graph traversals?
- Is it possible to apply the concepts of graphs to such problems as searching for a value, or sorting a sequence of values?
- Are there any important relationships existing between graphs and other data structures, such as trees?

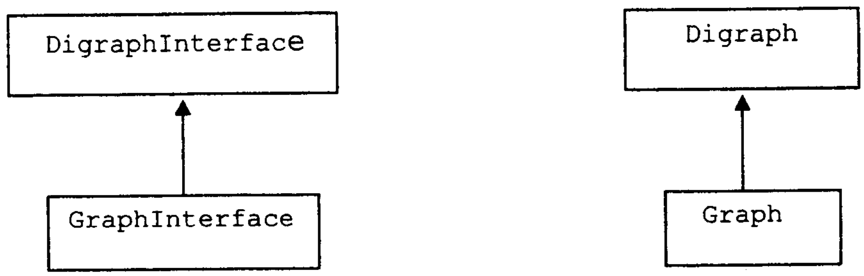
In an object-oriented design, a graph ADT should address the construction and initialization of graph objects. We will define a constructor initializing graph objects with no vertices or edges. In addition, the user interface should contain descriptions of instance methods allowing for changes in the structure of the graph object (mutators), either by inserting or removing vertices, edges, or both. Furthermore, we should be able to test whether certain vertices or edges appear in that object. Finally, the design should include the definition and implementation of methods for traversing the vertices of the graph in some particular order from some designated starting vertex.

In brief, our plan is to define interfaces and classes that implement the methods described in these interfaces for undirected graphs and for digraphs. For the sake of simplicity, we will assume at the outset that the value of each node is integer-valued and matches the subscript assigned to that node. Further, these classes should possess the following functionality:

1. constructors, allowing for the creation of graph (digraph) objects with vertices storing values of some specific `Comparable` type;
2. insertion methods for new vertices and edges in any graph (digraph) object;
3. removal methods for vertices and edges in the current graph (digraph) object;
4. a `boolean`-valued method testing whether the current graph (digraph) object is empty – that is, whether the object has any vertices or edges;
5. a `boolean`-valued method with two vertex parameters testing whether these vertices are joined by an edge in the current graph (digraph) object;
6. a method with a single vertex parameter `v` returning the collection of all vertices in the current graph (digraph) object that are adjacent to `v`;
7. methods implementing traversals of vertices in the current graph (digraph) object beginning with a designated start vertex and terminating with a given final (or goal) vertex, given as parameters.

Our design will involve the definition of two interfaces and consequently two implementation classes – one for (undirected) graphs and one for digraphs. These classes will be related by inheritance. Specifically, these interfaces and their corresponding implementation classes will have the digraph class as the base class and the undirected graph class as the subclass. This will involve a number of fundamental observations about graphs and digraphs that exploit the inheritance relationship (see (Figure 5)).

We describe an inheritance relationship between `DigraphInterface` and `GraphInterface`, naming the respective interfaces for digraph and (undirected) graph objects.



(Figure 5)

3. Initial Implementation Details of Graph ADTs in Java.

We sketch the coding of `DigraphInterface` and `GraphInterface`, leaving some of the formal coding details as an exercise.

```
public interface DigraphInterface
{
 // Tests whether current digraph is empty.
 // Returns true if so, false if not.
 public boolean isEmpty();

 // Returns the number of distinct vertices in the
 // current digraph.
 public int size();

 // Returns whether v is joined to w by an edge.
 // Returns true if so, false if not.
 public boolean isAdjacent(Object v, Object w);

 // Inserts edge from v to w.
 // Constructs from v to w, if no such edge is already present
 // and throws an exception otherwise.
 // Precondition: v, w are vertices in the current digraph.
 public void insertEdge(Object v, Object w);

 // Inserts vertex into digraph.
 // Inserts a new vertex if that vertex is not already present
 // and raises an exception if no vertex is inserted, since it is
 // already a vertex of the current digraph.
 public void insertVertex(Object v);

 // Removes vertex from current digraph if present, along with
 // all incident edges.  Raises an exception if that vertex is
 // not in the present digraph.
 public void eraseVertex(Object v);
```

```
// Removes edge from v to w if currently present in digraph.
// Precondition: v,w are vertices in current digraph.
public void eraseEdge(Object v, Object w);

// Outputs specifications of the current digraph.
public void output();

} // terminates text of DigraphInterface.
```

How do we arrive at the design decision that `GraphInterface` is to be inherited from `DigraphInterface`? The decision is based on an elementary observation: each edge in an undirected graph can be viewed as a pair of edges in a digraph whose initial and terminating vertices exchange positions. For instance, we can view the edge $\{A_i, A_j\}$ in an undirected graph as a pair of edges (A_i, A_j) and (A_j, A_i) in a digraph. Thus, in particular, inserting and removing edges in an undirected graph simply amounts to inserting and deleting edges in a digraph. Applying this idea and inheritance, we can code `GraphInterface` as

```
public interface GraphInterface extends DigraphInterface
{
 // Inserts edge connecting v and w.
 // Precondition: v,w are vertices in undirected graph.
 public void insertEdge(Object v, Object w);

 // Removes edge from v to w, if currently present in graph.
 // Precondition: v,w are vertices in current undirected graph.
 public void eraseEdge(Object v, Object w);

} // terminates text of GraphInterface.
```

How are `Digraph` and `Graph` objects represented internally? In other words, how do we describe the `private` data members implementing `DigraphInterface`? One possible representation is through *adjacency matrices.*

4. Adjacency Matrices

We define an *adjacency matrix* as a two-dimensional display of `boolean` values whose rows and columns are indexed by the vertices of the underlying digraph (or graph). If the adjacency matrix is to represent a specific (undirected) graph with vertices $A_0, A_1, \ldots, A_{n-1}$, the adjacency matrix will be a symmetric matrix with n rows and n columns, where `true` appears in position i,j (row index i and column index j) if and only if there is an edge connecting A_i and A_j in the graph. For undirected graphs, the `boolean` value in each location i,j will be the same as that for location j,i. In the case of the undirected graph shown in (Figure 1), the associated adjacency matrix is that given in (Figure 6),and the adjacency matrix for the directed graph in (Figure 3) is given in (Figure 7). Note that this last matrix is not symmetric.

