Soft sequence heaps - Theory and experimentation

Peter Ellerup Frank, 201504938

Lars G. S. Lundqvist, 201504741

Master's Thesis, Computer Science

June 2020

Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Soft heaps are heaps that are allowed to return wrong answers some of the time. If an element is stored with an increased key then we call it a *corrupt* element. The number of allowed corruptions in the soft heap a any given time is εN, where N is the number of inserts and for a fixed $0 < \varepsilon \leq \frac{1}{2}$. Corruptions happens in what is called *car-pooling* [4]. Here multiple elements are grouped together under one representative. This is what enables soft heaps to beat the theoretical time bounds a normal heap has.

The first soft heap was introduced by Chazelle [6] and its purpose was to solve the MST [5]. Later, Kaplan et al. [13] introduced a new version with tighter analysis. Kaplan et al. [14] introduces how a number of selection problems can be solved using soft heaps. Brodal [3] tried to further simplify the construction by basing it on sorted sequences. All of these soft heap implementations achieve amortized constant time for each operation, but depending on the variant either EXTRACT-MIN or INSERT will be affected by ε.

Before implementing soft heaps we do a number of tests on lists to understand what is important when working with hardware. We describe and see the effects of the cache and vectorization. We see the importance of the density of the structure, and we see that array based data structures can be very efficient compared to a node based data structure.

We implement the non-lazy and lazy version of simplified soft heaps, and implement soft sequence heaps. Using these heaps we do a number of experiments. We learn that the real number of corruptions does not come close to the limit. We also see that for our implementation, simplified soft heaps performs better in inserts, but soft sequence heaps perform better when extracting elements.

We implement and test two applications of soft heaps, finding the k-th smallest element and heap selection. We also implement some non-soft solutions to see if it makes sense to use a soft implementation. We find that while the soft heap can find a good candidate median when finding the k-th smallest element, the price is simply too high. A simple randomized algorithm that can utilize the hardware better performs much better. For heap selection we implement an algorithm that uses a soft heap as priority queue and one that uses a normal heap. The latter outperforms the first consistently and interestingly we observe that applying more corruptions does not give a better running time under certain circumstances.

One of the disadvantages of using soft heaps as part of an algorithm is that instead of just making some calculation one has to setup the data structure by inserting data into it. This take some time, especially if you compare to another algorithm that can work on the data in place.

Contents

List of Figures

1 Introduction

The project is made as a group, all in the group are responsible for all of the thesis. What referenced material we have read, use, and in extension are responsible for can be seen in the Curriculum section which comes after Bibliography.

Our project is an experimental study of soft heaps with focus on soft sequence heaps. We will cover the theory, but we also want to test it with experiments because theory will only take you so far. We hypothesize that soft heaps will be faster in some areas because of car-pooling which decreases the number of items that needs to be compared.

Heaps, also known as priority queues, are a common data structure that can contain a number of elements. Each element will have a key, that comes from a totally ordered universe. We can query the heap at any point and get the smallest element in the heap. Other common operations that can be performed on a heap includes insert, extracting the minimum element, and melding — combining two heaps into one, preserving all the elements.

A heap can be used for many purposes, but its performance is limited by the same theoretical bound as sorting. This means we cannot insert and extract all elements faster than $\mathcal{O}(N \lg N)$ in the comparison based model, where N is the number of insertions.[1]

This is where soft heaps enter the picture. A soft heap is a heap, but it is allowed to return the wrong answer some of the time in a controlled manner. Because soft heaps are allowed to be wrong from time to time, allows us to improve the speed of the data structure. We measure if an answer is wrong by whether it is *corrupt* or not. Corrupt elements are elements that are stored in the structure according to a key, that is not its original key. To bound the number of corruptions allowed in the soft heap, we first select a variable $0 < \varepsilon \leq \frac{1}{2}$. The number of corruptions allowed at any given point is εN. This means we can decide how much corruption we allow. It is important to notice that as we remove elements, N does *not* decrease. This means we can be in a case, where every element currently inside the soft heap could be corrupt. The different operations on the soft heap will all be amortized $\mathcal{O}(1)$ time after having selected an ε value. This value will however affect the performance. Picking a value too close to zero will result

[1] We use lg to denote the binary logarithm

in performance similar to a non-soft solution.

The first soft heap implementation and the concept was introduced by Chazelle [6]. He combined binomial trees and car-pooling. Car-pooling was a concept Chazelle [4] introduced earlier, and is one of the main parts that make soft heaps possible. The idea of car-pooling is to pool elements together and move them as a single element inside the data structure. This means you can save a lot of comparisons since you only compare one element from each pool. This first implementation achieves amortized time bound of $\mathcal{O}(\lg \frac{1}{\varepsilon})$ for INSERT and $\mathcal{O}(1)$ time for all other operations.

The next milestone in soft heap developments was made by Kaplan and Zwick [12] which introduced a simpler version of Chazelle's soft heap, this time using binary trees and car-pooling. This have the same running time as the original by Chazelle. Later this was improved by Kaplan et al. [13]. They call the implementation soft heaps simplified. The improvement is done in the analysis. Their analysis gives the amortized time $\mathcal{O}(\lg \frac{1}{\varepsilon})$ per deletion and $\mathcal{O}(1)$ for all other operations. This is preferred over having the time complexities the other way around. The authors state that this is a simpler, improved analysis compared to the original soft heap by Chazelle. We will often reefer to this implementation as simplified soft heap. There also exists a version with lazy inserts presented in [14], this version fulfils a stronger interface described in the same paper.

After Kaplan et al. [13] a new version of soft heaps was introduced by Brodal [3]. Brodal's version is based on sorted sequences and car-pooling. This implementation gives the same running times as Chazelle [5] which were amortized $\mathcal{O}(\lg \frac{1}{\varepsilon})$ time for INSERT and $\mathcal{O}(1)$ for everything else. Brodal in [3, Section 4] introduces an alternative version of [13] that avoids lazy insertions and double even fills where fill is an operation that is specific to that implementation. This version is based on ternary trees. This version has the same running time as Kaplan et al. [13], where DELETE-MIN takes $\mathcal{O}(\lg \frac{1}{\varepsilon})$ amortized time and everything else is $\mathcal{O}(1)$ amortized time.

Applications Introducing a new data structure in itself is not that interesting. It is what you can use it for that is most interesting. Chazelle developed the original soft heap to solve the minimum spanning tree (MST) problem. He describes how to solve it in [5]. He is able to solve the problem in $\mathcal{O}(m \cdot \alpha(m, n))$ time where α is the inverse of Ackermann's function, n is the number of vertices, and m is the number of edges. This function is deterministic compared to earlier randomized linear time algorithms [15]. Later Pettie and Ramachandran [16, 17] made an algorithm for MST that have optimal number of comparisons using soft heaps. The original goal for Chazelle was to solve the MST problem, but he also mentioned other use cases. For example finding exact or approximate medians.

Other applications includes the ones introduced by Kaplan et al. [14], which solves a lot of selection problems using soft heaps. These applications includes finding k-th smallest element, finding the k smallest elements, heap selection, and others. Here he also introduces a new interface where you have to return which elements are newly corrupt when you call DELETE-MIN. This also dictates that you can only corrupt elements when DELETE-MIN called. In addition, you shall also return the original key, the element had then it was inserted. This can always just be added to the element and should therefore not be a program. The authors also describes how to update his simplified soft heap to fit this new interface as it originally did not.

Alternative computational models Thorup et al. [22] describes how you can do it in non-comparison based RAM model. Losing the comparison restriction can improve the time. There also exist a soft heap implementation focused on the I/O model. This one is made by Bhushan and Gopalan [1].

1.1 Report structure

Our project is an experimental study of soft heaps where we do experiments that focuses on soft heap implementations and also experiments that focus on some soft heap applications. First we cover the theory in Chapter 2 of the different concepts we are going to use. This includes theory around all the data structures we will see, from lists to heaps to soft heaps. In the section we also cover the theory of the applications we will be testing. Chapter 3 covers information about our test setup and the tools we will be using. This is useful for the upcoming experiments. We look at the performance of different C++ implementations of a list in Chapter 4. The purpose of this is to get a better understanding of how we can utilize the hardware when implementing soft heaps. We will translate the theory we saw for the soft heaps into C++ implementations in Chapter 5. This is followed by a number of benchmarks in Chapter 6 where we test how they perform. After getting some understanding for how they perform we move on to testing them in some applications. The first application is finding the k-th smallest element in Chapter 7, followed by heap selection in Chapter 8. Finally we end with a conclusion in Chapter 9.

Our contributions The contributions of the report is giving a more explicit and easy to understand explanation of soft heaps in general, particularly simplified soft heaps and soft sequence heaps. Furthermore, we present experimental results which there are not many of in the field of soft heaps.

2
Theory

In the chapter we will cover the theory about the different concepts we will be using. We start by introducing the comparison based RAM model, which will be the one we will be using. This is followed by an explanation of $\mathcal{O}$-notation and related notations. We see a simple example of an analysis using $\mathcal{O}$-notation and then go more in depth with how to do an amortized analysis. Next we cover some theory around lists, since we are going to do experiments with them later. We introduce array based and node based lists. After that, we cover how the lists perform under a number of operations that we will see used inside the soft heap implementations. We also prove that inserting into an array based solution is amortized $\mathcal{O}(1)$ time.

Before we get to the soft implementations, we introduce the building blocks used inside of them. This means we quickly cover binary heaps, binomial heaps, and sequence heaps. After covering the building blocks we only need one more ingredient to make the soft implementations, and that is car-pooling. In the car-pooling section we cover what it is in general and show examples of how it works.

At this point we are ready to tackle the theory around the soft heaps. We shortly talk about the original soft heap by Chazelle, but the focus is the two newer implementations, soft heap simplified and soft sequence heap. For both of these we prove that they do not exceed the corruption limit and that they have the correct running times. The last concepts we cover are the theory of the applications that we later make experiments about.

The scope of our project is limited to single-threaded, non-blocking data structures under the RAM model.

2.1 Computational model

To make an analysis of any algorithm we first have to decide on a computational model. A computational model is a formal model for what operations can be performed and what their cost is.

RAM Model A commonly used model and the model we are going to use is called the *Random Access Machine* model, or *RAM* model for short. This model helps us make machine and language independent analyses of algorithms and can be used to argue about the performance of algorithms.

In the model simple operations take one time step. This includes simple mathematical operations such as plus, minus, and multiplication. It also includes assignments and control flow operations such as function calls or if-statements. This however does not mean that you can execute the whole if-statement in 1 time step, you still need to consider all the individual operations that make up the if-statement. This also applies for while-loops. The model also makes you able to perform memory access in one time step, with unlimited memory.

The model is relatively simple which makes it easier to work with, but also might not capture all the characteristics of a modern computer. Most of the time the RAM model should not give you drastically misleading results. The RAM model is often used together with $\mathcal{O}$-notation to describe the performance of algorithms as the input grows.

Comparison-based In the comparison-based model the only operation you can do on two elements is comparison. Comparing two elements will then tell you which of the elements are greater or if they are equal. In a non-comparison based model you can use information from the elements. An example of this could be radix sort. Here we can use the fact that we want to sort a number of integers and can make decision based on the what the integers are, instead of just which is greater or smaller. We use the comparison-based model in our analysis. This is the most applicable since it does not make any assumptions about what the key is. This means the key could be a simple integer, but it could also be a more complex object.

Other models There exists other models of computations that you can use, that can be relevant in certain cases. An example of this is the I/O model. The focus of this model is to measure how much data needs to be go between the hard disk and RAM. It works under the assumption that because reading from the disk is so slow, it becomes the bottleneck in the system. All other operation are instant in this model.

Another model is one described in [11]. Here a model called the virtual address translation model is described. It considers the cost of address translation. The motivation for its creation was that the authors realised the running time of algorithms did not match the analysed time. They noticed that algorithms that did random access would not follow the analysed time, while sequential access algorithms world.

2.2 Big O-notation

In essence, $\mathcal{O}$-notation is a mathematical notation used to describe how mathematical functions change as the input gets larger. The formal definition states that given two functions f and g then for the statement $f(n) = \mathcal{O}(g(n))$ to be true, there needs to be some n_0 and some positive c such that forall $n \geq n_0$ it holds that $|f(n)| \leq c \cdot |g(n)|$. Intuitively, this means as n gets sufficiently large then g will always be equal or larger than f assuming you can apply a fixed scaling c. We can write this as one expression:

$$f(n) = \mathcal{O}(g(n)) \iff (\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{R}^+ \forall n \in \mathbb{R} : n \geq n_0 \implies |f(n)| \leq c \cdot |g(n)|)$$

This means we can write true statements such as $n + cn^2 = \mathcal{O}(n^2)$ or $\lg n + c + c \lg \lg n = \mathcal{O}(\lg n)$ where c can be any constant. Mathematicians usually write the double implication as seen above where the input is from $\mathbb{R}$. In computer science the input is often the number of elements, or the size in bytes. One cannot have half an element, therefore $\mathbb{N}$ is used instead. We write the version favoured by mathematicians because we are going to need a variable ε in some of our $\mathcal{O}$-notation estimations and ε is from $\mathbb{R}$.

Usage in computer science Using the RAM model assumptions, we can make a function that describes the time an algorithm takes to complete. One of the assumptions states that simple operations take constant time. This means that as the input increases these constants become irrelevant, leaving only the important information about how many times something specific is repeated based on the input. This is exactly what $\mathcal{O}$-notation will capture. It will capture the term that is most affected by an increase in the input size. It also makes it such that we do not have to count specific the number of operations, but can focus more on the control flow. For anything but the smallest inputs the algorithm with the lowest $\mathcal{O}$-notation, will most likely perform better. We can measure the input either in the number of bytes, the number of elements, or something third. For lists a natural thing is to use the number of elements.

We can see how we can use this for an algorithm that operates on a list. The algorithm might take longer as the list gets larger. $\mathcal{O}$-notation can quickly tell us if this is the case or if the performance is unaffected by the length of the list. By analyzing the code we might be able to make statements such as: "This operation takes $\mathcal{O}(n)$ time," where n refer to the number of elements in the list. Here $\mathcal{O}(n)$ means that it takes at most linear time in n to complete the algorithm. In this case n could have been any number of expressions like $\lg n$, n^2, n^n, or just 1. $\mathcal{O}(1)$ means that the operation takes a fixed constant time to complete in the worst-case.

Some limitations of $\mathcal{O}$-notation are that it only captures the main complexity. That is, any constant multiplication of what we have in the $\mathcal{O}$-notation can be discarded since the notion of $\mathcal{O}$-notation already contains a constant multiplier that could just be changed. We also have that we only get the part of the expression with the biggest complexity as we saw in the earlier example where $n + cn^2 = \mathcal{O}(n^2)$. This limitation might make it seem like the analysis might not be that useful, but we are okay with this is because as n increases the plus n part in the example becomes negligible relative to n^2. When comparing two functions performing the same task with e.g., a time complexity of $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, then we would prefer the function that has $\mathcal{O}(n^2)$ time complexity assuming the analyses are tight. Another limitation is that the analysis relies on the model we are using, which in our case is the RAM model.

One thing to consider is two different algorithms that perform the same task and both have the same $\mathcal{O}$-notation time complexity. These two algorithms might still differ quite drastically in actual performance. Here the constant might be important for your choice, or you might also start to consider how friendly the algorithm is to the hardware you are using. You might also look into any patterns there are in the input that might make you prefer one algorithm over the other. In some cases these considerations might also apply to things that have different complexities.

$\mathcal{O}$-notation is a good tool, but if you want to know the actual performance of a given implementation, then the only way to really test it is to do in practice. Here unexpected things can often happen because of how complex computers are. So far we have talked

about the time performance, but $\mathcal{O}$-notation can also be used when talking about the space complexity of an algorithm.

2.2.1 Analyzing an algorithm

We now have a good idea of what $\mathcal{O}$-notation is and what it tells us about a given function, but we do not know that much about how to calculate it from a given algorithm. To show how it works let us explain it with an example.

Finding index of element Imagine you have a function that takes a list containing n elements and an element to find the index of. The function returns the index of said element or -1 if the element is not in the list. In code, it could look like Algorithm 1.

Algorithm 1 Finding an element's index

```
1: function FindIndexOfElement(list, element)
2:     for all i ∈ 0, ..., list.size() − 1 do
3:         if list[i] == element then
4:             return i
5:     return −1
```

When analyzing the function we see that there are two things that can happen, either you will find the element and return the index, or you will go through the whole list without finding the element and return -1.

If you find the element, then the amount of work you have to do is bounded by $c + c_l \cdot n_c = \mathcal{O}(n_c)$, where c_l is the constant time it takes to complete one iteration of the loop, n_c is the number of times you have repeated the loop, and c for extra work outside the loop. We can write this as $\mathcal{O}(n_c)$ since we discard constants. Since n_c is only bounded by n in the worst-case, we would normally just write $\mathcal{O}(n)$ instead.

If we do not find the element then we are going to go through the whole list doing $c + c_l \cdot n + c_r = \mathcal{O}(n)$ work, where c_r is the constant work it takes to returning -1. This means that we are also bounded by $\mathcal{O}(n)$ in this case.

The time for an algorithm is defined to be the worst-case time over all inputs for a given size. Since we see that the operation takes $\mathcal{O}(n)$ in both cases, we say the time complexity of the function is $\mathcal{O}(n)$. If the two cases were not the same then we would have taken the one with the worst complexity, or you could write down what the complexity is under different circumstances.

2.2.2 Other notations

Besides $\mathcal{O}$ we also have some other notations we can use to tell something about how a function behaves. $\mathcal{O}$ tells us that the worst-case performance never exceeds the expression given to $\mathcal{O}$ when the input gets large enough and assuming you can apply a constant multiplier. Ω is the opposite of $\mathcal{O}$. As the input increases it says that after some point, the function f will be greater than g, again considering a constant multiplier that is positive. This means that Ω makes a lower bound for the worst-case. We can see an example of this:

$$n^4 + n^3 = \Omega(n^4)$$

Θ combines $\mathcal{O}$ and Ω into one. This means that it tells not only about the worst-case, but also the best-case. This is therefore a stronger statement. The constant in these two cases can be different, but have to be positive. Even if this is stronger, people will often still just use $\mathcal{O}$ since it is usually the upper bound on the performance we care about. We again can do an example of this:

$$n^4 + n^3 = \Theta(n^4)$$

The small *o*-notation tells us about the relation between two functions as they grow. If one function grows faster than the other, then you can use small *o*-notation to denote that. E.g., if you have two functions f and g and state that $f(n) = o(g(n))$ then what that means is that as n increases then $f(n)$ gets small compared to $g(n)$. In mathematical terms this means what we have:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

2.2.3 Amortized time

If we want to talk about the performance over a sequence of operations, just summing the worst-case of the individual operations can give an overestimate of the time needed. This happens because the operations might perform better than the worst-case some of the time. What we want is an analysis that considers how the operations affect the data structure and thereby each other. It needs to look at the algorithm as a whole instead of thinking about the operations in isolation.

We do this by talking about the running time per operation over a worst-case *sequence* of operations and not the individual operations. This kind of analysis is called an amortized analysis and is presented in the paper [21] which contains some ideas that we also use in our explanation.

We can imagine a function that has a fast running time most of the time when we call it, but occasionally something happens that makes it slow. An example could be that a pre-allocated array ran out of space, making the next insert call slow. Here a worst-case analysis might not be very telling about its performance, so we want to do an amortized analysis. The original worst-case analysis might tell you that the operation takes at most $\mathcal{O}(n)$ time to complete. If we do an amortized analysis, we might be able to show that the frequency of which it is slow, is so low that the average time over a sequence of operations is never worse than $\mathcal{O}(1)$ time per operation. This is what we would call $\mathcal{O}(1)$ amortized time.

Potential function To do amortized analysis we can define a potential function. This function will take the current data structure and then output the structure's potential. The potential is like a savings account for time. You can imagine that you are putting aside time as the potential increases to use later. This means that for fast, common operations we can add small amounts of time to our savings account, but not too much because we do not want to change the complexity of the fast calls. Later you might be in the situation where an expensive operation is called. At that point we want to lower the potential of the structure and then hopefully use the time released from our savings account to pay for the expensive operation. If we can show that we always have enough potential stored before the expensive operations, then we are able to complete

the amortized analysis. Overall, the expensive operation is not going to be performed any faster, but we can show that the expensive operation is so rare, that over a worst-case sequence of operations, the average performance never exceed a certain limit.

How it works in practice is that you have the potential function Φ that outputs the potential of a given structure D. This function needs to be designed on a case by case basis depending on what you want to prove and the given structure. The amortized time of a function is $t_a + \Phi(D') - \Phi(D)$, where t_a is the time it takes to perform the operation, D' is the structure after the operation, and D is the structure before the operation.

Example Imagine you have a list where there are two operations that can be performed. You can insert elements into to list, and you can perform an operation on all the elements, deleting them in the process. The potential function for this could be defined to be the number of elements n times some constant c in a structure D. This means we have $\Phi(D) = cn$. Thereby, every time we insert an element the potential will increase by c. Let us assume that insert can be done in constant time. This means that the amortized cost of inserting is the insert cost plus the change in potential. This means that the amortized cost of inserting is still $\mathcal{O}(1)$.

The other operation could be e.g., printing the sum while emptying the list. The real time this would take could be $\mathcal{O}(n)$; the number of elements times some constant for each element. To calculate the amortized cost of this operation we take the real cost, which is $\mathcal{O}(n)$, and add the change in potential. Here we will see that the change in potential is negative $\Omega(n)$. Therefore, the amortized time for this operation is constant.

This works because every insert makes the other operation slower, but at the same time adds potential to offset that when the other operation is called. This means that an amortized analysis will tell you that these two operations both use amortized $\mathcal{O}(1)$ time. All of this also means that you cannot simply do an operation by operation analysis.

2.3 Lists

A *list* is one of the most basic data structures. It is defined as a collection of elements that we can add and remove from. Unlike a set, a list has an order and it can have duplicates. Furthermore, we can access any element inside a list unlike e.g., a queue.

We go through the analysis needed to get the theoretical list time complexities because many data structures — including soft heaps — employ a lot of lists. With that said then this section is mostly written for the sake of completeness. The analysis is rather straightforward so the reader should go over this section lightly.

In Chapter 4 we test the real world performance, compare it against the theoretical time, and observe factors that effect the real world running time. By combining the knowledge gained from these two sections should give us a good foundation for making the most efficient soft heap implementations. An overview of the theoretical time complexities are in Table 2.1.

2.3.1 Classes of lists

We have identified two primary classes of lists to limit the number of lists we look at in this section. These two list classes cover most types of lists. Many variations on these

	Push back	Insert	Read	Iterate	Pop back	Delete	Merge	Concat.
Node	$\mathcal{O}(1)$	$\mathcal{O}(m)$*	$\mathcal{O}(m)$*	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$*	$\mathcal{O}(m)$	$\mathcal{O}(1)$
Array	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$

Table 2.1: Overview of the results we get in the section. *: given pointer to node then $\mathcal{O}(1)$

two classes can be made, but most often they will share many of the same performance characteristics for most of the operations.

Node based The first class is *node based* lists. Lists in this class consists of nodes that contain pointers to each other. In addition to the pointers, the nodes also contain the elements you are storing. Examples of this could be a singly-linked list or a doubly-linked list. In a doubly-linked list each node hold an element and two pointers, that point to the previous node and to the next node in the list. Often, access to a node based list is facilitated by having a pointer to the head of the list, and from there we can follow the pointers to get to the other elements. An advantage of the node based approach is that nodes can be allocated at any time and do not need to be at a specific place in memory.

Array based The second class is *array based* lists. This covers lists that are based on arrays and use elements' position in memory to denote their position in a data structure. This means that their position are not defined by pointers between each element, but are defined by their position relative to a fixed start position. Examples of this is C++'s native array or a dynamic sized array like C++'s `vector`. Access is done via a base pointer that points to the first element. Given the base pointer we can calculate an offset to any element easily.

An oblivious downside is that if an element needs to change position in the data structure, then it also needs to change position in memory since there is a one-to-one mapping. However, this one-to-one mapping also makes the data very densely packed compared to node based lists. We will also see that since arrays have a fixed size, there are some problems to be solved if you need your list to support dynamic sizes.

We note that a node based data structure can use an array to hold its elements, this however does not make it an array based solution in the sense that we have defined.

Classes for other data structures than lists Just as a side note we want to point out that the ideas that form these two list classes can also be applied to other data structures like trees. Making a tree can both be done by making nodes that holds pointers to the other elements, or you can define its position in a tree by cleverly defining how a position can map to a position in a tree.

2.3.2 Inserting elements

We will now describe some list operations and their theoretical running times. The first one we will look at is insert.

One of the advantages of *node based* lists is how each element's position is defined by a small set of pointers. These pointers are very localized; meaning if an element changes

position, then it often only needs to update the pointers of the elements right next to it. This also goes for inserting. If you have a pointer to where you want to insert, insertion can be as simple as changing a fixed number of pointers. This means that the insertion itself takes $\mathcal{O}(1)$ time. Normally a node based list will store a pointer to the beginning and maybe the end, so you can insert an element there in $\mathcal{O}(1)$ time. As noted, if you want to insert other places too, then you need a pointer to that place. This means that it can take up to $\mathcal{O}(m)$ to get a pointer to an arbitrary place.

Array based lists can hit a bit of a road block in more ways than one. If we assume that there is enough space, then inserting into the back of an array is the easiest insert to do. This only requires us to store the element at a given position. This can be done in $\mathcal{O}(1)$ time assuming you know the current size. Inserting anywhere other than the end is going to be a bit of a problem. Inserting in the middle will change the position of all elements after the element we insert. This means that since the position in the list denotes where an element is located in memory, then we need to change the position of a lot of elements. Inserting close to the end might not be that bad, but as we insert closer and closer to the beginning then the begin to need to move most of the array. This implies that the worst-case is that we need to move all the elements and insert therefore takes $\mathcal{O}(m)$ time in the worst-case. This becomes expensive if it happens often.

Dynamic arrays If you do not know how big the array needs to be when creating your array, then you need a dynamic array. Here dynamic means that the number of spaces in the array M can change. When we create an array we need to provide a fixed size, because that is how we allocate memory. This means we have a problem when that array gets full and we still want to insert elements. There is no option to easily extend an array. The solution chosen to solve this problem is often to move all the elements to a new array that have double the size of the original array.

