SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

IMPLEMENTATION AND ANALYSIS OF BINOMIAL QUEUE ALGORITHMS*

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.

Key words. analysis of algorithms, binomial queue, priority queue, heap

Introduction. 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 the qualification "mergeable" is generally reserved for those priority queues which can be merged quickly: merging should not require examining a positive fraction of the items in the queues.

The priority queue is recognized as a useful abstraction due to the large number of applications in which it arises [1], [2], [13]. Priority queues are also interesting simply because a number of subtle data structures have been devised for representing them, including heaps [13], leftist trees [13], and 2-3 trees with heap ordering [1]; the last two of these structures are mergeable. Recently the *binomial queue*, a new data structure for implementing mergeable priority queues, was described by Vuillemin [21]. This structure does not improve upon the asymptotic time bounds already known for the operations it performs, but is interesting because of its intrinsic beauty and simplicity, and because it uses less storage than other methods.

One goal of this paper is to show that the binomial queue is not just another pretty data structure, but is the most practical structure for priority queues in many situations. Section 1 defines binomial queues, and § 2 gives algorithms for operating on them. Section 3 discusses the underlying structures which seem most suitable for

* Received by the editors October 29, 1976, and in revised form May 27, 1977.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. This research was supported by a National Science Foundation graduate fellowship at Stanford University.

implementing binomial queues under various assumptions. One of the structures allows an arbitrary 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. (Implementation of the priority queue primitives, using two of the structures introduced in § 3, are given in [2].)

A second goal of the paper is to give a complete average-case analysis of the time required for insertion and deletion using binomial queues. In § 4 we show that our intuitive sense of the binomial queue's simplicity is reflected by the fact that binomial queues do not degenerate from their "random" state (the state brought about by consecutive random insertions) when deletions occur. This leads to the analysis of binomial queue algorithms given in § 5. The results of this analysis also help to establish the practicality of binomial queues by aiding in a comparison between binomial queues and other structures; the conclusion is that binomial queues are the fastest known mergeable priority queue organization, and may have some advantages even when fast merging is not required.

1. Binomial trees, forests and queues.

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

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

(1.2) Suppose that Y and Z are disjoint B_{k-1} trees for $k \geqq 1$. 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.

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 [15]; B_0 through B_4 trees are shown explicitly in an algorithm for prime implicant determination [17]; a B_5 tree is given as the frontispiece for [11]; and oriented binomial trees, called S_n trees, were used by Fischer in an analysis of set union algorithms [6].

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 illustrates rule (1.2) for building binomial trees, and Figure 2 displays the first few cases.

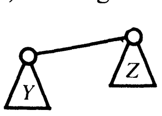


FIG. 1. *Construction of a binomial tree.*

An alternative construction rule, equivalent to (1.2), is often useful:

(1.3) Suppose that $Z_{k-1}, \cdots, Z_0$ are disjoint trees such that Z_l is a B_l tree for $0 \leqq l \leqq k-1$. 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 $Z_{k-1}, \cdots, Z_0$ the offspring of R, left to right in this order, is a B_k tree.

Figure 3 illustrates rule (1.3) for building binomial trees. The equivalence of (1.2) and (1.3) 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*;

