Container Libraries - Reference Manual

Table of Contents

Table of Contents

Container Libraries - Reference Manual

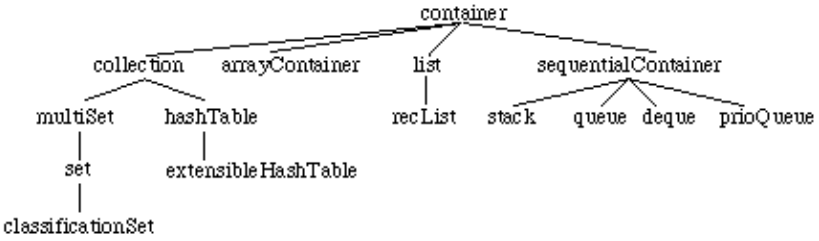
Mjølner Informatics Report
MIA 92-22
August 1999

Containers Reference Manual

The Container Libraries

This document describes version 1.6 of the container libraries. The container libraries implements many fairly often used data structures, such as sets, lists, stacks, etc., giving solutions to a very broad range of data structuring issues. It is intended that the container libraries will be extended in the future to include data structures such as trees, and mappings. The container libraries contains a number of abstract patterns which can be used as the basis for other container data structures.

The entire inheritance hierarchy for the container libraries is:



The container libraries are fragmented into relatively small fragments to enable the programmers to be very selective in picking the proper data structures without being burdened with the overhead of the other data structures.

We will in the following discuss the patterns in the container libraries individually.

Container

Container is the abstract superpattern for all patterns in the container libraries, defining all common attributes of container patterns.

Container patterns are generic patterns in the sense that the element type of the elements kept in the container is specified through the element virtual attribute.

Container defines the following additional attributes:

- init: To be invoked before the container object may be used.
- clear: empties the container.
- empty: returns true, if the container does not contain any elements.
- size: returns the number of elements in the container.
- capacity: returns the current capacity of the container. Most containers will dynamically expand as usage demands, and capacity returns the current capacity (subject to changes later, when usage demands). If the capacity of the container is in principle indefinite, capacity returns -1.
- has: takes a reference to an element object, and returns true if the element is in the container.
- scan: scans through the entire container, invoking INNER for each element in the container satisfying a predicate.
- find: scans through the entire container, invoking INNER first time an element is found, satisfying a predicate. Find also returns a reference to the element found.
- copy: creates an exact copy of the container. Only elements satisfying a predicate is copied to the new container.
- equal: a virtual procedure pattern, taking two element references. Equal returns true is the two element references are identical. Equal is used in the implementation of the above operations (and some of the later mentioned operations). If some other equality test is wanted for specific containers, this can be accomplished by further binding equal in that container.

Container also defines two exceptions:

- emptyContainerError: is invoked if the container is empty, and some container operation is invoked which demands a non empty container.
- illegalCellReference: is invoked, if some container operation is making illegal references internally in the container private data structures.

Collection

Collection is a subpattern of the container pattern and is the abstract superpattern for all position independent container patterns in the container libraries, defining all common attributes of these patterns.

Collection defines the following additional attributes:

- insert: takes an element reference and inserts the element in the collection.
- delete: takes an element reference and deletes that element from the collection.
- union: takes a reference to another collection, and unifies the elements in both collections into this(collection).
- diff, sect, symDiff: similar to union.

MultiSet

MultiSet is a subpattern of the collection pattern. MultiSet is an unstructured collection of element references, where duplicates are allowed. MultiSet does not define any additional operations.

Set

Set is a subpattern of the multiSet pattern. Set is like multiSet, except that duplicated are not allowed. Set does not define any additional operations.

Set

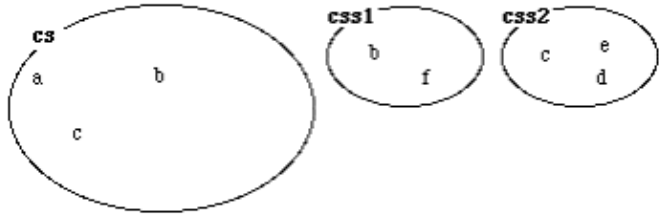
Set is a subpattern of the multiSet pattern. Set is like multiSet, except that duplicated are not allowed. Set does not define any additional operations.

ClassificationSet

ClassificationSet is a subpattern of the set pattern. ClassificationSet is used for dynamic classification of elements. ClassificationSet defines two additional attributes:

- subsets: is a set containing the subsets of this classificationSet.
- insertSubset: takes a classificationSet and inserts it as a subset of this classificationSet. The exception illegalSubset will be invoked if the subset inserted contains elements, that are not instances of a subpattern of the element type of this classificationSet.
- superSet: if this classificationSet is a subset of another classificationSet, superSet will reference that classificationSet.
- scanUnclassified: like scan except that it only scans those elements that are not member of any subsets of this classificationSet.

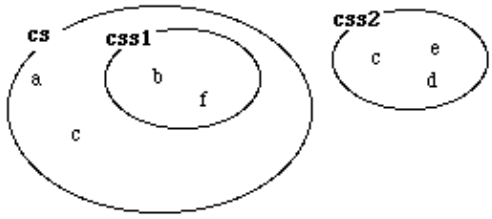
Let us illustrate by an abstract example, assuming that css1 contains the elements a, b and c, css2 contains the elements c. d and e. and finallv cs contains the elements b and f:



Then after

```
css1[]->cs.insertSubset
```

cs will contain the original elements, and the css1 subset:

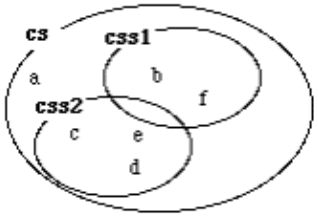


Note that the element b, originally located in both cs and css1, now is located exclusively in css1. In this situation, cs.scan(# ... #) will run through the elements a, b, c, d, e and f. and css1.scan(# ... #) will run through the elements b and f.

If we then execute

```
css2[]->cs.insertSubset
```

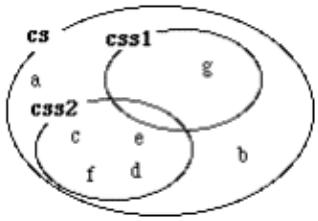
we will have the following situation:



Note again, that c is exclusively in css2. If we now execute

```
g[]->css1.insert; f[]->css2.insert; b[]->cs.delete
```

we will have the following situation:



Finally,

```
  css1.scanUnclassified(# ... #)
```

will only scan the elements a and b, whereas

```
  css1.scan(# ... #)
```

will scan all elements (i.e. a, b, c, d, e, f and g).

HashTable

HashTable is a subpattern of the collection pattern. HashTable implements a hash table data structure with the following additional attributes:

- rangeInitial: defines the range of internal hash values of the hash table.
- range: returns the actual range of internal hash values of the hash table.
- hashFunction: takes an element reference and returns a hash value for that element. In most uses of hashTable, this function must be further bound to give a more intelligent distribution of hash values.
- scanIndexed: takes a hash value and scans all elements in the hash table with the same hash value.
- findIndexed: takes a hash value and finds an element with the same hash value, satisfying predicate.
- statistics: enables calculation of various statistics about the usage of the hash table.

ExtensibleHashTable

ExtensibleHashTable is a subpattern of the hashTable pattern. ExtensibleHashTable allows for extension of the range of hash values of the hashTable. Unless the specified hashFunction somehow explicitly can cope with dynamic changes to the hash range, the entire hashTable needs to be rehashed. ExtensibleHashTable implements the following additional attributes:

- extend: extends the range of hash values of the hash table.
- rehash: makes a total rehash of the contents of the hash table.

ExtensibleHashTable

ExtensibleHashTable is a subpattern of the hashTable pattern. ExtensibleHashTable allows for extension of the range of hash values of the hashTable. Unless the specified hashFunction somehow explicitly can cope with dynamic changes to the hash range, the entire hashTable needs to be rehashed. ExtensibleHashTable implements the following additional attributes:

extend: extends the range of hash values of the hash table.

rehash: makes a total rehash of the contents of the hash table.

ArrayContainer

ArrayContainer is a subpattern of the container pattern. ArrayContainer is an abstraction of an element repetition, implementing the following additional attributes:

- put, get: inserts/returns an element reference at a given position.
- bubble-, shell- and quicksort: sorting operations.
- capacityInitial, capacityIncrement, capacityExtend: are used for controlling the strategy for extending the ArrayContainer. capacityInitial is used to specify the initial size of the repetition. capacityIncrement is used to specify the additional index values to be allocated when the capacityExtend is invoked.

SequentialContainer

SequentialContainer is a subpattern of the container pattern and is the abstract superpattern for patterns in the container libraries that implements data structures, where the elements are sequentially ordered, and where only the elements in the front or back of this sequence can be manipulated (e.g. stacks and queues). SequentialContainer does not define any additional operations.

Stack

Stack is a subpattern of the sequentialContainer pattern. Stack implements an ordinary stack with push, pop and top operations.

Queue

Queue is a subpattern of the sequentialContainer pattern. Queue implements an ordinary queue with insert, remove and front operations.

