

Lazy Queue: A new approach to implementing 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
S-100 44 Stockholm, Sweden

Email: robertr@tds.kth.se
FAX: +46 - 8 24 77 84
Phone: +46 - 8 790 78 82

Lazy Queue: A new approach to implementing the Pending-event Set¹

Abstract

In discrete event simulation, very often the future event set is represented by a priority queue. The data structure used to implement the queue and the way operations are performed on it are often crucial to the execution time of a simulation. In this paper a new priority queue implementation strategy, the Lazy Queue, is presented. It is tailored to handle operations on the pending event set efficiently. The Lazy Queue is a kind of multi-list data structure that delays the sorting process until a point near the time where the elements are to be dequeued. In this way, the time needed to sort new elements in the queue is reduced. We have performed several experiments comparing queue access times with the access times of the implicit heap and the calendar queue. Our experimental results indicate that the Lazy Queue is superior to these priority queue implementations.

Key words: Discrete Event Simulation, Priority Queue, Event List implementation, performance measurement.

1 Introduction

In discrete event simulation, the pending event set (PES) is the set of all generated but not yet evaluated events. The implementation of the PES is often crucial to the execution speed of a simulation. Normally the PES is represented by a priority queue, where the time of the events are used as priorities. Several methods for implementing priority queues have been proposed in the literature, e.g. [1, 2, 3, 4, 5]; and performance of these methods has been studied by many researchers, e.g. [6, 7, 8, 9, 10, 11].

Priority queue implementations can be roughly classified into two categories, tree and multi-list oriented ones. Implementations such as Implicit Heap, Explicit Heap, Pagodas, Skew Heaps, and Splay Trees [8] belong to the former category, while the latter includes Two List [1], Multi List [6] and Calendar Queue [3].

The Two List structure is a simple multi-list structure that divides the queue elements into a short sorted list, containing the elements belonging to the near future, and a long unsorted list, containing the other elements. A basic multi-list structure consists of an array of sorted lists, where the last list serves

¹ This work is part of a distributed simulation project financed by the Swedish National Board for Technical Development (STU). This paper is based on "Lazy Queue: An Efficient Implementation of the Pending-event Set" by R. Rönngren, J. Riboe and R. Ayani, which appeared in: *Proceedings of The 24th Annual Simulation Symposium*, New Orleans, Louisiana, April 1 - 5, pp. 194 - 204, ©1991 IEEE.

as an overflow bucket. For simplicity, the length of all the intervals, except for the last (overflow) one, can be assumed to be equal. This simplification, however, makes the list more sensitive to skewnesses or peaks in the priority distribution.

The Calendar Queue [3] is a kind of multi-list structure with ordered sub-lists. It removes the problem of overflow management by spreading the overflow elements over the sub-lists. In the Calendar Queue, each sub-interval represents a *day*, while a year stands for the total interval the sub-intervals span. The overflow elements belong to future years, as opposed to the non-overflow elements belonging to the current year.

The dequeue operation removes the element with the earliest time (highest priority). In order to find the smallest element, which may reside in front of any of the sub-lists, the Calendar Queue structure maintains an index that points to the last visited day. This index serves as a starting point to search the smallest element. If the smallest element belongs to the last visited day, it is immediately accessible and the dequeue operation will take a constant time. However, if the first element of the last visited day does not belong to the current year, i.e. it belongs to some future year, then the index is incremented. This scheme will be repeated until the smallest element of the current year is found or a whole year (all sub-lists) has been examined.

The Calendar Queue also performs resize operations to keep the sub-lists short, two to four elements in the best case. This is achieved 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 twice and half the current number of days per year. 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, but the cost is amortized by the cheap ordinary 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 situation is not handled explicitly by the original algorithm given in [3].

In this paper, a new method, the Lazy Queue, is presented for implementing the PES. The Lazy Queue is specifically tailored to handle the priority queue encountered 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 portion of the elements ordered. The elements are divided into three parts : a near future that is kept ordered, a far future that is partially ordered 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: in Section 2, the Lazy Queue is presented; Section 3 gives details on the self-adjustment policies of the queue; an analysis of expected performance and of the memory requirement and management are

found in Section 4; Section 5 introduces the experimental testbed; and some experimental results are presented in Section 6 where the Lazy Queue is compared to other queue implementations; Section 7 contains a discussion on a possible parallel implementation. Pseudo code for the basic operations on the Lazy Queue can be found in Appendix 1.

2 The Lazy Queue

The objective of this investigation was to design a priority queue implementation tailored to discrete event simulation. As a motivation for the approach we have some observations concerning how we, as human beings, maintain information on future events that we are to participate in (our pending event set). We can, as in the Calendar Queue [3], make analogies with the way 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. We can distinguish three intervals; a **near future (NF)**, where most of the events are known and thus we may keep a detailed schedule. A **far future (FF)**, where we expect most of the yet unknown events to 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 space 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 the scheduling of events until there is good reason to believe that most new events will occur beyond the period for which the detailed schedule is maintained.

In the present implementations of the Lazy Queue, the NF consists of a sorted array of elements and a data structure for insertion of new elements that fall into the NF part. This part of the NF is referred to as the insertion part of the NF. The FF consists of several months where each month contains an unsorted array of events. A QuickSort algorithm [12] is used for sorting the months that are transferred from the FF to the NF. The choice of data structures to implement the insertion part of the NF and the VFF will affect both the average and the worst case performance of the Lazy Queue. We have implemented two versions of the Lazy Queue, one that uses linked lists and another that uses binary heaps to implement these data structures. These implementations are referred to as Lazy Queue(linked list) (see figure 1) and Lazy Queue(binary heap).

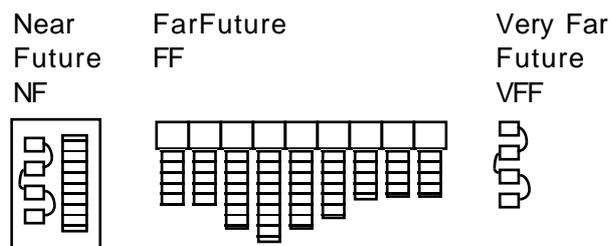


Figure 1. Schematic picture of a Lazy Queue(linked list).

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 the FF. Thus, in most cases, a new event is simply appended to one of the sub-lists of the FF. If the new element falls into the NF it is inserted into the the insertion part of the NF. A dequeue operation can either be a dequeue-min operation where the element with the smallest time stamp is dequeued or a dequeue of an arbitrary element. To dequeue an arbitrary element we have to search one of the sub-intervals of the queue. As we can expect the sub-intervals to contain only a small number of elements, this can be performed efficiently. When performing a dequeue-min 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 the following the term dequeue should be interpreted as dequeue-min.

In order to obtain good performance from any multi-list based queue, the width and number of the sub-intervals have to be chosen appropriately. As we expect the Lazy Queue to be less sensitive to distributions that are skew or have peaks than other multi-lists, a rectangular distribution is used as an approximation of the priority distributions. Using this approximation, the length of the months² is initially 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 **ElementsPerMonth** (Expected number of elements per Month). The actual average number of elements that fall into a month is called **AverageElementsPerMonth**. The number of months is chosen so that only a small number of elements will fall into the VFF. We call the length of a month **LengthOfMonth** (all months have equal length) and the number of months **NumberOfMonths**. Resize operations are introduced to adjust the queue to dynamic changes in both the queue size and the distribution of the elements. These operations are discussed in Section 3.

The reasons to believe that the Lazy Queue will render good performance can be summarized as:

- (1) most of the operations on the queue will access arrays with the indices known when the operation starts, giving constant access times for these operations;
- (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);

² 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.

(3) the fact that we can use a sorting method with a $O(\log(n))$ per element behaviour will reduce the sensitivity to skewness or peaks in the priority distribution compared to other multi-list based priority queue implementations, e.g. the Calendar Queue.

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 infrequent in most cases.

3 Adjustment of the queue

The performance of all multi-list structures depends heavily on an appropriate choice of the number and length of the sub-intervals. As we can expect both the queue size and the distribution of the elements to change dynamically, it is crucial for the queue to be able to adapt itself to such changes. We call these operations *resize operations*. The Lazy Queue has to be able to adjust the length of the NF as well as LengthOfMonth and NumberOfMonths. Normally LengthOfMonth and NumberOfMonths are increased or decreased by a factor of 2. In the following sections we introduce the criteria that are used to detect when it is necessary to perform resize operations, how these criteria are tested and how the resize operations are performed.

