THE COMPOSITION OF FAST PRIORITY QUEUES

Jan van Leeuwen
Department of Computer Science
University of Utrecht
3508 TA Utrecht, the Netherlands

Technical Report RUU-CS-78-5

May 1978

Department of Computer Science
University of Utrecht
3508 TA Utrecht, the Netherlands

all correspondence to:

Dr. Jan van Leeuwen
Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

THE COMPOSITION OF FAST PRIORITY QUEUES*

Jan van Leeuwen

Abstract. Priority queues are datastructures which effectively support INSERT, DELETE, MIN and UPDATE commands in some application environment. In order to achieve fast response times, priority queues have often been designed from balanced or binomial tree structures. We present additional techniques for structuring priority queues which will be useful (or otherwise entertaining) to programmers. First we characterize a class of common algorithms which perform MIN - DELETE - INSERT cycles, where the new element to be inserted is generated by some monotone function from the extracted min. The class includes e.g. Huffman's construction of minimum redundancy codes. We show that the algorithms can be implemented using a simple priority queue structure whose response time is bounded by a constant for each of the commands involved. Next, in a more general approach, we consider ways of composing fast priority queues from simpler, more conventional sub-queues. We obtain a priority queue, for instance, whose response time for INSERT commands is always bounded by a constant. DELETE and MIN commands may take O(log n) time, where n is the number of elements in the queue. The same approach leads to another priority queue organization (which is less efficient in space), whose response time for MIN commands is always bounded by a constant at the expense of a moderate increase in the time for INSERT commands (essentially to loglog n). The constructions shed new light on the structure of binomial queues, and illustrate various techniques and current ideas used for efficient datastructuring.

1. Introduction

In recent years many new, efficient data-organizations have been devised for the benefit of combinatorial algorithms which require extensive data manipulation during execution (see Knuth [3] or Aho, Hopcroft and Ullman [1]). In most instances, fast priority queues were obtained as a "corollary" to some data-structuring technique of potentially greater applicability. In this paper we shall attempt to approach the inherent structure of priority queues more directly, by analysing a scheme which is fundamental to virtually all priority queues.

* These notes were prepared for a tutorial presentation in the Workshop on "Fast Algorithms" at the GI Jahrestagung, Berlin, October 5, 1978.

Priority queues are datastructures which effectively support INSERT, DELETE, MIN and UPDATE commands. The problem of designing efficient priority queues has been familiar to programmers for many years. Currently, priority queues are designed from some underlying balanced or binomial tree structure to give an O(log n) worst case response time for the basic commands. A typical, conventional priority queue can process INSERT, DELETE and UPDATE commands in O(log n) time while MIN commands can be answered instantly, i.e., in time bounded by a constant. We shall attempt to lower the worst case time bound for some commands to improve the (worst case) performance, perhaps at the expensive of space.

Relaxing the constraint of minimum storage can affect the practicality of our approach to some extent. Priority queues in systems applications tend to be of limited size and some extra pointer-space per element may not be disastrous (on the other hand, no sophisticated queue structure may be needed either), but blowing up the memory requirement by some constant factor may not be feasible for any of the algorithms which use potentially unbounded priority queues. See Vuillemin [10] for a listing of various software tasks for which "large" priority queues are required. Indeed, Vuillemin's binomial queues [10] may be the most efficient priority queue structure from a combined storage and time performance viewpoint. It will not be surprising that in optimizing time we loose on the other end.

In section 2 we shall demonstrate that a general priority queue may not always be the best tool in an application where INSERT, DELETE and MIN commands are used. We characterize a class of fairly common algorithms which perform MIN - DELETE - INSERT cycles, where preprocessing of initial data makes it possible to proceed with a simple, "tuned" priority queue whose response time remains bounded by a constant for each command issued during execution! Applications include an alternative implementation of Huffman's algorithm (van Leeuwen [7]) and a linear time procedure for generating a particular data-compression code of Verhoeff (van Leeuwen [8]).

In section 3 we shall investigate a general scheme for composing fast priority queues from simpler, more conventional subqueues. The scheme has substantial similarities to the structure of Vuillemin's binomial queues, but its more abstract appearance makes it easier to devise local optimizations during the execution of commands. The particular optimization we strive for concerns the elimination of much of the internal restructuring triggered in e.g. binomial queues upon the insertion of a new element. We obtain a priority queue whose response time for INSERT commands is reduced to a constant, whereas the time for MIN commands remains at O(log n). A

simple addition to the scheme results in a modified priority queue structure in which the response time for MIN commands goes down to a constant (as for conventional queues), at the expense of only a moderate increase (to loglog n) in the response time for INSERT commands. However, the response time for INSERTs will generally be much smaller than that.

The results have been presented largely to show what one can do with priority queue structure. We believe that the techniques will be inspiring (or otherwise entertaining) to programmers who need to design a priority queue in software. We have tried to develop the ideas of this paper in a way which requires virtually no technical background. The paper can be considered as being to large extent a tutorial on the structure and further tuning of binomial queues, although we shall actually be using a modified version of them which appears to perform more elegantly. In general, the constructions and tools used illustrate several of the recent developments in the design of efficient datastructures.

2. A useful, special priority queue structure

We shall describe a construction principle involving MIN - DELETE - INSERT cycles, which can be recognized in a number of algorithms. It is commonly implemented using a general priority queue. We have observed in the context of some diverse applications (van Leeuwen [7, 8]) that for the particular kind of algorithms where it occurs a much simpler datastructure can be used, with response times bounded by a constant for all commands involved (perhaps only so after preprocessing the initial data). We shall explain it here to demonstrate that a priority queue doesn't always have to be a "complicated" data-structure. Chances are that the same technique has been applied by some clever programmer before, but we have found no record of an attempt to characterize a general class of algorithms where the same implementation can be applied with profit.

