Singly Linked List and Binary Heap Based Priority Queues

Albert Asratyan & Mandar Joshi
asratyan@kth.se mandarj@kth.se

Royal Institute of Technology

Stockholm, Sweden 2018

Abstract

A priority queue is one of the most fundamental data structures that are used by software programmers on a daily basis. In this study priority queues based on singly linked list and binary heap had been investigated and compared. To study this, multiple priority queues of each of the implementations of sizes ranging from 100000 elements to 10000000 elements had been built and tested. Two main functions, insert(), which inserts an element into the queue and deleteMax(), which removes the element with the highest priority, have been tested, and the execution times in clock cycles have been recorded. Singly linked list version has shown O(N) time complexity for insertion on average, whereas binary heap version has shown O(log(N)) time complexity. Regarding deletions, linked list has shown to be constant O(1), whereas binary heap is logarithmic again with O(log(N)). Therefore, each of the implementations should be used, based on the importance of each of the operations. If deletion is crucial, use linked list version, but otherwise binary heap implementation suits general needs better.

Sammanfattning

En prioritetskö är en av de mest grundläggande datastrukturer som dagligen används av programmerare. I denna studie har prioritetsköer undersökts och jämförts med både en enbart länkad lista och en binär hög. För att studera detta, har flera prioritetsköer för varje implementering av storlekar från 100000 element till 10000000 element byggts och testats. Två huvudfunktioner, insert(), som sätter in ett element i kön och deleteMax(), som tar bort elementet med högsta prioritet, har testats och vi har mätt exekveringstiderna i klockcykler. Enbart länkad list versionen har visat O(N) tidskomplexitet för “insert” i genomsnitt, medan den binära högen har visat O(log (N)) tidskomplexitet. När det gäller “deleteMax” har den enbart länkad lista visat sig vara konstant O(1), medan den binära högen är logaritmisk igen med O (log (N)). Därför kan man använda båda implementeringar, baserat på vikten av var och en av operationerna. Om borttagning av element är avgörande, bör man använda enbart länkad lista, men annars är den binära högen bättre vid allmänna behov.

Table of Contents

1. Introduction

1.1 Background

When writing code, programmers use data structures subconsciously most of the time, never realizing how crucial they are for the code. Data structures can be defined as a management and storage format that allows to access and modify data efficiently [1]. However, because of its definition, data structures include many different forms of storing data, ranging from the basic primitive types, such as booleans and integers, to more complex trees and heaps.

In this report, we are going to narrow down data structures to priority queues only and analyze advantages and disadvantages of the different ways there exist of implementing priority queues.

1.2 Problem

A priority queue is a data structure that allows at least the following two operations to be performed: inserting data to the queue in respect to its priority, and deletion, which finds, returns, and removes the minimum element in the priority queue. In other words, we remove the element with the highest priority.

Since a priority queue can be implemented with different algorithms using different data structures, the question arises of the differences between various implementations, will some be more efficient than others?

In this report we will be comparing two implementations of priority queues, one using singly linked lists and another using a binary heap.

Thus, the research question can be formulated as following: which of the priority queue implementations is better, the singly linked list or heap based? The evaluation will include best, worst and normal case performance for insert() and deleteMax() operations. Insert() inserts an element into the queue with an associated priority. deleteMax() removes an element with highest priority from the queue.

1.3 Objectives

The purpose of this study is to investigate the hypothesis stated above. Most likely, each of the implementations will have its advantages, and, therefore, this evaluation will be able to serve as a basis for choosing the most fitting priority queue implementation in general cases.

The study will compare the best, worst, and average time complexity cases for insert() and deleteMax() operations for each of the priority queue implementations. The study will then serve as a guideline for choosing priority queue implementations for different situations.

1.4 Social benefits, ethics and sustainable development

In this investigation we are going to replicate standard data structures that are used widely in computer science and software engineering. Data structures can be considered a basic “building material” for writing code which can be used for building any sort of software. Due to this nature of data structures there are a wide range of ethical consequences they can be applied to.

Regarding sustainability, programming languages evolve over time. However, data structures are theoretical concepts that have a specific implementation, not dependent on any programming language. In this report we will be discussing data structures with an implementation in the programming language C, it might be the case that the specific code we write will not run as the language changes. Nonetheless, the conclusions will be related to data structures which will not be affected by changes in the programing language, which is beneficial for sustainability.

1.5 Delimitations

Testing did not involve priority queues with less than 1000 elements because they are too small to compare and see a substantial difference in time complexity. For queues with a small number of elements it is then up to the users preference as to what data structure should be used for implementation.

2. Method

When comparing different priority queue implementations, time complexity is usually used. Time complexity is the computational complexity that describes the amount of time it takes to run an algorithm to its completion. Each of the functions (insert() and deleteMax()) will have its own time complexity. Because of this, multiple tests are required to compare every aspect of each of the implementations. Tests will be conducted for each of the functions, required by the definition of priority queue. Clock ticks from the <time.h> library are going to be used for time complexity measurements.

In order to compare time complexities, some requirements must be met. The tests must be run on the same machine, because of the hardware differences that may affect the final results. Then the results are going to be compared between each other for later analysis and conclusion. However, since clock ticks are system specific, it will not be possible to replicate the exact results as in this investigation, but the general differences between the implementations will still be the same, with the only difference being measured times of a slightly different scale, but moreless the same proportion.

2.1 Data collection

In order to collect the most unbiased data, simulation conditions should match for all of the test cases. Multiple runs for each of the tests should be recorded, to reduce any errors. Because of this, for each of the priority queues an input sample of test data is going to consist of a 500 numbers. Each of the test for each of the functions (insert() and deleteMax()) is going to be repeated multiple times. The same machine is going to be used for conducting all of the experiments. The tests will also account for the best, worst and average cases. This is explained in more detail below.

2.2 Data analysis

After the raw data is collected, it must be analyzed. First, the averages are going to be calculated for each of the tests with standard deviation to understand how spread the results are from the calculated mean. These averages can then be used for determining which of the implementation is better in what cases. These will also be used to plot confidence intervals in the graphs for the data we will present.

3. System

In this study we have designed a priority queue implemented with both a linked list and a binary heap.

3.1 System development method

We have used an iterative development model, where we have broken down the implementations into smaller once. Specifically, we started with implementing a linked list, testing it and making sure it is valid, and then going on and implementing a priority queue based on that linked list. The same principle applied to the binary heap and the priority queue based on that. On each iteration we have done testing, and if we found anything that did not work as intended then we had to backtrack and re-implement our solution.

3.2 Implemented algorithms

3.2.1 Priority Queue

A queue is a abstract data structure where elements are inserted and deleted according to the first-in first-out principle, or FIFO. In a normal queue the typical operations you can perform for inserting and deleting elements are enqueue and dequeue, these insert an element to the back of the queue respectively remove an element from the front of the queue. A priority queue is similar to a queue, in which elements are inserted with an added factor, a priority. These elements are then removed from the queue based on the highest priority. These elements are therefore stored in the queue based on their priority [2] [3].

3.2.2 Linked list

A linked list is a linear data structure where elements are separated objects. Each element contains two pieces of data, the data itself and a reference to the next element, also called node. The last node has a reference to null. A linked list is therefore a dynamic data structure, it can be enlarged or shrunken at any given moment. The disadvantage of a linked list is that one can not access an individual element directly, instead you would be required to traverse the list from the head (first node) until the node you are searching for. This is a description for a singly linked list, and a doubly linked list is a variation of this where each node has two references instead of one. These point to the next node and the previous node. Yet another type of linked list is a circular linked list. In this the last node of the list points back to the first node of the list [4].

