Algorithms and Data Structures

Daniel Sleator Editor

Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem

RANDY BROWN

ABSTRACT: A new priority queue implementation for the future event set problem is described in this article. The new implementation is shown experimentally to be O(1) in queue size for the priority increment distributions recently considered by Jones in his review article. It displays hold times three times shorter than splay trees for a queue size of 10,000 events. The new implementation, called a calendar queue, is a very simple structure of the multiple list variety using a novel solution to the overflow problem.

1. INTRODUCTION

A priority queue is a queue for which each element has an associated priority, and for which the dequeue operation always removes the lowest (or highest) priority item remaining in the queue. An important application of priority queues is representation of the pending event set in discrete event simulation [1, 8, 10, 11].

Priority queue implementations can be classified according to the manner in which execution time varies with queue size. Let a hold operation refer to a dequeue followed immediately by an enqueue. This operation leaves the queue size unchanged. For a simple linear list implementation, the time for a hold operation is proportional to the queue size. The two list method reduces the queue size dependence of the hold time to $O(n^{0.5})$, where n is the queue size [1]. Binomial queues, pagodas, skew heaps, pairing heaps, and splay trees run with $O(\log n)$ time per hold operation [2, 4, 7, 11–15, 17].

At least two priority queue implementations for the special case of the event set problem have been proposed which were claimed to have a hold time of $O(1)$ in the queue size. In 1977 Franta and Maly proposed a two level pointer structure which has $O(n^{.5})$ worst case behavior but which their experimental data showed to have nearly $O(1)$ average performance [5, 6]. While comparing it to other event set implementations, however, McCormack and Sargent found it did not work nearly as well as originally indicated [11]. They suggested that this might be due to large overhead, or to an error in the original code.

In 1980 Davey and Vaucher proposed a partitioned list structure which they proved to be $O(1)$ except for the problem of an overflow list. This structure also was tested by McCormack and Sargent and found to work no better than a p-tree implementation, which is $O(\log n)$, for queues of modest size [11]. They pointed out that Davey and Vaucher's partitioned list is $O(n)$ if a fixed non-negligible percentage of events end up in the overflow list, as happens for some distributions. As with Franta and Maly's method, high overhead seems to be a problem. A complicated algorithm may be slow even though its execution time is $O(1)$ in queue size. Such an algorithm will be useful only for extremely large queues.

Another event set implementation worth mentioning is that proposed by Ulrich in 1978 [16]. He did not claim $O(1)$ performance for it, but analysis of his algorithm suggests that it should be $O(1)$ in average performance except for the presence of an overflow list.

The purpose of this article is to present a priority queue implementation for the event set problem which has relatively low overhead, and which on the basis of intuition and experimental evidence has $O(1)$ average performance. The proposed data structure is called a calendar queue. It is modeled after a desk calendar, which is the means by which a human being solves the problem of ordering a future event set. It bears strong similarities to Davey and Vaucher's partitioned list and to Ulrich's multiple list structure, but does not require an overflow list and is considerably simpler.

2. THE CALENDAR QUEUE

One schedules an event on a desk calendar by simply writing it on the appropriate page, one page for each day. There may be several events scheduled on a particular day, or there may be none. The time at which an event is scheduled is its priority. The enqueue operation corresponds to scheduling an event. The earliest event on the calendar is dequeued by scanning the page for today's date and removing the earliest event written on that page.

In the computer implementation, each page of the calendar is represented by a sorted linked list of the events scheduled for that day. An array containing one pointer for each day of the year is used to find the linked list for a particular day. If the array name is "bucket," then bucket[12] is a pointer to the list of events scheduled for the 12th day of the year. The complete calendar thus consists of an array of 365 pointers and a collection of up to 365 linked lists.

Events can be scheduled for up to one year from the present date by making the calendar circular. If today is December 5, then one can schedule an event for November 12 of next year by simply flipping back to November 12 on the present calendar and writing the event on that page. If events are erased from the calendar as they are dequeued, it will be clear that the event on November 12 is scheduled for next year. Upon dequeueing the last event from December 31 of this year, one moves back to January 1 and begins dequeueing events scheduled for next year. The same calendar can be used indefinitely. Notice that there is no need to move pointers or perform any other maintenance on the data structure as one moves from one year to the next.