Deque

Deque is a subpattern of the sequentialContainer pattern. Deque implements an ordinary double-ended queue with insertFront, insertBack, removeFront, removeBack, front and back operations.

PrioQueue

PrioQueue is a subpattern of the sequentialContainer pattern. PrioQueue implements a priority queue with insert, remove, front and scanPriority operations. All operations takes a number, identifying a priority. ScanPriority scans through all elements of a given priority.

The Container Libraries

Queue

Queue is a subpattern of the sequentialContainer pattern. Queue implements an ordinary queue with insert, remove and front operations.

The Container Libraries

Deque

Deque is a subpattern of the sequentialContainer pattern. Deque implements an ordinary double-ended queue with insertFront, insertBack, removeFront, removeBack, front and back operations.

PrioQueue

PrioQueue is a subpattern of the sequentialContainer pattern. PrioQueue implements a priority queue with insert, remove, front and scanPriority operations. All operations takes a number, identifying a priority. ScanPriority scans through all elements of a given priority.

The Container Libraries

List

List is a subpattern of the container pattern. It implements a double-chained list data structure. The list pattern is a little special, since many operations takes list positions as arguments instead of elements. The reason is, that the typical usage of lists are to manipulate the list itself (finding the element just before a given position in the list, etc.). This adds a little extra complexity to the operations of the list pattern, relative to the other container patterns.

Central to the understanding of the list pattern, is the understanding of the theCellType pattern. TheCellType is a virtual pattern, defined in container pattern (but it is a private attribute, except for the list subpattern). In the list pattern, theCellType contains the references to the successor and predecessor list positions, as well as the elm reference to the element object contained at this list position. TheCellType describes the list positions (also often called list elements).

List defines the following additional operations:

- prepend: takes an element (i.e. a reference to an element) and adds it to the front of the list. Prepend returns the position of the element in the list (i.e. a reference to the instance of theCellType, containing the new element reference). This position may later be used to reference the element in the list (e.g. finding the succ of pred positions in the list).
- append: Similar to prepend, except that it inserts the element at the end of the list.
- head: Returns the position in the beginning of the list.
- last: Returns the position in the end of the list.
- tail: Returns a copy of the list, except for the first list element which have been removed.
- preample: Returns a copy of the list, except for the last list element which have been removed.
- concatenate: takes two lists and returns a new list which is the concatenation of the two lists.
- insertBefore, insertAfter: Takes an element and a list position, and inserts the element before (respectively after) the list position.
- delete, deleteBefore, deleteAfter: takes a list position and deletes the list position (respectively the list position before or after) from the list.
- splitBefore, splitAfter: takes a list position and returns two lists that are the lists from the beginning up to the list position, respectively from the list position until the end of the list. The list position will be in one of the lists.
- at: takes an element and returns the first list position, containing a reference to that element.
- locate: returns the position of the element in the list, satisfying predicate. This operation is similar to find, except that it returns the position, not the element.
- iterate, iterateFrom, iterateReverse, iterateReverseFrom: iterates through the list structure in different ways. In contrast to scan (inherited from container), all iterate variants respects the sublist structure such that a sublist will not be traversed automatically.
- scanFrom, scanReverse, scanReverseFrom: similar to the iterate operations above.

RecList

RecList is a subpattern of the list pattern, and makes it possible to represent recursive lists. TheCellType is further bound to include the sublist reference, and if the list position represents a sublist, sublist will contain a reference to the sublist. TheCellType further defines the when attribute, which may be used to specify actions to be executed in case the list position contains an element reference, respectively a sublist. Finally, recList defines the following additional operations:

insertSublistBefore, insertSublistAfter: Takes a list and a list position, and inserts the list as a sublist before (respectively after) the list position

RecList defines the exception illegalSublist, which is invoked if the elements of an sublist are not instances of a subpattern of the element type of this recList.

Containers Reference Manual

Using the Container Libraries

The container libraries consists of a number of fragments: container, collection, sets, classification, arrayContainer, hashTable, seqContainers, list and recList:

- container.bet contains the definition of the abstract pattern container. It is the origin fragment for all the other container fragments.
- collection.bet contains the definition of the abstract pattern collection. It is the origin for the sets, classification and hashTable fragments.
- sets.bet contains the definition of the **multiSet and set** patterns. It is the origin of the classification fragment.
- classification.bet contains the definition of the **classificationSet** pattern.
- hashTable.bet contains the definition of the **hashTable and extensibleHashTable** patterns.
- arrayContainer.bet contains the definition of the **ArrayContainer** pattern.
- list.bet contains the definition of the **list** pattern.
- recList.bet contains the definition of the **recList** pattern.
- seqContainers.bet contains the definition of the abstract pattern sequentialContainer along with the **stack, queue, prioQueue and deque** patterns.

An application may utilize the container libraries by including the appropriate library fragment. E.g. in order for an application to utilize the multiSet pattern, the application must have the following outline:

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/containers/sets'
--- program: descriptor ---
(# ...
   recordBag: @multiSet(# element:: record #);
   aRecord: @record;
   aPerson: @person; (* person is a subpattern of record *)
   ...
do ...
   recordBag.init;
   ...
   aPerson[] -> recordBag.insert;
   ...
   aRecord[] -> recordBag.insert;
   ...
   recordBag.scan(# where:: (# current.key<3000 -> value #)
                  do current.print
                  #);
   (* prints all records in recordBag that has a key less that
    * 3000. It is assumed that record implements a virtual print
    * operation
    *)
   ...
#)
```

In order to utilize the other fragments, just replace

```
INCLUDE '~beta/containers/sets'
```

with e.g.

```
INCLUDE '~beta/containers/hashTable'
```

and in order to use more than one container fragment, just insert several INCLUDEs:

```
INCLUDE '~beta/containers/sets';
```

```
INCLUDE '~beta/containers/hashTable'
```

Note that since the container fragments has ORIGIN in betaenv, the above is equivalent to:

```
ORIGIN '~beta/containers/sets'
--- program: descriptor ---
(# ...
   recordBag: @multiSet(# element:: record #);
   ...
do ...
   ...
#)
```

Using the Container Libraries

Examples of Use of the Container Libraries

The following will illustrate the usage of some of the container fragments. Following each example, the output from the example program is given.

Set Example

```
ORIGIN '~beta/containers/sets';
--- program:descriptor ---
(* This demo program illustrates the usage of the set pattern.  The first part
 * of the demo illustrates inserting elements from the sets, and
 * the last part illustrates the union and symDiff operations.
 *
 * At the end of this file, a copy of the output of this program is given
 *)
(# intSet: set
     (* intSet is a set containing integerObjects *)
     (# element::< integerObject #);

   intSet1, intSet2, intSet3: @intSet;
   i: [10]^integerObject;

do (* initializing the integerObjects *)
   (for int:10 repeat &integerObject[]->i[int][]; int->i[int] for);

   (* initializating intSet1 and inserting integerObjects into it *)
   intSet1.init;
   'intSet1.capacity: '->puttext; intSet1.capacity->putInt; newline;
   (for int:4 repeat '*'->put;
      i[int][]->intSet1.insert
   for);
   (* printing the size of intSet1 *) '+'->put;
   'intSet1.size: '->puttext; intSet1.size->putInt; newline;
   (* printing intSet1 *)
   'intSet1.elements: '->puttext;
   intSet1.scan(# do current->putint; ','->put #); newline;

   (* initializating intSet2 and inserting integerObjects into it *)
   intSet2.init;
   (for int:4 repeat
      i[int+3][]->intSet2.insert
   for);
   (* printing intSet2 *)
   'intSet2.elements: '->puttext;
   intSet2.scan(# do current->putint; ','->put #); newline;

   (* illustrating the use of the union, diff, sect and symDiff operations *)
   'intSet2->intSet3; intSet1->intSet3.union: '->puttext;
   intSet2->intSet3; intSet1->intSet3.union;
   intSet3.scan(# do current->putint; ','->put #); newline;

   'intSet2->intSet3; intSet1->intSet3.symDiff: '->puttext;
   intSet2->intSet3; intSet1->intSet3.symDiff;
   intSet3.scan(# do current->putint; ','->put #); newline;

   (* illustrating finding some integerObject in insSets *)
   'intSet3.find(# predicate::< (# "elm=7" #) #): '->puttext;
   intSet3.find(# predicate::< (# do (current=7)->value #)
               do current->putint
               #); newline;
   '7->intSet3.has: '->puttext;
   (if i[7][]->intSet3.has
    //true then 'yes'->putline
    else 'no'->putline
   if);
   '10->intSet3.has: '->puttext;
   (if i[10][]->intSet3.has
    //true then 'yes'->putline
```

```
    else 'no'->putline
   if);

   (********** OUTPUT **************
    * intSet1.capacity: -1
    * intSet1.size: 4
    * intSet1.elements: 4,3,2,1,
    * intSet2.elements: 7,6,5,4,
    * intSet2->intSet3; intSet1->intSet3.union: 1,2,3,7,6,5,4,
    * intSet2->intSet3; intSet1->intSet3.symDiff: 1,2,3,7,6,5,
    * intSet3.find(# predicate::< (# "elm=7" #) #): 7
    * 7->intSet3.has: yes
    * 10->intSet3.has: no
    ********************************)
#)
```

HashTable Example

