A Laboratory Sequence for Data Structures

Thomas E. O'Neil
Computer Science Department
University of North Dakota
oneil@cs.und.edu

Abstract

It is common for computer science curricula to include a course on data structures and algorithm analysis following a two-semester sequence that covers the fundamentals of computer programming. Emphasis in such a course may be placed on the mathematical techniques of algorithm analysis or on the design and implementation of algorithms that perform fundamental operations on the underlying data structures. For instructors who prefer an emphasis on implementation, this paper describes a sequence of four laboratory projects that reinforce learning about fundamental operations on common data structures, about trade-offs between flexibility and efficiency, and about the techniques of abstraction, implementation-hiding, and code re-use in the object-oriented approach to software development.

The four projects are centered on implementing the following structures: general-purpose lists, priority queues, general trees, and undirected graphs. These projects are consistent with the organization of most standard textbooks for courses on data structures and algorithms. Each project has three components: an abstract definition of a data structure and some fundamental operations on it, a concrete implementation of the abstract definition, and a graphical user interface for testing the implementation. In the simplest delivery of each project, the instructor provides the abstract definition and the testing interface, and the student is responsible for supplying the concrete class that implements the abstract definition. Students may also be engaged as a group in designing the abstract definitions. Java is used as the development language for the projects presented here. The graphical testing interfaces use Java Swing classes. Depending on how the testing interfaces are presented to the students, these projects may also provide an introduction to the coding of graphical user interfaces.

Introduction

A course on data structures and algorithm analysis is a central component of most computer science curricula. The course typically follows a two-semester sequence that covers the fundamentals of computer programming. At that point students are ready to explore the science behind programming. They learn methods for mathematical analysis of the time and space required to run programs. They learn to implement the structures that serve as models for the central tasks of data processing. They learn the best-known algorithms for various operations on these structures, and they are introduced to the hierarchical classification of algorithms based on time and space analysis.

Coursework for the study of algorithms and data structures may include a mix of mathematical analysis and computer programming, and, depending on the position of the course in the prerequisite structure of the curriculum, more emphasis may be placed on one or the other. When it immediately follows the introductory programming sequence, it may be important to require significant laboratory work involving the implementation of the data structures and algorithms being studied. For instructors who prefer an emphasis on implementation, this paper describes a sequence of four laboratory projects that reinforce learning about fundamental operations on common data structures, the time and space requirements for these operations, and trade-offs between flexibility and efficiency. In addition, the projects illustrate use of the object oriented paradigm in software development. The project sequence emphasizes data and object abstraction, implementation, code reuse, and a clean separation of the user interface from the data model and its functional components.

The Project Sequence

In the sequence of four projects described here, students produce general implementations of the following fundamental structures: a simple list, a priority queue, a general tree, and an undirected graph. All the projects have a similar format involving three components: an abstract definition of a data object and some fundamental operations on it, a concrete implementation of the abstract definition, and a graphical user interface for testing the implementation. The abstract definition is presented as a Java abstract class with a set of abstract methods providing general purpose operations on the object. No implementation is given for the data object or its methods in the abstract class – it encapsulates a set of specifications which the students must use to create an implementation. This set of specifications is also used to design a graphical user interface that can be used to exercise all the operations defined for the data object. The testing interface and the implementation class are developed independently of one another. Both rely only on the method specifications in the abstract class. The testing interface does not even know the name of the implementation class – the implementation class is selected by the user and dynamically loaded at runtime. Thus the implementation of the object is strictly hidden from the client application that uses it.

In the simplest delivery of the projects, the student is responsible for producing only the concrete class that implements the abstract definition. The abstract class and the client

interface code are supplied by the instructor. There are several options for giving the students more responsibility for specifications and testing. Students may be engaged as a group in discussing and designing the abstract definitions. They can also be given some level of responsibility for producing the client application code. The client programs presented here employ graphical interfaces using Java Swing classes. Assuming students have not seen Java Swing before, the client programs can be given to them a day before the project deadline. Students can use the GUI clients in the final phase of testing their implementations. The rationale for withholding the test code (until the end) is to force students to develop the object implementation code without knowing how a specific client program will use it. Students are expected to produce their own test programs as needed to do incremental testing during development of the implementation.

