Edgar H. Sibley
Panel Editor

Execution times for a variety of priority-queue implementations are compared under the hold model, showing many to be faster than implicit heaps.

AN EMPIRICAL COMPARISON OF PRIORITY-QUEUE AND EVENT-SET IMPLEMENTATIONS

DOUGLAS W. JONES

During the last decade, a number of new priority-queue implementations have evolved, including pagodas [6, 20], skew heaps [21, 22], splay trees [21, 23], pairing heaps [8], and binomial queues [4, 5, 24]. In addition, a number of special priority-queue implementations for representing the pending event set in discrete event simulation have been developed, including the two-list implementation [3] and Henriksen's implementation [10, 15]. The resulting variety of implementations is reminiscent of the variety of sorting algorithms. Comparison between the newer priority-queue implementations and the older ones suggests that many of the former are nearly optimal, but that no single implementation is best for all applications.

The basic operations on the priority-queue abstract data type are enqueue and dequeue (sometimes called insert and delete-min [1]). Enqueue places an item in a priority queue, and dequeue removes and returns the highest priority item from a priority queue. By convention, increasing priorities correspond to decreasing numerical values. As with all abstract data types, there are many ways to implement a priority queue; each implementation involves a data structure used to represent the queue, and an algorithm that implements each abstract operation in terms of this data structure.

In the priority-queue testing reported on in this article, a variant of the hold model [12, 15, 19] was used because it allows long sequences of operations to be performed on a queue of constant size, facilitating direct measurement of the average time per operation as a function of queue size. Unlike measurements made using actual priority-queue applications (e.g., those reported in [3] and [11] for various simulation models), the hold model allows direct comparison with theoretical predictions and straightforward extrapolation to new applications.

In this article, the performance of some of the newer and special-purpose priority-queue implementations is compared against the older, classic implementations and the results analyzed. The first section of this article describes the measurement methodology used (which readers primarily interested in practical advice about the selection of a priority-queue implementation may choose to skip). The balance of the article, in three main sections—"Results for Classical Implementations," "Simulation Event-Set Implementations," and "Nearly Optimal Implementations"—describes each of the 11 implementations tested, presenting results and giving general recommendations.

MEASUREMENT METHODOLOGY

The hold model is the most widely used approach to priority-queue performance measurement. (It is

used, for example, in [7], [10]–[13], [15], [16], and [19].) The model derives from the use of a priority queue to represent the pending event set in discrete event simulation, where the priority of each item in the event set represents the time at which some event is to happen. The enqueue operation schedules events, while the dequeue operation finds the next pending event. Simulation progresses by repeatedly dequeueing events, computing their consequences, and reporting the consequences either by updating the global state of the simulated system or enqueueing notices of additional future events. Any number of events may be scheduled as a consequence of one event. Some events only change the global simulation state, while others schedule large numbers of new events.

The hold model is based on the simplest of discrete-event-simulation models—those in which all events cause exactly one new event to be scheduled at some future time. From the standpoint of performance measurement, the advantage of this is that the size of the priority queue remains constant for the duration of the run. As a result, a series of hold-model runs with different queue sizes allows direct measurement of the average time for a dequeue followed by an enqueue as a function of queue size. (In the Simula language, the hold operation performs a dequeue followed by an enqueue, hence the name of the model.) Measurements are made by preparing an initial priority queue and then repeatedly dequeueing and enqueueing items and dividing the total time by the number of trials. The priority of each item enqueued during the test is determined by adding a random value to the priority of the most recently dequeued item, as described in the following code fragment:

```
{ create initial queue }
initialize( q );
for i := 1 to size do begin
     n := allocate;
     n↑.prio :=
       random_initial_priority;
     enqueue( n, q );
end;
{ perform sequence of hold operations }
t1 := time_of_day;
for i := 1 to trials do begin
          { one hold operation }
     n := dequeue( q );
     n↑.prio := n↑.prio +
       random_priority_increment;
     enqueue( n, q );
end;
t2 := time_of_day;
time_per_trial := (t2 - t1)/trials;
```

Aside from the choice of priority-queue implementation, the variables that can affect the times measured by the hold model are the number of items initially placed in the queue, the initial priority distribution of these items, the initial shape of the data structure representing the queue, and the priority increment distribution. It is shown [15] that the distribution of priority values in the queue approaches a steady state as the number of hold operations increases. This steady state depends only on the priority increment distribution.

Unfortunately, the queue shape does not approach a steady state under repeated hold operations; there are binary-tree-based queue implementations where the balance of the tree is preserved by hold operations and only changes when multiple enqueue or dequeue operations occur in sequence. To obtain a representative queue shape, the initial queue used for the measurements reported here was built using random sequences of enqueues and dequeues, halting as soon as a queue of the desired size was built. The probability of an enqueue was set slightly larger than the probability of a dequeue so that the queue would grow slowly to the desired size. To ensure that the priority distribution in the resulting queue was near the steady-state distribution, the priority of each item enqueued during this construction process was determined by adding a random increment to the priority of the most recently dequeued item.

Most analytical treatments of average priority-queue performance have assumed "random" queues [5, 13, 20] in which the distribution of values in the queue is the same as the distribution of expected values for newly enqueued items. As shown in [12], this corresponds to using a negative exponential priority increment distribution under the hold model. However, empirical observation of real simulation models shows that other distributions are quite common [19]. At the two extremes are distributions that result in first-in–first-out queue behavior, and those that result in nearly last-in–first-out behavior. It is therefore important to measure how sensitive the different priority-queue implementations are to changes in the priority distribution. The priority increment distributions used in the tests reported here are given in Table I (on the next page). These same distributions were used in [15] for both analytic and empirical tests of event-set implementations; the first four were also used in the empirical tests reported in [10] and [19].

In our testing, all measurements were made on machines with no interactive users, and all nonessential background processes were suspended, so that the results are largely unaffected by interrupts, context switches, or memory contention. All measurement runs were repeated three times with the

TABLE I. Priority Increment Distributions

Distribution[a]	Expression to compute random values[b]	Bias[c]
1. Exponential	$-ln(\text{rand})$	0.50
2. Uniform 0.0–2.0	2 rand	0.66
3. Biased 0.9–1.1	0.9 + 0.2 rand	0.97
4. Bimodal	0.95238 rand + if rand < 0.1 then 9.5238 else 0	0.13
5. Triangular	$1.5\,\text{rand}^{0.5}$	0.80

[a] All distributions have an expected value of 1.
[b] rand returns a random value uniformly distributed between 0 and 1.
[c] 1.0 corresponds to purely FIFO queue access; 0.0 is purely LIFO. In [19], this is called $\%F$; in [15], it is $E(G(x))$.

same random number seed so that obviously perturbed data could be detected and discarded. All measurements were based on 1,000 trials, and the cost of 10,000 iterations of the measurement loop was determined so that the measurement overhead could be subtracted from the times reported for each queue size. The number 10,000 was used to ensure that errors in the overhead measurement would not contribute significantly to errors in reported times. For most queue implementations and priority distributions, measurements were made for 28 exponentially-spaced queue sizes ranging from 1 to 11,585 ($2^{13.5}$).