The type of construction we are about to describe seems to occur often in algorithms which try to minimize "something". There is an initial pool of data from some space V, a function f from V^p to V^q (some fixed p and q) and a loop of the following global form

```
repeat
    [pick p of the currently smallest items]
        for i from 1 to p do begin u_i ← MIN (pool);
                                   DELETE (u_i, pool)
                           end;
```

```
        [construct q new items]
            (y_1,...,y_q) ⇐ f(u_1,...,u_p);
        [insert the new items in the pool]
            for i from 1 to q do INSERT (y_i,pool)
    until <no more>;
```

The cycle is repeated until the pool is exhausted (typically when $p > q$) or until a predetermined number of iterations is exceeded. In between any of the steps shown some processing of data may take place, provided that no x_i or y_j is changed and no additional data transfers to or from the "pool" take place in the meantime. We shall refer to the program-segment as the "fundamental loop". It is represented again in Figure 1.

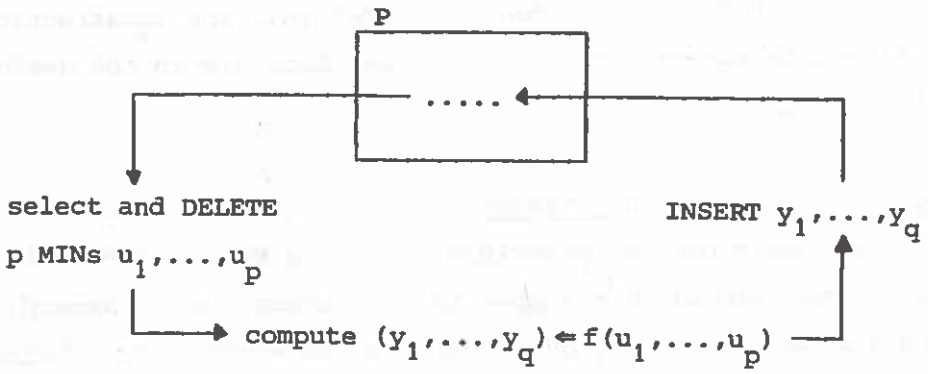


Figure 1. The fundamental loop.

The data-pool (P) is left unstructured, but obviously a general priority queue will do for the intended operations. We may be able to improve on it when more is known about f. Let π_i be the projection onto the i^{th} coordinate.

Notation. For tuples $\bar{u} = (u_1,\ldots,u_p)$ let $\bar{u}_{min} = \min\{u_i \mid 1 \le i \le p\}$ and $\bar{u}_{max} = \max\{u_i \mid 1 \le i \le p\}$.

Definition. A function f is said to be H-monotone if and only if for all $\bar{u},\bar{v}$

(a) $\bar{u}_{max} < \pi_i f(\bar{u})$, for all i and

(b) $\bar{u}_{max} < \bar{v}_{min} \Rightarrow \pi_i f(\bar{u}) < \pi_i f(\bar{v})$, for all i.

Note that we allow any feasible ordering < on V to be used in the definition. It will all become less mysterious in the following examples.

Example 2.1. Huffman's algorithm (see Knuth [4], p. 402) basically consists of a fundamental loop with a function f defined by $f(u_1,u_2) = u_1+u_2$. This function is H-monotone, using $\le$ for <.

Example 2.2. Verhoeff's algorithm for generating a particular data-compression code (see [9]) consists of a fundamental loop which employs the function $f(u) = (p_1.u,\ldots,p_k.u)$, where p_1 to p_k are certain numbers between 0 and 1.

This function is H-monotone also, now using ≥ for <.

We claim that for fundamental loops involving an f which is H-monotone, the data-pool can be structured so as to allow for more efficient command-processing than a general priority queue can. The structure consists, in fact, of a combination of ordinary queues as shown in Figure 2. We shall enter the initial data in sorted order in Q_0. The implementation of the fundamental loop for H-monotone f will be such that all queues remain sorted, throughout the course of the procedure.

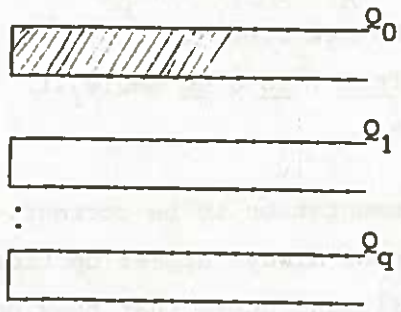


Figure 2.

Queues are unsophisticated datastructures which efficiently support the primitives of Figure 3. We assume that elements in a queue are stored simply as a linked list, with a queue descriptor keeping track of head and tail.