In each of the four projects, the abstract class definition leaves the student free to design the private fields and structures that will be used to implement the object. Whenever possible, the collection of abstract methods is defined without suggesting any particular implementation strategy. The students are left to make choices involving the perennial trade-offs of flexibility vs. efficiency and time vs. space. The final project defines an abstract graph class that uses the abstract list class and the abstract tree class as return types for some of its methods. Students are thus required to reuse the code for lists and trees that they developed in the earlier projects. This drives home the principle that in the world of object oriented programming, objects should be defined as generally as possible and reused whenever it is appropriate.

The List Project

In the first project, students are asked to implement a simple list of objects. The AbstractList class contains an integer field called *position* that indicates which item in the list is considered to be the current item. The initial value for *position* is zero. There are two methods for inserting new items in the list (see Figure 1), *insert(Object item)* and *append(Object item)*. The *append()* method simply places the new item at the end of the list. The *insert()* method inserts the new item at the current *position*. There are several methods dedicated to using and modifying the current *position*: *getPosition()*, *setPosition(int pos)*, *setStart()*, *setEnd()*, *prev()*, and *next()*. The *remove()* method removes and returns the item at the current position, and the *getItem()* method just returns the item at the current position. The *findPosition(Object target)* method searches the list for one that matches the specified object and returns either the *position* of the matching object or -1 if there is no match. The *length()* method returns the number of items currently in the list.

Of course there are several predefined versions of lists in the Java Foundation Classes, such as ArrayList and LinkedList, but the students are prohibited from using these in their implementation. They must build the implementation from primitive Java types and internal classes of their own creation. The implementation class must declare an array of Objects or define a list node class and dynamically allocate new nodes to hold newly inserted objects. And the classical trade-offs must be considered. An array implementation makes the *position*-modifying methods constant time, but insertion in the

```
abstract class AbstractList
{
   protected int position = 0;
         // The list's internal position marker.
   public abstract void clear();
         // Reinitialize the list
   public abstract void insert(Object item) throws InsertionFailure;
         // Insert an item at the current position in the list.  Throws InsertionFailure if the operation is
         // not successful.
   public abstract void append(Object item) throws InsertionFailure;
         // Append an item at the end of the list.  Throws InsertionFailure if the operation is not
         // successful.
   public abstract Object remove() throws EmptyListError, PositionError;
         // Remove the item at the current position marker.  Throws EmptyListError if the list is empty.
         // Throws PositionError if the current position is at the end of the list.
   public abstract void setStart();
         // Set position marker to 0.
   public abstract void setEnd();
         // Set position marker to length of list.
   public abstract void prev() throws PositionError;
         // Move position marker one step left.  Throws PositionError if already at start.
   public abstract void next() throws PositionError;
         // Move position marker one step right.  Throws PositionError if already at end.
   public abstract int length();
         // Return number of items in the list.
   public abstract boolean setPosition(int pos) throws PositionError;
         // If pos or more elements are in the list, set the position marker to pos and return true.
         // Throws PositionError if pos exceeds the number of items in the list.
   public abstract int getPosition();
         // Return the current position.
   public abstract int findPosition(Object target);
         // Return the position of the target item in the list.  If the target is not in the list, return -1.
   public abstract Object getItem() throws EmptyListError;
         // Return the item at the position marker without removing it.  Throws EmptyListError if there
         // are no items in the list.
}
```

Figure 1. Method signatures for the AbstractList class.

middle requires linear time to move items over and make a space for the new one. A linked implementation allows constant time insertion, but modifying the current *position* can take linear time. The array implementation also raises the question of whether the list has a maximum capacity, and what to do if the capacity is exceeded.

