CRD

Corporate Research and Development

Schenectady, New York

ADA* GENERIC LIBRARY
LINEAR DATA STRUCTURE PACKAGES, VOLUME TWO

D.R. Musser† and A.A. Stepanov‡

Information Systems Laboratory

April 1988 88CRD113

*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 TWO		
Author(s)	D.R. Musser† A.A. Stepanov‡	**Phone**	(518)387-6120 8*833-6120
Component	Information Systems Laboratory		
Report Number	88CRD113	**Date**	April 1988
Number of Pages	92	**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 Two

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 3, 1988

Contents

Chapter 1

Introduction

This is the second volume of a library of Linear Data Structures facilities in the Ada programming language. The purpose of this library and of the broader Ada Generic Library project 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. In this volume several useful linear data structures are provided as Ada packages, designed and programmed according to the principles of generic algorithms as explained in Volume 1. Familiarity with those principles and with the particular data structures covered in Volume 1 is assumed, as the packages presented here build upon that work.

The following packages are included in this volume:

- `Double_Ended_Lists` 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` provides the familiar linear data structure in which insertions and deletions are restricted to one end.
- `Output_Restricted_Deques` provides a data structure that restricts insertions to both ends and deletions to one end.

These three packages are representational abstractions that produce different structural abstractions from different representations of sequences. For example, any of the three different low-level representations of singly-linked lists provided in Volume 1 (Chapters 3, 4, 5) can easily be plugged together with `Double_Ended_Lists` to produce three different versions of this data structure and its associated algorithms. Each version is provided in the library as a Partially Instantiated Package (PIP), which is a generic package with only the element type, and perhaps some configuration parameters, as generic parameters. See Chapter 5 for further details on the form and usage of PIPs.

Similarly, three more PIPs are provided for plugging together each of the low-level representations of singly-linked lists with `Stacks`. The `Stacks` package can also be combined with low-level representations other than linked lists, since the generic parameters of these packages do not need all of the characteristics of linked-lists (in particular, no `Set_Next` operation is needed). As an illustration of this, Appendix B shows how to supply the needed low-level operations using a simple vector representation.

The parameterization of `Output_Restricted_Deques` is such that the operations assumed are easily provided by `Double_Ended_Lists`. Thus we obtain PIPs by plugging together each of the three PIPs for `Double_Ended_Lists` with with `Output_Restricted_Deques`, producing three different versions of that data structure and its operations. One could, however, produce other versions in terms of a vector representation, since the operations assumed as parameters for `Output_Restricted_Deques`, like those of `Stacks`, can also be efficiently performed in terms of a vector representation.

Chapter 2

Double_Ended_Lists Package

2.1 Overview

This package creates a data type called `Del` and provides 47 subprograms for manipulating values of this type. Basically `Dels` are finite sequences and the operations provided are similar to to those of `Singly_Linked_Lists` (Chapter 6 of Volume 1), but some operations such as concatenation are more efficient (constant time rather than linear in the length of the arguments). In addition, more security against certain kinds of semantic errors is provided, since the package user does not have direct access to pointer values. For example, with `Singly_Linked_Lists` it is possible using the `Set_Next` operation to create a circular list, causing other operations to loop indefinitely, but this is not possible with `Dels`.

The package is generic in the type of elements stored and in the subprograms that provide operations on a singly-linked-list representation of finite sequences. This is a representational abstraction package in which the parameterization is the same as that for `Singly_Linked_Lists`, so that any low-level representation package that can be plugged together with `Singly_Linked_Lists` can also be plugged together with `Double_Ended_Lists`.

2.1.1 A model of double-ended-lists

The internal representation of the `Del` type is as a record containing three pointers into a singly-linked-list representation of a sequence: first-element, last-element, and current-element. While this representation is not directly accessible to the package user, it is nonetheless useful to think in terms of the three pointers as a model of double-ended-lists, both for understanding of what the operations do and of how to use them most effectively.

- The first-element pointer gives the same kind of access to a sequence as one has with `Singly_Linked_Lists`.
- The last-element pointer makes it possible to access the last element in constant time, rather than having to traverse the sequence, and consequently concatenatation of two sequences can be done in constant time.
- The current-element pointer is used as a marker within the sequence; many of the subprograms operate only on the elements starting with the current element through the end of the sequence, and some of these convey their result by moving the current-element pointer to a new position (always to the right).

2.1.2 Invariants

The user of `Double_Ended_Lists` does not have direct access to any of the three pointers; only through certain subprograms can changes in these pointers be effected. The main consequence of this fact, and of the selection of operations actually provided, is that certain properties (called *invariants*) of the representation are maintained, which in turn implies that certain kinds of errors are ruled out. These invariants are as follows: For each value of type `Del`, there is a finite sequence such that either the sequence is empty, in which case the generic formal subprogram `Is_End` returns true on all three pointers; or, letting the sequence be

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

1. There is a sequence of pointers

   $$p_0, p_1, \ldots, p_n$$

   such that p_i points to x_i for $i = 0, \ldots, n-1$; $p_i = \text{Next}(p_{i-1})$ for $i = 1, \ldots, n$; and `Is_End`(p_n) is true.

2. The first-element pointer equals p_0.

3. The last-element pointer equals p_{n-1}.

4. The current-element pointer equals p_i for some i, $i = 0, \ldots, n$.

A direct consequence of these invariants is that there can be no loops in double-ended lists, unlike the case with `Singly_Linked_Lists`.

Note that possibly `Is_End` is true of the current-element pointer. In this case we say that the current-element pointer is *off the end* of the sequence.

2.1.3 Classification of operations

As is the case with `Singly_Linked_Lists`, the operations on `Double_Ended_Lists` can be classified as follows:

1. Construction and modification of sequences

2. Examination of sequences

3. Computing with sequences

The following three subsections give a brief overview of these categories, leaving the details and examples of usage to the individual subprogram descriptions. In comparison with the selection of operations on `Singly_Linked_Lists`, the operations on `Double_Ended_Lists` differ in the following general ways:

- Construction, modification, and examination of sequences includes operations that take advantage of the last-element and current-element pointers.

- Many of the operations operate on the current element or on all of the elements from the current element to the end.

- There are no operations like `Set_Next` that permit pointers to be changed to arbitrary values.

- There is no sharing of list structure.
- Construction and modification operations are provided as procedures rather than functions, and there are no Copy versions of the operations, since it is expected that in most cases Dels will be treated as objects on which computation will be performed by modification.

The Del type is a limited private type, and thus assignment from one variable of type Del to another is prohibited by the language rules. There is, however, a Copy_Sequence operation that can be used in place of assignment.

2.1.4 Construction and modification of sequences

All of the operations in this category are procedures.

Basic construction

Declaration of a variable to be of type Del initializes the variable to represent an empty sequence. There are three operations for adding a single element to a sequence: `Add_First(The_Element, S)`, `Add_Last(The_Element, S)`, and `Add_Current(The_Element, S)`.

`Copy_Sequence(S1, S2)` produces a copy of sequence S2 in S1 that does not overlap with S2 in its memory representation.

Basic Modification

`Set_First(S, E)` changes S so that its first element is E but the following elements are unchanged. Similarly, `Set_Last(S, E)` and `Set_Current(S, E)` change the last and current elements, respectively. `Advance(S)` moves the current-element pointer one element forward. `Initialize(S)` resets the current-element pointer to the first element.

`Drop_Head(S)` removes the elements of S from the first element up to and including the the current element. The complementary operation `Drop_Tail(S)` removes the elements beyond the current element. `Free(S)` removes all the elements; its use is to return the cells occupied by S to the available space pool. The header cell is retained, but is made empty.

Reversing

There is one operation for reversing the order of elements in a sequence: `Invert(S)`.

Splitting and Concatenation

`Split(S1, S2)` splits S1 into two parts: all elements up to and including its current element (this becomes the new value of S1) and all elements following the current element of S1 (this becomes the new value of S2). The old value of S2 is lost (the cells it occupies are returned to available space). The current element of the new S1 its last element and of the new S2 is the first element.

