CRD

Corporate Research and Development

Schenectady, New York

ADA* GENERIC LIBRARY
LINEAR DATA STRUCTURE PACKAGES, VOLUME ONE

D.R. Musser† and A.A. Stepanov‡

Information Systems Laboratory

April 1988 **88CRD112**

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)
†Rensselaer Polytechnic Institute, Troy, NY 12180
‡Polytechnic University, Brooklyn, NY 11201

Technical Information Series

Class 1

GENERAL ELECTRIC

CLASSES OF GENERAL ELECTRIC TECHNICAL REPORTS

CLASS 1 -- GENERAL INFORMATION

Available to anyone on request. Patent, legal, and commercial review required before issue.

CLASS 2 -- GENERAL COMPANY INFORMATION

Available to any General Electric Company employee on request. Available to any General Electric Subsidiary or Licensee, subject to existing agreements. Disclosure outside General Electric Company requires approval of originating component.

CLASS 3 -- LIMITED AVAILABILITY INFORMATION

Original distribution to those individuals with specific need for information. Subsequent Company availability requires originating component approval. Disclosure outside General Electric Company requires approval of originating component.

CLASS 4 -- HIGHLY RESTRICTED DISTRIBUTION

Original distribution to those individuals personally responsible for the Company's interests in the subject. Copies serially numbered, assigned, and recorded by name. Material content and knowledge of existence restricted to copy holder.

Requests for Class 2, 3, or 4 reports from non-resident aliens or disclosure of Class 2, 3, or 4 reports to foreign locations, except Canada, require review for export by the CRD Counsel.

Corporate Research and Development

Technical Report Abstract Page

Title	ADA* GENERIC LIBRARY LINEAR DATA STRUCTURE PACKAGES, VOLUME ONE		
Author(s)	D.R. Musser† A.A. Stepanov‡	**Phone**	(518)387-6120 8*833-6120
Component	Information Systems Laboratory		
Report Number	88CRD112	**Date**	April 1988
Number of Pages	150	**Class**	1
Key Words	generic algorithms, generic packages, generic subprograms, list manipulation, software library, software productivity, software reliability, software reuse		

The purpose of the Ada Generic Library is to provide Ada programmers with an extensive, well-structured and well-documented library of generic packages whose use can substantially increase productivity and reliability. The construction of the library follows a new approach, whose principles include the following:

- Extensive use of generic algorithms, such as generic *sort* and *merge* algorithms that can be specialized to work for many different data representations and comparison functions.
- Building up functionally in layers (practicing software reuse within the library itself).
- Obtaining high efficiency in spite of the layering (using Ada's *inline* compiler directive).

Volumes 1 and 2 contain eight Ada packages, with over 170 subprograms, for various linear data structures based on linked lists.

Manuscript received March 30, 1988

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

†Rensselaer Polytechnic Institute, Troy, NY 12180
‡Polytechnic University, Brooklyn, NY 11201

Ada® Generic Library
Linear Data Structure Packages

Volume One

David R. Musser
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Polytechnic University
Computer Science Department
333 Jay Street
Brooklyn, New York 11201

Release 1.1
March 4, 1988

Contents

Chapter 1

Introduction

1.1 Purpose of the library

The purpose of the Ada Generic Library is to provide Ada programmers with an extensive, well-structured and well-documented library of generic packages whose use can substantially increase productivity and reliability. Our main goal in this introduction is to explain both the structure of this particular library and the general principles we have followed in creating that structure. We believe these principles, which are quite different from those on which other libraries such as in [1] have been founded, have broad applicability to the goal of widely-usable software components in Ada.

The first phase of the library concentrates on a significant subset of the data structures problem: an extensive set of *linear data structure* manipulation facilities. The data structures and algorithms included have been selected based on their well-established usefulness in a wide variety of applications. Volumes 1 and 2 include generic Ada packages for Singly_Linked_Lists, Double_Ended_Lists, Stacks, and Output_Restricted_Deques, containing over 170 subprograms and structured to allow plugging together interchangeably with three packages providing different storage allocation strategies.

One note of warning about the current status of these packages, or, more accurately, about the current state of Ada compilers. In the course of developing these packages we had occasion to attempt to compile them with four different Ada compilers: the Alsys compiler for the IBM PC, the Verdix and Telesoft compilers for the SUN workstation, and the DEC Ada compiler for VAX computers. *Only the DEC compiler succeeds in compiling all the packages in this library.* The others cannot handle the heavily layered generics we use in structuring the library. We are working with the vendors of these compilers to make them aware of these problems.

Acknowledgements This initial phase of Ada Generic Library was developed with support from GE Aerospace (for Volume 1) and GE Corporate Research and Development (for Volume 2). Susan Mickel and William Novak of GE Western Systems provided useful suggestions and comments on early versions of Volume 1.

1.2 Principles behind the library

The main principles we have followed in building the library are the following:

1. Extensive use of generic algorithms, such as generic *sort* and *merge* algorithms that can be specialized to work for many different data representations and comparison

functions.

2. Building up functionality in layers, separating, to as large an extent as possible, concerns about representations from those of algorithms.

3. Obtaining high efficiency in spite of the layering (using Ada's *inline* compiler directive).

4. Emphasis on careful selection and expert programming of highly efficient algorithms.

5. High quality documentation that makes it easy to find operations in the library and select the best algorithm and data structure for the application at hand.

The most important technical idea is that of generic algorithms, which are a means of providing functionality in a way that abstracts away from details of representation and basic operations. Instead of referring directly to the host language facilities, generic algorithms are defined in terms a few primitive operations that are considered to be *parameters*. By plugging in actual operations for these parameters, one obtains specific instances of the algorithms for a specific data structure. By carefully choosing the parameterization and the algorithms, one obtains in a small amount of code the capability to produce many different useful operations. It becomes much easier to obtain the operations needed for a particular application by plugging components together than it would be to program them directly.

1.3 Related technology

The notion of generic algorithms is not entirely new, but there has not been any attempt to structure a general software library founded on this idea. Older program libraries, written in Fortran or other languages without the facilities for generic programming, could not take advantage of the algorithm abstractions that were known. But even the recent improvements in abstraction facilities in contemporary programming languages, such as Ada, have not precipitated widespread use of algorithmic abstraction. (Booch, for example, makes some use of generic algorithms for list and tree structures, but almost as an afterthought in a chapter on utilities.) For the benefits of this approach to be fully realized, great care must be exercised in selecting and structuring algorithms, especially in determining how they are parameterized and how they are used to develop more concrete levels of the library. Indeed, we view algorithm selection, abstraction, and structuring as being of far greater importance to software reusability than any language or other human-interface issues; experience with Unix tools provides ample evidence of this point.

1.4 Structure of the library

The key structuring mechanism used in building the library is *abstraction*. We discuss four classes of abstractions that we have found useful in structuring the library, as shown in Table 1.1, which lists a few examples of packages in the library. Each of these Ada packages has been written to provide generic algorithms and generic data structures that fall into the corresponding abstraction class. (The packages marked with a * are not included in this release of the library.) These classes are defined as follows:

Data Abstractions Data types with operations defined on them	System_Allocated_Singly_Linked User_Allocated_Singly_Linked {Instantiations of representational abstractions}
Algorithmic Abstractions Families of data abstractions with common algorithms	Sequence_Algorithms* Linked_List_Algorithms Vector_Algorithms
Structural Abstractions Intersections of algorithmic abstractions	Singly_Linked_Lists Doubly_Linked_Lists* Vectors*
Representational Abstractions Mappings from one structural abstraction to another	Double_Ended_Lists Stacks Output_Restricted_Deques

Table 1.1: Classification of Abstractions and Example Ada Packages

1.4.1 Data abstractions

Data abstractions are data types and sets of operations defined on them (the usual definition); they are abstractions mainly in that they can be understood (and formally specified by such techniques as algebraic axioms) independently of their actual implementation. In Ada, data abstractions can be written as packages which define a new type and procedures and functions on that type. Another degree of abstractness is achieved by using a generic package in which the type of elements being stored is a generic formal parameter. In our library, we program only a few such data abstractions directly—those necessary to create some fundamental data representations and define how they are implemented in terms of Ada types such as arrays, records and access types. Three such packages, which we refer to as "low-level data abstraction packages," are presented in Chapters 3, 4, and 5. Most other data abstractions are obtained by combining existing data abstraction packages with packages from the structural or representational classes defined below.

1.4.2 Algorithmic abstractions

These are families of data abstractions that have a set of efficient algorithms in common; we refer to the algorithms themselves as *generic algorithms*. For example, in our library there is a package of generic algorithms for linked-lists; in a future release there will be a more general package of sequence algorithms whose members can be used on either linked-list or vector representations of sequences. The linked-list generic algorithms package contains 31 different algorithms such as, for example, generic merge and sort algorithms that are instantiated in various ways to produce merge and sort subprograms in structural abstraction packages such as singly-linked lists and doubly-linked lists.

We stress that the algorithms at this level are derived by abstraction from concrete, efficient algorithms. As an example of algorithmic abstraction, consider the task of choosing and implementing a sorting algorithm for linked list data structures. The merge sort algorithm can be used and, if properly implemented, provides one of the most efficient sorting algorithms for linked lists. Ordinarily one might program this algorithm directly in terms of whatever pointer and record field access operations are provided in the programming language. Instead, however, one can abstract away a concrete representation and express the algorithm in terms of the smallest possible number of generic operations. In this case, we essentially need just three operations: **Next** and **Set_Next** for accessing the next cell in a list, and **Is_End** for detecting the end of a list. For a particular representation of linked lists, one then obtains the corresponding version of a merge sorting algorithm by instantiating the generic access operations to be subprograms that access that representation.

Thus in Ada one programs generic algorithms in a generic package whose parameters are a small number of types and access operations—e. g.,

```
generic
  type Cell is private;
  with function Next(S : Cell) return Cell;
  with procedure Set_Next(S1, S2 : Cell);
  with function Is_End(S : Cell) return Boolean;
  with function Copy_Cell(S1, S2 : Cell) return Cell;
package Linked_List_Algorithms is
  . . .
```

The subprograms in the package are algorithms such as **Merge** and **Sort** that are efficient when **Next**, **Set_Next**, etc., are instantiated with constant time operations.

1.4.3 Structural abstractions

Structural abstractions (with respect to a given set of algorithmic abstractions) are also families of data abstractions: a data abstraction A belongs to a structural abstraction S if and only if S is an intersection of some of the algorithmic abstractions to which A belongs. An example is singly-linked-lists, the intersection of sequence- , linked-list-, and singly-linkedlistalgorithmic abstractions. It is a family of all data abstractions that implement a singly-linked representation of sequences (it is this connection with more detailed structure of representations that inspires the name "structural abstraction"). (In this release, the **Singly_Linked_Lists** package (Chapter 6) is actually programmed just in terms of the **Linked_List_Algorithms** package.)

Note that, as an intersection of algorithmic abstractions, such a family of data abstractions is smaller than the algorithm abstraction classes in which it is contained, but a *larger* number of algorithms are possible, because the structure on which they operate is more completely defined.

Programming of structural abstractions can be accomplished in Ada with the same kind of generic package structure as for generic algorithms. The **Singly_Linked_Lists** package contains 66 subprograms, most of which are obtained by instantiating or calling in various ways some member of the **Linked_List_Algorithms** package. In Ada, to actually place one data abstraction in the singly-linked-lists family, one instantiates the **Singly_Linked_Lists** package, using as actual parameters a type and the set of operations on this type from a data abstraction package such as **System_Allocated_Singly_Linked** that defines an appropriate representation.

1.4.4 Representational abstractions

These are mappings from one structural abstraction to another, creating a new type and implementing a set of operations on that type by means of the operations of the domain structural abstraction. For example, stacks can easily be obtained as a structural abstraction from singly-linked-lists, and this is carried out in Ada using generic packages in a manner that will be demonstrated in Volume 2. Note that what one obtains is really a family of stack data abstractions, whereas the usual programming techniques give only a single data abstraction.

1.5 Selection from the library

The first observation we would make is that proper classification of software components for maximum usability may well depend more on *internal structure* than on functional (input-output) behavior. In searching the library, the programmer needs to know not only whether there is a subprogram that performs the right operation, but also what kind of data representation it uses (if it is not a completely generic algorithm), since in all but the simplest cases it will be used in a particular context that may strongly favor one representation over another.

Experienced programmers will sometimes want to use generic algorithms directly, instantiating the generic access operations to be subprograms accessing a particular data representation. Although generic, these algorithms are tailored to be used with data representations with particular complexity characteristics, such as linked-list- versus array-like representations, and the programmer must be aware of these issues.

This is not to say that intelligent use of the library necessarily requires the programmer to examine the bodies of the subprograms. If construction of the library is, as we have recommended, algorithmically-driven and draws upon the best books and articles on algorithms and data structures, then it should be possible to develop sufficiently precise and complete *selection criteria* based on the advice in those books and articles. Again, the preparation of these selection criteria and other documentation must be done very carefully and thoroughly to make later usage by programmers as simple as possible. (The selection criteria contained in this release of the library are mainly for choosing between different subprograms within a package; criteria for choosing between different packages will be supplied in a later release.)

1.6 Using the library

The packages in the Ada Generic Library are intended to be included in a local site's Ada library structure (using the library mechanism supported by the Ada system in use locally), so that a programmer can use them simply by including appropriate `with` statements in his or her source code. In most cases the programmer will not use packages from the four abstraction classes directly; instead it is simpler to use what we call Partially Instantiated Packages, or PIPs. Each PIP effectively "plugs together" a low-level data abstraction package with a structural or representational package, presenting a generic package interface in which the only generic parameters are the element type and perhaps some size or other control parameters. In this release of the library there are twelve PIPs provided, one for each combination of one of the three low-level data abstraction packages in Chapters 3, 4, and 5 with the Singly_Linked_Lists package in Chapter 6 or one of the three packages in Volume 2. PIPs are discussed further in Chapter 7.

Bibliography

[1] G. Booch, G., *Software Components in Ada.* Benjamin/Cummings, 1987.

[2] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

[3] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1968.

[4] R. Sedgewick, *Algorithms*, Addison-Wesley, 1983.

[5] G. Steele, *Common Lisp: The Language*, Digital Press, 1984.

[6] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

Chapter 2

Linear Data Structures

2.1 Sequences

The first phase of the Ada Generic Library, Linear Data Structures, can be described in terms of the different data structures that are implemented, most of which are relatively simple and familiar structures such as linked lists, vectors (one dimensional arrays), stacks, queues, deques, etc. However, a highly unifying way to organize one's understanding of these structures and the algorithms associated with them is in terms of the mathematical notion of (finite) *sequences.* (In a later release, we will include a `Sequences` package of generic algorithms, but for now we discuss sequences just as a way of understanding many aspects of linked-list and vector representations.)

For a given data type T, the set of all sequences

$$x_0, x_1, \ldots, x_{n-1}$$

for all integers $n \geq 0$, where each x_i is a member of type T, is called the set (or type) of sequences of T. If $n = 0$, we have the unique *empty sequence* of T. The number of elements, n, in a sequence is called the *length* of the sequence. The index i of an element x_i within a sequence is also called a *position* in the sequence.

As mathematical objects, sequences are not of great interest (at least not the finite variety we are discussing here), but their computational use introduces a great many interesting and sometimes complex issues. The issue of insertion or deletion of elements in a sequence comes immediately to mind; the need to frequently insert or delete elements somewhere in the middle of a sequence favors a linked list representation; whereas the need to access elements in random positions, as opposed to consecutive positions, favors a vector representation.

Another discriminator between linked representations and vector representations is whether it is possible to assume a fixed upper limit on the length of sequences, in which case we refer to them as *bounded sequences.* Bounded sequences allow vector representations, whereas unbounded sequences are generally implemented as linked lists. (However, a less well-known representation called "extensible vectors" can also be used for unbounded sequences, as will be discussed in a later volume.)

We will not attempt to give a complete discussion of the tradeoffs between various linear data structures or to justify all of the assertions made in this overview or in the descriptions of the packages and subprograms given in later chapters. We have, however, tried to remain consistent with widely used terminology and notation, so that the reader can use textbooks on data structures such as [3], [4], as sources of reference in conjunction with these packages.

In the remainder of this section, we give some additional terminology for sequences that will be used in the subprogram descriptions. For a sequence S of length n, say

$$x_0, x_1, \ldots, x_{n-1}$$

we refer to x_0 as the *first* element (not the zeroth) and x_{n-1} as the *last* element.

We also commonly refer to x_0 as the left end and x_{n-1} as the right end of the sequence. Thus, if there are one or more elements $x_i, x_j, \ldots$ equal to some element x, then we refer to the element in the sequence with smallest index as the *left-most occurrence* of x in S.

In this discussion of sequences, the indices, or positions, of elements play a major role, but computationally this is not necessarily the case. When using a linked list representation, it is best to de-emphasize the calculation and use of numerical positions in favor of operations that move through sequences element by element.

2.2 Organization

2.2.1 Low-level data abstractions

In this release we have provided three different low-level data abstractions using singly-linked list representations:

- The `System_Allocated_Singly_Linked` package provides records containing datum and link fields, allocated using the standard heap allocation and deallocation procedures.
- `Once_User_Allocated_Singly_Linked` provides more efficient allocation and deallocation by allocating an array of records as a storage pool, but is less flexible than the system allocated package since this array and the system heap are managed separately.
- `Auto_Reallocating_Singly_Linked` also uses an array of records for efficiency but automatically allocates a larger array whenever necessary; its disadvantage is that the parameters controlling the reallocation may need to be tuned to achieve optimum reallocation behavior.