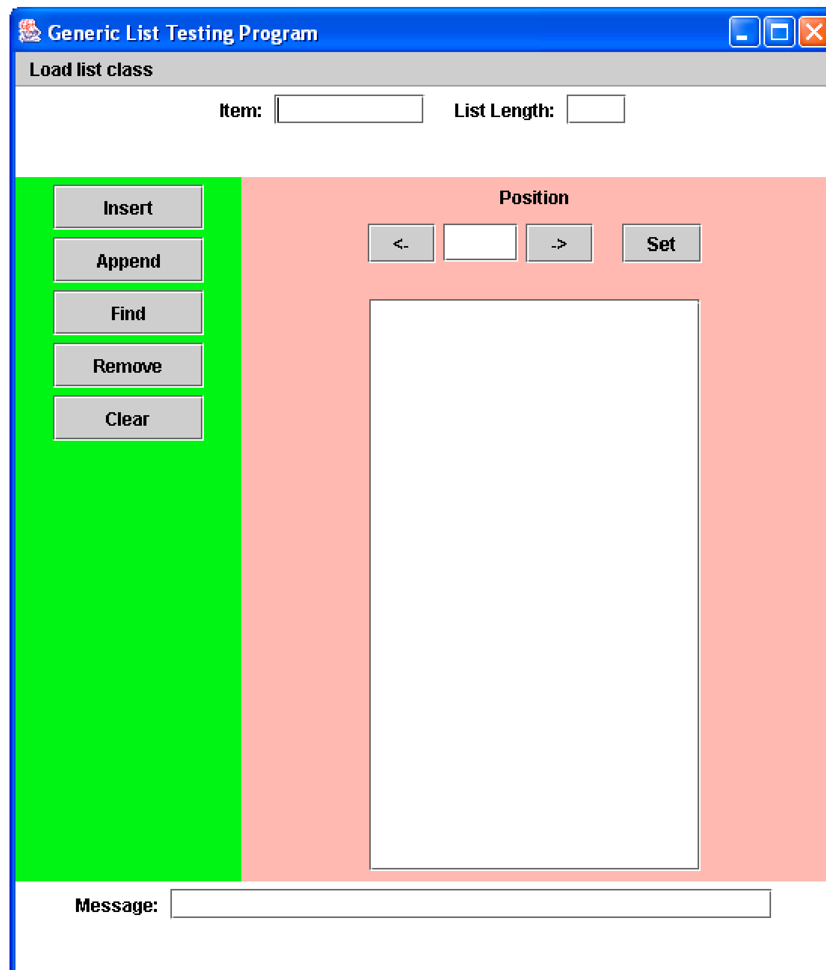
The AbstractList class includes the definition of three exceptions to be thrown by the implementation class and caught by the client applications (see Figure 2). The exception *InsertionFailure()* is thrown if an insertion fails for any reason. If the implementation uses dynamically allocated nodes, there should be no need to use this exception. For array implementations, students have the option of throwing this exception when the array is full. The exception *EmpyListError()* is thrown if an attempt is made to get or

```
public class InsertionFailure extends Exception
{
    public InsertionFailure() { super("Unable to insert item"); }
}
public class EmptyListError extends Exception
{
    public EmptyListError() { super("The list is empty"); }
}
public class PositionError extends Exception
{
    public PositionError() { super("Invalid list position"); }
}
```

Figure 2. Exception classes defined within AbstractList.

remove an item from an empty list. The exception *PositionError()* is used for any operation that would cause the current *position* to become invalid, such as a *next()* operation when the *position* is already at the end of the list.

The client application used to test the list implementation is a relatively simple program



that uses Java Swing interface components (see Figure 3). It contains methods that call AbstractList methods in response to user actions. It also calls AbstractList methods in a *refreshDisplay()* method that is executed after every list operation. It exercises every method of AbstractList, and the application methods that make these calls are localized within the application code. The application allows the user to insert, delete, and search for strings in a list. The list contents are displayed in a list box (a Java JList object) after every operation. The user can also use arrow keys or selections from the list box to modify the current position in the list. If exceptions occur during any operations, a message is displayed in a text field.