The choice of 1000 trials represents a trade-off between a number of factors. Generally, more trials would lead to more accurate measurements if it were not for the fact that they also increase the possibility of disturbing the results by interrupts, cache flushes, or other system activity. As a result, an a priori error estimate could not be made, and the difficulty of obtaining exclusive use of large computer systems prevented the additional runs needed to measure the error at each data point. Therefore, the primary error estimation tool available involves an examination of the scatter of the data around a curve fit to the data. For most of the priority-queue implementations presented here, the data are well behaved for queue sizes of between 50 and 1000 elements, suggesting errors in measurement of approximately three percent of the full-scale value shown. As expected for smaller queues, some implementations perform quite erratically; the performance analyses of many of these implementations include ill-behaved terms that vanish as the queue size grows. Ill-behaved data for queues of larger than 1000 elements are probably due to an insufficient number of trials, since for some queue implementations average performance emerges only as the entire contents of the queue are replaced.

Most measurements were taken on a VAX 11/780 running UNIX® (BSD 4.2), using the Berkeley Pascal compiler, with optimization enabled. Although admittedly this compiler does not generate particularly good code, primarily because it does not make effective use of registers, the quality of code generation was considered adequate for purposes of queue implementation comparison, where code generation quality is not as critical as it would be in comparing machine architectures. What matters more in this case is code generation *consistency*, and this is easier to ensure with a high-level language implementation than it is with low-level languages like C or assembly.

Performance measurements made on the VAX 11/780 are unavoidably perturbed by the machine's 8-kbyte cache memory and 64-entry virtual address translation cache. This means that effective memory access time should improve when the set of addresses referenced by a program is smaller than 64 pages (32,768 bytes or 2,048 typical queue entries) and further improved at 8 kbytes (512 typical queue entries).

To identify the extent to which the relative performance of different queue implementations depends on architecture or compiler code generation strategy, measurements were also taken on an HP Series 200 model 236 workstation (9836A, based on the Motorola 68000 CPU clocked at 8 MHz) using HP Pascal (Rev. 3.0), and on a Prime 850 under Primos (Rev. 19.2.7) using Hull V-mode Pascal (Rev. 3.4A). The HP machine has no cache or virtual memory addressing mechanisms, while the Prime is comparable in size, speed, and complexity to the VAX.

Programming Conventions

All the priority-queue implementations tested were coded to conform to a common test driver interface. Conceptually, this interface exports the types queue, item, and pointer to item for use by the driver, and it exports the operations enqueue, dequeue, initqueue, and empty. (The last two operations initialize and test a queue to see if it is empty.) The data structures needed by each queue implementation

UNIX is a trademark of AT&T Bell Laboratories.

were simplified, when possible, by avoiding the inclusion of information needed for other operations like deletion of arbitrary items from queues or changing the priority of items already in a queue.

Some priority-queue implementations allow an especially efficient implementation of the hold operation: that is, a dequeue immediately followed by an enqueue. These implementations were not tested in this comparison, however, because, although the hold operation is the dominant priority-queue operation in some applications, it is of little or no use in many others.

All queue implementations were coded for speed: Recursion was eliminated from all code; small procedures and functions were expanded inline; and many common subexpressions, state variables, and redundant tests or assignments were eliminated. As a result, no implementation of enqueue or dequeue involved more than one extra procedure call. For many of the queue implementations, alternative forms of the code were tested empirically to determine which worked best. These additional tests were in addition to the tests of significant algorithmic variants discussed below. All code used in the final tests is available from the author.

Existing correctness proofs of the queue implementations tested were supplemented with two empirical tests to verify that no items in the queue were gained or lost, and that successive dequeues removed items in priority order. A test of stability was also included for implementations that were supposed to preserve the order of items with equal priority. The inclusion of these tests provides some protection against the kind of mistakes published in [7] and exposed in [11] and [19]. In terms of the results reported here, this testing exposed a number of minor errors in transliteration and one significant algorithmic error.

RESULTS FOR CLASSICAL IMPLEMENTATIONS

Linear List

In linear list implementations of priority queues, for any n item queue, an $O(n)$ sequential search is required either when an item is enqueued (in a sorted list) or dequeued (from an unsorted list). For sorted list implementations, the average performance depends on the relation between the distribution of priorities in the queue and the distribution of priorities of newly enqueued items. For example, if the linear search is done from the head of the queue, distributions leading to last-in–first-out access patterns can be done in constant time (as discussed in more detail in [19]).

The version of the linear list implementation tested here uses a singly linked list with searching from the head at insertion time. This minimizes storage requirements for the queue, since only one pointer is needed per item, and favors last-in–first-out behavior. An additional pointer per item would be required to support arbitrary deletion or searching from the tail. The results of the linear list implementation for the five priority distributions tested—exponential, uniform, biased, bimodal, and triangular—are given in Figure 1. When these data are compared with the results for other implementations, it is clear that no other implementation is better than linear list for queues of fewer than about 10 items.

Implicit Heaps

Implicit heaps, as used in heapsort [2, 18, 25], are the oldest priority-queue implementation with $O(\log n)$ performance. In a heap, the priority structure is represented as a binary tree that obeys the heap invariant, which holds that each item always has a higher priority than its children. In an implicit heap, the tree is embedded in an array, using the rule that location 1 is the root of the tree, and that locations $2i$ and $2i + 1$ are the children of location i.

The enqueue operation on an implicit heap involves a search from the leaf at the upper bound of the heap up toward the root for a place to put the new item; each item passed along the way is demoted to make space for the new item. The dequeue operation returns the root and then promotes items while searching down from the root for a place to put the most distant leaf. An efficient hold operation is easily implemented with implicit heaps; the im-

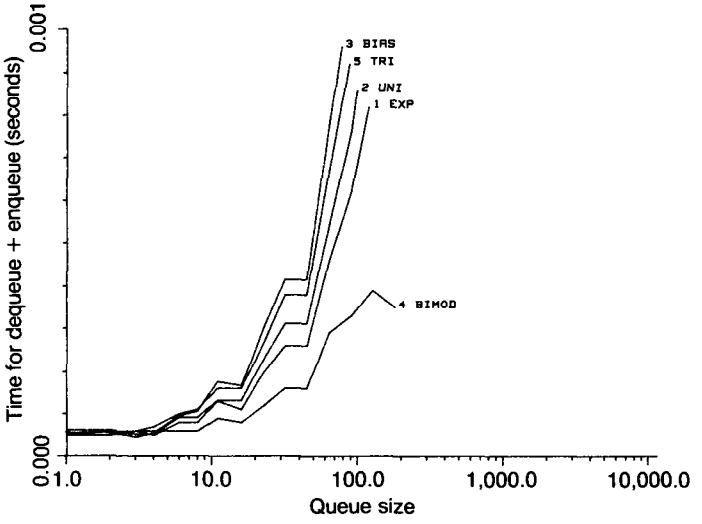

FIGURE 1. Linked List Data from the VAX 11/780

provement resulting from its use is reported to be about 30 percent [7, 19].

When items being enqueued contain data associated with their priority, there are three alternative implicit-heap implementations: Heap entries can contain entire items; heap entries can be limited to a priority and a pointer to the remainder of the item; or heap entries can consist only of a pointer, with even the priority moved out of the heap. Since the third alternative proved marginally faster on the VAX 11/780, it was used in all measurements reported here.