These data abstractions are described in Chapters 3, 4, and 5.

2.2.2 Algorithmic, structural and representational abstractions

This release of the library provides the following algorithmic, structural and representational abstraction packages:

- `Singly_Linked_Lists` is a structural abstraction package that provides over 60 subprograms for operations on a singly-linked list representation, including numerous kinds of concatenation, deletion, substitution, searching and sorting operations.
- `Linked_List_Algorithms` is a generic algorithms package that is the source of most of the algorithms used in `Singly_Linked_Lists`; many of the algorithms will also be used in implementing the `Doubly_Linked_Lists` package.
- `Double_Ended_Lists` (described in Volume 2) employs header cells with singly-linked lists to make some operations such as concatenation more efficient and to provide more security in various computations with lists.

- `Stacks` (Volume 2) provides the familiar linear data structure in which insertions and deletions are restricted to one end.

- `Output_Restricted_Deques` (Volume 2) provides a data structure that restricts insertions to both ends and deletions to one end.

The latter three packages are representational abstractions that produce different structural abstractions from different representations of singly-linked lists. Any of the four structural or representational abstraction packages can be plugged together with any of the three low-level data abstraction packages provided, for a total of 12 different possible combinations. Each of these 12 combinations, called a *Partially Instantiated Package*, or *PIP* for short, is included in the library. To use them one only has to instantiate the element type to a specific type. See Chapter 8 for further details on the form and usage of the PIPs.

A later release will also include:

- `Sequences`

- `Doubly_Linked_Lists`

- `Simple_Vectors`

- `Extensible_Vectors`

packages, along with several low-level data abstraction packages that plug together with them.

2.3 Selection from the library

There are, at a minimum, three kinds of selections to be made in using these packages:

1. the choice of a low-level data abstraction package

2. the choice of a structural or representational abstraction package

3. the choice of operations within the structural or representational package

The fact that the structure of our library allows separate choices for 1 and 2 means that there are many more selections available than would be the case with more conventional organizations. However, it is not the case that these choices are entirely independent of each other or of the choices in 3. In fact, the programmer will often have to give careful consideration to the the combination of operations that he or she expects to use in an application, and make a package selection based on algorithmic issues of time and space efficiency of the subprograms as documented in the subprogram descriptions. Another issue that might dictate a choice would be the possible exceptions raised by the operations to be used.

Chapter 3

System_Allocated_Singly_Linked Package

3.1 Overview

This is the simplest of the three low-level data abstraction packages provided in this release. It provides records containing datum and link fields, allocated using the standard heap allocation and deallocation procedures.

The exceptions that are raised by the subprograms in this package (and the other two low-level representation packages) are renamings of those defined in the package `Linked_Exceptions`, (which contains nothing but exception specifications). `Linked_Exceptions` is used in a context clause of the the low-level representation packages and the data abstraction packages with which they might be plugged together, so that both packages are referring to the same set of exceptions; renamings are done to make the exceptions visible outside. (The way that exceptions are set up in these packages may be revised in a future release. Under consideration is the possibility of eliminating the exceptions entirely, allowing system exceptions such as `Contraint_Error` and `Storage_Error` to surface from the packages.)

3.2 Package specification

The package specification is as follows:

```
with Linked_Exceptions;
generic

  type Element is private;

package System_Allocated_Singly_Linked is

  type Sequence is private;

  Nil : constant Sequence;

  First_Of_Nil : exception
      renames Linked_Exceptions.First_Of_Nil;
```

```
  Set_First_Of_Nil : exception
      renames Linked_Exceptions.Set_First_Of_Nil;
  Next_Of_Nil : exception
      renames Linked_Exceptions.Next_Of_Nil;
  Set_Next_Of_Nil : exception
      renames Linked_Exceptions.Set_Next_Of_Nil;
  Out_Of_Construct_Storage : exception
      renames Linked_Exceptions.Out_Of_Construct_Storage;


 {The subprogram specifications}

private

  type Node;

  type Sequence is access Node;

  Nil                      : constant Sequence := null;

end System_Allocated_Singly_Linked;
```

3.3 Package body

The package body is as follows:

```
with Unchecked_Deallocation;
package body System_Allocated_Singly_Linked is

  type Node is record
    Datum : Element;
    Link  : Sequence;
  end record;

  procedure Free_Aux is new Unchecked_Deallocation(Node, Sequence);


 {The subprogram bodies}

end System_Allocated_Singly_Linked;
```

3.4 Subprograms

3.4.1 Construct

Specification

```
function Construct(The_Element : Element; S : Sequence)
      return Sequence;
pragma inline(Construct);
```

Description Returns the sequence whose first element is The_Element and whose following elements are those of S. S is shared.

Time constant

Space constant

Mutative? No

Shares? Yes

Details May raise an exception, Out_Of_Construct_Storage. The relations

$$\text{First}(\text{Construct}(E,S)) = E$$
$$\text{Next}(\text{Construct}(E,S)) = S$$

always hold unless an exception is raised.

See also First, Next, Set_First, Set_Next

Implementation

```
begin
  return new Node'(The_Element, S);
exception
  when Storage_Error =>
    raise Out_Of_Construct_Storage;
end Construct;
```

3.4.2 First

Specification

```
function First(S : Sequence)
       return Element;
pragma inline(First);
```

Description Returns the first element of S

Time constant

Space 0

Mutative? No

Shares? No

Details Raises an exception, First_Of_Nil, if S = Nil.

See also Set_First, Next

Implementation

```
begin
  return S.Datum;
exception
  when Constraint_Error =>
    raise First_Of_Nil;
end First;
```

3.4.3 Free

Specification

```
procedure Free(S : Sequence);
pragma inline(Free);
```

Description Causes the first cell of S to be made available for reuse. S is destroyed.

Time constant

Space 0 (makes space available)

where $n = \text{length}(S)$

Mutative? Yes

Shares? No

See also

Implementation

```
  Temp : Sequence := S;
begin
  Free_Aux(Temp);
end Free;
```

3.4.4 Next

Specification

```
function Next(S : Sequence)
       return Sequence;
pragma inline(Next);
```

Description Returns the sequence consisting of all the elements of S, except the first. S is shared.

Time constant

Space 0

Mutative? No

Shares? Yes

Details Raises an exception, Next_Of_Nil, if S is Nil.

See also First, Set_Next

Implementation

```
begin
  return S.Link;
exception
  when Constraint_Error =>
    raise Next_Of_Nil;
end Next;
```

3.4.5 Set_First

Specification

```
procedure Set_First(S : Sequence; X : Element);
pragma inline(Set_First);
```

Description Changes S so that its first element is X but the following elements are unchanged.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Raises an exception, Set_First_Of_Nil, if S is Nil.

See also First, Set_Next

Implementation

```
begin
  S.Datum := X;
exception
  when Constraint_Error =>
    raise Set_First_Of_Nil;
end Set_First;
```

3.4.6 Set_Next

Specification

```
procedure Set_Next(S1, S2 : Sequence);
pragma inline(Set_Next);
```

Description Changes S1 so that its first element is unchanged but the following elements are those of S2. S2 is shared.

Time constant

Space 0

Mutative? Yes

Shares? Yes

Details Raises an exception, Set_Next_Of_Nil, if S1 is Nil.

See also Next, Set_First

Implementation

```
begin
  S1.Link := S2;
exception
  when Constraint_Error =>
    raise Set_Next_Of_Nil;
end Set_Next;
```

Chapter 4

User_Allocated_Singly_Linked Package

4.1 Overview

Compared to the System_Allocated_Singly_Linked low-level data abstraction, this package provides more efficient allocation and deallocation of list nodes by allocating an array of records as a storage pool. This however makes it less flexible than the system allocated package since the array and the system heap are managed separately, producing a greater possibility of running out of storage.

See the discussion of exceptions in Section 3.1, which applies here also.

The subprogram descriptions are identical to those for System_Allocated_Singly_Linked in all respects except the implementations.

4.2 Package specification

The package specification is as follows:

```
with Linked_Exceptions;
generic
  Heap_Size : in Natural;
  type Element is private;

package User_Allocated_Singly_Linked is

  type Sequence is private;

  Nil : constant Sequence;

  First_Of_Nil : exception
      renames Linked_Exceptions.First_Of_Nil;
  Set_First_Of_Nil : exception
      renames Linked_Exceptions.Set_First_Of_Nil;
  Next_Of_Nil : exception
      renames Linked_Exceptions.Next_Of_Nil;
```

```
  Set_Next_Of_Nil : exception
      renames Linked_Exceptions.Set_Next_Of_Nil;
  Out_Of_Construct_Storage : exception
      renames Linked_Exceptions.Out_Of_Construct_Storage;


 {The subprogram specifications}

private

  type Sequence is new Natural;

  Nil                          : constant Sequence := 0;

end User_Allocated_Singly_Linked;
```

4.3 Package body

The package body is as follows:

```
package body User_Allocated_Singly_Linked is

  type Node is record
    Datum : Element;
    Link  : Sequence;
  end record;

  type Heap_Of_Records is array(Sequence range <>) of Node;

  Heap          : Heap_Of_Records(1 .. Sequence(Heap_Size));

  Free_List     : Sequence := Nil;

  Fill_Pointer : Sequence := 1;


 {The subprogram bodies}

end User_Allocated_Singly_Linked;
```

4.4 Subprograms

4.4.1 Construct

Specification

```
function Construct(The_Element : Element; S : Sequence)
      return Sequence;
pragma inline(Construct);
```

Description Returns the sequence whose first element is The_Element and whose following elements are those of S. S is shared.

Time constant

Space constant

Mutative? No

Shares? Yes

Details May raise an exception, Out_Of_Construct_Storage. The relations

First(Construct(E,S)) = E
Next(Construct(E,S)) = S

always hold unless an exception is raised.

See also First, Next, Set_First, Set_Next

Implementation

```
  Temp : Sequence;
begin
  if Free_List /= Nil then
    Temp := Free_List;
    Free_List := Next(Free_List);
  elsif Fill_Pointer <= Sequence(Heap_Size) then
    Temp := Fill_Pointer;
    Fill_Pointer := Fill_Pointer + 1;
  else
    raise Out_Of_Construct_Storage;
  end if;
  Set_First(Temp, The_Element);
  Set_Next(Temp, S);
  return (Temp);
end Construct;
```

4.4.2 First

Specification

```
function First(S : Sequence)
      return Element;
pragma inline(First);
```

Description Returns the first element of S

Time constant

Space 0

Mutative? No

Shares? No

Details Raises an exception, First_Of_Nil, if S = Nil.

See also Set_First, Next

Implementation

```
begin
  return Heap(S).Datum;
exception
  when Constraint_Error =>
    raise First_Of_Nil;
end First;
```

4.4.3 Free

Specification

```
procedure Free(S : Sequence);
pragma inline(Free);
```

Description Causes the first cell of S to be made available for reuse. S is destroyed.

Time constant

Space 0 (makes space available)

where $n = \text{length}(S)$

Mutative? Yes

Shares? No

See also

Implementation

```
begin
  Set_Next(S, Free_List);
  Free_List := S;
end Free;
```

4.4.4 Next

Specification

```
function Next(S : Sequence)
       return Sequence;
pragma inline(Next);
```

Description Returns the sequence consisting of all the elements of S, except the first. S is shared.

Time constant

Space 0

Mutative? No

Shares? Yes

Details Raises an exception, Next_Of_Nil, if S is Nil.

See also First, Set_Next

Implementation

```
begin
  return Heap(S).Link;
exception
  when Constraint_Error =>
    raise Next_Of_Nil;
end Next;
```

4.4.5 Set_First

Specification

```
procedure Set_First(S : Sequence; X : Element);
pragma inline(Set_First);
```

Description Changes S so that its first element is X but the following elements are unchanged.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Raises an exception, Set_First_Of_Nil, if S is Nil.

See also First, Set_Next

Implementation

```
begin
  Heap(S).Datum := X;
exception
  when Constraint_Error =>
    raise Set_First_Of_Nil;
end Set_First;
```

4.4.6 Set_Next

Specification

```
procedure Set_Next(S1, S2 : Sequence);
pragma inline(Set_Next);
```

Description Changes S1 so that its first element is unchanged but the following elements are those of S2. S2 is shared.

Time constant

Space 0

Mutative? Yes

Shares? Yes

Details Raises an exception, Set_Next_Of_Nil, if S1 is Nil.

See also Next, Set_First

Implementation

```
begin
  Heap(S1).Link := S2;
exception
  when Constraint_Error =>
    raise Set_Next_Of_Nil;
end Set_Next;
```

Chapter 5

Auto_Reallocating_Singly_Linked Package

5.1 Overview

Compared to the `System_Allocated_Singly_Linked` low-level data abstraction, this package provides more efficient allocation and deallocation of list nodes by allocating an array of records as a storage pool. It is also more flexible than the `User_Allocated_Singly_Linked` data abstraction, since it automatically reallocates a larger array whenever necessary. A disadvantage is that it may be necessary to tune the parameters controlling the reallocation based on characteristics of a particular application.

See the discussion of exceptions in Section 3.1, which applies here also.

The subprogram descriptions are identical to those for `System_Allocated_Singly_Linked` in all respects except the implementations.

5.2 Package specification

The package specification is as follows:

```
with Linked_Exceptions;
generic
  Initial_Number_Of_Blocks : in Positive;
  Block_Size               : in Positive;
  Coefficient              : in Float;
  type Element is private;

package Auto_Reallocating_Singly_Linked is

  type Sequence is private;

  Nil : constant Sequence;

  First_Of_Nil : exception
      renames Linked_Exceptions.First_Of_Nil;
  Set_First_Of_Nil : exception
```

```
      renames Linked_Exceptions.Set_First_Of_Nil;
  Next_Of_Nil : exception
      renames Linked_Exceptions.Next_Of_Nil;
  Set_Next_Of_Nil : exception
      renames Linked_Exceptions.Set_Next_Of_Nil;
  Out_Of_Construct_Storage : exception
      renames Linked_Exceptions.Out_Of_Construct_Storage;


 {The subprogram specifications}

private

  type Sequence is new Natural;

  Nil                       : constant Sequence := 0;

end Auto_Reallocating_Singly_Linked;
```

5.3 Package body

The package body is as follows:

```
with Unchecked_Deallocation;
package body Auto_Reallocating_Singly_Linked is

  Number_Of_Blocks : Positive := Initial_Number_Of_Blocks;

  Heap_Size        : Sequence := Sequence(Number_Of_Blocks * Block_Size);

  type Node is record
    Datum : Element;
    Link  : Sequence;
  end record;

  type Vector_Of_Nodes is array(Sequence range <>) of Node;

  type Heap_Of_Nodes   is access Vector_Of_Nodes;

  procedure Free_Heap is new Unchecked_Deallocation(Vector_Of_Nodes,
                                                    Heap_Of_Nodes);

  Heap          : Heap_Of_Nodes;

  Free_List     : Sequence := Nil;

  Fill_Pointer : Sequence := 1;
```

```
  procedure Reallocate is
    New_Number_Of_Blocks : Natural        :=
       Positive(Float(Number_Of_Blocks) * Coefficient + 0.5);
    New_Heap_Size        : Sequence       :=
       Sequence(New_Number_Of_Blocks * Block_Size);
    New_Heap             : Heap_Of_Nodes :=
     new Vector_Of_Nodes(1 .. New_Heap_Size);
  begin
    for I in Heap'range loop
      New_Heap(I) := Heap(I);
    end loop;
    Free_Heap(Heap);
    Heap := New_Heap;
    Number_Of_Blocks := New_Number_Of_Blocks;
    Heap_Size := New_Heap_Size;
  end Reallocate;


 {The subprogram bodies}

begin

  Heap := new Vector_Of_Nodes(1 .. Heap_Size);

exception

  when Storage_Error =>
    raise Out_Of_Construct_Storage;

end Auto_Reallocating_Singly_Linked;
```

5.4 Subprograms

5.4.1 Construct

Specification

```
function Construct(The_Element : Element; S : Sequence)
      return Sequence;
pragma inline(Construct);
```

Description Returns the sequence whose first element is The_Element and whose following elements are those of S. S is shared.

Time constant except when reallocation is necessary

Space constant

Mutative? No

Shares? Yes

Details May raise an exception, Out_Of_Construct_Storage. The relations

$$\begin{aligned} \text{First}(\text{Construct}(E,S)) &= E \\ \text{Next}(\text{Construct}(E,S)) &= S \end{aligned}$$

always hold unless an exception is raised.

See also First, Next, Set_First, Set_Next

Implementation

```
  Temp : Sequence;
begin
  if Free_List /= Nil then
    Temp := Free_List;
    Free_List := Next(Free_List);
  else
    if Fill_Pointer > Sequence(Heap_Size) then
      Reallocate;
    end if;
    Temp := Fill_Pointer;
    Fill_Pointer := Fill_Pointer + 1;
  end if;
  Set_First(Temp, The_Element);
  Set_Next(Temp, S);
  return (Temp);
end Construct;
```

5.4.2 First

Specification

```
function First(S : Sequence)
      return Element;
pragma inline(First);
```

Description Returns the first element of S

Time constant

Space 0

Mutative? No

Shares? No

Details Raises an exception, First_Of_Nil, if S = Nil.