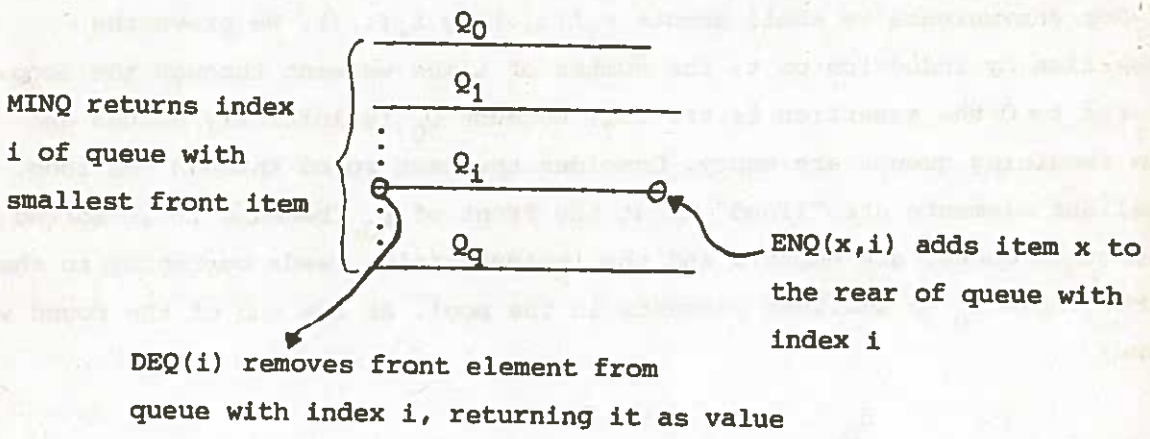


Figure 3.

Theorem 2.3. Let f be H-monotone. The fundamental loop can be implemented in such a way that the response times are O(q) for each MIN, O(1) for each DELETE and O(1) for each INSERT as encountered during the execution of the loop.

Proof

The plot has already been given away. The idea is to store the q elements generated in a step in their "own" queue. More specifically, we can implement

the loop as follows. Watch the implementation of selecting a next set of smallest elements.

```
repeat
    [pick p of the currently smallest items]
      for i from 1 to p do begin index⇐MINQ;
                                 u_i ⇐ DEQ(index)
                           end;
    [construct q new items]
      (y_1,...,y_q) ⇐ f(u_1,...,u_p);
    [insert the new items in the pool]
      for i from 1 to q do ENQ(y_i,i)
until <no more>;
```

In order for this implementation to be correct, we must show that the smallest elements of the pool always appear up front in the queues. Clearly, the H-monotonicity of f will guarantee that they do! We shall prove a slightly stronger assertion ("invariant"), which must hold each time after completing another "round" through the loop:

(i) the elements DELETEd in the last round are all less than or equal to each of the elements remaining in or just added to the queues,

(ii) the queues are sorted.

For convenience we shall denote $\pi_i f(...)$ by $f_i(...)$. We prove the assertion by induction on t, the number of times we went through the loop.

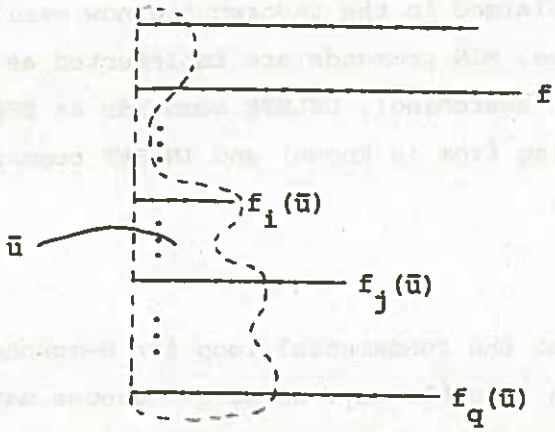
For $t = 0$ the assertion is trivial, because Q_0 is initially sorted and the remaining queues are empty. Consider the next round through the loop. The smallest elements are "lined" up at the front of Q_0 (because it is sorted and the other queues are empty), and the implementation leads correctly to the first tuple u_0 of smallest elements in the pool. At the end of the round we have:

$$\underbrace{\bar{u}_0}$$
$$f_1(\bar{u}_0)$$
$$\vdots$$
$$f_q(\bar{u}_0)$$

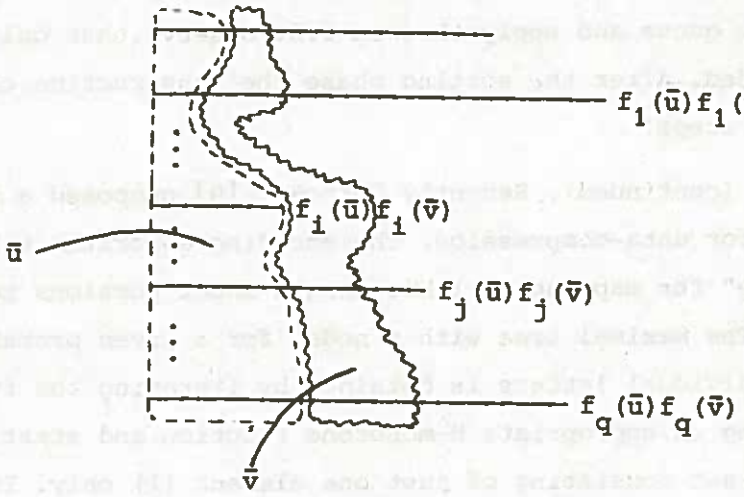
Note that (i) holds because $(\bar{u}_0)_{max} < f_i(\bar{u}_0)$, by definition of H-monotonicity. (This particular case of the definition is needed only for starting consistently.) The queues are obviously sorted, because Q_0 remained so and the other

queues have just one element. Hence, the assertion is valid for $t = 1$.

Now suppose the inductive assertion is valid after the t^{th} round through. Suppose we have just picked and DELETEd a tuple $\bar{u}$ and updated the queues to



The area enclosed by dotted lines indicates where the $\bar{u}$-elements were taken from. Note that by inductive hypothesis, $\bar{u}_{max}$ is less than or equal to all elements to the right of the dotted contour. As all queues are sorted (by hypothesis), the next tuple of smallest elements $\bar{v}$ can be selected off the front of the queues as implemented!