3.1 Resize criteria

A set of parameters and criteria must be defined in order to make decision on 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 ElementsPerMonth. The selected parameters are:

- 1 ElementsInNF, the number of elements present in the insertion part of the NF
- 2 ElementsInVFF, the number of elements present in the VFF
- 3 AverageElementsPerMonth, the actual average number of elements in each month
- 4 ElementsInFarHalfFF, the number of elements that are present in the upper half of FF³, we call this part of the queue FarHalfFF
- 5 the height and width of peaks detected in the priority distribution

Based on these parameters we can define a series of criteria that should be met. If they are not all met, a resize operation should possibly be performed. These criteria are:

³ 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.

- 1 $AverageElementsPerMonth \in [LowerBoundAEPM, UpperBoundAEPM]$
 where $LowerBoundAEPM$ and $UpperBoundAEPM$ are the boundaries of the actual average number of elements that are present in the months of FF
- 2 $ElementsInNF < MaxElementsInNF$
 where $MaxElementsInNF$ is the upper threshold for the number of elements present in NF
- 3 $ElementsInVFF < MaxElementsInVFF$
 where $MaxElementsInVFF$ is the upper threshold for the number of elements present in VFF
- 4 $ElementsInFarHalfFF + ElementsInVFF > MaxElementsInVFF/2$
 this criterion is set to ensure a good utilisation of the FF
- 5 the average number of elements present in the $PeakWidth$ first month of the FF $< PeakThreshold$
 $PeakWidth$ is defined as the minimum of a constant number and half the number of months in FF; $PeakThreshold$ is a constant number

Criteria 1–3 are strong criteria in the sense that they normally never should be violated. Criterion 3 may however not always be met in cases where the elements in the queue are strongly concentrated to the ends of the interval (towards NF and VFF). In such a case it may be necessary to violate criterion 3, and implicitly criterion 4, to be able to meet criterion 1. Violation of Criteria 4 or 5 only results in resizes if the other criteria can be met after the resize. This resolves possible conflicts between Criterion 4, 5 and the other criteria that could occur in some cases. Such a case could be when the elements are concentrated to the front and to the end of a long interval.

The reason for introducing Criterion 5 is that we can detect peaks close to the NF and spread these elements over several months before these peaks have propagated into the NF (this is achieved by halving $LengthOfMonth$ and adjusting $NumberOfMonths$).

3.2 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 its elements to the NF, Criteria 1,4,5 are checked and implied resize operations will be performed. We must also check that these resize operations do not result in a state violating Criteria 2 or 3.

When performing an enqueue operation where the element falls into NF or VFF, Criteria 2 and 3 are checked respectively. The resulting resize operations are performed so that Criteria 1 and 4 are met (since this only involves halving LengthOfMonth and/or doubling NumberOfMonths, Criterion 5 is not affected). In a build up phase, when no dequeue operation yet has been performed, Criteria 1 and 4 are also checked and if they are violated LengthOfMonth and NumberOfMonths will be directly recomputed.

3.3 Resize operations

The basic resize operations performed are:

- 1 Halve or double NumberOfMonths
- 2 Halve or double LengthOfMonth

Changes to LengthOfMonth and NumberOfMonths must often be combined, i.e. when LengthOfMonth is halved NumberOfMonths often has to be doubled. It should be noticed that it may happen that NumberOfMonths and/or LengthOfMonth have to be adjusted in smaller or greater steps to ensure good performance of the queue.

Halving LengthOfMonth 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 adjust NF, we calculate $\text{ElementsInFarHalfFF} + \text{ElementsInVFF}$. If Criterion 4 is not met, in this case $\text{ElementsInFarHalfFF} + \text{ElementsInVFF} \leq \text{MaxElementsInVFF}/2$, so it is not necessary to increase NumberOfMonths. Instead, the elements in FarHalfFF are moved to VFF. Otherwise NumberOfMonths is recalculated, which normally results in a doubling of NumberOfMonths. Finally, each month is split into two, and the size of the array holding the elements in each month is adjusted.

Doubling LengthOfMonth is performed by first pairwise joining the months and moving them into the lower half of the FF. After joining the months, before VFF has been adjusted, FarHalfFF is empty. Hence we can check Criterion 4 by checking if $\text{ElementsInVFF} < \text{MaxElementsInVFF}/2$, if this is the case, it is possible to halve NumberOfMonths to achieve a good utilization of the FF part. Otherwise 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.

The resize operations could be implemented as, if the number of months is increased creating some $O(n)$ months, and then copying $O(n)$ elements into these new months. Thus the number of operations needed for the resize operations are $O(n)$ except for the case of halving the LengthOfMonth. This case could, if the elements are concentrated to some months, require $O(n \cdot \log(n))$ operations since it would

require a nearly complete sorting of the queue. This sorting effort is not wasted since the sorted parts do not have to be sorted again, if necessary, however they may be merged with other sorted parts. It should be noticed that the number of operations needed in subsequent halvings of LengthOfMonth would be lower as the queue would be nearly sorted.

The first resize operation performed on the Lazy Queue should be treated as a special case. In this case the LengthOfMonth and NumberOfMonths are directly calculated from the number of elements in the queue, the priority interval spanned by the elements and ElementsPerMonth, using a rectangular approximation of the actual distribution.

When comparing these strategies to those proposed for the Calendar Queue [3], it can be seen that the Lazy Queue adapts the queue to changes in the priority distribution without changes in the queue size. This case is not considered by the original Calendar Queue approach as proposed in [3]. The resize operations in the Lazy Queue can also be expected to occur less frequently than they will occur if this case were to be treated by the Calendar Queue. This is because the average length of the sub-lists in the Calendar Queue will be near 2, where as AverageElementsPerMonth can vary within a wider range without affecting overall performance of the Lazy Queue (see Figure 3). However, by reintroducing an overflow structure we have also reintroduced the problems that will follow with it.

4 Performance considerations

For the Lazy Queue to be of practical use we must know what resources in the form of time and memory it can be expected to use.

4.1 Effects of resize operations on performance

How the queue can adapt itself to the actual distribution of the elements and how frequently resize operations have to be performed to accomplish this, will largely affect the performance of the queue.

Are there distributions to which the Lazy Queue can not successfully adapt itself? Here successfully means that resize criteria 1–3 can be met. Such failure of adjustment will occur when the approximation of the actual distribution to a rectangular distribution fails. This implies that the distribution of the elements must be concentrated towards the ends of the interval (towards NF and VFF). In such a case, no resize operations will be performed and the performance of the queue will be determined by the performance of the data structures used to implement the NF and the VFF parts. That is for the Lazy Queue(linked list) the performance per access will be $O(n)$ and for the Lazy Queue(binary heap) the performance will be $O(\log(n))$.

How frequently are resize operations invoked? To answer this question we consider distributions to which the queue eventually can adapt itself, i.e. distributions for which the rectangular approximation holds. The queue size may be approximately constant, a 'steady-state' case, or it may vary. The distribution from which the priorities of the new elements are generated may change as a function of the time or the number of priorities drawn from it or it may be constant with respect to these parameters. We call the first category of distributions 'compound' and the latter category 'non-compound'. (See also Section 5.1)

Non-compound distributions: In a steady state case no resize operations will be performed as the approximation to a rectangular distribution is successful. In the case where the number of elements in the queue is varying, the resize operations will be dominated by the operations due to changes in the `AverageElementsPerMonth`. This is explained by the fact that once a good enough approximation of the actual distribution is established other resize criteria will be violated very seldomly. This is accomplished in most cases for fairly moderate queue sizes. The resize operations inflicted by criterion 1 will thus at most take place when the size of the queue has changed by a factor 2. This allows us to use the same analysis as Brown used for the Calendar Queue [3]. According to that analysis each element is only involved in at most 2 resize operations on the average. Under the assumption that the rectangular distribution approximation is successful we get a time complexity of $O(n)$ for a resize operation. When this is amortized over n elements we get an $O(1)$ access time. In the worst case, if most elements have fallen into a few months and a halving of the `LengthOfMonth` is performed, the resize operation takes $O(n \cdot \log(n))$ time resulting in $O(\log(n))$ access time.

Compound distributions: In this case the number of resize operations could increase rapidly. The new elements could all fall into the NF, VFF or some single months. In the case of a varying number of elements, resize operations could occur on every `MaxElementsInNF/2`, `MaxElementsInVFF/2` or `PeakThreshold*PeakWidth/2` access. These operations could include series of halving the `LengthOfMonth`. It should be noticed that after a halving of the `LengthOfMonth` the queue will be completely sorted greatly reducing the need for sorting in subsequent halvings of the `LengthOfMonth`. Thus the access time could be $O(n)$. We, however, believe that priority distributions that consistently changes are unlikely to occur in real simulations. For the Lazy Queue(linked list) the worst case time complexity could be constrained if an upper limit on the frequency of performed resize operations is imposed. If such a limit is imposed the queue could in a worst case degenerate into what is essentially a data structure equivalent to that of the NF insertion part and the VFF. In such a case it is possible to achieve a worst case behaviour of $O(\log(n))$. Such a limit on the resize frequency has not been considered in the current implementation of the Lazy Queue. None of the experiments that have been performed have implied that this should be necessary to do.