The results obtained using implicit heaps for the five priority distributions are given in Figure 2. Note that a semilog coordinate system is used for all performance data: Thus, the performance data for $O(\log n)$ queue implementations should appear as a straight line. The consistent departure of the VAX 11/780 data from a straight line is the result of the cache effects discussed under "Measurement Methodology" on page 300; no such departure was observed on the HP 9836 or the Prime 850. The most notable characteristic of implicit heaps is that their performance under the hold model is essentially independent of the priority distribution. Implicit heaps are also quite fast, although many of the more recently developed queue implementations are faster. In testing implicit heaps on the HP 9836, a disturbing problem arose: Queues larger than 8000 items could not be tested because the compiler imposes a limit of 32 kbytes on statically allocated contiguous data structures! The existence of such limits serves as an important motivation for the use of pointers instead of contiguous allocation in software that is intended to be portable.

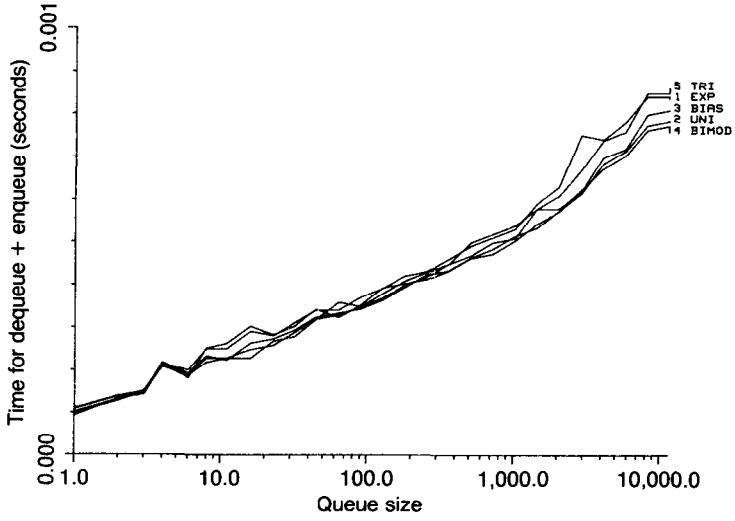

FIGURE 2. Implicit Heap Data from the VAX 11/780

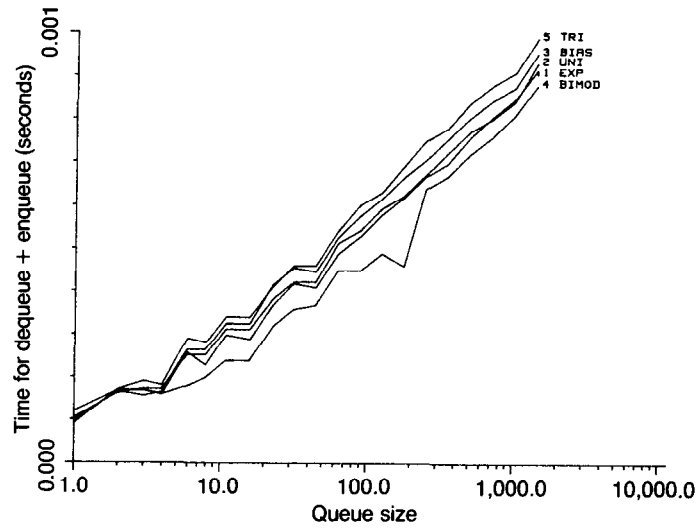

FIGURE 3. Leftist Tree Data from the VAX 11/780

Leftist Trees

Leftist trees [18] also use the heap structure, but in this case the heap is explicitly represented with pointers from parents to their children. Both the enqueue and dequeue operations on leftist trees are done using the merge operation. For enqueue, the new item is initialized as a one-node tree before being merged with the original tree. For dequeue, the root of the original tree is returned, and the resulting subtrees are merged. Two leftist trees are merged by merging their rightmost branches. The balance of the tree is maintained by recording, for each item, the distance to the nearest leaf, and always sorting the two children of an item so that the path to the nearest leaf is through the right child. As a result, although leftist trees require an extra field in each item, they guarantee an $O(\log n)$ bound on the time taken for the merge operation.

Experimentally, leftist trees were consistently about 30 percent slower than implicit heaps on all machines tested. The results obtained using leftist trees are given in Figure 3. Aside from a change in scale, these results are very similar to those for implicit heaps as given in Figure 2.

SIMULATION EVENT-SET IMPLEMENTATIONS

The special demands of discrete event simulation have sparked the development of a number of special-purpose priority-queue implementations for the pending event set. Many simulation formalisms require a stable event set so that events scheduled to happen at the same time can be processed in first-

in–first-out order, and none of the heap-based queue implementations can guarantee such stable behavior without the addition of auxiliary keys. The emphasis on stable behavior in the simulation event set appears to have originated with Gordon's early work leading to the development of the GPSS language; in [9], the reason for this emphasis is explained, and it is noted that, in most simulations, a random ordering of items with equal priority is sufficient. Implicit heaps, leftist trees, and related priority-queue implementations should all satisfy this criterion.

Two List

The two-list event-set implementation [3] operates by dividing the set of items in the queue into two lists: a short sorted list of items near the head of the queue, and a long unsorted list of more distant events. When a new item is enqueued, it is compared with a threshold priority to determine in which list it should be placed. When an item is dequeued, it will normally be removed from the sorted list, but occasionally, the sorted list will be empty, in which case the threshold must be advanced and all items in the unsorted list ahead of the new threshold must be moved to the sorted list. The threshold advance mechanism is adaptive; it tries to keep the average length of the sorted list near $n^{0.5}$. The result is an average enqueue time of $O(n^{0.5})$; when the sorted list must be rebuilt, the worst-case dequeue time becomes $O(n)$, but the average remains $O(n^{0.5})$ because most of the dequeues are done in $O(1)$ time.

Using the hold model with the two-list event-set implementation for the five priority distributions produces the results shown in Figure 4. Note that two-list implementation works very poorly for distribution 4, the bimodal distribution. The reason for this is that, for queue sizes greater than 200, a significant fraction of the high traffic, short lifetime items are enqueued in the unsorted list, which necessitates frequent rebuilding of the sorted list. As long as such distributions are not involved, however, the performance of the two-list implementation is considerably better than linear list and is among the better implementations for queues of up to a few hundred items.

Henriksen's

Henriksen's event-set implementation [10, 11, 15] uses a simple linear list, but maintains an auxiliary array of pointers into the list. This array allows a binary search to be used to find the range of entries in the list where a newly enqueued item should be placed. The binary search takes $O(\log n)$ time, but there is also a significant cost associated with maintaining the auxiliary array and searching the subsection of the main list pointed to by an array entry. Kingston has shown that the average performance of Henriksen's implementation is bounded by $O(n^{0.5})$ [15], and that, although single operations may take $O(n)$ time, the time per operation is bounded by $O(n^{0.5})$ if amortized over sufficiently many operations [17]. Henriksen's implementation, compared with other event-set implementations in [10], [15], and [19], fared well in each study.

