Introduction to Systematic Programming
Unit 18 - Sorting

18.1 More on `FOR` loops

Occasionally it is convenient to use a 'count-down' `FOR` loop in which the control variable is *decreased by one* at the end of each repetition. This kind of Ada `FOR` loop has the general form:

```
FOR Index IN REVERSE Start_expression .. Finish_expression LOOP
     Step(s)_to_be_repeated;
END LOOP;
```

Such a statement means:

i) Evaluate the *`Start_expression`*.
ii) Evaluate the *`Finish_expression`*.
iii) Assign the value of the *`Finish_expression`* to the *`Index`* variable.
iv) Compare the index variable's value with the starting value;
if the *`Index`* variable is less than the value of the *`Start_expression`*, the execution of the `FOR` step is terminated; otherwise (ie. if the *`Index`* variable is greater than or equal to the value of the *`Start_expression`*) the *`Step(s)_to_be_repeated`* part is obeyed, and then 1 is subtracted from the *`Index`* variable[1].
v) The repetition process is then continued (from (iv)).

Note that the above `FOR` repetition includes the keyword `REVERSE` before the index range. Of course, rather than using an explicit index range as shown above, it is also possible to use a named discrete type or subtype as the index range (as was illustrated in [14.5]).

For example, to input a line of text and print it in reverse order we might write:

```
Get_Line(Item => Line, Last => LineLen);
FOR Index IN REVERSE 1 .. LineLen LOOP
   Put(Line(Index));
END LOOP;
```

where the following declarations are assumed:

```
MaxLineLen : CONSTANT Positive := 80;  -- say.
Line : String(1 .. MaxLineLen);
LineLen : Natural;
```

18.2 The 'short-circuit' logical operators `AND THEN` and `OR ELSE`.

Consider the following fragment:

```
IF X >= 0.0 AND SQRT(X) < Y THEN
     Do_something;
END IF;
```

where `X` and `Y` are variables of type `Float`. Presumably the intention of the first comparison is to avoid the run-time error that would occur if there was an attempt to calculate the square root of a negative number. However when `X` has a negative value we would still find that a run-time error would occur because in Ada both operands of the logical operators `AND` (and `OR`) are always evaluated.

We could of course overcome this problem using nested `IF's`:

```
IF X >= 0.0 THEN
     IF SQRT(X) < Y THEN
          Do_something;
     END IF;
END IF;
```

However this is rather clumsy. For these sort of circumstances Ada provides two 'short-circuit' logical operators `AND THEN` and `OR ELSE`. The boolean expression C_1 `AND THEN` C_2 is evaluated as follows:

i) evaluate the condition C_1
ii) if this value is `False`, then the whole expression is `False` (and note that C_2 is not evaluated in this case), otherwise the expression result is the same as the boolean value of condition C_2

[1] If the `Index` variable is of some non-integer discrete type, then the value of `Index` is replaced by its predecessor value after each repetition.

Thus to avoid a possible run-time error we should write the above selection as:

```
IF X >= 0.0 AND THEN SQRT(X) < Y THEN
     Do_something;
END IF;
```

Thus, if X held the value 3.2 (say) then the first comparison X>=0.0 is True, and the value of the whole condition controlling the IF step depends on the value of SQRT(X)<Y. However, if X held the value -4.8 (say), then the first comparison X>=0.0 is False, and the value of the whole condition controlling the IF step is established as False (without the need for the computer to evaluate the second condition SQRT(X)<Y).

Similarly the boolean expression C_1 OR ELSE C_2 is evaluated as follows:

i) evaluate the condition C_1
ii) if this value is True, then the whole expression is True (and note that C_2 is not evaluated in this case), otherwise the expression result is the same as the boolean value of condition C_2

It is easy to check that when the values of both conditions C_1 and C_2 may be evaluated without error, then the short-circuit operators AND THEN and OR ELSE produce the same results as the corresponding standard logical operators AND and OR.

A typical case where short-circuit logical operators may be used to advantage is in comparisons involving array elements where we also need to check that the array index lies in the permitted range:

```
SUBTYPE SomeRange IS Integer RANGE 1000 .. 9999;  -- say.
TYPE SomeArrayType IS ARRAY(SomeRange) OF Float;
A, B : SomeArrayType;
...........
WHILE Index IN SomeRange AND THEN A(Index) < B(Index) LOOP
   Step(s)_to_be_repeated;
END LOOP;
```

Here use of plain AND would run the risk of a Constraint_Error if Index held a value outside the range SomeRange.