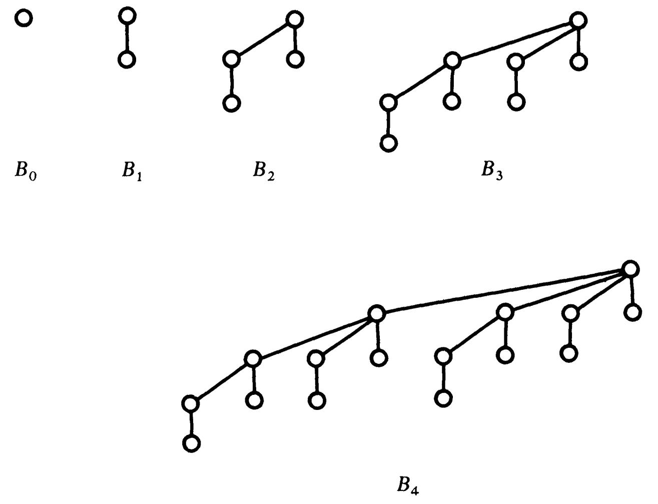


FIG. 2. *Small binomial trees.*

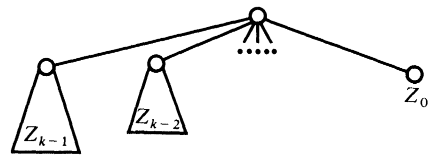


FIG. 3. *Alternative construction of a binomial tree.*

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

Proof. The proof is trivial by induction on k. □

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

(1.4) The forest contains m nodes.

(1.5) 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.)

Since by (1.5) 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 $\sum_{j\geqq 0} b_j \cdot 2^j$; 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 $m \geqq 0$, and that all binomial forests of a given size are isomorphic. Figure 4 shows some small binomial forests.

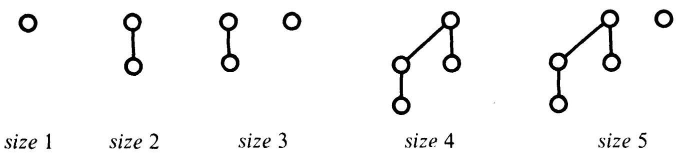


FIG. 4. *Small binomial forests.*

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. The proof is obvious. □

Consider a binomial forest of size m such that each node has an associated *key*, where a linear order $\leqq$ 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.

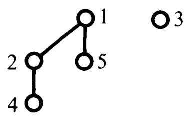


FIG. 5. *A binomial queue of size 5 (with integer keys).*

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 (1.2). 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 (1.3) and the fact that subtrees of a heap-ordered tree are heap-ordered. □

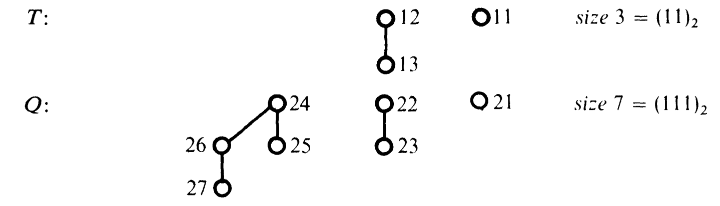
2. Binomial queue algorithms.

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 defined in the Introduction. In the following informal description of the algorithms we let $\|Q\|$ denote the number of elements in a queue Q.

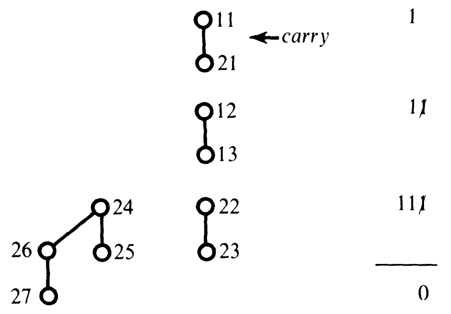
Consider first the operation Union (T, Q), which merges the elements of T into Q. If $\|T\| = t$ and $\|Q\| = q$, then the process of merging the binomial queues for T and Q is analogous to the process of adding t and q in binary. We successively "add" pairs of heap-ordered B_k trees, as described in 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.

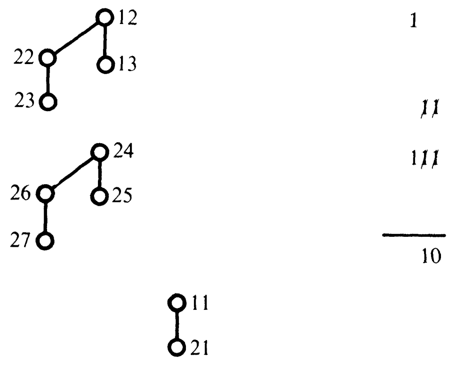
Given the ability to perform Union, the operation Insert (x, Q), which adds item x to queue Q, is trivial to specify: just let X be the binomial queue containing only the item x, and perform Union (X, Q). By this method, the time required for an insertion into Q is $O(\log \|Q\|)$.