(column index)

(row index)

	0	1	2	3	4
0	false	true	true	false	true
1	true	false	true	false	false
2	true	true	false	false	true
3	false	false	false	false	false
4	true	false	true	false	false

(Figure 6)

(column index)

(row index)

	0	1	2	3	4
0	false	true	false	false	true
1	false	false	true	false	false
2	true	false	true	false	false
3	false	false	false	false	false
4	true	false	false	false	false

(Figure 7)

We can use these ideas to fill in the data members of the implementation of `DigraphInterface` by defining a two-dimensional `boolean`-valued matrix type. Using adjacency matrices, it is a simple matter to determine whether an edge connecting vertices A_i and A_j exists in the current graph (or digraph). However, using adjacency matrices for graphs containing a relatively large number of vertices is inefficient. This is because the matrix that is constructed has a location for <u>every</u> pair of vertices currently lying in the graph, whether an edge exists between a pair of such vertices or not. Thus, the size of the matrix depends solely upon the number of vertices present, but in no way upon the number of edges. Ordinarily, the graphs have a relatively few number of edges compared to the number of vertices. Such matrices are known as sparse matrices, since all but a relatively small number of entries are `false`. Therefore, we seek a more space-efficient alternative to represent graphs. This occurs in the form of an *adjacency list*.

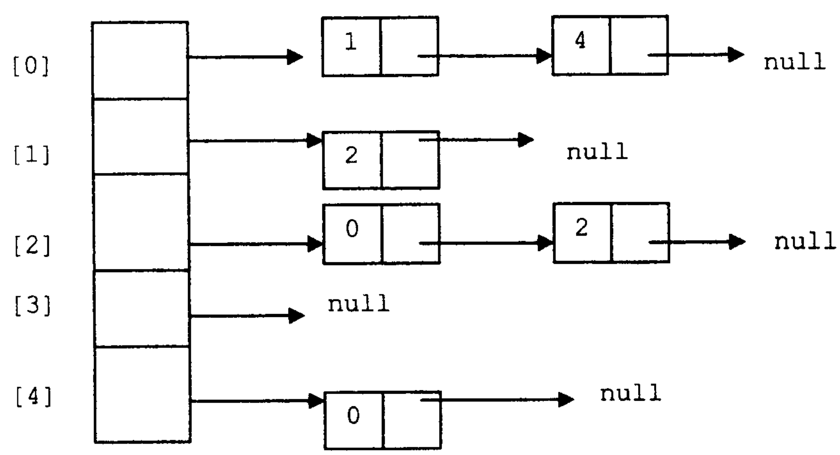
5. Adjacency Lists

Let `G` be a graph (or digraph) with vertices $V = \{A_0, A_1, \ldots, A_{n-1}\}$, and edges in the set `E`. The *adjacency list* of `G` is a one-dimensional array of `n` components, each of which is a reference to a linearly linked list or to `null`, determined by the following rules:

(I) Suppose `G` is a digraph. If (A_i, A_j) is a member of `E`; that is, if `G` contains the directed edge beginning at A_i and terminating at A_j, then a node whose `info` component is `j` appears in the list coming from component `i` in the array.

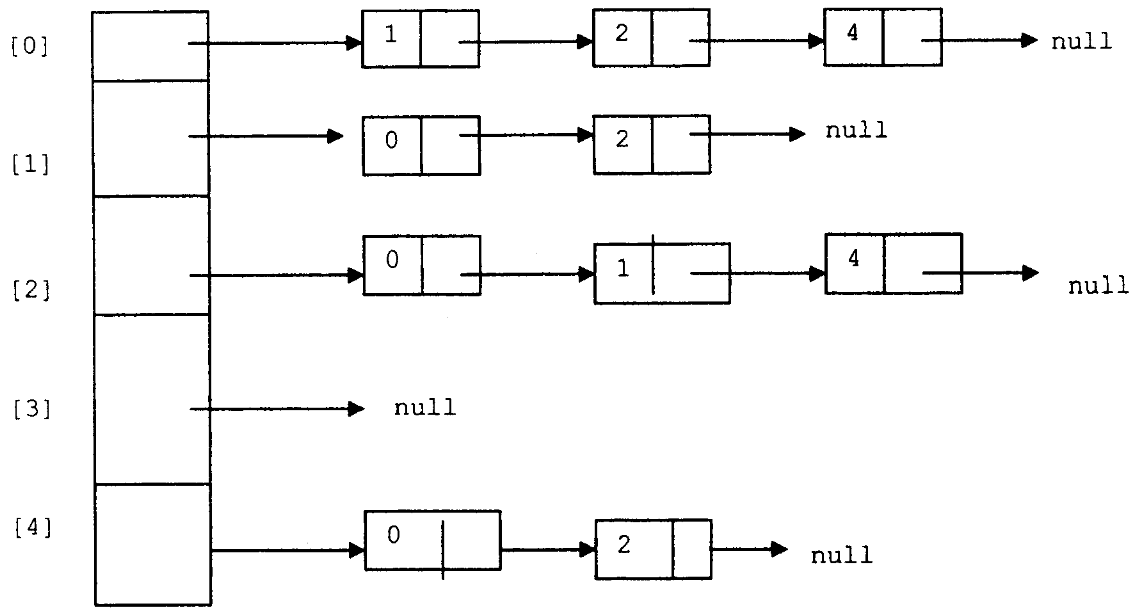
(II) If `G` is an undirected graph, and $\{A_i, A_j\}$ is in `E`, than a node whose `info` component is `j` appears in the list coming from component `i`, and a node whose `info` component is `i` appears in the list coming from component `j`.

We first illustrate the adjacency list for the digraph of (Figure 3) in (Figure 8).



(Figure 8)

The adjacency list for the (undirected) graph of (Figure 1) is represented as (Figure 9).



(Figure 9)

This results in a substantial savings of storage, since the only nodes that are required are those for which an edge exists between the vertex representing the subscript of the array and the vertex appearing as the subscript of the linearly linked list. The only other entries are `null`, where no edge exists starting at the vertex represented by the array subscript, and any vertex in the graph.

