THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

A COMPARATIVE PERFORMANCE ANALYSIS OF REAL-TIME PRIORITY QUEUES

By

Nikhil Mhatre

Major Professor: **Dr. T.P. Baker**

In partial fulfillment of the
Master of Science degree, Fall 2001

Acknowledgements

I would like to express my deep appreciation to my advisor, Dr. Baker without whose constant support and encouragement, this project would not have seen completion. His original ideas, comments and suggestions were invaluable to me at all times. I also wish to thank my committee members, Dr. Lacher and Dr. Van Engelen for spending their time and effort reviewing this report. I am grateful to Mr. Gaitros, who has been very helpful to me by lending me the resources needed to complete this work.

My sincerest gratitude goes to my friends, peers and colleagues, Bharath Allam, Nikhil Iyer, and Nikhil Patel, who were constantly with me with their suggestions and support.

I wish to thank Smita Bhatia for her immense patience and inexhaustible support, and my parents, Shaila and Jayprakash for bearing with me through this considerable length of time

Last but not the least, a special thanks goes to Douglas Adams for helping me understand the concept of the Babel fish.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

Many real-time applications require the use of priority queues, for priority dispatching or for timed events. The stringent performance requirements of real-time systems stipulate that randomness and unpredictability in all software components be avoided. This suggests that priority queue algorithms used in real-time software be well understood and have bounded worst-case execution time.

A survey of existing priority queue algorithms is conducted to select ones that are suitable for real-time applications. Empirical average and worst-case performance evaluation of selected ones is performed and the results analyzed. This survey is improved by the availability of a higher resolution, reliable clock function that enabled point-to-point measurement of execution times as compared to the restrictive method of averaging the execution times over a large number of runs.

1. Introduction

1.1 Definition

A *priority queue* is a set of items each of which has a *priority* drawn from a totally ordered set. Elements of a priority queue can be thought of as awaiting service, where the item with the smallest priority is always to be served next. Ordinary stacks and queues are special cases of priority queues [2].

Typically, applications require the primitive operations: Make_Empty, Insert, Delete, Min, Member and Empty. The primitive operation Make_Empty (Q) returns an empty queue Q. Insert (Q, item) inserts the specified item in the queue Q. The operation Delete (Q, item) removes the specified item from Q. Min (Q) returns an item in Q having the least priority and Member (Q, item) returns *true* if an item having that name is in Q, else it returns *false*. Finally, Empty (Q) returns *true* if there are no items in the queue Q and *false* otherwise.

Priority queues find applications in job scheduling, discrete event simulation, and various sorting problems. Priority queues have also been used for optimal code construction, shortest path computations and computational geometry.

1.2 Objective

This study addresses the problem of choosing an appropriate priority queue algorithm for an application. There are many different priority queue algorithms and many applications; no priority queue algorithm is best for all applications. The applications that we are primarily interested in here are real-time systems. This analysis is based on a prior study by Maheshwari [1] and is aimed at bringing it up to date. We were aided in this exercise by the availability of a higher resolution clock function provided by RT-Linux [17], which has enabled us to gather more accurate results.

1.3 Requirements for Real-Time Systems

Typical use of priority queues in real-time systems is in job scheduling and in timed-event or “wakeup” queues. In the former, jobs waiting to be dispatched are ordered by priority in a queue and the dispatcher serves the job with the highest priority. The latter consists of jobs that are currently suspended but which need to be activated at some future time. These jobs are arranged in the queue in order of increasing wakeup times. When the first job’s wakeup time equals the system time, it is removed from the queue and appropriate actions are performed.

The fact that distinguishes the use of priority queues for real-time applications from other applications is that in real-time systems we are usually interested in the worst-case performance rather than the average-case performance. The criticality of “hard” real-time systems requires that unpredictability and randomness be avoided. Moreover, in real-time systems, queue sizes are small or moderate. This eliminates priority queue algorithms that perform well only for very large queue sizes. Finally, the algorithms must efficiently support all the operations that may be required in real-time applications (e.g. arbitrary deletion).

1.4 Terminology

The time taken by an algorithm can vary considerably between two different executions of the same size. In the worst–case analysis of an algorithm we only consider those instances of that size on which the algorithm requires the most time. However, on the average, the required time is appreciably shorter. This analysis of the average execution time on instances of size n is called *average-case* analysis. The analysis depends on the distribution of test cases.

In most applications of data structures, we wish to perform not just a single operation but also a sequence of operations, possibly having correlated behavior. By averaging the running time per operation over a worst-case sequence of operations, one can sometimes obtain an overall time bound much smaller than the worst-case time per operation multiplied by the number of operations. This kind of averaging is called *amortization* [3].

A priority queue algorithm is said to be *stable* if it maintains first-in, first-out (FIFO) order within elements of the same priority.

Finally, "little-oh" is defined as: $f(n) \sim o(g(n))$ iff there exists a positive constant n_o such that for

$n > n_o$

$$\lim_{n \to \infty} f(n) / g(n) = 1$$

2. Background

As with all abstract data types, there are many ways to implement priority queues; each implementation involves a data structure used to represent the queue, and an algorithm that implements each abstract operation in terms of this structure.

What follows is a brief overview of the algorithms surveyed and justification as to why they were selected, or eliminated from testing here. The selection is in accordance with the requirements given in the first chapter.

2.1 Rings

Rings are ordered linear lists that are connected in a circular fashion. For an n item ring, an $O(n)$ search is required when an item is inserted or deleted. Min can be implemented in constant time and Delete_Min also requires constant time. There are many versions of rings including singly linked rings and doubly linked rings. Both forms were tested empirically and singly linked rings were retained for further analysis as they are simpler to implement and the results obtained can be extrapolated to doubly linked lists. They also have the advantage of requiring less storage. The biggest disadvantage of Rings is that they have an $O(n)$ worst case execution time. It is possible to implement rings using linked lists or arrays. The disadvantage of the former is that it requires dynamic storage allocation, which may be undesirable in some real time systems. The latter has the disadvantage of placing an upper bound on the number of elements that can be accommodated. The implemented test here uses linked lists.

Rings were selected because we expected good performance for small sized queued. They also have the advantage of placing no restriction on the number of items that can be accommodated and are easy to implement. Finally, rings are stable. A version of rings was also implemented in assembly language and tested to measure the performance improvement in going from a high level language to assembly language.

2.2 Implicit Heaps

Implicit Heaps, or simply *heaps* [4] are the oldest priority queue algorithm with $O(\log n)$ performance. The heap data structure is based on binary trees in which each item always has a higher priority than its children. In an implicit heap, the tree is embedded in an array. The rule is that location 1 is the root of the tree, and that locations $2i$ and $2i+1$ are the children of location i. Since the smallest item resides at the root of the heap, Min can be implemented in constant time. The Insert operation appends the item at the last (i.e. rightmost) leaf and lets it rise towards the root so as to restore heap order. Delete replaces the item by the last item and then lets this item rise or fall so as to restore heap order.

Heaps were a natural choice for measurements as they have good performance, are well understood and are a basis of comparison in the literature. A drawback of using implicit heaps is that storage for the data structure must be statically allocated. This places an upper bound on the number of items that can be accommodated. Finally, heaps are not stable.

2.3 Leftist Trees

Leftist trees [5], which were invented by C.A. Crane, use the heap structure, but here the heap is explicitly represented with pointers from parents to their children. The two children are sorted so that the right path is the shortest path from the root to an external node. The fundamental operation on leftist trees is *melding*, i.e. joining two trees. To insert an item into a leftist tree, we make the item into a one-node tree and meld it with the existing tree. To delete a minimum item, we remove the root and meld its left and right sub-trees. Both these operations take $O(\log n)$ time. Arbitrary deletion is more difficult and requires adding information to the tree representation or using a more intricate algorithm in which an item to be deleted is merely marked as deleted, the actual deletion is carried out during subsequent Min operations.

Leftist trees were eliminated from this analysis because it has been shown by Jones [6] that they are consistently about 30 percent slower than implicit heaps.

2.4 B-Trees

B-Trees, introduced by Bayer and McCreight [7], are a generalization of the binary search tree. They maintain the linear left-right order condition on the node that binary search trees possess, while maintaining the balance of the tree. This linear left-right ordering is what sets them apart from heaps. The price of these improvements is added complexity in the programming. B-trees have insertion and deletion defined so that the number of children of each node always lies within fixed lower and upper limits (*a* and *b* respectively). B-trees are widely used to provide fast access to externally stored files [4].

B-trees were selected for analysis because we believed that they would perform well for large queue sizes if the number of items per node were large. However, after we finished implementing them, it was apparent that they would be inefficient. The measurements presented here are just for the 2, 4 version. Other *a*, *b* versions did not produce significantly different results and hence were left out. Their dismal average-case performance ruled them out from the worst-case measurements.

2.5 AVL Trees

AVL trees, also called height-balanced trees, were developed by G.M. Adel'son-Vel'skii and E.M. Landis [8]. An AVL tree is a binary search tree in which the heights of the left and right sub-trees of any node differ by at most one. Insertion and deletion in AVL trees is the same as for any binary search tree. However, to restore balance, single and double rotations are performed [9]. Insertion and deletion take $O(\log n)$ time.

Like B-trees, AVL trees are not simple to implement They were eliminated from the tests here as they have been shown to be inferior by Jonassen and Dahl [10].

2.6 D-Trees

D-trees are due to T.P. Baker and were first described in [1]. D-trees are a special form of heaps implemented as binary *d*ecision *trees*. As in heaps, the tree is embedded in an array with the root at location 1 and for each node i, the left child of i is at $2i$ and the right child at $2i + 1$. They differ from heaps in that all items are stored at leaves. Any node that is a parent of a node to which an item is assigned is always an interior node. The value stored at an interior node is the one with the higher priority of its two children. Thus the root has the item with the highest priority and Min can be computed in $O(1)$ time. Inserting a sentinel value at all the nodes initializes the D-Tree. This sentinel has a priority that is lower than the lowest possible priority. For a real item Insert simply inserts the item in its designated place at one of the leaves and propagates the change in priority of parent nodes up to the root. Replacing the item with the sentinel value and propagating this change up the tree performs deletion.

D-trees were included in these tests as it is a new algorithm that had never been tested, and promised good performance. The disadvantages of D-Trees are that it requires almost twice the storage of heaps, and like heaps, are not stable

2.7 Binomial Queues

Developed by Jean Vuillemin [2], the binomial queues are a very interesting and conceptually simple data structure. A binomial queue is represented as a forest of heap-ordered trees where the number of elements in each tree is an exact power of two. The times taken by operations on a binomial queue are bounded by $O(\log n)$. Empirical and theoretical evidence gathered by Brown suggests that binomial trees give the fastest implementation of meldable heaps when constant factors are taken into account. Binomial queues use less storage than leftist or AVL trees but are complex to code.

Binomial queues were left out from the experiments here because Brown [11] has shown that heaps are slightly faster than binomial queues on the average and considerably faster in the worst case. These results were reopened and questioned in [6], where it was claimed that binomial queues are nearly optimal.

2.8 Pagodas

Pagodas [12], like leftist trees, are based on heap ordered binary trees. However, unlike leftist trees, in a Pagoda the primary pointers lead from the leaves of the tree towards the root. A secondary pointer in each item of the tree points downwards from that item to its otherwise unreachable leftmost or rightmost descendant. The root, having no upward pointer, has pointers to both its leftmost and rightmost descendants. As a result, all items in a pagoda are reachable from the root by somewhat complex paths, and all branches of the structure are circularly linked. The insert operation is based on merging the right branch of one pagoda with the left branch of another. Deletion requires finding the two pointers to the item, which can be done because all branches are circularly linked.

Unlike leftist trees and binomial queues, no effort is made to maintain the balance of a pagoda. Therefore, although the average time for operations on pagodas is $O(\log n)$, there are infinite sequences of operations that take $O(n)$ time per operation [6]. This was the reason pagodas were not included in the tests here.

2.9 Fishspear

The Fishspear [13] algorithm is one of the recent priority queue algorithms. Fishspear is an instance of a general class of (non-deterministic) algorithms, which operate on a data structure called a Fishspear. Fishspear can be implemented with sequential storage and is more efficient than heaps in two senses. First, both have the same amortized $O(\log n)$ worst-case comparison per queue operation, but in Fishspear the coefficient of $\log n$ is smaller. Secondly, the number of comparisons is “little-oh” of the number made by heaps for many classes of input sequences that are likely to occur in practice. However, it is more complicated to implement than heaps and the overhead per comparison is greater.