Here it should be clear moving all the elements is going to be an *expensive operation* compared to the *cheap operation* that is simply adding an element to an array that have space. Let us analyze this and refer to these two cases as the expensive case and the cheap case. The reason we double the size is that we want to have a balance between the number of times we need to move all the elements and the amount of wasted space we are going to take up.

A worst-case analysis will tell us that inserting takes $\mathcal{O}(m)$ time where m is the current number of elements in the array. In the expensive case we calculate this from the fact that we need to create a new array of size $2m$, move all the current m elements to the new list, and insert the element we tried to insert originally. For the cheap case the analysis tells us it takes $\mathcal{O}(1)$ time. If we did a worst-case analysis over a sequence of insertions then this analysis would become overly pessimistic.

We can do better. It is clear that after we have hit the expensive case, then we will have $m - 1$ cheap inserts before another expensive insert comes around. The tool we can use to capture this important property is amortized analysis as we described in Subsection 2.2.3. Using this we are able to improve the analysis such that insert takes amortized constant time.

We first define the potential function Φ and then do a case analysis to show that we get the desired bounds. We define the potential function to be $\Phi = 2m - M$. We can confirm that we will never have negative potential if we only allow insertion. In the cheap case we see that each insertion of an element increases the potential function by

two. This means that it cannot become negative if it was not negative before. Before the expensive case is performed we know that $M = m$. Since we double the array we then know that the new size is $M = 2m$. Furthermore, we also add the element increasing m by one. This gives us $\Phi = 2(m + 1) - 2m$ which is clearly positive. This means that the potential is never negative.

Let us look at the amortized time the operation takes. For cheap case insert took $\mathcal{O}(1)$ real time. It increases the potential by a constant. This means the amortized time for this case is $\mathcal{O}(1)$. If we are in the expensive case then it took $\mathcal{O}(m)$ real time. If we look at the change is potential we see that we have increased the M by a factor two and only increased m by one. This means that the total change in potential is negative $m - 2$. This is enough to pay for this expensive operation and tells us that the expensive case is also amortized $\mathcal{O}(1)$ time. Since both cases of insert tell us that the amortized time is constant then we conclude that insert can be done in amortized $\mathcal{O}(1)$ time. Recall that we are only allowing insertions.

2.3.3 Reading elements

Depending on the index of the element you want to read, the node based list might perform poorly. The problem is that getting to the element can be hard. This is the same problem we saw with inserting where you need to traverse the whole list until you get to the index. The worst-case time is $\mathcal{O}(m)$.

Reading from an array based list is easy. As noted in the beginning, if you have the base pointer to the first element, know the size of each object, and the index you want; then we can perform a simple calculation to figure out where in memory to read. Furthermore, given a pointer to an element, then since a pointer is just an address, we can offset that address by the size of the object go get the next element in the array. Any array based list solution should do look up in $\mathcal{O}(1)$ time.

Reading all elements If we instead of just reading a single element look at how long it takes to read all the elements in a list, then node based lists can compete. The problem with node based lists was getting to the element, not reading it. Therefore, we can do this operation efficiently since the way we get to an element is by iterating through the elements that leads up to it. This implies that a node based list have the same time complexity for reading a single element as it have for reading all the elements, $\mathcal{O}(m)$. Array based lists will have the same time complexity. Here we need to read m elements, and each read takes $\mathcal{O}(1)$ time. Logically it is impossible to read m elements faster than $\mathcal{O}(m)$ time.

2.3.4 Deleting elements

Deleting an element mostly follows the same analysis as we saw for inserting. In the node based approach the hard part is still getting to the element. The actual deletion only takes constant time and is done by updating some pointers. Therefore, if we have a pointer to the element we want to delete, then it only takes $\mathcal{O}(1)$ time. If we do not have a pointer, then it will take up to $\mathcal{O}(m)$ time. This means that we can delete the first or last element in constant time.

Deleting elements in array based lists suffer from the same problems as array insertion. Deleting in the end can be done in constant time. If we delete any other place,

it will require us to move all the elements after the deleted element back one location. This gives us a worst-case time of $\mathcal{O}(m)$. If you want to delete from the front, then there exists a variation that allows this in $\mathcal{O}(1)$ time.

2.3.5 Merging

In the different implementations of soft heaps we will see two different operations used to combine two lists; merging and concatenating. Merging means taking two lists that are in order and making one list that is ordered. Concatenating means that after the elements in the first list comes all the elements in the second list.

Merging two sorted lists l_1 and l_2 in a node based list will use $\mathcal{O}(l_1.\text{size} + l_2.\text{size})$. This is done by having an iterator for each list and then merging them while iterating. When one of the lists runs out of nodes we can link the rest of the nodes in $\mathcal{O}(1)$ time. For an array based list we need to move all the elements, even after one list is empty. This uses $\mathcal{O}(l_1.\text{size} + l_2.\text{size})$ time.

2.3.6 Concatenation

Concatenating two lists will give you a single list where all the elements in the second list comes after all the elements in the first list. For any array based approach having to move all the elements of one of the lists is a must. If none of the lists can contain all the elements then we need to move all the elements to a new location, therefore $\mathcal{O}(l_1.\text{size} + l_2.\text{size})$ time. If space permits then we might be able to save some time since we do not need to move all the elements.

In amortized time we can bring this down to amortized constant time for each element in the smaller list which is $\mathcal{O}(\min(l_1.\text{size}, l_2.\text{size}))$. The reason for this is that we have constant look up and can therefore look up all the element of the smallest list linearly in the number of elements. Next we insert all the elements into the other list. From the analysis of insert we know this is amortized $\mathcal{O}(1)$ time per element.

Node based lists can do much better. Since it is all pointers we only need to update the pointers of the last element in the first list and the first element in the second list. No elements need to change position. This can all be done in $\mathcal{O}(1)$ time.

2.3.7 Space usage

Time complexity is not the only important performance benchmark. Space usage can also be important. We calculate space usage relative to the number of elements m currently in a given list. It makes sense that you will always need to at least store the elements you have in your list. This gives us a theoretical lower bound on the space that is $\mathcal{O}(m)$.

For the array based lists we need the space to store the data and then depending on the implementation we might have a size value stored somewhere. This implies that the total space usage is $\mathcal{O}(m)$. Node based lists clearly stores more data since it also needs to store pointers for each element. However, the number of pointers are fixed and therefore constant extra space per element. This makes it so that as long as the number of pointers are fixed per element the space usage is $\mathcal{O}(m)$.

Here $\mathcal{O}$-notation do not tell the full story since both list classes take $\mathcal{O}(m)$ space, but arrays can be way more dense. Like we noted with time performance, then when you

have two $\mathcal{O}$-notation-estimations with the same complexity, looking into the constants might be a good idea. Note that space usage will affect real world performance.

2.4 Non-soft heaps

The purpose of this section is to explain non-soft heaps before talking about how each of them can be converted into a original soft heap, a simplified soft heap, and a soft sequence heap, respectively.

A *heap* is defined as a tree that have the constraint that the children of a given node never have a lower value than their parent. This means that the root of the tree will always be the lowest element. This is why they can be used as implementations of priority queues. A heap has no requirements on the relation between siblings.

Heaps support the following operations. MAKE-HEAP will create a new empty heap, this will often be used indirectly. INSERT will insert into a given heap. MELD will combine two heaps into one. Then we have three functions that works on the minimum element. First is FIND-MIN, which will just return the minimum element. DELETE-MIN will delete the minimum element from the heap and not return anything. EXTRACT-MIN is FIND-MIN and DELETE-MIN combined. There is also a DELETE operation that can delete an arbitrary element given a pointer. We will not be focusing on this since the heap applications we look at does not use it. A simple way to implement it would be to do lazy deletion. When you call DELETE on an element, mark it as deleted using the pointer and then remove it once it reaches the minimum position.

2.4.1 Binary heaps

A binary heap consists of a single tree. The tree is made up of nodes that can have up to two children. The children are often refereed to as left and right child. It can also be generalized to a d-ary heap where instead of two children it can hold d.

A crucial part of the performance of the binary heap depends on the balance of the tree. If we extract the minimum element, then you have to *fill* the root. This is done by comparing the two children and then moving the content of which ever is smaller up. To fill the node you just emptied you recursively fill. The running time of this is the height of the tree. If the tree with B nodes is not balanced, this operation can take up to $\mathcal{O}(B)$ time. In a balanced tree we can limit this to $\mathcal{O}(\lg B)$.

2.4.2 Binomial heaps

This heap, instead of just being a single tree, consists of several heap-ordered trees. We call structures with several trees a forest. We get a single tree with one node when we insert a node into an empty binomial heap. A node with one element starts with rank 0. We only want one root of each rank. We can combine roots of rank $0, \ldots, k-1$ into a new tree of rank k. This is done simply by making a new node with rank k and making the nodes with rank $0, \ldots, k-1$ children of it. On Figure 2.1 we can see how each tree is made up of trees of smaller rank.

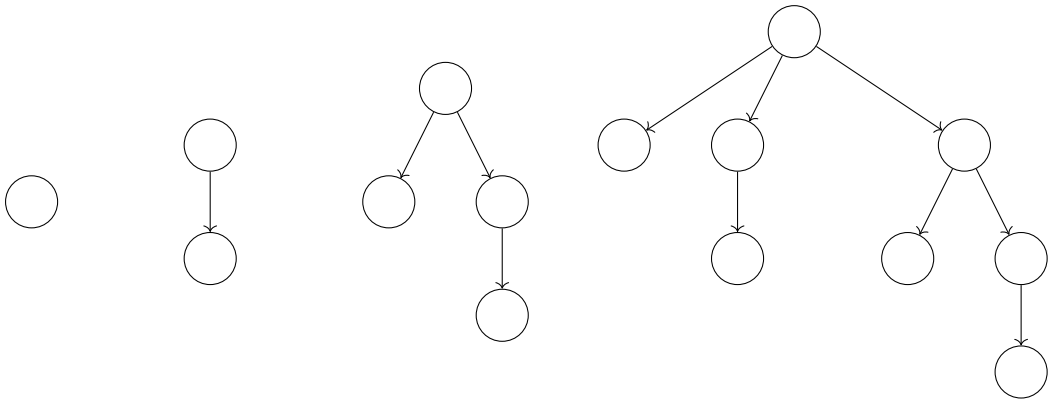


Figure 2.1: Here we see a number of binomial trees that can make up a binomial heap. From left to right we have trees of rank 0 to 3. Notice how each tree have a full copy of each tree of lower rank in its children.

2.4.3 Sequence heaps

Sequence heaps were coined by Sanders [18]. A sequence heap is a priority queue that is characterized by storing its items in a logarithmic number of sorted sequences $L_1, \ldots, L_\ell$. A sequence is said to have *rank* r if it contains 2^r items, also counting deleted items. So the rank of a sequence is not affected by EXTRACT-MIN and DELETE, but INSERT can change it as we will see in a moment. A list $\mathcal{L}$ stores the sequences in increasing rank order.

Given an item e then INSERT(e) makes a new sequence containing only e so its rank is zero. Zero is the smallest possible rank so the new sequence will be placed at the front of $\mathcal{L}$. If the first two sequences of $\mathcal{L}$ have equal rank then they are merged into a new sequence of rank $r + 1$. This is repeated until all sequences have distinct rank. EXTRACT-MIN removes the first item in the sequence containing the minimum item and returns the item. An example of INSERT and EXTRACT-MIN can be seen on Figure 2.2.

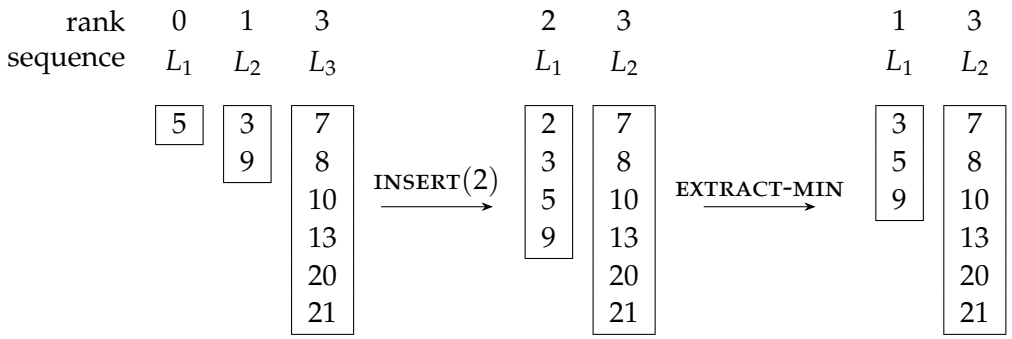


Figure 2.2: INSERT(2) creates a rank 0 sequence (2) which is merged with (5) resulting in $(2, 5)$. That sequence is merged with $(3, 9)$ which becomes $(2, 3, 5, 9)$. EXTRACT-MIN extracts the minimum item which is the head of $(2, 3, 5, 9)$.

Analysis

We will now prove the running time for the operations EXTRACT-MIN and the operation INSERT.

Lemma 2.4.1. *Sequence heap's* INSERT *and* EXTRACT-MIN *take amortized* $\mathcal{O}(\lg N)$ *time.*

Proof. The proof is done by making three observations and applying them. Recall the definition of rank of a sequence

$$\text{rank}(L_i) = r \iff \text{size}(L_i) = 2^r$$

When we have the size 2^r, then we can isolate the rank r like so $r = \lg 2^r$. We are doing amortized analysis so we are given a mixed sequence of operations and we say that N is the number of insertions that has occurred during the sequence of operations.

We will now determine an upper bound for the highest possible rank of a sequence. The largest rank comes from the sequence that has most items. The case where a sequence has the most items possible is the case where a sequence contains all items that has ever been inserted i.e., N. That means that the largest possible sequence contains N items. We find the rank by taking the logarithm of the size, so $\lg N$. The logarithm of N can give a decimal number if there are sequences of lesser rank. Therefore, we need to round down to get the highest possible rank. This gives us the first observation:

$$\text{rank}(L_i) \leq r_{max} = \lfloor \lg N \rfloor$$

Second observations is that we can bound the number of merges an item can participate in by r_{max}. This is done by looking at how INSERT works and noting that a merge will always increase the rank by one.

The third observation is that we can observe that by construction INSERT ensures distinct sequence ranks.

We now have the three observations and can therefore argue about the time complexity of INSERT. The cost of inserting an element is going to be proportional to the rank of the final sequence the element ends up in. Each time the element moves to a sequence of one rank higher it appears in one merge, where it takes $\mathcal{O}(1)$ time per element. This means that since the max rank is bounded by $\lfloor \lg N \rfloor$, then the total work is bounded by amortized $\mathcal{O}(\lg N)$. We need it to be amortized time since it might not get its final rank right away, and we therefore need to deposit the potential, so we can later withdraw it when we need to move it up in sequence rank.

Now we will argue for the time complexity of EXTRACT-MIN. EXTRACT-MIN have to look at the first element of each sequence to determine the minimum item. As previously observed the ranks are distinct so the largest rank can be used to make an expression of how many sequences there can be at most. The number of sequences is at most $\lfloor \lg N \rfloor + 1$. Therefore, EXTRACT-MIN takes amortized $\mathcal{O}(\lg N)$ time. □

2.5 Car-pooling

We just covered the normal non-soft heap implementation. These are the building blocks for the different soft heap implementations. In addition to these there is another important part that makes soft heaps work and that is car-pooling. In this section we will cover what car-pooling is, and give an example of it using linked lists.

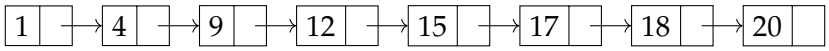


Figure 2.3: Example of a normal singly linked list containing integers and pointers to the next element in the list

One of the ideas that is going to be used to get the wanted running time in soft heaps is the idea of car-pooling. The term was coined in a report by Chazelle [4] from 1998 which was later made into a paper [6]. The basic idea is to combine the items you store in your structure into groups of items. A group, in combination with a representative, will be denoted as a *node* by us. A given node then acts as if all the elements in its group are of the key value of the representative. The effect of this is that some items do not moves according to their own key, we call these items corrupt. The performance increase comes from the fact that a node will move around the data structure based on the representative. This means that comparisons are only done with the representatives, thereby saving all the comparisons with the elements in the node's group.

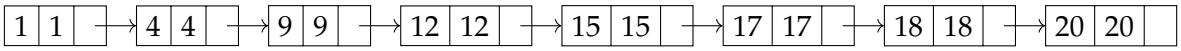
A data structure will perform faster if you build it around nodes containing multiple items instead of items directly. Sadly, there is also a downside to doing it; it is not free performance. Since we group elements which might not have the same keys we might introduce inconsistencies. The difference in speed and accuracy will be dependent on how much grouping you do and the data structure.

Soft heaps In the different soft heap variants the representative will be the item with the greatest key in the group of the node. As mentioned, we convert a structure into one using car-pooling by using nodes instead of items.

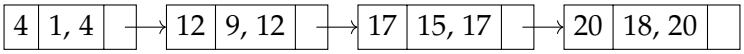
When moving things around it is important that we are oblivious to the contents of a node's group and only look at the representative. However, in some cases we might need to make changes such that we touch a node's group. In the case of a soft heap we need to change the operations that need to do something to a specific item i.e. EXTRACT-MIN and FIND-MIN. These operations need to return a single item and not a node. We also need to add additional code somewhere that is going to perform the car-pooling.

Example Car-pooling can be used on different data structures and is not limited to the binomial heap as seen in the original soft heap implementation Chazelle [6]. In the following example we use sorted lists as input, but the example can easily be expanded to nodes in a tree or something else. On Figure 2.3 we see a normal singly-linked list that is sorted and where we store the value of an entry and a pointer to the next element. The same list can also exist when our list needs to support car-pooling. On Figure 2.4a we see how this would look, here we converted each item into a node. But we also see that since no items are grouped yet there have yet to be any active car-pooling. The second field of each node is the *group* of the node. In this case the group containing only a single item. The first field is the representative. The last field still points to the next element.

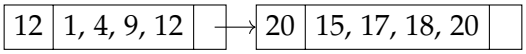
Currently we have not gained anything besides introducing linearly more space usage, this however changes once we do car-pooling. At certain points in the data structure you want to perform car-pooling. In the example we will now start doing some arbitrary car-pooling. We do this by combining the groups of two nodes and then choosing the right representative. We use the greater as the representative, just like the soft heap



(a) Example of a singly-linked list based on *nodes* instead of items. The first field is the representative. In the second field is the group of the node. Currently, there is only one item in the group. The last field is a pointer to the next value.



(b) Compared to (a), car-pooling has now been applied to every other element. The representative in the first field is now the greater of the two values in the set found in the second field



(c) Applying car-pooling one additional time on (b) we get even fewer elements

Figure 2.4: Examples of singly-linked-lists with various levels of car-pooling applied

variants. Finding the new greater can always be done in constant time by comparing the two representatives before the car-pooling. The lowest item can also be the representative, but if you want another representative then you might not be able to do it in constant time.

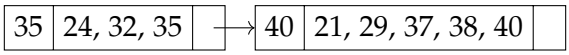
If we do car-pooling on the example we had on Figure 2.4a by combining every other node with the previous node, then we end up with the list we see on Figure 2.4b. We are of course not limited to only doing it once, it can be done as many times as wanted, as long as you have multiple nodes. Doing it a second time yields the result seen on Figure 2.4c.

Now, let us look at how you would extract items. On Figure 2.4b and Figure 2.4c we are able to extract an ordered list efficiently. Here we would simply take the first node and extract elements from its group until it is empty. At that point we would then move on the the next node and repeat. In the figures we have seen so far the groups have always been ordered, this however is not a guarantee; the items in the groups may not always be in order. This means that if you want it to give a fully ordered list, more work is needed to be done. Note that by the definition of car-pooling nodes are ordered by the representative.

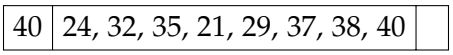
Unsorted corruption set Now we want to give an example of the situation where the corruption set is unsorted. Depending on the data structure, you can imagine you have multiple lists of nodes and need to merge these. When merging you merge such that you get an ordered list based on the representatives without looking at items inside the nodes' groups. After this step you might end up with a list of items that is not ordered if you were to extract them as before by just going though the groups inside the node.

A simple example of this, could be where you have two lists of nodes where each list only contains one node. Merging these two lists could give you a list as seen on Figure 2.5a. We see that they are ordered by the representatives, as required, but they are not ordered if you were to concatenate the two groups inside the nodes. This is where we pay for the increased speed in operations, by reduced accuracy. If we imagine we wanted to extract all the numbers from this list with the minimum element first, we would end up with a list containing $24, 32, 35, 21, 29, 37, 38, 40$, not exactly an ordered list. So this sequence of extractions which is incorrect orderwise is the negative consequence or trade-off of using car-pooling.

We further note that even the groups inside a node can be unordered. In the exam-



(a) After a merge with another list you might end up with a list looking like this. This list is valid since they are sorted according to the representative



(b) Applying car-pooling to (a) we see that the set is not guaranteed to be sorted as they were on the examples in Figure 2.4. Because we want to combine the two sets in constant time we cannot sort them

Figure 2.5: More examples of a singly-linked-link with various levels of car-pooling applied showing that the set does not have to be strictly ordered.

ples we have only seen ordered groups. When we do car-pooling we want item lists to be combined in constant time. A side effect of this is that we cannot sort the final list since this will take linear time in the number of elements in the longest list. This means that if you had used car-pooling where you combined two nodes in Figure 2.5a you would end up with a single node containing the item group $24, 32, 35, 21, 29, 37, 38, 40$ with the representative 40 as seen on Figure 2.5b.

Recap Now we have shown the main concepts of car-pooling. Depending on the data structure the actual implementation of how it works will be a bit different. But this section should serve as a good foundation and help to make understanding car-pooling in the later implementation easier. Let us recap the main parts of car-pooling: (*i*) items will be grouped into nodes, (*ii*) each node will have a representative, and (*iii*) the data structure will then mostly use nodes as if they were items.

It should also be clear that depending on the amount of car-pooling you do, the less guaranties you have on the order. We saw that before you did any car-pooling, the list had to be fully ordered. If you combined every other element, up to half the elements can be out of order since we only have that they are sorted by the representative, which is the greater of the two. When we get to the soft heap implementations, we will see the different implementations do car-pooling at specific times. When to do car-pooling is important when it comes to the analysis, as you do not want too many nodes.

2.6 Soft heaps

We will now explain what a soft heap is and how it works without going into a specific implementation. Specific implementations will be explained in the sections 2.7-2.9 which will also contain the corresponding theory. Table 2.2 shows the time complexities that we are going to prove in the upcoming analyses except ternary trees which we do not cover.

Soft heaps in general Soft heaps were first introduced by Chazelle [6]. A soft heap has the same behavior as a regular priority queue in the sense that it stores a set of pairs (key, value) called *items* and the keys dictates an ordering. The main idea in soft heaps is to move items across the data structure in groups instead of moving items one at a time [6, p. 1012]. Chazelle calls this *car-pooling* which we have already covered in Section 2.5.

	INSERT	EXTRACT-MIN	
Chazelle 2000 [6]	$\mathcal{O}(\lg \frac{1}{\varepsilon})$	$\mathcal{O}(1)$	Binomial trees
Kaplan and Zwick 2009 [12]	$\mathcal{O}(\lg \frac{1}{\varepsilon})$	$\mathcal{O}(1)$	Binary trees
Brodal 2020 [3]	$\mathcal{O}(\lg \frac{1}{\varepsilon})$	$\mathcal{O}(1)$	Sorted sequences
Kaplan et al. 2013 [14]	$\mathcal{O}(1)$	$\mathcal{O}(\lg \frac{1}{\varepsilon})$	Binary trees
Brodal 2020 [3]	$\mathcal{O}(1)$	$\mathcal{O}(\lg \frac{1}{\varepsilon})$	Ternary trees

Table 2.2: Amortized time complexities for the different soft heap implementations.

To refresh what car-pooling means we need to first remember what a corrupted item is. A soft heap can *artificially* increase the key of an item. We highlight the word artificially because increasing the key is not achieved by modifying actual value of the key in the item. We denote the original key as the *real* key and the possibly increased key as the *current* key. A corrupted item is then defined as the following.

Definition 2.6.1 (General definition of a corrupted item)**.** *An item e is corrupted if e.key* $<$ *e.key*$'$*, where e.key is the real key and e.key*$'$ *is the current key.*

In Subsection 2.8.3 we will see later see that the strengthened interface introduced by Kaplan et al. [14, p. 5] changes the definition of when an item is called corrupted. This enables structure to first report items as corrupt when they can affect the output.

Car-pooling can now be defined as treating a group of items as a single item and the *current* key of all the items in the group is the maximum real key in the group. Observe that assigning the current keys of the items in a group to be the maximum real key in the group, is what artificially increases an item's key. The reason why an item with an increased key is called corrupt is that the real key will never be used again in any soft heap operation. Only the *common* key which is the maximum real key in the group will be compared against other common keys. This means that you cannot know if any item you insert into a soft heap will be corrupted. This is the reason it is called a soft heap.

The difference between a regular priority queue and a soft heap is car-pooling, which will cause corrupted items. This means that given a mixed sequence of INSERT and EXTRACT-MIN operations, a soft heap will return items according to the current key and not the real key. That we return items according to the current key is caused by corruptions and the effects of corruptions on a extracted sequence of items. When an item gets corrupted in the soft heap, it should be seen as an item that that may occur later in the sequence of extractions then it should have, because of its artificial high key.

Error parameter How often this happens is determined by an error parameter ε which becomes fixed when a soft heap is instantiated. In this paper we define ε as $0 < \varepsilon \leq \frac{1}{2}$ unless we specify otherwise.

Definition 2.6.2 (Corruption limit)**.** *The corruption limit is how many items that are allowed to be corrupt in the soft heap and is defined as* εN *after a total of N insertions.*

Note that the corruption limit is independent of EXTRACT-MIN or DELETE calls so it can be larger than the current number of items in the soft heap. Even though soft heaps can make these so called errors in FIND-MIN and EXTRACT-MIN, then it is done in a controlled way in the sense that soft heaps respect the corruption limit.