6. Graph Traversals.

In Chapter Eight, we studied several ways to traverse the nodes of a binary tree. Three of these traversals were preorder, inorder, and postorder. We may now classify each of these as a version of a *depth-first traversal*. This characterizes a traversal of the nodes of a binary tree that begins at a specific node, usually the root, and then descends the levels of the tree as much as possible until a leaf is visited, and then follows some other path (if such is available). In fact, this form of traversal is not limited to binary trees. In fact, depth-first traversal may be defined for general `n`-way trees as the natural generalization of depth-first traversals of a binary tree.

In direct contrast to this, the level-order traversal of a binary tree is an example of a *breadth-first traversal*. Here, the nodes of the tree are traversed in such a way that, beginning at the root, all nodes of a particular level are visited before progressing to nodes at the next lowest level. This process continues until all of the nodes of the tree are traversed. Again, as before, there is the natural generalization of breadth-first traversal of a general `n`-way tree.

No matter whether the tree is a binary tree or a general n-way tree, the concept of tree traversal presumes that all of the nodes of the tree are visited. How does this generalize to graphs? In the case of graph traversals, there is a predetermined ***start vertex***, and the traversal proceeds to the nodes that can be reached from that vertex. For example, if we consider the (undirected) graph of (Figure 1), any traversal beginning at any node other than A_3 will contain the other nodes A_0, A_1, A_2, A_4, in some order, but no such traversal can contain A_3. In fact, it is obvious that if a graph is connected, then any traversal visits all of its vertices.

In the case of directed graphs, a traversal starting from a designated vertex must follow the directed paths beginning at that vertex, and ending when we can go no further.

We begin by looking at an algorithm for the depth-first traversal of a directed graph. The strategy underlying depth-first traversal starting at a designated beginning vertex is to follow a path that proceeds as deeply into the graph as possible without baking up. In other words, after visiting a specific vertex, then (whenever possible) depth-first traversal moves to some adjacent vertex that has not as yet been visited.

As an example, suppose we consider the directed graph of (Figure 11.3). Then a possible depth-first traversal beginning at A_0 is given by the sequence

A_0 A_1 A_2 A_4

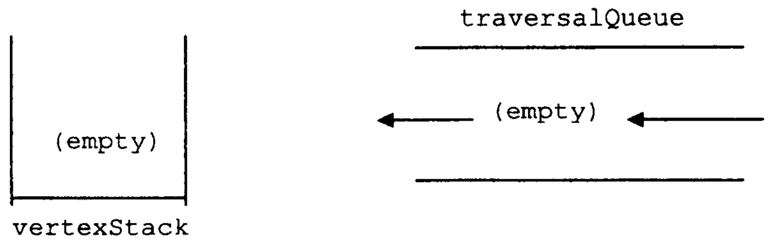
Note that although A_3 is a vertex, there is no path from A_0 to A_3.

How do we design the algorithm for depth-first traversal? We begin by fixing some vertex of the graph as the designated `startVertex`. Then we create an initially empty stack, called `vertexStack`, to hold the vertices of the graph, once these vertices have been visited. In addition, we create an initially empty queue of vertices, called `traversalQueue`. The members of `traversalQueue` will represent, in order, the traversal of the vertices of the graph completed so far. After these initializations, the algorithm for depth-first traversal proceeds as follows:

```
Mark startVertex as visited;
traversalQueue.insert(startVertex);
vertexStack.push(startVertex);
while(!vertexStack.isEmpty())
{
 topVertex = vertexStack.top();
 if(topVertex has an unvisited adjacent vertex)
 {
  nextVertex = next unvisited neighbor of topVertex;
  Mark nextVertex as visited;
  traversalQueue.insert(nextVertex);
  vertexStack.push(nextVertex);
 } // terminates text of if-clause
 else
  vertexStack.pop();
} // terminates text of while-loop
```

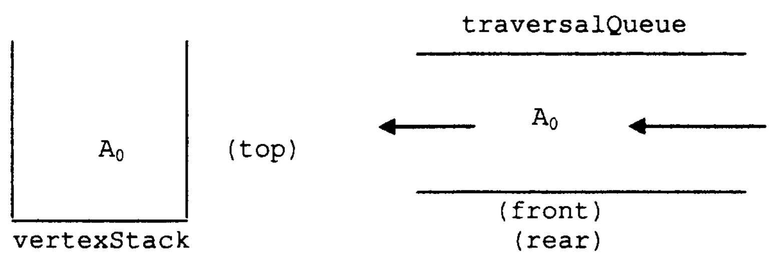
```
return traversalQueue;
```

We apply this algorithm to the graph of (Figure 11.3), starting at A_0. Initially `vertexStack` and `traversalQueue` are empty, as in (Figure 10(a)):



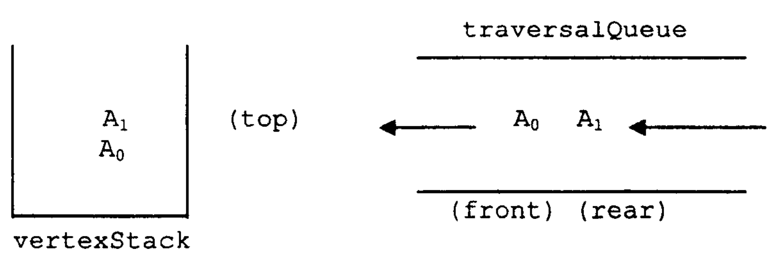
(Figure 10(a))

Beginning the traversal at `startVertex` = A_0, we mark A_0 as visited, then insert A_0 at the rear of `traversalQueue`, and push A_0 onto `vertexStack`, as seen in (Figure 10(b)):



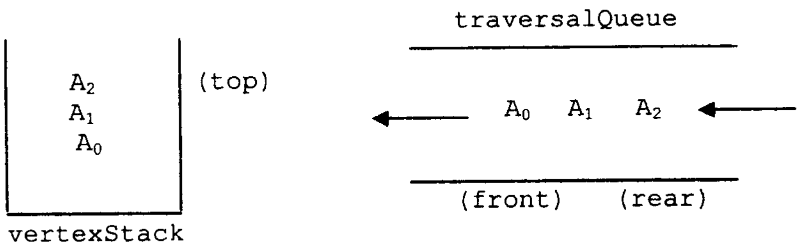
(Figure 10(b))