Fishspear is similar to self-adjusting heaps in that the behavior depends dynamically on the data and the cost per operation is low only in the amortized sense – individual operations can take time $\Omega(n)$. Clearly, Fishspear was not suitable for our purposes and was not treated.

2.10 Splay Trees

Splay trees, developed by Sleator and Tarjan [14] are a form of binary search tree based on the concept of *self-adjustment* and *amortized execution time*. The restructuring heuristic used in splay trees is *splaying* which moves a specified node to the root of the tree by performing a sequence of rotations along the (original) path from the node to the root. Splay trees avoid the costs associated with tree balancing by blindly performing pointer rotations to shorten each path followed in the tree. This avoids the necessity of maintaining or testing records of the balance of the tree but it also increases the number of rotations performed. Splay trees are stable.

We see that in splay trees, individual operations within a sequence can be expensive making them unsuitable for real time applications. Hence they were rejected.

2.11 Skew Heaps

Skew heaps are also due to Sleator and Tarjan [3] and they too do not rely on any mechanism that limits the cost of individual operations As a result, individual inserts or deletes may take O(n) time However skew heaps guarantee that the cost per operation will never exceed log(*n*) if the cost is amortized over a sufficiently long sequence of operations. The basic operations on a skew heap are very similar to those on a leftist tree, except that no record of the path length to the nearest leaf is maintained with each item; instead, the children of each item visited on the merge path are simply exchanged, thereby effectively randomizing the tree structure.

Skew heaps were left out from our tests for the same reason as splay trees.

2.12 Pairing Heaps

Pairing heaps correspond to binomial queues in the same way that skew heaps correspond to leftist trees. A pairing heap is represented as a heap ordered tree where the insert operation can be executed in constant time either by making the new item the root of the tree, or by adding the new item as an additional child of the root. The Delete operation removes the sub-tree headed by the item from the heap and links it to the root after deleting the item from the sub-tree. The key to the efficiency of pairing heaps is the way in which the new root is found and the heap reorganized as the result of a delete operation. This is done by linking successive children of the old root in pairs and then linking each of the pairs to the last pair produced.

Pairing heaps have an $O(\log n)$ amortized performance bound, but as can be seen from the above description, they are not suitable for real time applications.

2.13 Bit Vectors

Most modern microprocessors now provide bit scan instructions or floating point normalization instructions. Bit vectors can be implemented in assembly language on any processor that has either of these facilities. Our implementation used bit scan instructions.

The data structure used by this algorithm in its simplest form is an array of *n*, where *n* is the number of jobs to be accommodated. Thus every job has a bit associated with it. The bit corresponding to a job can be calculated based on its job number, priority level or time-stamp. An Insert is performed by calculating the bit corresponding to that job and setting it. Similarly, Delete is performed by resetting the bit associated with that job. In this form of implementation, Min can be determined only by scanning through the entire array and determining the first bit that is set. This overhead can be avoided by implementing a hierarchical structure of bit vectors as shown below.

Here, the bits in the first two levels of bit vectors correspond to intervals of priority. Thus, if the first bit in the top level is set, it means that there is at least one job with priority from 0 through 255 present in the queue. The first bit in the second level being set indicates that there is at least one job with priority from 0 through 15 is present in the queue and so on. For example, if a job with priority 17 is the highest priority job in the queue, then Min can be determined as follows:

1. Scan the first level to determine that the first bit is set.

2. Scan the cell of the second level corresponding to the first bit of level one (the first element).

3. Scan the cell of the third level corresponding to the first bit of level two (the second element).

The third scan instruction will determine that a job with priority 17 exists which is the current Min. Hence, with this implementation, it can be guaranteed that Min can be determined by three bit scan instructions i.e. in constant time.

Bit-Vectors were included in these tests as they promised good performance because of the speed of the scan instructions and other bit-manipulation instructions. However, Bit-Vectors have the disadvantage of requiring disproportionately large amounts of memory if a large number of items and priorities are to be accommodated. For example a 512 * 512 bit-vector requires over 32Kbytes of memory. Bit-vectors are unstable and they also dictate that the number of priority levels be fixed. Finally, some machines do not have bit-scan instructions.

2.14 Lazy Queue

The lazy queue [15] is based on conventional multi-list structures with some distinct differences. It is divided into three parts: 1) the near future (NF) where most of the events are known, 2) the far future (FF) where most of the yet unknown events are expected to occur, and 3) the very far future (VFF), which is an overflow list. The hypothesis here is that a lot of work can be saved by postponing the scheduling of events until they are needed. As time advances, the boundaries of the NF, FF, and the VFF are advanced. If the NF is found to be empty, then the next interval in the FF is sorted and placed into the NF. In this way, the sorting process can be delayed until it is necessary. An Insert operation is carried out by determining into which part of the queue the element falls and inserting the element into it. A Delete-Min operation is carried out by removing the smallest elements in the NF.

As mentioned before, real-time systems usually have small queue size and it is obvious that Lazy queues are designed for long queues. Also, arbitrary deletion from a lazy queue can be expensive since the FF and the VFF are not sorted. Hence, we conclude that Lazy queues are not suitable for real-time applications.

2.15 Calendar Queue

The calendar queue [11] is a simple algorithm based on the concept of a day-to-day calendar. It assumes that the future event set is divided into a pre-determined number of intervals, and each interval has a list of all events scheduled for that time interval. Thus it can be said to resemble the concept of a hash structure. An Insert operation is carried out by determining the time interval it belongs to and inserting that element into the list of that interval. The Delete-Min operation simply removes the first element from the earliest time interval.

The calendar queue, however is not suitable for real-time systems because of two reasons:

1. The maximum size of the queue needs to be estimated before-hand which is difficult in the applications that we are interested in, and
2. Brown [11] hypothesis that if the queue is sparsely filled, the algorithm will perform poorly. Hence, the queue size will have to be determined to a very accurate degree in order for this algorithm to perform efficiently.

3. Implementation

3.1 Programming Conventions

The GNAT Ada [16] compiler was used to develop all of the algorithms being measured for performance. Each algorithm was implemented as a package using the following specification:

```
package < algorithm > is

    subtype Item_Type is Integer range 1 .. < last_item >;

    type Queue_Type is limited private;

    procedure Make_Empty (Queue : in out Queue_Type);

    procedure Delete (Queue : Queue_Type; Item : Item_Type);

    procedure Insert (Queue : Queue_Type; Item : Item_Type);

    function Min (Queue : Queue_Type);

end < algorithm >;
```

3.2 Measurement Environment

The test environment used for the measurement was Real-Time Linux (RT-Linux) [17]. RT-Linux is an extension to the regular Linux operating system. Its basic design is illustrated in Figure 1. Instead of changing the Linux kernel itself to make it suitable for real-time tasks, a simple and efficient real-time kernel was introduced underneath the Linux kernel. The real-time kernel only schedules the Linux kernel (and therefore all the normal Linux processes under its control) when there are no real-time tasks to be executed.

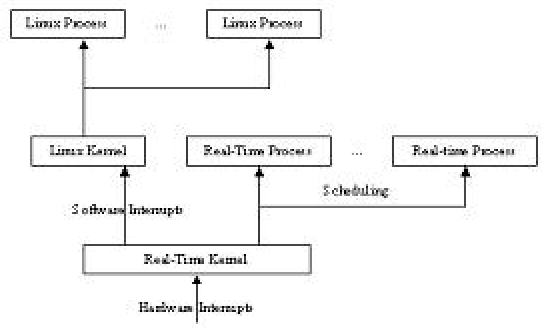

Figure 1. Real-Time Linux

It will preempt the Linux kernel whenever there is a real-time task ready to run, so that the time constraints of real-time tasks can be satisfied. In other words, the original Linux kernel is running as a task of the real-time kernel with the lowest priority.

RT-Linux provides a clock function that was used for our performance measurements. This function has a resolution of one nanosecond. This enables us to measure the execution times point-to-point within code, as compared to [1] which relied on averaging the execution times because the clock was too coarse.

3.3 Measurement Overheads

In all cases of performance measurements, there were two overheads that were taken into account:

3.3.1 Clock function overhead

The overhead associated with calling the clock function was not determined by taking the average of a large number of calls. Instead, the clock was called and timed a thousand times. The execution times were examined manually and it was found that it executes in 448ns more than 90% of the time, 416ns ~10% of the times and has a random value only once. This random value occurs the first time the clock function is called. Thus, it can be assumed that once the clock function is called in rapid succession multiple numbers of times, it begins taking advantage of the cache and can execute in the same amount of time consistently. In multiple runs of this test, the values given above did not change. Hence, the overhead of the clock function was assumed to be 448ns.

3.3.2 Random number generator overhead

The random number generator provided by the GNAT [16] compiler was used for our measurements. The overhead for a single call to the random number generating routine was 5583ns. This overhead of a generating a single random number was found by making a large number of calls, timing them individually and taking the average of all the calls. In multiple runs of this test the maximum variation from the average was less than 0.8%.

3.4 Average case

Average case measurements were carried out on the Insert and the Delete-Min operations. The Delete-Min is a combination of the Delete and the Min operations, which calculates and deletes the element with the smallest numerical value of priority on the queue. The smallest numerical value is assumed to be the highest priority in all cases.

3.4.1 Test methodology

The test case for the average case measurements was based on the "Hold Model" [18]. This model is based on the Simula "hold" operation, and has been used often for research on event sets and priority queues. A hold operation is defined as "*An operation which determines and removes the record with the minimum time value from the event set being considered, increases the time value of that record by T, where T is a random variant distributed according to some distribution F(t), and inserts it back into the event set*" [18]. To suit the current measurements, this definition was adapted as *An operation that determines and removes the job with the smallest value of* *priority from the priority queue being considered, generates a new job whose priority is determined at random and inserts this new job into the priority queue* This model has three steps. The steps and how they were adapted for the current measurements are as follows:

1. INITIALIZATION: *Insert N event records into the future event set with the time value determined from the random distribution F(t).* Insert *N* jobs into the priority queue. The priorities of these jobs are determined at random.

2. TRANSIENT: *Execute N1 hold operations to permit the event set to reach steady state. In analytical work, N1 is assumed to be infinity. In practice, N1 is some small multiple of the size of the event set.* Execute *N1* hold operations to permit the queue to grow to a steady state. In practice, *N1* is assumed to be the size of the queue that was being tested.

3. STEADY STATE: *Execute N2 hold operations and measure the performance of the event set algorithm to do this.* Execute *N2* hold operations and measure the performance of the algorithm that is being tested.

For the tests being carried out F(*t*) is the random number generator provided by the GNAT Ada compiler, *N* is 0, *N1* is the queue size being tested and *N2* is 1000.

3.4.2 Tested algorithms

Average case measurements were performed on the following algorithms:

- Rings
- Rings (assembly version)
- Implicit heaps
- D-Trees
- B-Trees

- Bit-vectors.

The results obtained from these measurements are presented in the next chapter.

3.5 Worst case

Worst-case analysis was performed on the Rings (Ada and Assembly versions), Heaps and D-Trees algorithms. The Insert and Delete operations of each algorithm were tested for performance. The Min operation in all three algorithms can be carried out in $O(1)$ time and hence was omitted from the measurements.

The test method for the worst-case analysis was different for each data structure. A brief description of each method follows.

3.5.1 Rings

The Rings algorithm maintains a linked list in a sorted order. The worst-case for such a data structure is when the largest element is inserted and deleted for the Insert and Delete operations respectively. Thus, inserting a thousand consecutive elements into the list, and then timing the insert and delete operations successively for the largest element tested the worst-case performance for a queue size of one thousand.

3.5.2 Heaps

The worst case for the Heaps algorithm is when the smallest element is inserted or deleted, for the Insert and Delete operations, respectively. This case leads to maximum *sifting*, that is, movement of elements towards or away from the root to restore the heap order. When an item is deleted from the heap, its two children are compared to determine which will replace the element deleted. If it can be ensured that this comparison always fails, that is, the element that is assumed to be the greater is always the smaller, there will be two additional assignments at each node. This brings out the worst case in Heaps.

The worst case occurs when an insert is done for queue sizes of $2n-1$ for $n = 3, 4, \ldots$ The case when $n = 2$, is not very interesting. Hence, the queue size of 3 was chosen to be the first measurement length.

3.5.3 D-Trees

D-Trees have a data structure that is very similar to Heaps, and hence have similar worst-case scenarios. Thus, the worst case for D-Trees is also when the smallest element is inserted or deleted. This is the case when a new *min* has to be propagated to the root.