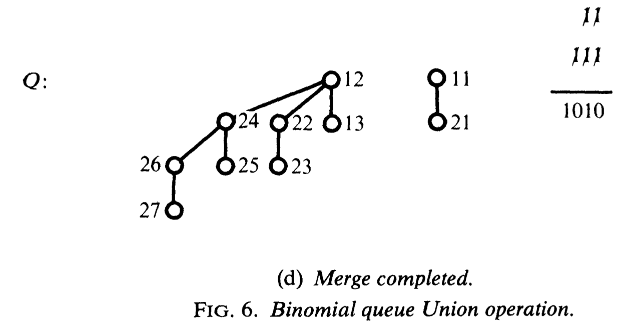
(a) *Binomial queues of size* 3 *and* 7 *to be merged for Union operation.*



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



(c) *After merge of* B_1*'s.*



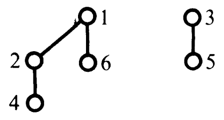
(d) *Merge completed.*

FIG. 6. *Binomial queue Union operation.*

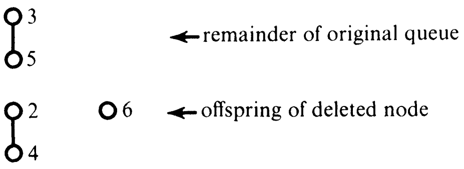
The operation DeleteSmallest (Q) is a bit more complicated. The first step is to locate the node x containing the smallest key. Since x is the root of one of the queue's B_k trees, it can be found by examining each of these roots once. By Lemma 2 this requires $O(\log \|Q\|)$ time.

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.

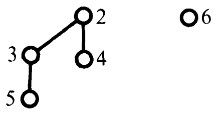
The final step consists of merging the two queues formed in the second step: the queue T' formed from T, and the queue Q' formed by removing T from Q. Since each queue is smaller than $\|Q\|$, the operation Union (T', Q') requires $O(\log \|Q\|)$ time; therefore the entire deletion requires $O(\log \|Q\|)$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



(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.*

FIG. 7. *DeleteSmallest on a binomial queue.*

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 Fig. 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 consistitue 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 should be mentioned that there is an alternative deletion algorithm for binomial queues which is analogous to deletion from a heap [13]. The first step of this DeleteSmallest procedure is to locate the node x containing the smallest key in the queue Q. Then x is removed from the tree T containing it, and if T was the smallest binomial tree in Q then the procedure terminates. Otherwise we remove the root node y from the smallest tree in Q and make y the root of T. At this point T is a binomial tree, but may no longer be heap-ordered; thus we "sift down" the node y by repeatedly exchanging y with the smallest of its offspring until y is smaller than all of its offspring. This establishes heap-order in T and completes the deletion process. Unfortunately, the siftdown step is relatively expensive: in a B_k tree, it requires $k-1$ comparisons to find the smallest offspring of the root, and may require $k-2$ comparisons to find the smallest offspring of this node, etc. Hence a worst-case siftdown may use $O(\log^2 \|Q\|)$ steps. (It should be clear how to generalize this DeleteSmallest procedure to handle arbitrary deletions in $O(\log^2 \|Q\|)$ steps.)

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 insertions. Consider the sequence of instructions

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

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 the Introduction, 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 $O(k+\log m)$.