Since `vertexStack` is not empty, processing enters the `while`-loop. Set `topVertex` = A_0, and test whether `topVertex` has an unvisited neighbor. It has the unvisited neighbor A_1. Set `nextVertex` = A_1, mark A_1 as visited, insert A_1 at the rear of `traversalQueue`, and push A_1 onto `vertexStack`. This produces



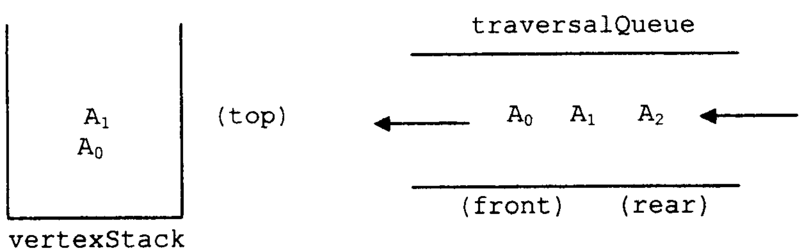
(Figure 10(c))

Since `vertexStack` is not empty, processing re-enters the `while`-loop. Set `topVertex = `A_1, and test whether `topVertex` has an unvisited neighbor. It has A_2 as an unvisited neighbor, so set `nextVertex = `A_2. Then mark A_2 as visited, insert A_2 at the rear of `traversalQueue`, and push A_2 onto `vertexStack`. This produces the results seen in (Figure 10(d)).



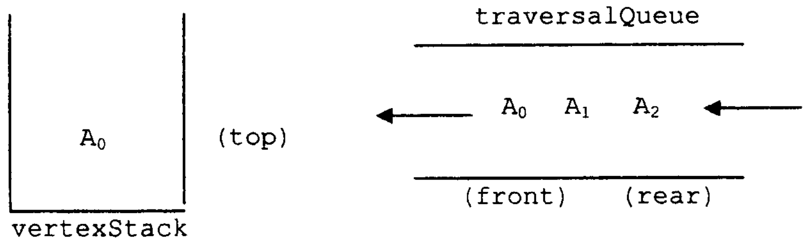
(Figure 10(d))

Since `vertexStack` is not empty, processing once again re-enters the `while`-loop. Set `topVertex = `A_2, and since A_2 has no unvisited neighbor, the `else`-clause executes. Thus, `vertexStack.pop()` executes, yielding the results seen in (Figure 10(e)):



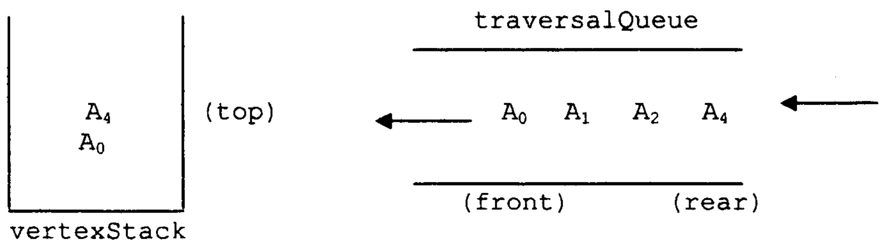
(Figure 10(e))

Since `vertexStack` is not empty, processing once again re-enters the `while`-loop. Thus, `topVertex = `A_1, and since A_1 has no unvisited neighbor, the `else`-clause executes once again, popping `vertexStack` and yielding the result described in (Figure 10(f)):



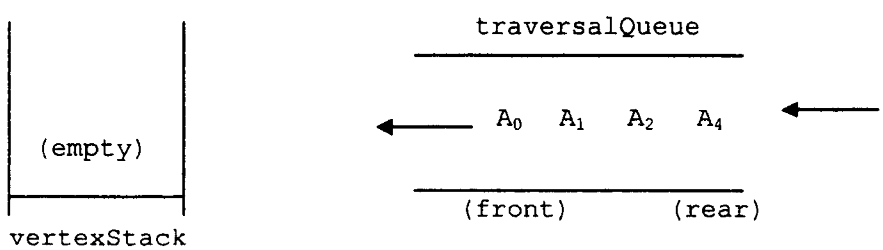
(Figure 10(f))

Since `vertexStack` is still not empty, processing re-enters the `while`-loop. Thus, `topVertex = A`$_0$, and `topVertex` has `A`$_4$ as an unvisited neighbor. Thus, `A`$_4$ is marked as visited, is inserted at the rear of `traversalQueue` and pushed onto `vertexStack`, yielding the results seen in (**Figure 10(g)**):



(Figure 10(g))

Processing re-enters the `while`-loop twice in succession, eventually producing the results depicted in (**Figure 11.10(h)**):



(Figure 10(h))

This completes execution of the `while`-loop, and control transfers to the return of the current contents of `traversalQueue`, yielding

A_0 A_1 A_2 A_4

as the result of the depth-first traversal of the graph of (Figure .3). Note that the algorithm does not permit the output of A_3, since that vertex cannot be reached by a depth-first search beginning at A_0.

Unlike its depth-first counterpart, the breadth-first traversal of a graph adopts the strategy that once a vertex A_i has been visited, every vertex adjacent to A_i is visited before attempting to visit another vertex. Using the directed graph of (Figure 3), a possible breadth-first traversal beginning at A_0 is given by

A_0 A_1 A_4 A_2

This uses the fact that each of A_1 and A_4 is adjacent to A_0 – consequently, these are visited before visiting A_2.

The algorithm for breadth-first traversal again begins by fixing `startVertex`, and employs two queues. One is `traversalQueue`, and is defined exactly as in the case of depth-first traversal. The second is `vertexQueue`, initially empty, and holding the neighbors of the current vertex. The algorithm for breadth-first traversal may then be described as

```
Mark startVertex as visited;
traversalQueue.insert(startVertex);
vertexQueue.insert(startVertex);
while(!vertexQueue.isEmpty())
{
 frontVertex = vertexQueue.front();
 vertexQueue.remove();
 while(frontVertex has an unvisited neighbor)
 {
  nextNeighbor = next unvisited neighbor of frontVertex;
  Mark nextNeighbor as visited;
  traversalQueue.insert(nextNeighbor);
  vertexQueue.insert(nextNeighbor);
 } // terminates text of inner while-loop.
} // terminates text of outer while-loop.
return traversalQueue;
```