```
ORIGIN '~beta/containers/hashTable';
-- program: Descriptor --
(* This demo program illustrates the usage of the hashTable and
 * extensibleHashTable patterns.  The first part of the demo
 * illustrates inserting and deleting elements from the hashtables,
 * the middle part illustrates the diff and sect operations, and the
 * last part of this program illustrates the possibilities for
 * specifying extend the hashTable dynamically (illustrated by the
 * hashTable4 object).  The use of the statistics operations are also
 * illustrated here.
 *
 * At the end of this file, a copy of the output of this program is
 * given
 *)
  (#
     intHashTable: hashTable (* a hashTable containing integerObjects *)
       (# element::< integerObject; hashFunction::<  (#  do e->value #);
       #);
     intEHashTable: extensibleHashTable
     (* a hashTable containing integerObjects *)
       (# element::< integerObject; hashFunction::<  (#  do e->value #);
       #);
     hashtable1: @intHashTable
       (# rangeInitial::<  (#  do 2->value #) #);
     hashtable2: @intHashTable
       (# rangeInitial::<  (#  do 4->value #) #);
     hashtable3: @intHashTable;
     hashtable4: @intEHashTable
       (# rangeInitial::<  (#  do 2->value #) #);
     i: [10] ^integerObject;

  do (* initializing the integerObjects *)
     (for int: 10 repeat
       &integerObject[]->i[int][]; int->i[int]
     for);
     (* initializing hashTable1 and putting some integerObjects into it. *)
     hashtable1.init;
     'hashtable1.capacity: '->puttext;
     hashtable1.capacity->putInt;
     newline;
     'hashtable1.range: '->puttext;
     hashtable1.range->putInt;
     newline;
     (for int: 4 repeat i[int][]->hashtable1.insert for);
     'hashtable1.elements: '->puttext;
     hashtable1.scan
       (#  do current->putint; ','->put #);
     newline;
     'hashtable1.size: '->puttext;
     hashtable1.size->putInt;
     newline;
     newline;
     (* initializing hashTable2 and putting some integerObjects into it. *)
     hashtable2.init;
     'hashtable2.capacity: '->puttext;
     hashtable2.capacity->putInt;
     newline;
     'hashtable2.range: '->puttext;
     hashtable2.range->putInt;
     newline;
     (for int: 4 repeat i[int+3][]->hashtable2.insert for);
     'hashtable2.elements: '->puttext;
```

```
hashtable2.scan
  (#  do current->putint; ','->put #);
newline;
'hashtable2.size: '->puttext;
hashtable2.size->putInt;
newline;
newline;
(* initializing hashTable3 and putting some integerObjects into it. *)
hashtable3.init;
(for int: 4 repeat i[int+6][]->hashtable3.insert for);
'hashtable3.elements: '->puttext;
hashtable3.scan
  (#  do current->putint; ','->put #);
newline;
newline;
(* deleting integerObject 3 from hashTable3 *)
'3->hashtable3.delete: '->puttext;
i[3][]->hashtable3.delete;
hashtable3.scan
  (#  do current->putint; ','->put #);
newline;
newline;
(* illustrating the diff, and sect operations on hashTables *)
'hashtable2->hashtable3; hashtable1[]->hashtable3.diff: '->puttext;
hashtable2->hashtable3;
hashtable1[]->hashtable3.diff;
hashtable3.scan
  (#  do current->putint; ','->put #);
newline;
'hashtable2->hashtable3; hashtable1[]->hashtable3.sect: '->puttext;
hashtable2->hashtable3;
hashtable1[]->hashtable3.sect;
hashtable3.scan
  (#  do current->putint; ','->put #);
newline;
(* Illustrating the use of extensibleHashTable:
 * Initializing hashTable4 and putting some integerObjects into it.
 *)
hashtable4.init;
'hashtable4.capacity: '->puttext;
hashtable4.capacity->putInt;
newline;
'hashtable4.range: '->puttext;
hashtable4.range->putInt;
newline;
(for int: 4 repeat i[int+3][]->hashtable4.insert for);
'hashtable4.elements: '->puttext;
hashtable4.scan
  (#  do current->putint; ','->put #);
newline;
'hashtable4.size: '->putText;
hashtable4.size->putInt;
newline;
(* illustrating extending an expensibleHashTable *)
'hashtable4.statistics(# do screen[]->print #): '->putline;
hashtable4.statistics
  (#  do screen[]->print #);
(for int: hashTable4.range repeat
  int->putint;
  '->hashtable4.scanIndexed(# ... #): '->puttext;
  i[int][]->hashTable4.hashFunction
    ->hashtable4.scanIndexed (#  do current->putint; ' '->put #);
  'found'->putline;

for);
'3->hashTable4.extend(# do hashTable4.rehash #)'->putline;
```

```
3->hashTable4.extend (#  do hashTable4.rehash #);
(for int: 10 repeat
  int->putint;
  '->hashtable4.has: '->puttext;
  (if i[int][]->hashtable4.has
   // true then 'yes'->putline
   else
      'no'->putline
  if);

for);
(for int: hashTable4.range repeat
  int->putint;
  '->hashtable4.scanIndexed(# ... #): '->puttext;
  i[int][]->hashTable4.hashFunction
    ->hashtable4.scanIndexed
      (# end::<  (#  do 'found'->putline #)
      do current->putint; ' '->put
      #);

for);
'hashtable4.range: '->puttext;
hashtable4.range->putInt;
newline;
'hashtable4.elements: '->puttext;
hashtable4.scan
  (#  do current->putint; ','->put #);
newline;
'hashtable4.size: '->putText;
hashtable4.size->putInt;
newline;
'hashtable4.statistics(# do screen[]->print #): '->putline;
hashtable4.statistics
  (#  do screen[]->print #);
(************ OUTPUT ***************
 * hashtable1.capacity: -1
 * hashtable1.range: 2
 * hashtable1.elements: 3,1,4,2,
 * hashtable1.size: 4
 *
 * hashtable2.capacity: -1
 * hashtable2.range: 4
 * hashtable2.elements: 5,6,7,4,
 * hashtable2.size: 4
 *
 * hashtable3.elements: 7,8,9,10,
 *
 * 3->hashtable3.delete: 7,8,9,10,
 *
 * hashtable2->hashtable3; hashtable1[]->hashtable3.diff: 5,6,7,
 * hashtable2->hashtable3; hashtable1[]->hashtable3.sect: 4,
 * hashtable4.capacity: -1
 * hashtable4.range: 2
 * hashtable4.elements: 7,5,6,4,
 * hashtable4.size: 4
 * hashtable4.statistics(# do screen[]->print #):
 * Histogram: (2,2)
 * Maximum Collisions: 2
 * Minimum Collisions: 2
 * Average Collisions: 2
 * 1->hashtable4.scanIndexed(# ... #): 7 5 found
 * 2->hashtable4.scanIndexed(# ... #): 6 4 found
 * 3->hashTable4.extend(# do hashTable4.rehash #)
 * 1->hashtable4.has: no
 * 2->hashtable4.has: no
 * 3->hashtable4.has: no
```

```
    * 4->hashtable4.has: yes
    * 5->hashtable4.has: yes
    * 6->hashtable4.has: yes
    * 7->hashtable4.has: yes
    * 8->hashtable4.has: no
    * 9->hashtable4.has: no
    * 10->hashtable4.has: no
    * 1->hashtable4.scanIndexed(# ... #): 6 found
    * 2->hashtable4.scanIndexed(# ... #): 7 found
    * 3->hashtable4.scanIndexed(# ... #): found
    * 4->hashtable4.scanIndexed(# ... #): 4 found
    * 5->hashtable4.scanIndexed(# ... #): 5 found
    * hashtable4.range: 5
    * hashtable4.elements: 6,7,4,5,
    * hashtable4.size: 4
    * hashtable4.statistics(# do screen[]->print #):
    * Histogram: (1,1,0,1,1)
    * Maximum Collisions: 1
    * Minimum Collisions: 0
    * Average Collisions: 1
    ********************************)
#)
```

List Example

```
ORIGIN '~beta/containers/list';
--- program:descriptor ---
(* This demo program illustrates the usage of the list pattern.  The first
 * part of the demo illustrates inserting and deleting elements from the lists.
 *
 * At the end of this file, a copy of the output of this program is given
 *)
(# intList: list(# element::< integerObject #);

   intList1, intList2: @intList; intList3: ^intList;
   i: [10]^integerObject;
   listPosition: ^intList3.theCellType;

do (* initializing the integerObjects *)
   (for int:10 repeat &integerObject[]->i[int][]; int->i[int] for);

   (* initializing intList1 and putting some integerObjects into it *)
   intList1.init;
   'intList1.prepend: '->puttext;
   (for int:4 repeat
      i[int][]->intList1.prepend
   for);
   intList1.scan(# do current->putint; ','->put #); newline;

   (* initializing intList3 and putting some integerObjects into it *)
   intList2.init;
   'intList2.append: '->puttext;
   (for int:4 repeat
      i[int+3][]->intList2.append
   for);
   intList2.scan(# do current->putint; ','->put #); newline;

