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

A researcher plan in advance to conduct a successful and credible scientific work. However, students and engineers occasionally approach the planning phase in a careless manner. As a result, researchers waste time to redo the changes due to poor design and execution. Students and engineers lack a deeper knowledge of the scientific method to contribute with quality work. That is, science emphasizes the importance of theoretical knowledge and its application in practice. Thus, in our study, we used the scientific method to evaluate algorithms with a rigorous, honest and transparent method.

1.1 Background

A data structure represent a model to store data with a methodical approach [1]. Software engineers use queues in software development to manage arbitrary entities in a first-in-first-out (FIFO) order. For example, developers use a queue to develop a memory buffer or scheduling in an operating system.

Higher level programming languages often provide a standardized library with finished implementations, such as a queue. Software developers use these implementations frequently, and sometimes take the underlying architecture for granted. Thus, an inexperienced developer possess a limited knowledge of the theory and technical aspects of the used implementations.

Developers use the data structure, or model, linked list to store data linearly. The simple linked list consists of entities. An entity stores data and includes a connection to the next entity in the list. Computer scientist and developers refer the entity as a node or an element, and the connection as a pointer. That is, an element stores data and consists of a pointer to the next element in the list. Thus, developers frequently use linked list implementations to store and manage data linearly, for example, in a queue implementation.

1.2 Problem

An engineer must learn the skill to apply the scientific method in practice. That is, to later on possess the ability to contribute with further development and exploration within the scientific community. Also, a software engineer needs to grasp the basics of features in an API[1]-library. However, developers sometimes use these library features in projects, and take these algorithms for granted to behave in a specific manner. Thus, we applied the scientific method rigorously to evaluate algorithms in depth, both in theory and practice.

Our task was to evaluate four variations of a list-based data structure to represent a queue [2]. Each element compromises of a number to represent its priority in the queue. For clarification, an element with a low number represents higher priority, because the element lies closer to the present time and therefore ready for execution. We implemented the queues in the low-level programming language `C`, and the following list presents 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 of the first and last element's priority.
    - In the head – The new element's priority is higher than the mean, that is,
      numerical value $<$ mean value.
    - In the rear – The new element's priority is lower than the mean, that is, numerical value $\geq$ mean value.
4. **Scheduling queue** – The queue consists of an FIFO-queue for each priority in the $[0, 40]$ interval.

Computer scientists evaluate algorithms in terms of space and time, such as execution time and memory usage. That is, we 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 either enqueues or dequeues elements with an FIFO policy. Because of the FIFO policy, we assumed with a correct implementation and theory, the dequeue-operation take $\mathcal{O}(1)$ time. However, the enqueue-operation becomes more sensitive to an element's priority, because the algorithm insert each element in the correct spot in the queue. Thus, the implementations perform differently because of priority distribution. For example, the implementation reaches $\mathcal{O}(n)$ time if the algorithm traverses through the entire list to enqueue an element.

The following list describes our hypotheses:

- 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 was to raise awareness of the features in API-libraries with the scientific method, which developers and engineers often take for granted. Thus, we conducted experiments to achieve a deeper knowledge to apply the scientific method, and to evaluate four versions of a queue implementation. Consequently, we wanted to contribute with a study students and engineers can learn from and further develop.

1.5 Goal

The bigger goal was to attain deeper knowledge of the scientific method, and apply the method 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.
- Grasp the importance of scientific method.
- Conduct a reproducible study.
- Attain the ability to present, analyze and draw conclusions from experiments, both in theory and practice.
- Implement queues correctly, that is, verify its behavior and validate the data.
- Create tests to generate the execution time and memory usage from the algorithms.

1.6 Social Benefit, Ethics and Sustainable Development

Our study influence students and engineers to apply the scientific method more with a more serious approach. Also, the study facilitate the process for software engineers to choose an implementation of a queue, which fits their specific purpose. Finally, we delve into the foundations of algorithms, data structure and complexity theory. Thus, our work help students and engineers to acquire a deeper comprehension of computer science.

Within science, a researcher undertake a moral responsibility to conduct a reproducible work with honesty and transparency. However, sometimes scientists encounter difficult circumstances, which limits their ability to produce a reproducible work. For example, due to economical and practical reasons, such as expensive hardware. We documented each step and presented findings with rigor, honesty and transparency. That is, we encourage students to reproduce our findings.

We conducted the experiment on a virtual machine on Microsoft's cloud platform. Thus, we used the exact quantity of resources needed to conduct the experiments. That is, we avoided to 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 to reach practical conclusions from observations or data [3]. That is, in sequential order to 1) collect data, 2) detect pattern in the data, 3) set up a hypothesis and 4) explain the findings with established theory to reach 5) a conclusion [4]. The method is sometimes informally called a bottom-up approach due to its style to first observe data. Software engineers frequently apply an inductive method to observe algorithms and a system's behavior. For example, a developer put in a specific input in the implementation and generate an output to observe the algorithm's performance.

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

2.2 Quantitative and Qualitative Method

Researches apply a quantitative method in studies, which rely on measurable data in large quantities to explain findings [5]. That is, a quantitative study use measurable metrics, such as time, temperature and age. These studies apply statistical analysis to summarize the data in tables and graphs with error margins. For example, scientists use a quantitative method 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 a qualitative method in studies when the data is significantly more difficult to quantify, such as opinions, thoughts and happiness. The researchers conduct, for example, interviews and surveys, to collect data from individuals. Thus, psychologists often implement a qualitative method to examine people's behavior.

2.3 Data Collection

We used quantifiable metrics in our study. That is, execution time and memory usage, because computer scientists commonly apply these for algorithm och complexity analysis. In contrast to, for example, interviews and surveys, which the researchers within social sciences regularly use in qualitative studies [3, 5]. That is, qualitative studies reaches conclusions from the quality of the data. Thus, our study uses an experimental quantitative method to reach a deeper insight into data structures and queue implementations.

For each of the four queue-implementations, we tested and generated the execution time for the best, average and worst case. That is, we created input for the queues with varying distributions of the element's priorities. Besides execution time, we log the memory usage for each implementation. Finally, we wrote scripts to execute the tests and logged 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 analyzed the algorithms with help of established complexity theory. For example, to determine an algorithm's performance based solely on theoretical knowledge. However, the usage of only a deductive methods can yield unexpected results when put into practice. Thus, we generate realistic data from our experiments, and use an inductive method to further analyze the algorithms. That is, to explore if we can notice observable patters in the data with help of complexity models.