As seen in [14, p. 4] ε can be bounded depending on the application you want to solve with soft heaps. The most general bound is $0 < \varepsilon < 1$ since it does not make sense for ε to be equal to or larger than 1. If $\varepsilon \geq 1$ then you will have no guarantees from the corruption limit since the number of elements will always be below the corruption limit. With the strengthened interface you might be able to use it, but you would have no guarantees. Most of the time you would not want to fix ε to something larger than $\frac{1}{2}$ because then it will not take long before all items in the soft heap are corrupted. At that point you might as well stop extracting items from the soft heap.

Corruption motivation The brief explanation of why we want corruption is that it introduces common keys of which there are fewer than real keys. Recall, that common keys are used for comparison and they are only compared against each other. So common keys results in fewer comparisons. The precise reason why corruption works can be explained using *entropy* which is an idea from information-theory. Entropy is a measure of information or uncertainty. Entropy is calculated as a function of a probability distribution. For example, a coin toss contains the event head and the event tail each occurring with probability $\frac{1}{2}$. The amount of bits we need to convey the outcome of the toss is 1 bit. So entropy can be thought of as wanting to convey some information using as few bits as possible.

To get the intuition behind the definition of entropy lets look at an example from [20, p. 55]. Let X be a random variable that can assume the values x_1, x_2, x_3 with probabilities $\frac{1}{2}, \frac{1}{4}$, and $\frac{1}{4}$. Encode the events x_1, x_2, x_3 as $0, 10, 11$. The average number of bits in Xs encoding is

$$\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{4} \cdot 2 = \frac{3}{2}$$

$\frac{3}{2}$ is less than our number of events so this suggests that an event that occurs with probability 2^{-n} can be encoded using n bits. We replace n with p in the aforementioned probability and isolate p by taking the logarithm. This suggests the more general statement that an event that occurs with probability p can be encoded using approximately $-\lg p$ bits. For multiple events this means that given a random variable X with probabilities $p_1, \ldots, p_n$ defined by a given probability distribution, we define our measure of information to be the weighted average of the quantities $-\lg p_i$ where $i = 1 \ldots, n$. This is the motivation behind the formal definition: Given a discrete random variable X which can assume values from a finite set X, then the entropy of X is $H(X) = -\sum \Pr(x) \lg \Pr(x)$ [20, p. 55].

Let us now return back to talking about why corruption makes sense. In soft heaps we do not have events, we have data in the form of items that each have a key. So our items will act like the events we just saw in the example. We have N distinct key assignments so using the notation from before we can write $|X| = N$. Each of our inserted keys occurs with the same probability, $\Pr x = \frac{1}{N}$ for all $x \in X$. This implies that $H(X) = \lg N$ as also stated in [6, p. 1013].

To convince one self that corruption reduces the entropy of the data in the heap then one can take it to the extreme by increasing to ∞ when corrupting a key and considering the case where every key is corrupted. The set of keys would have zero entropy meaning that we could perform all operations in constant time [6, p. 2]. We will see later that soft heaps do not need to increase keys this aggressively to attain zero entropy and thereby achieve amortized constant time operations.

The operations that a soft heap needs to support are make-heap, insert, delete, meld and find-min. There can also be other operations like e.g., extract-min. As previously mentioned, soft heaps can achieve amortized constant time or $\mathcal{O}(1)$ time operations by using corruptions which can be seen as allowing errors in the minimum related operations. More precisely either insert or extract-min will take $\mathcal{O}(\lg \frac{1}{\varepsilon})$ amortized time depending on the given soft heap implementation. That time is more desirable than $\mathcal{O}(1)$. Some refer to this as amortized constant time since ε becomes fixed at soft heap instantiation, but we just write the dependency on ε explicitly to be precise.

2.7 Original soft heaps

Soft heaps was introduced by Chazelle in [6]. Since then other implementations have been introduced so we refer to Chazelle's implementation as *original soft heaps*. We decided not to implement original soft heaps in order to focus more on simplified soft heaps and soft sequence heaps. Therefore we will give a very brief overview where we do not cover the operations [6, Section 3].

Original soft heaps are achieved by applying car-pooling on binomial trees. A *binomial tree* of rank h is a tree made up of 2^h nodes. A rank h binomial tree is created by making the root of one rank $h-1$ binomial tree to the child of another binomial tree that also has rank $h-1$. This can be seen on Figure 2.1. Chazelle makes two modifications on a binomial tree and calls the result a *soft queue* q. (i) q is derived from a binomial tree which is called its *master tree*. The difference between q and its master tree is that q is allowed to be missing some subtrees. A node of q has a *rank* which is defined to be the number of children of the corresponding node in the master tree. The rank of a root has to satisfy the invariant that its number of children is not smaller than $\lceil \text{rank}(\text{root})/2 \rceil$. (ii) A node g of q can hold a list of items denoted *item-list*. g.cKey is the shared value of all current keys in item-list(g) and it at most the maximum current key of the items in list-item(g). The cKey of a node is not larger than any of its children.

A original soft heap is a sequence of soft queues. The rest of the explanation which is how this soft queue abstraction can be implemented is found in [6, sections 3-4]. For the sake of briefness we will only mention the analytical result which is that original soft heaps achieves amortized $\mathcal{O}(\lg \frac{1}{\varepsilon})$ time per insertion and $\mathcal{O}(1)$ for all other operations.

2.8 Soft Heaps Simplified

Kaplan et al. took the original soft heap and made a simplified version [13]. This simplified version uses less space and avoids some operations found in the original by Chazelle. This simplified version also tightens and simplifies the analysis and can perform all the operations we would expect. The simplified version of soft heap combines 3 things. (*i*) It uses heap-ordered binary trees as its building blocks where the smallest elements are in the nodes at the top. (*ii*) It uses car-pooling to bring down the number of nodes to increase the performance, just like all the other soft heap implementations. (*iii*) It uses multiple binary trees to ensure that the structures becomes relatively balanced.

In this section we will give an overview of how the algorithm works. This is followed by an analysis. The analysis will show that we are below the corruption limit and analyze the running time. The last thing we will do is to describe how you would make a small change to the implementation to fit the strengthened interface that some

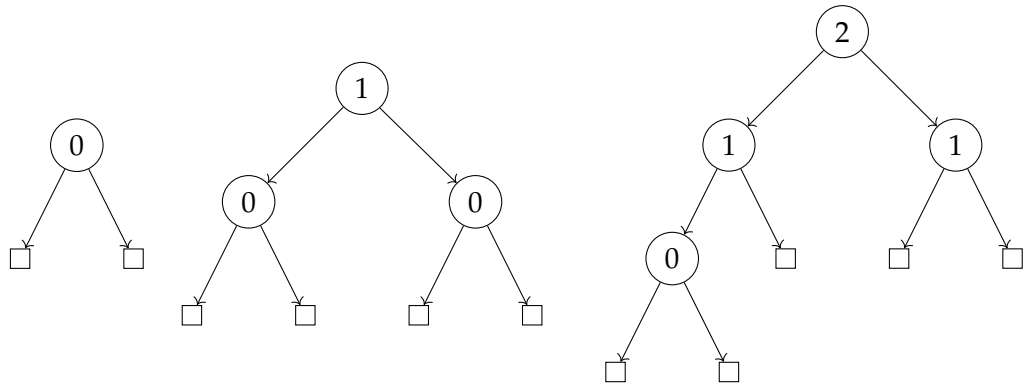


Figure 2.6: Showing the rank of different nodes. The first structure is a single binary node, it is what we have after creating a new root node. The next structure show how two nodes of rank 0 can be used to create one of rank 1. Please note that this can only be an intermediate structure since the node with rank 1 cannot contain zero elements and would fill itself with items from one of the children, deleting the child in the process. The last one shows how two nodes of rank 1 creates one of rank 2. Here we also see how some nodes are missing since they have been used to fill the structure after linking.

applications will need. This is one of the implementations that we are going to test. More specifics about the implementation can be found in Section 5.1.

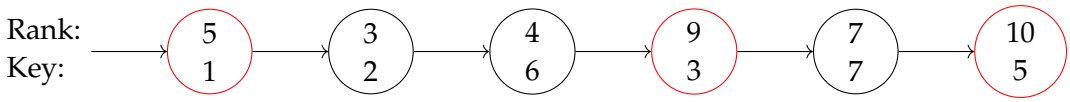
2.8.1 Algorithm

The building blocks of the simplified soft heap are binary trees. In the simplified soft heap we can create binary trees in two ways. The first is when we insert a new item into the soft heap. At that point we create a new binary tree consisting of only a root node that hold the item we just inserted. Each node will have a rank, and the root node we just created will have a rank of 0. Currently, the nodes will hold one item each, once we get to the car-pooling part a node can hold multiple items. In addition to each node having a rank and items, it also has a key. The key is going to be equal to the item inside with the biggest key. Here we see how items can become corrupt in the data structure. If there are multiple items inside a node with different keys then they are treated as if they have the key the node have.

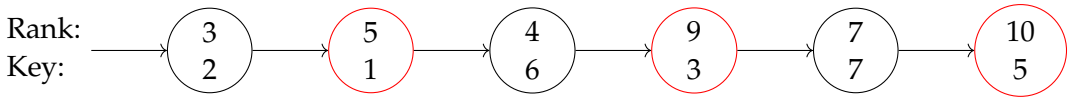
The other way to create binary trees is to combine two roots into a new tree in what we call the linking phase. The LINK operation takes two tree roots of rank k and then creates a new node that have rank $k + 1$ with each of the nodes with rank k as children. After the nodes have been linked, additional code will fill the new node, we will see this later. An example of how LINK works is illustrated on Figure 2.6. We can see the larger the tree is and the closer to the root you are, the higher rank you have.

The order of the root list As noted, in this implementation we have multiple trees. The way we keep track of the trees is to have all the roots form a singly-linked list where each root points to another root. The order of roots is very important as it will affect the performance. We define two orders that the list can be in: *findable order* or *meldable order*.

Findable order means that the root with the lowest *key* is first. This node is then followed by the roots that have a lower rank than the first in increasing order by rank.



(a) This in an example of root nodes findable order



(b) This in an example of root nodes meldable order

Figure 2.7: This is an example of how the same root nodes differs in order under the two orderings we have defined. Imagine each node have subtree below that we have left out. Here it is also easier to see why the changing between the two orderings works. Red nodes indicates where a findable order starts

After that the roots with greater rank comes in findable order. This order is useful when you need to access the node with the lowest key, e.g., in the operations FIND-MIN and DELETE-MIN.

Meldable order means that the root with lowest *rank* is first followed by everything in findable order. This is better for MELD and INSERT since MELD needs to go through elements starting from minimum rank.

Both the orders are illustrated on Figure 2.7 where we can see how the same nodes are arranged differently under different ordering. We can convert between these two orders by swapping the two first elements in the list, if the first node does not already have both the lowest rank and the lowest key. If that is the case then the list is in both orders at the same time.

Findable order is the default order in the sense that it is the order we want we heap to be in once a given operation we have called is complete. While performing subroutines on the data structure you might find it in no order or meldable order.

In addition to how we order the elements there is another important thing to note. By construction the structure will never have two roots with the same rank. This means that there is never doubt about what node has the lowest rank and also makes sure that we do not get too many trees.

Operation definitions We now know we can create trees in two ways and that these trees are in a root list that can be in two orders. The next thing is to be a bit more precise about how each of the operations modify the data structure. Before we talk about the functions you call from the outside let us talk about internal functions.

FILL will fill a given node that it is called on. First it will figure out which of the children have the smallest key. It will then take all the items from that child node and update its own key. After that it will recursively fill the just emptied child. The filling will stop when there are no children in a node, and will delete the node if you took the items from a node that have no children. We do not want nodes without items in them.

LINK will take two roots as arguments. It will then make a new root note with the two arguments as children. Next it is going to fill the new root using FILL. These two functions will be used by the exposed functions, which we will look at now.

INSERT is done by creating a new node x consisting only of a root containing the single item we have inserted. You then change the heap you want to insert into to

meldable order. If x have smaller rank than the first element in the heap, then you change the heap back into findable order and insert x in the front, this makes it into meldable order, and we therefore again convert it into findable order. At this point you would be done.

If x does not have smaller rank, then it must mean that x and the heap have equal rank since we only increase x by one for each iteration. If this is the case then we take the first node in the root list and LINK it x. We now convert the root list, where we just removed the first node from, into meldable order and again check if the rank of the newly linked root is smaller and recurse from there.

MELD takes two root lists and link trees until all roots have a distinct rank. This is done by first converting both structures into meldable order. We look at the first node of each of the lists and find the one with the biggest rank. We then take the node with the biggest rank x_b and put it aside for now. We turn the list we just took a node from into meldable order. We then recurse. On the result of the recursion we add x_b using the insert technique described in INSERT. The effect of this is that we find the tree with the smallest rank and then insert all other trees from smallest rank to biggest rank. Once all the recursion and inserting is done, then we convert the list back to findable order.

DELETE-MIN will look at the first node x. Since the root list is in findable order, x will have the smallest key of the nodes. DELETE-MIN then delete a single item from x. We are done if this was not the last item in the x. If this was the last item in x then that triggers fill of x. After the fill is complete we update the key of x. At this point we need to make sure we leave the list in findable order. To do this we go through the list until we get to the first node with greater rank than x. This would be the second red node on Figure 2.7a. While doing this we are also moving x along list.

At this point we are sure the node with the minimum key is either the node we have reached, or it is to the left of us. This is because to the left is all nodes that have less rank than x, stored in order of increasing rank. The first node we meet with have higher rank than x is the node with the lowest key of the nodes that have higher rank than x. At this point we traverse back thought the list. While traversing, we swap nodes if the key is smaller, such that we are pulling the one with the minimum key along. This will result in a list in findable order.

Let us see an example of what would happen to the heap on Figure 2.7a if we called DELETE-MIN. We could end up with a root list that looks like Figure 2.8 after the operation. Let us explain. If we deleted the last item in the node with rank 5 then that could change its key from 1 to 8 because of filling. To restore findable order we would first go through the list, swapping the node with rank 5 along until we reached the node with rank 8, the first node with rank greater than 5. Now we need to go back. Since the node with rank 8 have the smallest key it will be pulled along until it reaches the node with rank 3 since its node has smaller key. Since the rank 3 node is already at the beginning then we stop. We will also now notice that the list is in findable order.

FIND-MIN can just read an item from the first node. Since the list is in findable order the first node will have the smallest key and therefore contain the item that should be returned. You probably want to make sure if a node have multiple items, that it is the same item that FIND-MIN will return that DELETE-MIN will remove.

Car-pooling So far we have only seen a single item in every node. For that reason it might seem weird why we have been careful to distinguish these two. The reason for

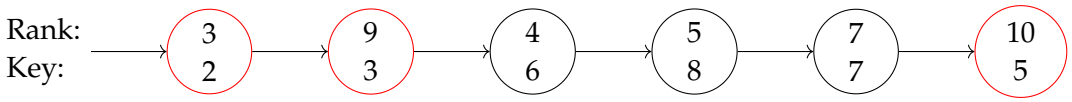


Figure 2.8: Shows the new findable order of Figure 2.7a after a delete opertaion has been performed and changed the key of the node with rank 5 from 1 to 8. Also note that the list is currently both in findable order and meldable order

is that when we add car-pooling to the mix, then we might have nodes that contain multiple items. To do car-pooling we define a new function that is called DEFILL, which stands for double even fill. DEFILL will call the FILL function we have already defined, but it will call it twice if a node's rank is above a certain threshold, the rank of the current node is even, and the node has children. We can get multiple item per node since we are filling nodes that are not empty. Now that we have defined the new function we also want to use it. This is done by simply replacing all calls to FILL in the previous operations with calls to DEFILL. Once this is done you should be good to go. An item will now be corrupt if the item is in a node that does not have the key value it have.

Arbitrary delete So far we have only covered deleting the minimum item. This is the most important part since it is a priority queue and all we need for the applications we will see later. This is also what we have implemented. You can however implement arbitrary delete, which lets you delete any item inside the soft heap assuming you have a pointer to the location. How to do it is described in [13, Section 6]. You can do it lazily, meaning we mark an item as deleted instead of removing it right away. Once a deleted item becomes we minimum item, then we just remove it using the DELETE-MIN. This means that every time we need get a new minimum we need to check that it is not marked. If you want to do it eagerly then additional pointers are needed.

2.8.2 Analysis

There are a number of things we want to prove about this new data structure. The first is that we adhere to the corruption limit that is needed to be a soft heap. The next thing we will focus on is how many fillings happens. This filling result is used when we want to analyze the running times for the different operations. We will mostly follow the order of lemmas in [13], but we will explain the proofs in more detail and in our own words.

Before we start the analysis we want to define some things that is going to be used in the analysis. After you have decided on the variable for ε then we can calculate the threshold $t = \lceil \lg \frac{3}{\varepsilon} \rceil$. In the analysis we will consistently use the variable N to mean the total number inserts and D to mean the total number of deletions. In addition to the nodes having a rank we also define the item to have the rank of the nodes they are in. This is only used as part of the analysis and should therefore not be implemented in code.

Corruption limit

The first thing we want to prove is that we do not go over the corruption limit specified by soft heap. This means that we want to show that the corrupted number of items

never exceeds εN. To get there we first prove a number of smaller lemmas.

Lemma 2.8.1 (Kaplan et al. [13, Lemma 4.1])**.** *The number of nodes of rank k is at most* $\lfloor \frac{N}{2^k} \rfloor$.

Proof. To prove this we use induction in the rank k. The only way to make nodes of rank zero is when we insert elements. Since there are at most N inserts this means there can at most be N nodes of rank 0. Therefore the expression holds: $N \leq \lfloor \frac{N}{2^0} \rfloor$.

For the induction case we point out that a node of rank $k+1$ gets created by linking two nodes of rank k. These nodes of rank k can then not be linked again. This means that there at most can be half as many nodes of rank $k+1$ as there are nodes of rank k. E.g., we can limit the number of rank 1 nodes to be half of rank 0 nodes, which makes the statement true again: $\lfloor \frac{N}{2} \rfloor \leq \lfloor \frac{N}{2^1} \rfloor$, we floor since we cannot make half a node if we only have one node of rank 0. □

Next, we define a function s that is only used in our analysis. The specific definition of s is that $s(k)$ is the maximum size of a node of rank k, where size means how many items it contains. With s defined the first thing we do is to prove some properties about s.

Lemma 2.8.2 (Kaplan et al. [13, Lemma 4.2])**.** *If* $k \leq t$, $s(k) = 1$. *Suppose* $k > t$ *then* $s(k) \leq s(k-1)$ *if k is odd, and* $s(k) \leq 2s(k-1)$ *if k is even.*

Proof. If a node has rank 0 then it has one item from creation and cannot have been filled since it has no children. This means that $s(0) = 1$. If $k \leq t$ or k is odd then DEFILL will never trigger the double fillings. This means that these nodes will only have been filled while empty. Since it gets filled by a node of rank $k-1$ then $s(k) \leq s(k-1)$. Using induction we can therefore say that if $k \leq t$ then $s(k) = 1$. If $k > t$ and k is even then double fillings can happen from a node of rank $k-1$. This gives us $s(k) \leq 2 \cdot s(k-1)$ since two fillings can happen instead of one. □

Instead of just having a recursive expression we want to limit it to a number which we can use in later lemmas.

Lemma 2.8.3 (Kaplan et al. [13, Corollary 4.3])**.** *If* $k > t$ *then* $s(k) \leq 2^{\lceil \frac{k-t}{2} \rceil}$.

Proof. We use Lemma 2.8.2 which gave us that:

$$s(k) = \begin{cases} 1 & \text{for } k \leq t \\ s(k-1) & \text{for } k > t \text{ and } k \text{ is odd} \\ 2s(k-1) & \text{for } k > t \text{ and } k \text{ is odd} \end{cases}$$

$k - t$ is going to denote the number of ranks above the threshold. Only every other rank above the threshold can be double filled, hence we divide by two. We round this result up because we do not know if t is odd or even, and we therefore do not know if the first node after t got double filled or not. Since the double filling doubles the items and we rounded up to be conservative, then the expression have to be true. □

In the next lemma we are going to use that we defined a rank for items. Here we want to show that only a limited number of items will have a high rank.

Lemma 2.8.4 (Kaplan et al. [13, Lemma 4.4])**.** *The number of items of rank greater than t is at most* εN.

Proof. From Lemma 2.8.1 we know that the number of nodes of a given rank k is at most $\lfloor \frac{N}{2^k} \rfloor$. This means we can get a bound on the number of items grater than rank t by taking the number of nodes greater than rank t and multiple it by the maximum size they can have. We will leave out the flooring in the following.

$$\sum_{k>t} \frac{N}{2^k} \cdot s(k)$$

The next step is to notice that for a given value of k we can define a new variable i that is equal to $k - t$. This means that we can replace k with $t + i$ since $t + i = t + k - t = k$. Since the sum works on $k > t$ we can use $i \geq 1$ in the sum to get the same result as using k:

$$\sum_{k>t} \frac{N}{2^k} \cdot s(k) = \sum_{i \geq 1} \frac{N}{2^{t+i}} \cdot s(t+i)$$

Using this new expression we can expand the sum such that we get two iterations for each summing:

$$\sum_{i \geq 1} \frac{N}{2^{t+i}} \cdot s(t+i) = \sum_{i \geq 1} \left(\frac{N}{2^{t+2i}} \cdot s(t+2i) + \frac{N}{2^{t+2i-1}} \cdot s(t+2i-1) \right) \tag{2.1}$$

Lemma 2.8.3 tells us that $s(t+2i) \leq 2^i$ and $s(t+2i-1) \leq 2^i$ if $i \geq 1$. This means we can rewrite the sizes s to 2^i since the sum have that $i \geq 1$.

$$\sum_{i \geq 1} \left(\frac{N}{2^{t+2i}} \cdot s(t+2i) + \frac{N}{2^{t+2i-1}} \cdot s(t+2i-1) \right) \leq \sum_{i \geq 1} \left(\frac{N}{2^{t+2i}} + \frac{N}{2^{t+2i-1}} \right) \cdot 2^i$$

On this expression we can move some of the variables out of the sum and then simplify the expression like so:

$$\begin{aligned}
\sum_{i \geq 1} \left(\frac{N}{2^{t+2i}} + \frac{N}{2^{t+2i-1}} \right) \cdot 2^i &= \frac{N}{2^t} \sum_{i \geq 1} \left(\frac{1}{2^{2i}} + \frac{1}{2^{2i-1}} \right) \cdot 2^i \\
&= \frac{N}{2^t} \sum_{i \geq 1} \left(\frac{2^i}{2^{2i}} + \frac{2^i}{2^{2i-1}} \right) \\
&= \frac{N}{2^t} \sum_{i \geq 1} \left(\frac{1}{2^i} + \frac{2}{2^i} \right) \\
&= \frac{N}{2^t} \sum_{i \geq 1} \frac{3}{2^i} \\
&= \frac{3N}{2^t}
\end{aligned}$$

Now, using that $t = \lceil \lg \frac{3}{\varepsilon} \rceil$ we can conclude that the number of items with rank higher than t can at most be εN since

$$\frac{3N}{2^t} = \frac{3N}{2^{\lceil \lg \frac{3}{\varepsilon} \rceil}} \leq \varepsilon N$$

□

With this we have all we need to conclude that we do not break the corruption limit and fulfill that requirement.

Theorem 2.8.5 (Kaplan et al. [13, Theorem 4.5])**.** *The number of corrupted items is at most εN.*

Proof. From Lemma 2.8.2 we know items that have a lower rank or equal rank to t is not corrupted since the size is always 1. Furthermore, we just learned in Lemma 2.8.4 that there can at most be εN items above the rank t. This means that even if all those items were corrupt then we can still conclude that there cannot be more than εN items corrupt. □

Fillings

Next we will study the fillings. Fillings are essential because they are what we are using to do car-pooling in addition to filling empty nodes. Compared to Chazelle's bound Kaplan improve it by splitting the analysis into to two groups. We define a low fillings to be if you fill a node of rank k where $k \leq t$. If $k > t$ then we call it a high filling.

Lemma 2.8.6 (Kaplan et al. [13, Lemma 4.6])**.** *The number of low fillings is at most $tD + \mathcal{O}(N)$.*

Proof. We analyze two groups, the fillings from items that gets deleted, and the other fillings. We know that we delete a total of D items. Before an item gets deleted it can atmost have been part of t low fillings. This means that we have at most tD low fillings on the items that ends up being deleted.

To calculate how many fillings the rest of the items can have been in, we start by looking at Lemma 2.8.1. We again split into cases. We case for items that are above or below the threshold.

First we want to get the number of low fillings for items of rank $k \leq t$. We know from Lemma 2.8.2 that since we are below rank t, the number of items per node is at most one. This means that each filling only moves one item. Therefore the the number of fillings is bounded by the number of times an item can move while having low rank. To calculate this we know that an item of rank k can at most have been in k fillings. This means we just need to sum over the amount of nodes for each rank up to t. We know the number of nodes of a given rank from Lemma 2.8.1. This value is then multiply it by the rank k. This will tell us that we do $\mathcal{O}(N)$ low fillings in this case:

$$\sum_{k=1}^{t} k \cdot \frac{N}{2^k} = \mathcal{O}(N)$$

For items with rank $k > t$ we use Lemma 2.8.4 to know the number of items that can be above t. Each of these items can at most have been in t low fillings. Using these facts and that $\varepsilon \cdot \lceil \lg \frac{3}{\varepsilon} \rceil$ is less than 3 for any $0 < \varepsilon < 1$ we can calculate how many low fillings the items with rank greater than t have been in.

$$t \cdot \varepsilon N = \lceil \lg \frac{3}{\varepsilon} \rceil \cdot \varepsilon N = \mathcal{O}(N)$$

Now we just combine the tree calculations and get that we can at most have done $tD + \mathcal{O}(N)$ low fillings. □

We defined a new function like we did with size s. We call the new function $f(k)$ and it measures the maximum number of times a node of rank $k \geq 1$ have been filled. If $k = 0$ we define $f(0) = 1$, since we count making the node as a filling.