We apply the algorithm to the graph of (Figure 11.3), beginning with A_0. Initially, `traversalQueue` and `vertexQueue` are empty, as in (Figure 11.11(a)):

vertexQueue

← (empty) ←

traversalQueue

← (empty) ←

(Figure 11(a))

Set `startVertex = ` A_0, mark A_0 as visited, then insert A_0 at the rear of each of `traversalQueue` and `vertexQueue`, as described in (Figure 11(b)):

vertexQueue

← A_0 ←

(front)
(rear)

traversalQueue

← A_0 ←

(front)
(rear)

(Figure 11(b))

Since `vertexQueue` is not empty, processing enters the outer `while`-loop. Thus, set `frontVertex = ` A_0, and remove A_0 from `vertexQueue`. Now A_0 has an unvisited neighbor A_1. Set `nextNeighbor = ` A_1, mark A_1 as visited, and insert A_1 at the rear of each of `traversalQueue` and `vertexQueue`, yielding the results as described in (Figure 11(c)):

vertexQueue

← A_1 ←

(front)
(rear)

traversalQueue

← A_0 A_1 ←

(front) (rear)

(Figure 11(c))

Since `frontVertex = ` A_0 has another unvisited neighbor A_4, processing re-enters the inner `while`-loop. Set `nextNeighbor = ` A_4, mark A_4 as visited, and insert A_4 at the rear of each of `traversalQueue` and `vertexQueue`, yielding the results depicted in (Figure 11.11(d)):

vertexQueue	traversalQueue
$\leftarrow$ A_1 A_4 $\leftarrow$	$\leftarrow$ A_0 A_1 A_4 $\leftarrow$
(front) (rear)	(front) (rear)

(Figure 11(d))

Processing cannot re-enter the inner `while`-loop at this point, since A_0 has no other unvisited neighbors. Since `vertexQueue` is not empty, processing re-enters the outer `while`-loop. Set `frontVertex =` A_1, remove A_1 from `vertexQueue`, producing the results seen in **(Figure 11(e))**:

vertexQueue	traversalQueue
$\leftarrow$ A_4 $\leftarrow$	$\leftarrow$ A_0 A_1 A_4 $\leftarrow$
(front) (rear)	(front) (rear)

(Figure 11(e))

Now since `frontVertex (=` A_1`)` has unvisited neighbor A_2, set `nextNeighbor =` A_2, mark A_2 as visited, and insert A_2 at the rear of each of `traversalQueue` and `vertexQueue`, as in **(Figure 11.11(f))**:

vertexQueue	traversalQueue
$\leftarrow$ A_4 A_2 $\leftarrow$	$\leftarrow$ A_0 A_1 A_4 $\leftarrow$
(front) (rear)	(front) (rear)

(Figure 11(f))

At this point, there are no unvisited neighbors of `frontVertex =` A_4; thus the inner `while`-loop is not entered. Since `vertexQueue` is not empty, processing re-enters the outer `while`-loop. Thus, set `frontVertex =` A_4, and remove A_4 from the front of `vertexQueue`. This situation is described in **(Figure 11(g))**:

vertexQueue traversalQueue

← A_2 ←

(front)
(rear)

← A_0 A_1 A_4 A_2 ←

(front) (rear)

(Figure 11.11(g))

Again, since `frontVertex (=` A_4`)` has no unvisited neighbor, the inner `while`-loop is not entered. Since `vertexQueue` is not empty, processing re-enters the outer `while`-loop. Set `frontVertex =` A_2, and remove A_2 from `vertexQueue`, producing the result depicted in **(Figure 11.11(h))**:

`vertexQueue`

← (empty) ←

`traversalQueue`

← A_0 A_1 A_4 A_2 ←

(front) (rear)

(Figure 11.11(h))

Since `frontVertex (=` A_2`)` has no unvisited neighbor, the inner `while`-loop is not entered. Further, since `vertexQueue` now is empty, the outer `while`-loop is not entered. Consequently, control passes to the return of the current contents of `traversalQueue`, producing

A_0 A_1 A_4 A_2

and execution terminates.

7. Implementation of the `Digraph` ADT Using Adjacency Lists.

We described the adjacency list representation and `DigraphInterface` earlier. In this section, we present some of the preliminary details of an implementation of `DigraphInterface` using adjacency lists. Here we will make use of the `LinkedList` interface of `java.util` as an indispensable tool in this construction.

In providing some of the details of the design of this implementation class, which we call `Digraph`, we introduce several constraints. The first involves the size of any `Digraph` object. We place an upper bound on the size of the graph as the size of the array used in the adjacency list representation. For example, suppose that size is stored in the `int`-valued variable `ArraySize`; that is, suppose we assume the definition

```
protected int ArraySize = <maximum possible number of vertices
                                     of any digraph>;
```

Specifically, suppose `ArraySize` is `10`. Then the maximum number of vertices allowed in any `Digraph` object is `10`. Thus, at any time during the course of processing any `Digraph` object, the maximum allowable number of vertices in that object is `10`. In order to keep track of the number of vertices in the current `Digraph` object, we use the `int`-valued variable `currSize`. As empty digraph (one with no vertices or edges) will have its value of `currSize` equal to zero, making the implementation of a constructor for this class and the implementation of the `isEmpty()` method quite straightforward. For example, the implementation of the constructor of any `Digraph` object will contain

```
currSize = 0;
```

and the coding of `isEmpty()` may be given by

```
// Tests whether the current digraph is empty.
// Returns true if so, false if not.
public boolean isEmpty()
{
 return currSize == 0;
} // terminates text of isEmpty()
```

In addition, the implementation of the `size()` method of `DigraphInterface` simply involves returning the current value stored in `currSize`, as in

```
// Returns the number of distinct vertices in the
//  current digraph.
public int size()
{
 return currSize;
} // terminates text of size()
```

A number of delicate issues must be considered using this implementation. First of all, while the value of `currSize` always yields the number of distinct vertices currently appearing in the `Digraph` object, this does not necessarily imply that all of the vertices of the digraph with subscript less than or equal to that value of `currSize` still remain in the current digraph. This is because of the possibility that several of the previous vertices may have been removed using the `eraseVertex()` method.

Secondly, imposing an upper bound on the value of `currSize` of `ArraySize - 1` introduces the possibility of an overflow condition, similar to that already seen in the array implementation of stacks. This prompts the need for the definition and implementation of an accompanying `GraphException` class. In fact, the definition of this class would have been required anyway for the underflow condition; that is, to safeguard against any attempt to remove a vertex from a currently empty `Digraph` object.