We used both graphs and tables to visualize the execution time and memory usage. And, ran the experiments multiple times to generate the sample mean and standard deviation. The standard deviation adds a deeper insight into the data and spread around the mean. Also, it help us to identify the external or internal error sources. If the standard deviation is relatively high, then we can suspect that something is wrong with, for example, the underlying system.

Chapter 3

System Description

3.1 System Development Method

Our implementation produces an executable for each of the four queue-implementations. The experiment consists of primarily three files, `main.c` with the tests, `queues.h` and `queues.c` which respectively describes and implements the functionalities of the queue. Finally, we used 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 developer a Python-script, which 1) calculated the sample mean and standard deviation, and 2) summarized and visualized the data with tables and graphs in LaTeX. Finally, we used Valgrind on our implementations to examine memory allocation and deallocation.

We added an optional solution to run the entire experiment in a container. That is, if anyone wants to run the entire experiment, and spend minimum time on the setup process, then the user only has to install Docker and execute a command to achieve the similar results we produced. Docker containers introduce an effective approach to package software, because containers use 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 priority queue.
- **Dequeue** – dequeues the element with the highest priority, that is, the first element in the queue.

The queues approach to enqueue varies for the implementations, because the algorithm traverses and inserts the new element in the correct position in the queue. Each element stores at least 1) a unique number identifier of the data type integer, 2) a priority number of the data type double and 3) a pointer to refer to additional elements.

The following sections describe our implementation of the queues. Additionally, we analyzed the time complexity for the enqueue-operation. We found the average case time complexity more difficult to theoretically analyze compared to the best and worst case. Thus, we examined the best and worst cases in theory to interpret the average case for each implementation's enqueue-operation.

3.2.1 Singly Linked List

This queue 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. The algorithm enqueues new elements from head. The following list analyzes the enqueue-operation's time complexity:

- **Best Case** $\mathcal{O}(1)$ – The best case occurs when the enqueue-operation of new elements always takes place in the head. That is, the algorithm avoids to traverse the list to find the correct spot to insert the new element. On the other hand, the queue 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** $\mathcal{O}(n)$ – The worst case happens when the implementation 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 priority distribution of the same numerical value, because the queue uses FIFO policy to manage elements of the same priority.

3.2.2 Doubly Linked List

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

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

- **Worst Case** $\mathcal{O}(n)$ – The queue reaches the worst case when the priority's numerical value constantly descends from a higher one. Thus, the algorithm traverses the list from the rear n-times to insert the new element correctly.

3.2.3 Doubly Linked List with Modification

This queue is similar to the previous section's implementation. That is, an element stores the same properties as the last one. However, for this algorithm, the insertion of new elements takes place in either the head or rear. The enqueue-operation depends on the average of the first and last element in the list. The following list demonstrates the enqueue 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 analyzed 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 take place in either head or rear. For example, let us say that we create a distribution of the same priority. The average will always be the same as the new element's priority. Thus, the insertion for each of new elements takes place in the rear.
- **Worst Case** $\mathcal{O}(n)$ – To achieve the worst case, we must traverse through the list for each insertion of a new element. It is achievable if the queue, for example, first enqueues two elements with significantly differing 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 for each new element, it receives a lower priority than the previous one, then the insertion takes place in the head and traverses $(n-1)$-times to insert new element correctly in the queue.

3.2.4 Scheduling Queue

This queue uses a static list, which consists of 41 pointers. These entries represent the priority in the [0, 40] interval, and each pointer connects to an FIFO-queue. A singly linked list represent each of the FIFO-queues, as described in Section 3.2.1. The following list presents the time complexity for the best and worst case:

- **Best Case** $\mathcal{O}(n)$ – The algorithm achieve the best time complexity if we distrubute the elements equally in the queue. The simplest distribution enqueues

elements from highest to the lowest priority and resets from lowest. That is, from priority 0 to 40, and then start over again 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 algorithm achieves the same worst case the singly linked list, which we described in Section 3.2.1. That is, the implementation traverse n-times for a priority distribution of the same priority.

3.3 Experimental Setup

We conducted the experiments on a Microsoft Azure virtual private server (VPS). We used a VPS service because we gain full access over the machine. 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 important to take into account as these can influence the results of the study. That is, if we identify error sources early in the planning phase, we can then mitigate unexpected behaviors and results. Software and computational research depend heavily on the underlying system, both hardware and software. Thus, we run the experiments multiple times and calculate statistical features, such as mean and standard deviation for the sample. Standard deviation is relevant, because we can then identify how the execution time vary around the mean. A large standard deviation indicates, for example, that another process consume significant quantity of resources.

Another error source comes from poor design and implementation of models. Thus, we must grasp and follow scientific method rigorously, verify and validate the implementations. For example, a developer wastes time to debug an implementation due to poor design. And, especially in a relative low-level programming language, such as C.

Table 3.1. Experiment parameters for each implementation and test case

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

3.3.2 Verification and Validation

We developed a test case to verify the implementation' correctness, specifically the dequeue-operation to insert new elements at the correct position in the queue. Our Python script generates an input- and output-file. The former consists of elements of randomized priorities, and the output file contains the elements in correct order. Finally, we used the input and output files in our implementations to verify its correctness. We used Valgrind and verified the memory allocation and deallocation of our implementations.

We developed tests for the best, average and worst cases to validate its behavior. That is, we analyzed the running time and complexity theoretically. And, used statistical analysis and complexity theory to validate the algorithms. Finally, we examined the edge cases, for example, an empty or a list containing only one element.

3.3.3 Experiments

To test the best, average and worst case, we created a variety of priority distributions as input based on our complexity analysis of our algorithms. The analysis takes place in Section 3.2. We did not analyze the time complexity for the average case theoretically. However, we assumed the average case somewhere lies between the best and worst case. Thus, We randomly assigned priorities in $[0, 40]$ interval to the elements in the average case experiments. 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, p and n represent a set of elements, a numerical priority and the number of elements, respectively: illustrates

- **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 elements 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,

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

  - 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 help of 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

Our running time results for the enqueue-operation shows 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. Each implementation reaches similar time complexity in the best 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 the average case tests. That is, around 10 seconds and 1 second, respectively, for 60 000 elements. The average case tests for the scheduling queue yielded running time that were only milliseconds higher than for its best case. Whereas, the rest of implementations' average case tests produced running times that were several seconds higher than their best case. That is, at least for more than 25 000 elements.