The figure indicates a typical contour for $\bar{v}$ (deletion of the selected elements could momentary empty a queue). Since $\bar{u}_{max}$ is less than or equal to all elements where $\bar{v}$ was selected from, i.e. $\bar{u}_{max} < \bar{v}_{min}$, we have $f_i(\bar{u}) < f_i(\bar{v})$. Clearly, all queues remain sorted! We have to be a bit more careful in ascertaining the validity of (i) this time. If a queue was not emptied entirely after deleting $\bar{v}$, then a front element remained which must be greater than or equal to $\bar{v}_{max}$. Hence, all elements in this queue (the added element included) must be greater than or equal to $\bar{v}_{max}$. If a queue

got emptied upon deleting $\bar{v}$, then we have no "element of reference" left and we cannot conclude (i) by the ordering of the queue. Fortunately, $\bar{v}_{max} < f_i(\bar{v})$ by definition of H-monotonicity and we are saved. It proves the induction hypothesis for t+1.

The response times claimed in the theorem are now easily verified. Using the implementation above, MIN commands are implemented as MINQ's (requiring O(q) by straightforward searching), DELETE commands as DEQ's (O(1) because the queue you're deleting from is known) and INSERT commands as ENQ's (taking O(1) each also).

□

The implementation of the fundamental loop for H-monotone functions is simple, but it works in a subtle way. Using q+1 queues may seem esoteric, but it is easy to compress the entire structure in one array. Let us indicate some typical applications where the approach is vindicated.

Example 2.1. (continued). The construction of a Huffman-tree of n weighted items is usually implemented by means of a general priority queue structure (see e.g. Knuth [5]). We could now proceed differently (van Leeuwen [7]). Just sort the n given items by some dedicated routine, store the resulting sequence in a queue and apply theorem 2.3. Observe that only one extra queue is needed. After the sorting phase the construction can be completed in O(n) more steps!

Example 2.2. (continued). Recently Verhoeff [9] proposed a new type of fixed length code for data-compression. The encoding algorithm makes use of a "maximal tree" for mapping variable-length input portions to fixed-length code words. The maximal tree with n nodes for a given probability distribution of individual letters is obtained by iterating the fundamental loop n-times, using an appropriate H-monotone function and starting from an initial data-set consisting of just one element (1) only. It was shown (van Leeuwen [8]) that this algorithm is essentially unique for obtaining maximal n-trees. Applying theorem 2.3 it follows that "Verhoeff-codes" with n code-words can be generated in only O(n) time, faster than Huffman-codes of comparable size.

Although it doesn't appear to be of great practical relevance, one may wonder how well the implementation actually performs when p and q are large (or, perhaps, when just q is large). It is immediate from the program that much time may get wasted on MIN selection, to a total of O(p.q) each round through. We shall see that it can be reduced to O(p.log q) for each round,

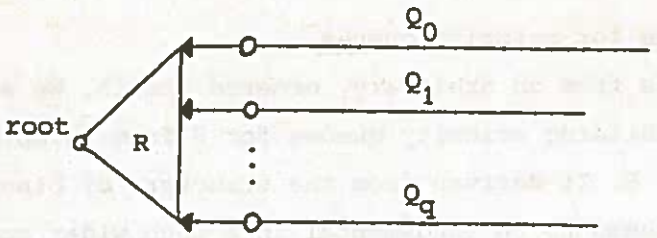


Figure 4.

at the expense of a slight increase in the time needed to INSERT new items. Just observe that we constantly perform MIN, DELETE and INSERT (or rather, UPDATE) operations on the front positions of the queues! It means that we'd better connect these positions into a dynamic priority queue R (see Figure 4).

Theorem 2.4. Let f be H-monotone. The fundamental loop can be implemented in such a way that (after a start-up phase) the response times are O(1) for each MIN, O(log q) for each DELETE and "usually" O(1) for each INSERT as encountered during execution.

Proof

Choose R to be a conventional priority queue. Let the fundamental loop be implemented as in theorem 2.3. We can leave the description as it is, and merely change the actions of the queueing primitives MINQ, DEQ and ENQ (which correspond to the MIN, DELETE and INSERT commands in the fundamental loop).

The current value of MINQ can be immediately read off from the root-record of R. It gives the smallest queue-element in constant time. A DEQ(i) operation should remove the front element (a) of Q_i from R and "drop" it from the pool, and push the next element of Q_i (b, if it exists) as a new item into R. If b exists, then b > a and we might implement the transaction as an UPDATE instead. In either way, the time required is O(log q) in worst case. ENQ(x,i) will merely add x to the rear of Q_i and finish, if Q_i wasn't empty. If Q_i was empty, then x gets into the front position right away and must be pushed into R like all other front-elements. The time required will be O(1) or O(log q), depending on whether the latter case occurs.

□

Different response times may result if we choose a different, perhaps more adaptive structure for R than just a conventional priority queue.

3. A general structuring scheme for priority queues.

Let S be a set of n elements from an arbitrary, ordered domain. We shall outline a general scheme for building priority queues for S from "simpler" priority queues for subsets of S. It derives from the structure of binomial queues (Vuillemin [10]), but seems to be fundamental in a much wider context. Proper "tuning" gives priority queues with better worst case response times for the basic commands than were achieved before. At the same time it will enable us to introduce binomial queues themselves, and to expose some difficulties in applying the same structural improvements developed below to this particular priority queue structure. Throughout this section we shall de-emphasize the space-requirements of our structures, and freely introduce pointers and linked lists when needed. The total space used will remain within a constant factor from the optimum.