Events can be scheduled more than a year in advance if one writes the date beside each event. Suppose the present date is June 5, 1987. If today's calendar page contains the entry "Open time capsule—June 5, 1988," it is clear that this event was scheduled more than a year in advance and that it should not be dequeued until next year. Before dequeueing an event from today's calendar page, one checks the date written beside it to make certain that it is scheduled for this year. If it is, one dequeues it and erases it; if not, it is ignored and left for next year or some year thereafter. This simple mechanism avoids the need for an overflow list and its associated problems. In the computer implementation, the time of each event is already recorded in its event node, and so no new record keeping is required. All that is necessary is to examine the date of each event before dequeueing it to make certain that it belongs to the current year. If not, it is simply skipped and left for a future traversal of the calendar.

The length of a year is chosen to be long enough that most events, say at least 75 percent, will be scheduled for no more than a year after the time at which they are enqueued; and the number of days in a year is chosen to be large enough that no one day contains too many events. The length of a year and the length of a day will both have to be adjusted periodically as the queue grows and shrinks. Figure 1 is an example of a calendar queue with an eight day year. The length of a year is 4.0 time units and the length of a day is 0.5 time units. The next event to be dequeued is at time 14.5. Notice that the event at time 19.1 in bucket 6 will be skipped the first time it is encountered and dequeued the following year. The year shown begins at time 12.0. If year 0 begins at time 0.0, year 1 begins at time 4.0, etc., then the current year is year 4.

	Bucket 0:	16.2			/* 12–12.5 */
	Bucket 1:	16.6			/* 12.5–13 */
	Bucket 2:				/* 13–13.5 */
	Bucket 3:	17.8			/* 13.5–14 */
	Bucket 4:				/* 14–14.5 */
→	Bucket 5:	14.5	14.7	14.8	/* 14.5–15 */
	Bucket 6:	15.2	15.3	19.1	/* 15–15.5 */
	Bucket 7:	15.9			/* 15.5–16 */

The current year begins at 12.0 and ends at 16.0. Bucket 5 is the current date. Note that events scheduled in buckets 0 through 4 are scheduled for next year and are in the range 16–20. 0.5 is the length of a day. The numbers on the right are the time ranges for each day in the current year. Note that the event at 19.1 in bucket 6 is scheduled for next year.

FIGURE 1. Eight Day Calendar Queue

3. ADJUSTMENT OF CALENDAR SIZE

If the number of events in the queue is much smaller or much larger than the number of buckets, it will not function efficiently. If each bucket contains 100 entries, the enqueue time will be excessive because of the time needed to locate the position in the sorted list at which to enter the new event. If only one bucket out of a hundred contains an entry, the dequeue time will be long since an average of a hundred empty buckets will have to be examined during each dequeue to find the next entry.

Consider an empty queue which gradually grows to contain 10,000 events. Since it will eventually hold 10,000 events, the number of buckets should be on the order of 10,000. While the queue is still small, say 10 events, only one bucket in 1000 will be occupied. This will make the queue extremely inefficient while it is small. The solution is to allow the number of buckets to correspondingly grow and shrink as the queue grows and shrinks. One can start with a small number of buckets and copy the queue onto another calendar whenever the number of events exceeds twice the number of buckets. Similarly, if the queue shrinks so that the number of events is less than half the number of buckets, it can be copied onto a smaller calendar. If the number of buckets is doubled when the queue size reaches twice the number of buckets, the new queue will have the same number of events as buckets. Similarly, halving the number of buckets when the queue size falls to half the number of buckets will result in the new calendar having the same number of buckets as events.

Consider a queue which has grown steadily from 0 to 1024 events, and suppose that the initial number of buckets was two. The events were copied onto a new calendar when the queue size reached 5, again when it reached 9, again at 17, 33, 65, 129, etc. The queue was recopied each time its size became one greater than a power of two. How many times has the average event been copied? Half of the events were not copied at all, since they were added after the last doubling of the number of buckets. Half of the remaining 512 were added between the doubling before last and the last one, and so have been copied once. Continuing in this manner one sees that $\frac{1}{4}N$ were copied once, $\frac{1}{8}N$ were copied twice, $\frac{1}{16}N$ were copied 3 times, etc., where $N = 1024$. Thus the average number of times an event was copied is

$$(1/4) + 2(1/8) + 3(1/16) + 4(1/32) + 5(1/64) + \cdots$$

The sum terminates when one reaches the original queue size. The finite sum is bounded by the infinite sum, whose value is 1.0. In this case the queue size is exactly a power of two. The worst case is when the queue size is one greater than a power of two. When the queue size reaches 1025, all of the events will be copied to another calendar and the average event will have been copied about twice. The number of times an average event has been copied will fluctuate between one and two as the queue grows, never becoming larger than two. The recopying time per event is $O(1)$ in the queue size.