A similar situation occurs when using one of the searching methods described in [11.7]. When searching the elements numbered from 1 to NumStudents of an array User to find they contain a particular value (held in the variable Wanted say) we wrote:

```
Position := 1;
Found := False;
WHILE NOT Found AND Position <= NumStudents LOOP
     IF User(Position) = Wanted THEN
          Found := True;
     ELSE
          Position := Position + 1;
     END IF;
END LOOP;
-- After the search, if Found is True then Position holds the
-- subscript of the array element holding the value wanted.
```

ie. on each repetition we need to check to see if the search has not 'run-off' the end of the array before checking whether the element contains the value we are searching for. However, using the AND THEN operator we could write the above more compactly as:

```
Position := 1;
WHILE Position <= NumStudents AND THEN User(Position) /= Wanted LOOP
     Position := Position + 1;
END LOOP;
```

18.3 An introduction to sorting algorithms

Until now we have been mainly concerned with details of the Ada language and with general programming methodology, but the rest of this unit introduces a specific class of computer algorithms, namely **sorting algorithms**, that is algorithms which put items into some defined order. Most commercial applications of computers involve a significant amount of sorting. For example, we might wish to sort an array of records of type OrderList storing information on the orders received by a firm (as considered in Unit 16) into numerical order based upon the part number of the item ordered. The need for sorting also commonly arises in other types of work, such as computer systems programming. Thus a knowledge of sorting techniques is important to most programmers.

Most sorting applications involve ordering collections of records, according to the value of one particular field (as with the `OrderList` records mentioned above), rather than ordering collections of individual values. However the principles involved are readily demonstrated by the latter, simpler problem. The first example considered will involve sorting a collection of `Integer` values stored in an array (such sorts are called **internal sorts**). In [18.6] we shall see how the algorithm developed for this case may easily be adapted to sort arrays with elements of a different type.

Obviously sorting methods need to be correct, but the time taken by sorting schemes tends to rise rapidly with the number of items to be sorted, so it is also important to use methods that are reasonably efficient. In fact, the best sorting techniques known to Computer Science require some advanced programming techniques, and are hence beyond the scope of this course; even so, it is still possible to improve significantly on the most naive methods, that is on methods which might typically be invented by someone who had not studied the area at all.

18.4 Straight Insertion Sort (SIS)

Suppose that the values we want to sort into ascending order are stored in the elements indexed from `1` to `N` of an array. The method is such that, at each stage in the algorithm, as a count `K` advances from `2` to `N`, the elements of the array indexed from `1` to `K-1` have already been put in ascending order.

Each major step of the algorithm is then as follows:

a) Consider the K^{th} element of the array.

b) Determine where in the index range `1` to `K` it should 'fit' in the (already ordered part of the) array, say at position `Pos`.

c) 'Slide' the elements subscripted from `Pos` to `K-1` one place to the 'right', and insert the newly considered value at the `Pos` element so that the elements in the range `1` to `K` are now in order.

d) The elements indexed from `1` to `K` are now in order, so `K` is now increased by `1` and the process is repeated from (a) until all `N` elements have been brought into order.

Notice that this method performs a sort 'in place' without the need for any additional arrays. Also, we have assumed ascending order is required, but note that only trivial changes are needed to produce descending order.

18.4.1 Analysis of the SIS

Let us analyse this algorithm to get some idea of the likely 'cost' of executing it. In order to find the correct position of the K^{th} element, on average about `(K-1)/2` comparisons are required. Hence the total number of comparisons for all the stages is approximately:

$$(1 + 2 + ... + (N-1))/2 = N(N - 1)/4 \approx N^2/4$$

Similarly in order to place the K^{th} element in the correct position, on average about `K/2` assignments are required (as the elements are 'slid' one place to the 'right'). Hence the total number of assignments is approximately:

$$(1 + 2 + ... + N)/2 = N(N + 1)/4 \approx N^2/4$$

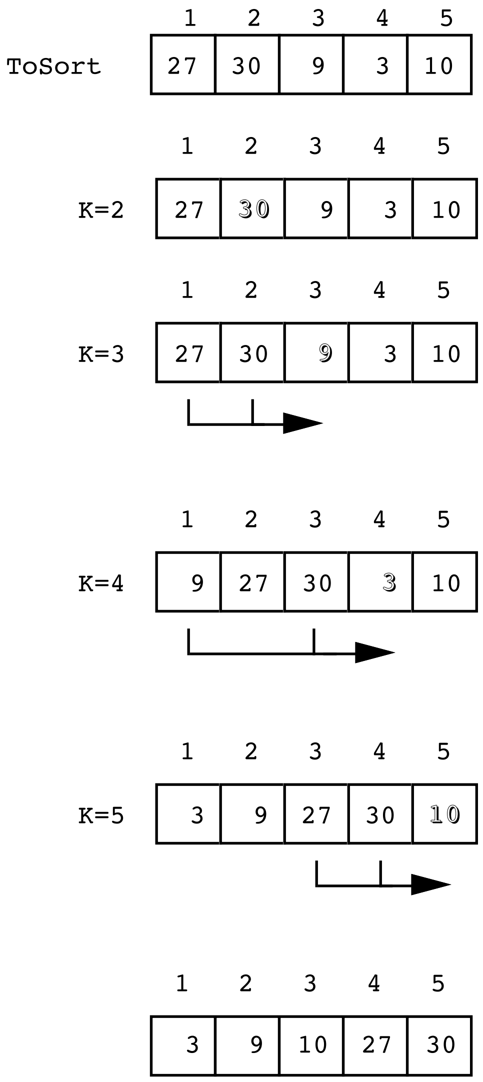
Counting both assignments and comparisons, for large values of `N` the Straight Insertion Sort requires roughly $N^2/2$ operations. Some of you may know about another sorting method called the Straight Exchange Sort or 'Bubble Sort'. This has little to recommend it compared with the SIS, since the average numbers of comparisons and assignments required to sort `N` values using the 'Bubble Sort' are approximately $N^2/2$ and $3N^2/2$ respectively, ie. in total, approximately four times the cost of applying the SIS.

18.4.2 An illustration of the SIS

We assume that the array to be sorted has five elements indexed from 1 to 5, that is we assume that declarations of the following form have been made:

```
N : CONSTANT Positive := 5;
TYPE SmallArray IS ARRAY (1 .. N) OF Integer;
ToSort : SmallArray;
```

Suppose also that the array `ToSort` has had values assigned to its elements as follows:



Elements `1..2` already in order => no change

Element `3` should move to position `1` =>

Slide the elements `1..2` one place to the right,and assign the value `9` to element `1`

Element `4` should move to position `1` =>

Slide the elements `1..3` one place to the right,and assign the value `3` to element `1`

Element `5` should move to position `3` =>

Slide the elements `3..4` one place to the right,and assign the value `10` to element `3`

Sort complete.

18.5 An Ada procedure for SIS

We assume, as above, that we are sorting an array with `Integer` elements. In order to make our sort procedure reasonably flexible we will define an unconstrained array type `IntArray` (say):

```
TYPE IntArray IS ARRAY (Positive RANGE <>) OF Integer;
```

and design our procedure to sort an array of any size of this type into ascending order. Thus our procedure will have the heading:

```
PROCEDURE SISort (ToSort : IN OUT IntArray);
```

In developing the algorithm we will suppose that the upper and lower index bounds of the array are `High` and `Low` respectively (which are assumed to be `Positive` values) instead of the values `1` and `N` in the discussion and illustration above.

18.5.1 Outline algorithm

```
FOR K IN Low+1 .. High LOOP

    Find correct position Pos (amongst the ordered elements
      indexed from Low to K-1) at which the NextValue
      (stored at index K) should be placed ........................ A

    Slide elements with indices Pos to K-1 one place to the right ... B

    IF Pos /= K THEN
      Insert the NextValue at position Pos ........................ C
    END IF

END LOOP
```

18.5.2 Refinements

A Search down through the array from position `K-1` to find the first position `Pos` where `ToSort(Pos) <= NextValue`. However, take care not to run off the lower end of the array.

```
NextValue := ToSort(K);  -- Initialise for 'scan down' through array.
Pos := K;
WHILE Pos > Low AND THEN ToSort(Pos-1) > NextValue LOOP
   Pos := Pos - 1;
END LOOP;
```

Another (slightly less elegant) way to program the 'scan-down' loop is as a `LOOP` with an `EXIT` involving an `OR ELSE` condition:

```
LOOP
   -- Quit when the scan down has reached the end of the array
   -- or when the correct position is found.
   EXIT WHEN Pos = Low OR ELSE ToSort(Pos-1) <= NextValue;
   Pos := Pos - 1;
END LOOP;
```

B One way we can refine this is as a 'count-down' `FOR` loop as discussed in [18.1] above:

```
    FOR I IN REVERSE Pos .. K-1 LOOP
        ToSort(I+1) := ToSort(I);
    END LOOP;
```

Thus in this example the index variable `I` takes in turn (ie. on each repetition) the values of:

```
      K-1, K-2, K-3, ..., Pos+2, Pos+1, Pos
```

Hence the element in position `K-1` is copied to position `K`, the element in position `K-2` is copied to position `K-1` and so on until finally the element in position `Pos` is copied to position `Pos+1`.