4. Measurements

4.1 Average Case

Average case measurements were performed on the following algorithms:

- Rings
- Rings (assembly version)
- B-Trees
- Bit Vector
- D-Trees
- Heaps

Each algorithm was tested with four different random seeds. The resulting graphs are shown on the following pages. The graphs have been shown in a logarithmic scale since the behavior at smaller sizes is emphasized. In the context of real-time systems, the behavior for small queue sizes is more interesting since the queue size in such systems is usually small. The variation of execution time as seen from the graph is not significant for different random seeds used. A brief explanation of the performance of each algorithm also follows.

4.1.1 Rings

Rings have $O(n)$ performance for the Insert operation as can be seen from Figure 4. The performance is impressive for small sizes of the queue, but as the queue size increases, the performance degrades linearly.

The measurements for the Delete-Min operation of the Rings algorithm are not significant because it always deletes the first element from the list. This can be done in constant time regardless of the size of the queue. The chart for the delete operation proves this (Figure 5). The slight increase in execution time for longer queue sizes can be attributed to the fact that more system resources are needed as the length of the queue increases.

4.1.2 Rings (Assembly version)

The assembly version of the Rings algorithm was implemented to record the performance improvement that could be gained by such an exercise. As seen from Figure 2 and Figure 3 which are Insert and Delete-Min operation measurements for this algorithm, the assembly version of the Rings algorithm performs significantly better than the version implemented in an high level language. However, it does behave in a similar manner to the version implemented in a high level language.

4.1.3 B-Trees

The Insert and Delete operations for the B-Trees algorithm exhibit approximately the same behavior. The charts are shown in Figure 8 and Figure 9. As seen, this algorithm behaves erratically. This erratic performance is due to re-arranging of the nodes when the Insert or Delete operation is carried out. For some queue sizes, this needs to be done very often resulting in larger than average execution times.

4.1.4 Bit Vector

The results of the bit vector algorithm are shown in Figure 6 and Figure 7. The Insert operation executes in near constant time regardless of the queue size, and hence performs as expected.

The Delete-Min operation of this algorithm is interesting. As seen, the execution time drops as the queue size increases. This is due to the implementation of the data structure as a three-level hierarchy. The place where the element is expected to be present is checked first and if the element exists, it is deleted. Once that particular element is deleted, the algorithm checks if there are any more elements present in that interval of priority. If none exist, then it traverses up the hierarchy clearing the bits until it reaches the top-most level. Hence, if the queue has a large number of elements present, the probability of the interval under consideration being empty is lesser and the traversal up the hierarchy might not be needed. Thus, the Delete operation performs slightly more efficiently at longer queue sizes.

4.1.5 Heaps

Figure 10 shows the results for the measurements on the Insert operation for the Heaps algorithm. The execution time for the Insert operation initially increases as the queue size increases. But at a certain size of the queue, the Heap is populated to the point where very little iteration is needed to insert a new element. At this stage the execution time for inserting an element begins dropping. Thus, this algorithm behaves exceptionally well for insertion when the queue sizes are large.

The Delete-Min operation for Heaps is the same as the worst-case behavior. The following section describes the worst-case behavior for Heaps.

4.1.6 D-Trees

The D-Trees algorithm, being similar to the Heaps algorithm behaves in a similar manner. As the queue size increases, the time needed for the insertion drops, since fewer numbers of iterations need to be carried out when a new element is inserted. The plot is slightly different here. The execution time for delete begins dropping immediately as the queue size increases.

D-Trees like Heaps also exhibit worst-case behavior for the Delete-Min algorithm. The results are shown in Figure 13.