The Priority Queue Project

A priority queue is a list of objects from which removal always returns the object with the

```
abstract class AbstractPriorityQueue
{
   protected Comparator prioritytest;
   protected Comparator searchtest;
   public void setPriorityComparator(Comparator pc) { prioritytest = pc; }
   public void setSearchComparator(Comparator sc) { searchtest = sc; }
   public abstract void clear();
         // Reinitialize the queue
   public abstract void insert(Object item) throws InsertionFailure;
         // Insert an item in the queue.  The Comparator prioritytest is used to compare the new item
         // with other items in the queue and determine the proper insertion point.  Throws
         // InsertionFailure  if the operation  is not successful.
   public abstract Object remove() throws EmptyQueueError;
         // Remove and return the highest priority item from the queue.  Throws EmptyQueueError
         //  if the queue is empty.
   public abstract int length();
         // Return number of items currently in the queue.
   public abstract boolean contains(Object key);
         // Returns true if some object in the queue matches key object, as determined by the
         //  Comparator searchtest.  If no object in the queue matches the key, false is returned.
   public abstract Object getItem(int position) throws EmptyQueueError;
         // Return the item at the indicated position without removing it. Throws EmptyQueueError if
         // there are no items in the queue. This method is included to allow the contents of the queue to
         // be displayed.  The item at position 0 should be the highest priority item, but for the other
         // items, the position does not represent a ranking according to priority.
}
```

Figure 4. The Methods of the AbstractPriorityQueue Class.

highest priority. The user has no control over the position of the objects in the queue, so the collection of methods is much smaller than for a general-purpose list. As shown in Figure 4, there is one insertion method and one removal method. The user is also given a boolean *contains(Object key)* method that can be used to determine whether objects with a specified key value are already in the queue.

This project illustrates the use of the predefined Comparator interface from the java.util package. The implementation of AbstractPriorityQueue must be capable of storing and comparing any objects that are placed in it. Many objects, like number and strings, have a natural ordering. But it can't be assumed that the objects in the queue are numbers or strings. So the implementation code must rely on the client application to supply a comparator that can be used to compare the objects in the queue. The Comparator interface specifies a comparison method *compare(Object o1, Object o2)* that returns an integer following the convention of the string comparison function in the C language. The priority queue implementation requires two of these – one for comparing priorities and one for comparing whatever key field in the objects will be used to identify them. So the specification of AbstractPriorityQueue includes two methods that can be used by the client application to supply comparators: *setPriorityComparator(Comparator pc)* and *setSearchComparator(sc)*. The client code must call these methods after instantiating a new priority queue, preferably before objects are inserted in the queue. The implementation class uses *pc.compare(o1, o2)* or *sc.compare(o1, o2)* when it needs to compare objects in the queue.

AbstactPriorityQueue defines exceptions similar to those of AbstractList and the interface for the queue-testing client can be the same as the list-testing code (Figure 3). Specifications for the project can also include restrictions on how the queue is

```
abstract class AbstractTree
{
   public abstract void clear();
         // Reinitialize the tree
   public abstract void insertItem(Object item, int i) throws PositionError;
         // inserts a new item as the i'th child of the current node, where the index of the leftmost child
         // is 0.  Throws PositionError if the current node doesn't already have at least i-1 children.
   public abstract void insertTree(AbstractTree subtree, int i) throws PositionError;
         // inserts the subtree as the i-th subtree of the current node, where the index of the leftmost
         //subtree  is 0.   Throws PositionError if the current  node doesn't already have at least i-1
         // children.
   public abstract Object removeItem() throws CursorError, RemovalError;
         // Removes and returns the current node.  Throws CursorError if the cursor is null.  Throws
         // RemovalError if the current node has children.
   public abstract AbstractTree removeTree() throws CursorError;
         // Removes and returns the current node and all its subtrees.  Throws CursorError if the
         // cursor  is null.
   public abstract AbstractCursor getCursor();
         // Returns the tree's cursor object.  Returns null if the tree is empty.
   public abstract String toString();
         // Returns a string representation of the tree (e.g. a preorder listing of item strings with
         // parentheses to indicate subtree structure).
   public abstract int weight();
         // Returns the number of items in the tree.
   public abstract int height();
         // Returns the height of the tree.
}
```

