Math. Systems Theory 15, 295–309 (1982)

Mathematical
Systems Theory

A Priority Queue in Which Initialization and Queue Operations Take $O(\log\log D)$ Time[1]

Donald B. Johnson

Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania

Abstract. Many computer algorithms have embedded in them a subalgorithm called a priority queue which produces on demand an element of extreme priority among elements in the queue. Queues on unrestricted priority domains have a running time of $\Theta(n\log n)$ for sequences of n queue operations. We describe a simple priority queue over the priority domain $\{1,\ldots,N\}$ in which initialization, insertion, and deletion take $O(\log\log D)$ time, where D is the difference between the next lowest and next highest priority elements in the queue. In the case of initialization, $D=\Theta(N)$. Finding a least element, greatest element, and the neighbor in priority order of some specified element take constant time. We also consider dynamic space allocation for the data structures used. Space can be allocated in blocks of size $\Theta(N^{1/p})$, for small integer p.

1. Introduction

A priority queue is an on-line algorithm for processing a sequence of operations on a universe of elements. The operations we consider are

i. inserting an element into the queue,
ii. deleting an element from the queue, and
iii. finding among elements in the queue an element which has the maximum or minimum priority,

a priority being associated with each element in the universe.

One way to measure the efficiency of a priority queue is to employ worst case bounds for time and space consumed when processing an entire operation sequence, taking the worst case among all sequences of length n. In this case we assume that the queue is empty before the sequence is executed and we include the time to initialize the queue in the total execution time for the sequence. Another way is to consider the cost of individual operations, taking into account

[1]This research was supported by the National Science Foundation under grants MCS 77-21092 and MCS 80-002684.

the environment in which they occur, specifically, the distance to other priorities in the queue. In each case we assume the universe of elements and their priority assignments are given at no cost in time or space.

When priorities are drawn from an arbitrary domain with a total order, pairwise comparison is the only means of discriminating between priorities. Since a sequence of n insertions followed by n deletions of the element of minimum priority will sort a universe of n elements on their priorities, it is not possible to conceive of a priority queue algorithm that will run uniformly in $o(n \log n)$ time given only a total order on an arbitrary priority domain. Queues which achieve insertion and deletion time of $\Theta(\log n)$ on a queue of n elements, and therefore exhibit a running time of $O(n \log n)$ and use $O(n)$ space on sequences of length n, are described in an extensive literature. See, for example, [1, 2, 3, 7, 8].

Priority queues are often embedded in other algorithms, the efficiency of which depends on the efficiency of the priority queue. Such applications are too numerous to reference here, but it is relevant to mention that among these applications are some with special properties which allow more efficient queues than those of general applicability. In certain optimization problems on dense graphs, operation sequences typically have more insertions and deletions than determinations of extreme elements. We have treated such unbalanced sequences in [4, 5, 6] where additional references may be found.

Other applications pose the priority queue problem on a finite priority domain. This is the case we study here. Our work extends the results of [10, 11, 12] which were subsequently simplified in [9].[2] One of our extensions improves the processing time for short sequences over relatively large domains. For the priority domain $\{1,\ldots,N\}$ the algorithm cited runs in $O(N + n \log\log N)$ time. Our algorithm runs in $O(n \log\log N)$ time and therefore is superior in the worst case whenever $n = o(N/\log\log N)$.[3]

The previous work cited maintains a tree and transits paths in this tree through binary search, recursively implemented. One aim of our work has been to see how simple a representation can be developed in which all operations on the tree are performed non-recursively. In relation to our work, the presentation in [9] bears out the commonly expressed view that recursive algorithms can be more elegant than non-recursive ones. Interestingly, however, it turns out that our non-recursive approach allows a reduction in the cost of the expensive individual operations of insertion and deletion from $O(\log\log N)$ to $O(\log\log D)$. The parameter D is the length of the smallest interval, between distinct priorities present in the queue, which properly contains the priority inserted or deleted. For the purpose of this definition it is assumed that priorities 0 and $N+1$ are always present in the queue. Though $D = \Theta(N)$ in the worst case, this improvement from $O(\log\log N)$ to $O(\log\log D)$ is substantial when the queue tends to be active over

---

[2] There is also a less inaccessible simplified exposition by the original author. See Developments in Data Structures (Section 10: Priority Deques), in *Interface Between Computer Science and Operations Research*, *Mathematical Centre Tracts 99*, J. K. Lenstra, A. H. G. Rinnooy Kan, and P. van Emde Boas (eds.), Mathematisch Centrum, Amsterdam, 1979.

[3] Unless otherwise indicated, all logarithms are taken to the base 2. Also, to avoid cumbersome notation, we assume that whenever the expression $\log x$ appears in an asymptotic expression, it is to be taken as $\max(1, \log x)$.

small subranges of the entire priority range $\{1,\ldots,N\}$ or when many operations occur when the queue is densely populated. Under the assumption that no priority repeats, the former time bound [10, 11, 12] is always $\Theta(\log\log N)$. Our bound can be $O(1)$ in the best case even when D is large.

A consequence of these facts is that our algorithm will sort n numbers drawn from $\{1,\ldots,N\}$ in $O(n\log\log(N/n))$ time. This reduces to linear time for the conventional bucket sorting problem on $n=\Omega(N)$ elements. In fact, a sorting algorithm based on this method is better than bucket sort whenever $n\log\log(N/n)=o(N)$. The bound for such an algorithm is also better than $\Theta(n\log n)$, the bound for generally applicable comparison sorts, whenever $(\log(N/n))^c < n$ for some fixed constant c which depends on the algorithms. Presumably $c>1$, but is not too large. Thus, new bounds for sorting are demonstrated for certain ranges of n as a function of N. Of course, because of the space consumed by our algorithm, this observation is of theoretical interest only.

We also show how to allocate space for our data structure dynamically in blocks of size $\Theta(N^{1/p})$ at an increase in cost for insertion and deletion of a factor of at most p. As observed in [10], a space-saving scheme, with p inherently limited to 2, arises naturally from the recursive structure. Our method rests on a different recursive decomposition which allows space to be balanced for other than $p=2$.

The elements in our priority queue are kept in N buckets, each of which contains elements of a single priority from the domain $\{1,\ldots,N\}$. The non-empty buckets are kept in the *bucket list*, a doubly-linked list sorted on priority. Given such a list, the operations of finding a maximum element, a minimum element, or an element nearest in priority to that of the elements in a specified bucket can be done in constant time. The remainder of this paper describes a data structure for which initialization of the structure and insertion or deletion of a bucket in the sorted list can each be performed in $O(\log\log D)$ time.

The idea is to conceive of a complete binary tree with N leaves on the lowest level, but dynamically to leave as much of the tree as possible unconstructed. The leaves of the binary tree on its lowest level are numbered consecutively from the left so that they correspond to the priority domain $\{1,\ldots,N\}$ taken as an ordered set. Each priority therefore defines a unique path to the root of the tree.

We would like to construct the path for a priority whenever the corresponding bucket becomes non-empty. We would like to delete that portion of the path which is no longer needed when the bucket becomes empty again. With this arrangement it would be possible to maintain the bucket list as follows. Whenever it became necessary to insert a new bucket, its path would be constructed rootward until the new path intersected the path of some non-empty bucket. Then this path would be followed leafward to find the non-empty bucket adjacent to which the new bucket must be inserted into the list. Deletion would be a reversal of this process.

Such an algorithm has one feature which we require: parts of the tree which are never used are never constructed. Thus our development departs from [11, 12] in which an entire tree is initialized at a cost of $\Theta(N)$.[4] However, the above description is still inadequate. We cannot afford to construct or even traverse

[4] Even under the deferred allocation mentioned in [10] an initialization cost of $\Theta(N)$ can be incurred by the first $N^{1/2}$ insertions.

entire new path segments at each queue operation since traversal or construction would cost time proportional to the length of the path segment involved. This length is $\Theta(\log D)$, in the worst case. We show below how to get by with constructing only a logarithmic number of nodes in each new path segment, once space has been allocated for the entire tree. Our discussion begins in Section 2 where we explain the data structure. In Section 3, we show how insertion of buckets is done in $O(\log\log D)$ time. In Section 4, we describe bucket deletion. In the final section we discuss tree initialization and how to allocate space dynamically to reduce space requirements.

We will assume that an array of some size $M > n$, in which only n elements are ever accessed, may be initialized to some uniform value at a cost of $\Theta(n)$, not $\Theta(M)$. The well-known technique for doing so [1] distributes initialization over the sequence of array accesses, initializing an element when a first access occurs. Let $\Theta(n)$ space be allocated for a stack. To mark an array element as having been accessed we will insert in the element a pointer to the next stack location and in this stack location will be placed a pointer to the array element. Thus an array element has had a previous access if and only if the element points to a stack location which points back to the array element. Actually, the only data that must be stored in the array are pointers into the stack. All data associated with array elements can be stored in the stack.

Our analysis employs a unit cost arithmetic complexity measure [1]. All numbers generated by our algorithm are $O(N)$ in absolute value. The algorithm uses division by powers of two, which is assumed to be of unit cost, but not general multiplication or division. In our method for saving space we also employ multiplication by powers of two. It may be said parenthetically that it is unlikely that these operations add computational power to the unit cost plus-RAM. We notice, for example, that the shifts required can be implemented, on any Turing machine that represents numbers in binary, using no more steps than required for addition. It is clear that allowing these operations would not change the time bounds in [10, 11, 12]. However, much of the detail there could have been avoided if these operations had not been forbidden out of concern for their possible power.

2. The Structure of the Tree

As we have already indicated, the fundamental data structure our algorithm manipulates is a list of non-empty buckets in which buckets are ordered by priority. A bucket is inserted when a first instance of the priority to which it corresponds is added to the queue. When a bucket becomes empty, it is deleted from the queue.

Buckets on the list comprise the leaves of a binary tree T. The nodes of T are indexed by defining a complete binary *host tree* $H = \{1,\ldots,2^l + N\}$ for $l = \lceil \log(N+1) \rceil$. Each node q in $H - \{1\}$ is a *child* of $\lfloor q/2 \rfloor$. If q is even, q is the *left* child; if q is odd, q is the *right* child. Node 1 is the *root* of H.

As with the usual definitions, a *path* is a sequence of nodes $(p = p_1, p_2, \ldots, p_k = q)$ in which p_{i+1} is a child of p_i in H for all $i = 1,\ldots,k-1$. We denote such a path as $(p \rightsquigarrow q)$. We conceive of our trees as rooted at the "top" so that each

path is directed "downward." The fact that node s is somewhere on path $(p \rightsquigarrow q)$ we denote as $s \in (p \rightsquigarrow q)$.

To obtain the embedding of T in H, the leaves $\{2^l, \ldots, 2^l + N\}$ in H are identified in order with the priorities $\{0, 1, \ldots, N\}$. Bucket 0 is used as the header for the bucket list, and is always present in the queue. Tree T is then defined as the union of all paths $(1 \rightsquigarrow f)$ in H for which leaf f corresponds to a bucket on the bucket list. Figures 1 and 2 show a host tree H and a tree T which can be embedded in H. (Equivalent constructions can be had in which the list header is fixed to precede any given priority i. In this case each priority $j < i$ is mapped to leaf $j - 1$ to allow the header to use leaf $i - 1$.)

Under these definitions only leaves at the lowest level of H are ever in T. From now on, we shall use the term *leaf* to refer only to such leaves. Every path from the root to a leaf is of the form $(1 = f_l, f_{l-1}, \ldots, f_0 = f)$, where $f_i = \lfloor f/2^i \rfloor$ for $i = 0, \ldots, l$, as may be verified from the identity $\lfloor f/2^k \rfloor = \lfloor \lfloor f/2^{k-1} \rfloor /2 \rfloor$ for positive integers k. For example, in Figures 1 and 2 the path $(1 \rightsquigarrow 10)$ is $(1,2,5,10) = (\lfloor 10/2^3 \rfloor, \lfloor 10/2^2 \rfloor, \lfloor 10/2^1 \rfloor, \lfloor 10/2^0 \rfloor)$.

Paths $(q \rightsquigarrow f)$ in H and T are $k+1$ nodes in length when $q = \lfloor f/2^k \rfloor$, too long to be traversed by a procedure which must run in $O(\log k)$ time. Consequently we introduce the notion of a chain $(f \overset{*}{\Rightarrow} q)$ which is a listing of the unique one-sided binary search on $(q \rightsquigarrow f)$ from f for node q. First, in a sequence of steps of expanding size $1, 2, 4, \cdots$ up the path (toward nodes of smaller value), $(f \overset{*}{\Rightarrow} q)$ consists of nodes $(f = f_0, f_1, \ldots, f_i)$ where $f_i \leqslant q$ and $f_j > q$ for $j = 0, \ldots, i - 1$. Then, if necessary, the sequence completes with an ordinary binary search on the path $(f_i \rightsquigarrow f_{i-1})$. Thus

$$(f \overset{*}{\Rightarrow} q) = (f = f_0, f_1, \ldots, f_r = q),$$

as is algorithmically defined by the following procedure.

```
procedure (f ⇒* q)
    {one-sided binary search}
    f_0 ← f
    β_0 ← 0
    j ← 1
    i ← −1
    while f_{j−1} > q do
        β_j ← β_{j−1} + 2^{j−1}
        f_j ← max{1, ⌊f/2^{β_j}⌋}
        j ← j + 1
```

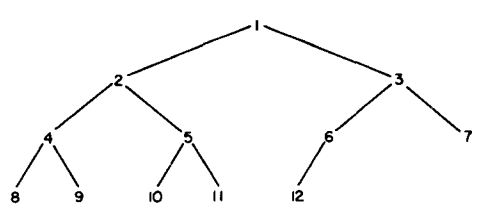

Fig. 1. Pictorial representation of host tree $H = \{1, \ldots, 12\}$ for the priority set $\{1,2,3,4\}$. In this example, $l = 3$.

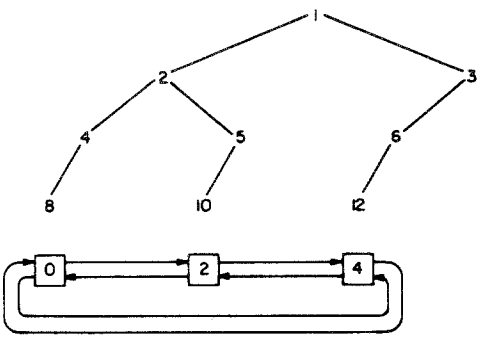

Fig. 2. Pictorial representation of a tree T for buckets $\langle 2,4\rangle$ hosted by H in Figure 1. The list of buckets is shown below the leaves of T.

endwhile
$i \leftarrow j-1$
while $f_{j-1} \neq q$ **do**
  if $f_{j-1} < q$ **then** $\beta_j \leftarrow \beta_{j-1} - 2^{2i-j-1}$
    else $\beta_j \leftarrow \beta_{j-1} + 2^{2i-j-1}$
  $f_j \leftarrow \max\{1, \lfloor f/2^{\beta_j} \rfloor\}$
  $j \leftarrow j+1$
endwhile
return $((f_0, f_1, \ldots, f_{j-1}))$

For all $q \in (1 \rightsquigarrow f)$, the result $(f \stackrel{*}{\Rightarrow} q) = (f_0, \ldots, f_r)$ is well defined, with $r+1 \leqslant 2\lceil \log(k+1) \rceil$. Figure 3 shows chain $(742 \stackrel{*}{\Rightarrow} 11)$ on path $(1 \rightsquigarrow 742)$ in a tree T with height $l = 9$. It is convenient also to use the notation $(f: \beta_0, \ldots, \beta_r)$ for chain $(f \stackrel{*}{\Rightarrow} q) = (f_0, \ldots, f_r)$.

In addition to the evident use of the procedure to construct chains from f when q is known, we will also use it when only f is known. In these cases the procedure is implemented by specifying the outcomes of comparisons $f_{j-1}: q$, as we will explain shortly.

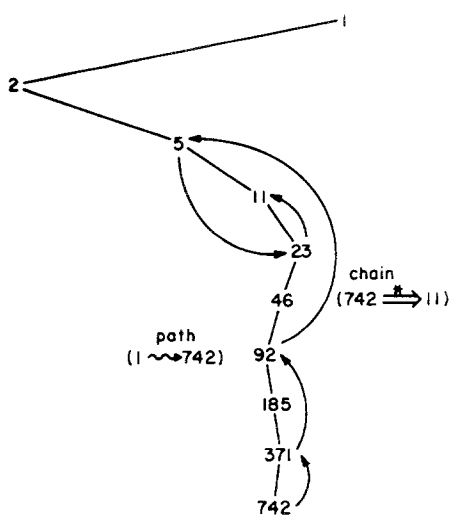

Fig. 3. Path $(1 \rightsquigarrow 742)$ and trace of chain $(742 \stackrel{*}{\Rightarrow} 11)$ in a tree T with $l = 9$.

Using the fact that either $|\beta_{j+1}-\beta_j| = 2|\beta_j-\beta_{j-1}|$ or $2|\beta_{j+1}-\beta_j| = |\beta_j-\beta_{j-1}|$, it is easy to show that every node in a chain is *extremal* in the sense that each node is either an upper bound or a lower bound on the nodes following it in the chain. This bound is strict in every case except node 1, the only node that can repeat on a chain. It follows from the above that two chains

$$(a:\alpha_0,\ldots,\alpha_m) = (a=a_0,\ldots,a_m), \text{ and}$$
$$(b:\beta_0,\ldots,\beta_n) = (b=b_0,\ldots,b_n),$$

$a_m \leqslant b_n$, have the following *coincidence properties*. Since $\alpha_0 = \beta_0$ for any two chains, there exists an index k, $0 \leqslant k \leqslant \min\{m, n\}$, for which

(i) either $\alpha_i = \beta_i > \alpha_m$ or $\alpha_i = \beta_i < \beta_n$, $i = 0,\ldots,k$,
(ii) $\alpha_m \geqslant \alpha_k = \beta_k \geqslant \beta_n$,
(iii) $\alpha_i > \alpha_k = \beta_k > \beta_j$, for all $i = k+1,\ldots,m, j = k+1,\ldots,n$,

and furthermore, if $(1 \rightsquigarrow a)$ and $(1 \rightsquigarrow b)$ coincide on a maximal subpath $(1 \rightsquigarrow p)$, then

(iv) $a_i = b_i$, $0 \leqslant i \leqslant k$ and $a_i \leqslant p$,
(v) $a_i \neq b_i$, otherwise.

If nodes $\lfloor q/2 \rfloor$, q, and $q+1$ are present in T, then q is the *left sibling* of $q+1$. For a node $q+1$ (either the root or with left sibling q) we call the maximal subpath $(q+1 \rightsquigarrow f)$, ending in leaf f and on which there are no other nodes with left siblings, the *left base* of leaf f. The left bases of any tree T partition the nodes of T. For example, the left bases of the tree in Figure 2 are $\{(1,2,/.,8), (5,10), (3,6,12)\}$. Under the partition of any tree into left bases, there is defined for each node in a base the *top* of its base and the *leaf* of its base. Thus, let w belong to left base $(p \rightsquigarrow f)$. Then $lefttop(w) = p$ and $leftleaf(w) = f$. Referring for example to Figure 2, $lefttop(2) = 1$ and $leftleaf(2) = 8$.

Similar definitions hold on the right. Right bases also partition T, but in general the partition is different. For example, in Figure 2, $righttop(2) = 2$ and $rightleaf(2) = 10$.

We now show how T is recorded in terms of variables $LLEAF(q)$, $RLEAF(q)$, $LTOP(q)$, $RTOP(q)$, $LREF(q)$, and $RREF(q)$ for each node $q \in H$. The idea is that while *top* and *leaf* values are defined for every node in T, it is sufficient to record these values only on the chains of T that connect the *leaf* node to the *top* node of each base, right or left. When H is labeled in this way as shown precisely below, H is *properly labeled* for T. The symbol Λ means "no label." Thus, for example, when $LTOP(q) = p \neq \Lambda$ we will say that "q is labeled on the left."

Proper Labeling

$$LLEAF(q) = \begin{cases} leftleaf(q) & \text{if } (q \rightsquigarrow leftleaf(q)) \text{ is a left base,} \\ \Lambda & \text{otherwise.} \end{cases}$$

$$RLEAF(q) = \begin{cases} rightleaf(q) & \text{if } (q \rightsquigarrow rightleaf(q)) \text{ is a right base,} \\ \Lambda & \text{otherwise.} \end{cases}$$

$$LTOP(q) = \begin{cases} \Lambda & \text{if } q \text{ is in no left chain,} \\ lefttop(q) & \text{if } q \text{ is in } (leftleaf(q) \overset{*}{\Rightarrow} lefttop(leftleaf(q))), \\ 0 & \text{otherwise.} \end{cases}$$

$$RTOP(q) = \begin{cases} \Lambda & \text{if } q \text{ is in no right chain,} \\ righttop(q) & \text{if } q \text{ is in } (rightleaf(q) \overset{*}{\Rightarrow} \\ & \qquad righttop(rightleaf(q))), \\ 0 & \text{otherwise.} \end{cases}$$

$LREF(q)$ = number of distinct left chains containing q.

$RREF(q)$ = number of distinct right chains containing q.

It should be noticed that a node has a TOP value of zero when it lies on a chain but lies in no base for any chain to which it belongs.

We have defined proper labeling in a somewhat redundant manner. This is for purposes of exposition. It is possible to use the reference counts to distinguish between top values 0 and Λ. Also *LEAF* and *TOP* can use the same storage location on a given side, left or right. If the value stored for w is a node greater than or equal to w, then w is known to be a top node and the value stored is $LEAF(w)$. Otherwise, the value stored is $TOP(w)$.

3. Inserting a Bucket

In order to insert $(1 \rightsquigarrow f)$ into T for some new leaf f, it would be sufficient to find some leaf $t \in T$ which maximizes p, the largest node common to $(1 \rightsquigarrow f)$ and $(1 \rightsquigarrow t)$. The maximality of p would guarantee that $(1 \rightsquigarrow f) = (1 \rightsquigarrow p, q \rightsquigarrow f)$ where $(1 \rightsquigarrow p) \in T$ and $(q \rightsquigarrow f) \notin T$. Thus, for example, if $f < t$ it would be correct to insert f into the bucket list ahead of $g = leftleaf(p)$. See Figure 4.

The problem is that p may not be labeled, so our procedure must find a labeled surrogate, p', for which $leftleaf(p') = leftleaf(p)$, for example, in the case $f < t$. In the following procedure, the surrogates are nodes d and e for cases $f < t$ and $f > t$, respectively. In the procedure it is assumed that H is properly labeled for T and that T is not empty.

INSERT_BUCKET(f)
  $(f_0,\ldots,f_m) \leftarrow$ **labelrule**(f, *left*)
  $d \leftarrow \max_i\{f_i \in (f_0,\ldots,f_m)$ and f_i has a non-zero label on the left$\}$
  $(f_0,\ldots,f_m) \leftarrow$ **labelrule**(f, *right*)
  $e \leftarrow \max_i\{f_i \in (f_0,\ldots,f_m)$ and f_i has a non-zero label on the right$\}$
  $b \leftarrow LTOP(d)$
  $c \leftarrow RTOP(e)$
  $g \leftarrow LLEAF(b)$
  $h \leftarrow RLEAF(c)$
  if $f < g$ **then** link the bucket for f ahead of g
  else link it after h
  {restore proper labeling for T}
  if $f < g$ **then Irestoreleft**(f, g)
  else Irestoreright(f, h)

where **labelrule** specifies outcomes of comparisons $f_{j-1} : q$ in procedure $(f \overset{*}{\Rightarrow} q)$ of the preceding section as follows,

procedure labelrule(f, *side*)
{Move down when f_{j-1} is labeled and up otherwise.}
return(($f \stackrel{*}{\Rightarrow} x$) where
if $i \geq 0$ **and** $2i - j - 1 < 0$ **then** $f_{j-1} = x$
else if f_{j-1} is labeled on *side* **then** $f_{j-1} < x$
else $f_{j-1} > x$)

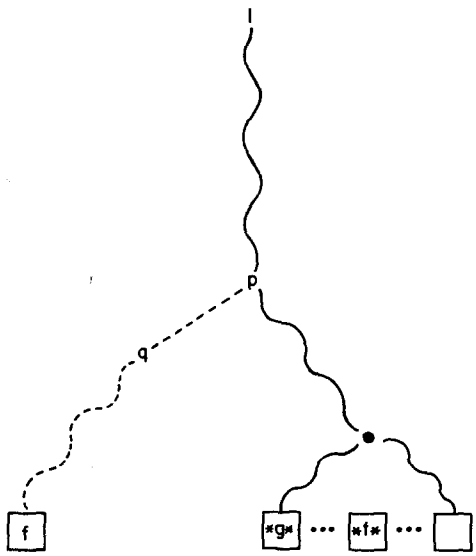

Fig. 4. Choice of t so as to maximize p guarantees ($q \rightsquigarrow f$) not to be in T.

Procedures **lrestoreleft** and **lrestoreright** will be discussed shortly.

Lemma 1.

$$d = \max\{q \in (1 \rightsquigarrow f) \textit{ and } q \textit{ has a non-zero label on the left}\}$$

$$e = \max\{q \in (1 \rightsquigarrow f) \textit{ and } q \textit{ has a non-zero label on the right}\}$$

Proof. Let $d' = \max\{q \in (1 \rightsquigarrow f)$ and q has a non-zero label on the left$\}$. Since d' has a label on the left, there is a leaf t in T for which the chain ($t \stackrel{*}{\Rightarrow} \textit{lefttop}(t)$) contains d'. Let d'' be the first node on this chain to be in ($1 \rightsquigarrow f$) and to satisfy $d'' \geq \textit{lefttop}(t)$. Such a node must exist since d' itself has these properties. In fact, by the extremal property we may conclude that d'' is the largest node on chain ($t \stackrel{*}{\Rightarrow} \textit{lefttop}(t)$) to have these properties. Therefore, since d'' must have a non-zero left label in order to satisfy proper labeling, it follows that $d' = d''$.

Since chains ($t \stackrel{*}{\Rightarrow} \textit{lefttop}(t)$) and ($f \stackrel{*}{\Rightarrow} f_m$) = **labelrule**($f$, *left*) move upward below the intersection of ($1 \rightsquigarrow f$) and ($1 \rightsquigarrow t$), and move downward above $\textit{lefttop}(t)$, it follows from the coincidence properties of chains that these chains coincide at $d' = d''$. Since every node on ($f \stackrel{*}{\Rightarrow} f_m$) which follows d' is greater than d', the lemma holds for d.

The result for e is shown similarly. □

Given this result for d, consider the case where $f < g$ (as shown in Figure 5a). Let some leaf t in T maximize p, the largest node common to ($1 \rightsquigarrow f$) and ($1 \rightsquigarrow t$).

Since the top node of each left base has a label on the left, it follows from the maximality of d that every node on the subpath $(d \rightsquigarrow p)$ belongs to the same left base $(b \rightsquigarrow g)$. Thus node g is correctly computed, and the insertion is correct. The argument involving e is made similarly, referring to the right side.

It may be deduced from the extremal property of chains that the search for d (as well as for e) is confined to a path of length $O(length(p \rightsquigarrow f))$.

It may be seen from Figures 5a and 5b that what is needed to restore proper labeling to H for T in the case where $f < g$ is to delete the labeling for left base $(b \rightsquigarrow g)$ and label the new bases, right base $(r \rightsquigarrow f)$ and left bases $(b \rightsquigarrow f)$ and $(s \rightsquigarrow g)$. We accomplish this with procedure **lrestoreleft**. (Procedure **lrestoreright** deals with the case $h < f$ and will not be shown.)

procedure lrestoreleft(f, g)
  {restore proper labeling when new leaf $f < g$}
  $(f_0,\ldots,f_m), (g_0,\ldots,g_m) \leftarrow$ **equalrule**(f, g)
  $p \leftarrow \max_i\{f_i = g_i, f_i \in (f_0,\ldots,f_m), g_i \in (g_0,\ldots,g_m)\}$
  $r \leftarrow 2p$
  $s \leftarrow r + 1$
  $LLEAF(b) \leftarrow f$
  $LLEAF(s) \leftarrow g$
  $RLEAF(r) \leftarrow f$
  for $w \in (g \overset{*}{\Rightarrow} b)|s$ **do**
    $LREF(w) \leftarrow LREF(w) - 1$
    if $LREF(w) = 0$ **then** $LTOP(w) \leftarrow \Lambda$
    else $LTOP(w) \leftarrow 0$
  endfor
  for $w \in (f \overset{*}{\Rightarrow} b)|r$ **do**
    $LREF(w) \leftarrow LREF(w) + 1$
    if $w \geqslant b$ **then** $LTOP(w) \leftarrow b$
    else if $LTOP(w) = \Lambda$ **then** $LTOP(w) \leftarrow 0$
  endfor
  for $w \in (g \overset{*}{\Rightarrow} s)$ **do**
    $LREF(w) \leftarrow LREF(w) + 1$
    if $w \geqslant s$ **then** $LTOP(w) \leftarrow s$
    else if $LTOP(w) = \Lambda$ **then** $LTOP(w) = 0$
  endfor
  for $w \in (f \overset{*}{\Rightarrow} r)$ **do**
    $RREF(w) \leftarrow RREF(w) + 1$
    if $w \geqslant r$ **then** $RTOP(w) \leftarrow r$
    else if $RTOP(w) = \Lambda$ **then** $RTOP(w) = 0$
  endfor

where the procedure **equalrule** is

procedure equalrule(f, g)
  {move down when $f_{j-1} = g_{j-1}$ and up otherwise.}
  return$((f \overset{*}{\Rightarrow} x)$ and $(g \overset{*}{\Rightarrow} y)$ where
    if $i \geqslant 0$ **and** $2i - j - 1 < 0$ **then** $f_{j-1} = x$ **and** $g_{j-1} = y$
    else if $f_{j-1} \neq g_{j-1}$ **then** $f_{j-1} > x$ **and** $g_{j-1} > y$
      else $f_{j-1} < x$ **and** $g_{j-1} < y$)

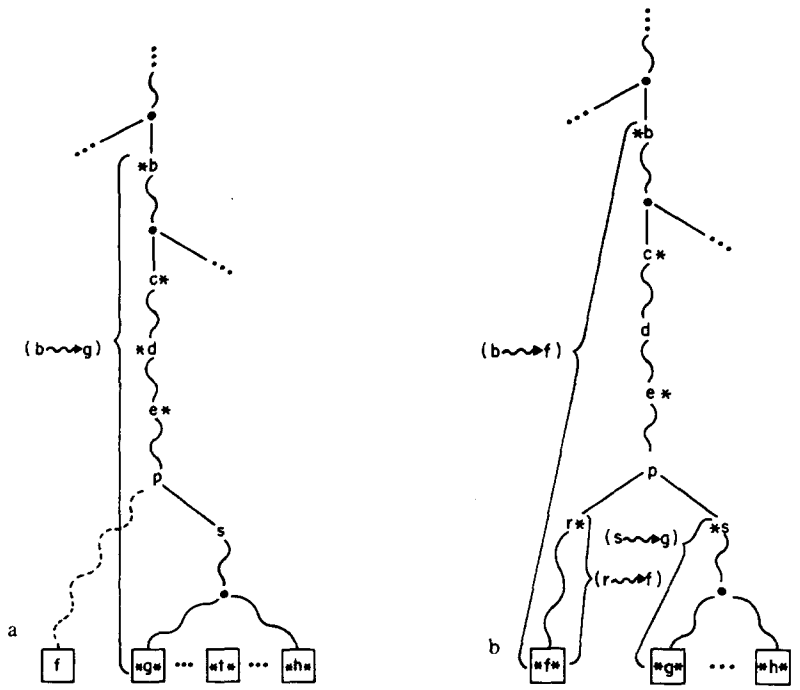

Fig. 5. a. Part of T before insertion of f, showing left base $(b \rightsquigarrow g)$. A star (*) at a node indicates that there must be a label on the side where the star appears. b. Part of T before deletion of f, showing left bases $(b \rightsquigarrow f)$ and $(s \rightsquigarrow g)$, and right base $(r \rightsquigarrow f)$. A star (*) at a node indicates that there must be a label on the side where the star appears.

The notation $(f \stackrel{*}{\Rightarrow} q)|w$ means a shortest prefix of $(f \stackrel{*}{\Rightarrow} q)$ ending in nodes f_i and f_{i+1}, where $w \geqslant f_i > f_{i+1}$. This refinement is necessary to control running time.

Lemma 2. *Node p in procedure* **Irestoreleft** *is the largest node common to* $(1 \rightsquigarrow f)$ *and* $(1 \rightsquigarrow g)$.

Proof. Let p' be the largest node common to $(1 \rightsquigarrow f)$ and $(1 \rightsquigarrow g)$. Thus $(f \stackrel{*}{\Rightarrow} p')$ and $(g \stackrel{*}{\Rightarrow} p')$ are prefixes, respectively, of the chains returned by **equalrule**(f, g). But no nodes following p' on the chains from **equalrule**(f, g) can coincide since they are all greater than p'. Therefore, $p = p'$. □

With p determined, the next five assignment statements in **Irestoreleft** find r and s and restore the *LEAF* labels for T. The remaining statements correct the *TOP* and *REF* information by traversing the old and new chains involved: $(g \stackrel{*}{\Rightarrow} b)$ (where non-zero labels are removed in left base $(b \rightsquigarrow g)$), and $(f \stackrel{*}{\Rightarrow} b)$, $(g \stackrel{*}{\Rightarrow} s)$, and $(f \stackrel{*}{\Rightarrow} r)$ (where non-zero labels are introduced in the respective bases). Reference counts are adjusted in the obvious manner during these traversals. The values 0 and Λ are also placed properly during these same traversals. When reference counts reveal whether a node belongs to some chain outside of its base, in this case $TOP(w)$ for a chain node w gets 0 on the

appropriate side if it does not already have a non-zero value. Otherwise, when $REF(w) = 0$, it must be that $TOP(w) = \Lambda$ on the appropriate side.

It remains to comment on the restricted traversals $(g \stackrel{*}{\Rightarrow} b)|s$ and $(f \stackrel{*}{\Rightarrow} b)|r$. The extremal property of chains guarantees that the suffixes deleted by the restrictions $|s$ and $|r$ never enter the regions at or below s and r, respectively. Therefore, no node is missed in these regions. Again, applying the extremal property it can be shown that chains $(f \stackrel{*}{\Rightarrow} b)$ and $(g \stackrel{*}{\Rightarrow} b)$ coincide on $(1 \rightsquigarrow p)$ and therefore neither reference counts nor labels need to be changed in this region because whatever would be altered for the removal of $(g \stackrel{*}{\Rightarrow} b)$ would be restored for $(f \stackrel{*}{\Rightarrow} b)$. While we have not succeeded in keeping path $(1 \rightsquigarrow p)$ entirely free of traversal, whatever nodes are touched above s in the restricted traversal $(g \stackrel{*}{\Rightarrow} b)|s$ have their labels restored by the restricted traversal $(f \stackrel{*}{\Rightarrow} b)|r$, since the same nodes are touched in each case.

All of the chain traversals can be done in $O(\log(length(r \rightsquigarrow f)))$ time. We have already made this observation in the case of finding d and e. It is also clearly true for finding p and for the traversals of chains $(g \stackrel{*}{\Rightarrow} s)$, $(f \stackrel{*}{\Rightarrow} r)$, $(g \stackrel{*}{\Rightarrow} b)|s$, and $(f \stackrel{*}{\Rightarrow} b)|r$. It follows that **INSERT_BUCKET** can be made to run in time proportional to the logarithm of the length of the path appended to T.

4. Deleting a Bucket

The deletion procedure is very simple since we know the position in the bucket list of the bucket f to be deleted.

DELETE_BUCKET(f)
    delete bucket for f from bucket list
    ⟨restore proper labeling for T⟩
    if $LTOP(f) < RTOP(f)$ **then** **Drestoreleft**(f)
    else **Drestoreright**(f)

Proper labeling is unique since it is defined by the tree T and not by the sequence of operations by which T is constructed. Thus, restoring proper labeling after f is removed from T is the reverse of restoring the labeling after a hypothetical insertion of leaf f into tree $T - f$. We may therefore refer to Figures 5a and 5b for the case where $LTOP(f) < RTOP(f)$. The procedure for this case is as follows.

procedure **Drestoreleft**(f)
    ⟨restore proper labeling when deleted f satisfies $LTOP(f) < RTOP(f)$⟩
    $r \leftarrow RTOP(f)$
    $s \leftarrow r + 1$
    $g \leftarrow LLEAF(s)$
    $(f_0, \ldots, f_m), (g_0, \ldots, g_m) \leftarrow$ **equalrule**(f, g)
    $e \leftarrow \max_i \{f_i = g_i, f_i \in (f_0, \ldots, f_m), g_i \in (g_0, \ldots, g_m)$
        and f_i has a nonzero label on the right⟩
    $b \leftarrow LTOP(f)$
    $c \leftarrow RTOP(e)$
    $RLEAF(r) \leftarrow \Lambda$
    $LLEAF(s) \leftarrow \Lambda$
    $LLEAF(b) \leftarrow g$

```
for w ∈ (f ⇒* r) do
    RREF(w) ← RREF(w) − 1
    if RREF(w) = 0 then RTOP(w) ← Λ
    else RTOP(w) ← 0
endfor
for w ∈ (g ⇒* s) do
    LREF(w) ← LREF(w) − 1
    if LREF(w) = 0 then LTOP(w) ← Λ
    else LTOP(w) ← 0
endfor
for w ∈ (f ⇒* b)|r do
    LREF(w) ← LREF(w) − 1
    if LREF(w) = 0 then LTOP(w) ← Λ
    else LTOP(w) ← 0
endfor
for w ∈ (g ⇒* b)|s do
    LREF(w) ← LREF(w) + 1
    if w ≥ b then LTOP(w) ← b
    else if LTOP(w) = Λ then LTOP(w) = 0
endfor
```

Lemma 3. *Node e is the largest node with a nonzero label on the right and common to* $(1 \rightsquigarrow f)$ *and* $(1 \rightsquigarrow g)$.

Proof. Similar to the proof of Lemma 2. □

Given this result, the correctness and complexity analysis follow essentially as in the previous section. We conclude that **DELETE_BUCKET** runs in time proportional to the logarithm of the length of the path deleted from T.

5. Conclusion

The entire data structure is initialized in two steps. First an array $H(1:2^l + N)$ is allocated. Then for each q in H, the values $LTOP(q)$, $RTOP(q)$, $LLEAF(q)$, and $RLEAF(q)$ are set to Λ, and $LREF(q)$ and $RREF(q)$ are set to zero. If the initialization of these values is distributed over subsequent operations as the well-known technique alluded to in the introduction allows, it will cost constant time to set up the host tree H prior to executing the sequence of queue operations. The second step of initialization creates an empty bucket list which is associated with leaf 2^l. Leaf 2^l is put into T by setting $LLEAF(1) = RLEAF(1) = 2^l$ and, for $q \in (2^l \overset{*}{\Rightarrow} 1)$, $LTOP(q) = RTOP(q) = LREF(q) = RREF(q) = 1$. It follows from previous discussion that the entire initialization costs $\Theta(\log\log N)$ time. This completes our exposition of the basic algorithm and the analysis of its complexity.

Even though allocation of $\Theta(N)$ space can be done in constant time, the requirement for this much space at the outset may be burdensome when N is large. Even when the length of the access sequence is large, much of the $\Theta(N)$ space may go unused. This observation was exploited in one of the original papers [10] (and left unmentioned in the others) by a simple scheme which allocates

space according to the recursive decomposition of the tree used for searching it. We observe that this space economy arises naturally in the recursive formulation of [9] and needs only to be mentioned explicitly in the earlier formulation in which the entire data structure is set up at the start.

Our approach is to partition H, together with 2^k dummy nodes, with respect to a parameter k. When $N+1$ is chosen to be a power of two, the partition is based on a decomposition into a *root subtree* on nodes $\{1,\ldots,2^{k+1}-1\}$ and the 2^k *bottom subtrees* rooted in the leaves of the root subtree. When $N+1$ is not a power of two, some of the 2^k bottom subtrees may be missing. The root in each of the 2^k subtrees is replaced with a dummy node so the leaves of the root subtree are not duplicated in the bottom subtrees.

If, for instance, k is chosen equal to $l/2$ for even l, this partition yields 2^k+1 subtrees each with $2^{k+1}-1$ nodes.

Given a host tree H and node $q = \lfloor f/2^\beta \rfloor$ for some leaf f, the subtree (of the partition) in which q belongs is the root subtree when $l-\beta \leqslant k$. Otherwise, q is in the bottom subtree rooted in the dummy node corresponding to node $\lfloor f/2^{l-k} \rfloor$. Since every node q that our algorithm finds is generated from a leaf f and the parameter β in an expression of the form $\lfloor f/2^\beta \rfloor$, a directory of size 2^k+1 suffices for locating in constant time a pointer to the subtree containing q. This directory can be generated in constant time by initializing it in the distributed fashion described earlier. The location problem within the bottom subtree is then to find node $q' = \lfloor f'/2^\beta \rfloor$ in this subtree taken as rooted at 1, where it can be shown that $f' = f - 2^{l-k}\lfloor f/2^{l-k} \rfloor + 2^{l-k}$.

Suppose we choose to have no more than $p \leqslant l$ probes for each node access. When $p=1$, there is no partition of H, and $\Theta(N)$ space is allocated as described for our basic algorithm. When $p>1$, space can be allocated in blocks of size $\Theta(2^{l/p})$, that is, $\Theta(N^{1/p})$, instead of $\Theta(N)$. This is accomplished by choosing $k = \lceil l/p \rceil$, giving a directory of size $2^{\lceil l/p \rceil}+1$, and a root subtree with $2^{\lceil l/p \rceil+1} - 1$ nodes which is set up to be probed directly without being partitioned. The bottom subtrees have no more than $2^{l(p-1)/p}-1$ nodes. Therefore, in any one of these bottom subtrees, $p-1$ probes suffice for access when the same construction is applied recursively to the bottom subtree, using the initial value of k.

To determine space used as a function of n, the length of the operation sequence, the worst case assumption is that there are n insertions. Thus the space used is no larger than a quantity proportional to the number of nodes in the union of the root subtrees, generated in our recursive decomposition of H, that contain segments of the paths of the tree T generated by the n insertions. The number of such root subtrees is itself no larger than the largest number of nodes coverable by paths to n leaves in a tree of degree 2^k with 2^{l-k} leaves. When $n \leqslant 2^{l-k}$, the number of such subtrees, each of size 2^k, is no more than $2n + n(\lceil l/k \rceil - \log_{(2^k)} n)$. Thus, since the union of root subtrees can never be larger than $O(N)$, the space used can be expressed in terms of parameters p and l as

$$O(np(1-(\log n)/l)N^{1/p}), \qquad 1 \leqslant n \leqslant 2^{l(p-1)/p}, \text{ and}$$

$$O(N), \qquad n \geqslant 2^{l(p-1)/p}.$$

With this added discussion, we have shown a priority queue on the domain $\{1,\ldots,N\}$ where initialization, insertion, and deletion take $O(p \log\log D)$ time,

the other operations of finding extreme and neighbor elements take constant time, and space can be allocated in blocks of size $\Theta(N^{1/p})$.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley (1974)
2. M. R. Brown. Implementation and analysis of binomial queue algorithms, *SIAM J. Comput.* 7, 298–319 (1978)
3. C. A. Crane. Linear lists and priority queues as balanced binary trees, (Ph.D. Thesis) *STAN-CS-72-259*, Computer Science Department. Stanford University (February 1972)
4. D. B. Johnson. Efficient special purpose priority queues, *Proc. 15th Annual Allerton Conference on Communication, Control, and Computing.* Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois, 1–7 (1977)
5. D. B. Johnson. Efficient algorithms for shortest paths in sparse networks, *J. ACM* 24, 1–13 (1977)
6. D. B. Johnson. Priority queues with update and finding minimum spanning trees, *Information Processing Letters* 4, 53–57 (1975)
7. A. Jonassen and O.-J. Dahl. Analysis of an algorithm for priority queue administration, *BIT* 15, 409–422 (1975)
8. D. E. Knuth. *The Art of Computer Programming, Volume 3/Sorting and Searching.* Addison-Wesley (1973)
9. D. E. Knuth. *Notes on the van Emde Boas construction of priority deques: An instructive use of recursion* (unpublished notes March 1977)
10. P. Van Emde Boas. Preserving order in a forest in less than logarithmic time, *Proc. Sixteenth IEEE Conference on Foundations of Computer Science.* Berkeley, California, 75–84 (1975)
11. P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue, *Mathematical Systems Theory* 10, 99–127 (1977)
12. P. Van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space, *Information Processing Letters* 6, 80–82 (1977)

Received July 20, 1981, and in revised form March 26, 1982, and in final form July 12, 1982.