A

Extracts from the Base libraries

Throughout the discussion, we have encountered references to a set of libraries collectively known as the "Base libraries", from which the most fundamental classes are grouped into the "Kernel library".

Reading such classes is a good way to learn more about the method by benefiting from the example of widely reused software components, which have been around for a long time and continue to evolve.

This page and the next are only the introduction to the appendix; the actual class texts, made available in electronic form so as to facilitate browsing, appear only on the CD-ROM version of this book **(starting next page)**.

See "Criteria for view inheritance", page 856.

A detailed presentation of the libraries has been published separately [M 1994a], which also describes the theoretical underpinnings — the general taxonomy principles used to classify the major data structures of computing science. A few of the basic ideas were summarized in the discussion of view inheritance.

Among the most important classes whose concepts were discussed in the previous chapters and whose text you will find on the following pages on the CD-ROM are:

- *ARRAY*, describing one-dimensional arrays and relying on a flexible and general view of this notion (in particular, arrays can be freely resized to any dimension during the execution of a system).
- *LINKABLE*, describing cells of linked structures, chained one-way to similar cells.
- *BI_LINKABLE*, the equivalent for two-way linked cells.
- *LIST*, a deferred class representing the general notion of list as "active data structure" with cursor, without commitment to a particular representation. (The next three classes provide specific implementations, using multiple inheritance through the "marriage of convenience" technique.)
- *ARRAYED_LIST*, giving an implementation by an array (whose resizability of is particularly useful here).
- *LINKED_LIST*, a one-way linked list implementation, relying internally on class *LINKABLE*.

- *TWO_WAY_LIST*, a one-way linked list implementation, relying internally on class *BI_LINKABLE*.
- *TWO_WAY_TREE*, a widely used implementation of general trees, based on *TWO_WAY_LIST* for its representation and relying on the observation made on the chapter on multiple inheritance: if we merge the notion of tree and node, we can consider that a tree is both a list (as in *TWO_WAY_LIST*) and a list element (as in *BI_LINKABLE*).

All these classes, representing containers, are generic, with a single generic parameter representing the type of elements.

The classes are given "as is", without further formating. Note that the following page numbers are of the form 1266.1, 1266.2 etc. to avoid any confusion with the numbering of the pages in the printed book.

