Priority Queues Experiment

December 3, 2017

WONG, SAI MAN
TIGERSTRÖM, GABRIEL

Abstract

This is a skeleton for KTH theses. More documentation regarding the KTH thesis class file can be found in the package documentation.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis. Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasellus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Referat

Lorem ipsum dolor sit amet, sed diam nonummy nibh eui mod tincidunt ut laoreet dol

Denna fil ger ett avhandlingsskelett. Mer information om LaTeX-mallen finns i dokumentationen till paketet.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis. Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasellus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Contents

Chapter 1

Introduction

To plan in advance and thoroughly is one of the cores to successfully carry out a credible scientific work. However, students and engineers sometimes takes this step unseriously and carelessly. As a result, time can get wasted to redo everything due to poor design and execution. To prevent this, students and engineers need to possess a deeper knowledge of the scientific method. That is, not only theoretical knowledge, but also how to apply it in practice is equally as important. Thus, in this study, we used the scientific method to evaluate algorithms with a rigorous, honest and transparent method.

1.1 Background

A data structure is a model to represent data, and there is a large number of different types [1]. The common challenge is to identify the most feasible data structure for a specific purpose, such as a queue. Software engineers frequently use queues in software development. For example, to manage arbitrary entities in a first-in-first-out (FIFO) order, such as a memory buffer or scheduling in an operating system.

In high-level programming languages, there are libraries that provide finished features, such as a queue-implementation. However, this may result in that developers often take the underlying implementations for granted. Thus, they may possess insufficient theoretical and technical knowledge about the features they use.

Another model to represent data linearly is a linked list. Linked list comes in different variations. However, the basics of a simple linked list is that an entity stores data and has a connection to the next entity. In computer science, the entity is referred to as a node or an element, and the connection is called a pointer. The most simple linked list implementation represent data linearly, because each node stores data and has a pointer to the next node. Thus, developer frequently use different variations of linked lists to represent a list or queue.

1.2 Problem

The scientific method is crucial for further development within any research field. Also, to understand the underlying implementation of features in an API[1]-library. However, the problem is that engineers and students sometimes takes these for granted. Thus, in this study we applied the scientific method rigorously to evaluate algorithms in depth, both theoretically and practically.

Our task was to evaluate four variations of a list-based data structure to represent a queue [2]. Each element has a numerical value to represent its priority in the queue. In this study and to clarify, an element with a low numerical value represents higher priority, because it is closer to the time and ready for execution. We implemented the queues in the low-level programming language `C`, and the following are the implementation specifications:

1. **Singly linked list** – Insertion of new elements takes place in the head.
2. **Doubly linked list** – Insertion of new elements takes place in the rear.
3. **Doubly linked list** – Insertion of new elements takes place based on the mean value of the first and last element's mean value.
    - In the head – New element has a higher priority than the mean value, that is,
      numerical value $<$ mean value.
    - In the rear – New element has a lower or equal priority to the mean value, that is,
      numerical value $\geq$ mean value.
4. **Scheduling queue** – There is an FIFO-queue for each priority in the $[0, 40]$ interval.

The relevant parameters to use and examine are primarily time and space for algorithms, such as execution time and memory usage. That is, we can gather a large quantity of these metric to determine best, average and worst cases. Thus, we asked ourselves: "Do these queues achieve similar performance, or does the performance vary depending on the circumstance?"

[1]API – Application Programming Interface

1.3 Hypothesis

A queue's purpose is to enqueue and dequeue elements with an FIFO approach. Because of the FIFO policy, we assumed with a correct implementation and theory that the dequeue-operation should take $\mathcal{O}(1)$ time. However, the enqueue-operation is more sensitive to the element's priority, because it must insert each element in the correct spot in the queue. Thus, the implementations can perform differently due to the distribution of the priorities. For example, the implementation reaches $\mathcal{O}(n)$ time if it always has to traverse through the entire list to enqueue an element.

Our hypotheses were the following:

- The dequeue-operation for all the queues takes $\mathcal{O}(1)$ time.
- The enqueue-operation for all the queues takes $\mathcal{O}(n)$ time.

1.4 Purpose

Our purpose with this paper was to raise awareness about the features in API-libraries and scientific aspects, which are often taken for granted. Thus, we conduct experiments to get a deeper insight into four varieties of a queue-implementation and the scientific method. Consequently, we wanted to contribute with a study other students and engineers can learn from and further develop.

1.5 Goal

The bigger goal was to get a deeper knowledge about the scientific method, and apply it to evaluate four queue-implementations. However, the following list demonstrates the sub-goals:

- Apply scientific method to plan, design, experiment, document, analyze and present the findings.
- Understand the importance of scientific method.
- Conduct a study which is reproducible.
- Attain the ability to present, analyze and draw conclusions from experiments, both theoretically and practically.
- Implement queues correctly, that is, to verify its behavior and validate the data.
- Create tests to generate the implementation's execution time and memory usage.

1.6 Social Benefit, Ethics and Sustainable Development

This study can influence students and engineers to apply the scientific method more seriously. Also, it can facilitate the process for software engineers to choose an implementation of a queue that fits their specific purpose. Finally, this study delves into the foundations of algorithms, data structure and complexity theory. Thus, it can help students and engineers to get a deeper knowledge within computer science.

In scientific community, it is a researcher's moral responsibility to try to make the work as easily reproducible as best to their ability and circumstances. That is, so that other people can reproduce and further develop the work. We documented each step and presented findings with rigor, honesty and transparency. That is, we emphasize that other people try to reproduce our findings.

We conducted the experiment on a virtual machine on Microsoft's cloud platform. Thus, we only used the exact amount of resources needed to conduct this experiment. That is, we did not waste unnecessary electricity due to, for example, system idle.

1.7 Outline

Chapter 2

Method

2.1 Inductive and Deductive Method

Scientists apply an inductive method when they try to reach practical conclusions from observations or data [3]. That is, in sequential order to 1) collect data, 2) detect pattern in it, 3) set up a hypothesis and 4) try to explain the findings with established theory to reach 5) a conclusion [4]. It is sometimes informally called a bottom-up approach due to its style to start with observations. Software engineers frequently apply this method to observe algorithms and a system's behavior. For example, a developer with specific a input can generate output from an algorithm to create an understanding of how it works.

On the contrary, scientists use a deductive method, or top-down approach, to try to reach logical conclusions from theory. For example, mathematicians use a deductive method to try to explain something unknown with help of well-established theory (axioms and theorems). This also applies to researchers within computer science. That is, they use mathematical notations to represent theoretical models in, for example, algorithm analysis and complexity theory. Thus, researchers use these methods for different purposes, and they sometimes combine them to reach both a logical and practical conclusion.

2.2 Quantitative and Qualitative Method

Researches apply a quantitative method in studies that rely on measurable data in large quantities to explain findings [5]. That is, the metrics in a quantitative study are measurable, such as time, temperature and age. These studies frequently apply statistical analysis to summarize the data in tables and graphs with error margins. For example, scientists use a quantitative method to try to predict a nation's growth based on the birth and death rate over a longer period.

In contrast, a qualitative method relies on data of quality to interpret the results [6].

