Accessing the Internal Organization of Data Structures in the JDSL Library*

Michael T. Goodrich[1], Mark Handy[2], Benoît Hudson[2], and Roberto Tamassia[2]

[1] Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218
goodrich@cs.jhu.edu

[2] Department of Computer Science
Brown University
Providence, Rhode Island 02912
{mdh, bh, rt}@cs.brown.edu

Abstract. Many applications require data structures that allow efficient access to their internal organization and to their elements. This feature has been implemented in some libraries with *iterators* or *items*. We present an alternative implementation, used in the Library of Data Structures for Java (JDSL). We refine the notion of an item and split it into two related concepts: *position* and *locator*. Positions are an abstraction of a pointer to a node or an index into an array; they provide direct access to the in-memory structure of the container. Locators add a level of indirection and allow the user to find a specific element even if the position holding the element changes.

1 Introduction

In using a data structure, the user must be granted some form of access to its elements, preferably an efficient form. In some cases, the access can be very limited: in a stack, for example, we can look only at the top element. But in many cases, we need a more general mechanism to allow the user some handle on the elements. In an array structure, we can use indices. In a linked structure, we can use pointers to the nodes.

From the perspective of object-oriented design, however, we need to restrict access to the internals of the data structures. Otherwise, the user can make a container invalid by modifying data required to maintain the consistency of the data structure. For instance, if a list gives the user pointers to its nodes, the user should not be able to modify explicitly the successor and predecessor pointers of the node.

* Work supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grants CCR–9625289, CCR–9732327, and CDA-9703080.

1.1 Previous Work

Two well-known abstractions for safely granting access to the internals of data structures are *iterators* and *items*. STL [13], JGL [20] and the Java Collections Framework of Java 1.2 [18] use iterators. LEDA [11, 21] uses items.

Iterators provide a means to walk over a collection of elements in some linear order. At any time, an iterator is essentially a pointer to one of the elements; this pointer can be modified by using the incrementing functions the iterator provides. In C/C++, a pointer (other than to `void`) fits this profile: it points to a position (an address in memory), and the increment operation '++' makes the pointer point to the next element in an array. In a linked list, a simple class with a pointer to a node of the list will do. The access function will return the element of the node, and the increment function will move to the next node in the list. This discussion is of an STL *input iterator*. Other iterators in the STL hierarchy can also move backwards, write to the current element, and so on. The capabilities of iterators do fulfill the requirement to hide the data structure organization from the user, while still allowing the user to refer to an element efficiently. They also make it easy to act on every element of a list.

Iterators, however, have their limitations. For example, if a program gets an iterator over a container and subsequently modifies the container, the question arises as to how the iterator will behave. The modification may be reflected by the iterator, or it may not, or the iterator may simply be invalidated. Furthermore, iterators work well on linearly arranged data, but it is not clear how to extend them to non-linear data structures, such as trees and graphs.

Items, as they are referred to in LEDA, are the nodes of a data structure. In order to keep the user from having access to the internal structure, all fields of an item are private; all access is done through the container, passing in the item. This works quite well to describe arbitrary linked structures, including linear data structures. This approach does not preclude the use of iterators: one can easily be written by having it store a current item, which it changes on incrementing. For array-based structures, LEDA does not provide items. For uniformity's sake, it would be possible to provide them, either by storing a pointer to an item-object in each slot of the array or by creating an item class that simply hides an array index.

1.2 Overview

We have refined the notion of an item and split it into two abstractions: *position* and *locator*. Positions abstract the notion of a place in memory, and provide direct but safe access to the internals of a data structure. Locators are a handle to an element; while the position of the locator within a container may change, the element will not.

In the following sections, we discuss these concepts at greater length, and introduce two container types which use positions and locators in their interfaces. We then present our implementation of these abstractions in the framework of JDSL, a Java library that provides a comprehensive set of data structures.

Examples show how one would use positions and locators, notably in the context of Dijkstra's shortest-path algorithm. We also note some of the constant-factor costs of the design and how they can be reduced. Finally, we discuss the current state of JDSL in the contexts of teaching and of research.