Here we could also note that the worst case behaviour of the Calendar Queue also is $O(n)$. If the distribution changes so that nearly all elements fall into one sub list. Then the queue would degenerate to a linked list. This could happen also if the number of elements in the queue is constant. Even if resize strategies for the Calendar Queue were developed that cover these cases the worst case would still have a time complexity of $O(n)$. This may happen, because the distribution of the elements could continuously change so that the elements either fall into a few sub lists or into different years. These cases can not be alleviated by performing resize operations more frequently since they also would have a time complexity of $O(n)$. They would involve some operation on every day in the queue and the number of days in the queue is approximately $n/2$.

4.2 Memory considerations

In the worst case memory usage, the months of the FF are only half filled. This results in a maximum memory requirement of $2 * n * \mathcal{E} + f(n)$ where \mathcal{E} is the storage size in bytes of an element in the queue and its associated priority, and $f(n)$ is the storage in bytes for data structures other than the arrays of the months in FF. These data structures essentially consists of an array holding information on the months of the FF, (such as size of the array of elements in the month, next free slot etc). Since the number of months in the FF is bounded by $n / \text{LowerBoundAEPM}$, $f(n)$ is proportional to $O(n)$. Experimentally, $f(n)$ was observed to be approximately equal to $n * \mathcal{E}$ for all of the experimental distributions and queue sizes, giving a total memory requirement of $3 * n * \mathcal{E}$. This can be compared to the Implicit Binary Heap which has a worst case of $2 * n * \mathcal{E}$. It should also be noted 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 to enhance performance and avoiding fragmentation. Thus actual memory requirements in practice may be somewhat greater than the $3 * n * \mathcal{E}$.

5 Performance measurements

When performing the performance measurements decisions have to be made regarding: How the measurements should be done. What experiments, i.e. what priority distributions to use. Implementation issues regarding the NF,FF and VFF parts of the Lazy Queue and associated parameters. These decisions are crucial for the validity of the experimental results.

5.1 Priority distributions

There exists a wide variety of priority distributions. The problem is to make a good selection of test distributions to use as priority generators. The selection should reflect typical distributions that occur in real simulation models as well as provide an adequate testbed for the queues. We have used five non-compound distributions, Rect, NegTriang, Triang, Camel and Exp, and one compound distribution that we call Change. 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 and at the end of a queue respectively.

The Camel distribution [13] models a distribution with peaks defined on a closed interval. Besides the interval bounds, a Camel distribution is dependent on three parameters: the number of humps (n), the fraction of the probability mass distributed in the humps (m) and the fraction of the total width the sum of the hump widths occupies (w). In the rest of this paper, we assume that $n = 2$. Because m and w denotes fractions they belong to the interval $[0,1]$.

The Camel 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 by 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 was $[0,1000]$ if not otherwise stated.

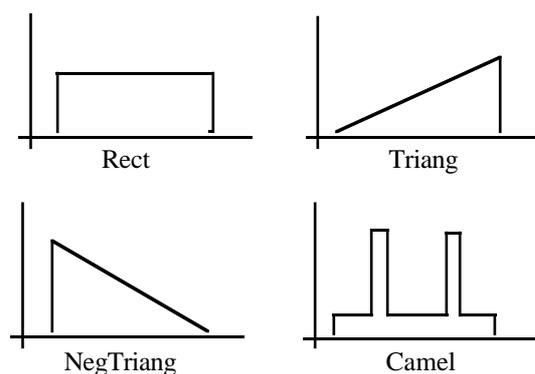


Figure 2. Bounded interval priority distributions used in the experiments.

Exp is an exponential distribution with a mean of 1. This distribution was primarily chosen because it is defined on an open ended interval $[0, \infty[$. In the following we collectively refer to the distributions Rect, Triang, NegTriang, Camel and Exp as ordinary distributions.

Change is a combination of two distributions. It takes three parameters: $\text{Change}(d1, d2, k)$ where $d1$ and $d2$ are priority distributions and k is an integer constant. It combines distributions $d1$ and $d2$ by interleaving sequences of k priorities drawn from distributions $d1$ and $d2$ beginning with $d1$. For instance, $\text{change}(\text{Exp}, \text{Triang}(90000, 100000), 10000)$ defines a combination of an exponential distribution with a mean of 1 with a triangular distribution defined on the interval $[90000, 100000]$ so that the first 10000 priorities are drawn from the exponential distribution, the next 10000 priorities from the triangular distribution and so on. The Change combination of distributions with various parameters is used to test the queues regarding the worst case behaviour and the sensitivity to compound distributions.

5.2 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 (dequeue-min) operation. The parameters we have varied, beside those that are specific to the Lazy Queue approach, 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 [8] and the latter is Up and Down. The 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 the equal number of deletes and inserts. Up and Down models a transient phase of a queue. The queue is loaded up to some queue size and then emptied again. Classic Hold and Up and Down represents two extremes and serve to show the bounds of the performance.

The queue sizes used range from small to very large sizes (20000). The interesting question here is: is the Lazy Queue generally applicable, usable for arbitrary sizes, or just for some particular queue size interval?

The experiments were performed on a Sequent Symmetry S81⁴ [14]. Actually, the machine is a multi-processor but this is of no interest as we are discussing sequential algorithms executing on one computation unit. Nevertheless, it has one very important feature which we have based the experiments

⁴ Sequent is a registered trade mark of Sequent Computer System, Inc.

on. Each processor is equipped with a register based clock, which enables timings with microsecond resolution and accuracy.

Each queue operation was timed separately and the results were accumulated in a table for statistics collection. This is in contrast to the 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 operations (see for example [8]). Of course this is necessary if the clock (as in most computers) has low resolution, but it introduces inaccuracy due to loop overhead.

All performance measurements were performed with the needed memory pre-allocated. Thus effects of the performance of the under-lying memory management system were eliminated from the measurements.

All code was written in the C programming language and the queue specific parameters used for the Lazy Queue were those described in section 5.4.

5.3 Implementation of NF, FF and VFF

As discussed earlier we have implemented two versions of the Lazy Queue, one where the insertion part of the NF and the VFF are implemented as linked lists, Lazy Queue(linked list) and a Lazy Queue(binary heap) where these data structures are implemented as binary heaps. The tradeoff between these two implementations are a slightly better average case and a worse worst case for the linked list version contra a worse average case and a better worst case for the binary heap version.

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 on demand and deallocated as soon as possible. The initial allocation size is chosen as ElementsPerMonth 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 (ElementsPerMonth, 2*ElementsPerMonth, 4*ElementsPerMonth etc) to avoid memory fragmentation.

As sorting method, a Quick Sort that resorts to Insertionsort for short sub-lists has been chosen [12]. This method gives acceptable performance both for short and long sub-lists. It has an average time complexity of $O(\log(n))$ per element.

5.4 Parameters of the Lazy Queue

There are a number of parameters to the Lazy Queue that have to be determined by the user. These parameters are: ElementsPerMonth, LowerBoundAEPM, UpperBoundAEPM, PeakThreshold, PeakWidth, MaxElementsInNF, MaxElementsInVFF. It is possible to formulate some guidelines for the choice of these parameters. A series of experiments were performed to provide a basis for choosing these values as well as to evaluate how they may influence the performance of the queue.

The following guide-lines can be formulated for the choice of the parameters:

- UpperBoundAEPM should not be greater than an approximate limit of the size for which the data structure implementing the insertion part of the NF has good performance. It should not be too small since this could impose more frequent resize operations.
- $\text{UpperBoundAEPM} \leq \text{PeakThreshold} \leq \text{MaxElementsInNF}$
- MaxElementsInNF should not be much greater than the approximate limit of the size for which the data structure implementing the insertion part of the NF has good performance. It should neither be too small, since that could increase the number of halving of the month length that have to be performed.
- MaxElementsInVFF should not be much greater than the approximate limit of the size for which the data structure implementing the insertion part of the VFF has good performance. It should not be too great since that could slow down the adaption of the queue to compound priority distributions.