See also Set_First, Next

Implementation

```
begin
  return Heap(S).Datum;
exception
  when Constraint_Error =>
    raise First_Of_Nil;
end First;
```

5.4.3 Free

Specification

```
procedure Free(S : Sequence);
pragma inline(Free);
```

Description Causes the first cell of S to be made available for reuse. S is destroyed.

Time constant

Space 0 (makes space available)

where n = length(S)

Mutative? Yes

Shares? No

See also

Implementation

```
begin
  Set_Next(S, Free_List);
  Free_List := S;
end Free;
```

5.4.4 Next

Specification

```
function Next(S : Sequence)
       return Sequence;
pragma inline(Next);
```

Description Returns the sequence consisting of all the elements of S, except the first. S is shared.

Time constant

Space 0

Mutative? No

Shares? Yes

Details Raises an exception, Next_Of_Nil, if S is Nil.

See also First, Set_Next

Implementation

```
begin
  return Heap(S).Link;
exception
  when Constraint_Error =>
    raise Next_Of_Nil;
end Next;
```

5.4.5 Set_First

Specification

```
procedure Set_First(S : Sequence; X : Element);
pragma inline(Set_First);
```

Description Changes S so that its first element is X but the following elements are unchanged.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Raises an exception, Set_First_Of_Nil, if S is Nil.

See also First, Set_Next

Implementation

```
begin
  Heap(S).Datum := X;
exception
  when Constraint_Error =>
    raise Set_First_Of_Nil;
end Set_First;
```

5.4.6 Set_Next

Specification

```
procedure Set_Next(S1, S2 : Sequence);
pragma inline(Set_Next);
```

Description Changes S1 so that its first element is unchanged but the following elements are those of S2. S2 is shared.

Time constant

Space 0

Mutative? Yes

Shares? Yes

Details Raises an exception, Set_Next_Of_Nil, if S1 is Nil.

See also Next, Set_First

Implementation

```
begin
  Heap(S1).Link := S2;
exception
  when Constraint_Error =>
    raise Set_Next_Of_Nil;
end Set_Next;
```

Chapter 6

Singly_Linked_Lists Package

6.1 Overview

This package provides 66 subprograms (including those that are generic formal parameters) for manipulating a singly-linked-list representation of sequences, in which the elements are of any type (supplied by a generic parameter). The purposes of the these subprograms may be classified into the following three categories:

1. Construction and modification of sequences
2. Examining sequences
3. Computing with sequences

In this section we give a brief overview of these categories, leaving the details and examples of usage to the individual subprogram descriptions.

The selection of operations in this package and many details of their behavior were inspired by the sequence and list operations defined for the Common Lisp language in [5].

6.1.1 Construction and modification of sequences

Basic construction

The most basic operation is `Construct`, which is actually a generic formal parameter to the package and is therefore supplied by another package (such as `System_Allocated_Singly_Linked`). It is assumed that `Construct` takes an element `E` and a sequence `S` and produces a new sequence whose elements are `E` followed by all the elements of `S`. By using the constant `Nil`, which is also a generic formal and represents the empty sequence, and calls to `Construct`, one can obtain particular sequences; e.g., assuming the element type is `Integer`, the expression

```
Construct(1,Construct(3,Construct(5,Nil)))
```

produces a sequence of the first three odd numbers.

The `Make_Sequence` function, given an integer `N` and an element `E`, produces a sequence of `N` elements all equal to `E`.

`Copy_Sequence(S)` returns a sequence containing the same elements as `S`, but using new cells. `Copy_First_N(S,N)` produces a sequence consisting of the first `N` elements of `S`, using new cells.

Basic modification

All of the subprograms for basic modification of sequences are procedures. `Set_First(S,E)` changes `S` so that its first element is `E` but the following elements are unchanged. Similarly, `Set_Next(S1,S2)` changes `S1` so that it retains its first element but the following elements are all the elements of `S2`. `S2` is unchanged, but the issue of argument *sharing* comes into play here. `S2` is shared in the sense that the cells making it up are used also in the representation of `S1`. Thus if `S2` is referred to later, one must remember that any change to `S1` may also change `S2`, and vice versa.

`Set_Nth(S,N,E)` is a more general version of `Set_First` allowing change of an element in an arbitrary position. Note however that its execution time is a linear function of `N`, rather than constant as in the case of vector accesses. Linked list representations are most appropriate when the computation can be arranged so that operations like `Set_Nth(S,N,E)` that reference arbitrary positions in the list are only rarely if ever used.

There are two procedures for returning cells to the available space pool: `Free(S)` returns just the first cell of `S`, while `Free_Sequence(S)` returns all cells of `S`. Note that `Set_Next(S1,S2)` does not free any cells; however, it is almost always applied when `S1` is the tail of a sequence, hence no cells need to be freed.

`Set_First`, `Set_Next`, and `Free` are actually generic parameters of the package, hence these descriptions should be regarded as requirements on these parameters.

Reversing

There are two functions for computing the reverse of a given sequence, `Invert` and `Invert_Copy`. The difference between them illustrates an important distinction that appears in numerous other pairs of operations in this package: we say that `Invert(S)` *mutates* its argument `S`, since it uses the cells of `S` to hold the result, while `Invert_Copy` leaves `S` intact by using newly allocated cells to hold the result. One way to implement `Invert_Copy(S)` would simply be

```
Invert(Copy_Sequence(S))
```

but the actual implementation is more efficient. (It might in fact be reasonable to implement `Copy_Sequence(S)` as

```
Invert(Invert_Copy(S))
```

although a different implementation is actually used.)

Mutative operations, such as `Invert` and many of the operations described below, must be used with care since they can introduce subtle bugs, but they are essential to some kinds of uses of sequences, such as data base applications, and their use in other cases can mean enormous improvements in efficiency.

In some cases, no non-mutative version of an operation is supplied; when it is necessary to perform such an operation on an argument that should not be mutated, one should first copy the argument; e.g., `Sort`, described below, is mutative and there is no `Sort_Copy`, so one should write

```
Sort(Copy_Sequence(S))
```

if `S` will be needed later.

Concatenation

In a similar way, the two functions Concatenate and Concatenate_Copy provide for concatenating two sequences with or without mutating their arguments. More precisely, Concatenate(S1,S2) mutates S1 and shares S2, while Concatenate_Copy(S1,S2) builds its result out of completely new cells, leaving both S1 and S2 intact for further use.

There is another concatenation function, Append(S1,S2), which is equivalent to

```
Concatenate(Copy_Sequence(S1),S2)
```

i.e., S1 is left intact and S2 is shared. The implementation is however slightly more efficient.

There are two functions which combine the functions of reversing and concatenation. Reverse_Append(S1,S2) produces a sequence containing all the elements of S1, in reverse order, followed by those of S2, in order, with S1 left intact and S2 shared. Reverse_Concatenate(S1,S2) returns the same result, but mutating S1 and sharing S2.

Merging and sorting

Merge(S1,S2) merges its arguments into a single sequence, using its generic parameter Test to compare two elements; e.g., Test might be "<=" or "<". If S1 and S2 are in order as determined by Test, then the result will be in order as determined by Test (see Section 6.1.7 for further discussion of ordering). S1 and S2 are both mutated.

If either S1 or either S2 is not in order, Merge(S1,S2) will not be in order, but it nevertheless will be an *interleaving* of S1 and S2: if element X precedes element Y in S1 then X will precede Y in Merge(S1,S2), and similarly for X and Y in S2.

Sort(S) takes a comparison function Test and returns a sequence containing the same elements as S, but in order as determined by Test; S is mutated.

Both Merge and Sort are *stable*: elements considered equal by Test (see the discussion in 6.1.7) will remain in their original order.

Deletion and substitution

There are eight different operations for deleting elements from a sequence, all of which have a generic parameter Test(X) or Test(X,Y), which are Boolean valued functions on element values X and Y. For example, Delete_If(S) returns a sequence consisting of the elements E of S except those satisfying Test(E) = True, mutating S. Delete_Copy_If(S) does the same thing while leaving S intact. See also Delete, Delete_If_Not, Delete_Duplicates, and the corresponding Copy versions.

Similarly, there are six generic subprograms for substituting a new element for some of the elements in a sequence: Substitute(New_Item, Old_Item,S), Substitute_If(New_Item,S), Substitute_If_Not(New_Item,S), and the corresponding Copy versions.

6.1.2 Examining sequences

Basic queries

Is_End(S) returns the Boolean value True if S = Nil, False otherwise. Is_Not_End(S) is the same as not Is_End(S); it is provided purely for convenience. Length(S) returns the number of elements in S.

Counting

The remaining operations for examining sequences are generic, all having either Test(X) or Test(X,Y) as a generic parameter. For example, Count, Count_If, and Count_If_Not are Integer valued functions for counting the elements in a sequence satisfying or not satisfying Test.

Equality and matching

Equal(S1,S2) returns true if S1 and S2 contain the same elements in the same order, using Test as the test for the element equality. Using "=" for Test one obtains the ordinary check for equality of two sequences, but this function can be used to extend other equivalence relations on elements to an equivalence relation on sequences.

A more general operation is the procedure Mismatch, which scans its two inputs in parallel until the first position is found at which they disagree, again using Test as the test for element equality. Mismatch sets its two output parameters to be the subsequences of its inputs beginning at the disagreement position and going to the end. S1 and S2 are shared. (One use of Mismatch is to implement Equal.)

Searching

There are a number of functions for searching a sequence. If S contains an element E such that Test(Item,E) is true, then Find(Item,S) returns the sequence containing the elements of S beginning with the leftmost such element; otherwise Nil is returned. S is shared. Find_If and Find_If_Not are related functions. Position, Position_If, and Position_If_Not are similar, but return as an integer the position of the leftmost occurrence of Item satisfying Test, or -1 if there is none. Search(S1,S2) returns leftmost occurrence in S2 of a subsequence that element-wise matches S1, using Test as the test for element-wise equality; Nil is returned if there is no match.

The other operations for searching are all Boolean valued. Some(S) returns True if Test is true of some element of S, false otherwise. Similarly, Every(S) checks if Test is true of every element of S, Not_Every(S) checks if Test is false for some element, and Not_Any(S) checks if Test is false for every element. All of these operations start with elements indexed $0, 1, \ldots$ and stop performing Test after the first element that determines the answer (e.g., if S is a sequence of integers 2, 3, 5, 7, 11, the operation is Some, and Test(X) checks for X being odd, then Test is performed only on 2 and 3).

6.1.3 Computing with sequences

Procedural iteration

The functions and procedures in this category are generic subprograms for iterating over a sequence, applying some given subprogram to each element. For_Each, for example, is a procedure that takes a generic parameter called The_Procedure; For_Each(S) computes The_Procedure(E) for each element E of S. For_Each_2 takes two sequences and a procedure with two arguments and applies the procedure to corresponding pairs of elements in the sequences.

Mapping

Map(S) applies its generic argument F to each element of S and returns the results as a sequence. F must be a function from the Element type to the Element type. Map mutates S, while Map_Copy leaves it intact. Map_2 and Map_Copy_2 are similar functions for application of a function F of two arguments to corresponding pairs of elements of two sequences S1 and S2.

Reduction

Reduce applies a function of two arguments, F(X,Y), to reduce a sequence to a single value; for example, if F is "+", Reduce(S) sums up the elements of S. It is also necessary to supply Reduce with an element that is the identity for F; e.g., 0 in the case of "+" when the elements are integers.

6.1.4 Exception handling

The exceptions that are raised by the subprograms in this package are renamings of those defined in the package Linked_Exceptions; see the discussion in Section 3.1.

With all the subprograms that have subprograms as generic formal parameters, such as Test or The_Procedure, there is a question of what happens when an unhandled exception is raised by the actual subprogram to which the parameter is instantiated. In all cases, such an exception would end the processing being performed; e.g., with procedure For_Each, if an unhandled exception is raised during execution of The_Procedure on some cell X in S, the following cells are not processed.

6.1.5 Notes on efficiency

All of the subprograms in this package have either constant or linear time and space efficiency, with the exception of Sort, Delete_Duplicates, and Delete_Copy_Duplicates. That is, the computing time and space required to obtain the the answer is a linear function of the length of the input(s), or is a constant. In most cases, subprograms that do not have "Copy" in their names use no space at all in the sense that no new cells are used in constructing sequences, since they reuse the cells in one or more of their arguments to represent the result. The exceptions are Construct, Make_Sequence, Append, and Reverse_Append, which do use new cells in representing all or part of the results they compute.

The computing time for Sort is order $n \log n$, where n is the length of its argument. This is the maximum as well as average and minimum time for sorting (a merge-sort algorithm is used).

In the case of Delete_Duplicates and Delete_Copy_Duplicates, the computing time is order n^2, which can be very time consuming for long lists. In certain cases a faster algorithm could be used; e.g., if the elements can be totally ordered (see Section 6.1.7) then it would be faster to sort them and then eliminate the duplicates in one pass, for a total time of order $n \log n$. This assumes that order is not important in the result. Another possibility would be to use a hashing scheme, which could produce essentially linear time behavior. Neither of these alternatives may be available, however, for cases when Test is not just an equality test; e.g., see the example given in the subprogram description, in which Test is a divisibility check.

6.1.6 Implementation notes

As most of these subprograms are implemented as instances or calls of subprograms in `Linked_List_Algorithms`, one should refer to that package in Chapter 7 for algorithmic details. As with the algorithms in that package, there is no use of recursion and the inline pragma plays an important role in achieving efficiency.

6.1.7 Orderings for `Merge` and `Sort`

A precise description of the kind of function that can be used for comparing values when using the `Merge` and `Sort` subprograms can be given in terms of the notion of a *total order relation.* The generic subprogram parameter `Test` must be either a total order relation (e.g., `"<"` or `">"`) or the negation of a total order relation (e.g., `">="` or `"<="`).

The requirements of a total order relation $\prec$ are:

1. For all X, Y, Z, if $X \prec Y$ and $Y \prec Z$, then $X \prec Z$ (Transitive law).

2. For all X, Y, exactly one of $X \prec Y$, $Y \prec X$, or $X = Y$ holds (Trichotomy law).

In determining whether a proposed relation satisfies the trichotomy law, it is not necessary to have a strict interpretation of "="; one can introduce a notion of equivalence and define the total order relation on the equivalence classes thus defined. Or, looked at another way, we consider X and Y to be equivalent if both $X \prec Y$ and $Y \prec X$ are false. For example, X and Y might be records that have integer values in one field and the records are compared using `"<"` on that field. Thus two records that have the same integer in that field would be equivalent, but might not be equal because of having different values in other fields.

If `Test` is a total order relation or the negation of a total order relation, we can define the notion of a sequence `S` being "in order as determined by `Test`" as follows: for any two elements X and Y that are not equivalent (in the sense defined above), then `Test`(X, Y) is true if and only if `X` precedes `Y` in `S` . (Thus `"<"` or `"<="` will produce ascending order, while `">"` or `">="` will produce descending order.)

6.2 Package specification

The package specification is as follows:

```
with Linked_Exceptions;
 generic

  type Element0  is private;
  type Sequence0 is private;
  Nil0 : Sequence0;
  with function First0(S : Sequence0) return Element0;
  with function Next0(S : Sequence0) return Sequence0;
  with function Construct0(E : Element0; S : Sequence0) return Sequence0;
  with procedure Set_First0(S : Sequence0; E : Element0);
  with procedure Set_Next0(S1, S2 : Sequence0);
  with procedure Free0(S : Sequence0);

package Singly_Linked_Lists is
```

```
  subtype Element  is  Element0;
  subtype Sequence is  Sequence0;
  Nil : Sequence renames Nil0;

  First_Of_Nil : exception
      renames Linked_Exceptions.First_Of_Nil;
  Set_First_Of_Nil : exception
      renames Linked_Exceptions.Set_First_Of_Nil;
  Next_Of_Nil : exception
      renames Linked_Exceptions.Next_Of_Nil;
  Set_Next_Of_Nil : exception
      renames Linked_Exceptions.Set_Next_Of_Nil;
  Out_Of_Construct_Storage : exception
      renames Linked_Exceptions.Out_Of_Construct_Storage;


 {The subprogram specifications}

end Singly_Linked_Lists;
```

6.3 Package body

The package body is as follows:

```
with Linked_List_Algorithms;
package body Singly_Linked_Lists is

  function Copy_Cell(S1, S2 : Sequence) return Sequence is
  begin
    return Construct(First(S1), S2);
  end Copy_Cell;

  pragma Inline(Copy_Cell);

  package Algorithms is new Linked_List_Algorithms(Cell => Sequence,
   Next => Next, Set_Next => Set_Next, Is_End => Is_End,
   Copy_Cell => Copy_Cell);

  generic
    Item : Element;
    with function Test(X, Y : Element) return Boolean;
  function Make_Test(S : Sequence) return Boolean;

  function Make_Test(S : Sequence) return Boolean is
  begin
    return Test(Item, First(S));
  end Make_Test;
```

```
    pragma Inline(Make_Test);

    generic
      with function Test(X : Element) return Boolean;
    function Make_Test_If(S : Sequence) return Boolean;

    function Make_Test_If(S : Sequence) return Boolean is
    begin
      return Test(First(S));
    end Make_Test_If;

    pragma Inline(Make_Test_If);

    generic
      with function Test(X : Element) return Boolean;
    function Make_Test_If_Not(S : Sequence) return Boolean;

    function Make_Test_If_Not(S : Sequence) return Boolean is
    begin
      return not Test(First(S));
    end Make_Test_If_Not;

    pragma Inline(Make_Test_If_Not);

    generic
      with function Test(X, Y : Element) return Boolean;
    function Make_Test_Both(S1, S2 : Sequence) return Boolean;

    function Make_Test_Both(S1, S2 : Sequence) return Boolean is
    begin
      return Test(First(S1), First(S2));
    end Make_Test_Both;

    pragma Inline(Make_Test_Both);

   {The subprogram bodies}


  end Singly_Linked_Lists;
```

6.4 Definitions for the examples

The following definitions are referenced in the examples included in the subprogram descriptions. (This is the skeleton of a test suite in which the examples are included.)

```
                             with System_Allocated_Singly_Linked_Lists;
  package Integer_Linked_Lists is new
```

```
  System_Allocated_Singly_Linked_Lists(Integer);

                 with Integer_Linked_Lists, Text_Io, Examples_Help;
procedure Examples is
  use Integer_Linked_Lists.Inner, Text_Io, Examples_Help;
  Flag : Boolean := True;  -- used in Shuffle_Test

  function Shuffle_Test(X, Y : Integer) return Boolean is
    -- used in examples of Sort and Merge subprograms to
    -- produce merge with every-other-one interleaving;
    -- ignores X and Y
  begin
    Flag := not Flag;
    return Flag;
  end Shuffle_Test;

  function Iota(N : Integer) return Sequence is
    -- returns a sequence of the integers 0, 1, . . . , N - 1
    Result : Sequence := Nil;
  begin
    for I in reverse 0 .. N - 1 loop
      Result := Construct(I, Result);
    end loop;
    return Result;
  end Iota;

  procedure Show_List(S : Sequence) is
    -- prints the sequence S on a line beginning with --:
    -- using Print_Integer from Examples_Help
    procedure Show_List_Aux is new For_Each(Print_Integer);
  begin
    Put("--:"); Show_List_Aux(S); New_Line;
  end Show_List;

begin

 {Examples from the subprograms}

end Examples;
```

6.5 Subprograms

6.5.1 Append

Specification

```
function Append(S1, S2 : Sequence)
      return Sequence;
```

Description Returns a sequence containing all the elements of S1 followed by those of S2. S2 is shared.