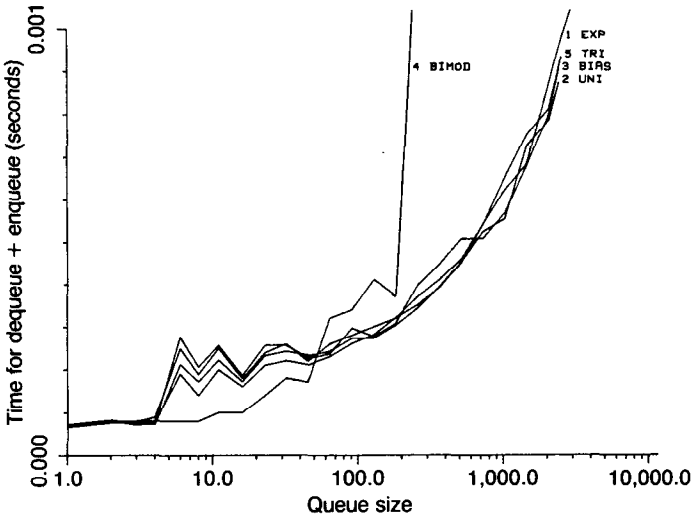

FIGURE 4. Two-List Data from the VAX 11/780

The code used here to test Henriksen's implementation was a modification of the Pascal code given in Figure 16 of [15]: The results of these tests for the five priority distributions are given in Figure 5. Un-

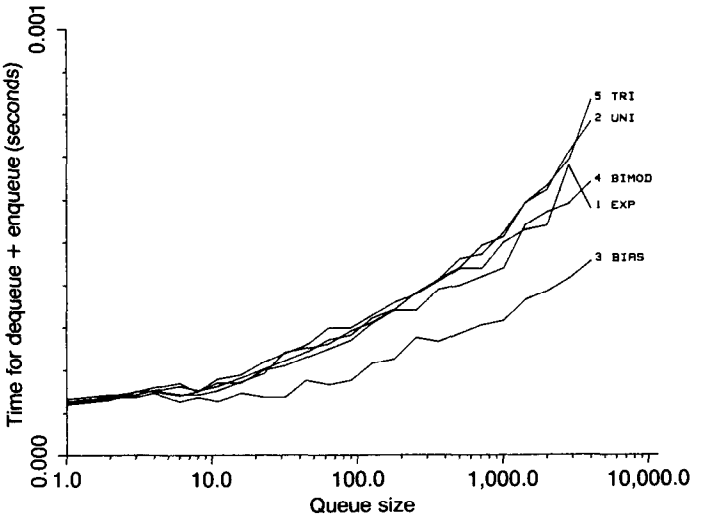

FIGURE 5. Henriksen's Implementation Data from the VAX 11/780

like the heap-based priority-queue implementations, the performance of Henriksen's implementation varies by almost a factor of 2 depending on the priority distribution, as was the case in [10] and [15]. The relative speeds measured for Henriksen's implementation and implicit heaps are also consistent with those reported in [19]. For queues of 4000 elements (the largest measured), Henriksen's worst case is comparable to the implicit heap, and the best case is considerably better. Since Henriksen's implementation relies on a contiguous array to store the auxiliary index (as do implicit heaps), compiler-imposed limits on the size of contiguously allocated data structures may interfere with the use of Henriksen's implementation for very large queues—a problem that can be overcome (at some expense) by replacing the array with a binary tree, as described in [11].

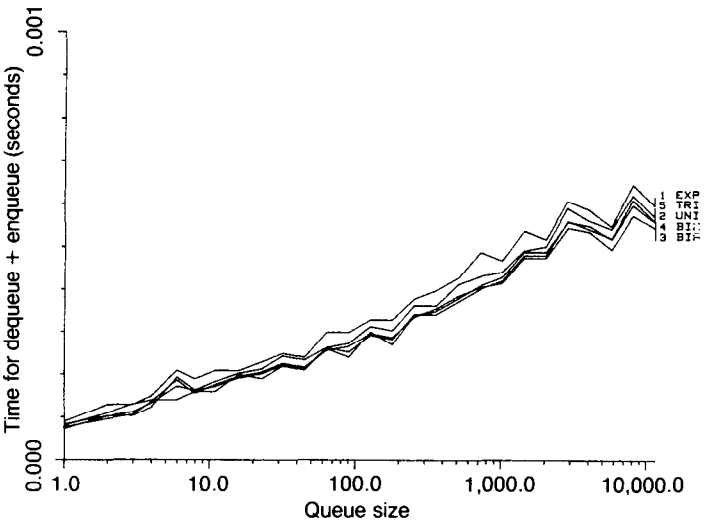

FIGURE 6. Binomial Queue Data from the VAX 11/780

NEARLY OPTIMAL IMPLEMENTATIONS

Binomial Queues

Binomial queues, which were developed by Jean Vuillemin in the mid 1970s [5, 24], have been characterized as a practical and nearly optimal priority-queue implementation [4]. A binomial queue is represented as a forest of heap-ordered trees where the number of elements in each tree is an exact power of two. The $O(\log n)$ bound on the time taken by operations on a binomial queue follows from the fact that any number n can be represented as the sum of $\log_2 n$ or fewer distinct powers of two. Each of the trees in the forest making up a binomial queue is a binomial tree, which is to say that, if there are 2^n items in the tree, the children of the root are binomial trees with sizes 2^{n-1}, 2^{n-2}, 2^{n-3}, and so on.

The times taken for operations on binomial queues are not very sensitive to the priority distribution, but they vary considerably with relatively small changes in queue size. This is because the time for an operation on a binomial queue depends in part on the number of 1 bits in the binary representation of the queue size. This is minimized for sizes of the form 2^n, and maximized for sizes of the form $2^n - 1$.

The code tested here was a Pascal transliteration of the SAIL code for the structure R variant of binomial queues as taken from Appendix 1.1 of [4]. Of all the priority-queue implementations tested, the binomial queue is clearly the most complex, although, as can be seen in Figure 6, it performs extremely well. Only three other implementations ever performed significantly better: Henriksen's, splay trees, and pairing heaps. These results are not consistent with the comparison of implicit heaps and binomial queues presented in [4], where implicit heaps were found to be about 20 percent faster than binomial queues. This difference is probably due to the fact that the items enqueued in the tests reported in [4] consisted of simple priority values with no auxiliary fields.

Pagodas

Pagodas, which were developed at about the same time as binomial queues [6, 20], are, like leftist trees, based on heap ordered binary trees. However, unlike leftist trees, in a pagoda the primary pointers lead from the leaves of the tree toward the root. A secondary pointer in each item of the tree points downwards from that item to its otherwise unreachable left- or rightmost descendant. The root, having no upward pointer, has pointers to both its left- and rightmost descendants. As a result, all items in a pagoda are reachable from the root by somewhat complex paths, and all branches of the structure are circularly linked.