The most important parameters are ElementsPerMonth and its boundaries LowerBoundAEPM and UpperBoundAEPM. An experiment was designed where a Lazy Queue(linked list) with the resize capabilities disabled were tested for different values of ElementsPerMonth. A fixed queue size and a rectangular distribution was used. LengthOfMonth and NumberOfMonth were calculated in advance so that AverageElementsPerMonth could be expected to be equal to ElementsPerMonth. The result of this experiment is shown in figure 3. It tells us that a wide range of ElementsPerMonth could be used with only little effect on the performance of the queue. This also indicates that the queue could be expected to be little sensitive to inhomogeneous distributions. That is, distributions for which the number of elements in the months of FF vary from only a few elements up to nearly a thousand elements.

Mean access time μ s

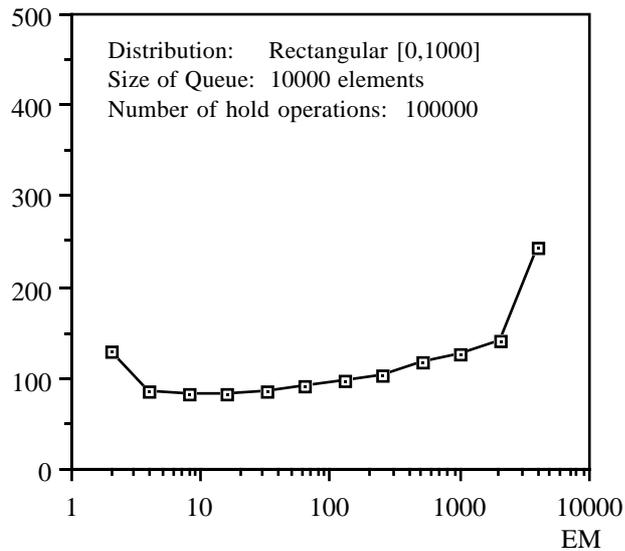


Figure 3. Access times of the Lazy Queue for various choices of ElementsPerMonth.
(Expected number of elements per month)

For the choice of the other parameters a series of experiments has been conducted. In these experiments one parameter at a time was varied keeping the other at fixed default values. The default values and the ranges for each parameter are found in table 1. The priority distributions used were the set of non-compound distributions and $\text{Change}(\text{Exp}, \text{Triang}, 1000)$ and $\text{Change}(\text{Triang}, \text{Exp}, 1000)$. Steady state experiments were performed where the queue size was varied from 1000 elements up to 20000 elements. Changes in the access time was only observed in a few cases for the Lazy Queue(linked list) when MaxElementsInNF and MaxElementsInVFF were varied. The results of these experiments are found in figures 4 and 5. The absence of variations in the access time can be explained by the fact that for the non-compound distributions there are no changes or huge peaks and the rectangular approximation of the distribution is hence valid. Thus only a few resize operations are performed once the LengtOfMonth and NumberOfMonths have been given appropriate values. This is done in the first resize operation.

	Lazy Queue (linked list)		Lazy Queue (binary heap)	
	Default value	Tested values	Default value	Tested values
MaxElementsInVFF	256	[128,256,...2048]	2048	[512,1024,...8192]
MaxElementsInNF	1024	[128,256,...2048]	2048	[512,1024,...8192]
LowerBoundAEPM	4	4	4	4
UpperBoundAEPM	64	[64,128,...2048]	512	[64,128,...4096]
PeakThreshold	64	[64,128,...2048]	2048	[256,512,...4096]
PeakWidth	8	8	8	8

Table 1. Default values for different parameters of the Lazy Queue

Figure 4 shows that if a too small value is chosen for MaxElementsInNF, the number of elements falling into the NF part would often exceed this maximum value generating frequent resize operations that could degrade the overall performance. From figure 5 one can see that a too high value of MaxElementsInVFF could degrade the performance due to the increased length of the linked list implementing the VFF.

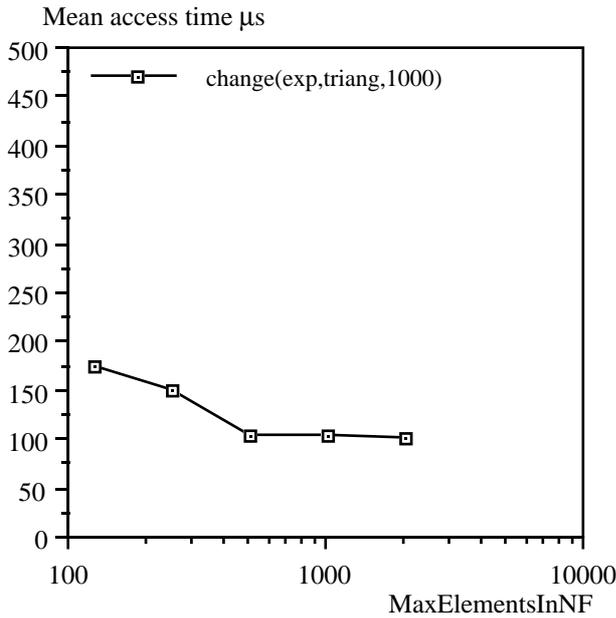


Figure 4. Lazy Queue(linked list) dependency on MaxElementsInNF

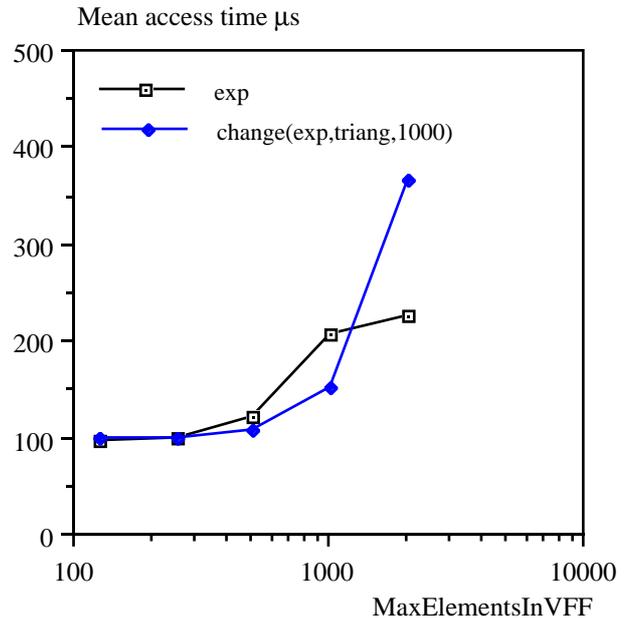


Figure 5. Lazy Queue(linked list) dependency on MaxElementsInVFF

In the experiments in the following section we have used the set of default values given in table 1 for our implementations of the Lazy Queue. The implementations have not imposed an upper limit on the

frequency of resize operations. For pseudo code implementations of the basic queue operations (enqueue and dequeue) on a Lazy Queue(linked list) see Appendix 1.

6 Experimental results

Experiments were performed to measure access times of the Lazy Queue, Calendar Queue and Implicit Binary Heap for both non-compound distributions (Exp, Triang, Negtriang, Rect, Camel) and compound distributions. In the steady state experiments the number of hold operations performed was five times the queue size. This method provides a constant ratio of the number of elements in the queue and the hold operations performed. Consequently, it reduces the deformation effect of successive hold operations on the actual distribution of the queue elements. If a constant number of hold operations were performed, as in [8], this would affect the smaller queue sizes more than the larger.

6.1 Non-compound distributions

The first set of experiments tests the sensitivity of the Lazy queue to various non-compound priority distributions and queue sizes. Figures 6a-b, 7 and 9a-b show the results of the steady state experiments.

The Binary Heap has an access time that is essentially independent of the actual distribution used [8]. Therefore, only figures for the Binary Heap with a rectangular distribution are shown (dashed lines in the figures) for the purpose of comparison.