Let S be partitioned into subsets $S_0, S_1, \ldots, S_k$. If we structure S_0 to S_k as conventional priority queues, then we can finish it off to a priority queue for S by entering the min's into a separate structure P as in Figure 5. There will be fewer than n min's, so we can afford P to be a "sophisticated" priority queue. For the time being, we leave P as a black box and merely list what is required of it.

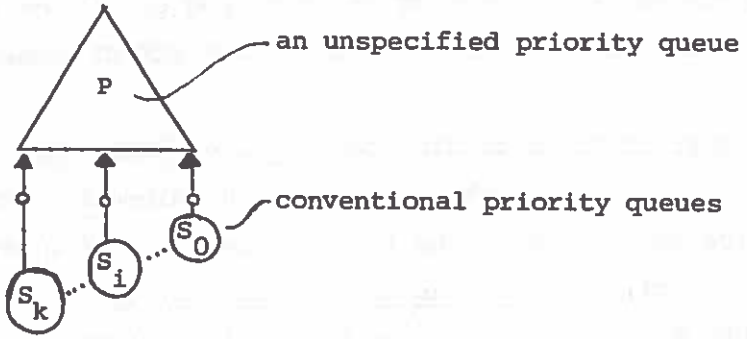


Figure 5.

A set S_i will typically contain 2^i elements. (It follows that some of the subsets are empty, depending on the binary representation of n.) We shall not structure S_i as a binomial queue, but rather as a perfect binary tree with 2^i leaves or i-tree. Thus, the priority queue structure for S_i is a full tree of tournaments among its elements with the winner (i.e. its smallest element) at the top. Actually, we allow S_i to be made up of 0, 1 or 2 i-trees! The observant reader will sense a connection with a redundant representation of integers n using the digits 0, 1 and 2. It is instructive to explore such a representation before we continue. (Such representations date back to e.g. Avizienis [2], but the applicability for data-structuring purposes seems to have been observed first in class-notes by Clancy and Knuth in 1977.)

Beginning from 0 for 0 and 1 for 1, we shall be applying one of two operations on a varying integer n:

+1 ("add one")

step 1. fix the rightmost 2 in the representation of n,

step 2. add 1 to the last digit of n (changing 0 to 1 or 1 to 2, whatever the case may be).

-1 ("subtract one")

step 1. if the last digit is non-zero, subtract 1 from the last digit of n and stop,

step 2. if the last digit is zero, borrow a "one" from the last non-zero digit (changing 2 to 1 or 1 to 0, whatever the case may be) and put 1's in all positions to the right.

There are ways of also restricting the work for -1 to a bounded number of digit-positions, but it does not seem to be feasible here. Fixing a 2 means to push it to the left:

...02... ➡ ...10...

...12... ➡ ...20...

It is the only risky step, because we have to make sure that at no moment in time we are requested to fix a 2 whose neighbour is a 2 also

...22... ➡ ✕

Proposition 3.1. The digits 0, 1 and 2 are sufficient to support the routines for +1 and -1.

Proof

We only need to argue that no representation can wind up having a 22 substring. The reader can easily verify this by observing that 2's can only occur in one of the following possible ways

a) 21*

b) ....21*0...

□

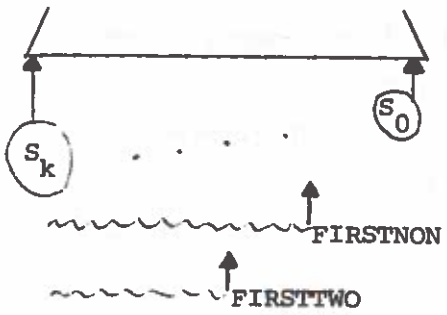
Back to priority queues, we shall introduce a digit-value $d(S_i)$ for each set indicating how many i-trees it contains (0, 1 or 2). At all times the string

$$d(S_k) \ldots d(S_1)\ d(S_0)$$

will be a redundant representation of n, obtained through a history of adding and subtracting "one" (i.e. one element). We shall now demonstrate

how the routines for +1 and -1 are made into efficient priority queue routines for INSERT and DELETE.

By a doubly linked list we shall keep track of the position of non-zero S_i's and in another, singly linked push-down list we record the S_i's with $d(S_i)=2$. By inspecting the top-pointers FIRSTNON and FIRSTTWO we can immediately access the "rightmost non-zero S_i" and the "rightmost S_i with $d(S_i)=2$", respectively.



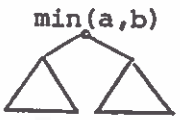
To add an element x to S we proceed as follows, completely in analogy to "+1":

INSERT(x):

IN 1. i⇐FIRSTTWO; if i=Ω then goto IN5

IN 2. combine the i-trees

from S_i into one (i+1)-tree



and move it into S_{i+1}

IN 3. update the non-zero and two-lists accordingly

IN 4. delete max(a,b) from P

IN 5. add x, as a 0-tree, to S_0

IN 6. update the non-zero and two-lists accordingly

IN 7. insert x into P

The steps are all self-explanatory, and are easily performed for the subset structuring.

Lemma 3.2. The implementation of INSERT(x) leaves the data-structure consistent and requires only O(1) time modulo one delete and one insert on P.