The enqueue and dequeue operations on pagodas are based on merging the right branch of one pagoda with the left branch of another. Unlike leftist trees, the merge operation is performed bottom up. As a result, random insertions can be performed in constant time, assuming there is a uniform distribution of values over the set of values already in the pagoda. Unlike leftist trees and binomial queues, no effort is made to maintain the balance of a pagoda; therefore, although the average time for operations on pagodas is $O(\log n)$, there are infinite sequences of operations that take $O(n)$ time per operation. Deletion of an arbitrary item from a pagoda requires

finding the two pointers to that item, which can be done in an average time of $O(\log n)$ because all branches are circularly linked.

The code tested in this study was a Pascal transliteration of the PDP-10 assembly code implementation of pagodas that is given in the appendix of [20]. Although the code is moderately large, it performs very well: consistently faster than implicit heaps, as fast as binomial queues on the VAX, and only slightly slower on the HP 9836 and the Prime 850. This performance is consistent with the comparison of binomial queues and pagodas presented in [20]. The results obtained using pagodas are shown in Figure 7; except for a change of scale, the results are quite similar to those shown for implicit heaps in Figure 2.

Skew Heaps

The skew heap priority-queue implementation developed by Sleator and Tarjan in 1982 [21, 22] represents an important advance over its predecessors because it does not rely on any mechanism that limits the cost of individual operations. As a result, individual enqueues or dequeues may take $O(n)$ time. On the other hand, skew heaps guarantee that the cost per operation will never exceed $O(\log n)$ if the cost is amortized over a sufficiently long sequence of operations (or over any sequence starting from an empty queue). The basic operations on a skew heap are very similar to those on a leftist tree, except that no record of the path length to the nearest leaf is maintained with each item; instead, the children of each item visited on the merge path are simply exchanged, thereby effectively randomizing the tree structure.

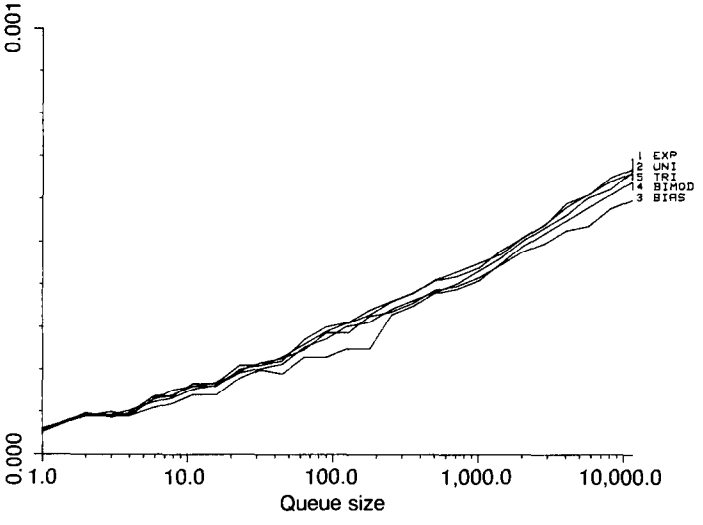

FIGURE 7. Pagoda Data from the VAX 11/780

In [22], two major variants of skew heaps are proposed: the bottom-up variant and the top-down variant. The top-down variant uses a simple binary tree analogous to that used with leftist trees. This variant is the second simplest of the priority-queue implementations we tested; only the linked list implementation was simpler. In [21] and [22], a triangularly linked data structure for the top-down variant is proposed that allows arbitrary items to be deleted without using any additional storage for back pointers. This top-down variant is the first of the fast priority-queue implementations we examined that can be modified to allow concurrent manipulation of a priority queue by multiple processes [14].

The bottom-up variant of skew heaps is similar in many ways to pagodas: The pointers along the skeleton of the tree structure point from leaves toward the root, and each item has a second pointer pointing down to an otherwise unreachable item. This downward pointer always points to the rightmost descendant of the left child, and the root contains pointers to the rightmost descendants of both children, which allows bottom-up merging of skew heaps. As with pagodas, this data structure is cyclic, allowing arbitrary items to be deleted in $O(\log n)$ time.

Both the top-down and bottom-up skew heap variants were implemented and tested in this study. After the bottom-up version was repaired so that it handled equal keys correctly, both the top-down and bottom-up skew heaps performed better than implicit heaps for all five priority distributions. The bottom-up version was the faster of the two on the VAX 11/780 and the HP 9836, and as fast as pagodas and binomial queues on the Prime 850 and the VAX 11/780. As with the other implementations based on heap-ordered binary trees, skew heaps are relatively insensitive to changes in the priority distribution. The results obtained using the top-down and bottom-up skew heaps are given in Figures 8 and 9 (on the next page), respectively.

Splay Trees

Splay trees are a relatively new form of binary search tree [21, 23]. As shown in [15] and [16], simple binary search trees have $O(n^{0.5})$ performance when used as priority queues—a poor performance resulting when the tree becomes systematically unbalanced by the repeated deletion of the leftmost item in the tree. Clearly, balanced trees can solve this problem, but tree balancing entails a not insignificant cost. In [13], a comparison between AVL

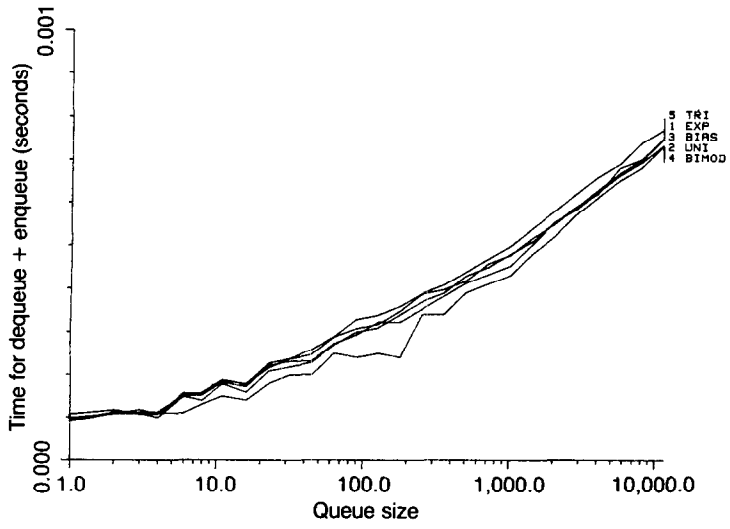

FIGURE 8. Top-Down Skew Heap Data from the VAX 11/780

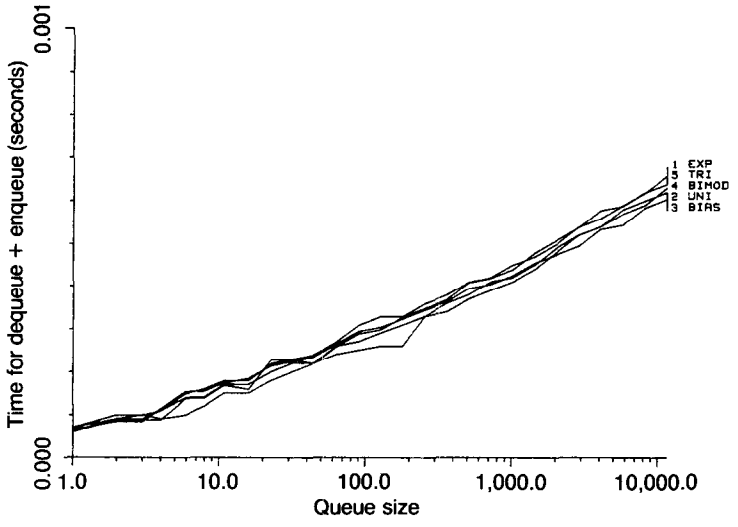

FIGURE 9. Bottom-Up Skew Heap Data from the VAX 11/780

trees [18] and implicit heaps under the hold model for a negative exponential distribution revealed that, although AVL trees required fewer comparisons, they ran only slightly faster than implicit heaps. Splay trees avoid some of the costs associated with tree balancing by blindly performing pointer rotations (the basic balancing operation) to shorten each path followed in the tree. This avoids the necessity of maintaining or testing records of the balance of the tree, but it also increases the number of rotations performed.