However, an alternative (more succinct) way of refining this step would be to use array slices as follows (this works correctly even though the slices overlap!):

```
      ToSort(Pos+1 .. K) := ToSort(Pos .. K-1);
```

Of course, we can't use the following 'normal' `FOR` loop to do the 'sliding' (why?).

```
    FOR I IN Pos .. K-1 LOOP
        ToSort(I+1) := ToSort(I);
    END LOOP;
```

C `ToSort(Pos) := NextValue;`

18.5.3 Full procedure

```
PROCEDURE SISort (ToSort : IN OUT IntArray) IS
   Low       : CONSTANT Positive := ToSort'First;
   High      : CONSTANT Positive := ToSort'Last;
  NextValue : Integer;
  Pos       : Positive;
BEGIN
   FOR K IN Low+1 .. High LOOP
      -- First initialise for 'scan down' through array
      NextValue := ToSort(K);
      Pos := K;
      -- Scan down through array to find the insertion position
      WHILE Pos > Low AND THEN ToSort(Pos-1) > NextValue LOOP
         Pos := Pos - 1;
      END LOOP;
      IF Pos /= K THEN
      -- NextValue is not already in correct place, so 'slide' the
      -- elements one place to the 'right' and insert NextValue.
         ToSort(Pos+1 .. K) := ToSort(Pos .. K-1);
         ToSort(Pos) := NextValue;
      END IF;
   END LOOP;
END SISort;
```

The procedure SISort may be used to sort arrays of any (constrained) subtype of the unconstrained type IntArray no matter what the size. Note that if we want the procedure to sort a portion of an array (rather than the whole of that array), no modification to SISort is necessary, we may simply supply the appropriate slice of the array as the AP when we call SISort.

As it is could be useful in many programs, it would be sensible to put the procedure SISort in a package, SIS_Pack (say). In outline this would have the form:

```
PACKAGE SIS_Pack IS
   TYPE IntArray IS ARRAY (Positive RANGE <>) OF Integer;
   PROCEDURE SISort (ToSort : IN OUT IntArray);
END SIS_Pack;

PACKAGE BODY SIS_Pack IS

    PROCEDURE SISort (ToSort : IN OUT IntArray) IS
    -- As in [18.5.3] above
    ........

    END SISort;

END SIS_Pack;
```

18.6 Sorting arrays of other types

Suppose that we wished to sort an array of string values into lexicographic (dictionary) order, it is easy to modify our procedure SISort to handle this situation. For definiteness suppose we wanted to sort, into alphabetical order, an array containing all the user ID's of computer users in a computer science department.

Given the following declarations:

```
SUBTYPE UnixUserID IS String(1 .. 8);
TYPE IDArray IS ARRAY (Positive RANGE <>) OF UnixUserID;
```

the only changes needed would be to the first two lines of our procedure SISort:

```
PROCEDURE SISort (ToSort : IN OUT IDArray) IS
    NextValue : UnixUserID;
    .....
```

Note that no changes are necessary to the executable steps of the procedure since string values may be compared using the comparison operator > and the assignment steps are valid for string values of the same type and length.

Suppose instead that we wished to sort an array of records into ascending order of some field (Key, say) of the records, it is also easy to modify our procedure SISort to handle this situation. Assuming the following declarations:

```
TYPE RecType IS RECORD
                   Key : Integer;
                   ..............
                END RECORD;
TYPE RecArray IS ARRAY (Positive RANGE <>) OF RecType;
```

only the highlighted modifications in the procedure that follows would be needed (it is not, in fact, necessary that the Key field is of type Integer; in fact it could be of any type for which the comparison operator > is meaningful):

```
PROCEDURE SISort(ToSort : IN OUT RecArray) IS
   NextValue : RecType;
   Pos       : Positive;
   Low       : CONSTANT Positive := ToSort'First;
   High      : CONSTANT Positive := ToSort'Last;
BEGIN
   FOR K IN Low+1 .. High LOOP
      NextValue := ToSort(K);
      Pos := K;
      WHILE Pos > Low
            AND THEN ToSort(Pos-1).Key  >  NextValue.Key LOOP
         Pos := Pos - 1;
      END LOOP;
      IF Pos /= K THEN
         ToSort(Pos+1 .. K) := ToSort(Pos .. K-1);
         ToSort(Pos) := NextValue;
      END IF;
   END LOOP;
END SISort;
```

18.7 Programming example

Each day a small library issues at most 1000 books. The serial number of each book issued is recorded in a file; each serial number on a separate line. It is desired to print out a list of the serial numbers (five to a line) of one day's book issues in ascending order.

