# COMPARATIVE STUDY OF PRIORITY QUEUES IMPLEMENTATION

By

DONGHONG WEI

Bachelor of Foreign Languages

Shanxi University

Taiyuan, China

1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
December, 1999

COMPARATIVE STUDY OF PRIORITY QUEUES

IMPLEMENTATION

Thesis Approved:

Jacques E. LaFrance

J Chandler

Helen C

Wayne B. Powell

Dean of the Graduate College

## ACKNOWLEDGMENTS

I wish to express my deepest appreciation to my advisor Dr. Jacques LaFrance for accepting to be my major advisor. His enthusiastic support, constructive guidance, encouragement, and friendship throughout time of acquaintance have been a constant source of inspiration and motivation that helped me gain confidence academically and professionally. My sincere appreciation also extends to Dr. John P. Chandler whose intelligent suggestion and guidance have made this thesis feasible. I would like to thank Dr. H. K. Dai for his constructive criticism, direction and wisdom. Grateful appreciation also goes to the faculty and students of the Department of Computer Science for their interest, guidance, and friendship.

I wish to sincerely thank my husband for his unconditional love, spiritual support and endless encouragement.

To my parents, Shuren Wei and Shuzhen Ma, I want to tell them how much I love them and that I could not have made it through graduate school without their total support and consistent encouragement.

I dedicate this thesis to my dearest daughter, Connie Yang, who is the reason why I have worked so feverishly to complete this project.

Finally, I would like to praise God for providing me with confidence, wisdom, persistence and strength that guided me through the entire process.

TABLE OF CONTENTS

## LIST OF FIGURES

Figure Page

# CHAPTER I

# INTRODUCTION

Priority queues are used in a wide variety of applications including operating systems, real-time systems and discrete event simulations [Ronngren, 1997]. In a priority queue, each element is ordered by its associated priority [Ayanne, 1990]. The basic operations are dequeue and enqueue. A dequeue operation removes the element with the highest priority, and an enqueue inserts a new element into the queue. Ordinary stacks and queues are special cases of priority queues [Brown, 1988].

The implementation of the priority queues may have a profound effect on the performance of such applications. Over the years, several performance studies on priority queues have appeared in the literature. A significant number of these studies have been performed in the context of discrete event simulation (DES) [Jones, 1986]. The reason for studying priority queues is twofold: the specific implementation is often crucial to the performance of the simulator, and the way operations are performed on the pending event set provides an excellent test case for studying priority queues [Fujimoto, 1990]. The impact of the implementation of the pending event set can have a super linear effect on the performance [Ronngren, 1993]. Although priority queues are used in various contexts, they are some general quality measures of interest. The most important metric is the time required to perform the most common operations that are dequeue and enqueue, and we refer to this time as access time. In most cases, the measure of interest

is the amortized access time. Thus it is important to find methods that allow a realistic and accurate assessment of the access time.

Up till now, the most widely used method for performance studies of priority queues has been the Classic Hold introduced by Vaucher and Duval [Duval, 1975] and refined by Jones [Jones, 1986]. It models operation on a fixed-size queue where a series of hold operations (a dequeue followed by an enqueue) are performed. An Up/Down model is proposed by Ronngren et al [Ronngren, 1993], where a sequence of enqueues is followed by an equally long sequence of dequeues. Markov Hold is proposed by Chung et al. [Chung, 1993], where operations on the queue are determined by a two-state Markov process with states insert (enqueue) and delete (dequeue). By changing the transition probabilities, the Markov Hold model can represent random sequences of enqueue and dequeue operations.

Although the three methods mentioned above have been introduced to perform the studies of priority queues, comparatively studying priority queues with the three methods and examining the different priority queue algorithms for performing the event-list in discrete event simulation programs have not been studied before.

In this thesis I explored the three methods to study the behavior of three different priority queues, which are array-based Implicit Binary Heap, linked-list-based Median Pointer Linked List and lazy-queue-based SPEEDESQ. Comparing the performances of the three different typical priority queues with the three models, this thesis provides readers with an accurate and realistic analysis of each priority queue's access time complexity for the simulation. Readers can choose the best priority queue or access

pattern depending on the application. The thrust of this thesis is to provide readers a paradigm for the study of computation complexity of comparison based problems.

Chapter 2 provides a review of the priority queue data structures and the implementation of the pending event set by use of priority queues. Chapter 3 provides a discussion of the design and the implementation details of the software that is developed as part of the thesis. The analysis and evaluation of the software developed are discussed in Chapter 4. The thesis ends with Chapter 5 that provides a summary and the conclusions drawn from the study.

# CHAPTER II

## LITERATURE REVIEW

An abstract data type (ADT) is a data type along with the set of operations of that type. Abstract data types are mathematical abstractions. They can be viewed as an extension of modular design [Weiss, 1997]. A Queue is one of the most simple and basic ADT, as John Beldler mentioned [Beldler, 1996]:

> A queue is a sequential, homogeneous, variable-sized, possibly empty collection of objects whose attributes and operations satisfy the following:
> 1. A queue is said to be empty when it contains no objects.
> 2. A queue has two ends, called the front and the rear.
> 3. The only object in a queue that is visible is the object at the front of the queue.
> 4. The dequeue operation removes the object currently at the front of the queue. All remaining objects in the queue, if any, move one position forward toward the front of the queue. The object immediately following the front object becomes the new front of the queue. If there are no other objects in the queue, the queue becomes empty.
> 5 The enqueue operation inserts new objects at the rear of the queue. If the queue was empty, the enqueued object becomes the front object in the queue. At any time, new objects may be enqueued. When an object is enqueued, it becomes the rear object in the queue.

The queue constructors modify the queue either by removing the object at the front of the queue or by adding new objects to the rear of the queue [Beldler, 1996], therefore, the basic operations are dequeue and enqueue. As the enqueue inserts an element in the rear of the list, the queue expands. As the dequeue deletes the element at the front of the list, the queue shrinks [Weiss, 1997]. The order of the objects in the queue, from front to rear, is the order in which the objects were enqueued. The term FIFO, first-in-first-out, describes the order of processing of the objects in a queue.

However, sometimes some types of data in the queue require the data to be deleted according to a priority rather than FIFO. Any data structure that supports the operations of searching, inserting, and deleting minimum or maximum is called a priority queue [Horowitz, 1996].

The priority queue is a structure for storing and retrieving information [Beldler, 1996]. It has at least the following two operations: dequeue and enqueue. A dequeue operation removes the element with the highest priority, and an enqueue operation inserts a new element into the queue. Figure 1 shows the basic model of a priority queue.

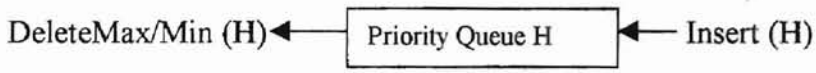



**Figure 1. The Basic Operation Model of a Priority Queue**

In the following two sections, I will review seven well-known priority queue data structures and discrete event simulation.

### 2.1 Priority Queue Data Structures

There are obviously several ways to implement the priority queues. The following seven well-known data structures will be introduced: Implicit Binary Heaps [Bentley, 1985], Median Pointer Linked Lists, Skew Heaps [Sleator and Tarjan, 1986], Calendar Queue [Brown, 1988], Henriksen's [Henriksen, 1997], the Lazy Queue [Ronngren,1993a] and the SPEEDESQ [Steinman, 1992].

#### 2.1.1 Implicit Binary Heap

A heap [Aho, 1974] is a standard data structure for implementing priority queues. An Implicit Binary Heap is one of the most elegant of storage structures to represent a priority queue. The Implicit Binary Heap is defined as a structure on locations 1 through i of an array with the property that the element in location i is smaller than that in location

[i/2], thus including a complete binary tree with the property that the value of the parent is greater than that of the child [Munro,1979]. Such a pointer free representation has been called an implicit data structure. It is well known that the Implicit Binary Heap enables us to perform the basic priority queue operations. In an enqueue operation, the new element is placed at the base of the heap (i.e., as the rightmost leaf at the lowest level). To restore the heap property (i.e., that each parent has a higher priority than any of its children), the new element is compared to its parent and they are swapped if necessary. This process has to be repeated upward in the heap until either the root is reached or at some level no swap is needed. In the dequeue operation, delete the element with the highest priority and then re-order the heap. Both the enqueue and dequeue operations have a worst-case behavior of O (log N) [Gaston, 1986], where N is the number of elements in the queue.

### 2.1.2 Median Pointer Linked List

The Median Pointer Linked List was implemented as a Doubly Linked Circular List that allows insertions to take place from both the front and the back [Weiss, 1997]. A pointer to the median element (with respect to the number of elements) in the list is used to identify whether the insertions should be made from the front (time-stamp of the new element is less than or equal to that of the median element) or the back end of the list [McCormack and Sargent, 1981]. A dequeue operation on a Median Pointer Linked list is performed in constant time, whereas the enqueue operation is O(N).

### 2.1.3 Skew Heap

The Skew Heap [Sleator and Tarjan 1986](sometimes called a priority queue or mergeable heap) is a self-adjusting forming heap-ordered binary tree where any

descendant of a node has lower priority than the node itself. The self-adjusting data structures have the following advantages: 1) They need less space, since no balance information is kept. 2) Their access and update algorithms are easy to understand and to implement. 3) In an amortized sense, ignoring constant factors, they can be at least as efficient as balanced structures [Sleator, 1986]. The fundamental operation on the Skew Heaps is a meld operation. A meld operation merges two Skew Heaps into one, preserving the heap property. Thus a dequeue operation is performed by removing the topmost (root) node and melding the two resulting subheaps into a single heap. In the top-down Skew Heap, an enqueue operation is performed as a meld of a one-node Skew Heap and the existing Skew Heap. The amortized time to perform a dequeue or an enqueue operation is O(logN) [Sleator and Tarjan 1986] although individual operations may be O(N) if the heuristics for the balancing of the heap fails.