3.2.3 Priority queue with linked list

A priority queue with a linked list is created in the following way. The highest priority element is always at the head of the list. The other elements are arranged in descending order based on their priority. The benefit of this is that it allows us to remove an element very quickly since it is always at the head of the list. This is done with a time complexity of O(1), or constant time. To insert an element, we are required to traverse the list to find the exact position where the element should be inserted depending on its priority. This way the order of the list is maintained. However, the time complexity will be O(N), where N is the number of elements in the list [5].

3.2.4 Binary Heap

We start by defining a binary tree. A binary tree is based on the mathematical concept from graph theory that has the following definition: "A binary tree comprises a triple of sets (L, S, R) where L and R are binary trees (or are empty) and S is a singleton set. The single element of S is the root, and L and R are called, respectively, the left and right subtrees of the root." [6] A binary heap then is a data structure which is a heap that takes the form of a binary tree. Every element at every node is less than (or equal to) the element at its left child and the element at its right child. The heap will then have the property of finding the minimum element at the root of the tree [7].

3.2.5 Priority queue with binary heap

Binary heaps are a very common data structure used to implement binary heaps. In order to implement the binary heap, the standard approach is to use an array, which we have done in our implementation. Each element in the array corresponds to a node in the heap. Specifically, this can be illustrated in the following example. These are the properties:

1. the root of the heap is always in array[1]
2. the left child is found in array[2]
3. the right child is found in array[3]
4. in general, if a node is in array[k], then the left child is in array[k*2], and the right child is in array[k*2 + 1]
5. If a node is in array[k], then its parent is in array[k/2] (using integer division)

Therefore, when inserting an element, we first add it to the end of the array. However, this might lead to the heap not being in the proper order because of the priority. Then we have to compare the value of priorities to its parent, if the parent is smaller we swap the values. Then this check is done recursively until we find nodes where the priorities keep the order, or if we get to the root. Regarding the complexity, for the insert operation, we start by adding an element to the end of the array. This is constant, if we assume that the array does not need to be expanded. We then swap values up the tree until the heap is ordered. In the worst case, we have to get to the top of the tree, and then the insert operation is O(log(N)). This is because the tree is balanced. Removing the maximum element is similar in time complexity. We then have to follow the path down the tree from the top to the bottom.

The time complexity for deletion is then also O(log(N)), where N is the number of values stored in the heap [8].

3.3 Test Bed

This study requires multiple tests to be performed for performance comparison. For this purpose, primarily a Windows laptop with a virtualized Linux (Ubuntu 18) was used with the Virtualbox software. The virtual machine had 2 GB of RAM, and 1 core (2 thread) of Intel Core i5-8250U 1.8 GHz processor dedicated to it. To eliminate extra uncertainties, the laptop was always plugged in and in the "performance mode" to stop processor from locking the core frequencies. To validate the results, some of the tests were replicated on a Macbook Air running macos mojave 10.14 with a 1.4 GHz Intel Core i5 processor. It had 4GB of 1600MHz DDR3 RAM, and used the gcc compiler to see if the general trends with the tests match.

For coding, Atom text editor and GCC compiler were used. No additional GCC optimizations were used. All of the tests were conducted consecutively to eliminate possible inconsistencies based on running background system processes. Before running all of the tests, the test bed was validated. All of the used software (Linux OS, gcc compiler) was using latest stable versions.

3.3.1 Measurement methods

For measurements the <time.h> library was used, with the clock function called in the beginning of the code and the end of the code for the code segment we are executing to measure it. We then subtract these values to get the amount of clock ticks it took for the processor to execute the lines of code we timed.

3.4 Experiments

To test the performance of the singly linked list based priority queue and binary heap based priority queue, the main functions, insert() and deleteMax(), were tested.

Due to the nature of singly linked list based priority queues, the delete() function does not have best and worst cases, because there is only one case where delete() always removes the first element of the queue. The same goes for the binary heap based priority queue.

Regarding the insert() function, three cases have to be taken into consideration: best, worst, and average. For linked list based priority queue, the best case is when an element is entered at the beginning of the list, so there would be no list traversal through the list. The worst case would be to enter an element at the end of the list, because then there would be a need to go through every single element of the list, to get to the end of the list. For the average case, a random number generator from the standard libraries was used to simulate average performance.

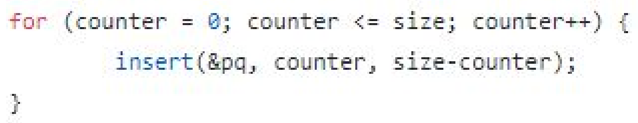
For binary heap based priority queue, the cases are opposite. The worst case would happen when an element has to be inserted at the beginning of the tree, because then the whole tree has to be reshaped based on the new highest priority element. In the best case, the element is inserted at the end of the heap, so the heap does not have to be reshaped. The average case is simulated in the same way as the linked list average case, random priorities were generated using the library function rand().

Both these implementations use the rand() function from the C, and this function returns a pseudo-random number between 0 and RAND_MAX, which in the library is a the value 2147483647, the highest signed integer that can be represented with 32 bits [9]. A pseudo-random number generator or PNRG is an algorithm that use formulas from mathematics or lists that are already pre-calculated in order to produce random numbers. However, it results that these numbers are not truly random, they are predetermined. In particular, the rand() function in C uses a Linear congruential generator, or LCG, as the mathematical algorithm to obtain its random values [10].

There are some other random functions one could have used, for example, another study regarding a similar problem statement has used the following distributions: uniform distribution, also known as the rectangular distribution, triangular distribution, negative triangular distribution and the exponential distribution [11].

The tests were performed for priority queues of the following sizes: 1 000 000 elements, 2 000 000 elements, 3 000 000 elements, 4 000 000 elements, 5 000 000 elements, 6 000 000 elements, 7 000 000 elements, 8 000 000 elements, 9 000 000 elements, 10 000 000 elements. Queues of smaller sizes (1 000 elements, 10 000 elements, 100 000 elements, 300 000 elements, 500 000 elements) have also been tested, and the raw data can be found in the appendix 4.

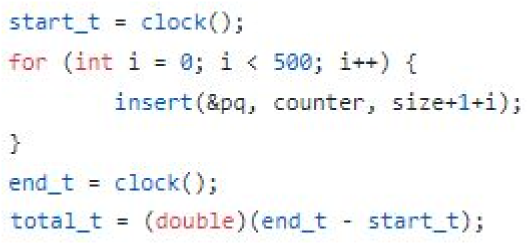
For each of the priority queues listed above, 500 insertions (or deletions, depending on the test) were performed. For each of the tests, a priority queue of a required size must be created first. To achieve this efficiently, best case types of queue building up insertions have been performed. For example, when creating a linked list efficiently and fast:



```
for (counter = 0; counter <= size; counter++) {
        insert(&pq, counter, size-counter);
}
```

Picture 1. Creating a singly linked list priority queue fast

In the code snippet above, for loop inserts a node in the beginning of the list by calling insert(priorityQueue, value, priority) with priority higher, than the previous highest priority, indicated by size - counter. This is necessary for building up priority queues of big sizes, which will become apparent in the discussion later where the results for linked list priority queue running times will be analyzed.
Insertions were timed in the following manner, this was described in the section 3.3.1:



```
start_t = clock();
for (int i = 0; i < 500; i++) {
        insert(&pq, counter, size+1+i);
}
end_t = clock();
total_t = (double)(end_t - start_t);
```