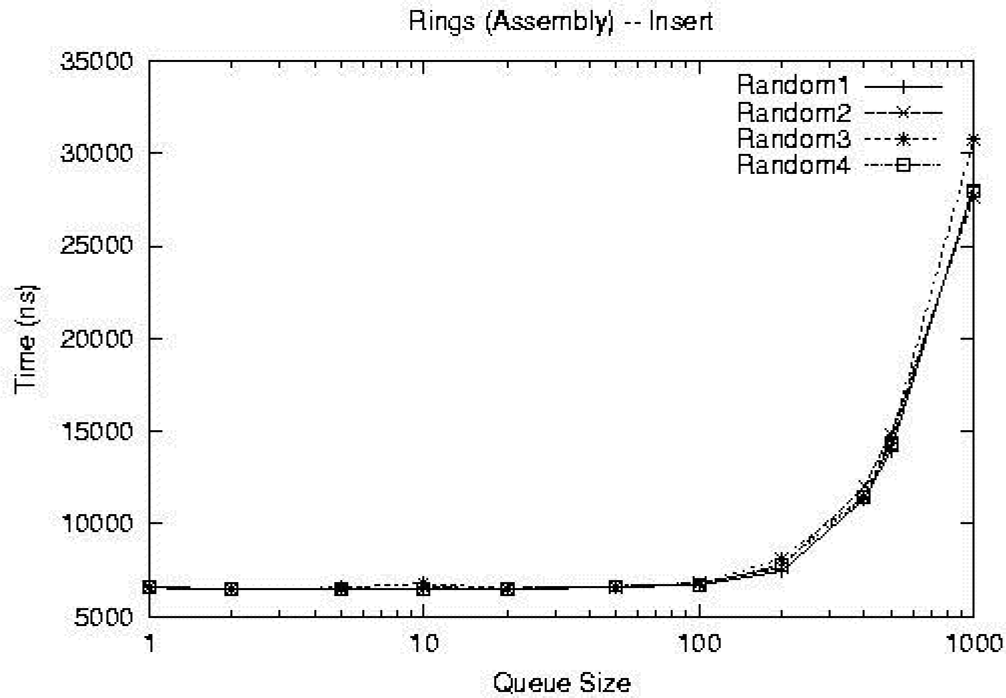

Figure 2. **Rings (Assembly) – Average case Insert Performance**

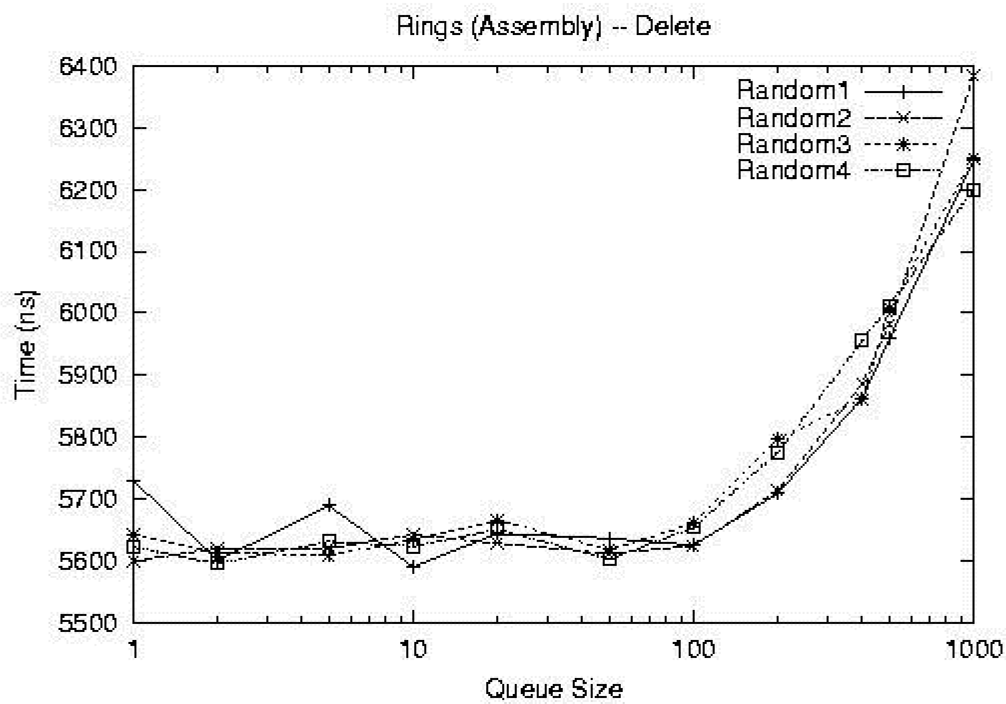

Figure 3. **Rings (Assembly) – Average case Delete performance**

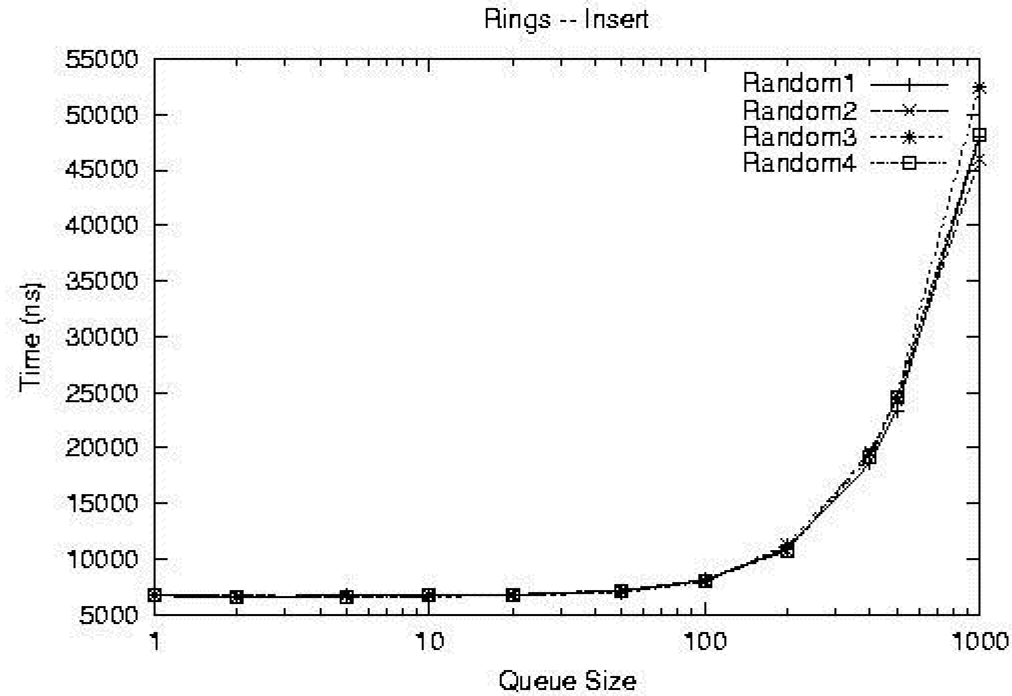

Figure 4. Rings – Average case Insert performance

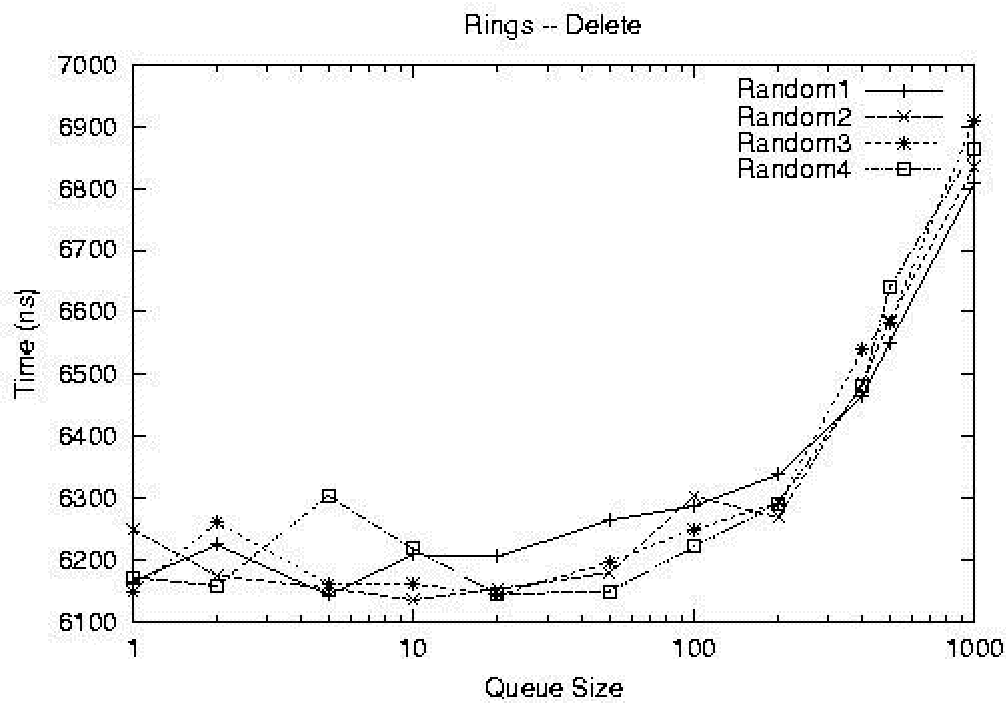

Figure 5. Rings – Average case Delete performance

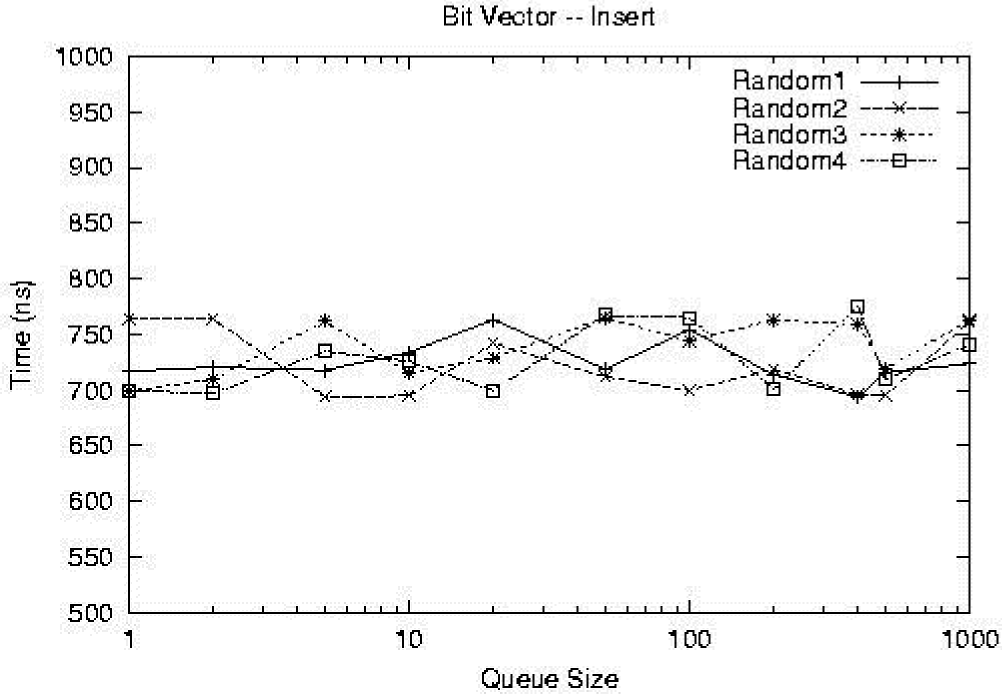

Figure 6. Bit Vector – Average case Insert performance

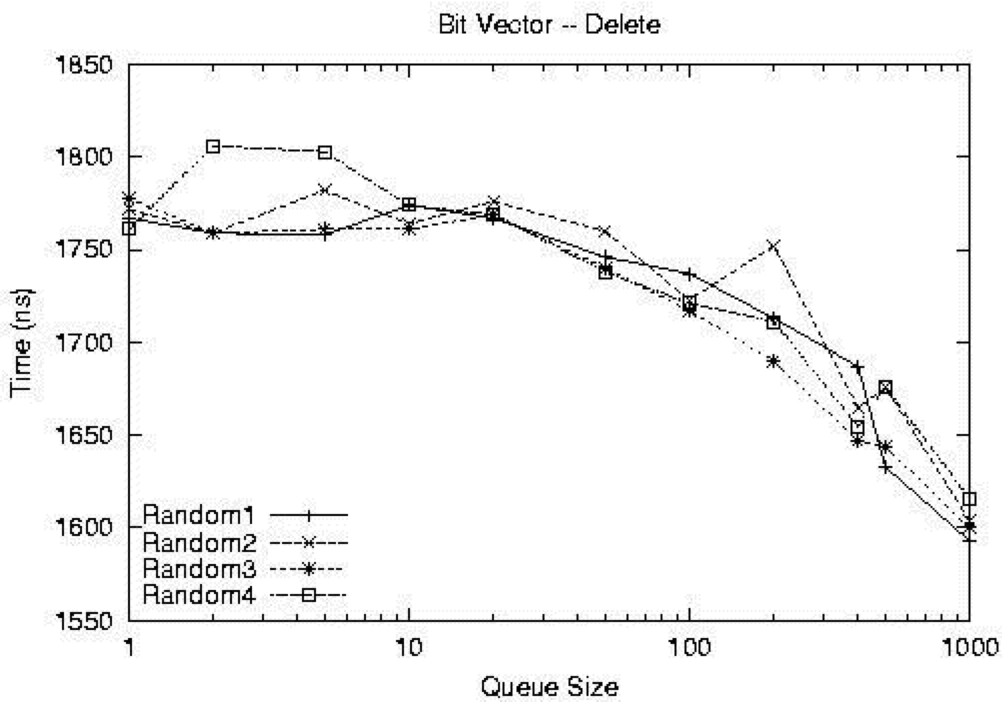

Figure 7. Bit Vector – Average case Delete performance

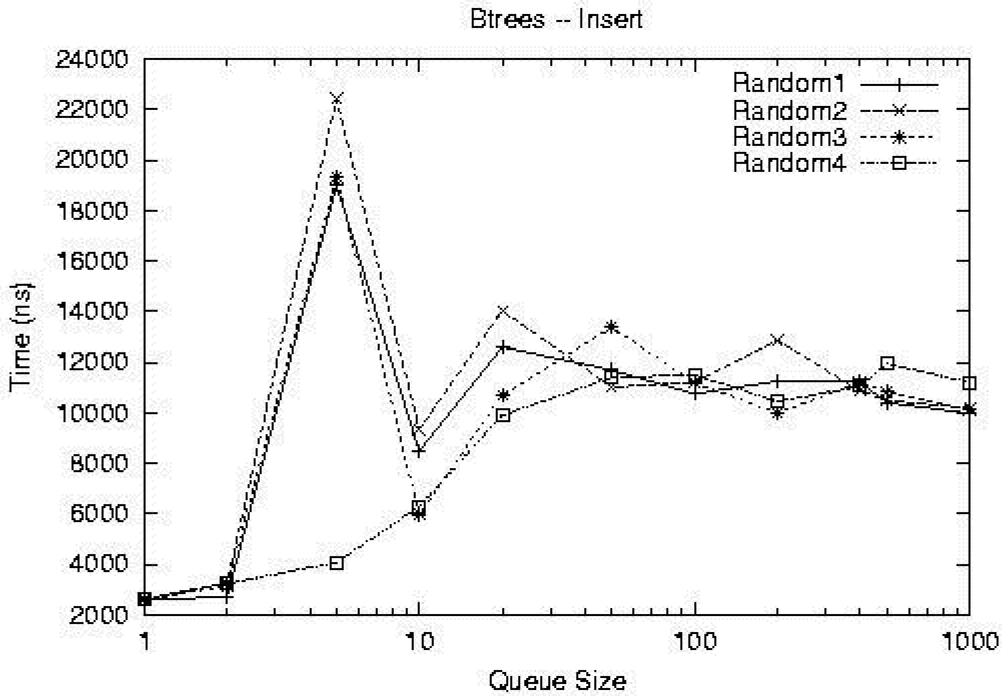

Figure 8. **B-Trees – Average case Insert performance**

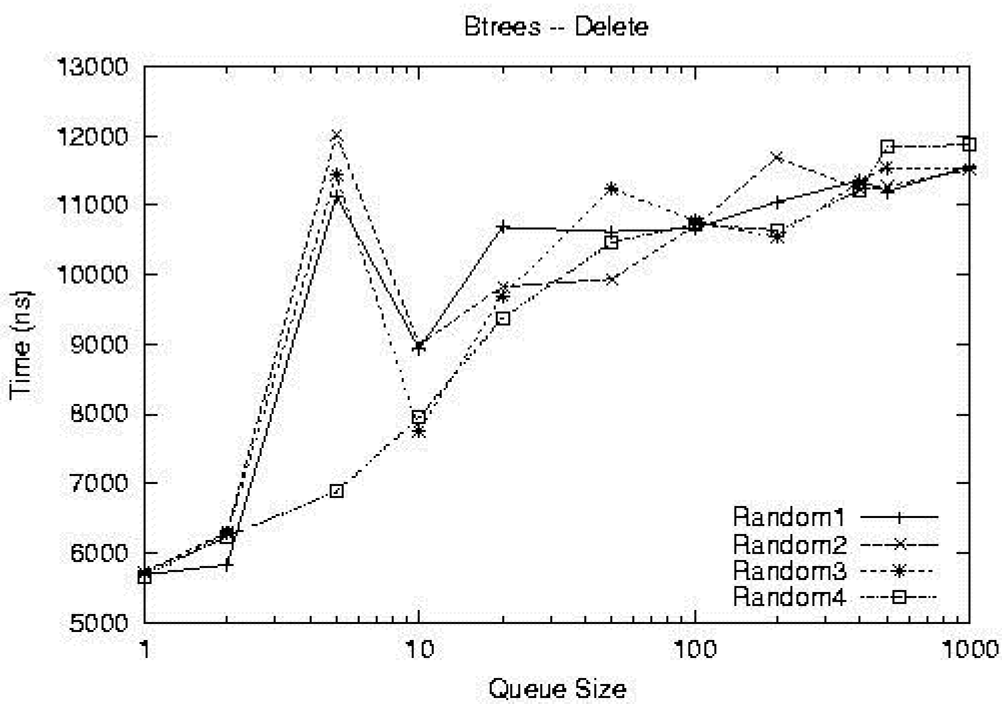

Figure 9. **B-Trees – Average case Delete performance**

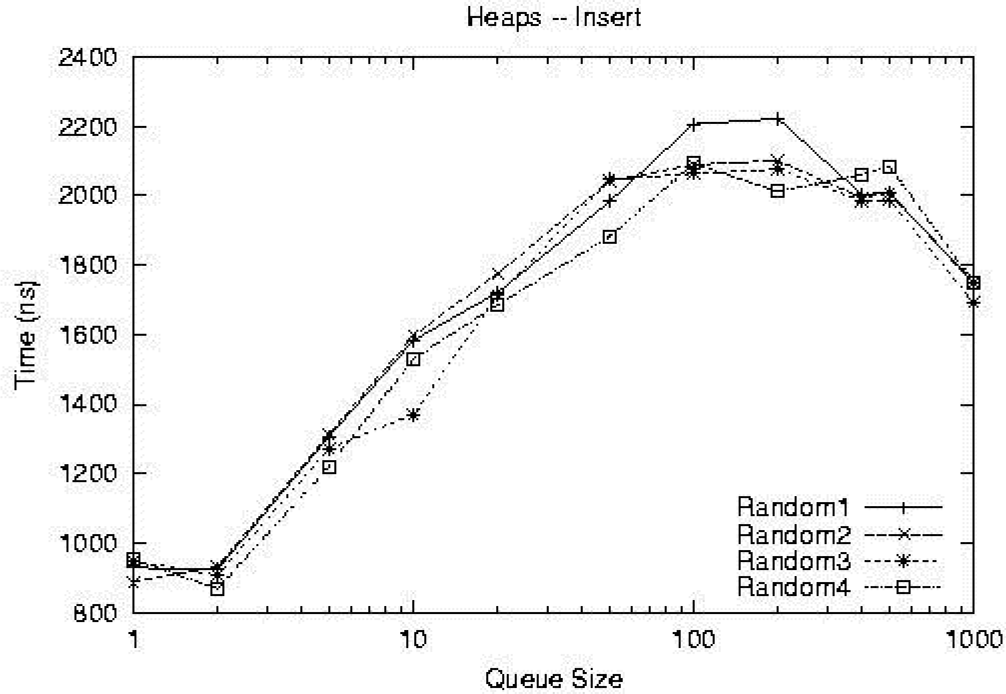

Figure 10. Heaps – Average case Insert performance

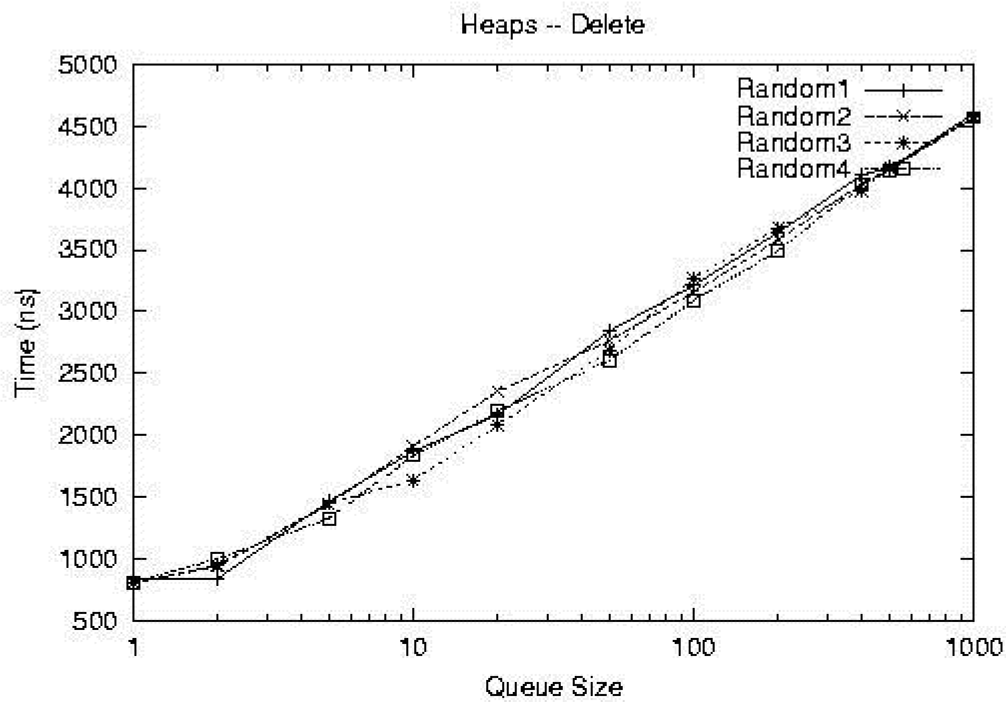

Figure 11. Heaps – Average case Delete performance

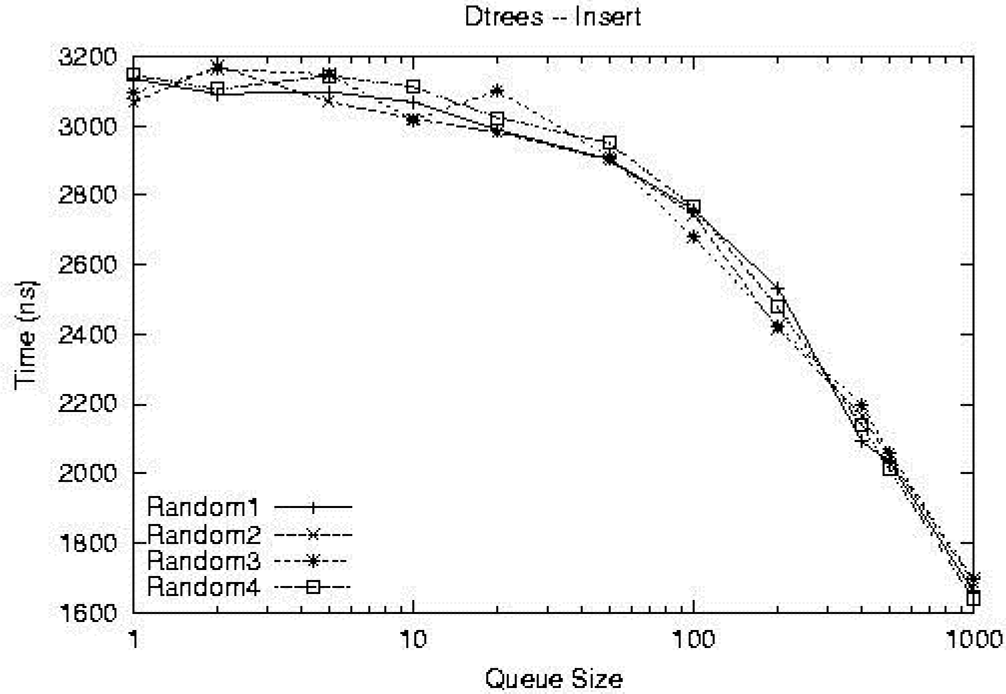

Figure 12. D-Trees – Average case Insert performance

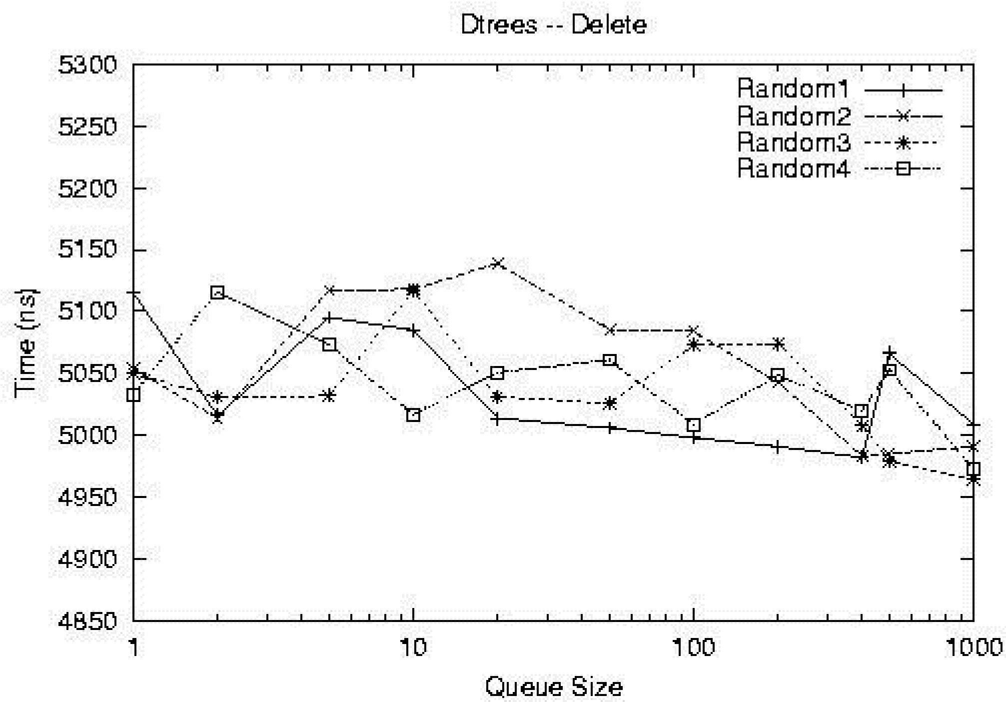

Figure 13. D-Trees – Average case Delete performance

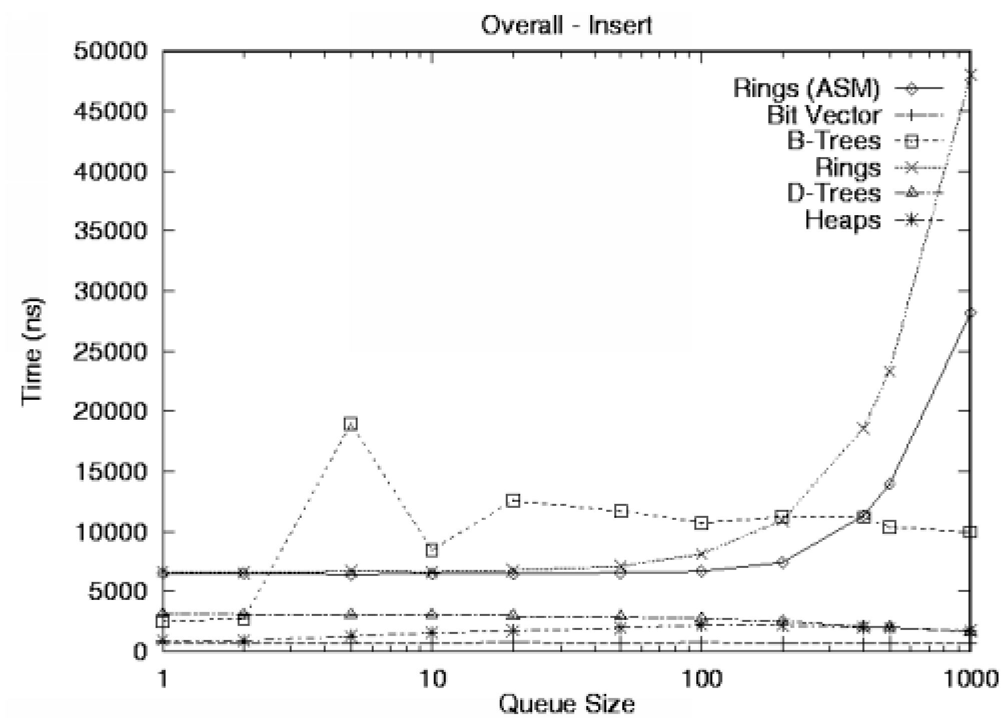

Figure 14. Overall – Average case Insert performance

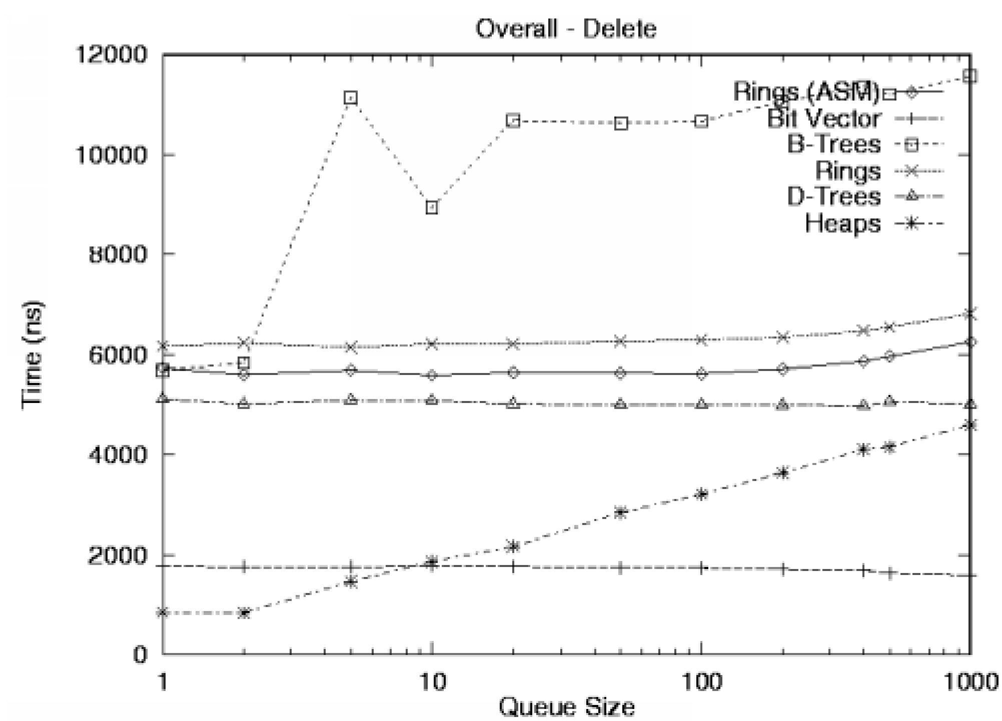

Figure 15. Overall – Average case Insert performance

4.2 Worst Case

Worst-case results are shown as charts in the following pages. The first two plots (Figure 16 and Figure 17) compare the performance of all the algorithms for the insert and delete operations respectively. The next two (Figure 18 and Figure 19) show the comparison between Heaps and D-Trees for the same operations. As mentioned previously, the algorithm for the worst-case analysis in each case is different depending on the algorithm being tested. All the four algorithms tested here show similar results for both the Insert and Delete operations. A brief commentary on their behavior follows.

4.2.1 Rings

Both the Ada and Assembly versions of the Rings algorithms were tested under the worst case. As seen in the case of average case analysis, the plots for both algorithms are similar but the assembly version performs much more efficiently than the Ada version.

Rings do not perform well under worst-case conditions. Figure 16 and Figure 17 show the behavior of these algorithms as compared to the other algorithms. The performance declines linearly as the queue size increases

4.2.2 Heaps

Heaps have a logarithmic degradation of performance as the queue size increases for both the Insert and Delete operations under the worst case. They behave exceptionally well for small queue sizes, but efficiency drops as the length of the queue increases.

