Formal Specification, Design and Implementation of the List Data Structure

M. Devi Prasad
School of Information Sciences, Manipal University.
devi.prasad@manipal.edu

Abstract

We present a formal specification of the *List* data structure. First, we describe the essential properties of the list abstract data type. We subsequently develop a behavioral specification of the ADT. This step introduces a few design decisions that influence the representation and efficiency of operations on lists. This, in turn, helps in shaping a concrete implementation in the C programming language. This entire exercise demonstrates how formal specification may guide different design choices, and thus lead to different implementations.

1 Introduction

Most textbooks employ a pattern in describing data structures: first informally describe the behavior of a particular data structure, and follow it up with an implementation in a high level programming language or pseudo code. The informal descriptions use diagrams, operation traces and textual narratives. The time and space complexity of algorithms, and implementation efficiency are discussed to some extent in many books. The depth and breadth of exposition depends on factors such as author's predispositions and the target audience.

We wish to take a reasonably different approach to the way data structures course is taught. We wish to bring a little more engineering discipline to describing and developing data structures: we first bring out a specification, in some detail, of the interfaces of the underlying ADT. We follow it up with the design of the representation and associated behaviors of the data structure. We finally implement at the least two different versions in order to understand the pros and cons of different implementation strategies.

2 Notation and Conventions

We employ sets, sequences and tuples to develop an expressive formal specification of the list data structure. We use the following notation:

Sequence	$[a_1, ..., a_n]$	*where* $0 \leq n$
n-tuple	$(a_1, a_2, ..., a_n)$	

In the following sections, we will be using sequences to represent the actual data elements held by list instances. An empty list content is represented as $[]$ and a non-empty sequence of list elements is shown as $[a_1, ... a_n]$.

We write $d :: s$ to mean a sequence of at least one element with item 'd' at the head of the sequence; 's' in this case represents the rest of the sequence which may be empty.

Certain list operations defined in the next section employ 2-tuples to represent the `Result` of their execution. A 2-tuple of type $\mathbb{N} \times \{OK, EMPTY\}$ is used to hold the answer ($\mathbb{N}$ component) and the (potential error) status of the operation `OK` and `EMPTY`.

3 The Definition of List and Result Interfaces

Lists are very simple data containers. In our definition, new elements can be added to the list at the first and last positions, also called the *head* and *tail* of the list.

To indicate anomalies that may occur while attempting list operations, we define an auxiliary interface named **Result**. A Result object indicates either the answer of the successful operation or an anomalous state.

3.1 The List Interface

A list instance has no logical limit on its size. It may grow or shrink in size. The list constructor has no arguments, and thus, has the following signature:

new : $() \rightarrow$ List

We define only a few operations to inspect the state of a list instance.

empty : List $\rightarrow$ Boolean

full : List $\rightarrow$ Boolean

length : List $\rightarrow \mathbb{N}$

The **head** and **tail** operations access the data stored at the first and last positions in the array:

head : List $\rightarrow$ Result

tail : List $\rightarrow$ Result

The **add_head** and **add_tail** operations mutate the list. On an empty list, **remove_head** and **remove_tail** indicate the status as EMPTY in the Result component.

add_head : List $\times \mathbb{N} \rightarrow$ List

add_tail : List $\times \mathbb{N} \rightarrow$ List

remove_head : List $\rightarrow$ List $\times$ Result

remove_tail : List $\rightarrow$ List $\times$ Result

3.2 The Result Interface

The **Result** interface has a constructor, and value and status component extractors:

result : $\mathbb{N} \times$ Status $\rightarrow$ Result

val : Result $\rightarrow \mathbb{N}$

status : Result $\rightarrow$ Status

Status is a set of two values that indicate whether an operation succeeded or not:

Status $= \{OK, EMPTY\}$

4 The Semantics of List Operations

We use sequences to represent lists. So, **new** creates an empty sequence:

$$\textbf{new}() = [\,]$$

We define two state inspector operations, *empty* and *full*, by pattern-matching over the structure of list.

$$\textbf{empty}(\texttt{list}) = \begin{cases} true & \text{if } \texttt{list} = [\,] \\ false & \text{otherwise.} \end{cases}$$

In our definition, there is no logical limit on the capacity of list instances:

$$\textbf{full}(\texttt{list}) = false$$

The **length** operation is inductively defined over the list structure. The length of an empty list is 0, and the length of a general list is one more than the length of the list obtained by removing its head.

$$\textbf{length}(\texttt{list}) = \begin{cases} 0 & \text{if } \texttt{list} = [\,] \\ 1 + \textbf{length}(\texttt{list}') & \text{if } \texttt{list} = \_ :: \texttt{list}' \end{cases}$$