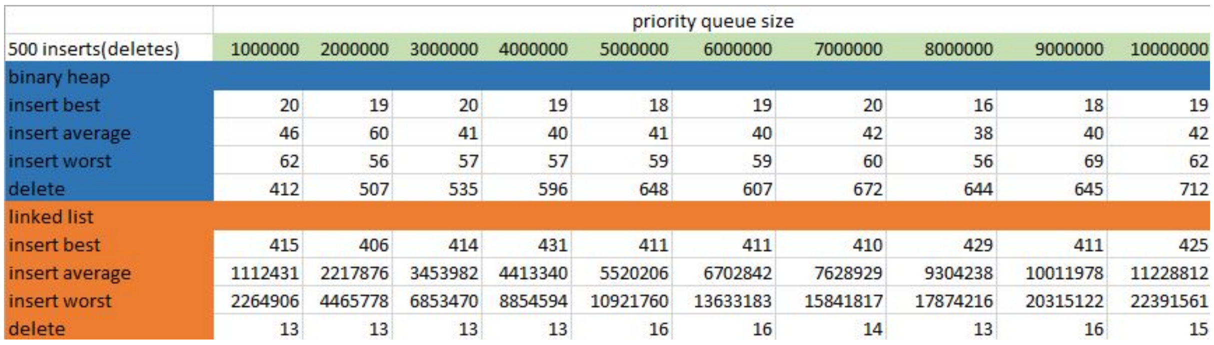
Picture 2. Timing insertions

Starting clock measurement was taken, 500 insertions were completed, and the final clock measurement was recorded. The difference between the final and the initial equal to the amount of the processor clocks taken to complete 500 insertions. 500 may seem like a very specific number, but it was chosen because each individual insertion/deletion is too quick to measure, and there is no measurable time difference between 1 insertion and 10 insertions. That is why they were measured in batches. Moreover, we only need to know the relative measurements to be able to compare two different priority queues. This was described in section 3.3 where we discussed our operating system and processor. The values acquired from these tests are for this processor and operating system, the results could differ with different processors and/or operating system. As long as conditions of the tests remain the same for all of the tests, it satisfies the requirements that we set out.

As a side note, 500 tested together one after another one measurements do not affect overall performance of the implementations, because of the initial size of the queues. Even for the smallest of the sizes, 1000000 elements, 500 is only 500/1000000=0,0005 (or 0,05%) of the whole amount. Therefore, the difference between making 1000000 or 1000499 insertions can be neglected.

4 Results

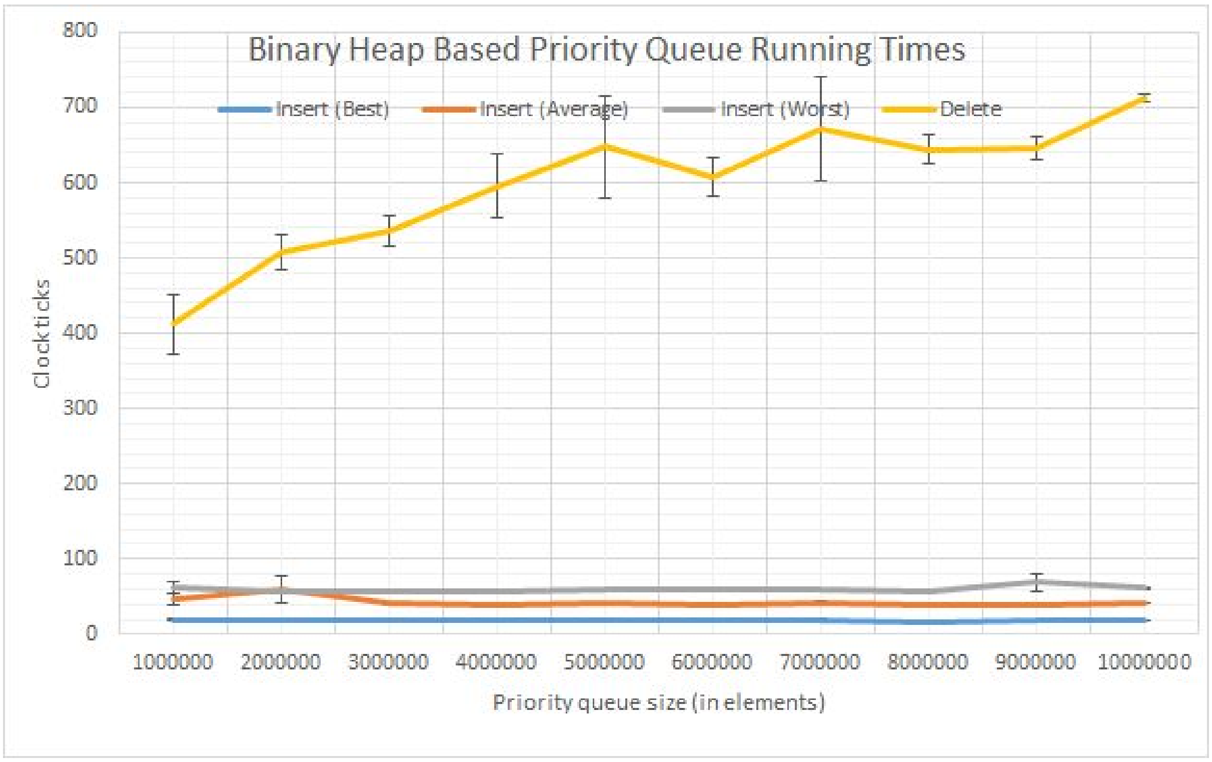
Based on the tests conducted above, the following graph contains the mean values of the operations we tested. The full table is available in Appendix 4. The full table has values for each of the tries, mean values, and 95% confidence intervals for each of the found means. The table below is an extract from the full table and it contains average clock cycles for operations with elements of the two implementations of priority queues with varying sizes.



	priority queue size									
500 inserts(deletes)	1000000	2000000	3000000	4000000	5000000	6000000	7000000	8000000	9000000	10000000
binary heap										
insert best	20	19	20	19	18	19	20	16	18	19
insert average	46	60	41	40	41	40	42	38	40	42
insert worst	62	56	57	57	59	59	60	56	69	62
delete	412	507	535	596	648	607	672	644	645	712
linked list										
insert best	415	406	414	431	411	411	410	429	411	425
insert average	1112431	2217876	3453982	4413340	5520206	6702842	7628929	9304238	10011978	11228812
insert worst	2264906	4465778	6853470	8854594	10921760	13633183	15841817	17874216	20315122	22391561
delete	13	13	13	13	16	16	14	13	16	15

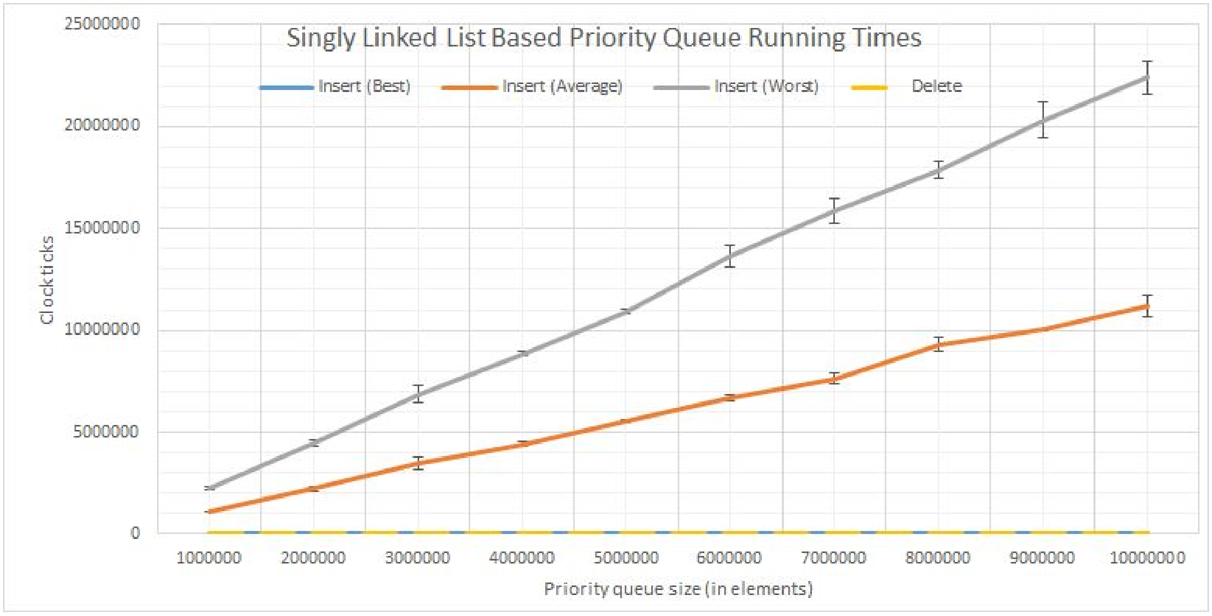
Table 1. Mean values for each of the 500 insertions(deletions) in processor clock ticks