A.1 ARRAYS

```
indexing


    description:
        "Sequences of values, all of the same type or of a conforming one, %
        %accessible through integer indices in a contiguous interval";


    status: "See notice at end of class";
    date: "$Date: 1996/06/05 14:19:05 $";
    revision: "$Revision: 1.28 $"


class ARRAY [G] inherit


    RESIZABLE [G]
        redefine
            full, copy, is_equal,
            consistent, setup
        end;


    INDEXABLE [G, INTEGER]
        redefine
```

```
                copy, is_equal,
                consistent, setup
            end;

        TO_SPECIAL [G]
            export
                {ARRAY} set_area
            redefine
                copy, is_equal,
                consistent, setup
            end

    creation

        make

    feature -- Initialization

        make (minindex, maxindex: INTEGER) is
                -- Allocate array; set index interval to
                -- `minindex' .. `maxindex'; set all values to default.
                -- (Make array empty if `minindex' = `maxindex' + 1).
            require
                valid_indices: minindex <= maxindex or (minindex = maxindex + 1)
            do
                lower := minindex;
                upper := maxindex;
                if minindex <= maxindex then
                    make_area (maxindex - minindex + 1)
                else
                    make_area (0)
            end;
        ensure
            lower = minindex;
            upper = maxindex
        end;

    make_from_array (a: ARRAY [G]) is
            -- Initialize from the items of `a'.
            -- (Useful in proper descendants of class `ARRAY',
            -- to initialize an array-like object from a manifest array.)
        require
            array_exists: a /= Void
        do
            area := a.area;
            lower := a.lower;
            upper := a.upper
        end;

    setup (other: like Current) is
            -- Perform actions on a freshly created object so that
            -- the contents of `other' can be safely copied onto it.
        do
            make_area (other.capacity)
        end;

feature -- Access

    frozen item, frozen infix "@", entry (i: INTEGER): G is
            -- Entry at index `i', if in index interval
        do
            Result := area.item (i - lower);
        end;
        has (v: G): BOOLEAN is
                -- Does `v' appear in array?
                -- (Reference or object equality,
                -- based on `object_comparison'.)
            local
                i: INTEGER
            do
                if object_comparison then
                    if v = void then
                        i := upper + 1
                    else
                        from
                            i := lower
                        until
                            i > upper or else (item (i) /= Void and then item (i).
    is_equal(v))
                        loop
                            i := i + 1;
                        end;
                    end
                else
                    from
                        i := lower
                    until
                        i > upper or else (item (i) = v)
                    loop
                        i := i + 1;
                    end;
                end
                Result := not (i > upper);
            end;
```

```
feature -- Measurement

    lower: INTEGER;
            -- Minimum index

    upper: INTEGER;
            -- Maximum index

    count, capacity: INTEGER is
            -- Number of available indices
        do
            Result := upper - lower + 1
        end;

    occurrences (v: G): INTEGER is
            -- Number of times `v' appears in structure
        local
            i: INTEGER
        do
            if object_comparison then
                if v /= Void then
                    from
                        i := lower
                    until
                        i > upper
                    loop
                        if item (i) /= Void and then v.is_equal (item (i))
then
                            Result := Result + 1
                        end
                        i := i + 1
```

```
                                        end
                                end
                        else
                                from
                                        i := lower
                                until
                                        i > upper
                                loop
                                        if item (i) = v then
                                                Result := Result +1
                                        end;
                                        i := i + 1
                                end
                        end;
                end;

feature -- Comparison

        is_equal (other: like Current): BOOLEAN is
                        -- Is array made of the same items as `other'?
                do
                        Result := area.is_equal (other.area)
                end;

feature -- Status report

        consistent (other: like Current): BOOLEAN is
                                -- Is object in a consistent state so that `other'
                                -- may be copied onto it? (Default answer: yes).
                do
                        Result := (capacity = other.capacity)
                end;
```

```
full: BOOLEAN is
            -- Is structure filled to capacity? (Answer: yes)
      do
            Result := true
      end;

all_cleared: BOOLEAN is
            -- Are all items set to default values?
      local
            i: INTEGER;
            dead_element: G;
      do
            from
                  i := lower
            variant
                  upper + 1 - i
            until
                  (i > upper) or else not (dead_element = item (i))
            loop
                  i := i + 1
            end;
            Result := i > upper;
      end;

valid_index (i: INTEGER): BOOLEAN is
            -- Is `i' within the bounds of the array?
      do
            Result := (lower <= i) and then (i <= upper)
      end;

extendible: BOOLEAN is
```

```
                -- May items be added?
                -- (Answer: no, although array may be resized.)
            do
                Result := false
            end;

        prunable: BOOLEAN is
                -- May items be removed? (Answer: no.)
            do
                Result := false
            end;

feature -- Element change

        frozen put, enter (v: like item; i: INTEGER) is
                -- Replace `i'-th entry, if in index interval, by `v'.
            do
                area.put (v, i - lower);
            end;

        force (v: like item; i: INTEGER) is
                -- Assign item `v' to `i'-th entry.
                -- Always applicable: resize the array if `i' falls out of
                -- currently defined bounds; preserve existing items.
            do
                if i < lower then
                    auto_resize (i, upper);
                elseif i > upper then
                    auto_resize (lower, i);
                end;
                put (v, i)
            ensure
```

```
                inserted: item (i) = v;
                higher_count: count >= old count
        end;


    subcopy (other: like Current; start_pos, end_pos, index_pos: INTEGER) is
                -- Copy items of `other’ within bounds `start_pos’ and `end_pos’
                -- to current array starting at index `index_pos’.
        require
                other_not_void: other /= Void;
                valid_start_pos: other.valid_index (start_pos)
                valid_end_pos: other.valid_index (end_pos)
                valid_bounds: (start_pos <= end_pos) or (start_pos = end_pos + 1)
                valid_index_pos: valid_index (index_pos)
                enough_space: (upper - index_pos) >= (end_pos - start_pos)
        local
                other_area: like area;
                other_lower: INTEGER;
                start0, end0, index0: INTEGER
        do
                other_area := other.area;
                other_lower := other.lower;
                start0 := start_pos - other_lower;
                end0 := end_pos - other_lower;
                index0 := index_pos - lower;
                spsubcopy ($other_area, $area, start0, end0, index0)
        ensure
                -- copied: forall `i’ in 0 .. (`end_pos’-`start_pos’),
                --     item (index_pos + i) = other.item (start_pos + i)
        end


feature -- Removal
```

```
        wipe_out is
                -- Make array empty.
            do
                make_area (capacity)
            end;

        clear_all is
                -- Reset all items to default values.
            do
                spclearall ($area)
            ensure
                all_cleared: all_cleared
            end;

    feature -- Resizing

        grow (i: INTEGER) is
                -- Change the capacity to at least `i'.
            do
                if i > capacity then
                    resize (lower, upper + i - capacity)
                end
            end;

        resize (minindex, maxindex: INTEGER) is
                -- Rearrange array so that it can accommodate
                -- indices down to `minindex' and up to `maxindex'.
                -- Do not lose any previously entered item.
            require
                good_indices: minindex <= maxindex
            local
                old_size, new_size, old_count: INTEGER;
        new_lower, new_upper: INTEGER;
do
        if empty_area then
                new_lower := minindex;
                new_upper := maxindex
        else
                if minindex < lower then
                        new_lower := minindex
                else
                        new_lower := lower
                end;
                if maxindex > upper then
                        new_upper := maxindex
                else
                        new_upper := upper
                end
        end;
        new_size := new_upper - new_lower + 1;
        if not empty_area then
                old_size := area.count;
                old_count := upper - lower + 1
        end;
        if empty_area then
                make_area (new_size);
        elseif new_size > old_size or new_lower < lower then
                area := arycpy ($area, new_size,
                        lower - new_lower, old_count)
        end;
        lower := new_lower;
        upper := new_upper
ensure
        no_low_lost: lower = minindex.min (old lower);
```

```
         no_high_lost: upper = maxindex.max (old upper)
      end;

feature -- Conversion

   to_c: ANY is
         -- Address of actual sequence of values,
         -- for passing to external (non-Eiffel) routines.
      do
         Result := area
      end;

   linear_representation: LINEAR [G] is
         -- Representation as a linear structure
      local
         temp: ARRAYED_LIST [G];
         i: INTEGER;
      do
         !! temp.make (capacity);
         from
            i := lower;
         until
            i > upper
         loop
            temp.extend (item (i));
            i := i + 1;
         end;
         Result := temp;
      end;

feature -- Duplication
```

```
	copy (other: like Current) is
			-- Reinitialize by copying all the items of `other'.
			-- (This is also used by `clone'.)
		do
			standard_copy (other);
			set_area (standard_clone (other.area));
		ensure then
			equal_areas: area.is_equal (other.area)
		end;

	subarray (start_pos, end_pos: INTEGER): like Current is
			-- Array made of items of current array within
			-- bounds `start_pos' and `end_pos'.
		require
			valid_start_pos: valid_index (start_pos)
			valid_end_pos: valid_index (end_pos)
			valid_bounds: (start_pos <= end_pos) or (start_pos = end_pos + 1)
		do
			!! Result.make (start_pos, end_pos);
			Result.subcopy (Current, start_pos, end_pos, start_pos)
		ensure
			lower: Result.lower = start_pos;
			upper: Result.upper = end_pos;
			-- copied: forall `i' in `start_pos' .. `end_pos',
			--     Result.item (i) = item (i)
		end

feature -- Obsolete

	duplicate: like Current is obsolete "Use ``clone''"
		do
			Result := clone (Current)
```

```
            end;

feature {NONE} -- Inapplicable

    prune (v: G) is
                -- Remove first occurrence of `v’ if any.
                -- (Precondition is false.)
            do
            end;

    extend (v: G) is
                -- Add `v’ to structure.
                -- (Precondition is false.)
            do
            end;

feature {ARRAY} -- Implementation

    arycpy (old_area: POINTER; newsize, s, n: INTEGER): like area is
                -- New area of size `newsize’ containing `n’ items
                -- from `oldarea’.
                -- Old items are at position `s’ in new area.
            external
                “C”
            end;

feature {NONE} -- Implementation

    auto_resize (minindex, maxindex: INTEGER) is
                -- Rearrange array so that it can accommodate
                -- indices down to `minindex’ and up to `maxindex’.
                -- Do not lose any previously entered item.
```

```
        -- If area must be extended, ensure that space for at least
        -- additional_space item is added.
    require
        valid_indices: minindex <= maxindex
    local
        old_size, new_size: INTEGER;
        new_lower, new_upper: INTEGER;
    do
        if empty_area then
            new_lower := minindex;
            new_upper := maxindex
        else
            if minindex < lower then
                new_lower := minindex
            else
                new_lower := lower
            end;
            if maxindex > upper then
                new_upper := maxindex
            else
                new_upper := upper
            end
        end;
        new_size := new_upper - new_lower + 1;
        if not empty_area then
            old_size := area.count;
            if new_size > old_size
                 and new_size - old_size < additional_space
            then
                new_size := old_size + additional_space
            end
        end;
```

```
                if empty_area then
                    make_area (new_size);
                elseif new_size > old_size or new_lower < lower then
                        area := arycpy ($area, new_size,
                            lower - new_lower, capacity)
                end;
                lower := new_lower;
                upper := new_upper
            end;

        empty_area: BOOLEAN is
            do
                Result := area = Void or else area.count = 0
            end;

        spsubcopy (source, target: POINTER; s, e, i: INTEGER) is
                -- Copy elements of `source' within bounds `s'
                -- and `e' to `target' starting at index `i'.
            external
                "C"
            end

        spclearall (p: POINTER) is
                -- Reset all items to default value.
            external
                "C"
            end

    invariant

        consistent_size: count = upper - lower + 1;
        non_negative_count: count >= 0
    end -- class ARRAY



    --|---------------------------------------------------------------
    --| EiffelBase: library of reusable components for ISE Eiffel 3.
    --| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
    --|   Engineering Inc.
    --| All rights reserved. Duplication and distribution prohibited.
    --|
    --| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
    --| Telephone 805-685-1006
    --| Fax 805-685-6869
    --| Electronic mail <info@eiffel.com>
    --| Customer support e-mail <support@eiffel.com>
    --|---------------------------------------------------------------
```

