The Class LIST_BAG

The class LIST_SET did not allow repeated items, each item appeared once in the list. We consider a related class which allows repeated items and these items are implicitly kept in a ‘Last-in First-out’ order. As an example of using LIST_BAG, we give a class for quick_sorting the items in the list.

In order that we can implement the routine, join, more efficiently, we change the ‘export’ rules of the inherited attributes, first_node and cursor, so that they are available to the class LIST_BAG itself.

We also have to redefine the routine, add, as previously this routine checked if the list had the item already. The new version of the routine just puts the new item at the front of the list.

The routine, remove, is as before but in the case of LIST_BAG, the routine removes only the first occurrence of an item, if it is in the list.

```
class  LIST_BAG [G]
    inherit LIST_SET[G]
            export
                {LIST_BAG}
                first_node, cursor
            redefine
                add
            end
feature

    add(x : G) is  -- Add x, maybe again
        local
            n : NODE [G]
        do
                !!n
                n.set_item(x)
                n.set_next(first_node)
                first_node := n
                count := count + 1
        end -- add
        join(other : LIST_BAG[G]) is -- join to the end of current
                require
                        other /= void
                do
                        if not other.empty then
                                if not empty then
                                        finish
                                        cursor.set_next(other.first_node)
                                        count := count + other.count
                                else
                                        first_node := other.first_node
                                        count := other.count
                                end
                        end
                end -- join
end -- class LIST_BAG
```

The class LIST_BAG has, since it inherits from LIST_SET, all the features of LIST_SET, including a redefined version of add.
tf. the class LIST_BAG also contains the features:

count : INTEGER

empty : BOOLEAN

has (x : G) : BOOLEAN

add(x : G)

remove (x : G)

copy(other: like current)

is_equal(other: like current):BOOLEAN

-- traversal routines.

item : G -- item at cursor

start -- set cursor back to start

first : G -- The item at first_node

finish -- set cursor to last node

last : G -- return last item in list

forth -- move cursor forward

off : Boolean -- Is cursor beyond end

Quicksort on Lists

The algorithm for quicksort is the same for lists as for arrays; “split the list into a left and right partition about a pivot item and recursively quicksort each partition”.
We choose as pivot the item at the first node in the list.

With arrays we used a procedure to implement the algorithm, with lists we use a function. The list version of quicksort is not an in-place sort due to convenience and also because we want the functions to be free of side-effects. In sorting a list using a function we want the original list to remain intact.

The function for partition returns a pair of lists; the left and right partition. We therefore need a simple class for a pair of objects.

```
class PAIR[G]
feature
        first, second : G

        set_first(item:G) is
                do
                        first := item
                end -- set_first

        set_second(item:G) is
                do
                        second := item
                end -- set_second

end -- PAIR
```

Partition of a list.

Given a list S, Partition(S,P) returns two lists, L and R, say, such that all the items in L are less than the pivot, P, and all the items in R are not less (greater or equal) than the pivot.

The list S is traversed and the appropriate items in the list, S, are copied to L to R which are created and returned by the function.

```
partition (s:LIST_BAG[G]; pivot:G):PAIR[LIST_BAG [G]] is
        require
            s /= void and then  not  s.empty
        local
            left, right: LIST_BAG [G]
        do
            !! left;
            !! right;
            from
                s.start
            until
                s.off
            loop
                if  s.item < pivot then
                    left.add (s.item)
                else
                    right.add (s.item)
                end ;
                s.forth
            end ;
            !! Result;
            Result.set_first (left);
            Result.set_second (right)
        end ;
```

The function for Quicksort

In the function for quicksort, we use the function for partition. We don’t partition the full original list but this list with the pivot item removed. Otherwise, the recursive call may call a list of the same size. Since we removed the pivot item, we later add it back so as to preserve the original list.

```
quicksort (s: LIST_BAG [G]): LIST_BAG [G] is
    require
        s /= void and then  not  s.empty
    local
        left_part, right_part: LIST_BAG [G];
        p: PAIR [LIST_BAG [G]];
        pivot: G
    do
        if  s.count = 1 then
            result := clone (s)
        else
            !! left_part;
            !! right_part;
            pivot := s.first;
            s.remove (pivot);
            p := partition (s, pivot);
            if  not  p.first.empty then
                left_part := quicksort (p.first)
            end ;
            if  not  p.second.empty then
                right_part := quicksort (p.second)
            end ;
            right_part.add (pivot);
            left_part.join (right_part);
            result := left_part;
            s.add (pivot)
        end
    end ; -- quicksort
```

```
class  SORT_TEST
creation make
feature
    make is
        local
            s, s_new: LIST_BAG [STRING];
            p: QUICKSORT_LIST [STRING]
        do
            !! s;
            io.put_string("%NEnter words: %'quit%' to quit%N");
            from
                io.read_word
            until
                equal (io.last_string, "quit")
            loop
                s.add (io.last_string);
                io.read_word
            end ;
            print_list("%NOriginal list is: ->",s);
            !! p;
            s_new := p.quicksort (s);
            print_list("%N Sorted list is: ->%N",s_new);
            print_list("%NOriginal list was: ->",s);
        end ; -- make

    print_list (msg:STRING; s: LIST_BAG [STRING]) is
        do
            if s.empty then  print("List is empty")
            else
                print(msg);
                from
                  s.start
                until
                  s.off
                loop
                  io.put_string(s.item);io.putchar(' ');
                  s.forth
                end
            end
        end ;
end  -- class SORT_TEST
```