Scientists use this method in studies when the data significantly more difficult to quantify, such as opinions, thoughts and happiness. The data are usually collected from other people in form of interviews and surveys. Thus, psychologists frequently implement a qualitative method to try to understand people's behavior.

2.3 Data Collection

The most relevant metrics to examine algorithms with are time and space, such as execution time and memory usage. These metrics are all quantifiable. In contrast to, for example, interviews and surveys, which are common methods in qualitative studies [3, 5]. That is, qualitative studies reaches conclusions from the quality of the data rather quantity. Thus, this study uses an experimental quantitative method to try to get a deeper insight about queues.

For each of the four queue-implementations, we tested and generated the execution time for best, average and worst case. That is, we created input for the queues based on different distributions of the element's priorities. Besides execution time, we also log the memory usage for each implementation. Finally, we wrote scripts to execute the tests and logged all the raw data.

2.4 Data Analysis

We reasoned with a deductive method to explore the theory within computer science, such as algorithm complexity theory. That is, we can reach theoretical conclusions from complexity theory about the algorithms. For example, it is possible to determine an algorithm's performance based solely on theoretical knowledge. On the contrary, the usage of deductive methods can also yield unexpected results when put into practice. Thus, we generate realistic data from our experiments, and use an inductive method to further analyze it. That is, to explore if we can notice any observable patters in the data with help of models.

We used both graphs and tables to visualize the execution time and memory usage And, ran the experiments multiple times to get the sample mean and standard deviation. Standard deviation is important because it provides a deeper insight about the results around the sample mean. Also, it can eventually help us identify the external or internal error sources. For example, if the standard deviation is relatively high, then we can suspect that something is wrong.

Chapter 3

System Description

3.1 System Development Method

Our implementation produces an executable for each of the four queue-implementations. It consists of primarily three files, `main.c` with all the tests, `queues.h` and `queues.c` which respectively describes and implements the functionalities. Thus, we use compile time macros to determine which of the four queue-implementation we create an executable from.

We used Bash-scripts to automatically run tests and log the results in plain text. Also, we wrote a Python-script that 1) calculated the sample mean and standard deviation, and 2) summarized it in a specific format so that LaTeX can visualize it in either tables or graphs. Finally, we also use Valgrind to examine that our implementation allocates and deallocates memory correctly.

We also added an optional solution to run the entire experiment in a container. That is, if anyone only wants to run the entire experiment without thinking about the set up process, then the user only need to install Docker and run one command to achieve the results as we did. It is efficient, because containers shares the same resources as the host operating system.

3.2 Queue Algorithms

Our task is to implement four different queue-implementation in the programming language `C`, as described in Section 1.2. The following are common operations for a queue:

- **Enqueue** – enqueues the element in the correct spot in the queue, because it is a priority queue.
- **Dequeue** – dequeues the element with the highest priority, that is, the first element in the queue.

The only operation that can vary in the queue-implementations is the enqueue-operation, because it must insert the element in the correct position in the queue. Another common feature is that each element stores at least 1) an unique number identifier of the data type integer, 2) a priority number of the data type double and 3) a pointer to refer to other elements.

The following sections describe how we implemented the queues. Additionally, we analyze the time complexity for the enqueue-operation as it varies. Average cases are difficult to analyze theoretically. However, best and worst cases are easier to analyze in theory. Thus, we analyze the best and worst case to try to understand the average case for each implementation's enqueue-operation.

3.2.1 Singly Linked List

This implementation uses a singly linked list to represent a queue. That is, an element in the queue stores 1) a unique number identifier, 2) a priority and 3) a pointer to the next element in the queue. This algorithm enqueues new elements from head. The following list analyzes the enqueue-operation's time complexity:

- **Best Case** $\mathcal{O}(1)$ – This occurs when the enqueue-operation of new elements always takes place in the head. That is, the implementation does not have to traverse the list to find the correct spot to insert the new element. Instead, it needs to only perform one operation to put the new element in the head. Thus, we can reach the best case with a distribution of priorities where the numerical value descends from a higher one.

- **Worst Case** $\mathcal{O}(n)$ – This happens when the implementation must traverse through the entire list, or n-times, to insert the new element in the correct place. Thus, the implementation reaches the worst case scenario with a distribution of priorities with the same numerical value, because the queue uses an FIFO approach to manage elements with the same priority.

3.2.2 Doubly Linked List

This implementation uses a doubly linked list data structure. An element stores 1) an unique number identifier, 2) priority, and 3-4) two pointers that connects next and previous element. In this implementation, the enqueue-operation of new elements takes place in the rear. The following examines the time complexity of this implementation:

- **Best Case** $\mathcal{O}(1)$ – This implementation reaches the best case when the priority distribution consists of the same number. Because, it applies FIFO to manage elements with same priorities. Thus, it requires only one operation to insert the new element.

- **Worst Case** $\mathcal{O}(n)$ – This queue reaches the worst case when the priority's numerical value constantly descends from a higher one. With such numerical distribution, it has to traverse the list from the rear n-times to insert the element correctly.

3.2.3 Doubly Linked List with Modification

This queue is similar to the previous section's implementations. That is, an element stores the same properties as the last one. However, in this implementation the insertion of new elements takes place in either the head or rear. It depends on the average of the first and last element in the list. The following list clearly demonstrates the rules of the modified version:

- In the head – New element has a higher priority than the mean value, that is, numerical value $<$ mean value.

- In the rear – New element has a lower or equal priority to the mean value, that is,
  numerical value $\geq$ mean value.

We can also analyze the best and worst case for this implementation. The following summarizes the analysis:

- **Best Case** $\mathcal{O}(1)$ – For the best case, we can either design the distribution of priorities to only take place in either head or rear. For example, let us say that we create a distribution with only the same priority. The average will always be the same as the new element's priority. Thus, the insertion of new elements always takes place in the rear.

- **Worst Case** $\mathcal{O}(n)$ – To achieve the worst case, we must traverse through the list for each enqueue-operation. This is possible if the queue, for example, first enqueues two elements with significantly different priorities. One with a high priority, and another with significant lower priority. The average will skew towards the lower priority, that is, a notably high numerical value. Thus, if the new element always will have a little bit lower priority than the previous one, then the insertion takes place in the head and traverses $(n-1)$-times to insert new element correctly.

3.2.4 Scheduling Queue

This implementation uses a static list with 41 entries of pointers These entries represent the priority in the [0, 40] interval, and each pointer is connected to an FIFO-queue. Our implementation of the FIFO-queue is a singly linked list, as described in Section 3.2.1 The time complexity for best and worst case is:

- **Best Case** $\mathcal{O}(n)$ – The implementation achieve the best time complexity if the elements are equally distributed in the queue. The simplest distribution is to enqueue elements from highest to lowest priority and start over. That is, from priority 0 to 40, and then start over from priority 0. However, when the implementation starts over from priority 0, the enqueue operation still has to traverse through the FIFO-queues to insert the new elements correctly. For example, if we assume that the elements are distributed equally, we can represent the running time as:

$$1 + \left\lfloor \frac{n}{41} \right\rfloor$$

  The one represents the cost to access the priority queue. The rest represents the cost to traverse through the list.