Lemma 2.8.7 (Kaplan et al. [13, Lemma 4.7])**.** *For every $k \geq 1$ then $f(k) \leq 2f(k-1)$. If $k > t$ and k is even, then $f(k) \leq f(k-1)$.*

Proof. We define a new function g that works like f with some changes. Like f, g counts the number of fillings, but *only* counts *while not being a root*. In addition to this it also counts the deletion. We can see that $g(k) \leq f(k)$ since f would count the first filling where g would not. g does count the deletion, but since f is already in the lead by one, makes it so that g cannot overtake f. For values larger than zero f will always increase faster or at the same speed.

Given a node of rank $k \geq 1$ we know that it has two children of rank $k-1$ it can get filled from. Every time we fill it will either cause a recursive fill or delete the child. This means that the number of fills is limited to the fills of both children plus deletions, which is what we defined g to be. This means we can say that $f(k) \leq 2g(k-1) \leq 2f(k-1)$. We can add the last part since we had $g \leq f$. This proves the first claim.

For $k > t$ and k is even we want an even tighter bound. If $k > t$ and k is even then a filling will trigger two calls to FILL, except maybe the last one when it only have one child. We can therefore say that $2f(k) - 1 \leq 2g(k-1)$. Again, since $f \geq g$ then $2g(k-1) \leq 2f(k-1)$. Since f gives a whole number we can conclude $f(k) \leq f(k-1)$ because the -1 cannot offset the first f since we multiply it by 2. This proves the second claim. □

Like we did with the size function we also want to get some numeric bound for f instead of just a recursion bound.

Lemma 2.8.8 (Kaplan et al. [13, Corollary 4.8])**.** *If $k \leq t$ then $f(k) \leq 2^k$. If $k > t$ then $f(k) \leq 2^{\lceil \frac{k+t}{2} \rceil}$.*

Proof. The first part is easy to prove by induction. If $k = 0$ then $f(0) = 1$. If we put this into the expression we see that it holds $1 \leq 2^0$. Given $k < t$ we know from Lemma 2.8.7 that $f(k) \leq 2f(k-1)$. Assuming by induction that the expression was true for $k-1$ then it also holds true for k.

The second part is proved the same way we did Lemma 2.8.3 with minor differences. One thing you need to notice is that $\frac{k+t}{2}$ is the same as $t + \frac{i}{2}$ where $i = k - t$ since $k > t$. The expression is therefore true since the fillings will increase in size due to DEFILL, lowering the number of fillings you need to make. □

Now we just need to know the number of high fillings to be done with the filling part.

Lemma 2.8.9 (Kaplan et al. [13, Lemma 4.9])**.** *The number of high fillings is at most $3N$*

Proof. This almost exactly follows the proof we had for size Lemma 2.8.4 but this time we have $2^{\lceil \frac{k+t}{2} \rceil}$ from Lemma 2.8.8 instead of $2^{\lceil \frac{k-t}{2} \rceil}$. This makes the conclusion $3N$ instead of εN.

$$\begin{aligned}\sum_{k>t} \frac{N}{2^k} \cdot f(k) &= \sum_{i\geq 1} \frac{N}{2^{t+i}} \cdot f(t+i) \\ &= \frac{N}{2^t} \sum_{i\geq 1} \left(\frac{1}{2^{2i}} + \frac{1}{2^{2i-1}} \right) \cdot 2^{t+i} \\ &= N \sum_{i\geq 1} \frac{3}{2^i} \\ &= 3N\end{aligned}$$

□

Running time

The last thing we want to analyze is the running time. Let us first note the less interesting operations MAKE-HEAP and FIND-MIN. Both of these are $\mathcal{O}(1)$ time worst-case, also amortized. The MAKE-HEAP operation takes an element as input and creates an empty heap and adds the element to it. FIND-MIN returns a value that we can get to in constant time because of the order we store the root elements in findable order.

For the rest of the operations we first want to figure out how much time have been spend on fillings. We can look at the total filling time caused by multiple operations because we are going to do an amortized analysis. After we have calculated the total time used on fillings we can ignore the filling calls inside the other operations.

Lemma 2.8.10 (Kaplan et al. [13, Lemma 5.1])**.** *The time spend doing fillings is $\mathcal{O}(tD + N)$.*

Proof. If we analyze the time it takes to complete DEFILL without the mutually recursive calls to DEFILL from inside FILL then it uses $\mathcal{O}(1)$ time. Since we know the maximum number of low fillings is $tD + \mathcal{O}(N)$ from Lemma 2.8.6 and maximum number of high fillings is $3N$ from Lemma 2.8.9, then we know that the maximum number of calls to DEFILL is $\mathcal{O}(tD + N)$. Therefore, when we look at the total time doing fillings is bounded by $\mathcal{O}(tD + N)$. □

From this we see that we are able to pay for the fillings in the amortized analysis by paying $\mathcal{O}(t)$ and $\mathcal{O}(1)$ in DELETE and INSERT respectively. From here on out we just calculate the running time of the other operations without including the time it takes to fill since that is already paid for.

Lemma 2.8.11 (Kaplan et al. [13, Lemma 5.2])**.** *Excluding filling, the time spend extracting minimum elements is $\mathcal{O}(tD + N)$.*

Proof. If a DELETE-MIN does not empty the root then the operation completes in $\mathcal{O}(1)$ time. However, if DELETE-MIN empties the root of rank k then it takes up to $\mathcal{O}(k + 1)$ time excluding fillings. This covers the time it would take to destroy the node if it had no children, but also covers the cost it would take to reorder the list into findable order. We scale all our calculations around $\mathcal{O}(k + 1)$ such that we get $k + 1$ instead, since $\mathcal{O}$-notation have a fixed multiplier. We can do better than just saying that k is bounded by $\lceil \lg N \rceil$ and saying that the total time spend on deletion, not including fillings, are $\mathcal{O}(D \lg N)$. We use the fact that we know there are a limited number of nodes that can

exceed t in rank. If we instead of paying $k+1$ for every deletion pay $\min(t+1, k+1)$ then we will have paid fully for all nodes of rank $k \leq t$. The total time for this is $\mathcal{O}(tD)$, since we make D deletions at the cost of at most t.

Since all nodes are not $k \leq t$ we need to pay the rest for these, which is per node $k-t$. We sum over the number of nodes of rank $k > t$, since we already calculated the full price for those under t. Each of these nodes needs to be filled before we can remove elements from them. We therefore multiply by the number of nodes with $f(k)$. For the last part we multiply by the part we did not already pay for. This gives us the following expression:

$$\begin{aligned}
\sum_{k>t} \frac{N}{2^k} \cdot f(k) \cdot (k-t) &= \sum_{k>t} \frac{N}{2^k} \cdot 2^{\lceil (k+t)/2 \rceil} \cdot (k-t) \\
&= \sum_{i \geq 1} \frac{N}{2^{i+t}} \cdot 2^{\lceil i/2 \rceil} \cdot 2^t \cdot i \\
&= \frac{N}{2^t} \sum_{i \geq 1} \frac{i \cdot 2^{\lceil i/2 \rceil}}{2^i} \cdot 2^t \\
&= \frac{N}{2^t} \sum_{i \geq 1} \left(\frac{2i-1}{2^{2i-1}} + \frac{2i}{2^{2i}} \right) 2^{t+i} \\
&= N \sum_{i \geq 1} \left(\frac{2i-1}{2^{2i-1}} + \frac{2i}{2^{2i}} \right) 2^i \\
&= N \sum_{i \geq 1} \left(\frac{2i-1}{2^{i-1}} + \frac{2i}{2^i} \right) \\
&= \mathcal{O}(N)
\end{aligned}$$

The reduction works by using the $i = k - t$ technique we have seen multiple times now as in Lemma 2.8.4. In the same lemma we also saw how we can expand the sum. To get the value for $f(k)$ we use Lemma 2.8.8.

If we combine this expression with what we learned before from the case where $k \leq t$, then we get the total time spend on deletion, excluding the time spend on fillings, is $\mathcal{O}(tD + N)$. □

We can now use the same argument as we did for fillings.

Lemma 2.8.12 (Kaplan et al. [13, Lemma 5.3])**.** *Excluding filling, the time spent doing insertions and melds is $\mathcal{O}(1)$ per insertion or meld, amortized.*

Proof. As we defined in Subsection 2.2.3 we can define a potential function on the data structure. The goal is to prove the two operations, INSERT and MELD, are $\mathcal{O}(1)$ amortized time using the potential function. We define the following to be our potential function:

$$\Phi = \text{number of trees} + \text{max node rank}$$

Melding We first look at melding of two heaps H_1 and H_2. We define h_i to be the number of trees and r_i to the root with the height rank in H_i. We assume w.l.o.g. that $r_1 \leq r_2$. After the melding of two heaps is complete it will have performed some number of linkings, we define this variable to be k. The real time for each of meld is $\mathcal{O}(r_1 + k + 1)$.

r_1 because we need to at least iterate to the smallest rank of the two heaps. While doing this we will perform at k links. The plus one covers the case where we meld an empty heap.

Let us look at the potential change when melding. The total potential before the meld is the sum of the potential for each of the soft heaps $(h_1 + r_1) + (h_2 + r_2)$. The total potential after the meld is now the sum of the roots minus the number of links we have made. In addition to this, the highest rank can at most have increased by one. This means the potential is at most $(h_1 + h_2 - k) + (r_2 + 1)$. Now we want to write out the total time plus the change in potential to get the amortized time:

$$\mathcal{O}(r_1 + k + 1) + ((h_1 + h_2 - k) + (r_2 + 1)) - (h_1 + r_1) + (h_2 + r_2)$$

We have a bit of a problem with we $\mathcal{O}$ part, but we can fix this. $\mathcal{O}$ means there is some constant scaling c. We can scale the whole potential function with c. The effect of this would be that we can remove the $\mathcal{O}$ part and get that the amortized time is $\mathcal{O}(1)$:

$$(r_1 + k + 1) + ((h_1 + h_2 - k) + (r_2 + 1)) - ((h_1 + r_1) + (h_2 + r_2)) = 2 = \mathcal{O}(1)$$

Insertion Inserting require us to make a new tree of rank zero and then meld it into the other structure. The creation part takes constant time and increases the potential by a constant. Combining this with what we learned about meld, means we can say that the amortized time for inserting is $\mathcal{O}(1)$. □

Theorem 2.8.13 (Kaplan et al. [13, Theorem 5.4])**.** *The amortized time per heap operation is* $\mathcal{O}(\lg \frac{1}{\varepsilon})$ *per delete and* $\mathcal{O}(1)$ *for each other operation.*

Proof. We conclude this directly from the lemmas above. □

2.8.3 Strengthened interface

The interface we have described so far might not be enough for some applications that exists. Some of the authors of simplified soft heap and more introduces a new interface in [14]. Here they also states how this implementation can be changed to fit the new interface. The new interface makes it such that DELETE-MIN returns the elements that can be returned later as corrupted. This means that no other function can make corruptions, or at least the results cannot be affected before an DELETE-MIN operation have been called. This also means that you can have things that are internally corrupt, but cannot have affected anything the output yet, and therefore not needed to be externally corrupt. We will see this in the next soft sequence heap.

In the current implementation the corruptions happens while inserting, but we want to return what can be corrupted on DELETE-MIN. The way we change the current implementation to fit this requirement is to do the insert lazily. This means that we create a list that can hold the items we insert. Once we try to delete the minimum element then we will do all the inserts and note down the corruption that happens. Since the analysis is amortized it should sill work since we only move the work somewhere else. One thing however is that the DELETE-MIN after a lot of inserts might become very expensive. In a real-time application you might have a bigger desire to spread out the work.

2.9 Soft Sequence Heaps

With the concepts sequence heaps and car-pooling covered, we are now ready to explain how a sequence heap is converted into a *soft sequence heap*.

Soft sequence heaps are introduced by Brodal [3] and are a fairly recent addition to the family of soft heap implementations. A soft sequence heap is a sequence heap with car-pooling applied. More precisely, car-pooling is a general idea and it has been adapted by Brodal to work in the context of a sequence heap. The motivation behind soft sequence heaps is to be an alternative to simplified soft heaps that satisfy the strengthened interface, but have an even simpler implementation.

Corruption- and witness-sets We need to apply car-pooling to make sequence heaps attain the same time complexities as the original soft heap by Chazelle. This will be done by using two ideas called corruption- and witness-sets. We previously mentioned that a soft heap is instantiated with an error parameter ε. Soft sequence heaps use the same bound as Chazelle [5, p. 1012] which is $t = \lceil \lg \frac{1}{\varepsilon} \rceil$ and refers to it as the *rank threshold*.

Each item e of each sorted sequence has a *corruption-set* $C(e)$. $C(e)$ stores items whose keys should be increased to $\text{key}(e)$. Items are added to $C(e)$ when $\textsc{reduce}(L)$ is called on a sequence L. This happens when a merge of two rank $r-1$ sequences yield a sequence with rank $r > t$ where $r - t$ is even. $\textsc{reduce}(L)$ then *prunes* every second item from L excluding the first and last item. Here prune means that an item e is removed from L, but before removing it, it and the set $C(e)$ is moved into $C(e')$ where e' is the successor of e. That we need to move items into a successor item is why we do not prune the last item.

Counting corrupted elements as the one in a corruption set will only leave $o(N)$ items unpruned as the number of items increase. This is true because of the following. In Lemma 2.9.6 we will learn that the *total* number of unpruned items in the heap is $\mathcal{O}(\sqrt{N/\varepsilon})$. If we use that expression as the numerator in the fraction from o-notation and use N as the denominator then we get that indeed only $o(N)$ items are left unpruned. This means that when we insert items then the number of items left not corrupted is vanishing small. We therefore break the corruption limit which is εN.

This is solved by redefining when an item is considered corrupted externally. The new definition is that an item is not seen as corrupt until it is returned by EXTRACT-MIN as corrupt. We first return an item as corrupt when it can affect the order of the items returned by EXTRACT-MIN. This new definition make it so that we do not corrupt too many items. We will achieve this new behavior by using *witnesses*. When an item e in a sequence gets pruned i.e., moved to a corruption-set, then the predecessor of e, e'' becomes the witness for e and stores a copy of e in its witness-set $W(e'')$. $W(e'')$ is all the items that e'' witnessed. A witness e'' can be deleted by EXTRACT-MIN which reports all items in $W(e'')$ as corrupted by returning the witness-set. Observe that for e'' to be a witness then it needs to be in a sequence i.e., it has not been corrupted or deleted. The intuition about what witnesses does will be given later using a figure.

Representation details We will now go over some notation and details about how we represent a soft sequence heap. Some notation might seem familiar from when we introduced sequence heaps. We denote an item, its real key, and its value as e, $\text{key}(e)$, and $\text{value}(e)$ respectively. In Subsection 2.4.3 we explained that a sequence heap has

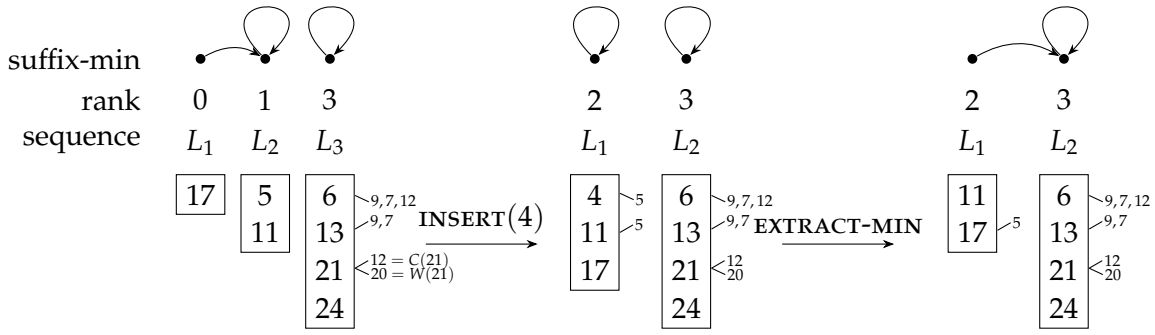


Figure 2.9: A soft sequence heap instantiated with $t = 0$. The lines going up from a number in a sequence denotes the elements in its corruption-set. The downward line denotes the witness-set. To keep the example short we use $t = 0$. This means that we are actually using an ε that is higher than $\frac{1}{2}$ which is what we have upper bounded ε to. INSERT(4) creates sequence (4) which is merged with (17) creating $(4, 17)$. That is merged with $(5, 11)$ creating $(4, 5, 11, 17)$ where 5 is pruned. EXTRACT-MIN removes 4 and reports 5 as corrupted.

sequences and they have ranks. The same is the case for a soft sequence heap S which we represent using a list $\mathcal{L}$ of non-empty sequences $L_1, \ldots, L_\ell$ of items.

As before, sequence rank is denoted rank(L_i) and $\mathcal{L}$ is ordered such that rank$(L_i) <$ rank(L_{i+1}) for $1 \leq i < \ell$. Both $C(e)$ and $W(e)$ are represented by linked-lists. e is either in a single sequence L_i or it is in a single corruption-set $C(e')$ and possibly a single witness-set $C(e'')$. Observe that $e \in W(e'')$ and $e \in C(e')$ implies about e'' and e' that they are both in the same sequence. Also observe that the items of a sequence that are not only *internally* corrupted, but fully corrupted are those that are in a corruption-set, but not a witness-set. This means that the *total number of corrupted items* is calculated by subtracting the total number of witnesses from the total number of internally corrupted items. The car-pooling effect comes from the fact that if $e \in C(e')$ then the current key of e is key(e').

To get the desired running time for EXTRACT-MIN, a reference to the minimum item has to be maintained efficiently. To do this we use a suffix-min idea, also used in [6, p. 1017]. This is done by making each L_i have a *suffix-min* reference. A suffix-min reference is defined as:

$$\text{key}(\text{head}(L_j)) \leq \text{key}(\text{head}(L_{i\ldots\ell})), \text{ where } i \leq j \leq \ell \implies \text{suffix-min}(L_i) = L_j$$

This definition implies that suffix-min$(L_\ell) = L_\ell$. The definition also implies that head(suffix-min(L_1)) is the minimum item so the suffix-min of the first sequence is a reference to the minimum item. suffix-min(L_i) can be updated like so

```
1: function UPDATE-SUFFIX-MIN(L_i)
2:     if i = ℓ ∨ key(head(suffix-min(L_{i+1}))) then
3:         suffix-min(L_i) = L_i
4:     else
5:         suffix-min(L_i) = suffix-min(L_{i+1})
```

Here is a helper function that we are going to use in the next part. It basically

removes every second item in a sequence which is referred to as *pruning* an item. The resulting sequence has length $\lceil \frac{m+1}{2} \rceil$.

```
1: function REDUCE(L = e_1, ..., e_m)
2:     for all 1 ≤ i < m/2 do
3:         Append {e_2i} ∪ C(e_2i) to C(e_2i+1)
4:         Append {e_2i} ∪ W(e_2i) to W(e_2i−1)
5:         prune e_2i from L
```

Operations implementation We now explain how to implement the operations. Each of the primary soft heap operations has been written below in pseudo code. The pseudo code follow the original explanation from [3, p. 6] closely, but one should note two things. First of all, just because the operations are written as pseudo code then that should not be interpreted as this is exactly the way the different operations need to be implemented. The second thing is that some parts of the original explanation are just suggestions. For example, MELD uses multiple while-loops, but it can also be implemented using a single loop. We will explain a bit about the operations in words here, but otherwise the pseudo code should be able to stand on its own. Note that in MELD when we write comma in subscript then the value next to the comma is the rank.

DELETE is implemented lazily which means that when DELETE is called then it only mark the item as deleted. The item will then remain in the heap until it becomes the minimum item and it will then be truly removed when EXTRACT-MIN is called.

Like [6, 14], soft sequence heaps define its own threshold in the following way.

Definition 2.9.1 (Soft sequence heap threshold)**.** *A soft sequence heap is instantiated with* $0 < \varepsilon \leq \frac{1}{2}$ *which is used to define the threshold* $t = \lceil \lg \frac{1}{\varepsilon} \rceil$.

An example of INSERT and EXTRACT-MIN can be seen on Figure 2.9. Now that the explanation and figure have been covered it is easier to give some intuition as to why we wait to report an item e as corrupted until the moment where an item is removed that has e in its witness-set. On Figure 2.9 consider the key 9 which is in the witness-set of 6. This implies that item 9 was pruned at some point and item 6 either has directly witnessed this happen or it has gotten 9 by adopting the witness-set of another item. In either case then 6 will always have come before 9 in the sequence. When 6 is extracted at some point and the items in its witness-set is reported as corrupted then this means that in the future when 9, 7, and 12 are extracted, then we know that these items may have been extracted later then they should have been. The intuition behind this is that a witness-set for a given e will always be higher in a sequence than the corruption-set for that e. This means that when we extract and remove items from the top, then the witness-set has to go first. One can think of it as the item 6 protects 9, 7, and 12.

Remark the following about REDUCE. Because rank is only changed after a merge where it is increased from $r - 1$ to r then if a sequence of e.g., rank 3 gets items extracted multiple times then even though it might now only contain 1 unpruned item then it can still only be merged with another sequence of rank 3.

```
1: function DELETE(S, e)
2:     (e', k') = FIND-MIN(S)
3:     if e ≠ e' then
4:         Mark e as deleted
5:     else
6:         (e, k, C) = EXTRACT-MIN(S)
7:         return C
```

```
1: function INSERT(S, e)
2:     L' = new sequence{e}
3:     ℒ.pushFront(L')
4:     while rank(ℒ[0]) = rank(ℒ[1]) do
5:         r = rank(ℒ[0])
6:         ℒ[0] = merge(ℒ[0], ℒ[1])// Post merge rank is r + 1
7:         if r + 1 > t ∧ isEven(r − t) then
8:             REDUCE(ℒ[0])
9:     UPDATE-SUFFIX-MIN(L_1)
```

```
1: function EXTRACT-MIN(S)
2:     L_i = suffix-min(L_1)
3:     e = head(L_i)
4:     k = key(e)
5:     if C(e) ≠ ∅ then
6:         e' = head(C(e))
7:         C(e).remove(e')
8:         if FIND-MIN(S) = lazyDeletedItem then
9:             EXTRACT-MIN(S)// Return w-set instead of ∅
10:        return (e', k, ∅)
11:    else
12:        C = W(e)
13:        removeLazyDeletedItems(C)
14:        L_i.remove(e)
15:        if L_i = ∅ then
16:            ℒ.remove(L_i)
17:        for all j ∈ i, ..., 0 do
18:            UPDATE-SUFFIX-MIN(L_j)
19:        if FIND-MIN(S) = lazyDeletedItem then
20:            EXTRACT-MIN(S)// Add witnesses to C
21:        return (e, k, C)
```

```
1: function MAKE-HEAP(ε)
2:     return S// ℒ = ∅
```

```
1: function FIND-MIN(S)
2:     L_i = suffix-min(L_1)
3:     e = head(L_i)
4:     k = key(e)
5:     if C(e) = ∅ then
6:         return (e, k)
7:     else
8:         e' = head(C(e))
9:         return (e', k)
```

```
1: function MELD(S_1, S_2)
2:     L_1 = S_1.L
3:     L_2 = S_2.L
4:     r_1 = maxRank(L_1)
5:     r_2 = maxRank(L_2)
6:     L = merge(L_1, L_2) // We merge the list of sequences, not the sequences themselves
7:     while ∃(i, j) : i ≠ j ∧ L_i, L_j ∈ L ∧ rank(L_i) = rank(L_j) do// i, j should both be picked in decreasing order such that sequences are merged right to left
8:         L' = merge(L_i, L_j) // Rank is increased by one, and L' is in L
9:         if rank(L') > t ∧ isEven(rank(L') − t) then
10:            L' = REDUCE(L')
11:    for all i ∈ indexOfFirstMerge(L), ..., 0 do
12:        UPDATE-SUFFIX-MIN(L_i)
13:    return reference(L)
```

2.9.1 Analysis

A soft sequence heap is an approximate priority queue so its purpose is to deliver the inserted items in the specified ordering with some amount of inaccuracy. To satisfy this, the operations needs to be correct, especially FIND-MIN which informs about the ordering and EXTRACT-MIN which deliver the items. That they are correct comes from the fact that the item they return is always the item whose current key is the minimum of all non-corrupted items in the sequences.

For the remaining part of this section we will be moving towards proving Theorem 2.9.2. We do this in a number of smaller lemmas. The main outline of the lemmas mostly follows the same as in the original soft sequence heap paper [3, p. 7-10]. We provide or own explanation and break the proofs into smaller steps to make them more explicit.

Theorem 2.9.2 (Brodal [3, Theorem 1])**.** *A soft sequence heap supports insert in amortized $\mathcal{O}(\lg \frac{1}{\varepsilon})$ time and all other soft heap operations in amortized constant time, for a fixed error parameter $0 < \varepsilon \leq \frac{1}{2}$. After a total of N insertions the soft heap contains at most εN items with corrupted keys.*

Lemma 2.9.3 (Brodal [3, Lemma 2])**.** *A sequence with rank r contains at most 2^r items (corrupted or not), and after N insertions all sequences have rank at most $\lfloor \lg N \rfloor$.*

Proof. We do induction in the rank r. When we create a sequence it will have a rank of 0 and contain one element. Therefore, the base case holds: $1 \leq 2^0$.

Items can be removed from the sequences, but this will only lower the number of items and not affect the rank since the rank only changes after a merge or when the sequence becomes empty and is therefore removed. To increase the rank to r we will take two sequences of $r - 1$. By the induction hypothesis we assume the sequences of rank $r - 1$ had at most 2^{r-1} items. Combining these two lists therefore gives us at most $2^{r-1} \cdot 2$ items which is the same as 2^r. This induction proof proves a sequence of rank r can contain at most 2^r items.

Since it takes 2^r items to make a sequence of rank r then the largest possible rank a sequence can have in a soft sequence heap is $\lfloor \lg N \rfloor$. □

In the following proofs we will use the variable s_r which is defined below. Note that when we write maximum length of a sequence in the following definition then we mean the items that have not been pruned so corrupted items are not counted.

Definition 2.9.4 (s_r)**.** *We define s_r to be the maximum length of a sequence of rank r.*

The next lemma shows the impact that REDUCE has on s_r.

