Data Structures and Algorithm Analysis

Edition 3.2 (C++ Version)

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

March 28, 2013
Update 3.2.0.10
For a list of changes, see
http://people.cs.vt.edu/~shaffer/Book/errata.html

If you wish to have a printed version of this document, print copies are published by Dover Publications
(see http://store.doverpublications.com/048648582x.html).
Further information about this text is available at
http://people.cs.vt.edu/~shaffer/Book/.

4

Lists, Stacks, and Queues

If your program needs to store a few things — numbers, payroll records, or job descriptions for example — the simplest and most effective approach might be to put them in a list. Only when you have to organize and search through a large number of things do more sophisticated data structures usually become necessary. (We will study how to organize and search through medium amounts of data in Chapters 5, 7, and 9, and discuss how to deal with large amounts of data in Chapters 8–10.) Many applications don't require any form of search, and they do not require that any ordering be placed on the objects being stored. Some applications require processing in a strict chronological order, processing objects in the order that they arrived, or perhaps processing objects in the reverse of the order that they arrived. For all these situations, a simple list structure is appropriate.

This chapter describes representations for lists in general, as well as two important list-like structures called the stack and the queue. Along with presenting these fundamental data structures, the other goals of the chapter are to: (1) Give examples of separating a logical representation in the form of an ADT from a physical implementation for a data structure. (2) Illustrate the use of asymptotic analysis in the context of some simple operations that you might already be familiar with. In this way you can begin to see how asymptotic analysis works, without the complications that arise when analyzing more sophisticated algorithms and data structures. (3) Introduce the concept and use of dictionaries.

We begin by defining an ADT for lists in Section 4.1. Two implementations for the list ADT — the array-based list and the linked list — are covered in detail and their relative merits discussed. Sections 4.2 and 4.3 cover stacks and queues, respectively. Sample implementations for each of these data structures are presented. Section 4.4 presents the Dictionary ADT for storing and retrieving data, which sets a context for implementing search structures such as the Binary Search Tree of Section 5.4.

4.1 Lists

We all have an intuitive understanding of what we mean by a "list." Our first step is to define precisely what is meant so that this intuitive understanding can eventually be converted into a concrete data structure and its operations. The most important concept related to lists is that of **position**. In other words, we perceive that there is a first element in the list, a second element, and so on. We should view a list as embodying the mathematical concepts of a sequence, as defined in Section 2.1.

We define a **list** to be a finite, ordered sequence of data items known as **elements**. "Ordered" in this definition means that each element has a position in the list. (We will not use "ordered" in this context to mean that the list elements are sorted by value.) Each list element has a data type. In the simple list implementations discussed in this chapter, all elements of the list have the same data type, although there is no conceptual objection to lists whose elements have differing data types if the application requires it (see Section 12.1). The operations defined as part of the list ADT do not depend on the elemental data type. For example, the list ADT can be used for lists of integers, lists of characters, lists of payroll records, even lists of lists.

A list is said to be **empty** when it contains no elements. The number of elements currently stored is called the **length** of the list. The beginning of the list is called the **head**, the end of the list is called the **tail**. There might or might not be some relationship between the value of an element and its position in the list. For example, **sorted lists** have their elements positioned in ascending order of value, while **unsorted lists** have no particular relationship between element values and positions. This section will consider only unsorted lists. Chapters 7 and 9 treat the problems of how to create and search sorted lists efficiently.

When presenting the contents of a list, we use the same notation as was introduced for sequences in Section 2.1. To be consistent with **C**++ array indexing, the first position on the list is denoted as 0. Thus, if there are n elements in the list, they are given positions 0 through $n-1$ as $\langle a_0,\ a_1,\ ...,\ a_{n-1}\rangle$. The subscript indicates an element's position within the list. Using this notation, the empty list would appear as $\langle\rangle$.

Before selecting a list implementation, a program designer should first consider what basic operations the implementation must support. Our common intuition about lists tells us that a list should be able to grow and shrink in size as we insert and remove elements. We should be able to insert and remove elements from anywhere in the list. We should be able to gain access to any element's value, either to read it or to change it. We must be able to create and clear (or reinitialize) lists. It is also convenient to access the next or previous element from the "current" one.

The next step is to define the ADT for a list object in terms of a set of operations on that object. We will use the **C**++ notation of anabstract class to formally define

the list ADT. An abstract class is one whose member functions are all declared to be "pure virtual" as indicated by the "**=0**" notation at the end of the member function declarations. Class **List** defines the member functions that any list implementation inheriting from it must support, along with their parameters and return types. We increase the flexibility of the list ADT by writing it as a **C**++ template.

True to the notion of an ADT, anabstract class does not specify how operations are implemented. Two complete implementations are presented later in this section, both of which use the same list ADT to define their operations, but they are considerably different in approaches and in their space/time tradeoffs.

Figure 4.1 presents our list ADT. Class **List** is a template of one parameter, named **E** for "element". **E** serves as a placeholder for whatever element type the user would like to store in a list. The comments given in Figure 4.1 describe precisely what each member function is intended to do. However, some explanation of the basic design is in order. Given that we wish to support the concept of a sequence, with access to any position in the list, the need for many of the member functions such as **insert** and **moveToPos** is clear. The key design decision embodied in this ADT is support for the concept of a **current position**. For example, member **moveToStart** sets the current position to be the first element on the list, while methods **next** and **prev** move the current position to the next and previous elements, respectively. The intention is that any implementation for this ADT support the concept of a current position. The current position is where any action such as insertion or deletion will take place.

Since insertions take place at the current position, and since we want to be able to insert to the front or the back of the list as well as anywhere in between, there are actually $n + 1$ possible "current positions" when there are n elements in the list.

It is helpful to modify our list display notation to show the position of the current element. I will use a vertical bar, such as $\langle 20,\ 23 \mid 12,\ 15\rangle$ to indicate the list of four elements, with the current position being to the right of the bar at element 12. Given this configuration, calling **insert** with value 10 will change the list to be $\langle 20,\ 23 \mid 10,\ 12,\ 15\rangle$.

If you examine Figure 4.1, you should find that the list member functions provided allow you to build a list with elements in any desired order, and to access any desired position in the list. You might notice that the **clear** method is not necessary, in that it could be implemented by means of the other member functions in the same asymptotic time. It is included merely for convenience.

Method **getValue** returns a pointer to the current element. It is considered a violation of **getValue**'s preconditions to ask for the value of a non-existent element (i.e., there must be something to the right of the vertical bar). In our concrete list implementations, assertions are used to enforce such preconditions. In a commercial implementation, such violations would be best implemented by the **C**++ exception mechanism.