   (* illustrates the reverse scanning *)
   'intList2.scanReverse: '->puttext;
   intList2.scanReverse(# do current->putint; ','->put #); newline;

   (* initializing intList3 *)
   &intList[]->intList3[]; intList3.init;

   (* illustrating concatenation of lists *)
   'intList1[]->intList2.concatenate->intList3[]: '->puttext;
   intList1[]->intList2.concatenate->intList3[];
   intList3.scan(# do current->putint; ','->put #); newline;

   (* illustration getting the position of some integerObject in the list, and
    * replacing the element at that position
    *)
   '"10"->("2"->intList3.at).elm[]: '->puttext;
   i[2][]->intList3.at->listPosition[];
   i[10][]->listPosition.elm[];
   intList3.scan(# do current->putint; ','->put #); newline;

   (* illustrates deleting an element from the list *)
   '6->intList2.at->intList2.delete: '->puttext;
   i[6][]->intList2.at->intList2.delete;
   intList2.scan(# do current->putint; ','->put #); newline;

    i[7][]->intList3.at->listPosition[];
  (* illustrating inserting integerObjects before some list position *)
   '(8,7)->intList3.insertBefore: '->puttext;
   (i[8][],listPosition[])->intList3.insertBefore;
   intList3.scan(# do current->putint; ','->put #); newline;
```

```
   (* illustrating deleting integerObjects after some list position *)
   '7->intList3.deleteAfter: '->puttext;
   listPosition[]->intList3.deleteAfter;
   intList3.scan(# do current->putint; ','->put #); newline;

   (********** OUTPUT **************
    * intList1.prepend: 4,3,2,1,
    * intList2.append: 4,5,6,7,
    * intList2.scanReverse: 7,6,5,4,
    * intList1[]->intList2.concatenate->intList3[]: 4,5,6,7,4,3,2,1,
    * "10"->("2"->intList3.at).elm[]: 4,5,6,7,4,3,10,1,
    * 6->intList2.at->intList2.delete: 4,5,7,
    * (8,7)->intList3.insertBefore: 4,5,6,8,7,4,3,10,1,
    * 7->intList3.deleteAfter: 4,5,6,8,7,3,10,1,
    ****************************)
#)
```

Container Interface

```
ORIGIN '~beta/basiclib/betaenv';
LIB_DEF 'container' '../lib';
BODY 'private/containerBody';
(*
 * COPYRIGHT
 *
 *       Copyright (C) Mjolner Informatics, 1992-95
 *
 *       All rights reserved.
 *)
-- lib: Attributes --
container:
(* The top most pattern in the data structures hierarchy.
 * Container is an abstract superpattern and the currently available
 * subpatterns are (indentation specifies specialization):
 *
 *   container
 *     collection
 *       multiSet
 *         set
 *           classificationSet --- allows for sets of sets, etc.
 *       hashTable
 *         extensibleHashTable --- allows for extending the index
 *                                 range
 *     list
 *       recList --- allows for recursive lists
 *     arrayContainer --- includes bubble/shell/quicksort
 *     sequentialContainer
 *       stack
 *       queue
 *       prioqueue
 *       deque
 *)
  (#
     <<SLOT containerLib:Attributes>>;
     element:< Object
     (* The qualification of the objects, contained in this
      * container.  When using one of the subpatterns of container,
      * further binding of element is used to restrict the kinds of
      * objects, that can be inserted into the container.  None of
      * the predefined container types specializes element.
      *) ;
     init:<
     (* Should be invoked before any usages of any type of
      * container
      *) Object;
     clear:<
     (* Removes all elements currently in the container, making it
      * empty
      *) Object;
     empty:< booleanObject; (* Returns true if the container is empty *)
     size:< integerValue
     (* Returns the number of elements currently in the container
      *) ;
     capacity:< integerValue
     (* Returns the current capacity of this container. Some
      * specializations may dynamically extend its capacity.  In
      * this case, capacity returns the current capacity (subject to
      * extension at some later stage).  If -1 is returned, the
      * capacity of the container is infinite (limited only by
      * memory).
      *) (#  do - 1->value; INNER #);
```

```
equal:< booleanValue
(* Defines the equality test, used in the various subpatterns
 * of container in the implementation of the different
 * operations.  Users of container patterns must further bind
 * equal to contain the proper equality test for the specified
 * element type.  Default equality test for equal references
 * (i.e. the same object)
 *)
  (# left,right: ^element
  enter (left[],right[])
  do (left[] = right[])->value; INNER
  #);
has:< booleanValue
(* Takes an element, and checks whether it is in the container
 *)
  (#
     elm: ^element;
     doneInInner: @boolean;
  enter elm[]
  do false->value;
     INNER has;
  #);
theScanner:<  (* private *)
  (# w:^s.where;
     s:^scan;
     aCell:^theCellType;
  enter (s[],w[])
  ...
  #);
scan:
(* Scans through the container, invoking INNER for each
 * element in the container, for which "where" returns true.
 * "Start" is invoked at the start of the scanning, and "end"
 * is invoked at the end of the scan.  In each turn of the
 * scan, "current" refers to the current element in the
 * container.  The elements will be scanned in unpredictable
 * order, unless otherwise explicitly mentioned otherwise in
 * the subpatterns
 *)
  (#
     first:@boolean; (* private *)
     where:< elementPredicate;
     current: ^element;
     start:< object;
     end:< object;
  ...
  #);
find:<
(* Searches through the container, executing INNER for the
 * element found and returning the first element found that
 * satisfies the predicate.  "Start" is invoked at the start of
 * the search, and "end" is invoked at the end of the search.
 * In each turn of the scan, "current" refers to the current
 * element in the container.  If no element satisfying the
 * predicate is found, the notification "notFound" is invoked.
 *)
  (#
     predicate:< elementPredicate;
     current: ^element;
     notFound:< Notification
       (#
       do none->current[];
          'Element not found in container'->msg.putLine; INNER
       #);
     start:< object;
     end:< object;
```

```
      ...
      exit current[]
      #);
    copy:<
    (* Default copy is one-level (shallow) copying.  I.e. copying
     * the container and all objects in the container.  Only
     * elements satisfying "predicate" will be copied.  During the
     * copying, "current" will refer to the element being
     * considered for copying.  If another copying in needed, this
     * can be done by further binding "copy", and assigning "true"
     * to "doneInInner", which will result in the default copying
     * being ignored
     *)
      (#
         doneInInner: @boolean;
         theCopy: ^container;
         predicate:< elementPredicate;
         current: ^element;
      do INNER ;
      exit theCopy[]
      #);
    emptyContainer: Exception
    (* Invoked if some operation not valid for empty containers
     * are invoked on an empty container
     *) (#  do 'Empty container'->msg.putLine; INNER #);
    emptyContainerError:< emptyContainer
    (* Used to handle emptyContainer errors, not handled by local
     * exception handlers in operations for containers
     *) ;
    illegalCellReference:< Exception
    (* Invoked if some operation tries to reference a non-existing
     * cell
     *)
      (#
      do 'Reference to nonexisting cellObject in container'->msg.putLine;
         INNER
      #);
    elementPredicate: booleanValue
    (* This pattern is used as the superpattern for all predicates
     * in the find, scan and copy operations.
     *) (# current: ^element enter current[] do true->value; INNER #);
    theCellType:< (* Private *)
    (* This pattern is used as the common superpattern for all
     * data structure cells, used by implementations of the
     * subpatterns of container.
     *)
      (# elm:^element;
         copy:<
           (# theCellCopy: ^theCellType
           do &theCellType[]->theCellCopy[];
              elm[]->theCellCopy.elm[];
              INNER
           exit theCellCopy[]
           #)
      do INNER
      #);
    doEnter:<
      (# containerType:<container;
         theOther: ^containerType;
         thePred:@elementPredicate;
      enter theOther[]
      ...
      #)
  enter doEnter
    (* will copy the contents of theOther[] into THIS(container),
     * but only elements verified with thePred;
```

```
    *)
do INNER
exit THIS(container)[]
#)
```

Collection Interface

```
ORIGIN 'container';
LIB_DEF 'collection' '../lib';
BODY 'private/collectionBody';
(*
 * COPYRIGHT
 *
 *       Copyright (C) Mjolner Informatics, 1992-94
 *
 *       All rights reserved.
 *)
--- lib: attributes ---
collection: container
  (* Collection is the superpattern for all collection data
   * structures.  Defines the operations:
   *     insert, delete, union, diff, sect, xclOr
   * Container is an abstract superpattern and the currently available
   * subpatterns are (indentation specifies specialization):
   *
   *     collection
   *       multiSet
   *         set
   *           classificationSet --- allows for sets of sets, etc.
   *       hashTable
   *         extensibleHashTable --- allows for extending the index
   *                                 range
   *)
  (# <<SLOT collectionLib: attributes>>;
     insert:<
       (* Takes an element and inserts it in THIS(collection) *)
       (# elm: ^element
       enter elm[]
       do INNER
       #);
     delete:<
       (* Takes an element and removes the object from
        * THIS(collection).  Invokes the notification notFound, if the
        * element cannot be found.
        *)
       (# elm: ^element;
          notFound:< Notification
            (#
            do 'Element not found in container'->msg.putLine;
               INNER
            #);
       enter elm[]
       do INNER
       #);
     union:<
       (* Takes a collection, and unifies its elements into
        *         THIS(collection)
        *)
       (# theOther: ^collection
       enter theOther[]
       ...
       exit this(collection)[]
       #);
     diff:<
       (* Takes a collection, and removes its elements from
        *         THIS(collection)
        *)
       (# theOther: ^collection
       enter theOther[]
```