Thus, new elements always enter the set from the right (through S_0). When we call for deletion of an element x, it may have migrated to the left into

some bigger S_i. We can easily tell by chasing pointers from the record for x upwards, until we hit a root. (See Figure 6)

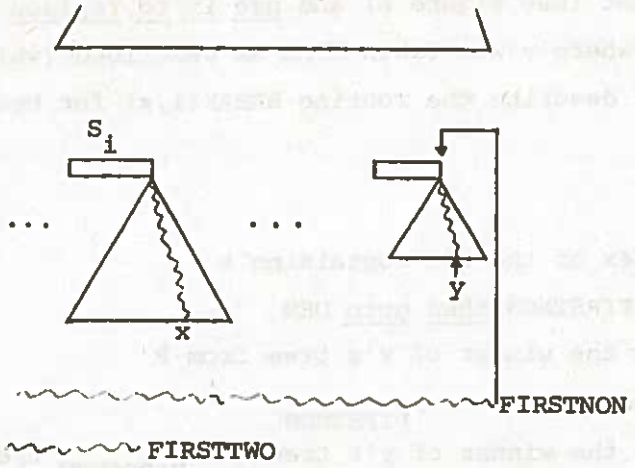


Figure 6

We must be careful in deleting x. One could consider the following general routine: while traversing the path from x to the root, break the i-tree into constituent j-trees (one for each j from 0 to i-1). We shall elaborate on it later, but for the moment we just note that effectively the "0-tree" (i.e., the leaf) containing x is dropped as desired (see Figure 7).

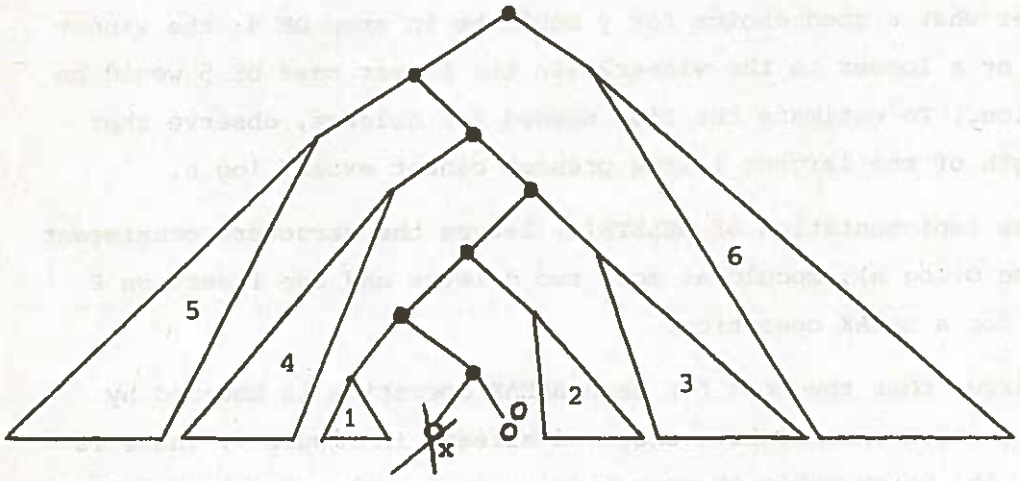


Figure 7

What to do with each of the dangling j-trees? They shouldn't stay in S_i. The obvious step would be to put the j-tree into S_j, for j from 0 to i-1. This would be very dangerous, because there may be S_j's here which already have 2 j-trees. Also, adjacent 1's would become adjacent 2's and there is no way we can handle that even numerically in the representation. It does work correctly only if we are breaking the rightmost non-zero tree! If x doesn't

belong to the rightmost non-zero set, then we shouldn't break its tree but proceed by a trick instead. Just "borrow" some arbitrary element y from the rightmost non-zero set (see Figure 6) and use it to replace x in its i-tree, then break the tree where y was taken from as described (while eliminating y from it). We shall describe the routine BREAK(i,x) for breaking an i-tree later.

DELETE(x):

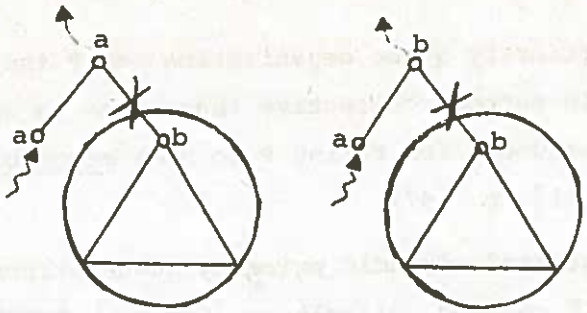
DE 1. i⇐index of the set containing x
DE 2. if i = FIRSTNON then goto DE9
DE 3. delete the winner of x's tree from P
DE 4. y⇐some element of $S_{FIRSTNON}$
DE 5. delete the winner of y's tree in $S_{FIRSTNON}$ from P
DE 6. put y in the leaf-position occupied by x, and chase it up the tournament tree
DE 7. insert the new winner of the i-tree into P
DE 8. i⇐FIRSTNON; x⇐y
DE 9. BREAK(i,x)

Several optimizations of this routine can be considered, but none will influence the time-complexity by more than a constant factor (... which is important in practice, but not for the present treatment). For instance, one may wonder what a good choice for y could be in step DE 4: the winner of $S_{FIRSTNON}$ or a looser to the winner? (In the former case DE 5 would be a direct action.) To estimate the time needed for deletes, observe that the path-length of the largest i-tree present cannot exceed log n.

Lemma 3.3. The implementation of DELETE(x) leaves the structure consistent and takes time O(log n), modulo at most two deletes and one insert on P and the cost for a BREAK operation.

We shall argue that the cost for each BREAK operation is bounded by O(log n) also, which one may have observed already in Figure 7. There is a bit more to it, which makes it conceptually to a rather involved operation.