4.2.3 D-Trees

D-Trees exhibit a consistent behavior for all queue sizes for both the queue operations tested. In the case of the Insert operation, they show a very slight decline in performance for large queue sizes.

It must be noted here that the consistent and almost constant performance of D-Trees is dependant on the maximum number of elements that can be inserted into the queue. In our experiments, this limit was 2000 items. If this limit were increased or decreased, the value of the seemingly constant function would increase or decrease accordingly. This is due to the fact that as the number of items that can be accommodated increases, the height of the tree structure increases, and hence more amount of time is required for traversal from the leaf nodes to the root whenever an item is inserted or deleted.

4.2.4 Comparison of worst case results

Comparison of the results obtained from worst-case analysis shows that Rings are not suitable as priority queue data structures. Heaps behave very well when the queue size is small, but their performance declines as the queue size is increased. This indicates that they can be used as priority queue structures where the queue size is guaranteed to be small. However, the overall worst-case behavior of D-Trees is much more consistent. They perform very well at large queue sizes and the difference between Heaps and D-Trees for small queue sizes is not very great indicating that D-Trees could be used as priority queue structures for all queue sizes.

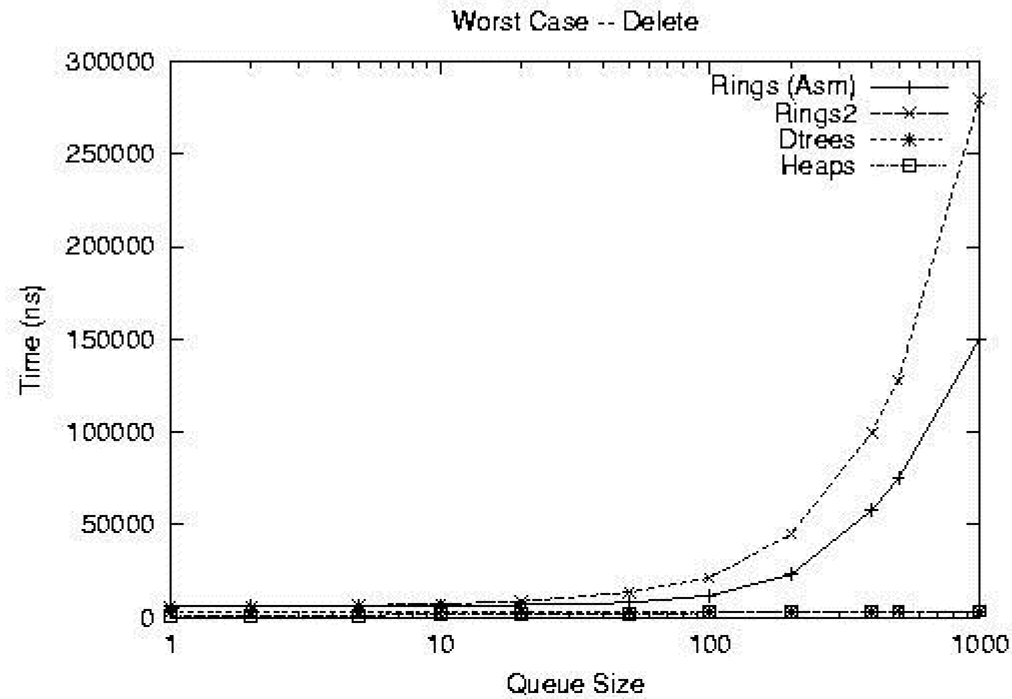

Figure 16. **Worst case Insert performance**

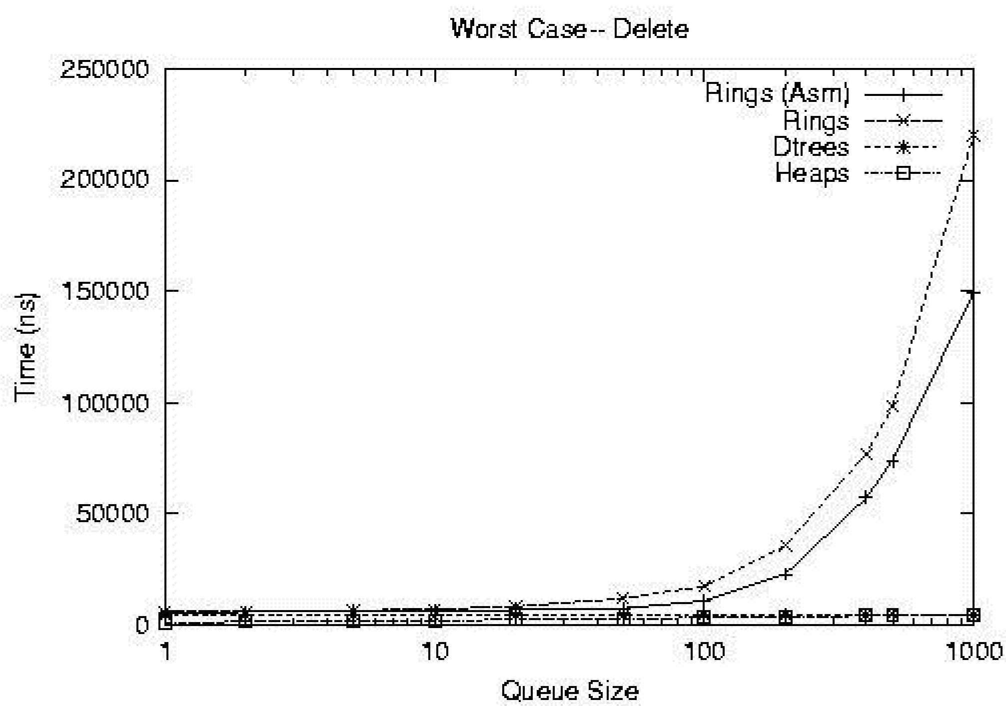

Figure 17. **Worst case Delete performance**

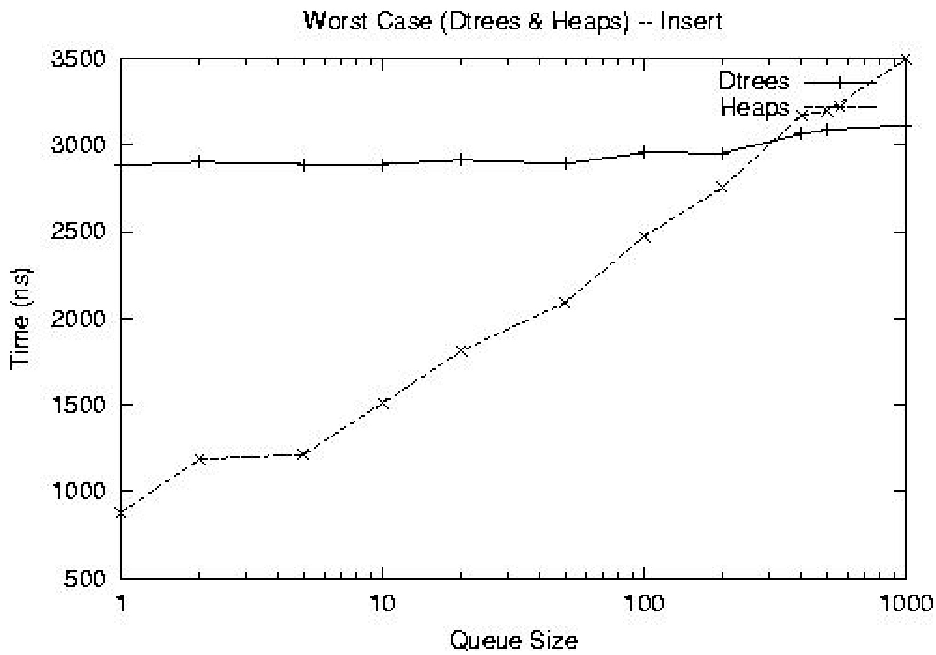

Figure 18. **Worst case results for D-Trees and Heaps (Insert)**

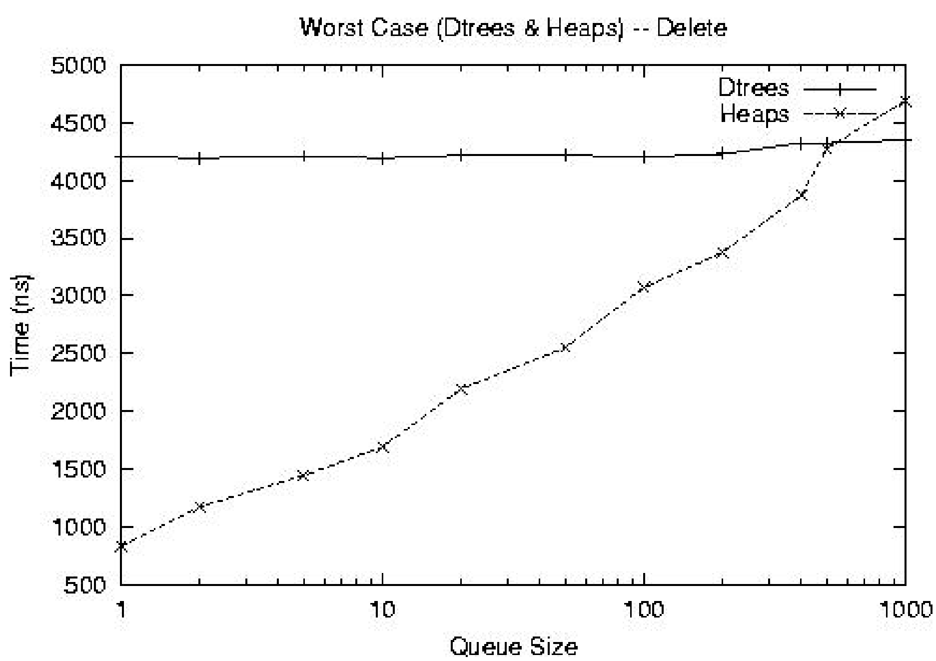

Figure 19. **Worst case results for D-Trees and Heaps (Delete)**

5. Conclusions

5.1 Recommendations

The algorithms tested here have been distinguished and compared based on their average-case and worst-case performance. Here we present a brief summary of our observations for each of the algorithms.

The Rings algorithm is stable and simple to implement. It has good average-case performance for small queue sizes, but extremely poor worst-case performance. However, implementing this algorithm in assembly language improves performance and increases the threshold of the queue size until which it can be used efficiently. The B-Trees algorithm is not stable and not a simple algorithm to implement and maintain. It has erratic average-case and poor worst-case performance. It is not recommended for use in a real-time environment. Both the Rings and B-Trees algorithms have the advantage that the queue size is not pre-determined by a certain limit.

The Heaps and D-Trees algorithms are both not stable, and have a pre-determined limit on the queue size. Heaps has fair average-case performance and good worst-case performance. D-Trees always perform better than Heaps for the average-case test, but were not as efficient as Rings for small queue sizes. They have extremely good worst-case performance that is as good as Heaps and more consistent as can be seen from the smooth graphs for its worst-case graphs. They however, use almost twice the memory required by Heaps.

The Bit Vector algorithm has the same disadvantages that Heaps and D-Trees posses. It is not stable, needs a lot of memory, and the maximum queue size has to be pre-determined. It shows constant time for Inserts, which is highly desirable. For the Delete operation, however, they perform better at longer queue sizes than for smaller queue sizes. In real-time systems, efficient performance at smaller queue sizes is more desirable.

5.2 Incomparable features

For some applications, features other performance may be the decisive factor in choosing a priority queue algorithm. Of all the algorithms tested here, only rings are stable. Bit-Vectors is the only one that must be implemented in assembly language. Finally, all algorithms except Rings and B-Trees require that the maximum number of elements that need to be accommodated in the queue be known *a priori*.

5.3 Future Work

We have analyzed some promising algorithms that can be used for priority queue implementation in real time systems. This work can be extended in the following areas:

- Two types of queues are usually needed for implementing a scheduling system, an active queue, and a timed queue. The active queue contains the currently active jobs sorted by priority, and the timed queue contains sleeping jobs sorted by wakeup times. At the end of a given time interval, usually more than one job wakes up, and thus we need to delete a set of jobs from the timed queue, and insert them in the active queue. The algorithms studied can be analyzed with respect to this requirement, i.e. insertion of a set of elements into the queue instead of one job at a time.
- The analysis can be performed with respect to multiprocessor systems. Performance measurements with respect to single queue and multiple queue multiprocessor systems can be carried out.

- Multiprocessor systems with a single queue have concurrency issues that can be studied. Resolution of these issues can be done using techniques like semaphores, mutexes, etc. These techniques can also be analyzed and measured empirically.