Time order n_1

Space order n_1

where $n_1 = \text{length(S1)}$

Mutative? No

Shares? Yes

See also Concatenate, Concatenate_Copy

Examples

```
Show_List(Append(Iota(5), Iota(6)));
-- 0 1 2 3 4 0 1 2 3 4 5
Show_List(Append(Nil, Iota(6)));
-- 0 1 2 3 4 5
Show_List(Append(Iota(5), Nil));
-- 0 1 2 3 4
```

Implementation

```
begin
  return Algorithms.Append(S1, S2);
end Append;
```

6.5.2 Butlast

Specification

```
function Butlast(S : Sequence; N : Integer := 1)
      return Sequence;
```

Description Returns a sequence containing all of the elements of S except the last N elements. S is mutated.

Time order n

Space 0

where $n = \text{length(S)}$

Mutative? Yes

Shares? No

See also Butlast_Copy, Subseqeunce

Examples

```
Show_List(Butlast(Iota(5)));
-- 0 1 2 3
Show_List(Butlast(Iota(5), 3));
-- 0 1
Show_List(Butlast(Iota(5), 5));
--
```

Implementation

```
  I : Integer := Length(S) - N;
begin
  if I <= 0 then
    return Nil;
  elsif N > 0 then
    Set_Next(Nth_Rest(I - 1, S), Nil);
  end if;
  return S;
end Butlast;
```

6.5.3 Butlast_Copy

Specification

```
function Butlast_Copy(S : Sequence; N : Integer := 1)
      return Sequence;
```

Description Returns a sequence containing all of the elements of S except the last N elements.

Time order n

Space $n - \mathrm{N}$

where $n = \mathrm{length(S)}$

Mutative? No

Shares? No

See also Butlast, Subsequence

Examples

```
Show_List(Butlast_Copy(Iota(5)));
-- 0 1 2 3
Show_List(Butlast_Copy(Iota(5), 3));
-- 0 1
Show_List(Butlast_Copy(Iota(5), 5));
--
```

Implementation

```
begin
  return Copy_First_N(S, Length(S) - N);
end Butlast_Copy;
```

6.5.4 Concatenate

Specification

```
function Concatenate(S1, S2 : Sequence)
      return Sequence;
```

Description Returns a sequence containing all the elements of S1 followed by those of S2. S1 is mutated and S2 is shared.

Time order n_1

Space 0

where $n_1 = \text{length}(S1)$

Mutative? Yes

Shares? Yes

See also Append, Concatenate_Copy

Examples

```
Show_List(Concatenate(Iota(5), Iota(6)));
-- 0 1 2 3 4 0 1 2 3 4 5
Show_List(Concatenate(Nil, Iota(6)));
-- 0 1 2 3 4 5
Show_List(Concatenate(Iota(5), Nil));
-- 0 1 2 3 4
```

Implementation

```
begin
  if Is_End(S1) then
    return S2;
  end if;
  Set_Next(Last(S1), S2);
  return S1;
end Concatenate;
```

6.5.5 Concatenate_Copy

Specification

```
function Concatenate_Copy(S1, S2 : Sequence)
      return Sequence;
```

Description Returns a sequence containing all the elements of S1 followed by those of S2.

Time order $n_1 + n_2$

Space order $n_1 + n_2$

where n_1 = length(S1) and n_2 = length(S2)

Mutative? No

Shares? No

See also Append, Concatenate

Implementation

```
begin
  return Append(S1, Append(S2, Nil));
end Concatenate_Copy;
```

6.5.6 Construct

Specification

```
function Construct(E: Element; S : Sequence)
      return Sequence renames Construct0;
```

Description Returns the sequence whose first element is E and whose following elements are those of S. S is shared.

Time constant

Space constant

Mutative? No

Shares? Yes

Details This description is actually a requirement on Construct0, a generic formal parameter of the package. May raise an exception, Out_Of_Construct_Storage. The relations First(Construct(E,S)) = E and Next(Construct(E,S)) = S always hold unless an exception is raised.

See also First, Next, Set_First, Set_Next

6.5.7 Copy_First_N

Specification

```
function Copy_First_N(S : Sequence; N : Integer)
       return Sequence;
```

Description Returns a copy of the first N elements of S.

Time order N

Space order N

Mutative? No

Shares? No

See also Butlast, Butlast_Copy, Copy_Sequence

Implementation

```
begin
  return Algorithms.Append_First_N(S, Nil, N);
end Copy_First_N;
```

6.5.8 Copy_Sequence

Specification

```
function Copy_Sequence(S : Sequence)
      return Sequence;
```

Description Returns a sequence containing the same elements as S, in the same order, but using separate storage cells.

Time order n

Space order n

where $n = \text{length}(S)$

Mutative? No

Shares? No

See also Butlast, Butlast_Copy, Copy_First_N

Implementation

```
begin
  return Append(S, Nil);
end Copy_Sequence;
```

6.5.9 Count

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Count(Item : Element; S : Sequence)
      return Integer;
```

Description Returns a non-negative integer equal to the number of elements E of S such that Test(Item,E) is true.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Count_If, Count_If_Not, Find

Examples

```
declare
  function Count_When_Divides is
    new Integer_Linked_Lists.Inner.Count(Test => Divides);
begin
  Show_Integer(Count_When_Divides(3, Iota(10)));
-- 4
end;
```

Implementation

```
  function Test_Aux is new Make_Test(Item, Test);
  function Count_Aux is new Algorithms.Count(Test_Aux);
begin
  return Count_Aux(S);
end Count;
```

6.5.10 Count_If

Specification

```
generic
with function Test(X : Element) return Boolean;
function Count_If(S : Sequence)
      return Integer;
```

Description Returns a non-negative integer equal to the number of elements E of S such that Test(E) is true.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Count, Count_If_Not, Find, Find_If

Examples

```
declare
   function Count_If_Odd is new Count_If(Test => Odd);
begin
  Show_Integer(Count_If_Odd(Iota(9)));
--  4
end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Count_Aux is new Algorithms.Count(Test_Aux);
begin
  return Count_Aux(S);
end Count_If;
```

6.5.11 Count_If_Not

Specification

```
generic
with function Test(X : Element) return Boolean;
function Count_If_Not(S : Sequence)
       return Integer;
```

Description Returns a non-negative integer equal to the number of elements E of S such that Test(E) is false.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Count, Count_If, Find, Find_If_Not

Examples

```
 declare
    function Count_If_Not_Odd is new Count_If_Not(Test => Odd);
 begin
   Show_Integer(Count_If_Not_Odd(Iota(9)));
--  5
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If_Not(Test);
  function Count_Aux is new Algorithms.Count(Test_Aux);
begin
  return Count_Aux(S);
end Count_If_Not;
```

6.5.12 Delete

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Delete(Item : Element; S : Sequence)
       return Sequence;
```

Description Returns a sequence consisting of all the elements E of S except those for which Test(Item,E) is true. S is mutated.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

See also Delete_If, Delete_If_Not

Examples

```
declare
    function Delete_When_Divides
       is new Integer_Linked_Lists.Inner.Delete(Test => Divides);
begin
   Show_List(Delete_When_Divides(3, Iota(15)));
-- 1 2 4 5 7 8 10 11 13 14
end;
```

Implementation

```
  function Test_Aux is new Make_Test(Item, Test);
  procedure Partition_Aux
    is new Algorithms.Invert_Partition(Test_Aux);
  Temp_1, Temp_2: Sequence := Nil;
begin
  Partition_Aux(S, Temp_1, Temp_2);
  Free_Sequence(Temp_1);
  return Invert(Temp_2);
end Delete;
```

6.5.13 Delete_Copy

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Delete_Copy(Item : Element; S : Sequence)
       return Sequence;
```

Description Returns a sequence consisting of all the elements E of S except those for which Test(Item,E) is true.

Time order nm

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Delete

Examples

```
declare
  function Delete_Copy_When_Divides
    is new Integer_Linked_Lists.Inner.Delete_Copy(Test => Divides);
begin
   Show_List(Delete_Copy_When_Divides(3, Iota(15)));
--  1  2  4  5  7  8  10  11  13  14
end;
```

Implementation

```
  function Test_Aux is new Make_Test(Item, Test);
  function Delete_Copy_Aux
    is new Algorithms.Delete_Copy_Append(Test_Aux);
begin
  return Delete_Copy_Aux(S, Nil);
end Delete_Copy;
```

6.5.14 Delete_Copy_Duplicates

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Delete_Copy_Duplicates(S : Sequence)
      return Sequence;
```

Description Returns a sequence of the elements of S but with only one occurrence of each, using Test as the test for equality.

Time order n^2m

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details The left-most occurrence of each duplicated item is retained.

See also Delete_Duplicates

Examples

```
 declare
    function Delete_Copy_Duplicates_When_Divides
      is new Delete_Copy_Duplicates(Test=>Divides);
 begin
   Show_List(Delete_Copy_Duplicates_When_Divides(Next(Next(Iota(20)))));
--  2  3  5  7  11  13  17  19
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_Both(Test);
  function Delete_Copy_Aux
    is new Algorithms.Delete_Copy_Duplicates_Append(Test_Aux);
begin
  return Delete_Copy_Aux(S, Nil);
end Delete_Copy_Duplicates;
```

6.5.15 Delete_Copy_If

Specification

```
generic
with function Test(X : Element) return Boolean;
function Delete_Copy_If(S : Sequence)
      return Sequence;
```

Description Returns a sequence consisting of all the elements E of S except those for which Test(E) is true.

Time order nm

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Delete_If, Delete_Copy_If_Not

Examples

```
 declare
     function Delete_Copy_If_Odd is new Delete_Copy_If(Test => Odd);
 begin
    Show_List(Delete_Copy_If_Odd(Iota(10)));
--  0  2  4  6  8
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Delete_Copy_Aux
    is new Algorithms.Delete_Copy_Append(Test_Aux);
begin
  return Delete_Copy_Aux(S, Nil);
end Delete_Copy_If;
```

6.5.16 Delete_Copy_If_Not

Specification

```
generic
with function Test(X : Element) return Boolean;
function Delete_Copy_If_Not(S : Sequence)
       return Sequence;
```

Description Returns a sequence consisting of all the elements E of S except those for which Test(E) is false.

Time order nm

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

See also Delete_If_Not, Delete_Copy_If

Examples

```
declare
    function Delete_Copy_If_Not_Odd is new Delete_Copy_If_Not(Test => Odd);
begin
   Show_List(Delete_Copy_If_Not_Odd(Iota(10)));
-- 1  3  5  7  9
end;
```

Implementation

```
  function Test_Aux is new Make_Test_If_Not(Test);
  function Delete_Copy_Aux
    is new Algorithms.Delete_Copy_Append(Test_Aux);
begin
  return Delete_Copy_Aux(S, Nil);
end Delete_Copy_If_Not;
```

6.5.17 Delete_Duplicates

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Delete_Duplicates(S : Sequence)
       return Sequence;
```

Description Returns a sequence of the elements of S but with only one occurrence of each, using Test as the test for equality. S is mutated.

Time order n^2m

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

Details The left-most occurrence of each duplicated item is retained.

See also Delete_Copy_Duplicates

Examples

```
declare
     function Delete_Duplicates_When_Divides is
        new Delete_Duplicates(Test=>Divides);
begin
   Show_List(Delete_Duplicates_When_Divides(Next(Next(Iota(20)))));
--  2  3  5  7  11  13  17  19
end;
```

Implementation

```
  function Test_Aux is new Make_Test_Both(Test);
  function Delete_Aux is
     new Algorithms.Delete_Duplicates(Test_Aux, Free);
begin
  return Delete_Aux(S);
end Delete_Duplicates;
```

6.5.18 Delete_If

Specification

```
generic
with function Test(X : Element) return Boolean;
function Delete_If(S : Sequence)
      return Sequence;
```

Description Returns a sequence consisting of all the elements E of S except those for which Test(E) is true.

Time order nm

Space order n

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

See also Delete_Copy_If, Delete_If_Not

Examples

```
 declare
        function Delete_If_Odd is new Delete_If(Test => Odd);
 begin
    Show_List(Delete_If_Odd(Iota(10)));
--  0  2  4  6  8
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  procedure Partition_Aux
    is new Algorithms.Invert_Partition(Test_Aux);
  Temp_1, Temp_2: Sequence := Nil;
begin
  Partition_Aux(S, Temp_1, Temp_2);
  Free_Sequence(Temp_1);
  return Invert(Temp_2);
end Delete_If;
```

6.5.19 Delete_If_Not

Specification

```
generic
with function Test(X : Element) return Boolean;
function Delete_If_Not(S : Sequence)
       return Sequence;
```

Description Returns a sequence consisting of all the elements E of S except those for which Test(E) is false. S is mutated.

Time order nm

Space order n

where $n = \text{length(S)}$ and $m = \text{average(time for Test)}$

Mutative? Yes

Shares? No

See also Delete_Copy_If_Not, Delete_If

Examples

```
 declare
       function Delete_If_Not_Odd is new Delete_If_Not(Test => Odd);
 begin
    Show_List(Delete_If_Not_Odd(Iota(10)));
--  1  3  5  7  9
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  procedure Partition_Aux is
    new Algorithms.Invert_Partition(Test_Aux);
  Temp_1, Temp_2: Sequence := Nil;
begin
  Partition_Aux(S, Temp_1, Temp_2);
  Free_Sequence(Temp_2);
  return Invert(Temp_1);
end Delete_If_Not;
```

6.5.20 Equal

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Equal(S1, S2 : Sequence)
       return Boolean;
```

Description Returns true if S1 and S2 contain the same elements in the same order, using Test as the test for element equality.

Time order $m\,\text{min}(\text{length}(S1), \text{length}(S2))$

Space 0

where $m = \text{average}(\text{time for Test})$

Mutative? No

Shares? No

See also Mismatch

Examples

```
declare
     function Equal_Equal is new Equal(Test => "=");
begin
   Show_Boolean(Equal_Equal(Iota(10),Iota(10)));
--True
   Show_Boolean(Equal_Equal(Iota(10),Iota(11)));
--False
   Show_Boolean(Equal_Equal(Invert(Iota(10)),Iota(10)));
--False
   Show_Boolean(Equal_Equal(Iota(10),Nil));
--False
   Show_Boolean(Equal_Equal(Nil,Iota(10)));
--False
   Show_Boolean(Equal_Equal(Nil,Nil));
--True
end;
```

Implementation

```
  function Test_Aux is new Make_Test_Both(Test);
  function Equal_Aux is new Algorithms.Equal(Test_Aux);
begin
  return Equal_Aux(S1, S2);
end Equal;
```

6.5.21 Every

Specification

```
generic
with function Test(X : Element) return Boolean;
function Every(S : Sequence)
       return Boolean;
```

Description Returns true if Test is true of every element of S, false otherwise. Elements numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns true if S is Nil.

See also Not_Every, Some

Examples

```
 declare
     function Every_Odd is new Every(Test => Odd);
     function Delete_If_Not_Odd is new Delete_If_Not(Test => Odd);
  begin
    Show_Boolean(Every_Odd(Iota(10)));
--False
    Show_Boolean(Every_Odd(Delete_If_Not_Odd(Iota(10))));
--True
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Every_Aux is new Algorithms.Every(Test_Aux);
begin
  return Every_Aux(S);
end Every;
```

6.5.22 Find

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Find(Item : Element; S : Sequence)
       return Sequence;
```

Description If S contains an element E such that Test(Item,E) is true, then the sequence containing elements of S beginning with the leftmost such element is returned; otherwise Nil is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

See also Find_If, Find_If_Not, Some, Search

Examples

```
declare
    function Find_When_Greater is new Find(Test => "<");
begin
   Show_List(Find_When_Greater(9, Iota(20)));
--  10  11  12  13  14  15  16  17  18  19
end;
```

Implementation

```
    function Test_Aux is new Make_Test(Item, Test);
    function Find_Aux is new Algorithms.Find(Test_Aux);
  begin
    return Find_Aux(S);
  end Find;
```

6.5.23 Find_If

Specification

```
generic
with function Test(X : Element) return Boolean;
function Find_If(S : Sequence)
       return Sequence;
```

Description If S contains an element E such that Test(E) is true, then a sequence containing the elements of S beginning with the leftmost such element is returned; otherwise Nil is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

See also Find, Find_If_Not, Some, Search

Examples

```
declare
    function Find_If_Greater_Than_7
      is new Find_If(Test => Greater_Than_7);
begin
   Show_List(Find_If_Greater_Than_7(Iota (15)));
--  8  9  10  11  12  13  14
end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Find_Aux is new Algorithms.Find(Test_Aux);
begin
  return Find_Aux(S);
end Find_If;
```

6.5.24 Find_If_Not

Specification

```
generic
with function Test(X : Element) return Boolean;
function Find_If_Not(S : Sequence)
       return Sequence;
```

Description If S contains an element E such that Test(E) is false, then a sequence containing the elements of S beginning with the leftmost such element is returned; otherwise Nil is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

See also Find, Find_If, Some, Search

Examples

```
declare
    function Find_If_Not_Greater_Than_7
      is new Find_If_Not(Test => Greater_Than_7);
begin
   Show_List(Find_If_Not_Greater_Than_7(Invert(Iota (15))));
-- 7  6  5  4  3  2  1  0
end;
```

Implementation

```
  function Test_Aux is new Make_Test_If_Not(Test);
  function Find_Aux is new Algorithms.Find(Test_Aux);
begin
  return Find_Aux(S);
end Find_If_Not;
```

6.5.25 First

Specification

```
function First(S : Sequence)
      return Element renames First0;
```

Description Returns the first element of S

Time constant

Space 0

Mutative? No

Shares? No

Details This description is actually a requirement on First0, a generic formal parameter of the package. Raises an exception, First_Of_Nil, if S = Nil.

See also Set_First, Next

6.5.26 For_Each

Specification

```
generic
with procedure The_Procedure(X : Element);
procedure For_Each(S : Sequence);
```

Description Applies The_Procedure to each element of S.

Time order np

Space 0

where n = length(S) and p = average(time for The_Procedure)

Mutative? No

Shares? No

Details S : Sequence

See also For_Each_2, Map

Implementation

```
  procedure The_Procedure_Aux(X : Sequence) is
  begin
    The_Procedure(First(X));
  end The_Procedure_Aux;
  pragma Inline(The_Procedure_Aux);
  procedure For_Each_Aux
    is new Algorithms.For_Each_Cell(The_Procedure_Aux);
begin
  For_Each_Aux(S);
end For_Each;
```

6.5.27 For_Each_Cell

Specification

```
generic
with procedure The_Procedure(X : Sequence);
procedure For_Each_Cell(S : Sequence);
```

Description Applies The_Procedure to each storage cell of S.

Time order np

Space 0

where n = length(S) and p = average(time for The_Procedure)

Mutative? No

Shares? No

See also For_Each, Map

Implementation

```
  procedure For_Each_Aux
    is new Algorithms.For_Each_Cell(The_Procedure);
begin
  For_Each_Aux(S);
end For_Each_Cell;
```

6.5.28 For_Each_2

Specification

```
generic
     with procedure The_Procedure(X, Y : Element);
procedure For_Each_2(S1, S2 : Sequence);
```

Description Applies The_Procedure to pairs of elements of S1 and S2 in the same position.

Time order np

Space order n

where p = average(time for The_Procedure) , $n = \min(n_1, n_2)$, n_1 = length(S1) , n_2 = length(S2)

Mutative? No

Shares? No

Details Stops when the end of either S1 or S2 is reached.

See also For_Each, For_Each_Cell_2, Map

Implementation

```
  procedure The_Procedure_Aux(X, Y : Sequence) is
  begin
    The_Procedure(First(X), First(Y));
  end The_Procedure_Aux;
  pragma Inline(The_Procedure_Aux);
  procedure For_Each_Aux
    is new Algorithms.For_Each_Cell_2(The_Procedure_Aux);
begin
  For_Each_Aux(S1,S2);
end For_Each_2;
```

6.5.29 For_Each_Cell_2

Specification

```
generic
      with procedure The_Procedure(X, Y : Sequence);
procedure For_Each_Cell_2(S1, S2 : Sequence);
```

Description Applies The_Procedure to pairs of cells of S1 and S2 in the same position.

Time order np

Space order n

where p = average(time for The_Procedure) , $n = \min(n_1, n_2)$, n_1 = length(S1) , n_2 = length(S2)

Mutative? No

Shares? No

Details Stops when the end of either S1 or S2 is reached.

See also For_Each_Cell, For_Each_2, Map

Implementation

```
  procedure For_Each_Aux
    is new Algorithms.For_Each_Cell_2(The_Procedure);
begin
  For_Each_Aux(S1,S2);
end For_Each_Cell_2;
```

6.5.30 Free

Specification

```
procedure Free(S : Sequence) renames Free0;
```

Description Causes the first cell of S to be made available for reuse. S is mutated.

Time constant

Space 0 (makes space available)

where $n = \text{length}(S)$

Mutative? Yes

Shares? No

See also Free_Sequence

6.5.31 Free_Sequence

Specification

```
procedure Free_Sequence(S : Sequence);
```

Description Causes the storage cells occupied by S to be made available for reuse. No further reference should be made to S or to any sequence that shares with S.

Time order n

Space 0 (makes space available)

where $n =$ length(S)

Mutative? Yes

Shares? No

See also Free

Implementation

```
  procedure Free_Sequence_Aux is new Algorithms.For_Each_Cell(Free);
begin
  Free_Sequence_Aux(S);
end Free_Sequence;
```

6.5.32 Invert

Specification

```
function Invert(S : Sequence)
       return Sequence;
```

Description Returns a sequence containing the same elements as S but in reverse order. S is mutated.

Time order n

Space 0

where n = length(S)

Mutative? Yes

Shares? No

See also Invert_Copy, Reverse_Append, Reverse_Concatenate

Examples

```
    Show_List(Invert(Iota(6)));
--  5  4  3  2  1  0
```

Implementation

```
begin
  return Reverse_Concatenate(S, Nil);
end Invert;
```

6.5.33 Invert_Copy

Specification

```
function Invert_Copy(S : Sequence)
      return Sequence;
```

Description Returns a new sequence containing the same elements as S but in reverse order.

Time order n

Space order n

where $n = \text{length}(S)$

Mutative? No

Shares? No

See also Invert, Reverse_Append, Reverse_Concatenate

Examples

```
    Show_List(Invert_Copy(Iota(6)));
--  5  4  3  2  1  0
```

Implementation

```
begin
  return Reverse_Append(S, Nil);
end Invert_Copy;
```

6.5.34 Is_End

Specification

```
function Is_End(S : Sequence)
      return Boolean;
```

Description Returns true if S is the Nil sequence, false otherwise.

Time constant

Space 0

Mutative? No

Shares? No

See also Is_Not_End

Implementation

```
begin
  return S = Nil;
end Is_End;
```

6.5.35 Is_Not_End

Specification

```
function Is_Not_End(S : Sequence)
       return Boolean;
```

Description Returns false if S is the Nil sequence, true otherwise.

Time constant

Space 0

Mutative? No

Shares? No

See also Is_End

Implementation

```
begin
  return not Is_End(S);
end Is_Not_End;
```

6.5.36 Last

Specification

```
function Last(S : Sequence)
      return Sequence;
```

Description Returns the sequence containing just the last element of S.

Time order n

Space 0

where n = length(S)

Mutative? No

Shares? Yes

Details Raises an exception, Next_Of_Nil, if S is Nil.

See also First

Examples

```
    Show_List(Last(Iota(6)));
--  5
```

Implementation

```
begin
  return Algorithms.Last(S);
end Last;
```

6.5.37 Length

Specification

```
function Length(S : Sequence)
       return Integer;
```

Description The number of elements in S is returned as a non-negative integer.

Time order n

Space 0

where $n = \text{length}(S)$

Mutative? No

Shares? No

See also

Implementation

```
begin
  return Algorithms.Length(S);
end Length;
```

6.5.38 Make_Sequence

Specification

```
function Make_Sequence(Size : Integer; Initial : Element)
      return Sequence;
```

Description Returns a sequence of length Size in which each element has the value of Initial.

Time order Size

Space order Size

Mutative? No

Shares? No

See also

Examples

```
    Show_List(Make_Sequence(5, 9));
--  9 9 9 9 9
```

Implementation

```
  Result : Sequence := Nil;
  I      : Integer  := Size;
begin
  while I > 0 loop
    Result := Construct(Initial, Result);
    I := I - 1;
  end loop;
  return Result;
end Make_Sequence;
```

6.5.39 Map

Specification

```
generic
with function F(E : Element) return Element;
function Map(S : Sequence)
      return Sequence;
```

Description Returns a sequence consisting of the results of applying F to each element of S. S is mutated.

Time order nf

Space order n

where n = length(S) and f = average(time for F)

Mutative? Yes

Shares? No

See also Map_Copy, Map_2, For_Each

Examples

```
 declare
       function Map_Square is new Map(F => Square);
 begin
    Show_List(Map_Square(Iota(5)));
-- 0  1  4  9  16
 end;
```

Implementation

```
  procedure The_Procedure_Aux(S : Sequence) is
  begin
    Set_First(S, F(First(S)));
  end The_Procedure_Aux;
  pragma Inline(The_Procedure_Aux);
  procedure Map_Aux
    is new Algorithms.For_Each_Cell(The_Procedure_Aux);
begin
  Map_Aux(S);
  return S;
end Map;
```

6.5.40 Map_2

Specification

```
generic
with function F(X, Y : Element) return Element;
function Map_2(S1, S2 : Sequence)
       return Sequence;
```

Description Returns a sequence consisting of the results of applying F to corresponding elements of S1 and S2. S1 is mutated.

Time order nf

Space order n

where f = average(time for F) , $n = \min(n_1, n_2)$, n_1 = length(S1) , n_2 = length(S2)

Mutative? Yes

Shares? No

Details Let $X_0, X_1, \ldots, X_{n_1-1}$ be the elements of S1 and $Y_0, Y_1, \ldots, Y_{n_2-1}$ be those of S2. The sequence returned by Map_2 consists of F(X_0,Y_0), F(X_1,Y_1), ..., F(X_{n-1},Y_{n-1}), where $n = \min(n_1, n_2)$.

See also Map, Map_Copy_2, For_Each

Examples

```
 declare
      function Map_2_Times is new Map_2(F => "*");
 begin
    Show_List(Map_2_Times(Iota(5), Invert(Iota(5))));
--  0  3  4  3  0
 end;
```

Implementation

```
    procedure The_Procedure_Aux(S1, S2 : Sequence) is
    begin
      Set_First(S1, F(First(S1), First(S2)));
    end The_Procedure_Aux;
    pragma Inline(The_Procedure_Aux);
    procedure Map_Aux
      is new Algorithms.For_Each_Cell_2(The_Procedure_Aux);
  begin
    Map_Aux(S1, S2);
    return S1;
  end Map_2;
```

6.5.41 Map_Copy

Specification

```
generic
with function F(E : Element) return Element;
function Map_Copy(S : Sequence)
      return Sequence;
```

Description Returns a sequence consisting of the results of applying F to each element of S.

Time order nf

Space order n

where n = length(S) and f = average(time for F)

Mutative? No

Shares? No

See also Map, For_Each

Examples

```
 declare
       function Map_Copy_Square is new Map_Copy(F => Square);
 begin
    Show_List(Map_Copy_Square(Iota(5)));
--  0  1  4  9  16
 end;
```

Implementation

```
  function Make_Cell(S1, S2 : Sequence) return Sequence is
  begin
    return Construct(F(First(S1)), S2);
  end Make_Cell;
  pragma Inline(Make_Cell);
  function Map_Copy_Aux
    is new Algorithms.Map_Copy_Append(Make_Cell);
begin
  return Map_Copy_Aux(S, Nil);
end Map_Copy;
```

6.5.42 Map_Copy_2

Specification

```
generic
with function F(X, Y : Element) return Element;
function Map_Copy_2(S1, S2 : Sequence)
       return Sequence;
```

Description Returns a sequence consisting of the results of applying F to corresponding elements of S1 and S2.

Time order nf

Space order n

where f = average(time for F) , $n = \min(n_1, n_2)$, n_1 = length(S1) , n_2 = length(S2)

Mutative? No

Shares? No

Details Let $X_0, X_1, \ldots, X_{n_1-1}$ be the elements of S1 and $Y_0, Y_1, \ldots, Y_{n_2-1}$ be those of S2. The sequence returned by Map_Copy_2 consists of F(X_0,Y_0), F(X_1,Y_1), ..., F(X_{n-1},Y_{n-1}), where $n = \min(n_1, n_2)$.

See also Map_2

Examples

```
declare
      function Map_Copy_2_Times is new Map_Copy_2(F => "*");
begin
   Show_List(Map_Copy_2_Times(Iota(5), Invert(Iota(5))));
--  0  3  4  3  0
end;
```

Implementation

```
  function Make_Cell(S1, S2, S3 : Sequence) return Sequence is
  begin
    return Construct(F(First(S1), First(S2)), S3);
  end Make_Cell;
  pragma Inline(Make_Cell);
  function Map_Copy_Aux
    is new Algorithms.Map_Copy_2_Append(Make_Cell);
begin
  return Map_Copy_Aux(S1, S2, Nil);
end Map_Copy_2;
```

6.5.43 Merge

Specification

```
generic
with function Test(X, Y :Element) return Boolean;
function Merge(S1, S2 : Sequence)
       return Sequence;
```

Description Returns a sequence containing the same elements as S1 and S2, interleaved. If S1 and S2 are in order as determined by Test, then the result will be also. Both S1 and S2 are mutated.

Time order $(n_1 + n_2)m$

Space order $n_1 + n_2$

where n_1 = length(S1) , n_2 = length(S2) , and m = average(time for Test)

Mutative? Yes

Shares? No

Details By "interleaved" is meant that if X precedes Y in S1 then X will precede Y in Merge(S1,S2) and similarly for X and Y in S2 (even if S1 or S2 is not in order). The property of stability also holds. See Section 6.1.7 for discussion of the restrictions on Test and definition of "in order as determined by Test."

See also Sort, Concatenate

Implementation

```
function Test_Aux is new Make_Test_Both(Test);
     function Merge_Aux is new Algorithms.Merge(Test_Aux);
   begin
     return Merge_Aux(S1, S2);
   end Merge;
```

6.5.44 Mismatch

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
procedure Mismatch(S1, S2 : in Sequence; S3, S4 : out Sequence);
```

Description S1 and S2 are scanned in parallel until the first position is found at which they disagree, using Test as the test for element equality. S3 and S4 are set to be the subsequences of S1 and S2, respectively, beginning at this disagreement position and going to the end. S1 and S2 are shared.

Time order $\min(n_1, n_2)m$

Space 0

where n_1 = length(S1) and n_2 = length(S2) and m = average(time for Test)

Mutative? No

Shares? Yes

Details S3 and S4 are both set to Nil if S1 and S2 agree entirely.

See also Equal

Examples

```
 declare
     Temp_1, Temp_2 : Sequence;
     procedure Mismatch_Equal is new Mismatch(Test => "=");
 begin
Mismatch_Equal(Iota(10),Iota(10), Temp_1, Temp_2);
    Show_List(Temp_1); Show_List(Temp_2);
--
--
    Mismatch_Equal(Iota(10),Iota(11), Temp_1, Temp_2);
    Show_List(Temp_1); Show_List(Temp_2);
--
-- 10
    Mismatch_Equal(Invert(Iota(10)),Iota(10), Temp_1, Temp_2);
    Show_List(Temp_1); Show_List(Temp_2);
-- 9 8 7 6 5 4 3 2 1 0
-- 0 1 2 3 4 5 6 7 8 9
    Mismatch_Equal(Iota(10),Nil, Temp_1, Temp_2);
    Show_List(Temp_1); Show_List(Temp_2);
-- 0 1 2 3 4 5 6 7 8 9
--
    Mismatch_Equal(Nil,Iota(10), Temp_1, Temp_2);
    Show_List(Temp_1); Show_List(Temp_2);
```

```
--
-- 0  1  2  3  4  5  6  7  8  9
   Mismatch_Equal(Nil,Nil, Temp_1, Temp_2);
   Show_List(Temp_1); Show_List(Temp_2);
--
--
 end;
```

Implementation

```
      function Test_Aux is new Make_Test_Both(Test);
      procedure Mismatch_Aux is new Algorithms.Mismatch(Test_Aux);
    begin
      Mismatch_Aux(S1, S2, S3, S4);
    end Mismatch;
```

6.5.45 Next

Specification

```
function Next(S : Sequence)
      return Sequence renames Next0;
```

Description Returns the sequence consisting of all the elements of S, except the first. S is shared.