Calculated with confidence intervals, the following graph shows the performance of the binary heap implementation.



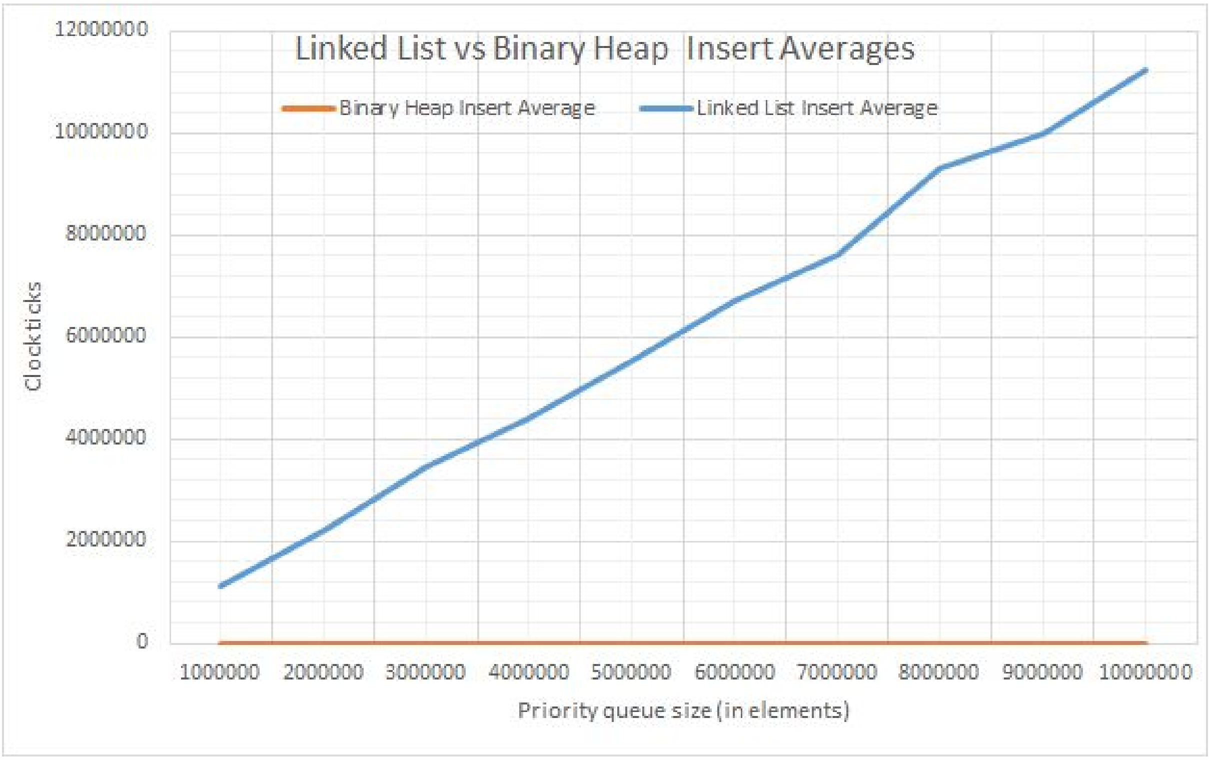
Graph 1. Binary heap based priority queue performance

Similarly, the following graph shows the singly linked list performance with confidence intervals.



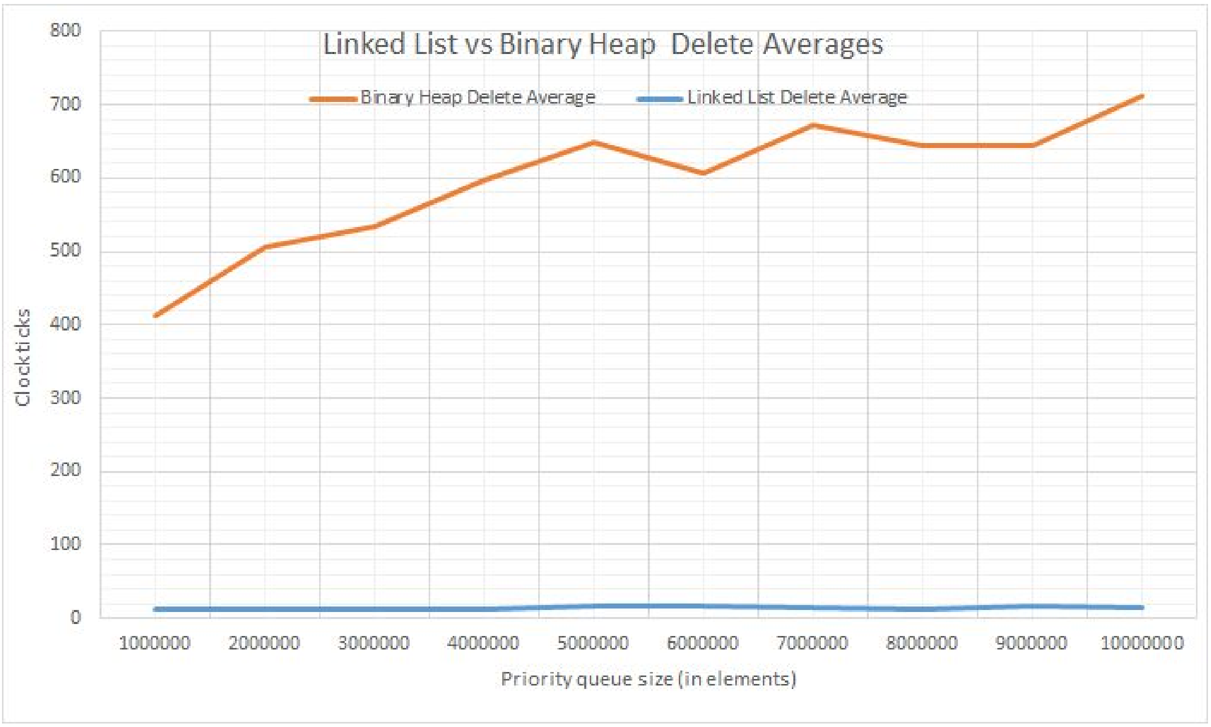
Graph 2. Singly linked list based priority queue performance

When compared to each other and presented on the same graph, average insert running times of singly linked list and binary heap versions look like the following:



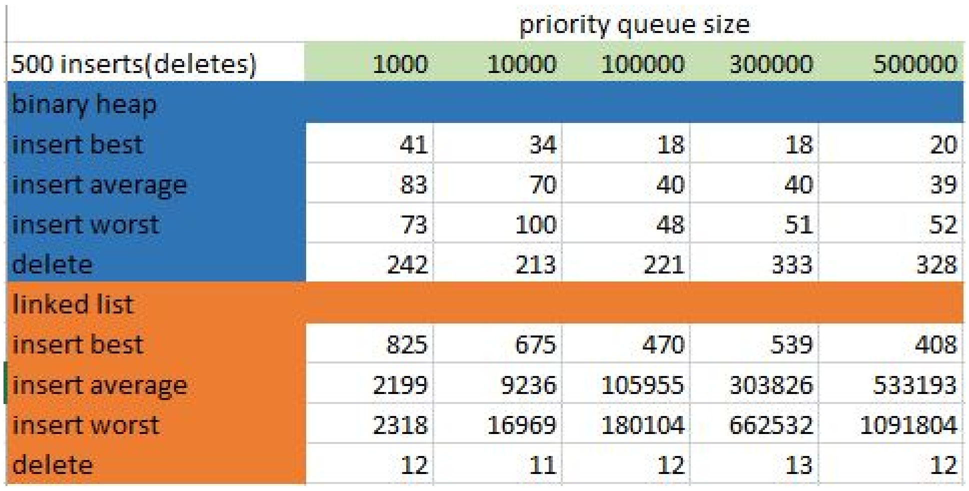
Graph 3. Comparison of insert average times for linked list and binary heap

Finally, the delete() function running time comparison:



Graph 4. Comparison of delete() running times for linked list and binary priority queues

It must be noted, that on some of the lines on the first two graphs, error bars appear invisible, because the values for confidence interval are very small, when compared to the scale of the graph. Refer to the appendix 4 to check the confidence intervals for the functions.



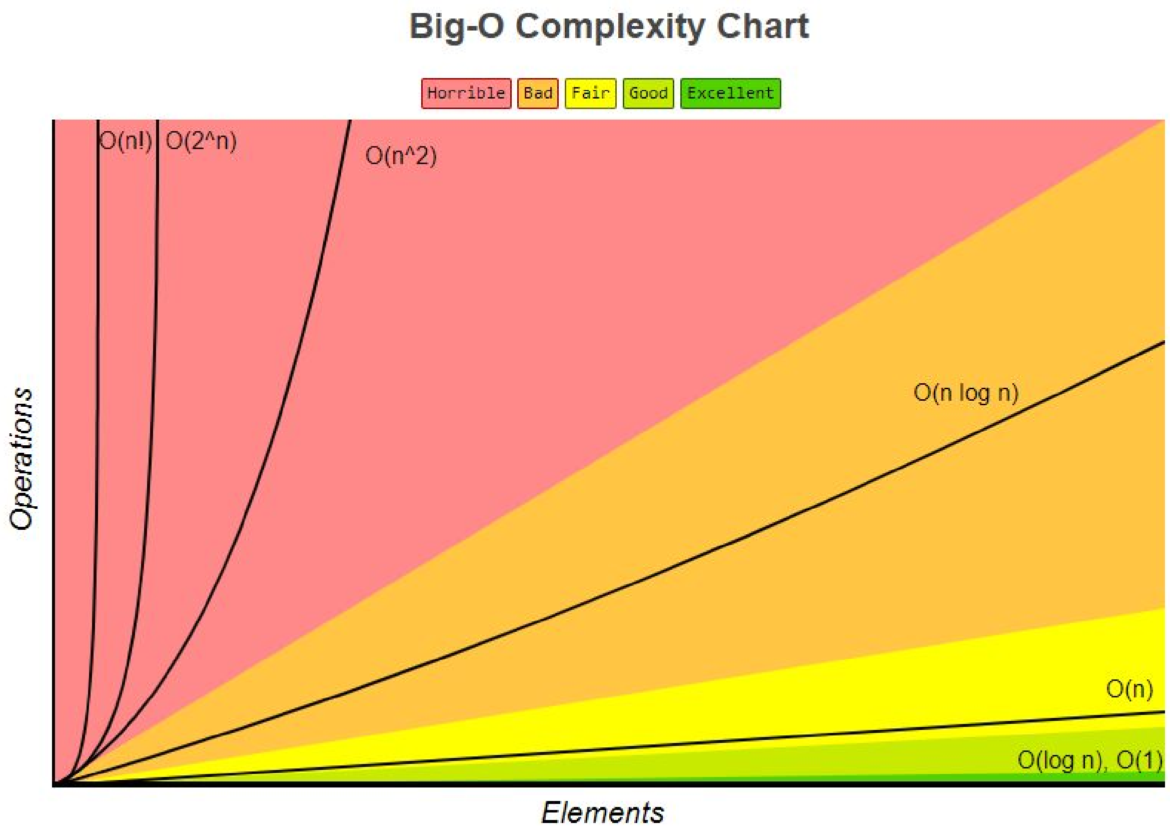
	priority queue size				
500 inserts(deletes)	1000	10000	100000	300000	500000
binary heap					
insert best	41	34	18	18	20
insert average	83	70	40	40	39
insert worst	73	100	48	51	52
delete	242	213	221	333	328
linked list					
insert best	825	675	470	539	408
insert average	2199	9236	105955	303826	533193
insert worst	2318	16969	180104	662532	1091804
delete	12	11	12	13	12

Table 2. Raw data for smaller priority queue sizes

4.1 Discussion

4.1.1 Binary heap

As can be seen on the graph 1, the best, worst, and average performances correspond to the theoretical performance of the binary heap based priority queue, where the performance is described by O(log(N)), because there is barely growth, if any, when increasing from 1 million to 10 million elements. Regarding delete() function, the growth can be described as logarithmic as well, because with the queue size increased from 1 million to 10 million elements, the deletion time has increased not even by a factor of 2. When compared to the O(log(N)), such as on the graph below, there is an apparent resemblance between best, worst, average, and delete cases of the tested algorithm and the actual O(log(N)) graph. It must be noted, that deletion takes significantly more time than insertion for binary heap



Graph 5. Big-O Notation Time Complexity Chart [12]

4.1.2 Linked list

The results achieved for analysing the singly linked list corresponded to the mathematical theory of time complexity. For the best case, the insertion is always done at the head of the list, resulting in a time complexity of O(1). This can be seen on graph 2 with the blue line. The worst case is when inserting an element requires us to traverse the entire list. This is when the element we insert has the lowest priority. The time complexity of this is then O(N), and can be seen with the grey line in graph 2. For the average case we have a random function which assigns each new element with a random priority. This is shown on graph 2 with the orange line. This is still a linear graph with the time complexity O(N), however when compared with the worst case scenario, the conclusion that the average case takes twice as less time can be made.

4.1.3 Comparing insertions between singly linked list and binary heap

Comparing inserting elements in both the singly linked list implementation and the binary heap, it can be observed that the average time taken for insertions is, as the binary heap increases in size, constant throughout. However in comparison to the singly linked list, as the list size grows bigger with more elements, the average time it takes to insert an element grows linearly with the size of the list, thereby making it much more efficient to use a binary heap implementation for a priority queue, especially in relation when working with elements numbering millions.

4.1.4 Comparing deletions between singly linked list and binary heap

When comparing delete values in the table 1, deletion for linked list stays constant, while deletion for binary heap grows very slowly. The bigger the priority queue becomes, bigger the performance difference between two implementation becomes. No matter how big the linked list will be, the deletion operation will always be constant, and because of this, linked list implementation is clearly superior. However, this does not mean that the binary heap version is bad. It is still O(log(N)), which is extremely good, but between the two, linked list is more efficient.