Lemma 2.9.5 (Brodal [3, Lemma 3])**.** *A sequence of rank r contains at most $s_r = 2^r$ items for $r \leq t$, and at most $s_r = (2^t + 1) \cdot 2^{\lceil (r-t)/2 \rceil}$ items for $r > t$.*

Proof. We want to take into consideration that car-pooling is going to affect the maximum length of a sequence. For sequences of rank $r \leq t$ no car-pooling has been applied, and the bound from Lemma 2.9.3 still stands. This implies $s_r = 2^r$.

For $r > t$ car-pooling has been applied. This means we do a merge and then alternate between pruning and not pruning. Merging of two sequences can at most double the length compared to the previous rank's max length i.e., $s_{r+1} \leq 2s_r$. If REDUCE get applied then the length would be halved, but we preserve the first and last item. Given that the length before the REDUCE was bounded by $2s_r$ then the length after reduce is bounded by

$$s_{r+1} \leq \left\lceil \frac{2s_r + 1}{2} \right\rceil = s_r + 1$$

We can write up the cases like this:

$$s_r = \begin{cases} 1 & \text{for } r = 0 \\ s_{r-1} + 1 & \text{for } r > t \text{ and } r - t \text{ is even} \\ 2 \cdot s_{r-1} & \text{otherwise} \end{cases}$$

Since we know that when we merge we alternate between using REDUCE and not using it then we can calculate the length of a sequence given $r = t + 2p$ for a $p \geq 1$ since we had that $r > t$. Now each p represents an application of merge and another merge with a reduce afterwards. The length of s_{t+2} is therefore

$$\overbrace{2^t}^{s_t} \overbrace{\cdot 2 + 1}^{\text{merge, merge, reduce}}$$

If p is larger then we just repeat this pattern of multiplying by two and increasing by one as so:

$$(\cdots((2^t \cdot 2 + 1) \cdot 2 + 1) \cdots)$$

We can simplify this expression by first focusing on what the multiplying by two part does and what the addition part does. If the addition part was not there then it would be $2^t \cdot 2^p$. This can be further simplified

$$\begin{aligned} (\cdots((2^t \cdot 2 + 1) \cdot 2 + 1) \cdots) &= 2^t \cdot 2^p + \sum_{i=0}^{p-1} 2^i \\ &= 2^t \cdot 2^p + 2^p - 1 \\ &= (2^t + 1) \cdot 2^p - 1 \end{aligned}$$

From this we can see that the bound that we wanted to prove holds since when we isolate p in its definition we get $p = \frac{r-t}{2}$.

If we cannot describe r by the expression above then we should be able to describe it by $r = t + 2p - 1$. We know that in this case s_r would only increase by one compared to s_{r-1} since $s_{t+2p-1} = s_{t+2p} - 1$.

We can therefore conclude that we never go over the length specified by s_r. □

From the result of the following lemma then one can derive that REDUCE only leaves $o(N)$ items in the sequences and reduce therefore has a big impact on the data structure. This is mitigated though by the fact that majority of pruned items will have witnesses and are therefore not corrupted.

Lemma 2.9.6 (Brodal [3, Lemma 4])**.** *For a soft sequence heap the total number of items (unpruned items) in $L_1, ..., L_\ell$ is $\mathcal{O}(\sqrt{N/\varepsilon})$.*

Proof. To bound the number of non-corrupted items we combine the bound of each of the sequences. We can split this into the case where $r \leq t$ and the case where $r > t$ which has the two bounds we found in Lemma 2.9.5. This gives the expression with the summations below. The addition operand is the non-REDUCE case and the right is the REDUCE case. From Lemma 2.9.3 we know $\lfloor \lg N \rfloor$ is the largest attainable rank of a sequence. As explained in the end of Lemma 2.4.1 then this means that the largest possible number of sequences in a soft sequence heap is $\lfloor \lg N \rfloor + 1$. So this is the value we sum over in the right addition operand, just zero indexed.

$$\sum_{r=0}^{\lfloor \lg N \rfloor} s_r \leq \sum_{r=0}^{t} 2^r + \sum_{r=t+1}^{\lfloor \lg N \rfloor} (2^t + 1) \cdot 2^{\lceil (r-t)/2 \rceil}$$

This result can be bounded to the expression below. The sum on the left part of the addition gives $2^{t+1} - 1$, which is $\mathcal{O}(2^t)$ or constant for a given t. We like to keep the constants that use t since they show that changing ε does affect this. We now look on the right side and do a case analysis. If $\lg N > t$ then since r is an positive exponent, the last iteration is going to be the dominant term in $\mathcal{O}$-notation. This means we have that $r = \lg N$. If we then simplify the last term we get $\mathcal{O}(2^{t/2} \cdot 2^{(\lg N)/2})$. If $\lg N \leq t$ then the right side becomes zero. This means that

$$\sum_{r=0}^{t} 2^r + \sum_{r=t+1}^{\lfloor \lg N \rfloor} (2^t + 1) \cdot 2^{\lceil (r-t)/2 \rceil} = \mathcal{O}(2^{t/2} \cdot 2^{(\lg N)/2})$$

We then use the fact that $t = \lceil \lg \frac{1}{\varepsilon} \rceil$ and for a variable a we have $2^{(\lg a)/2} = \sqrt{a}$ to get the thing we wanted to proof.

$$\mathcal{O}(2^{t/2} \cdot 2^{(\lg N)/2}) = \mathcal{O}(\sqrt{N/\varepsilon})$$

□

Lemma 2.9.5 bounded the total length of a single sequence, the following lemma takes it a step further by making the same bound, but for all sequences added together.

Lemma 2.9.7 (Brodal [3, Lemma 5])**.** *Over a sequence of heap operations containing N insertions, the total length of all sequences created is bounded by $\mathcal{O}(N \lg \frac{1}{\varepsilon})$.*

Proof. We can at most create $\lfloor N/2^r \rfloor$ distinct sequences of rank r since it takes 2^r items to create one sequence of rank r and an item can never become part of a lower ranked structure. To get the length we sum all possible ranks. So for each rank we take the number of possible sequences for that rank times the maximum possible length of that rank.

$$\sum_{r=0}^{\lfloor \lg N \rfloor} \left\lfloor \frac{N}{2^r} \right\rfloor \cdot s_r$$

We can do a number of things to this equation. The first thing we do is use Lemma 2.9.5 to split it into the case where we are below t and above. Next we write the equation in $\mathcal{O}$-notation-notation and in the last step we substitute t by its definition. For the step with $(t-r)/2$ in the exponent, it is the first term that is going to dominate the expression since r is increasing and it is subtracted.

$$\begin{aligned}
\sum_{r=0}^{\lfloor \lg N \rfloor} \left\lfloor \frac{N}{2^r} \right\rfloor \cdot s_r &= \mathcal{O}\left(\sum_{r=0}^{t} \frac{N}{2^r} 2^r + \sum_{r=t+1}^{\lfloor \lg N \rfloor} \frac{N}{2^r} \cdot 2^{(r+t)/2} \right) \\
&= \mathcal{O}\left(\sum_{r=0}^{t} N + \sum_{r=t+1}^{\lfloor \lg N \rfloor} N \cdot 2^{(t-r)/2} \right) \\
&= \mathcal{O}\left(N \cdot t \right) \\
&= \mathcal{O}\left(N \cdot \lg \frac{1}{\varepsilon} \right)
\end{aligned}$$

From this the lemma follows directly. □

The amortized running times of the operations is proved in the next lemma.

Lemma 2.9.8 (Brodal [3, Lemma 6])**.** *Soft sequence heaps support* INSERT *in amortized* $\mathcal{O}(\lg \frac{1}{\varepsilon})$ *time, and the remaining operations in amortized constant time.*

Proof. We first look at FIND-MIN and MAKE-HEAP. Both of these take $\mathcal{O}(1)$ time since their amount of work does not dependent on the input or current state of the data structure.

We want to look at how many times the different operations can be called based on the number of items that have been inserted. We first note that an item can only be pruned once, since that moves it to a corruption-set and a witness-set. An item can also only be extracted once either from the corruption-set or the sequence directly. An item can also at most be reported as corrupt once. All this takes $\mathcal{O}(N)$ time, not including the time to update suffix-min. Besides these operations we have some that can vary in time it takes. The variation can come from two places. The first is when merging sequences including reducing, and the other is when updating suffix min.

Merging Merge can happen from INSERT and MELD. Lemma 2.9.7 tells us the max number of sequences created over time is bounded by $\mathcal{O}(N \lg \frac{1}{\varepsilon})$. Merging takes linear time in the length of the list, the same with REDUCE, so the total time used on merging and reducing can at most be $\mathcal{O}(N \lg \frac{1}{\varepsilon})$.

Suffix-min updates Suffix-min updates will happen when the minimum item changes. This means that the operations INSERT, MELD, and EXTRACT-MIN can trigger that we need to update the suffix-min pointers for some sequences. We however do not need

to update all the pointers each time. For INSERT only the first suffix-min needs to be updated, since INSERT can only have affected the first sequence in the post-operation data structure.

For EXTRACT-MIN we need to update suffix-min in all the sequences that come before the sequence we have extracted from. If the sequence with the lowest item have rank r then it takes $\mathcal{O}(r)$ time to update. There we can at most do N extractions, since we remove an item each time. We want the total time spent on suffix-min updates, so we split the analysis depending on rank r. Since we can at most have N extractions total, this must imply that we also have at most have N extractions where the rank $r \leq t$. Extracting all these items can at most take $\mathcal{O}(N \cdot t)$ time.

For the sequences with $r > t$, corruption can happen. We remind the reader that the maximum number of possible sequences of a given rank is $\lfloor \frac{N}{2^r} \rfloor$, we learned this inside Lemma 2.9.7. The number of extracts that forces us to update suffix-min is the number of sequences times the length of each sequence times the work at a given sequence's rank. This gives us the following expression, which we can simplify:

$$\begin{aligned}\sum_{r=t+1}^{\lfloor \lg N \rfloor} \left\lfloor \frac{N}{2^r} \right\rfloor \cdot s_r \cdot r &\leq \sum_{r=t+1}^{\lfloor \lg N \rfloor} \frac{N}{2^r} \cdot 2^{(r+t)/2} \cdot r \\ &= \sum_{r=t+1}^{\lfloor \lg N \rfloor} N \cdot 2^{(t-r)/2} \cdot r \\ &= \mathcal{O}(N \cdot t)\end{aligned}$$

From this we see that the total time spent on EXTRACT-MIN and INSERT is $\mathcal{O}(N \cdot t)$.

The last operation we need to look at is melding two soft sequence heaps $\mathcal{L}_1$ and $\mathcal{L}_2$. Here we will use a potential argument to bound the time. We first define a rather simple potential function Φ.

$$\Phi = \text{sequence with max rank}$$

We note that this potential function does not ruin the analysis for the operations we have already looked at. The only other operation that can increase the potential function is INSERT. But, INSERT can only increase the max rank by one, this only adds a constant to INSERT.

For MELD we define r_1 and r_2 to be the maximum rank of $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. The potential after the melding is at most $\max(r_1, r_2) + 1$, since it can only have increased the maximum rank by one. This means that the change in potential is negative $\min(r_1, r_2) - 1$, which is potential that we can use. By adding one to the potential on meld calls we can make this into $\min(r_1, r_2)$. Since we meld $\mathcal{L}_1$ into $\mathcal{L}_2$ then we only need to update at most $\min(r_1, r_2)$ pointers since after rank $\min(r_1, r_2)$ no changes can happen, or if they do, then it comes from a cascading merge. There can only be one cascading merge that extends rank $\min(r_1, r_2)$. A merge will combine sequences into one implies we can still pay for this suffix-min update with $\min(r_1, r_2)$. Assuming you never meld a heap with zero items, then we know that the amount of melds cannot exceed $N - 1$. This means that the time spent on updating suffix-min in meld can at most be $\mathcal{O}(N)$.

From this we have that the total work over a sequence of operations, except the constant operations MAKE-HEAP, FIND-MIN and melding empty heaps, is $\mathcal{O}(N \lg \frac{1}{\varepsilon})$, this means we do $\mathcal{O}(\lg \frac{1}{\varepsilon})$ for each insertion made. The rest was constant as we noted. □

Like we had a length bound s_r we now define a corruption-set length bound c_r and witness-set length bound w_r of an item in a sequence of rank r.

Lemma 2.9.9 (Brodal [3, Lemma 7]). *$c_r = w_r = 0$ is true for $r \leq t$, and $c_r = w_r = 2^{\lceil (r-t)/2 \rceil} - 1$ is true for $r > t$.*

Proof. Just merging never affects the corruption-set or witness-set, only REDUCE affects them. While $r \leq t$ we do no car-pooling and therefore REDUCE is never called. This means that items are never added to c_r and w_r and therefore remains 0 all the time.

For $r > t$, car-pooling only happens then $r - t$ is even. This means we set up *three cases*. (*i*) From before we know that if $r \leq t$ then c_r and w_r are 0. (*ii*) If no car-pooling is done, then the corruption-set and witness-set does not change from the previous rank. (*iii*) If car-pooling happens, then every other sequence item adopts the corruptions of its predecessor, thereby doubling the length and we also adopt the predecessor. We can write these three cases up as follows to give us a length limit on the number of corruptions on an item in a sequence of rank r (witnesses have the same formula):

$$c_r = \begin{cases} 0 & \text{for } r \leq t \\ c_{r-1} & \text{for } r > t \text{ and } r - t \text{ is odd} \\ 2 \cdot c_{r-1} + 1 & \text{for } r > t \text{ and } r - t \text{ is even} \end{cases}$$

From here we do the same as we did in Lemma 2.9.5. We write out $r = t + 2p$ for $p \geq 1$. This time however, the base case is 0, the odd case does nothing, and the even case does the same as the odd and the even did in Lemma 2.9.5. This means we get:

$$\begin{aligned} ((\cdots((0 \cdot 2 + 1) \cdot 2 + 1) \cdots) \cdot 2 + 1) &= \sum_{i=0}^{p-1} 2^i \\ &= 2^p - 1 \end{aligned}$$

Since we have that $2p = r - t$ we get what we need to prove in the lemma: $2^{\lceil (r-t)/2 \rceil} - 1$. We also see that if we need $r = t + 2p - 1$ then the odd case does not change c_r. The same arguments can be applied for w_r to get the same bound as for c_r. □

We define an interval $I(e)$ for each item e in a corruption-set of a given item e' in a sequence. The key of e is always in the interval $I(e)$. If e has no witness then $I(e) =]-\infty, key(e')]$. If e'' holds a witness for e then the interval is $I(e) =]key(e''), key(e')]$. Next we define $D(L_i, x)$ where L_i is a sequence and x is a possible key value. D then denotes the items e_s in L_i that are corrupt and where the $x \in I(e_s)$. We can write this as:

$$D(L_i, x) = \{e_s | \exists e'_s \in L_i : e_s \in C(e'_s) \wedge x \in I(e_s)\}$$

An example of this would be that if we called $D(L_2, x)$ for any $6 < x \leq 13$ in the soft heap we in the sequences after extracting in Figure 2.9 then we would get the items with keys 9, 7 and 12. Going lower will return an empty set since they all have witnesses. Going higher will only return 12 until x passes 21.

$D(L_i, -\infty)$ therefore gives all externally corrupted items in L_i. We define d_r to bound the number of external corruptions for a sequence of rank r.

Lemma 2.9.10 (Brodal [3, Lemma 8]). *$d_r = 0$ for $r \leq t$, and $d_r = 2^{r-t-1}$ for $r > t$.*

Proof. For $r \leq t$ we already know that there are no corruptions. Therefore, the first part of the lemma follows directly.

When two sequences L_i and L_{i+1} gets merged together into $L_{i'}$ then all the corrupted items from each of the sequences come together in the new one. This means that $D(L_{i'}, x) = D(L_i, x) \cup D(L_{i+1}, x)$. However, REDUCE is going to affect this if it gets called. So we look at how REDUCE can change the number of d_r.

If an item e with predecessor e'' and successor e' is pruned from the sequence then its interval becomes $I(e) =]key(e''), key(e')]$ where it had none before. Furthermore, all the items that were in the corruption-set of e have been moved to e'. This extends all their intervals with $]key(e), key(e')]$. The number of items getting moved here are bound by the number of corruptions from the previous rank c_{r-1}. The same happens for the witnesses. Their new bound is extended with $]key(e''), key(e)]$.

So let us see how this changes $D(L_i, x)$. If $x \in I(e)$ then D will increase by one. Furthermore, since $]key(e''), key(e)]$ and $]key(e), key(e')]$ are disjoint implies that for a given x in $D(L_i, x)$ we can only be in the of the ranges. If we are in the first range then D will increase by at most w_{r-1} and for the other at most c_{r-1}. This implies that we can at most increase D by $\max(c_{r-1}, w_{r-1}) + 1$. The bounds for corruption-set and witness-set are the same for a given rank, we can therefore remove the max part. We therefore end up with an upper bound of $2 \cdot d_{r-1} + c_{r-1} + 1$ when REDUCE is applied. Like we have done before we can write up the different cases.

$$d_r = \begin{cases} 0 & \text{for } r \leq t \\ 2 \cdot d_{r-1} & \text{for } r > t \text{ and } r - t \text{ is odd} \\ 2 \cdot d_{r-1} + c_{r-1} + 1 & \text{for } r > t \text{ and } r - t \text{ is even} \end{cases}$$

Using Lemma 2.9.9 we can replace c_r and simplify the expression. We use the same trick where we note that it alternates between two cases and then write out those two alternating cases in the sum. This is the same we have done in the simplified analysis Equation 2.1 on page 28:

$$\begin{aligned} d_r &= \sum_{i=1}^{\lfloor \frac{r-t}{2} \rfloor} (c_{(t+2i)-1} + 1) \cdot 2^{r-(t+2i)} \\ &= \sum_{i=1}^{\lfloor \frac{r-t}{2} \rfloor} (2^{\lfloor (t+2i-1-t)/2 \rfloor} - 1 + 1) \cdot 2^{r-t-2i} \\ &= \sum_{i=1}^{\lfloor \frac{r-t}{2} \rfloor} 2^{i-1} \cdot 2^{r-t-2i} \\ &= 2^{r-t-1} \sum_{i=1}^{\lfloor \frac{r-t}{2} \rfloor} 2^{-i} \\ &< 2^{r-t-1} \end{aligned}$$

This gives us the bound we are looking for. We however need to look at EXTRACT-MIN as this can also affect the value of D. When we extract the first element e in a sequence with empty corruption-set then elements can lose their witness and thereby get their interval extended by $]-\infty, key(e)]$. Since the corruption-set is empty, it is only the items in the witness that gets affected. Furthermore, since it is the first item in the sequence, then no

intervals starts lower than $key(e)$. This means that $D(L_i, key(e) - a) = D(L_i, key(e) + a)$, for an arbitrary small a. Since we have not changed $D(L_i, key(e) + a)$ it must therefore still be below or equal to d_r.

□

Lemma 2.9.11 (Brodal [3, Lemma 9])**.** *The total number of corruptions in a soft sequence heap after N insertions is bounded by εN.*

Proof. Now that we have a bound for a sequence from Lemma 2.9.10 we can simply sum over all possible ranks. We bounded the max rank all the way back in Lemma 2.9.3. We write up the expression and simplify the expression:

$$\begin{aligned}
\sum_{r=0}^{\lfloor \lg N \rfloor} d_r &= \sum_{r=t+1}^{\lfloor \lg N \rfloor} 2^{r-t-1} \\
&= \sum_{i=0}^{\lfloor \lg N \rfloor - t - 1} 2^i \\
&= 2^{\lfloor \lg N \rfloor - t} - 1 \\
&< \frac{N}{2^t} \\
&\leq \varepsilon N
\end{aligned}$$

From this we can see that the maximum number of corruptions can at most be εN. □

We can now prove Theorem 2.9.2.

Proof. The running time follows from the analysis we made in Lemma 2.9.8 and the corruption limit follows directly from Lemma 2.9.11. □

2.10 Applications

In our testing we are going to cover two practical applications of soft heap. The first one is finding the k-th smallest element in an unsorted list. We will also call this finding the element with rank k. This problem can also easily be modified to return the k smallest elements instead of just the k-th smallest. The other problem we will cover is heap selection. This problem covers selecting the k-th smallest element when the input is a heap-ordered binary tree. Interesting for this problem is that it needs the strengthened interface that was introduced in [14]. Soft sequence heap supports this out of the box, but for the simplified soft heap we need to use the lazy insert version.

Approxiate the media Before we go further we want to show how you can use the corruption bounds of soft heap to give some guaranties. This will show how soft heaps can be used to get an approximate result. In the later applications we can see how we can get a deterministic result even when using a soft heap. The first problem we want to talk about is approximating the median. Chazelle already in his first paper [6] talked about an algorithm to approximate the median of N elements using the soft heap. The approximate median you find will at most be εN ranks away from the correct median. The algorithm works by adding all your elements into a soft heap. From there you would then perform $\lfloor \frac{N}{2} \rfloor$ FIND-MIN and DELETE-MIN operations. While removing the

elements you want to note down the element that have the biggest real key, this element will also have the biggest rank of the elements you have removed. Now, the statement is that this element will be at most εN ranks away from the correct median.

Lemma 2.10.1. *After extracting $\lfloor \frac{N}{2} \rfloor$ elements, the one with the biggest real key of the extracted elements, will be within εN ranks of the median.*

Proof. By definition of soft heap we have that at most εN elements can be corrupt. This statement sill holds true after extracting half the elements. This means that out of the remaining $\frac{N}{2}$ elements at most εN can be corrupt.

We first look at the upper bound. After having extracted $\frac{N}{2}$ elements the one with the highest key will at most have a rank of $\frac{N}{2} + \varepsilon N$. To show why this is the case, let us assume that we had the maximum number of corruptions and that all the corrupted elements have lower rank than the biggest real key element we end up extracting. Any corrupted elements with higher rank will not affect the result. Let us say that they are corrupted to some arbitrary high value, higher than any other real key. If you now imagine a list where the elements were in order of the current key, then you would see that this pushes all the highest εN elements down the list. This therefore means that the biggest ranked element we would be able to read would be the element with rank $\frac{N}{2} + \varepsilon N$. This means that the bound hold on the upper side.

Let us look at the lower side. Since we have extracted $\frac{N}{2}$ elements then if no elements were corrupt then the highest element would be the median. Any corruptions that happens can only make us read elements with higher rank. This means that the highest element we have read cannot be lower than the median. From this the lemma follows. □

Lemma 2.10.2. *The running time of approximating the median can be done in $\mathcal{O}(N)$ time.*

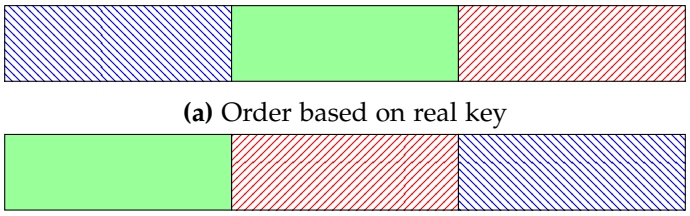
Proof. The algorithm does N INSERT and $\frac{N}{2}$ FIND-MIN and DELETE-MIN. These take amortized $\mathcal{O}(1)$ time per element for a fixed ε. This means that the total running time is $\mathcal{O}(N)$. □

Chazelle [6] notes that other solutions to this problem that also solves it in linear time requires a variant of the median-finding algorithm described in [2].

2.10.1 Finding the k-th smallest element

Instead of making an approximation, can also get the extract median, or in fact we can get any k-th element in the structure. This is refereed to as the *standard selection problem*. Chazelle [6, Application (3)] said that this is possible and comes with an example using $\varepsilon = \frac{1}{3}$, this gives him a running time of $\mathcal{O}(N)$. Kaplan et al. [14, p. 5] later explains how to do this too referring to Chazelle [6], but this time shows you can use any $\varepsilon < \frac{1}{2}$ and uses $\varepsilon = \frac{1}{4}$ to also get a running time of $\mathcal{O}(N)$. Before moving on, let us be specific about the problem definition.

Finding the k-th smallest element is defined to be the problem where you are given N elements from a totally ordered domain and then for any integer $1 \leq k \leq N$ you have to return the k-th smallest element. If k is equal to either 1 or N then it is easy to solve this problem in $\mathcal{O}(N)$ time. In this case we would go through the whole data set, only remembering the smallest or largest element seen so far. It should also be clear we



(a) Order based on real key

(b) Order based on current key

Figure 2.10: Each colored block represents a number of element. These elements can be sorted according to their (real) key (a) such that all the smallest elements are in the first box and the biggest elements are in the last box. If you put all of these elements into a soft heap, then key can get corrupt, making the current key different from the real key. This means that if you extracted all the elements from a soft heap, then the element might not be in order of the real keys (b)

cannot do it faster than $\mathcal{O}(N)$ since we at least need to check every element once since every element might affect the k-th rank.

For our example we will use $\varepsilon = \frac{1}{3}$. Solving this problem using soft heaps requires you to first insert all N elements into a soft heap, followed by $N\frac{1}{3}$ extractions. As you extract elements you note down the element e with the largest rank/real key, just like we did when find the approximated median. Now, the rank of e is going to be bounded to ranks between $N\frac{1}{3}$ and $N\frac{2}{3}$. For the lower bound we use the exact same argumentation as we did in Lemma 2.10.1; since we have read $N\frac{1}{3}$ elements e must have a rank of at least that. The upper bound also follows closely from the lemma. After having read $N\frac{1}{3}$ elements, there are $N\frac{2}{3}$ elements left in the heap. At least $N\frac{1}{3}$ of these are not corrupted this means that the element with the largest rank we can have seen, is the element with rank $N\frac{2}{3}$. We illustrate this on Figure 2.10 where we can see how given that the real order is what we seen on (a), then even if all the smallest elements (blue elements) are corrupt, then we are never able to read any of the last $N\frac{1}{3}$ elements (red elements) if we only read $N\frac{1}{3}$ elements. This is illustrated on (b).

Now that we have an element e that we know have rank between $N\frac{1}{3}$ and $N\frac{2}{3}$ we calculate the rank r of e just by counting the number of elements lower than e. If $r = k$ then we have found the element we are looking for. If $r > k$ then we can remove all elements larger than the element that had rank r. If $r < k$ then we do the same but for the smaller elements. Here it is important to notice that we remove at least $N\frac{1}{3}$ elements from consideration.