A.2 LINKABLE AND BI-LINKABLE ELEMENTS

```
indexing

        description:
                "Linkable cells containing a reference to their right neighbor";

        status: "See notice at end of class";
        names: linkable, cell;
        representation: linked;
        contents: generic;
        date: "$Date: 1995/07/26 00:54:01 $";
        revision: "$Revision: 1.6 $"

    class LINKABLE [G] inherit
    CELL [G]
        export
            {CELL, CHAIN}
                put;
            {ANY}
                item
        end

feature -- Access

    right: like Current;
            -- Right neighbor

feature {CELL, CHAIN} -- Implementation

    put_right (other: like Current) is
            -- Put `other’ to the right of current cell.
        do
            right := other
        ensure
            chained: right = other
        end;

    forget_right is
            -- Remove right link.
        do
            right := Void
        ensure
            not_chained: right = Void
        end;

end -- class LINKABLE
```

```
    --|----------------------------------------------------------------
    --| EiffelBase: library of reusable components for ISE Eiffel 3.
    --| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
    --|   Engineering Inc.
    --| All rights reserved. Duplication and distribution prohibited.
    --|
    --| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
    --| Telephone 805-685-1006
    --| Fax 805-685-6869
    --| Electronic mail <info@eiffel.com>
    --| Customer support e-mail <support@eiffel.com>
    --|----------------------------------------------------------------


indexing

        description:
            "Linkable cells with a reference to the left and right neighbors";

        status: "See notice at end of class";
        names: bi_linkable, cell;
        representation: linked;
        contents: generic;
        date: "$Date: 1995/07/26 00:53:49 $";
        revision: "$Revision: 1.7 $"

    class BI_LINKABLE [G] inherit

        LINKABLE [G]
        redefine
            put_right, forget_right
        end

feature -- Access

    left: like Current;
            -- Left neighbor

feature {CELL, CHAIN} -- Implementation

    put_right (other: like Current) is
            -- Put `other' to the right of current cell.
        do
            if right /= Void then
                right.simple_forget_left
            end;
            right := other;
            if (other /= Void) then
                other.simple_put_left (Current)
            end
        end;

    put_left (other: like Current) is
            -- Put `other' to the left of current cell.
        do
            if left /= Void then
                left.simple_forget_right
            end;
            left := other;
            if (other /= Void) then
                other.simple_put_right (Current)
            end
        ensure
            chained: left = other
        end;

    forget_right is
            -- Remove links with right neighbor.
        do
            if right /= Void then
                right.simple_forget_left;
                right := Void
            end
        ensure then
            right_not_chained:
                (old right /= Void) implies ((old right).left = Void)
        end;

    forget_left is
            -- Remove links with left neighbor.
        do
            if left /= Void then
                left.simple_forget_right;
                left := Void
            end
        ensure
            left_not_chained:
                left = Void;
                (old left /= Void) implies ((old left).right = Void)
        end;

feature {BI_LINKABLE, TWO_WAY_LIST} -- Implementation
```

```
simple_put_right (other: like Current) is
        -- set `right’ to `other’
    do
        if right /= Void then
            right.simple_forget_left;
        end;
        right := other
    end;

simple_put_left (other: like Current) is
        -- set `left’ to `other’ is
    do
        if left /= Void then
            left.simple_forget_right
        end;
        left := other
    end;

simple_forget_right is
        -- Remove right link (do nothing to right neighbor).
    do
        right := Void
    end;

simple_forget_left is
        -- Remove left link (do nothing to left neighbor).
    do
        left := Void
    ensure
        not_chained: left = Void
    end;
    invariant

        right_symmetry:
            (right /= Void) implies (right.left = Current);
        left_symmetry:
            (left /= Void) implies (left.right = Current)

    end -- class BI_LINKABLE



    --|----------------------------------------------------------------
    --| EiffelBase: library of reusable components for ISE Eiffel 3.
    --| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
    --|   Engineering Inc.
    --| All rights reserved. Duplication and distribution prohibited.
    --|
    --| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
    --| Telephone 805-685-1006
    --| Fax 805-685-6869
    --| Electronic mail <info@eiffel.com>
    --| Customer support e-mail <support@eiffel.com>
    --|----------------------------------------------------------------
```

A.3 LISTS

```
indexing

        description:
            "Sequential lists, without commitment to a particular representation";

        status: "See notice at end of class";
        names: list, sequence;
        access: index, cursor, membership;
        contents: generic;
        date: “$Date: 1995/07/26 00:54:06 $”;
        revision: “$Revision: 1.9 $”

deferred class LIST [G] inherit

        CHAIN [G]
                redefine
                        forth
                end;
feature -- Cursor movement

        forth is
                        -- Move to next position; if no next position,
                        -- ensure that `exhausted’ will be true.
                deferred
                ensure then
                        moved_forth: index = old index + 1
                end;

feature -- Status report

        after: BOOLEAN is
                        -- Is there no valid cursor position to the right of cursor?
                do
                        Result := (index = count + 1)
                end;

        before: BOOLEAN is
                        -- Is there no valid cursor position to the left of cursor?
                do
```

```
            Result := (index = 0)
        end;

feature -- Obsolete

    offleft: BOOLEAN is obsolete “Use ``before’’”
        do
            Result := before or empty
        end;

    offright: BOOLEAN is obsolete “Use ``after’’”
        do
            Result := after or empty
        end;

invariant

    before_definition: before = (index = 0);
    after_definition: after = (index = count + 1);

end -- class LIST


--|----------------------------------------------------------------
--| EiffelBase: library of reusable components for ISE Eiffel 3.
--| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
--|   Engineering Inc.
--| All rights reserved. Duplication and distribution prohibited.
--|
--| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
--| Telephone 805-685-1006
--| Fax 805-685-6869
```

```
    --| Electronic mail <info@eiffel.com>
    --| Customer support e-mail <support@eiffel.com>
    --|--------------------------------------------------------------
```