- **Worst Case** $\mathcal{O}(n)$ – The worst case is the same as described Section 3.2.1. That is, the implementation has to traverse n-times for a distribution of elements with only the same priority. The cost to access the priority queue can influence the running time. However, in complexity theory, the cost to access the it is insignificant.

3.3 Experimental Setup

We conducted the experiments on a Microsoft Azure virtual machine. Basically, it is a high-performance virtual private server. We used this service because we have full control over the environment. The system consisted of a 64 bit architecture virtual CPU with two cores (Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz), 8 GB memory and Ubuntu 16.04.3 LTS (Xenial Xerus) operating system.

3.3.1 Error Sources

Error sources are also necessary to take into account as these can influence the results of this study. That is, if we identify error sources early in the planning phase, we can then mitigate unexpected behaviors and results. Every software and computational research depend heavily on the underlying system, both hardware and software. Assume that the implementation is correct, the results depend on memory and CPU to produce accurate output, but also the operating system. Thus, we run the experiments multiple times and calculate statistical features, such as mean value and standard deviation. Standard deviation is relevant, because we can then identify how the execution time vary around the mean. For example, a large standard deviation indicates, for example, that another process irregularly consume a large amount of resources.

Another error source is poor design and implementation of models. Thus, it important to grasp and follow scientific methodology rigorously, verify and validate

Table 3.1. Experiment parameters for each implementation and test case

Iterations	50
Elements	10 000 – 60 000 (multiples of 5000)

the implementations. It gets significantly more difficult to identify error sources, if the latter mentioned points are not taken seriously. For example, a poor implementation design results in that the researcher waste valuable time to debug. And, especially in a relative low-level programming language, such as C.

3.3.2 Verification and Validation

We developed a test case to verify the implementations correctness. That is, to test that the queues inserted the elements correctly. We wrote a Python script that generates an input- and output-file. The former consists of elements with random priorities, and the output file contains the elements in correct order. Finally, we used these files in our implementations to verify its correctness. We also used Valgrind to verify that we allocated and deallocated memory correctly.

We developed tests for best, average and worst cases to validate its behavior. That is, we analyzed the running time and complexity theoretically. And, used statistical analysis to validate the algorithm with the theory, and vice versa. Finally, we tried to use different inputs and examine different edge cases, for example, an empty or a list with only one element.

3.3.3 Experiments

We developed a function that creates n element with randomized priorities in the $[0, 40]$ interval. Furthermore, the verification use the latter mentioned distribution of elements as input to enqueue it in both the queue implementation based on the standard library and our implementation. Finally, we dequeue the elements from the two queues to to verify our implementation's correctness.

To test best, average and worst case, we created different priority distributions as input based on our complexity analysis of our algorithms. This analysis takes place in Section 3.2. The average case is more difficult to analyze theoretically. However, we can assume that average case is between the best and worst case. We randomly assigned priorities in $[0, 40]$ interval to the elements in the average case tests. Finally, Table 3.1 presents the remaining parameters we used for each test case.

The following list summarize the element's priority distribution, where N is the set of elements, p is a numerical priority and n is the number of elements:

- **Singly linked list**
  - Best case – Each element's priority increases linearly, for example,

    $$N = \left\{p_1, (p-1)_2, (p-2)_3, (p-3)_4, ..., (p-n)_n \,\middle|\, p \geq n \geq 0\right\}$$

  - Worst case – Each element's priority is the same, for example,

    $$N = \left\{p_n \,\middle|\, p \geq 0,\ n \geq 0\right\}$$

- **Doubly linked list**
  - Best case – Each element's priority is the same, for example,

    $$N = \left\{p_n \,\middle|\, p \geq 0, n \geq 0\right\}$$

  - Worst case – Each element's priority increases linearly, for example,

    $$N = \left\{p_1, (p-1)_2, (p-2)_3, (p-3)_4, ..., (p-n)_n \,\middle|\, p \geq n \geq 0\right\}$$

- **Doubly linked list with modification**
  - Best case – Each element's priority is the same, for example,

    $$N = \left\{p_n \,\middle|\, p \geq 0, n \geq 0\right\}$$

  - Worst case – Insert two element with significant different priority to skew the average. Then, linearly decrease the priority from one with the highest one, for example:

    $$N = \left\{p_1, (p*2)_2, (p+1)_3, (p+2)_4, (p+3)_5, ..., (p+n-2)_{(n-2)} \,\middle|\, p \geq n \geq 3\right\}$$

- **Scheduling queue**
  - Best case – The element's priority is equally distributed, for example,

    $$N = \left\{((p+1) \bmod 41)_1, ((p+2) \bmod 41)_2, ..., ((p+n) \bmod 41)_n \,\middle|\, 0 \leq p \leq 40, n \geq 0\right\}$$

  - Worst case – Each element's priority is the same, for example,

    $$N = \left\{p_n \,\middle|\, p \geq 0, n \geq 0\right\}$$

Chapter 4

Results and Discussion

4.1 Running Time

We visualized the running time results from the implementations with graphs and tables in Appendix B, Appendix C, Appendix D and Appendix E. That is, the singly linked list, doubly linked list, doubly linked list with modification and scheduling queue respectively.

4.1.1 Enqueue

The results similarity is that the worst case tests represent an upper bound limit of time complexity for each of the algorithms. The same goes for best case tests, that is, these show a lower bound limit. And the average case's time complexity lies somewhere between these limits. Another observation is that all the implementations perform similarly in best case and worst case. Finally, the singly and doubly linked list yielded similar running time for the average case. That is, around 18 seconds with 60 000 elements.

The doubly linked list with modification and scheduling queue algorithms produced a lower running time in average case tests. That is, around 10 seconds and 1 second, respectively, for 60 000 elements. The average case tests for the scheduling queue are only milliseconds higher than for its best case. On the contrary, the other implementation's average case tests produced running times that are several seconds higher than their average case. That is, at least for more than 25 000 elements.

4.1.2 Dequeue

There is no significant changes in the results for these experiments, because the dequeue operation is similar for all the implementations. That is, the elements are in the correct order in the queue. The implementation only needs to dequeue and remove the element from the head of the list. Thus, the running time slightly differ in milliseconds.

4.2 Memory Usage

Appendix F summarizes the memory usage of all the implementations for best, average and worst case tests. The memory is the same for each test on a specific implementation, because it allocates the same number of elements. Thus, we did not have to calculate the mean and standard deviation value for memory usage.

All the implementations memory usage increase linearly with the number of elements. The doubly linked list with and without modification reached similar memory usage. That is, around 4 Megabytes for 60 000 elements. And, singly linked list and priority queue implementation had similar memory usage. That is, around 6 Megabytes for 60 000 elements. The singly linked list and queue implementation use significantly less memory than the variations of doubly linked list. For example, for 60 000 elements, the former implementations use almost 2 Megabytes less memory than the queue based on doubly linked lists.

4.3 Discussion

4.3.1 Running Time Analysis

Our complexity analysis matched with the results we generated from best, average and worst case tests. That is, the worst case test represents an upper bound limit for the running time of the algorithms, and the best case the lower bound. However, the most noticeable observation is that the scheduling queue implementation produced almost as low running time as for its lower bound limit. Thus, the scheduling queue implementation performed the best in terms of time.