As with top-down skew heaps, it should be possible to construct a version of splay trees that allows concurrent access to the priority queue. Unlike the other priority-queue implementations discussed in this section, splay trees are stable in the sense that items with equal priority can be handled in first-in-first-out order.

Simple presentations of the basic operations on splay trees suggest that there will be one pointer rotation and one comparison per item visited during each search of the tree. In fact, no priority comparisons are needed to find and delete the leftmost item in a tree, and half of the pointer rotations serve only to bring the target of the search to the root, an operation that can be performed directly. The code tested for this implementation was based on the top-down code given in Section 4 of [21], but modified considerably to eliminate redundant operations and to specialize it for the enqueue and dequeue operations. The resulting implementation was quite fast, as shown in Figure 10.

As with Henriksen's event-set implementation, splay trees perform exceptionally well for distribution 3, the biased distribution. In fact, splay trees were as fast as Henriksen's implementation on the HP 9836, and even faster on the Prime 850 and VAX 11/780. This excellent performance reinforces the conjecture (made in [23]) that splay trees are, in a sense, optimal. Even with parent links added to allow for arbitrary deletion, splay trees are still slightly faster than Henriksen's implementation on the VAX 11/780, but slightly slower on the HP 9836 or the Prime 850.

Pairing Heaps

Pairing heaps have recently been developed as yet another priority-queue implementation with an

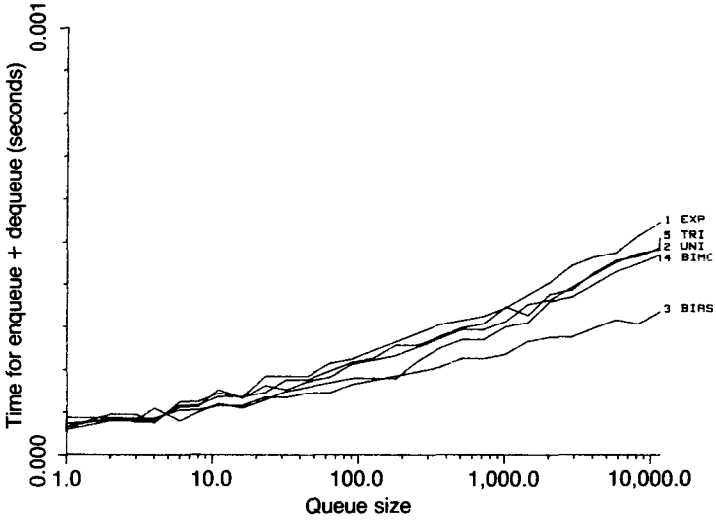

FIGURE 10. Splay Tree Data from the VAX 11/780

$O(\log n)$ amortized performance bound [8]. Pairing heaps correspond to binomial queues in the same way that skew heaps correspond to leftist trees. A pairing heap is represented as a heap-ordered tree where the enqueue operation can be executed in constant time either by making the new item the root of the tree, or by adding the new item as an additional child of the root. The dequeue operation returns the root of the tree and searches the list of children for the item that will become the new root.

The key to the efficiency of pairing heaps is the way in which the new root is found and the heap reorganized as a result of the dequeue operation. This is done by linking successive children of the old root in pairs and then linking each of the pairs to the last pair produced. The link operation combines two pairing heaps by adding the heap with the lower priority root as a new child of the heap with the higher priority root. If the pairing heap is modified to allow arbitrary deletion, items can be promoted in $O(1)$ time by removing the subtree headed by the item from the heap, raising the item's priority, and linking it to the root.

In [8], three pairing heap variants are discussed that differ in the way the list of pairs of children of the old root is linked to find the new root. The original variant involves a back-to-front linking pass over the list of pairs: The alternatives are a front-to-back linking pass, and a multipass scheme where pairs of elements of the list of pairs are repeatedly linked until only one remains. When all three variants were implemented and tested on the VAX 11/780 with distribution 1, the exponential distribution, the original, and the front-to-back variants were found to be best, whereas the multipass variant was significantly slower, possibly because of the bookkeeping expense involved in making multiple passes.

Further testing was done only on the original pairing heap variant. As can be seen in Figure 11, this variant ran at essentially the same speed as the bottom-up skew heap on all machines; as with Henriksen's implementation and splay trees, it ran especially well for distribution 3, the biased distribution.

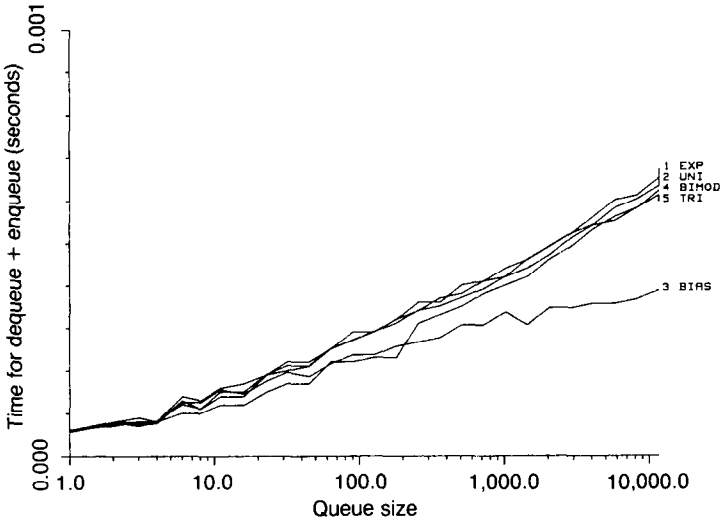

FIGURE 11. Pairing Heap Data from the VAX 11/780

SUMMARY

Figures 12–14 (pp. 310–311) show the relative performance of each of the tested queue implementations on the VAX 11/780, the HP 9836, and the Prime 850, respectively, using distribution 1, the exponential distribution. In interpreting this data, it is useful to keep in mind that, while the exponential distribution is the worst case for some implementations, such as splay trees and binomial queues, it is closer to the average case for others, such as Henriksen's.