Conversely, `Concatenate(S1, S2)` modifies S1 to be the concatenation of its input value and S2. The output value of S2 is made empty. The current element of the new S1 is the same as in the input value.

Thus, if S2 is empty, the net effect of

```
Split(S1, S2); Concatenate(S1, S2);
```

is a no-op. (If S2 is non-empty the effect is the same as that of `Free(S2)`.)

Merging and Sorting

`Merge(S1, S2)` modifies `S1` to be a sequence containing the same elements as the input values of `S1` and `S2`, interleaved. If `S1` and `S2` are in order as determined by its generic parameter `Test`, then the result will be also.

By "interleaved" is meant that if X precedes Y in `S1` then X will precede Y in the new `S1` and similarly for X and Y in `S2` (even if `S1` or `S2` is not in order). See Section C for discussion of the restrictions on Test and definition of "in order as determined by Test.""

`Sort(S)` takes a comparison function `Test` and modifies `S` to be a sequence containing the same elements as `S`, but in order as determined by `Test`.

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

Deletion and substitution

There are four 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)` modifies `S` by removing those elements `E` of the input value of `S` that satisfy `Test(E) = True`. See also `Delete`, `Delete_If_Not`, and `Delete_Duplicates`.

Similarly, there are three 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)`, and `Substitute_If_Not(New_Item, S)`.

2.1.5 Examining sequences

All of the operations in this category are functions, except `Mismatch`, `Find`, `Find_If`, `Find_If_Not` and `Search`.

Basic queries

`Is_End(S)` returns the `Boolean` value `True` if the current-element pointer of `S` is off the end, `False` otherwise. `Is_Empty(S)` returns `True` if `S` has no elements, `False` otherwise. `Length(S)` returns the number of elements in `S`. `First(S)`, `Last(S)`, and `Current(S)` return the first, last, and current elements of a non-empty sequence `S`; if `S` is empty they all apply the generic formal parameter `First` to a `Sequence` with no elements, raising an exception.

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, beginning with their current 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(S1, S2), which scans the input values of S1 and S2 in parallel until the first position is found at which they disagree, again starting with the current elements and using Test as the test for element equality. Mismatch modifies the current-element pointers of S1 and S2 to be the subsequences of its inputs beginning at the disagreement position and going to the end.

Searching

There are eight operations for searching a sequence. If S contains an element E such that Test(Item,E) is true, at or to the right of its current-element pointer, then Find(Item, S) moves the current-element pointer of S to the the leftmost such element; otherwise the current-element pointer is moved off the end of S. Find_If and Find_If_Not are related procedures. Search(S1, S2) searches S2, starting with the current element, for the leftmost occurrence of a subsequence that element-wise matches S1, and moves the current-element pointer of S2 to this subsequence. If no matching subsequence is found, the current element pointer of S2 is set off the end.

The other operations for searching are all Boolean valued functions. 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 the current element and proceed to the right, just through the first element that determines the answer (e.g., if S from its current element to the end 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).

2.1.6 Computing with sequences

Procedural iteration

The five 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, starting with the current element and going to the end. For_Each_2 takes two sequences and a procedure with two arguments and applies the procedure to corresponding pairs of elements in the sequences, starting with their current elements.

Mapping

Map(S) modifies S to consist of the results of applying its generic parameter F to each element of S, from the current element to the end. F must be a function from the Element type to the Element type. Map_2 is a similar procedure 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. The elements included in the reduction are those from the current element of S to the end. 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.

2.2 Package specification

The package specification is as follows:

```
generic

  type Element  is private;
  type Sequence is private;
  Nil : Sequence;
  with function First(S : Sequence) return Element;
  with function Next(S : Sequence) return Sequence;
  with function Construct(E : Element; S : Sequence) return Sequence;
  with procedure Set_First(S : Sequence; E : Element);
  with procedure Set_Next(S1, S2 : Sequence);
  with procedure Free_Construct(S : Sequence);

package Double_Ended_Lists is

  type Del is limited private;


 {The subprogram specifications}

private

  type Del is record
    First   : Sequence := Nil;
    Current : Sequence := Nil;
    Last    : Sequence := Nil;
  end record;

end Double_Ended_Lists;
```

2.3 Package body

The package body is as follows:

```
with Singly_Linked_Lists;
package body Double_Ended_Lists is

  package Regular_Lists is
```

```
      new Singly_Linked_Lists(Element, Sequence, Nil, First,
               Next, Construct, Set_First, Set_Next, Free_Construct);

  procedure Make_Empty(S : out Del) is
  begin
    S.First := Nil;
    S.Current := Nil;
    S.Last := Nil;
  end Make_Empty;
  pragma Inline(Make_Empty);

  procedure Put_List(S : out Del; L : Sequence) is
  begin
    S.First := L;
    S.Current := L;
    S.Last := Regular_Lists.Last(L);
  end Put_List;
  pragma Inline(Put_List);


 {The subprogram bodies}

end Double_Ended_Lists;
```

2.4 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 Double_Ended_Lists_1; -- a PIP;
package Integer_Double_Ended_Lists is
   new Double_Ended_Lists_1(Integer);

with Integer_Double_Ended_Lists, Text_Io, Examples_Help;
procedure Test_Del is
  use Integer_Double_Ended_Lists.Inner, Text_Io, Examples_Help;
  Flag : Boolean := True;

  function Shuffle_Test(X, Y : Integer) return Boolean is
  begin
    Flag := not Flag;
    return Flag;
  end Shuffle_Test;

  procedure Iota(N : Integer; Result : in out Del) is
  begin
    for I in 0 .. N - 1 loop
      Add_Last(I, Result);
```

```
      end loop;
    end Iota;

    procedure Show_List(S : Del) is
      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}



    Show("End Of Tests");
end;
```

2.5 Subprograms

2.5.1 Add_Current

Specification

```
procedure Add_Current(The_Element : Element; S : in out Del);
pragma inline(Add_Current);
```

Description Inserts The_Element in S after the current element.

Time constant

Space constant

Mutative? Yes

Shares? No

Details The current element is unchanged. Attempts to apply Next to the current element pointer even if Is_End is true of this pointer.

See also Add_First, Add_Last

Examples

```
    declare
      Temp : Del;
    begin
      Iota(3, Temp);
      Add_Current(5, Temp);
      Show_List(Temp);
--  0  5  1  2
      Add_Current(6, Temp);
      Show_List(Temp);
--  0  6  5  1  2
    end;
```

Implementation

```
      Next_One, New_One : Sequence;
    begin
      Next_One := Next(S.Current);
      New_One := Construct(The_Element, Next_One);
      Set_Next(S.Current, New_One);
      if Regular_Lists.Is_End(Next_One) then
        S.Last := New_One;
      end if;
    end Add_Current;
```

2.5.2 Add_First

Specification

```
procedure Add_First(The_Element : Element; S : in out Del);
pragma inline(Add_First);
```

Description Inserts The_Element as the first element of S.

Time constant

Space constant

Mutative? Yes

Shares? No

Details The current element is unchanged, unless S was empty.

See also Add_Current, Add_Last

Examples

```
    declare
      Temp : Del;
    begin
      Iota(3, Temp);
      Add_First(5, Temp);
      Initialize(Temp);
      Show_List(Temp);
--  5  0  1  2
    end;
```

Implementation

```
begin
  S.First := Construct(The_Element, S.First);
  if Regular_Lists.Is_End(S.Last) then
    S.Last := S.First;
    Initialize(S);
  end if;
end Add_First;
```

2.5.3 Add_Last

Specification

```
procedure Add_Last(The_Element : Element; S : in out Del);
pragma inline(Add_Last);
```

Description Inserts The_Element as the last element of S.

Time constant

Space constant

Mutative? Yes

Shares? No

Details The current element is unchanged, unless S was empty.

See also Add_Current, Add_First

Examples

```
    declare
      Temp : Del;
    begin
      Iota(3, Temp);
      Add_Last(5, Temp);
      Show_List(Temp);
--  0 1 2 5
    end;
```

Implementation