A.4 ARRAYED LISTS

```
indexing

        description:
            “Lists implemented by resizable arrays”;

        status: “See notice at end of class”;
        names: sequence;
        representation: array;
        access: index, cursor, membership;
        size: fixed;
        contents: generic;
        date: “$Date: 1995/10/30 16:55:22 $”;
        revision: “$Revision: 1.20 $”

    class ARRAYED_LIST [G] inherit

        ARRAY [G]
            rename
                duplicate as array_duplicate,
                force as force_i_th,
                item as i_th,
                make as array_make,
                put as put_i_th,
                wipe_out as array_wipe_out,
                count as array_count,
                bag_put as put
    export
        {NONE}
            all;
        {ARRAYED_LIST}
            array_make;
        {ANY}
            capacity
    undefine
        linear_representation, prunable, put,
        prune, consistent, is_equal, occurrences,
        extendible, has
    redefine
        extend, setup, copy, prune_all, full, valid_index
    end;

ARRAY [G]
    rename
        duplicate as array_duplicate,
        force as force_i_th,
        item as i_th,
        make as array_make,
        put as put_i_th,
        count as array_count,
        bag_put as put
    export
        {NONE}
            all;
        {ARRAYED_LIST}
            array_make;
        {ANY}
            capacity
    undefine
```

```
            linear_representation, prunable, full, put,
            prune, consistent, is_equal, occurrences,
            extendible, has
        redefine
            wipe_out, extend,
            setup, copy, prune_all, valid_index
        select
            wipe_out
        end;

    DYNAMIC_LIST [G]
        undefine
            valid_index, infix "@", i_th, put_i_th,
            force
        redefine
            first, last, swap, wipe_out,
            go_i_th, move, prunable, start, finish,
            count, prune, remove,
            setup, copy, put_left, merge_left,
            merge_right, duplicate, prune_all
        select
            count
        end;

creation

    make, make_filled

feature -- Initialization

    make (n: INTEGER) is
            -- Allocate list with `n' items.
```

```
                    -- (`n’ may be zero for empty list.)
          require
                    valid_number_of_items: n >= 0
          do
                    array_make (1, n)
          ensure
                    correct_position: before
          end;

     make_filled (n: INTEGER) is
                    -- Allocate list with `n’ items.
                    -- (`n’ may be zero for empty list.)
                    -- This list will be full.
          require
                    valid_number_of_items: n >= 0
          do
                    array_make (1, n)
                    count := n
          ensure
                    correct_position: before
                    filled: full
          end;

feature -- Access

     item: like first is
                    -- Current item
          require else
                    index_is_valid: valid_index (index)
          do
                    Result := area.item (index - 1);
          end;
          first: G is
                    -- Item at first position
               do
                    Result := area.item (0);
               end;

          last: like first is
                    -- Item at last position
               do
                    Result := area.item (count - 1)
               end;

          index: INTEGER;
                    -- Index of `item', if valid.

          cursor: CURSOR is
                    -- Current cursor position
               do
                    !ARRAYED_LIST_CURSOR! Result.make (index)
               end;

     feature -- Measurement

          count: INTEGER;
               -- Number of items.

     feature -- Status report

          prunable: BOOLEAN is
                    -- May items be removed? (Answer: yes.)
               do
```

```
                    Result := true
          end;

     full: BOOLEAN is
                    -- Is structure filled to capacity? (Answer: no.)
          do
                    Result := (count = capacity)
          end;

     valid_cursor (p: CURSOR): BOOLEAN is
                    -- Can the cursor be moved to position `p'?
          local
                    al_c: ARRAYED_LIST_CURSOR
          do
                    al_c ?= p;
                    if al_c /= Void then
                              Result := valid_cursor_index (al_c.index)
                    end
          end;

     valid_index (i: INTEGER): BOOLEAN is
                    -- Is `i' a valid index?
          do
                    Result := (1 <= i) and (i <= count)
          end

feature -- Cursor movement

     move (i: INTEGER) is
                    -- Move cursor `i' positions.
          do
                    index := index + i;
```

```
        if (index > count + 1) then
            index := count + 1
        elseif (index < 0) then
            index := 0
        end
    end;

start is
        -- Move cursor to first position if any.
    do
        index := 1
    ensure then
        after_when_empty: empty implies after
    end;

finish is
        -- Move cursor to last position if any.
    do
        index := count
    --| Temporary patch. Start moves the cursor
    --| to the first element. If the list is empty
    --| the cursor is before. The parents (CHAIN, LIST...)
    --| and decendants (ARRAYED_TREE...) need to be revised.
    ensure then
        before_when_empty: empty implies before
    end;

forth is
        -- Move cursor one position forward.
    do
        index := index + 1
    end;
```

```
        back is
                    -- Move cursor one position backward.
            do
                    index := index - 1
            end;

        go_i_th (i: INTEGER) is
                    -- Move cursor to `i'-th position.
            do
                    index := i;
            end;

        go_to (p: CURSOR) is
                    -- Move cursor to position `p'.
            local
                    al_c: ARRAYED_LIST_CURSOR
            do
                    al_c ?= p;
                        check
                            al_c /= Void
                        end;
                    index := al_c.index
            end;

    feature -- Transformation

        swap (i: INTEGER) is
                    -- Exchange item at `i'-th position with item
                    -- at cursor position.
            local
                    old_item: like item
            do
                old_item := item;
                replace (area.item (i - 1));
                area.put (old_item, i - 1);
            end;

feature -- Element change

        put_front (v: like item) is
                -- Add `v’ to the beginning.
                -- Do not move cursor.
            do
                if empty then
                    extend (v)
                else
                    insert (v, 1)
                end;
            end;

        force, extend (v: like item) is
                -- Add `v’ to end.
                -- Do not move cursor.
            do
                count := count + 1;
                force_i_th (v, count)
            end;

        put_left (v: like item) is
                -- Add `v’ to the left of current position.
                -- Do not move cursor.
            do
                if after or empty then
                extend (v);
                index := index + 1
            else
                insert (v, index)
            end
        end;

    put_right (v: like item) is
            -- Add `v' to the right of current position.
            -- Do not move cursor.
        do
            if index = count then
                extend (v)
            else
                insert (v, index + 1)
            end;
        end;



    replace (v: like first) is
            -- Replace current item by `v'.
        do
            put_i_th (v, index)
        end;

    merge_left (other: ARRAYED_LIST [G]) is
        local
            i, l_count: INTEGER;
        do
            if not other.empty then
                resize (1, count + other.count);
                from
                i := count - 1;
                l_count := other.count
            until
                i < index - 1
            loop
                area.put (area.item (i), i + l_count);
                i := i - 1
            end;
            from
                other.start;
                i := index - 1
            until
                other.after
            loop
                area.put (other.item, i);
                i := i + 1;
                other.forth
            end;
            index := index + l_count;
            count := count + l_count;
            other.wipe_out
        end
    end;

merge_right (other: ARRAYED_LIST [G]) is
    local
        old_index: INTEGER;
    do
        old_index := index;
        index := index + 1;
        merge_left (other);
        index := old_index
```

```
            end;

    feature -- Removal

        prune (v: like item) is
                -- Remove first occurrence of `v', if any,
                -- after cursor position.
                -- Move cursor to right neighbor
                -- (or `after' if no right neighbor or `v'does not occur)
            do
                if before then index := 1 end;
                if object_comparison then
                    if v /= Void then
                        from
                        until
                            after or else (item /= Void and then v.is_equal
(item))
                        loop
                            forth;
                        end
                    end
                else
                    from
                    until
                        after or else item = v
                    loop
                        forth;
                    end
                end;
                if not after then remove end;
            end;
```

```
remove is
        -- Remove current item.
        -- Move cursor to right neighbor
        -- (or `after’ if no right neighbor)
    local
        i,j: INTEGER;
        default_value: G;
        l_count: INTEGER
    do
        if not off then
            from
                i := index - 1;
                l_count := count - 1
            until
                i >= l_count
            loop
                j := i + 1;
                area.put (area.item (j), i);
                i := j;
            end;
            put_i_th (default_value, count);
            count := count -1;
        end
    end;

prune_all (v: like item) is
        -- Remove all occurrences of `v’.
        -- (Reference or object equality,
        -- based on `object_comparison’.)
        -- Leave cursor `after’.
    local
        i: INTEGER;
```

```
                val, default_value: like item;
            do
                if object_comparison then
                    if v /= void then
                        from
                            start
                        until
                            after or else (item /= Void and then v.is_equal
(item))
                        loop
                            index := index + 1;
                        end;
                        from
                            if not after then
                                i := index;
                                index := index + 1
                            end
                        until
                            after
                        loop
                            val := item;
                            if val /= Void and then not v.is_equal (val) then
                                put_i_th (val, i);
                                i := i + 1
                            end;
                            index := index + 1;
                        end
                    end
                else
                    from
                        start
                    until
            after or else (item = v)
        loop
            index := index + 1;
        end;
        from
            if not after then
                i := index;
                index := index + 1
            end
        until
            after
        loop
            val := item;
            if val /= v then
                put_i_th (val, i);
                i := i + 1;
            end;
            index := index + 1
        end
    end;
    if i > 0 then
        index := i
        from
        until
            i >= count
        loop
            put_i_th (default_value, i);
            i := i + 1;
        end;
        count := index - 1;
    end
ensure then
```

```
                    is_after: after;
          end;

     remove_left is
                    -- Remove item to the left of cursor position.
                    -- Do not move cursor.
          do
                    index := index - 1;
                    remove;
          end;

     remove_right is
                    -- Remove item to the right of cursor position
                    -- Do not move cursor
          do
                    index := index + 1;
                    remove;
                    index := index - 1;
          end;

     wipe_out is
                    -- Remove all items.
          do
                    count := 0;
                    index := 0;
                    array_wipe_out;
          end;

feature -- Duplication

     setup (other: like Current) is
                    -- Prepare current object so that `other'
```

```
            -- can be easily copied into it.
            -- It is not necessary to call `setup'
            -- (since `consistent' is always true)
            -- but it will make copying quicker.
        do
            if other.empty then
                wipe_out
            else
                resize (1, other.count)
            end
        end;

    copy (other: like Current) is
        local
            c: like cursor;
        do
            count := 0;
            c := other.cursor;
            from
                other.start
            until
                other.after
            loop
                extend (other.item);
                other.forth
            end;
            other.go_to (c);
        end;

    duplicate (n: INTEGER): like Current is
            -- Copy of sub-list beginning at current position
            -- and having min (`n', `count' - `index' + 1) items.
        local
            pos: INTEGER
        do
            !! Result.make (n.min (count - index + 1));
            from
                Result.start;
                pos := index
            until
                Result.count = Result.capacity
            loop
                Result.extend (item);
                forth;
            end;
            Result.start;
            go_i_th (pos);
        end;


feature {NONE} --Internal

    insert (v: like item; pos: INTEGER) is
            -- Add `v' at `pos', moving subsequent items
            -- to the right.
        require
            index_small_enough: pos <= count;
            index_large_enough: pos >= 1;
        local
            i,j: INTEGER;
            p : INTEGER;
            last_value: like item;
            last_item: like item;
        do
```

```
                        if index >= pos then
                                index := index + 1
                        end;
                        last_item := last;
                        count := count + 1;
                        force_i_th (last_item, count);
                        from
                                i := count - 2
                        until
                                i < pos
                        loop
                                j := i - 1;
                                area.put (area.item (j), i);
                                i := j;
                        end;
                        put_i_th (v, pos);
                ensure
                        new_count: count = old count + 1;
                        insertion_done: i_th (pos) = v
                end;

        new_chain: like Current is
                        -- unused
                do
                end;

invariant

        prunable: prunable;

end -- class ARRAYED_LIST
```

```
    --|--------------------------------------------------------------
    --| EiffelBase: library of reusable components for ISE Eiffel 3.
    --| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
    --|   Engineering Inc.
    --| All rights reserved. Duplication and distribution prohibited.
    --|
    --| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
    --| Telephone 805-685-1006
    --| Fax 805-685-6869
    --| Electronic mail <info@eiffel.com>
    --| Customer support e-mail <support@eiffel.com>
    --|--------------------------------------------------------------
```