When BREAK(i,x) is called we must split a given i-tree into parts and put each part into the appropriate S_j. We start the splitting at the leaf containing x and work our way up, at each node saving the subtree which wasn't cut to pieces. (In the beginning we consider the 0-tree containing x as "cut to pieces".) The saved subtree will be a perfect tournament, no matter what:



In the end we need not delete the overall winner of the tree from P, it was already taken care of in DE 5. (In fact, it wasn't necessary to do so unless the winner was x, but it doesn't harm in view of the next step.) However, a nasty task remains, the insertion of the winner of each j-tree obtained into P.

We conclude

Lemma 3.3'. The implementation of DELETE(x) leaves the structure consistent and requires only O(log n) time, modulo at most two deletes and the collective insertion of up to O(log n) elements into P.

It follows that we need only design a convenient priority queue structure for P to allow for MIN selection and DELETE and INSERT commands for the entire set. Note that UPDATE commands are supported by the structure easily, at a cost of only O(log n) and perhaps an update in P.

Theorem 3.4. There is a general, dynamic priority queue structure which requires O(log n) for each MIN command, O(1) for each INSERT command and again O(log n) for each DELETE or UPDATE command.

Proof

Let P be the unordered set of winners from the lower i-trees. There are at most O(log n) of them. MIN selection follows by a linear scan over P. Deletion and insertion require O(1) per element for P. Lemmas 3.2 and 3.3' easily give the conclusion for INSERT, DELETE and UPDATE commands as stated.

□

Perhaps the only interesting feature of theorem 3.4 is the O(1) worst case response time for INSERTs, which hasn't been achieved for priority queues within the O(log n) range for commands before. We must "pay" for it by an O(log n) response time for MINs, which may be acceptable if MINs aren't tested very often but less desirable when they are. Let's see what trade-off we can make.

Choosing a different priority queue organization for P isn't alltogether trivial, because it should support collective insertions in an efficient manner. It is a succint argument for taking P to be a mergeable heap! (see Aho, Hopcroft and Ullman [1], p. 147).

Theorem 3.5. There is a general, dynamic priority queue structure which requires O(1) for each MIN command, O(loglog n) for each INSERT and O(log n) for each DELETE or UPDATE command.

Proof

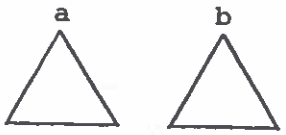
Organize P as a mergeable heap on O(log n) elements, for instance using a 2-3 tree. MINs are now available in constant time, and deletion and insertion only cost O(loglog n) per element for P. The collective insertion of up to O(log n) elements is easily dealt with also. Just build a 2-3 tree based priority queue out of the elements in O(log n) steps, and merge it into P.

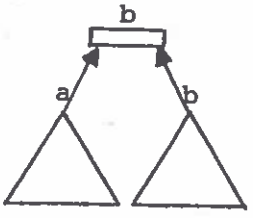
□

The latter result is interesting when compared to, for instance, the study of van Emde Boas et al. [6] concerning priority queues for subsets of a fixed universe.

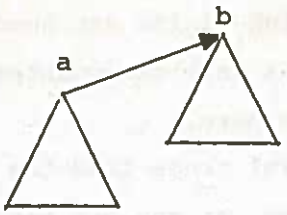
4. Application to binomial queues.

There has clearly been a reason for using the perfect binary tree instead of any other structure to represent each component of an S_i, apart from the fact that it works. It was perhaps the simplest structure familiar to all readers which could be used to explain the tricky data manipulation routines. The space required remains within acceptable limits also: at each internal node we merely need one pointer to the record of the "winner" at this node and a father link. A feature of binomial queues is that it embeds the linkage structure in the element-records and puts loosers in an organized manner "under" winners. The idea is easily grasped when considering the combination of two small trees into a bigger one, like we did in the INSERT routine:

instead of combining 

into 

and allocating a fresh record for an internal node, we attach the looser a directly to the winner b



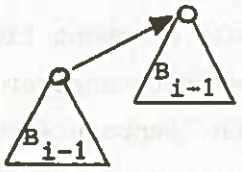
No duplication of elements is needed and an element is always there "where it wins". However, we now need additional pointers at each node to facilitate manipulation of the structures just the same, and it is largely a matter of context to determine if the structuring using perfect binary trees isn't more expedient after all. We shall demonstrate that the routines explained in section 3 can be worked out for the binomial tree representation as well, but there will be unsuspected difficulties which make it all less elegant than before.

The pattern for combining trees of 2^i elements into a bigger one inspires to the following, known family of trees $\{B_i\}_{i \geq 0}$.

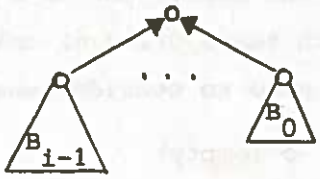
Definition

(i) $B_0 = 0$ (a single node)

(ii) $B_i =$



The B_i's are what we call "binomial trees". They have some nice structural properties of which we only mention one. Each B_i-tree can be decomposed as



It holds the key to a simple, linked representation which will be essential for the data-manipulation routines (notably for BREAK) later. See Figure 8.

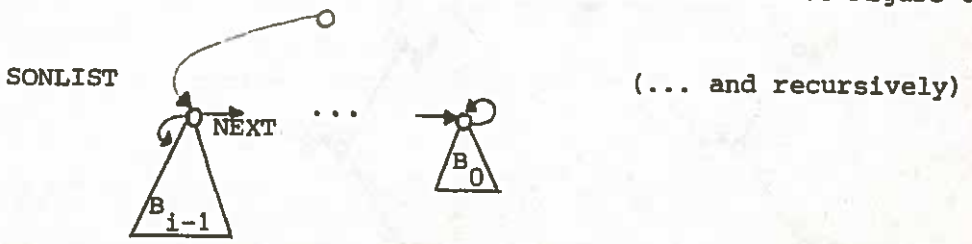


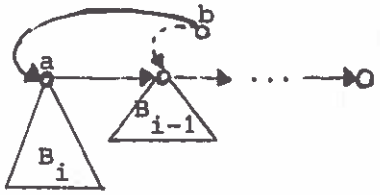
Figure 8

Apart from the father link only two more link-fields are needed in each record. (A method of Mark Brown [3] requires only two pointer fields total, but it would only complicate matters here.)