Figure 5. The method of the AbstractTree class.

implemented. For example, students can be required to implement both insertion and removal of items with a time complexity of O(*lg n*), where *n* is the number of items in the queue. This rules out simply implementing the queue as an ordered list and requires the underlying implementation to employ a search tree or a heap.

The General Tree Project

In the third project of the sequence, students are asked to implement a general tree of objects. In a general tree, there is no restriction on the number of subtrees under any node. It provides a model for any hierarchical structure, such as a file system directory. The class AbstractTree specifies methods for insertion and removal of individual items and of subtrees (see Figure 5). It has a *toString()* method that can be called to get a string representing the entire tree, such as a parenthesized preorder traversal.

The insertion and removal methods rely on a position marker within the tree that

```
public abstract class AbstractCursor
{
    // This class is used to specify and control which node is the current node in the tree.  When the
    // first node is inserted, it automatically becomes the current node.
    public abstract void toRoot();
            // Makes the root the current node.
    public abstract void toChild(int position) throws PositionError;
            // The specified child of current becomes the new current node. Throws PositionError
            // if current does not have the specified child.
    public abstract void toParent() throws CursorError;
            // The parent of current becomes the new current node.  Throws CursorError if
            // current has no parent.
    public abstract void toRight() throws CursorError;
            // The right sibling of current becomes the new current node.  Throws CursorError if
            // current has no right sibling.
    public abstract void toLeft() throws CursorError;
            // The left sibling of current becomes the new current node.  Throws CursorError if
            // current has no left sibling.
    public abstract boolean isRoot();
            // Returns true if the current node is the root, false otherwise.
    public abstract boolean isLeaf();
            // Returns true if the current node is a leaf, false otherwise.
    public abstract boolean hasRightSibling();
            // Returns true if the current node has a right sibling, false otherwise.
    public abstract boolean hasLeftSibling();
            // Returns true if the current node has a left sibling, false otherwise.
    public abstract Object getItem();
            // Returns the object stored at the current node.
    public abstract String getPath() throws CursorError;
            // Returns as a string the sequence of child indexes that lead from the root to the
            // current node (e.g. 1/0/2 for some node on the third level).  The root's path is an
            // empty string.
}
```

Figure 6. The methods of the AbstractCursor class.

indicates what node is the current node. The AbstractTree class contains an internal class called AbstractCursor that serves this purpose. It has a number of methods that allow modification of the tree cursor and retrieval of cursor-related information, such as a method that returns a string representation of the path from the root of the tree to the current node (see Figure 6). The cursor object is made available to the client application through the tree's *getCursor()* method. The AbstractCursor class plays a role similar to that of the predefined interface ListIterator in the java.util package. Three exceptions are defined for use by the tree and cursor implementations: *CursorError()*, *PositionError()*, and *RemovalError()*.

Students must create an implementation that contains a subclass for tree nodes and employs arrays of nodes, single nodes dynamically allocated as needed, or some combination of the two strategies. No matter which implementation they choose, they discover that some operations are made easy, while others become more difficult. The use of fixed-size arrays of child nodes places an unauthorized restriction on the number of children per node.

The client program for testing and editing trees (as shown in Figure 7) contains navigational buttons for controlling the cursor, buttons for insertion and removal of individual items, and buttons for insertion and removal of subtrees. After each operation, the entire tree is displayed using the string representation returned by the *toString()* method. The editing client has the capability of storing one removed subtree for later reinsertion at another location in the tree.