### 2.1.4 Calendar Queue

The Calendar Queue suggested by Brown is a multilist-based data structure[Brown, 1988]. It is a new priority queue implementation for the future event set problem. It uses an elegant technique to manage the overflow problem encountered in multilist-based implementations. Calendar Queue is modeled after a desk calendar. In the Calendar queue, there exists no dedicated overflow structure. All elements, including those that would fall into an overflow structure in an ordinary multilist, are inserted into the sublists. This is accomplished in the following way: all sublists span equally long priority (time) intervals where the total length of these subintervals is called a year. When a new element is inserted into the queue, the sublist into which the new element will fall is calculated. To ensure good performance, it is required that the sublists, which are

implemented as linked lists, are kept short (an average length of two elements). This is accomplished by a resize operation. The resize operation is performed when the queue size has changed by a factor of two. The new length of the subintervals is calculated by using an approximation of the distribution based on the first few elements [Brown 1998]. The Calendar Queue has O(1) access time under many operating conditions although the resize operations are O(N). The worst-case amortized access time for the Calendar Queue is O(N).

### 2.1.5 Lazy Queue

The Lazy Queue [Ronngren et al., 1993] is a multilist-oriented data structure. The fundamental idea is to divide the future events into several parts and keep only a small portion of the elements completely sorted [Robert, 1997]. The elements are divided into: 1) A near future that is kept sorted, 2) A far future that is partially sorted, and 3) A very far future that is used as an overflow bucket. As time advances, part of the far future is sorted by standard sorting techniques and transferred into the near future. This lazy sorting behavior gives the queue its name. The far future part of the queue is implemented as an array of unsorted sublists where each sublist corresponds to an equally long priority interval. The near future consists of a sorted array of elements (transferred from the far future) and a skew heap is used to insert the new elements that belong to the near future. Skew heaps are used to implement the very far future of the lazy queue. To ensure good utilization of the data structures, a set of resize operations was introduced. Some modifications in the implementations resize operations have been performed that improve the performance for smaller queue sizes as compared to the results presented in [Ronngren et al. 1993a]. In particular, there is a resize operation that recalculates both

the length of a subinterval and the number of subintervals. Ronngren's operation is used whenever a resize operation is initiated. However, the criteria for expensive, worst case of O(NlogN), but they are amortized over the relatively inexpensive ordinary operations. This results in a near O(1) access time for many operation conditions. The worst-case amortized access time can be restricted to O(N) [Ronngren et al., 1993]. A minimum queue size of 256 elements is requested to perform a resize operation. The Lazy queue is stable only if the sorting algorithm and the implementations of the near and very far futures are stable.

### 2.1.6 Henriksen's Data-Structure

Henriksen's [Henriksen, 1977] implementation of the pending event set employed in GPSS/H [Gordon, 1981] uses a linked list and an array of pointers into the list. The array of pointers is used to perform binary searches in the list to find the place where a new element should be put in an enqueue operation. A heuristic algorithm is employed to update the auxiliary pointers. The implementation was based on the code given in Kingston where the array of pointers is used as a circular list of pointers [Kingston, 1986]. The size of the array of pointers is doubled when necessary as the size grows. However, the array is not decreased in size if the queue size shrinks. The amortized access time of Henriksen's algorithm is often O (log N), and limited by $O\sqrt{n}$ in the worst case [Kingston, 1986]. Dequeue operations are performed in constant time, whereas enqueue operation can be O ($\sqrt{n}$).

### 2.1.7 SPEEDESQ

The SPEEDESQ [Steinman 1992] consists of two single linear linked lists. One list, referred to as the "dequeue-list", is kept sorted and the other list, the "enqueue-list" is

unsorted. New elements are added to the enqueue-list, which can be done in constant time since the list is kept unordered. The queue also maintains a variable recording of the highest priority (smallest time-stamp) of any element present in the enqueue-list. A dequeue operation removes the element with the highest priority from the dequeue-list. In a dequeue operation, the enqueue-list is sorted and merged with the dequeue-list if the dequeue-list is exhausted or whenever the highest priority element is present in the enqueue-list. The merge operation is potentially an O(N) which, if frequent, result in a worst-case performance of O(N). SPEEDESQ has constant enqueue time and a constant time for many of the dequeue operations. However, dequeue operations that involve sorting of the enqueue-list have an O(NlogN) time complexity.

### 2.1.8 Expected Performance of the Priority Queues

Table I Summarizes the theoretically expected performance of the sequential priority queues.

**Table 1. Expected Performance of Sequential Priority Queues**

| Queue | Enqueue (best, average) | Enqueue (worst case) | Dequeue (best, average) | Dequeue (worst case) |
|---|---|---|---|---|
| Calendar queue | O(1), O(N) | O(N) | O(1), O(N) | O(N) |
| Henriksen's | O(log N), O($\sqrt{n}$) | O(N) | O(1), O(1) | O(1) |
| Skew Heap | O(1), O(log N) | O(N) | O(1), O(log(N)) | O(N) |
| Lazy Queue | O(1), O(N) | O(Nlog N) | O(1), O(N) | O(NlogN) |
| Implicit Binary Heap | O(1), O(1) | O(logN) | O(1), O(logN) | O(logN) |
| SPEEDESQ | O(1), O(1) | O(1) | O(1), O(N) | O(NlogN) |
| Median Pointer Linked List | O(1), O(N) | O(N) | O(1), O(1) | O(1) |

## 2.2 Priority Queue and Discrete Event Simulation (DES)

Priority queues are used in many applications including real-time systems, operating systems and simulations. Typical applications require primitive operations among the following five: INSERT, DELETE, MIN, UPDATE, AND UNION. The

operation INSERT (name, label, Q) adds an element to queue Q. DELETE (name) removes the element. Operation MIN (Q) returns the name of the element in Q having the least label, and UPDATE (name, label) changes the label of the element named. UNION (Q1, Q2, Q3) merges into Q3 all elements of Q1 and Q2; the sets Q1 and Q2 become empty [Chung, 1993].

A variety of applications directly require using priority queues are: job scheduling, discrete simulation languages where labels represent the time at which events are to occur, as well as various sorting problems [Jean, 1997]. Priority queues also play a central role in several good algorithms such as optimal code constructions, Chartre's prime number generator and Brown's power series multiplication. The applications have also been found in numerical analysis algorithms and in graph algorithms for such problems as finding shortest paths and minimum cost spanning tree [Jean, 1997].

The most popular use of priority queues is in the area of discrete event simulations (DES) [Ronngren, 1997]. Here, a priority queue is used to hold the pending event set (PES) which contains the generated but not yet evaluated events. In discrete event simulations, events are simulated as occurring at discrete times determined by random variables. A data structure stores unprocessed events. The basic action is to remove the event with the earliest time and process it. This processing may create new events which must be inserted into the data structure and which will be processed in the future. To accomplish this, the data structure should support two operations: insert and delete-min. The data structure, which stores the events (according to their execution times), is called an event-list. The event-list can be represented as a priority queue in which priorities are assigned according to the time, the higher priority given to the item

with the smallest value. The Pending event set (PES) is the set of all generated but not yet evaluated events and, in general, is represented by a priority queue. The implementation of the PES is often crucial to simulation performance. An empirical study by Comfort [Comfort,1984] indicated that up to 40% of the simulation execution time might be spent on the management of the PES alone. Therefore, as systems become more complex and a demand for fast simulators arises efficient implementation of the PES becomes increasingly important [Ronngren, 1997].

In a PES, the simulated times at which the events are scheduled to be executed (time-stamps) are used as priorities. A sequential discrete event simulator operates in a three-step cycle: remove the events with the smallest time stamp (i.e., highest priority) from the PES; execute this event; and insert any new events resulting from this execution into the PES. Thus the two most common operations on the PES data structure are: dequeues, the removal of the event with the highest priority, and enqueue, the insertion of a new event. Empirical studies of real simulations [Comfort, 1984] indicate that these two operations can account for as much as 98% of all operations on the PES, the rest being other operations such as deletion of arbitrary events and the like. The performance of the PES is influenced by a number of variables including the initial distribution of events, the priority increment distributions, access patterns, and the size of the event set. Thus the event set implementation must be efficient under a wide variety of operation conditions and possibly by adaptive to take advantage of these conditions [Jones, 1986].

The requirements of high performance simulation of complex systems and the observation that these systems are often inherently parallel have motivated the development of parallel discrete event simulation[ Fujimoto, 1990]. In parallel DES

(PDES), the inherent parallelism that exists in most simulation models is realized by allowing logical processes (LPs) to be executed in parallel using several processing units[Steinman, 1992]. In PDES priority queues are also often used as scheduling queues for LPs that are ready to execute. This queue may also be shared by several processing elements.

In this thesis I comparatively studied the performance of the different priority queues in the discrete event simulation programs.

# CHAPTER III

# DESIGN AND IMPLEMENTATION ISSUES

## 3.1 Measurement Techniques

When designing experiments to study priority queues, it is important to carefully choose access patterns and measurement techniques. The choices should reflect the operating conditions under which the priority queue will be used as well as enabling accurate measurement of the performance metrics of interest. When selecting access patterns for this thesis, I chose synthetic experiments over real simulations. Synthetic experiments provide better control over the variables affecting performance, therefore they better expose the factors that influence performance [Jones, 1986].

### 3.1.1 Access Patterns

There are three classes of access patterns used to implement the simulation: Classic Hold, Up/Down Model, and Markov Hold.