Priority queues of smaller sizes have also been tested briefly (table 2). The data for smaller sizes follows the exact same trends as the data for the bigger sizes, thus making all of the conclusions above applicable to priority queues of smaller sizes.

5 Summary and future work

To sum up, each of the implementations has its own advantages. If deletion time is critical in a system, a linked list based priority queue should be used. However, it has a significant drawback, when compared to the binary heap implementation, which is insertion time for bigger sizes. When working with big data structures, binary heap based priority queue should be used instead. Even with longer deletion times, it is still a better all-round option for bigger queues. However, if it is known by the user that no big queues are going to be used, then the choice of implementation can be based on personal preference or code simplicity, which in this case is much simpler for linked list (can be found in appendix 1). Arguably, a linked list based priority queues up to the size of 1000000 can be used in this case, because even with its linear time complexity it still takes milliseconds to execute, at least on the machines used for testing.

In this study, we have not discussed space complexity of the algorithms that have been implemented. This could be a potential continuation of the study.

References

[1] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to algorithms, 3rd ed. Cambridge, Mass.: MIT Press, 2009.

[2] V. Adamchik, "Stacks and Queues", Cs.cmu.edu, 2009. [Online]. Available: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html. [Accessed: 13- Jan- 2019].

[3] "Priority queue - Rosetta Code", Rosettacode.org, 2018. [Online]. Available: https://rosettacode.org/wiki/Priority_queue#C. [Accessed: 13- Jan- 2019].

[4] V. Adamchik, "LinkedLists", Cs.cmu.edu, 2009. [Online]. Available: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html. [Accessed: 13- Jan- 2019].

[5] S. Mahapatra, "Priority Queue using Linked List - GeeksforGeeks", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/priority-queue-using-linked-list/. [Accessed: 13- Jan- 2019].

[6] R. Garnier and J. Taylor, Discrete mathematics, 3rd ed. Boca Raton: CRC Press, 2010, p. 620.

[7] "6.1 Binary Heaps", Lcm.csa.iisc.ernet.in. [Online]. Available: http://lcm.csa.iisc.ernet.in/dsa/node137.html. [Accessed: 13- Jan- 2019].

[8] "Priority Queues", Pages.cs.wisc.edu. [Online]. Available: http://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html. [Accessed: 13- Jan- 2019].

[9]"ISO C Random Number Functions", Gnu.org. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/ISO-Random.html. [Accessed: 29- Jan- 2019].

[10]P. Selinger, "The GLIBC random number generator", Mscs.dal.ca, 2007. [Online]. Available: https://www.mscs.dal.ca/~selinger/random/. [Accessed: 29- Jan- 2019].

[11]R. Rönngren and R. Ayani, "A Comparative Study of Parallel and Sequential Priority Queue Algorithms", www.cs.auckland.ac.nz/, 1997. [Online]. Available: https://www.cs.auckland.ac.nz/~mcw/Teaching/320/refs/priority-queue/pq-comparative-study.pdf. [Accessed: 29- Jan- 2019].

[12] E. Rowell, "Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell", Bigocheatsheet.com, 2013. [Online]. Available: http://bigocheatsheet.com/. [Accessed: 13- Jan- 2019].

Appendix 1: Source code for the *algorithms*

The whole project is available at: https://github.com/Goradux/Priority-Queues

Singly linked list based priority queue:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct node {
	int data;
	double priority;
	struct node* next;
} Node;

Node* newNode(int dataNew, double priorityNew) {
	Node* temp = (Node*)malloc(sizeof(Node));
	temp->data = dataNew;
	temp->priority = priorityNew;
	temp->next = NULL;

	return temp;
}

// value of the highest priority element
int headValue(Node** head) {
	return (*head)->data;
}

void deleteMax(Node** head) {
	Node* temp = *head;
	(*head) = (*head)->next;
	free(temp);
}

void insert(Node** head, int dataNew, double priorityNew) {
	Node* start = (*head);

	Node* temp = newNode(dataNew, priorityNew);

	// "<=" ensures FIFO
	while (start->next != NULL && start->next->priority <= priorityNew) {
		start = start->next;
	}

	// here we are at the right position
	temp->next = start->next;
	start->next = temp;
}

int isEmpty(Node** head) {
	return (*head) == NULL;
}

void insertBest(long size) {
        clock_t start_t;
        clock_t end_t;
        clock_t total_t;
        long counter;

        Node* pq = newNode(0, 555%size);
        for (counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size-counter);
        }

        start_t = clock();
        for (int i = 0; i < 500; i++){
                insert(&pq, -1, -1);
        }
        end_t = clock();
        total_t = (double)(end_t - start_t);

        printf("Best case, insert\n");
        printf("Total CPU clocks to queue up %ld elements: %ld\n", size, total_t  );
        return;
}

void insertWorst(long size) {
        clock_t start_t;
        clock_t end_t;
        clock_t total_t;
        long counter;

        Node* pq = newNode(0, 555%size);
        for (counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size-counter);
        }
        start_t = clock();
        for (int i = 0; i < 500; i++) {
                insert(&pq, counter, size+1+i);
        }
        end_t = clock();

/*
for (int i = 0; i < 1000; i++) {
        insert(&pq, 1, size+1);
}
*/

        total_t = (double)(end_t - start_t);

        printf("Worst case, insert\n");
        printf("Total CPU clocks to queue up %ld elements: %ld\n", size, total_t  );
        return;
}

void insertAverage(long size){
        srand(time(NULL));

        clock_t start_t;
        clock_t end_t;
        clock_t total_t;
        long counter;

        Node* pq = newNode(0, rand()%size);
        for (counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size-counter);
        }
        start_t = clock();
        for (int i = 0; i < 500; i++) {
                insert(&pq, counter, rand()%size);
        }
        end_t = clock();

        total_t = (double)(end_t - start_t);

        printf("Average case, insert\n");
        printf("Total CPU clocks to queue up %ld elements: %ld\n", size, total_t  );
        return;
}
```

Binary heap based priority queue:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define LEFTCHILD(x) 2*x+1
#define RIGHTCHILD(x) 2*x+2
#define PARENT(x) (x-1)/2

typedef struct node {
  int data;
  int priority;
} node;

typedef struct minHeap {
  int size;
  node *element;
} minHeap;

minHeap initMinHeap() {
  minHeap hp;
  hp.size = 0;
  return hp;
}

void insertNode(minHeap *hp, int data, int priority) {
  //allocating space
  if (hp->size) {
    hp->element = realloc(hp->element, (hp->size+1)*sizeof(node));
  } else {
    hp->element = malloc(sizeof(node));
  }

  //initializing the node with value
  node nd;
  nd.data = data;
  nd.priority = priority;

  //positioning the node at the right position in the min heap
  int i = (hp->size)++;
  while(i && nd.priority < hp->element[PARENT(i)].priority) {
    hp->element[i] = hp->element[PARENT(i)];
    i = PARENT(i);
  }
  hp->element[i] = nd;
}

void swap(node *n1, node *n2) {
  node temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}

void heapify(minHeap *hp, int i) {
  int smallest = (LEFTCHILD(i) < hp-> size && hp->element[LEFTCHILD(i)].priority <
hp->element[i].priority) ? LEFTCHILD(i) : i;
  if (RIGHTCHILD(i) < hp->size && hp->element[RIGHTCHILD(i)].priority <
hp->element[smallest].priority) {
    smallest = RIGHTCHILD(i);
  }
  if (smallest != i) {
    swap(&(hp->element[i]), &(hp->element[smallest]));
    heapify(hp, smallest);
  }
}

void deleteNode(minHeap *hp) {
  if (hp->size) {
    hp->element[0] = hp->element[--(hp->size)];
    hp->element = realloc(hp->element, hp->size*sizeof(node));
    heapify(hp, 0);
  } else {
    printf("\n OOPS! Heap is empty already!\n");
    free(hp->element);
  }
}
```