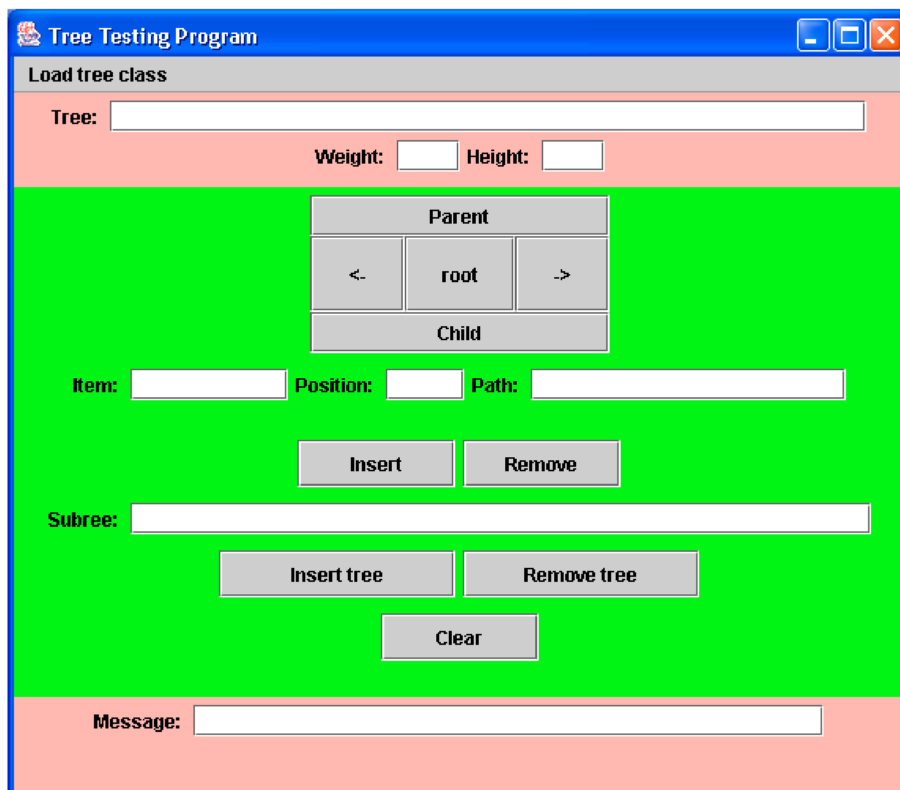


Figure 7. The tree viewer/editor.