We make one final preliminary observation. To simplify matters, we assume that the vertices of any of the digraphs (or graphs) will be integer-valued. Consequently, our implementation of either `DigraphInterface` or `GraphInterface` will replace any reference to `Comparable` by `int`. With these initial observations and constraints accounted for, we continue our coding details of the implementation of `DigraphInterface`.

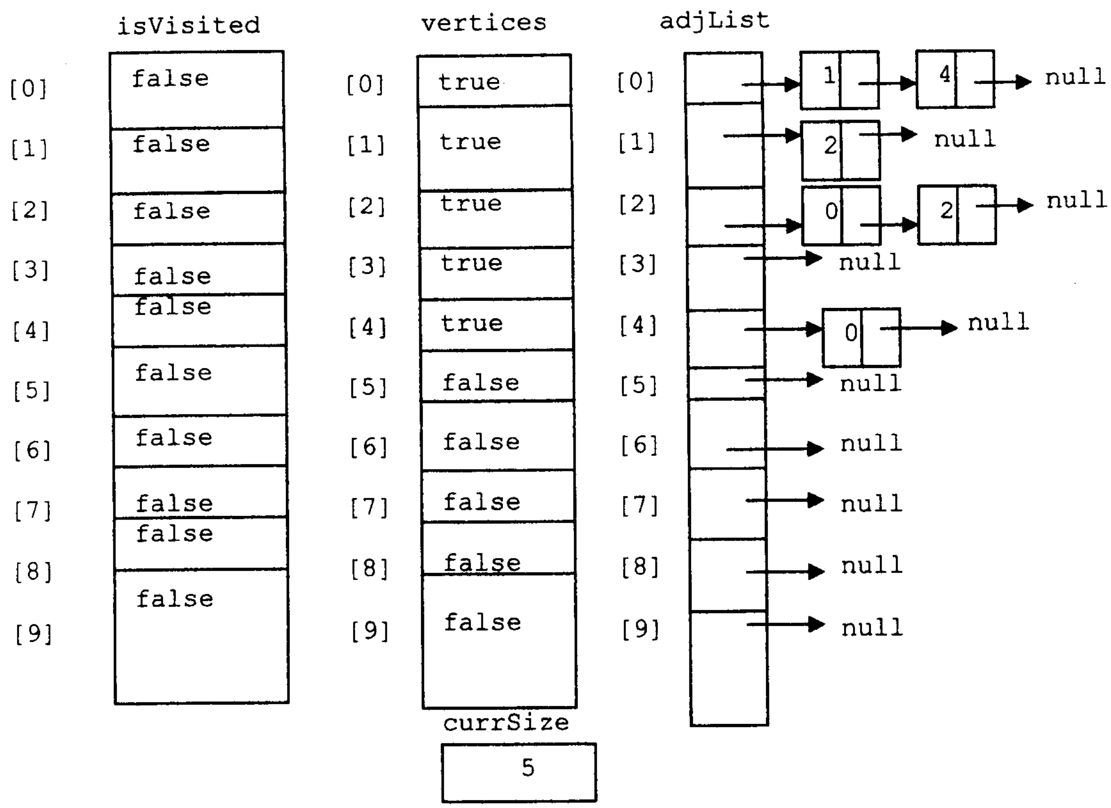
Besides the definition of `ArraySize` given above, the remaining instance variables of `Digraph` are listed as

```
protected int currSize;
protected boolean vertices[] = new boolean[ArraySize];
protected boolean isVisited[] = new boolean[ArraySize];
protected LinkedList adjList[] =
                            new LinkedList[ArraySize];[1]
```

If any component of the `vertices` array is `true`, that signifies that the subscript of that component is an actual vertex of the current digraph. Conversely, if instead that component is `false`, the corresponding subscript does not represent a corresponding vertex of the current digraph. The components of the `isVisited` array indicate whether the corresponding vertex has already been visited (if `true`) in some traversal of that digraph, and `false` if that vertex has not as yet been visited. Accordingly, that vertex must also be designated as true in the `vertices` array. Finally, we declare an array `adjList` of `LinkedList` components, formally describing the current adjacency list of the associated digraph. The next example illustrates how the digraph of (Figure 3) is represented by these instance variables, where we assume the value of `ArraySize` is `10`.

[1] Here we do not parametrize `adjList`, since this involves parametrizing an array, which is not allowed.

Example 1: The representation of the digraph object of (Figure 3) is



(Figure 13)

The code for the constructor for `Digraph` may then be described by

```
// Constructor.  Constructs empty Digraph object represented
// by an empty adjacency list, with current size zero, and
// with no vertices.  Hence, each component of the vertices
// and isVisited arrays is initialized as false.
public Digraph()
{
 currSize = 0;
 for(int index = 0; index < ArraySize; ++index)
 {
  vertices[index] = false;
  isVisited[index] = false;
  adjList[index] = new LinkedList<Integer>();
 } // terminates text of for-loop
} // terminates text of constructor.
```

Note also that we use the `LinkedList` implementation of the `List` interface from the `Collections` hierarchy of `java.util`. In so doing, we free ourselves from any source of run-time error arising from the normal processing of linearly linked lists. The reader should also note that we could just as easily used the `ArrayList` implementation of `List` instead of `LinkedList`. We leave these alternative coding details as an exercise.

