Lazy Queue: An Efficient Implementation of the Pending-event Set

Robert Rönngren, Jens Riboe and Rassul Ayani
Department of Telecommunication and Computer Systems
The Royal Institute of Technology
P.O.Box 70043
10044 Stockholm, Sweden
Email: robertr@tds.kth.se
FAX: +46 - 8 24 77 84

Abstract

A new priority queue implementation, the Lazy Queue, is presented in this paper. It is tailored to handle the pending event set encountered in discrete event simulation. The Lazy Queue is a multi-list variety where the sorting process is delayed until a point near the time where the elements are to be dequeued. The queue access time has been measured and compared with the access times of an implicit heap and a calendar queue. The experimental results indicate that the Lazy Queue is superior to these priority queue implementations.

1 Introduction

In discrete event simulation, very often the pending event set (PES) is represented by a priority queue. The data structure used to represent the queue and the way in which the operations are performed on it, i.e. sequentially or concurrently, is often crucial to the execution of a simulation.

Priority queues for the pending even set problem can roughly be characterized into two sub-groups, tree oriented and multi-list oriented priority queue algorithms. In the former class resides queues such as Binary Heap (Implicit Heap), Leftist Tree (Explicit Heap), Pagodas, Skew Heaps, Splay Trees (balanced binary search-tree) [Jones86]. The latter group includes Two List [Jones86], Multi List [Ayani87] and Calendar Queue [Brown88]. The classical linear list could be excluded from the classification or treated as a degenerated multi-list. That also holds for Henriksen's multi-entrance linear list [Jones86]. The queue algorithm in focus for this paper belongs to the multi-list group and we therefore concentrate the discussion of related work to multi-list algorithms.

A simple multi-list structure is the Two List structure that divides the queue elements into two lists. One short sorted list for elements belonging to the near future and a long unsorted list of more distant elements. The decision of where an enqueued element should be placed mandates of a threshold priority. The first list requires a traversal of the list to find the right place. The second list provides a fixed insertion time. A dequeue operation normally removes the first element of the sorted list. If the list becomes empty however, a new threshold need to be computed (for example, by sampling a couple of priorities to establish a desirable length of the sorted list). Then, all elements ahead of the new threshold must be moved and inserted in sorted order into the first list.

A plain multi-list structure consists of an array of sorted lists, where the last list serves as the overflow bucket. The threshold priority is generalized to a couple of priority intervals, one for each sub-list—excluding the overflow list. For simplicity the length of the intervals are assumed to be equal. This simplification, however, makes the list more sensitive to skewness or peaks in the priority distribution.

The Calendar Queue is a sort of multi-list structure with sorted sub-lists. It solves the problem of the overflow bucket management, by letting the overflow elements be spread over the lists. It is performed by introducing the concept of a year. A year stands for the total interval the sub-intervals span. Each sub-interval represents a day. The overflow elements belong to future years, as opposed to the non-overflow elements which belongs to the current year.

The dequeue operation removes the element with the smallest time (highest priority). In order to find the least element, because it can reside in front of any of the sub-lists, the Calendar Queue structure need to maintain a last visited day index, which serves as a starting point for a search for the least element. If the least element belongs to the last visited day, then a dequeue operation will take a fixed amount of time. If the first element of the last visited day has a time (priority) greater than the current year length, it belongs to some future year. Then the last visited day index needs to be incremented modulo the number of days. This scheme will be repeated until the
least element of the current year is found or a whole year (all sub-lists) have been examined. The last case triggers a simple linear search through all days, examining the first element of each sub-list, looking for the least element belonging to some future year.

Besides of the overflow management, Calendar Queue also improves Multi List by maintainence of short sub-lists, typically two to four elements in the best case. This is done by recomputation of the day length and the number of days per year, when the queue size exceeds an upper threshold or falls below a lower threshold. The thresholds are typically two times the current number of days per year and half the ditto. A resize operation consists of a doubling or a halving of the number of days, a recomputation of the day length and a complete re-ordering (movement) of the queue elements. The resize operation is very expensive in consumption of time, but the time is amortized over the very cheap plain enqueue and dequeue operations. As the queue size increases there seldom exists a need for a resize operation, with the exception of when too many empty sub-lists (unbalanced queue) arises. This phenomenon is not catched by the original algorithm.