Suppose that the queue size fluctuates, rising and falling either randomly or periodically. The worst case would be for the queue size to rise rapidly to just above a power of two, then fall rapidly to just below the next lower power of two, repeating this cycle indefinitely. Suppose for example that the queue size is fluctuating rapidly between 511 and 1025. The fastest rise and fall possible is with a string of 514 enqueues followed by 514 dequeues. The number of buckets will oscillate in this case between 512 and 1024. 1025 + 511 events will be copied each cycle, and the equivalent of 514 hold operations will be performed per cycle. This gives an average of not quite 3 events copied per hold operation. The same bound of 3 is obtained for oscillations between just below 2^m and just above 2^n for any m and n.

One way to allocate space for the array bucket[ ] is to allow it to oscillate between the top and bottom of a larger array. Suppose that bucket[ ] is a subarray of A[ ], which has 15,000 events, and that the current calendar has 64 buckets. If bucket[0] = A[0], then the next calendar will be at the top of A[ ]. When the queue is copied onto a 128 bucket calendar, bucket will be moved so that bucket[0] = A[15000 − 128]. Each time the queue is copied to a new calendar, bucket[ ] moves from the top to the bottom or from the bottom to the top of A[ ]. The original calendar will be at the bottom and the new calendar at the top, or vice versa. Since one calendar is always half the size of the other, the total space needed to contain both the original calendar and the new one is 1.5 times the space needed for the larger calendar. Moving one array within another is especially easy in C, since an array name in C is simply a pointer to the first element of the array, and pointer arithmetic is supported. The same effect can be obtained in other languages using subscript arithmetic. If space is at a premium, the old and new calendars can be stored in the same array, and mark fields can be used during copying to keep track of which entries have already been copied to the correct bucket for the new calendar.

The length of a day (bucket width) should be adjusted each time the queue is copied onto another calendar. If the bucket width is much greater than the average separation of adjacent queue elements, the queue elements will be clustered in a small number of buckets near the current date and the rest will be empty. If the bucket width is too small, most queue elements will not be in the current year. Enqueue and dequeue times are smallest when the bucket width is somewhere in the vicinity of the average separation of adjacent queue elements. Since the separation of adjacent queue elements generally decreases as more elements are added to the queue, the bucket width should be readjusted when the queue is copied to another calendar. Buckets containing events about to be dequeued normally contain the most queue elements, since they have not been emptied for a full year. An easy way to calculate the new bucket width is to dequeue a few elements, calculate the average separation of the dequeued events, and then put them back on the queue.

There is one last difficulty to be considered. It can occasionally happen that all of the priorities in the queue become tightly clustered around two or more points. For example, half of the priorities might be in the interval [0, 0.1] and the other half in the interval [5.6, 5.7]. If the queue contains 1000 events, 500 in each interval, the average separation between events within either interval will be about 0.0002. If the bucket width is 0.0002, then the length of a year will be 0.1024 (assuming 512 buckets). The distance between the two intervals is about 54 years, with 512 days per year.

The first 500 events could be dequeued quickly, but 54 years of empty buckets would have to be searched to find the first event in the interval [5.6, 5.7]. One would cycle through the calendar 54 times before finding an event in the current year. The remaining 500 events could then be dequeued quickly after the first event in the second set is found. The cost of searching one or two empty years would not be bad when amortized over 1000 elements, but 54 years is excessive.

A simple solution is to resort to a direct search for the minimum priority event in the queue whenever an entire year of empty buckets is encountered. Whenever the dequeue operation cycles through all of the buckets in the calendar without finding an event in the current year, it cycles through again looking for the lowest priority item in the queue. Since the events in each bucket are sorted, it suffices to examine the first event in each bucket. Normal operation is resumed at the priority and bucket of the lowest priority element found.

A side benefit of this strategy is that it provides protection against a poorly chosen initial bucket width. If the queue begins with two buckets, the average separation between queue elements will not be calculated until the queue size reaches 5. Until then the bucket width will have its initial value. If the initial bucket width is too large, then all of the queue elements will be in one bucket. This is not a serious problem with only 4 elements in the queue; if the bucket width is too small, however, the queue elements may be separated by hundreds or thousands of years. This would be a serious problem without the direct search mode, but a direct search is efficient when there are only two buckets and 4 elements in the queue. Consequently, it is quite safe to arbitrarily initialize the bucket width to 1.0.

