An Overview of the net.datastructures package[*]

Michael T. Goodrich[†‡] *Roberto Tamassia* [§‡] *Eric Zamore*[§]

`http://net.datastructures.net`
`java@datastructures.net`

(Version 3.0)

Abstract

We introduce *the net.datastructures package*, written entirely in Java. The net.datastructures package includes implementations of a variety of simple and complex data structures, such as lists, dictionaries, maps, trees, graphs, and priority queues, with powerful and flexible access to the elements stored, both by means of traditional iterators and by means of relavitvely new types of accessors called *positions* and *entries*. The net.datastructures package also includes implementations of a few fundamental algorithms, such as a DFS traversal of a graph, and Dijkstra's algorithm for shortest paths.

[*]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 and CCR–9732327.

[†]Center for Geometric Computing, Department of Computer Science, The Johns Hopkins University, 3400 N. Charles Street, Baltimore, MD 21218, USA.

[‡]Work on this project performed by this author as a coauthor of the book *Data Structures and Algorithms in Java*.

[§]Center for Geometric Computing, Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912–1910, USA.

1 Introduction

Computer programs organize information into *data structures* and process it according to *algorithms*. Thus, providing component libraries of efficient data structures and algorithms can enable rapid software development for advanced applications.

1.1 Data Structure Libraries in Java

Java is evolving into a premier development language for advanced software applications, particularly for the Internet. A small library of data structures and algorithms, which we refer to as *Java Collections* (JC) is included in the standard Java package `java.util`. An alternative library is the *Generic Library for Java* (*JGL*) by ObjectSpace, which is patterned after the STL, the *Standard Template Library* written in C++. Both the Java Collections and JGL provide implementations of basic data structures such as maps, sets, dictionaries, and sequences. JGL also provides a considerable number of template-based algorithms for permuting data. The *Graph Foundation Classes for Java* (*GFC*) by alphaWorks is a framework for programming with graphs in Java. It provides a set of data structures to represent trees and graphs and some graph drawing algorithms based on these data structures.

1.2 JDSL

The *Data Structures Library in Java* (*JDSL*), is a data structures library that provides advanced data structures and algorithms not found in the Java Collections and JGL. In addition to basic data structures such as lists and dictionaries, JDSL includes implementations of a variety of complex data structures such as trees, graphs, and priority queues, with powerful and flexible access to the elements stored.

The net.datastructures package was created to function as a simpler, more user-friendly alternative to JDSL. While the underlying philosophies behind each library differ, many of the features in JDSL are also included in the net.datastructures package, and much of the functionality of the two libraries is identical. They were both in-part developed at the *Center for Geometric Computing* at Brown University. See Section 5 for a full comparison between the two.

1.3 The net.datastructures package

Like JDSL, the net.datastructures package provides basic and advanced data structures such as lists, trees, graphs, dictionaries, hash tables, priority queues, and graphs. The net.datastructures package provides several algorithms such as Dijkstra's shortest-paths algorithm for graphs, and sorting algorithms such as quick-sort and merge-sort.

The net.datastructures package allows access to stored data through a variety of mechanisms. Besides providing iterators, a simple mechanism for iteratively listing through a collection of objects, the net.datastructures package provides two types of accessors to data, called *positions* and *entries*, which allows a user to "track" elements within data structures. For example, the method for inserting an element into a dictionary returns an entry, containing a key and a value, for the element, which allows to later access the element in constant time without having to search.

Another feature of the net.datastructures package is the support for *decorations* (or *attributes*), which can be used to "label" positions within a data structure. For example, in a traversal of a graph, we can

	JC	*JGL*	*GFC*	*JDSL*	*DSP*
Stacks and Queues	$\surd$	$\surd$			$\surd$
Sequences (lists, vectors)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
General-purpose trees			$\surd$	$\surd$	
Binary trees			$\surd$	$\surd$	$\surd$
Priority queues (heaps)		$\surd$		$\surd$	$\surd$
Hash Tables	$\surd$	$\surd$		$\surd$	$\surd$
Search Trees	$\surd$	$\surd$		$\surd$	$\surd$
Sets			$\surd$		
Graphs			$\surd$	$\surd$	$\surd$
Templated algorithms				$\surd$	$\surd$
Sorting algorithms	$\surd$	$\surd$		$\surd$	$\surd$
Data permutation algorithms		$\surd$			
Graph traversals			$\surd$	$\surd$	$\surd$
Shortest path				$\surd$	$\surd$
Minimum spanning tree				$\surd$	
Graph drawing algorithms			$\surd$		
Iterators	$\surd$	$\surd$		$\surd$	$\surd$
Accessors (positions and entries/locators)				$\surd$	$\surd$
Range views	$\surd$	$\surd$			
Decorations (attributes)			$\surd$	$\surd$	$\surd$
Thread-safety and full serializability	$\surd$	$\surd$			

Table 1: A Comparison of the Java Collections, the Generic Library for Java, the Graph Foundation Classes for Java, the Data Structures Library in Java, and the net.datastructures package.

use decorations to mark the vertices and edges visited so far.