A.5 LINKED LISTS

```
indexing

        description:
            “Sequential, one-way linked lists”;

        status: “See notice at end of class”;
        names: linked_list, sequence;
        representation: linked;
        access: index, cursor, membership;
        contents: generic;
        date: “$Date: 1996/01/15 16:31:59 $”;
        revision: “$Revision: 1.17 $”

    class LINKED_LIST [G] inherit

        DYNAMIC_LIST [G]
            redefine
```

```
                    go_i_th, put_left, move, wipe_out,
                    isfirst, islast,
                    first, last, finish, merge_left, merge_right,
                    readable, start, before, after, off
                end

creation

        make

feature -- Initialization

        make is
                    -- Create an empty list.
                do
                    before := true
                ensure
                    is_before: before;
                end;

feature -- Access

        item: G is
                    -- Current item
                do
                    Result := active.item
                end;

        first: like item is
                    -- Item at first position
                do
                    Result := first_element.item
```

```
    end;

last: like item is
        -- Item at last position
    do
        Result := last_element.item
    end;

index: INTEGER is
        -- Index of current position
    local
        p: LINKED_LIST_CURSOR [G]
    do
        if after then
            Result := count + 1
        elseif not before then
            p ?= cursor;
                    check p /= Void end;
            from
                start; Result := 1
            until
                p.active = active
            loop
                forth
                Result := Result + 1
            end;
            go_to (p)
        end
    end;

cursor: CURSOR is
        -- Current cursor position
```

```
                do
                        !LINKED_LIST_CURSOR [G]! Result.make (active, after, before)
                end;

        feature -- Measurement

            count: INTEGER;
                        -- Number of items

        feature -- Status report

            readable: BOOLEAN is
                        -- Is there a current item that may be read?
                do
                        Result := not off
                end;

            after: BOOLEAN;
                        -- Is there no valid cursor position to the right of cursor?

            before: BOOLEAN;
                        -- Is there no valid cursor position to the left of cursor?

            off: BOOLEAN is
                        -- Is there no current item?
                do
                        Result := after or before
                end;

            isfirst: BOOLEAN is
                        -- Is cursor at first position?
                do
```

```
				Result := not after and not before and (active = first_element)
			end;

		islast: BOOLEAN is
				-- Is cursor at last position?
			do
				Result := not after and not before and
							(active /= Void) and then (active.right = Void)
			end;

		valid_cursor (p: CURSOR): BOOLEAN is
				-- Can the cursor be moved to position `p'?
			local
				ll_c: LINKED_LIST_CURSOR [G];
				temp, sought: like first_element
			do
				ll_c ?= p;
				if ll_c /= Void then
					from
						temp := first_element;
						sought := ll_c.active;
						Result := ll_c.after or else ll_c.before
					until
						Result or else temp = Void
					loop
						Result := (temp = sought);
						temp := temp.right
					end;
				end
			end;

		full: BOOLEAN is false;
```

```
                        -- Is structured filled to capacity? (Answer: no.)

            feature -- Cursor movement

                start is
                            -- Move cursor to first position.
                        do
                            if first_element /= Void then
                                active := first_element;
                                after := false
                            else
                                after := true
                            end;
                            before := false
                        ensure then
                            empty_convention: empty implies after
                        end;

                finish is
                            -- Move cursor to last position.
                            -- (Go before if empty)
                        local
                            p: like first_element
                        do
                            if not empty then
                                from
                                    p := active
                                until
                                    p.right = Void
                                loop
                                    p := p.right
                                end;
            active := p;
            after := false;
            before := false
        else
            before := true;
            after := false
        end;
    ensure then
        Empty_convention: empty implies before
    end;

forth is
        -- Move cursor to next position.
    local
        old_active: like first_element
    do
        if before then
            before := false;
            if empty then after := true end
        else
            old_active := active;
            active := active.right;
            if active = Void then
                active := old_active;
                after := true
            end
        end
    end;

back is
        -- Move to previous item.
    do
```

```
        if empty then
            before := true;
            after := false
        elseif after then
            after := false
        elseif isfirst then
            before := true
        else
            active := previous
        end
    end;

move (i: INTEGER) is
        -- Move cursor `i’ positions. The cursor
        -- may end up `off’ if the offset is too big.
    local
        counter, new_index: INTEGER;
        p: like first_element
    do
        if i > 0 then
            if before then
                before := false;
                counter := 1
            end;
            from
                p := active
            until
                (counter = i) or else (p = Void)
            loop
                active := p;
                p := p.right;
                counter := counter + 1
                end;
                if p = Void then
                    after := true
                else
                    active := p
                end
            elseif i < 0 then
                new_index := index + i;
                before := true;
                after := false;
                active := first_element;
                if (new_index > 0) then
                    move (new_index)
                end
            end
        ensure then
            moved_if_inbounds:
                ((old index + i) >= 0 and
                (old index + i) <= (count + 1))
                    implies index = (old index + i);
            before_set: (old index + i) <= 0 implies before;
            after_set: (old index + i) >= (count + 1) implies after
        end;

    go_i_th (i: INTEGER) is
            -- Move cursor to `i’-th position.
        do
            if i = 0 then
                before := true;
                after := false;
                active := first_element
            elseif i = count + 1 then
```

```
                before := false;
                after := true;
                active := last_element
            else
                move (i - index)
            end
        end;

    go_to (p: CURSOR) is
            -- Move cursor to position `p'.
        local
            ll_c: LINKED_LIST_CURSOR [G]
        do
            ll_c ?= p;
                check
                    ll_c /= Void
                end;
            after := ll_c.after;
            before := ll_c.before;
            if before then
                active := first_element
            elseif after then
                active := last_element
            else
                active := ll_c.active;
            end
        end;

feature -- Element change

    put_front (v: like item) is
            -- Add `v' to beginning.
        -- Do not move cursor.
    local
        p: like first_element
    do
        p := new_cell (v);
        p.put_right (first_element);
        first_element := p;
        if before or empty then
            active := p
        end;
        count := count + 1;
    end;

extend (v: like item) is
        -- Add `v’ to end.
        -- Do not move cursor.
    local
        p: like first_element
    do
        p := new_cell (v);
        if empty then
            first_element := p;
            active := p;
        else
            last_element.put_right (p);
            if after then active := p end
        end;
        count := count + 1
    end;

put_left (v: like item) is
        -- Add `v’ to the left of cursor position.
```

```
                        -- Do not move cursor.
                local
                        p: like first_element
                do
                        if empty then
                                put_front (v)
                        elseif after then
                                back;
                                put_right (v);
                                move (2)
                        else