Among our conclusions (which are summarized in Table II), we find that simple linked lists are the best

TABLE II. Summary of Conclusions

Priority-queue implementation	Code size[a]	Performance		Relative speed[b]	Comments
		Average	Worst		
Linked list	47	$O(n)$	$O(n)$	11	Best for $n < 10$
Implicit heap	72	$O(\log n)$	$O(\log n)$	8	
Leftist tree	79	$O(\log n)$	$O(\log n)$	9–10	
Two list	104	$O(n^{0.5})$	$O(n)$	9–10	Good for $n < 200$
Henriksen's	68	$O(n^{0.5})$	$O(n^{0.5})$[c]	1–7	Stable
Binomial queue	188	$O(\log n)$	$O(\log n)$	1–7	
Pagoda	110	$O(\log n)$	$O(n)$	4–8	Delete in $O(\log n)$
Skew heap, top down	56	$O(\log n)$	$O(\log n)$[c]	5–7	
Skew heap, bottom up	103	$O(\log n)$	$O(\log n)$[c]	4–6	Delete in $O(\log n)$
Splay tree	119	$O(\log n)$	$O(\log n)$[c]	1–3	Stable
Pairing heap	84	$O(\log n)$	$O(\log n)$[c]	3–6	Promote in $O(1)$

[a] The total lines of Pascal code for initqueue, emptyqueue, enqueue, and dequeue.
[b] 1 is fastest; 11 is slowest: The rankings are based on Figures 12–14.
[c] An amortized bound; single operations may take $O(n)$ time!

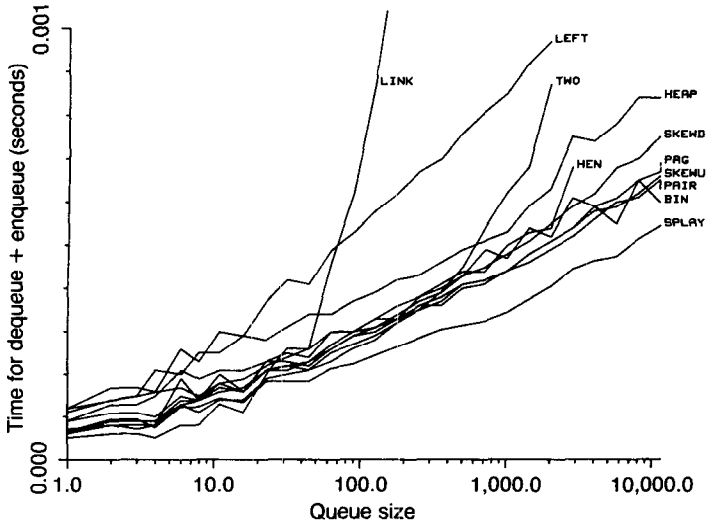

FIGURE 12. VAX 11/780 Data for the Exponential Distribution

priority-queue implementation for fewer than 10 items, although they perform very poorly for queues of more than 50 items. The two-list implementation may perform well for queues of up to a few hundred items, but for some priority distributions, it is as bad as a simple linked list. Although leftist trees have an $O(\log n)$ performance bound, they never perform well enough to warrant consideration in any application. Henriksen's implementation, although it has an $O(n^{0.5})$ performance bound, performs acceptably for all queue sizes tested; only splay trees challenge it in applications where stable behavior is required.

Implicit heaps are among the worst choices for queues smaller than 20 elements—and consistently worse than many other priority-queue implementations—but they can be modified to improve their performance under strictly alternating enqueues and dequeues. However, unless such alternation dominates an application or contiguous storage allocation has special advantages, implicit heaps are not a particularly good choice.

Binomial queues perform erratically and require the most complex code of all the queue implementations examined. Furthermore, splay trees perform at least as well as binomial queues on all machines tested. Skew heaps, pairing heaps, and pagodas all perform almost as well; of these, the top-down variant of skew heaps is one of the simplest. Although these implementations all have $O(\log n)$ average running times, they have different worst cases: Pagodas have an $O(n)$ worst-case bound; skew heaps, splay trees, and pairing heaps have $O(n)$ bounds for single operations, but $O(\log n)$ amortized bounds; and binomial queues have an $O(\log n)$ bound for single operations.

If other priority-queue operations such as arbitrary deletion or priority changes are needed, bottom-up skew heaps, splay trees, and pairing heaps emerge as the best alternatives. All priority-queue operations on pagodas and bottom-up skew heaps can be done in $O(\log n)$ average time with no performance penalty because of circularly linked branches. Although the data structures used for splay trees and pairing heaps must be modified to allow arbitrary deletion (with some performance penalty), splay trees still compete well, and pairing heaps may be acceptable because of the possibility of promoting arbitrary items in constant time.

The empirical results presented here should serve not only as a practical guide for priority-queue users, but also as an introductory survey for the more general audience. It is hoped that the necessary update and amalgamation of previous comparisons provided by this work will stimulate additional work on algorithms for both priority queues and other applications.

Acknowledgments. I would like to thank George Singer for the preliminary work he did measuring the performance of pagodas, and Robert Tarjan for bringing skew heaps, pairing heaps, and splay trees to my attention. I also thank Hewlett Packard for the loan of an HP 9020 workstation used for preliminary performance studies.