#### Classic Hold

Up till now, the most widely used method for performance studies for priority queues has been the Classic Hold (See Figure 2) introduced by Vaucher and Duval and refined by Jones. A hold operation is defined as a dequeue followed by an enqueue operation. The Classic Hold models the behavior of a discrete event simulation system performing a sequence of operations. In the Classic Hold experiments, the queue is initialized to a fixed size and the measurement phase, which consists of a number of

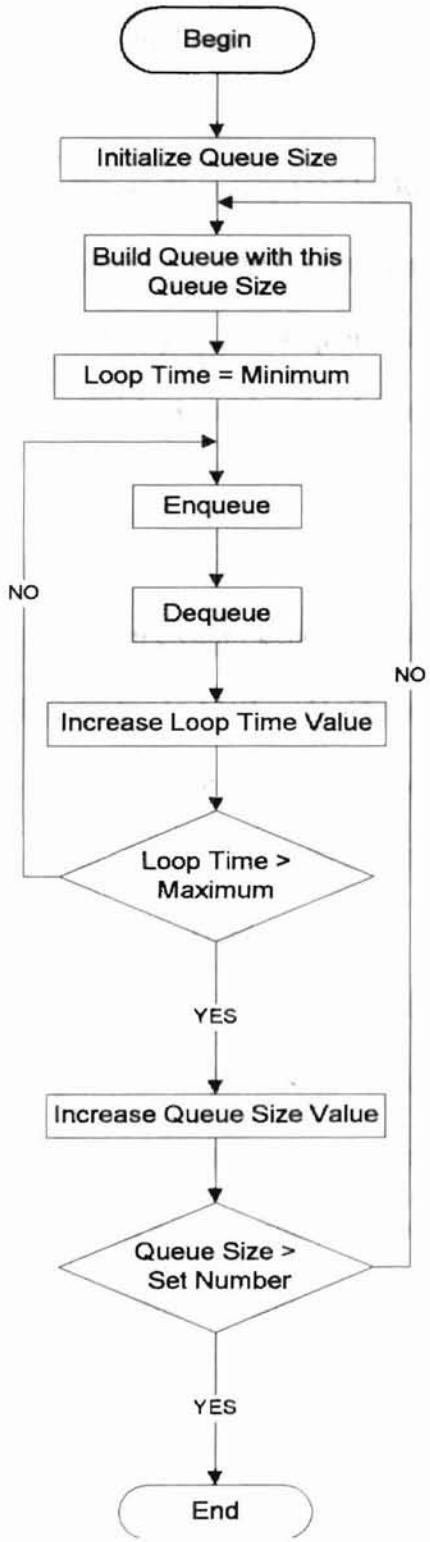



**Figure 2. Classic Hold Flow Chart**

hold operations, is performed. The queue size having been built remains fixed throughout the process of the Classic Hold operation.

**Up/Down Model**

An Up/Down Model (See Figure 3) is proposed by Ronngren et al. [Ronngren, 1993], model the PES's transient behavior where the queue grows to a certain size by a sequence of enqueues and then shrinks by a same sequence of dequeues. In the experiment, the measurements start and end with a fixed size queue for this access pattern.

**Markov Hold**

Chung et al. [1993] proposed an elegant generalization of the Classic Hold, called the Markov Hold model (See Figure 4). In Markov Hold, the operations on the priority queue are generated by a two-state Markov process that may be in either of the states insert (enqueue) or delete (dequeue). By changing the state transition probabilities this model can be used to represent the Classic Hold, Up/Down, and a generalized random sequence of enqueue and dequeue operations. The Markov model can be used to generate access patterns equivalent to the three classes mentioned.

In the Markov Hold simulation, the state transition is generated by the function *randomNum* ( ). If the function returns even number, enqueue operation is performed. If the function returns odd number , dequeue operation is performed.

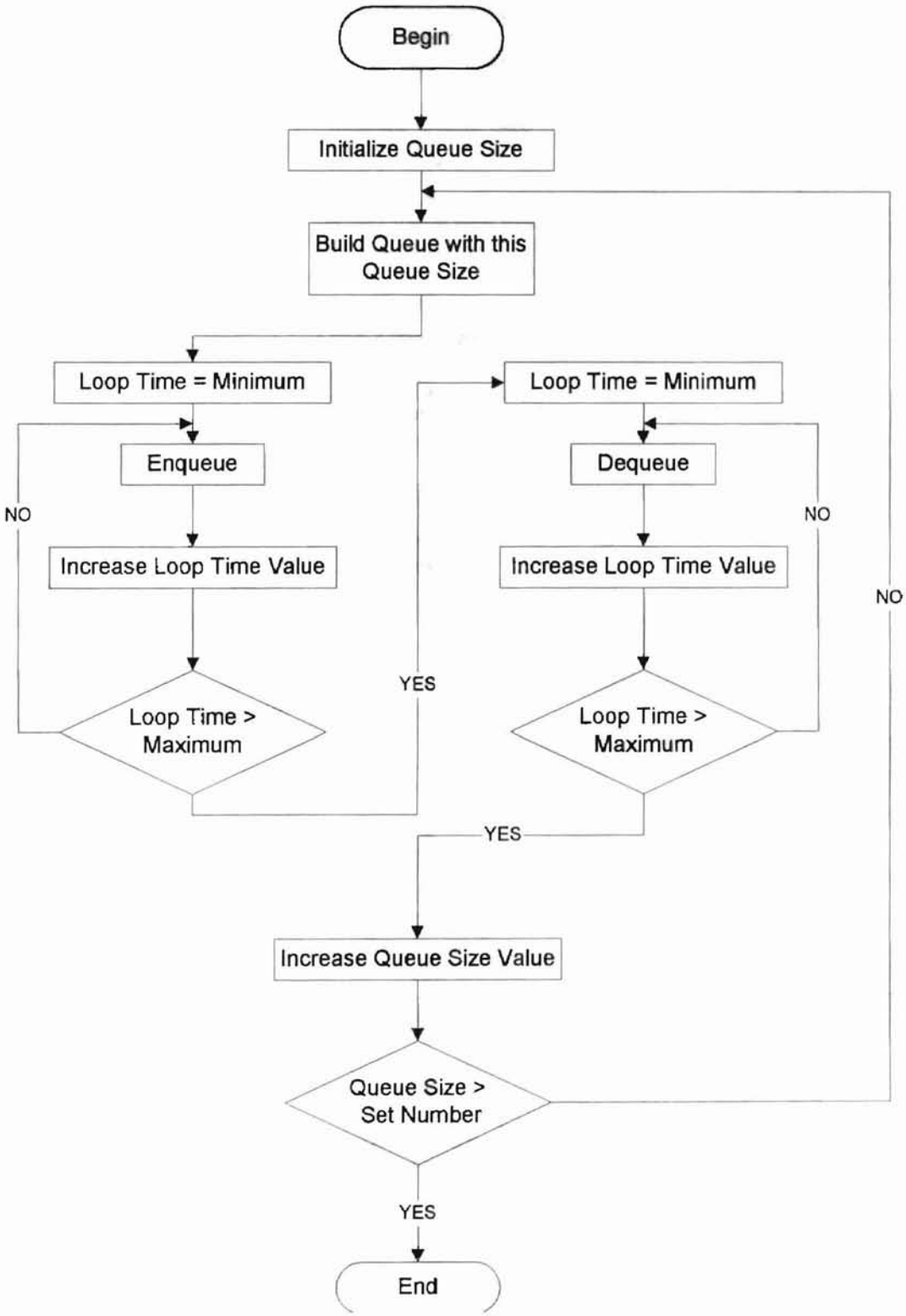



**Figure 3. Up/Down Model Flow Chart**

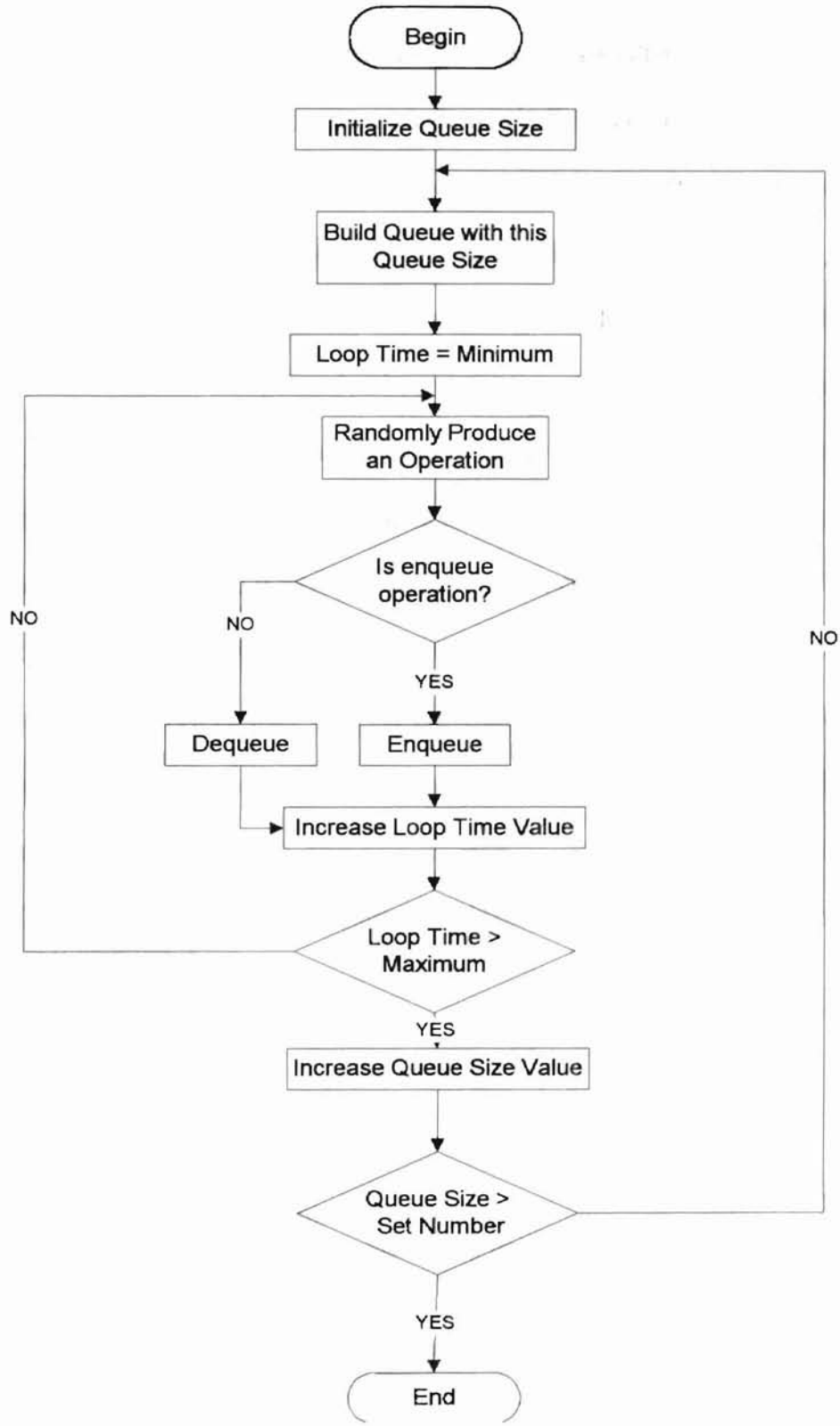



**Figure 4. Markov Hold Model Flow Chart**

### 3.1.2 Time Measurements

All the simulation was performed on the Compaq Proliant 300 Server with 2 processors of 300MHz. The program code was built and compiled with the Microsoft Visual C++ under the Windows 95 environment. Due to insufficient resolution of the available time measurement mechanisms, the function *void _ftime (***struct _timeb** ***timeptr)*** was used to measure the access time. *_ftime* does not return a value, but fills in the fields of the structure pointed to by pointer *timeptr* that points to **_timeb** structure. The *_ftime* function gets the current local time and stores it in the structure pointed to by *timeptr*. The **_timeb** structure is defined in SYS\TIMEB.H. It contains four fields: *dstflag* which is nonzero if daylight savings time is currently in effect for the local time zone; *millitm* which is fraction of a second in milliseconds; *time* which is in seconds since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC); *timezone* the value of which is set from the value of the global variable *_timezone* (See Example).