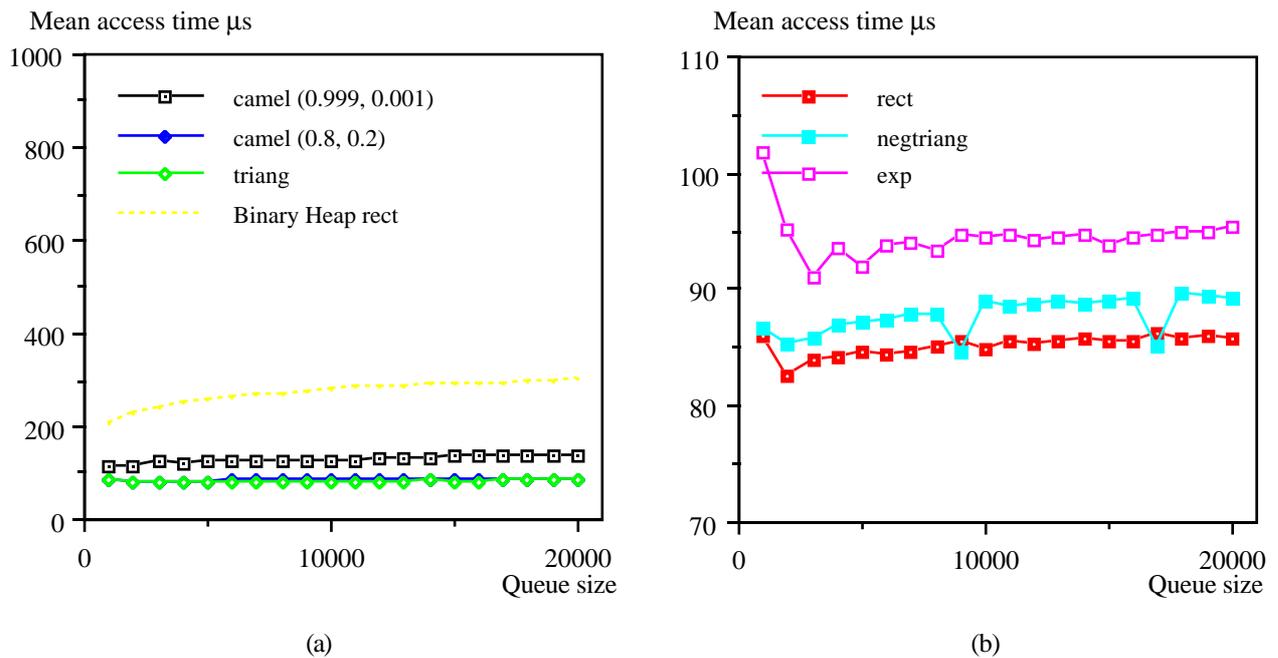


Figure 6. Performance of Lazy Queue (linked list) in Steady State experiments.

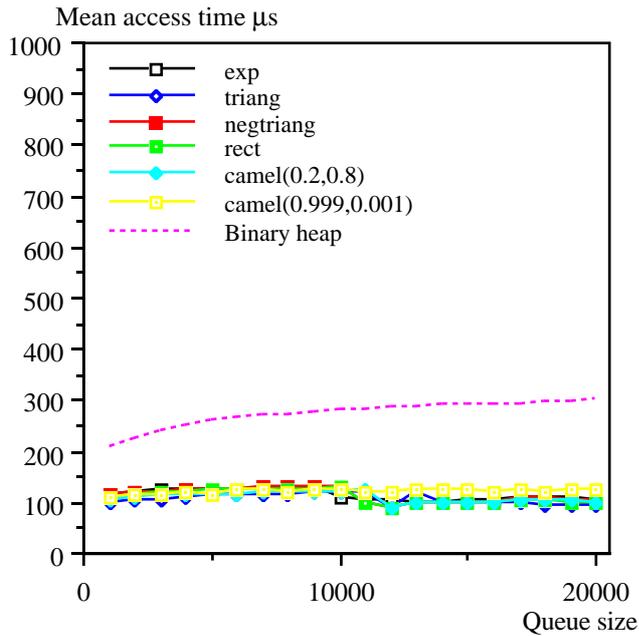


Figure 7. Performance of Lazy Queue(binary heap) in Steady State experiments

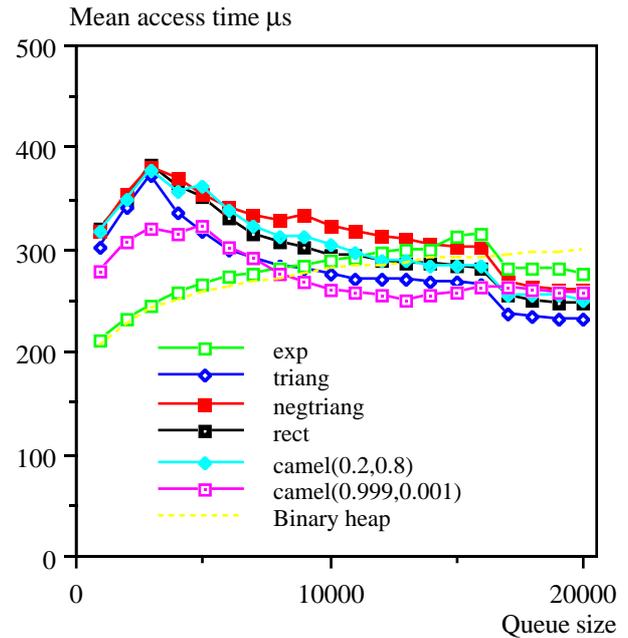


Figure 8. Performance of Lazy Queue(binary heap) in Up and Down experiments

As can be seen from Figures 6a-b and 7, the Lazy Queue illustrates a stable behavior also for distributions with high peaks. Compared to the Binary Heap (dashed line), the Lazy Queue performs well. If we compare these figures with those for the Calendar Queue, Figures 9a-b, we see that the Lazy Queue exhibits a more stable performance and compares favourably in most cases. The Calendar Queue approaches and surpasses the Binary Heap for some distributions. It is notable that this occurs although the distribution does not change over time and even for the triangular distribution. 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 by examining the first few elements present in the queue. In the cases of the triangular and the camel distribution the Calendar Queue makes less accurate calculations of the day length as the queue size grows. Consequently, many elements fall into a few sub-lists where the insertion time approaches an $O(n)$ behavior. At the same time, many consecutive elements fall into separate years, which imposes a linear search through the queue that also has an $O(n)$ time complexity. The leaps in the curves of the Calendar Queue, Figures 9a-b and 11, coincide with the resize points.

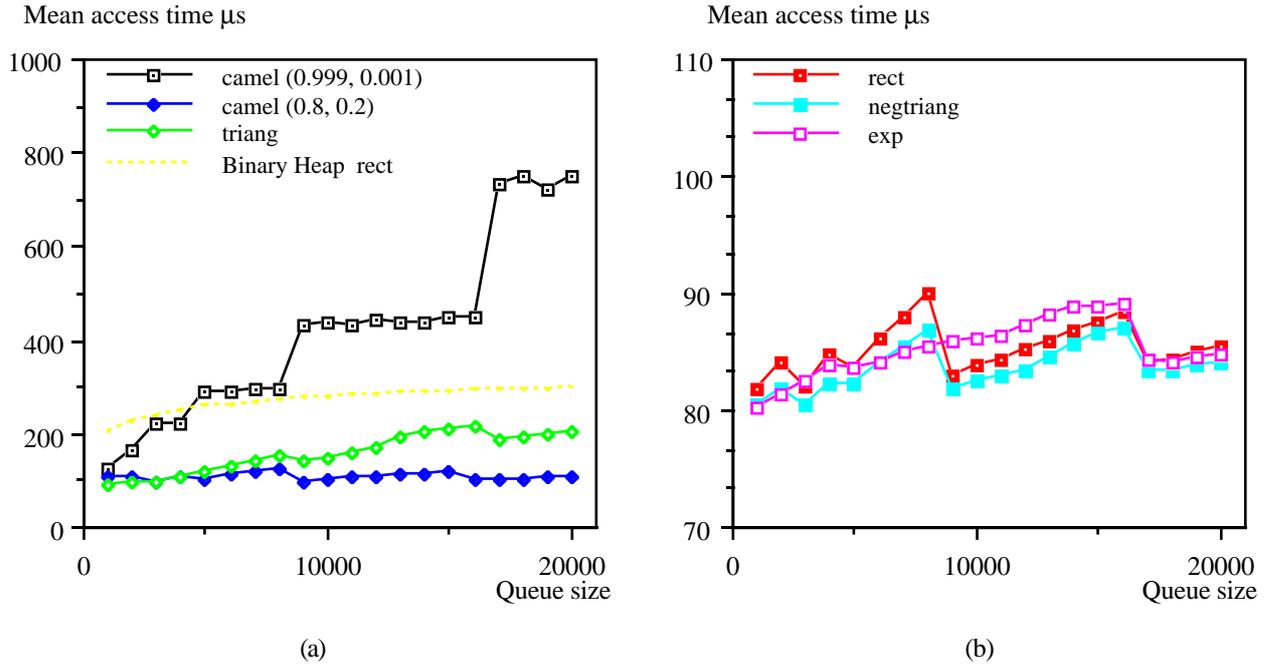


Figure 9. Performance of Calendar Queue in Steady State experiments.