The scheduling queue has to traverse through a list, in general, fewer times than the rest of the implementations, because it has a assigned list for each priority. We can use our complexity analysis and math notation to express the average running time, where n represents number of elements:

- **Singly Linked List and Doubly Linked List** – In worst case, these algorithms must traverse n times. Thus, we assume in average that these algorithms must at most traverse $n/2$ times,

- **Doubly Linked List with Modification** – It worst case, this algorithm must traverses the list n times. However, due to its modification, the insertion of new elements takes place in either the head or rear based on an average of first and last element's priority in the queue. Thus, we assume in average that this algorithm must at most traverse $n/4$ times.

- **Scheduling Priority** – This algorithm traverses the list n times in worst case. However, due to its 41 different FIFO queues, each separate queues are in general shorter than the previous mentioned queues. Thus, we assume in average that this algorithm must at most traverse $n/41$ times.

Hence, based on the assumptions in the list above, we can summarize each implementation's average time complexity in relation to the number of elements as the following:

$$\begin{bmatrix} (1) & \text{Singly Linked List} & = & \frac{n}{2} \\ (2) & \text{Doubly Linked List} & = & \frac{n}{2} \\ (3) & \text{Doubly Linked List with Modification} & = & \frac{n}{4} \\ (4) & \text{Scheduling Queue} & = & \frac{n}{41} \end{bmatrix} \implies \frac{n}{2} \geq \frac{n}{4} \geq \frac{n}{41}$$

In average, we noticed that time complexity of (4) is approximately 10× higher than (3), and 20× higher than (1) and (2). Also, the time complexity of (1) and (2) is roughly 2× higher than (1). Thus, our running time results from the average case tests yielded similar time complexity. For example, our running time for $n = 60000$ was the following:

$$\begin{bmatrix} (1) & 17695.48 \pm 2 \times 201.24 \text{ ms} & \approx 20000 \text{ ms} & \approx 2 \times (3) \text{ or } 20 \times (4) \\ (2) & 18662.72 \pm 2 \times 201.24 \text{ ms} & \approx 20000 \text{ ms} & \approx 2 \times (3) \text{ or } 20 \times (4) \\ (3) & 9595.70 \pm 2 \times 118.06 \text{ ms} & \approx 10000 \text{ ms} & \approx 10 \times (4) \\ (4) & 950.29 \pm 2 \times 22.56 \text{ ms} & \approx 1000 \text{ ms} & \end{bmatrix}$$

Finally, the dequeue operation for all the algorithms have similar behavior. That is, all the dequeue operations take $\mathcal{O}(1)$ time.

4.3.2 Memory Usage Analysis

The reason both variations of doubly linked list has similar memory usage, and higher memory usage than singly linked list and scheduling queue, is because these implementations store in total two pointers. That is, doubly linked lists have a pointer that connects to the previous element, and the second connects to the next element. On the contrary, singly linked list and scheduling queue only store one pointer, which refers to the next element in the queue. Thus, the difference in memory usage is because of an extra pointer allocation.

Chapter 5

Conclusions and Future Work

Appendix A

Requirement Overview

In [7], the author describes that a requirement analysis is a summary of all the requirement the researcher can find. The author clearly states that this document is *not* about how one should implement something to achieve a specific requirement. On the contrary, it should only cover the meaning of the requirement. Thus, we need to set up requirements that describes how to pass the task about the scientific method, specifically experimental quantitative method, as shown in Table A.1.

Table A.1. Requirement analysis about the scientific method

Requirement number	Requirement type	Name or source	Clarified description of the requirement, followed by what should be fulfilled	Fulfilled or not fulfilled or partly fulfilled
1	**Must** mandatory task	Purpose [8]	Ask good questions and identify the purpose. That is, why is it necessary to conduct an experimental evaluation of algorithms, and what is the application of it. The specific task description is described in [2].	
2	**Must** mandatory task	Prestudy [8]	Demonstrate an understanding about the task and field of study. Also, identify which parameters to use, how these can depend on and vary from each other	
3	**Must** mandatory task	Methods [8]	Identify which scientific method(s) to use. For example, quantitative, qualitative, deductive and/or inductive method	
4	**Must** mandatory task	Experimental Plan [8]	Identify experiments to conduct. Describe why these are relevant. Examine metrics to measure and elaborate on why these are relevant to this study.	

5	**Must** mandatory task	Experimental Setup [8]	Describe in details the resources needed. Identify and examine the influence of error sources, both external and internal ones. Set up an implementation, verification and validation plan.	
5	**Must** mandatory task	Goals [8]	Demonstrate how to determine when a goal is reached and an experiment is completed.	
6	**Must** mandatory task	Communication Part I [8]	Communicate and summarize the discussions of requirement 1-5 (Appendix A) in a document and hand it in to the mentor/supervisor	
7	**Must** mandatory task	Experimental Execution [8, 2]	Follow the discussed requirements 1-5, or only 6, to conduct the experiments on a real system. And, regularly document the findings with honesty and transparency.	
8	**Must** mandatory task	Communication Part II [9]	Summarize the entire work and its requirements, that is 1-7, in a smaller version of a technical report. Use the IMRAD-structure (Introduction, Method, Results, and Discussion). It is a common communication structure in scientific reports.	

Appendix B

Results – Singly Linked List

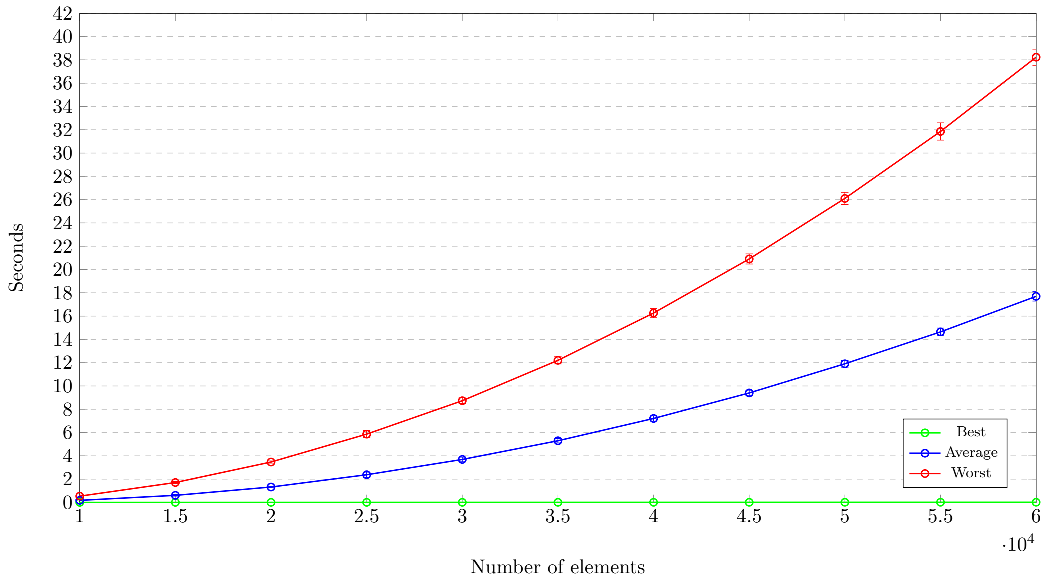

Figure B.1. Running Time Enqueue Results – Singly Linked List

Table B.1. Best Case Enqueue – Singly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.60	0.40
15000	3.97	1.48
20000	5.25	1.85
25000	6.48	1.09
30000	8.00	2.32
35000	9.09	1.49
40000	10.12	1.03
45000	11.54	2.61
50000	13.02	4.07
55000	14.24	2.28
60000	15.42	2.33

