Experimental Method - Planning

January 31, 2018

WONG, SAI MAN
TIGERSTRÖM, GABRIEL

Contents

1 Experiment Method – Planning **1**
1.1 Introduction 1
1.1.1 Background 1
1.1.2 Problem 2
1.1.3 Hypothesis 3
1.1.4 Purpose 3
1.1.5 Goal 3
1.1.6 Social Benefit, Ethics and Sustainable Development 4
1.2 Method 5
1.2.1 Inductive and Deductive Method 5
1.2.2 Quantitative and Qualitative Method 5
1.2.3 Data Collection 5
1.2.4 Data Analysis 6
1.3 System Description 7
1.3.1 System Development Method 7
1.3.2 Queue Algorithms 7
1.3.3 Experimental Setup 10

Appendices **12**

A Requirement Analysis **13**

Bibliography **15**

Chapter 1

Experiment Method – Planning

1.1 Introduction

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

1.1.1 Background

A data structure represents a model to store data methodically [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 complete implementations, such as a queue. Software developers use these implementations frequently, and sometimes take the underlying architecture for granted. Thus, an inexperienced developer possesses 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 scientists and developers refer the entity as a node or an element, and the connection as a pointer. Developers frequently use linked list implementations to store and manage data linearly, for example, in a queue implementation.

1.1.2 Problem

Our task was to evaluate four variations of a list-based data structure to represent a queue [2]. Each element comprises 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 must implement 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 by their execution time and memory usage. We plan to gather a large quantity of these metrics to determine best, average and worst cases. Thus, we ask ourselves: "Do these queues achieve similar performance, or does the performance vary depending on the circumstance?"

1.1.3 Hypothesis

A queue either enqueues or dequeues elements with a FIFO policy. Because of the FIFO policy, based on a correct implementation and theory, the dequeue-operation takes $\mathcal{O}(1)$ time. However, the enqueue-operation needs to consider an element's priority, because the algorithm inserts 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.

1.1.4 Purpose

Our purpose is to raise awareness of the features in standardized libraries, which developers and engineers often take for granted. Thus, we applied the scientific method to conduct experiments and evaluate four versions of a priority queue implementation. Consequently, we want to produce a credible and scientific work, so that other students and engineers can learn from, reproduce, and further develop.

1.1.5 Goal

The goal is to attain deeper knowledge of the scientific method, and apply the method to evaluate four queue-implementations. The following list demonstrates the sub-goals:

- Apply the 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.

1.1.6 Social Benefit, Ethics and Sustainable Development

Our study might influence students and engineers to apply the scientific method with a more serious approach. Also, the study facilitates 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 helps students and engineers to acquire a deeper comprehension of computer science.

Within science, a researcher undertakes 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 economic and practical reasons, such as expensive hardware. We will document each step and present our findings with rigor, honesty, and transparency. That is, we encourage students to reproduce our findings.

We will experiment in a virtual machine on Microsoft's cloud platform. Thus, we use the exact quantity of resources needed to conduct the experiments. Thereby, avoiding waste of electricity due to, for example, system idle.

1.2 Method

1.2.1 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 the pattern in the data, 3) set up a hypothesis and 4) explain the findings with the 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 a theory first. For example, mathematicians use a deductive method to explain the unknown with the 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.

1.2.2 Quantitative and Qualitative Method

Researchers apply a quantitative method in studies, which rely on measurable data in large quantities to explain findings [5]. That is, a quantitative study uses 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 more extended period.

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

1.2.3 Data Collection

We will be handling quantifiable metrics in our study. That is, execution time and memory usage because computer scientists commonly apply these for algorithm 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 reach conclusions from the quality of the data. Thus, our

study use an experimental quantitative method to reach a broader insight into data structures and queue implementations.

For each of the four queue-implementations, we will test and generate the execution time for the best, average and worst case. We will create input for the queues with varying distributions of the element's priorities. Besides execution time, we will log the memory usage for each implementation. Finally, we will write scripts to execute the tests and log the raw data.