Appendix 2: Source code for the tests (validation)

```
void algoTestLinkedList(int size){
       Node* pq = newNode(0, 0);
```

```
        for (int counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size);
        }
        printf("List built!\n");
        for (int counter = 0; counter <= size; counter++) {
                deleteMax(&pq);
        }
        printf("List deleted!\n");
}

void algoTestBinaryHeap(int size) {
  minHeap hp = initMinHeap();
  for (int i = 0; i < size; i++) {
    insertNode(&hp, i, i);
  }
  printf("Heap built!\n");
  for (int i = 0; i < size+1; i++) {
    deleteNode(&hp);
  }
  printf("Heap deleted!\n");
}
```

Appendix 3: Source code for the experiments

Singly linked list tests:

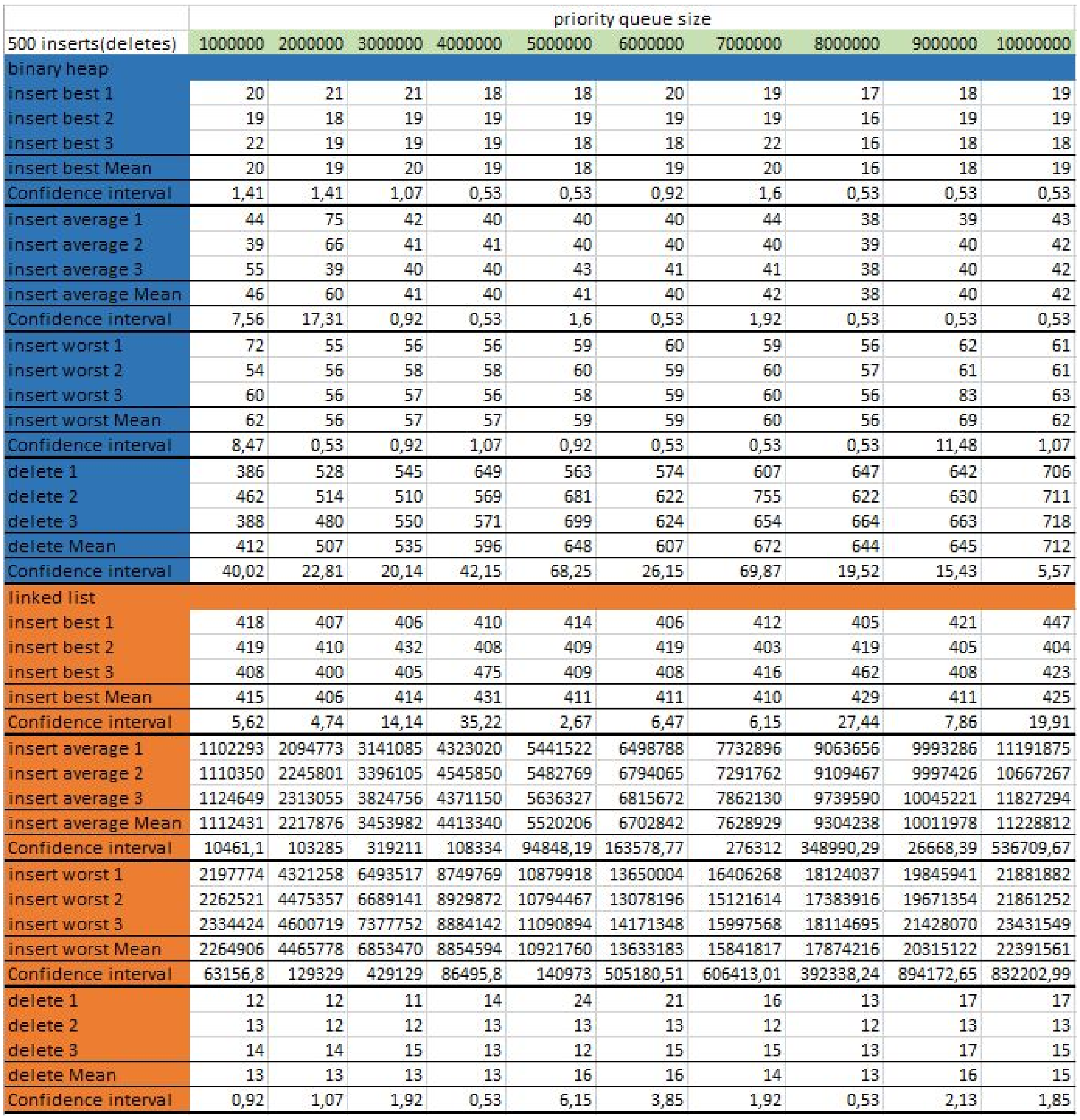
```
void insertBest(long size) {
        clock_t start_t;
        clock_t end_t;
        clock_t total_t;
        long counter;

        Node* pq = newNode(0, 555%size);
        for (counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size-counter);
        }

        start_t = clock();
        for (int i = 0; i < 500; i++){
                insert(&pq, -1, -1);
        }
        end_t = clock();
        total_t = (double)(end_t - start_t);

        printf("Best case, insert\n");
        printf("Total CPU clocks to queue up %ld elements: %ld\n", size, total_t  );
        return;
}

void insertWorst(long size) {
        clock_t start_t;
        clock_t end_t;
        clock_t total_t;
        long counter;

        Node* pq = newNode(0, 555%size);
        for (counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size-counter);
        }
        start_t = clock();
        for (int i = 0; i < 500; i++) {
                insert(&pq, counter, size+1+i);
        }
        end_t = clock();

/*
for (int i = 0; i < 1000; i++) {
        insert(&pq, 1, size+1);
}
*/

        total_t = (double)(end_t - start_t);

        printf("Worst case, insert\n");
        printf("Total CPU clocks to queue up %ld elements: %ld\n", size, total_t  );
        return;
}

void insertAverage(long size){

        srand(time(NULL));

        clock_t start_t;
        clock_t end_t;
        clock_t total_t;
        long counter;

        Node* pq = newNode(0, rand()%size);
        for (counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size-counter);
        }
        start_t = clock();
        for (int i = 0; i < 500; i++) {
                insert(&pq, counter, rand()%size);
        }
        end_t = clock();

        total_t = (double)(end_t - start_t);

        printf("Average case, insert\n");
        printf("Total CPU clocks to queue up %ld elements: %ld\n", size, total_t  );
        return;
}

void testInsert(long size) {
        insertBest(size);
        insertAverage(size);
        insertWorst(size);

        return;
}

void testDeleteMax(long size){
        clock_t start_t;
        clock_t end_t;
        clock_t total_t;
        long counter;

        Node* pq = newNode(0, 5555%size);
        for (counter = 0; counter <= size; counter++) {
                insert(&pq, counter, size-counter);
        }

        start_t = clock();
        for (int i = 0; i < 500; i++) {
                deleteMax(&pq);
        }
        end_t = clock();

        total_t = (double)(end_t - start_t);

        printf("Any case, deleteMax\n");
        printf("Total CPU clocks to delete %ld elements: %ld\n", size, total_t  );
        return;
}
```

Binary heap tests:

```
void insertWorst(int input) {
  clock_t start_t;
  clock_t end_t;
  clock_t total_t;

  int i;
  int size = input;

  minHeap hp = initMinHeap();
  for (i = 0; i < size; i++) {
    insertNode(&hp, i, i);
  }
  printf("Worst case\n");
  start_t = clock();
  for (i = 0; i < 500; i++) {
    insertNode(&hp, 0, -1 - i);
  }
  end_t = clock();
  total_t = (double)(end_t - start_t);
  printf("Clock ticks taken: %ld\n", total_t);
```

