The Asymptotic Complexity Comparison Between Double Linked List and Skew Heap Implementation of Priority Queue

Qi Li (qi5@kth.se)
Hussam Hassanein (hussamh@kth.se)

Abstract

A lot of the perspectives have been discussed, when it came to coming up with a solution to the complexity problem when comparing the double linked list and skew heap Implementation of priority queue. The solution of the problem is clearly the end product of the process, but a lot of focus in this report has been on the process in all phases of development and analysis. It is important that the details are covered in a way that the reader can understand the scope of the solution in every step. In the problem section, an explanation of problem and its scope is discussed and its purpose, and requirements to fulfill. The report includes a description of how things are done, and what are the constants and variables that are being used during the experiment. The second part of the report has they analysis of methods used and an evaluation of the process and the results of the experiment, to give enough depth to the solution.

Sammanfattning

Det var ju många perspektiv till komplexitet problemet, som blev diskuterat, I den här rapporten. Lösningen till problemet var en hel del av den. Det är som viktig att tänka på, är att processen skulle var förenklat till läsaren, med att ha hur mycket detaljer som helst, för att uppnå den behövs förståelsen. I problemet delen av rapporten, finns det en beskrivning av problemet, anledning till det, och hur experimentet utföras I praktiken. Efter den, så kommer det en analysering metoderna som används I experimentet och hur lösningen till problemet blev uppfattat. Det var väldigt viktigt att förklara de olika varianterna och konstanterna som används I experimentet för att de räknas som villkoren på att lösningen ska gälla.

Table of content

1 Introduction

In this report, the experiment of complexity comparison of the double linked list and skew heap Implementation of priority queue is explained and analysed in details. The first section of the report is dedicated to the problem explanation from different dimensions. The other section of the report is dedicated for the analysis of the process and the evaluation of the outcome of the experiment.

This experiment was done as a part of an assignment introduced in Engineering Skills course, with Identification number II1304, given by KTH. This report gives details on the process of the coming up to a solution, and the results of the mentioned experiment. Some of the results of this experiment are introduced in the form of charts and graphs, for a better clarification.

1.1 Background

Priority queue is a First In First Out (FIFO) abstract data structure that has broad implementation in information technology. The queue is sorted by the “priorities” as the priority queue that has been named[1]. A priority queue usually supports the is_empty to check whether the queue has no elements, the insert_with_priority to add an element to the queue with an associated priority and the pull_highest_priority_element to remove the element from the queue that has the highest priority, and return it [2]. The most naive implementation is to implement the priority queue as a single linked list. This implementation is quite expensive as known. Therefore the experiment uses double linked list as first comparison material. The most typically implementation of priority queue is heap [1]. There are multiple heap implementations. The Skew heap, a priority queue based on a self-adjusting heap which was suggested by Daniel Sleator and Robert Tarjan, has been used as second comparison material [3].

1.2 Problem

Different implementation of the priority queue can have a different efficiency. Since the priority queue has a broad usage for system design in basic level, an efficient implementation of priority queue can save a lot of time for the rest of the system. For example, Dijkstra's Shortest Path Algorithm using priority queue, Prim's algorithm, Data compression [1], and Operating system Process Queue.

From literature study, it is shown that, the linked list has O(n) as an average asymptotic complexity for the whole operation and O(log n) for skew heap. Ones can also learn that linked list should be slower than the skew heap implementation.

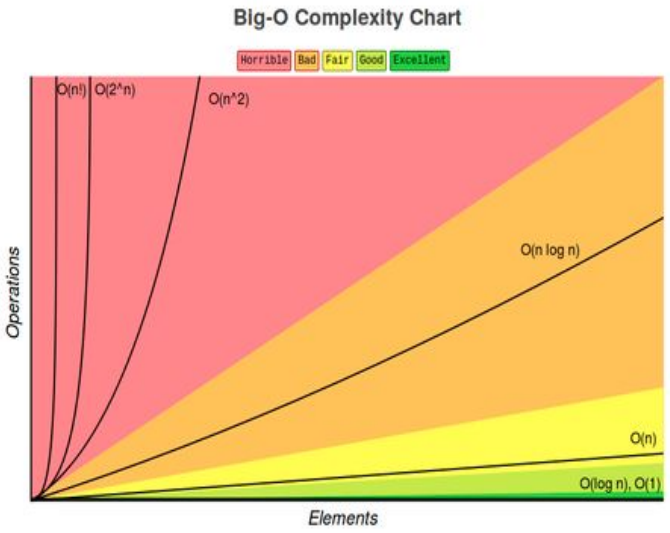

Figure1. Big O complexity chart [4]

In this experiment, the double linked list has been selected over the singly linked list. As mentioned in the literature studies, the results are expected to remain the same. Implementing a self-organized skew heap structure takes more like than implementing a doubly linked list. If eventually the experiment can prove that with the extra link the efficiency would be better than in skew heap, the result would save time during future development of Operating system and kernel for other system.

The problem that is needed to be solved for the experiment, is to find out if a double linked list makes any difference compared to a singly linked list, when it comes to complexity comparison of the implementation of priority queue.

1.3 Purpose

The purpose of the experiment is to find out the most efficient implementation of priority queue from double linked list and skew heap, and solve the problem during the process to reach the goal. The experiment is considered to be done only when both methods are executed and run through different values under the same requirements, and that results in having a similar pattern for these different inputs. The end goal is to determine which method is more efficient and that could be decided by running different algorithm to measure the speed/ effectivity unit for skew heap and linked list implementation as a priority queue. The comparison is to be done under the same requirements to have pass the verification test for each of the methods and be able to validate and test the results.

1.4 Objectives

The experiment is expected that the linked list is still slower than skew heap when comes to implementation for priority queue with exactly same asymptotic complexity. The expectation is based on the the double linked list has average complexity O(n) as figure 2 stated, which is same as single linked list [4].

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Figure 2. Common data structure operations [4]

So regardless of the usage of the double linked list, the result should remain the same. The experiment are expected to have a O(n) line and O(logn) line as stated in figure 1.

During the planning phase, ones has set up the short term goals for different phases of the experiment as figure 3 stated. Most of the goals have been successfully reached, but there are few that were not reached due to time box has been reached. To make sure the goal would actually be reached, different verification and validation process have been proceeded. The validation and verification method for each single goal is explained into details in the method section.