Precautions should be taken to insure that the bucket width is set to separate points within a cluster rather than to separate the clusters when the distribution is severely bimodal. Recall that the bucket width is set using the average distance between adjacent elements in the queue. All that is necessary is to calculate an average, throw out any separation that is more than twice this initial average, and then calculate a new average using only the remaining separations. This will exclude the separation between the last point in one cluster and the first point in the next from the final average. Best results have been obtained using a bucket width of 3.0 to 4.0 times the average separation between queue elements. This puts an average of 3 or 4 elements in each bucket about to be dequeued.

4. ALGORITHMS

The enqueue operation can be implemented as follows (in C pseudocode):

```
enqueue(entry, priority)
double priority;
struct nodetype *entry;
/* This adds one entry to the queue. */
  {
  int i;
  /* Calculate the number of the bucket in which to
      place the new entry. */
  i = priority/width;          /* Find virtual bucket. */
  i = i % nbuckets;            /* Find actual bucket. */
  Insert entry into bucket i in sorted list;
  ++qsize;    /* Update record of queue size. */
  /* Double the calendar size if needed. */
  if (qsize > top_threshold) resize(2 * nbuckets);
  }
```

If the calendar length were infinite, the bucket containing an entry would be found by dividing its priority by the bucket width and rounding down to the nearest integer. The bucket found this way is called the virtual bucket. The actual bucket number is the virtual bucket number modulo the number of buckets in the calendar. Since the number of buckets is normally a power of two, the modulo operation can be performed by anding with a bit mask containing zeros in the high order bits. Top_threshold is normally twice the current number of buckets, and resize( ) is the procedure that copies the queue onto a new calendar.

The following procedure implements the dequeue operation:

```
struct nodetype *dequeue( )
/* This removes the lowest priority node from the
    queue and returns a pointer to the node containing
    it. */
{
register int i;
if (qsize == 0) return(NULL);
for (i = lastbucket; ; )   /* Search buckets */
  {  /* Check bucket i */
  if (bucket[i] != NULL && bucket[i] → prio
  < buckettop)
    {  /* Item to dequeue has been found. */
    Remove item from list;
    /* Update position on calendar. */
    lastbucket = i; lastprio = priority of item removed;
    --qsize;
    /* Halve calendar size if needed. */
    if (qsize < bot_threshold) resize(nbuckets/2);
    return item found;
    }
  else
    {  /* Prepare to check next bucket or else go to a
           direct search. */
    ++i; if(i == nbuckets) i = 0;
    buckettop += width;
    if(i == lastbucket) break; /* Go to direct search */
    }
  }
/* Directly search for minimum priority event. */
Find lowest priority by examining first event of each
bucket;
Set lastbucket, lastprio, and buckettop for this event;
return(dequeue( ));   /* Resume search at minnode. */
}
```

Three variables are used to keep track of the position in the calendar from which the last event was dequeued: lastbucket, buckettop, and lastprio. Lastbucket is simply the bucket number from which the last event was dequeued. Buckettop is the priority at the top of that bucket, i.e., it is the highest priority that could go into the bucket. Actually, it is one-half width greater than the top of the bucket. Lastprio is the priority of the last event dequeued. Lastbucket and lastprio are updated just before returning a dequeued event, and buckettop is updated every time the search advances to another bucket. Lastbucket is needed so that dequeue( ) will know where to begin the search. Buckettop is used to determine whether an entry is for the current year; *it is the top of the bucket for the current year. It is* 0.5 * width greater than the actual buckettop to guard against rounding error. Buckettop and width must be double precision to avoid fatal accumulation of roundoff error in the often repeated update operation:

```
buckettop += width;
```

Roundoff error will quickly cause buckettop to become less than the actual top of the bucket if this addition is done in 32 bit floating point. Other floating point variables in the program can be 32 bits long.

The main search loop is exited by a break command whenever *i* becomes equal to lastbucket. This only happens when an entire year of buckets have been searched without finding a queue element to dequeue. A direct search is then begun for the smallest element in the queue. When it is found, lastbucket, lastprio, and buckettop are reset at its position and the search is resumed by a recursive call to dequeue( ).