Reading the value at the head of the list is trivial:

$$\textbf{head}(\texttt{list}) = \begin{cases} (\_, EMPTY) & \text{if } \texttt{list} = [\,] \\ (x, OK)) & \text{if } \texttt{list} = \texttt{x} :: \texttt{list}' \end{cases}$$

Accessing the tail of the list requires careful thinking. Since we have not defined an operation to quickly reach the end of the list, we need to iterate over the entire list. We specify this operation in three separate cases:

$$\textbf{tail}(\texttt{list}) = \begin{cases} (\_, \mathit{EMPTY}) & \text{if } \texttt{list} = [\,] \\ (\texttt{x}, \mathit{OK})) & \text{if } \texttt{list} = \texttt{x} :: \texttt{list}' \\ & \quad \wedge\ \textbf{tail}(\texttt{list}') = (\_, \mathit{EMPTY}) \\ (\texttt{y}, \mathit{OK})) & \text{if } \texttt{list} = \texttt{x} :: \texttt{list}' \\ & \quad \wedge\ \textbf{tail}(\texttt{list}') = (\texttt{y}, \mathit{OK}) \end{cases}$$

Let us try to understand the three cases shown above. The case of an empty list is trivial. The second case deals with (sub)lists that hold only one element. This, in fact, may represent the final suffix of the sequence of elements - one that contains the last element of the list. The third case is the most common case wherein we strip the head of the list and look for the tail element in the rest of the list, named $\texttt{list}'$, in the above. The result of this step defines the result of the entire operation.

We will now specify two simple operations: **add_head** and **remove_head**. We should keep in mind that the former simply returns a list while the latter answers with a 2-tuple containing the new list and the status of the operation:

$$\textbf{add\_head}(\texttt{list}, \texttt{x}) = \texttt{x} :: \texttt{list}$$

$$\textbf{remove\_head}(\texttt{list}) = \begin{cases} ([\,], (\_, \mathit{EMPTY})) & \text{if } \texttt{list} = [\,] \\ (\texttt{list}', (\texttt{x}, \mathit{OK})) & \text{if } \texttt{list} = \texttt{x} :: \texttt{list}' \end{cases}$$

The specification of **add_tail** operation is reasonably easy. This operation returns a new list containing the given element in the tail position. Because it is not possible (in our case, in this specification) to simply access the last position in the list, we need to iterate over list's structure.

$$\textbf{add\_tail}(\texttt{list}, \texttt{x}) = \begin{cases} [\texttt{x}] & \text{if } \texttt{list} = [\,] \\ \texttt{a} :: \textbf{add\_tail}(\texttt{list}', \texttt{x}) & \text{if } \texttt{list} = \texttt{a} :: \texttt{list}' \end{cases}$$

Adding an element to an empty list yields a list containing the element. The most common scenario, represented by the second case, requires attention. Notice how we retain the structure of the origin list. As long as the list is not empty we keep prodding. We retain the contents of original list by appending the head of the list to the result of adding the given element to the tail of the current list (referred to as $\texttt{list}'$ above).

Finally, we come to the specification of **remove_tail**. The basic idea is to iterate over the list and when we reach the last element, return an empty list as an indication of dropping the last element from the result. In the other case, we retain contents of the original list just as **add_tail** does.

$$\textbf{remove\_tail}(\texttt{list}) = \begin{cases} ([\,], (\_, \mathit{EMPTY})) & \text{if } \texttt{list} = [\,] \\ ([\,], (\texttt{x}, \mathit{OK})) & \text{if } \texttt{list} = \texttt{x} :: \texttt{list}' \\ & \quad \wedge\ \textbf{remove\_tail}(\texttt{list}') = ([\,], (\_, \mathit{EMPTY})) \\ (\texttt{x} :: \texttt{list}'', (\texttt{y}, \mathit{OK})) & \text{if } \texttt{list} = \texttt{x} :: \texttt{list}' \\ & \quad \wedge\ \textbf{remove\_tail}(\texttt{list}') = (\texttt{list}'', (\texttt{y}, \mathit{OK})) \end{cases}$$

Note that in the second case, we return a Result tuple containing the element in the end of the given list. This is the case that detects the singleton list (that is, a list of only one element). Although the Result tuple contains the last element, we are actually interested in the first component - the list component which is empty in this particular case. In other words, this case strips the last element from the list and returns an empty list as its answer.

In the third case, **remove_tail** is recursively called to process the sublist. The head element, `x`, of the current list is appended to $\texttt{list}''$, the list returned by processing the sublist. The element, `y`, which was at the tail of the original list, is propagated all way up the call chain.