Requirements:

MoSCoW: Must have, Should have, Could have, and Won't have[5]

Requirements number	Type of requirement	Designation Source	Description of the requirement / what must be met	Completed / Not met / Partially met
1	Must have	Queue structure [3]	Queue must be FIFO structure. To verify it the first item that goes into the queue must be the first one be popped [3].	Completely reached
2	Must have	Multiple input data pattern. [4]	The experiment must be carry out at least 3 different input data patterns. Each data pattern must be recursively done 3 times. This is to minimize the random error of the data.	Completely reached

3	Should have	Use Usual implementation [3]	The implementation of the queue is preferable in usual implementation than naive implementation as Wikipedia stated [3].	Completely reached
4.	Should have	Automation experiment step (experiment requirement)	The experiment is preferable to be implemented as automotive as possible. This is to eliminate Random error and systematic error.	Completely reached
5	Should have	Pass the Unit testing and Integration testing	Eliminate the systematic error from the implementation of the code. This can be verified when SonarQube start showing passing the Quality gate and TravisCI finished with build success.	Not reached due to the the time box has been excess

Figure 3. the goal of the experiment

1.5 Social benefits, ethics and sustainable development

The experiment can benefit the Operating System kernel and software developer. As mentioned in 1.2 problem part before, the skew heap implementation is complicated and easily to make mistake. The skew heap implementation is even complicate to verificate the correction of the implementation due to the structure of the skew heap does not look like a priority queue. The reason skew heap can be used as a priority queue is due to its FIFO property. If one proves that double linked list can have better efficiency compared to skew heap, with adding one pointer to the original linked list data structure, this can save a lot of efforts when comes to system development. Since the double linked list is also easier to be visualize, the validation and verification process can be simplified. The time, which is saved on validation and verification, implementation and preparation for the skew heap as priority queue for basic architecture of developing software or kernel, can be used for other tasks: to improve scrum meeting, to develop other modules, and for developer just to have a cup of tea in order to perform a higher-productivity work afterwards. Since the verification and validation process are easier for double linked list, the result of the experiment may help other developers to simplify the future development process for their own software. In long run, the experiment result can create a sustainable develop environment.

1.6 Delimitations

The experiment has been run at a dual core personal used intel i5 linux laptop. This means the result of the experiment might be limited only on this hardware and operating system. Since the laptop is meant for personal usage, the operating system is not clean and there are also other background applications running with one's notice. The delimitation can also be based on linux operating system. This means on different operating systems like windows or mac OS might create different result due to the operating system’s architecture.

1.7 Disposition

The report describes the method one used for reach the conclusion of the experiment. In this section, it is expected to find the methods used for data collection and analysis.

The implementation and testing method explains in the System section. In system section one can also expect to find the system design for measurement before the data collection. The experiment are expected to be automotive, ones can also find how the automotive part has been construct in system section.

The raw data and graph can be found in the result section. The results are expected to be raw and not analyzed. Ones can expect to find the analyzed data with conclusion in the conclusion section afterwards.

2 Method

The experiment has been done by implementing the linked list and skew heap test platform, collecting the data from the test platform and analyze the data through gnuplot.

2.1 Data collection

The data collection are expected to be done automatically by the bash script in plot.sh combine with run1.sh for skew heap and run2.sh file for linked list. To eliminate the the systematic uncertainty, the experiment has been perform with 3 data pattern for each input. Each of them is meant for max, min and medium value. Each data pattern was performed 3 times. After all the data has been outputted into the plot folder under root directory, the gnuplot was added all of them together and divide by 3*3 which is 9 times to obtain the average value of each raw. Eventually gnuplot can expect to plot the graph with only one ‘dat’ output file. The reason to have separate plot.sh ,run1.sh and run2.sh file for automatic data collection is to prevent the false implementation in the system development like memory leak and dependency between each experiment. Since after operating system execute each process of the operating system would freed all the page it reserve for this process[8], by separating each experiment run into different process can eliminate the issue that ones have done for the implementation. The detail about the implementation of two different priority queue is in the section 3 System.

To be able to do automation data collection correctly, one has to do certain amount of literature study and experiment before the automated data collection works perfectly. Priority queue experiment is the first time one engage automotive experiment. The KTH operating system and Engineering skill in ICT course leader has present few automotive method during the lecture.
The lecture note that taken from these course by the team member would be used as preliminary study. To be able to solve the problem during implement the automotive data collection system, the post on stackexchange website would be used and studied [6]. Ones is also expecting to use some existence *.p file with some modification from operating system course for gnuplot [7].

2.2 Data analysis

The data has been automatically analyze and output into graph after the data collection phase. The analyze result is base on X coordinate = the number of element in queue that initially been scheduled in the queue (Queue size) and Y coordinates = time cost to clean up the queue. The graph output is shown as figure 4 below.

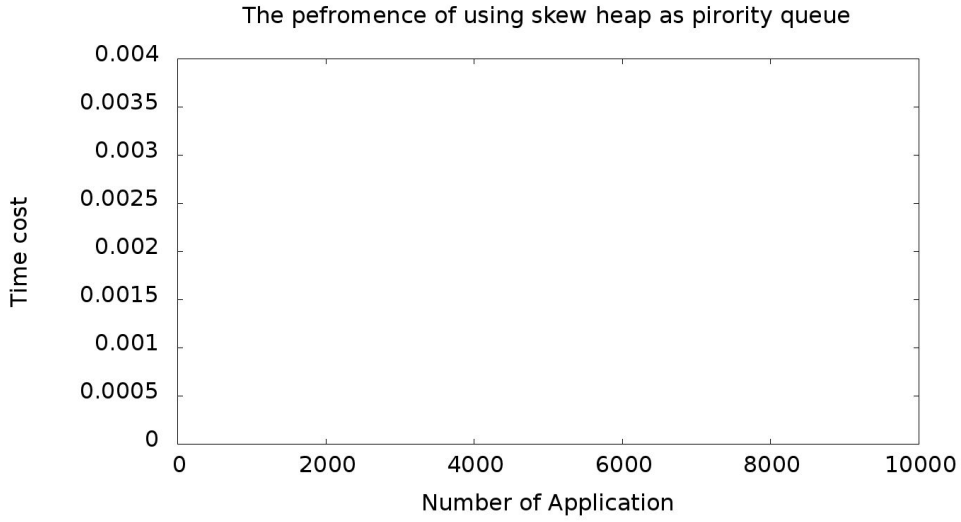

figure 4. empty graph output for data analyze

Figure 4 only showed a empty graph that what the output might looks like instead a complete graph. The complete graph is in section 4 Result. The automotive program would also output a basic gnuplot stat analyze result for both implementation in the command line. The statistical analyze from gnuplot would also be showed in section 4, in the result section. Base on these analytical result that obtain from gnuplot, ones can output the manual data analyze and conclusion individually.

3 System

The program has treated each queue as a process queue and each element as a small process. Each process has same action.This is also the basic idea when one implement the code. Previous section has explain how the automatic data collection and analyze script has been implemented. This section explains how each queue is actually been implemented. The system section explains what is happened after run1.sh and run2.sh.

3.1 System development method

The program has been developed under C programming environment with Atom editor. To be able to valid each modul and if the implementation fits the purpose that one served for, the debug tool GDB has been used for debug and verification. Since computer has its own clock embedded, the C Library - <time.h> has been used for data measurement for each run of the experiment. The system library math.h has also been used combine with time.h for implement the pseudo random number generator.

During development phase, one has treat each queue as a process queue and each element as a small process. Each process has same action. After each process has done with its action, it would be taken away from the queue. The details of actions that each element/process has done are explained in section 3.2 implement algorithms.

3.2 Implemented algorithms

3.2.1 Literature study for the algorithms

Since linked list implementation is quite traditional way to implement the priority queue, revision on the KTH course material for ID1020 Algorithms and Data Structures is sufficient. Base on this study, ones would evaluate how the double linked list should be implemented as priority queue. The material is available on KTH course web. The section 11 course material from Oregon State University has been used as study material due to [9] detailed efficiency comparison with graph presented. The document has also multiple previous sample data stated which can be used as an idea for analytical study. The efficiency measurement system has been implement from dedication C library. The system could help

system to measure the speed and the complexity for each of the structure and then compare each of the structure to come up with an answer for the problem that the experiment are trying to solve. In case of system error that caused by wrong implementation, ones has study the "C Library - <time.h> “ article from Tutorials Point[10] before implement the efficiency measurement system.