```
  Temp : Sequence := S.Last;
begin
  S.Last := Construct(The_Element, Nil);
  if Regular_Lists.Is_End(Temp) then
    S.First := S.Last;
    Initialize(S);
  else
    Set_Next(Temp, S.Last);
  end if;
end Add_Last;
```

2.5.4 Advance

Specification

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

Description Moves the current element pointer forward one element.

Time constant

Space 0

Mutative? No

Shares? No

Details Tries to compute Next of the current element pointer even if Is_End is true of this pointer.

See also

Implementation

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

2.5.5 Concatenate

Specification

```
procedure Concatenate(S1, S2 : in out Del);
pragma inline(Concatenate);
```

Description S1 is modified to be the concatenation of its input value and S2.

Time constant

Space 0

Mutative? Yes

Shares? No

Details The output value of S2 is made empty. The current element of the new S1 is the same as in the input value.

See also

Examples

```
    declare
      Temp_1, Temp_2 : Del;
    begin
      Iota(5, Temp_1);
      Iota(6, Temp_2);
      Concatenate(Temp_1, Temp_2);
      Show_List(Temp_1);
--  0 1 2 3 4 0 1 2 3 4 5
    end;
    declare
      Temp_1, Temp_2 : Del;
    begin
      Iota(6, Temp_2);
      Concatenate(Temp_1, Temp_2);
      Show_List(Temp_1);
--  0 1 2 3 4 5
    end;
    declare
      Temp_1, Temp_2 : Del;
    begin
      Iota(5, Temp_1);
      Concatenate(Temp_1, Temp_2);
      Show_List(Temp_1);
--  0 1 2 3 4
    end;
```

Implementation

```
begin
  if Is_Empty(S1) then
    S1 := S2;
    Make_Empty(S2);
  elsif not Is_Empty(S2) then
    Set_Next(S1.Last, S2.First);
    S1.Last := S2.Last;
    Make_Empty(S2);
  end if;
end Concatenate;
```

2.5.6 Copy_Sequence

Specification

```
procedure Copy_Sequence(S1 : out Del; S2 : Del);
```

Description S1 is made to be a copy of S2.

Time order n_2

Space order n_2

where n_2 = length(S2)

Mutative? No

Shares? No

Details The current element of S1 becomes the first element (and thus may differ from the current element of S2).

See also

Examples

```
    declare
      Temp_1, Temp_2 : Del;
    begin
      Iota(3, Temp_1);
      Copy_Sequence(Temp_2, Temp_1);
      Show_List(Temp_2);
--  0  1  2
    end;
```

Implementation

```
  Temp : Sequence := Regular_Lists.Copy_Sequence(S2.First);
begin
  S1.First := Temp;
  S1.Current := Temp;
  S1.Last := Regular_Lists.Last(Temp);
end Copy_Sequence;
```

2.5.7 Count

Specification

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

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

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

Implementation

```
  function Count_Aux is new Regular_Lists.Count(Test);
begin
  return Count_Aux(Item, S.Current);
end Count;
```

2.5.8 Count_If

Specification

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

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

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

Implementation

```
      function Count_Aux is new Regular_Lists.Count_If(Test);
    begin
      return Count_Aux(S.Current);
    end Count_If;
```

2.5.9 Count_If_Not

Specification

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

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

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

Implementation

```
  function Count_Aux is new Regular_Lists.Count_If_Not(Test);
begin
  return Count_Aux(S.Current);
end Count_If_Not;
```

2.5.10 Current

Specification

```
function Current(S : Del)
      return Element;
pragma inline(Current);
```

Description Returns the current element of S.

Time constant

Space 0

Mutative? No

Shares? No

Details If the current element pointer of S is off the end, this function will apply First to a Sequence with no elements, raising an exception.

See also

Implementation

```
begin
  return First(S.Current);
end Current;
```

2.5.11 Delete

Specification

```
generic
     with function Test(X, Y : Element) return Boolean;
procedure Delete(Item : Element; S : in out Del);
```

Description Modifies S by deleting all elements E of S for which Test(Item,E) is true.

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, Delete_Duplicates

Examples

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

Implementation

```
  function Delete_Aux is new Regular_Lists.Delete(Test);
begin
  Put_List(S, Delete_Aux(Item, S.First));
end Delete;
```

2.5.12 Delete_Duplicates

Specification

```
generic
      with function Test(X, Y : Element) return Boolean;
procedure Delete_Duplicates(S : in out Del);
```

Description Modifies S by deleting all duplicated occurrences of elements, using Test as the test for equality.

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 element is retained.

See also Delete, Delete_If

Examples

```
    declare
      Temp : Del;
      procedure Delete_Duplicates_When_Divides is
          new Delete_Duplicates(Test=>Divides);
    begin
      Iota(20, Temp);
      Advance(Temp);
      Drop_Head(Temp);
      Delete_Duplicates_When_Divides(Temp);
      Show_List(Temp);
--  2  3  5  7  11  13  17  19
    end;
```

Implementation

```
  function Delete_Aux is new Regular_Lists.Delete_Duplicates(Test);
begin
  Put_List(S, Delete_Aux(S.First));
end Delete_Duplicates;
```

2.5.13 Delete_If

Specification

```
generic
     with function Test(X : Element) return Boolean;
procedure Delete_If(S : in out Del);
```

Description Modifies S by deleting all elements E 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, Delete_If_Not

Examples

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

Implementation

```
  function Delete_Aux is new Regular_Lists.Delete_If(Test);
begin
  Put_List(S, Delete_Aux(S.First));
end Delete_If;
```

2.5.14 Delete_If_Not

Specification

```
generic
      with function Test(X : Element) return Boolean;
procedure Delete_If_Not(S : in out Del);
```

Description Modifies S by deleting all elements E for which Test(E) is false.

Time order nm

Space order n

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

Mutative? Yes

Shares? No

See also Delete, Delete_If

Examples

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

Implementation

```
  function Delete_Aux is new Regular_Lists.Delete_If_Not(Test);
begin
  Put_List(S, Delete_Aux(S.First));
end Delete_If_Not;
```

2.5.15 Drop_Head

Specification

```
procedure Drop_Head(S : in out Del);
pragma inline(Drop_Head);
```

Description S is modified by removing all elements up to and including the current element.

Time order k

Space 0

where k = the number of elements up to and including the current element

Mutative? Yes

Shares? No

Details The elements removed are returned to the storage allocator. If Is_End is true of the current element or the current element is the last element, all elements of S are removed.

See also

Examples

```
    declare
      Temp : Del;
    begin
      Iota(4, Temp);
      Advance(Temp);
      Drop_Head(Temp);
      Show_List(Temp);
--  2  3
    end;
```

Implementation

```
      Next_One : Sequence;
    begin
      if Is_End(S) then
        Regular_Lists.Free_Sequence(S.First);
        Make_Empty(S);
      else
        Next_One := Next(S.Current);
        if Regular_Lists.Is_End(Next_One) then
          Regular_Lists.Free_Sequence(S.First);
          Make_Empty(S);
        else
```

```
        Set_Next(S.Current, Nil);
        Regular_Lists.Free_Sequence(S.First);
        S.First := Next_One;
        Initialize(S);
      end if;
    end if;
  end Drop_Head;
```

2.5.16 Drop_Tail

Specification

```
procedure Drop_Tail(S : in out Del);
pragma inline(Drop_Tail);
```

Description S is modified by removing all elements following the current element.

Time order k

Space 0

where k = the number of elements following the current element

Mutative? Yes

Shares? No

Details The elements removed are returned to the storage allocator. If Is_End is true of the current element or the current element is the last element, no elements of S are removed.

See also Drop_Head

Examples

```
        declare
          Temp : Del;
        begin
          Iota(4, Temp);
          Advance(Temp);
          Drop_Tail(Temp);
          Initialize(Temp);
          Show_List(Temp);
    --  0  1
        end;
```

Implementation