4.1.2 Dequeue

There were no significant changes in the results for the dequeue-operation experiments, because the dequeue operation is similar for each implementation. 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 barely differ in milliseconds.

4.2 Memory Usage

Appendix F summarizes the memory usage of the implementations for the best, average and worst case tests. The memory usage was the same across implementations, because the algorithms allocate the same number of elements for each test. Thus, we did not have to calculate the mean and standard deviation for memory usage.

The implementations' memory usage increase linearly and in relation to 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 latter 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 time 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 and memory usage.

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, generated data and math notation to express the average running time, where n represents number of elements:

- **Singly Linked List and Doubly Linked List** – In the worst case, these algorithms must traverse n times, and in the best case $\mathcal{O}(1)$. Thus, we assume in average that these algorithms must at most traverse $n/2$ times,
- **Doubly Linked List with Modification** – In the worst case, the algorithm 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 the worst case. However, due to its 41 FIFO queues, each separate queue is in general shorter than the previous mentioned queues. Thus, we assume in average that this algorithm must at most traverse $n/41$ times.

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:

$$\left[\begin{array}{lll} (A) \quad \text{Singly Linked List} & = & \frac{n}{2} \\ (B) \quad \text{Doubly Linked List} & = & \frac{n}{2} \\ (C) \quad \text{Doubly Linked List with Modification} & = & \frac{n}{4} \\ (D) \quad \text{Scheduling Queue} & = & \frac{n}{41} \end{array}\right] \implies \frac{n}{2} \geq \frac{n}{4} \geq \frac{n}{41}$$

In average, we noticed that time complexity of (D) is approximately $10\times$ higher than (C), and $20\times$ higher than (A) and (B). Also, the time complexity of (A) and (B) is roughly $2\times$ higher than (C). 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:

$$\left[\begin{array}{llll} (A) & 17695.48 \pm 2 \times 201.24 \text{ ms} & \approx 20000 \text{ ms} & \approx 2 \times \text{(C) or } 20 \times \text{(D)} \\ (B) & 18662.72 \pm 2 \times 201.24 \text{ ms} & \approx 20000 \text{ ms} & \approx 2 \times \text{(C) or } 20 \times \text{(D)} \\ (C) & 9595.70 \pm 2 \times 118.06 \text{ ms} & \approx 10000 \text{ ms} & \approx 10 \times \text{(D)} \\ (D) & 950.29 \pm 2 \times 22.56 \text{ ms} & \approx 1000 \text{ ms} & \end{array}\right]$$

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 generated similar memory usage, and higher memory usage than singly linked list and scheduling queue, because these implementations store in total two pointers for each element. That is, doubly linked lists have a pointer that connects to the previous element, and the second connects to the next element. 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.

4.3.3 Choice of Methods

We used a deductive method to analyze each implementation's best and worst case. That is, we took help of established data structure and complexity theory to determine the best and worst case. And, assumed the average case's time complexity lies somewhere between the best and worst.

An inductive method helped us to validate our best and worst case analysis. Additionally, the average case experiments contributed to a better comprehension of the average time complexities of the algorithms, That is, we observed patterns in the data from the tests, which resulted in a theoretical analysis and explanation of the algorithms average time complexity, as demonstrated in Section 4.3.1. Additionally, the data showed that the average case's time complexity lies indeed between the best (lower bound limit) and worst case (upper bound limit). Thus, a quantitative experimental study with both inductive and deductive reasoning was a feasible methodology to evaluate algorithms.

Chapter 5

Conclusions and Future Work

This quantitative experimental study shows that the calculated theoretical space and time complexity of a number of list implementations hold. The study also shows that the priority queue implementation provides a time complexity better than a double linked list with a low space complexity as the singly linked list.

We applied the scientific method to conduct an experiment with honesty, transparency and rigor. The experiment helped us to verify and validate our implementations. We produced a reproducible study and reached the goals to grasp the scientific method. Finally and hopefully, our analysis and findings can help engineering students to achieve a deeper theoretical and practical comprehension of algorithms and data structures.