```
      ...
      exit this(collection)[]
      #);
    sect:<
      (* Takes a collection, and keeps in THIS(collection) those
       * elements that are in both collections
       *)
      (# theOther: ^collection
      enter theOther[]
      ...
      exit this(collection)[]
      #);
    symDiff:<
      (* Takes a collection, and inserts into THIS(collection) those
       * elements not being in both collections
       *)
      (# theOther: ^collection; tmp1, tmp2: ^collection
      enter theOther[]
      ...
      exit this(collection)[]
      #);
  do INNER
  #)
```

Sets Interface

```
ORIGIN 'collection';
BODY 'private/setsBody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-94
 *       All rights reserved.
 *)
--- lib: attributes ---
multiSet: collection
  (* MultiSet is a data structure for collecting elements (dublicates
   * are allowed of the same element).  Defines no new operations.
   * MultiSet is the superpattern for Set, which ignores dublicates.
   * The available subpatterns are (indentation specifies
   * specialization):
   *       multiSet
   *          set
   *             classification
   *)
  (# <<SLOT multiSetLib: attributes>>;
     theScanner::<  (* private *)
       (#
       ...
       #);
     find::
       (#
       ...
       #);
     copy::<
       (#
       ...
       #);
     empty::<
       (#
       ...
       #);
     capacity::<
       (#
       ...
       #);
     has::<
       (# myEqual:@equal;
       ...
       #);
     clear::< (# ... #);
     size::< (# ... #);
     insert::<
       (* inserting in a multiSet is somewhat dependent on the
        * particular subpattern.  In some cases, insertion in only
        * done if the element is not already in the multiSet
        * (e.g. sets).  The local attribute, allowedToInsert, controls
        * whether the element is actually inserted in the multiSet.
        *)
       (# allowedToInsert: @boolean;
          multiSetInsertPrivate: @ (* O(1) amort. *)
            ...
       do true->allowedToInsert;
          INNER;
          (if allowedToInsert then multiSetInsertPrivate if)
       #);
     delete::<
       (# multiSetDeletePrivate: @...
       do multiSetDeletePrivate;
```

```
          INNER
        #);
      theCellType::< (* Private *)
        (# pred, succ: (* Private *) ^theCellType;
        do INNER #);
      doEnter::<
        (# containerType::<multiSet;
        ...
        #);
      private:@...;
  do INNER
  #) (* multiSet *);

(*-------------------------------------------------------------*)
(*--- set: multiSet -------------------------------------------*)
(*-------------------------------------------------------------*)

set: multiSet
  (* A set is a multiSet in which dublicates are removed (ie. not inserted).
   * Defines no new operations.
   *)
  (# <<SLOT setLib: attributes>>;
     insert::<
       (* Inserts only the element if it is not in the set already *)
       (# ... #);
     doEnter::<
       (# containerType::<Set;
       do INNER
       #);
  do INNER
  #)
```

Classification Interface

```
ORIGIN 'sets';
BODY 'private/classificationBody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-94
 *       All rights reserved.
 *)
--- lib: attributes ---
classificationSet: set
  (* A classification set is a set in which subsets may be defined.
   * The subsets of a classificationSet are such that inserting
   * elements in the subsets, makes the element a member of the
   * classificationSet.  Inserting an element in one of the subsets of
   * the classificationSet, which is already in the classificationSet
   * (or one of its subsets), will move the element from the
   * classificationSet (or the subset) into the proper subset.  If a
   * classificationSet is inserted as a subset of
   * THIS(classificationSet), it is ensured that inserting elements in
   * the subset will not result in dublicates being present in
   * THIS(classificationSet).  The reference SuperSet in
   * THIS(classificationSet) refers to the classificationSet,
   * THIS(classificationSet) might be a subset of.  SuperSet is NONE,
   * if THIS(classificationSet) is not a subset of any other
   * classificationSet.  Defines two new operations:
   *       insertSubset, scanUnclassified
   *)
  (# <<SLOT classificationSetLib: attributes>>;
     clear::< (# ... #);
     size::< (# ... #);
     has::< (# ... #);
     copy::< (# ... #);
     delete::<(# ... #);
     subsets: @
       (* The subsets currently registered in THIS(classificationSet)
        *)
       set(# element::< classificationSet #);
     superSet: ^classificationSet;
     insertSubset:
       (* Takes a classificationSet, and defines it as a subset of
        * this classificationSet.  The elements of this subset will
        * now be member of THIS(classificationSet)
        *)
       (# ss: ^classificationSet;
          illegalSubset:< Exception
            (* Will be invoked if a subset is inserted with an
             * element type is not a subpattern of the element type of
             * THIS(classificationSet).
             *)
            (#
            do 'The elements of the subset does not share '->msg.putText;
               'qualifications with the elements of this set'->msg.putLine;
               INNER
            #)
       enter ss[]
       ...
       #);
     insert::< (# ... #);
     scanUnclassified:
       (* similar to scan, except that it only scans those elements
        * that are NOT member of any subsets
        *)
       (# where:< elementPredicate;
```

```
         current: ^element;
         start:< object;
         end:< object
      do start; ...; end
      #);
    theScanner::<  (* private *)
      (#
      ...
      #);
    doEnter::<
      (# containerType::<classificationSet;
         ...
      #);
#)
```

HashTable Interface

```
ORIGIN 'collection';
LIB_DEF 'hashtable' '../lib';
BODY 'private/hashTableBody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-94
 *       All rights reserved.
 *)
--- lib:attributes ---
hashTable: collection
  (* A hashTable is an efficient data structure for storing a unknown
   * number of elements for quick access.  Defines the following new
   * operations: range, hashFunction, scanIndexed, findIndexed and
   * statistics.  The available subpatterns are (indentation specifies
   * specialization):
   *       hashTable
   *          extensibleHashTable
   *)
  (# <<SLOT hashTableLib:attributes>>;
     init::< (# ... #);
     clear::< (# ... #);
     size::< (# ... #);
     rangeInitial:< integerValue
       (* This pattern is used to define the value domain for the
        * hashFunction and thus the number of indices in
        * this(hashTable).
        *)
       (# do 17->value; INNER #);
     range: integerValue
       (* This pattern returns the number of indices in
        * this(hashTable).
        *)
       (# ... #);
     indexValue: integerValue
       (#
       do INNER;
          (if value < 0 then -value->value if);
          value mod range->value;
          (if value = 0 then
              range->value;
          if)
       #);
     index: indexValue(# enter value do INNER #);
     theIndex: @index;
     hashFunction:< indexValue
       (* This pattern is invoked on all elements inserted into
        * THIS(hashTable).  The default behaviour will generate many
        * collisions, and specialization is therefore strongly advised
        *)
       (# e: ^element
       enter e[]
       do 0->value;
          INNER;
       #);
     empty::< (# ... #);
     has::< (# ... #);
     insert::< (# ... #);
     delete::< (# ... #);
     copy::< (# ... #);
     theScanner::<(# ... #);
     scanIndexed:
       (* enters an index value, and scans all elements in
```

```
    * this(hashTable) with the same index value (i.e. all
    * collisions on that index value.  That is, scanIndexed is
    * similar to scan, except that it only scans the elements
    * which happen to be indexed at the same index in the
    * hashTable.  Each time INNER is invoked, current refers to
    * the actual element.  Note that if you want to scan all
    * element with the same index as some known element, elm, you
    * can do so by: elm[]->table.hashFunction->table.scanIndexed(#
    * ... do ... #)
    *)
   (# inx: @integer;
      where:< elementPredicate;
      current: ^element;
      start:< object;
      end:< object;
   enter theIndex->inx
   ...
   #);
 find::
   (#
   ...
   #);
 findIndexed:
   (* enters an index value, and seeks among the elements for the
    * first element, for which predicate holds true.  If such an
    * element is found, INNER is called, and then that element is
    * returned by findIndexed.  While INNER is invoked, current
    * refers to the element found.  Note that findIndexed behavies
    * somewhat similar to Find, except that findIndexed only scans
    * through the elements in the hashTable with the given hash
    * index (i.e. the collisions).  This implies that findIndexed
    * takes advantage of the fact that this is a hashTable,
    * meaning that findIndexed is a lot faster than Find.  Note
    * that if you want to find some element with the same index as
    * some known element, elm1, you can do so by:
    * elm1[]->table.hashFunction ->table.findIndexed(#
    * predicate::< (# ... do ... #) #); ->elm2[]
    *)
   (# inx: @integer;
      predicate:< elementPredicate;
      current: ^element;
      notFound:< Notification
        (#
        do 'Element not found in hashtable'->msg.putline;
           INNER
        #);
      start:< object;
      end:< object;
   enter theIndex->inx
   ...
   exit current[]
   #);
 statistics:
   (* calculates statistics on the current status of
    * this(hashTable).  Returns a histogram in the form of a
    * table, where each entry in the table contains the number on
    * collisions for that hash index.  Also returned are the
    * maximum, minimum and avegare number of collisions found in
    * the hashTable.  The local print pattern prints this
    * information on some stream
    *)
   (# histogram: [range] @integer;
      max, min, average, usedIndices: @integer;
      print:
        (# s: ^stream; i: @integer
        enter s[]
```