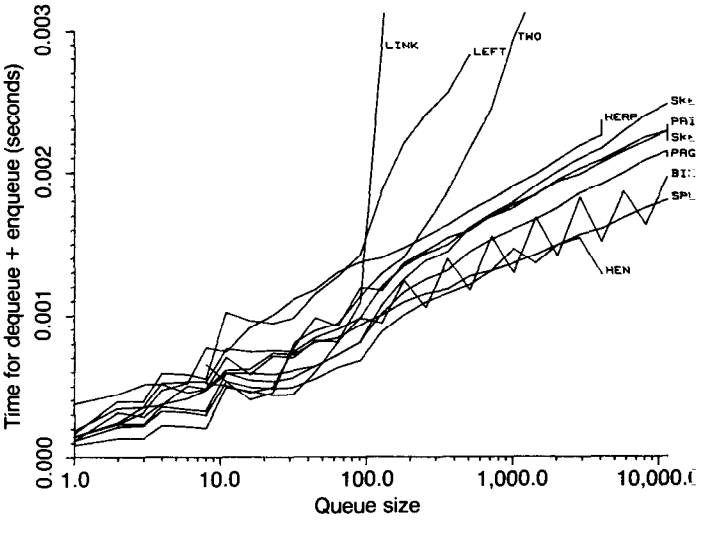

FIGURE 13. HP 9836 Data for the Exponential Distribution

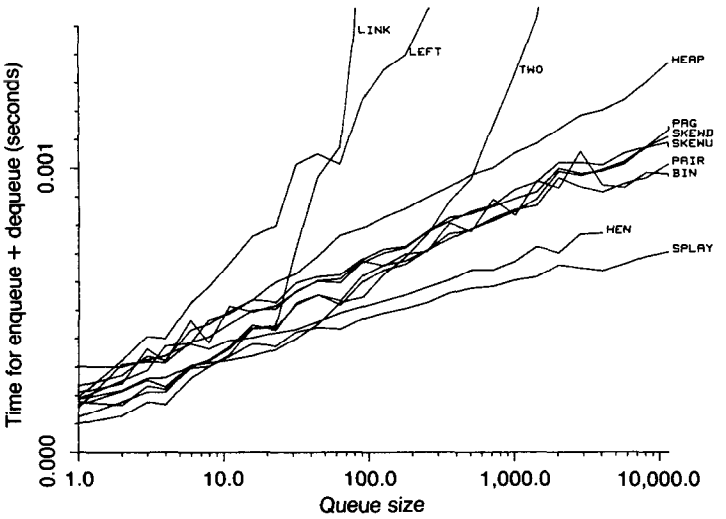

FIGURE 14. Prime 850 Data for the Exponential Distribution

REFERENCES

1. Aho, A.V., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974. Sections 4.9–4.11. A discussion of applications of 2–3 trees to priority queues.
2. Bentley, J. Programming pearls: Thanks, heaps. *Commun. ACM 28,* 3 (Mar. 1985), 245–250. Clear introduction to implicit heaps for sorting and priority queues.
3. Blackstone, J.H., Hogg, G.L., and Phillips, D.T. A two-list synchronization procedure for discrete event simulation. *Commun. ACM 24,* 12 (Dec. 1981), 825–829. A priority-queue implementation is described and recommended as the best choice for general use. The recommendation is disputed in [11].
4. Brown, M.R. The analysis of a practical and nearly optimal priority queue. Comput. Sci. Rep. STAN-CS-77-600, Dept. of Computer Science, Stanford Univ., Calif., Mar. 1977. This is the same as [5], but with an appendix containing working SAIL code.
5. Brown, M.R. Implementation and analysis of binomial queue algorithms. *SIAM J. Comput. 7,* 3 (Aug. 1978), 298–319. An assortment of implementations of binomial queues is presented along with a detailed performance analysis and empirical comparisons with leftist trees and heaps.
6. Francon, J., Viennot, G., and Vuillemin, J. Description and analysis of an efficient priority queue representation. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science* (Ann Arbor, Mich., Oct. 16–18). IEEE, Piscataway, N.J., 1978, pp. 1–7. (Also published as IRIA Rapport de Recherche No 287.) The original presentation of pagodas.
7. Franta, W.R., and Maly, K. A comparison of heaps and the TL structure for the simulation event set. *Commun. ACM 21,* 10 (Oct. 1978), 873–875. Claim, based on empirical data, that the TL implementation runs in $O(1)$ time, later disputed by [11] and [19]; convincing data presented comparing three heap implementations under the hold model.
8. Fredman, M.L., Sedgewick, R., Sleator, D., and Tarjan, R. The pairing heap: A new form of self-adjusting heap. Submitted for publication. Pairing heaps and their variants are described and analyzed.
9. Gordon, G. The development of the general-purpose simulation system (GPSS). *ACM Hist. Program. Lang. Conf., SIGPLAN Not. 13,* 8 (Aug. 1978), 183–198. (Edited by R.L. Wexelblat and republished in the ACM Monograph Series by Academic Press, New York, 1981, pp. 403–426.) Sections 3.5 and 4.6. The history of the requirement for a stable event set in discrete event simulation is presented (all in terms of linear lists).
10. Henriksen, J.O. An improved events list algorithm. In *Proceedings of the 1977 Winter Simulation Conference* (Gaithersburg, Md., Dec. 5–7). IEEE, Piscataway, N.J., 1977, pp. 547–557. A new implementation is described, and empirical measurements are presented.
11. Henriksen, J.O. Event list management—A tutorial. In *Proceedings of the 1983 Winter Simulation Conference* (Arlington, Va., Dec. 12–14). IEEE, Piscataway, N.J., 1983, pp. 543–551. A tutorial introduction to the event set, the hold model, and Henriksen's implementation; attacks claims made in [3] and [7].
12. Jonassen, A. Priority queue processes with biased insertion of exponentially distributed input keys. Rep. 14, Univ. of Oslo Institute of Informatics, Norway, May 1977, ISBN 82-90230-02-8. Clear description of the hold model. Proof that the exponential distribution results in a uniform insertion distribution in the steady state.
13. Jonassen, A., and Dahl, O.J. Analysis of an algorithm for priority queue administration. *BIT 15,* 4 (1975), 409–422. Presents the p-tree algorithm, an analysis (later disputed in [15]), and comparisons under the hold model with implicit heaps, leftist trees, AVL-trees, and linear lists.
14. Jones, D.W. Concurrent operations on priority queues. Submitted for publication. Skew heaps allow priority-queue operations in $O(1)$ time given $O(\log n)$ processors in a shared-memory environment.
15. Kingston, J.H. Analysis of algorithms for the simulation event list. Ph.D. thesis, Basser Dept. of Computer Science, Univ. of Sydney, Australia, July 1984. Average case analysis of binary trees, p-trees, and Henriksen's under the hold model.
16. Kingston, J.H. Analysis of tree algorithms for the simulation event list. *Acta Inf. 22,* 1 (Apr. 1985), 15–33. Partial presentation of material from [15].
17. Kingston, J.H. The amortized complexity of Henriksen's algorithm. Comput. Sci. Tech. Rep. 85-06, Dept. of Computer Science, Univ. of Iowa, Iowa City, July 1985. Proof of an $O(n^{1/2})$ amortized bound.
18. Knuth, D.E. *The Art of Computer Programming.* Vol. 3, *Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973. Section 5.2.3 discusses heapsort and leftist trees; section 6.2.3 discusses AVL trees.
19. McCormack, W.M., and Sargent, R.G. Analysis of future event-set algorithms for discrete event simulation. *Commun. ACM 24,* 12 (Dec. 1981), 801–812. Empirical comparisons, under the hold model and in discrete event simulation models, of implicit heaps, Henriksen's, and older implementations.
20. Nix, R. An evaluation of pagodas. Res. Rep. 164, Dept. of Computer Science, Yale Univ., New Haven, Conn., no date. Analysis and empirical comparisons with binomial queues, heaps, and leftist trees. Well-commented PDP-10 assembly language code is given.
21. Sleator, D.D., and Tarjan, R.E. Self-adjusting binary trees. In *Proceedings of the ACM SIGACT Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, 1983, pp. 235–245. Original presentation and amortized complexity analysis of skew heaps and splay trees.
22. Sleator, D.D., and Tarjan, R.E. Self adjusting heaps. *SIAM J. Comput.* To be published. Expansion of the material on skew heaps from [21].
23. Tarjan, R.E., and Sleator, D.D. Self-adjusting binary search trees. *J. ACM 32,* 3 (July 1985), 652–686. Expansion of the material on splay trees from [21].
24. Vuillemin, J. A data structure for manipulating priority queues. *Commun. ACM 21,* 4 (Apr. 1978), 309–315. Original presentation of binomial queues; code presented is very hard to follow.
25. Williams, J.W.J. Algorithm 232: Heapsort. *Commun. ACM 7,* 6 (June 1964), 347–348. Original presentation of implicit heaps.

CR Categories and Subject Descriptors: E.1 [**Data Structures**]: *trees*; E.2 [**Data Storage Representations**]: *linked representations*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*; I.6.1 [**Simulation and Modeling**]: Simulation Theory—*types of simulation (discrete)*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: binomial queue, event set, heap, leftist tree, pagoda, pairing heap, priority queue, sequencing set, skew heap, splay tree

Received 6/85; accepted 11/85

Author's Present Address: Douglas W. Jones, Dept. of Computer Science, Univ. of Iowa, Iowa City, IA 52242; CSNet address: jones@uiowa.