```
          Next_One : Sequence;
        begin
          if not Is_End(S) then
            Next_One := Next(S.Current);
            if not Regular_Lists.Is_End(Next_One) then
              Set_Next(S.Current, Nil);
              Regular_Lists.Free_Sequence(Next_One);
              S.Last := S.Current;
            end if;
          end if;
        end Drop_Tail;
```

2.5.17 Equal

Specification

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

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

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

Space 0

where m = average(time for Test)

Mutative? No

Shares? No

See also Mismatch

Implementation

```
  function Equal_Aux is new Regular_Lists.Equal(Test);
begin
  return Equal_Aux(S1.Current, S2.Current);
end Equal;
```

2.5.18 Every

Specification

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

Description Returns true if Test is true of every element of S from the current element to the end, false otherwise. Elements starting with the current element and in successively higher positions are considered 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 the current pointer of S is off the end.

See also Not_Every, Some

Examples

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

Implementation

```
  function Every_Aux is new Regular_Lists.Every(Test);
begin
  return Every_Aux(S.Current);
end Every;
```

2.5.19 Find

Specification

```
generic
     with function Test(X, Y : Element) return Boolean;
procedure Find(Item : Element; S : in out Del);
```

Description If S contains an element E such that Test(Item,E) is true, at the current element or beyond, then the leftmost such element is made to be the current element; otherwise the current element pointer falls off the end of S.

Time order nm

Space 0

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

Mutative? No

Shares? No

See also Find_If, Find_If_Not, Some, Search

Examples

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

Implementation

```
  function Find_Aux is new Regular_Lists.Find(Test);
begin
  S.Current := Find_Aux(Item, S.Current);
end Find;
```

2.5.20 Find_If

Specification

```
generic
      with function Test(X : Element) return Boolean;
procedure Find_If(S : in out Del);
```

Description If S contains an element E such that Test(E) is true, at the current element or beyond, then the current element is set to the leftmost such element; otherwise the current element pointer falls off the end of S.

Time order nm

Space 0

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

Mutative? No

Shares? No

See also Find, Find_If_Not, Some, Search

Examples

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

Implementation

```
  function Find_Aux is new Regular_Lists.Find_If(Test);
begin
  S.Current := Find_Aux(S.Current);
end Find_If;
```

2.5.21 Find_If_Not

Specification

```
generic
     with function Test(X : Element) return Boolean;
procedure Find_If_Not(S : in out Del);
```

Description If S contains an element E such that Test(E) is false, at the current element or beyond, then the current element is set to the leftmost such element; otherwise the current element pointer falls off the end of S.

Time order nm

Space 0

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

Mutative? No

Shares? No

See also Find, Find_If, Some, Search

Examples

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

Implementation

```
  function Find_Aux is new Regular_Lists.Find_If_Not(Test);
begin
  S.Current := Find_Aux(S.Current);
end Find_If_Not;
```

2.5.22 First

Specification

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

Description Returns the first (left-most) element of S.

Time constant

Space 0

Mutative? No

Shares? No

Details Attempts to apply the generic formal First even if S has no elements.

See also

Implementation

```
begin
  return First(S.First);
end First;
```

2.5.23 For_Each

Specification

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

Description Applies The_Procedure to each element of S starting with the current element and going to the end.

Time order np

Space 0

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

Mutative? No

Shares? No

See also For_Each_2, Map

Implementation

```
  procedure For_Each_Aux is
        new Regular_Lists.For_Each(The_Procedure);
begin
  For_Each_Aux(S.Current);
end For_Each;
```

2.5.24 For_Each_2

Specification

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

Description Applies The_Procedure to pairs of elements of S1 and S2 in the same position, starting with the current elements and going to the end.

Time order np

Space 0

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, Map, Map_2

Implementation

```
  procedure For_Each_Aux is
        new Regular_Lists.For_Each_2(The_Procedure);
begin
  For_Each_Aux(S1.Current, S2.Current);
end For_Each_2;
```

2.5.25 Free

Specification

```
procedure Free(S : in out Del);
pragma inline(Free);
```

Description Causes the storage cells occupied by S to be made available for reuse.

Time order n

Space 0 (makes space available)

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

Mutative? Yes

Shares? No

Details The header record of S is retained, but is made empty.

See also

Implementation

```
begin
  Regular_Lists.Free_Sequence(S.First);
  Make_Empty(S);
end Free;
```

2.5.26 Initialize

Specification

```
procedure Initialize(S : in out Del);
pragma inline(Initialize);
```

Description The current element of S is reset to the first element.

Time constant

Space 0

Mutative? No

Shares? No

See also Make_Empty

Implementation

```
begin
  S.Current := S.First;
end Initialize;
```

2.5.27 Invert

Specification

```
procedure Invert(S : in out Del);
```

Description Modifies S to contain the same elements as its input value, but in reverse order.

Time order n

Space 0

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

Mutative? Yes

Shares? No

Details The element referred to by the current element is the same as before the inversion, but its position is changed: if initially it was i, the new current element position is $n - 1 - i$.

See also

Examples

```
    declare
      Temp : Del;
    begin
      Iota(6, Temp);
      Invert(Temp);
      Initialize(Temp);
      Show_List(Temp);
--  5  4  3  2  1  0
    end;
    declare
      Temp : Del;
    begin
      Invert(Temp);
      Show_List(Temp);
--
    end;
```

Implementation

```
      Temp : Sequence := Regular_Lists.Invert(S.First);
    begin
      S.Last := S.First;
      S.First := Temp;
    end Invert;
```

2.5.28 Is_Empty

Specification

```
function Is_Empty(S : Del)
      return Boolean;
pragma inline(Is_Empty);
```

Description Returns true if S has no elements, false otherwise.

Time constant

Space 0

Mutative? No

Shares? No

See also Is_End

Implementation

```
begin
  return Regular_Lists.Is_End(S.First);
end Is_Empty;
```

2.5.29 Is_End

Specification

```
function Is_End(S : Del)
       return Boolean;
pragma inline(Is_End);
```

Description Returns true if the current element of S has fallen off the end, false otherwise.

Time constant

Space 0

Mutative? No

Shares? No

See also Is_Empty

Implementation

```
begin
  return Regular_Lists.Is_End(S.Current);
end Is_End;
```

2.5.30 Last

Specification

```
function Last(S : Del)
       return Element;
pragma inline(Last);
```

Description Returns the last element of S.

Time constant

Space 0

Mutative? No

Shares? No

Details Attempts to apply the generic formal First even if S is empty.

See also First, Current

Implementation

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

2.5.31 Length

Specification

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

Description Returns the number of elements in S from the current element to the end, as a non-negative integer.

Time constant

Space 0

Mutative? No

Shares? No

Details The current element is included in the count.

See also

Implementation

```
begin
  return Regular_Lists.Length(S.Current);
end Length;
```

2.5.32 Map

Specification

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

Description Modifies S to consist of the results of applying F to each element of S, from the current element to the end.

Time order nf

Space order n

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

Mutative? Yes

Shares? No

See also For_Each

Examples

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

Implementation

```
          Dummy : Sequence;
          function Map_Aux is new Regular_Lists.Map(F);
        begin
          Dummy := Map_Aux(S.Current);
        end Map;
```

2.5.33 Map_2

Specification

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

Description Modifies S1 to be a sequence of the results of applying F to corresponding elements of S1 and S2, starting with the current elements and going to the end.

Time order nf

Space order n

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

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 new value of S1 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 For_Each

Examples

```
    declare
      Temp_1, Temp_2 : Del;
      procedure Map_2_Times is new Map_2(F => "*");
    begin
      Iota(5, Temp_1);
      Iota(5, Temp_2);
      Invert(Temp_2);
      Initialize(Temp_2);
      Map_2_Times(Temp_1, Temp_2);
      Show_List(Temp_1);
--  0 3 4 3 0
    end;
```

Implementation

```
      Dummy : Sequence;
      function Map_2_Aux is new Regular_Lists.Map_2(F);
    begin
      Dummy := Map_2_Aux(S1.Current, S2.Current);
    end Map_2;
```

2.5.34 Merge

Specification