The design of a number of the instance methods that follow may raise exceptions. For example, we wish to throw an exception if we attempt to insert a new vertex in a graph that is already full, causing an overflow error to occur. Similarly, an underflow condition is raised if we attempt to erase a vertex from a `Digraph` (or `Graph`) object that is currently empty. The formal coding of the `GraphException` class is similar to its counterpart for stacks, queues, and binary trees, and may then be coded as

```
public class GraphException extends RuntimeException
{
 // Constructor
 public GraphException(String str)
 {
  super(str);
 } // terminates text of constructor
} // terminates text of GraphException class
```

The code for `insertVertex` is straightforward. The `boolean` value `true` is returned just when a new vertex is actually inserted, and `false` if that vertex is already in the graph. In addition, a `GraphException` exception is thrown in the case of an overflow.

```
// Inserts vertex into digraph.
// Inserts a new vertex if that vertex is not already present
// and raises an exception if no vertex is inserted, since it is
// already a vertex of the current digraph.
public void insertVertex(int index)
{
 if(!vertices[index]) // Vertex is not in present digraph.
                      // Perform legitimate insert operation.
 {
  ++currSize;
  vertices[index] = true;
 } // terminates text of if-clause
 else if((currSize < ArraySize) && vertices[index])
        // vertex is already in digraph.  Throw exception.
  throw new GraphException("Vertex already in digraph");
 else // overflow
  throw new GraphException("Overflow - digraph is already full");
} // terminates text of insertVertex
```

Note that `insertVertex` does not make any changes in `adjList`, since `insertVertex` simply adds a new vertex to the current digraph. It does not add any new edges to that digraph. The insertion of new edges is the task of the `insertEdge` method.

The purpose of the `isAdjacent` method is to test whether the two parameters, representing vertices of the current digraph, are joined by an edge. It does not matter which of the two is the initial or the terminal vertex. The code uses the `contains` method applied to the `LinkedList` components of `adjList`, inherited from the `Collections` interface.

```
// Returns whether u is joined to v by an edge.
// Returns true if so, false if not.
public boolean isAdjacent(int u, int v)
{
 return (adjList[u].contains(v)
                    || adjList[v].contains(u));
} // terminates text of isAdjacent.
```

The `insertEdge` method first checks whether the two parameters are currently vertices of the digraph. If so, the method inserts a directed edge beginning at the value of the first parameter and terminating at the value of the second parameter.

```
// Inserts an edge from v to w.  Throws an exception
// if no edge is constructed.
// Precondition: v, w are vertices of the current digraph,
// and w is not currently joined to v by an edge.
public void insertEdge(int v,int w)
{
 // v, w are vertices of the current digraph and w is not currently
 // joined to v by an edge.
 if(vertices[v] && vertices[w] &&
  !(adjList[v].contains(w)))
  adjList[v].add(new Integer(w));
 else  // if any other condition applies, throw GraphException
  throw new GraphException("Illegal attempt to join edges.");
} // terminates text of insertEdge
```

The next method removes a vertex from the current digraph if that vertex is present, and also removes all incident edges involving that vertex. In addition, the value of `currSize` decreases by one. A `GraphException` exception is thrown if the vertex to be removed is not a member of the current digraph.

```
// Removes vertex from the current digraph if present,
// along with all incident edges.
// Precondition: parameter represents a vertex of the current digraph.
public void eraseVertex(int v)
{
 if(vertices[v]) // if v is a vertex
 {
  for(int w = 0;w < ArraySize;++w)
  // w is a vertex and v, w are connected by an
  // incident edge.
   if(vertices[w] && adjList[v].contains(w))
     eraseEdge(v,w);
   // terminates text of for-loop
  // Remove vertex from graph
```

```
  vertices[v] = false;
  // Reduce size of digraph
  --currSize;
 } // terminates text of if-clause
 else  // v is not a vertex.  Throw GraphException
  throw new GraphException("Parameter not a vertex of current
                              digraph.");
} // terminates text of eraseVertex
```

The code for `eraseEdge` removes a directed edge beginning at the value of the first parameter and terminating at the value of the second parameter. We presume that this edge exists in the current digraph, as well as each of the vertices defining that edge. The method removes a value from the appropriate adjacency list, using the `remove()` method inherited from the `Collections` interface. The formal coding of the method is given by

```
// Removes edge from v to w, if present in the current digraph.
// Precondition: v, w are vertices in the current digraph, and
// there is an edge from v to w.
public void eraseEdge(int v, int w)
{
 // There is an edge from v to w, and each of v, w is a vertex
 // in the current digraph.
 if(isAdjacent(v,w) && vertices[v] && vertices[w])
  adjList[v].remove(w);
 else  // throw a GraphException exception
  throw new GraphException("Edge removal aborted");
} // terminates text of eraseEdge
```

The next instance method outputs the specifications of the current digraph. That is, the values of the vertices, the (directed) edges, and the size of the current digraph are output, along with the decision as to whether the current digraph is empty.

```
// Outputs specifications of the current digraph.
public void output()
{
 System.out.println("Vertices are:");
 for(int index = 0; index < ArraySize; ++index)
  if(vertices[index]) System.out.print(index + "  ");
 // Terminates text of for-loop
 System.out.println();
 System.out.println("Edges are:");
 for(int index = 0; index < ArraySize; ++index)
  if(vertices[index])
   System.out.println(index + "  " + adjList[index]);
 // terminates text of for-loop
 if(isEmpty())
  System.out.println("Current digraph is empty.");
 else
  System.out.println("Current digraph is not empty.");
 System.out.println("Current digraph has " + size() + " vertices.");
} // terminates text of output method.
```

We illustrate the implementation of the `Digraph` class by constructing and then outputting the specifications of the digraph of (Figure 3). This is accomplished by

```
public static void main(String [] args)
 {
  // Constructs Digraph object of Figure 11.3.

  // Construct empty Digraph object.
  Digraph digraphObj = new Digraph();
  // Insert vertices
  digraphObj.insertVertex(0);
  digraphObj.insertVertex(1);
  digraphObj.insertVertex(2);
  digraphObj.insertVertex(3);
  digraphObj.insertVertex(4);
  // Insert edges
  digraphObj.insertEdge(0,1);
  digraphObj.insertEdge(0,4);
  digraphObj.insertEdge(1,2);
  digraphObj.insertEdge(2,0);
  digraphObj.insertEdge(2,2);
  digraphObj.insertEdge(4,0);

  // Output specifications of digraphObj
  digraphObj.output();
 } // terminates text of main method
```

The output obtained by executing this main method is

```
Vertices are:
0  1  2  3  4
Edges are:
0  [1,4]
1  [2]
2  [0,2]
3  []
4  [0]
Current digraph is not empty.
Current digraph has 5 vertices.
```

If we now continue with

```
digraphObj.erase(3);
```

and then invoke

```
digraphObj.output();
```

the resulting output continues as

```
Vertices are:
0  1  2  4
```

```
Edges are:
0  [1,4]
1  [2]
2  [0,2]
3  [0]
Current digraph is not empty.
Current digraph has 4 vertices.
```