**Example**

```
/* FTIME.C: This program uses _ftime to obtain the current time and then stores this
time in timebuffer*/
#include <stdio.h>
#include <sty/timeb.h>
void main (void)
{
        struct_timeb timebuffer;
        char *timeline;
        _ftime (&timebuffer);
        timeline = ctime(& (timebuffer.time));
        printf (" The time is %.19s.%hu %s", timeline, timebuffer.millitm,
        &timeline[20]);

}
Output
The time is Tue Mar 21 15:26:41.341 1995
```

The *_ftime* routines use the TZ environment variable. If TZ is not set, the run time library attempts to use the time-zone information specified by the operation system. *_ftime* store current system time in variable of type **struct _timeb**.

### 3.2 Design and Implementation

I used the three measurement methods (Classic Hold, the Markov Hold, and Up/Down model) mentioned above to design and implement the performance of the three different priority queues. These priority queues are the Implicit Binary Heap [Bentley, 1985], the Median Pointer Linked List [McCormack and Sargent, 1981] and the SPEEDESQ [Steinman 1992]. The followings are the reasons why I choose these three priority queues:

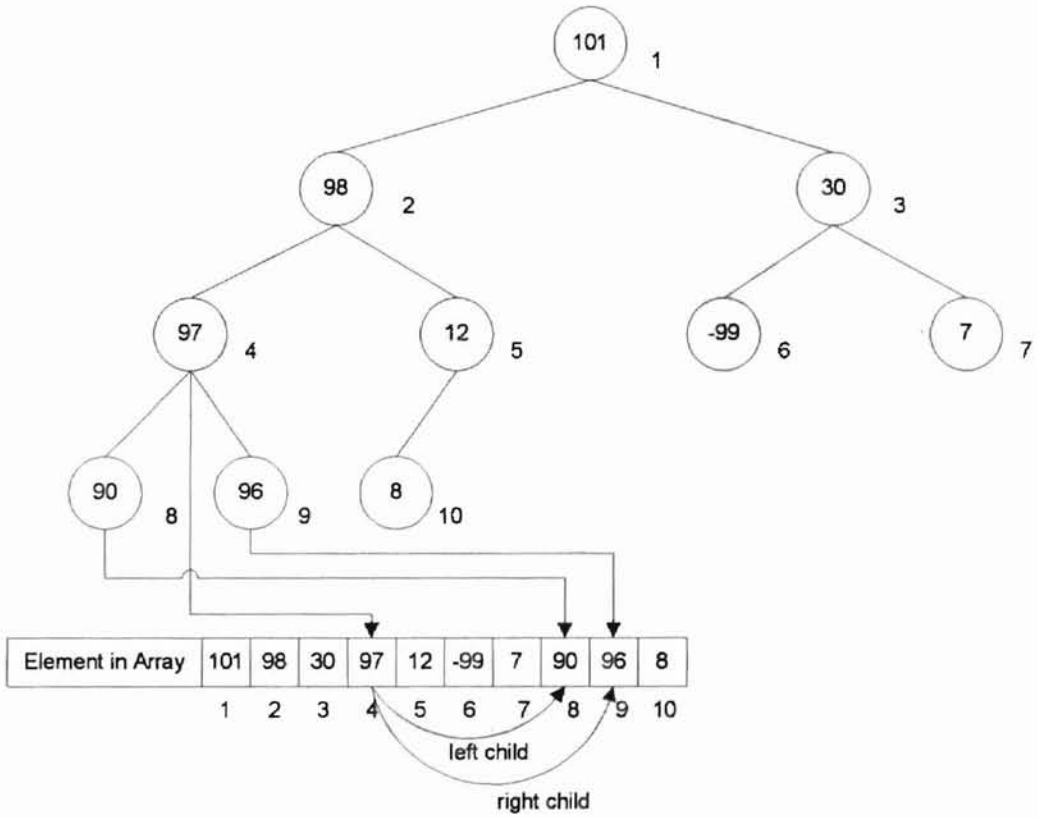
1. Implicit Binary Heap is a well-known priority queue. It stands for the array-based structures that are popular because of the potential for a logarithmic relationship between queue size (array size) and the complexity of insert and delete operations.
2. Median Pointer Linked List is a typical and classic ADT to represent priority queues. It stands for the linked-list-based structure with more sophisticated pointer-based algorithms.
3. SPEEDESQ is a two-linked-list priority queue data structure. It stands for the lazy-queue-oriented data structures that are more recently proposed queue implementations.

In this simulation program, all codes were written in the C programming language, since C programming language obtains the advantages of few restrictions, few complaints, block structures, stand-alone functions and a compact set of keywords

[Schildt, 1990]. Moreover, C programming language has the powerful portability and efficiency.

Four operations on the priority queues were implemented in the experiment: create-queue, dequeue, enqueue and free-queue. *_ftime* () function was used to measure the mean access time.

### 3.2.1 Implicit Binary Heap Simulation




Parent: i
Left Child: 2i
Right Child: 2i + 1

**Figure 5. Implicit Binary Heap and an Array Representation**

Since the Implicit Binary Heap data structure (heap) is a binary tree completely filled, it can be regarded as an array object. The Implicit Binary Heap thus consists of an array and an integer element representing the priority (see Figure 5).

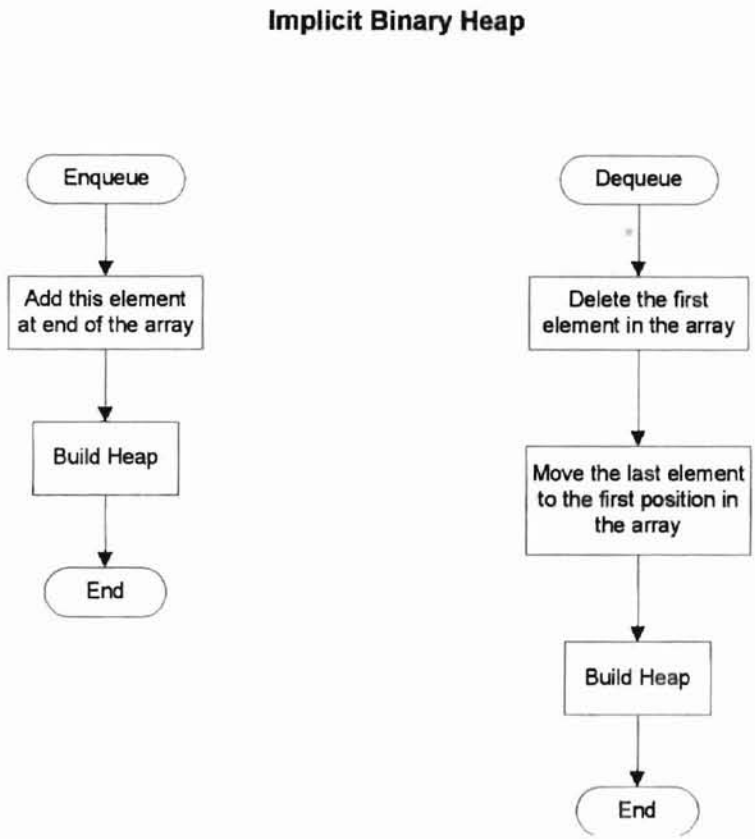



**Figure 6. Implicit Binary Heap Flow Chart**

In the heap simulation program (see figure 6), in order to avoid the transient startup period of the simulation, the initial size of the heap is set up to 16, which is built by *buildFixedSizeQu* () function. The element of the heap is generated by *RandomNum ()* function. The enqueue operation is performed by the *enqueueHeap* () function and the dequeue operation by *dequeueHeap* () function. Both operations involve ensuring that the heap order property is maintained by *Build_heap* () function. The heap size is increased

by twice for each time from 16 to 100,000. For each fixed size heap, each access pattern loops 5 million times.

### 3.2.2 Median Pointer Linked List Simulation

The Median Pointer Linked list (See Figure 7) is implemented as a doubly linked list. The structure is utilized in the design of each cell in the linked list.

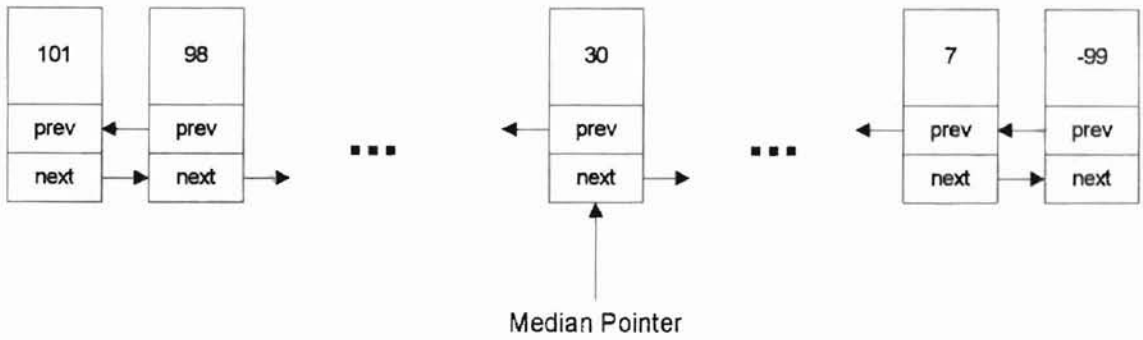



**Figure 7. Median Pointer Linked List Data Structure**

In the simulation program each cell is composed of an integer element and two pointers referring to the previous and next cell. The queue size of the Median Pointer Linked List is from 10 to 60,000 and for each queue size, the operations repeat 100,000 times to calculate the average. The list is built by the *buildFixedSizeMedlst* () function. Each element of the cell is generated by the *RandomNum* () function. A median pointer is set up to point to the middle position of the linked list. Enqueue and dequeue operations were performed by calling the functions *enquToMedLst* () and *dequFrmMedLst* () respectively. In the enqueue operation, the element of the new cell will compare with the value of the median pointer. If the element is larger than the median pointer node, the cell will be inserted from the front of the list, otherwise, the cell will be inserted from the back of the list. The median pointer is updated after finishing the two operations: when dequeue or equeue operation performs twice, the median

pointer moves backward or forward once, otherwise, the median pointer remains still. If the linked list is empty, median pointer points to the head of the list (See Figure 8).