1.2.4 Data Analysis

We reason with a deductive method to explore the theory within computer science, such as algorithm complexity theory. That is, we analyze the algorithms with the 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 method can yield unexpected results when put into practice. Thus, we will generate realistic data from our experiments and use an inductive method to analyze the algorithms further. That is, to explore if we can notice observable patterns in the data with the help of complexity models.

By running the experiments multiple times we can obtain the sample mean and standard deviation. The standard deviation adds a deeper insight into the data and spread around the mean. Also, it helps 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.

1.3 System Description

1.3.1 System Development Method

In our implementation, we plan to generate an executable for each of the four queue-implementations. The experiment would consists of, for example, `main.c` containing the tests, `queues.h` and `queues.c` which respectively would describe and implement the functionalities of the queue. Thus, we must use compile-time macros to determine the executable to produce based on the implementations, and check for memory leaks with, for example, Valgrind.

We plan to 1) use scripts and container technology (Docker) to automate the experiments. Also, 2) to process and analyze the data, we can use mathematical tools, such as MATLAB or the Python library NumPy for scientific computing. Lastly, we must 3) present these findings, analysis, and interpretation in form of tables and graphs in LaTeX. Thus, our procedure is to implement 1) to gather, 2) analyze and 3) present the data. Docker containers introduce a practical approach to package software because containers use the same resources as the host operating system.

1.3.2 Queue Algorithms

Our task is to implement four different queue-implementation in the programming language `C`, as described in Section 1.1.2. The following are standard 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 challenging to analyze compared to the best and worst case theoretically. Thus, we examined the best and worst cases in theory to interpret the average case for each implementation's enqueue-operation.

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 only to 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.

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 connect to the 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 continually descends from a higher one. Thus, the algorithm traverses the list from the rear n-times to insert the new element correctly.

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 significantly 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.

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 1.3.2. 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:

  $$1 + \frac{n}{41}$$

  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 1.3.2. That is, the implementation traverse n-times for a priority distribution of the same priority.

1.3.3 Experimental Setup

We conducted the experiments on a Microsoft Azure virtual private server (VPS). We used a VPS service because we gain full access to 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.

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 varies around the mean. A large standard deviation indicates, for example, that another process a 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. Furthermore, especially in a relatively low-level programming language, such as `C`.

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 contain 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. Also, 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.

Table 1.1. Experiment parameters for each implementation and test case

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

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 1.3.2. 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 1.1 presents the remaining parameters we used for each test case.

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

- **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 difference in 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,

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

Appendix A

Requirement Analysis

In [6], 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 [7]	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].	Fulfilled Section 1.1.4
2	**Must** mandatory task	Prestudy [7]	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	Fulfilled Section 1.1.1
3	**Must** mandatory task	Methods [7]	Identify which scientific method(s) to use. For example, quantitative, qualitative, deductive and/or inductive method	Fulfilled Section 1.2
4	**Must** mandatory task	Experimental Plan [7]	Identify experiments to conduct. Describe why these are relevant. Examine metrics to measure and elaborate on why these are relevant to this study.	Fulfilled Section 1.3.2

5	**Must** mandatory task	Experimental Setup [7]	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.	Fulfilled **??**
6	**Must** mandatory task	Goals [7]	Demonstrate how to determine when a goal is reached and an experiment is completed.	Fulfilled Section 1.1.5
7	**Must** mandatory task	Communication Part I [7]	Communicate and summarize the discussions of requirement 1-6 (Appendix A) in a document and hand it in to the mentor/supervisor	Fulfilled
8	**Must** mandatory task	Experimental Execution [7, 2]	Follow the discussed requirements 1-6, or only 7, to conduct the experiments on a real system. And, regularly document the findings with honesty and transparency.	
9	**Must** mandatory task	Communication Part II [8]	Summarize the entire work and its requirements, that is 1-8, 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.	

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: 2017-11-28]

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

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

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