Each new calendar generated by resize( ) is initialized by a procedure called localinit( ).

```
localinit(qbase, nbuck, bwidth, startprio)
int qbase, nbuck;
double bwidth, startprio;
/* This initializes a bucket array within the array a[].
   Bucket width is set equal to bwidth. Bucket[0] is
   made equal to a[qbase]; and the number of buckets
   is nbuck. Startprio is the priority at which dequeue-
   ing begins. All external variables except resizeen-
   abled are initialized. */
{
int i;
long int n;
/* Set position and size of new queue. */
firstsub = qbase;
bucket = pointer to start of bucket[] in a[];
width = bwidth;
nbuckets = nbuck;
Calculate bit mask for modulo nbuckets operation;
/* Initialize as empty. */
qsize = 0;
for(i = 0; i < nbuckets; ++i) bucket[i] = NULL;
/* Set up initial position in queue. */
lastprio = startprio;
n = startprio/width;   /* Virtual bucket */
lastbucket = n % nbuckets;
buckettop = (n + 1) * width + 0.5 * width;
/* Set up queue size change thresholds. */
bot_threshold = nbuckets/2 - 2;
top_threshold = 2 * nbuckets;
}  /* end */
```

An empty queue is initialized by initqueue( ).

```
initqueue( )
/* This initializes an empty queue. */
  {
  localinit(0, 2, 1.0, 0.0);
  resizeenabled = TRUE;
  }
```

The first calendar is initialized at the bottom of a[ ] with two buckets. The initial bucket width is 1.0 and the starting priority is 0.0. This assumes that all priorities are positive. A large negative starting priority could be used otherwise. It would trigger an initial direct search at the first dequeue. Resizeenabled is used to disable copying to a new calendar when temporarily dequeueing elements to calculate the new bucket width.

Resize can be implemented as follows:

```
resize(newsize)
int newsize;
/* This copies the queue onto a calendar with newsize
   buckets. The new bucket array is on the opposite
   end of the array a[QSPACE] from the original. */
  {
  double bwidth;
  int i;
  int oldnbuckets;
  struct nodetype ** oldbucket;
  if (!resizeenabled) return;
  bwidth = newwidth( );   /* Find new bucket
  width. */
  /* Save location and size of old calendar for use
     when copying calendar. */
  oldbucket = bucket; oldnbuckets = nbuckets;
  /* Initialize new calendar. */
  if(firstsub == 0)
    localinit(QSPACE-newsize, newsize, bwidth,
    lastprio);
  else
    localinit(0, newsize, bwidth, lastprio);
  /* Copy queue elements to new calendar. */
  for (i = oldnbuckets - 1; i >= 0; --i)
    Transfer elements from bucket i to new calendar
    by enqueueing them;
  }  /* end */
```

The final procedure needed is newwidth( ). Newwidth( ) calculates the bucket width to use for the new calendar.

```
double newwidth( )
/* This calculates the width to use for buckets. */
  {
  int nsamples;
/* Decide how many queue elements to sample. */
if(qsize < 2) return(1.0);
if(qsize <= 5)
   nsamples = qsize;
else
   nsamples = 5 + qsize/10;
if(nsamples > 25) nsamples = 25;
Record lastprio, lastbucket, buckettop;
Dequeue nsample events from the queue and record
their priorities with resizeenabled equal to FALSE;
Restore the sampled events to the queue using en-
queue( );
Restore lastprio, lastbucket, and buckettop;
Calculate average separation of sampled events;
Recalculate average using only separations smaller than
twice the original average;
return(3.0 times final average);
} /* end */
```

5. EXPERIMENTAL RESULTS

All measurements were performed on a Harris HCX-7 super minicomputer or on a Texas Instruments (T.I.) PC. The HCX-7 is a 32-bit 10 MIP machine with a 4096 word instruction cache and a data cache. Programming was done in C under the Unix operating system. The Texas Instruments PC is a very small computer similar to the IBM PC. It has an 8088 CPU with a 5 Mhz clock and an 8087 coprocessor. Although the T.I. PC is a very small machine, it has the advantage of being a single user system. Most of the measurements were made on the PC rather than on the Harris minicomputer because timing measurements could be made much more accurately in the absence of task switching between users. All programming was done in C and all routines were coded for the maximum speed that could be obtained without resorting to assembly language. The Microsoft version 4.0 C compiler was used on the T.I. PC with extended (32-bit) pointers and the big memory model. The standard Unix cc compiler was used on the Harris minicomputer. Global optimization was used on each.