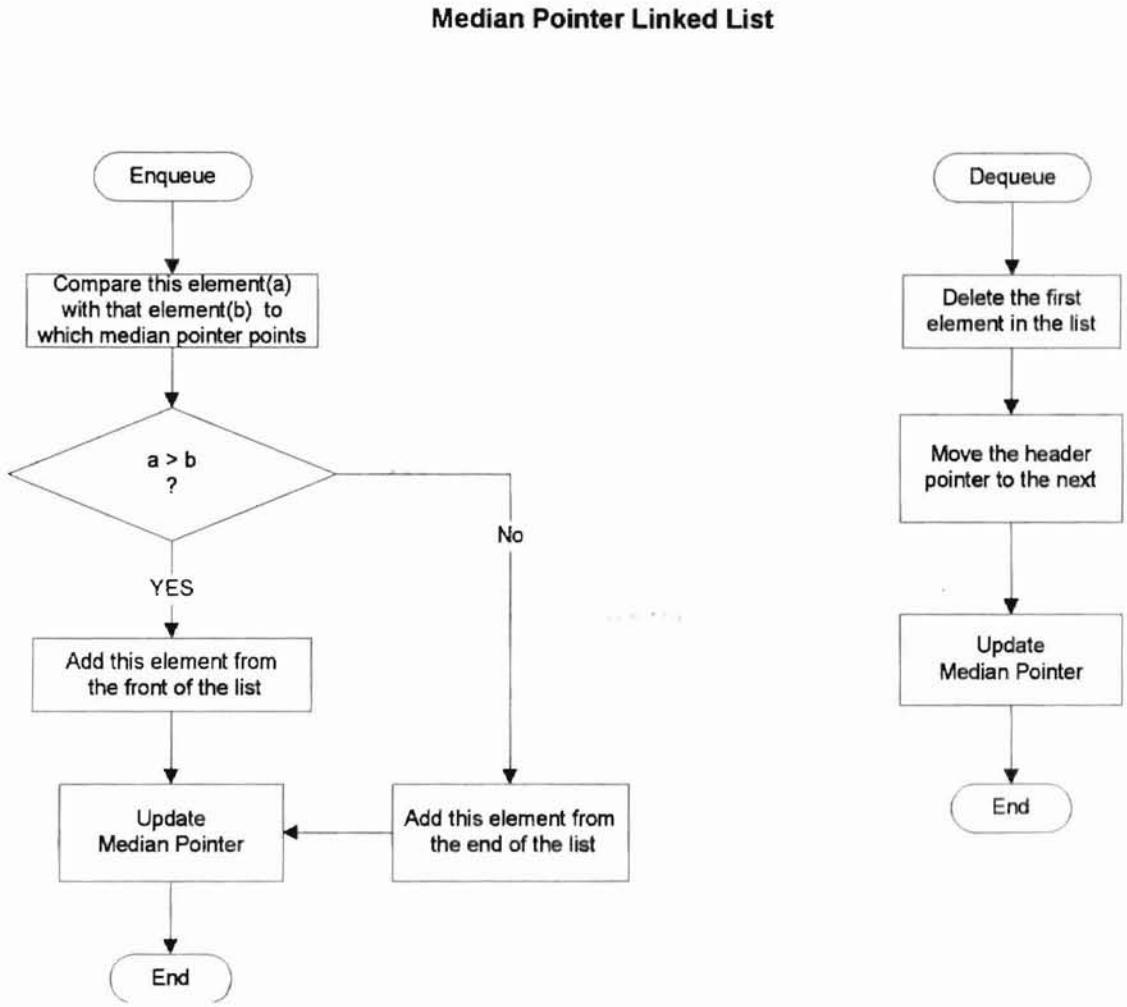



**Figure 8. Median Pointer Linked List Flow Chart**

### 3.2.3 SPEEDESQ Simulation

The SPEEDESQ (See Figure 9) consists of two single linear linked lists: one is sorted dequeue linked list and the other is unsorted enqueue linked list. The merge

operation will occur whenever the dequeue linked list is empty or the highest priority element is present in the enqueue linked list.

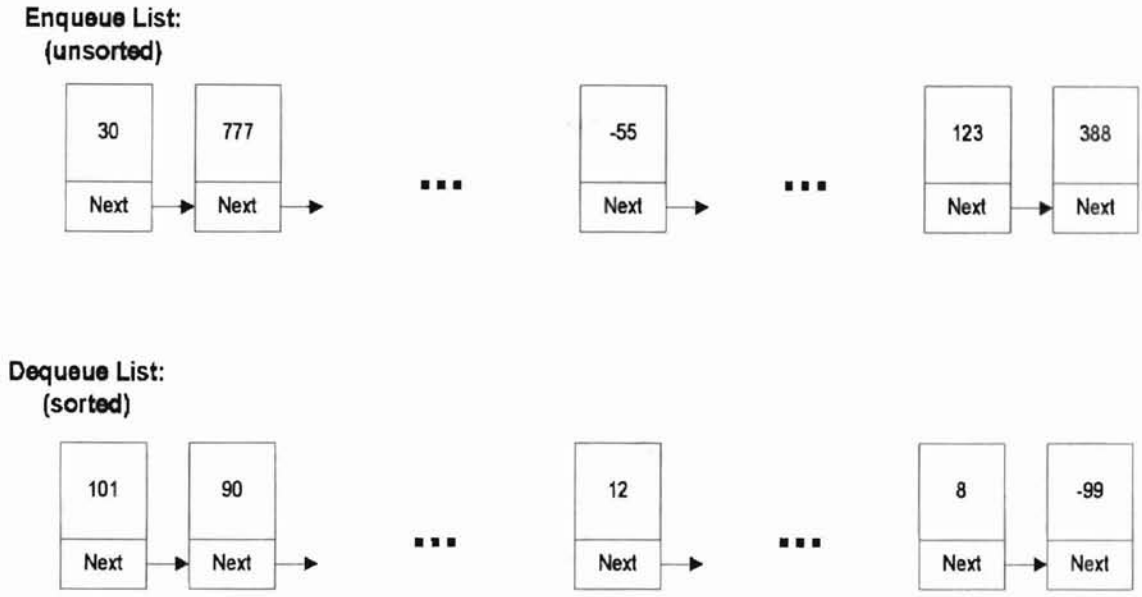



**Figure 9. SPEEDESQ Data Structure**

In the design of the SPEEDESQ simulation, the sizes of the list increased from 10 to 50,000 and for each queue size, the operations repeat 100,000 times to calculate the average. Two linked list data structures are set up: enqueue-linked list and dequeue-linked list. Two global variables are declared to record the highest priority of both the linked lists. The enqueue operation is performed by *enquToLst* ( ) function, in which the new element is directly inserted to the enqueue linked list without comparison and sorting. Each time the enqueue operation is performed, the highest priority variable is updated by comparing with the new element. The dequeue operation is performed by *dequFrmLst* () function that is much more complicated. In the dequeue operation, the head of the linked list needs to be compared with the highest priority variable. If the

element is larger dequeue the element from the list. If the element is smaller then sort the enqueue list and then merge the two linked lists (See Figure 10).

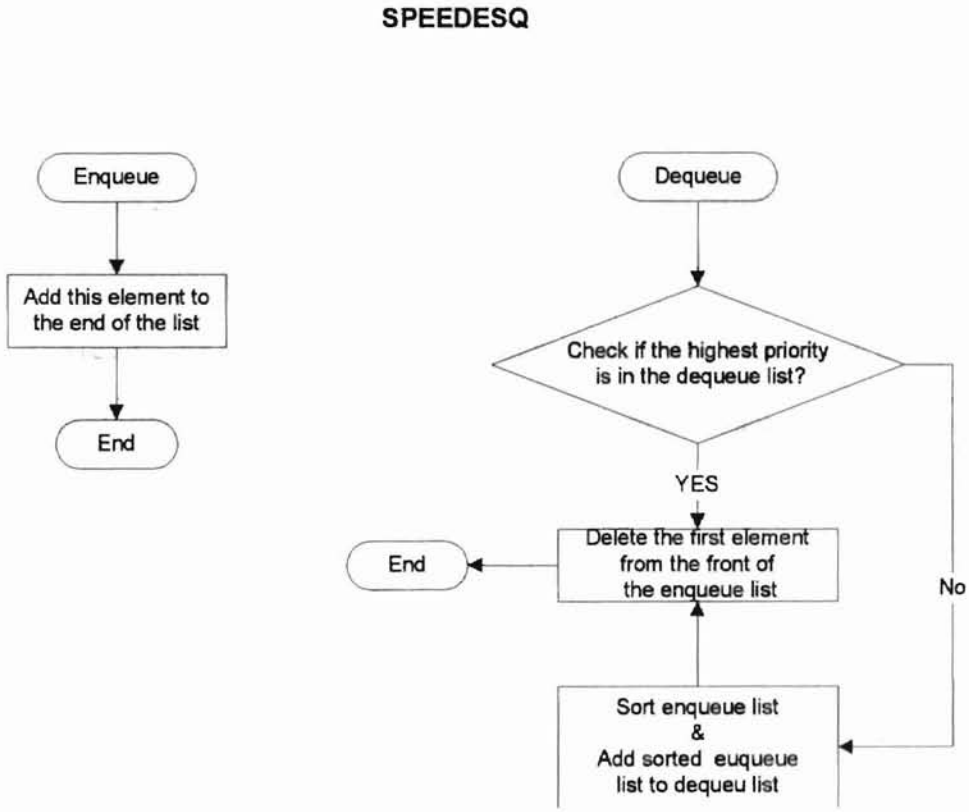



**Figure 10. SPEEDESQ Flow Chart**

# CHAPTER IV

# EVALUATION

## 4.1 Program

The program simulated the performances of the Implicit Binary Heap, the Median Pointer Linked List, and the SPEEDESQ. For each priority queue, the Classic Hold, Up/Down Model and Markov Hold were used as the access patterns. Figure 11 shows the flow chart of the simulation program.