                                p := new_cell (active.item);
                                p.put_right (active.right);
                                active.put (v);
                                active.put_right (p);
                                active := p;
                                count := count + 1
                        end
                ensure then
                        previous_exists: previous /= Void;
                        item_inserted: previous.item = v
                end;

        put_right (v: like item) is
                        -- Add `v’ to the right of cursor position.
                        -- Do not move cursor.
                local
                        p: like first_element;
                do
                        p := new_cell (v);
                        check empty implies before end;
                        if before then
```

```
            p.put_right (first_element);
            first_element := p;
            active := p;
        else
            p.put_right (active.right);
            active.put_right (p);
        end;
        count := count + 1
    ensure then
        next_exists: next /= Void;
        item_inserted: not old before implies next.item = v
        item_inserted_before: old before implies active.item = v
    end;

replace (v: like item) is
        -- Replace current item by `v'.
    do
        active.put (v)
    end;

merge_left (other: like Current) is
        -- Merge `other' into current structure before cursor
        -- position. Do not move cursor. Empty `other'.
    local
        other_first_element: like first_element;
        other_last_element: like first_element;
        p: like first_element;
        other_count: INTEGER
    do
        if not other.empty then
            other_first_element := other.first_element;
            other_last_element := other.last_element;
            other_count := other.count;
                check
                    other_first_element /= Void;
                    other_last_element /= Void
                end;
            if empty then
                first_element := other_first_element;
                active := first_element
            elseif isfirst then
                p := first_element;
                other_last_element.put_right (p);
                first_element := other_first_element
            else
                p := previous;
                if p /= Void then
                    p.put_right (other_first_element)
                end;
                other_last_element.put_right (active)
            end;
            count := count + other_count;
            other.wipe_out;
        end
    end;


merge_right (other: like Current) is
        -- Merge `other’ into current structure after cursor
        -- position. Do not move cursor. Empty `other’.
    local
        other_first_element: like first_element;
        other_last_element: like first_element;
        p: like first_element;
        other_count: INTEGER;
```

```
            do
                if not other.empty then
                    other_first_element := other.first_element;
                    other_last_element := other.last_element;
                    other_count := other.count;
                        check
                            other_first_element /= Void;
                            other_last_element /= Void
                        end;
                    if empty then
                        first_element := other_first_element;
                        active := first_element;
                    else
                        if not islast then
                            other_last_element.put_right (active.right);
                        end;
                        active.put_right (other_first_element);
                    end;
                    count := count + other_count;
                    other.wipe_out;
                end
            end;

    feature -- Removal

        remove is
                -- Remove current item.
                -- Move cursor to right neighbor
                -- (or `after’ if no right neighbor).
            local
                removed, succ: like first_element
            do
            removed := active;
            if isfirst then
                first_element := first_element.right;
                active.forget_right;
                active := first_element;
                if count = 1 then
                    check
                        no_active: active = Void
                    end;
                    after := true;
                end
            elseif islast then
                active := previous;
                if active /= Void then
                    active.forget_right
                end;
                after := true
            else
                succ := active.right;
                previous.put_right (succ);
                active.forget_right;
                active := succ
            end;
            count := count - 1;
            cleanup_after_remove (removed)
        end;

    remove_left is
            -- Remove item to the left of cursor position.
            -- Do not move cursor.
        do
            move (-2);
        remove_right;
        forth
    end;

remove_right is
        -- Remove item to the right of cursor position.
        -- Do not move cursor.
    local
        removed, succ: like first_element
    do
        if before then
            removed := first_element;
            first_element := first_element.right;
            active.forget_right;
            active := first_element
        else
            succ := active.right;
            removed := succ;
            active.put_right (succ.right);
            succ.forget_right
        end;
        count := count - 1;
        cleanup_after_remove (removed)
    end;

wipe_out is
        -- Remove all items.
    do
        active := Void;
        first_element := Void;
        before := true;
        after := false;
```

```
            count := 0
        end;

feature {LINKED_LIST} -- Implementation

    new_chain: like Current is
            -- A newly created instance of the same type.
            -- This feature may be redefined in descendants so as to
            -- produce an adequately allocated and initialized object.
        do
            !! Result.make
        end;

    new_cell (v: like item): like first_element is
            -- A newly created instance of the same type as `first_element'.
            -- This feature may be redefined in descendants so as to
            -- produce an adequately allocated and initialized object.
        do
            !! Result;
            Result.put (v)
        ensure
            result_exists: Result /= Void
        end;

    previous: like first_element is
            -- Element left of cursor
        local
            p: like first_element;
        do
            if after then
                Result := active
            elseif not (isfirst or before) then
```

```
                    from
                        p := first_element
                    until
                        p.right = active
                    loop
                        p := p.right
                    end;
                    Result := p;
                end
            end;

        next: like first_element is
                -- Element right of cursor
            do
                if before then
                    Result := active
                elseif active /= Void then
                    Result := active.right
                end
            end;


        active: like first_element;
                -- Element at cursor position

        first_element: LINKABLE [G];
                -- Head of list

        last_element: like first_element is
                -- Tail of list
            local
                p: like first_element
                do
                    if not empty then
                        from
                            Result := active;
                            p := active.right
                        until
                            p = Void
                        loop
                            Result := p;
                            p := p.right
                        end
                    end
                end;

            cleanup_after_remove (v: like first_element) is
                    -- Clean-up a just removed cell.
                require
                    non_void_cell: v /= Void
                do
                end;



        invariant


            prunable: prunable;
            empty_constraint: empty implies ((first_element = Void) and (active = Void));
            not_void_unless_empty: (active = Void) implies empty;
            before_constraint: before implies (active = first_element);
            after_constraint: after implies (active = last_element)



        end -- class LINKED_LIST
```

```
--|----------------------------------------------------------------
--| EiffelBase: library of reusable components for ISE Eiffel 3.
--| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
--|   Engineering Inc.
--| All rights reserved. Duplication and distribution prohibited.
--|
--| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
--| Telephone 805-685-1006
--| Fax 805-685-6869
--| Electronic mail <info@eiffel.com>
--| Customer support e-mail <support@eiffel.com>
--|----------------------------------------------------------------
```

