6 The Singly-Linked Bounded List

Now that the unbounded list forms have been covered we can move on to their bounded counterparts. The topic of this chapter is the singly-linked form, while the doubly-linked variant is presented in Chapter 7.

6.1 Singly-Linked Bounded List Interface

The interface to the singly-linked bounded list composed of generic Items is presented below. For reasons (to be explained in Section 6.1.1) the bounded list differs somewhat from the unbounded. Like the unbounded list, the bounded list does not concern itself with the data type of the objects or items manipulated.

Table 6.1 follows the interface module with a summary of algorithmic complexity measures for the exported routines.

DEFINITION MODULE ListSBM;
(*------------------------------------------------------------------*)
  Version : 1.00 26 Apr 1988 C. Lins
  Compiler : TML Modula-2 Compiler for the Apple Macintosh
  Component: Polylithic Structures - List
           Singly-Linked Bounded Managed

  Revision History
  v1.00 26 Apr 1988  C. Lins
         Initial implementation for TML Modula-2.
(*------------------------------------------------------------------*)

FROM Items IMPORT
  (*---Type*) Item;

FROM ErrorHandling IMPORT
  (*---Type*) HandlerProc;

FROM ListEnum IMPORT
  (*---Type*) Exceptions;

(*------------------------------------------------------------------*)
6.1.1 Type Declarations

6.1.1.1 Approach # 1

The reader may remember from the bounded structures in Volume 1 that it was sufficient to modify the unbounded form of the Create operation to include a parameter specifying the structure's desired maximum size. The selector SizeOf was added as a convenience, allowing the client to easily retrieve this limit. This approach could be taken for bounded lists as well, associating each list created with its own pool of nodes. This is not necessarily bad, but could be very wasteful of space since the pools (and the nodes resident there) could not be shared by all bounded lists of the same type.

6.1.1.2 Approach # 2

Another approach involves exporting an additional data type representing the node pool along with operations to create and destroy node pools. In this case, each list operation would be required to provide the appropriate pool as a parameter. An arbitrary number of lists could then be associated with a given node pool, drawing list nodes from the same pool. Drawbacks inherent in this approach are:

- the additional pool parameter for each list operation makes it more difficult to replace unbounded lists with bounded lists (and vice versa) in client modules due to the extra work needed to either add or remove references to the node pool.

- operations with more than one list parameter, such as Assign and IsEqual, would be constrained to lists from a single node pool, or would necessitate two node pool parameters (one for each list).

- the security of the list abstraction is compromised when employing multiple node pools since the client could inadvertently use the wrong node pool for a given list causing unpredictable results.