Time constant

Space 0

Mutative? No

Shares? Yes

Details This description is actually a requirement on Next0, a generic formal parameter of the package. Raises an exception, Next_Of_Nil, if S is Nil.

See also Set_Next, First

6.5.46 Not_Any

Specification

```
generic
with function Test(X : Element) return Boolean;
function Not_Any(S : Sequence)
      return Boolean;
```

Description Returns true if Test is false of every element of S, false otherwise. Elements numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns true if S is Nil.

See also Every, Some, Not_Every

Examples

```
declare
    function Not_Any_Odd is new Not_Any(Test => Odd);
    function Delete_If_Odd is new Delete_If(Test => Odd);
begin
   Show_Boolean(Not_Any_Odd(Iota(10)));
--False
   Show_Boolean(Not_Any_Odd(Delete_If_Odd(Iota(10))));
--True
end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Not_Any_Aux is new Algorithms.Not_Any(Test_Aux);
begin
  return Not_Any_Aux(S);
end Not_Any;
```

6.5.47 Not_Every

Specification

```
generic
with function Test(X : Element) return Boolean;
function Not_Every(S : Sequence)
       return Boolean;
```

Description Returns true if Test is false of some element of S, false otherwise. Elements numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns false if S is Nil.

See also Every, Some

Examples

```
 declare
      function Not_Every_Odd is new Not_Every(Test => Odd);
      function Delete_If_Not_Odd is new Delete_If_Not(Test => Odd);
 begin
    Show_Boolean(Not_Every_Odd(Iota(10)));
--True
    Show_Boolean(Not_Every_Odd(Delete_If_Not_Odd(Iota(10))));
--False
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Not_Every_Aux is new Algorithms.Not_Every(Test_Aux);
begin
  return Not_Every_Aux(S);
end Not_Every;
```

6.5.48 Nth

Specification

```
function Nth(N : Integer; S : Sequence)
       return Element;
```

Description Returns the N-th element of S.

Time order N

Space 0

Mutative? No

Shares? No

Details The numbering of elements begins with 0, hence Nth(0,S) is the same as First(S) and Nth(Length(S)-1,S) is the same as First(Last(S)). An exception, Next_Of_Nil, is raised if $N > \text{Length}(S) - 1$. If $N < 0$, First(S) is returned.

See also Nth_Rest

Implementation

```
begin
  return First(Nth_Rest(N, S));
end Nth;
```

6.5.49 Nth_Rest

Specification

```
function Nth_Rest(N : Integer; S : Sequence)
      return Sequence;
```

Description Returns a sequence containing the elements of S numbered N, N+1, ..., Length(S)-1.

Time order N

Space order N

Mutative? No

Shares? Yes

Details The numbering of elements begins with 0, hence Nth_Rest(0,S) is the same as S and Nth_Rest(Length(S)-1,S) is the same as Last(S). An exception, Next_Of_Nil, is raised if $N > \text{Length}(S) - 1$. If $N < 0$, S is returned.

See also Nth, Butlast, Butlast_Copy

Implementation

```
begin
  return Algorithms.Nth_Rest(N, S);
end Nth_Rest;
```

6.5.50 Position

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Position(Item : Element; S : Sequence)
       return Integer;
```

Description If S contains an element E such that Test(Item,E) is true, then the index of the leftmost such item is returned; otherwise -1 is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details The index of the first item is 0, of the last is length(S)-1.

See also Position_If, Position_If_Not, Find, Some, Search

Examples

```
declare
    function Position_When_Greater is new Position(Test => "<");
begin

  Show_Integer(Position_When_Greater(3, Iota(7)));
--  4
end;
```

Implementation

```
  function Test_Aux is new Make_Test(Item, Test);
  function Position_Aux is new Algorithms.Position(Test_Aux);
begin
  return Position_Aux(S);
end Position;
```

6.5.51 Position_If

Specification

```
generic
with function Test(X : Element) return Boolean;
function Position_If(S : Sequence)
       return Integer;
```

Description If S contains an element E such that Test(E) is true, then the index of the leftmost such item is returned; otherwise -1 is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details The index of the first item is 0, of the last is length(S)-1.

See also Position_If_Not, Position, Find, Some, Search

Examples

```
declare
    function Position_If_Greater_Than_7 is
       new Position_If(Test => Greater_Than_7);
begin
  Show_Integer(Position_If_Greater_Than_7(Iota(10)));
-- 8
end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Position_Aux is new Algorithms.Position(Test_Aux);
begin
  return Position_Aux(S);
end Position_If;
```

6.5.52 Position_If_Not

Specification

```
generic
with function Test(X : Element) return Boolean;
function Position_If_Not(S : Sequence)
       return Integer;
```

Description If S contains an element E such that Test(E) is false, then the index of the leftmost such item is returned; otherwise -1 is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details The index of the first item is 0, of the last is length(S)-1.

See also Position_If_Not, Position, Find, Some, Search

Examples

```
 declare
     function Position_If_Not_Greater_Than_7 is
        new Position_If_Not(Test=>Greater_Than_7);
 begin
   Show_Integer(Position_If_Not_Greater_Than_7(Invert(Iota(10))));
--  2
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If_Not(Test);
  function Position_Aux is new Algorithms.Position(Test_Aux);
begin
  return Position_Aux(S);
end Position_If_Not;
```

6.5.53 Reduce

Specification

```
generic
Identity : Element;
with function F(X, Y : Element) return Element;
function Reduce(S : Sequence)
       return Element;
```

Description Combines all the elements of S using F; for example, using "+" for F and 0 for Identity one can add up a sequence of Integers.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also For_Each, Map

Examples

```
declare
    function Reduce_Times is new Reduce(Identity => 1, F => "*");
    function Reduce_Plus is new Reduce(Identity => 0, F => "+");
begin
  Show_Integer(Reduce_Times(Next(Iota(5))));
--  24
  Show_Integer(Reduce_Plus(Iota(100)));
--  4950
end;
```

Implementation

```
  function F_Aux(X : Element; S : Sequence) return Element is
  begin
    return F(X, First(S));
  end F_Aux;
  pragma Inline(F_Aux);
  function Reduce_Aux
    is new Algorithms.Accumulate(Element, F_Aux);
begin
  if Is_End(S) then
    return Identity;
  end if;
  return Reduce_Aux(Next(S), First(S));
end Reduce;
```

6.5.54 Reverse_Append

Specification

```
function Reverse_Append(S1, S2 : Sequence)
      return Sequence;
```

Description Returns a sequence consisting of the elements of S1, in reverse order, followed by those of S2 in order. S2 is shared.

Time order n_1

Space order n_1

where $n_1 = \text{length}(S1)$

Mutative? No

Shares? Yes

See also Reverse_Concatenate

Implementation

```
begin
  return Algorithms.Reverse_Append(S1, S2);
end Reverse_Append;
```

6.5.55 Reverse_Concatenate

Specification

```
function Reverse_Concatenate(S1, S2 : Sequence)
      return Sequence;
```

Description Returns a sequence consisting of the elements of S1, in reverse order, followed by those of S2 in order. S1 is mutated and S2 is shared.

Time order n_1

Space 0

where $n_1 = \text{length}(S1)$

Mutative? Yes

Shares? Yes

See also Reverse_Append

Examples

```
    Show_List(Reverse_Concatenate(Iota(5), Iota(6)));
--  4  3  2  1  0  0  1  2  3  4  5
    Show_List(Reverse_Concatenate(Nil, Iota(6)));
--  0  1  2  3  4  5
    Show_List(Reverse_Concatenate(Iota(5), Nil));
--  4  3  2  1  0
```

Implementation

```
begin
  return Algorithms.Reverse_Concatenate(S1, S2);
end Reverse_Concatenate;
```

6.5.56 Search

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Search(S1, S2 : Sequence)
       return Sequence;
```

Description Returns the leftmost occurrence of a subsequence in S2 that element-wise matches S1, using Test as the the test for element-wise equality. If no matching subsequence is found, Nil is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

See also Position, Find, Some, Search

Examples

```
 declare
     function Search_Equal is new Search(Test => "=");
  begin
    Show_List(Search_Equal(Construct(7, Construct(8, Construct(9, Nil))),
Iota(12)));
--  7  8  9  10  11
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_Both(Test);
  function Search_Aux is new Algorithms.Search(Test_Aux);
begin
  return Search_Aux(S1, S2);
end Search;
```

6.5.57 Set_First

Specification

```
procedure Set_First(S : Sequence; E : Element) renames Set_First0;
```

Description Changes S so that its first element is E but the following elements are unchanged.

Time constant

Space 0

Mutative? Yes

Shares? No

Details This description is actually a requirement on Set_Next0, a generic formal parameter of the package. Raises an exception, Set_First_Of_Nil, if S is Nil.

See also Next, Set_First

6.5.58 Set_Next

Specification

```
procedure Set_Next(S1, S2 : Sequence) renames Set_Next0;
```

Description Changes S1 so that its first element is unchanged but the following elements are those of S2. S2 is shared.

Time constant

Space 0

Mutative? Yes

Shares? Yes

Details This description is actually a requirement on Set_Next0, a generic formal parameter of the package. Raises an exception, Set_Next_Of_Nil, if S1 is Nil.

See also Next, Set_First

6.5.59 Set_Nth

Specification

```
procedure Set_Nth(S : Sequence; Index : Integer; New_Item : Element);
```

Description Replaces the element of S specified by Index with New_Item. S is mutated.

Time order Index

Space 0

Mutative? Yes

Shares? No

Details The numbering of elements begins with 0, hence Set_Nth(0,S,X) is the same as Set_First(S,X) and Set_Nth(Length(S)-1,S,X) is the same as Set_First(Last(S),X). An exception, Next_Of_Nil, is raised if S has fewer than Index+1 elements.

See also Nth

Implementation

```
begin
  Set_First(Nth_Rest(Index, S), New_Item);
end Set_Nth;
```

6.5.60 Some

Specification

```
generic
with function Test(X : Element) return Boolean;
function Some(S : Sequence)
       return Boolean;
```

Description Returns true if Test is true of some element of S, false otherwise. Elements numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns false if S is Nil.

See also Not_Every, Every, Not_Any

Examples

```
declare
    function Some_Odd is new Some(Test => Odd);
    function Delete_If_Odd is new Delete_If(Test => Odd);
 begin
   Show_Boolean(Some_Odd(Iota(10)));
--True
   Show_Boolean(Some_Odd(Delete_If_Odd(Iota(10))));
--False
 end;
```

Implementation

```
  function Test_Aux is new Make_Test_If(Test);
  function Some_Aux is new Algorithms.Some(Test_Aux);
begin
  return Some_Aux(S);
end Some;
```

6.5.61 Sort

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Sort(S : Sequence)
       return Sequence;
```

Description Returns a sequence containing the same elements as S, but in order as determined by Test. S is mutated.

Time order $(n \log n)m$

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

Details This is a stable sort. See Section 6.1.7 for discussion of the restrictions on Test and definition of "in order as determined by Test."

See also Merge

Examples

```
declare
    function Sort_Ascending is new Sort(Test => "<");
    function Shuffle_Merge is new Merge(Test => Shuffle_Test);
 begin
   Show_List(Sort_Ascending(Shuffle_Merge(Iota(5), Invert(Iota(5)))));
-- 0  0  1  1  2  2  3  3  4  4
end;
```

Implementation

```
  function Test_Aux is new Make_Test_Both(Test);
  function Sort_Aux is new Algorithms.Sort(32, Nil, Test_Aux);
begin
  return Sort_Aux(S);
end Sort;
```

6.5.62 Subsequence

Specification

```
function Subsequence(S : Sequence; Start, Stop : Integer)
      return Sequence;
```

Description Returns a sequence consisting of the elements of S numbered Start through Stop-1.

Time order Stop

Space order Stop - Start

Mutative? No

Shares? No

Details Start and Stop should satisfy $0 \leq \text{Start} \leq \text{Stop} \leq \text{Length(S)}$. The numbering of elements begins with 0, hence Subsequence(S,0,Length(S)) is a copy of S. An exception, Next_Of_Nil, is raised if Stop > Length(S).

See also Butlast, Butlast_Copy, Copy_First_N

Examples

```
    Show_List(Subsequence(Iota(10), 2, 5));
--  2 3 4
```

Implementation

```
begin
  return Copy_First_N(Nth_Rest(Start, S), Stop - Start);
end Subsequence;
```

6.5.63 Substitute

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Substitute(New_Item, Old_Item : Element; S : Sequence)
       return Sequence;
```

Description Returns a sequence of the elements of S except that those E such that Test(Old_Item,E) is true are replaced by New_Item. S is mutated.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

See also Substitute_Copy, Substitute_If, Substitute_If_Not

Examples

```
declare
    function Substitute_When_Divides
              is new Substitute(Test => Divides);
 begin
   Show_List(Substitute_When_Divides(-1, 3, Iota(15)));
--   -1  1  2 -1  4  5 -1  7  8 -1  10  11 -1  13  14
end;
```

Implementation

```
  procedure The_Procedure_Aux(S : Sequence) is
  begin
    if Test(Old_Item, First(S)) then
      Set_First(S, New_Item);
    end if;
  end The_Procedure_Aux;
  pragma Inline(The_Procedure_Aux);
  procedure Nsub_Aux
    is new Algorithms.For_Each_Cell(The_Procedure_Aux);
begin
  Nsub_Aux(S);
  return (S);
end Substitute;
```

6.5.64 Substitute_Copy

Specification

```
generic
with function Test(X, Y : Element) return Boolean;
function Substitute_Copy(New_Item, Old_Item : Element; S : Sequence)
      return Sequence;
```

Description Returns a sequence of the elements of S except that those E such that Test(Old_Item,E) is true are replaced by New_Item.

Time order nm

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Substitute, Substitute_Copy_If, Substitute_Copy_If_Not

Examples

```
declare
    function Substitute_Copy_When_Divides
              is new Substitute_Copy(Test => Divides);
 begin
   Show_List(Substitute_Copy_When_Divides(-1, 3, Iota(15)));
--   -1  1  2 -1  4  5 -1  7  8 -1  10  11 -1  13  14
end;
```

Implementation

```
  function F_Aux(X : Element) return Element is
  begin
    if Test(Old_Item, X) then
      return New_Item;
    else
      return X;
    end if;
  end F_Aux;
  pragma Inline(F_Aux);
  function Subst_Aux is new Map_Copy(F_Aux);
begin
  return Subst_Aux(S);
end Substitute_Copy;
```

6.5.65 Substitute_Copy_If

Specification

```
generic
with function Test(X : Element) return Boolean;
function Substitute_Copy_If(New_Item : Element; S : Sequence)
       return Sequence;
```

Description Returns a sequence of the elements of S except that those E such that Test(E) is true are replaced by New_Item.

Time order nm

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Substitute_Copy_If_Not, Substitute_If, Substitute_Copy

Examples

```
declare
    function Substitute_Copy_If_Odd
             is new Substitute_Copy_If(Test => Odd);
 begin
   Show_List(Substitute_Copy_If_Odd(-1, Iota(15)));
--  0 -1  2 -1  4 -1  6 -1  8 -1  10 -1  12 -1  14
end;
```

Implementation

```
  function F_Aux(X : Element) return Element is
  begin
    if Test(X) then
      return New_Item;
    else
      return X;
    end if;
  end F_Aux;
  pragma Inline(F_Aux);
  function Subst_Aux is new Map_Copy(F_Aux);
begin
  return Subst_Aux(S);
end Substitute_Copy_If;
```

6.5.66 Substitute_Copy_If_Not

Specification

```
generic
with function Test(X : Element) return Boolean;
function Substitute_Copy_If_Not(New_Item : Element; S : Sequence)
       return Sequence;
```

Description Returns a sequence of the elements of S except that those E such that Test(E) is false are replaced by New_Item.

