The class Priority Queue

The class has the following interface or specification

```
class PRIORITY_QUEUE [G -> COMPARABLE]
creation
     make
feature
     count_PQ : INTEGER -- #items in the Priority_Queue
     make -- create a Priority_Queue of default size
     empty : BOOLEAN  -- Is the Priority_Queue empty
                                    -- in effect,  is count_PQ = 0
     add (x : G)  -- Insert x into the Priority_Queue
     item : G  -- Get ‘largest’ item from the Priority_Queue
     remove       -- Remove ‘largest’ item from Priority_Queue
end -- PRIORITY_QUEUE
```

A Priority_Queue is in effect a suite or bag of items such that we can get and remove the largest item. A suite or bag of items is a 'set' where repeated items are allowed.

The Priority_Queue is useful in applications such as print queues, or any application where there is a priority on the items, e.g. a print queue where less pages are given higher priority.

Given the class PRIORITY_QUEUE we can implement a (not in-place) **sorting** routine as follows

Assume a class attribute A:ARRAY[G]

```
sort  is
      local
            k :INTEGER
            PQ : PRIORITY_QUEUE
      do
            !!PQ.make
            from
                  k := A.lower
            until
                  k > A.upper
            loop
                  PQ.add(A.item(k))
                  k := k+1
            end
      -- all array items put into Priority_Queue
            from
                  k := A.upper
            until
                  k < A.lower
            loop
                  A.put(PQ.item, k)
                  PQ.remove
                  k := k-1
            end
      -- the largest items are successively put back into A
      end -- sort
```

Efficient Implementation of PRIORITY_QUEUE

The above version of sort is not an **in-place** sort as the items in the array are ‘copied’ into a Priority_Queue and then ‘copied’ back out again in the correct order. By implementing a Priority Queue via a Heap the sort will have an O(n*log n) performance.

Heap implementation of PRIORITY_QUEUE

In the hidden part of the class assume we have an array

Heap : ARRAY[G]

Also let

default_size : INTEGER is 16

We will extend the array when necessary.

Implementing remove

In order to implement remove we will need the procedure Heapify, but in this context we will rename it to Sift_Down as we will also have a procedure Sift_Up.

The procedure, remove, over-writes the item at index 1 with the last item (the item at index count_PQ) and 'sifts down' this item. If the array gets too small it is re-sized.

```
Sift_Down (i,j:INTEGER) is       --Heapify segment A(i .. j)
local
    k : INTEGER
do
    k := 2*i
    if  k <= j then --  if k not a leaf in Heap(i .. j)
        if k < j and then
            Heap.item(k) < Heap.item(k+1) then
                k := k+1
        end
        if Heap.item(i) < Heap.item(k) then
            Exchange (i,k)
            Sift_Down (k,j) -- Heapify ‘subtree’ Heap(k .. j)
        end
    end
end -- Sift_Down
```

```
remove is -- Remove ‘largest’ item from the Priority_Queue
require
    Non_Empty_Q : not empty
do
    Heap.put(Heap.item(count_PQ), 1)
    count_PQ := count_PQ -1
    if count_PQ > 1 then
        Sift_Down(1, count_PQ)
    end
    if count_PQ < Heap.capacity // 4 and then
        Heap.capacity > default_size then
            Heap.grow(Heap.capacity // 2)
    end
end -- remove
```

Eiffel allows one to extend or contract arrays, if needed. If #items in the Priority_Queue is small relative to the capacity of the Heap array, then reduce Heap capacity.

Implementing add

To implement add, we need the procedure Sift_Up which, given an array with the Heap property, will add an item to the heap by 'sifting up' the item to its proper position.

```
Sift_Up (n:INTEGER) is
-- From below, put item at n into proper position
    local
        it:G
        k:INTEGER
    do
        from
            it := Heap.item(n)
            k := n
        until
            k=1 or else Heap.item(k // 2) >= it
        loop
            Heap.put(Heap.item(k // 2), k)
            k := k // 2
        end
        Heap.put(it, k)
    end -- Sift_Up
```

```
add (x:G) is
    do
        count_PQ := count_PQ+1
        if count_PQ > Heap.capacity then
            Heap.automatic_grow
        end         -- make (50%) more space  in Heap, if needed
        Heap.put(x, count_PQ)
        Sift_Up(count_PQ)
    end --add
```

Implementing make

Initialise, the array, Heap, to capacity of default_size. The Prioity_Queue is initially empty.

```
    make is
        do
            !!Heap.make(1, default_size)
            count_PQ := 0
        end -- make
```

Implementing a Priority Queue by a (linked) List

Implementing a Priority Queue by an array with the 'Heap Property' is very efficient except that the array may have to be resized.

Implementation by a linked list is possible. By maintaining the list in descending order, the largest item will be at the front of the list. In this case removing an item is very efficient but adding an item is more expensive as each insertion of an item into a ordered list will perform at O(n).

In implementing Heapsort, it will be more effort, $O(n^2)$, to built the queue, while the sorting phase will be $O(n)$.

Note:

By keeping the list ordered, we have already sorted the items except that that the order is reversed. It is possible to define heaps where the ‘smallest’ element is the one looked for. In using lists to store items that are comparable, it may be an advantage to keep the list of items always sorted when adding and removing items.