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

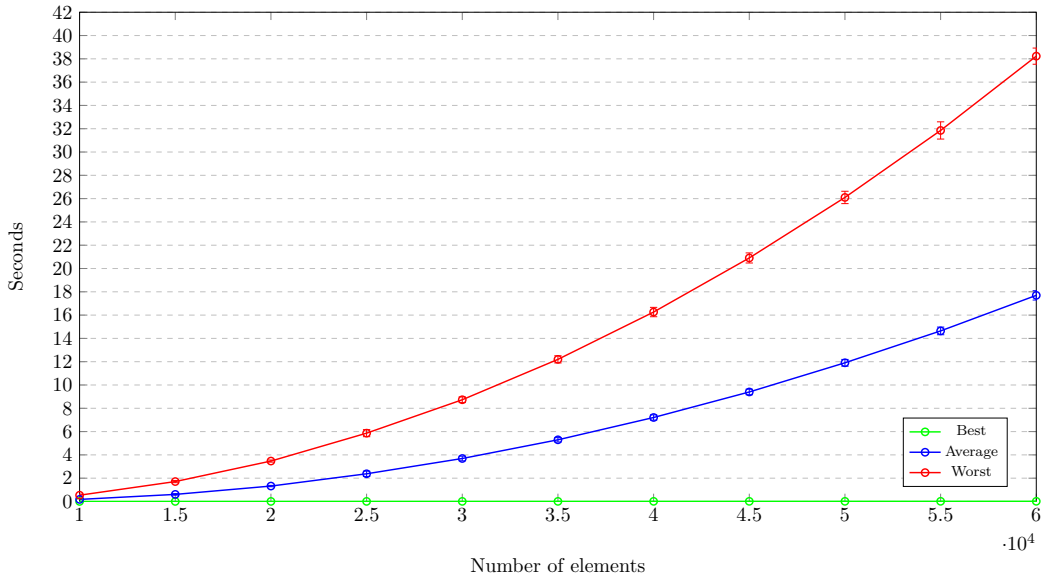

Figure B.1. Running Time Enqueue Results – Singly Linked List

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

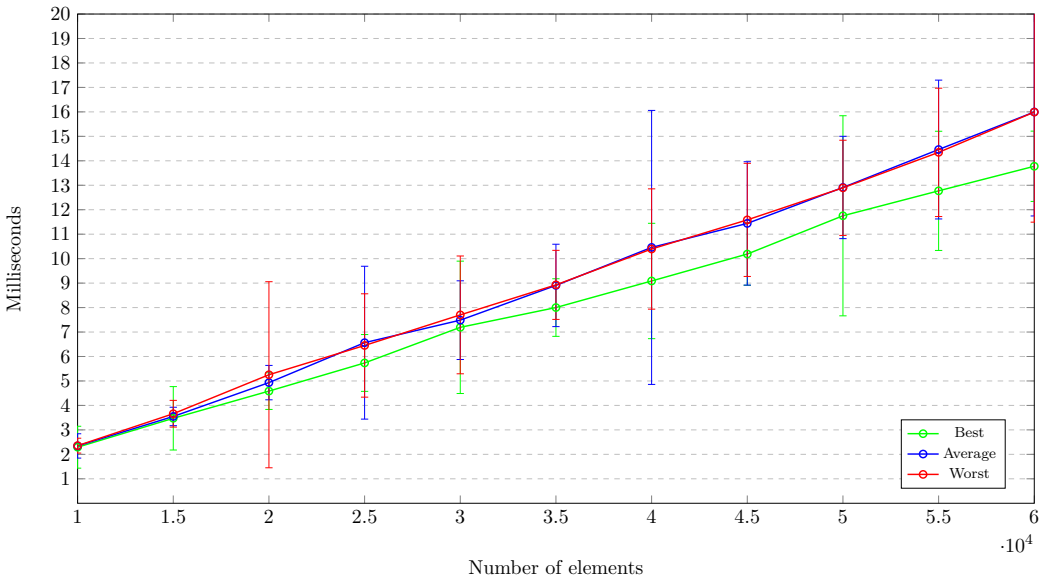

Figure B.2. Running Time Dequeue Results – Singly Linked List

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

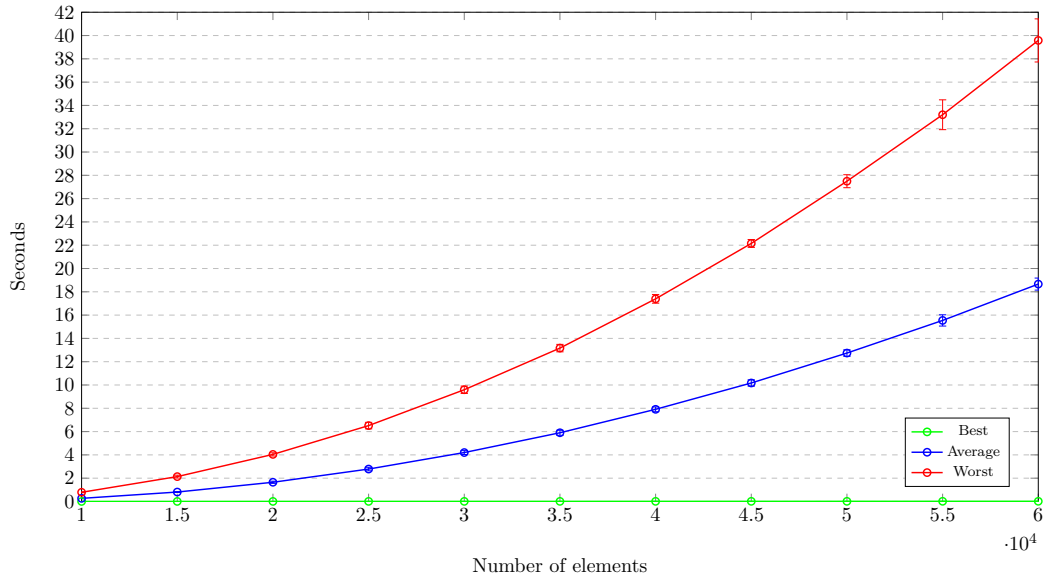

Figure C.1. Running Time Enqueue Results – Doubly Linked List

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

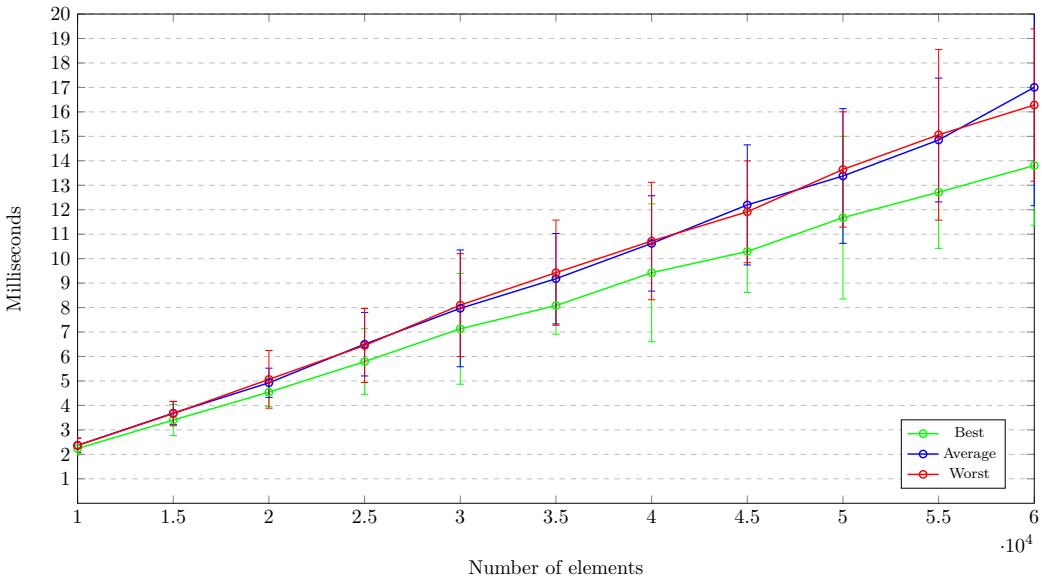

Figure C.2. Running Time Dequeue Results – Doubly Linked List

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

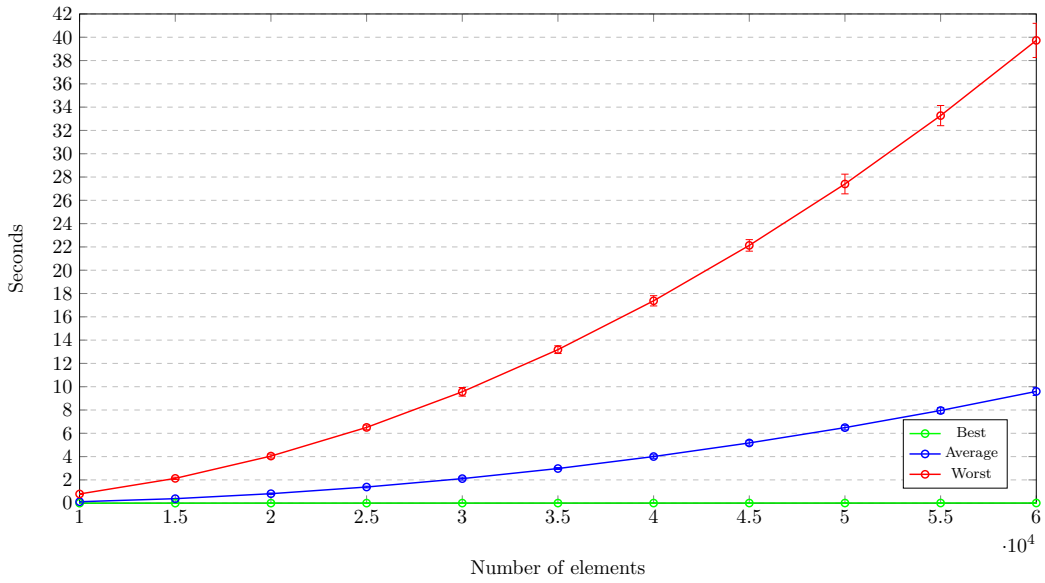

Figure D.1. Running Time Enqueue Results – Doubly Linked List with Modification

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

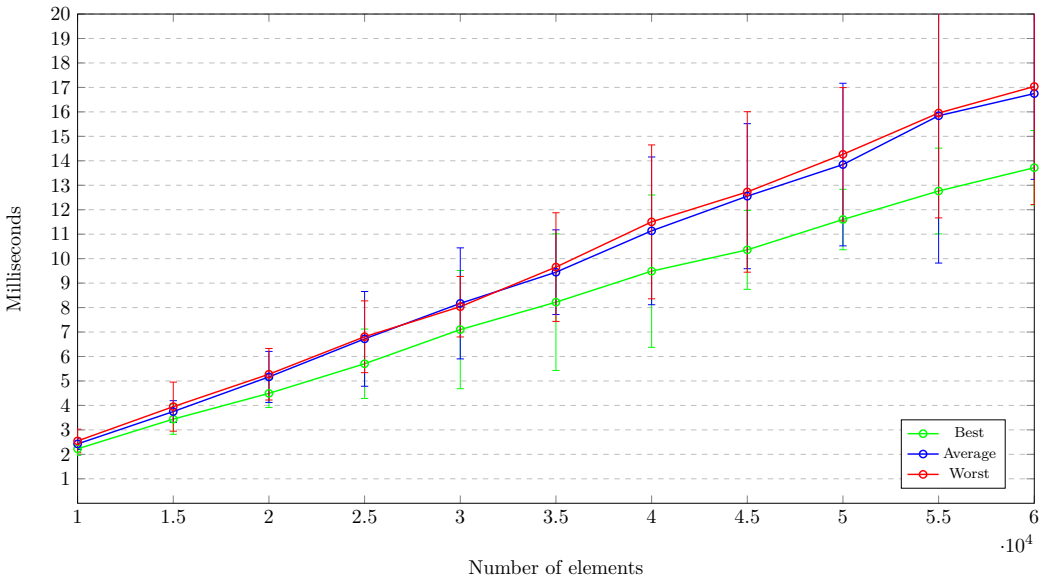

Figure D.2. Running Time Dequeue Results – Doubly Linked List with Modification

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

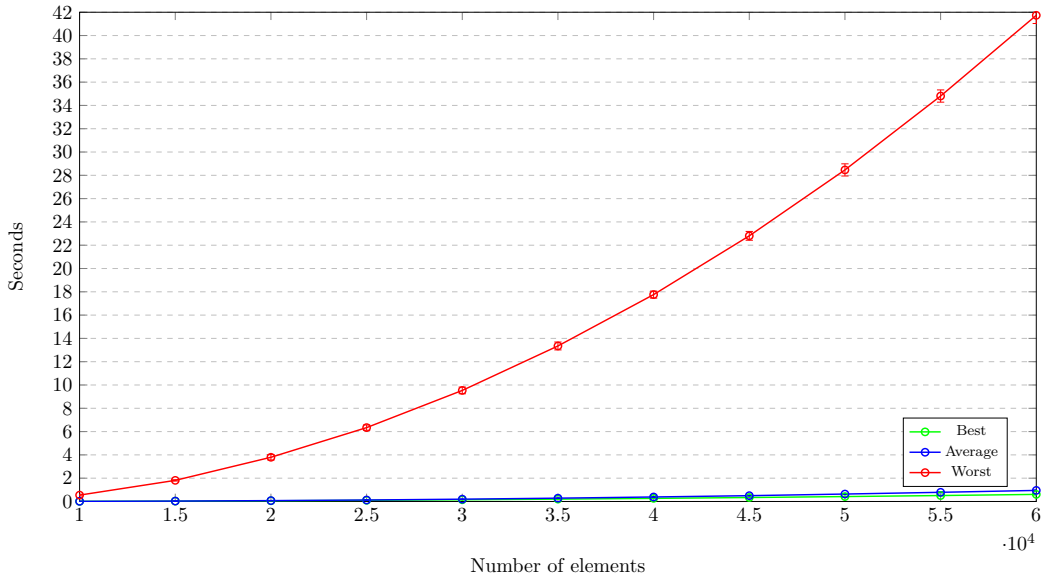