```
            do 'Histogram: '->s.putText;
               '('->s.put;
               1->i; histogram[i]->s.putInt;
               loop: (if i<range then
                         ','->s.put;
                         i+1->i; histogram[i]->s.putInt;
                         restart loop
                     if);
               ')'->s.putline;
               'Maximum Collisions: '->s.putText; max->s.putInt; s.newline;
               'Minimum Collisions: '->s.putText; min->s.putInt; s.newline;
               'Average Collisions: '->s.putText; average->s.putInt; s.newline;
               INNER
            #)
         do maxInt->min; minInt->max;
            (for i:range repeat
                 i->scanIndexed(# do histogram[i]+1->histogram[i] #)
            for);
            (for i:range repeat
                 (if histogram[i]>max then histogram[i]->max if);
                 (if histogram[i]<min then histogram[i]->min if);
                 (if histogram[i]>0 then
                     average + histogram[i]->average;
                     usedIndices+1->usedIndices
                 if);
            for);
            (if usedIndices = 0 then 0->average
             else average div usedIndices->average
            if);
            INNER
         exit (histogram, max, min, average, usedIndices)
         #);
      theCellType::< (* Private *)
         (# next: (* Private *) ^theCellType #);
      private:@...;
      doEnter::<
         (# containerType::<hashtable;
         ...
         #)
   do INNER
   #) (* hashTable *);

(*-------------------------------------------------------------------*)
(*--- extensibleHashTable-------------------------------------------*)
(*-------------------------------------------------------------------*)

ExtensibleHashTable: hashTable
   (* ExtensibleHashTable makes it possible to extend the range of
    * index values dynamically.  If the range of index value is
    * extended, the user of this(extensibleHashTable) needs to take
    * special care that the hash index of the existing objects in the
    * table is not changed, otherwise the table may not work properly.
    * In order to cope with hashFunctions that might be dependent on
    * the range of index values, extensibleHashTable defines a rehash
    * function which may be invoked to rearrange the hash indeces of
    * the existing elements in the table.  Note that rehash'ing the
    * table is a relatively costy operation, and should therefore only
    * be invoked when absolutely needed.  To make extending a table
    * safe from eventual changes in the hash indeces of elements
    * already in the tabel, the following is advised: table.extend(# do * table.rehash #) Define
    *)
   (# <<SLOT extensibleHashTableLib:attributes>>;
      extend:
         (* extends this(extensibleHashTable) with the given number of
          * index values
          *)
```

```
      (# increment: @integer
      enter increment
      ...
      #);
    rehash:
      (* takes all elements in this(extensibleHashTable) and
       * calculates new hash indexes for all of them
       *)
      (# ... #);
do INNER
#)
```

HashTable Interface Mjølner Informatics

Dictionary Interface

```
ORIGIN 'container';
BODY 'private/dictionaryBody'
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1997
 *       All rights reserved.
 *)
--- lib:attributes ---
dictionary: container
  (* A dictionary is an efficient data structure for storing associations
   * between two objects (the Key and the Entry).  The Key object acts as the
   * lookup value for the Entry object, such that the Entry object can be
   * located in the dictionary given the Key object.
   *
   * Defines the operations:
   *      hashFunction, associate, disassociate, lookup, scanAssociations
   *)
  (# <<SLOT dictionaryLib:attributes>>;

     key:< object
       (* The qualification of the Key objects of THIS(dictionary) *);
     keyEqual:< booleanValue
       (* Defines the equality test on key objetcs.  Users of
        * container patterns must further bind equal to contain the
        * proper equality test for the specified key type.  Default
        * equality test for equal references (i.e. the same object)
        *)
       (# left,right: ^key
       enter (left[],right[])
       do (left[] = right[])->value; INNER
       #);
     init::< (# ... #);
     clear::(# ... #);
     empty::(# ... #);
     size::(# ... #);
     has::(# ... #);
     find::(# ... #);
     copy::(# ... #);
     hashFunction:< integerValue
       (* Internally, the dictionary is organized as a hashTable, and this
        * pattern is invoked on all Keys defined in THIS(dictionary) to define
        * the location of the association.  The default behaviour will generate
        * many collisions, and specialization is therefore strongly advised
        *)
       (# k: ^key
       enter k[]
       do INNER
       #);
     associate:
       (* Associates the element 'e' with the key 'k' *)
       (# k: ^key; e: ^element
       enter (k[], e[])
       ...
       #);
     lookup:
       (* Returns the element associated with the key 'k'.  Returns
        * NONE of no associations exists with key 'k'
        *)
       (# k: ^key; e: ^element
       enter k[]
       ...
       exit e[]
```

```
      #);
    disassociate:
      (* Removes an possible association between key 'k' and element
       * 'e'.  Does nothing if no associateion exists.
       *)
      (# k: ^key; e: ^element
      enter k[]
      ...
      exit e[]
      #);
    scanAssociations:
      (* Scans this(dictionary).  For each assiciation in
       * this(dictionary), 'k' will refer to the key and 'e' will
       * refer to the associated element
       *)
      (# where:< elementPredicate(# k: ^key enter k[] do INNER #);
         k: ^key; e: ^element;
         start:< object;
         end:< object;
      ...
      #);
    theScanner::<
      (#
      ...
      #);
    storage: (* Private *) @...;
#)
```

ArrayContainer Interface

```
ORIGIN 'container';
BODY 'private/arrayContainerBody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-98
 *       All rights reserved.
 *)
--- lib: attributes ---
arrayContainer: container
  (* ArrayContainer is an abstraction of a repetition, offering
   * container capabilities as well as repetition capabilities.
   * Furthermore, arrayContainer implements three popular sorting
   * algorithms on the elements in the arrayContainer.  Defines the
   * operations:
   *      capacityInitial, capacityIncrement, capacityExtend,
   *      get, put, delete, bubbleSort, shellSort, quickSort
   *)
  (# <<SLOT arrayContainerLib: attributes>>;
     less:< booleanValue
       (* should be further bound to contain an ordering operation to
        * be used by the sorting algorithms.  Less is only used during
        * sorting, and can therefore be ignored if sorting is not
        * applied.
        *)
       (# left, right: ^element
       enter (left[],right[])
       do INNER
       #);
     init::< (# ... #);
     clear::< (# ... #);
     size::< (# ... #);
     capacity::< (# ... #);
     capacityInitial:< integerObject(# do 25->value; INNER #);
     capacityIncrement:< integerObject(# do 10->value; INNER #); (* -1 means double *)
     capacityExtend:< (# ... #);
     find::(# ... #);
     copy::(# ... #);
     has::(# ... #);
     put:<
       (* Takes an element and an index position and inserts the
        * element at that position
        *)
       (# elm: ^element; inx: @integer
       enter (elm[],inx)
       ...
       #);
     get:<
       (* Takes an index position and returns the element at that
        * position
        *)
       (# elm: ^element; inx: @integer
       enter inx
       ...
       exit elm[]
       #);
     delete:<
       (* Takes an index position and deletes the element at that
        * position (the deleted element is returned)
        *)
       (# inx: @integer; elm: ^element;
       enter inx
       ...
```

```
      exit elm[]
      #);
    bubbleSort:
      (* Takes an index position and sorts the elements in positions
       * [1: n], using the bubblesort algorithm
       *)
      (# n: @integer
      enter n
      do ...; INNER
      exit this(arrayContainer)[]
      #);
    shellSort:
      (* Takes an index position and sorts the elements in positions
       * [1: n], using the shellsort algorithm
       *)
      (# n: @integer
      enter n
      do ...; INNER
      exit this(arrayContainer)[]
      #);
    quickSort:
      (* Takes an index position and sorts the elements in positions
       * [1: n], using the quicksort algorithm
       *)
      (# n: @integer;
      enter n
      do ...; INNER
      exit this(arrayContainer)[]
      #);
    doEnter::<
      (# containerType::<arrayContainer
      ...
      #);
    theCellType::< (* Private *)
      (# occupied: @boolean;
         copy::< (#
                 do elm[]->theCellCopy.elm[];
                    occupied->theCellCopy.occupied; INNER
                 #)
      enter elm[]
      do INNER
      exit elm[]
      #);
    theScanner::<(# ... #);
    private: (* Private *) @...
  do INNER
  #)
```

List Interface