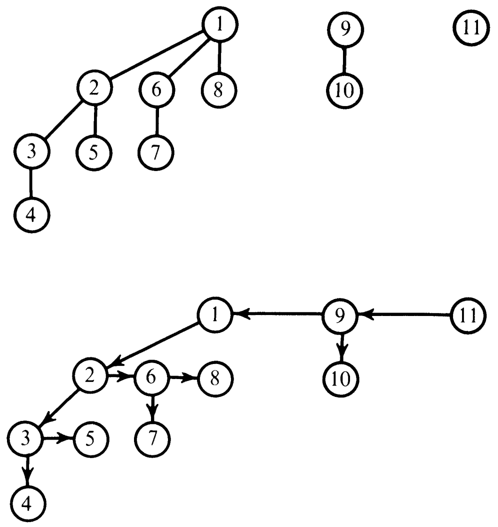
3. Structures for binomial queues.

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.

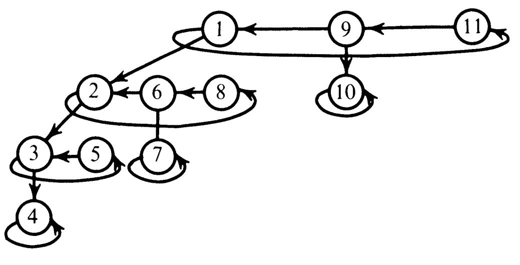
Since a binomial queue is a forest, it is natural to represent it as a binary tree [11]. But not all orientations of the binary tree links allow binomial queue operations to be performed efficiently. Evidently the individual trees of the binomial forest must be linked together from smaller to larger, 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 [21]; we shall call it structure V. An example of a binomial queue and its representation using structure V is given in Fig. 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 Fig. 8(b), eliminates this problem. In this structure the horizontal links point to the left, except that the leftmost tree among a group of siblings points to the rightmost. Downward links point to the