The Undirected Graph Project

The final project in the sequence is the implementation of an undirected graph. The methods of the AbstractGraph class are shown in Figure 8. Unlike the list and tree classes, the graph class has no internal position marker or iterator. The user is required to refer to vertices in the graph by integer indices that are assigned in order of insertion, starting with index 0. Removal of vertices can cause renumbering, so the user is supplied a *getIndex(Object item)* that can be used to retrieve the index of any object inserted in the graph. AbstractGraph includes methods for insertion and removal of vertices and for

```
abstract class AbstractGraph
{
   public abstract void clear();
         // Reinitializes the graph
   public abstract int addVertex(Object item);
         // Adds a vertex for the new item and returns its index.  The first vertex added has index 0.
   public abstract int getIndex(Object item) throws MissingError;
         // Returns the index of the specified item.  Throws MissingError if the item is not in any graph
         // vertex.
   public abstract Object getItem(int vindex) throws IndexError;
         // Returns the item at the vertex with the specified index.
   public abstract void addEdge(int vindex1, int vindex2) throws IndexError;
         // Adds an edge from the vertex with vindex1 to the vertex with vindex2.
   public abstract Object removeVertex(int vindex) throws IndexError;
         // Removes the vertex with the specified index.  Removing a vertex may cause the indexes of the
         //  remaining vertices to change.
   public abstract void removeEdge(int vindex1, int vindex2)
                                                   throws IndexError, MissingError;
         // Removes the specified edge.  Throws MissingError if there is no such edge.
   public abstract AbstractTree getSpanningTree(int vindex) throws IndexError;
         // Returns a spanning tree of items rooted at the specified vertex.  The spanning tree contains
         //all items in the graph.
   public abstract AbstractList getNeighborhood(int vindex) throws IndexError;
         // Returns a list of the items that are neighbors to the specified vertex, excluding the vertex
         //  itself.
   public abstract int getNeighborCount(int vindex) throws IndexError;
         // Returns the number of vertices adjacent to the specified vertex.
   public abstract int getVertexCount();
         // Returns the number of vertices in the graph.
   public abstract boolean areNeighbors(int vindex1, int vindex2) throws IndexError;
         // Returns true if the specified vertices are joined by an edge.
   public abstract boolean areConnected(int vindex1, int vindex2) throws IndexError;
         // Returns true if there is a path between the specified vertices.
   public abstract String getShortestPath(int vindex1, int vindex2) throws IndexError;
         // Returns the shortest path from the first vertex specified to the second, or a null string if no
         // path exists.  The path string contains vertex indexes separated by some delimiter, e.g. 3-5-
         // 10-2 as path from vertex 3 to vertex 2.  If the start and end vertices are the same, the path
         // returned is the shortest cycle from the vertex back to itself.
}
```

Figure 8. The methods of the AbstractGraph class.

insertion and removal of edges. It contains a *getSpanningTree(int vindex)* method that returns a spanning tree of objects rooted at the specified vertex and a *getNeighborhood (int vindex)* method that returns a list objects in the vertices adjacent to the specified vertex. Note that the return types of these two methods are AbstractTree and AbstractList, respectively. Reinforcing the principle of code reuse in the object-oriented paradigm, the final project in the sequence requires reuse of the implementations created for the first and third projects.

The graph class also contains boolean methods that determine whether two vertices are adjacent and whether two vertices are connected by some path. Finally, a method must be implemented to return a string representing the shortest path between two vertices. Students must employ breadth-first or depth-first traversal strategies to implement these methods. They must also create a structure to represent the graph: an adjacency matrix or some form of adjacency list. And again, the set of methods is robust enough to make the classical trade-offs apparent regardless of which implementation they choose.

The graph editing client (see Figure 9) provides panels for insertion and deletion of vertices and for insertion and deletion of edges. It displays a list of the vertices in the graph, a current vertex and its neighborhood, and the string representation of a spanning