```
ORIGIN 'container';
LIB_DEF 'list' '../lib';
BODY 'private/listBody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-96
 *       All rights reserved.
 *)
--- lib: attributes ---
list: container
  (* This list pattern defines a double-linked list data structure.
   * Most operations either enters or exits a list position.  List
   * positions are references into a particular place in the list,
   * containing an element.  These positions are instances of
   * theCellType.
   *
   *     list
   *       cyclicList --- prev and next operations to impl. cyclic list
   *       priorityList --- elements ordered by priority
   *       recList --- allows for recursive lists (in separate fragment)
   *
   * Defines the following new operations: prepend, append, head,
   *     tail, last, preample, at, locate, concatenate, splitBefore,
   *     splitAfter, insertBefore, insertAfter, delete, deleteBefore,
   *     deleteAfter, scanReverse, iterate, iterateFrom,
   *     iterateReverse, iterateReverseFrom
   *)
  (# <<SLOT listLib: attributes>>;
     theCellType::<
       (* theCellType is the pattern from which the individual list
        * positions are created. It defines the succ and pred
        * references to the list positions immediately before/after
        * this position in the list (NONE means the end of the list
        * (either end).  The elm attribute refers to the element at
        * this position in the list.
        *)
       (# succ, pred: ^theCelltype;
       do INNER
       #);
     clear::< (# ... #);
     size::< (# ... #);
     empty::< (# ... #);
     has::
       (# ... #);
     copy::<(#
            ...
            #);
     prepend: (* insert elm as first element *)
       (# elm: ^element;
          position: ^theCellType
       enter elm[]
       ...
       exit position[]
       #);
     append: (* insert elm as last element *)
       (# elm: ^element;
          position: ^theCellType
       enter elm[]
       ...
       exit position[]
       #);
     head:
```

```
  (* returns the position element of the first position in the
   * list
   *)
  (# position: ^theCellType
    ...
  exit position[]
  #);
tail:
  (* returns a copy of THIS(list), except the first element *)
  (# lst: ^list
    ...
  exit lst[]
  #);
last:
  (* returns the position element of the last position in the
   * list
   *)
  (# position: ^theCellType
    ...
  exit position[]
  #);
preample:
  (* returns a copy of THIS(list), except the last element *)
  (# lst: ^list
    ...
  exit lst[]
  #);
concatenate:
  (* returns a new list, containing the concatenated list *)
  (# otherList, newlist: ^list;
  enter otherList[]
  ...
  exit newlist[]
  #);
insertBefore: (* if position=NONE, insert elm as last element *)
  (# elm: ^element; position, newPosition: ^theCellType
  enter (elm[], position[])
  ...
  exit newPosition[]
  #);
insertAfter: (* if position=NONE, insert elm as first element *)
  (# elm: ^element; position, newPosition: ^theCellType
  enter (elm[], position[])
  ...
  exit newPosition[]
  #);
delete: (* if position=NONE, delete nothing *)
  (# deletedPosition, position: ^theCellType;
     empty:< emptyContainer
  enter position[]
  ...
  exit deletedPosition[]
  #);
deleteBefore:
  (* if position=NONE, delete last position element *)
  (# deletedPosition, position: ^theCellType;
     empty:< emptyContainer
  enter position[]
  ...
  exit deletedPosition[]
  #);
deleteAfter:
  (* if position=NONE, delete first position element *)
  (# deletedPosition, position: ^theCellType;
     empty:< emptyContainer
  enter position[]
```

```
  ...
  exit deletedPosition[]
  #);
splitBefore:
  (* splits this(list) into two lists, where the elements before
   * position (excluding position) is placed in preList and the
   * rest of this(list) is placed in postList.  PositionElm will
   * be the head of postList.  If position=NONE, preList will
   * become a copy of the entire list and postList will become
   * NONE
   *)
  (# position: ^theCellType;
     preList, postList: ^list;
     listSplitBeforePrivate: @...
  enter position[]
  do listSplitBeforePrivate; INNER
  exit (preList[], postList[])
  #);
splitAfter:
  (* splits this(list) into two lists, where the elements before
   * position (including position) is placed in preList and the
   * rest of this(list) is placed in postList. PositionElm will
   * be the last element in postList If position=NONE, preList
   * will become a copy of the entire list and postList will
   * become NONE
   *)
  (# position: ^theCellType;
     preList, postList: ^list;
     listSplitAfterPrivate: @...
  enter position[]
  do listSplitAfterPrivate; INNER
  exit (preList[], postList[])
  #);
at: (* returns the position of elm in the list *)
  (# elm: ^element;
     position: ^theCellType
  enter elm[]
  ...
  exit position[]
  #);
locate:
  (* returns the position of the element in the list, satisfying
   * predicate.  This operation is similar to find, except that
   * it returns the position, not the element.
   *)
  (# predicate:< cellPredicate;
     notFound:< Notification
       (#
       do none->position[];
          'Element not found in list'->msg.putLine;
          INNER
       #);
     start:< object;
     end:< object;
     position: ^theCellType;
  ...
  exit position[]
  #);
find::
  (#
  ...
  #);
thescanner::< (* private *)
  (# theCell:^theCelltype;
     notdone:@boolean;
  ...
```

```
  #);
theReverseScanner:< (* private *)
  (#
     w:^elementPredicate;
     s:^scanreverse;
     aCell:^theCellType;
     notdone:@boolean;
  enter (s[],w[])
  ...
  #);
theFromScanner:< (* private *)
  (#
     w:^elementPredicate;
     s:^scanFrom;
     aCell:^theCellType;
     notdone:@boolean;
  enter (s[],w[],aCell[])
  ...
  #);
scanReverse:
  (* similar to scan, except that it scans the list in reverse
   * direction
   *)
  (# first:@boolean;
     where:< elementPredicate;
     current: ^element;
     start:< object;
     end:< object;
  ...
  #);
scanFrom:
  (* similar to scan, except that it scans the list from the
   * given position
   *)
  (#
     first:@boolean;
     where:< elementPredicate;
     current: ^element;
     start:< object;
     end:< object;
     position:^theCelltype; (* start cell *)
  enter position[]
  ...
  #);
scanReverseFrom:
  (* similar to scanReverse, except that it scans the list in
   * reverse direction from the given position
   *)
  (# where:< elementPredicate;
     position: ^theCellType;
     current: ^element;
     theCell:^theCelltype;
     start:< object;
     end:< object;
  enter position[]
  ...
  #);
iterate:
  (* similar to scan, except that "current" refers to the
   * current cell in the list.  If furthermore the list is an
   * instance of the recList subpattern, iterate respects the
   * sublist structure by not scanning the elements in the
   * sublists (as scan does). I.e. during an iterate of a
   * recList, "current" may refer to a cell containing either an
   * element, or a sublist.
   *)
```

```
      (# where:< cellPredicate;
         current: ^theCellType;
         start:< object;
         end:< object;
         listIteratePrivate: @...
      ...
      #);
    iterateFrom:
      (* similar to iterate, except that it takes a position, and
       * starts the iteration from that position.
       *)
      (# where:< cellPredicate;
         position: ^theCellType;
         current: ^theCellType;
         start:< object;
         end:< object;
         listIterateFromPrivate: @...
      enter position[]
      ...
      #);
    iterateReverse:
      (* similar to iterate, except that it iterates backwards.
       *)
      (# where:< cellPredicate;
         current: ^theCellType;
         start:< object;
         end:< object;
         listIterateReversePrivate: @...
      ...
      #);
    iterateReverseFrom:
      (* similar to iterateReverse, except that it takes a position,
       * and starts the reverse iteration from that position.
       *)
      (# where:< cellPredicate;
         position: ^theCellType;
         current: ^theCellType;
         start:< object;
         end:< object;
         listIterateReverseFromPrivate: @...
      enter position[]
      ...
      #);
    cellPredicate: booleanValue
      (* This pattern is used as the superpattern for the predicates
       * in the locate and iterate operations.
       *)
      (# current: ^theCellType
      enter current[]
      do true->value;
         INNER
      #);
    doEnter::<
      (# containerType::<list;
         doneInInner:@boolean
      ...
      #);
    private:@...;
  do INNER
  #) (* list *);

cyclicList: list
  (* Add operations next and prev to enable traverse the list as a
   * cycle list
   *)
  (# theCellType::<
```

```
      (# prev:
           (# position: ^theCellType
           do pred[]->position[];
              (if position[]=NONE then last->position[] if);
           exit position[]
           #);
         next:
           (# position: ^theCellType
           do succ[]->position[];
              (if position[]=NONE then head->position[] if);
           exit position[]
           #);
      #);
  #);

priorityList: list
  (* implements a priority list where the elements added to the list
   * are ordered according to their priority (implemented by the
   * ordering relation 'less')
   *)
  (# less:< booleanValue
       (# left,right: ^element enter (left[],right[]) do INNER #);
     add:
       (# elm: ^element; added: @boolean
       enter elm[]
       do loop:
            iterate
            (#
            do (if (elm[],current.elm[])->less then
                   (elm[],current[])->insertBefore;
                   true->added;
                   leave loop
               if)
            #);
         (if not added then elm[]->append if)
       #)
  #)
```

RecList Interface