```
generic
     with function Test(X, Y : Element) return Boolean;
procedure Merge(S1, S2 : in out Del);
```

Description Modifies S1 to be 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 the new S1 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 C for discussion of the restrictions on Test and definition of "in order as determined by Test."

See also Sort, Concatenate

Examples

```
    declare
      Temp_1, Temp_2 : Del;
      procedure Shuffle_Merge is new Merge(Test => Shuffle_Test);
    begin
      Iota(5, Temp_1);
      Iota(5, Temp_2);
      Invert(Temp_2);
      Initialize(Temp_2);
      Shuffle_Merge(Temp_1, Temp_2);
      Show_List(Temp_1);
--  0  4  1  3  2  2  3  1  4  0
    end;
```

Implementation

```
      function Merge_Aux is new Regular_Lists.Merge(Test);
    begin
      Put_List(S1, Merge_Aux(S1.First, S2.First));
      Make_Empty(S2);
    end Merge;
```

2.5.35 Mismatch

Specification

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

Description S1 and S2 are scanned in parallel, starting from their current elements, until the first position is found at which they disagree, using Test as the test for element equality. S1 and S2 have their current elements set to the elements at which the first disagreement occurs.

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? No

Details S1 and S2 both have their current pointers set off the end if S1 and S2 agree entirely.

See also Equal

Implementation

```
  Temp_1, Temp_2 : Sequence;
  procedure Mismatch_Aux is new Regular_Lists.Mismatch(Test);
begin
  Mismatch_Aux(S1.Current, S2.Current, Temp_1, Temp_2);
  S1.Current := Temp_1;
  S2.Current := Temp_2;
end Mismatch;
```

2.5.36 Not_Any

Specification

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

Description Returns true if Test is false of every element of S, from its current element on, false otherwise. Elements numbered i, i + 1, i + 2, ... are tried in order, where the i-th element is current.

Time order nm

Space 0

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

Mutative? No

Shares? No

Details Returns true if the current element is off the end.

See also Every, Some, Not_Every

Examples

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

Implementation

```
  function Not_Any_Aux is new Regular_Lists.Not_Any(Test);
begin
  return Not_Any_Aux(S.Current);
end Not_Any;
```

2.5.37 Not_Every

Specification

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

Description Returns true if Test is false of some element of S, from its current element on, false otherwise. Elements numbered i, i + 1, i + 2, ... are tried in order, where the i-th element is current.

Time order nm

Space 0

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

Mutative? No

Shares? No

Details Returns false if the current element of S is off the end.

See also Every, Some

Examples

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

Implementation

```
  function Not_Every_Aux is new Regular_Lists.Not_Every(Test);
begin
  return Not_Every_Aux(S.Current);
end Not_Every;
```

2.5.38 Reduce

Specification

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

Description Combines all the elements of S using F, from the current element on; 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
      Temp : Del;
      function Reduce_Times is new Reduce(Identity => 1, F => "*");
    begin
      Iota(5, Temp);
      Advance(Temp);
      Show_Integer(Reduce_Times(Temp));
--  24
    end;
    declare
      Temp : Del;
      function Reduce_Plus is new Reduce(Identity => 0, F => "+");
    begin
      Iota(100, Temp);
      Show_Integer(Reduce_Plus(Temp));
--  4950
    end;
```

Implementation

```
  function Reduce_Aux is new Regular_Lists.Reduce(Identity, F);
begin
  return Reduce_Aux(S.Current);
end Reduce;
```

2.5.39 Search

Specification

```
generic
     with function Test(X, Y : Element) return Boolean;
procedure Search(S1 : Del; S2 : in out Del);
```

Description Searches S2, starting with the current element, for the leftmost occurrence of a subsequence that element-wise matches S1, using Test as the the test for element-wise equality, and moves the current element pointer of S2 to this subsequence. If no matching subsequence is found, the current element pointer of S2 is set off the end.

Time order nm

Space 0

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

Mutative? No

Shares? No

See also Position, Find, Some, Search

Examples

```
    declare
      Temp_1, Temp_2 : Del;
      procedure Search_Equal is new Search(Test => "=");
    begin
      Add_Last(7, Temp_1);
      Add_Last(8, Temp_1);
      Add_Last(9, Temp_1);
      Iota(12, Temp_2);
      Search_Equal(Temp_1, Temp_2);
      Show_List(Temp_2);
--  7  8  9  10  11
    end;
```

Implementation

```
  function Search_Aux is new Regular_Lists.Search(Test);
begin
  S2.Current := Search_Aux(S1.Current, S2.Current);
end Search;
```

2.5.40 Set_Current

Specification

```
procedure Set_Current(S : Del; X : Element);
pragma inline(Set_Current);
```

Description S is modified by replacing its current element by X.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Attempts to apply the generic formal Set_First even if the current element pointer is off the end of S.

See also Current, Set_First

Implementation

```
begin
  Set_First(S.Current, X);
end Set_Current;
```

2.5.41 Set_First

Specification

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

Description S is modified by replacing its first element by X.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Attempts to apply the generic formal Set_First even if Is_End is true of the first element pointer of S (which can only be true of S has no elements).

See also Current, Set_First

Implementation

```
begin
  Set_First(S.First, X);
end Set_First;
```

2.5.42 Set_Last

Specification

```
procedure Set_Last(S : Del; X : Element);
pragma inline(Set_Last);
```

Description S is modified by replacing its last element by X.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Attempts to apply the generic formal Set_First even if Is_End is true of the last element pointer of S (which can only be true of S has no elements).

See also Current, Set_First

Implementation

```
begin
  Set_First(S.Last, X);
end Set_Last;
```

2.5.43 Some

Specification

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

Description Returns true if Test is true of some element of S, from the current element on, false otherwise. Elements numbered i, i + 1, i + 2, ... are tried in order, where the i-th element is current.

Time order nm

Space 0

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

Mutative? No

Shares? No

Details Returns false if the current element of S is off the end.

See also Not_Every, Every, Not_Any

Examples

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

Implementation

```
  function Some_Aux is new Regular_Lists.Some(Test);
begin
  return Some_Aux(S.Current);
end Some;
```

2.5.44 Sort

Specification

```
generic
     with function Test(X, Y : Element) return Boolean;
procedure Sort(S : in out Del);
```

Description Modifies S to be a sequence containing the same elements as S, but in order as determined by Test.

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 C for discussion of the restrictions on Test and definition of "in order as determined by Test."

See also Merge

Examples

```
    declare
      Temp_1, Temp_2 : Del;
      procedure Sort_Ascending is new Sort(Test => "<");
      procedure Shuffle_Merge is new Merge(Test => Shuffle_Test);
    begin
      Iota(5, Temp_1);
      Iota(5, Temp_2);
      Invert(Temp_2);
      Initialize(Temp_2);
      Shuffle_Merge(Temp_1, Temp_2);
      Sort_Ascending(Temp_1);
      Show_List(Temp_1);
--  0  0  1  1  2  2  3  3  4  4
    end;
```

Implementation

```
  function Sort_Aux is new Regular_Lists.Sort(Test);
begin
  Put_List(S, Sort_Aux(S.First));
end Sort;
```

2.5.45 Split

Specification

```
procedure Split(S1, S2 : in out Del);
pragma inline(Split);
```

Description S1 is split into two parts: all elements up to and including its current element (this becomes the new value of S1) and all elements following the current element of S1 (this becomes the new value of S2).

Time constant

Space 0

Mutative? Yes

Shares? No

Details Procedure Free is applied to the input value of S2. The current element of the new S1 its last element and of the new S2 is the first element.

See also Concatenate

Examples

```
    declare
      Temp_1, Temp_2 : Del;
    begin
      Iota(4, Temp_1);
      Advance(Temp_1);
      Split(Temp_1, Temp_2);
      Initialize(Temp_1);
      Show_List(Temp_2);
--  2  3
      Show_List(Temp_1);
--  0  1
    end;
```

Implementation

```
      Next_One : Sequence;
    begin
      Free(S2);
      if not Is_End(S1) then
        Next_One := Next(S1.Current);
        if not Regular_Lists.Is_End(Next_One) then
          Set_Next(S1.Current, Nil);
          S2.First := Next_One;
          S2.Current := Next_One;
          S2.Last := S1.Last;
```

```
        S1.Last := S1.Current;
      end if;
    end if;
  end Split;
```

2.5.46 Substitute

Specification

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

Description Modifies S so that, from the current element on, the elements E such that Test(Old_Item,E) is true are replaced by New_Item.