We implement the `Graph` class as a subclass of `Digraph`. The coding of the constructor for `Graph` emulates that of `Digraph`, in that a `Graph` object is constructed with an empty adjacency list, with current size zero, and with no vertices.

```
// Constructor.  Constructs empty Graph object represented
// by an empty adjacency list with current size zero
// and with no vertices.  Hence each component of vertices and
// isVisited is initialized as false.  Constructs same objects
// as Digraph.
public Graph()
{
 super();
} // terminates text of constructor.
```

The `Graph` subclass contains a re-coding of the `isAdjacent` method, which tests whether two vertices of a `Graph` object are joined by an (undirected) edge.

```
// Returns whether v is joined to w by an (undirected) edge.
 public boolean isAdjacent(int v,int w)
 {
  return (adjList[v].contains(w)
              && adjList[w].contains(v));
 } // terminates text of isAdjacent.
```

The `insertEdge` method inserts an undirected edge joining two vertices of a graph that are not currently joined by an edge.

```
 // Inserts (undirected) edge joining v and w.
 // Precondition: v, w are edges in the current graph.
 public void insertEdge(int v,int w)
 {
  // v, w are edges in the current graph, and v and w are not
  // currently joined by an undirected edge.
  // Add an edge joining v and w by adding an edge from v to w
  // and an edge from w to v.
  if(vertices[v] && vertices[w] && !(adjList[v].contains(w))
      && !(adjList[w].contains(v)))
  {
   adjList[v].add(w);
   adjList[w].add(v);
  }
  else // if any other condition applies, raise GraphException.
   throw new GraphException("Illegal attempt to join vertices.");
} // terminates text of insertEdge.
```

There is no reason to override `eraseVertex` in the `Graph` subclass. All we need observe is that in an (undirected) graph pbject, applying `eraseVertex` from `Digraph` causes each of the `if`-clauses

```
if(vertices[w] && adjList[v].contains(w))
  adjList[v].remove(w);
if(vertices[w] && adjList[w].contains(v)))
  adjList[w].remove(v);
```

to execute, since an undirected edge is viewed as a pair of directed edges between the same two vertices. When `eraseVertex` is applied to a `Digraph` object, it may well be the case that exactly one of the `if`-caluses above executes, unless there are directed edges from the same two vertices in both directions. Further, since edges are not required in the execution of the `insertVertex` method, it is simply inherited in the same form in the `Graph` subclass.

The `eraseEdge` methos id re-coded in the `Graph` subclass, since the edge to be removed is undirected. Its code is given by

```
// Removes edge joining v and w, if currently present in
// (undirected) graph.
public void eraseEdge(int v,int w)
{
 // There are edges from v to w and w to v, and v, w are vertices
 if(isAdjacent(v,w) && vertices[v] && vertices[w])
 {
  adjList[v].remove(w);
  adjList[w].remove(v);
  // Removes edge joining v, w in undirected graph.
 } // terminates text of if-clause.
 else  // Illegal edge removal.  Throw GraphException exception.
  throw new GraphException("Illegal edge removal.");
} // terminates text of eraseEdge.
```

The `output` method is re-coded in the `Graph` subclass, since every reference to "digraph" is replaced by "graph."

```
// Outputs specifications of the current graph.
public void output()
{
 System.out.println("Vertices are:");
 for(int index = 0; index < ArraySize; ++index)
  if(vertices[index]) System.out.print(index + "  ");
 // Terminates text of for-loop
 System.out.println();
 System.out.println("Edges are:");
 for(int index = 0; index < ArraySize; ++index)
  if(vertices[index])
   System.out.println(index + "  " + adjList[index]);
```

```
 // terminates text of for-loop
 if(isEmpty())
  System.out.println("Current graph is empty.");
 else
  System.out.println("Current graph is not empty.");
 System.out.println("Current graph has " + size() + " vertices.");
} // terminates text of output method.
```

Consider the execution of the next driver program, which first constructs the (undirected) graph of (Figure 11.1), and then removes the vertex `2` from that graph.

```
public static void main(String [] args)
 {
  // Implement the (undirected) graph of Figure 11.1 of the
  // new data structures book.

  // First construct empty graph.
  Graph graphObj = new Graph();
  // Then insert vertices:
  graphObj.insertVertex(0);
  graphObj.insertVertex(1);
  graphObj.insertVertex(2);
  graphObj.insertVertex(3);
  graphObj.insertVertex(4);

  // Then insert edges:
  graphObj.insertEdge(0,1);
  graphObj.insertEdge(0,2);
  graphObj.insertEdge(0,4);
  graphObj.insertEdge(1,2);
  graphObj.insertEdge(2,4);

  // Output results
  graphObj.output();

  // Now erase vertex2:
  graphObj.eraseVertex(2);

  // Output the new result.
  graphObj.output();
 } // terminates text of main method
```

The resulting output is

```
Vertices are:
0  1  2  3  4
Edges are:
0  [1, 2, 4]
1  [0, 2]
2  [0, 1, 4]
3  []
4  [0, 2]
Current graph is not empty.
Current graph has 5 vertices.
```

```
Vertices are:
0  1  3  4
Edges are:
0  [1, 4]
1  [0]
3  []
4  [0]
Current graph is not empty.
Current graph has 4 vertices.
```

The implementation of traversals for `Graph` and `Digraph` objects, as well as implementation details for weighed graphs and digraphs, spanning trees, Prim's algorithm and Kruskal's algorithm on minimal spanning trees, and Dijkstra's algorithm on shortest paths will be discussed in the forthcoming Part Two of this paper.

8. Bibliography.

[1] De Lillo, Nicholas J. Object-Oriented Design in Java Using `java.util`, Brooks/Cole, 2004.

[2] Koffman, Elliot B. and Paul A. T. Wolfgang, Objects, Abstraction, Data Structures and Design Using Java 5.0, Wiley, 2005.

[3] Savitch, Walter Absolute Java, (Second Edition), Addison-Wesley, 2006.

The Ivan G. Seidenberg School of Computer Science and Information Systems Pace University

Technical Report Series

The School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. *Technical Reports* is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from your submission. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
The Ivan G. Seidenberg School of Computer Science and Information Systems
Goldstein Academic Center
Pace University
861 Bedford Road
Pleasantville, NY 10570-2799