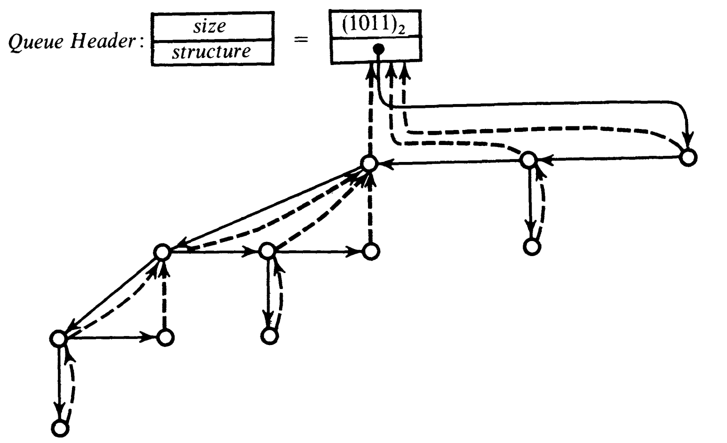


(a) *A binomial queue and its representation using structure V.*

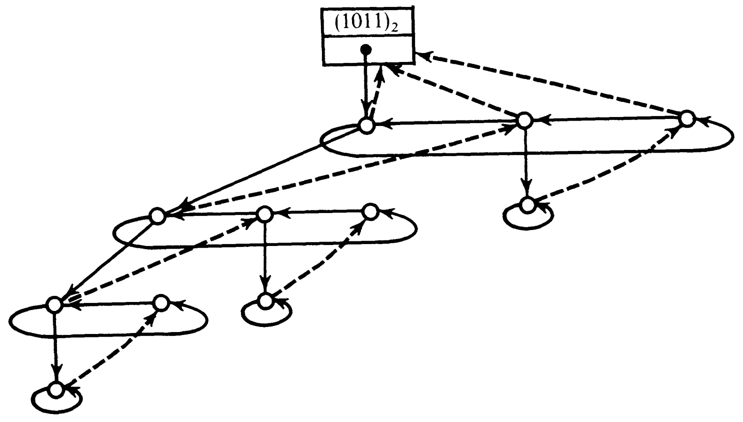


(b) *A representation for the same queue using structure R.*

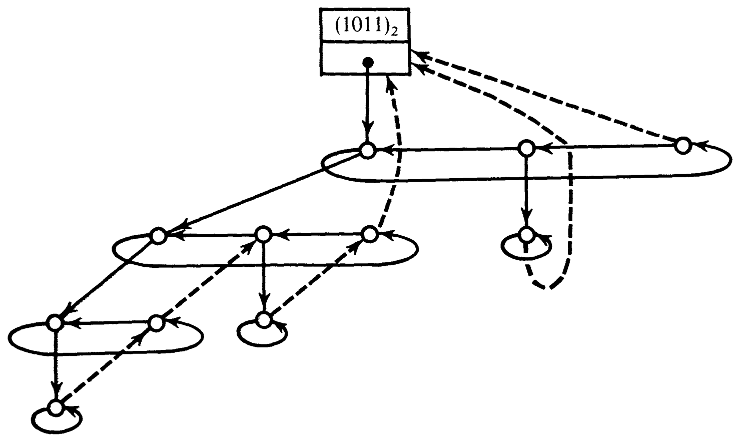
FIG. 8. *Structures for binomial queues.*



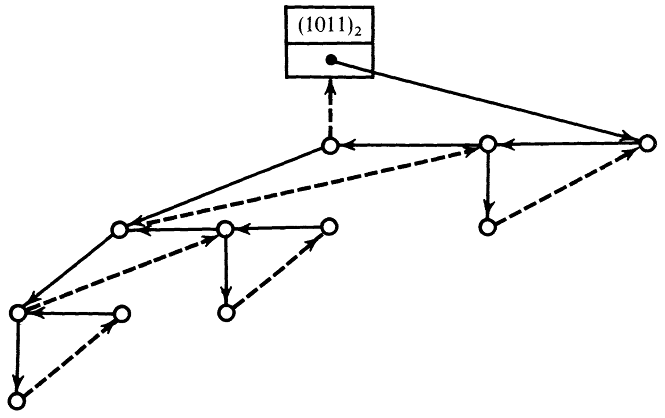
(a) *Structure V with upward pointers.*



(b) *Structure R with upward pointers.*



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



(d) *Structure K.*

FIG. 9. *Structures for binomial queues allowing arbitrary deletions.*

leftmost (largest) subtrees, as in structure V. 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 [3], 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 Fig. 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.

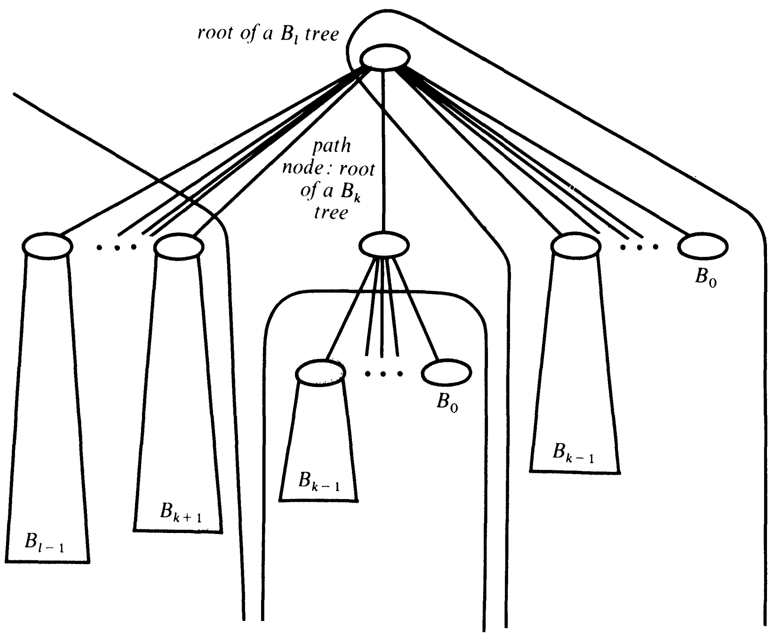


FIG. 10. *One step of the bottom-up deletion process.*

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*. (See Fig. 10.) 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 $B_{k-1}, \cdots, B_0$ 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 $B_{k+1}, \cdots, B_{l-1}$ 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. During the deletion process, nodes are visited in a zig-zag path moving upward to the queue header. We first move left among a group of siblings until the leftmost is reached, and then move up to the next level using the upward pointer. These steps are repeated until the queue header is reached.

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 Fig. 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 Fig. 9(d). This structure contains some null links, and seems to require less pointer updating per operation than the structure in Fig. 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.

4. Random binomial queues.

We define a *random binomial queue* of size m to be the queue formed by choosing a random permutation of $\{1, 2, \cdots, m\}$ 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 $\{1-\frac{1}{2}, 2-\frac{1}{2}, \cdots, m-\frac{1}{2}\}$, inserting x into the queue, and *renumbering* the queue such that the keys come from $\{1, 2, \cdots, m\}$ 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 mth 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 $X \leqq x$ 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 [8]; 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 $p=p_1p_2\cdots p_m$ be a permutation of $\{1, 2, \cdots, m\}$. Then in the binomial queue obtained by inserting $p_1, p_2, \cdots, p_m$ successively into an initially empty queue, the tree containing p_j is determined by j for $j=1, 2, \cdots, m$.*

Proof. We proceed by induction on m. The result is obvious for $m=1$. For $m>1$, let $l=2^{\lfloor \lg m \rfloor}$ 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 $m-l$ 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. $\square$

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 *do* insert in a manner which depends only on when the deleted element was inserted.

LEMMA 6. *Let* $p = p_1p_2 \cdots p_m$ *be a permutation of* $\{1, 2, \cdots, m\}$. *Then there is a mapping* $r = r_{j,m}$ *from* $\{1, 2, \cdots, m-1\}$ *onto* $\{1, 2, \cdots, j-1, j+1, \cdots, m\}$ *such that the result of inserting* $p_1p_2 \cdots p_m$ *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.

$$[0\ 0\ \bullet\ 0\ 0\ 0\ 0\ 0][\ ][0\ 0][\ ].$$

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.

$$[(0\ 0)\ \bullet\ (0)(0\ 0\ 0\ 0)][\ ][0\ 0][\ ].$$

This division clearly depends only on m and j.

Following the dismantling step is a merging step. Since the results of the deletion depend on details of the merging strategy, we must choose one; a 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

$$(0\ 0\ 0\ 0)(0\ 0)[0\ 0](0).$$

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 kth 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 kth 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 $\{1, 2, \cdots, m-1\}$. 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 $m \leqq n$ 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.

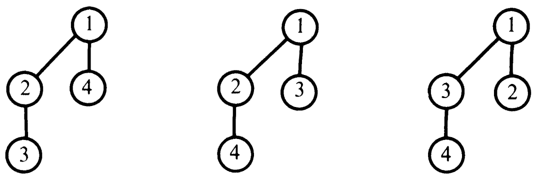
A similar argument shows that random binomial queues result when intermixed random deletions are performed; a simple argument appealing to Lemma 6 shows that intermixed deletions by age (how long an element has been in the queue) also lead to random queues. These types of deletions, especially deletions 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 [10] and Jonassen and Knuth [9]. 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.

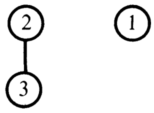
Binomial queues, however, are not sensitive to deletions in the search tree model. In a general study of deletion insensitivity, Knuth showed that Theorem 2 implies insensitivity to random deletions, and Lemma 6 implies insensitivity to deletions by age in this model [14]. Binomial queues are sensitive 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:



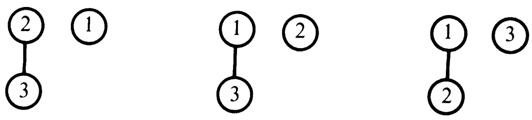
Pr = 1/3 Pr = 1/3 Pr = 1/3

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 jth 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 decompositon 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* $1, 2, \cdots, 2^k$ *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* $(2^k)!/2^{(2^k)-1}$ *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 $\{1, 2, \cdots, 2^j\}$ which give rise 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 Fig. 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 $2^{(2^i)-1}$ arrangements of the keys in tree Y which give rise to Y, and similarly for Z. So there are $2 \cdot 2^{(2^i)-1} \cdot 2^{(2^i)-1} = 2^{(2^{i+1})-1}$ 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 ith 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.