Figure E.1. Running Time Enqueue Results – Scheduling Queue

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

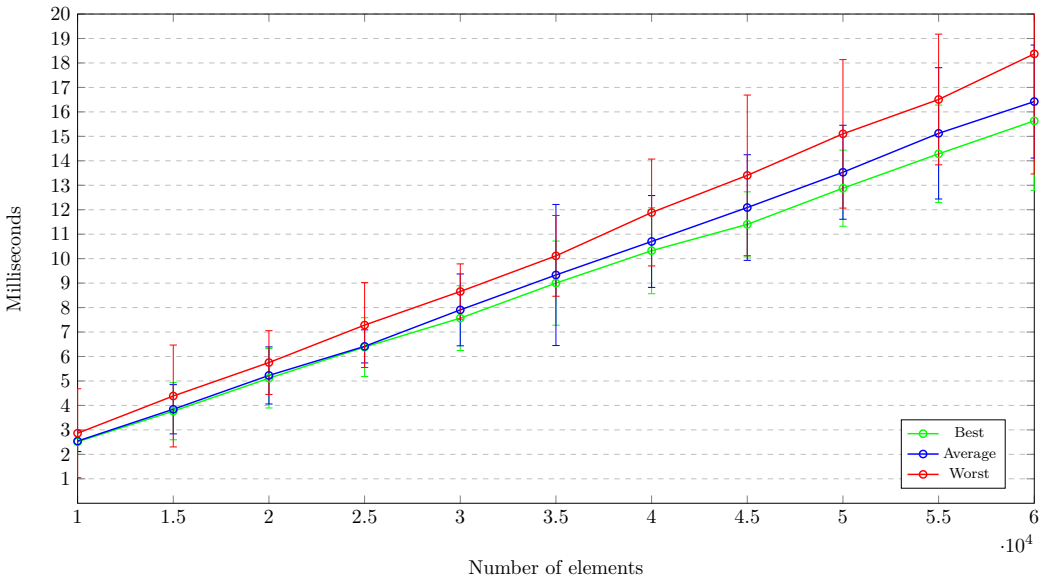

Figure E.2. Running Time Dequeue Results – Scheduling Queue

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.

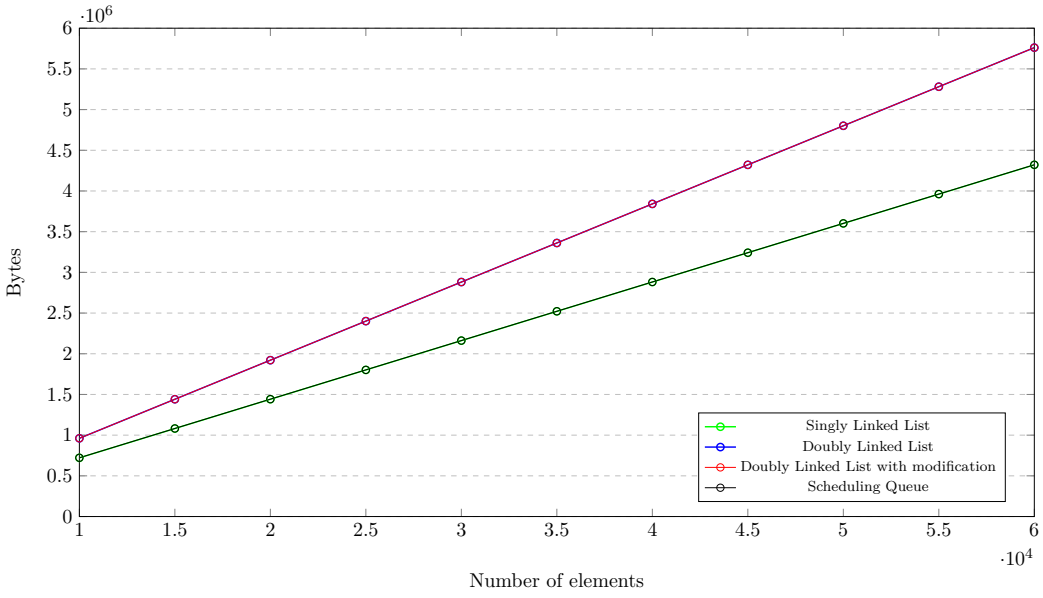

Figure F.1. Memory Usage Results of the All Queues

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

Appendix G

Code – `main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include <time.h>
#include <sys/types.h>
#include "queues.h"

/* #define TOTAL_ELEMENTS 40000 */
int TOTAL_ELEMENTS;

#if TEST_CORRECTNESS
void test_correctness(char* indata, char* outdata) {

    char * temp;

    list_t* list = new_queue();
    uint32_t pid = 0;
    double test_priority = 0;
    uint32_t test_pid = 0;

    char line[512];
    int i = 0;
    FILE* file = fopen(indata, "r");

    while (fgets(line, sizeof(line), file)) {
        temp = strtok(line,"␣");
        while (temp != NULL)
        {
            temp[strcspn(temp, "\n")] = '\0';

            if (i == 0) {
                test_pid = atoi(temp);
                i = 1;
            } else {
                test_priority = atof(temp);
                i = 0;
            }


            temp = strtok(NULL, "␣");
        }
        enqueue(list, test_pid, test_priority);
    }

    fclose(file);

    file = fopen(outdata, "r");

    while (fgets(line, sizeof(line), file)) {
```

```
        temp = strtok(line,"␣");
        while (temp != NULL)
        {
            temp[strcspn(temp, "\n")] = '\0';

            if (i == 0) {
                test_pid = atoi(temp);
                i = 1;
            } else {
                test_priority = atof(temp);
                i = 0;
            }

            temp = strtok (NULL, "␣");
        }

        pid = dequeue(list);
        printf("pid␣%u␣==␣test_pid␣%d\n", pid, test_pid);
        assert(pid == test_pid);
    }
    fclose(file);

    delete_list(list);

    printf("test␣correctness␣-␣Passed\n");
}
#else

list_t* test_best_case() {
    double test_priority;
    uint32_t test_pid;
    clock_t t;
    float time_taken = 0.0;

    list_t* list = new_queue();