Time order nm

Space 0

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

Mutative? Yes

Shares? No

See also Substitute_If, Substitute_If_Not

Examples

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

Implementation

```
      Dummy : Sequence;
      function Substitute_Aux is new Regular_Lists.Substitute(Test);
    begin
      Dummy := Substitute_Aux(New_Item, Old_Item, S.Current);
    end Substitute;
```

2.5.47 Substitute_If

Specification

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

Description Modifies S so that, from the current pointer on, the elements E such that Test(E) is true are replaced by New_Item.

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

Examples

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

Implementation

```
  Dummy : Sequence;
  function Substitute_If_Aux is new Regular_Lists.Substitute_If(Test);
begin
  Dummy := Substitute_If_Aux(New_Item, S.Current);
end Substitute_If;
```

2.5.48 Substitute_If_Not

Specification

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

Description Modifies S so that, from the current pointer on, the elements E such that Test(E) is false are replaced by New_Item.

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

Examples

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

Implementation

```
  Dummy : Sequence;
  function Substitute_If_Not_Aux is
      new Regular_Lists.Substitute_If_Not(Test);
begin
  Dummy := Substitute_If_Not_Aux(New_Item, S.Current);
end Substitute_If_Not;
```

Chapter 3

Stacks Package

This package provides one of the simplest of linear data structures, in which insertions and deletions of data are restricted to one end. Its name suggests the most appropriate model for understanding its behavior: a stack of papers on a desk, which can only be changed by placing a sheet of paper on top or by removing one from the top, and the one on top is the only one whose information can be examined. Another frequently used term for a stack discipline is "Last-In First-Out" (LIFO).

3.1 Package specification

The package specification is as follows:

```
generic
  type Element  is private;
  type Sequence is private;
  with procedure Create(S : out Sequence);
  with function Full(S : Sequence) return Boolean;
  with function Empty(S : Sequence) return Boolean;
  with function First(S : Sequence) return Element;
  with function Next(S : Sequence) return Sequence;
  with function Construct(E : Element; S : Sequence) return Sequence;
  with procedure Free_Construct(S : Sequence);
package Stacks is
  type Stack is limited private;
  Stack_Underflow, Stack_Overflow : exception;

 {The subprogram specifications}

  private
  type Stack is new Sequence;

end Stacks;
```

3.2 Package body

The package body is as follows:

```
package body Stacks is

 {The subprogram bodies}

end Stacks;
```

3.3 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 Stacks_1; -- a PIP;
  package Integer_Stacks is new Stacks_1(Integer);

  with Integer_Stacks, Text_Io, Examples_Help;
  procedure Test_Stacks is
    use Integer_Stacks.Inner, Text_Io, Examples_Help;

    procedure Show_Stack(S : in out Stack) is
      procedure Show_Stack_Aux is new For_Each(Print_Integer);
    begin
      Put("--:"); Show_Stack_Aux(S); New_Line;
    end Show_Stack;

begin


   {Examples from the subprograms}



    Show("End Of Tests");
end;
```

3.4 Subprograms

3.4.1 Create

Specification

```
procedure Create(S : out Stack);
pragma inline(Create);
```

Description Makes S be an empty stack.

Time constant

Space 0

Mutative? Yes

Shares? No

See also Push, Pop

Examples

```
-- See Push
```

Implementation

```
begin
  Create(Sequence(S));
end Create;
```

3.4.2 For_Each

Specification

```
generic
with procedure The_Procedure(E : Element);
procedure For_Each(S: in out Stack);
pragma inline(For_Each);
```

Description Successively removes each element E of S, from the top down, and applies The_Procedure to E.

Time order np

Space 0

where n is the number of elements in the stack, and p = average(time for The_Procedure)

Mutative? Yes

Shares? No

Details Does nothing if S is empty. If an unhandled exception is raised while executing The_Procedure on an element, those elements that were below it are left in S.

See also Pop, Top

Examples

```
-- See Push
```

Implementation

```
  An_Element: Element;
begin
  while not Is_Empty(S) loop
    Pop(An_Element, S);
    The_Procedure(An_Element);
  end loop;
end For_Each;
```

3.4.3 Is_Empty

Specification

```
function Is_Empty(S : Stack)
       return Boolean;
pragma inline(Is_Empty);
```

Description Returns true if S has no elements in it, false otherwise.

Time constant

Space 0

Mutative? No

Shares? No

See also Push, Pop

Examples

```
-- See Push
```

Implementation

```
begin
  return Empty(Sequence(S));
end Is_Empty;
```

3.4.4 Pop

Specification

```
procedure Pop(The_Element : out Element; S : in out Stack);
pragma inline(Pop);
```

Description Causes the top element of S to be removed and returned as the value of The_Element.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Raises an exception, Stack_Underflow, if S is empty.

See also Push, Top

Examples

```
-- See Push
```

Implementation

```
  Old : Sequence := Sequence(S);
begin
  if Empty(Sequence(S)) then raise Stack_Underflow;
  end if;
  The_Element := Top(S);
  S := Stack(Next(Sequence(S)));
  Free_Construct(Old);
end Pop;
```

3.4.5 Push

Specification

```
procedure Push(The_Element : in Element; S : in out Stack);
pragma inline(Push);
```

Description Places The_Element on top of S.

Time constant

Space constant

Mutative? Yes

Shares? No

Details Raises an exception, Stack_Overflow, if S is already full.

See also Pop, Top

Examples

```
declare
   S : Stack; E : Integer;
 begin
  Create(S);
  Push(2, S); Push(3, S); Push(5, S); Push(7, S);
  Show_Integer(Top(S));
--  7
  Pop(E, S);
  Show_Integer(E);
--  7
  Show_Integer(Top(S));
--  5
  Show_Boolean(Is_Empty(S));
-- False
  Show_Stack(S);
--  5  3  2
  Show_Boolean(Is_Empty(S));
-- True
 end;
```

Implementation

```
begin
  if Full(Sequence(S)) then raise Stack_Overflow;
  end if;
  S := Stack(Construct(The_Element, Sequence(S)));
end Push;
```

3.4.6 Top

Specification

```
function Top(S : Stack)
      return Element;
pragma inline(Top);
```

Description Returns the top element of S, without removing it.

Time constant

Space 0

Mutative? No

Shares? No

Details Raises an exception, Stack_Underflow, if S is empty.

See also Pop, Push

Examples

```
-- See Push
```

Implementation

```
begin
  if Is_Empty(S) then raise Stack_Underflow;
  end if;
  return First(Sequence(S));
end Top;
```

Chapter 4

Output_Restricted_Deques Package

A *deque* is a linear data structure consisting of finite sequences in which insertions and deletions are permitted only at the ends. Thus stacks and queues can be viewed as special cases of deques that have further restrictions on accesses: a stack prohibits both insertions and deletions at one end, while a queue can only have insertions at one end and only deletions at the other. One of the least restricted cases of a deque is that in which both insertions and deletions are permitted at one end (called the front), but at the other end (the rear) only insertions are allowed; hence it is called *output-restricted*. This package provides such a data structure, as a representational abstraction.

The generic parameters of the package are types and subprograms that allow the package to be easily plugged together with Double_Ended_Lists, but the parameters also could be satisfied with a vector representation of sequences.

4.1 Package specification

The package specification is as follows:

```
generic
  type Element  is private;
  type Sequence is limited private;
  with procedure Create(S : in out Sequence);
  with function Full(S : Sequence) return Boolean;
  with function Empty(S : Sequence) return Boolean;
  with function First(S : Sequence) return Element;
  with function Last(S : Sequence) return Element;
  with procedure Add_First(E : Element; S : in out Sequence);
  with procedure Add_Last(E : Element; S : in out Sequence);
  with procedure Drop_First(S : in out Sequence);
package Output_Restricted_Deques is
  type Deque is limited private;
  Deque_Underflow, Deque_Overflow : exception;