3.2.2 Implemented Method

In skew heap implementation, the code has been divided into 7 modules for the property reason. The modeles are heap/node creation, naive merge, swap, add, pop, decompose, and a random number generator ”increment”. To be able to do the following implementation correctly, the item of the heap is predefined with the type define”typedf structure” method.

The heap/node creation creates a typedef Tree object with all the property initialized. The property of the typedef structure has been initially planned as 6 parts: the address of parent, the left element, the right element, the content or priority of the queue, the length of the architecture and the address current node. The add method adds a node inside the skew heap and reorder it as the skew heap required. To be to add item inside the skew heap without breaking the regular of the skew heap, the add function need to have naive merage method and swap method. The naive merge merges an item or a skew heap with current skew heap but this could cause rightest tree. The swap method solves the problem by swapping each node from the tail of the heap and eventually make a even binary tree. The pop method returns, detaches the head of the tree and reorders the tree like the add method. During each pop, the heap generates certain amount of subtask. To be able to do subtask generation, the team has implement the decompose method. The decompose method generalizes the task with a timestep which base on the timestep that the item just pop and plus a random generate number. The random number generator is a easy math function rand() divide and obtain the remainder of the limitation that maximum number allowed to generate. Time step uses the same system as efficiency measurement system which is the dedicate c library “time.h”. Time.h function obtains current CPU time and returns a floating point number. The time stamp takes this number and puts it inside the node.

The double linked list implementation has same structure but

different implementation than skew heap inside each module.The linked list has inserted (add) to add value into the list, free_list to free the malloc space from heap, pop to take out the value from the list, decompose to generate 0-N subtask and eventually increment for generate random number. To be able to fits the basic requirement of the double linked list, each item would be typed define “typedef structure” as integer data to store the priority/content of the item, a pointer to the next item, a pointer to the previous item and a pointer to the tail address inside the list.

The insert function tries to append a item without breaking the FIFO order. When the current append item is smaller than the average between the first and last item in the queue, the program starts searching for properate position to insert from the front of the queue otherwise from end of the queue. By suitable position, it means that the previous item should be smaller than current item item and the next item should be bigger than current insertion item. When the suitable position is found, the queue detaches the link with the next compared item and previous compared item. After detach the link with next and previous compared item, the program would save the address of the next compared item before attach to the new item. Eventually, the newly append item would also attach to the next compared item.

The pop function is built based on the list and is always organized as a FIFO like priority queue structure. So the pop function can pop the item that is in the top of the list which should be the same as the head of the priority queue. Decompose function would act as same as skew heap’s decompose function with 0-N sub item generated after each pop. For increment function, since the random number generator does not depends on any data structure, the linked list would import the increment function from skew heap header file or just simple copy and paste the increment function from skew heap c code file.

One can see the appendix for detail implementation of the algorithm for skew heap and double linked list implementation of the priority queue.

3.3 Test bed

Following errors that affect the experiment:
Systematic error: CPU and Hardware load
Implementation issue during unappropriated code
Random error: Human mistake during experimental phase.

The experiment being proceeded under following equipment setup to minimize the uncertainties:
Operating System: Linux (Programming under Mac and Linux)
Programming language: C, bash script
Compiler: gcc
IDE: Atom
Plotting tools:gnuplots
Unit testing: SonarQube
Integration testing: TravisCI
Version control: Github
Debug tool: GDB

CPU and hardware load can cause systematic error during the experiment. Operating system has loaded with different background applications. At different timing the CPU and hardware load are different. This would cause uncertainties for the data. C language is the language that is closest to the hardware. Using C to programming and compiling the experiment can ideally eliminate systematic error from CPU and other hardware load as low as possible. However, windows and mac have many background applications running which affects the output data. Linux has relevantly ideally environment for this experiment since it only loads with what the experiment needed.

Operating mistake by human can cause Random errors for the experiment. Random errors can be eliminated by repeating the experiment. It can also been avoided by make the experiment step automotive. Since computer is good at doing stuff that have certain pattern and repeat itself. That is why the experiment is as automotive as possible. To be able to reach automotive goal, the experiment equipment should be able to use port or tunnel operation in the script. Gnuplot can be used as port “>” operation

from c code directly. This can be used in automation of the experiment to eliminate the random errors.

To be able to further eliminate the Random errors, as Experiment Plan stated the experiment for each data pattern was carried out 3 trials and there were 3 different data patterns. Each data pattern could be calculated by the mean value out of 3 trials. The graph was compared to make sure if the conclusion ones had is correct. Implementation the code in an unappropriated way can cause Systematic error. This kind of error cannot be eliminated from repeating the experiment and would create a fix amount of uncertainty in the output data. To avoid that, the develop phase is carried out with Continuous integration and deployment tools. Implementation phase would be carry out with SonarQube for unit testing and TravisCi for integration testing to eliminate the systematic error that caused during the develop phase. The standard for verification of this part is to pass the Quality gate from SonarQube and successfully build from TravisCI.

The GDB is used as important verification and validation tool during the implementation.Since each element has been saved in certain address (0x….) and each element has multiple linked to the previous,next element and for linked list even link to the tail. The only way to find out whether one has done the implementation correctly is though GDB. The command to uses here is GDB build/* .o to start the debug mode and set certain breakpoint though breakpoint <line Nr> and record the action for reversing debug though reverse-finish, reverse-continue and reverse-step. Ones can use print <container> to access certain address content. GDB is relative a good debug tool during C developer phase because GDB basicly can access all the memory content that a process is reserved for a program. Since GDB is also the official and most popular debug tool, it has a active community online that one can access if gets into problem.

3.3.1 Measurement methods

After being done with all the implementation of the skew heap and linked list priority queue, one should start implement the efficiency measurement system. This would still be done by “time.h” function in the dedicated c library. Program obtains the CPU time at the

beginning of the code. Then it obtains the CPU time again in the end of the code and uses the floating point value which program just gets in the end of the code to minus the floating point value from the beginning of the code. To convert the CPU time to normal clock second, one should use (CPU TIME)/CLOCKS_PER_SEC. The CLOCKS_PER_SEC is a macro expands to an expression representing the number of clock ticks per second [8].
One can see the appendix for detail implementation of the measurement methods for skew heap and linked list implementation.

3.4 Experiments

The experiment is designed based on the idea that “as automatic as possible” which can eliminate the random error. Computer is good at doing same thing over and over again and doing things that has certain rules.Human is good at creating the rules and create new element[11]. Each experiment run performed by individual call from script command to eliminate the random error that happened due to code implementation failure. This make sure each experiment run are independent to each other.

The Skew and Linked list is implemented based on the literature study. After each element has been pop, the software reads the priority level of of the popped element and generate random amount of sub element based on the priority of the popped element. The priority of the new element is the priority of the popped element add a random number. The random number generator is implemented based in time.h and math.h. The program uses rand() function from math.h library to generate the random number. Before the random number that has been generated, the program use srand to see the random number to the rand() function base on CPU clock. This makes sure each time that rand() been called the function provides a number that is random enough since the CPU clock is always different by the time goes on. The randomness is important because the randomness can guarantee that each queue operation for different part of the queue has been tested. This can give us a fair result with out uncertainties.