A.6 TWO-WAY LISTS

```
indexing

        description:
                "Sequential, two-way linked lists";

        status: "See notice at end of class";
        names: two_way_list, sequence;
        representation: linked;
        access: index, cursor, membership;
        contents: generic;
        date: "$Date: 1996/01/15 16:33:34 $";
        revision: "$Revision: 1.14 $"

    class TWO_WAY_LIST [G] inherit
```

```
LINKED_LIST [G]
    redefine
        first_element, last_element,
        extend, put_front, put_left, put_right,
        merge_right, merge_left, new_cell,
        remove, remove_left, remove_right, wipe_out,
        previous, finish, move, islast, new_chain,
        forth, back
    select
        put_front,
        merge_right,
        move, put_right,
        wipe_out
    end;

LINKED_LIST [G]
    rename
        put_front as ll_put_front,
        put_right as ll_put_right,
        merge_right as ll_merge_right,
        move as ll_move,
        wipe_out as ll_wipe_out
    export
        {NONE}
            ll_put_front, ll_put_right,
            ll_move, ll_merge_right, ll_wipe_out
    redefine
        put_left, merge_left, remove, new_chain,
        remove_left, finish, islast, first_element, extend,
        last_element, previous, new_cell, remove_right,
        forth, back
    end
```

```
creation

	make_sublist, make

feature -- Access

	first_element: BI_LINKABLE [G];
			-- Head of list
			-- (Anchor redefinition)

	last_element: like first_element;
			-- Tail of the list

	sublist: like Current;
			-- Result produced by last `split'

feature -- Status report

	islast: BOOLEAN is
			-- Is cursor at last position?
		do
			Result := (active = last_element)
				and then not after
				and then not before
		end;

feature -- Cursor movement

	forth is
			-- Move cursor to next position, if any.
		do
```

```
            if before then
                before := false;
                if empty then
                    after := true
                end
            else
                active := active.right;
                if active = Void then
                    active := last_element;
                    after := true
                end
            end
        end;

    back is
            -- Move cursor to previous position, if any.
        do
            if after then
                after := false;
                if empty then
                    before := true
                end
            else
                active := active.left;
                if active = Void then
                    active := first_element;
                    before := true
                end
            end
        end;

    finish is
```

```
                -- Move cursor to last position.
                -- (Go before if empty)
        do
                if not empty then
                        active := last_element;
                        after := false;
                        before := false
                else
                        after := false;
                        before := true;
                end;
        ensure then
                not_after: not after
        end;

move (i: INTEGER) is
                -- Move cursor `i’ positions. The cursor
                -- may end up `off’ if the offset is to big.
        local
c: CURSOR;
                counter: INTEGER;
                p: like first_element
        do
                if i > 0 then
                        ll_move (i)
                elseif i < 0 then
                        if after then
                                after := false;
                                counter := -1
                        end;
                        from
                                p := active
                until
                    (counter = i) or else (p = Void)
                loop
                    p := p.left;
                    counter := counter - 1
                end;
                if p = Void then
                    before := true;
                    active := first_element
                else
                    active := p
                end
            end
        end;

feature -- Element change

    put_front (v: like item) is
            -- Add `v' to beginning.
            -- Do not move cursor.
        do
            ll_put_front (v);
            if count = 1 then
                last_element := first_element
            end
        end;

    extend (v: like item) is
            -- Add `v' to end.
            -- Do not move cursor.
        local
            p : like first_element
    do
        p := new_cell (v);
        if empty then
            first_element := p;
            active := p
        else
            p.put_left (last_element)
        end;
        last_element := p;
        if after then
            active := p
        end;
        count := count + 1
    end;

put_left (v: like item) is
        -- Add `v’ to the left of cursor position.
        -- Do not move cursor.
    local
        p: like first_element;
    do
        p := new_cell (v);
        if empty then
            first_element := p;
            last_element := p;
            active := p;
            before := false;
        elseif after then
            p.put_left (last_element);
            last_element := p;
            active := p;
        elseif isfirst then
```

```
                p.put_right (active);
                first_element := p
            else
                p.put_left (active.left);
                p.put_right (active);
            End;
            count := count + 1
        end;

    put_right (v: like item) is
            -- Add `v' to the right of cursor position.
            -- Do not move cursor.
        local
            was_last: BOOLEAN;
        do
            was_last := islast;
            ll_put_right (v);
            if count = 1 then
                        -- `p' is only element in list
                last_element := active
            elseif was_last then
                        -- `p' is last element in list
                last_element := active.right;
            end;
        end;

    merge_left (other: like Current) is
            -- Merge `other' into current structure before cursor
            -- position. Do not move cursor. Empty `other'.
        local
            other_first_element: like first_element;
            other_last_element: like first_element;
```

```
                other_count: INTEGER;
            do
                if not other.empty then
                    other_first_element := other.first_element;
                    other_last_element := other.last_element;
                    other_count := other.count;
                        check
                            other_first_element /= Void;
                            other_last_element /= Void
                        end;
                    if empty then
                        last_element := other_last_element;
                        first_element := other_first_element;
                        active := first_element;
                    elseif isfirst then
                        other_last_element.put_right (first_element);
                        first_element := other_first_element;
                    elseif after then
                        other_first_element.put_left (last_element);
                        last_element := other_last_element;
                        active := last_element;
                    else
                        other_first_element.put_left (active.left);
                        active.put_left (other_last_element);
                    end;
                    count := count + other_count;
                    other.wipe_out
                end
            end;

        merge_right (other: like Current) is
                -- Merge `other' into current structure after cursor
```

```
                    -- position. Do not move cursor. Empty `other'.
          do
                    if empty or else islast then
                              last_element := other.last_element
                    end;
                    ll_merge_right (other);
          end;

feature -- Removal

     remove is
                    -- Remove current item.
                    -- Move cursor to right neighbor
                    -- (or `after' if no right neighbor).
          local
                    succ, pred, removed: like first_element;
          do
                    removed := active;
                    if isfirst then
                              active := first_element.right;
                              first_element.forget_right;
                              first_element := active;
                              if count = 1 then
                                        check
                                                  no_active: active = Void
                                        end;
                                        after := true;
                                        last_element := Void
                              end;
                    elseif islast then
                              active := last_element.left;
                              last_element.forget_left;
```

```
                        last_element := active;
                        after := true;
                else
                        pred := active.left;
                        succ := active.right;
                        pred.forget_right;
                        succ.forget_left;
                        pred.put_right (succ);
                        active := succ
                end;
                count := count - 1;
                cleanup_after_remove (removed)
        end;

remove_left is
                -- Remove item to the left of cursor position.
                -- Do not move cursor.
        do
                back; remove
        end;

remove_right is
                -- Remove item to the right of cursor position.
                -- Do not move cursor.
        do
                forth; remove; back
        end;

wipe_out is
                -- Remove all items.
        do
                ll_wipe_out;
                last_element := Void
            end;

        split (n: INTEGER) is
                -- Remove from current list
                -- min (`n’, `count’ - `index’ - 1) items
                -- starting at cursor position.
                -- Move cursor right one position.
                -- Make extracted sublist accessible
                -- through attribute `sublist’.
            require
                not_off: not off;
                valid_sublist: n >= 0
            local
                actual_number, active_index: INTEGER;
                p_elem, s_elem, e_elem, n_elem: like first_element;
            do
                    -- recognize first breakpoint
                active_index := index;
                if active_index + n > count + 1 then
                    actual_number := count + 1 - active_index
                else
                    actual_number := n
                end;
                s_elem := active;
                p_elem := previous;
                    -- recognize second breakpoint
                move (actual_number - 1);
                e_elem := active;
                n_elem := next;
                    -- make sublist
                s_elem.forget_left;
            e_elem.forget_right;
            !! sublist.make_sublist (s_elem, e_elem, actual_number);
                -- fix `Current’
            count := count - actual_number;
            if p_elem /= Void then
                p_elem.put_right (n_elem)
            else
                first_element := n_elem
            end;
            if n_elem /= Void then
                active := n_elem
            else
                last_element := p_elem;
                active := p_elem;
                after := true
            end
        end;

    remove_sublist is
        do
            sublist := Void;
        end;

feature {TWO_WAY_LIST} -- Implementation

    make_sublist (first_item, last_item: like first_element; n: INTEGER) is
            -- Create sublist
        do
            make;
            first_element := first_item;
            last_element := last_item;
            count := n
        end;

    new_chain: like Current is
            -- A newly created instance of the same type.
            -- This feature may be redefined in descendants so as to
            -- produce an adequately allocated and initialized object.
        do
            !! Result.make
        end;

    new_cell (v: like item): like first_element is
            -- A newly created instance of the type of `first_element'.
        do
            !! Result;
            Result.put (v)
        end;

    previous: like first_element is
            -- Element left of cursor
        do
            if after then
                Result := active
            elseif active /= Void then
                Result := active.left
            end
        end;

end -- class TWO_WAY_LIST


--|----------------------------------------------------------------
--| EiffelBase: library of reusable components for ISE Eiffel 3.
```

```
    --| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
    --|   Engineering Inc.
    --| All rights reserved. Duplication and distribution prohibited.
    --|
    --| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
    --| Telephone 805-685-1006
    --| Fax 805-685-6869
    --| Electronic mail <info@eiffel.com>
    --| Customer support e-mail <support@eiffel.com>
    --|--------------------------------------------------------------
```

