CATEGORIES OF DATA STRUCTURES*

Peter Falley
Professor Emeritus
Fairleigh Dickinson University, Madison, NJ 07940
Home Telephone: (505)388-2004
E-mail: falley@fdu.edu

ABSTRACT

In teaching Data Structures, we commonly discuss a sequence of different data types, grouping them according to certain commonalities, but often neglecting to emphasize the significant differences between the groups. When working in Java, the tendency to blur these distinctions is reinforced by the arrangement of the Java Collections Framework, which treats stacks as vectors and hash tables as dictionaries for example. This paper proposes grouping data structures into three categories: Storage structures (arrays, linked structures, hash tables), process-oriented data structures (stacks, queues, priority queues, iterators), and descriptive data structures (collections, sets, linear lists, binary trees, etc.), i.e., structures used to represent groups of objects. Storage structures are fundamental building blocks used in the implementation and representation of the other types of data structures. The paper seeks to emphasize the commonalities within each of the other two categories and the differences between the categories by employing implementation-independent specifications and contrasting the dynamic nature of the process-oriented with the static nature of the descriptive structures.

INTRODUCTION

The purpose of this paper is to lay out a division of the common data structures into three categories – storage structures, process-oriented data structures, and descriptive data structures. The data structures in each of these categories share a common purpose that distinguishes them from data structures in the other categories. The classification given here is not unique. For example, the C++ Standard Library (STL) [5] groups its container classes into sequences, container adapters, and associative containers, but that represents conceptually quite a different approach than the one proposed here.

The paper also presents coherent, implementation-independent approaches to giving abstract data type specifications for many of the data structures in the process-oriented and descriptive categories. Because textbooks often tie definitions of various data structures closely to their implementation, students tend to come away from Data Structures courses without a clear understanding of the distinction between an abstract data type and its implementation. The approach to specifying abstract data types given here may make it easier to clarify and maintain that distinction.

Throughout this paper, object-oriented and Java™ notation and terminology will be used.

STORAGE STRUCTURES

Storage structures (also referred to as Fundamental Data Structures in [1]) are data structures used to organize blocks of memory and to provide access to cells within those blocks. (Similar use of this terminology may be found in [3,7].) This category includes arrays and vectors, records, linked structures, and hash tables. Each such structure is characterized through use of a starting memory address, together with an access method for accessing individual components of the structure.

For instance, for an array or record, the access method involves using an access function to calculate the offset of a particular cell from the starting location. On the other hand, accessing components of a linked structure involves following a sequence of pointers built into the structure's individual components. For a hash table, the access method involves application of a hash function to the object being stored or to be retrieved, usually followed by some kind of linear search.

Each case involves physical memory structures as opposed to logical or conceptual constructs. Storage structures provide the basis for the implementation and representation of *logical data structures* of the kind discussed in the rest of this paper.

PROCESS-ORIENTED STRUCTURES

One category of logical data structures consists of those here called process-oriented. These are data structures for regulating the flow and utilization of data in programs, procedures or functions (methods). The most familiar of these are stacks and queues, which are often described as limited access linear lists [3,4,8]. However, that description focuses on their implementation rather than their functionality. Consistent with the purposes of process-oriented data structures, the approach taken here is to describe them in terms of their functionality, i.e., their input-output behavior rather than their internal structure.

The process-oriented data structures (PODS) share the following common operations protocol: Each PODS will include, at a minimum, methods (operations) of the form

```
PODS enter(Object o);     // enters o into the PODS changing its state. The
                          // modified PODS is returned
Object remove();          // provided the PODS is non-empty, removes an
                          // element from the PODS, changing the state of
                          // the latter, and returns the removed element
boolean isEmpty();        // determines if the PODS is in an empty state.
```

For purposes of discussing process-oriented data structures and describing differences between different types of such structures one can introduce the following terminology:

1. If po is an empty PODS and the instructions po.enter(o_1), po.enter(o_2), …, po.enter(o_n) are executed in succession without an intervening **remove** operation, po will be said to accept the sequence $\{o_1,o_2,\ldots,o_n\}$.
2. Similarly, if o_1=po.remove(), o_2=po.remove() , …, o_n=po.remove() is the sequence of objects returned by n successive applications of the **remove** method without intervening calls to **enter** and leaving the PODS empty, then po will be said to *produce* the sequence $\{o_1,o_2,\ldots,o_n\}$. In particular, if po.isEmpty() is true, then po produces the empty sequence {}.

The actual names of the **remove** and **enter** methods will change from data type to data type. (In the C++ STL, these operations are named *push*, *pop*, and *empty* for all container adapters, although in other contexts one usually employs the push-pop terminology only for stacks.)

A process-oriented data structure is dynamic in that its state, and thus the sequence it produces, changes each time an **enter** or **remove** operation is performed on it.

Different types of process-oriented data structures are distinguished by the relationship between the sequences they accept and those they produce, and in some cases by the types of data they can accept.

A *stack* is a PODS whose **enter** and **remove** operations are normally called *push* and *pop* respectively and satisfy the following condition: *If a stack, st, is in a state in which it would produce* $\{o_1,o_2,\ldots,o_n\}$*, then st.push(x) will produce* $\{x,o_1,o_2,\ldots,o_n\}$. In particular, this means that if st accepts the sequence $\{a_1,a_2,\ldots,a_n\}$ then it will produce $\{a_n,a_{n-1},\ldots,a_1\}$.

In the case of a *queue*, the **enter** and **remove** operations are normally called *enqueue* and *serve* and satisfy the following additional condition: *If a queue, q, is in a state in which it would produce* $\{o_1,o_2,\ldots,o_n\}$*, then q.enqueue(x) will produce* $\{o_1,o_2,\ldots,o_n,x\}$. If queue q accepts the sequence $\{a_1,a_2,\ldots,a_n\}$ then it produces the identical sequence.

For a *priority queue*, the key operations have the same names as those of a queue, the elements to be enqueued must be prioritized by means of a priority function which returns results of type float, and the following condition is satisfied*: The elements accepted by a priority queue are produced by it in non-increasing order of priority and elements of equal priority are produced in queue order*.

The specification of a *weak priority queue* is almost identical to that of a priority queue, except that there is no restriction on the order in which elements of equal priority are produced.

One could define a *priority stack* in almost the same way, except that elements of equal priority are produced in stack order. Another possible variation on the theme is a *randomizing queue* which produces a random permutation of any sequence it accepts.

It is interesting to note that an *iterator*, a companion data type to a storage or descriptive data structure, operates in a similar way to a process-oriented data structure. An iterator is used in sequentially processing elements of a storage structure or descriptive data structure. A minimal operations protocol for iterators consists of the following operations:

```
boolean hasNext();    // equivalent to !isEmpty()
Object next();        // returns the next element and removes it from the iterator
```

In this case, there is no **enter** operation. Instead, an iterator accepts all the elements it will process at the time of its creation. The order in which the elements are produced by an iterator is dependent on the type of the companion data structure and is determined as part of the specification of that data type.

Most implementations of the basic process-oriented data structures make use of linear storage structures, such as arrays or linked lists. However, a weak priority queue is best implemented as a binary heap, whose underlying structure is that of a complete binary tree, a non-linear concept. Iterators make use of the structure of their companion data structure, which may be non-linear.

DESCRIPTIVE DATA STRUCTURES

A *descriptive data structure* is a logical data structure for representing a group of objects and the relationships between those objects. The generic name commonly used for a descriptive data structure is *collection*, but the term covers a wide variety of different data types, including sets, multi-sets, simple linear and circular lists, binary and ordered tree collections, graphs, relations and maps.

Although descriptive data structures can be modified through addition, removal or updating of elements, they don't change state as a result of any of these operations the way process-oriented structures do. One should think of descriptive data structures as being static in comparison to the dynamic process-oriented data structures.

A minimal operations protocol for a collection is as follows:

```
Collection add(Object o);          // adds o to the Collection
Object remove(Object o);  // returns the first occurrence of o found in the
                                   // Collection, if any, and drops it from Collection
Iterator iterator();               // returns an iterator for the Collection
```

The first two operations are similar to the **enter** and **remove** operations for process-oriented data structures, but there are no rules as to the order in which elements are added or removed. Since there is no prescribed order of removal of elements, the **remove** operation usually requires an argument used to search the Collection for the element to be removed.

The iterator operation returns an iterator for the collection, which accesses the elements of the collection sequentially. The order in which the elements are produced by the iterator is part of the specification of the collection. For instance, for a set there is no prescribed order, while elements of a linear list are produced in the linear order of the list from first to last.

The existence of the iterator operation enables one to define a number of composite operations that are not provided for in the original specification, such as **isEmpty** or **size**.

The static nature of descriptive data structures makes it important to insure their integrity during execution. In particular, one must guard against inadvertent modification of their elements. For that reason, when the elements of a descriptive data structure are mutable (modifiable), it is best to populate the data structure with copies of the elements rather than the elements themselves. That insures against an external reference to such an element being used to modify the element and thereby the structure itself.

For example, consider a set S of arrays of integers, and assume that the following instructions add two new arrays to the collection:

```
int[] x = {1,2}, y = {4,2};
S.add(x);
S.add(y);
```

Then, the instruction x[0] = 4; would modify not only the x array, but would also affect the set S, since x still references the first object added to S. In this case, the side effect not only modifies the set, but actually destroys its set property, since the collection now contains two equal elements.

One avoids this kind of side effect by replacing the instruction S.add(x) with S.add(x.clone()). (The clone operation creates a copy of the object to which it is applied.)

For the same reason, when an element of a collection is accessed by an iterator or some other operation, a copy of the element rather than the element itself may be returned.

SPECIFICATIONS OF STRUCTURED COLLECTIONS

As in the process-oriented case, one would want to give specifications of descriptive data structures that do not suggest particular implementations. However, in this case, the focus has to be on the internal structure.

For a collection to be a set, the only additional proviso required is that all the elements be distinct, and that has no implementation implications. Also, relations and maps are just sets of ordered pairs of objects.

For structured collections, such as lists, or binary or ordered tree collections, the situation is more complicated. One often uses graphical representations to describe such structures, but that seems to hint at a linked implementation and may obscure the distinction between the abstract data type and its implementation.

Another approach is to describe the structure in set-theoretic terms. For example, a binary tree is defined as a set that is either empty or consists of three disjoint components: a root element, and two subsets, the left and right subtree, that are themselves binary trees. That definition describes the nodes of a binary tree storage structure, but it is often unclear how it relates to the collection of objects (not necessarily all distinct) stored at those nodes.

An alternative, implementation-independent approach uses the concept of indexing: A collection is said to be *indexed* by a set, IT, of integer tuples if there exists a mapping from IT onto the collection. If the *index set*, IT, has some underlying structure, then that structure translates into a structure for the indexed collection. 1-tuples are simply integers.

That approach is familiar for linear lists, which are often described as collections of elements numbered using integer indices belonging to an interval [0,n-1]. See for example [2,3]. The integer incrementing operation makes it possible to step through the index set and thus the collection. By modifying the incrementing operation to use modular arithmetic, one can use the same approach in specifying a circular list.

As noted in [4], indexing can also provide an implementation-independent way of specifying ordered or binary tree collections. For instance, one can define an ordered index tree to be a non-empty set, OT, of integer tuples satisfying the following: If $(x_0,\ldots,x_m)$ is a

tuple of minimum length in OT, and $(x_0,\ldots,x_n)$ is any other element of OT, then $\text{parent}(x_0,\ldots,x_n) = (x_0,\ldots,x_{n-1}) \in \text{OT}$, and if $x_n > 0$, $\text{leftSibling}(x_0,\ldots,x_n) = (x_0,\ldots,x_{n-1}) \in \text{OT}$.

An ordered tree collection is then defined as an indexed collection whose index set is an ordered index tree. Again, the specified ordered index tree operations translate into operations on the ordered tree collection. The Dewey Decimal numbering system in a table of contents tree basically makes use of this approach.

By restricting the integers in the index tuples to belong to the interval [0,k-1], we can define an ordered tree of order k. A slight variation on the theme yields a binary tree collection.

In the foregoing, the terms *binary tree collection* and *ordered tree collection* are used rather than just *binary tree* and *ordered tree* to emphasize that the discussion deals with structured collections rather than the graph theoretic tree concepts.

AN ILLUSTRATION

To illustrate the differences between the three types of data structures, consider how a company might handle orders for its products: It would maintain a log of such orders in a (chronological) linear list, but at the same time it would place each order into a queue for servicing by the shipping department. The data in the order list are permanent and can be accessed later for analysis or evaluation. On the other hand the data in queue are transient presented for service and removed from the queue in the order in which they are received. Although the two types of data structures have very different purposes, they can both be implemented using either an array or a simple linked list as the underlying storage structure.

SUMMARY

This paper has described a division of most data structures into three categories: storage structures and process-oriented and descriptive data structures, and it has pointed out the differences between the categories and the commonalities of data structures within each category. Some of these distinctions have been discussed elsewhere [6], but by naming the categories and giving implementation-independent specifications in the second and third category, the paper sought to clarify the relationship between the various types of data structures and to provide a useful framework for their study.

REFERENCES

[1] Borland C++ Classic Libraries Guide (1994), Scotts Valley, CA: Borland International, 1994

[2] Chan, P., Lee, R., Kramer, D., The Java™ Class Libraries, Second Edition, Volume 1, Reading, MA: Addison-Wesley, 1999.

[3] Collins, W.J., Data Structures and the Java Collections Framework, Boston, MA: McGraw-Hill, 2002

[4] Knuth, D. E., The Art of Computer Programming, Volume1, Third Edition, Reading, MA: Addison-Wesley , 1997.

[5] Kohl, N., C/C++ Reference, http://www.cppreference.com, last update 6/27/07.

[6] Lafore, R., Data Structures & Algorithms in Java™, Second Edition, Indianapolis, IN: Sams Publishing, 2003

[7] Tremblay, J.P., Sorensen, P.G., An Introduction to Data Structures with Applications, New York, NY, McGraw-Hill, 1984

[8] Weiss, M.A., Data Structures and Algorithm Analysis in C++, Menlo Park, CA: Addison-Wesley, 1994