Now the only thing left to do is to recurse. Each recursion will cut away at least $\frac{1}{3}$ of the remaining elements or maybe more. What we really are doing is finding an approximate median within some bounds and then recurse on that. Compared to just using a random median, this gives us the bounds we need to give a better analysis. A worst-case analysis if we picked random element to split around would give us $\mathcal{O}(N^2)$ running time.

Lemma 2.10.3. *The running time of finding the k-th element using soft heaps is* $\mathcal{O}(N)$.

Proof. We use same argumentation as Lemma 2.10.2 to get the running time of one iteration, if there is n elements then it takes $\mathcal{O}(n)$ time. This means that the first iteration takes $\mathcal{O}(N)$. Each iteration is going to take away at least $\frac{1}{3}$ of the elements. This means

that the second iteration takes at most $\mathcal{O}(\frac{2}{3}N)$, and the next one $\mathcal{O}(\frac{2}{3}^2N)$. This gives us the total running time of $\mathcal{O}(N) + \mathcal{O}(\frac{2}{3}N) + \mathcal{O}(\frac{2}{3}^2N) + \cdots = \mathcal{O}(N)$ since we have:

$$\begin{aligned}\sum_{i=0}^{\infty}\left(\frac{2}{3}\right)^i N &= N\sum_{i=0}^{\infty}\left(\frac{2}{3}\right)^i \\ &= N\left(\frac{1}{1-2/3}\right) \\ &= 3N \\ &= \mathcal{O}(N)\end{aligned}$$

□

Alternative solution

Solutions to this problem is not limited to only using soft heaps. Using a soft heap might now even make sense if the running time is bad compared to other, maybe simpler, alternatives. We have seen that the running time is linear for soft heap, but other solutions might still be faster.

As mentioned by Kaplan et al. [14] this problem has been looked at by multiple people. Blum et al. [2] have proved that this can be done deterministically in $\mathcal{O}(N)$ time. People have also looking into the number of comparisons needed for selection. Schönhage et al. [19] proved $3N$ comparisons were needed, and Dor and Zwick [8, 9] lowered it to $2.95N$.

The solutions we are going to test the soft heap implementation against is the following two.

Random select Random select works by selecting a random element and then splitting the elements into elements that are larger than the selected element and elements that are smaller. It is described in [7, Section 9.2]. After knowing how many elements are in each group makes us able to recurse into the correct group based on the rank we are looking for, just like the soft heap implementation. This algorithm borrows ideas from Quick sort. However here we only recurse to the side we are looking at.

Compared to the soft heap implementation we save a lot of work by just picking the element at random, but we also lose any guaranties on the element we pivot around. This can be done in an array which might improve the performance due to be cache utilization, since we will access the elements in order and only access a few places in memory at a given time.

Since we pick the element we sort around at random the worst-case total running is only bounded to $\mathcal{O}(N^2)$; each iteration we might select the item with rank 1 or the highest rank, this will only remove one element. However, this is very unlikely and the expected running time is $\mathcal{O}(N)$.

Deterministic select Based around the same concept as the previous, but this time we try to get a better pivot like in soft heap, but without using soft heap. This modified version can be found in [7, Section 9.3]. The way we do this is, that we imagine that our big input is made of small blocks of a few elements. For each of the blocks we want to find the median. This can be done by sorting each of the blocks. Since we only sort a

fixed amount elements, this should essentially be constant time per group of elements in $\mathcal{O}$-notation. Once sorted the median is just the middle element of the group. We move all medians next to each other. We now repeat the procedure on the medians to find the median of the medians. This is done until you have a single element. From there you use the element to partition the data into two, just like you would for random select.

Selecting a good element to split around is important as this directly affects the amount of recursion call we are going to have. Optimally we will recurse in the middle, splitting the input into two equal sizes. The new running time of this is $\mathcal{O}(N)$.

When implementing, if done correctly this can be done in a single array without having to make new. This should be good for the cache.

Other solutions Another solution that can solve the problem would be just to sort the data. Furthermore, in practice using complex algorithms might yield a better time complexity but perform poorly at low inputs compared to simple solutions. As the input gets smaller the better complexity might not be able to pay for all the extra overhead that the complex structure might create. This means that you might want to switch algorithm to a more simple one once the input get small enough. You can in fact do this without breaking your $\mathcal{O}$-notation analysis. If you decide a change algorithms at a fixed number of elements, then you can use that as a constant and therefore ignore it.

Finding the k-smallest elements

An alternative definition to this problem is to find the k smallest elements instead of just the k-th element. If we have found the k-th smallest element then we would be able to go through the list to find every element smaller than k. Going through the list and comparing each element once will take $\mathcal{O}(N)$ time. As noted before, finding the k-th element cannot be done faster than $\mathcal{O}(N)$ time, therefore the total time in $\mathcal{O}$-notation will still be the time it takes to find the k-th smallest element.

2.10.2 Heap selection

This problem is related to finding the k smallest elements for $1 \leq k \leq N$, but this time the input is in a heap-ordered binary heap instead of just a list. This means that the value of each node is greater than its parent. This means that the root of a given tree will be the element with the lowest value. You have no relation between two children other than that they are both smaller than the shared parent. The tree can either be balanced or not. We denote the two children of a given node `x` as `x.left` and `x.right`. The problem and how to solve it are introduced in [14, Section 3]. Here it is also noted that there exists a very complicated solution Frederickson [10] that can solve it in $\mathcal{O}(k)$ comparisons. This matches the information-theoretic lower bound. Compared to before we no longer have the argument that we need to visit every node at least once we than in finding the k-smallest. However, we can say that we need to at least do $\mathcal{O}(k)$.

Like before we will see that there are some relatively straightforward solutions and then the more complicated solution using soft heaps. This is again because some code takes advantage of the bounds that the soft heap creates. We will also see that for this application we need the strengthened interface. To ease into the problem we start by looking at the non-soft solution to the problem.

For the implementations we will assume that the input heap does contain k elements, and we also assume that you will not hit nodes in the tree that do not have two children. This makes the pseudo code much cleaner. In the real implementation you could either make null checks or generate a node to be used as leaf that would have a weight of ∞ such that it would never be picked in the priority queue, just like we did in the simplified soft heap.

Non-soft solution

We know by construction of the heap that the element with the lowest value is the root. If the element we want is the first element we just directly return the root. If not, then we take the two children (if they exist) and add them to a priority queue. Each of the children will be the smallest element of their own subtree. This means that if we take an element from the priority queue we get the smallest of the two subtrees, which will also be the second smallest element. Now, repeat this process of adding the two children and extracting the smallest element from the priority queue until you get to the k-th element. Pseudo code for this can be seen on Algorithm 2.

Algorithm 2 ([14, Figure 1]) Selecting the k smallest elements from a binary heap

```
1:  function heapSelect(r)
2:      S ← array()      // Can initialize with size k
3:      Q ← heap()
4:      Q.insert(r)
5:      for i ← 1 to k do
6:          e ← Q.extractMin()
7:          S.append(e)
8:          Q.insert(e.left)
9:          Q.insert(e.right)
10:     return S
```

Creating and inserting a single element into a heap can be done in $\mathcal{O}(1)$ time. For there we move into the for loop that will execute k times. Because of the priority queue it should be easy to extract the smallest element. Inserting the two children will each take $\mathcal{O}(\lg k)$ time. This extracts not only the k smallest elements, but the k smallest elements in order. A total of $2k+1$ elements have been inserted into the priority queue, the root plus the two children of every element extracted of which there were k. This gives a running time of $\mathcal{O}(k \lg k)$, which is the optimal if they need to be returned in sorted order.

Soft solution

The soft implementation of this is based around the same idea, but because of the corruptions the algorithm needs to have some changes made. It is also here we need the strengthened interface so we have more knowledge about which elements can be corrupt. The algorithm we will see first does not return the k smallest elements, but only a set of candidates. We then take these candidates and then use a standard selection algorithm to do the final selection. The first part, that uses soft heap, can be seen below:

Algorithm 3 ([14, Figure 2]) Selecting the k smallest elements from a heap using a Soft Heap

```
1:  function softSelect(r)
2:      S ← array() // Can initialize with size k, but need to have dynamic size
3:      Q ← softHeap(1/4)
4:      Q.insert(r)
5:      S.append(r)
6:      for i ← 1 to k-1 do
7:          (e, C) ← Q.extractMin()
8:          if not e is corrupted then
9:              C ← C ∪ {e}
10:         for e′ ∈ C do
11:             Q.insert(e′.left)
12:             Q.insert(e′.right)
13:             S.append(e′.left)
14:             S.append(e′.right)
15:     return select(S, k)
```

Like the with the non-soft solution we have a for loop that will extract elements, making sure we extract at least the minimum k elements. Because we want to be on the safe side any elements that gets returned as possible corrupt will also be added to the return list. We see this is the loop where it first extracts an element. If this element is corrupt, then it has already been added to S from another iteration. If it is not corrupt then we add it to C. Now we will go through all the elements in C and add the children both to Q and S. After $k-1$ iterations we should have $\mathcal{O}(k)$ elements where the smallest k elements are part of it. If we take the candidates we can then use a normal selection algorithm and get the smallest k elements.

3 Prerequisites for experiments

Before we get to the experiments part of the report we want to introduce some of the methodologies we have surrounding code quality and testing. We will also introduce some of the hardware components that is going to affect the performance and introduce some of the tools we are going to use to measure the performance.

3.1 Hardware knowledge

This section contains descriptions of some of the hardware components to watch out for when you want to write performance code. These observations will also be referenced in the later experiments' sections.

3.1.1 Cache

Cache is super fast memory that is located on the CPU itself. The cache memory is many times faster than the standard RAM, but is often very limited in size. The cache is often comprised of several so called levels. The first level, called level one cache, is the smallest but also the fastest cache. On a normal CPU you will often find up to three levels of cache, where the last is often referred to as last level cache. Each core on the CPU often have its own dedicated piece of cache at the lower levels, however on the upper levels this is often shared.

Cache is important for the performance of modern computers. While a program is running the contents in cache will usually be swapped many times over. As noted the cache is many times faster than the RAM, but since the cache is so small, then the cache have to wait for the RAM when it requests data. This can result in the CPU idling and thereby loosing performance. This means that limiting the amount we have to talk to the RAM can be very beneficial. This can be done both by reducing the amount of data we are working on and by improving the way we access memory.

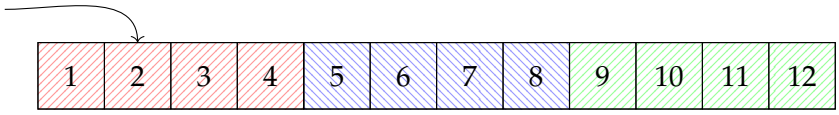


Figure 3.1: Reading a single value from RAM will load all the values found inside the same patterned boxes. Reading the value that the pointer points to will load all number in the red (1, 2, 3, 4) into cache. Therefore, accessing values in order can benefit the cache performance

3.1.2 Cache lines

Whenever you pull something from RAM into cache you pull a chunk that have a fixed size. This chunk is called a *cache line*. The size of the cache line is fixed and cannot be changed, but will affect how much data you get. If you design your algorithm around this knowledge then you can make it such that you do not have to pull from data constantly. This is something that can have a big effect on the performance. Figure 3.1 illustrates how multiple separated values can be in the same cache line, and how reading a single cache line can pull multiple items into cache. If you were iterating an array you would save a lot of time because often the next element will already be in cache.

We can get the specific cache line size on Linux by looking at output from the following command:

```
1 getconf LEVEL1_DCACHE_LINESIZE
```

On our test computer this command tell us that the cache line size is 64 bytes which is quite typical for current desktop CPUs. Often, even if we only needed 4 bytes for an integer, reading more than those bytes can be quite beneficial since the next data is often related. If we access a value that is already in cache, then we call it a *cache hit*, as noted this can happen when you access elements close to each other in memory. It can also happen if you use the same value again, before it have been pushed out of cache to make space for something else. If the data is not found in cache, then we call it a *cache miss*.

3.1.3 Branch prediction

Generally code contains a lot of branches. These can come from if-statements, but can also happen in constructs like loops. Each time there is a branch the computer needs to know which branch to take. A problem can happen because computers nowadays do instruction pipelining, where it actually works on multiple instructions at the same time. This means that it might not know which branch is going to be the correct one since it have yet to complete the instruction that will decide the branch. It can also happen that the value the branch depend on is not loaded from memory yet. To make it such that we do not just idle and wait we can do branch prediction.

Branch prediction is simply the CPU guesses at what branch we are going to take and starts executing that. While executing the branch it does not commit any of the calculations we make. When the data needed to decide the branch is ready then the CPU will either commit the calculations if the prediction was correct or go back if the prediction was wrong. This means we can gain performance when the CPU guesses correct, and if we were wrong we do not lose anything. All of this happens on the CPU

without intervention from the developer.

Branch prediction can often catch on to patterns. This means that when running a while-loop the CPU will often quickly realise that the loop loops, and predict that the loop does not break. Once we break out of the loop we will get a missed branch prediction, but if the loop makes many iterations, then we have gained a lot of performance.

3.1.4 Vectorization

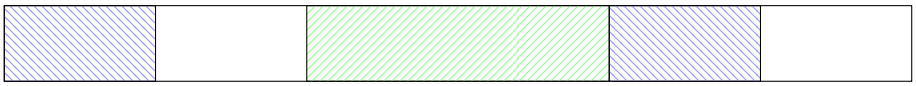
Vectorization is a feature that enables you to perform a **s**ingle **i**nstruction on **m**ultiple **d**ata, this is known as SIMD. It can be used when you want to use the same instruction on different data at the same time. Originally intended to help processors cope with multimedia content where a lot of pixels needed to be processed by the same operation, but have later been expanded to include a lot of different operations. These commands should enable great performance increase, but there are some limitation. The operations are quite picky about how the data is laid out in memory, this means that arrays are the way to go if you want vectorization. Applying vectorization is not something you would normally do yourself. It is the job of the compiler, but you still have to write your code such that it can get vectorized, and sometimes help the compiler by guaranteeing certain conditions that the compiler cannot infer from the code. This could be that two pointers cannot be aliases.

3.1.5 Stack alignment

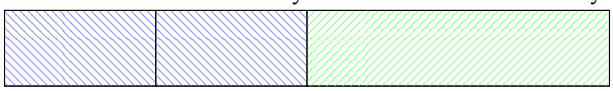
Stack alignment makes sure values are aligned in memory. What this means is that it tries to place memory with certain intervals in memory. The reason for doing this is that the computer can have an easier time reading from memory when it is aligned and some operations might even require it. Some architectures are more strict about this than others. x86 is quite versatile in this regard and can read unaligned data at some possible performance degradation. Other architectures might not support it or some instructions might not.

Below we see a practical example of how stack alignment can affect our code. We define two structs that each have the same elements inside of them. The size of each of the elements are $4 + 4 + 8 = 16$ bytes, but depending on how we pack them we might end up spending more space. Here the reason is stack alignment. For the `firstStruct` it is going to align the integer pointer to an 8 byte alignment, and therefore push it 4 bytes, leaving empty space. The same is going to happen in the end. We show how this looks on Figure 3.2. If we had a long array of these structs, just swapping these two could save a lot of space, and also improve performance because more can fit in cache, and because more can be in a single cache line.

```
1  struct firstStruct{
2      int a;          // Size of 4 bytes
3      int *b;         // Size of 8 bytes
4      int c;          // Size of 4 bytes
5  };
6  std::cout << sizeof(firstStruct) << std::endl; // Outputs: 24
7
8  struct secondStruct{
9      int a;          // Size of 4 bytes
10     int c;          // Size of 4 bytes
11     int *b;         // Size of 8 bytes
```



(a) `firstStruct` in memory with a total size of 24 bytes



(b) `secondStruct` in memory with a total size of 16 bytes

Figure 3.2: Here the blue areas indicates integers, the green areas indicate integer pointers, and the white space is unused. Here we can see how it does not like to cross the 8 byte alignment, if it does not need to

```
12 };
13 std::cout << sizeof(secondStruct) << std::endl; // Outputs: 16
```

3.2 Build environment

We use CMake to build executable targets of our code. The compiler we have used is GCC (GNU Compiler collection). GCC support many features which and be used by passing different flags. These flags include `-Wall` and `-Wextra` to enable a number of warnings about your code, `-g` for debugging, or `-ftree-vectorize` to enable vectorization. Many more flags exists, this is just some we have used. Another important one is `-O` followed by a number between 0 and 3 this will choose an optimization level, with 0 meaning no optimization. Some tools require us to compile for debugging mode. When debugging lowering or disabling the optimization can also make it a lot easier, as the optimizer might optimize a lot of code away.

3.3 Test tools

There exists a number of tool that can be used to capture information about a program. In this section we cover the tools we are going to be using in out testing.

3.3.1 GDB

GDB (GNU Project Debugger) is a very common debugger and integrated into CLion. We have used this while developing and use it in some of the tests. Through CLion we can also gain direct access to the GDB console. This enables us to use commands such as `disas` when the code is paused from a breakpoint. This will print the assembly instructions of the function we are currently in. `disas` is short for disassemble. Another feature we will be using is printing raw memory. This can be done using the `x` command. The `x` can then be followed by the address and it will print the values in memory at that location. GDB is a solid debugger that support loads of other stuff too.

3.3.2 perf

perf is a statistical profiler that features many different modes that can be used to measure different aspects of running programs. The part about perf being statistical comes

from the fact that it collects its results by performing samples. One of the ways to collect samples is to use the mode called `record` with an executable program as argument. This stores the observations in a file and which can then be displayed in different ways, one of them being `annotate` which presents the data in a tree-like view. Each C++ function that was called is shown and its children which is the functions that it calls. A percentage is shown next to each function. The percentage is based on the samples perf made. It samples by stopping the program execution at some frequency which can be specified as an argument. perf then *records* which functions are on the CPU stack. The more a function is on the stack the higher percentage it gets. That perf is a statistical profiler is one of the reasons we choose to use it for our profiling. Some profilers like Cachegrind gets some of its results by using a simulation that can be thought of as a virtual machine. A virtual machine is hardware independent i.e. it does not use the hardware directly. Using virtual machines for testing is not ideal so that is why we use perf for some of our results. With that said then the results of Cachegrind are still useful since they should be a depiction of what is really going on, on the hardware. perf also has a mode called stat `stat` which uses the counters provided by the hardware to obtain information. This mode also takes a program as input. It will execute the program while gathering information such as how long it took, cycles, branches, branch misses, context switches, and much more. If these stats are not of interest to you, then you can do `perf list` to get a complete list of all events it can track on a given CPU. This list also includes cache related events. From there we can use the `-e` to specify what variables we want to track. Like so: `perf stat -e [things to track] [program] [arguments]`.

To enable some features of pref we needed to change the value located in `/proc/sys/kernel/perf_event_paranoid`. We change this to -1 to enable the user to track most kernel events of the operating system. We later see that this actually makes a difference for one of our cache tests in Chapter 4.

Flamegraph

We mentioned that one of the ways of displaying the knowledge which can be derived from the samples is to use `annotate` which gives a tree-like view of the functions that have been called. The problem with this view is that it is very difficult to get an overview of what is actually the function that takes up most of the execution time and why that function does that. To remedy this we use the program call flamegraph which requires one to run `record` with flags that annotates debug information. We use flamegraphs in Section 5.2 to diagnose what code in our soft sequence heap implementation is making the overall runtime slow. An example of a flamegraph can be seen on Figure 5.4 on page 97. It may appear as some of the long C++ function names are not fully readable, but that is not the case when opening the SVG version in one's browser which we link to in the aforementioned section. Using the SVG one can also hover over the boxes to see the exact percentage which again is the time the function is on the CPU-stack. The width of the boxes is the time on the CPU-stack.

3.3.3 Valgrind

Valgrind is a set of tools that can help the programmer analyse the code he has written and find bugs or problems. The two tools we have used are *memcheck* and *Cachegrind*. We can run the different tools using `valgrind -tool=[tool name] [additional parameters]`.

Memcheck

As the name suggests, this tool is focused around checking memory. This will ensure the help ensure the correctness of the program and improve the quality. When programming C++ you can quickly make memory mistakes once a program gets complex. It is therefore nice to have a program that can tell you if there are some memory that is never deallocated or if there are some memory that have been deallocated and is still used. It works by analysing how memory is used for a specific run of the code. Since we already have test cases written it requires no additional work to just run memcheck on those. Memcheck cannot not tell you where you should have deallocated, but can tell you where an object is created that is never destroyed, so it is not a silver bullet. CLion has memcheck integration, so we can use the CLion interface to inspect the results from memcheck.

Cachegrind

Cachegrind is a tool that can profile cache usage. Compared to pref, Cachegrind simulates the execution of a given program. This means that instead of just getting some metrics for the whole program we can get metrics for each line of code. This enables much better reasoning about how a specific thing uses the cache. Another positive is that it limits the number of things that affect the results. This means that if you want to compare two different implementation, then you do not have to worry about how other programs might use the computer while running the test. This can also be a downside because the simulation is only a simulation. The simulation simulates the first level cache and the last level cache of the given system it is running on. The simulation also do not consider how other resources on the computer might use the computer. Moreover, the simulation will run much slower compared to just running the program normally. More information about the limitation can be found in the documentation[1].

Cachegrind confirms the importance of using the cache properly. According to the documentation of Cachegrind[2] a first level cache miss on a modern PC will cost around 10 cycles and a last level cache miss will cost around 200 cycles. They also note that a branch misprediction can cost around 10 to 30 cycles.

To use the program we simply call `valgrind -tool=cachegrind` and provide our binary at the end in addition to any arguments we have. When the command is done executing it outputs an overview of its findings. This includes the information about how many instructions and how many memory reads and write have taken place for the whole program. Cachegrind can also simulate branch prediction, this is done by simply adding `-branch-sim=yes` when you call it.

Annotating The output to console from Cachegrind only shows a resume, but it also creates a file with a lot more information. This file can be open with `cg_annotate [filename]`. Adding the `-auto=yes` will make add how each line of code affects the performance.

After using the tool for a bit it seems that the annotation might misplace where things are happening e.g., it might annotate that an assignment has zero writes while the above line have a lot while it is not assigning anything. We think this is due to the

[1] `https://valgrind.org/docs/manual/cg-manual.html#cg-manual.sim-details`
[2] `https://valgrind.org/docs/manual/cg-manual.html#cg-manual.overview`

optimizer optimizing the code and thereby making it hard for Cachegrind to figure out exactly which instruction originates from which line of code. This make sense since the optimizer might move the control flow around and combine multiple lines into new ones. Most of the time it should be pretty clear where the read and write should originate.

3.4 Timing the performance

In this section we will talk about how we will time the performance of a program. As noted before pref can measure the time it takes for a program to complete, but timing the performance inside the code lets us decide from where and to where we want to time.

To time a program we use a part of the standard library called *chrono*. It was introduced in C++11, before that you had to use other libraries or features provided by the operating system. Chrono provides three clocks `system_clock`, `steady_clock`, and `high_resolution_clock`.

The `system_clock` provides access to a real time wall clock. Since it is a wall clock it is system wide, and can be used across applications. The actual reference point in time is unspecified in the specification before C++20 where is was fixed to being the Unix Time measure. However, many implementations before already used this.

The `steady_clock` provides a monotonic clock and is not related to wall clock. This means that if you take the time at two different points t_1 and t_2, then $t_1 \leq t_2$ if t_2 was taken later. You might think this is the case for all clocks, but this is in fact not guaranteed for the others. The reason the other might not be is because they might sometimes have to do correction, thereby in a sense turning time backwards. You can check if a clock is steady by using the `is_steady` function.

The `high_resolution_clock` is the clock with the highest precision the system supports. Often this is just a wrapper for one of the previous mentioned clocks, but can also have its own implementation. The main problem with using this is that it is implemented differently depending on the system and the operating system. This means that in most cases you might be better off using one of the previous clocks directly, such that you know what you are getting.

We choose to use `steady_clock` because its precision should be high enough, even if there exist things with higher precision. A preliminary test shows that its accuracy is down to 100 nanoseconds between ticks, and should therefore easily be able to measure our workload[3]. It should hopefully provide more consistent results given that it is monotonic and will not make corrections. It is also the recommended for measuring intervals by public documentations[4].

The way we time our code is quite simple. Using the library and our desired clock you can ask for the time right now using `now()`, this will return a `time_point` which we store. Next we execute the thing we want to time and after that we note down the time again. Chrono provides functionality for figuring out the duration between two points in time and also converting it into a desired time scale. Nanoseconds is the lowest it provides. Once you have this you can use the `count()` function on duration to give you the amount of time passed, in this case in nanoseconds.

[3] As noted before this might change depending on the system.
[4] cppreference.com and cplusplus.com

Algorithm 4 Timing your code

```
1: auto start = chrono::steady_clock::now();
2: // Code to time
3: auto stop = chrono::steady_clock::now();
4: auto duration = chrono::duration_cast<chrono::nanoseconds>(stop -
   start);
```

3.4.1 Clock overhead problem and solution

We have already talked about how the operation cannot be too quick to complete or else we cannot measure it. We also tested that we could measure down to a tick of 100 nanoseconds, so for anything that are not instantly complete we should be fine, but there can be another problem. When we start to instrument our code we can affect the performance by adding overhead. We can minimize the effects of overhead by either repeating the thing we want to measure multiple times and then divide the by the number of times we did it. This will split the overhead across a wide number of runs and therefore become minimal. However, doing this approach also looses information. We no longer know how long each individual run takes and we cannot time individual part of the code. We elected not to do this because we wanted to be able to test individual parts of the program. We also know that our tests, as soon as the input reaches a certain point, should take minimal effect from the overhead. Since many of our tests are about adding a lot of elements and then removing them again, we almost already have the effect of repeating a function many times and dividing by the number of calls.

3.5 Experiments setup

We now know how to time our code properly we also need to run it. Here we explain how we set up a system that can run many tests without us having to manually do it. We try to run as little other stuff as possible to minimize variance and effect other programs can have on the tests.

There are two parts to our testing setup. Since the number of tests are so large, doing it manually is simply not an option. The first part to the setup is making a program that can execute all the different tests we have. We do this by making a program that can take a number of parameters on the command line and then use these input to decide that test to run and whit what input. An example would be that `[program] 1 2 10000`, would run test number 1 with the second variation of the implementation in that given test and an input size of 10000.