Time was measured on the T.I. PC using the system clock and a timing loop. The system clock only gives time to the nearest 0.1 second, so a timing loop was used to measure the time remaining until the next tick of the clock. This allowed time measurements on the T.I. PC to an accuracy of about one millisecond. Time was measured on the Harris minicomputer using the tms_utime field of the Unix tms function. Measurements were made when there were no other users on the system, but there was still variability of a few sixtieths of a second.

Figures 2, 3, and 4 show measurements of hold time. For each measurement the queue was built gradually to the required size using a random sequence of enqueue and dequeue operations, and then either one thousand or one hundred thousand hold operations were performed. One hundred thousand holds were used for measurement on the Harris and one thousand on the T.I. PC. Fewer holds were used for measurements on the PC because of its slower speed and greater accuracy of time measurement. The probability of an enqueue was 0.6 and the probability of a dequeue was 0.4 while the queue was being built to the required size. Measurements were made for 28 exponentially-spaced queue sizes ranging from 1 to 11,585 ($2^{13.5}$) as in the review article by Jones [9].

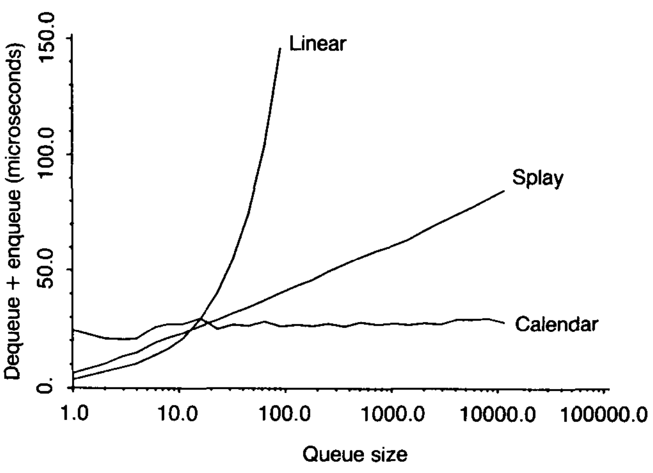

FIGURE 2. Comparison of Hold Times on Harris HCX-7

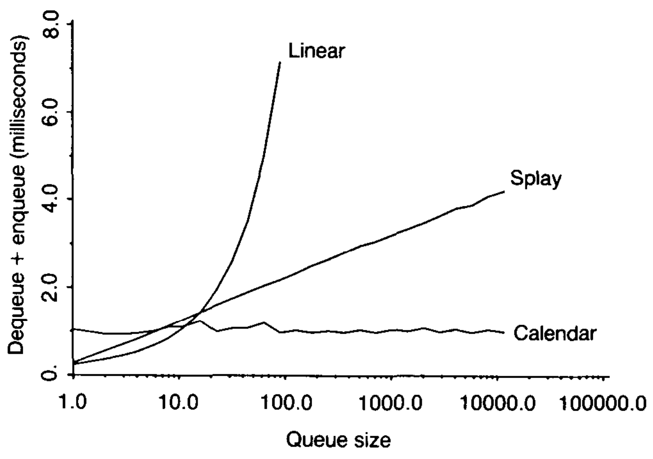

FIGURE 3. Comparison of Hold Times on Texas Instruments PC

Figures 2 and 3 compare the calendar queue hold time to the hold time for splay tree and linear linked list priority queue implementations. Figure 2 shows times on the Harris minicomputer and Figure 3 gives times on the T.I. PC. Jones provided a copy of the splay tree code which he used in his review article on priority queues. This code was translated from PASCAL to C and used for the comparisons shown in Figures 2 and 3. In each case the priority increment distribution was exponential. The priority of the next enqueued element was always the priority of the last dequeued element plus −ln(rand( )), where rand( ) returned a random value uniformly distributed between 0 and 1.

Figure 2 shows that on the Harris minicomputer the calendar queue hold time becomes smaller than the splay tree hold time when the queue size reaches 20, and stays smaller as the queue grows. Note that there is no tendency for the calendar queue hold time to increase as the queue size grows. The calendar queue hold time is approximately one-third that of the splay tree for a queue size of 11,585. On the T.I. PC (Figure 3), the calendar queue hold time becomes smaller than that of the splay tree at a queue size of 10, and the calendar queue hold time is approximately one-quarter that of the splay tree at a queue size of 11,585.