The final result is ported (> operation) into gnuplot by the format defined in heap.p in plot folder. Inside this folder, each row of the experiment result is added together and calculate the mean. This mean *.dat file is used as final plot material. By calculating the mean, the result can one step further eliminate the systematic error and random error that cause by human.

Before gnuplot plotting the graph, the heap.p file has been defined that 2 statistic analyse is the output into command line to help user to create the final conclusion. By the help of gnuplot, the random error of final conclusion can be eliminated and a precise and accurate conclusion can be created.

4 Results

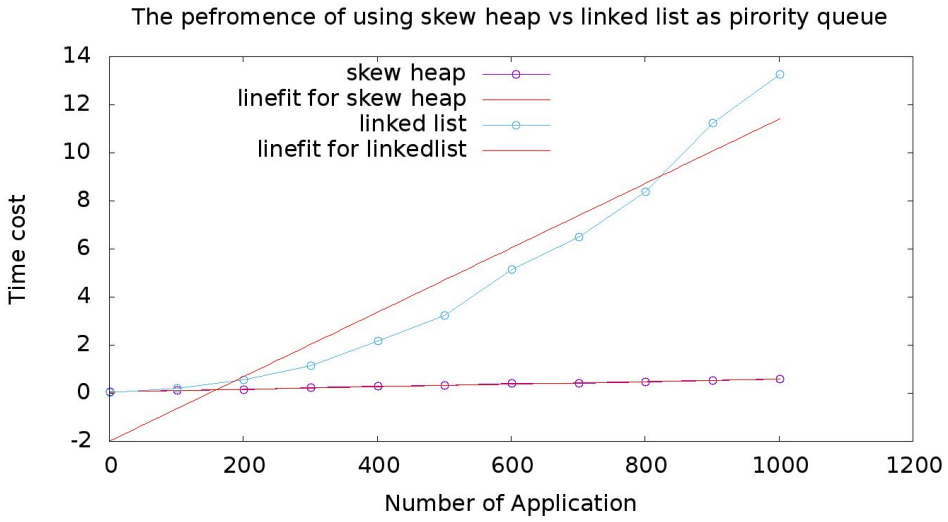

figure 5.The performance of using skew heap vs double linked list as priority queue

The result is same as the estimation ones get in the 1.4 Objective chapter but with a different outcome. Double Linked List is having a O(N log N) instead O(N). This conclusion is comes from the comparison between the output graph and figure 1. This is interesting since as imagination the double linked list implementation should only take half of the time than normal linked list since double linked list calculate the average between the first and last element than decide to search from head or tail. Does not matter which side it search from, the search operation will never pass the middle of the list. Apparently the conclusion is wrong and the graph shows the the answer is O(N log N). One can see the appendix 4 for detail raw data of each operation and appendix 5 for gnuplot statistic analyze result.

One can see from appendix 5 that the linear model for skew heap is
Linear Model: y = 0.0005333 x + 0.05802

slope is

Slope: 0.0005333 +- 1.122e-05

and for Linked List is

Linear Model: y = 0.01339 x - 1.995.

slope is

Slope: 0.01339 +- 0.001264

Linear model has also been called as best fit which means that linear model use a line to fit the original plot. From linear plot one should also getting information about the increment or decrement level. From these formula one can see that the slope of the Linked list is higher than skew heap by obtain the number before the x variable. At same time, the linked list is having lower value than skew heap during the start of the experiment by obtaining the value after the x variable. This is understandable due to skew heap need to compare, insert, merge and swap but linked list only compare and insert. Even the compare length might be different but in the start of the experiment definitely the linked list is much efficient. Since the skew heap is a self-organized heap and all it element is self organized with less comparison length this cause the skew heap start taking less time with in few experiment. By the time goes on the difference between skew heap and linked list is getting bigger and bigger [9].

4.1 Discussion

The research was first completed with the intention to explore the topic of algorithms and then the concepts of linked lists and skew heap as priority queues. The exploration helped a lot in deciding the approach to tackle the problem faced and finding the solution throughout a structural process. A lot of time has been invested in the research phase to achieve a full understanding of the scope of the experiment. With this sort of information in mind, the details of the experiment becomes clearer and clearer by the time and some variables are being given limits and some constants become part of the experiment. The methods as discussed earlier in their section were used to analyze and help test the results. The methods give a different perspective about how the problem was tackled in both technical and non technical matters. The results presented by graph here is very dependent on the mentioned requirements, which have been validated and tested through.

5 Summary and future work

The purpose of the experiment is to find out the most efficient implementation of priority queue from double linked list and skew heap, and solve the problem during the process to reach the goal. The answer has been stated in the result chapter which is the skew heap is more efficient than double linked list. By reaching our goal, the problem of the experiment has also been solved. By changing from single linked list to doubly linked list, the skew heap is still perform better but the double linked list has a asymptotic complexity of O(NlogN) instead O(N) as ones assumed when comes to priority queue implementation. By the times goes on the difference in complexity between skew heap and double linked list gets bigger. This clearly show that skew heap has a benefit in large amount of data/process management but double linked list has benifit in small data management.

The goal has mostly been reach except the bug fix from sonarQube. sonar has detect few formate and unused variable issue that need to fix. The issues does not affect the experiment result it self but definitely worth to fix to make the code looks nicer and benefit future development of the code. Since the purpose the the sonarQube is unit testing, if the unit testing is not completely pass it can cause interference with future implement module.

The experiment can also be continued by input the test bed as a module into some existing code on github and test the outcome and different between skew heap and linked list as priority queue.

References

[1]"Applications of Priority Queue - GeeksforGeeks", GeeksforGeeks, 2018. [Online]. Available: https://www.geeksforgeeks.org/applications-priority-queue/. [Accessed: 14- Jan- 2018].

[2]"Priority queue", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Priority_queue. [Accessed: 14- Jan- 2018].

[3]"On the assignment", Kth.instructure.com, 2018. [Online]. Available: https://kth.instructure.com/courses/2503/pages/on-the-assignment?module_item_id=43870. [Accessed: 14- Jan- 2018].

[4]"Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell", Bigocheatsheet.com, 2018. [Online]. Available: http://bigocheatsheet.com/. [Accessed: 14- Jan- 2018].

[5] En.wikipedia.org. (2017). MoSCoW method. [online] Available at: https://en.wikipedia.org/wiki/MoSCoW_method [Accessed 6 Dec. 2017].

[6]"Hot Questions - Stack Exchange", Stackexchange.com, 2017. [Online]. Available:

https://stackexchange.com/. [Accessed: 19- Dec- 2017].

[7]Gits-15.sys.kth.se. (2017). Lecture material for ID1200/06 . [online] Available at:

https://gits-15.sys.kth.se/johanmon/ID1206 [Accessed 20 Dec. 2017]

[8]R. Arpaci-Dusseau and A. Arpaci-Dusseau, Operating systems. [S. l.]: Arpaci-Dusseau Books, 2015.