The second part is our bash script. Inside the bash script we can easily define the parameter bounds we want to use and then run our program as many times as wanted. It will then note down the results after each run to a file that we can use later. This also makes all the runs independent since each run is a new instance of the program. This should minimize any effect different runs should have on each other. Doing it like this makes it easy to rerun all the test.

3.6 Plotting the results

Later in the experiments we will show a lot of plots about the running time of our program. In these plot where we used multiple samples it is going to be the minimum sample unless otherwise stated. The idea is that there is some ideal case where the computer was not disturbed and the minimum is the closest to it. Average have the problem that if we have some bad runs then they are going to affect the average, but not the minimum. For the minimum to be affected all runs needs to be bad, which can happen but are very unlikely, and still better than average where any number of bad runs affect the plot. In some case where the execution is heavily affected the by input, using minimum might is not ideal, and we will note when this is the case.

To create plots that that are made using vector graphics we use the Python packages called matplotlib and pandas.

3.7 Code quality

When programming you can often make mistakes and oversights as the program becomes complex, therefore it is important to relay on more than reading through the code for correctness. Here we describe some things we do to ensure that our implementations are correct. This also makes it so you can be less afraid about changing stuff in fear of breaking something.

Asserts To ensure the code correctness we use a number of asserts inside the code. These asserts will throw an error if something is wrong. We use these on invariant to spot errors. The asserts are only present when the code is run in debug mode, and will therefore not affect the performance of the program when the program is run in production mode for the timing tests

Tests We have written tests that will test the data structures in a number of ways to make sure they behave as one would expect. The tests consist of smaller handmade examples and bigger computer generated examples. In the computer generated ones we do a lot of experiments to try and catch rare cases where problems might happen. All the experiments are seeded such that it is possible to generate the same test case if it should fail.

Memory management Since C++ do not have a garbage collector we have to make sure to clean up after ourselves. We define our own destructors for objects that needs it and we also test with both test cases and mencheck. We make sure to test destructing structures that are not fully empty and melding structures.

Additional internal testing To test as many invariants as possible we have added additional functions for that. An example of this is a function added to the soft heap implementations that will return the current number of corrupted elements. We can then check this is in the range it needs to be in additional test cases.

3.8 Test machine

We will now describe our test setup i.e. the machine we run our experiments on. All our experiments have been run on the same computer such that we can make fair comparisons of the results. First we describe the hardware and then the software.

The test machine is a desktop and its CPU is an Intel i5-4670k. That CPU has a 64 kilobyte level 1 cache per core, a 256 kilobyte level 2 cache per core, and a shared level 3 cache of 6 megabyte. The test machine has three DDR3 4 GB memory sticks and therefore 12 GB memory in total. The speed of one the memory sticks is 1333 MT/s which stands for mega transfers per second. This should enable one to use pretty large input sizes without running out of memory. If we run out of memory then the OS pushes data from the memory to the 12 GB swap partition we have made. We try to avoid using the swap partition since it stored on disk which is much slower then memory.

We will now describe the software. We wanted a minimal system in the sense that it ran as few processes in the background as possible. We decided that the Linux distribution called Arch Linux would be good for this purpose. Specifically, we use Linux kernel version 5.6.13 on an x86_64 architecture. Furthermore, we use GCC version 10.1.0 to compile our code and CMake version 3.17.2-2 to generate our Makefile. The base install of Arch Linux does not feature a desktop environment, networking software, audio software or similar programs that runs in the background. This suites our needs fine since we purposely only use the machine for running tests. The reason for this is that if we used the machine for other purposes at the same time as running tests then that might impact the results. When we want to run a test or interact with the data then we use SSHFS to virtual mount the SSD of the test machine and SSH to run commands.

4
Experimenting with lists

In this section we will perform tests that use lists and see how they perform in the real world. This section will use the theory about list performance described in Section 2.3, and will also use the knowledge and tools discussed in Chapter 3.

The goal of this section is to get a understanding of what affects performance in practice. This can also show how the constants lost in $\mathcal{O}$-notation under the RAM model can be important when comparing two algorithms with the same dominant part in their time complexities. We hope to see how e.g., cache or vectorization affects the performance and see how accurate the $\mathcal{O}$-notation theory from Section 2.3 is. The tests can also show us how we can write code that better utilizes the hardware. This can help us later when we need to argue why specific implementations should be faster and help us pick which areas might yield the highest reward to explore if we want to improve the real performance of an algorithm — like soft heap.

When we talked about list theory in Section 2.3, we introduced the two main classes of lists - the node based and the array based lists. For each of these classes there are a number of implementations that we are going to test. This should not only show the difference between the two classes, but also show differences between implementations belonging to the same class. We will also see some data structures that do not fit directly into the two categories e.g., C++'s deque.

The main things we show in this chapter are the difference between node based solutions and array based solution. We see how array based solutions can utilize cache and RAM better, but are also able to do vectorization. We also take a deeper look into list and discovers that the minimum allocation size is 32 bytes, explaining with the performance of the singly-linked list is not better than the doubly-linked list. We also see how $\mathcal{O}$-notation does not always capture the trend of the performance as the input size increases. We also see how implementation with same $\mathcal{O}$-notation can behave very differently.

4.1 Test descriptions

We test lists in two different settings. The first test is reading from a given list variant, and the second test is creating a given list variant. For each test we try to not only observe the difference a certain change makes, but also understand why it results in better or worse performance. Therefore, when we observe something that we cannot immediately explain then we sometimes make a detour where we do further experiments to prove what is actually the cause in order to avoid as much speculation as possible. All the tests are executed on the machine described in Section 3.8.

Performance of iterating through a list The first test consists of summing all the elements of a list by iterating over it. In this chapter, summing the elements of a list by iterating through it, will be referred to as "iterating a list" and why we do that will be explained in a moment. Furthermore, the test we just described will be referred to as simply "the iteration test". The iteration test is performed using different list variants.

Before the iteration test can be performed then the given list variant is initialized and integers $0, \ldots, n-1$ are inserted in increasing order. We use integers because that is something which can be summed together which is something we are going to use in a moment. We are now ready to test how fast we can access elements in the list which we do by using a a loop which types depends on the given list. If we just retrieve an element from the list inside the loop and do not use it later in the code then the whole loop will be optimized away[1]. We avoid optimization by adding the value of each accessed element to a variable which is printed out after the loop.

To get a sense of how we collect time measurements for both the iteration test and the creation test then an example of the C++ function used to test STL vector can be seen on Section A.1. We have made such a function for each of the list variants we want to test and they all follow the same pattern. They initialize a list, fills it with as many elements as specified in the argument to the function and then iterate through the list. The chrono clock is queried three times. Before initialization, after the last insert, and after the last list access. The three timestamps are then used to collect the time for list creation and list iteration, respectively.

From theory, we know iterating through a list takes at least linear time for both of the two classes of lists. The lists, we will see, that do not fit into one of these two classes are also able to iterate a list in linear time in the number of elements n. This means that all the implementations we test should iterate a list in $\mathcal{O}(n)$ time.

We note that for the iteration test, because we doing list access right after the list have been created then the list will have some advantages when considering cache. The reason for this is that nothing else have had the chance to push the list out of cache. This applies to all the list implementations.

Performance of creating a list The second test is list creation and will measure the time it takes to initialize and insert a number of elements. We only insert into the back of a list. It is clear from the analysis in Section 2.3 that inserting anywhere but the back will hurt the array based solutions greatly, and inserting anywhere you do not have a pointer to will hurt the node based solutions greatly.

[1] We are compiling with optimization level 3 unless specified otherwise

Node based solutions often have a pointer to the end, since it is a place we often insert elements. This means we can insert in linear time in the number of elements. Array based solutions were a bit more interesting when considering inserts. Here we did the amortized analysis telling us that the amortized time for insert were $\mathcal{O}(1)$ per element. We however also noted how expanding the array will cost time and creating a big enough array will be preferable. We will test these scenarios.

4.2 Iterating through vector and list from STL

The first list implementations we wanted to test were the C++ containers called vector and list[2]. Both of these are part of C++'s standard library (STL) and are examples of an array based and a node based solution respectively. The implementation of vector is a dynamic array. STL list on the other hand is a doubly-linked list.

Since these are what the standard library gives you we expect them to not only be written very well, but also that the optimizer should be familiar with the data structures and should therefore be able to optimize them well.

4.2.1 Vector variants

When using the vector implementation we have two options for iterating through the list. We try both what we call a for-each loop and a for-i loop. The difference between the two implementations in code can be seen on Listing 4.1 and Listing 4.2. We hope to see the difference between using a for-each and a for-i loop. A for-each loop on a vector might work by increasing a pointer to get the next element: `previousElement += offset`. On the other hand, a for-i loop increases a integer that is used to index into the vector: `currentElement = arrayStart + offset·index`. In theory both of these use constant time. The latter might look like it would be slower, but in practice we expect the compiler to make these more or less equal because of optimization.

```
1 for (auto i : testArray) {
2     sum += i;
3 }
```

Listing 4.1: for-each loop

```
1 for (int j = 0; j < size; ++j) {
2     sum += testArray[j];
3 }
```

Listing 4.2: for-i loop

Note the following about for-each loops. If the objects in the variable `testArray` required more space than an integer, then it might be preferable to use a reference to the vector such that you do not copy multiple large objects. However, for small objects like `int`, passing it as a value is fine.

We just saw that there are two ways of iterating through a vector, but for STL list there is only one. When testing STL list we can only do a for-each loop since it does not support effective random access to the elements in the list.

4.2.2 Understanding the implementations

We already know vector and list from STL correspond to a dynamic array and doubly-linked list respectively. However, a lot can happen during compilation so to get a deeper

[2]When we talk about the general concept of a list we use "list" and when we speak about the list that is part of C++'s standard library we write "STL list"

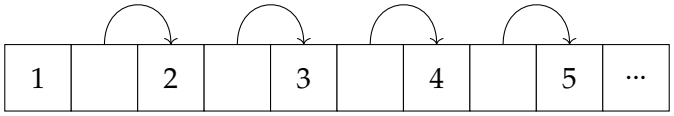


Figure 4.1: If each box is a piece of memory and you continuously make your objects for a linked-list, then you are likely to end up with memory where the elements are stored consecutively.

understanding of the data structure implementations we look at how the code behaves after being compiled. We look at how the two data structures place the elements exactly in memory. Here we see stack alignment in action and also come to grips with why one of the data structures will take better advantage of the cache.

Importance of elements' placement in memory for STL list When dynamically allocating memory it is done on a heap that grows in a direction. When doing several allocate memory calls in succession then it should be *likely* that the allocated memory are after each other i.e. placed consecutively. This means that since we create all the linked-list nodes in succession and in the order they appear in the list, it is likely that the elements will also be stored consecutively in memory and with the order preserved. This would look something like what can be seen on Figure 4.1. The reason for using the word "likely" is that we are not guarantied that the memory is allocated consecutively. Memory that is allocated consecutively improves STL list performance because of the following. When iterating through the linked-list then we need the pointer of each node. If each pointer is stored consecutively then when a block is pulled from memory into the cache then all pointers are hopefully retrieved with the use of that single block instead of having to make multiple read operations on the memory.

Whether it is realistic that the memory looks like this depends on how the data structure is implemented and used. If elements are created at different times or get moved around, you are unlikely to get a good memory layout. Allocations from other parts of the code that is interleaved with our list related allocations can affect the layout negatively. We will test the case where all the elements are created in the order we are going to go through them. We will check that this gives us the consecutive and ordered memory layout described before. Later we will see if it makes a difference when the layout is such that the elements does not come after each other in memory.

The relevance of if the elements being consecutive and ordered affects performance will be important for soft heaps. Soft heaps often performs concatenation of lists, and this can cause inefficient memory layouts. We test later how having the elements unordered affects the performance. For a vector this is not a concern since vector will ensure that all the elements are always in consecutive memory. This is the reason that many operations are cheap, but also the reason why some are expensive.

Verify consecutive memory for STL list Let us try to verify if it is true that STL list stores the elements one after another if we create them in succession. Trying to use the debugger from our IDE to verify this is not easy. If we look at the variable that is a STL list in the debugger, then it will not show you the pointers used in the linked-list, only the values of the elements. From this we are not able to see the exact placement of the elements in memory. If optimization is enabled, then it can also optimize code away,

further complicating the procedure when using the debugger. Thankfully, we can easily just add some additional code that goes through all the elements and output the address of the variables. Here we have to make sure we get the values by reference since if we get them by value it is no longer the elements inside STL list we get, but a copy. Here is a snippet of the output:

```
...
0x7fffda933310
0x7fffda933330
0x7fffda933350
0x7fffda933370
0x7fffda933390
...
```

Looking at the output we can see that the elements in the list are in order, consecutive, and that the space between each element is 32 bytes. This confirms that the layout looks somewhat like what we see on Figure 4.1. We can call `sizeof(int)` to get the size of an integer. It tells us that the size is 4 bytes[3]. Even though an integer only takes up 4 bytes of space, the spacing between elements are 32 bytes. Some of this space is probably used by internal pointers between the STL list nodes. Each pointer on a 64-bit machine is 8[4]. But some of it might also be due to *stack alignment*. Let us try to figure out if that is the case.

Internals of STL list We now know that the distance between elements are 32 bytes and that integers only take up 4, so let us have a look at what is stored around the value. We can get an address of a element inside STL list and use this address to read the memory around that position. We set a breakpoint after the code that outputs the addresses and then ask GDB for a direct memory dump at that memory location. This time the code tells us that the first element is located at $0x8420090$. Using GDB we then see what is stored around that address:

```
(gdb) x/64x 0x8420090-32
0x8420070: 0x0000 0x0000 0x0000 0x0000 0x0021 0x0000 0x0000 0x0000
0x8420080: 0x00a0 0x0842 0x0000 0x0000 0xd600 0xfffe 0x7fff 0x0000
0x8420090: 0x0000 0x0000 0x0000 0x0000 0x0021 0x0000 0x0000 0x0000
0x84200a0: 0x00c0 0x0842 0x0000 0x0000 0x0080 0x0842 0x0000 0x0000
0x84200b0: 0x0001 0x0000 0x0000 0x0000 0x0021 0x0000 0x0000 0x0000
0x84200c0: 0x00e0 0x0842 0x0000 0x0000 0x00a0 0x0842 0x0000 0x0000
0x84200d0: 0x0002 0x0000 0x0000 0x0000 0x0021 0x0000 0x0000 0x0000
0x84200e0: 0x0100 0x0842 0x0000 0x0000 0x00c0 0x0842 0x0000 0x0000
```

The green values to the left are memory addresses in hexadecimal. To the right of that number we see the values in memory, also in hexadecimal. The next couple of lines are just us very methodology verifying that the elements of the list actually are stored consecutively in memory. Therefore, feel free to go over it lightly.

As expected, we see that there are more values than just the numbers, 0 through $n-1$, that we have added to the list. We mark all the elements originating from the list with red. At the address of the first element, $0x8420090$, we see 0 as we would expect. It might not be clear that this is the correct address since there is quite a number of zeros here, but we can see both 1 and 2, 32 bytes and 64 bytes after the first element. These

[3]This is platform dependent.
[4]Function points can be more

are the addresses $0x84200b0$ and $0x84200d0$. These are the second and third values in our linked-list.

If we take a closer look at the other values you quickly see that some of them look like linked-list pointers since they have values that are just around the addresses we are working on. If we look at the addresses starting at address $0x84200a0$ we see it is a pointer to the next element in the list and that the next non-zero value is a pointer to the previous element. We mark all next pointers orange and all previous pointers blue.

From this we can then deduct that the internal structure of the list is something like this:

```
struct listItem {
    listItem *next
    listItem *previous;
    T item;
    // possible extra value or padding
};
```

Looking at the source code for STL list we can confirm that this is correct. We see that it has a next pointer, a previous pointer, and the user data, in that order[5].

Elements' placement in vector As noted earlier, vector is array based and therefore the elements are packed consecutively. Furthermore, it is also likely more dense compared to the node based approach because we do not need the internal pointers. By dense we mean that there are more elements for a given number of bytes. However, vector might also try to optimize for stack alignment which would increase the space usage to maybe the same as STL list. As before, we print out the addresses of the elements, to see how much space there is between elements:

```
...
0x7fffe6a9728c
0x7fffe6a97290
0x7fffe6a97294
0x7fffe6a97298
...
```

Here we see that the spacing between elements is in fact only 4 bytes. We learned before that the size of `int` were 4, so it is tightly packed. This means the space used to store the actual elements is 8 times larger for a STL list compared to vector!

4.2.3 Results overview

Now that we have a better understanding of what is going on we can look at the test results of the iterating test. We expect both the vector variants to perform better than STL list because as previously discovered, vectors are more densely packed which helps to improve the cache performance. We also think that the code for finding the next element in vector is going to be faster or equal to the time it takes to find the next element in STL list. In STL list it would have to both do a read of the pointer to the next element, and then follow the pointer. In an array based solution, adding a constant to the current pointer should suffice, since the spacing between elements is guaranteed constant. Both cases also need to make a check to see if they are at the end of the list.

[5]STL list source code. Struct is at line 80-177: `https://github.com/gcc-mirror/gcc/blob/4c1b27f961aadecf643d11f2e3e8abb89121fbfb/libstdc%2B%2B-v3/include/bits/stl_list.h#L80`

On Figure 4.2a we see the results from running the tests. The figure includes data from the other tests which we are yet to introduce. The data from these tests will mostly be ignored until we get to them. We will focus on comparing the differences between the three data structures introduced so far. The figure shows the time it takes per element for different sizes of input. Since we use a logarithmic scale for the number of elements then note that the size quickly grows to the point there it might not be able to fit in memory. If a list uses just 32 bytes per element then it would take over 3 gigabytes of space to store the data of 10^8 elements. Figure 4.2a gives good insight into how the algorithms evolve over time, and where the different algorithms starts to perform poorly and how the performance evolve over input size.

As pointed out in Section 3.6, each data point in the plots is the minimum value of 10 runs. Alternatively we could have plotted the average. Doing this for the same data that is used in Figure 4.2a we get the plots in Section A.2. They have the same trends, but overall a bit higher data points as expected.

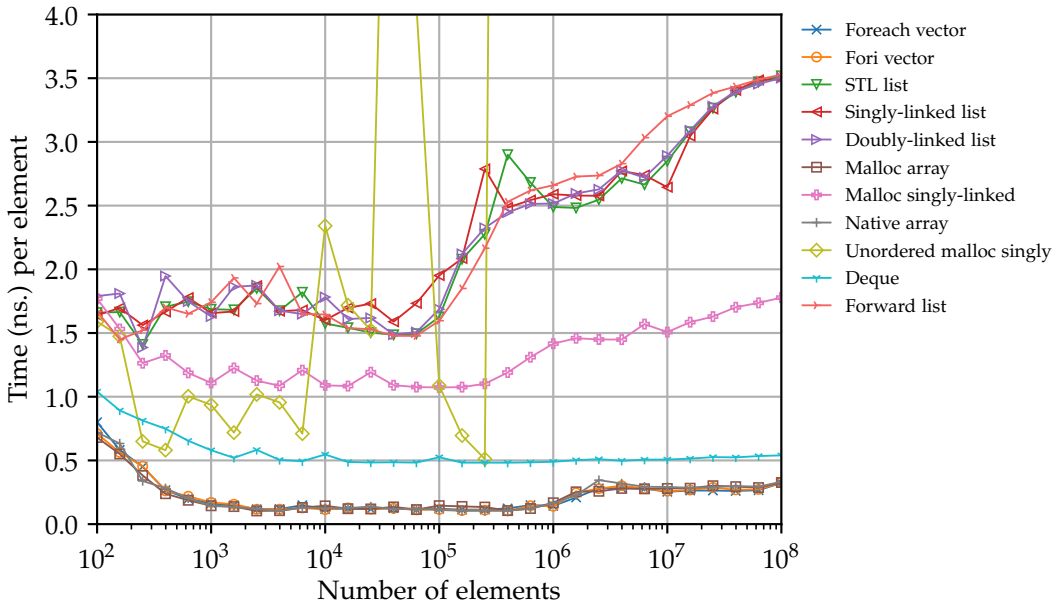
At a quick glance, one result we get from Figure 4.2a is that we can clearly see that the two vector implementations outperform the STL list implementation across all input sizes. We also see that while the performance per element is constant in certain ranges, it is not true over all input sizes. List theory told us that iterating though the list should take $\mathcal{O}(n)$ time, so we would have expected the per element time to take constant time.

We have made a normalized version of Figure 4.2a to get a better perspective the relation between the performances. The relative performance is plotted on Figure 4.2b. Here we have normalized it to the for-each vector implementation, hence the for-each vector is always at 1. We can see that once we are past the smaller input sizes i.e., 10^3, vector is around ten to twenty times faster in our iteration test code.

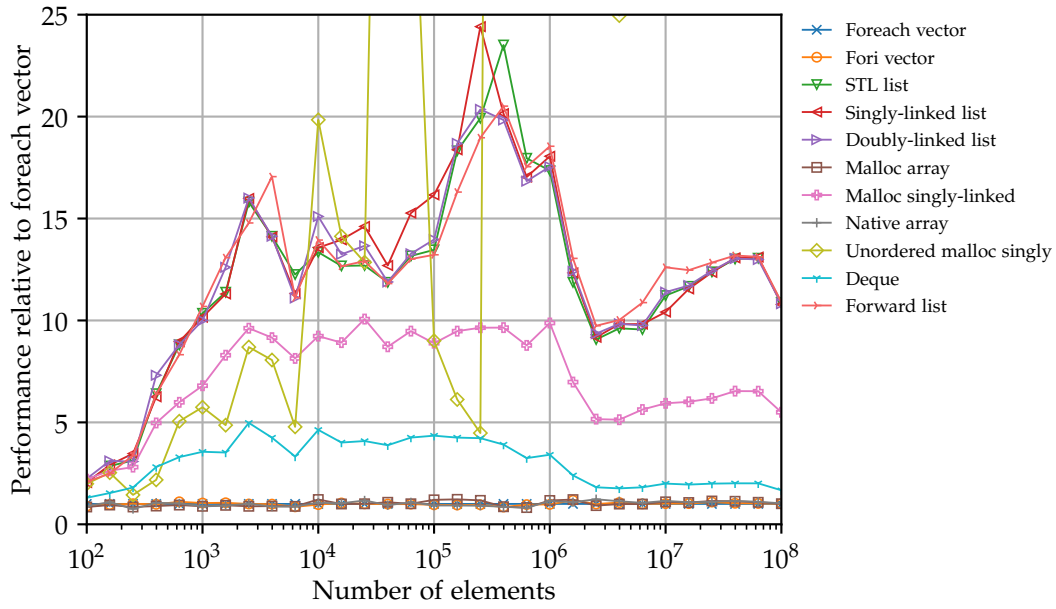
4.2.4 Results for vector

This section is about results for the two vector implementations, which are the blue and orange lines found at the bottom of both figures in Figure 4.2. As seen on Figure 4.2a, vector seems to be having a bit of a problem at the smaller input sizes compared to later. It still beats STL list, but does not perform that well compared to its performance as the input size increases. The reason for the bad performance might be because of some overhead. Some of the overhead might come from getting the CPU up to speed, since modern CPUs might clock down to preserve power when not used. Other overhead might also come from the timing operation, or the overhead in vectorization which is a concept we covered in Subsection 3.1.4. We will test the impact of vectorization in just a bit. If we plot the same data that we saw on Figure 4.2a differently, then we can see that there seems to be some overhead. On Figure 4.3 we see the same data without a logarithmic x-axis, the y-axis is total time instead of time per element, and we are only looking at small element sizes. Here we can see how the two vector implementations have a linear trend. If this line continued towards 0 on the x-axis, then it would not intersect 0 on the y-axis. Where it hits on the y-axis is an indicator for how much overhead there is.

Larger inputs Once the input gets past the small values and we get to input sizes that are around $2 \cdot 10^3$ we see great performance per element on Figure 4.2a. To better see exactly what is going on we have a zoomed in version of the plot that can be seen



(a) Absolute performance of iterating through a list while summing the values



(b) Relative performance of iterating through a list while summing the values compared to for-each vector

Figure 4.2: Here we see how long it takes per element to iterate through integers while summing the values of them using different data structures. All data points is the *minimum* value of 10 runs. In (a) we see the absolute performance in time whereas in (b) we see it as a relative performance for doing it with a for-each loop on vector.

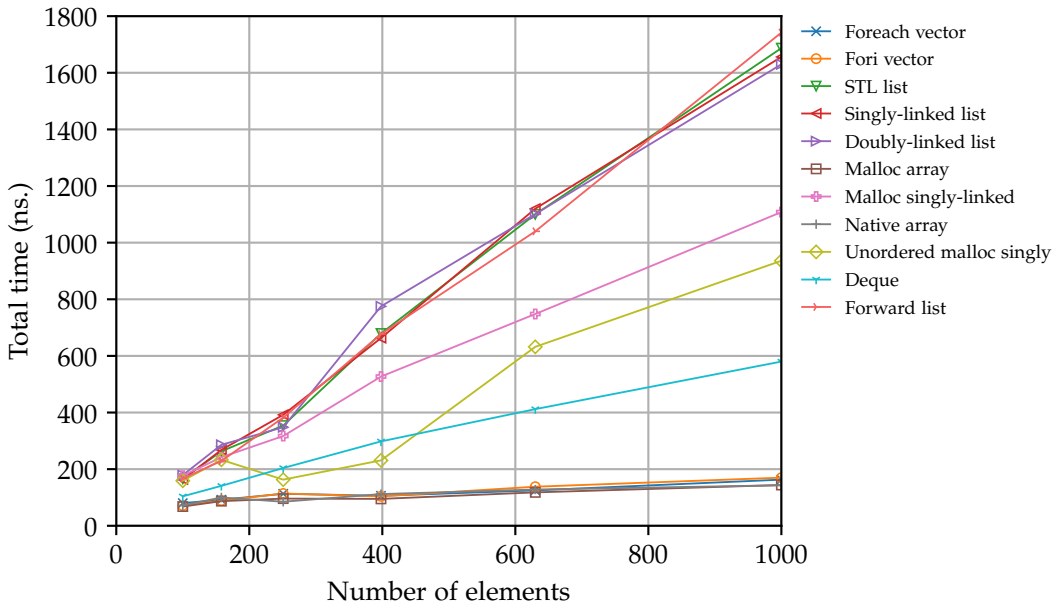


Figure 4.3: The total time it takes to iterate all the elements at a low number of elements. Unlike the other plots the x-axis includes 0

on Figure 4.4. The good performance continues all the way until the input size of $3 \cdot 10^5$. Up until this point, the performance is somewhat constant, this coincides with list theory. After $3 \cdot 10^5$ we start to see a degradation in performance. The per element time goes from around the $0.10 - 0.15$ nanoseconds range at input size 10^5 to the $0.25 - 0.30$ nanoseconds range at input size 10^7. This means that performance is more than halved. After $2 \cdot 10^6$ we see the performance plateauing again.