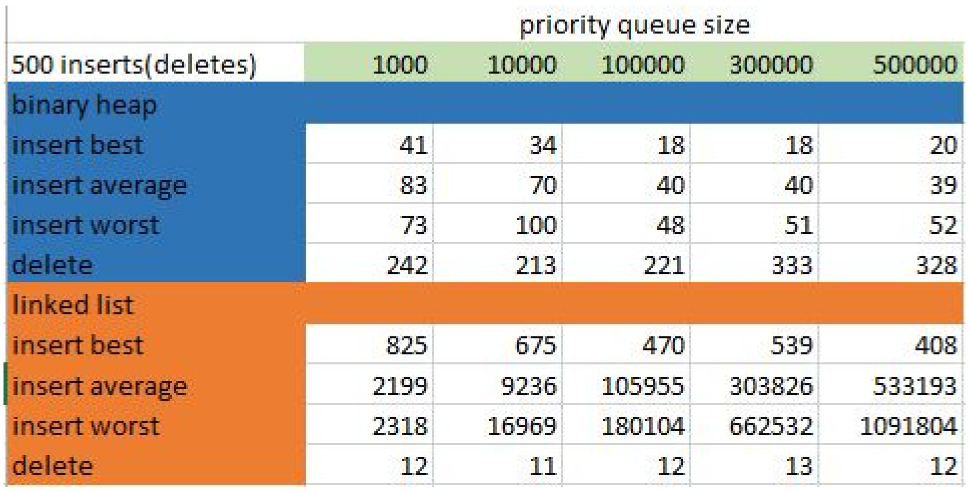
```c
}

//insert at the end of the heap
void insertBest(int input) {
  clock_t start_t;
  clock_t end_t;
  clock_t total_t;

  int i;
  int size = input;

  minHeap hp = initMinHeap();
  for (i = 0; i < size; i++) {
    insertNode(&hp, i, i);
  }
  printf("best case\n");
  start_t = clock();
  for (i = 0; i < 500; i++) {
    insertNode(&hp, 0, input + 1 + i);
  }
  end_t = clock();
  total_t = (double)(end_t - start_t);
  printf("Clock ticks taken: %ld\n", total_t);
}

void insertAverage(int input) {

  srand(time(NULL));

  clock_t start_t;
  clock_t end_t;
  clock_t total_t;

  int i;
  int size = input;

  minHeap hp = initMinHeap();
  for (i = 0; i < size; i++) {
    insertNode(&hp, i, i);
  }
  printf("Average case\n");
  start_t = clock();
  for (i = 0; i < 500; i++) {
    insertNode(&hp, 0, rand()%size);
  }
  end_t = clock();
  total_t = (double)(end_t - start_t);
  printf("Clock ticks taken: %ld\n", total_t);
}


void testDelete(int input) {
  clock_t start_t;
  clock_t end_t;
  clock_t total_t;

  int i;
  int size = input;

  minHeap hp = initMinHeap();
  for (i = 0; i < size; i++) {
    insertNode(&hp, i, i);
  }
  printf("Delete case\n");
  start_t = clock();
  for (i = 0; i < 500; i++) {
    deleteNode(&hp);
  }
  end_t = clock();
  total_t = (double)(end_t - start_t);
  printf("Clock ticks taken: %ld\n", total_t);
}
```

Appendix 4: Raw data from the experiments



	priority queue size									
500 inserts(deletes)	1000000	2000000	3000000	4000000	5000000	6000000	7000000	8000000	9000000	10000000
binary heap										
insert best 1	20	21	21	18	18	20	19	17	18	19
insert best 2	19	18	19	19	19	19	19	16	19	19
insert best 3	22	19	19	19	18	18	22	16	18	18
insert best Mean	20	19	20	19	18	19	20	16	18	19
Confidence interval	1,41	1,41	1,07	0,53	0,53	0,92	1,6	0,53	0,53	0,53
insert average 1	44	75	42	40	40	40	44	38	39	43
insert average 2	39	66	41	41	40	40	40	39	40	42
insert average 3	55	39	40	40	43	41	41	38	40	42
insert average Mean	46	60	41	40	41	40	42	38	40	42
Confidence interval	7,56	17,31	0,92	0,53	1,6	0,53	1,92	0,53	0,53	0,53
insert worst 1	72	55	56	56	59	60	59	56	62	61
insert worst 2	54	56	58	58	60	59	60	57	61	61
insert worst 3	60	56	57	56	58	59	60	56	83	63
insert worst Mean	62	56	57	57	59	59	60	56	69	62
Confidence interval	8,47	0,53	0,92	1,07	0,92	0,53	0,53	0,53	11,48	1,07
delete 1	386	528	545	649	563	574	607	647	642	706
delete 2	462	514	510	569	681	622	755	622	630	711
delete 3	388	480	550	571	699	624	654	664	663	718
delete Mean	412	507	535	596	648	607	672	644	645	712
Confidence interval	40,02	22,81	20,14	42,15	68,25	26,15	69,87	19,52	15,43	5,57
linked list										
insert best 1	418	407	406	410	414	406	412	405	421	447
insert best 2	419	410	432	408	409	419	403	419	405	404
insert best 3	408	400	405	475	409	408	416	462	408	423
insert best Mean	415	406	414	431	411	411	410	429	411	425
Confidence interval	5,62	4,74	14,14	35,22	2,67	6,47	6,15	27,44	7,86	19,91
insert average 1	1102293	2094773	3141085	4323020	5441522	6498788	7732896	9063656	9993286	11191875
insert average 2	1110350	2245801	3396105	4545850	5482769	6794065	7291762	9109467	9997426	10667267
insert average 3	1124649	2313055	3824756	4371150	5636327	6815672	7862130	9739590	10045221	11827294
insert average Mean	1112431	2217876	3453982	4413340	5520206	6702842	7628929	9304238	10011978	11228812
Confidence interval	10461,1	103285	319211	108334	94848,19	163578,77	276312	348990,29	26668,39	536709,67
insert worst 1	2197774	4321258	6493517	8749769	10879918	13650004	16406268	18124037	19845941	21881882
insert worst 2	2262521	4475357	6689141	8929872	10794467	13078196	15121614	17383916	19671354	21861252
insert worst 3	2334424	4600719	7377752	8884142	11090894	14171348	15997568	18114695	21428070	23431549
insert worst Mean	2264906	4465778	6853470	8854594	10921760	13633183	15841817	17874216	20315122	22391561
Confidence interval	63156,8	129329	429129	86495,8	140973	505180,51	606413,01	392338,24	894172,65	832202,99
delete 1	12	12	11	14	24	21	16	13	17	17
delete 2	13	12	12	13	13	13	12	12	13	13
delete 3	14	14	15	13	12	15	15	13	17	15
delete Mean	13	13	13	13	16	16	14	13	16	15
Confidence interval	0,92	1,07	1,92	0,53	6,15	3,85	1,92	0,53	2,13	1,85

Table 3. Raw data from the measurements (in clock ticks)



	priority queue size				
500 inserts(deletes)	1000	10000	100000	300000	500000
binary heap					
insert best	41	34	18	18	20
insert average	83	70	40	40	39
insert worst	73	100	48	51	52
delete	242	213	221	333	328
linked list					
insert best	825	675	470	539	408
insert average	2199	9236	105955	303826	533193
insert worst	2318	16969	180104	662532	1091804
delete	12	11	12	13	12

Table 4. Raw data from then smaller priority queue size