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
WITH CS_Int_IO;    USE CS_Int_IO;
WITH CS_File_IO;   USE CS_File_IO;
WITH SIS_Pack;   -- 'Import' Straight Insertion Sort package

PROCEDURE Library IS
   -- Program for Unit 18 of ISP Ada Course.
   -- Written by A Barnes,  November 1993.
   -- Updated for Ada95 by A Barnes, November 1996.

   MaxBooks : CONSTANT Integer := 1000;
   SUBTYPE BookList IS SIS_Pack.IntArray(1 .. MaxBooks);

   Issues : BookList;
   NumBooks : Natural := 0;

BEGIN  -- Main program of Library
   OpenInput(FileName => "issues.dat");
   OpenOutput(FileName => "sorted.txt");
   -- Input the serial numbers from the file.
   WHILE NOT End_Of_File LOOP
      NumBooks := NumBooks + 1;
      Get(Issues(NumBooks));
   END LOOP;

   -- Sort the serial numbers into ascending order.
   SIS_Pack.SISort(ToSort => Issues(1 .. NumBooks));

   -- Output the ordered serial numbers.
   Put_Line("Issued book numbers in ascending order:");
   FOR IssueNo IN 1 .. NumBooks LOOP
      Put(Item => Issues(IssueNo), Width => 10);
      IF IssueNo REM 5 = 0 THEN
         New_Line;
      END IF;
   END LOOP;
   New_Line;
   CloseInput;
   CloseOutput;
END Library;
```

18.8 Another programming example

In the library example above, suppose that it is possible for a reader to return a book so that it may be re-issued during the same day. Again the serial number of each book issued is recorded in a file, but in this case it is possible for the same serial number to appear more than once in the file. If it is desired to print out a list of the unique serial numbers of one day's book issues in ascending order, we must not only find a way of sorting the list of serial numbers into order, but also find some method of removing any duplicates. There are many possible approaches to the solution of this problem, but one way we might consider is to build up the ordered list in the array in a 'piece-wise' manner, ie:

a) Assume that the list stored in the array is initially empty.

b) Take the next serial number from the file.

c) Search the list of serial numbers stored in the array.

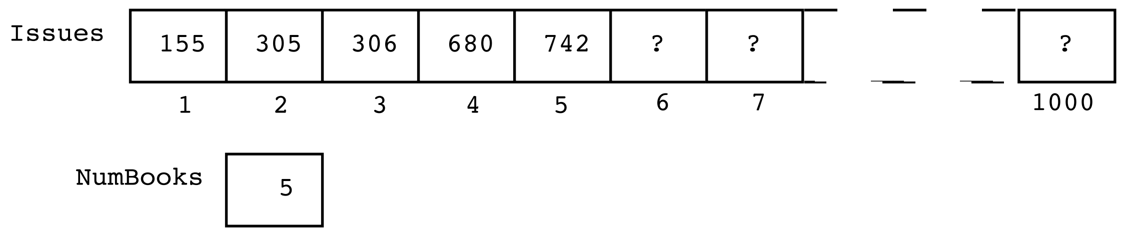
d) If the searched for serial number is not present, then insert it into its 'correct' position in the list stored in the array (ie. so that the list stored in the array is maintained in order).

e) Alternatively, if the searched for serial number is already present, then no actions are necessary since the list stored in the array already contains an entry for that serial number.

f) Continue the above from (b) until there are no more serial numbers in the file.

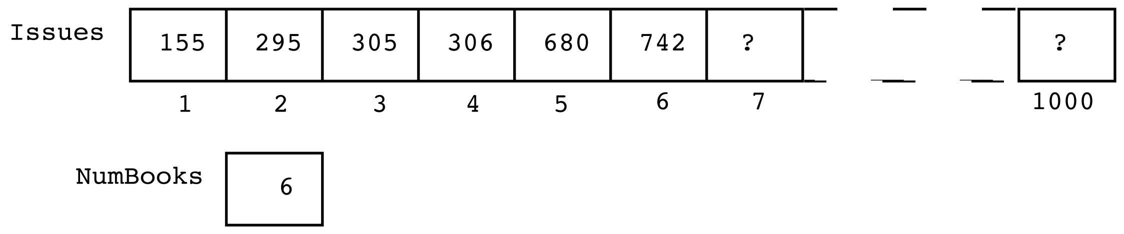
The list of values stored in the array does then not need to be sorted into order. Rather, it has been 'built-up' (serial number by serial number), maintaining the list in order throughout.

For example, suppose we represent the list using `Issues` and `NumBooks` as in the previous solution, and suppose that part way through processing the data file we have input the serial numbers `306`, `155`, `742`, `680` and `305`, and hence 'built-up' the list of values (so far) as:



Suppose that the next serial number in the file was the value `306`. In this case, in step (c), a search through the `Issues` elements indexed from `1` to `NumBooks` would reveal that the serial number `306` was already present, and hence could be ignored.