- The algorithms studied here were implemented using static priorities only. The implementation can be extended to use dynamic priorities. This has applications in hard real-time systems where the priorities might need to be adjusted at run-time to enable jobs to complete within their deadlines.

- Among the algorithms measured for worst-case performance, the algorithms showing better performance i.e. Heaps and D-Trees did not use linked allocation. This means that the number of jobs that can be accommodated has to be pre-determined. There is possibly an algorithm that can be evaluated that uses linked allocation (and thus dos have a bound on the number of jobs that it can accommodate) that has a performance comparable to these.

Appendix A

Measurement Data

The following tables contain all the measurement data obtained from the experiments. The average-case analysis data consists of four sets, corresponding to the four random number generator seeds used. The time measurements for average-case performance shown in the table are for 1000 Inserts or 1000 Deletes and are in terms of *nanoseconds*. The worst-case performance measurements are also in terms of *nanoseconds*. The data contained here was used for plotting the graphs presented earlier.

Queue Size	Set A	Set B	Set C	Set D
1	6541	6588	6554	6572
2	6519	6474	6481	6500
5	6439	6488	6550	6459
10	6493	6544	6787	6472
20	6477	6535	6489	6500
50	6587	6567	6571	6644
100	6700	6818	6736	6660
200	7443	7638	8138	7794
400	11339	12065	11470	11473
500	13971	14836	14572	14412
1000	28183	27655	30744	27953

Table 1. Rings (Assembly) – Average case Insert performance

Queue Size	Set A	Set B	Set C	Set D
1	5730	5599	5643	5622
2	5600	5620	5614	5595
5	5689	5620	5608	5631
10	5590	5642	5634	5623
20	5644	5628	5665	5650
50	5636	5611	5618	5603
100	5625	5624	5660	5653
200	5709	5714	5797	5776
400	5863	5887	5861	5955
500	5958	5981	6008	6012
1000	6247	6385	6251	6200

Table 2. Rings (Assembly) – Average case Delete performance

Queue Size	Set A	Set B	Set C	Set D
1	6681	6706	6759	6747
2	6580	6674	6600	6624
5	6720	6644	6725	6612
10	6670	6712	6634	6664
20	6776	6789	6696	6725
50	7116	7201	6993	7121
100	8135	8092	7945	7992
200	10961	10736	11307	10775
400	18627	19689	19446	19253
500	23383	24313	24653	24516
1000	47986	45976	52489	48186

Table 3. Rings – Average case Insert performance

Queue Size	Set A	Set B	Set C	Set D
1	6165	6249	6149	6171
2	6224	6174	6260	6158
5	6143	6154	6160	6304
10	6208	6135	6161	6218
20	6205	6153	6143	6144
50	6264	6179	6196	6149
100	6287	6303	6249	6221
200	6338	6269	6292	6291
400	6466	6487	6540	6483
500	6551	6586	6583	6640
1000	6808	6835	6910	6863

Table 4. Rings – Average case Delete performance

Queue Size	Set A	Set B	Set C	Set D
1	717	764	699	700
2	721	764	710	697
5	718	694	762	735
10	734	695	716	726
20	763	742	729	700
50	719	712	765	767
100	754	700	744	765
200	714	719	763	701
400	694	696	759	775
500	716	695	719	710
1000	724	761	763	741

Table 5. Bit Vector – Average case Insert performance

Queue Size	Set A	Set B	Set C	Set D
1	1767	1772	1778	1761
2	1759	1759	1759	1806
5	1758	1782	1761	1803
10	1774	1764	1761	1774
20	1767	1776	1769	1769
50	1746	1760	1740	1738
100	1737	1723	1717	1721
200	1713	1752	1690	1711
400	1687	1665	1647	1655
500	1633	1676	1644	1676
1000	1593	1604	1600	1616

Table 6. Bit Vector – Average case Delete performance

Queue Size	Set A	Set B	Set C	Set D
1	2574	2614	2570	2624
2	2696	3247	3148	3221
5	18997	22425	19313	4070
10	8479	9344	5968	6265
20	12586	14014	10696	9899
50	11694	11007	13379	11390
100	10741	11204	11184	11501
200	11251	12844	10001	10432
400	11219	10876	11181	11054
500	10401	10485	10820	11955
1000	9972	10182	10132	11200

Table 7. B-Trees – Average case Insert performance

Queue Size	Set A	Set B	Set C	Set C
1	5688	5730	5705	5660
2	5831	6302	6279	6242
5	11135	12013	11444	6881
10	8935	8992	7757	7958
20	10684	9829	9698	9373
50	10623	9939	11237	10468
100	10665	10701	10781	10731
200	11055	11692	10541	10642
400	11357	11230	11349	11219
500	11198	11274	11545	11839
1000	11564	11519	11526	11873

Table 8. B-Trees – Average case Delete performance

Queue Size	Set A	Set B	Set C	Set D
1	929	887	949	953
2	924	932	907	869
5	1306	1314	1272	1220
10	1583	1597	1371	1530
20	1721	1776	1712	1685
50	1983	2042	2046	1882
100	2207	2090	2064	2093
200	2222	2101	2076	2012
400	2003	1994	1985	2062
500	2009	2008	1988	2085
1000	1750	1747	1694	1752

Table 9. Heaps – Average case Insert performance

Queue Size	Set A	Set B	Set C	Set D
1	841	805	806	809
2	838	935	949	997
5	1465	1440	1446	1326
10	1868	1910	1635	1833
20	2163	2353	2080	2192
50	2844	2765	2690	2608
100	3212	3157	3270	3089
200	3640	3581	3678	3494
400	4110	4041	3977	4036
500	4162	4156	4171	4144
1000	4601	4570	4574	4572

Table 10. Heaps – Average case Delete performance

Queue Size	Set A	Set B	Set C	Set D
1	3136	3068	3096	3146
2	3092	3172	3164	3106
5	3098	3070	3149	3145
10	3069	3021	3017	3113
20	2989	2982	3102	3024
50	2904	2902	2911	2950
100	2758	2744	2679	2768
200	2534	2416	2424	2478
400	2092	2165	2197	2139
500	2034	2046	2058	2011
1000	1660	1687	1698	1641

Table 11. D-Trees – Average case Insert performance

Queue Size	Set A	Set B	Set C	Set D
1	5115	5054	5050	5033
2	5016	5014	5031	5116
5	5095	5117	5032	5073
10	5085	5118	5117	5017
20	5014	5139	5031	5051
50	5006	5085	5026	5061
100	4998	5084	5073	5009
200	4991	5043	5074	5049
400	4982	4983	5008	5020
500	5066	4985	4979	5053
1000	5009	4991	4964	4973

Table 12. D-Trees – Average case Delete performance

Queue Size	Rings (ASM)	Bit Vector	B-Trees	Rings (ADA)	D-Trees	Heaps
1	6541	717	2574	6681	3136	929
2	6519	721	2696	6580	3092	924
5	6439	718	18997	6720	3098	1306
10	6493	734	8479	6670	3069	1583
20	6477	763	12586	6776	2989	1721
50	6587	719	11694	7116	2904	1983
100	6700	754	10741	8135	2758	2207
200	7443	714	11251	10961	2534	2222
400	11339	694	11219	18627	2092	2003
500	13971	716	10401	23383	2034	2009
1000	28183	724	9972	47986	1660	1750

Table 13. Overall average case Insert performance

Queue Size	Rings (ASM)	Bit Vector	B-Trees	Rings (ADA)	D-Trees	Heaps
1	5730	1767	5688	6165	5115	841
2	5600	1759	5831	6224	5016	838
5	5689	1758	11135	6143	5095	1465
10	5590	1774	8935	6208	5085	1868
20	5644	1767	10684	6205	5014	2163
50	5636	1746	10623	6264	5006	2844
100	5625	1737	10665	6287	4998	3212
200	5709	1713	11055	6338	4991	3640
400	5863	1687	11357	6466	4982	4110
500	5958	1633	11198	6551	5066	4162
1000	6247	1593	11564	6808	5009	4601

Table 14. Overall average case Delete performance

Queue Size	Rings (ASM)	Rings (ADA)	D-Trees	Heaps
1	5973	5643	2882	879
2	6045	5810	2903	1187
5	6123	6472	2890	1214
10	6273	7226	2890	1510
20	6709	8706	2914	1811
50	7918	13522	2892	2092
100	11710	21339	2956	2472
200	23153	45145	2954	2757
400	58166	99986	3064	3172
500	74883	128242	3089	3196
1000	150437	279998	3114	3497

Table 15. Worst case Insert performance

Queue Size	Rings (ASM)	Rings (ADA)	D-Trees	Heaps
1	5528	5692	4209	831
2	5586	5769	4195	1173
5	5736	6261	4215	1444
10	5868	6849	4195	1693
20	6545	8064	4219	2194
50	7193	11705	4221	2554
100	10560	17295	4204	3073
200	22734	35496	4232	3379
400	57076	77017	4326	3874
500	74002	98169	4326	4277
1000	149335	220003	4353	4691

Table 16. Worst case Delete performance

Appendix B

This section contains three parts of the code that was developed for illustration purposes. The first part is the Ada specification that was used for implementing all the algorithms. The second contains the implementation of the D-Trees package body while the third contains the code for the Bit-Vector algorithm. D-Trees was chosen because it is as yet unpublished and was written in a high level language, while Bit Vector was chosen because it was implemented using assembly language.

Ada Package Specification

```
with Unchecked_Deallocation;

package Algorithm is
   subtype Item_Type is Integer range 1 .. 2000;
   type Queue_Record;
   type Queue_Type is access Queue_Record;
   type Queue_Record is
      record
         Item : Item_Type;
         Next  : Queue_Type;
         Prev  : Queue_Type;
      end record;

   function Empty (Queue : Queue_Type) return Boolean;

   function Member (Queue : Queue_Type; Item : Item_Type) return Boolean;

   function Min (Queue : Queue_Type) return Item_Type;
   pragma Inline (Min);

   procedure Delete (Queue : Queue_Type; Item : Item_Type);

   procedure Insert (Queue : Queue_Type; Item : Item_Type);

   procedure Make_Empty (Queue : out Queue_Type);

   function Allocate return Queue_Type;

   procedure Deallocate is new
     Unchecked_Deallocation (Queue_Record, Queue_Type);

end Algorithm;
```

D-Trees Package Implementation

```
package body Dtrees is

   function Min (Queue : Queue_Type) return Item_Type is
   begin
      return Queue (1);
   end Min;

   function Empty (Queue : Queue_Type) return Boolean is
   begin
      return Queue (1) = Non_Item;
   end Empty;

   procedure Make_Empty (Queue : in out Queue_Type) is
   begin
      if Queue = null then
         Queue := new Queue_Record;
      end if;
      for I in Tree_Position loop
         Queue (I) := Non_Item;
      end loop;
   end Make_Empty;

   procedure Apply (Queue : Queue_Type) is
   begin
      for I in Item_Type loop
         if Member (Queue, I) then
            P (I);
         end if;
      end loop;
   end Apply;

   function Member (Queue : Queue_Type; Item : Item_Type) return Boolean is
   begin
      return Queue (Item_Type'Pos (Item) + Off) = Item;
   end Member;

   procedure Insert (Queue : Queue_Type; Item : Item_Type) is
      L : Tree_Position := Item_Type'Pos (Item) + Off;
   begin
      loop
         Queue (L) := Item;
         exit when L = 1;       --  Stop when root position is reached.
         L := L / 2;            --  Move up to the parent.
         exit when Item >= Queue (L);
         --  Stop when there is no change.
      end loop;
   end Insert;

   procedure Delete (Queue : Queue_Type; Item : Item_Type) is
      L : Tree_Position := Item_Type'Pos (Item) + Off;
      P : Position;
      M : Possible_Item_Type := Non_Item;
      N : Possible_Item_Type;
   begin
      loop
         exit when Queue (L) = M;
         Queue (L) := M;
         exit when L = 1;
```

```
      P := L / 2;  --  L's parent.
      N := Queue (4 * P - L + 1); --  L's sibling.
      if M >= N then
         M := N;
      end if;
      L := P;
    end loop;
  end Delete;
end Dtrees;
```