The results are produced when the program terminates, which are used to comparatively study and analyze the priority queues and three access patterns as well.

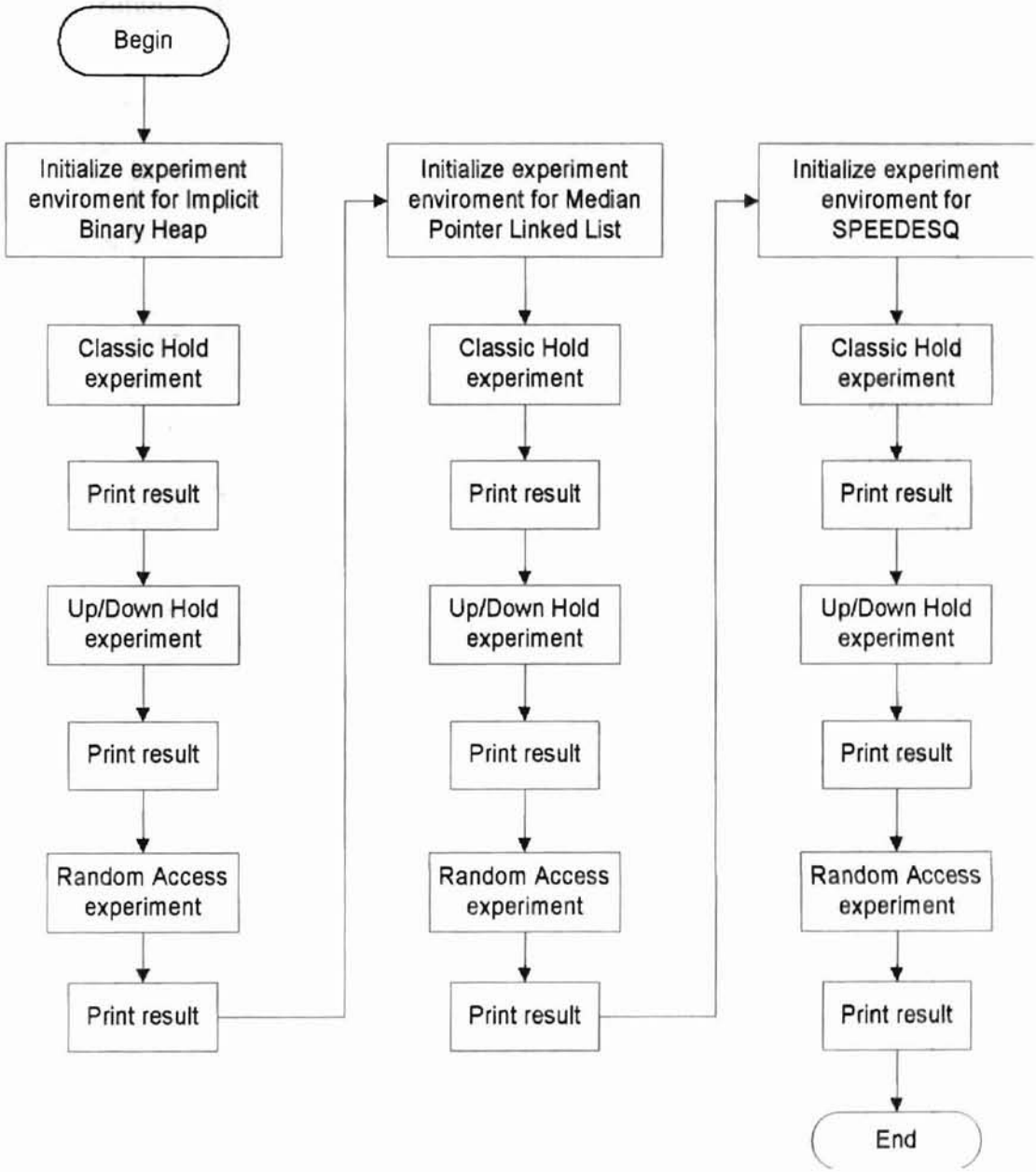



**Figure 11. The Simulation Program Flow Chart**

## 4.2 Performance of the Three Priority Queues

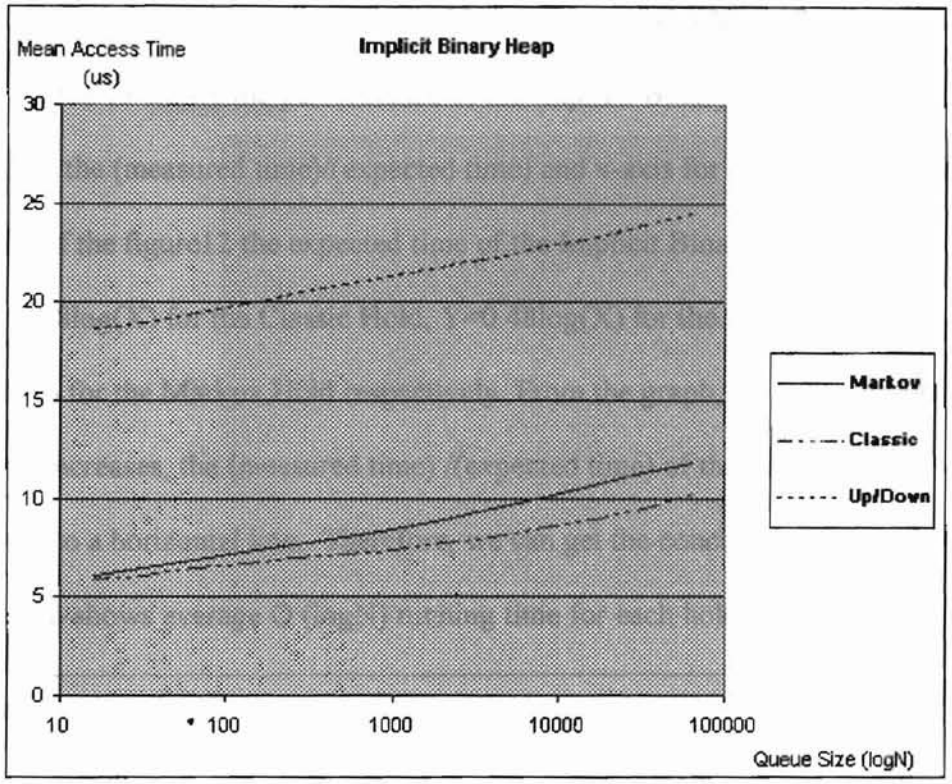



**Figure 12. Implicit Binary Heap with Classic Hold, Up/Down and Markov Hold**

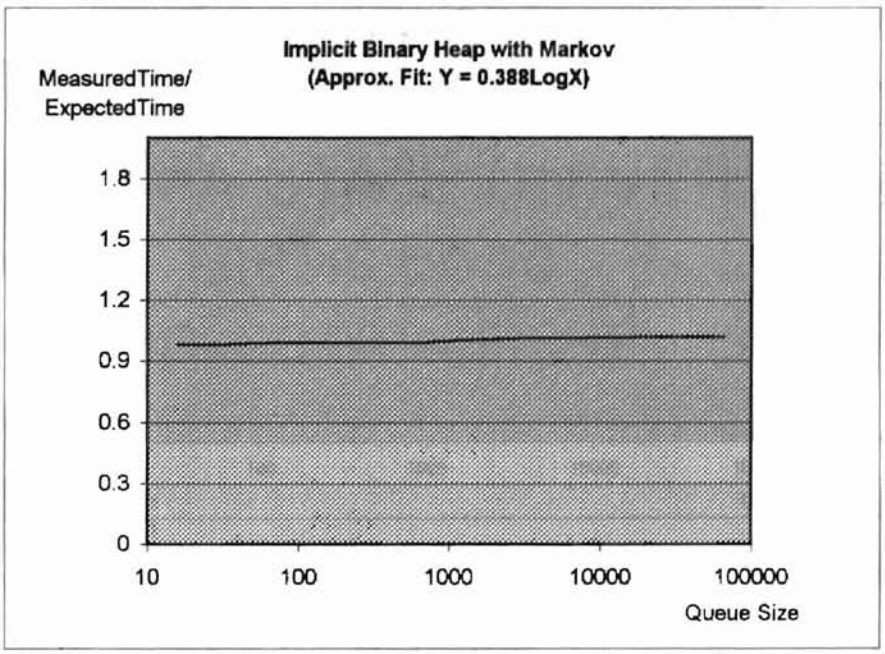



**Figure 13. Empirical Behavior at Large N of the Binary Heap with Markov Hold**

Figure 12 depicts the performance of the Implicit Binary Heap with the three models. As expected, the Implicit Binary Heap exhibits O(logN) performance.

Figures 13 shows the performance of the Implicit Binary Heap with y-axis standing for the (measured time)/(expected time) and x-axis for log (N). I derived from the results of the figure12 the expected time of the Implicit Binary Heap for each hold, i.e. Y= 0.298log(X) for the Classic Hold, Y=0.48log(X) for the Up/Down model and Y= 0.388log(X) for the Markov Hold respectively. From the graph, we find that when the queue size increases, the (measured time) /(expected time) of the Implicit Binary Heap approaches to a horizontal line. Therefore, we can get the conclusion that the Implicit Binary Heap shows average O (logN) running time for each hold.