Table B.2. Average Case Enqueue – Singly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	178.07	27.19
15000	608.04	65.45
20000	1323.36	58.71
25000	2377.35	226.27
30000	3690.39	198.54
35000	5295.07	187.06
40000	7210.06	227.22
45000	9401.20	239.48
50000	11905.78	289.38
55000	14643.43	329.18
60000	17695.48	402.49

Table B.3. Worst Case Enqueue – Singly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	538.40	98.16
15000	1709.69	96.15
20000	3469.97	90.95
25000	5865.22	303.27
30000	8733.55	243.98
35000	12200.96	307.68
40000	16262.86	402.18
45000	20906.17	432.43
50000	26101.44	529.77
55000	31852.84	739.59
60000	38230.61	694.06

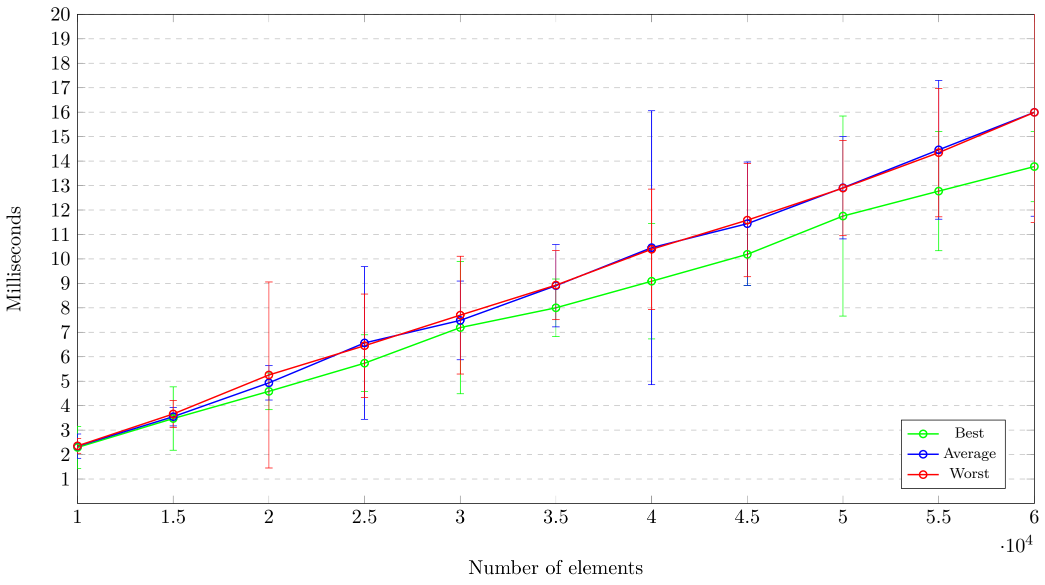

Figure B.2. Running Time Dequeue Results – Singly Linked List

Table B.4. Best Case Dequeue – Singly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.29	0.86
15000	3.47	1.30
20000	4.59	0.75
25000	5.74	1.16
30000	7.19	2.71
35000	8.00	1.18
40000	9.09	2.36
45000	10.19	1.28
50000	11.75	4.09
55000	12.77	2.44
60000	13.78	1.44

Table B.5. Average Case Dequeue – Singly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.34	0.50
15000	3.55	0.38
20000	4.93	0.70
25000	6.56	3.12
30000	7.49	1.61
35000	8.91	1.68
40000	10.46	5.60
45000	11.44	2.52
50000	12.91	2.09
55000	14.46	2.84
60000	16.00	4.25

Table B.6. Worst Case Dequeue – Singly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.36	0.30
15000	3.66	0.55
20000	5.25	3.80
25000	6.45	2.11
30000	7.70	2.41
35000	8.93	1.41
40000	10.39	2.46
45000	11.58	2.31
50000	12.90	1.95
55000	14.34	2.62
60000	15.99	4.50

Appendix C

Results – Doubly Linked List

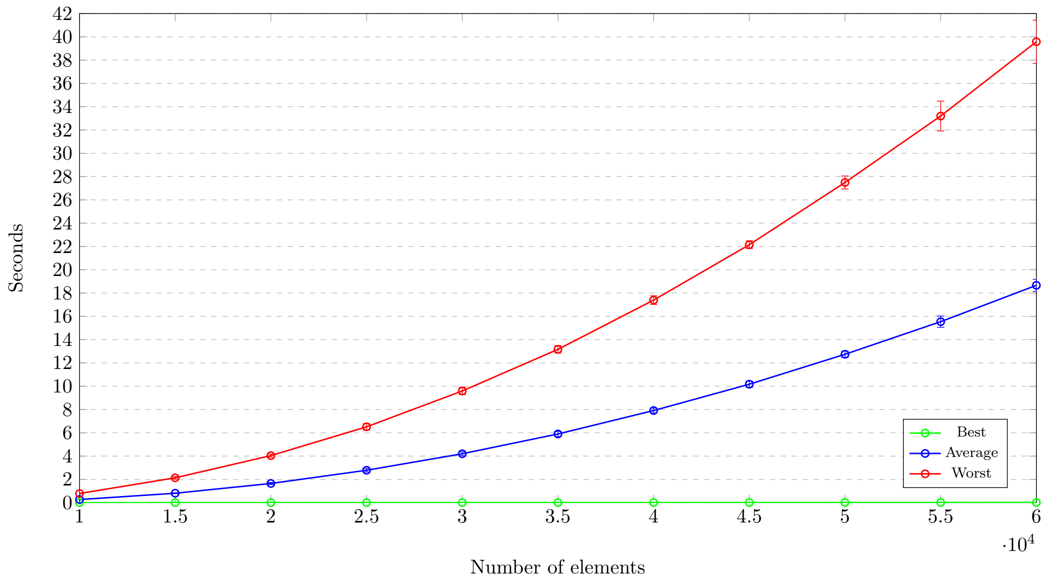

Figure C.1. Running Time Enqueue Results – Doubly Linked List

Table C.1. Best Case Enqueue – Doubly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.70	0.48
15000	4.10	0.64
20000	5.48	0.86
25000	7.01	2.17
30000	8.26	2.31
35000	9.50	1.63
40000	10.83	1.70
45000	12.26	2.64
50000	13.40	1.78
55000	14.80	2.75
60000	16.11	2.08

Table C.2. Average Case Enqueue – Doubly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	267.72	29.23
15000	810.01	37.10
20000	1649.22	93.75
25000	2783.48	151.00
30000	4200.80	165.67
35000	5899.17	216.82
40000	7911.52	206.70
45000	10172.26	269.84
50000	12741.62	278.69
55000	15543.18	483.50
60000	18662.72	516.74