```
ORIGIN 'list';
BODY 'private/recListBody';
(*
 * COPYRIGHT
 *
 *       Copyright (C) Mjolner Informatics, 1992-94
 *
 *       All rights reserved.
 *)
--- lib: attributes ---
recList: list
  (* This recList pattern extends the list pattern to allow for
   * sublists.  Most operations either enters or exits a list
   * position.  List positions in a recList may contain either an
   * element or a sublist.  Defines the following new operations:
   *
   *     insertSublistBefore, insertSublistAfter,
   *)
  (# <<SLOT recListLib: attributes>>;
     theCellType::<
       (* theCellType is the pattern from which the individual
        * recList positions are created.  It also defines the when
        * attribute which is used for accessing either the element or
        * the sublist at this position.  By further binding element,
        * respectively sublist, in when, the actions to be executed in
        * either case, may be specified.
        *)
       (# sublist: ^list;
          when:
            (* to differenciate between this(theCellType) holding an
             * element or a sublist
             *)
            (# elm:< (# elm: ^element
                        ...
                     exit elm[]
                     #);
               sublist:< (# lst: ^list
                            ...
                         exit lst[]
                         #);
               recListCellWhenPrivate: @...
            do INNER; recListCellWhenPrivate
            #);
          copy::< (#
                  do (if sublist[] <> NONE then
                         sublist.copy->theCellCopy.sublist[]
                     if)
                  #)
       do INNER
       #);
     insertSublistBefore:
       (* if position=NONE, insert lst as last element *)
       (# lst: ^list; position, newPosition: ^theCellType;
          illegalSublist:< illegalSublistException
       enter (lst[], position[])
       ...
       exit newPosition[]
       #);
     insertSublistAfter:
       (* if position=NONE, insert lst as first element *)
       (# lst: ^list; position, newPosition: ^theCellType;
          illegalSublist:< illegalSublistException
```

```
      enter (lst[], position[])
      ...
      exit newPosition[]
      #);
    illegalSublistException: Exception
      (#
      do 'The element of this sublist does not share '->msg.putText;
         'qualifications with the element type of the list'->msg.putLine;
         INNER
      #);
    theScanner::<
      (# ...
      #);
    theReverseScanner::<
      (# ...
      #);
    theFromScanner::<
      (# ...
      #);
    doEnter::<
      (# containerType::<recList;
      ...
      #);
  do INNER
  #)
```

SeqContainers Interface

```
ORIGIN 'container';
BODY 'private/seqContainersBody';
(*
 * COPYRIGHT
 *       Copyright (C) Mjolner Informatics, 1992-94
 *       All rights reserved.
 *)
--- lib: attributes ---
sequentialContainer: container
  (* sequentialContainer is an abstract superpattern and the
   * currently available subpatterns are (indentation specifies
   * specialization):
   *     sequentialContainer
   *       stack
   *       queue
   *       deque
   *)
  (# <<SLOT sequentialContainerLib: attributes>>;
     theScanner::<
       (#
       ...
       #);
     size::
       (#
       ...
       #);
     clear::
       (#
       ...
       #);
     empty::
       (#
       ...
       #);
     has::<
       (#
       ...
       #);
     theCellType::< (* Private *)
       (# succ, pred: (* Private *) ^theCellType;
       do INNER #);
     private:@...;
  do INNER
  #) (* sequentialContainer *);

(*-------------------------------------------------------------*)
(*--- stack-------------------------------------------------*)
(*-------------------------------------------------------------*)

stack: sequentialContainer
  (* Stack is an ordinary stack data structure.  Defines the
   * operations: top, push, pop
   *)
  (# <<SLOT stackLib: attributes>>;
     find::
       (#  privateFinder:@...;
       ...
       #);
     copy::<
       (#
       ...
       #);
```

```
      top: (* Returns the topmost element on THIS(stack) *)
        (# elm: ^element; empty:< emptyContainer
          ...
        exit elm[]
        #);
      push:
        (* Takes an element and places it on the top of THIS(stack) *)
        (# elm: ^element
        enter elm[]
        ...
        #);
      pop:
        (* Returns the topmost element, and removes it from the stack
         *)
        (# elm: ^element; empty:< emptyContainer
          ...
        exit elm[]
        #);
      doEnter::<
        (# containerType::<stack;
        ...
        #)
  do INNER
  #) (* stack *);

(*-------------------------------------------------------------*)
(*--- queue -------------------------------------------------*)
(*-------------------------------------------------------------*)

queue: sequentialContainer
  (* Queue is an ordinary queue data structure.  Defines the
   * operations: front, insert, remove
   *)
  (# <<SLOT queueLib: attributes>>;
     has::<
       (#
       ...
       #);
     find::
       (#  privateFinder:@...;
       ...
       #);
     copy::<
       (#
       ...
       #);
     front: (* Returns the element in the front of THIS(queue) *)
       (# elm: ^element; empty:< emptyContainer
         ...
       exit elm[]
       #);
     insert:
       (* Takes an element and inserts it at the end of THIS(queue) *)
       (# elm: ^element
       enter elm[]
       ...
       #);
     remove:
       (* Returns the element in front, and removes it from the queue
        *)
       (# elm: ^element; empty:< emptyContainer
         ...
       exit elm[]
       #);
      doEnter::<
       (# containerType::<queue;
```

```
      ...
      #)
    (*     private:@...;*)
  do INNER
  #) (* queue *);

(*-------------------------------------------------------------*)
(*--- deque -------------------------------------------------*)
(*-------------------------------------------------------------*)

deque: sequentialContainer
  (* Deque is a double-ended queue in which elements may be inserted
   * and removed from both ends of the queue.  Defines the operations:
   * front, insertFront, removeFront, back, insertBack, removeBack
   *)
  (# <<SLOT dequeLib: attributes>>;
     front: (* Returns the element in the front *)
       (# elm: ^element; empty:< emptyContainer
         ...
       exit elm[]
       #);
     insertFront: (* Takes an element and inserts it at the front *)
       (# elm: ^element
       enter elm[]
       ...
       #);
     removeFront:
       (* Remove and return the element in front of THIS(deque) *)
       (# elm: ^element; empty:< emptyContainer
         ...
       exit elm[]
       #);
     back: (* Returns the element in the back *)
       (# elm: ^element; empty:< emptyContainer
         ...
       exit elm[]
       #);
     insertBack: (* Takes an element and inserts it at the back *)
       (# elm: ^element
       enter elm[]
       ...
       #);
     removeBack: (* Returns the element in the back, removing it *)
       (# elm: ^element; empty:< emptyContainer
         ...
       exit elm[]
       #);
      doEnter::<
       (# containerType::<deque;
       ...
       #)
  do INNER
  #) (* deque *);

(* These are to different priorityqueues! *)

(*-------------------------------------------------------------*)
(*--- prioQueue ---------------------------------------------*)
(*-------------------------------------------------------------*)

prioQueue: sequentialContainer
  (* PrioQueue is a priority queue, in which the elements are kept in
   * SEPARATE queues for each priority.  Defines the operations:
   * front, insert, remove, scanPriority.
   *)
  (# <<SLOT prioQueueLib: attributes>>;
```

```
      theScanner::<
        (#
        ...
        #);
      front: (* Returns the element in front with the given priority *)
        (# prio: @integer; elm: ^element; empty:< emptyContainer
        enter prio
        ...
        exit elm[]
        #);
      insert:
        (* Insert an element in the queue with the given priority *)
        (# prio: @integer; elm: ^element
        enter (elm[],prio)
        ...
        #);
      remove:
        (* Remove and returns the front element with the given
         * priority
         *)
        (# prio: @integer; elm: ^element; empty:< emptyContainer
        enter prio
        ...
        exit elm[]
        #);
      scanPriority:
        (* scans through all elements of the given priority *)
        (# prio: @integer;
           where:< elementPredicate;
           current: ^element;
           start:< object;
           end:< object;
        enter prio
        ...
        #);
      copy::<
        (#
        ...
        #);
      theCellType::< (* Private *)
        (# prio: @integer;
           theQueue: ^queue;
           copy::< (#
                   do prio->theCellCopy.prio;
                      (if theQueue[] = NONE then
                       else
                          theQueue.copy->theCellCopy.theQueue[]
                      if)
                   #)
        do INNER #);
      doEnter::<
        (# containerType::<prioQueue;
        ...
        #)
    do INNER
    #) (* prioQueue *);

(*-------------------------------------------------------------------*)
(*--- priorityQueue -------------------------------------------------*)
(*-------------------------------------------------------------------*)
PriorityQueue:container
    (* This implements a general priorityqueue.
     * O(log n) time for insert and deletemin. (list based balanced treee)
     * Does not require elments to have and explicit
     * priority. It is based purely on the less attribute.
     * Futherbind element and less to use your own priotities
```

```
   * Note this redefines basic attributes!
   *)

(# <<SLOT priorityQueueLib: attributes>>;
   less:<booleanValue
     (# e1,e2:^element;
     enter (e1[],e2[])
     do INNER
     #);

   insert:
     (# elm: ^element;
        doInsert: @...;
     enter elm[]
     do doInsert
     #);

   deleteMin:
     (# elm: ^element;
        doDeleteMin: @...;
     do doDeleteMin;
     exit elm[]
     #);

   min:
     (# elm: ^element;
     ...
     exit elm[]
     #);

   scan:
     (# current: ^element;
     ...
     #);

   init::<(# ... #);

   clear:<(# ... #);
   size:IntegerValue
     (# ... #);
   empty:<BooleanValue
     (# ... #);
   has:<BooleanValue
     (# elm:^element;
     enter elm[]
     ...
     #);

   emptyContainer:< exception;
   storage: @...;
#)
```