2 Positions and Locators

We have developed a pair of notions related to the notion of items. On the one hand, we can talk about a *position* within a data structure; on the other hand, we can talk about a *locator* for an element within a data structure. A position is a topological construct: one position might be before another in a sequence, or the left child of another in a binary tree, for example. The user is the one who decides the relationships between positions; a container cannot change its topology unless the user requests a change specifically. An element in a container can always be found at some position, but as the element moves about in its container, or even from container to container, its position changes.

In order to have a handle to an element regardless of the position at which the element is found, we introduce the concept of a locator. Whereas the position-to-element binding can change without the user's knowledge, the locator-to-element binding cannot. Thus, positions closely resemble items, and the distinction between position and locator is new.

These two concepts can be distinguished more clearly by understanding, at a high level, their implementations. A position refers to a "place" in a collection. More precisely, a position is a piece of memory that holds

- the user's element
- adjacency information (for example, next and prev fields in a sequence, right-child and left-child fields in a binary tree, adjacency list in a graph)
- consistency information (for example, what container this position is in)

Thus, a position maps very closely to a single node or cell in the underlying representation of the container. Methods of some containers are written primarily in terms of positions, and a method that takes a position p can immediately map p to the corresponding part of the underlying representation. Section 3.1 gives more detail about the implementation of positions in JDSL.

A locator is a more abstract handle to an element, and its implementation does not map so directly to the underlying memory. Three principles constrain the implementation:

- For correctness, the container must guarantee that the locator-element binding will remain valid, even if the container moves the element to a different position.
- All containers, even containers with very abstract interfaces, must be realized ultimately in memory, and memory is positional.
- Methods of some containers are written in terms of locators, and a method that takes a locator ℓ needs to be able to map ℓ quickly to the underlying (positional) representation of the container.

Therefore, in implementation, a locator must hold some positional information (often a pointer to a position). The container uses this information to map the locator to the container's representation, and the container takes care to update the information when the locator changes position. Section 3.2 gives more detail about the implementation of locators in JDSL.

2.1 Positional Containers and Key-Based Containers

We distinguish between two kinds of containers, positional and key-based. In JDSL, the interfaces of positional containers are written primarily in terms of positions, and the interfaces of key-based containers are written primarily in terms of locators.

A *positional container* is a graph or some restriction of a graph. Examples are sequences, trees, and planar embeddings. A positional container stores the user's elements at vertices of the graph, and in some cases also at edges of the graph. Its interface maps very closely to its implementation, which usually involves linked data structures or arrays. The positions mentioned in its interface map to individual nodes or cells of the implementation.

A *key-based container* is an abstract data type that stores key-element pairs and performs some service on the pairs based on the keys. Its in-memory representation is entirely hidden by its interface; nothing is revealed about where it stores its elements. Since the hidden representation must be positional, a key-based container maps the locators mentioned in its interface to the representation using positional information stored in the locators.

2.2 Examples

We illustrate these concepts using the distinction between a binary tree and a red-black tree. Drawings of these two data structures might look the same, but the semantic differences are important: An unrestricted binary tree allows the user to modify the connections between nodes, and to move elements from node to node arbitrarily. Hence, a binary tree is positional, and its interface is written in terms of positions. A red-black tree, on the other hand, manages the structure of the tree and the placement of the elements on behalf of the user, based on the key associated with each element. It prevents the user from arbitrarily adjusting the tree; instead, it presents to the user a restricted interface appropriate to a dictionary. Hence, a red-black tree is key-based, and its interface is written in terms of locators. A locator guarantees access to a specific element no matter how the red-black tree modifies either the structure of the tree or the position at which the element is stored.

In both cases, the container provides access to its internal organization. The binary tree provides direct (but limited) access to its nodes (as positions). The red-black tree provides locators that specify only the order of the keys in the dictionary, without contemplating the existence of nodes.

A binary tree would support operations like these:

```
leftChild (Position internalNode) returns Position;
cut (Position subtreeRoot) returns BinaryTree; // removes and returns a subtree
swap (Position a, Position b); // exchanges elements and locators
```

A red-black tree would support operations like these:

```
first() returns Locator; // leftmost (key,element) pair
insert (Object key, Object element) returns Locator; // makes new locator
find (Object key) returns Locator;
replaceKey (Locator loc, Object newKey) returns Object; // old key
```

3 JDSL

JDSL, the Data Structures Library for Java, is a new library being developed at Brown and at Johns Hopkins, which seeks to provide a more complete set of data structures and algorithms than has previously been available in Java. Other goals are efficiency, run-time safety, and support for rapid prototyping of complex algorithms. The design of containers in JDSL is closer to that of LEDA and CGAL [5] than to that of STL and JGL. In addition to the standard set of vectors, linked lists, priority queues, and dictionaries, we provide trees, graphs, and others. We also have algorithms to run on many of the data structures, and we are developing a set of classes for geometric computing. The implementations of data structures are hidden behind abstract interfaces.

The interface hierarchy in figure 1 has two major divisions. One, between inspectable and modifiable containers, allows us to restrict subinterfaces to include only methods which are valid. The other, between PositionalContainer and KeyBasedContainer, implements the distinction we drew in section 2.1 between these two types of containers. Finally, two interfaces not pictured here implement the distinction between Position and Locator. PositionalContainer interfaces are written mainly in terms of Position, while KeyBasedContainer interfaces are written mainly in terms of Locator.

3.1 Position

The Position interface is present in most methods of positional containers, and implementations of the Position interface are the basic building blocks of those containers. Positions are most commonly used to represent nodes in a linked structure (such as a linked list), or indices in a matrix structure (such as a vector). In data structures which have different types of positions—such as vertices and edges in graphs—we use empty subinterfaces of Position useful only for type-checking purposes. While positions are closely tied to the internal workings of their container, their public interface is limited, so that they are safe to return to user code. Most operations must actually be done by calling upon the position's container, passing in the position as an argument—for example, Position

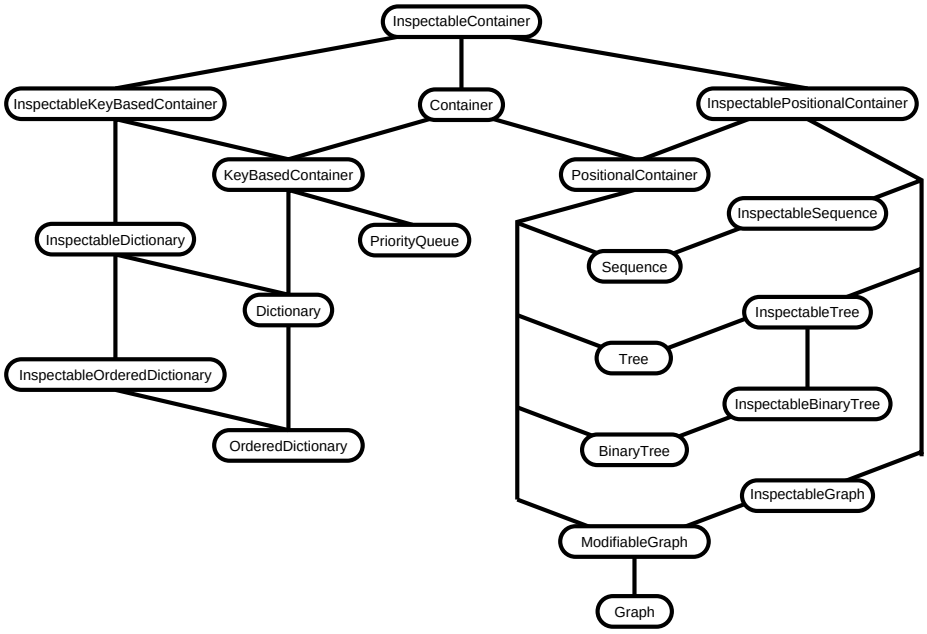

Fig. 1. A partial hierarchy of the JDSL interfaces.

Sequence.after(Position), which returns the position following the argument in the list. Only operations applicable to positions from any container are included in the interface.

```
public interface Position extends Decorable {
  public Object element() throws InvalidPositionException;
  public Locator locator() throws InvalidPositionException;
  public Container container() throws InvalidPositionException;
  public boolean isValid();
}
```

Note that locators are present even in positional containers, so the Position interface provides a way to find the locator stored at a given position.

In most implementations, the position classes are inner classes of the container that will use them, and have private methods allowing access by the container to their internals.

Since positions are so strongly tied to the internal structure of the data structures which contain them, they have no meaning once removed from a container. Hence, a deleted position is marked as invalid and any subsequent use of it will raise an exception. In addition, containers throw an exception if they are passed a position of the wrong class, a null position, or a position from a different container.

The Position interface is typically implemented either by a node in a linked data structure, or by a special object in an array-based data structure (see figure 2). We use a special object for two reasons: to store consistency checking information, and to adhere to the Java requirement that only objects can implement an interface. Without that requirement, we could use simply an integer index as the position.

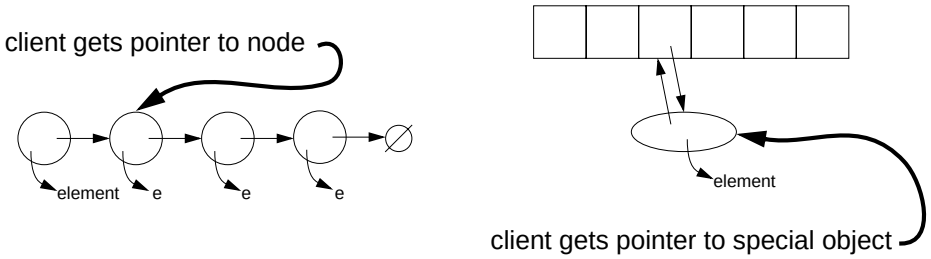

Fig. 2. Two sample implementations of positions in a sequence.

As an example of using Positions in a simple application, we can look at code which reverses a Sequence (such as a linked list or a vector):

```
void reverse(Sequence S) {
    if(S.isEmpty()) return;
    Position front = S.first(), back = S.last();
    for(int i=0; i<S.size()/2; i++) {
        S.swap(front, back);
        front = S.after(front);
        back = S.before(back);
    }
}
```

Positions are thus fairly similar to LEDA's items. The latter have only private members, and are friends (in the C++ sense) of their container, just as we hide most Position members except to the container.

Unlike an iterator, a position is always tied to a particular node, and cannot be used directly to traverse a data structure. Instead, an iterator would use a pointer to a position as its method of indexing into the data structure.

3.2 Locator

From the point of view of the user, Locators are essentially pointers to elements. They allow the user to efficiently locate an element within a data structure, typically in constant time, and to make a request of a data structure to act on an element. A locator provides a contract to its user that it will always be associated with a specific element even if the position of that element changes.

```
interface Locator extends Decorable {
  public Object element();
  public Container container();
  public boolean isContained();
  public Object invalidate();
  public boolean isValid();
}
```

Locators are typically used to access *(key, element)* pairs within KeyBasedContainers. Upon inserting an element, a locator is created and returned to the user. The locator can also be retrieved, in a dictionary, by calling find(Object). This locator can then be used to refer to the pair in constant time: the user need not search for it for each operation. For instance, the call replaceElement(Locator, Object) will typically take constant time, whereas a search would likely have taken logarithmic time. Similarly, replaceKey(Locator, Object) in a priority queue realized as a binary heap will take logarithmic time, while searching would take linear time.

In order to guarantee these time bounds, the container must be able to immediately map the locator to the underlying, positional data structure. Because of this constraint, we have found it useful to implement only one class, UniversalLocator, which stores its element and its position within its container. The user will only see the objects through the Locator interface above. However, the UniversalLocator class provides some additional methods which allow containers to modify the locator.

This allows us to implement the concept of *universality* of locators across a set of containers. A locator created by one container can be removed from it, and moved to another container within the set. The locator remains a valid handle to the element, even when it is not contained in any data structure. In JDSL, all containers accept UniversalLocator unless they specifically state otherwise.

Universality allows a user to move locators from one data structure to another, as long as all the data structures support UniversalLocator. This is useful, for example, in sorting applications, where the position-to-element binding is broken, but the locator-to-element binding need not be. By inserting and removing locators rather than elements, the author of the sort can allow the user to keep useful handles to the elements through the locators. For example, consider the code for a PQ-Sort in figure 3.

Universal locators require some overhead: a Position object needs to be created. In most cases, this overhead is acceptable: preliminary experiments indicate that our red-black tree, written under the paradigm of universal locators, is at least as fast as the red-black tree in JGL or JDK 1.2 [3]. But the interfaces do not require that universality be implemented by all locators. This allows potentially smaller or faster data structures in applications where the generality is not needed. For instance, a red-black tree locator could store its color, children, and parent, rather than requiring a separate position to store that information (see figure 4).

```
public void sort(Sequence S, Comparator c) {
    PriorityQueue Q = new VectorHeap(c); // use your favorite PriorityQueue

    // remove all elements from S, but retain the same locators
    while(!S.isEmpty()) {
        Locator loc = S.first().locator() ; // loc is in S
        S.removeFirst(); // after this, loc is in no container
        Q.insert(loc); // now loc is in Q
    }

    // remove all elements from Q in ascending order
    while(!Q.isEmpty()) {
        S.insertLast(Q.removeMin()); // move the locator from Q to S
    }
}
```

Fig. 3. A PQ-sort implementation. Note how the locators can be preserved as they move from the sequence into the priority queue and back.

4 Labeling and Iterating in JDSL

4.1 Decorable

Many algorithms need to store extra state associated with the positions or elements of the data structures they use. For example, in a breadth-first or depth-first search over a graph, we need to mark nodes as visited. Essentially, we want to add decorations or attributes to the positions of our data structures.

We can imagine the system as a two-dimensional matrix of values, with positions indexing the rows and attribute names indexing the columns [15]. We need to be able to find the value of a specific attribute for a specific position. The three solutions commonly used now are:

1. to copy the data structure into one which has the extra instance variables,
2. to use an external dictionary indexed by the positions (the *column-based* option), and
3. to associate an internal dictionaries with each positions (the *row-based* option); the dictionaries are indexed by attribute names.

The first solution clearly involves a large amount of copying of data structures. In a situation where a large number of relatively quick algorithms are being used, this could affect speed significantly.

The second solution is implemented in LEDA's node_map, edge_map, node_array, and edge_array. In JDSL, the column-based solution can be implemented simply by instantiating a hash table or other dictionary, using the hashcode() function of the positions as the key into the dictionary. Thus, no library support is required for the column-based solution.

JDSL supports the third solution; library support is required if the row-based solution is to exist at all. Position extends the Decorable interface, which requires that all positions have a dictionary associated with them. This is also the approach taken in the ffGraph library [17], which calls the attributes "labels". Under the assumption that there are more positions than there are decorations, the row-based solution is asymptotically faster than the column-based in the worst case. In the average case, the efficiencies are the same.

4.2 Enumerations

While one use of iterators is to encapsulate a position, they are also useful in iterating over an entire container. To support this functionality, JDSL requires that its data structures provide Enumerations over interesting sets of elements, positions, and locators.

One difficulty we noted with an iterator is its behavior upon modification of the underlying container. To avoid the issue, we specify that Enumerations provide a *snapshot* of the data. That is, subsequent modifications to the data structure over which we are enumerating do not modify outstanding enumerations. In general, a user who asks for an enumeration will use every element of it, so we are not affecting asymptotic efficiency. Also, we have developed optimizations to further reduce the cost (see section 4).

5 Examples of JDSL Containers

To bring together all the concepts we have discussed so far, we present examples implementations of three data structures. The following schematic figures show our standard implementations of binary trees, red-black trees, and general graphs:

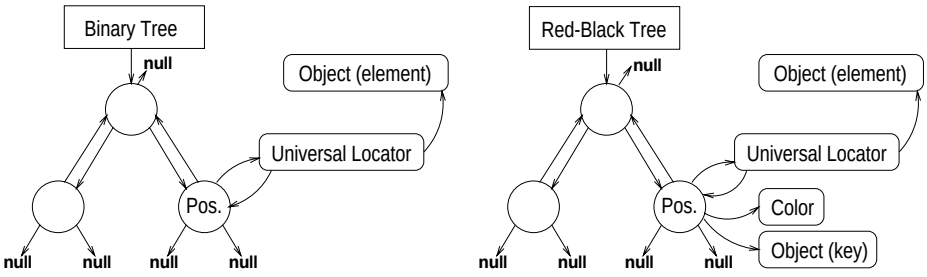

Fig. 4. Implementation of a BinaryTree, and implementation of a Dictionary as a red-black tree. Note the two have very similar diagrams, although the user sees very different interfaces.

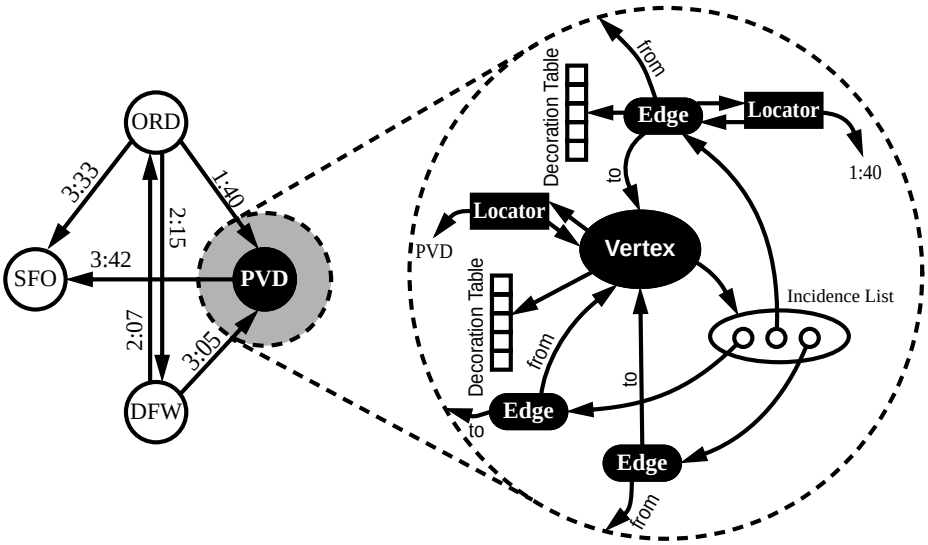

Fig. 5. Implementation of a Graph. Note Vertex and Edge are both empty subinterfaces of Position.

6 Example of an Algorithm

We present, as an example, a simple implementation of Dijkstra's algorithm for finding shortest paths; the implementation is based on [4]. The algorithm is typically used to illustrate the programming methodology of a library [15]. We assume the edges of the graph are weighted, using the Decorable mechanism, with non-negative Integer weights.

Here we use both positions and locators. The positions of a graph are its edges and vertices—this is reflected in JDSL by having Edge and Vertex extend Position. It is by using these that the algorithm traverses the graph. The locators we use allow us to find the vertices in the priority queue. We insert pairs *(distance, vertex)* into the queue, which represent the shortest yet known distance to the vertex. A vertex is chosen only if it is the closest vertex to the source that we have not yet encountered. When we choose a vertex, we discover all its edges. Through these edges, we may find a shorter path to another vertex. If we do, we need to update the key of the vertex within the queue (since the vertex should be removed from the queue earlier than previously believed), which we can do efficiently because we kept a locator to the vertex.

```
/**
 * Dijkstra's algorithm on an InspectableGraph, using integer weights.
 * The output is a list of edges which define the shortest path from s to t.
 */
public class Dijkstra {
    public Dijkstra(InspectableGraph g, Vertex s, Vertex t, Object weight) {
        graph_  = g;
        weight_ = weight;
        s_ = s; t_ = t;

        initialize();
        run();
        buildPath();
        cleanup();
    }

    private void run() {
        while( !pq_.isEmpty() ) {
            Vertex v = (Vertex) pq_.removeMin();
            if(v==t_) return; // if true, we've found the shortest path to t

            int distance = distance(v);

            // for all outgoing edges...
            for(Enumeration edges = graph_.outIncidentEdges(v);
                    edges.hasMoreElements() ; ) {
                Edge e = (Edge) edges.nextElement();
                Vertex dest = graph_.destination(e);

                int cumulative = weight(e) + distance;

                if( cumulative < distance(dest) ) { // relax the edge if applicable
                    Integer newdist = new Integer(cumulative);
                    pq_.replaceKey( locator(dest), newdist);
                    dest.set(incoming_, e);
                }
            }
        }
    }

    // Create appropriate decorations and put vertices into the heap
    private void initialize() {
        for(Enumeration verts = graph_.vertices(); verts.hasMoreElements(); ) {
            Vertex v = (Vertex)verts.nextElement();
            v.create(locator_ , pq_.insert(INFINITE, v));
            v.create(incoming_, null);
        }
        // we have set s to have infinite distance, but it has 0 distance
        pq_.replaceKey(locator(s_), ZERO);
    }
```

```
    // Build the list of edges from source to destination
    private void buildPath() {
        Vertex v = t_; // we're going backwards...
        while(v!=s_) {
            Edge e = incoming(v);
            output_.insertFirst(e); // so we insert at the head
            v = graph_.origin(e);
        }
    }

    // Clean up: destroy all decorations we created.
    private void cleanup() {
        for(Enumeration verts = graph_.vertices(); verts.hasMoreElements(); ) {
            Vertex v = (Vertex)verts.nextElement();
            v.destroy(locator_);
            v.destroy(incoming_);
        }
    }

    // Return the path as an enumeration of edges from s to t.
    public Enumeration getPath() { return output_.elements(); }

    // some accessors
    private Locator locator(Vertex v) { return (Locator)v.get(locator_); }
    private int distance(Vertex v) {
        return ((Integer)pq_.key(locator(v))).intValue();
    }
    private Edge incoming(Vertex v) { return (Edge)v.get(incoming_); }
    private int weight(Edge e) { return ((Integer)e.get(weight_)).intValue(); }

    // data structures we will be using
    private InspectableGraph graph_;
    private PriorityQueue pq_ = new VectorHeap(new IntegerComparator());
    private Sequence output_ = new NodeSequence();

    // the source and destination of the path
    private Vertex s_, t_;

    // keys for decorations
    private Object weight_ ;
    private Object locator_ = new Object();
    private Object incoming_ = new Object();

    // "infinity" and zero for the purposes of this program
    private static final Integer INFINITE = new Integer(Integer.MAX_VALUE);
    private static final Integer ZERO = new Integer(0);
}
```

7 Optimizations

JDSL's use of locators and positions is asymptotically efficient; algorithms do not suffer increased complexity, either in the worst case or in the average case, because access via locators and positions requires constant time. But a library which claims to be all-powerful invariably suffers from constant-factor efficiency losses: there are clear tradeoffs among power, elegance, and speed. We have developed some optimizations which should make JDSL competitive with other data structures libraries in terms of speed. Experiments are in progress to compare actual performance of JDSL with that of other libraries.

7.1 Lazy Allocation of Locators

One concern is caused by the requirement for each element to be associated with a locator. This obviously creates a large number of extra objects. Further, in PositionalContainers, the locators often go unused—they may be powerful, but many applications do not need the power. Therefore, most data structures in the library follow an *allocate-on-use* policy for creating locators. If the user asks to get a locator, via the Position.locator() method, for example, one is created. Otherwise, we short-circuit and store an element directly.

This can save considerable time and space, especially in situations where we are creating and destroying large numbers of positions, without ever querying them for a locator. The only cost here is a check and conditional branch whenever the element or locator is queried from a position, to see if the locator has been allocated or not.

7.2 Enumeration Caching

Another concern is the time used in building enumerations. Since these are snapshots of the data structures, creating them will take time at least linear in the number of elements to be included in the enumeration. Here we have applied a *copy-on-write* policy for some data structures: when an enumeration is asked for, we create an array which stores all the objects over which to enumerate, then return an enumeration over that array. If the same enumeration is asked for again, we can simply return another enumeration object, over the same array—in constant time. If a call is made to a function which modifies the correct data to return, we discard our array (Java's garbage-collection scheme takes care of disposing of it).

In this way, we are optimizing for having a number of read-only calls in between phases of modification. This is in fact realistic: a common use of a data structure is to fill it with data, then run some algorithms on it which only inspect the data.

The cost of this method is none in terms of time, but in most implementations, we have to keep an extra copy of the data around between the time an

enumeration has been called for and the next modification. If we have multiple enumerations simultaneously, however, we save space since we share a single copy among them.