 {The subprogram specifications}
```

```
  private
  type Deque is new Sequence;
end Output_Restricted_Deques;
```

4.2 Package body

The package body is as follows:

```
package body Output_Restricted_Deques is

 {The subprogram bodies}

end Output_Restricted_Deques;
```

4.3 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 Output_Restricted_Deques_1;  -- a PIP
package Integer_Output_Restricted_Deques is new
    Output_Restricted_Deques_1(Integer);

with Integer_Output_Restricted_Deques, Text_Io, Examples_Help;
procedure Test_Deques is
  use Integer_Output_Restricted_Deques.Inner, Text_Io, Examples_Help;

  procedure Show_Deque(D : in out Deque) is
  -- note that this makes D empty;
    procedure Show_Deque_Aux is new For_Each(Print_Integer);
  begin
    Put("--:"); Show_Deque_Aux(D); New_Line;
  end Show_Deque;

begin


   {Examples from the subprograms}



  Show("End Of Tests");
end;
```

4.4 Subprograms

4.4.1 Create

Specification

```
procedure Create(D : in out Deque);
pragma inline(Create);
```

Description Makes D be an empty deque.

Time constant

Space 0

Mutative? Yes

Shares? No

See also

Examples

```
-- See Push_Front
```

Implementation

```
begin
  Create(Sequence(D));
end Create;
```

4.4.2 For_Each

Specification

```
generic
with procedure The_Procedure(E : Element);
procedure For_Each(D: in out Deque);
pragma inline(For_Each);
```

Description Successively removes each element E of D, from the front to the rear, and applies The_Procedure to E.

Time order np

Space 0

where n is the number of elements in D, and p = average(time for The_Procedure)

Mutative? Yes

Shares? No

Details Does nothing if D is empty. If an unhandled exception is raised while executing The_Procedure on an element, those elements that were after it (from front to rear) are left in the deque.

See also

Examples

```
-- See Push_Front
```

Implementation

```
 An_Element: Element;
begin
  while not Is_Empty(D) loop
    Pop_Front(An_Element, D);
    The_Procedure(An_Element);
  end loop;
end For_Each;
```

4.4.3 Front

Specification

```
function Front(D : Deque)
      return Element;
pragma inline(Front);
```

Description Returns the front element of D, without removing it.

Time constant

Space 0

Mutative? No

Shares? No

Details Raises an exception, Deque_Underflow, if D is empty.

See also Pop_Front, Push_Front

Examples

```
-- See Push_Front, Push_Rear
```

Implementation

```
begin
  if Is_Empty(D) then raise Deque_Underflow;
  end if;
  return First(Sequence(D));
end Front;
```

4.4.4 Is_Empty

Specification

```
function Is_Empty(D : Deque)
      return Boolean;
pragma inline(Is_Empty);
```

Description Returns true if D has no elements in it, false otherwise.

Time constant

Space 0

Mutative? No

Shares? No

See also Push_Front, Push_Rear, Pop_Front

Examples

```
-- See Push_Front
```

Implementation

```
begin
  return Empty(Sequence(D));
end Is_Empty;
```

4.4.5 Pop_Front

Specification

```
procedure Pop_Front(The_Element : out Element; D : in out Deque);
pragma inline(Pop_Front);
```

Description Causes the front element of D to be removed and returned as the value of The_Element.

Time constant

Space 0

Mutative? Yes

Shares? No

Details Raises an exception, Deque_Underflow, if D is empty.

See also Push_Front, Front

Examples

```
-- See Push_Front, Push_Rear
```

Implementation

```
begin
  if Empty(Sequence(D)) then raise Deque_Underflow;
  else
    The_Element := Front(D);
    Drop_First(Sequence(D));
  end if;
end Pop_Front;
```

4.4.6 Push_Front

Specification

```
procedure Push_Front(The_Element : in Element; D : in out Deque);
pragma inline(Push_Front);
```

Description Places The_Element on the front of D.

Time constant

Space constant

Mutative? Yes

Shares? No

Details Raises an exception, Deque_Overflow, if D is already full.

See also Pop_Front, Front

Examples

```
declare
   D : Deque; E : Integer;
 begin
  Create(D);
  Push_Front(2, D); Push_Front(3, D); Push_Front(5, D); Push_Front(7, D);
  Show_Integer(Front(D));
--  7
  Pop_Front(E, D);
  Show_Integer(E);
--  7
  Show_Integer(Front(D));
--  5
  Show_Boolean(Is_Empty(D));
-- False
  Show_Deque(D);
--  5  3  2
  Show_Boolean(Is_Empty(D));
--  True
 end;
```

Implementation

```
begin
  if Full(Sequence(D)) then raise Deque_Overflow;
  end if;
  Add_First(The_Element, Sequence(D));
end Push_Front;
```

4.4.7 Push_Rear

Specification

```
procedure Push_Rear(The_Element : in Element; D : in out Deque);
pragma inline(Push_Rear);
```

Description Places The_Element on the rear of D.

Time constant

Space constant

Mutative? Yes

Shares? No

Details Raises an exception, Deque_Overflow, if D is already full.

See also Rear

Examples

```
declare
   D : Deque; E : Integer;
 begin
  Push_Rear(2, D); Push_Rear(3, D); Push_Rear(5, D); Push_Rear(7, D);
  Show_Integer(Rear(D));
--  7
  Pop_Front(E, D);
  Show_Integer(E);
--  2
  Show_Integer(Front(D));
--  3
  Show_Boolean(Is_Empty(D));
-- False
  Show_Deque(D);
--  3  5  7
  Show_Boolean(Is_Empty(D));
--  True
 end;
```

Implementation

```
begin
  if Full(Sequence(D)) then raise Deque_Overflow;
  end if;
  Add_Last(The_Element, Sequence(D));
end Push_Rear;
```

4.4.8 Rear

Specification

```
function Rear(D : Deque)
      return Element;
pragma inline(Rear);
```

Description Returns the rear element of D, without removing it.

Time constant

Space 0

Mutative? No

Shares? No

Details Raises an exception, Deque_Underflow, if D is empty.

See also Push_Rear

Examples

```
-- See Push_Rear
```

Implementation

```
begin
  if Is_Empty(D) then raise Deque_Underflow;
  end if;
  return Last(Sequence(D));
end Rear;
```

Chapter 5

Using the Packages

5.1 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. In Volume 1 we showed PIPs obtained from combining each of three low-level representations of singly-linked-lists with the `Singly_Linked_Lists` structural abstraction. For each of the representational abstractions in Chapters 2, 3, and 4 of this volume, there are three three PIPs included in the library for plugging the representational abstraction together with a particular representation.

5.1.1 PIPs for Double_Ended_Lists

From file delpip1.ada--

```
      with System_Allocated_Singly_Linked, Double_Ended_Lists;
generic
  type Element is private;
package Double_Ended_Lists_1 is

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

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

end Double_Ended_Lists_1;--
```

From file delpip2.ada--

```
      with User_Allocated_Singly_Linked, Double_Ended_Lists;
generic
  Heap_Size : in Natural;
  type Element is private;
package Double_Ended_Lists_2 is
```

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

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

end Double_Ended_Lists_2;--
```

From file delpip3.ada--

```
     with Auto_Reallocating_Singly_Linked, Double_Ended_Lists;
generic
  Initial_Number_Of_Blocks : in Positive;
  Block_Size               : in Positive;
  Coefficient              : in Float;
  type Element is private;
package Double_Ended_Lists_3 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 Double_Ended_Lists(Element, Sequence, Nil, First, Next,
      Construct, Set_First, Set_Next, Free);

end Double_Ended_Lists_3;--
```

5.1.2 PIPs for Stacks

In this case the low-level representation provided by System_Allocated_Singly_Linked does not provide exactly the operations needed by Stacks, but appropriate definitions of the missing operations (Create, Full, and Empty) are easily specified in the package specification and programmed in the package body.

From file stackp1.ada--