The net.datastructures package treats algorithms as objects that (*i*) are instantiated, (*ii*) passed the input data, (*iii*) run through some sort of *execute* method, and (*iv*) provide access to the output after their execution via various methods. Most algorithms implemented by the net.datastructures package can be parameterized by means of the *template method pattern*.

Table 1 compares key features of JC, JGL, GFC, JDSL, and the net.datastructures package (DSP).

The net.datastructures package has been developed at the *Center for Geometric Computing* at Brown University, in collaboration with Michael Goodrich, who is a professor at The Johns Hopkins University.

In the next section, we present the design goals for the net.datastructures package. Section 3 discusses the major concepts used in the net.datastructures package, and Section 4 examines the specific data structures and algorithms packaged in this release of the net.datastructures package. Section 5 outlines a comparason between JDSL and the net.datastructures package, and in Section 6, we briefly outline what is planned for the future of the net.datastructures package.

2 Design Goals

In this section, we overview the design goals followed in the development of the net.datastructures package.

2.1 Education

The main purpose of the net.datastructures package is to serve as an educational tool. The software is meant to explain concepts about algorithms and data structures by providing a set of functional, fully-usable components. The software is intended to be used for educational purposes, though it is suitable for any non-commercial use. The package has been thoroughly tested, but the input data sets were relatively small, making it impractical for use in a commercial product.

2.2 Code Readability

The code behind the net.datastructures package is intended to be easily readable - so that anyone familiar with the Java programming language will have full understanding of the code with one readthrough. Our choice of syntax is clear and concise, and many comments are interleaved to provide further clarification.

2.3 Simplicity

In an effort to make the net.datastructures package easy to use, simplicity was an important consideration in designing the package. All components of the net.datastructures package belong to a single Java package. There are very few different types of exception classes, no container classes, and quite basic inheritance hierarchies.

2.4 Efficiency

The data structures in the net.datastructures package typically offer the best-possible asymptotic time complexity for every operation supported. For example, in a hash table, searches are performed in expexted $O(1)$ time, while in a heap, insertions and removals are performed in actual $O(\log N)$ time. It should be noted that in order to keep the package as simple as possible, no constant-factor optimizations have been made.

2.5 Reliability

Reliability has been one of our primary goals during the construction of the net.datastructures package. We have thoroughly tested each component and ensured that it functions exactly according to our specifications. We have also paid close attention to error conditions, and made sure that appropriate user feedback is provided when an exception is thrown.

2.6 Object-Orientation

The net.datastructures package takes a strong object-oriented view of data structures and algorithms. The data structures and algorithms in the net.datastructures package are objects that themselves handle all the supported operations. Algorithm objects are instantiated with the input data and store the results of their execution and provide methods to access them.

3 Data Organization Concepts in the net.datastructures package

In this section, we examine some key data organization concepts used in the net.datastructures package.

3.1 Element

An element of a data structure is any `java.lang.Object`. The element is the underlying data stored in the data structure. Note that the same object can be stored in many data structures and can be also stored multiple times in the same data structure.

3.2 Key

Some data structures in the net.datastructures package store *keys* associated with elements. When keys are involved with a data structure, we use the term "value" instead of "element" to avoid confusion. Keys are typically used as an indexing mechanism for their associated value. An example of a key-based data structure is a *dictionary*, whose main methods support the following operations:

- inserting a (key, value) pair;
- searching for an value with a given key;
- removing an value with a given key.

A key can be any `java.lang.Object`. Note that a key and its value need not be distinct from each other. Typical keys are strings (e.g., names) and numbers (e.g., account numbers).

3.3 Accessor

The net.datastructures package provides unified and implementation-independent access to the elements of a data structure by means of *accessors*. An accessor provides constant-time access to an element stored in a data structure, independently from the implementation of the data structure. Every element in the net.datastructures package has an accessor which contains it. Most operations in the net.datastructures package refer to accessors, and not to the actual data itself. There are two main types of accessors in the net.datastructures package, *positions* and *entries*.

For example, a sequence `S` may be implemented either by means of an array or by means of a linked list. In the first case, to access an element we need its index in the array. In the second case, we need a pointer to the list node storing the element. However, the user of the net.datastructures package need not know which implementation of the sequence is being used. This is because in either case, given a position `pos` in the sequence, we can get the element by calling `pos.element()`. Also, we can delete the element from the sequence by calling `S.remove(pos)`.

3.3.1 Position

Positions denote virtual "places" in data structures such as sequences, trees, and graphs. These data structures establish topological relations between their positions, such as the adjacency relation on the vertices of a graph, the parent-child relation on the nodes of a tree, and the predecessor-successor relation on the nodes of a sequence. Positions represent the "nodes" in such structures.

3.3.2 Entry

Any key-based data structure uses an *entry* to represent the data associated with a key. Since these data structures do not necessarily define relationships among virtual places but operate on their elements through their associated keys, an entry is conceptually different from a position.

An entry is a mechanism for representing a (key,value) pair inside a key-based data structure `C`. If you give the data structure a pair to hold, it gives you back an entry `ent`, and you can later refer to the pair by means of the entry. For example, you can get the key with `ent.key()`, you can remove the pair by calling `C.remove(ent)`, and you can change the existing value with a new value `newVal` by calling `C.replaceValue(ent,newVal)`.

3.4 Iterator

Iterators provide a simple mechanism for iteratively listing through a collection of objects. The net.datastructures package provides two types of iterators - a position iterator and an element iterator. They each maintain a pointer to a "current" item in a list. The element iterator iterates over each element of a list, while the position iterator iterates over each *position* of a list. Both implement the `java.util.Iterator` interface, which has three methods: `hasNext()`, `next()`, and `remove()`. Neither iterator provided by the net.datastructures package supports the `remove()` operation; if `remove()` is called on an iterator, an exception is thrown.

Each iterator takes a list as a parameter to its constructor. Since the iterator does not copy the list, any changes that are made to the list will be reflected in the iterator. For example, if you generate an iterator over the elements of list `L`, and subsequently call `L.removeLast()`, the element that was removed from the list, `elem`, is also "removed" from the iterator. This means that the iterator will no longer return `elem` over its iteration.

Alternatively, one can use an iterator such that after one is generated over the elements of a data structure, changes to that data structure are *not* reflected in the iterator – the iterator refers to the state of the data structure when the iterator was created. This implies that in our above example, the iterator *would* return `elem`, the object removed from the list after the iterator was generated, over its iteration. To achieve this behavior, one could alter the `elements()` method of a list (which returns an iterator over the list) to first copy the list, and then pass the *copy* to the iterator. That way, the iterator refers to the copy, and changes to the original will not affect the iteration. This is what is currently done in the net.datastructures package.

3.5 Comparator

When using a key-based data structure, it is particularly important to be able to specify the relation for comparing the keys. In general, this relation depends on the type of the keys and on the specific application for which the key-based data structure is used. Keys of the same type may be compared differently in different applications.

To provide this capability, the net.datastructures package makes use of the `java.util.Comparator` interface. All comparators in the net.datastructures package implement this interface, and only provide one method - the `compare()` method. Also, a separate interface, named `EqualityTester`, is provided by the net.datastructures package, which is used to determine whether two objects are equal.

The net.datastructures package only provides one public comparator, the `DefaultComparator`, which uses the natural ordering of objects, as specified by the `compareTo(Object)` method in the `java.lang.Object` class. A few data structures use inner classes (§3.10) as comparators and others as equality testers.

3.6 Decoration

The net.datastructures package provides the `DecorablePosition` interface to allow the ability to "decorate" the positions of a data structure with various attributes. This mechanism is useful for storing intermediate or final results of the execution of an algorithm. For example, in a depth-first search traversal of a graph, we can use decorations to mark the vertices as visited or unvisited.

3.7 Map and Dictionary

The net.datastructures package provides the `Map` interface to uniquely associate a data item with a key, thus creating a 1-1 mapping from keys to data. The `Dictionary` interface provides a mechanism to store entries for convenient lookup. Both interfaces support insertion, search, and removal. The main difference is that each key in a map is associated with a unique element, whereas two or more entries with the same key may exist in a dictionary.

3.8 Binary Tree Format

Binary trees in the net.datastructures package are not proper, which means that external nodes have zero children, and internal nodes can either have one child or two children. A proper binary tree, on the other hand, requires that its internal nodes have exactly two children. In the net.datastructures package, if a node has only one child, it can be either a left or a right child. To facilitate this, methods such as `hasLeft(Position)`, which returns true if and only if a given node has a left child, are provided in the binary tree interface.

In the net.datastructures package, a heap, for instance, is built upon a complete binary tree. Because the tree is also non-proper, elements of the heap are stored in external (as well as internal) nodes. The binary search tree implementation, however, *is* proper, and thus does not store elements within external nodes. This decision to require binary search trees to be proper was made because many algorithms for binary search trees are much simpler to implement when built on a proper tree.

3.9 Algorithm/Template Method Pattern

The net.datastructures package views algorithms as objects that are instantiated with the input data, and provide access to the output after their execution via various methods. Algorithms in the net.datastructures package perform “generic” computations that can be parameterized by means of the *template method pattern* [GHJV95]. In short, this means that they can be specialized for specific tasks by subclassing them and overriding specific methods.

For example, the net.datastructures package contains an implementation of a depth-first search traversal of a graph, which contains a series of empty methods which get called by the main body of the algorithm when a vertex is first encountered, when an edge is traversed, etc. These empty methods can be overridden in subclasses to provide specific functionality.

3.10 Inner Class

The net.datastructures package heavily uses *inner classes*, which are classes completely defined within a regular public class. In Java applications, inner classes are typically used to define some small internal component (e.g. a button). In the net.datastructures package, inner classes are used when a data structure requires some component which would not be useful to classes outside the net.datastructures package. For example, the `BinarySearchClass` defines the inner class `BSTEntry`, which implements the `Entry` interface. There is no public implementation of an entry because different data structures specialize entries in various ways, and an outside user of the net.datastructures package would have no use for a common entry implementation. This means that each data structure that uses entries (or any type of accessor, for that matter) must define its own implementation. and this is done via an inner class. All inner classes in the net.datastructures package are declared `protected` so that they can be used in subclasses of their enclosing classes. They are declared `static` so that they do not have references to their enclosing classes, because such references would be unnecessary and would break encapsulation.

4 The Architecture of the net.datastructures package

In this section, we describe the interfaces defined in the net.datastructures package, their implementations, and the algorithms that operate on them. All interfaces and classes in the net.datastructures package are included, with the exception of inner classes (see §3.10).

In the rest of this section, we denote with N the current number of elements stored in the data structure being considered.

4.1 Packages

Each class and interface of the net.datastructures package is contained within a single Java package, called `net.datastructures`.

4.2 Accessors

There are several interfaces and classes that serve as accessors, implying they contain a single data item (see §3.3).

4.2.1 Position

The interface `Position` has one method, `element()`, which returns the data item stored at that position. `BTPosition` (an interface for a node in a binary tree) extends `Position`. The `BTNode` class implements the `BTPosition` interface, and is used in binary trees. The `DNode` class is an implementation of a position used for a doubly-linked list (see §3.3.1).

A decorable position is a position to which it is possible to assign attributes, such as the weight of an edge (see §3.6). Both the `Vertex` and `Edge` interfaces are decorable positions.

4.2.2 Entry

An entry is an accessor for key-based data structures (see §3.3.2). The `Entry` interface has two methods, `key()` and `value()`, which return the key and data, respectively, stored in a given entry.

4.3 Basic Data Structures

We classify the following data structures as "basic" because they support relatively few simple operations, and are often the first data structures presented to students.

4.3.1 Stack

A stack is a container of elements that are inserted and removed according to the *last-in, first-out (LIFO)* principle. A stack can hold an arbitrary number of elements, but only the most-recently inserted element (i.e., the "last-in" element) can be accessed or removed. The net.datastructures package provides the `Stack` interface, and two implementations of that interface. The `ArrayStack` interface implements a stack by means of a fixed-size array, and the `NodeStack` class implements a stack by means of a singly-linked list of nodes. In both implementations, all operations run in constant time.

The `Node` class is provided as an implementation of a node for a singly-linked list that does *not* implement the `Position` interface. This is done for simplification purposes, as students are generally introduced to basic data structures before accessors.

4.3.2 Queue

A queue is a container of elements that are inserted and removed according to the *first-in, first-out (FIFO)* principle. A queue can hold an arbitrary number of elements, but only the element that has been in the queue the longest (i.e., the "first-in" element) can be accessed or removed. The net.datastructures package provides the `Queue` interface, and the `NodeQueue` class, which implements a queue by means of a singly-linked list of nodes. All operations in this implementation run in constant time.

As in `NodeStack`, the `NodeQueue` class uses the `Node` class for the same reasons.

4.3.3 Deque

A deque (double-ended queue) is a collection of elements that are inserted and removed only from either end. It can be viewed as a queue that supports accesses at both ends. The net.datastructures package provides the `Deque` interface, and the `NodeDeque` class, which implements a deque by means of a doubly-linked list of nodes. All operations in this implementation run in constant time.

As in `NodeStack` and `NodeQueue`, the `NodeDeque` class uses an implementation of a doubly-linked list node (called `DLNode`) which does not implement the `Position` interface.

4.4 Sequential Data Structures

A sequential data structure stores its data in a linear sequence that is transparent to the user. Each item is "adjacent" to at most two other items.

4.4.1 List

A list is a linear collection of nodes, where each node is a position that contains a single data item. Access into the list is performed via the nodes. The the net.datastructures package provides the `List` interface and an linked-list implementation of that interface, the `NodeList` class. The `NodeList` class supports $O(1)$-time insertions, removals, and sequential accesses (e.g. finding the "next" node in the list).

4.4.2 Vector

A vector is a linear sequence with the notion of *rank*. Each item has a unique rank in the list, from 0 to $N-1$. The rank of an item represents its location in the list, and a vector supports multiple operations based on rank. The `Vector` interface and `ArrayVector` class are provided by the net.datastructures package. `ArrayVector` supports $O(1)$ rank-based accesses, and $O(N)$ insertions and removals.

4.4.3 Sequence

A sequence supports all the operations of a list and a vector, thus the `Sequence` interface inherits from both the `Vector` and `List` interfaces. It also contains two "bridging" methods which provide connections between ranks and positions. There currently is no implementation of a sequence in the net.datastructures package.

4.5 Iterators

The net.datastructures package provides two iterator classes, named `PositionIterator` and `ElementIterator`. Both iterate over a `List` implementation by maintaining a pointer to the current position (node of the list) or element (data item in the list), respectively (see §3.4).

4.6 Trees

The net.datastructures package contains many different types of trees. Trees allow more sophisticated relationships between elements than is possible with a linear structure. Trees allow relationships between a child and its parent, or between siblings of one parent. Trees have one root, which has no parent, and external leaves, which have no children. In this sub-section, we omit trees that maintain a key-based ordering (e.g. all binary search trees).

4.6.1 Tree

The most basic interface for a tree is the `Tree` interface, which lists a collection of methods which describe a generic tree where each node can have an arbitrary number of children. A node is external if it has zero children, and internal otherwise.

4.6.2 Binary Tree

The `BinaryTree` interface extends the `Tree` interface, adding support for a *non-proper* binary tree, implying that any given node can have zero, one, or two children (see §3.8). The `LinkedBinaryTree` class is a node-based implementation of a binary tree where each node has a reference to its parent and children.

4.6.3 Complete Binary Tree

A binary tree with height h is *complete* if the levels $0, 1, 2, \ldots, h-1$ have the maximum number of nodes possible (that is, level i has 2^i nodes, for $0 \leq i \leq h-1$) and in level $h-1$ all the internal nodes are to the left of the external nodes. The `CompleteBinaryTree` interface describes a complete binary tree, adding to the `BinaryTree` interface methods `add()` and `remove()`, which adds and removes, respectively, the node at the "end" of the tree, which is the rightmost node at the highest level of the tree. The `VectorCompleteBinaryTree` class is a realization of a complete binary tree by means of a vector.

4.7 Comparators

Comparators are used to compare keys in key-based data structures. They provide a generic way to specify many different orderings in a key-based data structure without changing the data structure itself.

4.7.1 Equality Tester

The `EqualityTester` interface is not a "comparator" per se, but it is used to determine whether two objects are equal. There currently is no public implementation of an equality tester in the net.datastructures package.

4.7.2 Default Comparator

The `DefaultComparator` is an implementation of the `java.util.Comparator` interface, and uses the natural ordering of objects, as specified by the `compareTo(Object)` method in the `java.lang.Comparable` interface.

4.8 Priority Queues

A priority queue is a data structure for storing a collection of keys-value pairs, where the smallest key value indicates the highest priority. It supports arbitrary insertions and deletions of entry and keeps track of the highest-priority entry. A priority queue is useful, for instance, in applications where the user wishes to store a queue of tasks of varying priority, and always process the most important task next. There are two interfaces describing priority queues in the net.datastructures package: the `PriorityQueue` interface and the `AdaptablePriorityQueue` interface. The former provides functionality to update the priority queue by removing the highest-priority entry (i.e., `removeMin()`) only, while the latter allows updates to any entry of the priority queue (i.e., `remove(Entry)` or `replaceKey(Entry,Object)`).

4.8.1 Heap

A heap is a *complete* binary tree satisfies the following property: for each node v other than the root, the key stored at v is greater than or equal to the key stored at the parent of v. Thus, if you were to traverse a path from the root to a leaf, you would encounter keys in non-decreasing order. This description allows a heap to support $O(\log N)$ insertions, removals, and key-based updates (i.e., `replaceKey(Entry,Object)`). The `HeapPriorityQueue` and `HeapAdaptablePriorityQueue` classes each implement a priority queue by means of a heap.

4.8.2 Sorted List Implementation

The sorted list implementation of a priority queue is simply a list sorted in non-decreasing order. Thus, the highest-priority item is the first element in the list. Removals take $O(1)$ time, but insertions and key-based updates take $O(N)$ time. The `SortedListPriorityQueue` and `SortedListAdaptablePriorityQueue` classes provide this implementation.

4.9 Maps

A map uniquely associates some data to a key. It differs from a dictionary in that a map does not allow duplicate keys. The map exists primarily to define "attributes" for an object and assign values to those attributes. (see §3.7, §3.6). The `Map` interface describes a map, and the `HashTable` class provides an implementation by means of a hash table. The `HashTable` class uses linear probling to resolve collisions, and has expected $O(1)$ time for insertions, removals, and searches.

4.10 Dictionaries

A dictionary is a data structure used to store key-value pairs and quickly search for them using their keys. All dictionaries in the net.datastructures package can store multiple key-value pairs with the same key. A dictionary needs some type of comparator to maintain the order of its keys, and thus all the dictionary implementations in the net.datastructures package have constructors which take a comparator as a parameter.

4.10.1 Binary Search Tree

A binary search tree is a proper binary tree in which each internal node stores a key according to the following ordering: if node v stores key k, then all keys stored in the left subtree of v are less than or equal to k, and all keys stored in the right subtree of v are greater than or equal to k. The `BinarySearchTree` class is a realization of a dictionary meeting these specifications. It extends `LinkedBinaryTree`.

4.10.2 AVL Tree

The `AVLTree` class is an implementation of an AVL tree, a binary search tree where insertion, removal, and access to key-value pairs require each $O(\log N)$ time. It maintains this property by requiring the heights of the left and right subtrees of each node not to differ by more than one. Its name comes from its founders, Adel'son-Vel'skii and Landis.

4.10.3 Red-Black Tree

The `RBTree` class is an implementation of a red-black tree, a bounded-depth binary search tree where insertion, removal, and access to key-value pairs require each $O(\log N)$ time. It maintains this property through a node-colorization scheme where each node is either colored red or black.

4.11 Graphs

A graph is a fundamental data structure used in a variety of application areas describing a binary relationship on a set of elements. Each vertex of the graph may be linked to other vertices through edges. Edges can be either one-way, *directed* edges, or two-way, *undirected* edges. In the net.datastructures package, there are no directed graphs, and both vertices and edges are positions of the graph. The `Graph` interface describes a generic, undirected graph.

4.11.1 Adjacency List Graph

The net.datastructures package provides a realization of a graph by means of an adjacency list structure, which is characterized by the following properties.

- There is a list of all vertices, V, and all edges, E, in the graph.
- Each vertex and edge has a reference to an element, and its position in V or E, respectively.
- Each vertex v holds a reference to a list, I, of its incident edges.
- Each edge e has a reference to its endpoint vertices (there are always two of them), and for each endpoint, a reference to e's position in the incidence list, I, of that endpoint.

The `AdjacencyListGraph` class is an implementation of an adjacency list graph. Both parallel edges (mutiple edges between the same two endpoints) and self-loops (edges for which both endpoints refer to the same vertex) are supported.

4.12 Algorithms

The net.datastructures package provides a series of algorithms designed for different applications. Each algorithm is a Java object that must be instantiated, passed input data, and executed. Currently, the net.datastructures package provides algorithms for sorting and graph-processing.

4.12.1 Sorting Algorithms

All sorting algorithms in the net.datastructures package are containted in the `Sort` class. There are two different sorting algorithms, quick-sort and merge-sort, and there are two different versions of each: an array-based sort and a list-based sort. Each is represented by a Java method which takes either an array or list and a comparator as parameters. Below is a description of these sorting algorithms.

- **Quick-Sort.** Quick-sort is an extremely fast sort, running in $O(N \log N)$ expected time. However, its performance degrades greatly if the sequence is already very close to being sorted. Also, it is not *stable* — that is, it does not guarantee that elements with the same value will remain in the same order they were in before sorting. In all cases whether neither of these caveats apply, it is the best choice.

  The array-based version of quick-sort is an in-place sort, meaning it uses a constant amount of additional space. The list-based version is not in-place. Both versions are recursive.

- **Merge-Sort.** Merge-sort is not as fast as quicksort, though it still runs in $O(N \log N)$ time. There are no cases where its performance will degrade due to peculiarities in the input data, and it is a stable sort.

  The array-based version of merge-sort is iterative, and the list-based version is recursive. Neither sort is in-place.

4.12.2 DFS

The depth-first search (DFS) traversal of a graph is available in the net.datastructures package. Depth-first search proceeds along one path, continuing until no new vertices can be found before backtracking. The implementation of depth-first search is a template method that allows the user to specify actions to occur when a vertex is first visited or is "finished" by being exited for the last time, and when different sorts of edges are reached (such as tree edges in the search tree DFS generates, or cross edges between different branches of the search tree).

The `DFS` class is provided as a template. It contains all necessary functionality to traverse the graph, in addition to callback methods which are called at certain points during the traversal. For instance, the method `traverseBack(Edge,Vertex)` is called whenever a back edge is traversed, and the method `startVisit(Vertex)` is called just before a vertex is marked as visited. To use these callbacks, they must be overridden in a subclass.

The `DFS` class is abstract, and therefore must be overridden to be used. The abstract `execute(Graph,Vertex,Object)` method is used to start the execution of the traversal, and return a "result" of the traversal, as defined by a subclass. The third parameter (of type `Object`) exists so that additional information can be supplied to the traversal. For example, the `FindPathDFS` class requires a destination vertex to be passed in via this parameter, and returns a path in the form of a list.

The following three subclasses of `DFS` are provided to perform specific functions.

- `FindPathDFS` finds a path between two vertices.
- `ConnectivityDFS` determines whether a graph is connected.
- `FindCycleDFS` finds a cycle in the graph, if one exists.

All three are good examples of how to use `DFS` through the template method pattern.

4.12.3 Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest path to every vertex of a connected graph from a specific source vertex. Each edge must have an integer "weight" attribute associated with it. The length of a path is determined by computing the sum of the weights of the edges along a path, and the shortest path is a path with minimum length. In the net.datastructures package, after the algorithm finishes its execution, a user can query each vertex for the shortest distance between it and the start vertex. The implementation of Dijkstra's algorithm — `Dijkstra` — uses the template method pattern; it can be easily extended to change its functionality, although extending is not necessary. Extending it makes it possible, for instance, to stop after computing the shortest path to a specific vertex, to alter the function for calculating the weight of an edge, and to change the way the results are stored.

To use the algorithm, you must call the `execute(Graph,Vertex,Object)` method. The first parameter is the graph on which to execute Dijkstra's algorithm, the second parameter is the source vertex, and the third parameter is the weight decoration object. The weight decoration object is the attribute through which all weights have been assigned to the edges. For example, suppose we had an object `WEIGHT`, and we set the edge weights by calling `edge.put(WEIGHT,new java.lang.Integer(intWeight))`, where `intWeight` was some `int` storing the weight of a given edge. To execute Dijkstra's algorithm, we would pass the object `WEIGHT` as the third parameter to the `execute` method. Note that it is required that for each edge, the value of the weight attribute be a `java.lang.Integer`. Also, if no weights are set, the `execute(Graph,Vertex,Object)` method will throw an exception.

After calling the `execute(Graph,Vertex,Object)` method, you can access length of the shortest path from the source to any given vertex `vert` by calling `getDist(vert)`, which returns an `int`. If this method is called before the algorithm is executed, an exception is thrown. If there is no path from the source to `vert` (and hence the graph is not connected), `getDist(vert)` will return the constant `java.lang.Integer.MAX_VALUE`.

4.13 Exceptions

The net.datastructures package defines the following exception classes. Each time an exception is thrown, a detailed error message is provided, which describes the error that occurred. The exceptions are the following.

- `BoundaryViolationException`. Signals that the boundaries of a data structure have been illegally traversed (e.g. past the end of a list).
- `EmptyDequeException`. Thrown when a deque cannot fulfill the requested operation because it is empty.
- `EmptyListException`. Thrown when a list cannot fulfill the requested operation because it is empty.

- `EmptyPriorityQueueException`. Thrown when a priority queue cannot fulfill the requested operation because it is empty.
- `EmptyQueueException`. Thrown when a queue cannot fulfill the requested operation because it is empty.
- `EmptyStackException`. Thrown when a stack cannot fulfill the requested operation because it is empty.
- `EmptyTreeException`. Thrown when a tree cannot fulfill the requested operation because it is empty.
- `FullStackException`. Thrown when an attempt is made to push an item onto a full stack. This only occurs in the array implementation.
- `InvalidEntryException`. Thrown when an entry is discovered to be invalid.
- `InvalidKeyException`. Thrown when an key is discovered to be invalid.
- `InvalidPositionException`. Thrown when an position is discovered to be invalid.
- `NonEmptyTreeException`. Thrown when a tree cannot fulfill the requested operation (e.g. adding a root) because it is not empty.

In addition to these, three exception classes provided by Java are used by the net.datastructures package. They are the following.

- `java.util.NoSuchElementException`. Thrown by an iterator to indicate that there are no remaining elements that haven't been traversed.
- `java.util.UnsupportedOperationException`. Thrown to indicate that the requested operation is not supported. This is used when `remove()` is called on an iterator.
- `java.lang.IllegalStateException`. Signals that a method has been invoked at an illegal or inappropriate time. This is used with the comparator is set in a non-empty priority queue.

5 Comparison to JDSL

This section assumes the reader is already familiar with JDSL. Readers who have not used JDSL should skip this section. The purpose of this section is to highlight the significant differences between the net.datastructures package and JDSL.

5.1 Purpose

The main goal of JDSL is to provide an efficient, reliable, flexible data structures library in Java containing many crucial components not available in other existing data structures libraries in Java. It is designed for both educational and high-performance use. It has been optimized and very well documented.

The primary goal of the net.datastructures package can be similarly phrased, but with two main differences. First of all, the package was not designed to fill holes left by other data structures libraries. Indeed, most, if not all, of the functionality in the net.datastructures package is also in available JDSL. Secondly, and more importantly, the net.datastructures package exists because JDSL was too advanced for some users. The net.datastructures package is a reduced, simpler, clearer, and easier-to-use version of JDSL. These advantages come with drawbacks, however. For instance, there are no constant-factor optimizations in the net.datastructures package.

5.2 Architecture

The net.datastructures package contains a much simpler architecture than does JDSL. Listed below are statistics that illustrate this fact.

- All classes and interfaces within the net.datastructures package are contained within a single Java package. JDSL currently consists of eight Java packages.
- The net.datastructures package contains about 60 Java files, whereas JDSL contains about 100.
- The net.datastructures package contains around 35 public classes and 20 public interfaces, whereas JDSL contains around 50 public classes and 40 public interfaces.
- The net.datastructures package and JDSL each contain about 10 concrete data structures.
- The net.datastructures package provides implementations for about 8 algorithms, whereas JDSL implements around 15 algorithms.

5.3 Binary Tree Format

One of the major design shifts from JDSL is the format of binary trees. In JDSL, all binary trees are required to be *proper*, that is, all nodes must either have exactly zero or two children, meaining that all internal nodes must have two children. As discussed in §3.8, binary trees in the net.datastructures package are *not* proper.

This difference is not noticeable, however, in key-based data structures that are built upon binary trees (e.g. red-black trees or heaps), because a user cannot access the underlying tree. For example, in a heap, if a (key-value) pair, a user does not know anything about the node (whether it is internal or external, for example) which contains the corresponding entry. The user also does not know how many children the node has. These details are all handled by the heap implementation.

The difference in binary tree format comes into play when using basic binary trees which do not serve a special purpose (e.g. general node-based binary trees). In this scenario, a user must properly handle both internal nodes that have have one child, and those that have two children. In addition, any user-created subclasses of key-based data structures (e.g. red-black trees, heaps) are given access to the underlying tree, which may or may not be proper, depending on the data structure (see §3.8).

5.4 Comparator

There are two main differences between the comparators provided in the net.datastructures package and the comparators provided in JDSL. In the net.datastructures package, there are fewer types of comparators, and they do not support most of the specialized operations that the JDSL comparators do.

In the net.datastructures package, there is only one public implementation of a comparator, which is a default comparator which simply uses the natural ordering of objects. JDSL uses specialized comparators (e.g. integer comparators) which allow it to define precisely the set of elements over which the comparator is valid. Thus, JDSL comparators provide the `isComparable(Object)` method which determines whether an object is in this set of elements. Comparators in the net.datastructures package have no such mechanism.

To compare objects in the net.datastructures package using a comparator, one must use the `compare(Object,Object)` method, which determines whether an object is greater than, equal to, or less than a second object. In JDSL, more specific operations to compare objects are available, such as the `isGreaterThanOrEqualTo(Object,Object)` method, which returns true if the first object is greater than or equal to the second. Such operations are optional, as they can be implemented using only the `compare(Object,Object)` method. Thus, in the interest of simplicity, they were omitted in the net.datastructures package.

5.5 Iterator

The semantics of iterators in the net.datastructures package are left to data structures that return them. In JDSL, on the other hand, iterators have specific *snapshot* semantics: they refer to the state of the data structure at the time the iterator was created, even in the data structure has been modified while stepping through the iterator. For example, if an iterator is created for all the nodes of a tree and then a subtree is cut off, the iterator will still include the nodes of the removed subtree. JDSL guarantees that all iterators have snapshot semantics, and thus its data structures are designed in a way that depends on this fact. In the net.datastructures package, no guarantees are made about iterator semantics.

Another difference is that JDSL iterators support a "reset" operation, which will reset the iterator to its initial state, implying that the iterator will subsequently return its first object. There is no such reset operation in the net.datastructures package.

5.6 Entry vs. Locator

JDSL introduced the concept of a *locator*, which is analagous to an *entry* in the net.datastructures package - both store an item in a key-based data structure and are both used to associate data with a given key.

There are a few slight differences, however.

- The term "element" is used to describe the data associated with a key in a locator. This is analagous to the term "value" in an entry.
- In JDSL, both locators and positions inherit from a common `Accessor` interface. In the net.datastructures package there is no such common inheritance.
- Classes that implement the `Locator` interface almost always are implemented efficiently. For instance, a locator class usually has a reference to (a) its position in a data structure, and (b) the data structure itself. This allows the data structure to perform efficient reorganizations and also perform a constant-time "contains" test to determine whether a given locator is a member of

itself. Classes that implement the `Entry` interface make no such guarantees about implementation efficiency.

5.7 Decorable Position

In the net.datastructures package, not all positions are decorable, as they are in JDSL. In the net.datastructures package, decorable positions are a subset of all positions. Decorations in the net.datastructures package are really just entries in a map.

5.8 Dictionary vs. Decorable

The JDSL equivalent to the `Map` interface (i.e., each key is uniquely associated with a value) is the `Decorable` interface. JDSL provides hash table implementations for both the `Decorable` and `Dictionary` interfaces, while the net.datastructures packageonly provides one for the `Map` interface.

6 Future Work

During the testing phase of the net.datastructures package, several visualizers, which draw a given data structure on the screen, were used to initially verify the correctness of each data structure. For almost every data structure, there is a visualizer that can display some sort of graphical representation of the data structure. These visualizers are currently under development, but it is planned that in the future they will be included with the rest of the net.datastructures package.

Also, numerous batch testers were created to ensure correctness over a large number of operations. Although they don't include much output or feedback, these batch testers may be released as well.

7 Acknowledgments

We would like to acknowledge suggestions, ideas, and code prototypes given by James Baker, Don Blaheta, Lubomir Bourdev, Jitchaya Buranahirun, Ming En Cho, Marco da Silva, John Kloss, David Jackson, Masi Oka, and Amit Sobti. Finally, we would like to thank Franco Preparata for encouragement and support.

References

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, MA, 1995.