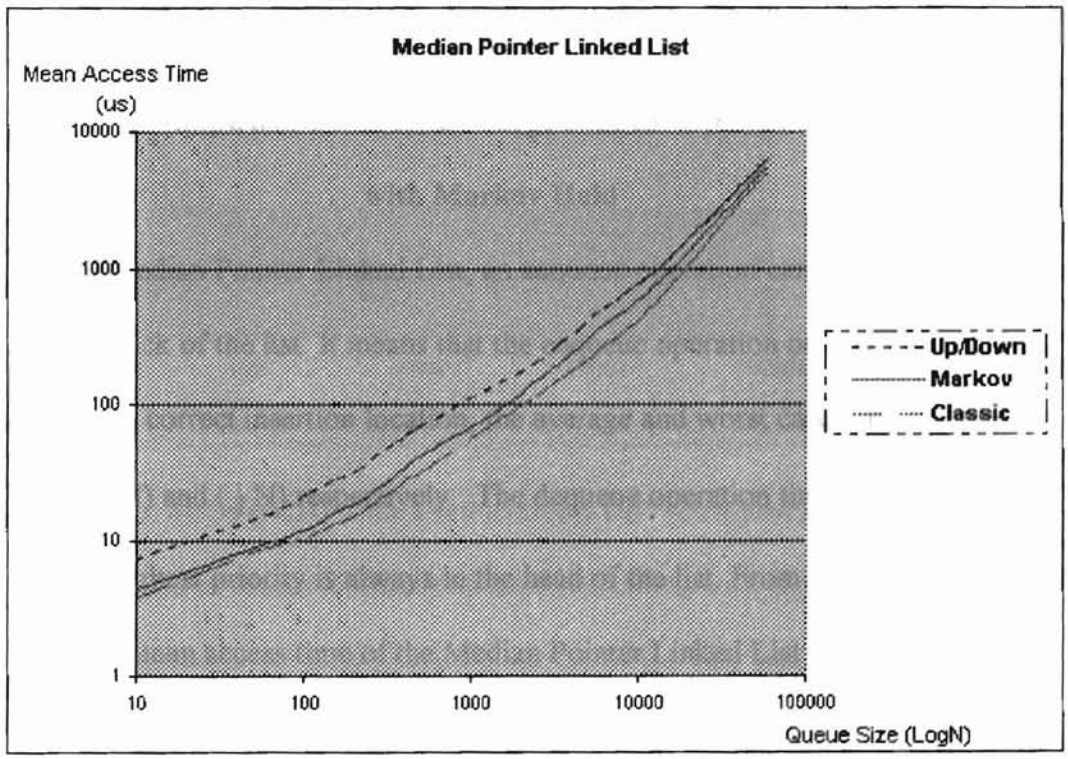



**Figure 14. The Median Pointer Linked List with Classic Hold, Up/Down and Markov Hold**

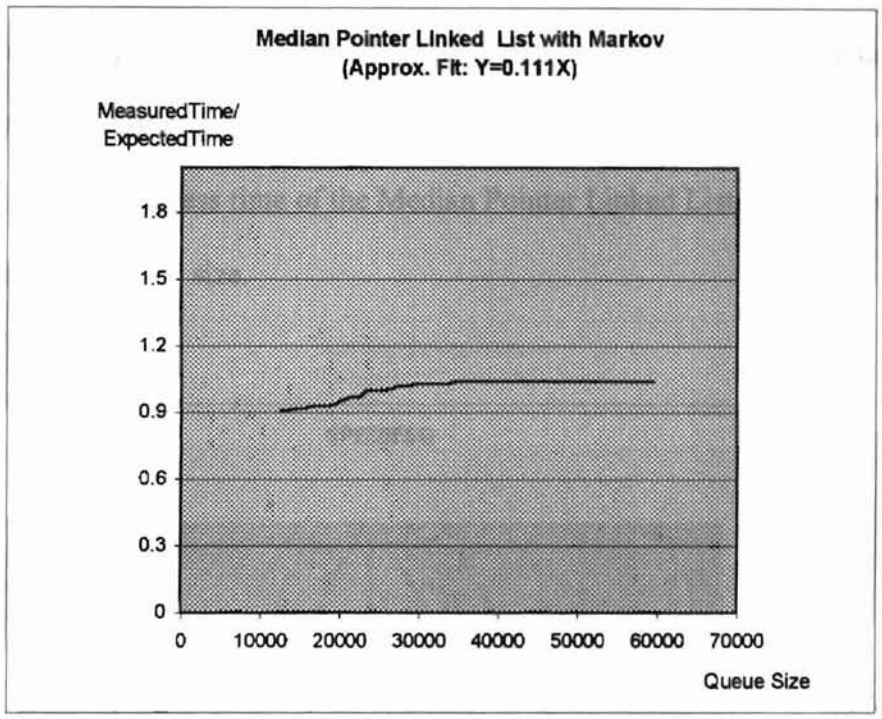



**Figure 15. Empirical Behavior at Large N of the Median Pointer Linked List with Markov Hold**

For the Median Pointer Linked List, an enqueue operation can be done either from the front or the back of the list. It means that the enqueue operation only searches half of the list to find the correct insertion location, the average and worst case time complexities are therefore ($\frac{1}{4}$ N) and ($\frac{1}{2}$ N) respectively. The dequeue operation time complexity is O(1), since the highest priority is always in the head of the list. From the figure 14 we can find that the mean access time of the Median Pointer Linked List grows linearly with the increase of elements in the queue. With the very small queue size (less than 30), the Median Pointer Linked List performs very well. From the result of the experiment, I got the expected time of the list for each hold. They are Y=0.116X for the Up/Down model, Y=0.101X for the Classic Hold and Y=0.111X for the Markov Hold. Figure 15 shows

the (measured time/expected time) performance of the Median Pointer Linked List with Markov Hold. The graph shows that when queue size (x-axis) increases a certain amount, the value of the (measured time)/ (expected time) approaches to a horizontal line. This proves that the mean access time of the Median Pointer Linked List grows linearly with the increase of the queue size.

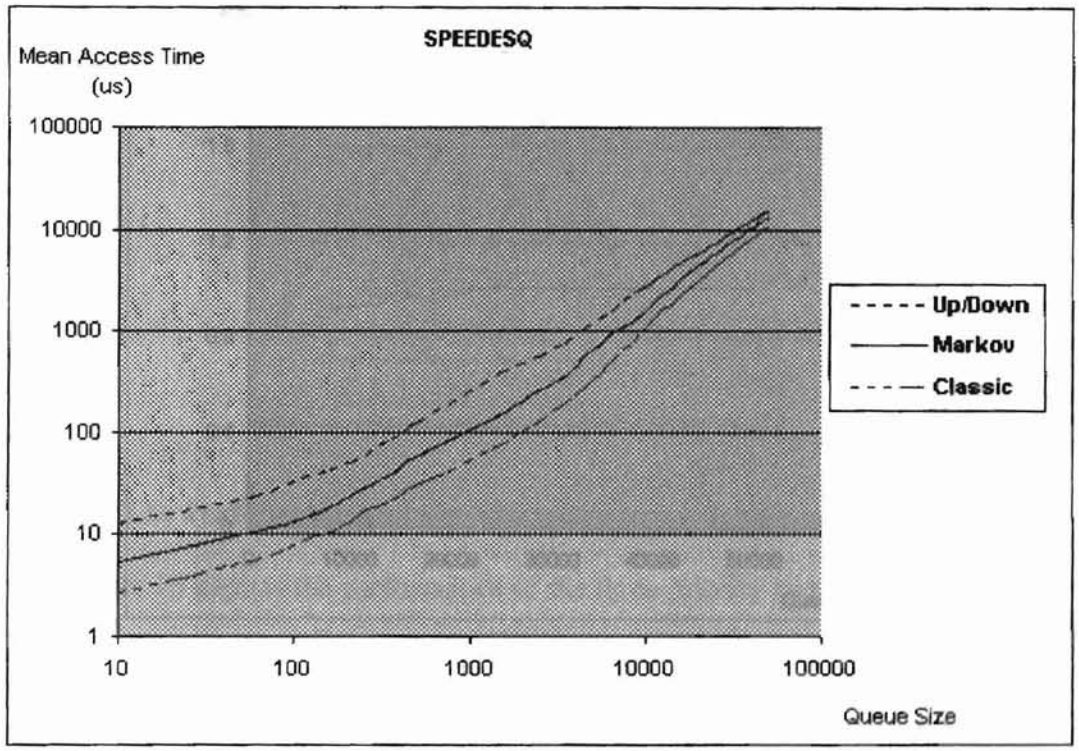



**Figure 16. SPEEDESQ with Classic Hold, Up/Down and Markov Hold**

Figure 16 shows the performance of the SPEEDESQ with three holds. Like the Median Pointer Linked List, the SPEEDESQ also performs very well for small queue size. The SPEEDESQ consists of two linked lists. One is an enqueue list that is unsorted, so the enqueue operation simply inserts an element at the end of the enqueue list, which costs O(1) time complexity. The other linked list is a dequeue list that is sorted. The dequeue operation depends on the location of the highest priority element. When the

highest priority element is present in the dequeue list, which is located in the head of the dequeue list, the dequeue operation just deletes the head costing O(1) running time, otherwise, it takes O(NlogN) worst-case time complexity to sort and merge the two linked lists.

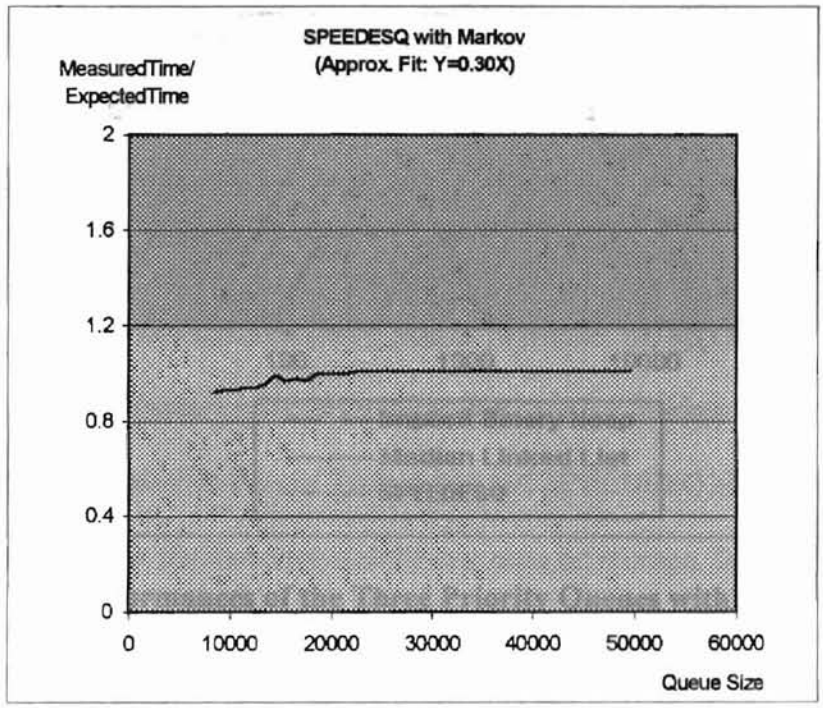



**Figure 17. Empirical Behavior at Large N of the SPEEDESQ with the Markov Hold**

The expected time of the three holds are Y=0.273X for Classic Hold, Y=0.324X for Up/Down and Y=0.30X for Markov Hold respectively. Figure 17 shows the (measured time) / (expected time) of the SPEEDESQ with Markov Hold. We find that the graph is a horizontal line. Therefore, we can prove that the performances of the SPEEDESQ grow linearly with the increase of queue size.