Suppose we organize elements into binomial trees from the start. It means that each S_i will contain 0, 1 or 2 B_i-trees, at any one time during execution. Let us see how the routines of section 3 can be adapted.

The INSERT(x) routine can be left completely as it was, except that in stage IN2 (obviously) we must combine the B_i-trees of the rightmost "2"

into one B_{i+1}-tree



(assuming that b is winner), a task which takes only constant time.

Great difficulties arise in DELETE(x), which seems to uncover a not-so-perfect feature of the binomial tree representation. Suppose we must delete x, occurring in the "interior" of some binomial tree. Note that the elements along any path towards the root must form a decreasing sequence, and that replacement of these elements by a decreasing sequence of smaller items leaves a consistent tree. We could fill the empty spot of x by shifting down its ancestors one node, a task which takes O(log n) only in one pass up (see Figure 9). It follows that we only need to consider what must happen

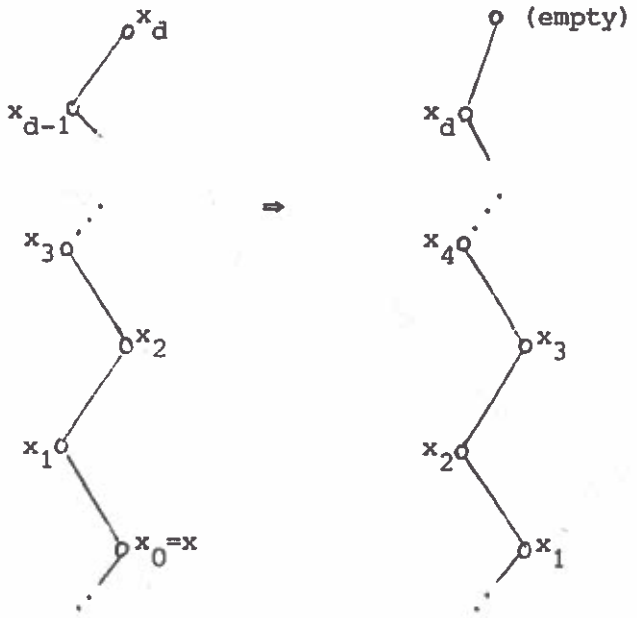


Figure 9

if we delete an x which happens to be the root of a B_i-tree. Fortunately the representation of a tree (see Figure 8) makes it very easy to BREAK a B_i-tree into its component B_j-trees if we cut its root off. This solves only part of the problem. As in section 3 we must try to avoid breaking a tree which isn't in the rightmost non-zero position, because it can lead to congestion in the "lower" positions. Borrowing an element y from the first non-zero tree like we did is hardly feasible this time, because it seems to be impossible to insert it in x's tree without wasting an inordinate amount extra comparisons. (We may be lucky to find an element y less than x, because it could be inserted for x and perculate upwards.) Sufficient structure is lacking for easily exchanging elements between subsets.

There seems to be no other way out than to use brute force: spend O(log n) time resolving all 2's to straighten the forest back into ordinary "binary" format, BREAK the B_i-tree and add the pieces according to a normal binary addition scheme to the lower order positions. The required "straightening" of the forest is not in harmony with and destroys much of the economy of the redundant representation. It may require many deletions from P in the course of one DELETE, which makes the worst case acceptable only if we let P be an unstructured set (theorem 3.4). It should be noted that the worst case will occur only rarely.

We conclude that binomial queues can be modified to guarantee an O(1) response time for INSERT commands, but DELETE commands will have to perform some additional O(log n) steps of clean-up work each time. Hence there may be some reason for using ordinary perfect binary trees after all?

5. References

[1] Aho, A.V., J. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley, Reading, Mass., 1974.

[2] Avizienis, A., Signed digit number representations for fast parallel arithmetic, IEEE Transact. on Electronic Computers 3 (1961) 389-399.

[3] Brown, M.R., The analysis of a practical and nearly optimal priority queue, Techn. Rep. STAN-CS-77-600, Dept. of Computer Science, Stanford University, 1977.

[4] Knuth, D.E., The art of computer programming, vol. 1: fundamental algorithms, Addison-Wesley, Reading, Mass., 1968.

[5] Knuth, D.E., The art of computer programming, vol. 3: sorting and searching, Addison-Wesley, Reading, Mass., 1973.

[6] van Emde Boas, P., R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory 10 (1977) 99-127.

[7] van Leeuwen, J., On the construction of Huffman-trees, in: S. Michaelson & R. Milner (eds.), Automata, languages and programming (Proceedings 3rd Colloq.), Edinburgh University Press, 1976, pp. 382-410.

[8] van Leeuwen, J., Linear time generation of a new fixed length data-compression code, Techn. Rep. RUU-CS-78-3, Dept. of Computer Science, University of Utrecht, 1978.

[9] Verhoeff, J., A new data-compression technique, Annals of Systems Research (1978) (to appear).

[10] Vuillemin, J., A data structure for manipulating priority queues, Comm. ACM 21 (1978) 309-315.