Time order nm

Space order n

where $n = \text{length(S)}$ and $m = \text{average(time for Test)}$

Mutative? No

Shares? No

See also Substitute_Copy_If, Substitute_If_Not, Substitute_Copy

Examples

```
declare
    function Substitute_Copy_If_Not_Odd
            is new Substitute_Copy_If_Not(Test => Odd);
 begin
   Show_List(Substitute_Copy_If_Not_Odd(-1, Iota(15)));
--  -1  1 -1  3 -1  5 -1  7 -1  9 -1  11 -1  13 -1
end;
```

Implementation

```
  function F_Aux(X : Element) return Element is
  begin
    if Test(X) then
      return X;
    else
      return New_Item;
    end if;
  end F_Aux;
  pragma Inline(F_Aux);
  function Subst_Aux is new Map_Copy(F_Aux);
begin
  return Subst_Aux(S);
end Substitute_Copy_If_Not;
```

6.5.67 Substitute_If

Specification

```
generic
with function Test(X : Element) return Boolean;
function Substitute_If(New_Item : Element; S : Sequence)
       return Sequence;
```

Description Returns a sequence of the elements of S except that those E such that Test(E) is true are replaced by New_Item. S is mutated.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

See also Substitute_If_Not, Substitute, Substitute_Copy

Examples

```
 declare
     function Substitute_If_Odd is new Substitute_If(Test => Odd);
  begin
    Show_List(Substitute_If_Odd(-1, Iota(15)));
--  0 -1  2 -1  4 -1  6 -1  8 -1  10 -1  12 -1  14
 end;
```

Implementation

```
  procedure The_Procedure_Aux(S : Sequence) is
  begin
    if Test(First(S)) then
      Set_First(S, New_Item);
    end if;
  end The_Procedure_Aux;
  pragma Inline(The_Procedure_Aux);
  procedure Nsub_Aux
    is new Algorithms.For_Each_Cell(The_Procedure_Aux);
begin
  Nsub_Aux(S);
  return S;
end Substitute_If;
```

6.5.68 Substitute_If_Not

Specification

```
generic
with function Test(X : Element) return Boolean;
function Substitute_If_Not(New_Item : Element; S : Sequence)
      return Sequence;
```

Description Returns a sequence of the elements of S except that those E such that Test(E) is false are replaced by New_Item. S is mutated.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

See also Substitute_If, Substitute, Substitute_Copy

Examples

```
declare
    function Substitute_If_Not_Odd
             is new Substitute_If_Not(Test => Odd);
 begin
   Show_List(Substitute_If_Not_Odd(-1, Iota(15)));
--   -1  1 -1  3 -1  5 -1  7 -1  9 -1  11 -1  13 -1
end;
```

Implementation

```
    procedure The_Procedure_Aux(S : Sequence) is
    begin
      if not Test(First(S)) then
        Set_First(S, New_Item);
      end if;
    end The_Procedure_Aux;
    pragma Inline(The_Procedure_Aux);
    procedure Nsub_Aux
      is new Algorithms.For_Each_Cell(The_Procedure_Aux);
  begin
    Nsub_Aux(S);
    return S;
  end Substitute_If_Not;
```

Chapter 7

Linked_List_Algorithms Package

7.1 Overview

This is a generic algorithms package that provides 31 algorithms for manipulating a linked list representation of sequences. Only a singly-linked representation is assumed, but many of the algorithms can also reasonably be used with other representations such as circular or non-circular doubly-linked representations. As can be seen from the subprogram implementations in the previous chapter, even for a singly-linked representation these algorithms can be instantiated in various ways to produce a substantially larger collection of useful operations.

Generic algorithm packages such as this are mainly for use in building the library, but nonetheless we include their full descriptions since they illustrate many principles of component reuse in addition to allowing the programmer to be fully aware of the algorithms used. Programmers familiar with the details of these algorithms and the principles of the library structure may also want to consider direct use of generic algorithms packages in some situations.

Perhaps the most interesting aspect of this package is the fact that more than 30 useful algorithms have been programmed entirely in terms of only four primitive operations, which have been made generic formal parameters along with a type, `Cell`:

- `function Next(S : Cell) return Cell;`
- `procedure Set_Next(S1, S2 :  Cell);`
- `function Is_End(S : Cell) return Boolean;`
- `function Copy_Cell(S1, S2 :  Cell) return Cell;`

It is assumed that

- `Next(S)` returns the sequence of cells of `S` except for its first cell;
- `Set_Next(S1, S2)` changes `S1` so that it retains its first cell but the following cells are all of those of `S2`;
- `Is_End(S)` returns true if `S` is the empty sequence of cells; false otherwise; and
- `Copy_Cell(S1, S2)` returns a sequence starting with a new cell containing some information from the first cell of `S1`; the following cells are those of `S2`.

All of the manipulation of *data* is therefore isolated in Copy_Cell.

Most of the algorithms in this package are straightforward; nevertheless, there is a major advantage of having them in a library since there are many small details that must be programmed correctly. Two of the operations, Merge_Non_Empty, and Sort, are of substantial interest from an algorithmic point of view. The Sort operation uses a merge-sort algorithm. In merge-sorting, it is essential that merging is always performed on sequences of the same length whenever possible, in order to produce $n \log n$ time behavior. With linked-lists this could be accomplished by traversing the initial list in order to divide it in two, and so on recursively, but this approach is both clumsy and inefficient (neither of which has prevented it from appearing in some textbooks). Instead, we employ a "binary counter" technique: an array, Register, is kept in which Register(I) always holds either an empty sequence or one of length 2^I, and single element sequences are "added" to the "count" in the register, with carries taking the form of merging of equal-length sequences.

7.2 Package specification

The package specification is as follows:

```
generic

  type Cell is private;
  with function Next(S : Cell) return Cell;
  with procedure Set_Next(S1, S2 : Cell);
  with function Is_End(S : Cell) return Boolean;
  with function Copy_Cell(S1, S2 : Cell) return Cell;

package Linked_List_Algorithms is


 {The subprogram specifications}

end Linked_List_Algorithms;
```

7.3 Package body

The package body is as follows:

```
package body Linked_List_Algorithms is


 {The subprogram bodies}

end Linked_List_Algorithms;
```

7.4 Subprograms

7.4.1 Accumulate

Specification

```
generic
      type Element is private;
  with function F(X : Element; Y : Cell) return Element;
function Accumulate(S : Cell; Initial_Value : Element)
       return Element;
```

Description Puts Initial_Value into an accumulator and successively updates the accumulator with F(accumulator,X) for each cell X of S.

Time order nm

Space 0

where n = length(S) and f = average(time for F)

Mutative? No

Shares? No

See also For_Each_Cell, Map

Implementation

```
  To_Be_Done : Cell    := S;
  Result     : Element := Initial_Value;
begin
  while not Is_End(To_Be_Done) loop
    Result := F(Result, To_Be_Done);
    Advance(To_Be_Done);
  end loop;
  return Result;
end Accumulate;
```

7.4.2 Advance

Specification

```
procedure Advance(S : in out Cell);
pragma inline(Advance);
```

Description Changes S to Next(S).

Time constant

Space 0

Mutative? No

Shares? Yes

Details Used for traversing a sequence, nondestructively—does not free any cells.

See also Next

Implementation

```
begin
  S := Next(S);
end Advance;
```

7.4.3 Append

Specification

```
function Append(S1, S2 : Cell)
      return Cell;
```

Description Returns a sequence containing copies of all the cells of S1 followed by the cells of S2. S2 is shared.

Time order n_1

Space order n_1

where $n_1 = \text{length}(S1)$

Mutative? No

Shares? Yes

See also Append_First_N, Reverse_Append

Implementation

```
  Result, Current : Cell;
  To_Be_Done      : Cell := S1;
begin
  if Is_End(S1) then
    return S2;
  end if;
  Result := Copy_Cell(To_Be_Done, S2);
  Current := Result;
  loop
    Advance(To_Be_Done);
    if Is_End(To_Be_Done) then
      return Result;
    end if;
    Attach_To_Tail(Current, Copy_Cell(To_Be_Done, S2));
  end loop;
end Append;
```

7.4.4 Append_First_N

Specification

```
function Append_First_N(S1, S2 : Cell; N : Integer)
      return Cell;
```

Description Returns a sequence containing the first N cells of S1 followed by all the cells of S2. S2 is shared.

Time order n_1

Space order n_1

where $n_1 = \min(\mathrm{N}, \mathrm{length}(\mathrm{S1}))$

Mutative? No

Shares? Yes

See also Append

Implementation

```
  Result, Current, Temp : Cell;
  To_Be_Done            : Cell    := S1;
  I                     : Integer := N - 1;
begin
  if Is_End(S1) or else I < 0 then
    return S2;
  end if;
  Result := Copy_Cell(To_Be_Done, S2);
  Current := Result;
  loop
    Advance(To_Be_Done);
    I := I - 1;
    if Is_End(To_Be_Done) or else I < 0 then
      return Result;
    end if;
    Attach_To_Tail(Current, Copy_Cell(To_Be_Done, S2));
  end loop;
end Append_First_N;
```

7.4.5 Attach_To_Tail

Specification

```
procedure Attach_To_Tail(X : in out Cell; Y : in Cell);
pragma inline(Attach_To_Tail);
```

Description Performs Set_Next(X,Y) followed by X := Y.

Time constant

Space 0

Mutative? Yes

Shares? Yes

See also

Implementation

```
begin
  Set_Next(X, Y);
  X := Y;
end Attach_To_Tail;
```

7.4.6 Count

Specification

```
generic
      with function Test(X : Cell) return Boolean;
function Count(S : Cell)
       return Integer;
```

Description Returns a non-negative integer equal to the number of cells X of S such that Test(X) is true.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

See also Find

Implementation

```
  Result     : Integer := 0;
  To_Be_Done : Cell    := S;
begin
  while not Is_End(To_Be_Done) loop
    if Test(To_Be_Done) then
      Result := Result + 1;
    end if;
    Advance(To_Be_Done);
  end loop;
  return Result;
end Count;
```

7.4.7 Delete_Copy_Append

Specification

```
generic
     with function Test(X : Cell) return Boolean;
function Delete_Copy_Append(S1, S2 : Cell)
       return Cell;
```

Description Returns a sequence consisting of copies of all the cells X of S1 except those for which Test(X) is true, followed by all the cells of S2. S2 is shared.

Time order nm

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

Details Copy_Cell (a generic parameter of the package) is used to do the copying.

See also Delete, Append

Implementation

```
  To_Be_Done       : Cell := S1;
  Result, Current : Cell;
begin
  while not Is_End(To_Be_Done) and then Test(To_Be_Done) loop
    Advance(To_Be_Done);
  end loop;
  if Is_End(To_Be_Done) then
    return To_Be_Done;
  end if;
  Result := Copy_Cell(To_Be_Done, S2);
  Current := Result;
  Advance(To_Be_Done);
  while not Is_End(To_Be_Done) loop
    if not Test(To_Be_Done) then
      Attach_To_Tail(Current, Copy_Cell(To_Be_Done, S2));
    end if;
    Advance(To_Be_Done);
  end loop;
  return Result;
end Delete_Copy_Append;
```

7.4.8 Delete_Copy_Duplicates_Append

Specification

```
generic
      with function Test(X, Y : Cell) return Boolean;
function Delete_Copy_Duplicates_Append(S1, S2 : Cell)
       return Cell;
```

Description Returns a sequence of copies of the cells of S1 but with only one occurrence of each, using Test as the test for equality, followed by all the cells of S2. S2 is shared.

Time order n^2m

Space order n

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

Details The left-most occurrence of each duplicated item is retained. Copy_Cell (a generic parameter of the package) is used to do the copying.

See also Delete_Duplicates

Implementation

```
  Result, Current, I : Cell;
  To_Be_Done         : Cell := S1;
begin
  if Is_End(S1) then
    return S1;
  end if;
  Result := Copy_Cell(To_Be_Done, S2);
  Current := Result;
  Advance(To_Be_Done);
  while not Is_End(To_Be_Done) loop
    I := Result;
    while not Is_End(I) and then not Test(I, To_Be_Done) loop
      Advance(I);
    end loop;
    if Is_End(I) then
      Attach_To_Tail(Current, Copy_Cell(To_Be_Done, S2));
    end if;
    Advance(To_Be_Done);
  end loop;
  return Result;
end Delete_Copy_Duplicates_Append;
```

7.4.9 Delete_Duplicates

Specification

```
generic
     with function Test(X, Y : Cell) return Boolean;
  with procedure Free(X : Cell);
function Delete_Duplicates(S : Cell)
       return Cell;
```

Description Returns a sequence of the cells of S but with only one occurrence of each, using Test as the test for equality. S is mutated.

Time order n^2m

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

Details The left-most occurrence of each duplicated cell is retained.

See also Delete_Copy_Duplicates

Implementation

```
  Tail, To_Be_Done, I : Cell := S;
begin
  if not Is_End(To_Be_Done) then
    Advance(To_Be_Done);
    while not Is_End(To_Be_Done) loop
      I := S;
      while I /= To_Be_Done and then not Test(I, To_Be_Done) loop
        Advance(I);
      end loop;
      if I = To_Be_Done then
        Tail := To_Be_Done;
        Advance(To_Be_Done);
      else
        I := To_Be_Done;
        Advance(To_Be_Done);
        Set_Next(Tail, To_Be_Done);
        Free(I);
      end if;
    end loop;
  end if;
  return S;
end Delete_Duplicates;
```

7.4.10 Equal

Specification

```
generic
      with function Test(X, Y : Cell) return Boolean;
function Equal(S1, S2 : Cell)
       return Boolean;
```

Description Returns true if S1 and S2 are of the same length and for each position the cells in that position in S1 and S2 are equal, using Test as the test for cell equality.

Time order m min(length(S1),length(S2))

Space 0

where m = average(time for Test)

Mutative? No

Shares? No

See also Mismatch

Implementation

```
  Tail_1, Tail_2 : Cell;
  procedure Mismatch_Aux is new Mismatch(Test);
begin
  Mismatch_Aux(S1, S2, Tail_1, Tail_2);
  return Is_End(Tail_1) and Is_End(Tail_2);
end Equal;
```

7.4.11 Every

Specification

```
generic
      with function Test(X : Cell) return Boolean;
function Every(S : Cell)
       return Boolean;
```

Description Returns true if Test is true of every cell of S, false otherwise. Cells numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns true if Is_End(S) is true.

See also Not_Every, Some

Implementation

```
  To_Be_Done : Cell := S;
begin
  while not Is_End(To_Be_Done) and then Test(To_Be_Done) loop
    Advance(To_Be_Done);
  end loop;
  return Is_End(To_Be_Done);
end Every;
```

7.4.12 Find

Specification

```
generic
      with function Test(X : Cell) return Boolean;
function Find(S : Cell)
       return Cell;
```

Description If S contains an cell X such that Test(X) is true, then the sequence of cells of S beginning with the leftmost such cell is returned; otherwise a cell X such that Is_End(X) is true is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

See also Some, Search

Implementation

```
  To_Be_Done : Cell := S;
begin
  while not Is_End(To_Be_Done) and then not Test(To_Be_Done) loop
    Advance(To_Be_Done);
  end loop;
  return To_Be_Done;
end Find;
```

7.4.13 For_Each_Cell

Specification

```
generic
      with procedure The_Procedure(X : Cell);
procedure For_Each_Cell(S : Cell);
```

Description Applies The_Procedure to each cell of S.

Time order np

Space 0

where n = length(S) and p = average(time for The_Procedure)

Mutative? No

Shares? No

Details 0

See also For_Each_Cell_2, Map

Implementation

```
  To_Be_Done : Cell := S;
  Temp       : Cell;
begin
  while not Is_End(To_Be_Done) loop
    Temp := Next(To_Be_Done);
    The_Procedure(To_Be_Done);
    To_Be_Done := Temp;
  end loop;
end For_Each_Cell;
```

7.4.14 For_Each_Cell_2

Specification

```
generic
      with procedure The_Procedure(X, Y : Cell);
procedure For_Each_Cell_2(S1, S2 : Cell);
```

Description Applies The_Procedure to pairs of cells of S1 and S2 in the same position.

Time order np

Space order n

where n_1 = length(S1) , n_2 = length(S2) , $n = \min(n_1, n_2)$, and p = average(time for The_Procedu

Mutative? No

Shares? No

Details Stops when a cell X is reached in either of S1 or S2 such that Is_End(X) is true.

See also For_Each_Cell, Map_2

Implementation

```
  To_Be_Done1 : Cell := S1;
  To_Be_Done2 : Cell := S2;
  Temp_1       : Cell;
  Temp_2       : Cell;
begin
  while not Is_End(To_Be_Done1)
        and then not Is_End(To_Be_Done2) loop
    Temp_1 := Next(To_Be_Done1);
    Temp_2 := Next(To_Be_Done2);
    The_Procedure(To_Be_Done1, To_Be_Done2);
    To_Be_Done1 := Temp_1;
    To_Be_Done2 := Temp_2;
  end loop;
end For_Each_Cell_2;
```

7.4.15 Invert_Partition

Specification

```
generic
      with function Test(S: Cell) return Boolean;
procedure Invert_Partition(S1: in Cell; S2, S3: in out Cell);
```

Description Partitions the cells of S1 into two sequences S2 and S3 with those in S2 satisfying Test and those in S3 failing Test. The cells in S2 and S3 are in reverse order of their occurrence in S1. S1 is mutated.

Time nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? Yes

See also

Implementation

```
  To_Be_Done, Temp: Cell := S1;
begin
  while not Is_End(To_Be_Done) loop
    Advance(To_Be_Done);
    if Test(Temp) then
      Set_Next(Temp, S2);
      S2 := Temp;
    else
      Set_Next(Temp, S3);
      S3 := Temp;
    end if;
    Temp := To_Be_Done;
  end loop;
end Invert_Partition;
```

7.4.16 Last

Specification

```
function Last(S : Cell)
      return Cell;
```

Description Returns the sequence consisting of just the last cell of S.

Time order n

Space 0

where $n = \text{length}(S)$

Mutative? No

Shares? Yes

Details An attempt is made to compute Next(S) even if Is_End(S) is true.

See also

Implementation

```
  I, J : Cell := S;
begin
  loop
    Advance(J);
    exit when Is_End(J);
    I := J;
  end loop;
  return I;
end Last;
```

7.4.17 Length

Specification

```
function Length(S : Cell)
      return Integer;
```

Description The number of cells in S is returned as a non-negative integer.

Time order n

Space 0

where n = length(S)

Mutative? No

Shares? No

See also

Implementation

```
  Result     : Integer := 0;
  To_Be_Done : Cell    := S;
begin
  while not Is_End(To_Be_Done) loop
    Result := Result + 1;
    Advance(To_Be_Done);
  end loop;
  return Result;
end Length;
```

7.4.18 Map_Copy_2_Append

Specification

```
generic
     with function Make_Cell(X, Y, Z : Cell) return Cell;
function Map_Copy_2_Append(S1, S2, S3 : Cell)
      return Cell;
```

Description Returns a sequence of cells consisting of the results of applying Make_Cell cells of S1 followed by the cells of S2, using Make_Cell to do the copying. S2 is shared.