To be able to make a fair judgement, we must also consider the access time during transient stages. We have measured performance of the queue in the transient stages by performing Up and Down experiments. As can be seen from Figure 10, the performance of the Lazy Queue(linked list) approaches that of the Binary Heap when the camel(0.999,0.001) distribution is used. This depends on the first approximations made, when the queue contains relatively few elements, not being accurate enough. The peaks in the distribution are detected only when criterion 5 is checked in the dequeue operations. This imposes resize operation where the LengthOfMonth is halved. Figure 8 shows that the Lazy Queue(binary heap) has a stable performance. It performs slightly worse than the Implicit Binary Heap. however, this is what could be expected as most elements initially fall into the VFF, implemented by a binary heap. This accounts for a $O(\log(n))$ access time that is further increased by the resize operations. After the initial resize operations, the access time will be near constant. Thus, the access time will decrease as the queue size is increased and the cost for the initial resizes are amortized over more accesses. When we compare performance of the Lazy Queue with the Calendar Queue (Figure 11), we see that the time to build the Calendar Queue shows more variation. The access time for the Calendar Queue sometimes grows rapidly, even passing the Binary Heap in some cases. This can be explained by the same reasons as for 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 $O(n^2)$ leading to $O(n)$ performance when amortized over n operations.

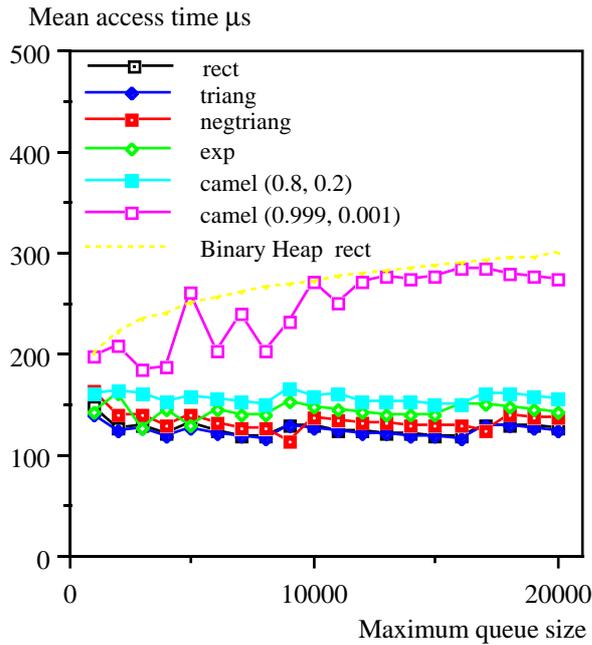


Figure 10. Performance of Lazy Queue(linked list) in Up and Down experiments.

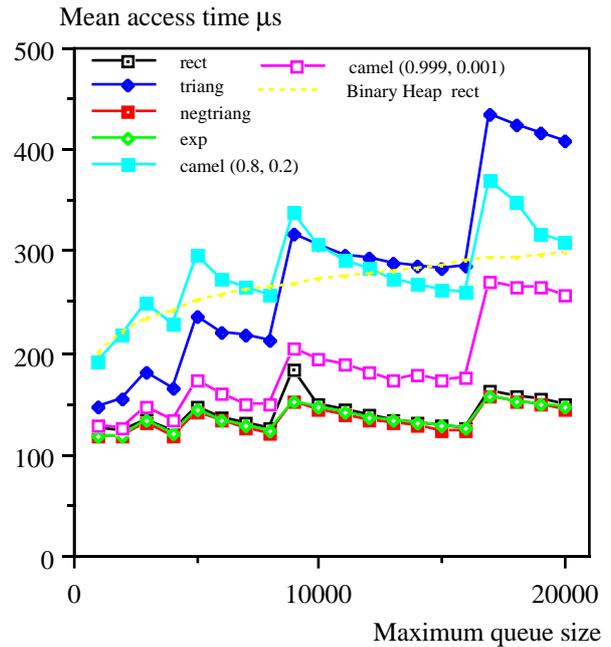


Figure 11. Performance of Calendar Queue in Up and Down experiments.

6.2 Compound distributions

This set of experiments tests the sensitivity to various compound priority distributions and queue sizes. In these experiments two sets of distributions were used. In the first set, an exponential distribution with mean 1 has been combined with a triangular distribution defined on the interval $[0,1000]$. The results of these experiments are shown in Figures 12a, 13a and 14a. The second set of experiments combines the exponential distribution with mean 1 with a triangular distribution defined on the interval $[90000, 100000]$. The results of this set of experiments are presented in figures 12b, 13b and 14b.

If these experiments were to be described in terms of a simulation; the first set would be equivalent to a simulation where the time between two events changes from seconds to hours; in the second set it changes from seconds to several days. Situations like these may arise for instance in battle field simulations. In such a simulation, the activity may be very low between battle periods, but it will be very high when a battle breaks out. The second set of experiments is also used to test the worst case performance of the queues.

As can be seen from figures 12a, 13a and 14a all queue implementations perform well for the first set of distributions. If we consider the figures 12b, 13b and 14b we can see that only the queue implementations based on binary heaps, Lazy Queue(binary heap) and the Implicit Binary Heap, shows good performance. In the case of the Lazy Queue(linked list), it encounters a series of halvings of the LengthOfMonth in the up and down experiment for the Change(Triang(90000,100000),Exp(1),10000)

distribution. It then exhibits $O(n)$ behaviour as predicted in the analysis. The Calendar Queue shows a rather unpredictable performance for the steady state experiment for the Change(Exp(1), Triang(90000, 100000), 10000) distribution. This can, in part, be explained by the fact that it adjusts the day length to the Exp(1) distribution and never is able to adjust it to the change in the distribution.

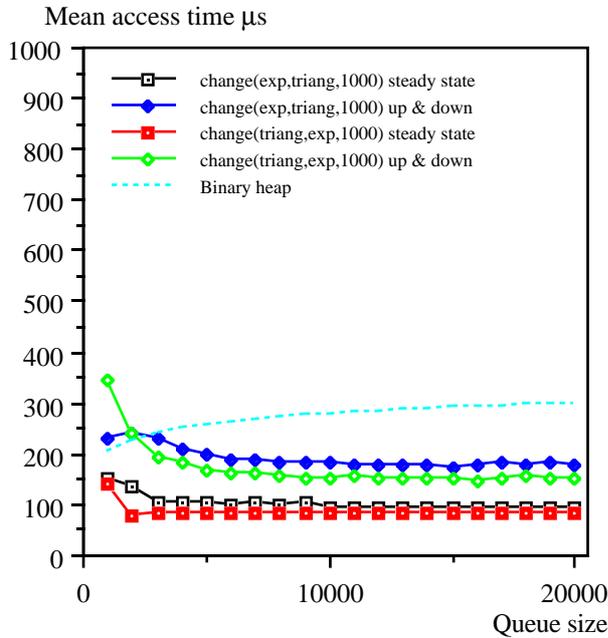


Figure 12a. Performance of Lazy Queue(linked list) for compound distributions.

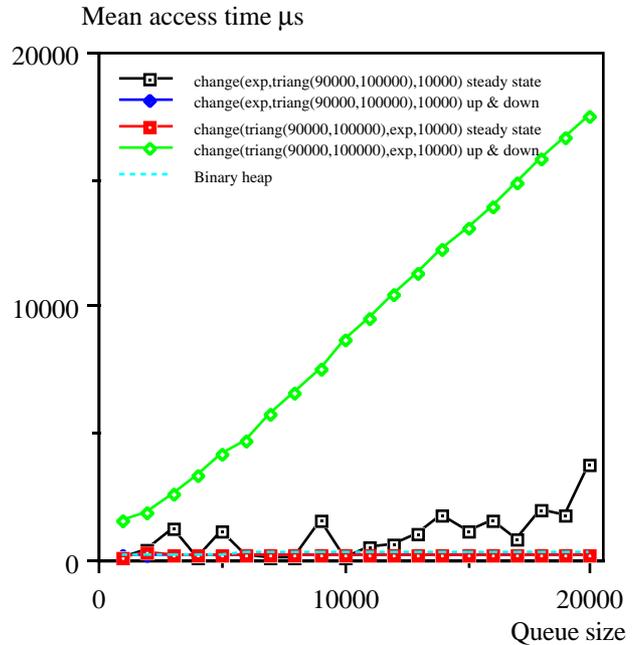


Figure 12b. Performance of Lazy Queue(linked list) for compound distributions.