In this paper a new method, the Lazy Queue, is presented for implementing the PES. The Lazy Queue is specifically tailored to handle the Pending Event Set in discrete event simulation. The fundamental idea of the Lazy Queue is to divide the future events into several parts and only keep a small part of the elements sorted. The elements are divided into three parts: a near future that is kept sorted, a far future that is partially sorted and a very far future that is used as an overflow bucket. As time advances parts of the far future are sorted and transferred into the near future. This lazy sorting behaviour has given the queue its name. The rest of this paper is organized as follows: first the Lazy Queue is presented, the details of the self-adjustment policies of the queue and the memory requirement and management are discussed, some experimental results are presented where the Lazy Queue is compared to other queue implementations. Pseudo code implementation of the basic operations on the Lazy Queue can be found in Appendix 1.

2 The Lazy Queue

The objectives of this investigation has been to design a priority queue implementation tailored to the discrete event simulation. As a foundation for the work we have some observations concerning how we, as human beings, maintain our agendas. We can, as in Calendar Queue [Brown88], make analogies to how we plan our future with the help of an agenda. One possible way of handling our agenda is to divide the future into a number of time intervals. There is a near future (NF), for which most of the events are known and thus we may have a detailed schedule. A far future (FF), where we expect that most of the yet unknown events will occur. For this period, we do not make any detailed plan yet. For those events that fall even further into the future, this period is called very far future (VFF), there may not even be any place in the actual agenda; the events occurring in VFF may be stored somewhere else. As time advances, the boundaries for NF, FF, and VFF are advanced. We hypothesize that a lot of work can be saved by postponing scheduling of events until there is good reason to believe that most new events will occur in a more or less distant future.

In our present implementation the NF consists of a sorted array of elements and a linked list. The FF consist of several months where each month contain an unsorted array of events. The VFF is implemented as a linked list (see Figure 1). A QuickSort algorithm [Sedgewick88] is used for sorting the months that are transferred from the FF to the NF.

An enqueue operation is carried out by determining into which part of the queue the event falls and then inserting the element into it. As mentioned, we expect most of the future events to occur in FF. Thus, in most of the cases, a new event is simply appended to one of the sublists of the FF. If the new element falls into the NF it is inserted into the linked list of the NF, thus this linked list is referred to as the insertion part of the NF. When doing a dequeue operation the smallest element is taken out from the NF. If there are no elements present in the NF the next non empty month of the FF is sorted and transferred to the NF; at the same time the borders between NF, FF and VFF are adjusted. In this way, the sorting process is delayed and is done only for a small part of the PES at a time. Hence, we call our implementation of the PES Lazy Queue.

In order to get good performance from any multi-list based queue the width and number of the sub-intervals has to be chosen appropriately. As we expect the Lazy Queue to be less sensible to distributions that are skew or have peaks a rectangular distribution is used as an approximation of the priority distributions. Using this approxi-
mation the length of the months\(^1\) is computed so that a constant number of elements can be expected to fall into each month (all months have equal length). This constant is called EM (Expected number of elements per Month), the actual average number of elements that fall into the months is called AEM. The number of months is chosen so that only a small number of elements will fall into the VFF.

As in other multi-list structures, some mechanism is needed to adjust the length and number of sub intervals (months) in order to adapt the queue to changes in size and changes in the priority distribution. These resize operations basically consist of halving or doubling the length and number of months.

### 2.1 Resize criteria

A set of parameters and criteria has to be defined to govern decisions of when and how the queue should be resized. These parameters and criteria must be carefully selected to get optimal performance from the queue and to avoid unnecessary resizes. The actual ranges that these parameters can vary within, that is the actual criteria to decide resizes upon, are dependent on the implementation and on the choice of EM.

The parameters selected are:

1. \#NF, the number of elements present in the insertion part of the NF
2. \#VFF, the number of elements present in the VFF
3. AEM, the actual average number of elements in each month
4. the height and width of peaks detected in the priority distribution
5. the average number of elements present in the upper half of FF

Based on these parameters the following criteria governing the resize operations can be stipulated:

1. \#NF \(\in [0, \text{MAXNF}]\)
   
   \text{MAXNF} is the upper threshold for the number of elements present in NF.

2. \#VFF \(\in [0, \text{MAXVFF}]\)
   
   \text{MAXVFF} is the upper threshold for the number of elements present in VFF.

3. AEM \(\in [\text{LOWERBOUND}, \text{UPPERBOUND}]\)
   
   LOWER and UPPERBOUND for the actual average number of elements in each month.

4. the number of elements present in the upper half of FF
   
   \(\#VFF > \frac{\text{MAXVFF}}{2}\)

   This criterion is set to ensure a good utilisation of the FF.

5. the average number of elements present in the K first months of the FF
   
   \(\text{K} \leq \text{PeakThreshold}\)

   K is normally the minimum of a constant number and half the number of months in FF. \text{PeakThreshold} is a constant number.

Criteria 1 - 4 are strong criteria in the sense that they should never be violated. Criterion 5 only results in resizes if the other criteria can be met after the resize.

Criterion 5 might seem to be redundant, but the reason for this criterion is twofold: (1) we can detect peaks close to the NF and spread these elements over several months to lower the risk of having too many elements falling into the NF (this is achieved by halving the length of the months); (2) it allows a choice of a higher value for \#NF as peaks and changes in the priority distributions are likely to be detected earlier. This lowers the risk of having unnecessary resizes imposed by criterion 1.

As can be seen from Figure 3, the boundaries for parameters 3 - 5 can be set rather wide. This reduces the number of resize operations necessary to perform. Note that if criterion 4 is violated this should not always imply a full halving of the number of months, normally a smaller adjustment of the number of months is performed. Otherwise this could cause a series of consecutive halving and doubling of the number of months.

### 2.2 Application of the resize criteria

The resize criteria are checked in the dequeue and enqueue operations as follows:

When a dequeue operation is performed and the NF is empty the FF is scanned for the next non-empty month. Before sorting this non-empty month and transferring it to the NF, criteria 3 - 5 are checked and implicated resize operations will be performed. We must also control that these resize operations do not result in a state violating criteria 1 or 2.

When performing an enqueue operation where the element falls into NF or VFF criteria 1 and 2 are checked respectively. The resulting resize operations are performed so that criteria 3 and 4 are met (since this only involves halving the length of the months and/or doubling the number of months criterion 5 is not affected). In a build up phase, when no dequeue operation yet has been performed, criteria 3 and 4 are also checked and if they are

\(^1\) By "length of month" we mean the time interval a month spans (not the time interval spanned by the elements present in a month). This time interval is equal for all months in the FF.
violated the length and number of months will be directly recomputed.

It should be noted that the resize criteria are controlled only at a few occasions and that the computations needed for these controls are small.

2.3 Resize operations

The four basic resize operations performed are:
1. Halve the number of months
2. Double the number of months
3. Halve the length of the months
4. Double the length of the months

Changing the length and number of months has often to be combined, i.e. when the length of the months is halved the number of months must often be doubled.

Halving the length of each month is performed by sorting the elements of each month (this means that the queue will be totally ordered). Then to check if it is necessary to increase the number of months, the number of elements in the upper half of the FF is calculated. If there are only a small number of elements in this part of the FF they are moved to the VFF and the boundary between FF and VFF is adjusted. Otherwise the number of months is recalculated, which normally results in doubling the number of months. Finally each month is split into two and the size of the array holding the elements in each month is adjusted.

Doubling the length of each month is performed by pairwise joining the months and moving them into the lower half of the FF. If the number of elements in the upper half of the FF is low after this operation these elements are moved into the VFF and the number of months is halved, this is done to assert a good utilization of the FF part. Finally the boundary between FF and VFF is adjusted. In each month we keep track of the sorted parts in order not to perform any unnecessary sorting.

When comparing these strategies to those proposed for the Calendar Queue [Brown88] it can be seen that:

The Lazy Queue strategies adapts the queue to changes in the priority distribution also without changes in the queue size. This case is not handled by the original Calendar Queue strategies. The resize operations in the Lazy Queue could also be expected to occur less frequently than they will occur if this case were to be treated by the Calendar Queue. This due to efficiency reasons, the length of the linked lists in the Calendar Queue can not vary as much as the number of elements that fall into each month in the FF of the Lazy Queue without affecting overall performance. But it should also be noticed that by reintroducing an overflow structure we have also reintroduced the problems that will follow with it. The Lazy Queue could be expected to perform poorly for example in cases when nearly all new elements fall into the VFF or the NF.

2.4 Memory requirements

As the worst case for the memory usage the months of the FF are only half filled. This and the descriptor for the queue (primarily the array of months) gives us a maximum memory requirement of $2nE + f(n)$ where $E$ is the storage size in bytes of an element in the queue and its associated priority. In our experiments we observed that $f(n)$ was approximately equal to $nE$ for all the used distributions and queue sizes, giving a total memory requirement of $3nE$. This can be compared to the Implicit Binary Heap which has a worst case of $2nE$.

It should also be noticed that problems with memory fragmentation may arise if naive strategies for allocation and resizing of the arrays of the months in the FF are adopted. Hence we use free-lists to store arrays of different sizes and we also preallocate arrays in these free-lists. This preallocation is done in order not to disturb the timing measurements in our experiments. This could also be desirable to do when putting the queue into actual use, enhancing performance and avoiding fragmentation. Thus the actual memory requirements may be greater than the $3nE$.

The reasons to believe that the Lazy Queue will render good performances can be summoned as:

1. most of the operations on the queue will access arrays with the indices known when the operation starts;
2. the sorting process, which is time consuming, is delayed and performed on demand (as opposed to the conventional multi-list implementations, where the sorting is done during the enqueue operation);
3. the fact that we can use a sorting method with a $O(\log(n))$ per element behaviour will reduce the sensibility to skewness or peaks in the priority distribution compared to other multi-list based priority queue implementations, e.g. Calendar Queue.

Moreover time consuming operations such as sorting a large number of events, resizing the queue and performing insertions in long linked lists can be expected to be scarce.

---

2 The upper half of the FF: e.g. if we have an FF that consists of months Jan-Dec and the present is at the beginning of the year (new years eve) the upper half of the FF would consist of months Jul-Dec. The lower half would be Jan-Jun.
3 Performance measurements

As discussed earlier it must be decided how to implement the different parts of the queue as well as how the queue should adapt itself to variations in the number of elements and priority distribution of the elements. These decisions will affect the overall performance as well as the worst case behaviour of the queue.

3.1 Implementation of NF, FF and VFF

We have chosen to implement the insertion part of the NF and the VFF as linked lists. This gives a worst case behaviour of O(n). It could have been alleviated by choosing data structures with O(log(n)) behaviour per access but this is likely to degrade average performance since these has longer access times for small sizes than lists, as well as it would make resize operations more complex.

The FF is implemented as an array of months. Each month consists of an array of elements with associated priorities. The element array of a month is allocated only when needed and deallocated as soon as possible. The initial allocation size is chosen as EM and the size of the array is doubled when needed. All memory management of these arrays is done by keeping free-lists of arrays of different sizes (EM, 2*EM, 4*EM etc) to avoid memory fragmentation.

As sorting method a Quick Sort that resorts to Insertionsort for short sublists has been used [Sedgewick88]. This method gives acceptable performance both for short and long sub-lists. It has an average time complexity of O(log(n)) per element.

3.2 Used parameters

We have chosen EM as 8. This allows us to set the boundaries for the actual average number of elements that fall into each month as [4,64] (see figure 2). Thus we lower the sensitivity to peaks in the distribution. A minimum queue size of 256 elements is imposed for any resize operation to take place. The number of elements allowed in the VFF (MAXVFF) is 256 and the number of elements allowed in the input part of the NF (MAXNF) is set to 1024.

The choice of MAXVFF as 256 depends on the fact that we want the queue to quickly adapt itself to the increasing size and to the priority distribution in a build up phase. When this transient phase is over we could increase the MAXVFF value. The choice of MAXNF is meant to make the time consuming resize operations involving halving the length of months less frequent. We can argue that we can choose a high value for MAXNF by examining Figure 2 that tells us that the queue tolerates rather long linked lists in the NF before the performance of the linked list starts to affect the overall performance.

To detect peaks we calculate the average number of elements present in the first months of the FF. By the first months we mean the minimum of 8 and half the number of months in the FF. If the average number of elements in these months is above 64 (PeakThreshold) and if it would not lower the average number of elements per month below the lower bound for AEM a resize is performed. Thus we can detect and spread peaks over more months and decrease the probability of having too many elements falling into the input part of the NF.

For pseudocode implementations of the basic operations (enqueue and dequeue) see Appendix 1.

3.3 Priority distributions

There exists a lot of different priority distributions. The problem is, which one to choose as the priority generator. One approach is to systematically use each distribution in turn or a favourite selection of them and measure the timings [Jones86]. One question mark which puts up is, does the selection reflect a typical usage? Our approach is to abstract characteristic priority distribution profiles [Riboe90b]. Or put into other words, the distributions used are models of real distributions. We have tested the Lazy queue against five distributions, Rect, NegTriang, Triang, Camel and Exp. Rect is the rectangular or uniform distribution and it models a priority distribution evenly spread over a closed interval. NegTriang and Triang are the Negative Triangular and the Positive Triangular distributions and they model distributions with high probability of insertions in the front of a queue, respectively at the end of a queue.

The Camel distribution [Riboe90a] models a distribution with peaks. Precisely as the other distributions, it is defined on a closed interval, which is needed if one wants to isolate the differences of the behaviour of the four distributions. Besides of the interval bounds, a Camel distribution is dependent of three parameters: the number of humps (n), the fraction of the probability mass distributed on the humps (m) and the fraction of the total width the sum of the hump widths occupies (w). For the rest of this paper, n = 2 and therefore is omitted in the figures. Because the m and w denotes fractions they belong to the interval [0,1].

The example distributions used in this paper are Camel(0.8, 0.2) and Camel(0.999, 0.001). The former is a normal two hump distribution with 80% of the mass in the humps (40+40) and 20% of the domain interval occupied of the two humps (10+10). The latter is an extreme and essentially forms a two point distribution (99.9% mass and 0.1% width for the two humps).

For the Rect, Triang, NegTriang and the Camel
distribution the interval used has been [0,1000].

![Distribution Diagram](image)

Figure 2: Bounded interval priority distributions used in the experiments

Exp is an exponential distribution with a mean of 1. This distribution was primarily chosen because it is defined on an open ended interval [0,∞[.

3.4 Measurement methods

The focus of the work was to get a picture of the mean access time for a queue under different loads. We define access time as an enqueue (insert) or a dequeue (delete minimum) operation. The parameters we have varied, beside the queue specific, are: the access pattern, the queue size and the priority distribution.

The access pattern captures a steady-state behaviour and a transient behaviour. The former is Classic Hold [Jones86] and the latter is Up and Down [Riboe90b]. Classic Hold models the behaviour of a discrete event simulation system performing a sequence of hold operations—a dequeue followed by an enqueue. The queue size remains unchanged, because of an equal number of deletes and inserts. Up and Down models a transient phase of a queue. The queue will be loaded up to some queue size and then emptied again. Classic Hold and Up and Down represents two extremes and serve to point out the bounds of the performance figures.

The queue sizes used range from small to very large sizes (20 000). The interesting question here is: is the queue in question an all-round queue, usable for arbitrary sizes or just for some particular queue size interval?

The experiments were performed on a Sequent Symmetry S81 [Sequent89]. Actually, the machine is a multi-processor but this is of no interest because we are discussing sequential algorithms executing on one computation unit. Nevertheless, it has one very important feature which we have based the experiment upon. Each processor is equipped with a register based clock, which makes it possible to do timings within a microsecond resolution and accuracy.

Each queue operation where timed separately and accumulated in a table for later statistics computations. This is in contrast to more conventional methods where the experiments are performed by starting the clock, running the whole queue experiment, stopping the clock and dividing the total time by the number of loops (see for example [Jones86]). Of course this is necessary if the clock (as in most computers) has low resolution, but it introduces inaccuracy due to the loop overhead.

All performance measurements were performed with the internal queue structures (memory blocks etc) scaled up to appropriate sizes. Thus effects of the performance of the under-lying memory management system is eliminated from the obtained figures.

All code was written in the C programming language and the queue specific parameters used for the Lazy Queue were those described in section 3.2.

4 Experimental results

The first experiment shows the impact of the choice of EM on the access time of the Lazy Queue. It tells us that we can use a wide range of EM with only small changes in performance of the queue, see Figure 3. This figure also indicates that the Lazy Queue is not sensitive to inhomogenous distributions.

Mean access time µs

![Mean Access Time Chart](image)

Figure 3: Access times for Lazy Queue for various choices of EM

(Expected number of elements per month)
experiments. Figures 4 - 6 show the results from steady state experiments. In these experiments a method has been used where the number of hold operations performed has been ten times the queue size. This method gives us a constant ratio of the number of elements in the queue and the hold operations performed. This lowers the effects that will be caused by the actual distribution of the elements in the queue changing as a result of the performed hold operations. If a constant number of hold operations where used, as in [Jones88], this would affect the smaller queue sizes more than the larger.

The Binary Heap has an algorithmical behaviour with $O(\log(n))$ access time and the access time is essentially independent of the actual distribution used [Jones86]. Figures for the Binary Heap used with a rectangular distribution are shown as dashed lines to facilitate comparisons between the Lazy Queue and the Calendar Queue.

![Diagram showing Mean access time μs](image)

Figure 4: Performance of Lazy Queue in Steady State experiments

As can be seen from Figure 4, the Lazy Queue has a stable behaviour even for distributions that have high peaks. Compared to the Binary Heap (dashed line) it performs very well. If we compare these figures with those for the Calendar Queue, Figure 5, we see that the Lazy Queue compares favourable in all cases. The Calendar Queue approaches and even surpasses the Binary Heap for some distributions. It is notable that this occurs though the distribution does not change over time and even for such a case as the triangular distribution (Figure 6). This can be explained by the fact that the heuristics for adjustment of the queue do not work well for cases where a good approximation of the distribution can not be made just by examining the first few elements present in the queue. For the cases of the triangular and the camel distribution this leads to:

1. the Calendar Queue makes less accurate calculations of the day length as the queue size grows;
2. this leads to almost all elements falling into a few sub-lists where the insertion time approaches an $O(n)$ behaviour.

The leaps in the curves of the Calendar Queue, Figure 5 and 8, coincide with the resize points.

![Diagram showing Mean access time μs](image)

Figure 5: Performance of Calendar Queue in Steady State experiments

To be able to fully judge the queue we must also see
that the time for transients, increasing and decreasing the queue size, is constrained. This gives us an idea of the penalty paid for adjusting the queue to changes in size and distribution. We have measured this by performing Up and Down experiments. As we see from Figure 7 we have a worst case for the Lazy Queue when a highly non uniform distribution is used. This depends on the first approximations made, when the queue contains relatively few elements, not being accurate enough. This leads to resize operations when the number of elements has increased. In the case of the camel distribution we do not detect the peaks early enough to avoid halving the length of the months as dequeue operations are performed. Thus an almost total ordering of the elements is performed. This can be done in $O(n \log(n))$ time and we get an access time that approaches that of the Binary Heap. This sorting effort is not wasted since the sorted parts do not have to be sorted again, if necessary they may however be merged with other sorted parts. When we compare the Lazy Queue performance to that of the Calendar Queue (Figure 8) we see that the time to build the Calendar Queue shows more variation. This is explained by the same reasons as in the steady state case. In the case of a resize the Calendar Queue scans through all its elements inserting them into the newly calculated days. If nearly all elements fall into a few sub-lists this operation has a time complexity of $O(n^2)$ that when amortized over $n$ operations results in $O(n)$ performance.

5 Conclusions

We have presented a new priority queue implementation, the Lazy Queue, for the pending event set problem. The data structure resembles earlier multi-list structures but has some distinctive differences. The queue is divided into three parts: (1) the near future, NF, where elements close to the actual simulation time is kept, (2) the far future, FF, that consists of several sub intervals (months) and (3) the very far future, VFF, which serves as an overflow list. The most interesting difference is that it normally delays the sorting of the elements until a point near the time when the elements are to be dequeued. This is achieved by transferring and sorting months from the FF to the NF on demand. It enables us to save some work and makes it possible to use arrays to store the elements in. Thus we can use standard sorting techniques when sorting the elements further improving performance. A general adjustment strategy for the queue has been implemented, enabling it to quickly adapt to changes in both the priority distribution and the size of the queue. The queue is also modularly built and we can change the implementations of the input part of the NF and the VFF as well as the sorting method to tailor it to special behaviour of the application.

Enqueue and dequeue times have been measured for a number of different priority distributions and queue sizes both in steady state and Up and Down experiments. These experiments have also been performed for two queue implementations reported in the literature: the Implicit Binary Heaps and the Calendar Queue. The experiments show that the Lazy Queue performs as well or better than
these queues in most cases. The experiments also show that the Lazy Queue seems to be more stable and handles skew priority distributions better than the Calendar Queue. When studying the performance figures we should bear in mind that the Calendar Queue probably would perform better if a more elaborate resize strategy for it could be developed.

Although the experiments show that the Lazy Queue has a stable performance that only depends little on the queue size and the distribution we can expect it to perform poorly in certain cases. These could be when we have distributions such that most of the elements fall into the NF or VFF part of the queue forcing the queue into a series of resize operations. This has still to be further investigated. It also has higher memory requirements than the other queues tested. Nevertheless we can argue that the Lazy Queue will work well for most practical simulations.

The Lazy Queue has some characteristics that makes it an interesting candidate for a parallel implementation. This issue will be further investigated.

Acknowledgement
This work is part of a distributed simulation project financed by the Swedish National Board for Technical Development (STU).

References


Appendix 1
In this section we present the data types of the Lazy Queue and the basic queue operations, enqueue and dequeue, in outline fashion. These outlined parts are then expanded into further detail. Last the resize controls are outlined. The actual resize operations are not presented in detail since they only would give minor contributions in the understanding of how the Lazy Queue works.