#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = TOTAL_ELEMENTS - test_pid;

        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[SINGLY_LINKED_LIST␣|␣BEST␣CASE␣|␣ENQUEUE]␣%f␣seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[SINGLY_LINKED_LIST␣|␣BEST␣CASE␣|␣DEQUEUE]␣%f␣seconds\n", time_taken);
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = test_pid;


        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST␣|␣BEST␣CASE␣|␣ENQUEUE]␣%f␣seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
```

```
    printf("[DOUBLY_LINKED_LIST | BEST CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = 1;

        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST_AVG | BEST CASE | ENQUEUE] %f seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST_AVG | BEST CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == ARRAY_PRIORITY
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = test_pid % 41;

        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[ARRAY_PRIORITY | BEST CASE | ENQUEUE] %f seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[ARRAY_PRIORITY | BEST CASE | DEQUEUE] %f seconds\n", time_taken);
#endif

    return list;
}

list_t* test_average_case() {
    double test_priority;
    uint32_t test_pid;
    clock_t t;
    double time_taken = 0;
    list_t* list = new_queue();

    srand(time(NULL));
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = rand() / (RAND_MAX / 40);

        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }

#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    printf("[SINGLY_LINKED_LIST | AVERAGE CASE | ENQUEUE] %f seconds\n", time_taken);
    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[SINGLY_LINKED_LIST | AVERAGE CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST
    printf("[DOUBLY_LINKED_LIST | AVERAGE CASE | ENQUEUE] %f seconds\n", time_taken);
    time_taken = 0.0;
```

```
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST | AVERAGE CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    printf("[DOUBLY_LINKED_LIST_AVG | AVERAGE CASE | ENQUEUE] %f seconds\n", time_taken);
    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST_AVG | AVERAGE CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == ARRAY_PRIORITY
    printf("[ARRAY_PRIORITY | AVERAGE CASE | ENQUEUE] %f seconds\n", time_taken);
    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[ARRAY_PRIORITY | AVERAGE CASE | DEQUEUE] %f seconds\n", time_taken);
#endif
    return list;
}


list_t* test_worst_case() {
    double test_priority;
    uint32_t test_pid;
    clock_t t;
    float time_taken = 0.0;

    list_t* list = new_queue();

#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    /*
     * Priority increments
     */
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = test_pid;

        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[SINGLY_LINKED_LIST | WORST CASE | ENQUEUE] %f seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[SINGLY_LINKED_LIST | WORST CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST
    /*
     * Priority decrements
     */
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = TOTAL_ELEMENTS - test_pid;


        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
```

```
    }
    printf("[DOUBLY_LINKED_LIST | WORST CASE | ENQUEUE] %f seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST | WORST CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    enqueue(list, 0, TOTAL_ELEMENTS*2);
    enqueue(list, 1, 0);
    for(test_pid = 2; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = test_pid - 1;

        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST_AVG | WORST CASE | ENQUEUE] %f seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[DOUBLY_LINKED_LIST_AVG | WORST CASE | DEQUEUE] %f seconds\n", time_taken);
#elif DATA_STRUCTURE == ARRAY_PRIORITY
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        test_priority = 40;

        t = clock();
        enqueue(list, test_pid, test_priority);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[ARRAY_PRIORITY | WORST CASE | ENQUEUE] %f seconds\n", time_taken);

    time_taken = 0.0;
    for(test_pid = 0; test_pid < TOTAL_ELEMENTS; test_pid++) {
        t = clock();
        dequeue(list);
        t = clock() - t;
        time_taken += ((double)t)/CLOCKS_PER_SEC;
    }
    printf("[ARRAY_PRIORITY | WORST CASE | DEQUEUE] %f seconds\n", time_taken);
#endif

    return list;
}
#endif

int main(int argc, char **argv) {
#if TEST_CORRECTNESS
    char* indata = argv[1];
    char* outdata = argv[2];
    test_correctness(indata, outdata);
#else
    list_t* list;

    TOTAL_ELEMENTS = atoi(argv[1]);

    list = test_best_case();
    delete_list(list);

    list = test_average_case();
    delete_list(list);