Table C.3. Worst Case Enqueue – Doubly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	787.97	34.65
15000	2135.36	122.84
20000	4041.46	137.08
25000	6513.78	279.49
30000	9592.81	315.90
35000	13167.27	319.54
40000	17394.38	370.42
45000	22152.37	336.91
50000	27494.68	557.58
55000	33200.68	1280.28
60000	39580.54	1850.11

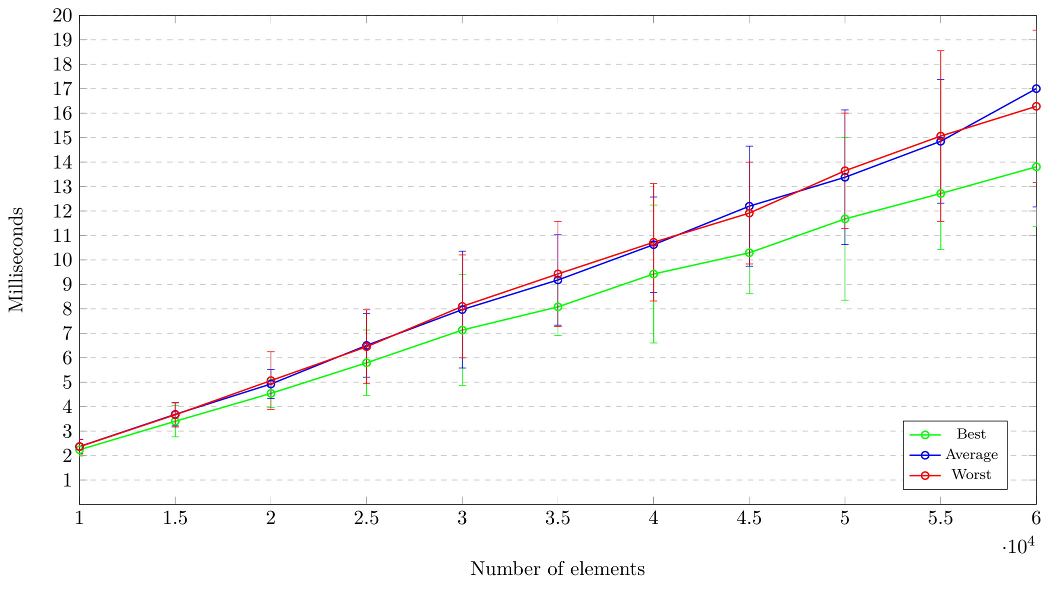

Figure C.2. Running Time Dequeue Results – Doubly Linked List

Table C.4. Best Case Dequeue – Doubly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.24	0.26
15000	3.40	0.64
20000	4.54	0.58
25000	5.79	1.34
30000	7.13	2.26
35000	8.08	1.17
40000	9.42	2.82
45000	10.29	1.67
50000	11.68	3.33
55000	12.72	2.30
60000	13.81	2.44

Table C.5. Average Case Dequeue – Doubly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.37	0.29
15000	3.68	0.47
20000	4.93	0.60
25000	6.50	1.30
30000	7.97	2.39
35000	9.18	1.85
40000	10.62	1.95
45000	12.20	2.45
50000	13.38	2.75
55000	14.85	2.53
60000	17.00	4.84

Table C.6. Worst Case Dequeue – Doubly Linked List

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.36	0.30
15000	3.67	0.50
20000	5.07	1.18
25000	6.45	1.51
30000	8.10	2.10
35000	9.43	2.15
40000	10.72	2.40
45000	11.92	2.08
50000	13.65	2.36
55000	15.06	3.49
60000	16.28	3.11

Appendix D

Results – Doubly Linked List with Modification

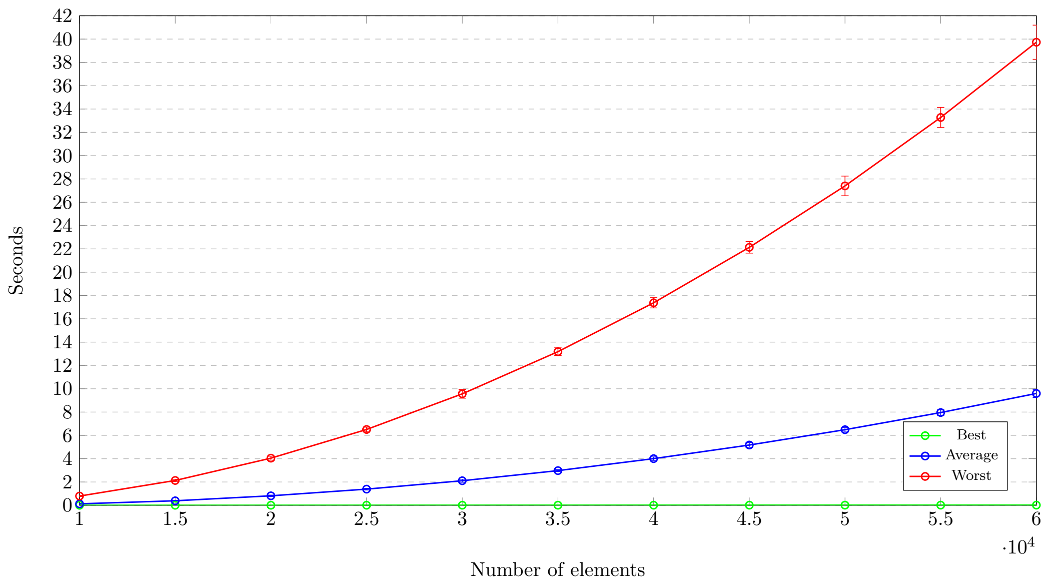

Figure D.1. Running Time Enqueue Results – Doubly Linked List with Modification

Table D.1. Best Case Enqueue – Doubly Linked List with Modification

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.62	0.31
15000	4.02	0.61
20000	5.24	0.54
25000	6.85	2.09
30000	8.19	2.10
35000	9.39	1.39
40000	10.62	1.36
45000	11.84	1.22
50000	13.38	1.76
55000	14.63	1.84
60000	15.89	2.02

Table D.2. Average Case Enqueue – Doubly Linked List with Modification

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	126.48	11.93
15000	388.12	22.12
20000	814.89	62.11
25000	1387.90	74.87
30000	2108.40	129.26
35000	2975.01	84.75
40000	4004.36	150.75
45000	5171.58	194.43
50000	6487.11	194.45
55000	7956.94	236.12
60000	9595.07	351.58

Table D.3. Worst Case Enqueue – Doubly Linked List with Modification

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	791.83	45.17
15000	2130.74	99.20
20000	4044.52	170.50
25000	6504.99	201.04
30000	9562.54	370.74
35000	13180.54	332.15
40000	17374.21	440.23
45000	22131.78	495.07
50000	27402.41	845.51
55000	33273.48	866.20
60000	39729.54	1465.50