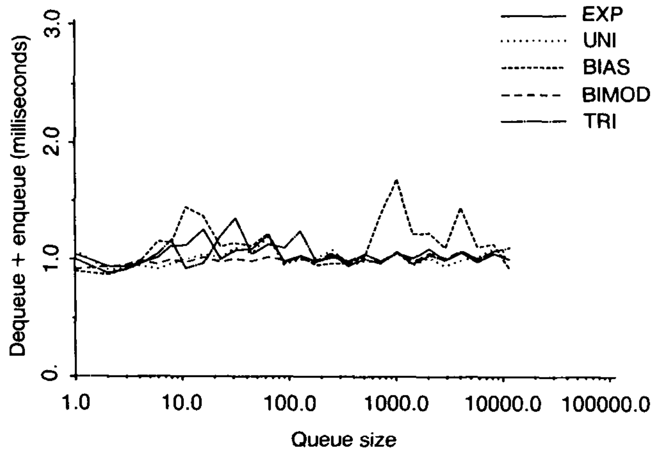

FIGURE 4. Calendar Queue Hold Times for Various Distributions

Figure 4 compares the performance of the calendar queue for several priority distributions. The distributions used are identical to those used by Jones in his survey [9]. They are shown in Table I. Performance is acceptable for all distributions, but is somewhat worse for the biased distribution than for the others. The reason seems to be oscillation of the density of events in the queue, which results in the bucket width not always being optimum. These measurements were all made on the T.I. PC.

Figure 5 shows the average enqueue plus dequeue times for non-static queues. Each point shows the average enqueue plus dequeue time for the complete process of building the queue to a specified size and then returning it to empty. Each curve gives results for a different rate of building and emptying the queue. Consider, for example, the curve labeled "prob = 0.75." The point at queue size 1024 is the average enqueue plus dequeue time over the entire process of building the queue up to 1024 elements and then returning it to empty. The queue was built with a random sequence of enqueues and dequeues, the probability of enqueue being 0.75; it was then emptied by a random sequence of enqueues and dequeues for which the probability of dequeue was 0.75. It can be seen that the probability determines the rate of increase and decrease of queue size. Since the queue size returns to zero, the total number of enqueues for the complete process equals the total number of dequeues. This means that the average enqueue plus dequeue time equals the total time for the process divided by the number of enqueues. The measurements were all made on the T.I. PC.

The enqueue plus dequeue time is seen to be greatest for rapidly changing queue size. This is as expected, since rapidly changing the queue size causes more frequent copying of the queue to new calendars. Probability equal to one corresponds to the most rapid increase and decrease possible, a string of enqueues followed by a string of dequeues. The enqueue plus dequeue time for this case is two to three times longer on the average than for hold operations. The oscillations are due to the fact that recopying of the calendar occurs at powers of two. The average time per enqueue suddenly increases every time the queue size exceeds a power of two, and then decreases until the next power of two is reached because the recopying time is being spread out over a larger number of enqueue and dequeue operations. Twice as many points were plotted per decade in Figure 5 as in the others, so that the oscillations could be seen more clearly. The queue size ratio between adjacent points is $2^{.25}$ rather than $2^{.5}$.

When the queue size rises and falls gradually, the average enqueue plus dequeue time is not much different than for hold operations. Consider the probability equal 0.6 curve. Prob. of enqueue/prob. of dequeue is

TABLE I. Priority Increment Distributions

Distribution[a]	Expression to compute random values[b]	Bias[c]
1. Exponential	−ln(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 [11], this is called %*F*; in [10], it is *E*(*G*(*x*)).

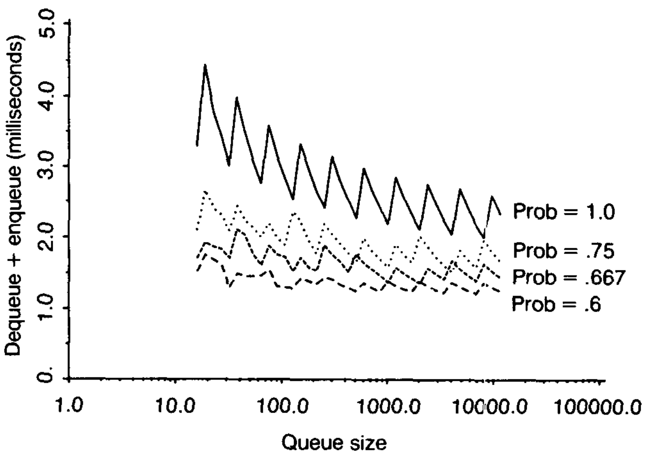

FIGURE 5. Enqueue + Dequeue Times for Non-Static Queues

1.5 while the queue is growing and ⅔ while it is shrinking. For this case the enqueue plus dequeue time is 30 to 50 percent greater than for hold operations. Notice that the enqueue plus dequeue time still shows no tendency to increase as the queue size increases. In fact it decreases, because the fixed costs associated with copying to a new calendar are less important for a large queue. Data is not shown for queue sizes less than 16 because the total time to build and empty the queue in these cases was too short for accurate measurement.