```
     with System_Allocated_Singly_Linked, Stacks;
generic
  type Element is private;
package Stacks_1 is

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

  procedure Create(S : out Sequence);
  pragma inline(Create);
```

```
  function Full(S : Sequence) return Boolean;
  pragma inline(Full);
  function Empty(S : Sequence) return Boolean;
  pragma inline(Empty);

  package Inner is
    new Stacks(Element, Sequence, Create, Full, Empty,
      First, Next, Construct, Free);

end Stacks_1;


package body Stacks_1 is

  use Low_Level;
  procedure Create(S : out Sequence) is
  begin
    S := Nil;
  end Create;

  function Full(S : Sequence) return Boolean is
  begin
    return False;  -- Stacks are unbounded when
                   -- represented as singly-linked-lists;
  end Full;

  function Empty(S : Sequence) return Boolean is
  begin
    return S = Nil;
  end Empty;

end Stacks_1;--
```

The other two PIPs, Stacks_2 and Stacks_3 for for plugging Stacks together with User_Allocated_Singly_Linked and Auto_Reallocating_Singly_Linked, respectively, are similar to Stacks_1.

5.1.3 PIPs for Output_Restricted_Deques

Another twist to the construction of PIPs is introduced here. The operations needed by Output_Restricted_Deques are conveniently supplied by Double_Ended_Lists, so we use an instance of a PIP for Double_Ended_Lists as the low-level representation. Since, as in the PIP for Stacks, not all of the operations needed are supplied directly, two are specified and programmed in this PIP's specification and body.

From file outdeqp1.ada--

```
      with Double_Ended_Lists_1, Output_Restricted_Deques;
generic
  type Element is private;
```

```
package Output_Restricted_Deques_1 is

  package Low_Level is new Double_Ended_Lists_1(Element);
  use Low_Level.Inner;

  function Full(D : Del) return Boolean;
  pragma inline(Full);
  procedure Drop_First(D : in out Del);
  pragma inline(Drop_First);

  package Inner is new
    Output_Restricted_Deques(Element, Del, Free, Full, Is_Empty, First,
      Last, Add_First, Add_Last, Drop_First);

end Output_Restricted_Deques_1;

package body Output_Restricted_Deques_1 is
  use Low_Level.Inner;

  function Full(D : Del) return Boolean is
  begin
    return False;  -- double-ended-lists are unbounded when
                   -- represented as singly-linked-lists;
  end Full;

  procedure Drop_First(D : in out Del) is
  begin
    Initialize(D);
    Drop_Head(D);
  end Drop_First;

end Output_Restricted_Deques_1;--
```

Similar PIPs, called Output_Restricted_Deques_2 and Output_Restricted_Deques_3, are provided for plugging Output_Restricted_Deques together with User_Allocated_Singly_Linked and Auto_Reallocating_Singly_Linked, respectively.

5.2 Test Suites and Output

Test suites are produced from the test suite package skeletons given in the chapters on the packages and the examples given with each subprogram.

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

Appendix A

Examples_Help Package

The following package defines a few procedures and functions that aid in the construction of examples and test cases for the various packages.

From file examhelp.ada--

```
package Examples_Help is

-- I/O procedures

  procedure Print_Integer(I : in Integer);
  procedure Show(The_String : String);
  procedure Show_Boolean(B : Boolean);
  procedure Show_Integer(I : Integer);

-- Some other little functions needed to construct examples

  function Divides(A, B : Integer) return Boolean;
  function Even(A : Integer) return Boolean;
  function Odd(A : Integer) return Boolean;
  function Greater_Than_7(A : Integer) return Boolean;
  function Square(A : Integer) return Integer;

end Examples_Help;

with Text_Io; use Text_Io;
package body Examples_Help is

-- I/O procedures

  procedure Print_Integer(I : in Integer) is
  begin
    Put(Integer'Image(I));
    Put(" ");
  end Print_Integer;

  procedure Show(The_String : String) is
```

```
  begin
    Put(The_String); New_Line;
  end Show;

  procedure Show_Boolean(B : Boolean) is
  begin
    if B then
      Show("--: True");
    else
      Show("--: False");
    end if;
  end Show_Boolean;

  procedure Show_Integer(I : Integer) is
  begin
    Put("--:"); Print_Integer(I); New_Line;
  end Show_Integer;

-- Some other little functions needed to construct examples

  function Divides(A, B : Integer) return Boolean is
  begin
    return B mod A = 0;
  end Divides;

  function Even(A : Integer) return Boolean is
  begin
    return Divides(2, A);
  end Even;

  function Odd(A : Integer) return Boolean is
  begin
    return not Divides(2, A);
  end Odd;

  function Greater_Than_7(A : Integer) return Boolean is
  begin
    return A > 7;
  end Greater_Than_7;

  function Square(A : Integer) return Integer is
  begin
    return A * A;
  end Square;

end Examples_Help;--
```

Appendix B

Combining Stacks with a Vector Representation

The Stacks and Output_Restricted_Deques packages can be combined with low-level representations other than linked lists, since the generic parameters of these packages do not need all of the characteristics of linked-lists (in particular, no Set_Next operation is needed). In order to give a concrete illustration of this point, we show a simple representation of vectors that supplies the operations needed for instantiation of Stacks. (A later volume will give more extensive vectors packages that will be documented in the same manner as the linked list packages.)

B.1 Simple_Indexed_Vectors Package Specification

From file sivects.ada--

```
generic

  Max_Size : in Natural;
  type Element is private;

package Simple_Indexed_Vectors is

  type Sequence is private;
  procedure Create(S : in out Sequence);
  function Full(S : Sequence) return Boolean;
  function Empty(S : Sequence) return Boolean;
  function First(S : Sequence) return Element;
  function Next(S : Sequence) return Sequence;
  function Construct(E : Element; S : Sequence) return Sequence;
  procedure Free_Construct(S : Sequence);

private

  type Node;
  type Sequence is access Node;
```

```
end Simple_Indexed_Vectors;--
```

B.2 Simple_Indexed_Vectors Package Body

From file sivectb.ada--

```
package body Simple_Indexed_Vectors is

type Storage is array(Integer range 1 .. Max_Size) of Element;

type Node is record
    Vector_Field : Storage;
    Index_Field  : Integer range 0 .. Max_Size := 0;
  end record;

procedure Create(S : in out Sequence) is
begin
  S := new Node;
end Create;

function Full(S : Sequence) return Boolean is
begin
  return (S.Index_Field = Max_Size);
end Full;

function Empty(S : Sequence) return Boolean is
begin
  return (S.Index_Field = 0);
end Empty;

function First(S : Sequence) return Element is
begin
  return S.Vector_Field(S.Index_Field);
end First;

function Next(S : Sequence) return Sequence is
begin
  S.Index_Field := S.Index_Field - 1;
  return S;
end Next;

function Construct(E : Element; S : Sequence) return Sequence is
begin
  S.Index_Field := S.Index_Field + 1;
  S.Vector_Field(S.Index_Field) := E;
  return S;
end Construct;
```

```
procedure Free_Construct(S : Sequence) is
begin
  null;
end Free_Construct;

end Simple_Indexed_Vectors;--
```

B.3 A PIP Combining Vectors and Stacks

From file stackp4.ada--

```
with Simple_Indexed_Vectors, Stacks;
generic
  Max_Size : in Natural;
  type Element is private;
package Stacks_4 is

  package Low_Level is new Simple_Indexed_Vectors(Max_Size, Element);
  use Low_Level;

  package Inner is new Stacks(Element, Sequence, Create, Full,
    Empty, First, Next, Construct, Free_Construct);

end Stacks_4;--
```

Appendix C

Orderings for Merge and Sort

This appendix is reproduced from a section in Volume 1.

A precise description of the kind of function that can be used for comparing values when using the `Merge` and `Sort` subprograms in the `Double_Ended_Lists` package 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.)

D.R. Musser
A.A. Stepanov

ADA GENERIC LIBRARY
LINEAR DATA STRUCTURE PACKAGES, VOLUME TWO

88CRD113
April 1988