Data types and constants

```c
/* Constants */
#define EM (8)
#define UPPERBOUND (64)
#define LOWERBOUND (4)
#define MAXNF (1024)
#define MAXVFF (256)
#define MINQ (256)
#define ListsToScan(queue) \  
  min(8,queue>cardinal/2)

/* Data types */
typedef struct {
priority pri;
/* priority = time stamp of event */
element elem;
/* normally a pointer to the element*/
} pri-elem;

typedef struct {
  int size,
  /* size of allocated array */
  next,
  /* next empty spot in array */
  sorted;
  /* array is sorted up to this index */
  pri-elem *elements;
  /* pointer to allocated array where
     elements are stored*/
} month;

typedef struct {
  int cardinal;
  /* number of elements present in the queue */
  /* NF part */
  int NF_cardinal;
  /* total number of elements in the NF part */
  int NF_size,
  /* size of array for element storage in NF */
  int NF_next;
  /* index to the first element present in the */
  /* element array in NF */
  pri-elem *NF_elements;
  /* pointer to element array of NF */
  linkedlist NF_head;
  /* pointer to input list of NF */
  /* FF part */
  priority month_length;
  /* interval each month spans in the FF part */
  priority FF_lowborder;
  /* border between NF and FF */
  int nr_months,
```
A further detailed code is shown in the following sections.

Further details of enqueue

The "InsertElementInto" operations of the enqueue operation can be expanded as follows:

InsertElementIntoNF:

```c
void InsertElementIntoNF(LazyQueue *queue, pri_el *el)
{
    queue->NF_cardinal++;
    insertlist(queue->NF_head, el);
    CheckResizeNF;
}
```

InsertElementIntoFF:

```c
int i;

if(el.pri < queue->FF_lowborder)
    InsertElementIntoNF;
else if(el.pri < queue->FF_highborder)
    InsertElementIntoFF;
else
    InsertElementIntoVFF;
```

The FF part of the queue is kept as a circular buffer of months. In this implementation, the number of months is not a power of two hence the need for a modulo operation when computing the index of the month in the FF part.
Further details of Dequeue

The "GetNextMonth" operation of the dequeue operation can be expanded as follows:

GetNextMonth:
/* get next nonempty month */
{  
    /* Search for next nonempty month */
    while(queue->FF[queue->first_month].next == 0)
    {  
        queue->first_month=(queue->first_month+
        1)%queue->nr_months;
        queue->FF_highborder+=
        queue->month_length;
        queue->FF_lowborder+=
        queue->month_length;
        Move elements from VFF into FF;
    }
}

CheckResizelnDequeue;
sort(queue->FF[queue->first_month]);
Transfer the first month to NF;
queue->first_month=(queue->first_month+
1)%queue->nr_months;

The resize checks performed in enqueue

The "CheckResize" operations of the enqueue operation can be expanded as follows:

CheckResizeNF:
if (The number of elements in queue->NF_head > MAXNF)
{  
    If we only have a small number of elements in
    the upper half of FF move these to VFF.
    Else if we can double the number of months
    without letting the average number of
    elements fall below LOWERBOUND after
    halving the length of months do so,
    Else recalculate the number of months needed.
    half_month_length(queue);
}

CheckBuildUpResize:
if (NO dequeue has been PERFORMED & &
queue->cardinal > MINQ)
{  
    If necessary adjust the queue so that we get an average
    number of elements per month that is EM and the
    tightest possible boundaries is set on FF.
}

CheckResizeVFF:
if((Number of elements in VFF > MAXVFF & &
queue->cardinal > MINQ)
{  
    Adjust the number of months and possibly the length
    of months so that the number of elements in VFF is
    less than 32;
}

The resize checks performed in dequeue

CheckResizelnDequeue:
if (Average number of elements per month > UPPERBOUND
   || (The average number of elements in the
       ListsToScan(queue) first months is greater than
       64 ||
       Average number of elements per month >
       3*LOWERBOUND))
{  
    if (The number of elements in the upper half of FF +
        the number of elements in VFF >
        MAXVFF*0.75)
    {  
        double_nr_months(queue);
    } 
    else
    {  
        move elements from upper half of FF to VFF;
        half_month_length(queue);
    }
}
else
if (Average number of elements per month <
LOWERBOUND)
{  
    double_month_length(queue);
    if (The number of elements in the upper half of FF +
        the number of elements in VFF >
        MAXVFF*0.75)
    {  
        move elements from upper half of FF to VFF;
        half_nr_months(queue);
    } 
}

204