Chapter **17**

Stack and List Implementations

Chapter 6 introduces the concept of an abstract data structure (ADS) and an abstract data type (ADT). It illustrates each of these abstractions with stacks and lists. An ADT differs from an ADS because the server module exports a type. The client module is then able to declare its own variable with that type and can have as many data structures as it needs. The data structures exist in the client module. For example, the module of Figure 7.10 declares two stacks, stackA and stackB. Because an ADS does not export a type, there is only one data structure and it is hidden in the server module. For example, the module of Figure 17.4 does not contain any stack variables. It simply manipulates the one stack contained in PboxStackADS.

Chapter 9 introduces the concept of a class, which is the object-oriented equivalent of an ADT. It introduces classes so you can see how to use the MVC objects provided by the BlackBox framework. The module in Figure 9.3 uses the stack class from PboxStackObj to manipulate two stacks. For now, there is no apparent advantage of using a class instead of an ADT. A demonstration of the advantage of using classes is postponed until a later chapter.

In all the previous chapters you accessed the ADS, the ADT, or the class by inspecting its corresponding interface, which consists of all the items exported by the server module. You then wrote your client modules accessing those items provided by the server. It was not necessary for you to know how the data structure is implemented in order to use the data structure. This chapter shows the implementations of the stack ADS, the stack class, and the list ADT that were previously hidden.

Stack ADS implementation

Figure 17.1 shows the interface for the stack abstract data structure from PboxStackADS. The server module exports five items—the constant capacity that specifies the maximum number of values that can be stored in the structure, and the four procedures Clear, NumItems, Pop, and Push. Procedure Push gives the client the ability to store a value in the data structure, and procedure Pop gives the ability to retrieve a value. Stacks are last in, first out (LIFO) structures, so that when a client executes the Pop procedure the value retrieved is the most recent value pushed. Figure 17.2 shows a sequence of push and pop operations on the stack.

```
DEFINITION PboxStackADS;

    CONST
        capacity = 8;

    PROCEDURE Clear;
    PROCEDURE NumItems (): INTEGER;
    PROCEDURE Pop (OUT val: REAL);
    PROCEDURE Push (val: REAL);

END PboxStackADS.
```

Figure 17.1
The interface of the stack abstract data structure.

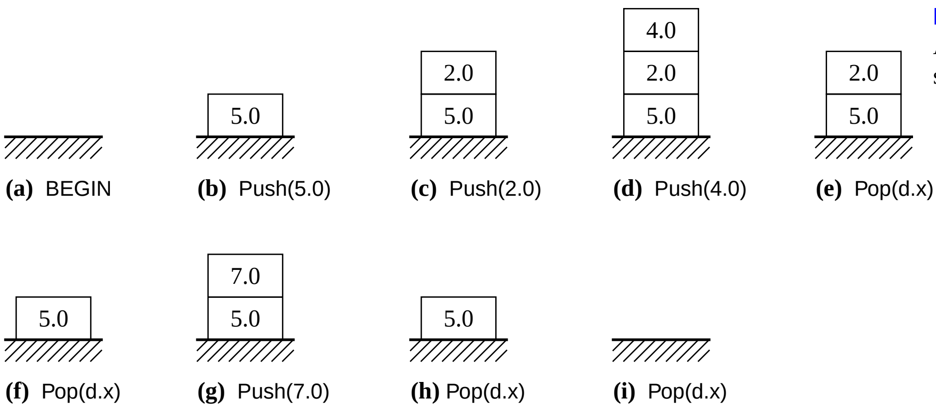


Figure 17.2
A sequence of operations on a stack.

As you might suspect by now, a straightforward implementation of the stack is to use an array to store the values. When the client executes the push operation, you simply store the value in the array. Of course, you need to keep track of where the most recent value was stored so you will know where to store the value pushed. You can use an integer variable whose value will be the index of the most recently stored value. Figure 17.3 is a diagram of the values that the array, named body, and the integer, named top, acquire assuming the same sequence of pushes and pops as in Figure 17.2.