[9]"Chapter 11: Priority Queues and Heaps", Web.engr.oregonstate.edu, 2017. [Online].Available:http://web.engr.oregonstate.edu/~sinisa/courses/OSU/CS261/CS261_Textbook/Chapter11.pdf.[Accessed: 19- Dec- 2017].

[10]"C Library - <time.h>", www.tutorialspoint.com, 2017. [Online]. Available:https://www.tutorialspoint.com/c_standard_library/time_h.htm. [Accessed: 19- Dec- 2017].

[11]W. Newman and M. Lamming, Interactive system design. Wokingham, Eng.: Addison-Wesley, 1995.

Appendices

Appendix 1: Source code for the algorithms

For skew heap(heap.c):

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <assert.h>

#include <time.h>



typedef struct Trees{

  struct Trees *parent;

  struct Trees *left;

  struct Trees *right;

  float value;

  int arch;

  struct Trees *self;
```

```
}Tree;

//public
struct Trees *empty =NULL;
struct Trees *head;
int maxTask;
int nrEvent;
int archtecture;
int dynAvg;
int chance;
int pCount;



//method declearation
Tree * creatHeap(Tree *heap,float value);
Tree * creatNode(float value,int arch);
int add(Tree *main, float item,int arch);
Tree * naiveMerge(Tree *main1, Tree *item);
int swap(Tree *tail);
Tree pop(Tree *main);
int merge(Tree *main, Tree *item);
int decompose(Tree element,clock_t timestemp);
```

```
int increment();


int debug(Tree *head) {
        if(head == NULL){
//      printf(" \t ");
printf("# dynavg value %d\n",dynAvg);
        return 0;
        }
        //printf(" \t ");
        debug(head -> left);
        //printf(" \t ");
        printf("# |____");
        printf(" %f\n",head -> value);
        //printf(" \t ");


        debug(head ->right);


}


//method
int main(int argc, char *argv[] ){
if(argc != 4) {
        printf("usage: list <maxTask> <nrEvent> <archtecture>\n");
```

```
        exit(0);
    }
    maxTask =atoi(argv[1]) ;
    nrEvent =atoi(argv[2]);
    archtecture=atoi(argv[3]);
    int counter=0;
    dynAvg=0;
    chance=0;
    pCount=0;


int current=nrEvent;
float dataList[3];
    clock_t t,timestemp;
    t=clock();//predefined  function in c
    srand(time(NULL));
    //add
    Tree *heap=creatHeap(heap,t);
    for (int i =1;i<current+1;i++){
        timestemp =clock();
        add(heap,timestemp,rand()%archtecture);
    }
    /*
```

```
  printf("\n#After head: \n");
  printf("#head is %f\n",head->value);
  debug(head);
  */
//pop
  while((heap->left)!=empty||(heap -> right)!=empty){
        Tree element=pop(heap);
        /*
        printf("\n#After pop: \n");
        printf("head is %f\n",head->value);
debug(head);
*/
        //decompose
        if(element.arch>0){
   decompose(element, timestemp);
   /*
   printf("\n#After decompose: \n");
   printf("head is %f\n",head->value);
   debug(head);
   */
}
   }
   Tree element=pop(heap);
```

```
  /*
  printf("\n#After last pop: \n");
  printf("head is %f\n",head->value);
  debug(head);
  */


 t=clock()-t;
 float effeciency=((float)t)/CLOCKS_PER_SEC*1000;
  printf("%d\t%.8f\n", current, effeciency);

//printf("# not crashed");

return 0;
}
int decompose(Tree element,clock_t timestemp){
  element.arch=element.arch-1;
  int n =rand()%maxTask;//random N
  int t =element.value;//random N
  for(int i=1;i<n;i++){
        add(head,t+increment(),element.arch);
  }
}
```

```
int increment(){
  int randomNr=rand()%(int)(500);
  return randomNr;
}

Tree * creatHeap(Tree *heap,float value){
  Tree *out=creatNode(value,rand()%archtecture);
  head=out;
  return out;
}

Tree * creatNode(float value,int arch){
  Tree *out=malloc(sizeof *out);
  out->parent=empty;
  out->left =empty;
  out->right=empty;
  out->value=value;
  out->arch=arch;
  out->self=&*out;
  return out;
}
int add(Tree *main, float item,int arch){
```

```
  int replacement=0;
if(item > head->value){
  replacement=item-head->value;
}else{
  replacement=head->value-item;
}
if(replacement>dynAvg){
  dynAvg=replacement;
}

Tree *itemT=creatNode(item,arch);
merge(main, itemT);
 return 0;
}

int merge(Tree *main, Tree *item){
  Tree *tail =  naiveMerge(main,item);
  swap(tail);
  return 0;
}

int swap(Tree *tail){
while((tail->parent) !=empty){//probelm infinity loop
```

```
  pCount++;
  tail=(tail ->parent);
  Tree *temp =(tail->left);
  (tail->left) = (tail-> right);
  (tail ->right)=temp;
}
//printf("# %d\n",pCount);
pCount=0;
  return 0;
}

Tree * naiveMerge(Tree *main1, Tree *item){
  Tree *container;
  Tree *main=malloc(sizeof *main);
  *main=*main1;
  //choose head
  if((main->value)>(item -> value)){

        //exchange item and main
        Tree *temp=main;
        main=item;
        item=temp;
        //head attach
```

```
        if((main->left)!=empty){
        main->left->parent=head;
  }
  if ((main -> right)!=empty){
        main->right->parent=head;

  }
  *head=*main;
  //free(main1);
  }else{
        if((main->left)!=empty){
        main->left->parent=head;

  }
  if ((main -> right)!=empty){
        main->right->parent=head;

  }
  *head=*main;
  //free(main1);
  }
container = head;
  while(1){
```

```
 //rest
 while((container->value)<=(item -> value)){
pCount++;
        //check last item
        if((container->right)==empty){//reason of infinity loop
        //position found
        //attach
        Tree *parent = container;
        //attach new tree
        (item->parent)=parent;
        (parent->right)=item;
        free(main);
        return parent->right; //return tail
        }else{
        container=(container->right);
 }
 }
 //position found
 //attach
 Tree *parent = (container->parent);
 //detch both child and parent
 (container->parent)=empty;
 (parent->right) = empty;
```

```
  //attach new tree
  (item->parent)=parent;
  (parent->right)=item;
  //prepare for next round
  item=container;
  container=(parent->right);
}
}

Tree pop(Tree *main){
  Tree *left=(main->left);
  Tree *right =(main->right);
  int leftFlag=0;
  int rightFlag=0;
pCount++;
  //detach head

  //detach left
  if((main->left)!=empty){
  (left->parent)=empty;
  (main->left)=empty;
  leftFlag=1;
}
```

```
//detach right
if ((main -> right)!=empty){
  (right->parent)=empty;
  (main->right)=empty;
  rightFlag=1;
}


  Tree target=*main;

  if(leftFlag==1&&rightFlag==1){
  merge(left,right);
  if(head->value==left->value){
          free(left);
  }else if(head->value ==right->value){
          //free(right);
  }
}else if (leftFlag==1&&rightFlag==0){
  //attach head to the next Item
  *head=*left;
  free(left);
}else if (leftFlag==0&&rightFlag==1){
  *head=*right;
```