The method works especially well for array-based data structures. In these, the array itself can be used as the cached copy, so that even the first enumeration call can be done in constant-time. Only if the container is modified while there are enumerations outstanding do we actually need to copy any data.

8 Experience with JDSL

JDSL has a companion *teach* version [19] geared for pedagogical use. It has been used successfully in first-year, one-semester, CS2-level classes at Brown [16] and Johns Hopkins, and is the library discussed in the textbook by Goodrich and Tamassia [9]. At Brown, over a hundred CS2 students implemented the following data structures and algorithms using the JDSL *teach* library:

- Sequence, implemented with a circular array
- Binary search and quicksort of a Sequence
- Binary tree
- Binary heap, reusing the binary tree
- Red-black tree, reusing the binary tree
- Hash table
- Rabin-Karp string-searching algorithm
- Convex hull algorithm, using package wrap
- Spanning tree of an undirected graph
- Directed graph
- Prim's minimum-spanning-tree algorithm
- Dijkstra's shortest-path algorithm

The methodology of the library allows projects that are more theoretically sophisticated than were possible in past years, and more full-featured. Data structures met the interfaces specified by JDSL. For instance, most data structures included a full suite of modifiers (insertion, deletion, and replacement) and thorough error handling via exceptions. Students were supported by visualizers and testers written within JDSL [1].

A number of implementation projects have been written based on the research version of the library, as well. Various point location algorithms have been implemented [14]. A planar map (an embedded planar graph, or EPG) has been implemented [10], with operations that preserve planarity. The EPG is built on top of an ordered graph from the library. On top of the EPG, there are algorithms for orthogonal drawings of graphs [7] and for finding the shortest path between two points in the interior of an arbitrary polygon [2]. Finally, a vector class with efficient insertion at arbitrary positions has been implemented [8].

Acknowledgments

We would like to thank the entire JDSL team for their work on the project—in particular, Andy Schwerin and Maurizio Pizzonia for their constructive criticism of this paper, and John Kloss for helpful comments regarding topics related to the paper.

References

1. R. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. In *Proc. ACM Symp. Computer Science Education*, 1999.
2. J. Beall. Shortest path between two points in a polygon. http://www.cs.brown.edu/courses/cs252/projects/jeb/html/cs252proj.html.
3. M. Boilen, A. Schwerin, and J. Kloss. Personal communication.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
5. A. Fabri et al. The CGAL kernel: A basis for geometric computation. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, pages 97–103, May 1996.
6. N. Gelfand, M. T. Goodrich, and R. Tamassia. Teaching data structure design patterns. In *Proc. ACM Symp. Computer Science Education*, 1998.
7. N. Gelfand and R. Tamassia. Algorithmic patterns for graph drawing. In *Proc. Graph Drawing '98*. Springer-Verlag, to appear.
8. M. T. Goodrich and J. Kloss. Tiered vector: An efficient dynamic array for JDSL. Poster at *OOPSLA'98*.
9. M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, New York, NY, 1998.
10. D. Jackson. The TripartiteEmbeddedPlanarGraph. Manuscript.
11. K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.
12. M. Nissen. Graph iterators: Decoupling graph structures from algorithms. Diploma thesis, Max-Planck-Institut für Informatik, Univ. Saarlandes, Saarbrücken, Germany, 1998.
13. B. Stroustrup. *The C++ Programming Language (3rd Edition)*. Addison-Welsey, Reading, MA, 1997.
14. R. Tamassia, L. Vismara, and J. E. Baker. A case study in algorithm engineering for geometric computing. In *Proc. Workshop on Algorithm Engineering*, pages 136–145, 1997.
15. K. Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proc. OOPSLA '97*, pages 34–48, 1997.
16. CS 16 home page. http://www.cs.brown.edu/courses/cs016.
17. ffGraph home page. http://www.fmi.uni-passau.de/~friedric/ffgraph/main.shtml.
18. Java 1.2 API. http://java.sun.com/products/jdk/1.2/docs/api/index.html.
19. JDSL home page. http://www.cs.brown.edu/cgc/jdsl.
20. JGL home page. http://www.objectspace.com/jgl.
21. LEDA home page. http://www.mpi-sb.mpg.de/LEDA.