Time order $n_1 + n_2$

Space order n_1

where $n_1 = \min(\text{length}(S1), \text{length}(S2))$ and $n_2 = \text{length}(S2)$

Mutative? No

Shares? Yes

Details Each application of Make_Cell has a cell of S1 as its first argument, the corresponding cell of S2 as its second argument, and S3 as its third argument. Stops when a cell C in either S1 or S2 is reached such that Is_End(C) is true, ignoring any remaining cells in the other sequence.

See also Append, Reverse_Append

Implementation

```
  Result, Current : Cell;
  To_Be_Done1     : Cell := S1;
  To_Be_Done2     : Cell := S2;
begin
  if Is_End(To_Be_Done1) or else Is_End(To_Be_Done2) then
    return S3;
  end if;
  Result := Make_Cell(To_Be_Done1, To_Be_Done2, S3);
  Current := Result;
  Advance(To_Be_Done1);
  Advance(To_Be_Done2);
  while not Is_End(To_Be_Done1)
        and then not Is_End(To_Be_Done2) loop
    Attach_To_Tail(Current,
                   Make_Cell(To_Be_Done1, To_Be_Done2, S3));
    Advance(To_Be_Done1);
    Advance(To_Be_Done2);
  end loop;
  return Result;
end Map_Copy_2_Append;
```

7.4.19 Map_Copy_Append

Specification

```
generic
      with function Make_Cell(X, Y : Cell) return Cell;
function Map_Copy_Append(S1, S2 : Cell)
       return Cell;
```

Description Returns a sequence of cells consisting of the results of applying Make_Cell to the cells of S1 followed by the cells of S2.

Time order $n_1 + n_2$

Space order n_1

where n_1 = length(S1) and n_2 = length(S2)

Mutative? No

Shares? Yes

Details Each application of Make_Cell has a cell of S1 as its first argument and S2 as its second argument.

See also Append, Reverse_Append

Implementation

```
  Result, Current : Cell;
  To_Be_Done      : Cell := S1;
begin
  if Is_End(To_Be_Done) then
    return S2;
  end if;
  Result := Make_Cell(To_Be_Done, S2);
  Current := Result;
  Advance(To_Be_Done);
  while not Is_End(To_Be_Done) loop
    Attach_To_Tail(Current, Make_Cell(To_Be_Done, S2));
    Advance(To_Be_Done);
  end loop;
  return Result;
end Map_Copy_Append;
```

7.4.20 Merge

Specification

```
generic
     with function Test(X, Y : Cell) return Boolean;
function Merge(S1, S2 : Cell)
       return Cell;
```

Description Returns a sequence containing the same cells as S1 and S2, interleaved. If S1 and S2 are in order as determined by Test, the result is also. Both S1 and S2 are mutated.

Time order $(n_1 + n_2)m$

Space order $n_1 + n_2$

where n_1 = length(S1) , n_2 = length(S2) , and m = average(time for Test)

Mutative? Yes

Shares? No

Details By "interleaved" is meant that if X precedes Y in S1 then X will precede Y in Merge(S1,S2) and similarly for X and Y in S2 (even if S1 or S2 is not in order). The property of stability also holds. See Section 6.1.7 for discussion of the restrictions on Test and definition of "in order as determined by Test."

See also Merge_Non_Empty, Sort

Implementation

```
  function Merge_Aux is new Merge_Non_Empty(Test);
begin
  if Is_End(S1) then
    return S2;
  elsif Is_End(S2) then
    return S1;
  else
    return Merge_Aux(S1, S2);
  end if;
end Merge;
```

7.4.21 Merge_Non_Empty

Specification

```
generic
     with function Test(X, Y : Cell) return Boolean;
function Merge_Non_Empty(S1, S2 : Cell)
       return Cell;
```

Description Returns a sequence containing the same cells as S1 and S2, interleaved. If S1 and S2 are in order as determined by Test, the result is also. Both S1 and S2 are mutated.

Time order $(n_1 + n_2)m$

Space order $n_1 + n_2$

where n_1 = length(S1) , n_2 = length(S2) , and m = average(time for Test)

Mutative? Yes

Shares? No

Details An attempt is made to compute Next(S1) even if Is_End(S1), and similarly for S2. (Merge avoids this potential problem; this subprogram exists mainly for use in implementing the Sort algorithm.) By "interleaved" is meant that if X precedes Y in S1 then X will precede Y in Merge(S1,S2) and similarly for X and Y in S2 (even if S1 or S2 is not in order). The property of stability also holds. See Section 6.1.7 for restrictions on Test and definition of "in order as determined by Test."

See also Merge, Sort

Implementation

```
  I, J, K, Result : Cell;
begin
  if Test(S2, S1) then
    Result := S2;
    I := Next(S2);
    J := S1;
    K := S2;
    goto Wrong_Loop;
  else
    Result := S1;
    I := Next(S1);
    J := S2;
    K := S1;
    goto Right_Loop;
  end if;
  << Right_Loop >>if Is_End(I) then
    Set_Next(K, J);
```

```
      return Result;
    elsif Test(J, I) then
      Attach_To_Tail(K, J);
      J := I;
      I := Next(K);
    else
      K := I;
      Advance(I);
      goto Right_Loop;
    end if;
    << Wrong_Loop >>if Is_End(I) then
      Set_Next(K, J);
      return Result;
    elsif Test(I, J) then
      K := I;
      Advance(I);
      goto Wrong_Loop;
    else
      Attach_To_Tail(K, J);
      J := I;
      I := Next(K);
      goto Right_Loop;
    end if;
  end Merge_Non_Empty;
```

7.4.22 Mismatch

Specification

```
generic
     with function Test(X, Y : Cell) return Boolean;
procedure Mismatch(S1, S2 : in Cell; S3, S4 : out Cell);
```

Description S1 and S2 are scanned in parallel until the first position is found at which they disagree, using Test as the test for cell equality. S3 and S4 are set to be the subsequences of S1 and S2, respectively, beginning at this disagreement position and going to the end. S1 and S2 are shared.

Time order $\min(n_1, n_2)m$

Space 0

where n_1 = length(S1) and n_2 = length(S2) and m = average(time for Test)

Mutative? No

Shares? Yes

Details Is_End(S3) and Is_End(S4) will both be true if S1 and S2 agree entirely.

See also Equal

Implementation

```
  To_Be_Done_1 : Cell := S1;
  To_Be_Done_2 : Cell := S2;
begin
  while not Is_End(To_Be_Done_1)
        and then not Is_End(To_Be_Done_2)
        and then Test(To_Be_Done_1, To_Be_Done_2) loop
    Advance(To_Be_Done_1);
    Advance(To_Be_Done_2);
  end loop;
  S3 := To_Be_Done_1;
  S4 := To_Be_Done_2;
end Mismatch;
```

7.4.23 Not_Any

Specification

```
generic
     with function Test(X : Cell) return Boolean;
function Not_Any(S : Cell)
       return Boolean;
```

Description Returns true if Test is false of every cell of S, false otherwise. Elements numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns true if Is_End(S) is true.

See also Every, Some, Not_Every

Implementation

```
  To_Be_Done : Cell := S;
begin
  while not Is_End(To_Be_Done) and then not Test(To_Be_Done) loop
    Advance(To_Be_Done);
  end loop;
  return Is_End(To_Be_Done);
end Not_Any;
```

7.4.24 Not_Every

Specification

```
generic
      with function Test(X : Cell) return Boolean;
function Not_Every(S : Cell)
       return Boolean;
```

Description Returns true if Test is false of some cell of S, false otherwise. Elements numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns false if Is_End(S) is true.

See also Every, Some

Implementation

```
  To_Be_Done : Cell := S;
begin
  while not Is_End(To_Be_Done) and then Test(To_Be_Done) loop
    Advance(To_Be_Done);
  end loop;
  return not Is_End(To_Be_Done);
end Not_Every;
```

7.4.25 Nth_Rest

Specification

```
function Nth_Rest(N : Integer; S : Cell)
      return Cell;
```

Description Returns a sequence containing the cells of S numbered N, N+1, ..., Length(S)-1.

Time order N

Space order N

Mutative? No

Shares? Yes

Details The numbering of cells begins with 0, hence Nth_Rest(0,S) is the same as S and Nth_Rest(Length(S)-1,S) is the same as Last(S). Assumes that $N \leq \text{Length}(S) - 1$. If $N < 0$, S is returned.

See also Next, Last

Implementation

```
  To_Be_Done : Cell    := S;
  I          : Integer := N;
begin
  while not Is_End(To_Be_Done) and then I > 0 loop
    I := I - 1;
    Advance(To_Be_Done);
  end loop;
  return To_Be_Done;
end Nth_Rest;
```

7.4.26 Position

Specification

```
generic
     with function Test(X : Cell) return Boolean;
function Position(S : Cell)
       return Integer;
```

Description If S contains an cell X such that Test(X) is true, then the index of the leftmost such item is returned; otherwise -1 is returned.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details The index of the first item is 0, of the last is length(S)-1.

See also Find, Some, Search

Implementation

```
  To_Be_Done : Cell    := S;
  I          : Integer := 0;
begin
  while not Is_End(To_Be_Done) and then not Test(To_Be_Done) loop
    I := I + 1;
    Advance(To_Be_Done);
  end loop;
  if Is_End(To_Be_Done) then
    return -1;
  else
    return I;
  end if;
end Position;
```

7.4.27 Reverse_Append

Specification

```
function Reverse_Append(S1, S2 : Cell)
      return Cell;
```

Description Returns a sequence consisting of the cells of S1, in reverse order, followed by those of S2 in order. S2 is shared.

Time order n_1

Space order n_1

where $n_1 = \text{length}(S1)$

Mutative? No

Shares? Yes

See also Reverse_Concatenate, Append

Implementation

```
  Result     : Cell := S2;
  To_Be_Done : Cell := S1;
begin
  while not Is_End(To_Be_Done) loop
    Result := Copy_Cell(To_Be_Done, Result);
    Advance(To_Be_Done);
  end loop;
  return Result;
end Reverse_Append;
```

7.4.28 Reverse_Concatenate

Specification

```
function Reverse_Concatenate(S1, S2 : Cell)
      return Cell;
```

Description Returns a sequence consisting of the cells of S1, in reverse order, followed by those of S2 in order. S1 is mutated and S2 is shared.

Time order n_1

Space 0

where $n_1 = \text{length}(S1)$

Mutative? Yes

Shares? Yes

See also Reverse_Append, Append

Implementation

```
  Result      : Cell := S2;
  To_Be_Done  : Cell := S1;
  Temp        : Cell;
begin
  while not Is_End(To_Be_Done) loop
    Temp := To_Be_Done;
    Advance(To_Be_Done);
    Set_Next(Temp, Result);
    Result := Temp;
  end loop;
  return Result;
end Reverse_Concatenate;
```

7.4.29 Search

Specification

```
generic
     with function Test(X, Y : Cell) return Boolean;
function Search(S1, S2 : Cell)
       return Cell;
```

Description Returns the leftmost occurrence of a subsequence in S2 that matches S1 cell for cell, using Test as the the test for cell equality. If no matching subsequence is found, a sequence S is returned such that Is_End(S) is true.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? Yes

See also Position, Find, Some, Search

Implementation

```
  To_Be_Done     : Cell := S2;
  Tail_1, Tail_2 : Cell;
  procedure Mismatch_Aux is new Mismatch(Test);
begin
  loop
    Mismatch_Aux(S1, To_Be_Done, Tail_1, Tail_2);
    if Is_End(Tail_1) then
      return To_Be_Done;
    elsif Is_End(Tail_2) then
      return Tail_2;
    end if;
    Advance(To_Be_Done);
  end loop;
end Search;
```

7.4.30 Some

Specification

```
generic
     with function Test(X : Cell) return Boolean;
function Some(S : Cell)
       return Boolean;
```

Description Returns true if Test is true of some cell of S, false otherwise. Elements numbered 0, 1, 2, ... are tried in order.

Time order nm

Space 0

where n = length(S) and m = average(time for Test)

Mutative? No

Shares? No

Details Returns false if Is_End(S) is true.

See also Not_Every, Every, Not_Any

Implementation

```
  To_Be_Done : Cell := S;
begin
  while not Is_End(To_Be_Done) and then not Test(To_Be_Done) loop
    Advance(To_Be_Done);
  end loop;
  return not Is_End(To_Be_Done);
end Some;
```

7.4.31 Sort

Specification

```
generic
     Log_Of_Max_Num : Integer;
  Empty          : Cell;
  with function Test(X, Y : Cell) return Boolean;
function Sort(S : Cell)
      return Cell;
```

Description Returns a sequence containing the same cells as S, but in order as determined by Test. S is mutated.

Time order $(n \log n)m$

Space 0

where n = length(S) and m = average(time for Test)

Mutative? Yes

Shares? No

Details This is a stable sorting algorithm. See Section 6.1.7 for restrictions on Test and definition of "in order as determined by Test."

See also Merge

Implementation

```
-- Merge-sort algorithm, using "register adder" technique
     type Table is array(0 .. Log_Of_Max_Num) of Cell;
     Register                : Table   := (others => Empty);
     I, Maximum_Bit_Position : Integer := 0;
     To_Be_Done              : Cell    := S;
     Bit, Carry              : Cell;
     function Merge_Aux is new Merge_Non_Empty(Test);
   begin
     while not (Is_End(To_Be_Done)) loop
       Carry := To_Be_Done;
       Advance(To_Be_Done);
       Set_Next(Carry, Empty);
       I := 0;
       loop
         Bit := Register(I);
         exit when Is_End(Bit);
         Carry := Merge_Aux(Bit, Carry);
         Register(I) := Empty;
         I := I + 1;
       end loop;
```

```
      Register(I) := Carry;
      if Maximum_Bit_Position < I then
        Maximum_Bit_Position := I;
      end if;
    end loop;
    Carry := Register(I);
    loop
      I := I + 1;
      exit when I > Maximum_Bit_Position;
      Bit := Register(I);
      if not Is_End(Bit) then
        Carry := Merge_Aux(Bit, Carry);
      end if;
    end loop;
    return Carry;
  end Sort;
```

Chapter 8

Using the Packages

8.1 Partially Instantiated Packages

The purpose of each of these packages, called "PIPs" is to plug together a low-level data abstraction package with a structural or representational abstraction package, while leaving the `Element` type (and perhaps other parameters) generic. Here we only show PIPs obtained from combining each of the three low-level representations with the `Singly_Linked_Lists` structural abstraction. (There are twelve PIPs included in this release of the library.)

8.1.1 Using System_Allocated_Singly_Linked

From file saslpip.ada--

```
with System_Allocated_Singly_Linked, Singly_Linked_Lists;
generic
  type Element is private;
package System_Allocated_Singly_Linked_Lists is

  package Low_Level is new System_Allocated_Singly_Linked(Element);
  use  Low_Level;

  package Inner is
   new Singly_Linked_Lists(Element, Sequence, Nil, First, Next,
         Construct, Set_First, Set_Next, Free);

end System_Allocated_Singly_Linked_Lists;--
```

8.1.2 Using User_Allocated_Singly_Linked

From file uaslpip.ada--

```
with User_Allocated_Singly_Linked, Singly_Linked_Lists;
generic
  Heap_Size : in Natural;
  type Element is private;
package User_Allocated_Singly_Linked_Lists is
```

```
    package Low_Level
      is new User_Allocated_Singly_Linked(Heap_Size, Element);
    use  Low_Level;

    package Inner is
     new Singly_Linked_Lists(Element, Sequence, Nil, First, Next,
           Construct, Set_First, Set_Next, Free);

  end User_Allocated_Singly_Linked_Lists;--
```

8.1.3 Using Auto_Reallocating_Singly_Linked

From file arslpip.ada--

```
  with Auto_Reallocating_Singly_Linked;
  with Singly_Linked_Lists;
  generic

    Initial_Number_Of_Blocks : in Positive;
    Block_Size               : in Positive;
    Coefficient              : in Float;
    type Element is private;

  package Auto_Reallocating_Singly_Linked_Lists is

    package Low_Level is new
      Auto_Reallocating_Singly_Linked(Initial_Number_Of_Blocks,
                                 Block_Size, Coefficient, Element);
    use  Low_Level;

    package Inner is
     new Singly_Linked_Lists(Element, Sequence, Nil, First, Next,
           Construct, Set_First, Set_Next, Free);

  end Auto_Reallocating_Singly_Linked_Lists;--
```

8.2 Integer Instantiation

A PIP can then be used by instantiating the Element type and any other remaining generic parameters. For example:

```
  with System_Allocated_Singly_Linked_Lists;
  package Integer_Linked is
    new System_Allocated_Singly_Linked_Lists(Integer);
```

Note that the **Inner** package of an instance of a PIP must be used in order to make available all of the subprogram names and other identifiers of the package without requiring a prefix, as in

```
with Integer_Linked;
procedure Application is
  use Integer_Linked.Inner;
  . . .
```

or

```
with Integer_Linked;
package Application is
  use Integer_Linked.Inner;
  . . .
```

8.3 Test Suite and Output

Using `Integer_Linked`, a test suite is produced from the test suite package skeleton given in Chapter 6 and the examples given with each subprogram.

The output that is produced is indicated in the comments in those examples.

D.R. Musser
A.A. Stepanov

ADA GENERIC LIBRARY
LINEAR DATA STRUCTURE PACKAGES, VOLUME ONE

88CRD112
April 1988