```
  free(right);
}
//reset flag
leftFlag =0;
rightFlag=0;
  return target;
}
```

For Linked List(linkedlist.c):

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <assert.h>
//https://stackoverflow.com/questions/21788598/c-inserting-into-linked-list-in-ascending-order
typedef struct node{
        float data;
        struct node *ptr;
        struct node *ptr_p;
        struct node *tail;
        int arch;
} node;
```

```
int archtecture=1;
int maxTask =1;
int pCount;
node* insert(node* head, float num, int arch) {
        node *temp, *prev, *next,*tail;
        temp = (node*)malloc(sizeof(node));
        temp->data = num;
        temp->ptr = NULL;
        temp->arch = arch;
        //first
        if(!head){
        head=temp;
        tail=temp;
        } else{
        tail=head->tail;
        int avg =head->data+tail->data;
        avg=avg/2;//average
        prev = NULL;
        next = head;
        if(num>avg){//end
        prev = tail;
        next = NULL;
        while(prev && prev->data>num){
```

```
            next = prev;
            prev = prev->ptr_p;
            }
            if(!next){
            prev->ptr = temp;
            temp->ptr = next;
            temp->ptr_p=prev;
            tail=temp;
            } else{
                  temp->ptr = prev->ptr;
                  temp->ptr_p = prev;
                  prev-> ptr = temp;
                  prev->ptr->ptr_p=temp;
 }



}else{//front
            while(next && next->data<=num){
            prev = next;
            next = next->ptr;
            }
            if(prev) {
```

```
                temp->ptr = prev->ptr;
                temp->ptr_p = prev;
                prev-> ptr = temp;
                prev->ptr->ptr_p=temp;
        } else {
                temp->ptr = head;
                head->ptr_p=temp;
                head = temp;
        }



        }
        }
        head->tail=tail;

        return head;
}

node *trace=NULL;
node pop(node* head){
  node *tail=head->tail;
  trace =head->ptr;
  if(trace!=NULL){
```

```
        trace->tail=head->tail;
  }
  node out = *head;
free(head);
pCount++;
  return out;
}


//free the malloced memory to avoid memory leakage
void free_list(node *head) {
        node *prev = head;
        node *cur = head;
        while(cur) {
        prev = cur;
        cur = prev->ptr;
        free(prev);
        }
}

int increment(node* head){
  int avg =500;
  int out=rand()%(avg+1);
```

```
  //printf("%d , %d\n",out,avg);
  return out ;
}

int decompose(node* head,node element){
  element.arch=element.arch-1;
  int n =rand()%maxTask;//random N
  int t =element.data;// obtain value from main node
  for(int i=1;i<n;i++){
        head = insert(head, t+increment(head),element.arch);
  }
  return 0;
}


int main(int argc, char *argv[]){

  if(argc != 4) {
        printf("usage: list <maxTask> <nrEvent> <archtecture>\n");
        exit(0);
        }

        int num;
```

```
        maxTask =atoi(argv[1]);//decompose
        int n =atoi(argv[2]);//nrEvent
        archtecture=atoi(argv[3]);//arch
        int r =0;
        clock_t timestemp,t;
        node *head, *p;
        head = NULL;
        t=clock();
        srand(time(NULL));
        //add
        for(int i=0;i<n;i++) {
        timestemp =clock();
        head = insert(head, timestemp,archtecture);
        }
        p = head;

        //DEBUG
/*
        printf("\n#The numbers are:\n");
        while(p) {
        printf("#%f ", p->data);
        p = p->ptr;
        }
```

```
*/
        //pop
        while(head->ptr!=NULL){
        node poped_node=pop(head);
        head=trace;

        //DEBUG
/*
        printf("\n#POPED %f\n",poped_node.data );
        p = head;
        printf("#Current list:\n");
        while(p) {
        printf("#%f ", p->data);
        p = p->ptr;
        }
        printf("\n ");
*/


        //decompose
        if(poped_node.arch>0){
        decompose(head,poped_node);
```

```
 }


 //DEBUG
/*
 p = head;
        printf("#Current list:\n");
        while(p) {
        printf("#%f ", p->data);
        p = p->ptr;
        }
        printf("\n ");
*/
        }

//debyg
        // printf("#LAST ONE!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");

        node poped_node=pop(head);//last
//debug
        //printf("\n#POPED %f\n",poped_node.data );

        t=clock()-t;
```

```
	float effeciency=((float)t)/CLOCKS_PER_SEC*1000;
	int current= n;
	printf("%d\t%.8f\n", current, effeciency);
	return 0;
}
```

Appendix 2: Source code for the tests (validation)

In heap.c:

```
int debug(Tree *head) {
	if(head == NULL){
//	printf(" \t ");
printf("# dynavg value %d\n",dynAvg);
	return 0;
	}
	//printf(" \t ");
	debug(head -> left);
	//printf(" \t ");
	printf("# |____");
	printf(" %f\n",head -> value);
	//printf(" \t ");

	debug(head ->right);

}
```

```
main():
/*
  printf("\n#After head: \n");
  printf("#head is %f\n",head->value);
  debug(head);
  */
/*
   printf("\n#After decompose: \n");
   printf("head is %f\n",head->value);
   debug(head);
   */
/*
   printf("\n#After last pop: \n");
   printf("head is %f\n",head->value);
   debug(head);
   */
//printf("# not crashed");
```

In linkedlist.c:

```
 //DEBUG
/*
          printf("\n#The numbers are:\n");
          while(p) {
          printf("#%f ", p->data);
```

```
            p = p->ptr;
            }

*/
 //DEBUG
/*
            printf("\n#POPED %f\n",poped_node.data );
            p = head;
            printf("#Current list:\n");
            while(p) {
            printf("#%f ", p->data);
            p = p->ptr;
            }
            printf("\n ");
*/
//DEBUG
/*
  p = head;
            printf("#Current list:\n");
            while(p) {
            printf("#%f ", p->data);
            p = p->ptr;
            }
```

```
        printf("\n ");
*/
        }

//debyg
        // printf("#LAST ONE!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
//debug
        //printf("\n#POPED %f\n",poped_node.data );
```

Appendix 3: Source code for the experiments

For define gnuplot graph and statistic analyze(heap.p):

```
 # Gnuplot script file for plotting data in file "heap.dat"
set terminal png
set output "heap.png"

set terminal png linewidth 1 size 1360,768  font verdana 24



# This is to set the color
set style line 1 lc rgb "black" lw 1 pt 1
set style line 2 lc rgb "red" lw 1 pt 1

set title "The pefromence of using skew heap vs linked list as
pirority queue"
```

```
set key left top

set xlabel "Number of Application"

set ylabel "Time cost"

data1 = "<( paste */heap.dat )"

data2 = "<( paste */linkedlist.dat )"

f1(x)=a1*x+b1

a1=1

b1=1

fit f1(x) data1 u 1:(($2+$4+$6+$8+$10+$12+$14+$16+$18)/9.0) via
a1,b1

f2(x)=a2*x+b2

a2=1

b2=1

fit f2(x) data2 u 1:(($2+$4+$6+$8+$10+$12+$14+$16+$18)/9.0) via
a2,b2

stat data1 u 1:(($2+$4+$6+$8+$10+$12+$14+$16+$18)/9.0)

stat data2 u 1:(($2+$4+$6+$8+$10+$12+$14+$16+$18)/9.0)

plot data1 u 1:(($2+$4+$6+$8+$10+$12+$14+$16+$18)/9.0) w lp pt
6 ps 2 title "skew heap",f1(x) lc rgb "red" title "linefit for skew
```

heap",data2 u 1:(($2+$4+$6+$8+$10+$12+$14+$16+$18)/9.0) w lp pt 6 ps 2 title "linked list",f2(x) lc rgb "red" title "linefit for linkedlist"