Figure 17.4 is the corresponding implementation of the stack abstract data structure. The module contains two global variables, body and top, that are necessary to maintain the state of the stack between invocations of the procedures. body is an array of eight reals and top is an integer. To clear the stack top is set to –1, which indicates that no items are stored in the array. Procedure Clear performs the operation with the simple assignment

```
top := -1
```

You can see from Figure 17.3 that at any point in time, the number of values stored in the stack is one more than the value of top. For example, in part (d) three

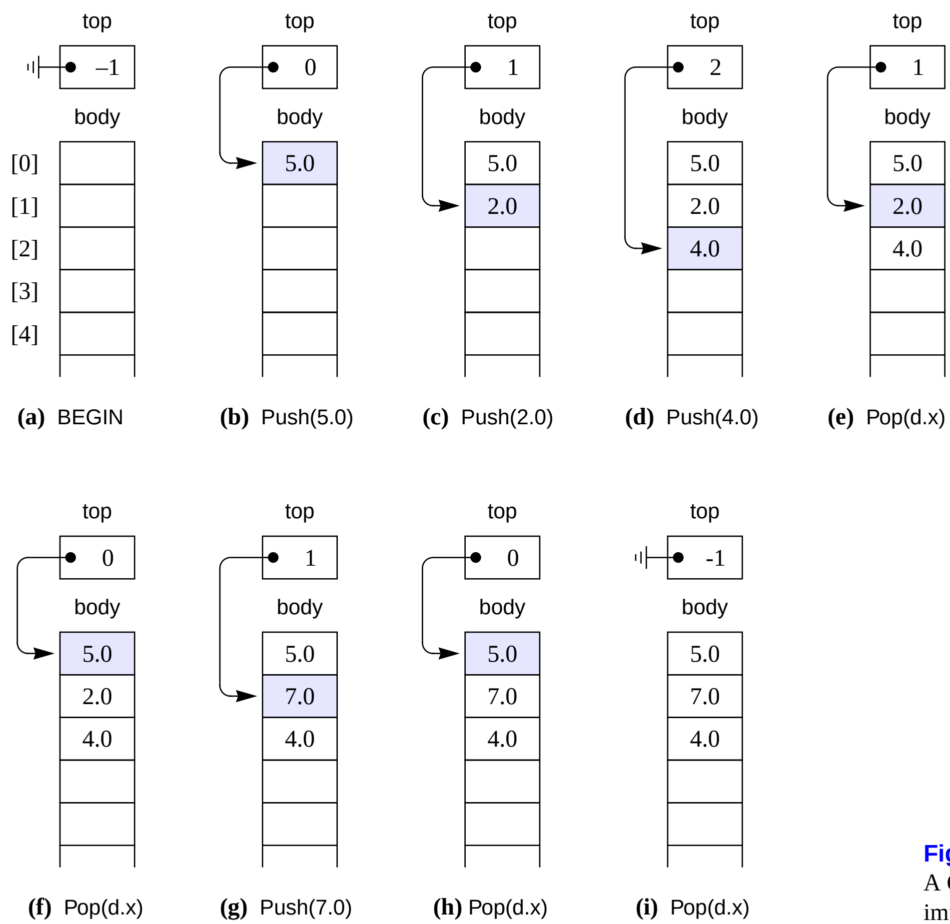


Figure 17.3
A Component Pascal implementation of a stack.

values are stored and the value of top is two. Therefore, function procedure NumItems simply executes the single statement

```
RETURN top + 1
```

The documentation for the push procedure is

PROCEDURE **Push** (val: REAL)
Pre
NumItems() < capacity   20
Post
val is pushed onto the top of the stack.

Procedure Push implements the precondition with the assertion

```
ASSERT(top < capacity - 1, 20)
```

Using the ASSERT statement to implement the precondition is consistent with the design-by-contract rule, which states

The design-by-contract rule

- IF in the client.
- ASSERT in the server.

```
MODULE  PboxStackADS;

    CONST
        capacity* = 8;
    VAR
        body: ARRAY capacity OF REAL;
        top: INTEGER;

    PROCEDURE  Clear*;
    BEGIN
        top := -1
    END Clear;

    PROCEDURE  NumItems* (): INTEGER;
    BEGIN
        RETURN top + 1
    END NumItems;

    PROCEDURE  Pop* (OUT val: REAL);
    BEGIN
        ASSERT(0 <= top, 20);
        val := body[top];
        DEC(top)
    END Pop;

    PROCEDURE  Push* (val: REAL);
    BEGIN
        ASSERT(top < capacity - 1, 20);
        INC(top);
        body[top] := val
    END Push;

END PboxStackADS.
```

Figure 17.4
The implementation of the stack abstract data structure.

The value of capacity -1 is the index of the last cell in the array. In this implementation, the capacity of the array is 8, and the index of the last cell is 7. To have room to put another value on the stack, variable top must be less than 7. The push operation is achieved with the assignments

```
INC(top);
body[top] := val
```

where val is the value supplied by the actual parameter from the client module. For example, in Figure 17.3(c) top has value 1. INC(top) gives it value 2, then body[2] gets val, as shown in part (d).

The documentation for procedure Pop is

```
PROCEDURE Pop (OUT val: REAL)
Pre
0 < NumItems()   20
Post
An item is removed from the top of the stack and val gets its value.
```

The precondition states that you cannot pop a value off the stack unless there is at least one value to be retrieved. Procedure Pop implements the precondition with the assertion

```
ASSERT(0 <= top, 20)
```

A value of zero for top indicates that one value is in the stack, as Figure 17.3(b) shows. Procedure Pop implements the retrieval with the statements

```
val := body[top];
DEC(top)
```

Because top is the index of body where the most recent value was stored, you must make the assignment to formal parameter x before you decrement top. Note how this is consistent with the implementation of procedure Push, in which the INC operation occurs before the assignment to body[top].

Stack class implementation

Figure 17.5 shows the interface of the stack class from module PboxStackObj. It differs from the stack ADS because the type Stack is exported.

```
DEFINITION PboxStackObj;

   CONST
      capacity = 8;

   TYPE
      Stack = RECORD
         (VAR s: Stack) Clear, NEW;
         (IN s: Stack) NumItems (): INTEGER, NEW;
         (VAR s: Stack) Pop (OUT val: REAL), NEW;
         (VAR s: Stack) Push (val: REAL), NEW
      END;

END PboxStackObj.
```

Figure 17.5
The interface of the stack class.

The concept of implementing the stack class with an array is identical to the concept of implementing the stack abstract data structure in the previous section. You have an array named body that stores the values, and you have an integer variable named top that stores the index of the array where the most recent value was pushed. You clear the array by setting top to –1, procedure NumItems returns one plus the value of top, Pop assigns to x then decrements top, and Push increments top then assigns to body[top]. The assertions are implemented as they are with the stack ADS. Figure 17.6 shows the implementation.

Figure 17.6
The implementation of the stack class.

```
MODULE PboxStackObj;

    CONST
        capacity* = 8;
    TYPE
        Stack* = RECORD
            body: ARRAY capacity OF REAL;
            top: INTEGER
        END;

    PROCEDURE (VAR s: Stack) Clear*, NEW;
    BEGIN
        s.top := -1
    END Clear;

    PROCEDURE (IN s: Stack) NumItems* (): INTEGER, NEW;
    BEGIN
        RETURN s.top + 1
    END NumItems;

    PROCEDURE (VAR s: Stack) Push* (val: REAL), NEW;
    BEGIN
        ASSERT(s.top < capacity - 1, 20);
        INC(s.top);
        s.body[s.top] := val
    END Push;

    PROCEDURE (VAR s: Stack) Pop* (OUT val: REAL), NEW;
    BEGIN
        ASSERT(0 <= s.top, 20);
        val := s.body[s.top];
        DEC(s.top)
    END Pop;

END PboxStackObj.
```

Compare the interface of the class in Figure 17.5 with the implementation in Figure 17.6. What is contained between the lines

```
Stack = RECORD

END
```

in each case? The interface does not show body or top in the record for the Stack class, but the implementation does. Furthermore, the interface shows the procedure headings within the record, but the procedures are contained outside the record in the implementation. Why are the interface and the implementation different in these two respects?

One big advantage of the Component Pascal language over many other object-oriented languages is that the interface is generated automatically by the compiler. With other languages, the programmer must write not only the implementation but the corresponding interface as well. So it is the Component Pascal compiler that generates the interface from the implementation. Figure 17.6 shows that Stack is exported with the * export mark but body and top are not. That is why body and top do not appear in the interface. They are both part of class Stack but are hidden from the client.

Automatic generation of the interface is also the reason for the procedure headings appearing inside the Stack record. When the compiler processes the source code, it detects the presence of an exported method by the existence of the receiver in front of the procedure name. The type of the receiver determines the placement of the procedure heading in the interface. For example, when the compiler scans the source line

```
PROCEDURE (VAR s: Stack) Clear*, NEW;
```

it detects the receiver (VAR s: Stack). The type of the receiver is Stack, so the line

```
(VAR s: Stack) Clear, NEW;
```

is inserted in the Stack record in the interface.

Why does the interface display class methods this way? To emphasize that class methods belong to the class record. The style is consistent with the manner in which methods are called. For example, suppose you have a record

```
d*: RECORD
    valuePushed*, valuePopped-: REAL;
    numItemsA-, numItemsB-: INTEGER
END;
```

How do you access one of the fields of the record, say valuePushed? You precede it with the record name with a period between the record name and the field. You refer to the valuePushed field of record d by writing

```
d.valuePushed
```

And how does a client module invoke a method? The style is the same as if the method were a field in the record. For example, if your client module declares

```
VAR
   stackA, stackB: PboxStackObj.Stack;
```

then to call the method to clear stackA, you write

```
StackA.Clear
```

This call is consistent with the interface

```
Stack = RECORD
   (VAR s: Stack) ClearStack, NEW;
END
```

In the same way that valuePushed belongs to the d record, Clear belongs to the Stack record.

The implementation of all the methods in the stack class is similar to the implementations in the stack ADS. Because body and top are part of a record, you simply use the record notation to refer to them. For example, the code for the push procedure with the ADS is

```
INC(top);
body[top] := val
```

where body and top are global variables in the server module. The corresponding code for the push method with the class is

```
INC(s.top);
s.body[s.top] := val
```

where record s is the formal parameter corresponding to the actual parameter in the client module.

List ADT implementation

Figure 17.7 shows the interface of the list ADT. As with the implementation of the stack, an array is a convenient data structure for implementing a list.

```
DEFINITION PboxListADT;

    CONST
        capacity = 8;

    TYPE
        List = RECORD END;
        T = ARRAY 16 OF CHAR;

    PROCEDURE Clear (VAR lst: List);
    PROCEDURE Display (IN lst: List);
    PROCEDURE GetElementN (IN lst: List; n: INTEGER; OUT val: T);
    PROCEDURE InsertAtN (VAR lst: List; n: INTEGER; IN val: T);
    PROCEDURE Length (IN lst: List): INTEGER;
    PROCEDURE RemoveN (VAR lst: List; n: INTEGER);
    PROCEDURE Search (VAR lst: List; IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN);

END PboxListADT.
```

Figure 17.7
The interface of the list abstract data type.

Figure 17.8 shows the implementation of the list ADT. The list record contains two fields—body, which is the array itself, and lastIndex, which is the index of the last item in the array.

The array has a capacity of eight, yet the body is declared to be an ARRAY 9 OF T. That is, there are nine cells in the array indexed from 0 to 8. The ninth cell at index 8 cannot be used by the client for storing a value. It is for storing the search value as a sentinel using the efficient version of the search algorithm in Figure 16.2. If the declaration for body did not allocate the extra cell, the algorithm could not do a sequential search when there are eight items in the list, because there would be no room for the sentinel.

```
MODULE  PboxListADT;
    IMPORT StdLog;

    CONST
        capacity* = 8;
    TYPE
        T* = ARRAY 16 OF CHAR;
        List* = RECORD
            body: ARRAY capacity + 1 OF T; (* + 1 necessary for procedure Search *)
            lastIndex: INTEGER
        END;

    PROCEDURE  Clear* (VAR lst: List);
    BEGIN
        lst.lastIndex := -1
    END Clear;
```

Figure 17.8
The implementation of the list abstract data type.

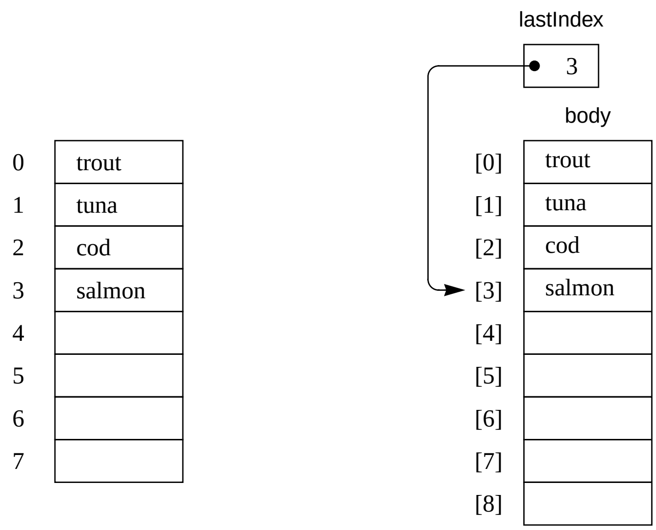
Figure 17.8
Continued.

```
PROCEDURE Display* (IN lst: List);
   VAR
      i: INTEGER;
BEGIN
   StdLog.Ln;
   FOR i := 0 TO lst.lastIndex DO
      StdLog.Int(i); StdLog.String(" "); StdLog.String(lst.body[i]); StdLog.Ln
   END
END Display;

PROCEDURE GetElementN* (IN lst: List; n: INTEGER; OUT val: T);
BEGIN
   ASSERT(0 <= n, 20);
   ASSERT(n <= lst.lastIndex, 21);
   val := lst.body[n]
END GetElementN;

PROCEDURE InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
   VAR
      i: INTEGER;
BEGIN
   ASSERT(0 <= n, 20);
   ASSERT(lst.lastIndex < capacity - 1, 21);
   IF n > lst.lastIndex + 1 THEN
      n := lst.lastIndex + 1
   END;
   FOR i := lst.lastIndex TO n BY -1 DO
      lst.body[i + 1] := lst.body[i]
   END;
   INC(lst.lastIndex);
   lst.body[n] := val
END InsertAtN;

PROCEDURE Length* (IN lst: List): INTEGER;
BEGIN
   RETURN lst.lastIndex + 1
END Length;

PROCEDURE RemoveN* (VAR lst: List; n: INTEGER);
   VAR
      i: INTEGER;
BEGIN
   ASSERT(0 <= n, 20);
   IF n <= lst.lastIndex THEN
      FOR i := n TO lst.lastIndex - 1 DO
         lst.body[i] := lst.body[i + 1]
      END;
      DEC(lst.lastIndex)
   END
END RemoveN;
```

```
PROCEDURE Search* (VAR lst: List; IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN);
BEGIN
   lst.body[lst.lastIndex + 1] := srchVal;
   n := 0;
   WHILE lst.body[n] # srchVal DO
      INC(n)
   END;
   fnd := n <= lst.lastIndex
END Search;

END PboxListADT.
```

Figure 17.8
Continued.

Figure 17.9(a) shows an abstract representation of a list containing four items. The first item is at position 0 and the last is at position 3. Figure 17.9(b) shows the implementation. The items are stored in the body part of the data structure. The position of each item corresponds to the index of the array. lastIndex has value 3, because that is the index of the last item in the array.



(a) The abstract list **(b)** The array implementation

Figure 17.9
The abstract list ADT and its array implementation.

The procedures for the list ADT are straightforward array manipulations. The Clear procedure simply sets lastIndex with

```
lst.lastIndex := -1
```

Any values remaining in the body of the array will be overwritten when the client inserts new values.

The Display procedure outputs lst.body[i] with

```
StdLog.Int(i); StdLog.String(" "); StdLog.String(lst.body[i]); StdLog.Ln
```

in the body of a FOR loop, with i ranging from 0 to lst.lastIndex.

Procedure GetElementN implements the preconditions

```
Pre
0 <= n   20
n < Length(lst)   21
```

with the ASSERT statements

```
ASSERT(0 <= n, 20);
ASSERT(n <= lst.lastIndex, 21)
```

Implementation of the second precondition is based on the fact that the index of the last item is one less than the length of the list. In Figure 17.9(b), lastIndex is 3 and the length of the list is 4. n is less than 4 if and only if it is less than or equal to 3. GetElementN has formal parameter val called by result. It sets the value of val by executing the statement

```
val := lst.body[n]
```

Procedure InsertAtN implements the preconditions

```
Pre
0 <= n   20
Length(lst) < capacity   21
```

with the ASSERT statements

```
ASSERT(0 <= n, 20);
ASSERT(lst.lastIndex < capacity - 1, 21)
```

If the preconditions are satisfied, it adjusts the value of n by comparing it with the index of the last item. The procedure allows the client to supply a large value of n, in which case the value gets inserted at the end of the list. The IF statement

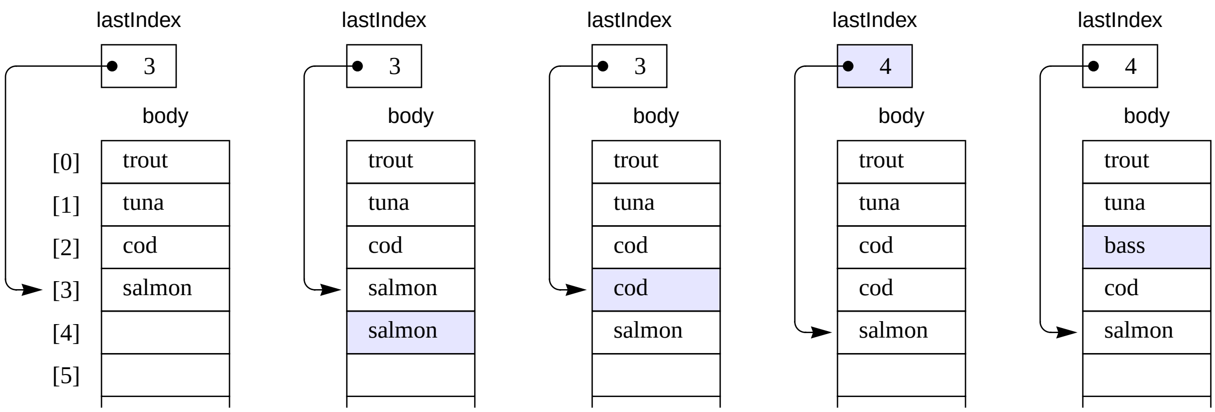
```
IF n > lst.lastIndex + 1 THEN
    n := lst.lastIndex + 1
END;
```

adjusts n to the position just after the last item, where the new value will be inserted, if the original value of n is beyond that position. Figure 17.10 shows the effect of

```
FOR i := lst.lastIndex TO n BY -1 DO
   lst.body[i + 1] := lst.body[i]
END;
INC(lst.lastIndex);
lst.body[n] := val
```

for the case of inserting value bass at position 2 for n. The elements of the array must be shifted down to make room for the inserted value.

Figure 17.10
Execution of InsertAtN with 2 for n and bass for val.