6. CONCLUSION

A new priority queue implementation for the future event set problem has been presented and tested. The new data structure is similar to a number of earlier multiple list and multiple pointer structures, but is a simplification and improvement on them. It does not have an overflow list, and is simple enough that execution time is quite fast for common distributions. Enqueue plus dequeue time was measured for each of the priority increment distributions considered by Jones in his review article on priority queues, and was found to not increase at all with queue size [9]. This is in agreement with what one would expect from an intuitive analysis of the calendar queue data structure and algorithm. The result is that for large queues the calendar queue hold time is much shorter than any others of which the author is aware.

In common with other similar structures, the calendar queue is somewhat sensitive to the priority increment distribution. If the priority distribution in the queue (not the priority increment distribution) is very non-uniform, then a bucket width which is small enough to prevent the existence of a large number of events in buckets at which the density is high will be so small that there are many empty buckets where the density is low. The calendar queue also will not work well if the distribution of priorities suddenly changes drastically.

Suppose, for instance, that one builds the queue size to 1000 and then begins doing only hold operations. Since the queue size is no longer changing, the bucket size will remain fixed. If the mean priority increment were then to change from 1.0 to 1000.0, the calendar queue would perform very poorly. If this is likely to happen in a proposed application, then the queue statistics should be continually monitored and the queue should be copied to a new calendar with a new bucket width as needed. This is an area in which more work could be done.

Nevertheless, the calendar queue seems to be fairly robust. It worked well for the priority increment distributions used by Jones in his measurements [9]. This is due in part to the number of elements per bucket becoming quite large, 15 or 20, before performance begins to degrade significantly. A sorted linked list must be fairly long before the time to insert an element in it is comparable to the total time needed for an enqueue and dequeue. The author will be pleased to provide a copy of the calendar queue code to anyone wishing to conduct further tests on it or desiring to use it in an application—see About The Author section appearing beneath the list of references for address information.

REFERENCES

1. 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.
2. Brown, M.R. Implementation and analysis of binomial queue algorithms. *SIAM J. Comput. 7*, 3 (Aug. 1978), 298–319.
3. Davey, D., and Vaucher, J. Self-optimizing partitioned sequencing sets for discrete event simulation. *Infor 18*, 1 (Feb. 1980), 41–61.
4. 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.
5. Franta, W.R., and Maly, K. An efficient data structure for the simulation event set. *Commun. ACM 20*, 8 (Aug. 1977), 596–602.
6. 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.
7. Fredman, M.L., Sedgewick, R., Sleator, D., and Tarjan, R. The pairing heap: A new form of self-adjusting heap. Submitted for publication.
8. 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.
9. Jones, D.W. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM 29*, 4 (Apr. 1986), 300–311.
10. 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.
11. 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.
12. Nix, R. An evaluation of pagodas. Res. Rep. 164, Dept. of Computer Science, Yale Univ., New Haven, Conn., no date.
13. 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.
14. Sleator, D.D., and Tarjan, R.E. Self adjusting heaps. *SIAM J. Comput.* To be published.
15. Tarjan, R.E., and Sleator, D.D. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.
16. Ulrich, E.G. Event manipulation for discrete simulations requiring large numbers of events. *Commun. ACM 21*, 9 (Sept. 1978), 777–785.
17. Vuillemin, J. A data structure for manipulating priority queues. *Commun. ACM 21*, 4 (Apr. 1978), 309–315.

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

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Event set, priority queue

ABOUT THE AUTHOR:

RANDY BROWN is an associate professor of electrical engineering at the University of Arkansas. His research interests include computer aids for the design of integrated circuits, and fundamental computer algorithms. Author's present address: Dr. Randy Brown, Department of Electrical Engineering, University of Arkansas, 232 Science-Engineering Building, Fayetteville, AR 72701.