A.7 TWO-WAY TREES

```
indexing

        description:
            “Trees implemented using a two way linked list representation”;

        status: “See notice at end of class”;
        names: two_way_tree, tree, two_way_list;
        representation: recursive, linked;
        access: cursor, membership;
        contents: generic;
        date: “$Date: 1995/07/26 00:55:12 $”;
        revision: “$Revision: 1.12 $”

    class TWO_WAY_TREE [G] inherit

        DYNAMIC_TREE [G]
            undefine
                child_after, child_before, child_item,
                child_off
            redefine
        parent
    select
        has
    end;

BI_LINKABLE [G]
    rename
        left as left_sibling,
        right as right_sibling,
        put_left as bl_put_left,
        put_right as bl_put_right
    export
        {ANY}
            left_sibling, right_sibling;
        {TWO_WAY_TREE}
            bl_put_left, bl_put_right,
            forget_left, forget_right;
    end;

TWO_WAY_LIST [G]
    rename
        active as child,
        put_left as child_put_left,
        put_right as child_put_right,
        after as child_after,
        back as child_back,
        before as child_before,
        count as arity,
        cursor as child_cursor,
        duplicate as twl_duplicate,
        empty as is_leaf,
        extend as child_extend,
        extendible as child_extendible,
        fill as twl_fill,
        finish as child_finish,
        first_element as first_child,
        forth as child_forth,
        full as twl_full,
        go_i_th as child_go_i_th,
        go_to as child_go_to,
        has as twl_has,
        index as child_index,
        isfirst as child_isfirst,
        islast as child_islast,
        item as child_item,
        last_element as last_child,
        make as twl_make,
        merge_left as twl_merge_left,
        merge_right as twl_merge_right,
        off as child_off,
        prune as twl_prune,
        put as child_put,
        readable as child_readable,
        remove as remove_child,
        remove_left as remove_left_child,
        remove_right as remove_right_child,
        replace as child_replace,
        search as search_child,
        start as child_start,
        writable as child_writable
    export
        {ANY}
            child;
        {NONE}
```

```
                twl_make, twl_has,
                twl_fill, twl_duplicate,
                twl_full
        undefine
            child_readable, is_leaf,
            child_writable,
            linear_representation,
            child_isfirst, child_islast, valid_cursor_index
        redefine
            first_child, last_child, new_cell
        select
            is_leaf
        end

creation

    make

feature -- Initialization

    make (v: like item) is
            -- Create single node with item `v’.
        do
            put (v);
            twl_make
        end;

feature -- Access

    parent: TWO_WAY_TREE [G];
            -- Parent node
    first_child: like parent;
            -- Leftmost child

    last_child: like parent

feature -- Element change

    put_child (n: like parent) is
            -- Add `n’ to the list of children.
            -- Do not move child cursor.
        do
            if is_leaf then
                first_child := n;
                child := n
            else
                last_child.bl_put_right (n);
                if child_after then
                    child := n
                end
            end;
            last_child := n;
            n.attach_to_parent (Current);
            arity := arity + 1
        end;

    replace_child (n: like parent) is
            -- Replace current child by `n’.
        do
            put_child_right (n);
            remove_child
        end;
```

```
put_child_left (n: like parent) is
        -- Add `n' to the left of cursor position.
        -- Do not move cursor.
    do
        child_back;
        put_child_right (n);
        child_forth; child_forth
    end;

put_child_right (n: like parent) is
        -- Add `n' to the right of cursor position.
        -- Do not move cursor.
    do
        if child_before then
            if is_leaf then
                last_child := n
            end;
            n.bl_put_right (first_child);
            first_child := n;
            child := n
        elseif child_islast then
            child.bl_put_right (n);
            last_child := n
        else
            n.bl_put_right (child.right_sibling);
            n.bl_put_left (child)
        end;
        n.attach_to_parent (Current);
        arity := arity + 1
    end;

merge_tree_before (other: like first_child) is
```

```
                        -- Merge children of `other' into current structure
                        -- after cursor position. Do not move cursor.
                        -- Make `other' a leaf.
                do
                        attach (other);
                        twl_merge_left (other)
                end;

        merge_tree_after (other: like first_child) is
                        -- Merge children of `other' into current structure
                        -- after cursor position. Do not move cursor.
                        -- Make `other' a leaf.
                do
                        attach (other);
                        twl_merge_right (other)
                end;

        prune (n: like first_child) is
                local
                        l_child: like first_child;
                do
                        from
                                l_child := first_child
                        until
                                l_child = Void or l_child = n
                        loop
                                first_child := first_child.right_sibling
                        end;
                        if l_child = first_child then
                                first_child := first_child.right_sibling
                        elseif l_child = last_child then
                                last_child := last_child.left_sibling
                elseif l_child /= void then
                    l_child.right_sibling.bl_put_left (l_child.left_sibling);
                end;
                n.attach_to_parent (Void)
            end;

feature {LINKED_TREE} -- Implementation



        new_cell (v: like item): like first_child is
            do
                !! Result.make (v);
                Result.attach_to_parent (Current)
            end;

        new_tree: like Current is
                -- A newly created instance of the same type, with
                -- the same node value.
                -- This feature may be redefined in descendants so as to
                -- produce an adequately allocated and initialized object.
            do
                !! Result.make (item)
            end;

feature {NONE} -- Implementation

        attach (other: like first_child) is
                    -- Attach all children of `other’ to current node.
            local
                cursor: CURSOR;
            do
                from
```

```
                    other.child_start
                until
                    other.child_off
                loop
                    other.child.attach_to_parent (Current);
                    other.child_forth
                end;
                other.child_go_to (cursor)
            end;

feature -- Obsolete

        child_add_left (v: like item) is
                -- Add `v' to the left of current child.
                -- Do not move child
            obsolete "Use %"child_put_left%" instead."
            do
                child_put_left (v)
            end

        child_add_right (v: like item) is
                -- Add `v' to the right of current child.
                -- Do not move child.
            obsolete "Use %"child_put_right%" instead."
            do
                child_put_right (v)
            end

invariant

        off_constraint: (child = Void) implies child_off
```

```
end -- class TWO_WAY_TREE



--|----------------------------------------------------------------
--| EiffelBase: library of reusable components for ISE Eiffel 3.
--| Copyright (C) 1986, 1990, 1993, 1994, Interactive Software
--|   Engineering Inc.
--| All rights reserved. Duplication and distribution prohibited.
--|
--| 270 Storke Road, Suite 7, Santa Barbara, CA 93117 USA
--| Telephone 805-685-1006
--| Fax 805-685-6869
--| Electronic mail <info@eiffel.com>
--| Customer support e-mail <support@eiffel.com>
--|----------------------------------------------------------------
```