    list = test_worst_case();
```

```
    delete_list(list);
#endif

    return 0;
}
```

Appendix H

Code – queue.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>
#include <sys/types.h>
#include "queues.h"

/*
 * Prints out the entire list
 */
void print_list(list_t* list) {
#if DATA_STRUCTURE == ARRAY_PRIORITY
    int i;
    node_t* current;
    for(i = 0; i <= PRIORITY_SIZE; i++) {
        current = list->priority_list[i];
        while(current != NULL) {
            printf("%d %f\n", current->pid, current->priority);
            current = current->next;
        }
    }
#else
    node_t* current = list->first;
    while(current != NULL) {
        printf("%d %f\n", current->pid, current->priority);
        current = current->next;
    }
#endif
}

list_t* new_queue() {
#if DATA_STRUCTURE == ARRAY_PRIORITY
    int i;
#endif

    list_t* list = malloc(sizeof(list_t));
    list->size = 0;

#if DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    list->mean = 0;
#endif

#if DATA_STRUCTURE == ARRAY_PRIORITY
    for(i = 0; i <= PRIORITY_SIZE; i++) {
        list->priority_list[i] = NULL;
    }
#endif
    return list;
}
```

```
/*
 * Delete the entire list
 */
void delete_list(list_t* list) {
    node_t* to_remove;

#if DATA_STRUCTURE != ARRAY_PRIORITY
    if(list->size > 0) {
        to_remove = list->first;
        list->first = list->first->next;
        free(to_remove);
        list->size--;
        delete_list(list);
    }
    else {
        free(list);
    }

#else

    int i;
    node_t* current;
    for(i = 0; i <= PRIORITY_SIZE; i++) {
        current = list->priority_list[i];
        while(current != NULL) {
            to_remove = current;
            current = current->next;
            list->size--;
            free(to_remove);
        }
    }
    free(list);
#endif



}


#if HELPER_FUNCTION
void helper_remove(list_t* list) {
    node_t* to_remove = NULL;
    if(list->first == list->last) {
        /*
         * It was the last node in the list.
         */
        assert(list->size == 0);
        free(list->first);
        list->first = NULL;
        list->last = NULL;
    }
    else{
        /*
         * Repoint and free the memory allocation.
         */
        to_remove = list->first;
        list->first = list->first->next;
        free(to_remove);
    }
}
#endif

/*
 * Dequeues the element with highest priority (low numerical value)
 */
int32_t dequeue(list_t* list) {
    int32_t to_return = 0;
#if DATA_STRUCTURE == ARRAY_PRIORITY
    int i;
    node_t* to_remove;
    for(i = 0; i <= PRIORITY_SIZE; i++) {
        to_remove = list->priority_list[i];
        if(to_remove != NULL) {
            to_return = to_remove->pid;
```

```
            list->priority_list[i] = list->priority_list[i]->next;
            free(to_remove);
            list->size--;
            break;
        }
    }
#endif

#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    node_t* to_remove;
    if(list->first != NULL) {
        to_return = list->first->pid;
    }
    else {
        assert(list->size == 0);
        free(list->first);
        list->first = NULL;

        return -1;
    }

    to_remove = list->first;
    list->first = list->first->next;
    free(to_remove);
    list->size--;
#endif

#if DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    if(list->first) {
        /*
         * There is a node to dequeue.
         */
        to_return = list->first->pid;
        if(list->first == list->last) {
            list->mean = list->first->priority;
        }
        else {
            list->mean = (list->first->next->priority
                    + list->last->priority) / 2;
        }
        --list->size;
    }
    helper_remove(list);
#endif

#if DATA_STRUCTURE == DOUBLY_LINKED_LIST
    if(list->first) {
        to_return = list->first->pid;
    }
    --list->size;
    helper_remove(list);
#endif

    return to_return;
}




#if HELPER_FUNCTION
void helper_insert(list_t* list, node_t* current, node_t* to_insert) {
    if(current->priority <= to_insert->priority) {
        to_insert->previous = current;
        to_insert->next = current->next;
        current->next = to_insert;

        if(to_insert->next == NULL) {
            list->last = to_insert;
        }
        else {
            to_insert->next->previous = to_insert;
        }
    }
```

```
    else {
        to_insert->next = current;
        to_insert->previous = current->previous;
        current->previous = to_insert;

        if(to_insert->previous == NULL) {
            list->first = to_insert;
        }
        else {
            to_insert->previous->next = to_insert;
        }
    }
}
#endif


/*
 * Puts the element in the right location
 */
list_t* enqueue(list_t* list, int32_t node, double priority) {
    node_t* current;

#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    node_t* temp = NULL;
#endif

    node_t* to_insert = malloc(sizeof(node_t));
    to_insert->pid = node;
    to_insert->priority = priority;


#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    if(list && list->size == 0) {
        to_insert->next = NULL;
        list->first = to_insert;
    }
    else {
        current = list->first;

        while(current->priority <= priority && current->next != NULL) {
            temp = current;
            current = current->next;
        }

        if(current->priority <= priority) {
            to_insert->next = current->next;
            current->next = to_insert;
        }
        else {
            if (current == list->first) {
                to_insert->next = current;
                list->first = to_insert;
            }
            else {
                temp->next = to_insert;
                to_insert->next = current;
            }
        }
    }
#endif

#if DATA_STRUCTURE == DOUBLY_LINKED_LIST
    if(list && list->size == 0) {
        to_insert->next = NULL;
        to_insert->previous = NULL;
        list->first = to_insert;
        list->last = to_insert;
    }
    else {
        current = list->last;
        while(current->priority > priority && current->previous != NULL) {
            current = current->previous;
        }
```

```
        helper_insert(list, current, to_insert);
    }

#endif

#if DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    if(list && list->size == 0) {
        /*
         * This is the first and last node in the list
         */
        to_insert->next = NULL;
        to_insert->previous = NULL;
        list->first = to_insert;
        list->last = to_insert;
    }
    else if(priority < list->mean) {
        /*
         * Insert from front
         */
        current = list->first;
        while(current->priority <= priority && current->next != NULL) {
            current = current->next;
        }

        helper_insert(list, current, to_insert);

    }
    else{
        /*
         * Insert from rear
         */
        current = list->last;
        while(current->priority > priority && current->previous != NULL) {
            current = current->previous;
        }

        helper_insert(list, current, to_insert);
    }

    list->mean = (list->first->priority +
            list->last->priority) / 2;
#endif

#if DATA_STRUCTURE == ARRAY_PRIORITY
    if (priority <= PRIORITY_SIZE) {
        current = list->priority_list[(int)priority];

        to_insert->next = NULL;
        if(current == NULL) {
            list->priority_list[(int)priority] = to_insert;
        }
        else{
            while(current->next != NULL) {
                current = current->next;
            }
            current->next = to_insert;
        }

    }
    else {
        printf("ERROR: Maximum priority is %d\n", PRIORITY_SIZE);
    }
#endif

    list->size++;
    return list;
}
```

Appendix I

Code – `queue.h`

```
#ifndef QUEUES
#define QUEUES

#define TRUE 1

#define PRIORITY_SIZE 40

#define SINGLY_LINKED_LIST 1
#define DOUBLY_LINKED_LIST 2
#define DOUBLY_LINKED_LIST_AVG 3
#define ARRAY_PRIORITY 4

#include "queues.h"


typedef struct node_t {
#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    struct node_t* next;
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST
    struct node_t* next;
    struct node_t* previous;
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    struct node_t* next;
    struct node_t* previous;
#elif DATA_STRUCTURE == ARRAY_PRIORITY
    struct node_t* next;
#endif
    int32_t pid;
    double priority;
} node_t;


typedef struct list_t {
#if DATA_STRUCTURE == SINGLY_LINKED_LIST
    node_t* first;
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST
    node_t* first;
    node_t* last;
#elif DATA_STRUCTURE == DOUBLY_LINKED_LIST_AVG
    node_t* first;
    node_t* last;
    double mean;
#elif DATA_STRUCTURE == ARRAY_PRIORITY
    node_t* priority_list[PRIORITY_SIZE + 1];
#endif

    uint32_t size;
} list_t;

void print_list(list_t* list);
list_t* new_queue();
```

```
list_t* enqueue(list_t* list, int32_t node, double priority);
int32_t dequeue(list_t* list);
void delete_list(list_t* list);
#endif
```

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]