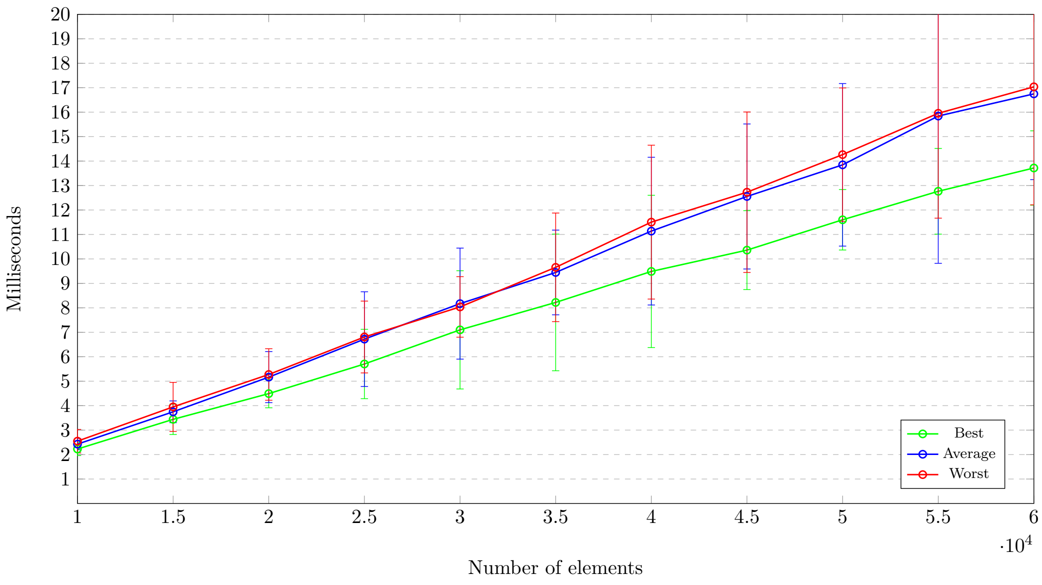

Figure D.2. Running Time Dequeue Results – Doubly Linked List with Modification

Table D.4. Best Case Dequeue – Doubly Linked List with Modification

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.23	0.27
15000	3.44	0.62
20000	4.49	0.58
25000	5.70	1.42
30000	7.10	2.42
35000	8.22	2.79
40000	9.49	3.11
45000	10.36	1.61
50000	11.60	1.24
55000	12.76	1.75
60000	13.72	1.52

Table D.5. Average Case Dequeue – Doubly Linked List with Modification

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.43	0.23
15000	3.75	0.45
20000	5.17	1.04
25000	6.72	1.94
30000	8.17	2.27
35000	9.45	1.73
40000	11.14	3.02
45000	12.55	2.96
50000	13.85	3.32
55000	15.84	6.02
60000	16.75	3.51

Table D.6. Worst Case Dequeue – Doubly Linked List with Modification

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.55	0.48
15000	3.95	1.00
20000	5.27	1.05
25000	6.81	1.47
30000	8.04	1.24
35000	9.65	2.22
40000	11.50	3.15
45000	12.73	3.28
50000	14.26	2.73
55000	15.95	4.29
60000	17.04	4.83

Appendix E

Results – Scheduling Queue

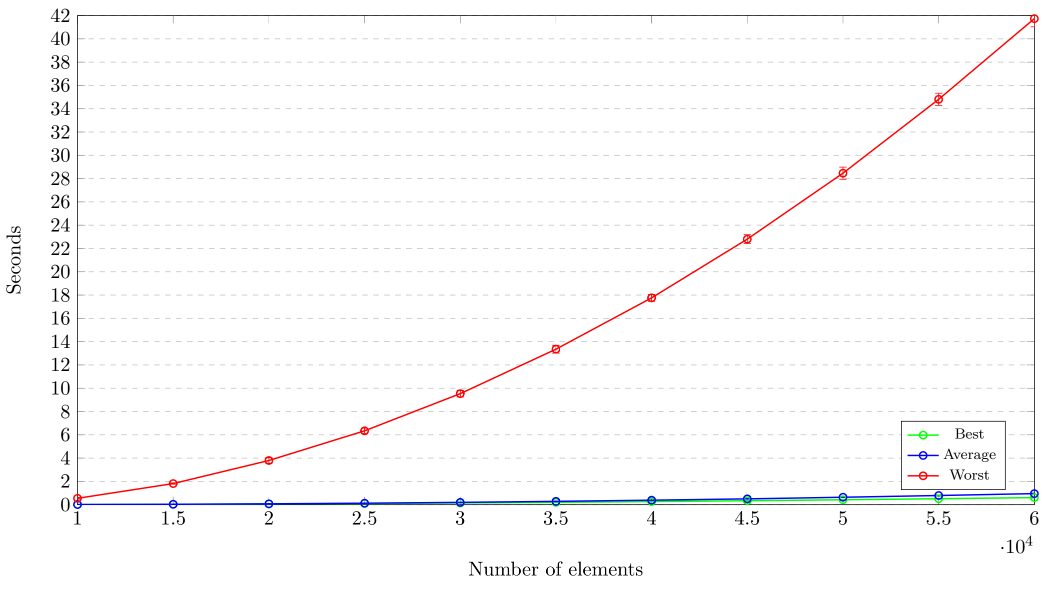

Figure E.1. Running Time Enqueue Results – Scheduling Queue

Table E.1. Best Case Enqueue – Scheduling Queue

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	10.92	1.95
15000	30.61	2.96
20000	60.23	6.53
25000	98.91	7.33
30000	146.22	20.04
35000	201.63	13.94
40000	266.62	17.70
45000	339.52	16.89
50000	421.26	24.07
55000	509.77	32.80
60000	608.66	53.49

Table E.2. Average Case Enqueue – Scheduling Queue

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	12.40	1.97
15000	34.48	2.76
20000	74.08	11.27
25000	126.81	8.90
30000	197.79	14.38
35000	285.06	21.35
40000	386.35	27.96
45000	498.61	23.97
50000	638.09	31.99
55000	786.87	49.92
60000	950.29	45.12

Table E.3. Worst Case Enqueue – Scheduling Queue

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	546.77	79.56
15000	1811.92	56.74
20000	3794.50	233.30
25000	6341.52	226.40
30000	9534.01	286.72
35000	13350.57	338.25
40000	17760.33	301.03
45000	22804.64	378.39
50000	28464.20	522.29
55000	34806.69	525.26
60000	41735.61	702.25