For Start simulation(plot.sh):

```
#!/bin/bash

#plot

echo "This script runs a complexity comparision for two different piority queue implementaion linkedlist vs skewheap"

echo "preperation....."

cd plot

rm -rf 2

rm -rf 4

rm -rf 6

rm -rf 8

rm -rf 10

rm -rf 12

rm -rf 14

rm -rf 16

rm -rf 18

mkdir 2

mkdir 4

mkdir 6

mkdir 8

mkdir 10

mkdir 12
```

```
mkdir 14
mkdir 16
mkdir 18
cd ..
echo "compliling all component"
make
echo "SUCESSFUL COMPILED"
echo "simulation start"
./run2.sh 5 1000 5 > plot/2/linkedlist.dat
echo "1st run"
./run2.sh 5 1000 5 > plot/4/linkedlist.dat
echo "2nd run"
./run2.sh 5 1000 5 > plot/6/linkedlist.dat
echo "3rd run"
echo "linked list done"
./run1.sh 5 1000 5 > plot/2/heap.dat
echo "1st run"
./run1.sh 5 1000 5 > plot/4/heap.dat
echo "2nd run"
./run1.sh 5 1000 5 > plot/6/heap.dat
echo "3rd run"
echo "skew heap done"
echo "1st data pattern done"
```

```
./run1.sh 3  1000 6 > plot/8/heap.dat
echo "1st run"
./run1.sh 3  1000 6 > plot/10/heap.dat
echo "2nd run"
./run1.sh 3  1000 6 > plot/12/heap.dat
echo "3rd run"
echo "skew heap done"
./run2.sh 3  1000 6 > plot/8/linkedlist.dat
echo "1st run"
./run2.sh 3  1000 6 > plot/10/linkedlist.dat
echo "2nd run"
./run2.sh 3  1000 6 > plot/12/linkedlist.dat
echo "3rd run"
echo "linkedlist done"
echo "2nd data pattern done"
./run2.sh 6 1000 3 > plot/14/linkedlist.dat
echo "1st run"
./run2.sh 6 1000 3 > plot/16/linkedlist.dat
echo "2nd run"
./run2.sh 6 1000 3 > plot/18/linkedlist.dat
echo "3rd run"
echo "linked list done"
./run1.sh 6 1000 3 > plot/14/heap.dat
```

```
echo "1st run"
./run1.sh 6 1000 3 > plot/16/heap.dat
echo "2nd run"
./run1.sh 6 1000 3 > plot/18/heap.dat
echo "3rd run"
echo "skew heap done"
echo "3rd data pattern done"
echo "SIMULATION SUECCESSED,PLOTING GRAPH....."
cd plot
gnuplot heap.p
echo "plot done!"
echo "open graph"
display heap.png
echo "not crashed!"
```

For Linked List Experiment start Trigger(run1.sh):

```
#!/bin/bash
#plot

counter=1
num=$(echo "$2+1" |bc )
while [ $counter -le $num ]
do
```

```
./build/heap.o $1 $counter $3
counter=$(echo "$counter+100" |bc )
done
```

For Skew Heap Experiment start Trigger(run2.sh):

```
#!/bin/bash
#plot

counter=1
num=$(echo "$2+1" |bc )
while [ $counter -le $num ]
do
./build/linkedlist.o $1 $counter $3
counter=$(echo "$counter+100" |bc )
done
```

Appendix 4: Raw data from the experiments

Data pattern:5 1000 5

First run:

Skew heap:

1 0.03700000

101 0.09300000

201 0.14500000

301 0.19900000

401 0.31599998

501 0.40200001

601 0.42299998

701 0.42999998

801 0.46700001

901 0.54299998

1001 0.62900001

Linked List:

1 0.04000000
101 0.38300002
201 0.99700004
301 1.96600008
401 3.05699992
501 4.03999996
601 6.20499992
701 7.96700001
801 10.36699963
901 13.67199993
1001 16.03299904

Second run:

Skew Heap:

1 0.04700000
101 0.10600000
201 0.15599999

301 0.21499999
401 0.26100001
501 0.31400001
601 0.37200001
701 0.42500001
801 0.47400001
901 0.55100000
1001 0.58499998

Linked List:

1 0.03900000
101 0.24300000
201 0.63700002
301 1.38699996
401 2.67999983
501 3.51200008
601 5.78900003
701 7.92600012
801 9.85699940
901 12.69000053
1001 14.39299965

Third run:

Skew Heap:

1 0.05000000
101 0.12000000
201 0.15900001
301 0.21600001
401 0.25999999
501 0.31199998
601 0.37799999
701 0.42900002
801 0.47499999
901 0.56000000
1001 0.57900000

Linked List:

1	0.04000000
101	0.23099999
201	0.65499997
301	1.31299996
401	2.52999997
501	3.48199987
601	5.27500010
701	7.82299995
801	10.16199970
901	12.37500000
1001	15.32400036

Data Pattern: 3 1000 6

Forth run:

Skew Heap:

1	0.03800000
101	0.11600000
201	0.20199999
301	0.25099999
401	0.26999998
501	0.30100000
601	0.35900000
701	0.42900002
801	0.47400001
901	0.52100003
1001	0.58099997

Linked List:

1	0.04400000
101	0.09200000
201	0.16500001
301	0.21400000
401	0.30500001
501	0.34500000

601 0.41299999
701 0.50800002
801 0.61900002
901 0.65900004
1001 0.76999998

Fifth run:

Skew Heap:

1 0.03800000
101 0.10300000
201 0.16300000
301 0.25400001
401 0.32800001
501 0.31199998
601 0.36100000
701 0.41600001
801 0.47000000
901 0.51800001
1001 0.57700002

Linked List:

1 0.03800000
101 0.09999999
201 0.15200000
301 0.21699999
401 0.27399999
501 0.34299999
601 0.42399999
701 0.49000001
801 0.59299999
901 0.74799997
1001 0.74000001

Sixth run:

Skew Heap:

1 0.03900000
101 0.09600000
201 0.15300000
301 0.20199999
401 0.37600002
501 0.30899999
601 0.42900002
701 0.42200002
801 0.46800002
901 0.51999998
1001 0.57600003

Linked List:

1 0.04600000
101 0.09100000
201 0.14700000
301 0.21699999
401 0.28299999
501 0.34400001
601 0.42500001
701 0.49599999
801 0.56900001
901 0.67799997
1001 0.92800003

Data Pattern: 6 1000 3

Seventh run:

Skew Heap:

1 0.04600000

101 0.09400000

201 0.14200000

301 0.20400000

401 0.30399999

501 0.32400000

601 0.50000000

701 0.42100000

801 0.46199998

901 0.50500000

1001 0.63400000

Linked List:

1 0.03600000

101 0.26699999

201 0.78100002

301 1.60800004

401 3.39299989

501 5.90599966

601 8.91699982

701 10.78599930

801 14.91600037

901 19.92600060

1001 23.50699997

Eighth run:

Skew Heap:

1 0.07100000

101 0.13100000

201 0.16000000

301 0.24500000

401 0.25400001

501 0.36000001

601 0.38300002

701 0.42600000

801 0.47999999

901 0.53299999

1001 0.58700001

Linked List:

1 0.03500000

101 0.25900000

201 0.74799997

301 1.73100007

401 3.45900011

501 5.50699997

601 9.94299984

701 11.07100010

801 14.25100040

901 21.04800034

1001 22.83499908

Ninth run:

Skew Heap:

1 0.05500000

101 0.12199999

201 0.15799999

301 0.21100001

401 0.26199999

501 0.32400000

601 0.37700000

701 0.42100000

801 0.48799998

901 0.53399998

1001 0.57900000

Linked List:

1 0.03600000

101 0.25299999

201 0.70700002

301 1.69099998

401 3.61100006

501 5.52099991

601 8.89299965

701 11.41200066

801 14.00199986

901 19.35300064

1001 24.80500031

Appendix 5: Statistic analyze of the result from gnuplot

iter chisq delta/lim lambda a1 b1

0 3.8672803678e+06 0.00e+00 4.19e+02 1.000000e+00 1.000000e+00

1 7.3133285317e+03 -5.28e+07 4.19e+01 4.270434e-02 9.986168e-01

2 2.7651585647e+00 -2.64e+08 4.19e+00 -7.869677e-04 9.968782e-01

3 1.9899851206e+00 -3.90e+04 4.19e-01 -6.036981e-04 8.546288e-01

4 6.8399163053e-03 -2.90e+07 4.19e-02 4.730447e-04 1.002625e-01

5 1.2469645857e-03 -4.49e+05 4.19e-03 5.333097e-04 5.804081e-02

6 1.2469628336e-03 -1.41e-01 4.19e-04 5.333434e-04 5.801716e-02

iter chisq delta/lim lambda a1 b1

After 6 iterations the fit converged.

final sum of squares of residuals : 0.00124696

rel. change during last iteration : -1.40506e-06

degrees of freedom (FIT_NDF) : 9

rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 0.0117708

variance of residuals (reduced chisquare) = WSSR/ndf : 0.000138551

Final set of parameters Asymptotic Standard Error

========== ==========

a1 = 0.000533343 +/- 1.122e-05(2.104%)

b1 = 0.0580172 +/- 0.006649 (11.46%)

correlation matrix of the fit parameters:

a1 b1

a1 1.000

b1 -0.846 1.000

iter chisq delta/lim lambda a2 b2

0 3.7909667965e+06 0.00e+00 4.19e+02 1.000000e+00 1.000000e+00

1 7.2101208892e+03 -5.25e+07 4.19e+01 5.220183e-02 9.985937e-01

2 4.3805399423e+01 -1.64e+07 4.19e+00 9.147246e-03 9.931971e-01

3 3.5969030807e+01 -2.18e+04 4.19e-01 9.773803e-03 5.405178e-01

4 1.5885744251e+01 -1.26e+05 4.19e-02 1.320031e-02 -1.860097e+00

5 1.5829104499e+01 -3.58e+02 4.19e-03 1.339209e-02 -1.994458e+00

6 1.5829104482e+01 -1.12e-04 4.19e-04 1.339220e-02 -1.994534e+00

iter chisq delta/lim lambda a2 b2

After 6 iterations the fit converged.

final sum of squares of residuals : 15.8291

rel. change during last iteration : -1.12091e-09

degrees of freedom (FIT_NDF) : 9

rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 1.32619

variance of residuals (reduced chisquare) = WSSR/ndf : 1.75879

Final set of parameters Asymptotic Standard Error

==========
==========

a2 = 0.0133922 +/- 0.001264 (9.442%)

b2 = -1.99453 +/- 0.7491 (37.56%)

correlation matrix of the fit parameters:

	a2	b2
a2	1.000	
b2	-0.846	1.000

* FILE:

Records:	11	
Out of range:	0	
Invalid:	0	
Blank:	0	
Data Blocks:	1	

* COLUMNS:

Mean:	501.0000	0.3252
Std Dev:	316.2278	0.1690
Sample StdDev:	331.6625	0.1772
Skewness:	0.0000	-0.0944
Kurtosis:	1.7800	1.8433
Avg Dev:	272.7273	0.1448
Sum:	5511.0000	3.5774
Sum Sq.:	3.86101e+06	1.4776
Mean Err.:	95.3463	0.0510

Std Dev Err.:	67.4200	0.0360
Skewness Err.:	0.7385	0.7385
Kurtosis Err.:	1.4771	1.4771
Minimum:	1.0000 [0]	0.0468 [0]
Maximum:	1001.0000 [10]	0.5919 [10]
Quartile:	201.0000	0.1598
Median:	501.0000	0.3287
Quartile:	801.0000	0.4731

Linear Model:	y = 0.0005333 x + 0.05802
Slope:	0.0005333 +- 1.122e-05
Intercept:	0.05802 +- 0.006649
Correlation:	r = 0.998
Sum xy:	2379

* FILE:

Records:	11
Out of range:	0
Invalid:	0
Blank:	0
Data Blocks:	1

* COLUMNS:

Mean:	501.0000	4.7150
Std Dev:	316.2278	4.4016
Sample StdDev:	331.6625	4.6164
Skewness:	0.0000	0.6669
Kurtosis:	1.7800	2.1136
Avg Dev:	272.7273	3.8063
Sum:	5511.0000	51.8646
Sum Sq.:	3.86101e+06	457.6546
Mean Err.:	95.3463	1.3271
Std Dev Err.:	67.4200	0.9384
Skewness Err.:	0.7385	0.7385
Kurtosis Err.:	1.4771	1.4771
Minimum:	1.0000 [0]	0.0393 [0]
Maximum:	1001.0000 [10]	13.2594 [10]
Quartile:	201.0000	0.5543
Median:	501.0000	3.2222
Quartile:	801.0000	8.3707

Linear Model: y = 0.01339 x - 1.995

Slope:	0.01339 +- 0.001264
Intercept:	-1.995 +- 0.7491
Correlation:	r = 0.9621
Sum xy:	4.072e+04

Appendix 6: README.md

Pioritory_Queue_ENGskill

TravisCI:https://travis-ci.org/GiantPanda0090/Pioritory_Queue_ENGskill/builds

SonarQube:
https://sonarcloud.io/dashboard?id=Pioritory_Queue_ENGskill

This repository is meant to compare the complexity of 2 type of Piority queue implementation which is skew heap and Linked List. The simulation will run with 3 input data pattern and each data patten will run 3 times. Eventually the result will be produced as the mean of the 3*3 simulation.

How to Compile:

Run make in the repository root path

How to run simulation:

Run ./plot.sh and wait for simulation complete. The result will be output into the /plot/ folder and named as heap.jpg.

or

If only need the create the graph under previous simulation results under plot folder, run ./just_plot.sh.

Currently awarded issue:

1.Skew heap implementation has memory leak

2.SonarQube issue has not been fixed yet

3.Report is pending.....