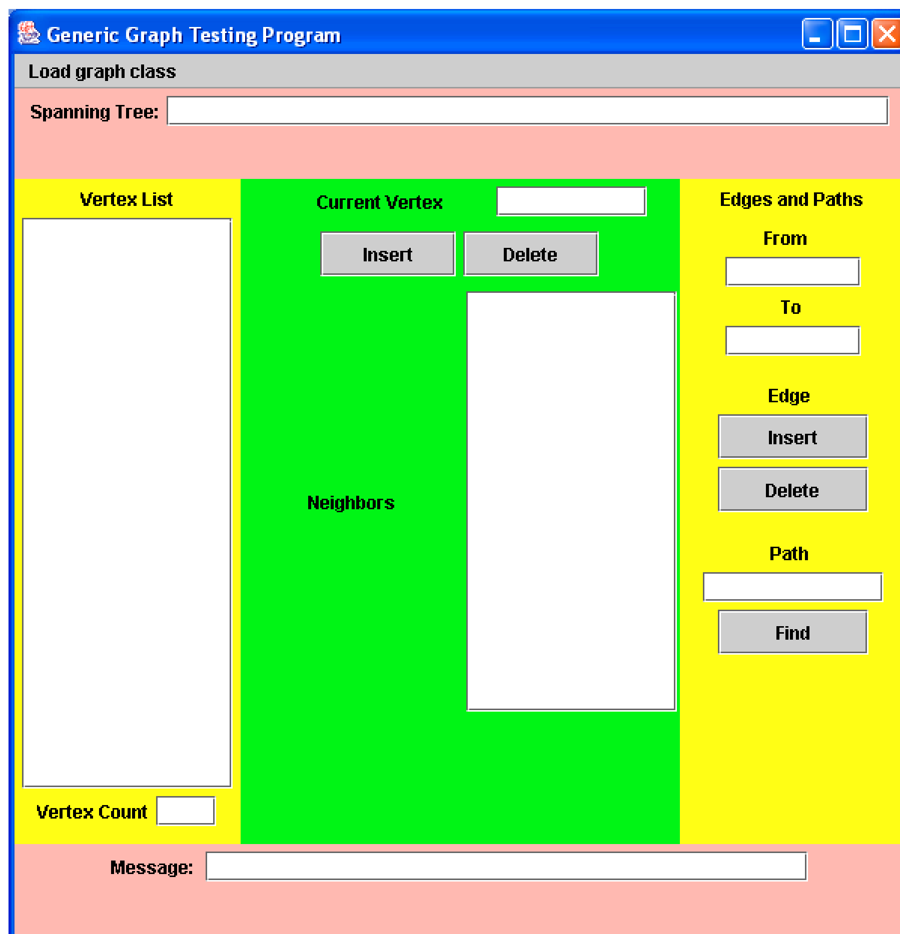


Figure 9. The graph viewing/editing program.

tree rooted at the current vertex. The interface will also display a path between any two vertices.

Levels of Student Responsibility

The projects described here were used in CSci 242 Algorithms and Data Structures in the Fall 2003 term at the University of North Dakota. In each project, the students were responsible for producing an implementation of the abstract class. They also participated in a design-level review of the abstract class when the assignment was first presented. The students were told that it was their responsibility to write auxiliary code to test their implementations incrementally, and that a GUI client application would be made available to them the day before the due date for final testing. The instructor used the same GUI client application to evaluate the student implementations.

This approach worked fairly well. With the first few projects, several students expressed disbelief and indignation when their code, which they had already tested, did not work with the GUI client application. Over the course of the semester, however, they learned to read through the virtual machine's screen dump to find the method calls that produced the errors. Invariably, they discovered that the errors resulted from sequences of operations they hadn't previously tried in their testing.

Students were apprehensive about reading the client source code that used Java Swing components. However, some class time was dedicated to viewing and discussing the organization of that code. All the GUI clients had the same structure, and the calls to the implementation methods were highly localized in the code. Once they learned to recognize the common code structure, they could ignore the GUI-building and event-capturing modules and more easily trace the method calls that resulted in errors.

Conclusion

The project sequence described here serves the following purposes:

a) It extends the students' maturity as programmers by requiring the implementation of increasingly complex structures.
b) It teaches students some fundamental structures and algorithms of computing and forces them to grapple with classical trade-offs in designing their implementations.
c) It reinforces the object oriented programming paradigm.
d) It gives an introduction to Java Swing and illustrates strict separation between the implementation of a data model and the client interface that uses the model.

The sequence is designed to provide a meaningful lab component to a course with considerable classroom emphasis on mathematical analysis of time and space requirements of computer algorithms. The combination of classroom analysis and laboratory implementation provides a solid foundation for must upper-level courses in Computer Science.

References

Eckstein, R., M. Loy, and D. Wood (1998). *Java Swing*. Sebastopol, CA: O'Reilly and Associates.

Flanagan, D. (1999). *Java in a Nutshell: A Desktop Quick Reference*. Sebastopol, CA: O'Reilly and Associates.

Goodrich, M., and Roberto Tamassia (2004). Data Structures and Algorithms in Java. Hoboken, NJ: John Wiley & Sons, Inc.

Java 2 Platform, Standard Edition, v 1.4.2 API Specification (2003). <www.java.sun.com>: Sun Microsystems, Inc.