Bit Vector Package Implementation

```
with System.Machine_Code;

use System.Machine_Code;

package body Algorithm is

  HT : constant String := "" & ASCII.HT;
  LFHT : constant String := ASCII.LF & ASCII.HT;

  function Allocate return Queue_Type is
    N : Queue_Type := new Queue_Record;
  begin
    --  aren't we returning a local pointer? this works because
    --  the memory is freed until the process exits, but should it be
    --  done this way?
    return N;
  end Allocate;

  function Empty (Queue : Queue_Type) return Boolean is
    Asmret : Integer := 0;
    Retval : Boolean := True;
  begin
    Asm ("pushl %%eax"          & LFHT &
         "pushl %%edx"          & LFHT &
         "pushl %%edi"          & LFHT &
         "movl  $0,%%eax"       & LFHT &
         "movl  4(%%edi),%%edx" & LFHT &        --  edx := Queue.Next
         "cmpl  %%edi,%%edx"    & LFHT &        --  Queue = Queue.Next ?
         "jnz   endempty"       & LFHT &
         "movl  $1,%%eax"       & LFHT &
         "endempty:"            &  HT  &
         "movl  %%eax,%0"       & LFHT &
         "popl %%edi"           & LFHT &
         "popl %%edx"           & LFHT &
         "popl %%eax",
         Outputs  => Integer'Asm_Output ("=m", Asmret),
         Inputs   => Queue_Type'Asm_Input ("D", Queue),
         Clobber  => "memory",
         Volatile => True);
    if Asmret /= 0 then
      Retval := True;
    else
      Retval := False;
    end if;
    return (Retval);
  end Empty;

  function Member (Queue : Queue_Type; Item : Item_Type) return Boolean is
```

```
  Asmret : Integer := 0;
  Retval : Boolean := True;
begin
  Asm ("pushl %%eax"          & LFHT &
         "pushl %%ebx"          & LFHT &
         "pushl %%ecx"          & LFHT &
         "pushl %%edx"          & LFHT &
         "pushl %%esi"          & LFHT &
         "pushl %%edi"          & LFHT &
         "movl  %%edx,%%ebx"    & LFHT &           --  N(BX) := QUEUE
         "back2:"               &  HT  &
         "movl  4(%%ebx),%%ebx" & LFHT &           --  N := N.NEXT
         "cmpl  %%edx,%%ebx"    & LFHT &           --  is N = QUEUE?
          "jz    false2"        & LFHT &           --  end of queue?
         "movl  (%%ebx),%%eax"  & LFHT &
         "cmpl  %%ecx,%%eax"    & LFHT &
         "jl    back2"          & LFHT &           --  is N.Item < Item?
         "je    true2"          & LFHT &           --  is N.Item = Item?
         "false2:"              &  HT  &
         "movl  $0,%%eax"       & LFHT &
         "jmp   exit2"          & LFHT &
         "true2:"               &  HT  &
         "movl  $1,%%eax"       & LFHT &
         "exit2:"               &  HT  &
         "movl  %%eax,%0"       & LFHT &
         "popl  %%edi"          & LFHT &
         "popl  %%esi"          & LFHT &
         "popl  %%edx"          & LFHT &
         "popl  %%ecx"          & LFHT &
         "popl  %%ebx"          & LFHT &
         "popl  %%eax",
         Outputs  => Integer'Asm_Output ("=m", Asmret),
         Inputs   => (Queue_Type'Asm_Input ("d", Queue),
                      Item_Type'Asm_Input ("c", Item)),
         Clobber  => "memory",
         Volatile => True);
  if Asmret /= 0 then
    Retval := True;
  else
    Retval := False;
  end if;
  return (Retval);
end Member;

function Min (Queue : Queue_Type) return Item_Type is
  Item : Item_Type;
begin
  Asm ("pushl  %%eax"             & LFHT &
      "movl   4(%%ebx),%%ebx"    & LFHT &   --  get first element
      "movl   (%%ebx),%%eax"     & LFHT &
      "movl   %%eax,%0"          & LFHT &
      "popl   %%eax",
      Outputs  => Item_Type'Asm_Output ("=m", Item),
      Inputs   => Queue_Type'Asm_Input ("b", Queue),
      Clobber  => "memory",
      Volatile => True);
  return (Item);
end Min;

procedure Delete (Queue : Queue_Type; Item : Item_Type) is
```

```
   Deleted_Item : Queue_Type := null;
begin
   Asm ("pushl    %%eax"          & LFHT &
        "pushl    %%ebx"          & LFHT &
        "pushl    %%ecx"          & LFHT &
        "pushl    %%edx"          & LFHT &
        "pushl    %%esi"          & LFHT &
        "pushl    %%edi"          & LFHT &
        "movl     %%edx,%%ebx"    & LFHT &        --  n(bx) := queue
        "back3:"                  &  HT  &
        "movl     4(%%ebx),%%ebx" & LFHT &        --  n := n.next
        "cmpl     %%edx,%%ebx"    & LFHT &        --  is n = queue?
        "jz       exit3"          & LFHT &
        "movl     (%%ebx),%%eax"  & LFHT &
        "cmpl     %%ecx,%%eax"    & LFHT &        --  n.item = item?
        "jl       back3"          & LFHT &        --  less
        "jg       exit3"          & LFHT &        --  greater, item absent
        "movl     4(%%ebx),%%eax" & LFHT &        --  found
        "movl     8(%%ebx),%%ecx" & LFHT &
        "movl     %%ebx,%%edx"    & LFHT &
        "movl     %%eax,%%ebx"    & LFHT &        --  memory leak?!!
        "movl     %%ecx,8(%%ebx)" & LFHT &
        "movl     %%ecx,%%ebx"    & LFHT &
        "movl     %%eax,4(%%ebx)" & LFHT &
        "movl     %%edx,%%eax"    & LFHT &
        "movl     %%edx,%0"       & LFHT &
        "exit3:"                  &  HT  &
        "popl     %%edi"          & LFHT &
        "popl     %%esi"          & LFHT &
        "popl     %%edx"          & LFHT &
        "popl     %%ecx"          & LFHT &
        "popl     %%ebx"          & LFHT &
        "popl     %%eax",
        Outputs  => Queue_Type'Asm_Output ("=m", Deleted_Item),
        Inputs   => (Queue_Type'Asm_Input ("d", Queue),
                     Item_Type'Asm_Input ("c", Item)),
        Clobber  => "memory",
        Volatile => True);
   Deallocate (Deleted_Item);
end Delete;

procedure Make_Empty (Queue : out Queue_Type) is
begin
   Asm ("pushl  %%eax"              & LFHT &
        "call   algorithm__allocate" & LFHT &     --  queue is in eax
        "movl   %%eax,4(%%eax)"     & LFHT &     --  Q.Next := Q
        "movl   %%eax,8(%%eax)"     & LFHT &     --  Q.Prev := Q
        "movl   %%eax,%0"           & LFHT &
        "popl   %%eax",
        Outputs  => Queue_Type'Asm_Output ("=m", Queue),
        Inputs   => No_Input_Operands,
        Clobber  => "memory",
        Volatile => True);
end Make_Empty;

procedure Insert (Queue : Queue_Type; Item : Item_Type) is
begin
   Asm ("pushl    %%eax"          & LFHT &
        "pushl    %%ebx"          & LFHT &
        "pushl    %%ecx"          & LFHT &
```

```
      "pushl    %%edx"          & LFHT &
      "pushl    %%esi"          & LFHT &
      "pushl    %%edi"          & LFHT &
      "movl     %%edx,%%ebx"    & LFHT &            --  BX := QUEUE
      "back4:"                  &  HT  &
      "movl     4(%%ebx),%%ebx" & LFHT &            --  N := N.Next
      "cmpl     %%edx,%%ebx"    & LFHT &            --  is N = Queue ?
      "jz       last4"          & LFHT &
      "movl     (%%ebx),%%eax"  & LFHT &
      "cmpl     %%ecx,%%eax"    & LFHT &            --  is N.Item = Item?
      "jl       back4"          & LFHT &
      "jz       exit4"          & LFHT &
      "last4:"                  &  HT  &
      "pushl    %%ebx"          & LFHT &
      "pushl    %%ecx"          & LFHT &
      "call     algorithm__allocate"    & LFHT &     --  new node in eax
      "popl     %%ecx"          & LFHT &
      "popl     %%ebx"          & LFHT &
      "movl     %%ebx,%%esi"    & LFHT &      --  N
      "movl     %%eax,%%edi"    & LFHT &      --  T
      "movl     %%ecx,(%%edi)"  & LFHT &      --  T.ITEM:=ITEM
      "movl     %%ebx,4(%%edi)" & LFHT &      --  T.NEXT:=N
      "movl     8(%%ebx),%%edx" & LFHT &      --  K:=N.PREV
      "movl     %%edx,8(%%edi)" & LFHT &      --  T.PREV:=K (N.PREV)
      "movl     %%edi,8(%%ebx)" & LFHT &      --  N.PREV:=T
      "movl     8(%%edi),%%esi" & LFHT &      --  L:=T.PREV
      "movl     %%edi,4(%%esi)" & LFHT &      --  L.NEXT (T.PREV.NEXT):=T
      "exit4:"                  &  HT  &
      "popl     %%edi"          & LFHT &
      "popl     %%esi"          & LFHT &
      "popl     %%edx"          & LFHT &
      "popl     %%ecx"          & LFHT &
      "popl     %%ebx"          & LFHT &
      "popl     %%eax",
      Outputs  => No_Output_Operands,
      Inputs   => (Queue_Type'Asm_Input ("d", Queue),
                   Item_Type'Asm_Input ("c", Item)),
      Clobber  => "memory",
      Volatile => True);
   end Insert;

end Algorithm;
```

References

1. Maheshwari, R., *An empirical evaluation of priority queue algorithms for real-time applications*, in *Department of Computer Science*. 1990, Florida State University.

2. Vuillemin, J., *A Data Structure for Manipulating Priority Queues.* Communications of the ACM, 1978. **21**(4): p. 309-315.

3. Sleator, D.D. and R.E. Tarjan, *Self-Adjusting Heaps.* SIAM Journal on Computing, 1986. **15**(1): p. 52-69.

4. Decker, R., *Data Structures*. 1989: Prentice Hall, Englewood Cliffs, New Jersey.

5. Knuth, D.E., *Sorting and Searching*, in *The Art of Computer Programming*. 1973, Addison-Wesley Publishing Company, Reading, Massachusetts.

6. Jones, D.W., *An Empirical Comparison of Priority-Queue and Event-Set Implementations.* Communications of the ACM, 1986. **29**(4): p. 300-311.

7. Bayer, R. and E.M. McCreight, *Organization and Maintainence of Large Ordered Indices.* Acta Informatica 1, 1972. **3**: p. 173-189.

8. Adel'son-Velskii, G.M. and G.M. Landis, *An Algorithm for the Organization of Information.* Soviet Math. Dokl., 1962. **3**: p. 1259-1262.

9. Van Wyck, C.J., *Data Structures and C Programs*. 1988: Addison Wesley Publishing Company, Reading, Massachusets.

10. Jonassen, A. and O.J. Dahl, *Analysis of an Algorithm for Priority Queue Administration.* BIT, 1975. **15**(4): p. 409-422.

11. Brown, M.R., *Implementation and Analysis of Binomial Queue Algorithms.* SIAM Journal on Computing 7, 1978. **3**: p. 298-319.

12. Francon, J., G. Vienot, and J. Vuillemin. *Description and Analysis of an Efficient Priority Queue Representation*. in *19th Annual Symposium on Foundations of Computer Science*. 1978. Ann Arbor, Michigan: IEEE, Piscataway, New Jersey.

13. Fischer, M.J. and M.S. Paterson. *Fishspear: A Priority Queue Algorithm*. in *25th Annual Symposium on Foundations of Computer Science*. 1984.

14. Sleator, D.D. and R.E. Tarjan, *Self-adjusting Binary Search Trees.* Journal of the ACM, 1985. **32**(3): p. 652-686.

15. Ronngren, R., J. Riboe, and R. Ayani. *Lazy Queue: An Efficient Implementation of the Pending-event Set*. in *The 24th Annual Simulation Symposium*. 1991. New Orleans, Louisiana: IEEE.

16. Ada Core Technologies, I., GNAT web page: http://www.gnat.com.

17. Barabanov, M. and V. Yodaiken, *Introducing Real-Time Linux.* Linux Journal, 1997(34): p. http://www.ssc.com/lj/issue34/0232.html.

18. McCormack, W.M. and R.G. Sargent, *Analysis of Future Event Set Algorithms for Discrete Event Simulation.* Communications of the ACM, 1981. **24**(12): p. 801-812.