Alternatively, if the next serial number in the file was the value `295`, a search through the `Issues` elements indexed from `1` to `NumBooks` would reveal that the serial number `295` was not already present, and should be positioned in the element indexed `2`. We could then 'slide' the elements indexed from `2` to `NumBooks` one place to the right to create a free element where `295` could be inserted. The value of `NumBooks` should then be incremented by `1` (to indicate that the size of the list stored in the array has increased by one value). Thus producing the updated ordered list of values:



This process is then continued for each serial number input from the data file.

Note the similarity between the above description with that for the Straight Insertion Sort. In fact the only difference between the processes is that in the example in [18.7] the serial numbers are inserted into their correct order by taking them one-by-one from where they have been stored in the unordered part of the array, whereas in this approach the serial numbers are taken directly from the input data file.

A solution to this problem might thus be as follows:

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
WITH CS_Int_IO;   USE CS_Int_IO;
WITH CS_File_IO;  USE CS_File_IO;

PROCEDURE NewLibrary IS
   -- Second program for Unit 18 of ISP Ada Course.
   -- Written by L Hazlewood, November 1994.
   -- Updated for Ada95 by L J Hazlewood, November 1996.

   MaxBooks : CONSTANT Integer := 1000;
   TYPE BookList IS ARRAY (1..MaxBooks) OF Integer;

   PROCEDURE UpdateList (NewEntry : IN Integer;
                         List : IN OUT BookList;
                         ListSize : IN OUT Natural) IS
      Pos : Natural;
   BEGIN
      -- Find in Pos the position where the NewEntry value is
      -- already stored in the List array, or the position
      -- where the NewEntry value should be inserted in order
      -- for the list to be in order.
      Pos := ListSize + 1;
      WHILE Pos > 1 AND THEN List(Pos-1) > NewEntry LOOP
         Pos := Pos - 1;
      END LOOP;

      IF Pos = 1 OR ELSE List(Pos-1) /= NewEntry THEN
         -- NewEntry should be inserted at position Pos.
         -- So "shuffle-up" list values to create a place for
         -- NewEntry to be inserted in the list.  Note that when
         -- Pos = ListSize+1 (ie. when NewEntry is to be added at
         -- the end of the list) the following slices are "empty",
         -- and no movement of the list values takes place.
         List(Pos+1 .. ListSize+1) := List(Pos .. ListSize);
         List(Pos) := NewEntry;
         ListSize := ListSize + 1;
      -- ELSE NewEntry is already present in the list,
      --      and so can be ignored.
      END IF;
   END UpdateList;

   -- Main program variables.
   Issues : BookList;
   NumBooks : Natural := 0;
   NextBook : Integer;

BEGIN  -- Main program of Library
   OpenInput(FileName => "issues.dat");
   OpenOutput(FileName => "sorted.txt");

   WHILE NOT End_Of_File LOOP
      Get(NextBook);
      UpdateList(NewEntry => NextBook,
                 List => Issues, ListSize => NumBooks);
   END LOOP;

   Put_Line("Issued book numbers in ascending order:");
   FOR  IssueNo IN 1 .. NumBooks LOOP
      Put(Item => Issues(IssueNo), Width => 10);
      IF IssueNo REM 5 = 0 THEN
         New_Line;
      END IF;
   END LOOP;
   New_Line;

   CloseInput;
   CloseOutput;
END NewLibrary;
```