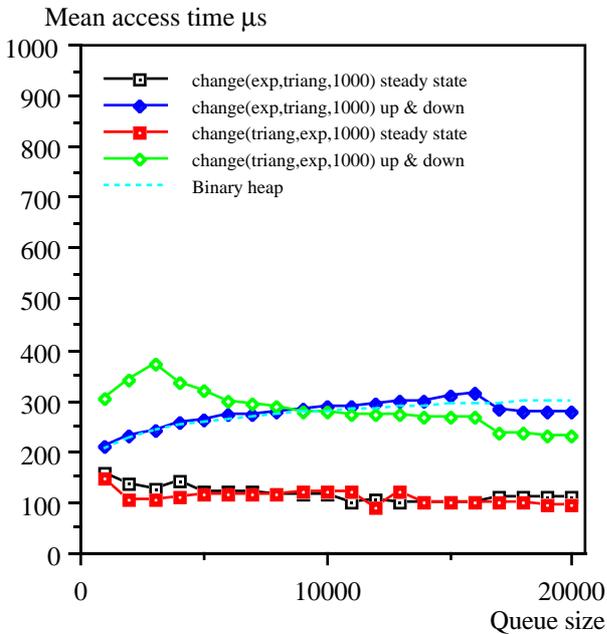


Figure 13a. Performance of Lazy Queue(binary heap) for compound distributions.

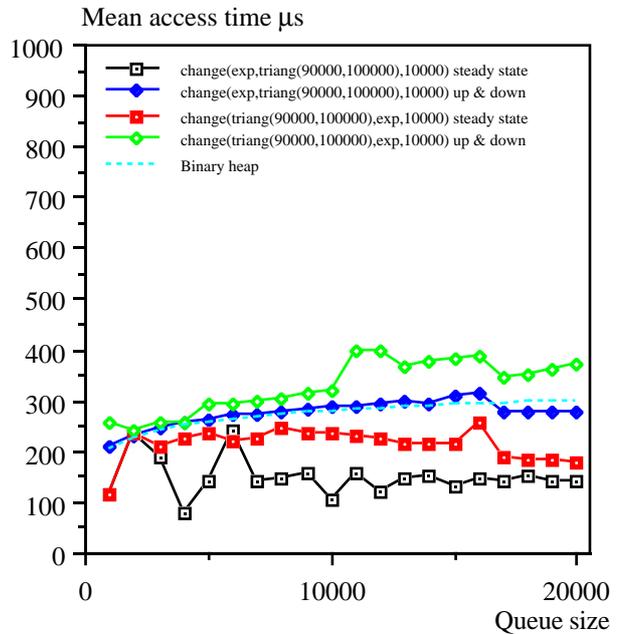


Figure 13b. Performance of Lazy Queue(binary heap) for compound distributions.

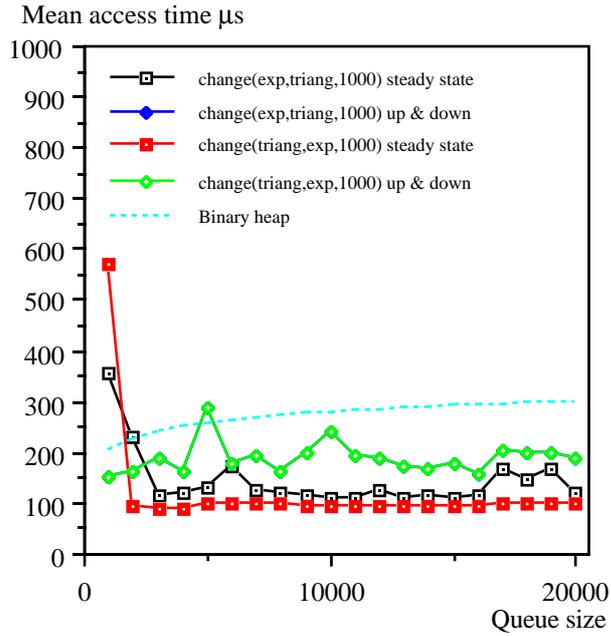


Figure 14a. Performance of Calendar Queue for compound distributions.

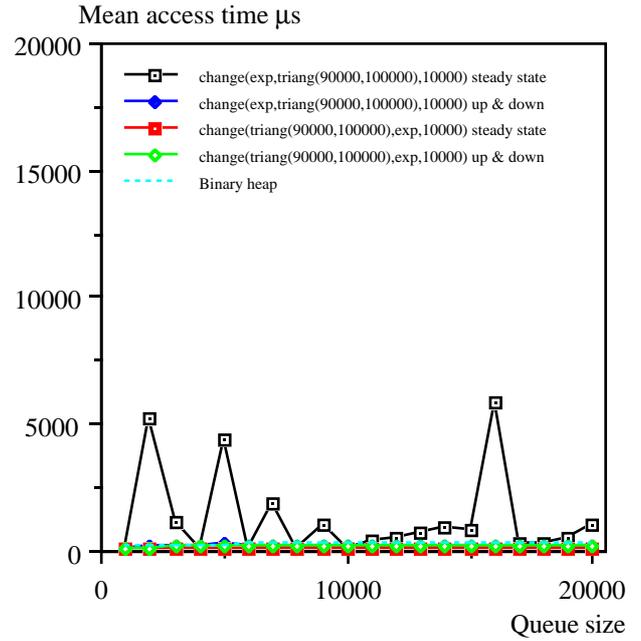


Figure 14b. Performance of Calendar Queue for compound distributions.

7 Future work

In the recent years, much interest has been devoted to parallel simulation schemes and their performance. Several concurrent implementation of the PES have appeared in the literature, e.g. [15, 9, 5]. Most of these implementations are based on the Implicit Binary Heap. However, an efficient implementation of the PES that can be used by the parallel simulation schemes, e.g. Time Warp, is still an open question. A simple analysis of the possible best performance that can be expected from the parallel binary heap implementations can easily be derived. In parallel binary heaps, all operations are performed in a top-down manner. There are two implications from this: First, we can note that the enqueue operation, that is otherwise performed bottom up will be slightly more compound, reducing the performance. Second, all operations traverse the heap, level by level, in a stepwise manner. The time spent at each level is approximately equal and constant. We denote this time by τ . The accesses are pipelined through the heap as all operations initially have to lock the topmost level. This gives a best possible performance limited to one completed operation every τ time units. This is true only if a level of parallelism can be sustained that allows $\lfloor \log(N)+1 \rfloor$ operations (on a heap with N elements) to be on-going in parallel.

We have implemented a sequential version of the parallel binary heap suggested by Rao and Kumar [5] leaving out the *lock* operations. Experimental results shows that this heap implementation has an access time of approximately $27\mu\text{s}$ on our Sequent Symmetry, while the mean access time of the Lazy Queue is approximately $100\mu\text{s}$ on the same machine. Two conclusions can be drawn from this result:

First, if it is not possible to sustain a level of parallel access to the PES that is higher than $\lfloor \log(N)+1 \rfloor / 3$ on the average, then it is better to use a fast sequential algorithm such as the Lazy Queue. Second, a parallel implementation of the Lazy Queue would perform better than the parallel binary heap, if it could exploit a degree of parallelism greater than three. This degree of parallelism can be achieved, for instance by allowing one dequeue operation to be overlapped with several enqueue operations that access different months of the FF.

Our initial investigation also indicates that the Lazy Queue is a good candidate to be used in the optimistic Time Warp parallel simulation paradigm. We intend to investigate these possibilities further.

8 Conclusions

We have presented a new data structure, the Lazy Queue, for implementing the pending event set encountered in discrete event simulation. The Lazy Queue is based on conventional multi-list structures but has some distinct differences. The Lazy Queue is divided into three parts: (1) the near future, NF, where elements close to the actual simulation time are 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. One interesting feature of the Lazy Queue 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 sorting and transferring the months from the FF to the NF on demand. As the elements in the NF and FF are stored in arrays, standard sorting techniques can be employed, which further improves the performance. A general adjustment strategy for the queue has been implemented, enabling it to 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 features of the applications.

An analysis of the Lazy Queue indicates that we can expect it to have a near constant access time for many distributions normally encountered in practical simulations. The analysis also shows that it is possible to restrict the worst case performance to $O(\log(n))$, if the insertion part of the NF and the VFF are implemented by binary heaps. This can be compared with the linked list based implementations, such as the Lazy Queue(linked list) and the Calendar Queue, where the access times are near constant for many distributions but $O(n)$ in the worst case.

The experimental results, where enqueue and dequeue times have been measured for a number of different priority distributions and queue sizes both in the steady state and Up and Down experiments, verify the analytical results. The same experiments were also performed on two other queue implementations: the Implicit Binary Heap and the Calendar Queue. The experiments show that the Lazy

Queue(linked list) is a good choice in the situations where the distributions do not change drastically over time. The Lazy Queue(binary heap), on the other hand, has showed a stable performance and is a good choice when nothing is known about the actual priority distribution.

There are some drawbacks associated with the Lazy Queue. It trades memory for efficiency and hence it requires more memory space than the other queues tested. The code to implement the Lazy Queue is rather more complex than for the other queues tested. There are also a number of parameters that have to be determined by the user. The best choice of these parameters has only been experimentally determined. It is possible to employ a technique which adjusts the parameters dynamically. However, the cost and effectiveness of such a technique needs further investigation.