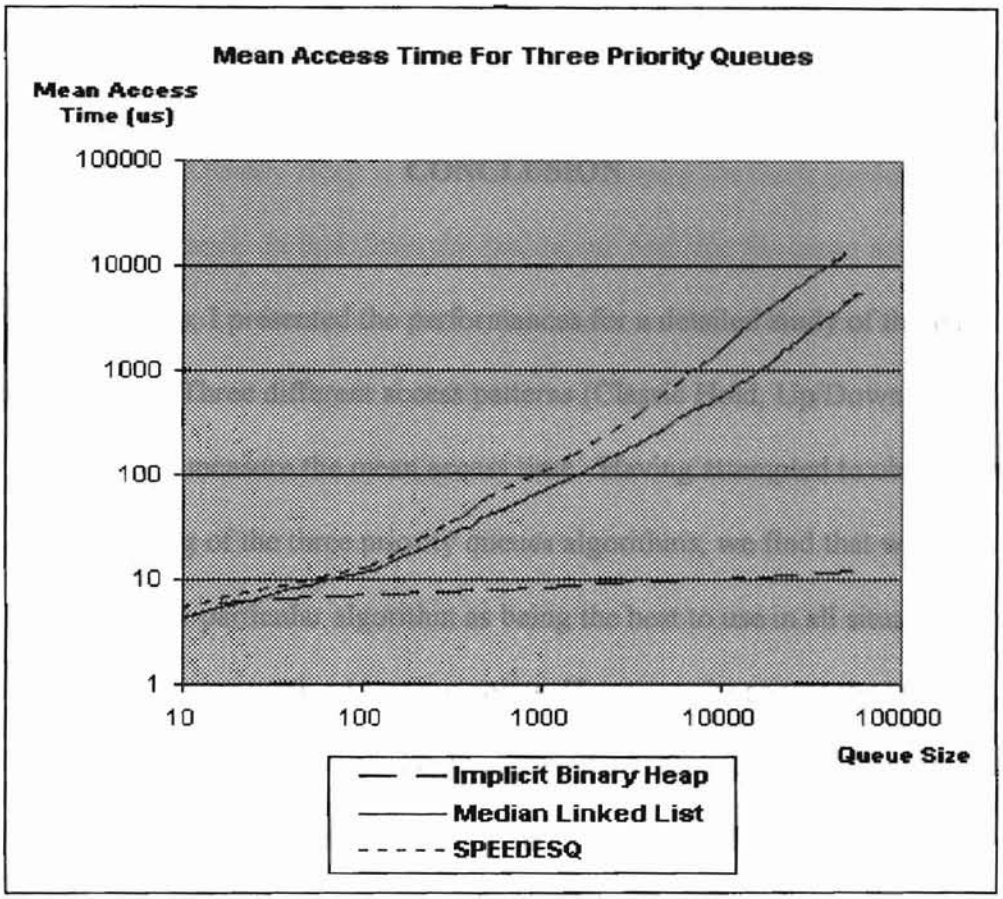



**Figure 18. The Performances of the Three Priority Queues with Markov Hold**

Figure 18 depicts the performance of the three priority queues with Markov Hold on one graph. We can clearly see the differences among the three priority queues. As expected, the Implicit Binary Heap exhibits O (log N) performance. We also noticed that both Median Pointer Linked List and SPEEDESQ perform very well when the queue size less than 30, however they can not compete with the Implicit Binary Heap when the queue size larger than 50.

# CHAPTER V

## CONCLUSION

In this thesis, I presented the performances for a detailed study of three priority queue algorithms. Three different access patterns (Classic Hold, Up/Down, Markov Hold) were used to measure the mean access time. Having attempted to obtain a comparative ranking of the three priority queues algorithms, we find that we are unable to recommend any one particular algorithm as being the best to use in all situations. All these queues have some weak and strong points. However, the approach to arriving at the conclusion has given readers some insight into methods for making the choice.

We note that Median Pointer Linked List and SPEEDESQ show good performance for queue sizes smaller than 30 elements on average. We also find that the mean access time of the SPEEDESQ is almost twice as much as the Median Pointer Linked List. This is because searching an element in the sorted linked list of the Median Pointer Linked List takes ($\frac{1}{4}$ N) on average whereas for the SPEEDESQ, it takes O ($\frac{1}{2}$ N) on average. The dequeue operation for the Median Pointer Linked List always takes O(1) time. The enqueue operation for SPEEDESQ always takes O (1) and dequeue operation also takes O (1) time if the merge is not necessary at this point. The Median Pointer Linked List can be used for the application that requests O (1) time complexity of the dequeue operation. The SPEEDESQ can be used for the application that requests the running time of the enqueue operation to be O (1) without concerning the worst case of

dequeue operation. SPEEDESQ is well suited for implementation on parallel machines because of its parallel sublists.

The Implicit Binary Heap is the only choice among the three queues that gives guaranteed performance. In fact, from the results, we find that the mean access time of Implicit Binary Heap performs so well that the other two linked lists can not compete with it when the queue size bigger than 50.

The results show that the standard Classic Hold yields results that correspond closely to the Markov Hold experiments. The Markov Hold model has been suggested as a generalization of the Classic Hold model. It allows random access patterns that could better mimic the behavior of real simulations. It has been claimed that this capability could reveal more information on the performance of priority queues. When comparing among the results obtained using the Classic Hold, Up/Down and Markov Hold, we draw the following conclusions. The Classic Hold and the Up/Down models represent two extreme cases. When the queue size remains nearly constant, the Classic Hold model gives as accurate and informative results as the more random access patterns generated by the Markov Hold Model. For changing queue sizes, the simple Up/Down access pattern often gives sufficient information. The simplicity of the Classic Hold and the Up/Down helps reveal more and clearer information on the dependencies of queue sizes on the performance.

## LITERATURE CITED


Aho, A.V., Hopcroft, J. E. and Ullman, J. D., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass, 1974;

Ayanne, R., "LR-algorithm: Concurrent operations on priority queues", Proceedings of the Second IEEE Symposium on Parallel and Distributed Systems, pp. 22-25, 1990;

Deldler, J., "Priority Queue", Data Structures and Algorithms, pp. 243-244, 1996;

Brown, R., "Calendar queues: A fast O (1) priority queue implementation for the simulation event set problem", Commun.ACM 31, 10 (Oct.), pp. 1220-1227, 1988;

Chung, K., Sang, J. and Rego, V., "A performance comparison of event calendar algorithms: An empirical approach", Softw. Pract. Exper. 23, 10 (Oct.), pp. 1107-1138, 1993;

Choi, B. D., "Priority queue with two-state Markov-modulated arrivals", IEEE Proceedings. Communications V. 145, No 3, 6 (June), pp. 152-159, 1998,

Floyd, R.W., "Algorithm 245, Tree sort 3", Comm. ACM, 7 (1964), pp. 701, 1964;

Fujimoto, R., "Parallel discrete event simulation", Commun.ACM 33, 10 (Oct.), pp. 31-53, 1990;

Gston, Gonnet, H. and Munro, J., "Heaps on Heaps", SIAM J. Comput. Vol. 15 No. 4, pp. 965-971, 1986;

Gonnet, G.H., A Handbook of Algorithms and Data Structures, Addison-Wesley, Reading, MA, 1984;

Henriksen, J. O., "An improved events list algorithms", Proceedings of the 1977 Winter Simulation Conference, pp. 547-557, 1988;

Horowitz, E., Sahui, S. and Sanguthevar, R., Computer Algorithm/C, 1996;

Knowlton and Kenneth, C., L6: Bell Telephone Laboratories Low-Level Linked List Language, 1966;

Knuth, D.E., "Sorting and Searching", The Art of Computer Programming,Vol.3., 1973;

McCormack, W. M. and Sargent, R. G., "Analysis of future event set algorithms for discrete event simulation", Commun.ACM 24, 12 (Dec.), pp. 801-812, 1981;

Munro, J.I. and Suwanda, H., "Implicit data structures for fast search and update", Comput. System Sci., 21 (1980), pp. 236-250, 1980;

Naor, D., Martel, C.U. and Matloff, N.S., "Performance of Priority Queue Structures in a Virtual Memory Environment", The Computer Journal, Vol. 34, No.5, pp. 1428-437, 1991;

Pugh, W., "Skip lists: A probabilistic alternative to balanced trees", Commun. ACM 33, 6 (June), pp. 668-676, 1990;

Ronngren, R., Ayani, R., Fujimoto, R. M. and Das, S. R., "Efficient implementation of event sets in time warp", In Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS'93), pp. 101-108, 1993;

Steinman, J. S., "SPEEDES: A unified approach to parallel simulation", In Proceedings of the Sixth Workshop on Parallel and Distributed Simulation, pp. 75-84, 1982;

Weiss, M. A., Data Structures and Algorithm Analysis in C, 1993;

Williams, J.W.J., "Algorithm 23: Heapsort", Comm. ACM, 7(1964), pp. 347-348, 1964.

# VITA

DONGHONG WEI

Candidate for the Degree of

Master of Science

Dissertation: COMPARATIVE STUDY OF PRIORITY QUEUES IMPLEMENTATION

Major field: Computer Science

Biographical:

Personal Data: Born in Tangshan, Hebei, P.R.China, July 26, 1967, the daughter of Shuren Wei and Shuzhen Ma.

Education: Graduated from 30th High School of Taiyuan, Taiyuan, Shanxi, China, July 1985; received Bachelor of Arts degree in Foreign Languages from Shanxi University, Taiyuan, Shanxi, China, July 1990; completed the requirements for the Master of Science degree at Oklahoma State University in December, 1999.