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 § 3 and pictured in Fig. 8(b). The priority queue structures to be used for comparison are the heap, leftist tree, sorted linear list, and unsorted linear list. (We do not perform a detailed comparison with 2-3 trees because they seem to be inferior to leftist trees when merging is needed, and to heaps when it is not.)

For each of the five structures, the operations Insert and DeleteSmallest have been carefully coded in the assembly language of a typical computer (the binomial queue and leftist tree implementations appear in [2]). 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 $1/2+O(1/m)$ for A and $1+O((\log m)/m)$ 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 [16] we find that

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

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

$$\frac{1}{2}\left(\lg m+\frac{1}{\ln 2}-\lg \frac{4}{3}\right) \leqq T_{\text{avg.}} \leqq \frac{1}{2}\left(\lg m+\frac{1}{\ln 2}\right),$$

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 [20] has shown that

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

using logarithmic averaging [5]; his result is based on the detailed analysis of $\sum_{1\leqq k\leqq m} \nu(k)$ performed by Hubert Delange [4]. 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 [19], [12, § 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 § 4.

The factors E and X are easy to dispose of. Since we only merge trees of equal size, our randomness assumptioin 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 § 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 $(1/m)F(m)$ where

$$F(m) = \sum_{\substack{k \geqq 0 \\ m=(b_l b_{l-1} \cdots b_0)_2}} b_k \cdot k \cdot 2^k.$$

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

LEMMA 7. *The function $F(m)$ defined above satisfies $\lceil m \lg m - 2m \rceil \leqq F(m) \leqq \lfloor m \lg m \rfloor$, $m \geqq 1$, 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 [16].) From the definition of $F(m)$, if $m = 2^k$ then

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

It is also clear from the definition that

$$F(2^k + i) = F(2^k) + F(i), \qquad 0 \leqq i < 2^k.$$

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

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

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

$$F(m) \geqq m \lg m - cm$$

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

$$\begin{aligned} F(m+i) &= F(2^k) + F(i) \qquad (0 \leqq i < 2^k) \\ &\geqq m \lg m + i \lg i - ci. \end{aligned}$$

It follows that if the inequality

$$m \lg m + i \lg i - ci \geqq (m+i) \lg (m+i) - c(m+i) \qquad (0 \leqq i < 2^k)$$

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

$$x \lg x \geqq (1+x) \lg (1+x) - c \qquad (0 \leqq x < 1).$$

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 [20] has shown that the logarithmically averaged value of S is

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

where

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

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 remaining 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 $m = (b_n b_{n-1} \cdots b_0)_2$ is

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

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

$$= \sum_{\substack{0 \leqq k \leqq n \\ b_k = 1}} \frac{2^k}{\sum_{0 \leqq l \leqq k,\, b_l = 1} 2^l}.$$

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

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

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

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

where

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

(The constant α also arises in connection with Heapsort; see [13, 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

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

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

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

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

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

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

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

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

where

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

(The constant α' arises in connection with merge sorting; see [13, 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 Fig. 11. A much simpler analysis [11, pp. 94–99] gives the corresponding results for sorted and unsorted linear lists (also shown in Fig. 11).

Priority queue algorithms based on heaps and leftist trees have not been completely analyzed; partial results are known for heaps [13], [18] but not for leftist trees. Therefore experiments were performed to estimate the average values of factors controlling the running time of these algorithms. (Note that the experiments did not consist of simply running the programs and timing them, but rather of running suitably instrumented programs which kept track of the number of times certain statements were executed. The averages computed in this way were then used in the expression for the program's running time, just as the mathematically derived averages were used in the case of binomial queues and linear lists.) 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 11 gives the experimentally determined running times for leftist trees and heaps.

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	$6m + 35$

FIG. 11. *Comparison of methods.*

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 $m \geqq 4$ when the measure of performance is the cost of an Insert followed by a DeleteSmallest. 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 $m \geqq 18$ 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. Using a redundant binary numbering scheme [7] it is possible to maintain the queue as a small number of 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 $m \geqq 30$.

Acknowledgments. The author would like to thank Robert Tarjan and Donald Knuth for helpful discussions relating to this work, and Daniel Boley, Lyle Ramshaw, and the referees for their criticisms of earlier drafts.

REFERENCES

[1] ALFRED V. AHO, JOHN E. HOPCROFT AND JEFFREY D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] MARK R. BROWN, *The analysis of a practical and nearly optimal priority queue*, Ph.D. thesis, STAN-CS-77-600, Computer Science Dept., Stanford Univ., Stanford, CA, March 1977, 99 pp.

[3] DAVID CHERITON AND ROBERT ENDRE TARJAN, *Finding minimum spanning trees*, this Journal, 5 (1976), pp. 724–742.

[4] HUBERT DELANGE, *Sur la Fonction Sommatoire de la Fonction « Somme des Chiffres »*, Enseignement Math. 21 (1975), no. 1, pp. 31–47.

[5] PERSI DIACONIS, *Examples in the theory of infinite iteration of summability methods*, Technical Report No. 86, Stanford Univ., Dept. of Statistics, May 1976.

[6] MICHAEL J. FISCHER, *Efficiency of equivalence algorithms*, Complexity of Computer Computations, Raymond E. Miller and James W. Thatcher, eds., Plenum Press, New York, 1972.

[7] LEO J. GUIBAS, EDWARD M. MCCREIGHT, MICHAEL F. PLASS AND JANET R. ROBERTS, *A new representation for linear lists*, Proc. 9th Annual ACM Symposium on the Theory of Computing, Boulder, CO, 1977, pp. 49–60.

[8] ARNE JONASSEN AND OLE-JOHAN DAHL, *Analysis of an algorithm for priority queue administration*, Math. Inst., Univ. of Oslo, 1975.

[9] ARNE T. JONASSEN AND DONALD E. KNUTH, *A trivial algorithm whose analysis isn't*, J. Comput. System Sci., to appear.

[10] GARY D. KNOTT, *Deletion in binary storage trees*, Ph.D. thesis, STAN-CS-75-491, Computer Science Dept., Stanford Univ., Stanford, CA, May 1975, 93 pp.

[11] DONALD E. KNUTH, *The Art of Computer Programming, Vol.* 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.

[12] ———, *The Art of Computer Programming, Vol.* 2, *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

[13] ———, *The Art of Computer Programming, Vol.* 3, *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[14] ———, *Deletions that preserve randomness*, IEEE Trans. Software Engrg., SE 3 (1977), pp. 351–359.

[15] DERRICK H. LEHMER, *The machine tools of combinatorics*, Applied Combinatorial Mathematics, Edwin F. Beckenbach, ed., John Wiley, New York, 1964.

[16] M. D. MCILROY, *The number of* 1*'s in binary integers*: *Bounds and extremal properties*, this Journal, 3 (1974), pp. 255–261.

[17] EUGENIO MORREALE, *Computational complexity of partitioned list algorithms*, IEEE Trans Computers, TC 19 (1970), pp. 421–428.

[18] THOMAS PORTER AND ISTVAN SIMON, *Random insertion into a priority queue structure*, IEEE Trans. Software Engrg., SE 1 (1975), pp. 292–298.

[19] RALPH A. RAIMI, *The first digit problem*, Amer. Math. Monthly, 83 (1976), no. 7, pp. 521–538.

[20] LYLE RAMSHAW, personal communication.

[21] JEAN VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, to appear.