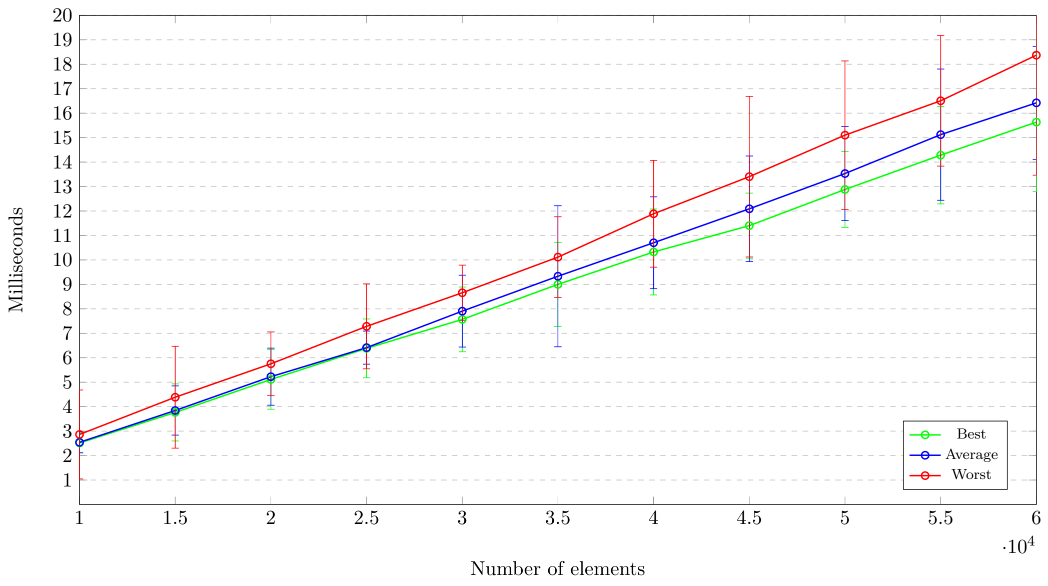

Figure E.2. Running Time Dequeue Results – Scheduling Queue

Table E.4. Best Case Dequeue – Scheduling Queue

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.51	0.41
15000	3.76	1.17
20000	5.11	1.22
25000	6.38	1.20
30000	7.57	1.32
35000	9.00	1.72
40000	10.33	1.76
45000	11.40	1.33
50000	12.88	1.56
55000	14.28	1.99
60000	15.63	2.84

Table E.5. Average Case Dequeue – Scheduling Queue

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.54	0.43
15000	3.84	1.01
20000	5.23	1.17
25000	6.41	0.68
30000	7.91	1.47
35000	9.33	2.88
40000	10.70	1.88
45000	12.09	2.16
50000	13.53	1.92
55000	15.12	2.68
60000	16.42	2.31

Table E.6. Worst Case Dequeue – Scheduling Queue

Number of Elements	Running Time in Milliseconds	2×SD ($N = 50$)
10000	2.86	1.82
15000	4.39	2.08
20000	5.75	1.30
25000	7.28	1.74
30000	8.66	1.13
35000	10.11	1.65
40000	11.89	2.18
45000	13.40	3.28
50000	15.10	3.04
55000	16.51	2.67
60000	18.37	4.91

Appendix F

Results – Memory Usage

The best, worst, average case tests are included in the total memory usage for each implementation. Thus, the higher memory usage.

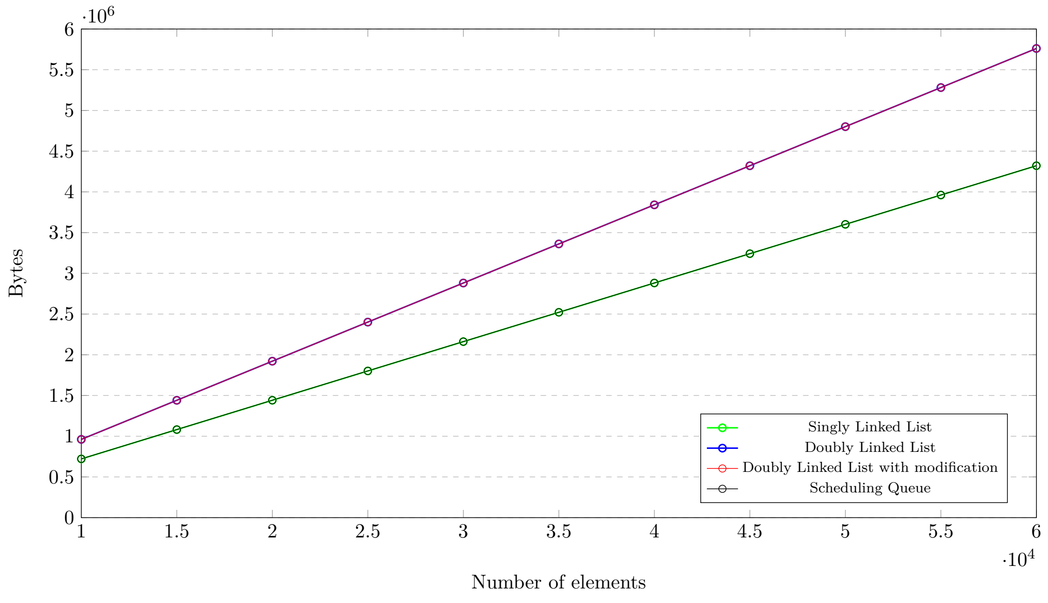

Figure F.1. Memory Usage Results of the All Queues

Table F.1. Memory Usage – Singly Linked List

Number of Elements	Memory Usage in Megabytes
10000	0.72
15000	1.08
20000	1.44
25000	1.80
30000	2.16
35000	2.52
40000	2.88
45000	3.24
50000	3.60
55000	3.96
60000	4.32

Table F.2. Memory Usage – Doubly Linked List

Number of Elements	Memory Usage in Megabytes
10000	0.96
15000	1.44
20000	1.92
25000	2.40
30000	2.88
35000	3.36
40000	3.84
45000	4.32
50000	4.80
55000	5.28
60000	5.76

Table F.3. Memory Usage – Doubly Linked List with Modification

Number of Elements	Memory Usage in Megabytes
10000	0.96
15000	1.44
20000	1.92
25000	2.40
30000	2.88
35000	3.36
40000	3.84
45000	4.32
50000	4.80
55000	5.28
60000	5.76

Table F.4. Memory Usage – Scheduling Queue

Number of Elements	Memory Usage in Megabytes
10000	0.72
15000	1.08
20000	1.44
25000	1.80
30000	2.16
35000	2.52
40000	2.88
45000	3.24
50000	3.60
55000	3.96
60000	4.32

Bibliography

[1] P. Deshpande and O. Kakde, *C & Data Structures*, ser. Charles River Media Computer Engineering. Charles River Media, 2004.

[2] R. Rönngren. Uppgift läsåret 15/16 | ingenjörskunskap och ingenjörsrollen ICT (II1304) | KTH. [Online]. Available: https://www.kth.se/social/course/II1304/page/uppgift-lasaret-1516/ [Accessed: 2015-12-08]

[3] R. Rönngren. Om experimentell och vetenskaplig metodik.pdf. [Online]. Available: https://www.kth.se/social/files/562ce17af2765431e02db7c3/Om%20experimentell%20och%20vetenskaplig%20metodik.pdf [Accessed: 2015-11-15]

[4] W. M. Trochim. (2006) Deduction and induction. [Online]. Available: http://www.socialresearchmethods.net/kb/dedind.php [Accessed: 2015-12-08]

[5] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects," in *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July*. CSREA Press USA, 2013, pp. 67–73.

[6] S. Merriam, *Qualitative Research: A Guide to Design and Implementation*, ser. Jossey-Bass higher and adult education series. John Wiley & Sons, 2009.

[7] R. Rönngren. Enkel kravanalys/kravsammanställning. [Online]. Available: https://www.kth.se/social/course/II1304/page/kravanalys/ [Accessed: 2015-12-08]

[8] R. Rönngren. Å3 experimentell metod – planering. [Online]. Available: https://kth.instructure.com/courses/2502/assignments/11804?module_item_id=33137 [Accessed: 2017-11-29]

[9] R. Rönngren. Å3 experimentell metod – rapport. [Online]. Available: https://kth.instructure.com/courses/2502/assignments/11805?module_item_id=33138 [Accessed: 2017-11-29]