(a) Initial **(b)** lst.body[4] := lst.body[3] **(c)** lst.body[3] := lst.body[2] **(d)** INC (lst.lastIndex) **(e)** lst.body[n] := val

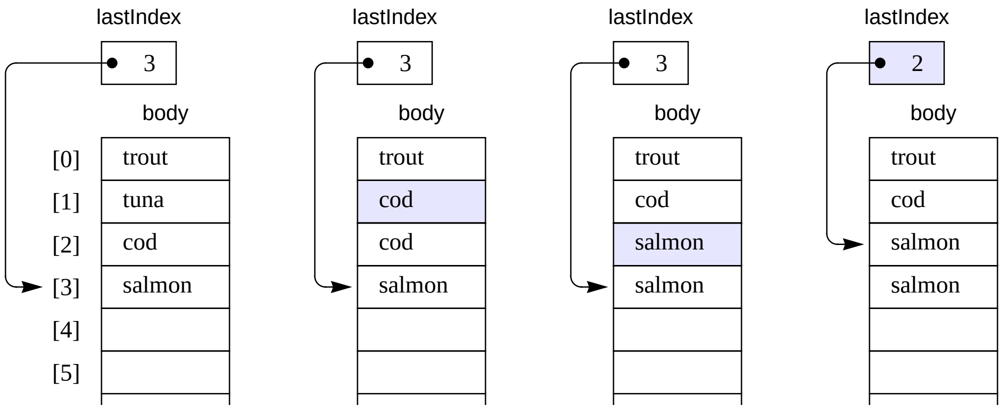
The implementation of procedure RemoveN first verifies the precondition that n is nonnegative with an appropriate ASSERT statement. The specification allows a large value of n, in which case the list is unchanged. No processing needs to be done unless n is less than or equal to lastIndex. Consequently, the processing is contained within the IF statement

```
IF n <= lst.lastIndex THEN
```

Figure 17.11 shows the effect of executing

```
FOR i := n TO lst.lastIndex - 1 DO
   lst.body[i] := lst.body[i + 1]
END;
DEC(lst.lastIndex)
```

for the case of removing the item at position 1. This time the items are shifted up and lastIndex is decremented. The garbage value at position 3 will be overwritten when the client inserts a new value later.



(a) Initial **(b)** lst.body[1] := lst.body[2] **(c)** lst.body[2] := lst.body[3] **(d)** DEC (lst.lastIndex)

Figure 17.11
Execution of RemoveN with 1 for n.

Problems

1. PboxStackADS in Figure 17.4 uses array body and integer top to implement a stack abstract data structure. At any given point in time, top has the index of the item on top of the stack. Modify the corresponding implementation of the abstract data type in PboxStackADT so that at any given point in time top will have the index of the location to push the next item. Hence, when the stack is cleared top will be initialized to 0 instead of to –1. Be sure to modify the ASSERT statements where necessary. Test your program by importing your implementation into a program similar to that in Figure 7.10.

2. Work Problem 1 but test your program by importing your implementation into the program you wrote for Chapter 7, Problem 20.

3. PboxStackObj in Figure 17.6 uses array body and integer top to implement a stack class. At any given point in time, top has the index of the item on top of the stack. Modify PboxStackObj so that at any given point in time top will have the index of the location to push the next item. Hence, when the stack is cleared top will be initialized to 0 instead of to –1. Be sure to modify the ASSERT statements where necessary. Test your program by importing your implementation into a program to construct an RPN calculator as described in Chapter 7, Problem 14. Verify in the calling module that the preconditions are met. If a precondition is not met, the calculator should do nothing and no trap should occur.

4. Work Problem 3, but construct a full-featured scientific calculator as described in Chapter 7, Problem 16.

5. Work Problem 3, but test your program by importing your implementation into a module that implements a dialog box for two stacks with an “A to B” button as described in Chapter 7, Problem 20. Verify in the calling module that the preconditions are met. If a precondition is not met, the dialog box should not change and no trap should occur.