References

- [1] Blackstone, J.H., G.L. Hogg and D.T. Philips, "A two list synchronization procedure for Discrete Event Simulation", *Comm. ACM Vol. 24*, No. 12, pp 825-829, Dec 1981.
- [2] Bently, J. "Programming Pealls", *Comm. ACM Vol. 28*, No 3, pp 245-250, Mar. 1985.
- [3] Brown, R. "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem", *Comm. ACM Vol. 31*, No. 10 , pp. 1220–1227, Oct. 1988.
- [4] Jit Biswas and J. C. Browne, " Simultaneous update of Priority Structures", *Proc. of the 1987 Int. Conf. on Parallel Processing*, pp 17-21, Aug. 1987.
- [5] Rao V. N. and V. Kumar, "Concurrent Access of Priority Queues", *IEEE trans. on Computers Vol. 37*, No. 12 , pp 1657-1665, Dec. 1988.
- [6] Ayani, R. "Performance of priority-queue implementations on shared memory multi-processor computer systems", Tech. Rep. TRITA–CS–8705, Dept. of Telecommunication and Computer Systems, The Royal Inst. of Technology, Stockholm, 1987.
- [7] Franta W.R. and K. Maly, "A comparison of heaps and the TL structure for the Simulation Event-Set", *Comm. ACM Vol. 21*, No. 10, pp. 873-875, Oct. 1978.
- [8] Jones, D.W. "An empirical comparison of priority-queue and event-set implementations", *Comm. ACM Vol. 29*, No. 4 , pp. 300–311, Apr. 1986.
- [9] Jones, D.W. "Concurrent Operations on Priority Queues", *Comm. ACM Vol. 32*, No. 1, pp. 132-137, Jan. 1989.
- [10] McComark, W.W and R.G. Sargen, "Analysis of Future Event-Set Algorithms for Discrete Event Simulation", *Comm. ACM Vol. 24*, No. 12, pp 801-812, Dec 1981.
- [11] Stasko J.T. and J. Scott Vitter, "Pairing Heaps: Experiments and Analysis", *Comm. ACM Vol. 29*, No. 3, pp 234-249, Mar. 1987.
- [12] Sedgewick, R. "Algorithms", second edition, Addison-Wesley Publishing Company, ISBN 0-201-06673-4, 1988.

- [13] Riboe, J. “The Camel Distribution”, Tech. Rep. TRITA-TCS-9104, Dept. of Telecommunication and Computer Systems, The Royal Inst. of Technology, Stockholm.
- [14] “Guide to Parallel Programming on Sequent Computer Systems”, Prentice-Hall, ISBN 0-13-370446-7, 1989.
- [15] Ayani, R. “LR-algorithm: Concurrent Operations on Priority Queues”, Proc. of the second IEEE Symposium on Parallel and Distributed Systems, Dallas, Texas, December 1990.

Appendix 1

In this section we present the data types of the Lazy Queue(linked list) and the basic queue operations, enqueue and dequeue, in outline fashion. These outlined parts are then expanded into further detail. 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

```
/* Constants */
#define ElementsPerMonth      (8)
#define UpperBoundAEPM      (64)
#define LowerBoundAEPM      (4)
#define MaxElementsInNF      (1024)
#define MaxElementsInVFF      (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 */
            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     lengthOfMonth;    /* interval each month spans in the FF part */
priority     FF_lowborder;     /* border between NF and FF */
int          numberOfMonths,   /* number of months in FF */
            first_month;      /* index of first month in FF, the FF part */
                                /* is organized as a circular list of months */
month        *FF;             /* pointer to array of months */
priority     FF_highborder;    /* border between FF and VFF */
/* VFF part */
linkedlist   VFF;             /* pointer to linked list representing VFF */
} LazyQueue;

```

enqueue and dequeue operations

The enqueue operation can be coded as follows:

```

void enqueue(queue,el)
LazyQueue *queue;
pri_elem el;
{
    /* add element to queue */
    queue->cardinal++;
    /* Check into what part of the queue the element falls */
    if(el.pri < queue->FF_lowborder)
        InsertElementIntoNF;
    else if(el.pri < queue->FF_highborder)
        InsertElementIntoFF;
    else
        InsertElementIntoVFF;
}

```

The dequeue operation can be coded as follows:

```
void dequeue(queue,el)
LazyQueue *queue;
pri_elem *el;
{
    /* If the queue is empty return special marker directly */
    if(queue->cardinal == 0)
    {
        *el = EMPTYMARKER;
        return;
    }

    /* If we have no elements in the NF part get the next non-empty month */
    if(queue->NF_cardinal <= 0)
        GetNextMonth;
    queue->cardinal--;
    queue->NF_cardinal--;
    *el = minimum element from NF_elements and NF_head;
}
```

Further details of enqueue

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

```
InsertElementIntoNF:
{
    queue->NF_cardinal++;
    insertlist(queue->NF_head,el);
    /* Check if the number of elements in the inputpart exceeds threshold value
       and possibly perform a resize */
    CheckResizeNF;
}
```

InsertElementIntoFF:

```
{
    int i;

    /* Calculate actual month to perform insertion into */
    i = (el.pri - queue->FF_lowborder) / queue->lengthOfMonth;
    i = (i + queue->first_month) % queue->numberOfMonths;

    /* Check if the allocated array of the month has sufficient size to hold one
       extra element */
    if(queue->FF[i].size < queue->FF[i]->next+1)
    {
        /* Allocate a sufficiently large array */
        allocate_month(queue,i,min(queue->FF[i].size*2,ElementsPerMonth));
        /* Add element to month */
        queue->FF[i].elements[queue->FF[i].next++] = el;
        /* In a build up phase, check for possible resize */
        CheckBuildUpResize;
    }

    /* Normal case perform insertion directly */
    else
        queue->FF[i].elements[queue->FF[i].next++] = el;
}
```

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.

InsertElementIntoVFF:

```
{
    insertlist(queue->VFF,el);
    CheckResizeVFF;
}
```

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->numberOfMonths;
        queue->FF_highborder+= queue->lengthOfMonth;
        queue->FF_lowborder+= queue->lengthOfMonth;
        Move elements from VFF into FF;
    }
}
```

CheckResizeInDequeue;

```
sort(queue->FF[queue->first_month]);
dealloc(queue->NF_elements);
Transfer the first month to NF;
queue->first_month=(queue->first_month + 1)%queue->numberOfMonths;
}
```

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 > MaxElementsInNF)
{
    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
        LowerBoundAEPM after halving the length of months do so,
    Else recalculate the number of months needed.
    half_lengthOfMonth(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 ElementsPerMonth and the

tightest possible boundaries is set on FF.

CheckResizeVFF:

```
if (Number of elements in VFF > MaxElementsInVFF && 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

CheckResizeInDequeue:

```
if (Average number of elements per month > UpperBoundAEPM ||
```

*(The average number of elements in the ListsToScan(queue) first months is greater than 64 && Average number of elements per month > 3*LowerBoundAEPM)*

```
{
```

```
    if (The number of elements in the upper half of FF + the number of elements in VFF > MaxElementsInVFF*0.75)
```

```
        double_numberOfMonths(queue);
```

```
    else
```

```
        move elements from upper half of FF to VFF;
```

```
        half_lengthOfMonth(queue);
```

```
}
```

```
else
```

```
if (Average number of elements per month < LowerBoundAEPM)
```

```
{
```

```
    double_lengthOfMonth(queue);
```

```
    if (The number of elements in the upper half of FF + the number of elements in VFF > MaxElementsInVFF*0.75)
```

```
    {
```

```
        move elements from upper half of FF to VFF;
```

```
        half_numberOfMonths(queue);
```

```
    }
```

```
}
```

About the authors:

Robert Rönngren: Is a PhD candidate at the Department of Telecommunication and Computer systems at the Royal Institute of Technology (KTH) in Stockholm, Sweden. His research interests focus on sequential and parallel discrete event simulation. He received a MS degree in engineering physics from the Royal Institute of Technology in 1986.

Rassul Ayani: Is an Associate Professor in the Department of Telecommunication and Computer Systems, Royal Institute of Technology (KTH) in Stockholm, Sweden. He received his Dipl. Ing. degree from Technische Hochschule in Vienna, Austria (1970) and his MS and Ph.D. degree from KTH. His research interests are in parallel architecture, parallel algorithms and parallel simulation. He is Associate Editor of the ACM Transaction on Modeling and Computer Simulation (TOMACS).

Jens Riboe: