THE ANALYSIS OF A PRACTICAL AND NEARLY OPTIMAL PRIORITY QUEUE

by

Mark R. Brown

STAN-CS-77-600
MARCH 1977

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

The Analysis of a Practical and Nearly Optimal Priority Queue

Mark R. Brown

Abstract.

The binomial queue, a new data structure for implementing priority queues that can be efficiently merged, was recently discovered by Jean Vuillemin; we explore the properties of this structure in detail. New methods of representing binomial queues are given which reduce the storage overhead of the structure and increase the efficiency of operations on it. One of these representations allows any element of an unknown priority queue to be deleted in log time, using only two pointers per element of the queue. A complete analysis of the average time for insertion into and deletion from a binomial queue is performed. This analysis is based on the result that the distribution of keys in a random binomial queue is also the stationary distribution obtained after repeated insertions and deletions.

An abstract notion of priority queue efficiency is defined, based on comparison counting. A good lower bound on the average and worst case number of comparisons is derived; several priority queue algorithms are exhibited which nearly attain the bound. It is shown that one of these algorithms, using binomial queues, can be characterized in a simple way based on the number and type of comparisons that it requires. The proof of this result involves an interesting problem on trees for which Huffman's construction gives a solution.

The printing of this paper was supported in part by National Science Foundation grant MCS 72-03752 A03, by the Office of Naval Research contract N00014-76-C-0330, and by IBM Corporation. Reproduction in whole or in part is permitted for any purpose of the United States Government.

Acknowledgments

I would like to thank Robert Tarjan, both for acquainting me with binomial queues and for many useful discussions relating to this work; Donald Knuth, for prompting me to write up my results and for pointing out innumerable improvements to them; Phyllis Winkler, for making the task of writing much easier through her encouragement and excellent typing; and Andrew Yao, for reading the finished product. Sam Bent, Daniel Boley, Lyle Ramshaw and Terry Roberts have also contributed to my efforts in an essential way. Finally, I am pleased to acknowledge the National Science Foundation's financial support of my graduate study.

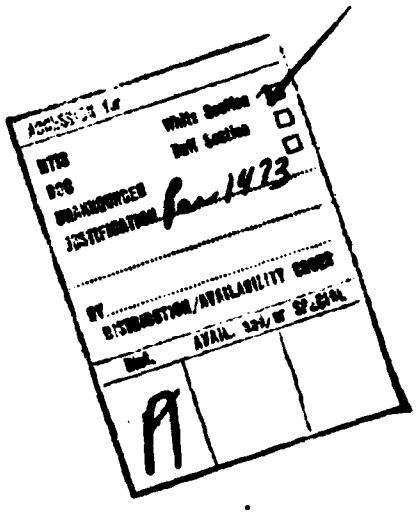


Table of Contents

Chapter One. Priority Queues

A priority queue is a structure for maintaining a collection of items, each having an associated key, such that the item with the smallest key is easily accessible. More precisely, if Q is a priority queue and x is an item containing a key from a linearly-ordered set, then the following operations are defined:

Insert(x, Q)	Add item x to the collection of items in Q.
DeleteSmallest(Q)	Remove the item containing the smallest key among all items in Q from Q; return the removed item.

These actions are referred to informally as insertion and deletion.

A mergeable priority queue is a priority queue with the additional property that two disjoint queues can be combined quickly into a single queue. That is, the operation

Union(T, Q)	Remove all items from T and add these items to Q

is defined when T and Q are mergeable priority queues; this operation is informally referred to as merging T into Q. Any pair of priority queues can be merged by using repeated applications of Insert and DeleteSmallest, but we reserve the qualification "mergeable" for those priority queues which can be merged quickly: merging should not require examining a positive fraction of the items in the queues.

The new results of this thesis, contained in Chapters 2 and 3, are concerned with a particular mergeable priority queue, and with priority queues in general. Chapter 1 is an introduction to the subject of

priority queues, and it should provide background for the later chapters.

The priority queue was not recognized as a fundamental data structure until quite recently. Several nontrivial priority queue organizations were developed for different applications before the usefulness of a priority queue as an abstraction was pointed out by Knuth in 1973 [28]. It follows that a good introduction to this subject requires not only a study of the data structures and their associated algorithms, but also an appreciation of the diverse applications in which priority queues are useful. We will devote Section 1.1 to a survey of these applications, and describe the known priority queue structures in Section 1.2. Finally, in Section 1.3 we present a summary of the results to be proved in Chapters 2 and 3.

1.1 Priority Queue Applications.

Possibly the earliest application of priority queues was in the implementation of simulation programming languages. Such languages are typically structured around an "event list" which is a record of actions to be performed at given instants of simulated time [2;5;15;30;34]. Thus adding a new action to the event list corresponds to an Insert, and executing the earliest event on the list requires a DeleteSmallest.

An event list generally has extra features which are not part of the definition of a priority queue. One of these is the FIFO property: events with equal times must be performed according to a FIFO discipline, in which the first event entered into the list is the first to be executed. In some situations it is important to be able to remove an arbitrary event (not just the earliest) from the list; in other cases, the ability to locate the event to be executed immediately before or after a given event may be necessary [41].

Another early application of priority queues was in sorting and selection problems. The idea of selection sorting [11;17;28, Section 5.2.3] is to repeatedly remove the smallest of a collection of items and move this item to an output area; hence we can accomplish a selection sort by first filling a priority queue using successive Insert operations, and then emptying the queue by using DeleteSmallest repeatedly.

Priority queues also play a role in external sorting [28, Section 5.4.1]. Many external sorting methods use a technique called replacement selection to form initial runs (sorted subsequences) and to merge runs together; replacement selection is based on alternating insertions and deletions from a priority queue. (Because the queue size does not change during

replacement selection, the full generality of a priority queue is not required.)

A typical selection problem is to find the k largest of n numbers, when n is much larger than k. One solution to this problem begins by inserting the first k numbers into a priority queue. Then for as long as there are numbers which have not been inserted, we insert a new number into the queue and then delete the smallest number from the queue. When this process terminates, the k largest numbers are contained in the queue [28, exercise 6.1-22]. This selection technique is used in an algorithm for random sampling [27, algorithm 3.4.2R].

An obvious application of priority queues, and one which helps motivate their name, is in job scheduling according to fixed priorities. In this situation jobs with priorities attached enter a system, and the job of highest priority is always the next to be executed. Examples of this procedure occur in operating systems and in industrial practice, though in both cases the restriction to fixed priorities may be violated in order to ensure fair scheduling (that is, to prevent a low-priority job from being delayed indefinitely).

Priority queues arise naturally in certain numerical iterations. One scheme for adaptive quadrature maintains a priority queue of subintervals whose union constitutes the interval of integration; each subinterval is labeled with the estimated error committed in the numerical integration over it. In each step of the iteration, the subinterval with the largest error is removed from the queue and bisected. Then the numerical integration is performed over these two smaller subintervals, which are inserted into the queue. The iteration stops when the total estimated error is reduced below a prescribed tolerance. This global

strategy is intended to result in subintervals over which the errors are roughly equal in magnitude [33].

It has been discovered that the use of fast priority queues can improve the efficiency of some well-known graph algorithms. In Kruskal's algorithm for computing minimum spanning trees, the procedure of sorting all edges and then scanning through the sorted list can be replaced by inserting all edges into a priority queue and then successively deleting the smallest edge [24]. If the priority queue is implemented properly this improves the algorithm on most graphs. Other ideas, one of which involves a good mergeable priority queue implementation, have led to more improvements in minimum spanning tree algorithms [3;19]. Similar applications have been found for priority queues in shortest path problems [18;20].

Finally, there is a collection of good algorithms which fall into none of the categories above but which depend on priority queues. Chartres' prime number generator uses a priority queue in a scheme to reduce its internal storage requirements [28, exercise 5.2.3-15]. B. L. Fox has mentioned that priority queues are useful in implementing some discrete programming algorithms [10]. Huffman's optimal code construction operates on a priority queue in just the opposite manner from the numerical iteration discussed above: it repeatedly selects the two smallest elements from a queue, combines them, and inserts the result back into the queue [26, pp. 400-405]. (For this problem, there is actually a better implementation which uses pre-sorting instead of priority queues.) The last of these algorithms that we will mention is the Hu-Tucker optimal binary search tree construction, whose asymptotic running time was greatly improved by using a good implementation of mergeable priority queues [28, pp. 439-444].

1.2 Priority Queue Structures.

The most obvious priority queue structure is certainly the linear list. If we keep a priority queue as a list of elements in arbitrary order, then an insertion consists of appending a new item to the front of the list, and a deletion requires searching the entire list to find the smallest key. (This is a mergeable priority queue since with linked lists, two queues can be merged in constant time.) A slightly more subtle method is to keep the list of elements sorted according to their keys; then a deletion is performed by removing the first item from the list, and an insertion requires searching down the sorted list to locate the proper place for the new element.

The sorted linear list was the structure first used to implement event lists, so it is not surprising that this structure can perform the extra operations required for event lists, and that it has the FIFO property. Both of the linear list schemes are easy to implement and are quite efficient when the queue size is small. But both schemes have the drawback that their running time for a single primitive operation grows linearly with the number of entries in the queue. A deletion from an unordered list always requires order m steps when there are m items in the list; an insertion into a sorted list requires $O(m)$ time on the average, whether the list is maintained in consecutive storage locations or in linked form. (Sorted list insertion can be made to run faster if the input has a known FIFO or LIFO tendency.) So both of these methods are slow when m is large.

A new priority queue scheme was discovered in 1964 by Arne Jonassen and Ole-Johan Dahl [21]. It represents a priority queue as a special type of binary tree, which they call a p-tree. Any node of a p-tree having

a null left link must also have a null right link, and the keys in a p-tree appear in increasing order when the tree is traversed in postorder.

By adding two extra links to each node, it is possible to perform a deletion from a p-tree in constant time. Insertion requires $O(m)$ steps in the worst case, but the analysis in [21] shows that an average insertion takes only $O(\log m)^2$ time. (The analysis applies only to a queue constructed by successive random insertions, but empirical tests indicate that deletions do not significantly affect the cost of subsequent insertions.) The p-tree structure has the FIFO property, and seems very well suited to event list applications.

In 1964, J. W. J. Williams introduced a data structure called a heap in connection with his heapsort algorithm [28, pp. 145-149]. The heap structure uses a linkless representation of a complete binary tree, storing the root in location 1 and the offspring of the node in location k in locations $2k$ and $2k+1$. A heap is further characterized by the requirement that the key contained in any node must be no larger than the keys of its offspring; a tree with this property is said to be heap-ordered. It is easy to see that in any heap-ordered tree, the smallest key appears in the root. Deletion from a heap requires $O(\log m)$ time on the average, and insertion takes $O(\log m)$ steps in the worst case but only $O(1)$ on the average [37]. Robert W. Floyd has demonstrated a bottom-up method which creates a heap containing m elements in $O(m)$ time [9].

Heaps are not difficult to implement, and they use storage efficiently since no space is needed for pointers within the structure. Items must move in order to perform insertions and deletions, so if the items are long it is more efficient to store pointers in the heap, instead of the items themselves.

A potential drawback of heaps is that they require a sufficiently large block of contiguous storage to be allocated in advance. It is possible to represent heaps as linked binary trees, with an upward pointer in each node, but this loses much of the simplicity of the method.

A sorted list becomes a practical priority queue structure for large N, given an efficient way of performing insertions into such a list. The balanced tree structure of Adel'son-Velskiĭ and Landis leads to such an efficient sorted list representation, as described in [4;28, pp. 463-468]. Both insertion and deletion can be performed in $O(\log m)$ steps. The algorithms are unfortunately quite complicated, but they should be useful in large problems when all of the flexibility that balanced trees offer is needed. An analogous sorted list representation is possible with 2-3 trees [1, pp. 155-157].

The leftist tree, a mergeable priority queue structure based on binary trees, was discovered in 1971 by Clark A. Crane [4; 28, pp. 150-152]. A leftist tree is heap-ordered, and satisfies the further condition that the shortest path from any node to a leaf may always be found by following right-links. This explains the designation "leftist", since these trees are generally slanted toward the left.

The basic operation on leftist trees is merging. It is possible to merge two leftist trees with a total of m nodes in $O(\log m)$ steps; to maintain the leftist structure during the merge it is necessary to keep an extra field in each node which records its minimum distance from a leaf. An insertion is accomplished by merging a single node into the tree; deletion is performed by removing the root and merging its two offspring. Thus individual insertions and deletions take $O(\log m)$ steps, and insertions and deletions take constant time in the case that insertions

obey a stack discipline. The leftist tree operations are not difficult to program, but since they require more time and space than corresponding heap operations it seems that leftist trees are only candidates for applications where fast merging is required.

Another mergeable priority queue was proposed in 1974 by Aho, Hopcroft, and Ullman [1, pp. 152-155]. The queue is based on 2-3 trees, a close relative of balanced trees. A 2-3 tree is a tree in which each non-leaf vertex has 2 or 3 sons, and all leaves appear on the same level. For the mergeable priority queue, assign items to the leaves of the 2-3 tree, and assign a label to each internal node v which gives the value of the smallest key contained in the leaves of the subtree rooted at v.

With this structure it is possible to perform insertions, deletions, and merges in $O(\log n)$ steps. The algorithms are only described informally in [1], and are rather involved although not difficult to follow. Although no careful study has been performed, it seems likely that this priority queue is harder to implement, requires more storage (because no items are stored in the internal nodes), and runs more slowly than leftist trees. The main reason for interest in this structure is that it supports a claim that anything is possible with 2-3 trees.

The binomial queue, a data structure for implementing mergeable priority queues, was discovered in 1975 by Jean Vuillemin [42]. The structure is a special type of forest, each of whose trees is heap-ordered; this forest can be represented as a binary tree. Chapter 2 considers this structure in detail, and concludes that binomial queues are preferable to leftist trees in most applications of mergeable priority queues. They

are also useful in other priority queue applications, particularly if the capability of deleting an arbitrary item from the queue is necessary.

If we assume that the keys in our queue are a subset of $\{1,2,\ldots,m\}$, then some interesting specialized priority queue structures are possible. A heap-like structure due to Luther C. Abel [28, p. 153] represents such a queue using only $2m-1$ bits of memory; it requires $O(\log m)$ steps for insertion and deletion, regardless of how many items are in the queue. A tree structure discovered by P. van Emde Boas [40] allows insertions and deletions in $O(\log \log m)$ steps, but the crossover point between this structure and the more straightforward $O(\log m)$ methods has not been determined.

1.3 Summary of the Results.

The previous section presented a maze of structures for implementing priority queues; how can we choose among them? It is not always possible to base such a choice on nicely quantifiable factors, since programming time and the number of times a program is to be used may weigh heavily in the consideration. The peculiarities of particular algorithms may turn out to be significant advantages or disadvantages in a given situation: the good performance of leftist trees when insertions follow a stack discipline may be essential to solve some problem efficiently, or the sequential allocation required by heaps may be impossible within a certain programming system. We have attempted to convey a feeling for these factors in the discussion of the preceding section.

In spite of these difficulties, it turns out that in many cases our choice of structures should be based on quantities such as how fast a given implementation will run, and how much storage it will use. The storage requirement is usually obvious, but the running time, especially "typical" running time, is generally more difficult to predict. It is possible to gain some feeling about the running time by executing the program several times on "random" inputs, but this procedure is unsatisfactory; it cannot give any significant increase in our understanding of the algorithm being tested. A method which can give us more insight is to determine the expected running time mathematically, under some plausible definition of what is meant by "random" inputs to our algorithm. This approach is called the analysis of an algorithm [26, Section 1.2.10].

Ideally, then, the following chapter should contain analyses of all of the algorithms mentioned in the previous section. But analysis turns

out to be very difficult for complicated priority queue structures. One reason for this is that the structures tend to degenerate from their "random" state (the state brought about by consecutive random insertions) when they are formed by a sequence of insertions and deletions. Such deletion sensitivity tends to complicate the analysis [23]. (Analyses of p-trees [21] and heaps [28;37] have been performed for the case of insertions only.)

Chapter 2 considers one priority queue structure, the binomial queue, in detail. When this structure was introduced by Vuillemin [42], it seemed to be of interest primarily due to its intrinsic beauty and simplicity. We show that the beauty of this structure is much deeper than was previously appreciated, by proving that a random binomial queue remains random even when deletions occur. This result allows us to perform a complete analysis of binomial queues.

We also demonstrate in Chapter 2 that the binomial queue is of greater practical importance than was previously acknowledged. We start by giving new methods for implementing binomial queues which improve the speed and reduce the storage requirements of the structure; one method allows any element of an unknown priority queue containing m elements to be deleted in $O(\log m)$ time, using only two pointers per element of the queue. We then compare the running time of a good implementation of binomial queues with the time used by other mergeable priority queues, and see that binomial queues are superior in most applications. This comparison is aided by our analysis of binomial queues, which allows the binomial queue implementation to be tuned for the best performance.

There are enough good priority queue structures in existence to make one wonder how fast any priority queue scheme can run. In general

it seems impossibly difficult to prove results about the minimum number of instructions that must be executed, or memory references performed, in order to accomplish a given task. But interesting optimality results have been proved about sorting, selection, and other problems within more restricted models of computation [28, Section 5.3]. This sort of investigation generally comes under the heading of "computational complexity".

Chapter 3 is concerned with optimality results about priority queues, a subject which has never previously been addressed. We define the efficiency of a priority queue scheme in terms of the number of inter-key comparisons it requires, and prove good upper and lower bounds on priority queue efficiency within this model. We also show that a certain form of the binomial queue algorithm, which is close to being optimal in our model, can be characterized in a simple way in terms of the number and type of comparisons it requires.

The Appendix contains implementations of the binomial queue algorithms in a high-level language. It also contains some of the assembly language implementations used to make the performance comparisons in Chapter 2.

Chapter Two. Implementation and Analysis of Binomial Queue Algorithms

The principal results of this chapter were summarized briefly in the previous section. In Section 2.1 we define the binomial queue structure in rather abstract terms, and in Section 2.2 we give informal descriptions of algorithms operating on this structure. No references to binomial queue implementations are made in these two sections; to a large degree, the conceptual simplicity of binomial queues depends on our ability to think of them in this abstract manner.

Section 2.3 presents several structures which can be used to implement binomial queues. While the original structure proposed for this purpose was a binary tree, none of our new structures are; several advantages are gained from abandoning the standard representation.

In Section 2.4 we define the notion of a random binomial queue, and prove that randomness is preserved in a wide variety of situations. Our analysis of binomial queue algorithms, based on these insensitivity results, is contained in Section 2.5; the end of that section contains a comparison of mergeable priority queue methods.

In what follows we use the terminology for trees given in [26]; in particular, the offspring of any node in a <u>tree</u> are ordered, while in an <u>oriented tree</u> they are unordered.

2.1 Binomial Trees, Forests and Queues.

For each $k \geq 0$ we define a class B_k of ordered trees as follows:

-- Any tree consisting of a single node is a B_0 tree. (1)

-- Suppose that Y and Z are disjoint B_{k-1} trees for $k \geq 1$. Then the tree obtained by adding an edge to make the root of Y become the leftmost offspring of the root of Z is a B_k tree. (2)

A binomial tree is a tree which is in class B_k for some k; the integer k is called the index of such a binomial tree. Binomial trees have appeared several times in the computer literature: they arise implicitly in backtrack algorithms for generating combinations [32]; B_0 through B_4 trees are shown explicitly in an algorithm for prime implicant determination [36]; a B_5 tree is given as the frontispiece for [26]; and oriented binomial trees, called S_n trees, were used by Fischer in an analysis of set union algorithms [8].

It should be clear from the definition above that all binomial trees having a given index are isomorphic in the sense that they have the same shape. Figure 1 on page 19 illustrates rule (2) for building binomial trees, and Figure 2 displays the first few cases.

An alternative construction rule, equivalent to (2), is often useful:

-- Suppose that $Z_{k-1}, \ldots, Z_0$ are disjoint trees such that Z_l is a B_l tree for $0 \leq l \leq k-1$. Let R be a node which is disjoint from each Z_l. Then the tree obtained by taking R as the root and making the roots of $Z_{k-1}, \ldots, Z_0$ the offspring of R, left to right in this order, is a B_k tree. (3)

Figure 3 illustrates rule (3) for building binomial trees. The equivalence of (2) and (3) follows by induction on k.

For future reference we record some properties of binomial trees, including the property which originally motivated their name:

Lemma 1. Let Z be a B_k tree. Then

(i) Z has 2^k nodes;

(ii) Z has $\binom{k}{l}$ nodes on level l.

Proof. Trivial induction on k. □

For each $m \geq 0$ we define a binomial forest of size m to be an ordered forest of binomial trees with the properties:

-- The forest contains m nodes. (4)

-- If a B_k tree Y is to the left of a B_l tree Z in the forest, then $k > l$. (That is, the indices of trees in the forest are strictly decreasing from left to right.) (5)

Since by (5) the indices of all trees in the forest are distinct, the structure of a binomial forest of size m can be encoded in a bit string $b_l b_{l-1} \cdots b_0$ such that b_j is the number (zero or one) of B_j trees in the forest. By Lemma 1, the number of nodes in the forest is $\sum_{j \geq 0} b_j \cdot 2^j$; hence $b_l b_{l-1} \cdots b_0$ is just the binary representation of m. This shows that a binomial forest of size m exists for each $m \geq 0$, and that all binomial forests of a given size are isomorphic. Figure 4 shows some small binomial forests.

Lemma 2. Let F be a binomial forest of size $m > 0$. Then

(i) The largest tree in F is a $B_{\lfloor \lg m \rfloor}$ tree;

(ii) There are $\nu(m) = (\#$ of 1's in binary representation of $m)$ trees in F; this is at most $\lfloor \lg(m+1) \rfloor$ trees;

(iii) There are $m-\nu(m)$ edges in F.

Proof. Obvious. □

Consider a binomial forest of size m such that each node has an associated key, where a linear order $\leq$ is defined on the set of possible key values. This forest is a binomial queue of size m if each binomial tree of the forest is heap-ordered: no offspring has a smaller key than its parent. This implies that no node in a tree has a smaller key than the root. Figure 5 gives an example of a binomial queue.

To avoid dwelling on details at this point, we shall defer discussion of representations for binomial queues until later sections. The timing bounds we give here and in the next section can only be fully justified by reference to a specific representation, but the bounds should be plausible as they stand.

The following propositions relating to binomial queues are essential:

Lemma 3. Two heap-ordered B_k trees can be merged into a single heap-ordered B_{k+1} tree in constant time.

Proof. We use construction rule (2). The merge is accomplished by first comparing the keys of the two roots, then adding an edge to make the larger root become the leftmost son of the smaller. (Ties can be broken in an arbitrary way.) This process requires making a single

comparison and adding a single edge to a tree; for an appropriate tree representation this requires constant time. □

Lemma 4. Let T be a heap-ordered B_k tree. Then the forest consisting of subtrees of T whose roots are the offspring of the root of T is a binomial queue of size 2^k-1.

Proof. This follows immediately from construction rule (3) and the fact that subtrees of a heap-ordered tree are heap-ordered. □

Figure 1. Construction of a binomial tree.

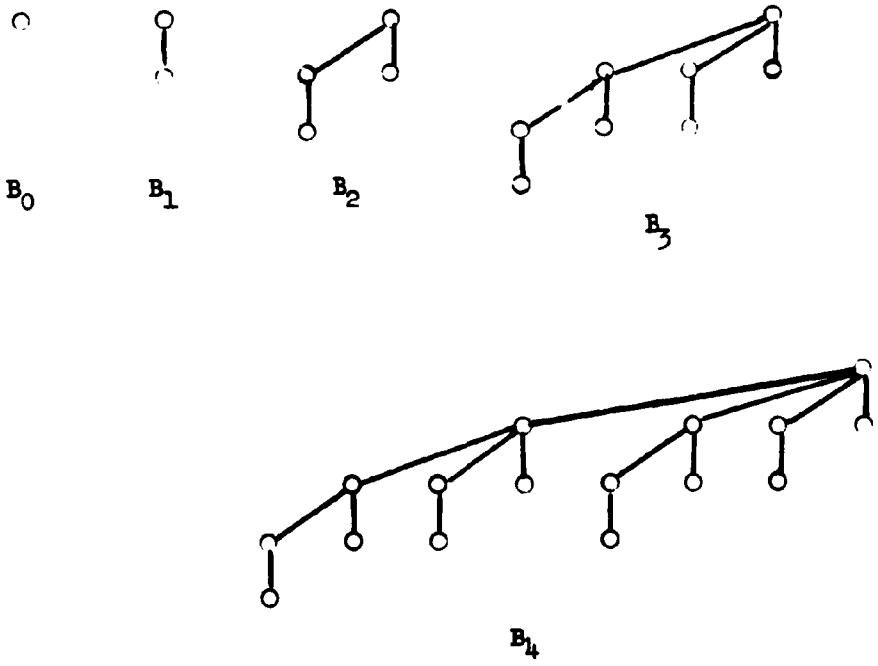


Figure 2. Small binomial trees.

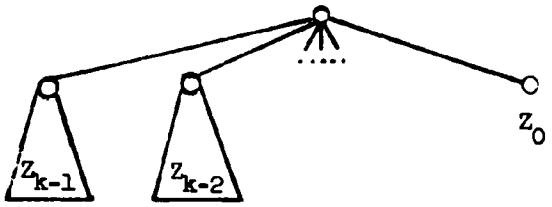


Figure 3. Alternative construction of a binomial tree.

Figure 4. Small binomial forests.

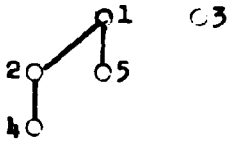


Figure 5. A binomial queue of size 5 (with integer keys).

2.2 Binomial Queue Algorithms.

In order to implement a mergeable priority queue using binomial queues, we must give binomial queue algorithms for the operations Insert , DeleteSmallest and Union which were introduced in Chapter 1. In the following informal description of the algorithms we let $\|Q\|$ denote the number of elements in a queue Q .

Consider first the operation Union(T, Q) , which merges the elements of T into Q . If $\|T\| = t$ and $\|Q\| = q$, then the process of merging the binomial queues for T and Q is analogous to the process of adding t and q in binary. We successively "add" pairs of heap-ordered B_k trees, as described in Lemma 3, for increasing values of k . In the initial step there are at most two B_0 trees present, one from each queue. If two are present, merge (add) them to produce a single B_1 tree, the carry. In the general step, there are at most three B_k trees present: one from each queue and a carry. If two or more are present we add two of them and carry the result, a B_{k+1} tree. Each step of this procedure requires constant time, and by Lemma 2 there are at most $\max(\lfloor \lg(t+1) \rfloor , \lfloor \lg(q+1) \rfloor)$ steps. Hence the entire operation requires $O(\max(\log\|T\| , \log\|Q\|))$ time. Figure 6 gives an example of Union with binomial queues.

Given the ability to perform Union , the operation Insert(x, Q) , which adds item x to queue Q , is trivial to specify: just let X be the binomial queue containing only the item x , and perform Union(X, Q) . By this method, the time required for an insertion into Q is $O(\log \|Q\|)$.

The operation DeleteSmallest(Q) is a bit more complicated. The first step is to locate the node x containing the smallest key. Since

x is the root of one of the queue's B_k trees, it can be found by examining each of these roots once. By Lemma 2 this requires $O(\log \|Q\|)$ time.

The second step of a deletion begins by removing the heap-ordered B_k tree T containing x from the binomial queue representing Q. Then T is partially dismantled by deleting all edges leaving the root x; this results in a binomial queue T' of size 2^k-1, as suggested by Lemma 4, plus the node x which will be returned as the value of DeleteSmallest.

The final step consists of merging the two queues formed in the second step: the queue T' formed from T, and the queue Q' formed by removing T from Q. Since each queue is smaller than $\|Q\|$, the operation Union(T',Q') requires $O(\log \|Q\|)$ time; therefore the entire deletion requires $O(\log \|Q\|)$ time. Figure 7 gives an example of DeleteSmallest with binomial queues.

In some situations it is useful to be able to delete an arbitrary element of a priority queue, not just the smallest. It is possible to accomplish this with binomial queues by generalizing the tree-dismantling step of DeleteSmallest. Suppose x is the node to be deleted, where x is contained in the B_k tree T. Referring back to Figure 1, we can decompose T into two B_{k-1} trees Y and Z. Now x lies in either Y or Z, and it lies in Y if and only if the root of Y is on the path from x to the root of T. So we remove the edge joining Y and Z, save the tree which does not contain x, and repeat the process on the tree containing x until x stands alone as a B_0 tree. When the process terminates, k subtrees have been saved, and they constitute a

binomial queue of size 2^k-1 . (Note that when x is the root of T , this procedure just deletes all edges leaving x .) The deletion is completed with a final Union , as before; the same time estimates also apply as long as we can delete each edge in constant time during the tree-dismantling step.

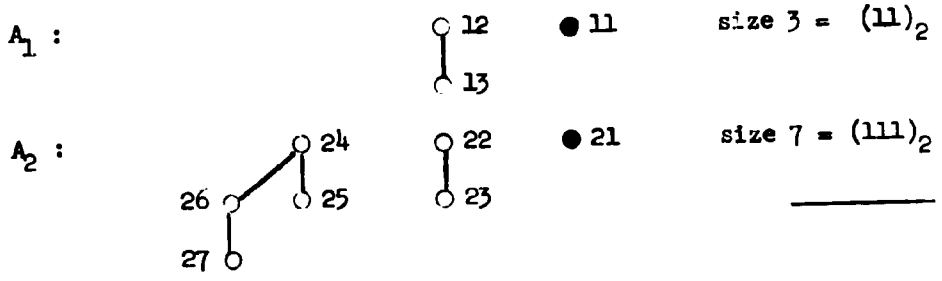
It is interesting to note that the time bound given for the Insert operation can be substantially improved if we study the effect of several consecutive instructions. Consider the sequence of instructions

$$\text{Insert}(x_1, Q);\ \text{Insert}(x_2, Q);\ \ldots;\ \text{Insert}(x_k, Q)\ .$$

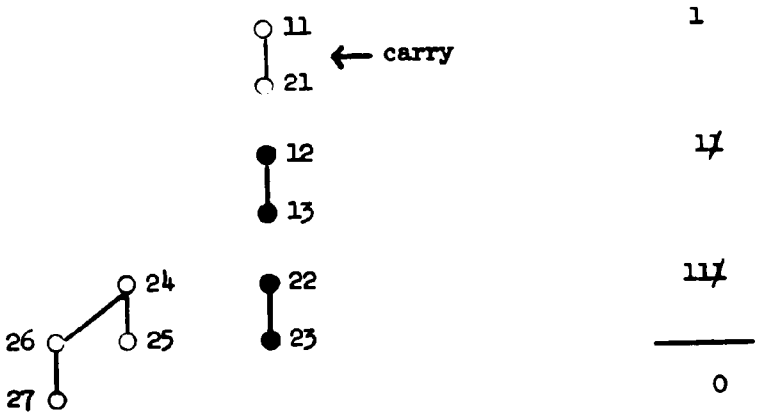
The time for each insertion is just $O(1)$ + O(number of edges created by the insertion) . If $\|Q\| = m$ initially, the number of edges created by this sequence of instructions is $(m+k - \nu(m+k)) - (m - \nu(m)) = k + \nu(m) - \nu(m+k)$ by Lemma 2. Hence the time for k insertions into a queue is $O(k) + O(k + \nu(m) - \nu(m+k)) = O(k + \log m)$ if the queue has size m initially.

As mentioned in Section 1.2, leftist trees and 2-3 trees can be used to implement mergeable priority queues. The time bounds for Insert , DeleteSmallest and Union using these structures have the same order of magnitude as those given above for binomial queues. But for both of these structures, insertions must be handled in a special way in order to achieve the $O(k + \log m)$ time bound for a sequence of Insert instructions. The naive approach, that of inserting elements individually into the leftist or 2-3 tree, can cost about $\log(k+m)$ per insertion for a total cost of $O(k \log(k+m))$. The faster approach is to buffer the insertions by maintaining the newly inserted elements as a forest of trees with graduated

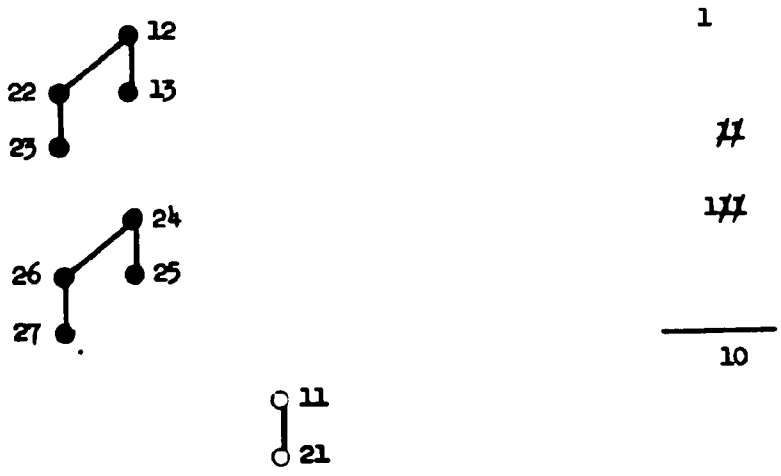
sizes, such as powers of two. Then insertions can be handled by balanced merges, just as with binomial queues. Individual merges require more than constant time, but the time for k insertions comes to $O(k + \log m)$.



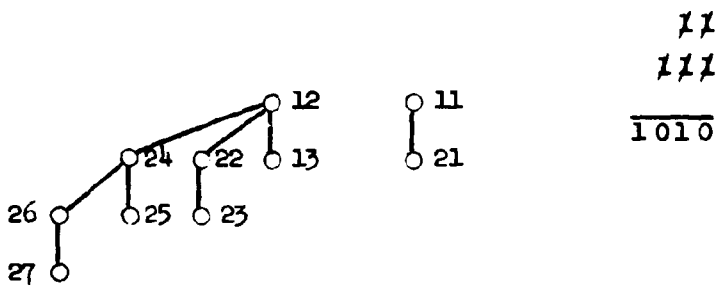
(a) Binomial queues of size 3 and 7 to be merged for UNION operation.



(b) After merge of B_0's; result is carry.

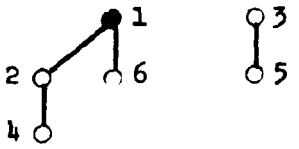


(c) After merge of B_1's.

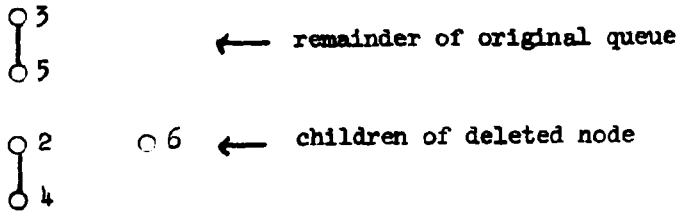


(d) Merge completed.

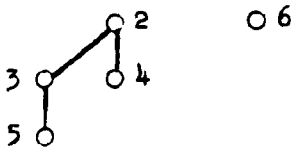
Figure 6. Binomial queue Union operation.



(a) Binomial queue of size 6. Node 1 is to be deleted.



(b) Two queues which result from removing node 1.



(c) Resulting queue of size 5 after merging.

Figure 7. DeleteSmallest on a binomial queue.

2.3 Structures for Binomial Queues.

In implementing binomial queues our objectives are to make the operations described in the previous section as efficient as possible while requiring a minimum of storage for each node. As usual, the most appropriate structure may depend on which operations are to be performed most frequently.

Since a binomial queue is a forest, it is natural to represent it as a binary tree [26]. But not all orientations of the binary tree links allow binomial queue operations to be performed efficiently. Evidently the individual trees of the binomial forest must be linked together from smaller to larger, in order to allow "carries" to propagate during the Union operation. But in order to allow two heap-ordered binomial trees to be merged in constant time, it seems necessary that the root of a binomial tree contain a pointer to its leftmost child; hence the subtrees must be linked from larger to smaller. This structure for binomial queues was suggested by Vuillemin [42]; we shall call it structure V . An example of a binomial queue and its representation using structure V is given in Figure 8(a).

The time bounds given in the preceding section for Insert , DeleteSmallest , and Union can be met using structure V , provided that the queue size is available during these operations. The queue size is necessary in order to determine efficiently the sizes of the trees in the queue as they are being processed. (The alternative is to store in each node the size of the tree of which it is the root; this will generally be less useful than keeping the queue size available, and it will use more storage.) In what follows we shall assume that the queue size is available

as part of the queue header; the other component of the queue header will be a pointer to the structure representing the queue.

One drawback of structure V for binomial queues is that the direction of the top-level links is special. This means that in this representation, the subforest consisting of trees whose roots are offspring of the root of a binomial tree is not represented as a binomial queue (as would be suggested by Lemma 4); the top level links are backwards. Structure R , the ring structure shown in Figure 8(b), eliminates this problem. In this structure smaller trees are always linked to larger ones, except that the largest tree points to the smallest. Downward links point to the largest subtrees, as before. It appears that structure R is slightly inferior to structure V for insertions, but is enough better for deletions to make it preferable for most priority queue applications. Structure V has some advantages for implementing the fast minimum spanning tree algorithm [4], since the ordering of subtrees helps to limit stack growth in that algorithm. (The stack can be stored in a linked fashion using the deleted nodes, thereby removing this objection to structure R .)

Neither of the structures described so far allows an arbitrary node to be deleted from a binomial queue, given only a pointer to the node. It is evident that in order for this to be possible, the structure must contain upward pointers of some sort which allow the path from any node to the root of the tree containing it to be found quickly. It must also be possible to find the queue header, since it will change during a deletion.

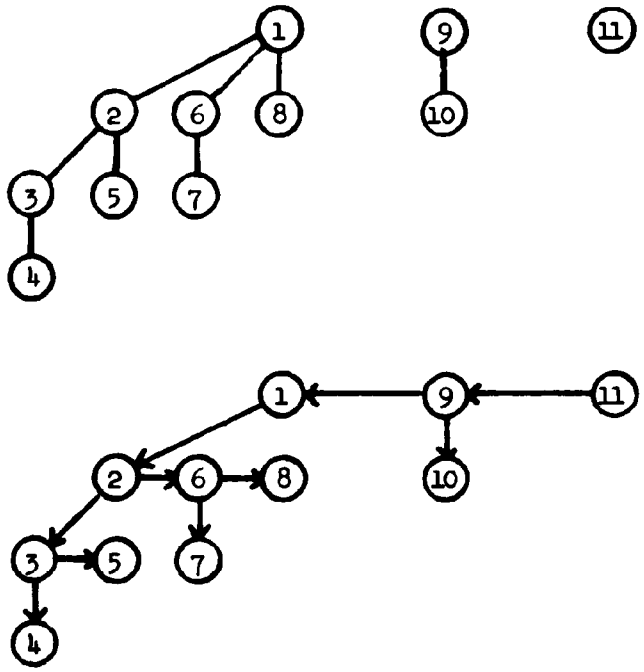
Simply adding a pointer from each node to its parent node (to the queue header in case of a root) in structure V results in a structure which allows arbitrary deletions to be performed. An example is given in Figure 9(a). Starting from any node in this structure, it is possible to follow the upward links and trace the path to the root of the binomial tree containing the node. The upward link from the root leads to the queue header, which we assume is distinguishable in some way from a queue node. Once the path to the root is known, the top-down deletion procedure described in the preceding section can be applied.

While the top-down deletion process is easy to describe, a more efficient bottom-up procedure would be used in practice. It is also essential to understand the bottom-up procedure in order to comprehend how alternative structures can be used. In the initial step of the bottom-up procedure we save all of the trees whose roots are offspring of the node to be deleted, and call this node the path node. In the general step the path node was originally the root of a B_k tree within the binomial tree being dismantled; its parent was the root of a B_ℓ tree, and we have saved $B_{k-1},\ldots,B_0$ trees so far. We first save the B_k tree formed by the right siblings of the path node, taking the path node's parent as a root. Then we save the $B_{k+1},\ldots,B_{\ell-1}$ trees which are left siblings of the path node, and make the parent of the path node the new path node. When the path node becomes the root, the process terminates. The forest of trees saved by this process is the same as that created by the top-down process, and the remaining steps of the two algorithms are identical.

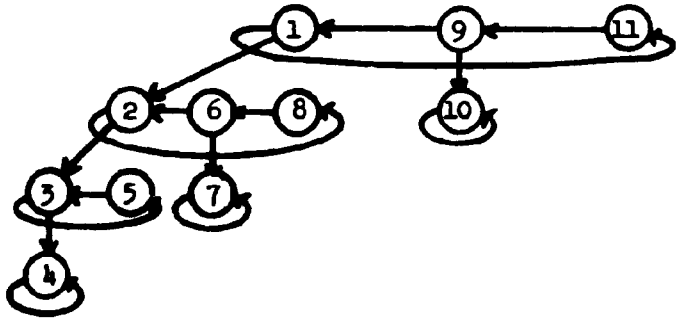
Figure 9(b) shows a modification of structure R which allows arbitrary deletions to be performed. This structure keeps an upward pointer only in the leftmost node among a group of siblings, and this pointer indicates the right sibling of the parent of nodes on this level. Note that the rightmost sibling in any family has no offspring, so the parent's right sibling always exists when needed. It is not too hard to convince oneself that the bottom-up deletion procedure just described can be performed on this structure.

Figure 9(c) shows a method of encoding the previous structure which uses only two pointers per node. The regularity of the binomial tree structure allows us to recover the information about which "child" pointers actually point upward, as follows: the rightmost node in any of the horizontal rings has no offspring (except perhaps on the top level of the forest), so its "child" pointer goes upward. If a node is an only child, or is the right sibling of a node having an only child, then it is one of these rightmost nodes. A node is an only child if and only if it is its own left sibling, so it is possible to test efficiently whether or not a "child" pointer goes upward. The upward pointer convention in Figure 9(c) is slightly irregular at the top levels; here the decoding depends on our ability to distinguish the queue header from other nodes.

Structure K, another structure which allows arbitrary deletions using only two pointers per node, is shown in Figure 9(d). This structure contains some null links, and seems to require less pointer updating per operation than the structure in Figure 9(c). Note that a path from an arbitrary node to the queue header can be found by always following "left" links, some of which go upwards. To move to the right on a given level we just follow the child pointer and then the "left" pointer.

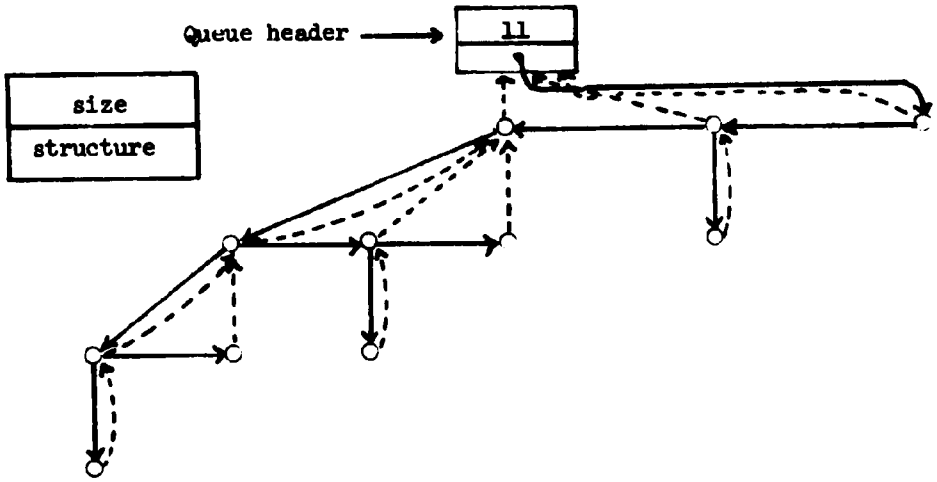


(a) A binomial queue and its representation using structure V .

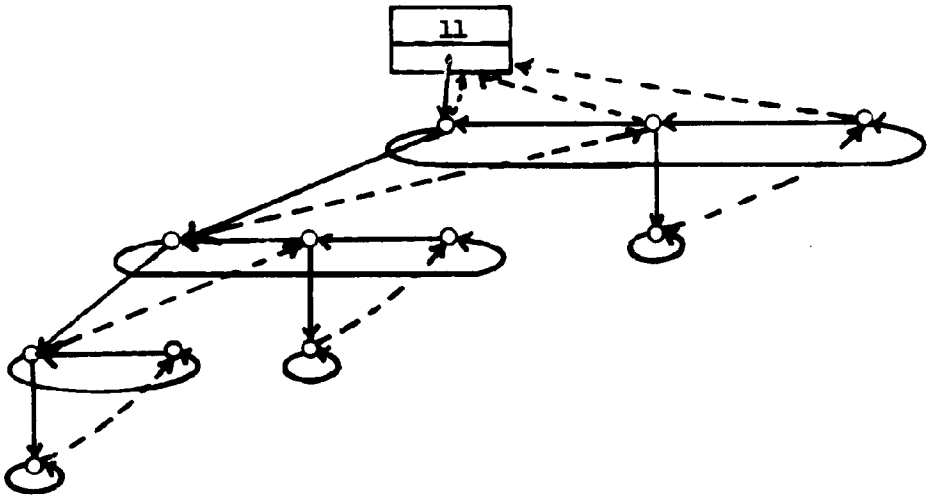


(b) A representation for the same queue using structure R .

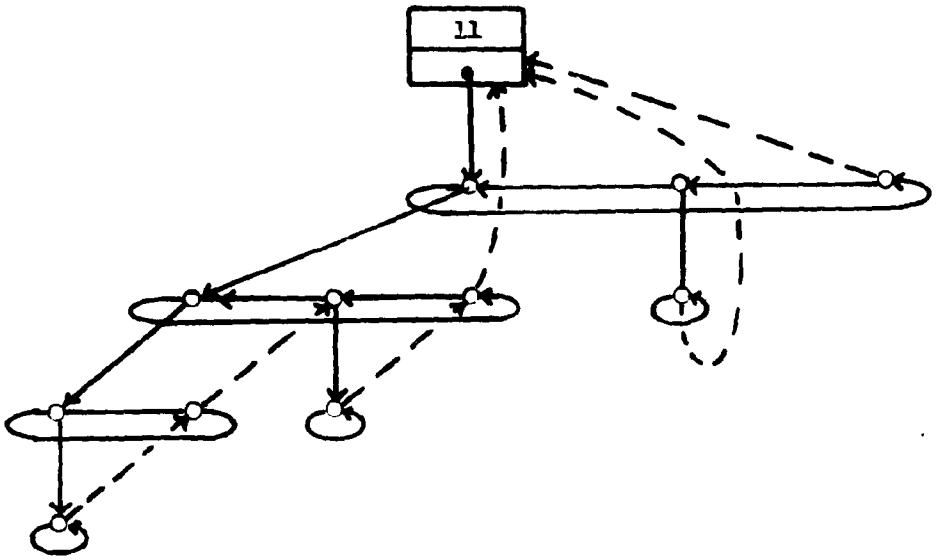
Figure 8. Structures for binomial queues.



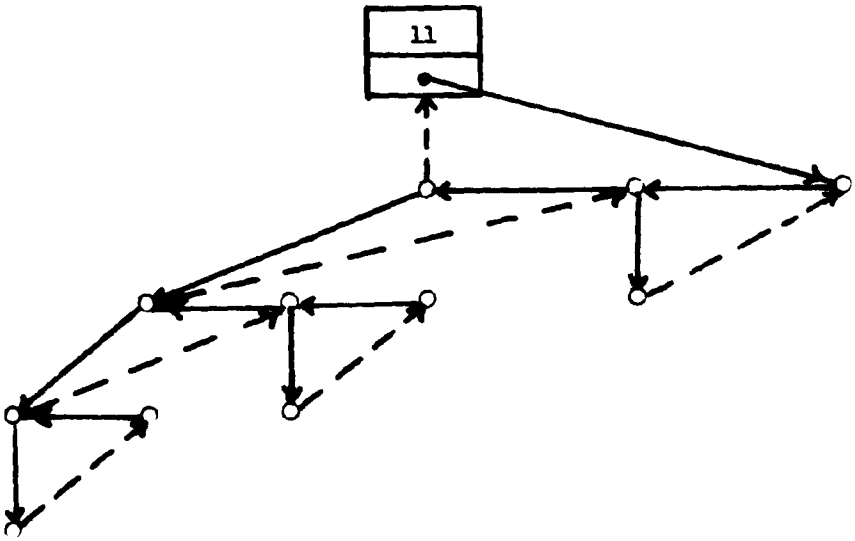
(a) Structure V with upward pointers.



(b) Structure R with upward pointers.



(c) A structure with only two pointers per node.



(d) Structure K .

Figure 9. Structures for binomial queues allowing arbitrary deletions.

2.4 Random Binomial Queues.

We define a random binomial queue of size m to be the queue formed by choosing a random permutation of $\{1,2,\ldots,m\}$ and inserting the permutation's elements successively into an initially empty binomial queue. (By a random permutation we mean a permutation drawn from the space in which all $m!$ permutations are equally likely.) Equivalently, a random binomial queue of size m is formed from a random binomial queue of size $m-1$ by choosing a random element x from $\{\frac{1}{2}, 1\frac{1}{2}, \ldots, m-\frac{1}{2}\}$, inserting x into the queue, and renumbering the queue such that the keys come from $\{1,2,\ldots,m\}$ and the ordering among nodes is preserved.

This definition of a random queue is simple, yet is not artificial. The second statement of the definition, which says that the m-th random insertion falls with equal probability into each of the m intervals defined by keys in the queue, is equivalent to another definition of random insertion which arises from event list applications. In these situations, a random insertion is obtained as follows: generate an independent random number X from the negative exponential distribution, in which the probability that $X \leq x$ is $1-e^{-x}$. Then insert the number $X+t$, where t is the key most recently removed from the queue (0 if no deletions have taken place). Here t is interpreted as the current instant of simulated time, and X is a random "waiting time" to the occurrence of some event. The fact that this definition of a random insertion is equivalent to the one we have adopted was proved by Jonassen and Dahl [22]; it follows without difficulty from the well-known "memoryless" property of the exponential distribution.

Our goal in this section is to study the structure of random binomial queues. The gross structure of such a queue is already evident; we observed earlier that all binomial forests of a given size are isomorphic. But more information about the distribution of keys in the forest is necessary to fully analyze the performance of binomial queue algorithms. For example, in order to analyze the behavior of DeleteSmallest it is necessary to determine the probability of finding the smallest element in the various trees of the binomial queue. It is also important to determine whether or not a random queue stays random after a DeleteSmallest has been performed, since if this is true then the analysis of random queues may apply even in situations where both insertions and deletions are used to build the queue.

Our first observation is that the insertion algorithm shows a certain indifference to the sizes of the elements inserted.

Lemma 5. Let $p = p_1 p_2 \ldots p_m$ be a permutation of $\{1,2,\ldots,m\}$. Then in the binomial queue obtained by inserting $p_1, p_2, \ldots, p_m$ successively into an initially empty queue, the tree containing p_j is determined by j for $j = 1,2,\ldots,m$.

Proof. We proceed by induction on m. The result is obvious for $m = 1$. For $m > 1$, let $\ell = 2^{\lfloor \lg m \rfloor}$ be the largest power of two less than or equal to m. After the first ℓ elements of p have been inserted, the queue consists of a single $B_{\lfloor \lg m \rfloor}$ tree. Later insertions have no effect on this tree, since it can only be merged with another tree of equal size. Hence the first ℓ elements of p must fall into the leftmost tree of the queue. Furthermore, since the leftmost tree is not touched, the remaining $m-\ell$ insertions distribute

the last elements of p into smaller trees as if the insertions were into an empty queue. This proves the result by induction. □

A quicker but less suggestive proof of Lemma 5 simply notes that comparisons between keys in the insertion algorithm only affect the relative placement of subtrees in the tree being constructed. Such comparisons never determine which tree is to receive a given node.

What the given proof of Lemma 5 says is that the input permutation p can be partitioned into blocks whose sizes are distinct powers of two, such that the 2^k elements of block b_k form a B_k tree when all m insertions are complete. The sizes of these blocks decrease from left to right, just as the sizes of trees in the forest decrease. (Another priority queue structure with this sort of indifferent behavior is an unsorted linear list; with the linear list, the blocks are all of size one.)

The deletion algorithm exhibits a similar dependence on when the deleted item was inserted, and a similar indifference to key sizes. What the following lemma states is that if we delete an element from a binomial queue, then the resulting queue is the same as we obtain by never inserting the element at all, but permuting the elements that we <u>do</u> insert in a manner which depends only on when the deleted element was inserted.

<u>Lemma 6.</u> Let $p = p_1 p_2 \cdots p_m$ be a permutation of $\{1,2,\ldots,m\}$. Then there is a mapping $r = r_{j,m}$ from $\{1,2,\ldots,m-1\}$ onto $\{1,2,\ldots,j-1,j+1,\ldots,m\}$ such that the result of inserting $p_1 p_2 \cdots p_m$ into an initially empty binomial queue and then deleting p_j is identical to the result of inserting $p_{r(1)} p_{r(2)} \cdots p_{r(m-1)}$ into an initially empty binomial queue.

Proof. We basically mimic the procedure for deleting p_j and then read the mapping from the result. The exact mapping depends on arbitrary choices made during the merging process and would be tedious to exhibit for general j and m, so we will give an example of the construction for $m = 10$, $j = 3$. First the input is divided into blocks as described above.

$$[00\bullet00000][][00][] \quad .$$

Then the block containing j, which holds all elements of the binomial tree T containing j in the queue, is further divided to exhibit the subtrees produced when T is dismantled.

$$[(00)\bullet(0)(0000)][][00][] \quad .$$

This division clearly depends only on m and j.

Following the dismantling step is a merging step. One possible strategy for this merge is as follows. If the dismantled binomial tree T was the smallest tree in the original queue, then no merging is required. Otherwise combine the smallest tree in the original queue with the forest just obtained by dismantling T. This produces a new tree which has the same size as T had, plus a forest of small trees; the merge is then complete. The same effect would be created (in the case we are considering) by reinserting all nodes in the order

$$(0000)(00)[00](0) \quad .$$

To see this, just simulate the insertion process on this input. The intermediate trees created during this process correspond to trees involved in the merge. (Note that the r map is far from being uniquely determined.) □

Here again we can draw an analogy with the unsorted linear list, which obviously has the behavior stated in the lemma.

Armed with this result, we can determine the effects of various types of deletions on random binomial queues.

Theorem 1. Let Q be a random binomial queue of size m. Suppose that p_k, the k-th element inserted in the process of building Q, is deleted from Q and Q is renumbered. Then the resulting Q is a random binomial queue of size $m-1$.

Proof. Consider the $m!$ equally-likely permutations used to build Q. When the k-th element of each permutation is discarded and the permutation renumbered, each of the $(m-1)!$ possible permutations occurs m times. The same is true if some fixed rearrangement of the permutation is made just before the renumbering. Hence by Lemma 6 the $m!$ queues obtained by inserting all possible permutations of length m and then deleting the k-th element (and renumbering) are just m copies of the $(m-1)!$ queues obtained by inserting all permutations of length $m-1$. □

Theorem 2. Let Q be a random binomial queue of size m. Suppose that k, the k-th smallest element inserted in the process of building Q, is deleted from Q and Q is renumbered. Then the resulting Q is a random binomial queue of size $m-1$.

Proof. Consider the $m!$ equally-likely permutations used to build Q. For fixed j, there are $(m-1)!$ of these permutations with $p_j = k$; if we ignore p_j and renumber, we get all $(m-1)!$ possible permutations of $\{1,2,\ldots,m-1\}$. The same is true if some fixed rearrangement of the

permutation is made before renumbering. Hence by Lemma 6 the $(m-1)!$ queues obtained by inserting all permutations of length m with $p_j = k$ and then deleting k (and renumbering) are just the $(m-1)!$ queues obtained by inserting all permutations of length $m-1$. This is true for each j, so the result follows. □

Corollary 1. If a random element (or randomly placed element) of the input is deleted from a random binomial queue of size m, the result is a random binomial queue of size $m-1$.

Proof. The two statements are obviously equivalent; they follow immediately from Theorem 1 or Theorem 2. □

These results are sufficient to show that binomial queues stay random in many situations. The most important of these is when a queue is formed by a sequence of n random Insert operations intermixed with $m \leq n$ occurrences of DeleteSmallest, arranged so that a deletion is never attempted when the queue is empty. Theorem 2 shows that a DeleteSmallest applied to a random queue leaves a random queue; a random Insert also preserves randomness. So under the most reasonable assumptions for priority queues, binomial queues can be treated as random. This is our rationale for assuming random binomial queues in the analysis of the next section.

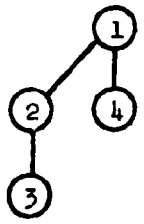
A similar argument shows that random binomial queues result when intermixed random deletions are performed; a simple argument appealing to Lemma 6 shows that intermixed deletions by age (how long an element has been in the queue) also lead to random queues. These types of deletions, especially deletions by age, are somewhat artificial for priority queues.

It is natural to ask whether randomness is preserved by the merging of binomial queues. Suppose that a random permutation of length m is given; its first k elements are inserted into one initially empty binomial queue, and the remaining $m-k$ elements are inserted into another. Then each of these queues is a random binomial queue, and the argument used to prove Lemma 6 shows that the result of merging these queues is also random as long as some fixed choice is made about which two trees to "add" when three are present during the merge. So in this sense merging does preserve randomness.

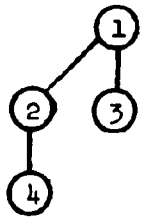
Sensitivity to deletions has been studied in the context of binary search trees by Knott [25]. The model used there considers a random insertion to be the insertion of a random real number drawn independently from some continuous distribution (for example, uniform on the interval $[0,1]$.) This definition is not equivalent to ours; Theorem 1 and Corollary 1 hold for deletions from binary search trees, but this does not imply that a tree built using intermixed random deletions is random. In fact, as Knott first noted, binary search trees are sensitive to deletions in this model.

Binomial queues, however, are not sensitive to deletions in the search tree model. In a general study of deletion insensitivity, Knuth showed that Theorem 2 implies insensitivity to random deletions, and Lemma 6 implies insensitivity to deletions by age in this model [29]. Binomial queues are sensitive to deletions by order (e.g., DeleteSmallest) in this model, but unsorted linear lists, as well as practically all other algorithms, are also sensitive to these deletions. So even with this alternative definition of a random insertion, random binomial queues tend to remain random when deletions are present.

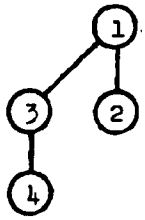
At this point it might seem that nothing can destroy a random binomial queue! This is not true; a deletion based on knowledge of the structure of the queue (or equivalently, knowledge of the entire input) can easily introduce bias. For example, random queues of size 4 are distributed as shown:



pr = 1/3

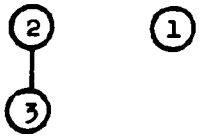


pr = 1/3



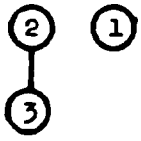
pr = 1/3

If we now delete the rightmost child of the root and renumber, we get:

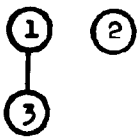


pr = 1

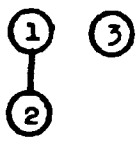
This isn't random; random binomial queues of size 3 have the distribution



pr = 1/3



pr = 1/3



pr = 1/3

Since the analysis of binomial queue algorithms performed in the next section is based on random binomial queues, we are interested in the distribution of keys in these queues. By Lemma 5, the probability that

a given element (e.g. the smallest) of a random permutation lies in a given binomial tree is simply the probability that the element lies in a certain block of positions within the permutation. Thus the probability that the j-th largest element in a binomial queue of size m lies in a B_k tree is just $2^k/m$, assuming that a B_k tree is present in a queue of size m. This decomposition of the input into blocks reduces the study of random binomial queues to the study of random heap-ordered binomial trees (i.e., random binomial queues of size 2^k).

As we observed earlier, the smallest key in a heap-ordered binomial tree must be in the root. The distribution of larger keys is not so highly constrained. The following result characterizes the distribution of keys without explicit reference to the $n!$ possible input permutations.

Theorem 3. Let a configuration of a heap-ordered B_k tree be any assignment of the integers $1,2,\ldots,2^k$ to the nodes of a B_k tree such that the tree is heap-ordered. Then in a random heap-ordered B_k tree all $\frac{(2^k)!}{2^{(2^k)-1}}$ configurations are equally likely. (That is, there are $2^{(2^k)-1}$ distinct input permutations which generate each possible configuration.)

Proof. We proceed by induction on k. The result is obvious for $k = 0$. Assume that for $k = j$ there are $2^{(2^j)-1}$ permutations of $\{1,2,\ldots,2^j\}$ which give size to each possible configuration.

Now consider any fixed configuration X of a B_{j+1} tree. This tree can be decomposed into the two B_j trees Y and Z, as shown in Figure 1. By the argument of Lemma 5, any permutation giving rise to

configuration X must consist of two blocks, one producing Y and the other Z; these blocks may appear in either order, since the relative position of Y and Z is determined by which tree contains the smallest key. By the induction hypothesis there are $2^{(2^j)-1}$ arrangements of the keys in tree Y which give rise to Y, and similarly for Z. So there are $2 \cdot 2^{(2^j)-1} \cdot 2^{(2^j)-1} = 2^{(2^{j+1})-1}$ permutations which produce X. Since this holds for any X, the result follows. □

In the inductive step above, we can note that the element 1 is equally likely to be contained in the first or the second block of a permutation producing X. This leads to an easy inductive proof of the proposition that the i-th inserted element is equally likely to fall into each of the 2^k nodes of a random heap-ordered B_k tree.

Unfortunately, Theorem 3 does not help much in determining the exact distribution of keys in a random binomial tree. There are fewer configurations than permutations, but the number of configurations still increases rapidly with k.

2.5 Analysis of Binomial Queue Algorithms.

We are now prepared to analyze the performance of Insert and DeleteSmallest, when implemented using binomial queues; this will allow a comparison with other priority queue organizations. The binomial queue implementation to be analyzed is based on structure R, discussed in Section 2.3 and pictured on page 32. The priority queue structures to be used for comparison are the heap, leftist tree, sorted linear list, and unsorted linear list.

For each of the five structures, the operations Insert and DeleteSmallest have been carefully coded in FAIL, a PDP-10 assembly language (the binomial queue and leftist tree implementations appear in Appendix A.) By inspecting these programs, we can write expressions for their running time as a function of how often certain statements are executed. It then remains to determine the average values of these factors.

The running times (in memory references for instructions and data) of the binomial queue operations are

Insert -- 16 + 19M + 2E + 6A

DeleteSmallest -- 38 + 11B + 6T + 4N - 2L + 4S + 14U + 2X

where

M is the number of merges required for the insertion;

E is the number of exchanges performed during these merges in order to preserve the heap-order property;

A is 1 if M = 0, and 0 otherwise;

B is 1 if the queue contains no B_0 tree, and 0 otherwise;

T is the number of binomial trees in the queue;

N is the number of times that the value of the smallest key seen so far is changed during the search for the root containing the smallest key;

L is 1 if the smallest key is contained in the leftmost root, and 0 otherwise;

S is the number of offspring of the root containing the smallest key;

U is the number of merges required for the deletion; and

X is the number of exchanges performed during these merges.

(To keep the expression for DeleteSmallest simple, certain unlikely paths through the program have been ignored. The expression above always overestimates the time required for these cases.)

As a first step in the analysis we note that several of the factors above depend only on the structure of the binomial queue Q and not on the distribution of keys in Q. Since the structure of Q is determined solely by its size, these factors are easy to determine. For example, if Q has size m then evidently M is the number of low-order 1 bits in the binary representation of m, and $A = 1$ if and only if m is even. Clearly $B = A$, and by Lemma 2 we can see that $T = \nu(m)$.

These factors are a bit unusual in that they do not vary smoothly with m. For example, when $m = 2^n-1$ we have $M = T = n$, while for $m = 2^n$ this changes to $M = 0$ and $T = 1$. Since in practice we are generally concerned not with a specific queue size m but rather with a range of queue sizes in the neighborhood of m, it makes sense to average the performance of our algorithms over such a neighborhood.

Factors A and M can be successfully smoothed by this approach: averaging over the interval $[m/2, 2m]$ gives an expected value of

$\frac{1}{2} + O\left(\frac{1}{m}\right)$ for A and $1 + O\left(\frac{\log m}{m}\right)$ for M. This agrees well with our intuition, since it says that about half of the integers in the interval are even, and that one carry is produced, on the average, by incrementing a number in the interval.

Properties of the factor $T = \nu(m)$ have been studied extensively. From [35] we find that

$$\left\lfloor \tfrac{1}{2} m \lg\left(\tfrac{3}{4} m\right) \right\rfloor \le \sum_{1 \le k \le m} \nu(k) \le \left\lfloor \tfrac{1}{2} m \lg m \right\rfloor ,$$

where each bound is tight for infinitely many m; it follows that our neighborhood averaging process will not completely smooth the sequence $\nu(m)$. But we have bounds on an "integrated" version of $\nu(m)$, so differentiating the bounds puts limits on the average growth rate of $\nu(m)$. Carrying out the differentiation gives

$$\tfrac{1}{2}\left(\lg m + \frac{1}{\ln 2} - \lg \tfrac{4}{e} \right) \le T_{\text{avg.}} \le \tfrac{1}{2}\left(\lg m + \frac{1}{\ln 2} \right) ,$$

which is about what we expect: half of the bits are 1, on the average. The remaining uncertainty in the constant term is about .21.

While this averaging technique fails to smooth the sequence $\nu(m)$ completely, there are other methods which succeed. There is no single "correct" method for handling problems of this type: different techniques may give different answers, and the usefulness of a result depends on how "natural" the smoothing method is in a given context. The more powerful averaging techniques which succeed in smoothing $\nu(m)$ seem artificial in connection with our analysis, but the results are quite interesting mathematically. Lyle Ramshaw [39] has shown that

$$\nu(m) \approx \frac{\lg m}{2} + \left(\frac{\lg \pi}{2} - \frac{1}{4} \right) \doteq \frac{\lg m}{2} + 0.57575$$

using logarithmic averaging [7]; his result is based on the detailed analysis of $\sum_{1 \le k \le m} \nu(k)$ performed by Hubert Delange [6]. The naturalness of logarithmic averaging is indicated by the fact that it also leads to the logarithmic distribution of leading digits which has been observed empirically [38;27, Section 4.2.4], and the fact that it is consistent with several other averaging methods (such as repeated Cesaro summing) when those methods define an average.

This analysis of factor T completes the purely "structural" analysis; the remaining factors depend on the distribution of keys in the queue. For the average-case analysis we shall assume that Q is a random binomial queue of size m and that the insertion is random. These assumptions are well justified by the discussion of Section 2.4.

The factors E and X are easy to dispose of. Since we only merge trees of equal size, our randomness assumption says that an exchange is required on half of the merges (on the average). More precisely, if there are n merges then the number of exchanges is binomially distributed with mean $n/2$ and variance $n/4$. The number of merges is just M in the case of E, and U in the case of X.

The factors L and S are also easy to analyze. We noted in Section 2.4 that the probability of having the queue's smallest key in a given tree is just proportional to the size of the tree. Therefore if there is a binomial tree of size 2^k in a queue of size m, this tree contains the queue's smallest key with probability $2^k/m$. The root of such a binomial tree has k offspring by Lemma 1, so the expected value

of S is $\frac{1}{m}F(m)$ where

$$F(m) = \sum_{\substack{k\geq 0 \\ m=(b_\ell b_{\ell-1}\ldots b_0)_2}} b_k \cdot k \cdot 2^k \quad .$$

While it seems hard to find a simpler closed form for $F(m)$, it is possible to derive good upper and lower bounds.

Lemma 8. The function $F(m)$ defined above satisfies $\lceil m \lg m - 2m \rceil \leq F(m) \leq \lfloor m \lg m \rfloor$, $m \geq 1$, and the upper bound is tight for infinitely many values of m.

Proof. (This argument is similar to the one used to prove Theorem 1 in [25].) From the definition of $F(m)$, if $m = 2^k$ then

$$F(m) = F(2^k) = k \cdot 2^k = m \lg m \quad .$$

It is also clear from the definition that

$$F(2^k+i) = F(2^k) + F(i) \quad , \quad 0 \leq i < 2^k \quad .$$

The upper bound on $F(m)$ is evidently attained whenever m is a power of two. It therefore holds when $m = 1$, and assuming that it holds up to $m = 2^k$, we have

$$\begin{aligned} F(m+i) &= F(2^k) + F(i) \qquad (0 \leq i < 2^k) \\ &\leq m \lg m + i \lg i \\ &\leq (m+i) \lg(m+i) \quad . \end{aligned}$$

So the upper bound holds for all m by induction.

The lower bound on $F(m)$ holds when $m = 1$, and whenever m is a power of two. Suppose that a bound of the form

$$F(m) \geq m \lg m - c\,m$$

is true for some $c > 0$ and all $m \leq 2^k$. Then

$$\begin{aligned} F(m+i) &= F(2^k) + F(i) \qquad (0 \leq i < 2^k) \\ &\geq m \lg m + i \lg i - c\,i \quad . \end{aligned}$$

It follows that if the inequality

$$m \lg m + i \lg i - c\,i \geq (m+i) \lg(m+i) - c(m+i) \qquad (0 \leq i < 2^k)$$

holds, the lower bound will hold for all m by induction. Replacing i by xm and simplifying gives another inequality which implies the result:

$$x \lg x \geq (1+x) \lg(1+x) - c \qquad (0 \leq x < 1) \quad .$$

But it is easy to verify that $x \lg x - (1+x) \lg(1+x)$ is decreasing on $[0,1]$, so we can take $x = 1$ to determine $c = 2$. (A tight lower bound can be found by using the value $F(2^k-1) = (k-2)2^k+2$.) □

According to these bounds, the average value of S lies between $\lg m - 2$ and $\lg m$. Lyle Ramshaw [39] has shown that the logarithmically averaged value of S is

$$\frac{F(m)}{m} = \lg m - C$$

where

$$C = \left(\frac{1}{\ln 2} \sum_{n \geq 1} \sum_{j > 2^n} \frac{(-1)^{j+1}}{j} \right) + \frac{1}{2} \doteq 1.10995 \quad .$$

The expected value of L is $2^{\lfloor \lg m \rfloor}/m$, which is between $1/2$ and 1.

Factor U is closely related to S. The number of merges required is equal to the number of trees (i.e., S) created by removing the node containing the smallest key, minus the number of these trees which are not merged. Since the first merge must take place with the smallest tree remining in the original forest, we see that the number of trees excluded from merging is equal to the number of low-order 0 bits in m. Since the least significant bits of m are distributed almost uniformly, the average value of this quantity $S-U$ is the same as the average value of M.

Factor N is more interesting. One way to search for the smallest root in the forest is to use the key contained in the rightmost root as an initial estimate for the smallest key, and then scan the forest from right to left, updating the estimate as smaller keys are seen. Since the trees increase in size from right to left, trees in the left of the forest are more likely to contain the smallest key; thus the estimate of smallest key will be changed often during the scan. To be more precise, the expected number of changes while searching a forest of size $m = (b_n b_{n-1} \ldots b_0)_2$ is

$$\sum_{\substack{0\le k\le n\\ b_k=1}} \Pr(\text{estimate changes when the } B_k \text{ tree is examined})$$

$$= \sum_{\substack{0\le k\le n\\ b_k=1}} \frac{(\text{number of nodes in the } B_k \text{ tree})}{(\text{total number of nodes in all } B_\ell \text{ trees examined, } 0\le \ell\le k)}$$

$$= \sum_{\substack{0\le k\le n\\ b_k=1}} \frac{2^k}{\sum_{\substack{0\le \ell\le k\\ b_\ell=1}} 2^\ell} \quad .$$

When $m = 2^n-1$ this has the simple form

$$\frac{2}{3}+\frac{4}{7}+\frac{8}{15}+\ldots+\frac{2^{n-1}}{2^n-1}$$

$$= \left(\frac{1}{2}+\frac{1}{2}+\frac{1}{2}+\ldots+\frac{1}{2}\right)+\frac{1}{2}\left(\frac{1}{3}+\frac{1}{7}+\frac{1}{15}+\ldots+\frac{1}{2^n-1}\right)$$

$$= \frac{n}{2}+\frac{1}{2}(\alpha-1)+O(2^{-n})$$

where $\alpha = \sum_{k\ge 1} \frac{1}{2^k-1} = 1.60669\ldots$.

(The constant α also arises in connection with Heapsort; see [28, 5.2.3(19)].)

A search strategy which intuitively seems superior to the one just described is to search the forest from left to right; for the above example the expected number of changes is reduced to

$$\frac{1}{3}+\frac{1}{7}+\frac{1}{15}+\ldots+\frac{1}{2^n-1} = \alpha-1+O(2^{-n}) \quad .$$

But this strategy is not practical; the links point in the wrong direction. With structure R we can improve the search by using the key contained in

the leftmost root as our initial estimate in a right to left search. This makes the expected number of changes in a queue of size 2^n-1 equal to

$$\frac{1}{2^{n-1}+1} + \frac{2}{2^{n-1}+2+1} + \frac{4}{2^{n-1}+4+2+1} + \cdots + \frac{2^{n-2}}{2^n-1} + \frac{2^{n-1}}{2^n-1} \quad .$$

By writing this sum in reverse order we can derive its asymptotic value:

$$\frac{2^{n-1}}{2^n-1} + \frac{2^{n-2}}{2^n-1} + \frac{2^{n-3}}{2^n-2^{n-2}-1} + \frac{2^{n-2}}{2^n-2^{n-2}-2^{n-3}-1} + \cdots \frac{1}{2^{n-1}+1}$$

$$= \frac{1/2}{1-2^{-n}} + \frac{1/4}{1-2^{-n}} + \frac{1/8}{1-\frac{1}{4}-2^{-n}} + \frac{1/16}{1-\frac{1}{4}-\frac{1}{8}-2^{-n}} + \cdots + \frac{1/2^n}{\frac{1}{2}+2^{-n}}$$

$$= \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{2}\left(\frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{2^{\lfloor n/2 \rfloor}+1}\right)\right)(1 + O(2^{-n/2})) + O(2^{-n/2})$$

$$= \frac{3}{4} + \frac{1}{2}\left(\alpha' - \frac{1}{2}\right) + O(2^{-n/2})$$

where $\alpha' = \sum_{k\geq 0} \frac{1}{2^k+1} = 1.26449\ldots$.

(The constant α' arises in connection with merge sorting; see [28, exercise 5.2.4-13].) So the expected value of this factor is about 1.13 for large n ; by modifying the search in this way we have effectively removed part of the inner loop.

This completes the analysis of Insert and DeleteSmallest for binomial queues. By plugging our average values into the running time expressions given above and simplifying, we get the results for binomial queues shown in Figure 10. A much simpler analysis [26, pp. 94-99] gives the corresponding results for sorted and unsorted linear lists (also shown in Figure 10.)

Priority queue algorithms based on heaps and leftist trees have not been completely analyzed; partial results are known for heaps [28;37] but not for leftist trees. Therefore experiments were performed to determine the average values of factors controlling the running time of these algorithms. Leftist trees and heaps are deletion sensitive, so the averages were taken from stationary structures (obtained after repeated insertions and deletions) rather than from random structures. Figure 10 gives the experimentally determined running times for leftist trees and heaps.

These results indicate that binomial queues completely dominate leftist trees. Not only do binomial queues require one fewer field per node, they also run faster, on the average, for $m \geq 4$ when the measure of performance is the cost of an Insert followed by a Delete . Linear lists are of course preferable to both of these algorithms for small m , but binomial queues are faster than unsorted linear lists, on the average, for $m \geq 18$ at a cost of one more pointer per node. So the binomial queue is a very practical structure for mergeable priority queues.

In some applications the queue size may constantly be in a range which causes the insertion and deletion operations on binomial queues to run more slowly than our averages indicate, due to the smoothed average we computed. If the queue size can be anticipated then dummy elements added to the queue might actually speed up the algorithms. At the expense of complicating the algorithms it is also possible to maintain a queue as two binomial forests in such a way that each insertion is guaranteed to take only constant time. But the binomial queue algorithms as they stand still dominate algorithms using leftist trees, even if the leftist tree

operations have average-case running times and the binomial queue operations always take the worst-case time. The only advantages which can apparently be claimed for leftist trees is that they are easier to implement and can take advantage of any tendency of insertions to follow a stack discipline.

The comparison of binomial queues with heaps and sorted linear lists is also interesting. The heap implementation stores pointers in the heap, instead of the items themselves; this is the usual approach when the items are large and should not be moved. In this situation heaps are slightly faster than binomial queues on the average, and considerably faster in the worst case. Heaps also save one pointer per node, so it seems that heaps are preferable to binomial queues when fast merging is not required. Binomial queues have an advantage when sequential allocation is a problem, or perhaps when arbitrary deletions must be performed. Sorted linear lists are better than both methods when m is small, but heaps are faster, on the average, when $m \geq 30$.

average case running times when $|Q| = m$.

queue	Insert(x,Q)	DeleteSmallest(Q)	Insert(x,Q); DeleteSmallest(Q)
binomial queue	39	22 lg m + 19	22 lg m + 58
leftist tree	17 lg m + 35	35 lg m - 27	52 lg m + 8
linear list	19	6m + 2 lg m + 20	6m + 2 lg m + 39
heap	32	18 lg m + 1	18 lg m + 33
sorted list	3m + 17	15	3m + 32

worst case running times when $|Q| = m$.

queue	Insert(x,Q)	DeleteSmallest(Q)	Insert(x,Q); DeleteSmallest(Q)
binomial queue	21 lg m + 16	30 lg m + 46	51 lg m + 62
leftist tree	32 lg m + 23	64 lg m - 7	96 lg m + 16
linear list	19	9m + 15	9m + 34
heap	12 lg m + 14	18 lg m + 16	30 lg m + 30
sorted list	6m + 20	15	6m + 35

Figure 10. Comparison of methods.

Chapter Three. The Complexity of Priority Queue Maintenance

The inherent complexity of sorting, selection, and related problems has been studied extensively [28]. The complexity of inserting and deleting items from a priority queue has not received such attention, possibly because the individual operations take constant time in certain cases. Priority queues may be used to perform sorting, however; hence it is clear that there is some limit to the average efficiency of a sequence of priority queue operations.

In Section 3.1 of this chapter we develop a definition of priority queue efficiency, based on the average number of comparisons required to execute a certain fixed pattern of insertions and deletions. We evaluate some known priority queue algorithms in Section 3.2 to obtain upper bounds on the average and worst case behavior of priority queues. In Section 3.3 we prove lower bounds on the average and worst case efficiency; these bounds are exact for infinitely many queue sizes.

One of the priority queue algorithms discussed in Section 3.2, based on binomial queues, has a simple characterization which is proved in Section 3.4: it is the only algorithm which compares only "unbeaten" nodes (nodes which are smaller than all nodes in the queue with which they have been compared) and which takes a number of comparisons independent of the key values involved. The proof given for this result uses a lemma involving two extremal problems on trees; we show that Huffman's construction [16;26, pp. 402-405] solves these problems. This is especially interesting since the problems lie outside the large domain for which Huffman trees were proved optimal in [13].

We conclude the chapter with a discussion of possible generalizations and open problems in Section 3.5.

3.1 A Setting for Priority Queue Complexity.

We shall investigate the complexity of priority queues in the following context. Initially we are given a priority queue containing m elements. We then perform an infinite number of cycles which consist of first deleting the smallest element of the queue, and then inserting a new element into the queue. A typical cycle of this infinite process is represented pictorially in Figure 1. (It is easy to imagine other settings in which to study the complexity of priority queues, but we defer discussion of this topic until Section 3.5.)

Our measure of the performance of a priority queue will be based on the number of comparisons made between keys during the above process. Therefore we restrict our attention to priority queue methods which are based entirely on the linear ordering among keys. This means, for example, that none of our methods may perform arithmetic on keys. We shall further assume that all keys are distinct, so that there are only two possible outcomes from any comparison.

Note that because we make so few assumptions about what goes on inside a priority queue, it is possible that the number of comparisons used on any particular cycle is zero. A deletion takes no comparisons if the smallest element is known prior to the deletion, and an insertion takes no comparisons if the queue just stores the inserted node without looking at it. In order to cope with this sort of anomaly, we use as our measure of cost the number of comparisons per cycle averaged over infinitely many cycles. More precisely, if a method uses $C(n,m)$ comparisons to perform n cycles on a queue of size m, then its limit cost per cycle is $\lim_{n\to\infty} \frac{C(n,m)}{n}$. We denote by $Q(m)$ the minimum limit cost per cycle

which suffices (in the worst case) to maintain a priority queue of size m. We define $\tilde{Q}(m)$ to be the minimum average limit cost per cycle, where insertion into each of the m intervals bounded by the $m-1$ key values in the queue is taken to be equally likely during each cycle.

Because we are averaging the number of comparisons over many cycles, it is possible to make a further assumption about the internal structure of our priority queues: we may assume that at the beginning of each cycle the smallest element of the queue is known. This is a valid assumption because it just amounts to charging the previous cycle for whatever comparisons are required at the start of a cycle to determine the smallest. Since we are averaging over infinitely many cycles this cannot change the result.

This additional property allows us to make a slightly less abstract interpretation of the situation. The state of a priority queue at the beginning of a cycle can be represented as a directed acyclic graph, where the arcs indicate comparisons which have been made in the process of maintaining the queue. An arc leads from a node with key K_i to a node with key K_j if the comparison $K_i : K_j$ was made and $K_i < K_j$ was the result. As indicated in Figure 2, the graph has a single source node, containing the smallest key, at the start of each cycle. Then this node is removed from a graph, corresponding to deleting the smallest element of the queue, and a new node is added to the graph. This node is inserted into the queue by performing enough comparisons (adding directed arcs) to again determine the smallest element (obtain a graph having a single source).

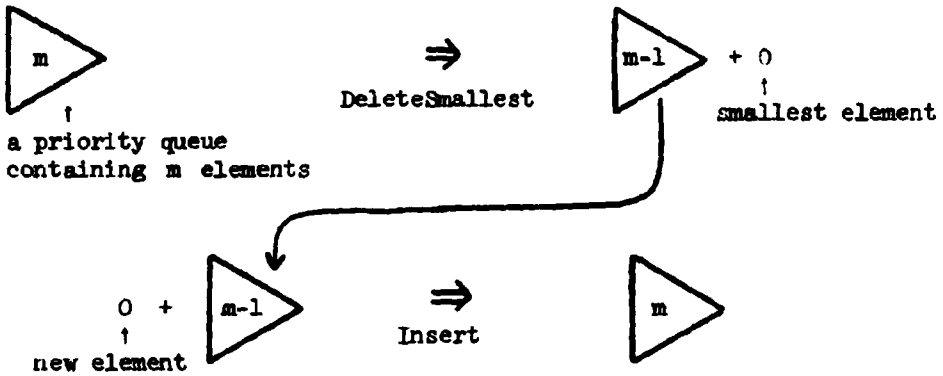
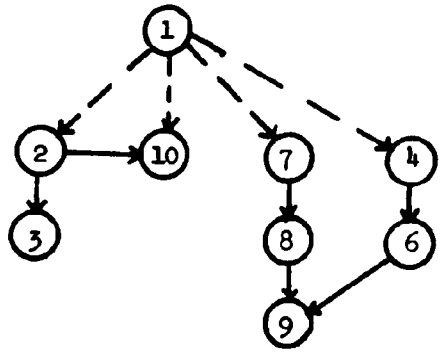
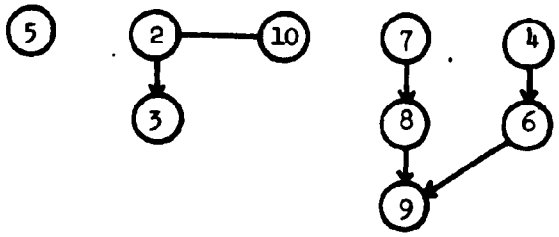


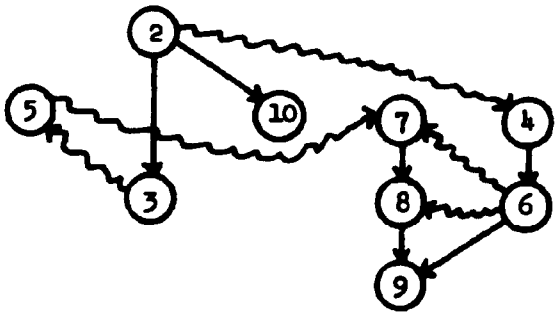
Figure 1. A priority queue cycle.



(a) The start of a cycle (comparisons to be lost by deletion of smallest are shown dashed).



(b) After removal of the smallest node and introduction of the inserted node (5).



(c) Insertion complete (added comparisons are shown wavy).

Figure 2. Directed graph interpretation of a priority queue cycle.

3.2 Upper Bounds.

Upper bound on $Q(m)$ are provided by the priority queue structures described in Section 1.2. Heaps and leftist trees each require about $2 \lg m$ comparisons, in the worst case, for a cycle on a queue of size m. Queues based on balanced trees also require at least $c \lg m$ comparisons for some constant $c > 1$, in the worst case. But some queue structures reduce the coefficient of $\lg m$ to 1.

One such structure is the sorted linear list. The smallest element of such a list can be deleted using no comparisons; an insertion into a list of $(m-1)$ items requires at most $\lceil \lg m \rceil$ comparisons, using binary search. It follows that $Q(m) \leq \lceil \lg m \rceil$. More detailed analysis of binary search [28, p. 194] shows that the average number of comparisons required is $\lg m + (1 + \theta - 2^{\theta})$ where $\theta = \lceil \lg m \rceil - \lg m$; hence $\bar{Q}(m) \leq \lg m + (1 + \theta - 2^{\theta})$. The function $(1 + \theta - 2^{\theta})$ is nonnegative and has a maximum value of about 0.0861 for θ in the range $0 \leq \theta < 1$.

There is good reason to feel uneasy about these bounds, since when this priority queue is implemented using simple linked or sequential list structures its average running time for a cycle is $O(m)$. Fortunately there is a more legitimate structure which gives exactly the same comparison bounds as a sorted linear list: the "loser-oriented" tree used for replacement selection [28, p. 253]. Using this structure, the running time for a cycle is proportional to the number of comparisons performed.

A third structure which gives a good upper bound is a variant of the binomial queue. The standard binomial queue algorithms given in Section 2.2 can require about $2 \lg m$ comparisons for DeleteSmallest;

it may take $\lg m$ to find the smallest, and $\lg m$ to merge its offspring back into the forest. An Insert operation also requires up to $\lg m$ comparisons, so an entire cycle may take about $3 \lg m$ comparisons.

To reduce the number of comparisons required by a binomial queue cycle, add nodes containing the key $+\infty$ to the queue, making the queue size 2^k where $k = \lceil \lg m \rceil$. Then at the start of a cycle the queue consists of a single heap-ordered B_k tree, and the DeleteSmallest operation requires no comparisons. The following Insert uses k comparisons, so this method requires exactly $\lceil \lg m \rceil$ comparisons per cycle. This shows that $Q(m) \leq \lceil \lg m \rceil$, giving the same bound on the worst-case number of comparisons per cycle as was given by the sorted linear list (or loser tree). Since this binomial queue algorithm requires a fixed number of comparisons per cycle, independent of key values, it does not give any better bound for $\tilde{Q}(m)$ than its bound for $Q(m)$.

3.3 Lower Bounds.

One approach to lower bounds on the number of comparisons required to maintain a priority queue is to analyze the possible internal states of the queue at the start of each cycle. This is possible under certain restrictions, as shown in the next section, but to get a general lower bound seems to require a different sort of argument which takes advantage of the long-term averaging present in our model.

It turns out to be quite easy to prove a good lower bound on $\bar{Q}(m)$. Suppose that a priority queue of size m requires $C(n,m)$ comparisons, on the average, to perform n cycles. The number of equally-likely outcomes of these cycles is m^n, since there are m equally-likely relative sizes for each of the n keys inserted during the cycles. So by the same decision-tree argument used to prove lower bounds for sorting [28, p. 194], the average number of comparisons required to determine which outcome has occurred is at least $n \lg m$. But it is possible to determine this outcome by observing the outputs of the n queue cycles and sorting the m keys which remain in the queue. Hence

$$n \lg m \leq C(n,m) + O(m \log m)$$

so

$$\lg m \leq \overline{\lim_{n \to \infty}} \frac{C(n,m) + O(m \log m)}{n} = \bar{Q}(m) .$$

This also implies that $Q(m) \geq \lg m$.

We can summarize the results of this and the previous section as follows.

Theorem 1. The functions $Q(m)$ and $\tilde{Q}(m)$ defined in Section 3.1 satisfy

$$\lg m \leq Q(m) \leq \lceil \lg m \rceil \quad , \quad \text{and}$$

$$\lg m \leq \tilde{Q}(m) \leq \lg m + (1 + \theta - 2^{\theta}) \quad ,$$

where $\theta = \lceil \lg m \rceil - \lg m$. □

3.4 A Characterization of Binomial Queues.

The foregoing results show that binomial queues are optimal when the queue size is a power of 2. When the queue size is not a power of 2, then dummy nodes can be added as described in Section 3.2 to make binomial queues nearly optimal; in the rest of this section, we shall use the term "binomial queue" to refer to such a structure.

We have seen that linear lists and loser trees also require $\lg m$ comparisons per cycle when $m = 2^k$, so binomial queues are not the only optimal structure for this problem. But neither linear lists nor loser trees are the basis for a practical priority queue algorithm. We have already noted that comparisons do not reflect the actual running time of a priority queue using a sorted linear list; loser trees work well for replacement selection but are awkward to use when the priority queue size may change with time. Our comparison model evidently does not capture the factors which make a given priority queue scheme difficult to implement.

It seems extremely difficult to evaluate the true complexity of data structuring problems; the linking automaton [31] appears to be a good setting for such questions, but few results have been obtained for this model. A more limited approach, closer in spirit to the comparison-counting model of complexity, is to restrict our consideration to algorithms such that the underlying structures must be easy to implement. Suppose that the only elements which may participate in comparisons are those which are not known to be larger than any other element of the queue. In terms of our directed graph description of a priority queue, such elements have no entering edges; they are candidates for being the smallest element in the queue. We call comparisons between such elements tree comparisons,

since they preserve the property that the directed graph structure of the queue is a tree. (It is not hard to see that a queue which makes a non-tree comparison can eventually have an internal structure which is not a tree.) Since a tree is much easier to represent and operate on than an arbitrary directed graph, this restriction may be a reasonable one.

We conjecture that binomial queues are uniquely optimal (in the sense of providing the tightest upper bound on $Q(m)$) among all priority queue algorithms which make only tree comparisons. The following result is a weakened form of this conjecture.

Theorem 2. The only priority queue which makes only tree comparisons and which makes a number of comparisons per cycle depending only on the queue size is the binomial queue.

Proof. We consider two algorithms to be the same if their directed graph structures are the same after each comparison. Suppose that an algorithm makes $k = k(m)$ comparisons per cycle on a queue of size m. At the start of each cycle the smallest element is known, so let the number of edges leaving this node at the start of the i-th cycle be d_i. Then during the i-th cycle the algorithm must determine which of d_i+1 nodes (the d_i which lost to the smallest, plus the inserted node) contains the smallest key. By the restriction to tree comparisons this takes exactly d_i comparisons; hence $d_i = k$ on every cycle. Thus it suffices to show that any tree comparison algorithm in which the smallest node (henceforth called the root) has a fixed number $k = k(m)$ of offspring for a queue of size m is the same as the binomial queue algorithm.

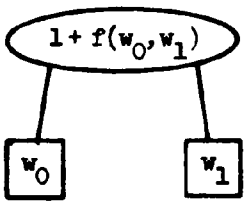
We are therefore led to consider adversaries which attempt to make the degree of the root fluctuate. It will be helpful to first consider two

simpler adversaries: one which attempts to maximize the degree of the root on a single cycle, and another which attempts to minimize it. Since max is an increasing function of its arguments, a maximizing adversary can do no better than to maximize the degree of the node which is smaller at each comparison; similarly, the minimizing adversary will minimize this quantity. If the two unbeaten nodes to be compared have degrees d_1 and d_2 then the maximum (minimum) resulting degree is $\max(d_1,d_2)+1$ ($\min(d_1,d_2)+1$) . What strategy can the algorithm use to minimize the degree of the root against a maximizing adversary, or maximize this quantity against a minimizing adversary?

It is useful to abstract this question into a problem on extended binary trees: we are given a vector $(w_0,w_1,\ldots,w_k)$ of $k+1$ real-valued weights, corresponding to the degrees of the $k+1$ nodes to be compared, and a function f which may be max or min . For any binary tree having $k+1$ external nodes we associate a real-valued <u>cost</u> with each node. The weights w_i are assigned as costs to the external nodes, and the costs of internal nodes are computed via the rule: if the two offspring of a node have costs u and v then the cost of the node is $1+f(u,v)$. Hence the cost of an internal node is just the degree of the node which wins the corresponding comparison under the adversaries considered above. We define the cost of the resulting binary tree to be the cost of the root, so our problem is to find a binary tree of minimum cost when $f = \max$, or maximum cost when $f = \min$. We shall call such trees <u>optimal</u>.

One method of constructing an appropriately labeled binary tree is to use Huffman's algorithm [16]. The first step in this procedure is to select the two smallest weights from the vector $(w_0,w_1,w_2,\ldots,w_k)$, say w_0

and w_1 . Then solve the problem for the k weights $(1 + f(w_0,w_1),w_2,\ldots,w_k)$. Finally, replace the external node containing $1 + f(w_0,w_1)$ with the binary tree



The tree which results from this procedure is called a Huffman tree.

Lemma 1. A Huffman tree is optimal when $f = \max$ and when $f = \min$.

Proof. (This proof is similar to the proof that Huffman trees have minimum weighted external path length, given in [26, p. 403]; a different proof for the case $f = \max$ is given in [14].) We argue by induction on k , the result being obvious when $k = 0$. It is sufficient to show that when $k > 0$, there is an optimal tree T in which the two smallest weights, say w_0 and w_1 , are contained in external nodes which are offspring of the same internal node. To see why this is enough, first note that by the induction hypothesis, the reduced problem of finding an optimal tree for the weights $(1 + f(w_0,w_1),w_2,\ldots,w_k)$ is solved by Huffman's algorithm. Call the Huffman tree for the reduced problem R ; then the cost of R is not worse than the cost of T because a solution to the reduced problem, which R solves optimally, is imbedded in T . But the tree R differs from the Huffman tree for the original problem only in the replacement of one external node, which does not change the cost of the root. Hence the Huffman tree for the original problem is optimal.

A second observation is that the cost of a tree is actually determined solely by the levels on which the weights $w_0, w_1, \ldots, w_k$ appear. The cost of the root is simply $f(l_0 + w_0, l_1 + w_1, \ldots, l_k + w_k)$ where l_i is the level on which weight w_i appears in an external node.

We shall prove that the two smallest weights, w_0 and w_1, may both appear on the deepest level in an optimal tree. Since there are at least two external nodes at this level and weights on the same level can be rearranged arbitrarily, this will give the result. Suppose that w_0 appears at level l_0 and that $w_j > w_0$ appears at level $l_j > l_0$. Consider the case $f = \max$; the effect of w_0 and w_j on the cost r of the root is to guarantee that $r \geq \max(l_0+w_0, l_j+w_j) = l_j+w_j$. If w_0 and w_j are exchanged, these nodes only force $r \geq \max(l_j+w_0, l_0+w_j) < l_j+w_j$, so the switch can only reduce r. In the case $f = \min$, we have $r \leq \min(l_0+w_0, l_j+w_j) = l_0+w_0$ before the switch, and $r \leq \min(l_j+w_0, l_0+w_j) < l_0+w_0$ after, so the exchange can only increase r. □

Using Lemma 1, we can construct the more complicated adversary needed to establish a strong restriction on the algorithms which make a fixed number of comparisons per cycle.

Lemma 2. A necessary condition for an algorithm using only tree comparisons to have a fixed degree k at the root at the start of each cycle is that the out-degrees of the offspring of the root be $k-1, k-2, \ldots, 1, 0$ at the start of each cycle.

Proof. Suppose that the degrees of the root's offspring, listed in decreasing order, are $d_{k-1}, d_{k-2}, \ldots, d_1, d_0$. We define an adversary which

causes the degree of the root to differ from k if the above degrees are not $k-1,k-2,\ldots,1,0$. The strategy depends on the first left-to-right discrepancy between the two sequences. Suppose ℓ is the largest index such that $d_\ell \neq \ell$; then there are two cases according to the relative sizes of d_ℓ and ℓ.

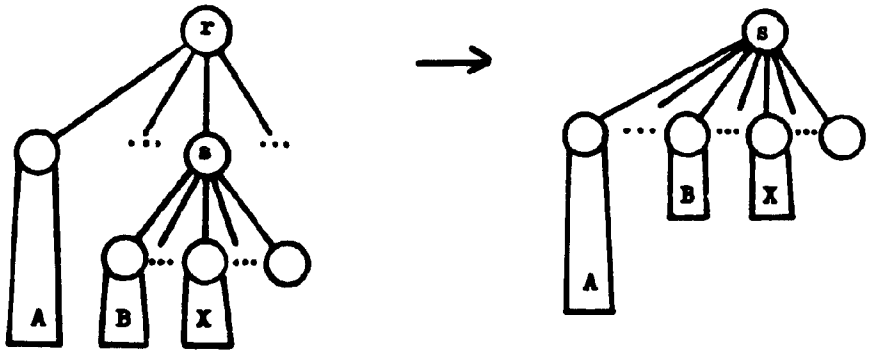
Case 1. $d_\ell > \ell$. The adversary attempts to maximize the degree of the root. Since offspring with degrees $k-1,k-2,\ldots,\ell+1$, and $d_\ell \geq \ell+1$ are present, along with the inserted node of degree 0, it is easy to see by Lemma 1 that even when the algorithm uses an optimal strategy against the maximizing adversary, the resulting degree is $\geq k+1$.

Case 2. $d_\ell < \ell$. The adversary attempts to minimize the degree of the root unless a premature comparison occurs; the i-th comparison is premature if it involves d_j where $j > \max(i-1,\ell)$. Informally, this means that to avoid a premature comparison, the algorithm must first perform a series of $\ell+1$ comparisons involving only the newly inserted element of degree 0 and the $\ell+1$ trees whose roots have degrees $d_0 \leq d_1 \leq \ldots \leq d_\ell < \ell$. This results in a single tree which is compared with the root of degree $d_{\ell+1} = \ell+1$; the result of this is then compared with the root of degree $d_{\ell+2} = \ell+2$, and so on until a single tree remains.

If no premature comparison occurs, then by Lemma 1 the tree which results from the first set of $\ell+1$ comparisons has degree $\leq \ell$, so the final result has degree $\leq k-1$. If a premature comparison occurs then the adversary changes to a maximizing strategy on that comparison and for all those which follow. An argument similar to the one used in Case 1 shows that the resulting degree is then $\geq k+1$. □

We can finally prove the theorem. First note that if the offspring of the root have out-degrees $k-1,k-2,\ldots,0$, then comparisons must proceed as in the binomial queue algorithm in order to guarantee that the root's degree remains at k. That is, the inserted node is compared to the degree 0 offspring, then this result is compared to the degree 1 offspring, and so on. This follows from Lemma 1 using either the maximizing or minimizing adversary.

Now order the offspring of each node in the directed graph structure of the queue by their out-degrees, decreasing from left to right. Find a structure arising during the operation of the given priority queue algorithm which differs from the binomial queue structure at some point, such that this discrepancy is as shallow (close to the root) as possible. The discrepancy can't occur at the root, and it can't occur at an offspring of the root by Lemma 2. If it occurs deeper in the tree, then one cycle of the queue can move it toward the root as shown (X is the subtree containing the mismatch);



This contradicts the assumption that the discrepancy was shallowest, and completes the proof. □

It seems quite unlikely that an exhaustive analysis of this kind can prove the conjecture stated above.

3.5 Discussion.

Huffman trees were originally used to find an extended binary tree solving the problem

$$\min \sum_{1 \le i \le n} l_i \cdot w_i \quad .$$

The proof of Lemma 1 shows that Huffman's algorithm also finds trees which optimize

$$\min \max_{1 \le i \le n} (l_i + w_i)$$

and

$$\max \min_{1 \le i \le n} (l_i + w_i) \quad .$$

The first of these problems can be interpreted as a scheduling problem on n parallel processors: we have n jobs with running times $w_1, w_2, \ldots, w_n$ whose results must be combined pairwise, at unit cost, before a final answer is obtained. Huffman trees minimize the time to compute the final answer in this situation. Both problems can be interpreted in terms of a circuit-design problem: we have n devices with propagation delays $w_1, w_2, \ldots, w_n$ whose outputs must be combined pairwise, at unit delay, to give a single output. One Huffman tree minimizes the maximum propagation delay; the other maximizes the minimum delay. The latter property can be significant if the circuit is part of a pipeline.

There are as many possible settings for priority queue complexity as there are applications of priority queues. The one chosen here is simple, yet seems representative. An obvious generalization is to allow the queue size to fluctuate in a regular fashion between two widely-spaced values, such as m and $m/2$. In this situation our lower bound on $\bar{Q}(m)$

becomes $\lg m - \lg(e/2)$, and the upper bounds on $Q(m)$ and $\bar{Q}(m)$ given by sorted linear lists are again nearly tight. But the loser tree and binomial queue structures used to prove upper bounds in the simpler model do not apply in this situation, since they do not grow and shrink gracefully. It would be interesting to find a structure which is nearly optimal in the more general model and can actually be implemented to run in time proportional to the number of comparisons.

It is an open problem to improve upon the upper and lower bounds on $Q(m)$ or $\bar{Q}(m)$ when m is not a power of 2. It would also be interesting to prove results to the effect that arithmetic on keys cannot help, in the worst case or on the average, when the key space is large compared to the queue size; results of this kind have been shown for selection problems [43;13]. When the key space is restricted to the integers from 1 to m and arithmetic on keys is allowed, then the priority queue operations can be implemented to run in $O(\log \log m)$ time [40].

Our abstract study of priority queues has shown that binomial queues of size $m = 2^k$ are a particularly simple and efficient structure for implementing alternating insertions and deletions, such as occur in replacement selection. When m is not a power of 2, fewer than $\lg m$ dummy nodes must be added to make the queue behave as such, since the algorithms never look below a node containing an infinitely large key. Hence binomial queues may be a useful alternative to loser trees for replacement selection.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass. (1974).

[2] J. N. Buxton, ed., Simulation Programming Languages, North-Holland, Amsterdam (1968).

[3] David Cheriton and Robert Endre Tarjan, "Finding Minimum Spanning Trees," SIAM Journal on Computing 5, 4 (December 1976), 724-742.

[4] Clark Allan Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," Ph.D. Thesis, Computer Science Department, Stanford University, STAN-CS-72-259, February 1972.

[5] Ole-Johan Dahl and Kristen Nygaard, "SIMULA - An ALGOL-Based Simulation Language," C.ACM 9, 9 (1966), 671-678.

[6] Hubert Delange, "Sur la Fonction Sommatoire de la Fonction ⟨⟨Somme des Chiffres⟩⟩," L'Enseignement Math. 21, 1 (1975), 31-47.

[7] Persi Diaconis, "Examples in the Theory of Infinite Iteration of Summability Methods," Stanford University Department of Statistics Technical Report No. 86, May 1976.

[8] Michael J. Fischer, "Efficiency of Equivalence Algorithms," in Raymond E. Miller and James W. Thatcher, eds., Complexity of Computer Computations, Plenum Press, New York (1972).

[9] Robert W. Floyd, "Algorithm 245: Treesort 3," C.ACM 7, 12 (December 1964), 701.

[10] B. L. Fox, "Accelerating List Processing in Discrete Programming," J.ACM 17, 2 (April 1970), 383-384.

[11] Edward H. Friend, "Sorting on Electronic Computer Systems," J.ACM 3, (1956), 134-168.

[12] Frank Fussenegger and Harold N. Gabow, "Using Comparison Trees to Derive Lower Bounds on Selection Problems," Proc. 17th Annual Symp. on Foundations of Computer Science, Houston, Texas, 1976, 178-182.

[13] C. R. Glassey and R. M. Karp, "On the Optimality of Huffman Trees," SIAM J. Appl. Math. 31 (1976), 368-378.

[14] Martin C. Golumbic, "Combinatorial Merging," IEEE Transactions on Computers, C-25 (November 1976), 1164-1167.

[15] G. Gordon, "A General-Purpose Systems Simulator," IBM Systems Journal 1, September 1962, 18-32.

[16] David A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE 40 (1951), 1098-1101.

[17] Kenneth E. Iverson, A Programming Language, John Wiley, New York (1962), 223-227.

[18] Ellis L. Johnson, "On Shortest Paths and Sorting," Proc. 25th Annual Conference of the ACM, 1972, 510-517.

[19] Donald B. Johnson, "Priority Queues with Update and Finding Minimum Spanning Trees," Information Processing Letters 4, 3 (December 1975), 53-57.

[20] Donald B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," J.ACM 24, 1 (Janaury 1977), 1-13.

[21] Arne Jonassen and Ole-Johan Dahl, "Analysis of an Algorithm for Priority Queue Administration," BIT 15 (1975), 409-422.

[22] Arne Jonassen and Ole-Johan Dahl, "Analysis of an Algorithm for Priority Queue Administration," Math. Inst., Univ. of Oslo (1975).

[23] Arne T. Jonassen and Donald E. Knuth, "A Trivial Algorithm Whose Analysis Isn't," submitted for publication.

[24] A. Kerschenbaum and R. Van Slyke, "Computing Minimum Spanning Trees Efficiently," Proc. 25th Annual Conference of the ACM, 1972, 518-527.

[25] Gary D. Knott, "Deletion in Binary Storage Trees," Ph.D. Thesis, Computer Science Department, Stanford University, STAN-CS-75-491, May 1975, 93 pp.

[26] Donald E. Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1973).

[27] Donald E. Knuth, The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, Mass. (1971).

[28] Donald E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass. (1973).

[29] Donald E. Knuth, "Deletions that Preserve Randomness," Computer Science Department, Stanford University, STAN-CS-76-584, December 1976.

[30] Donald E. Knuth and John L. McNeley, "SOL - A Symbolic Language for General Purpose Systems Simulation," IEEE Transactions on Computers EC-13, 4 (1964), 401-408.

[31] A. N. Kolmogorov and V. A. Uspenskiĭ, "K opredeleniio algoritma," [On the Definition of Algorithms], Uspekhi Mat. Nauk. 13, 4 (1958), 3-28. English Translation in Amer. Math. Soc. Transl. II Vol. 29 (1963), 217-245.

[32] Derrick H. Lehmer, "The Machine Tools of Combinatorics," in Edwin F. Bechenbach, ed., Applied Combinatorial Mathematics, John Wiley, New York (1964).

[33] M. A. Malcolm and R. B. Simpson, "Local Versus Global Strategies for Adaptive Quadrature," ACM Transactions in Math. Software, 1, 2 (June 1975), 130-146.

[34] H. Markowitz, B. Hausner, and H. Karr, SIMSCRIPT, A Simulation Programming Language, Prentice-Hall, Englewood Cliffs, N. J. (1963).

[35] M. D. McIlroy, "The Number of 1's in Binary Integers: Bounds and Extremal Properties," SIAM Journal on Computing 3, 4 (December 1974), 255-261.

[36] Eugenio Morreale, "Computational Complexity of Partitioned List Algorithms," IEEE Transactions C-19 (May 1970), 421-428.

[37] Thomas Porter and Istvan Simon, "Random Insertion into a Priority Queue Structure," IEEE Transactions SE-1 (September 1975), 292-298.

[38] Ralph A. Raimi, "The First Digit Problem," Amer. Math. Monthly 83, 7 (1976), 521-538.

[39] Lyle Ramshaw, personal communication.

[40] P. van Emde Boas, "Preserving Order in a Forest in Less Than Logarithmic Time," Proc. 16th Annual Symp. on Foundations of Computer Science, Berkeley, Calif. 1975, 75-84.

[41] Jean G. Vaucher and Pierre Duval, "A Comparison of Simulation Event List Algorithms," C.ACM 18 (1975), 223-230.

[42] Jean Vuillemin, "A Data Structure for Manipulating Priority Queues," C.ACM (to appear).

[43] Andrew C.-C. Yao, "On the Complexity of Comparison Problems Using Linear Functions," Proc. 16th Annual Symp. on Foundations of Computer Science, Berkeley, Calif., 1975, 85-89.

Appendix. Priority Queue Implementations

1. SAIL Implementations.

1.1 Binomial Queue using Structure R.

```
COMMENT Priority queue routines using structure R for binomial trees.

Notes on the subset of SAIL used here:
  1)  '<>' is the exchange operator.
  2)  'DONE "bn"' causes the loop on block named "bn" to be exited.
  3)  RECORD_POINTER parameters are passed by value.

About programming style:
  These routines are not intended to be an easy intoduction to binomial queues.
  Instead they are meant to be a guide to efficient implementation of binomial
  queue operations, as might be accomplished in assembly language.  This means
  that in these procedures a large amount of state is kept implicitly in the
  flow of control, rather than in program variables.  We have not attempted
  to perform other optimizations, such as the assignment of registers, since
  these can be performed without a global understanding of the algorithms.

About mnemonics:
  Identifier names are intended to convey meaning when possible, but have
  also been kept reasonably short.  Names are generally a concatenation of
  short tags which are abbreviations of something meaningful: Rt for root,
  Nxt for next, etc.  Capitalization is used to delimit the tags, so "the
  new rightmost root" is written newRtmRt.  Happy reading!;


BEGIN "BinomialQueue"
REQUIRE "{}<>" DELIMITERS;

RECORD_CLASS Node
  (RECORD_POINTER(Node) !Sibling!, !Child!; INTEGER Key!);

COMMENT  Abbreviations for Node fields:;
DEFINE !Sibling = !Node:!Sibling!!;
DEFINE !Child = !Node:!Child!!;
DEFINE Key = !Node:Key!!;

RECORD_CLASS QueueHeader (RECORD_POINTER(Node) leftmostRoot;  INTEGER Size);

BOOLEAN PROCEDURE ODD(INTEGER i);
  RETURN(i LAND 1);
```

```
PROCEDURE Insert (RECORD_POINTER(Node) x;  RECORD_POINTER(QueueHeader) Q);
  BEGIN  "Insert"
  RECORD_POINTER(Node) rtmRt, nxtRt;
  INTEGER s;
  s ← QueueHeader:Size[Q];
  IF s ≠ 0 THEN  rtmRt ← lSibling[QueueHeader:leftmostRoot[Q]];

  lChild[x] ← NULL_RECORD;
  IF ODD(s) THEN BEGIN
    "The rightmost tree in q consists of a single node; merge it into x. (The merge
     of H0's is treated specially to eliminate a test from the inner loop.)"
    nxtRt ← lSibling[rtmRt];
    IF Key[x] > Key[rtmRt] THEN x ↔ rtmRt;
    lChild[x] ← rtmRt;
    lSibling[rtmRt] ← rtmRt;
    WHILE TRUE DO BEGIN "MergeLoop"
      rtmRt ← nxtRt;  s ← s/2;
      IF ¬ODD(s) THEN DONE "MergeLoop";
      "The rightmost tree remaining is the same size as x; merge it into x."
      nxtRt ← lSibling[rtmRt];
      IF Key[x] > Key[rtmRt] THEN x ↔ rtmRt;
      lSibling[rtmRt] ← lSibling[lChild[x]];
      lSibling[lChild[x]] ← rtmRt;
      lChild[x] ← rtmRt
    END "MergeLoop"
  END;
  IF s = 0 THEN BEGIN
    "The entire forest has been merged into x. (The forest size is a power of two.)"
    lSibling[x] ← x;
    QueueHeader:leftmostRoot[Q] ← x
  END
  ELSE BEGIN
    "Some of the original forest remains; x is the rightmost root in the new forest."
    lSibling[QueueHeader:leftmostRoot[Q]] ← x;
    lSibling[x] ← rtmRt
  END;
  QueueHeader:Size[Q] ← QueueHeader:Size[Q] + 1;
END "Insert";
```

```
RECORD_POINTER(Node) PROCEDURE DeleteSmallest (RECORD_POINTER(QueueHeader) Q);
  BEGIN "DeleteSmallest"
  RECORD_POINTER(Node) lfmRt, rtmRt, smallest, pred, succ, smallestPred, rtmChild,
    newRtmRt, mrgTree, nxtTree;
  INTEGER s, smallestKey;
  lfmRt ← QueueHeader:leftmostRoot[Q];  rtmRt ← lSibling[lfmRt];
  s ← QueueHeader:Size[Q];

  IF lfmRt = rtmRt THEN BEGIN "There is only one tree in the forest."
    "Return the root, and make the new forest from its sons."
    smallest ← lfmRt;
    QueueHeader:leftmostRoot[Q] ← lChild[lfmRt]
  END "There is only one tree in the forest."
  ELSE BEGIN "There are two or more trees in the forest."

    "Search for the node containing the smallest key in the queue."

    lSibling[lfmRt] ← NULL_RECORD; "Mark the leftmost node to stop search."
    smallestKey ← Key[lfmRt];  pred ← lfmRt;  succ ← rtmRt;
    DO BEGIN
      IF smallestKey ≥ Key[succ] THEN BEGIN
        smallestKey ← Key[succ];  smallestPred ← pred  END;
      pred ← succ;  succ ← lSibling[succ]
    END UNTIL succ = NULL_RECORD;
    smallest ← lSibling[smallestPred];
    IF smallest = NULL_RECORD THEN BEGIN "The rightmost root is smallest."
      smallest ← rtmRt;
      IF ODD(s) THEN
        "The rightmost tree is a sing  node; just remove it from the forest."
        lSibling[lfmRt] ← lSibling[smallest]
      ELSE BEGIN
        "The sons of the rightmost root become the smallest trees in the new forest."
        lSibling[lfmRt] ← lSibling[lChild[smallest]];
        lSibling[lChild[smallest]] ← lSibling[smallest]
      END
    END "The rightmost root is smallest."
    ELSE BEGIN "A root other than the rightmost is smallest."
      "The tree containing this root must be replaced. A replacement tree is formed
       by merging the rightmost tree in the forest with the children of the removed
       root.  Children which are smaller in size than the rightmost tree become the
       smallest trees in the new forest."
      rtmChild ← lSibling[lChild[smallest]];
      lSibling[lChild[smallest]] ← NULL_RECORD; "Mark leftmost child to stop scan."
      IF ~ODD(s) THEN BEGIN
        "The queue size was even before the deletion, so some children of the
         removed root will move up to become the smallest trees in the new forest.
         Scan through the children until mrgTree, the one which will merge with
         the rightmost tree in the forest, is reached."
        newRtmRt ← rtmChild;
        s ← s/2;
        WHILE ~ODD(s) DO BEGIN
          rtmChild ← lSibling[rtmChild];  s ← s/2  END;
        mrgTree ← lSibling[rtmChild];
        IF smallestPred = rtmRt THEN
          "The tree to the right of the removed root is the rightmost tree, and thus
           will be consumed in building the replacement tree.  So the replacement's
           predecessor will be the leftmost child which moves up."
          smallestPred ← rtmChild
        ELSE
          "Link the children into the right of the forest now, since their lSibling
           is not the replacement and is therefore known."
          lSibling[rtmChild] ← lSibling[rtmRt]
```

```
      END
      ELSE BEGIN
        "The queue size is odd, so all children of the removed root will be used
         to make the replacement."
        newRtmRt ← lSibling[rtmRt];
        IF smallestPred = rtmRt THEN
          "The replacement tree will be the rightmost tree in the new forest; hence
           newRtmRt and smallestPred cannot be given true values now.  Flag
           this with smallestPred = NULL_RECORD."
          smallestPred ← NULL_RECORD;
        "Perform the first merge.  The merge of 80's is handled specially in order
         remove a test from the inner loop."
        mrgTree ← lSibling[rtmChild];
        IF Key[rtmRt] > Key[rtmChild] THEN  rtmRt ↔ rtmChild;
        lChild[rtmRt] ← rtmChild;
        lSibling[rtmChild] ← rtmChild
      END;
      "Complete the merge."
      WHILE mrgTree ≠ NULL_RECORD DO BEGIN
        nxtTree ← lSibling[mrgTree];
        IF Key[rtmRt] > Key[mrgTree] THEN  rtmRt ↔ mrgTree;
        lSibling[mrgTree] ← lSibling[lChild[rtmRt]];
        lSibling[lChild[rtmRt]] ← mrgTree;
        lChild[rtmRt] ← mrgTree;
        mrgTree ← nxtTree;
      END;
      "It remains to fix up some links between roots in the forest: 1) lSibling link
       from leftmost root to rightmost root, 2) lSibling link from replacement tree
       to the next larger tree, and 3) lSibling link to replacement tree from the
       next smaller tree.  Many forms of degeneracy are possible..."
      IF smallest = lfmRt THEN BEGIN
        "The leftmost tree was replaced, so link from header must be fixed."
        QueueHeader:leftmostRoot[0] ← rtmRt;
        IF smallestPred = NULL_RECORD THEN
          "The replacement tree is the only tree in the forest: 1), 2), and 3) are
           identical."
          lSibling[rtmRt] ← rtmRt
        ELSE BEGIN
          "There is another tree in the forest, besides the replacement: 1) and 2)
           are identical."
          lSibling[rtmRt] ← newRtmRt;
          lSibling[smallestPred] ← rtmRt
        END
      END
      ELSE BEGIN
        "There is a tree larger than the replacement, so 2) can be filled in now."
        lSibling[rtmRt] ← lSibling[smallest];
        IF smallestPred = NULL_RECORD THEN
          "There is no tree smaller than the replacement: 1) and 3) are identical."
          lSibling[lfmRt] ← rtmRt
        ELSE BEGIN
          "The nondegenerate case: 1), 2), and 3) are distinct."
          lSibling[lfmRt] ← newRtmRt;
          lSibling[smallestPred] ← rtmRt
        END
      END
    END "A root other than the rightmost is smallest."
  END "There are two or more trees in the forest.";
  QueueHeader:Size[0] ← QueueHeader:Size[0] - 1;
  lSibling[smallest] ← lChild[smallest] ← NULL_RECORD;
  RETURN(smallest)
END "DeleteSmallest";
```

```
PROCEDURE Union (RECORD_POINTER(QueueHeader) T, Q);
BEGIN "Union"
  RECORD_POINTER(Node) ARRAY Bk[1:3];
  INTEGER i;
  COMMENT  Bk is a stack of Bk trees which is accumulated for each stage of
    the "addition".  "Carries" are propogated through Bk[1].  The integer i
    is the stack pointer, i.e., it is the number of trees in the stack.;
  RECORD_POINTER(Node) rT, rQ, rF, dummy;
  COMMENT  rF points to the largest tree in the result forest which has been
    generated.;
  INTEGER sT, sQ;
  sT ← QueueHeader:Size[T];  IF sT≠0 THEN rT ← lSibling[QueueHeader:leftmostRoot[T]];
  sQ ← QueueHeader:Size[Q];  IF sQ≠0 THEN rQ ← lSibling[QueueHeader:leftmostRoot[Q]];
  dummy ← NEW_RECORD(Node);  rF ← dummy;
  i ← 0;

  "The binary addition algorithm."

  "The B0 trees are handled specially to remove tests from the inner loop."
  IF ODD(sT) THEN BEGIN
    i ← i+1;  Bk[i] ← rT;  rT ← lSibling[rT]  END;
  IF ODD(sQ) THEN BEGIN
    i ← i+1;  Bk[i] ← rQ;  rQ ← lSibling[rQ]  END;
  IF i = 1 THEN BEGIN
    lSibling[rF] ← Bk[1];  rF ← Bk[1];  i ← 0  END;
  IF i = 2 THEN BEGIN
    IF Key[Bk[1]] > Key[Bk[2]] THEN Bk[1] ↔ Bk[2];
    lChild[Bk[1]] ← Bk[2];
    lSibling[Bk[2]] ← Bk[2];
    i ← 1
  END;
  sT ← sT/2;  sQ ← sQ/2;
  "The general step."
  WHILE (sT ≠ 0) ∨ (sQ ≠ 0) DO BEGIN
    IF ODD(sT) THEN BEGIN
      i ← i+1;  Bk[i] ← rT;  rT ← lSibling[rT]  END;
    IF ODD(sQ) THEN BEGIN
      i ← i+1;  Bk[i] ← rQ;  rQ ← lSibling[rQ]  END;
    IF (i = 1) ∨ (i = 3) THEN BEGIN
      lSibling[rF] ← Bk[i];  rF ← Bk[i];  i ← i-1  END;
    IF i = 2 THEN BEGIN
      IF Key[Bk[1]] > Key[Bk[2]] THEN Bk[1] ↔ Bk[2];
      lSibling[Bk[2]] ← lSibling[lChild[Bk[1]]];
      lSibling[lChild[Bk[1]]] ← Bk[2];
      lChild[Bk[1]] ← Bk[2];
      i ← 1
    END;
    sT ← sT/2;  sQ ← sQ/2
  END;
  "Handle a carry off the end, if present."
  IF i = 1 THEN BEGIN
    lSibling[rF] ← Bk[1];  rF ← Bk[1]  END;

  "Link the result into Q, and clear out T."

  lSibling[rF] ← lSibling[dummy];
  "At this point the dummy node can be explicitly deallocated to save GC's."
  QueueHeader:leftmostRoot[Q] ← rF;
  QueueHeader:Size[Q] ← QueueHeader:Size[T] + QueueHeader:Size[Q];
  QueueHeader:leftmostRoot[T] ← NULL_RECORD;
  QueueHeader:Size[T] ← 0;
END "Union";
```

1.2 Binomial Queue using Structure K.

```
COMMENT Priority queue routines using structure K for binomial trees.

Notes on the subset of SAIL used here:
  1)  '<->' is the exchange operator.
  2)  'DONE "bn"' causes the loop on block named "bn" to be exited.
  3)  RECORD_POINTER parameters are passed by value.

About programming style:
  These routines are not intended to be an easy intoduction to binomial queues.
  Instead they are meant to be a guide to efficient implementation of binomial
  queue operations, as might be accomplished in assembly language.  This means
  that in these procedures a large amount of state is kept implicitly in the
  flow of control, rather than in program variables.  We have not attempted
  to perform other optimizations, such as the assignment of registers, since
  these can be performed without a global understanding of the algorithms.

About mnemonics:
  Identifier names are intended to convey meaning when possible, but have
  also been kept reasonably short.  Names are generally a concatenation of
  short tags which are abbreviations of something meaningful: Rt for root,
  Nxt for next, etc.  Capitalization is used to delimit the tags, so "the
  new rightmost root" is written newRtmRt.  Happy reading!;


BEGIN "BinomialQueue"
REQUIRE "(!<>" DELIMITERS;

RECORD_CLASS Node
  (RECORD_POINTER(ANY_CLASS) !Sibling!; RECORD_POINTER(Node) !Child!; INTEGER Key!);

COMMENT The !Sibling! field is declared ANY_CLASS because it must refer to records
  of class QueueHeader as well as class Node.  SAIL does not handle forward
  RECORD_CLASS declarations correctly.;

COMMENT  Abbreviations for Node fields:;
DEFINE !Sibling = !Node:!Sibling!!;
DEFINE !Child = !Node:!Child!!;
DEFINE Key = !Node:Key!!;

RECORD_CLASS QueueHeader
  (RECORD_POINTER(Node) !Sibling!, !Child!; INTEGER Size);

COMMENT  It is essential that !Sibling! have the same offset in both Node and
QueueHeader, since the condition !Sibling! = NULL_RECORD is what distinguishes
header nodes from others.  In some programming languages (e.g. Simula) there are
facilities for declaring such restrictions explicitly.;

BOOLEAN PROCEDURE ODD(INTEGER i);
  RETURN(i LAND 1);
```

```
PROCEDURE Insert (RECORD_POINTER(Node) x;  RECORD_POINTER(QueueHeader) Q);
BEGIN  "Insert"
  RECORD_POINTER(Node) rtmRt, nxtRt;
  INTEGER s;
  s ← QueueHeader:Size[Q];
  rtmRt ← QueueHeader:lChild![Q];

  lChild[x] ← NULL_RECORD;
  IF ODD(s) THEN BEGIN
    "The rightmost tree in Q consists of a single node; merge it into x. (The merge
     of BA's is treated specially to eliminate a test from the inner loop.)"
    nxtRt ← lSibling[rtmRt];
    IF Key[x] > Key[rtmRt] THEN x ↔ rtmRt;
    lChild[x] ← rtmRt;
    WHILE TRUE DO BEGIN "MergeLoop"
      rtmRt ← nxtRt;  s ← s/2;
      IF ~ODD(s) THEN DONE "MergeLoop";
      "The rightmost tree remaining is the same size as x; merge it into x."
      nxtRt ← lSibling[rtmRt];
      IF Key[x] > Key[rtmRt] THEN x ↔ rtmRt;
      lSibling[lChild[rtmRt]] ← lChild[x];
      lSibling[lChild[x]] ← rtmRt;
      lChild[x] ← rtmRt
    END "MergeLoop";
    lSibling[lChild[x]] ← Q
  END;
  QueueHeader:lChild![Q] ← x;
  IF s = 0 THEN
    "The entire forest has been merged into x. (The forest size is a power of two.)"
    lSibling[x] ← Q
  ELSE BEGIN
    "Some of the original forest remains; x is the rightmost root in the new forest."
    lSibling[x] ← rtmRt;
    lSibling[lChild[rtmRt]] ← x
  END;
  QueueHeader:Size[Q] ← QueueHeader:Size[Q] + 1;
END "Insert";
```

```
RECORD_POINTER(Node) PROCEDURE DeleteSmallest (RECORD_POINTER(QueueHeader) Q);
BEGIN "DeleteSmallest"
  RECORD_POINTER(Node) rtmRt, smallest, p, rtmChild, rightRtT, newRtmRt, mrgTree,
    nxtTree;
  INTEGER s, smallestKey;
  rtmRt ← QueueHeader:lChild![Q];  s ← QueueHeader:Size[Q];

  "Search for the node containing the smallest key in the queue."

  smallest ← rtmRt;  smallestKey ← Key[smallest];
  p ← lSibling[smallest];
  WHILE lSibling[p] ≠ NULL_RECORD DO BEGIN
    IF smallestKey > Key[p] THEN BEGIN
      smallest ← p;  smallestKey ← Key[smallest] END;
    p ← lSibling[p]
  END;

  "Merge the offspring of the smallest node with the rightmost tree in the
   forest, unless the rightmost tree contained the smallest."

  rtmChild ← lChild[smallest];
  IF rtmChild = NULL_RECORD THEN BEGIN "smallest is a B0."
    "The deletion can be completed now."
    IF s = 1 THEN
      QueueHeader:lChild![Q] ← NULL_RECORD
    ELSE BEGIN
      QueueHeader:lChild![Q] ← lSibling[smallest];
      lSibling[lChild[lSibling[smallest]]] ← Q
    END
  END   "smallest is a B0."
  ELSE BEGIN "smallest is not a B0."
    "Find the rightmost child of the smallest node."
    WHILE lChild[rtmChild] ≠ NULL_RECORD DO
      rtmChild ← lSibling[lChild[rtmChild]];
    IF smallest = rtmRt THEN BEGIN "smallest is the rightmost root."
      "The deletion can be completed now."
      QueueHeader:lChild![Q] ← rtmChild;
      lSibling[lChild[smallest]] ← lSibling[smallest];
      IF lSibling[smallest] ≠ 0 THEN
        lSibling[lChild[lSibling[smallest]]] ← lChild[smallest]
    END   "smallest is the rightmost root."
    ELSE BEGIN "smallest is not the rightmost root."

      "Set up for the merge."

      rightRtT ← lSibling[lChild[smallest]];
      lSibling[lChild[smallest]] ← NULL_RECORD; "Mark leftmost child."
      IF ~ODD(s) THEN BEGIN
        "The rightmost tree is not a B0, so some children of smallest will
         be the smallest trees in the new forest."
        newRtmRt ← rtmChild;
        s ← s/2;
        WHILE ~ODD(s) DO BEGIN
          rtmChild ← lSibling[rtmChild];  s ← s/2  END;
        mrgTree ← lSibling[rtmChild];
        IF rightRtT = rtmRt THEN
          rightRtT ← rtmChild
        ELSE BEGIN
          lSibling[rtmChild] ← lSibling[rtmRt];
          lSibling[lChild[lSibling[rtmRt]]] ← rtmChild
        END
      END
```

```
      ELSE BEGIN
        "The rightmost tree is a B0, so it combines with all of the children
         of smallest to form the replacement tree."
        IF rightRtT = rtmRt THEN
          rightRtT ← NULL_RECORD
        ELSE BEGIN
          newRtmRt ← lSibling[rtmRt];
          lSibling[lChild[newRtmRt]] ← 0
        END;
        "Perform the first merge.  The merge of B0's is handled specially in order
         eliminate a test from the inner loop."
        mrgTree ← lSibling[rtmChild];
        IF Key[rtmRt] > Key[rtmChild] THEN  rtmRt ↔ rtmChild;
        lChild[rtmRt] ← rtmChild;
      END;

      "Complete the merge."

      WHILE mrgTree ≠ NULL_RECORD DO BEGIN
        nxtTree ← lSibling[mrgTree];
        IF Key[rtmRt] > Key[mrgTree] THEN  rtmRt ↔ mrgTree;
        lSibling[lChild[mrgTree]] ← lChild[rtmRt];
        lSibling[lChild[rtmRt]] ← mrgTree;
        lChild[rtmRt] ← mrgTree;
        mrgTree ← nxtTree;
      END;

      "Link the tree created by the merge into the forest."

      lSibling[rtmRt] ← lSibling[smallest];
      IF lSibling[smallest] ≠ 0 THEN
        lSibling[lChild[lSibling[smallest]]] ← rtmRt;
      IF rightRtT = NULL_RECORD THEN BEGIN
        QueueHeader:lChild![0] ← rtmRt;
        lSibling[lChild[rtmRt]] ← 0;
      END
      ELSE BEGIN
        QueueHeader:lChild![0] ← newRtmRt;
        lSibling[rightRtT] ← rtmRt;
        lSibling[lChild[rtmRt]] ← rightRtT
      END
    END   "smallest is not the rightmost root."
  END   "smallest is not a B0.";
  QueueHeader:Size[0] ← QueueHeader:Size[0] - 1;
  lSibling[smallest] ← lChild[smallest] ← NULL_RECORD;
  RETURN(smallest)
END "DeleteSmallest";
```

```
RECORD_POINTER(QueueHeader) PROCEDURE Delete (RECORD_POINTER(Node) x);
BEGIN "Delete"
  RECORD_POINTER(QueueHeader) Q;
  RECORD_POINTER(Node) p, rtmRt, rightRtT, leftRtT, pN, Fr, Tr, newRtmRt, mrgTree,
    nxtTree;
  COMMENT Here T denotes the tree containing x, and Tr the tree which replaces T;
  RECORD_POINTER(Node) ARRAY path[1:18];  COMMENT 18 is lg(max queue size) + 1;
  INTEGER i, s, iSave;
  LABEL MergeForests, DeleteReturn;

  "Climb out of the binomial queue to reach the queue header; save the trail
   of nodes visited in the 'path' array."

  p ← x;  i ← 0;
  WHILE p ≠ NULL_RECORD DO BEGIN
    i ← i+1;  path[i] ← p;  p ← lSibling[p]  END;
  Q ← path[i];  i ← i-1;
  s ← QueueHeader:Size[Q];  rtmRt ← QueueHeader:lChild![Q];

  "How did we get to the queue header?  There are two possibilities: either
   from the lChild of the rightmost root, or from the leftmost root."

  IF path[i] = lChild[rtmRt] THEN BEGIN
    "x is a non-root node in the rightmost tree."
    rightRtT ← NULL_RECORD;
    leftRtT ← lSibling[rtmRt];
    pN ← rtmRt
  END
  ELSE BEGIN
    "Either x is the root of the rightmost tree, or it is in some tree other than
     the rightmost.  Locate the root of the tree containing x."
    WHILE TRUE DO BEGIN "RootSearch"
      IF i-2 ≤ 0 THEN BEGIN "x is a root."
        "The dismantling of the tree containing x can be completed now."
        leftRtT ← lSibling[x];
        IF lChild[x] = NULL_RECORD THEN BEGIN "x is a B0."
          "The deletion can be completed now."
          IF s = 1 THEN
            QueueHeader:lChild![Q] ← NULL_RECORD
          ELSE BEGIN
            QueueHeader:lChild![Q] ← leftRtT;
            lSibling[lChild[leftRtT]] ← Q
          END;
          GOTO DeleteReturn
        END "x is a B0.";
        rightRtT ← lSibling[lChild[x]];  IF rightRtT = Q THEN rightRtT ← NULL_RECORD;
        Fr ← lChild[x];  lSibling[Fr] ← NULL_RECORD;
        WHILE lChild[Fr] ≠ NULL_RECORD DO Fr ← lSibling[lChild[Fr]];
        GOTO MergeForests
      END "x is a root.";
      IF lChild[path[i]] = path[i-2] THEN DONE "RootSearch";
      i ← i-1
    END "RootSearch";
    "x is a non-root node in some tree other than the rightmost."
    pN ← path[i];
    rightRtT ← path[i-1];  leftRtT ← lSibling[pN];  i ← i-2
  END;

  "We are here to dismantle the tree containing the node x;  x is not the
   root of this tree, since that special case was handled above.

   The dismantling proceeds top down; large trees are generated before small
```

```
    ones.  At each step, there is a forest of trees already saved, linked from
    smaller to larger, in Fr.  There is also a node on the 'true' path from x to
    the root of the tree containing x, in pN.  Each step begins by finding the
    path node on the level on the next lower level.  Any trees to the left of
    this lower path node are added to the forest; then a tree formed from pN
    and the trees to the right of the lower path node is added.  Then the lower
    path node becomes pN, and the process repeats on the new level until the
    level of node x is reached.

    When the dismantling begins, pN is the root of the tree containing x
    and i is an index in path such that lChild[pN] = path[i]."

  Fr ← NULL_RECORD;
  WHILE TRUE DO BEGIN "DownLoop"
    iSave ← i;
    WHILE TRUE DO BEGIN "RightLoop"
      IF i-2 ≤ 0 THEN DONE "DownLoop";
      IF lChild[path[i]] = path[i-2] THEN DONE "RightLoop";
      i ← i-1
    END "RightLoop";
    IF iSave ≠ i THEN BEGIN
      rSibling[path[iSave]] ← Fr;  Fr ← path[i+1]  END;
    lChild[pN] ← path[i-1];  lSibling[pN] ← Fr;  Fr ← pN;
    pN ← path[i];  i ← i-2
  END "DownLoop";
  IF iSave ≠ 1 THEN BEGIN
    rSibling[path[iSave]] ← Fr;  Fr ← path[2]  END;
  lSibling[pN] ← Fr;  Fr ← pN;
  IF lChild[x] = NULL_RECORD THEN lChild[pN] ← NULL_RECORD
  ELSE BEGIN
    lChild[pN] ← lSibling[lChild[x]];
    lSibling[lChild[x]] ← Fr;  Fr ← lChild[x];
    WHILE lChild[Fr] ≠ NULL_RECORD DO Fr ← lSibling[lChild[Fr]]
  END;

MergeForests:

  "The variables passed from above are rtmRt, rightRtT, leftRtT, Fr, Q, and s."

  IF rightRtT = NULL_RECORD THEN BEGIN "the rightmost tree was dismantled."
    "The deletion can be completed now."
    QueueHeader:lChild!(Q) ← Fr;
    WHILE lSibling[Fr] ≠ NULL_RECORD DO BEGIN
      lSibling[lChild[lSibling[Fr]]] ← Fr;  Fr ← lSibling[Fr]  END;
    lSibling[Fr] ← leftRtT;  IF leftRtT ≠ Q THEN  lSibling[lChild[leftRtT]] ← Fr
  END "the rightmost tree was dismantled."
  ELSE BEGIN "a non-rightmost tree was dismantled."

    "Set up to merge Fr with the rightmost tree."

    Tr ← rtmRt;
    IF ~ODD(s) THEN BEGIN
      "The queue size is even before the deletion, so some trees in Tr'
       will move up to become the smallest trees in the new forest."
      newRtmRt ← Fr;
      s ← s/2;
      WHILE ~ODD(s) DO BEGIN
        lSibling[lChild[lSibling[Fr]]] ← Fr;  Fr ← lSibling[Fr];
        s ← s/2  END;
      mrgTree ← lSibling[Fr];
      IF rightRtT = rtmRt THEN
        rightRtT ← Fr
```

```
      ELSE BEGIN
        ISibling[Fr] ← ISibling[rtmRt];
        ISibling[IChild[ISibling[rtmRt]]] ← Fr
      END
    END
    ELSE BEGIN
      "The queue size is odd, so all children of the removed root will be used
       to make the replacement."
      IF rightRtT = rtmRt THEN
        rightRtT ← NULL_RECORD
      ELSE BEGIN
        newRtmRt ← ISibling[rtmRt];
        ISibling[IChild[newRtmRt]] ← Q
      END;
      "Perform the first merge.  The merge of B0's is handled specially in order
       speed up the inner loop."
      mrgTree ← ISibling[Fr];
      IF Key[Tr] > Key[Fr] THEN
        Tr ← Fr;
      IChild[Tr] ← Fr;
    END;

    "Complete the merge."

    WHILE mrgTree ≠ NULL_RECORD DO BEGIN
      nxtTree ← ISibling[mrgTree];
      IF Key[Tr] > Key[mrgTree] THEN  Tr ← mrgTree;
      ISibling[IChild[mrgTree]] ← IChild[Tr];
      ISibling[IChild[Tr]] ← mrgTree;
      IChild[Tr] ← mrgTree;
      mrgTree ← nxtTree;
    END;

    "The merge is complete;  link the new tree Tr into the forest."

    ISibling[Tr] ← leftRtT;  IF leftRtT ≠ Q THEN  ISibling[IChild[leftRtT]] ← Tr;
    IF rightRtT = NULL_RECORD THEN BEGIN
      QueueHeader:IChild![Q] ← Tr;
      ISibling[IChild[Tr]] ← Q
    END
    ELSE BEGIN
      QueueHeader:IChild![Q] ← newRtmRt;
      ISibling[rightRtT] ← Tr;  ISibling[IChild[Tr]] ← rightRtT
    END
  END "a non-rightmost tree was dismantled.";

DeleteReturn:
  QueueHeader:Size[Q] ← QueueHeader:Size[Q] - 1;
  ISibling[x] ← IChild[x] ← NULL_RECORD;
  RETURN(Q)
END "Delete";
```

```
PROCEDURE Union (RECORD_POINTER(QueueHeader) T, Q);
BEGIN "Union"
  RECORD_POINTER(Node) ARRAY Bk[1:3];
  INTEGER i;
  COMMENT  Bk is a stack of Bk trees which is accumulated for each stage of
    the "addition".  "Carries" are propogated through Bk[1].  The integer i
    is the stack pointer, i.e., it is the number of trees in the stack.;
  RECORD_POINTER(Node) rT, rQ, rF, dummy;
  COMMENT  rF points to the largest tree in the result forest which has been
    generated.;
  INTEGER sT, sQ;
  sT ← QueueHeader:Size[T];  rT ← QueueHeader:Child![T];
  sQ ← QueueHeader:Size[Q];  rQ ← QueueHeader:Child![Q];
  dummy ← NEW_RECORD(Node);
  rF ← dummy;
  i ← 0;

  "The binary addition algorithm."

  "The Bk trees are handled specially to remove tests from the inner loop."
  IF ODD(sT) THEN BEGIN
    i ← i+1;  Bk[i] ← rT;  rT ← lSibling[rT]  END;
  IF ODD(sQ) THEN BEGIN
    i ← i+1;  Bk[i] ← rQ;  rQ ← lSibling[rQ]  END;
  IF i = 1 THEN BEGIN
    lSibling[rF] ← Bk[1];  rF ← Bk[1];  i ← 0  END;
  IF i = 2 THEN BEGIN
    IF Key[Bk[1]] > Key[Bk[2]] THEN Bk[1] ↔ Bk[2];
    lChild[Bk[1]] ← Bk[2];
    i ← 1
  END;
  sT ← sT/2;  sQ ← sQ/2;
  "The general step."
  WHILE (sT ≠ 0) ∨ (sQ ≠ 0) DO BEGIN
    IF ODD(sT) THEN BEGIN
      i ← i+1;  Bk[i] ← rT;  rT ← lSibling[rT]  END;
    IF ODD(sQ) THEN BEGIN
      i ← i+1;  Bk[i] ← rQ;  rQ ← lSibling[rQ]  END;
    IF (i ≠ 1) ∨ (i = 3) THEN BEGIN
      lSibling[rF] ← Bk[i];  lSibling[lChild[Bk[i]]] ← rF;
      rF ← Bk[i];  i ← i-1
    END;
    IF i = 2 THEN BEGIN
      IF Key[Bk[1]] > Key[Bk[2]] THEN Bk[1] ↔ Bk[2];
      lSibling[lChild[Bk[2]]] ← lChild[Bk[1]];
      lSibling[lChild[Bk[1]]] ← Bk[2];
      lChild[Bk[1]] ← Bk[2];
      i ← 1
    END;
    sT ← sT/2;  sQ ← sQ/2
  END;
  "Handle a carry off the end, if present."
  IF i = 1 THEN BEGIN
    lSibling[rF] ← Bk[1];
    lSibling[lChild[Bk[1]]] ← rF;
    rF ← Bk[1]
  END;

  "Link the result into Q, and clear out T."

  lSibling[rF] ← Q;
  QueueHeader:lChild![Q] ← lSibling[dummy];
```

```
  "At this point the dummy node can be explicitly deallocated to save GC's "
  IF !Child[QueueHeader:!Child![Q]] ≠ NULL_RECORD THEN
    !Sibling[!Child[QueueHeader:!Child![Q]]] ← Q;
  QueueHeader:Size[Q] ← QueueHeader:Size[T] + QueueHeader:Size[Q];
  QueueHeader:!Child![T] ← NULL_RECORD;
  QueueHeader:Size[T] ← 0;
END "Union";
```

2. FAIL Implementations.

2.1 Binomial Queue using Structure R.

```
        TITLE   BQ
        SUBTTL  Binomial queue priority queue routines using structure R.

        ENTRY   INS_BQ,DEL_BQ

                        ;Registers:

        $SAIL_R←1       ;SAIL result register for typed procedures.
        $SAIL_P←17      ;SAIL regular PDL.


                        ;Node Fields:

                        ;Header node:
        $LMOST←0        ;Pointer to root of leftmost tree in forest.
        $SIZE← 1        ;Fullword integer count of number of elements in queue.

                        ;Queue element (binomial queue) node:
        $LCHILD←0       ;Pointer to leftmost child of this node (right half).
        $LSIBLING←0     ;Pointer to next sib to left of this node (left half).
                        ;If no left sib, then points to rightmost sib instead.
        $KEY← 1         ;Fullword integer or real key (i.e., node priority).


                        ;Trap handlers:
$INS_UFL:HALT           ;Here on insertion into queue with SIZE < 0.
$DEL_UFL:HALT           ;Here on deletion from queue with SIZE ≤ 0.
```

2. FAIL Implementations.

2.1 Binomial Queue using Structure R.

```
        TITLE   BQ
        SUBTTL  Binomial queue priority queue routines using structure R.

        ENTRY   INS_BQ,DEL_BQ

                        ;Registers:

        ↓SAIL_R←1       ;SAIL result register for typed procedures.
        ↓SAIL_P←17      ;SAIL regular PDL.


                        ;Node Fields:

                        ;Header node:
        ↓LMOST←←0       ;Pointer to root of leftmost tree in forest.
        ↓SIZE←← 1       ;Fullword integer count of number of elements in queue.

                        ;Queue element (binomial queue) node:
        ↓LCHILD←←0      ;Pointer to leftmost child of this node (right half).
        ↓RSIBLING←←0    ;Pointer to next sib to left of this node (left half).
                        ;If no left sib, then points to rightmost sib instead.
        ↓KEY←← 1        ;Fullword integer or real key (i.e., node priority).


                        ;Trap handlers:
↓INS_UFL:HALT           ;Here on insertion into queue with SIZE < 0.
↓DEL_UFL:HALT           ;Here on deletion from queue with SIZE ≤ 0.
```

```
        BEGIN   INS_BQ          ;Binomial queue insertion

        T0←     0
        T1←     1
        Q←      2
        X←      3
        LMR←    4               ;LMOST_ROOT
        RMR←    5               ;RMOST_ROOT
        NXTR←   6               ;NEXT_ROOT
        S←      7
        S1←     S+1

                                ;procedure INSERT(reference(NODE) X;
↑INS_BQ:HRRZ    Q,-1(SAIL_P)    ;                 reference(QUEUEHEADER) Q);
        AOSG    S,SIZE(Q)       ;S ← SIZE(Q) ← SIZE(Q) + 1
        JRST    INS_UFL         ;if S ≤ 0 then ERROR endif
        MOVE    LMR,LMOST(Q)    ;LMOST_ROOT ← LMOST(Q)
        HLRZ    RMR,LSIBLING(LMR);RMOST_ROOT ← LSIBLING(LMOST_ROOT)
        HRRZ    X,-2(SAIL_P)
        SETZM   LSIBLING(X)     ;LCHILD(X) ← NIL;
        LSHC    S,-1
        JUMPL   S1,M_DONE       ;if even(S) then  {Handle first merge specially.}
        HLRZ    NXTR,LSIBLING(RMR);NEXT_ROOT ← LSIBLING(RMOST_ROOT)
        MOVE    T0,KEY(X)
        CAMLE   T0,KEY(RMR)     ;  if  KEY(X) > KEY(RMOST_ROOT)
        EXCH    X,RMR           ;    then X ↔ RMOST_ROOT endif
        HRRM    RMR,LCHILD(X)   ;  LCHILD(X) ← RMOST_ROOT
        MOVSM   RMR,LSIBLING(RMR); LSIBLING(RMOST_ROOT) ← RMOST_ROOT;
        MOVEI   RMR,(NXTR)      ;  RMOST_ROOT ← NEXT_ROOT
        LSHC    S,-1            ;  S ← S/2
        JUMPL   S1,M_DONE       ;  loop while even(S);
M_LOOP: HLRZ    NXTR,LSIBLING(RMR);  NEXT_ROOT ← LSIBLING(RMOST_ROOT)
        MOVE    T0,KEY(X)
        CAMLE   T0,KEY(RMR)     ;    if  KEY(X) > KEY(RMOST_ROOT)
        EXCH    X,RMR           ;      then X ↔ RMOST_ROOT endif
        HRRZ    T1,LCHILD(X)    ;    T1 ← LCHILD(X)
        HRRM    RMR,LCHILD(X)   ;    LCHILD(X) ← RMOST_ROOT
        HLRZ    T0,LSIBLING(T1) ;    T0 ← LSIBLING(T1)
        HRLM    RMR,LSIBLING(T1);    LSIBLING(T1) ← RMOST_ROOT
        HRLM    T0,LSIBLING(RMR);    LSIBLING(RMOST_ROOT) ← T0
        MOVEI   RMR,(NXTR)      ;    RMOST_ROOT ← NEXT_ROOT
        LSHC    S,-1            ;    S ← S/2
        JUMPGE  S1,M_LOOP       ;  repeat
                                ;endif;
M_DONE: JUMPN   S,LINKIN        ;if S = 1 then
        MOVEM   X,LMOST(Q)      ;  LMOST(Q) ← X;
        HRLM    X,LSIBLING(X)   ;  LSIBLING(X) ← X
        JRST    EXIT            ;else
LINKIN: HRLM    X,LSIBLING(LMR) ;  LSIBLING(LMOST_ROOT) ← X;
        HRLM    RMR,LSIBLING(X) ;  LSIBLING(X) ← RMOST_ROOT
                                ;endif
EXIT:   SUB     SAIL_P,[3,,3]
        JRST    @3(SAIL_P)
        BEND    INS_BQ          ;end {INSERT}
```

```
        BEGIN   DEL_BQ          ;Binomial queue deletion

        T0=     0
        T1=     2
        Q=      3
        LMR=    4               ;LMOST_ROOT
        RMR=    5               ;RMOST_ROOT
        NRMR=   6               ;NEW_RTMOST_ROOT
        MC=     7               ;MERGE_CHILD
        RPRED=  10              ;RESULT_PRED
        RMC=    11              ;RMOST_CHILD
        S=      13
        S1=     S+1
        BEST_KEY=T0
        PRED=   T1
        SUCC=   MC
        NC=     S               ;NEXT_CHILD

                                ;reference(NODE) procedure DeleteSmallest
                                ;                   (reference (QUEUEHEADER) Q);
DEL_BQ: HRRZ    Q,-1(SAIL_P)    ;
        SOSGE   S,SIZE(Q)       ;S ← SIZE(Q) ← SIZE(Q) - 1;
        JRST    DEL_UFL         ;if S < 0 then ERROR endif;
        MOVE    LMR,LMOST(Q)    ;LMOST_ROOT ← LMOST(Q);
        HLRZ    RMR,LSIBLING(LMR);RMOST_ROOT ← LSIBLING(LMOST_ROOT);
        CAIN    RMR,(LMR)       ;if LMOST_ROOT = RMOST_ROOT then
                                ;  (The forest consists of a single tree, whose root
                                ;  contains the best key in the forest.  Remove the
                                ;  root, making the new forest from its children,
                                ;  and we're done.)
        JRST    (MOVE)  SAIL_R,(LMR);DEL_BQ ← LMOST_ROOT;
                HRRZ    T0,LCHILD(LMR)
                MOVEM   T0,LMOST(Q);LMOST(Q) ← LCHILD(LMOST_ROOT)
                JRST    EXIT)
                                ;else
                                ;  (The forest consists of more than one tree;
                                ;  search the roots of these trees for the best key.
                                ;  Scan the forest from right to left, but use the
                                ;  best key in the leftmost tree as an estimate of
                                ;  the best key in the forest.)
        HLLM    RMR,LSIBLING(LMR); LSIBLING(LMOST_ROOT) ← NIL;
        MOVE    DEST_KEY,KEY(LMR); BEST_KEY ← KEY(LMOST_ROOT);
        MOVEI   PRED,(LMR)      ;  PRED ← LMOST_ROOT;
        MOVEI   SUCC,(RMR)      ;  SUCC ← RMOST_ROOT;
        JRST    S_LOOP
NEW_BST:MOVE    BEST_KEY,KEY(SUCC)
        MOVEI   RPRED,(PRED)
        MOVEI   PRED,(SUCC)
        HLRZ    SUCC,LSIBLING(SUCC)
        JUMPE   SUCC,S_DONE     ;  loop until SUCC = NIL;
S_LOOP: CAML    BEST_KEY,KEY(SUCC); if BEST_KEY ≥ KEY(SUCC)
        JRST    NEW_BST         ;     then BEST_KEY ← KEY(SUCC);
                                ;     RESULT_PRED ← PRED
                                ;  endif;
        MOVEI   PRED,(SUCC)     ;  PRED ← SUCC;
        HLRZ    SUCC,LSIBLING(SUCC); SUCC ← LSIBLING(SUCC)
        JUMPN   SUCC,S_LOOP     ; repeat;

S_DONE: HLRZ    SAIL_R,LSIBLING(RPRED);DEL_BQ ← LSIBLING(RESULT_PRED);
        LSHC    S,-1
        JUMPN   SAIL_R,NOT_RM   ;  if DEL_BQ = NIL then
                                ;    (The best key is in the root of the smallest
                                ;    tree in the forest.  If this root has children,
```

```
                                ;    then they move up to become the roots of the
                                ;    smallest trees in the forest, and we're done.!
        MOVEI   SAIL_R,(RMR)    ;    DEL_BQ ← RMOST_ROOT;
        JUMPL   S1,SN_BEST      ;    if even(S) then
                                ;      !The best key is in an S0 tree (which is
                                ;      distinct from the leftmost tree.)  Remove
                                ;      the S0 and fix link from the leftmost tree.!
S0_BEST:HLRZ    T0,LSIBLING(SAIL_R)
        HRLM    T0,LSIBLING(LMR);      LSIBLING(LMOST_ROOT) ← LSIBLING(DEL_BQ)
        JRST    EXIT            ;    else
                                ;      !The best key has children.  Link them into
                                ;      the right of the forest.!
SN_BEST:HRRZ    T1,LCHILD(SAIL_R)
        HLRZ    T0,LSIBLING(T1)
        HRLM    T0,LSIBLING(LMR);      LSIBLING(LMOST_ROOT)←LSIBLING(LCHILD(DEL_BQ))
        HLRZ    T0,LSIBLING(SAIL_R)
        HRLM    T0,LSIBLING(T1) ;      LSIBLING(LCHILD(DEL_BQ)) ← LSIBLING(DEL_BQ);
        JRST    EXIT            ;    endif
                                ;  else
NOT_RM:                         ;    !The best key is in the root of some tree other
                                ;    than the rightmost in the forest.  Children of
                                ;    this root which are smaller than the rightmost
                                ;    tree will move up into the forest;  the other
                                ;    children will combine with the rightmost tree
                                ;    to form a replacement tree.!
        HRRZ    T1,LCHILD(SAIL_R)
        HLRZ    RMC,LSIBLING(T1);    RMOST_CHILD ← LSIBLING(LCHILD(DEL_BQ));
                                ;    !Mark end of children-list.!
        HLLM    T1,LSIBLING(T1) ;    LSIBLING(LCHILD(DEL_BQ)) ← NIL;
        JUMPGE  S1,S_EVEN       ;    if odd(S) then
                                ;      !Some, but not all, children of best root
                                ;       will move up to be roots in the forest.!
S_ODD:  MOVEI   NRMR,(RMC)      ;      NEW_RTMOST_ROOT ← RMOST_CHILD;
        LSHC    S,-1            ;      S ← S/2;
        JUMPGE  S1,C_DONE       ;      loop while odd(S):
C_LOOP: HLRZ    RMC,LSIBLING(RMC);       RMOST_CHILD ← LSIBLING(RMOST_CHILD);
        LSHC    S,-1            ;        S ← S/2
        JUMPL   S1,C_LOOP       ;      repeat;
C_DONE: HLRZ    MC,LSIBLING(RMC);      MERGE_CHILD ← LSIBLING(RMOST_CHILD);
                                ;      !Now RMOST_CHILD is really the leftmost
                                ;      which will move up.  MERGE_CHILD is the
                                ;      rightmost child which will participate in
                                ;      the merge to produce a replacement tree.!
        CAIN    RPRED,(RMR)     ;      if RESULT_PRED = RMOST_ROOT then
                                ;        !LSIBLING link from RMOST_CHILD is same
                                ;        as LSIBLING link to replacement tree.!
        JRST    [MOVEI RPRED,(RMC);      RESULT_PRED ← RMOST_CHILD
                 JRST   M_LOOP] ;      else
                                ;        !Link children into forest now.!
        HLRZ    T0,LSIBLING(RMR);        LSIBLING(RMOST_CHILD) ←
        HRLM    T0,LSIBLING(RMC);                              LSIBLING(RMOST_ROOT)
        JRST    M_LOOP          ;      endif
                                ;    else
                                ;      !The rightmost tree in the forest is an S0.
                                ;      This will combine with all children of the
                                ;      best root to produce the replacement tree.!
S_EVEN: HLRZ    NRMR,LSIBLING(RMR);    NEW_RTMOST_ROOT ← LSIBLING(RMOST_ROOT);
        CAIN    RPRED,(RMR)     ;      if RESULT_PRED = RMOST_ROOT then
                                ;        !The replacement tree will be the rightmost
                                ;        in the new forest, so the LSIBLING link
                                ;        from the leftmost root will be the LSIBLING
                                ;        link to the replacement tree.!
```

```
        MOVEI   RPRED,0         ;         RESULT_PRED ← NIL
                                ;       endif;
                                ;       {The merger of two S0's is a special case,
                                ;       handled here.}
        HLRZ    MC,LSIBLING(RMC);       MERGE_CHILD ← LSIBLING(RMOST_CHILD);
        MOVE    T0,KEY(RMR)
        CAMLE   T0,KEY(RMC)     ;       if KEY(RMOST_ROOT) > KEY(RMOST_CHILD)
        EXCH    RMR,RMC         ;         then RMOST_ROOT ↔ RMOST_CHILD endif;
        HRRM    RMC,LCHILD(RMR) ;       LCHILD(RMOST_ROOT) ← RMOST_CHILD;
        MOVSM   RMC,LSIBLING(RMC);      LSIBLING(RMOST_CHILD) ← RMOST_CHILD;
                                ;       LCHILD(RMOST_CHILD) ← NIL;
                                ;     endif;
                                ;     {Now finish the merge.}
        JUMPE   MC,M_DONE       ;     loop until MERGE_CHILD = NIL:
M_LOOP: HLRZ    NC,LSIBLING(MC) ;       NEXT_CHILD ← LSIBLING(MERGE_CHILD);
        MOVE    T0,KEY(RMR)
        CAMLE   T0,KEY(MC)      ;       if  KEY(RMOST_ROOT) > KEY(MERGE_CHILD)
        EXCH    RMR,MC          ;         then RMOST_ROOT ↔ MERGE_CHILD endif;
        HRRZ    T1,LCHILD(RMR)  ;       T1 ← LCHILD(RMOST_ROOT);
        HRRM    MC,LCHILD(RMR)  ;       LCHILD(RMOST_ROOT) ← MERGE_CHILD;
        HLRZ    T0,LSIBLING(T1) ;       T0 ← LSIBLING(T1);
        HRLM    MC,LSIBLING(T1) ;       LSIBLING(T1) ← MERGE_CHILD;
        HRLM    T0,LSIBLING(MC) ;       LSIBLING(MERGE_CHILD) ← T0;
        MOVEI   MC,(NC)         ;       MERGE_CHILD ← NEXT_CHILD
        JUMPN   MC,M_LOOP       ;     repeat;
                                ;     {Its time to tie up the loose ends...}
M_DONE: CAIE    SAIL_R,(LMR)    ;     if DEL_BQ = LMOST_ROOT
        JRST    R_N_LM          ;       then
R_LM:   MOVEM   RMR,LMOST(Q)    ;       LMOST(Q) ← RMOST_ROOT;
        JUMPN   RPRED,.+3       ;       if RESULT_PRED = NIL
        HRLM    RMR,LSIBLING(RMR);        then LSIBLING(RMOST_ROOT) ← RMOST_ROOT.
        JRST    EXIT            ;       else
        HRLM    NRMR,LSIBLING(RMR);       LSIBLING(RMOST_ROOT) ← NEW_RTMOST_ROOT;
        HRLM    RMR,LSIBLING(RPRED);      LSIBLING(RESULT_PRED) ← RMOST_ROOT
        JRST    EXIT            ;       endif
                                ;     else
R_N_LM: HLRZ    T0,LSIBLING(SAIL_R)
        HRLM    T0,LSIBLING(RMR);     LSIBLING(RMOST_ROOT) ← LSIBLING(DEL_BQ);
        JUMPN   RPRED,.+3       ;       if RESULT_PRED = NIL
        HRLM    RMR,LSIBLING(LMR);        then LISBLING(LMOST_ROOT) ← RMOST_ROOT
        JRST    EXIT            ;       else
        HRLM    RMR,LSIBLING(RPRED);      LSIBLING(RESULT_PRED) ← RMOST_ROOT;
        HRLM    NRMR,LSIBLING(LMR);       LSIBLING(LMOST_ROOT)←NEW_RTMOST_ROOT
                                ;       endif
                                ;     endif
                                ;   endif
                                ;endif;

EXIT:   SETZM   LSIBLING(SAIL_R);LSIBLING(DEL_BQ) ← LCHILD(DEL_BQ) ← NIL;
        SUB     SAIL_P,[2,,2]
        JRST    @2(SAIL_P)
        BEND    DFL_BQ          ;end {DEL_BQ};
```

2.2 Leftist Tree.

```
        TITLE   LT
        SUBTTL  Leftist tree priority queue routines.

        ENTRY   INS_LT,DEL_LT

                        ;Registers:

        $SAIL_R=1       ;SAIL result register for typed procedures.
        $SAIL_P=17      ;SAIL regular PDL.


                        ;Node Fields:

                        ;Header node:
        $ROOT== 0       ;Pointer to root of leftist tree.
        $SIZE== 1       ;Fullword integer count of number of elements in queue.

                        ;Queue element (leftist-tree) node:
        $DIST== 0       ;Length of shortest (rightest) path from this node to NIL.
        $LEFT== 1       ;Pointer to left child of this node (left half).
        $RIGHT==1       ;Pointer to right child of this node (right half).
        $KEY==  2       ;Fullword integer or real key (i.e., node priority).

                        ;Trap handlers:
$INS_UFL:HALT           ;Here on insertion into queue with SIZE < 0.
$DEL_UFL:HALT           ;Here on deletion from queue with SIZE ≤ 0.
```

```
                                ;Leftist tree insertion and deletion.

        .P1=    3               ;Registers used for linkage with MRG_LT
        .Q1=    4               ;by INS_LT, DEL_LT.
        .N=     15

        BEGIN   MRG_LT

        COMMENT • Procedure to merge two leftist trees.  Called by JSP N,MRG_LT,
        with the trees to be merged in P1 and Q1; returns with result tree in P1.
        This procedure uses AC DIST to eliminate special checks for NIL in the
        rebalancing phase; thus DIST should not conflict with ACs which the caller
        expects to be preserved: SAIL_P, SAIL_R, Q, and N. •

        R=      5
        T1=     6
        K=      T1
        D=      7

↑MRG_LT:                        ;reference(NODE) procedure MRG_LT
                                ;                       (reference(NODE) P,Q);
                                ;!The "rightmerge" phase merges the rightmost paths
                                ;of P and Q into a single path, preserving the
                                ;property that keys decrease from the root toward
                                ;the leaves.  Since the next phase will want to
                                ;traverse this path from the leaves toward the root,
                                ;the path is linked upward, using the RIGHT field.
                                ;At the end of this phase, R points to the lowest
                                ;node on the new path, and P contains a "leftover"
                                ;tree which will become a child of R.!
        MOVEI   R,0             ;R ← NIL;
        JRST    DNLOOP          ;loop until QNIL or PNIL;
QSMALL: HRRZ    T1,RIGHT(Q1)
        HRRM    R,RIGHT(Q1)
        MOVEI   R,(Q1)
        MOVEI   Q1,(T1)
DNLOOP: JUMPE   Q1,QNIL         ;  if Q = NIL then QNIL endif;
        JUMPE   P1,PNIL         ;  if P = NIL then PNIL endif;
        MOVE    K,KEY(P1)
        CAMLE   K,KEY(Q1)       ;  if KEY(P) ≤ KEY(Q) then
        JRST    QSMALL
PSMALL: HRRZ    T1,RIGHT(P1)    ;    T ← RIGHT(P);
        HRRM    R,RIGHT(P1)     ;    RIGHT(P) ← R;
        MOVEI   R,(P1)          ;    R ← P;
        MOVEI   P1,(T1)         ;    P ← T
        JRST    DNLOOP
                                ;  else
        COMMENT • see QSMALL •  ;    T ← RIGHT(Q);
                                ;    RIGHT(Q) ← R;
                                ;    R ← Q;
                                ;    Q ← T
                                ;  endif
                                ;repeat;
                                ;then
PNIL:   MOVEI   P1,(Q1)         ;  PNIL => P ← Q; Q ← DIST(P);
QNIL:   MOVE    Q,DIST(P1)      ;  QNIL => Q ← DIST(P);
                                ;end;
                                ;!The "rebalance" phase marches up the path
                                ;created by the rightmerge, interchanging the
                                ;left and right children of nodes as necessary
                                ;to guarantee that the result is leftist.  At the
                                ;end, P1 points to the result.!
```

```
        MOVEI   T1,0            ;DIST(NIL) = 0;
        JRST    UPLOOP          ;loop until R = NIL;
NOSWITCH:MOVEI  Q,1(Q)
        HRRM    P1,RIGHT(R)
MOVEUP: MOVEM   Q,DIST(R)
        MOVEI   P1,(R)
        MOVEI   R,(Q1)
UPLOOP: JUMPE   R,(N)
        HRRZ    Q1,RIGHT(R)     ;  Q ← RIGHT(R);  !Here Q is the next higher node on
        HLRZ    T1,LEFT(R)      ;  the path, P is our partial result and is leftist,
        CAMG    Q,DIST(T1)      ;  and Q = DIST(P). We may need to interchange the
        JRST    NOSWITCH        ;  children of R.!
SWITCH: MOVE    Q,DIST(T1)      ;  if Q > DIST(LEFT(R)) then  !Do the interchange.!
        MOVEI   Q,1(Q)          ;    Q ← DIST(LEFT(R)) + 1;
        HRRM    T1,RIGHT(R)     ;    RIGHT(R) ← LEFT(R);
        HRLM    P1,LEFT(R)      ;    LEFT(R) ← P
        JRST    MOVEUP
                                ;  else  !No interchange is needed.!
        COMMENT ← see NOSWITCH ←;    Q ← Q + 1;
                                ;    RIGHT(R) ← P
                                ;  endif;
                                ;  !Now complete R and move up the path.!
        COMMENT ← see MOVEUP ←  ;  DIST(R) ← Q;
                                ;  P ← R;
                                ;  R ← Q
                                ;repeat
        END     MRG_LT          ;end !MRG_LT!;

        BEGIN   INS_LT

        1←      0
        Q←      2
                                ;procedure INS_LT(reference(NODE) X;
↑INS_LT:HRRZ    Q,-1(SAIL_P)    ;                 reference(LT_QUEUE) Q);
        AOSG    SIZE(Q)         ;SIZE(Q) ← SIZE(Q) + 1;
        JRST    INS_UFL         ;if SIZE(Q) ≤ 0 then ERROR endif;
        MOVE    P1,ROOT(Q)
        HRRZ    Q1,-2(SAIL_P)
        MOVEI   T,1
        MOVEM   T,DIST(Q1)      ;DIST(X) ← 1;
        SETZM   RIGHT(Q1)       ;LEFT(X) ← RIGHT(X) ← NIL;
        JSP     N,MRG_LT
        MOVEM   P1,ROOT(Q)      ;ROOT(Q) ← MRG_LT(ROOT(Q),X)
        SUB     SAIL_P,[3,,3]
        JRST    @3(SAIL_P)
        END     INS_LT          ;end !INS_LT!;

        BEGIN   DEL_LT

        Q←      2
                                ;reference(NODE) procedure DEL_LT
↑DEL_LT:HRRZ    Q,-1(SAIL_P)    ;                          (reference(LT_QUEUE) Q);
        SOSGE   SIZE(Q)         ;SIZE(Q) ← SIZE(Q) - 1;
        JRST    DEL_UFL         ;if SIZE(Q) < 0 then ERROR endif;
        MOVE    SAIL_R,ROOT(Q)  ;DEL_LT ← ROOT(Q);
        HRRZ    Q1,RIGHT(SAIL_R)
        HLRZ    P1,LEFT(SAIL_R)
        JSP     N,MRG_LT
        MOVEM   P1,ROOT(Q)      ;ROOT(Q) ← MRG_LT(LEFT(ROOT(Q)),RIGHT(ROOT(Q)))
        SUB     SAIL_P,[2,,2]
        JRST    @2(SAIL_P)
        END     DEL_LT          ;end !DEL_LT!;
```

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: STAN-CS-77-600 ✓
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): The analysis of a practical and nearly optimal priority queue .
5. TYPE OF REPORT & PERIOD COVERED: technical, March 1977
6. PERFORMING ORG. REPORT NUMBER: STAN-CS-77-600 ✓

7. AUTHOR(s): Mark R. Brown
8. CONTRACT OR GRANT NUMBER(s): N00014-76-C-0330, NSF-MCS-72-03712

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Stanford University, Computer Science Department, Stanford, Ca. 94305
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Department of the Navy, Arlington, Va. 22217
12. REPORT DATE: March 1977
13. NUMBER OF PAGES: 102

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): ONR Representative: Philip Surra, Durand Aeronautics Bldg., Rm. 165, Stanford University, Stanford, Ca. 94305
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report): releasable without limitations on dissemination

Technical rept.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number): analysis of algorithms

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The binomial queue, a new data structure for implementing priority queues that can be efficiently merged, was recently discovered by Jean Vuillemin; we explore the properties of this structure in detail. New methods of representing binomial queues are given which reduce the storage overhead of the structure and increase the efficientcy of operations on it. One of these representations allows any element of an unknown priority queue to be deleted in log time, using only two pointers per element of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

094 120

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

the queue. A complete analysis of the average time for insertion into and deletion from a binomial queue is performed. This analysis is based on the result that the distribution obtained after repeated insertions and deletions.

An abstract notion of priority queue efficiency is defined, based on comparison counting. A good lower bound on the average and worst case number of comparisons is derived; several priority queue algorithms are exhibited which nearly attain the bound. It is shown that one of these algorithms, using binomial queues, can be characterized in a simple way based on the number and type of comparisons that it requires. The proof of this result involves an interesting problem on trees for which Huffman's construction gives a solution.