Cache experiment The explanation for why we see the performance starting to degrade around inputs of size 10^6 on Figure 4.4 is probably mostly due to cache. We will now prove that is indeed the case. Let us first calculate the number of integers that can at most be in the different levels of cache. For the level one cache we can hold $64\,\text{kilobytes}/4\,\text{bytes} = 16,000$ elements at most. We have to remember that other things are also going to be in the cache, so not everything can be the elements. For level two of cache we can have $64,000$ elements, and for level three we can have $1,500,000$ elements. At the level three cache we also have to remember it is shared between cores, so here there are more of a fight for the elements to get the cache space. We also note that the larger the input, the longer the time it takes before it need the elements. That makes the elements more likely to be pushed out of shared cache.

On Figure 4.4 we would expect to see a staircase effect every time we went from one level of cache to the next one. The place where we start to see the performance degrade on Figure 4.4 is around the place were we start using a lot of level three cache. It makes sense that the performance starts to degrade as we approach level three's capacity since it is shared as noted before.

We can use the *stat* mode from the profiler perf to gather stats about the number of *cache misses* and *cache references* we get running a given program. We run the list test code at a number of different inputs and note down the percentage of cache misses on

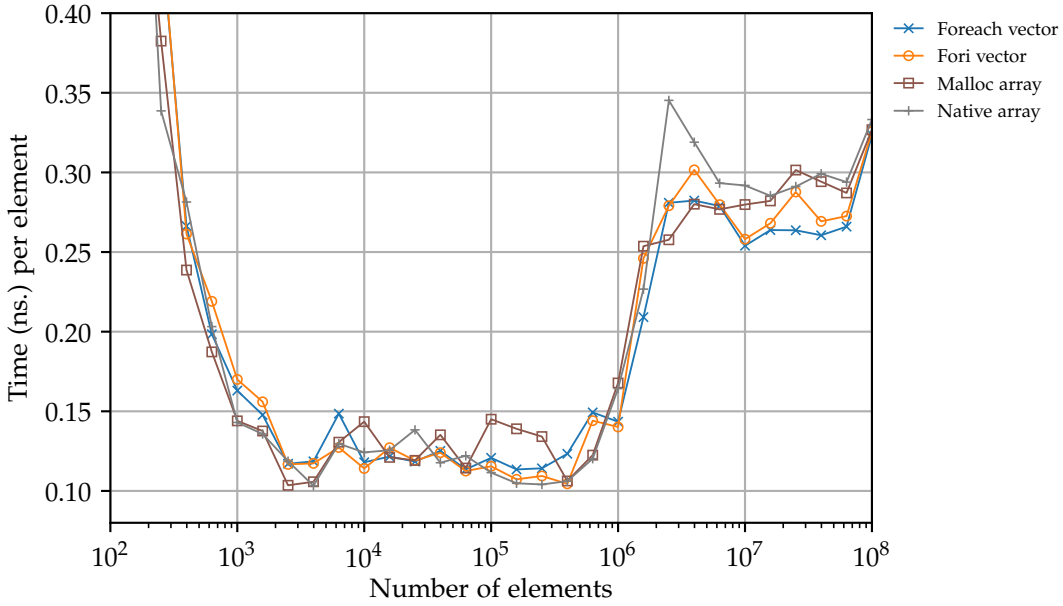


Figure 4.4: Zoomed in version of Figure 4.2a

the last level. We note that perf stat gathers stats for the whole execution, meaning that we do *not* only track the stats from iterating the loop.

In perf a cache misses is the number of memory accesses that misses the cache completely. Cache references count the number of requests made to the last level cache. The results of this test can be seen on Figure 4.5. We note that the results from perf stat will vary quite a lot between runs, so we ran it a number of times and took the number that occurred most consistently for a given input and used that. Therefore, the results on this particular graph should be taken with a grain of salt.

On Figure 4.5 we can see the results of the perf stat test. We see the percentage of cache look-ups that results in a miss. We see a sudden increase in the percentage of misses at a certain point. This point corresponds not only with a bit before the level three cache capacity, but also with where we saw the performance start to degrade on Figure 4.4. Just as we start to see a higher cache miss percentage, the per element time rise considerably. The cache misses also levels out as the performance plateaus.

On Figure 4.4 when we get to after the point where we cannot stay inside the level three cache i.e., approximately after $1,500,000$ elements, the performance levels out. At this point the cache is probably getting minimal usage no matter how many elements you add after that point.

This means we can see the effect of running out of the last level cache, but what about the lower levels of cache. We do see a small decrease in performance on Figure 4.4 after $3 \cdot 10^3$. This could be because of the first level cache running out, but it would be a bit early since we can hold a total of $16,000$ elements in the first level cache. We however also have to remember that the smaller the input the more variance we have and we are still fighting with some overhead. On Figure 4.4 there is no sign of any performance degradation where the level two cache runs out. This means that in this test it looks like it is more important to just be in cache, no matter the level. This also show in practice the difference between a last level cache miss vs. non-miss that we described to

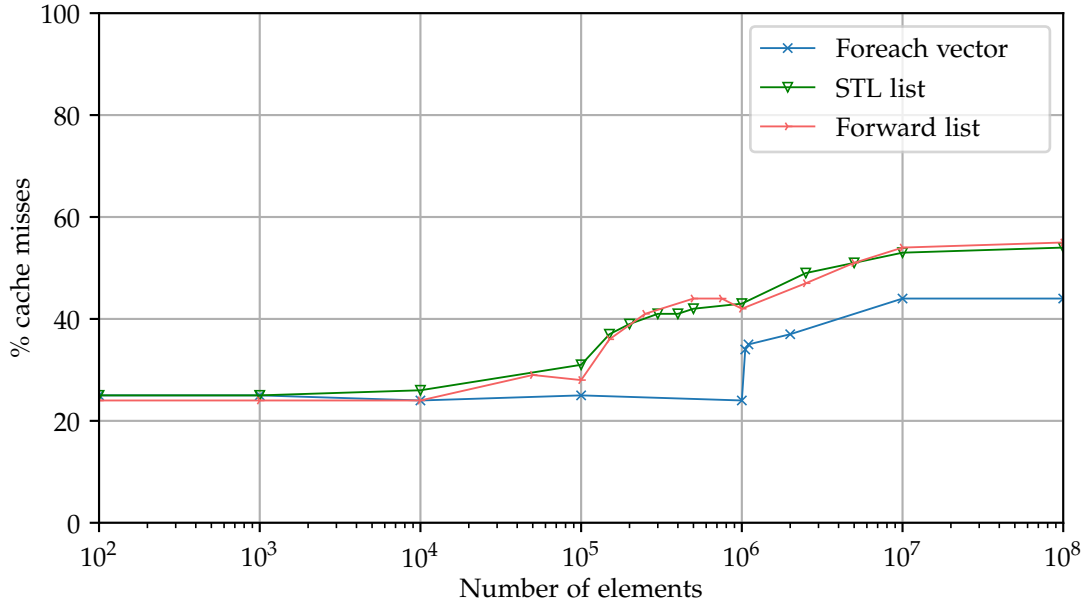


Figure 4.5: Percentage of the last level cache references that result in a cache miss.

be around 200 cycles in the section about Cachegrind in Subsection 3.3.3.

4.2.5 Results for STL list

When looking at Figure 4.2a then STL list does not seem to have any noticeable overhead in the beginning since its performance does not improve as the input size starts to increase. At first its performance is more or less constant taking around 1.5 to 2.0 nanoseconds per element all the way until input sizes of 10^5. At that point we see it slowly taking quite a big performance hit. It is probably due to the same cache properties we just described for vector.

The amount of elements that there can be at most in level three cache is 6 megabytes / 32 bytes $= 187,500$ elements. Just as with vector we see the performance degrade as we get to the limit of the last level cache. Just like we did with vector, we also performed the test on STL list where we monitor the percentage of cache misses. On Figure 4.5 we see that STL list's cache miss percentage increase in tandem with the performance on Figure 4.2a. After this point the performance on 4.2a seems to flatten out at inputs of size 10^6, but it starts to rise again. We do not exactly know why it stats to rise again. We have checked a number of things but nothing seems to clearly explain the reason. We should not be close to the RAM limit yet at that point. One possible explanation is that as we use more and more memory then we get a less favorable memory layout.

The reason why the overhead for STL list is not visible can come from a number of factors. For one, STL list already takes considerably longer to run, compared to STL list and is therefore less susceptible to a constant overhead. Furthermore, it will not have any of the overhead associated with vectorization, that might be present.

4.2.6 Testing cache usage further

We have now tested the real last level usage of the cache. The problem with using perf is that it captures the performance of the whole program. Furthermore, running the whole program on our test computer where only a handful of essential programs were running, it still gave us inconsistent results. We also explained how inconsistent the results were when making Figure 4.5. This means it might not be a good benchmark.

In this section we are going to test the cache usage in an alternative way. To do this we use Cachegrind. Like perf, Cachegrind have some limitations compared to running a real test as we mentioned in Chapter 3. Still, Cachegrind should enable us to look closer at how the cache performs in the loop where we iterate the data instead of the whole program.

Cachegrind (STL list) We first try Cachegrind out on STL list. Each time we run Cachegrind it will generate a large amount of data of which we only use a small portion. We will not be including the output of Cachegrind, but will note the values that we will be commenting on.

We start by running Cachegrind with an input size of 10^4, one of the places where it performs the best. After the Cachegrind simulation is complete, we then use the *cg_annotate* feature to inspect the result. The results will display a lot of information. Among the results is a table where the columns show various information, e.g. data reads. At each row of the table there is printed a line from our C++ test function. Hence the mode name, annotate. The information we are using is where how many cache misses and cache references our iteration loop code lines perform. On the output from annotate we see that a total of 10^4 data reads have happened on the first level cache while the last level cache had none. This shows that all the elements are inside the first level cache as expected.

We also see that $5,000$ of the data reads were misses on the first level cache. This must be because of cache lines. We learned earlier that the spacing between our elements in STL list were 32 bytes. This means that since the elements in STL list are likely packed back-to-back *in this test*, every time we get a cache miss and fetch a cache line then the next element is going to be a cache hit. The reason is that the test machine has a cache line size of 64 bytes. This all checks out with what we expected.

Increasing the input size to 10^5 we see that all the elements are still in level one cache, like before. We also see the same pattern where only every other read is a miss. This would support why we still see good performance at that input size on Figure 4.2a.

We run the same test with an input size of 10^6. At this point we see on the annotate output that with one million reads we get $500,012$ cache misses on the first level cache and the last level cache. As expected we were no longer able to hold all the elements in the last level cache.

Cachegrind (vector) Moving on to vector we expect to see the elements being packed more densely and therefore we expect less misses since every time we pull in a cache line we expect it to hold more elements. Considering an *int* is four bytes we should be able to pack a total of 16 elements into a single cache line.

We have done the tests for both the for-i and the for-each variant. Both variants shows the same results and we therefore just talk about the for-each results. At 10^4

elements the result tells us that we do get 625 first level cache misses while the last level cache is inactive. 625 cache misses $\cdot$ 16 elements in a cache line $= 10^4$ elements, so everything is exactly as we would expect.

If we look at the number of reads, we see 2500 reads on the level one cache. It is weird that we only have a total of 2500 reads, when there are 10^4 data points and we do need to read all of them. The answer, to why this is, is vectorization and we will prove that in a moment in Subsection 4.2.7. As we explained in Chapter 3, vectorization would enables us to process multiple data at the same time, in this case 4 elements in a single read. Hence only a forth of the reads.

Let us jump straight to input size of one million. Here we see mostly the same as before but with bigger numbers. On the sum statement we see a total of $250,000$ reads where $62,501$ were misses. This all happened on the level one cache. At this input size STL list were having misses in the last level cache. This shows that since data is packed more dense it can therefore fit better into the cache. We should therefore expect that vector performance starts to degrade at higher input sizes than e.g., STL list. Increasing the input size by an order of magnitude more, to ten millions, we start seeing it using the last level cache too.

This has confirmed what we saw from the perf results and showed us specifically how the cache acts in the iteration loop under simulation. We have seen that the cache lines does indeed load multiple elements into cache at the same time. Furthermore, we have confirmed that the "good" memory layout enables us to also use cache lines in the STL list implementation, resulting in only half the reads being misses. Finally, saw that something more was going on because we only read a forth of the values, and that is what we will look at now.

Branch prediction Cachegrind also provides a lot of other stats from its simulation. One of these is *branch prediction*. Every time the code has looked at an element it needs to figure out if there is more elements or not. This is a branch, and the CPU can use branch prediction here. The control flow in these iterations are very predictable and that means that for all the tests we only see it miss predict a couple of times, assumable in the start or end of the iteration. This means that the performance increase of the branch predictor should be active most of the time during these list experiments.

Even though this is true for both node based and array based solutions, array based solutions might benefit much more from this. Since array based solution supports random access, it knows where the next element is in memory before it have retrieved the next element. This is not true for the node based solution. This means that it might be able to start requesting data before the node based solution can.

4.2.7 Vectorization

From Cachegrind we learned that vector only performs a forth of the reads needed to read all the data. This suggests that something is going on, and we think that this is vectorization. Therefore we first make sure that vectorization is actually present. Level 3 optimization enables different optimizations and one of them is vectorization.

To confirm if it uses vectorization we look at the generated assembly code using GDB and the `disas` command. Looking at the assembly code it is quite surprising that a simple loop is turned into a quite large number of assembly lines. It generates many

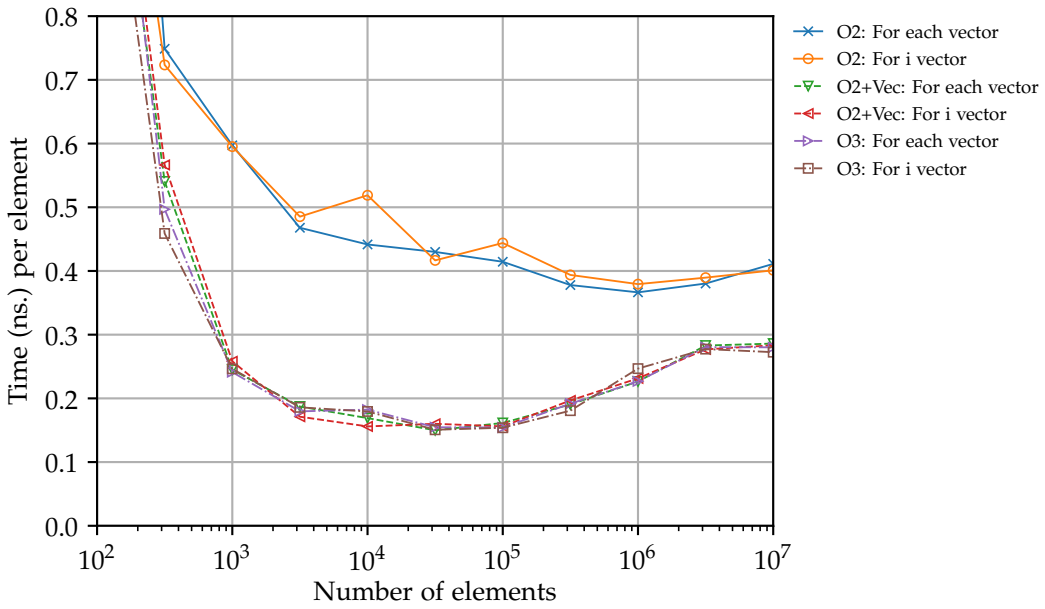


Figure 4.6: Comparing the absolute time for iterating and summing over vectors with different compiler optimization flags

lines of assembly code which are not easily understandable. This alone is not enough to hint at the use of vectorization. But we can see that the code uses special registers called `xmm0` and `xmm1`, which are vectorization registers. All this extra code is there to set up the data for vectorization, making sure that the data is aligned properly.

At this point we are quite sure the code actually uses vectorization and we move on to test how much of a difference it makes. There exist flags that prints a large amount of information about what the optimizer is doing and what choices the compiler makes. The amount of information here can be quite staggering so it is easier to just verify the presence of vectorization by just looking at timing results. Testing it also shows the actual performance difference vectorization makes.

As noted, level 3 optimization enables vectorization but also much more. So we dial the optimization level back to level 2 and use that as our base line. We also run the list test with level 2 optimization and vectorization enabled using `-ftree-vectorize`. We need the base line because going from level 3 optimization to level 2 optimization is also going to disabled other optimizations. This means that we can see the difference of vectorization alone. For the sake of completeness and curiosity we also test for level 3 optimization.

Running the experiments we get the results we see on Figure 4.6. Here it is clear to see that vectorization makes a big difference. This is interesting because this really shows another advantage of array based lists over node based lists. Across all inputs vectorization performs the best. At the really small inputs the difference is not that big, but we quickly get to an area where the vectorization more than double the performance. Interestingly, we see that the non-vectorized versions seem to improve for longer, and maybe even more surprisingly they do not seem to suffer of the same problem where the performance starts to degrade after we start using more and more of the level three cache.

This is probably because the vectorization are able to process the data so quickly that the bottleneck becomes the retrieval of data. This means that when we have to get the data from memory we are going to see a performance hit. The non-vectorized version on the other hand it is probably limited by the processing speed, and here speculative execution might also be able to better pre-fetch the next elements. Since we use level 3 as our default then we checked that our earlier discovery about spacing only being 4 bytes, was still the case when vectorization is disabled. This is indeed the case.

4.2.8 Recap of vector and STL list

From just looking at our STL list and vector sections we can see how many different parts affects the performance. Understanding each of these parts can help to estimate the final performance, but in the end a computer is a very complex system. Doing this experiment we have shown that major difference in running time can occur even if they have the same theoretical running time. This would also suggest that it is always a good idea to actually run some performance tests if performance is critical for your application.

What the test shows is that reading from an array based solution is faster than a node based solution, a result that is not that surprising. However, we did not do all of that work for just that simple conclusion, we dove deeper into understanding why we got the performance we did. This should hopefully make it clear that it can be worth to use a vector even if you might have to pay extra elsewhere, or at least places you might want to test it.

Like with all experiments we have to keep in mind that this test only shows what we are testing. This does not suggest that there is no reason for STL list to exists. This current test uses none of STL list's strong sides, that is concatenation or deletion of elements that are not in the end.

4.3 Iterating through other list types

We have now covered vector and STL list that each represent one of the two classes of lists that we defined. In this section we will cover some of the many variants from these classes, to see what affects performance. This includes other STL implementations, but also some implementations that we have implemented ourselves.

4.3.1 Forward list

Forward list is the STL implementation of a singly-linked list. This means that it is a node based implementation and closely related to STL list. We expect that maybe because forward list requires fewer pointers, we might be able to save some space, increase the density, and therefore improve the performance because more elements can be fetched in a single cache line and more elements can be in the cache.

Let us check if we actually save any space in our case. Depending on stack alignment, we might end up having the same spacing as STL list between elements. From our position printing technique we learn that there are 32 bytes between elements as seen on the addresses below. This means that they do indeed use the same spacing. This probably suggests that we will see the same performance.

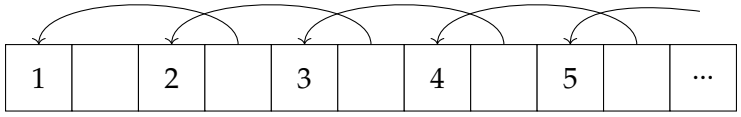


Figure 4.7: If each box is a piece of memory and you continuously add to a forward list, then you are likely to end up with memory where the elements are in reverse order.

```
...
0x55fad33d9368
0x55fad33d9348
0x55fad33d9328
0x55fad33d9308
...
```

Since it is a singly-linked list it does not support pushing elements to the back of the list, it only support pushing to the front. We need to keep this in mind since it is going to change the memory access pattern. After having added all the elements, then the structure is likely to look like Figure 4.7. This is also the reason why the addresses above is in descending order compared to the other times we have done the position printing technique.

One of the reasons why this might affect the performance is that the first elements that we read are the last elements that we added. This is the opposite of before where the last element we add is the last element we read. This might give us some performance since when all the items cannot be in the cache, we will start by going through all the elements still in cache from when we inserted them. In the other solution we would start with the elements not in cache. Starting with the elements not in the cache would then push the elements we have yet to read out of cache, only to be pulled into cache later. This should be the same effect as reading the list backwards with the other data structures.

You could make certain modifications that would enable you to push to the back of a singly-linked list, in a way that would only increase the operations with constant time. This could be done by having a pointer to the end of the list or use the list implementation defined in soft heaps simplified [13].

Results for forward list Looking at the graphs already introduced we see that forward list performs about the same as STL list. On Figure 4.2a forward list follows the trend of STL list very closely. At the point where STL list start to rise it might seem like this solution could have a slight edge. This can be because of less cache misses as explained before or just some variance. In the bigger picture the difference is not that big.

We also see the performance getting worse than STL list at around the input size of 10^6. This is the point where we no longer use the cache effectively. We have to remember that since we have added the elements from the front, we are also going through them in reverse order of where we added them. This is probably slightly negatively affecting some of the systems trying to predict what is going to happen. We also note that it means that we are reading the end of a cache line before reading the front.

Since forward list has the same trend as STL list on Figure 4.2a then we also want to see if its cache miss percentage is also similar. The minor difference in the implementation might show something interesting. On Figure 4.5 we see that it indeed does follow

the same trend as STL list. This means that iterating the list in reverse and reading from the back of the cache line seems to have minimal effect.

So in our test it looks like forward list do not bring any improvement to performance. There might be a slight improvement as the number of elements starts to fight for the last level cache, but at the same time it also seems to take a performance hit once the number of elements get large enough. We think the performance more comes down to the fact that the elements are stored in reverse order than the fact that it tries to save space.

There might exist a case where forward list could see a larger performance difference to STL list. So far all our test have used integers as payload. Changing the payload to something else, might make it so that an element in a doubly-linked list might take up an additional stack alignment jump compared to a singly-linked list. At this point we would expect a difference in performance because of the density in memory.

4.3.2 Own node based solutions

Instead of using some of the built in data structures we are also able to easily implement our own. Let us see how our own implementations perform much different.

Own singly-linked and doubly-linked lists

Making your own data structures is easy in C++. For the singly-linked list we just define a struct that holds the payload and a pointer to the next element.

```
1 struct item {
2     int i;
3     item *next;
4 };
```

And for the doubly-linked list we have two pointers instead.

```
1 struct item {
2     int i;
3     item *next;
4     item *previous;
5 };
```

Then to make the linked-list we construct multiple of these structs using `new` keyword and we do it in succession. As we have already explained this is likely to give us the favorable memory layout. As previously mentioned, if you use the `new` keyword then you also have to call `delete` on the element at some point to avoid memory leaks. We do not have a logical place to do it since our tests only focus on list creation i.e. insertion, not deletion.

Looking at the results on Figure 4.2a we see that both of them perform the same as STL list and forward list. This seems to be great since then you can use whatever one you want without losing performance. However, we would have expected the singly-linked list to perform better than the doubly-linked ones. We would expect this because we should be able to fit each element into 16 bytes after stack alignment, and therefore pack the elements more dense resulting in better use of cache and more elements pulled by each cache line.

Using `sizeof` tells us that the size is indeed 16. So that cannot be it. Next, we do the position printing test. Here we see that even though the elements only take up 16

bytes they still have 32 bytes between them. At first this puzzled us a bit. We checked the alignment of the struct using `alignof`, this would still not explain why. After a bit of research we find out that the minimum allocation you can make is 32 bytes[6]. So it makes sense that it gets the spacing that we see. This also means that there is space, and probably performance, to be saved if we malloc ourselves.

Malloc singly-linked

An alternative way to instantiate our linked-list node struct, is to `malloc` one big piece of memory and then use that. This does three things. It ensures that we can place the elements in the favorable memory layout since we control the exact position of the elements in memory. It also makes it such that we only do a single allocation instead of one for each element. And lastly, it makes it so that we can pack the elements more dense compared to what we say with our own singly-linked list implementation.

The downside to doing it this that when explicitly calling `new` *you* now have to manage the memory, and you can no longer just delete a single node once you are done with it. Furthermore, you can also run out of space having to allocate new space, which would require additional logic.

With all of this in mind let us look at the performance results. On Figure 4.2a we see some very interesting results. As soon as we get past the smallest sizes of elements then it pulls ahead of the other node based solutions. This is what we expected. Even though it performs better than the other node based solutions we still see it being almost ten times slower than vector for small to medium input sizes and a bit over 5 times slower for large input sizes as evident on Figure 4.2b.

For the larger input sizes we think this is a direct result of the total size it uses. We also see how its performance starts to degrade later than the other node based solutions, since it is more dense. Interestingly, it also seems to be much less affected when it does run out of cache.

Looking closer to at the relative performance on Figure A.5 show very interestingly that density is closely related to speed in this test. As we noted it have double the density as the other structures and we can see that it performs almost twice as good as the other node based structures at inputs of size 10^8.

Unordered malloc singly-linked

We talked about how the favorable memory layout is probably gives a performance boost, so let us test the case where the elements are not placed in the order that we are going to read them. We first allocate memory for our nodes, this gives a pointer A to the start of the block. We then interpret this allocated memory as a zeroed out array of nodes. This means that we can access the memory by $A[i]$ to get or set the i'th node. In code this would look like:

```
1 struct item {
2     int i;
3     item *next;
4 };
5
6 item *testArray = static_cast<item *>(malloc(sizeof(item) * testSize));
```

[6] https://prog21.dadgum.com/179.html

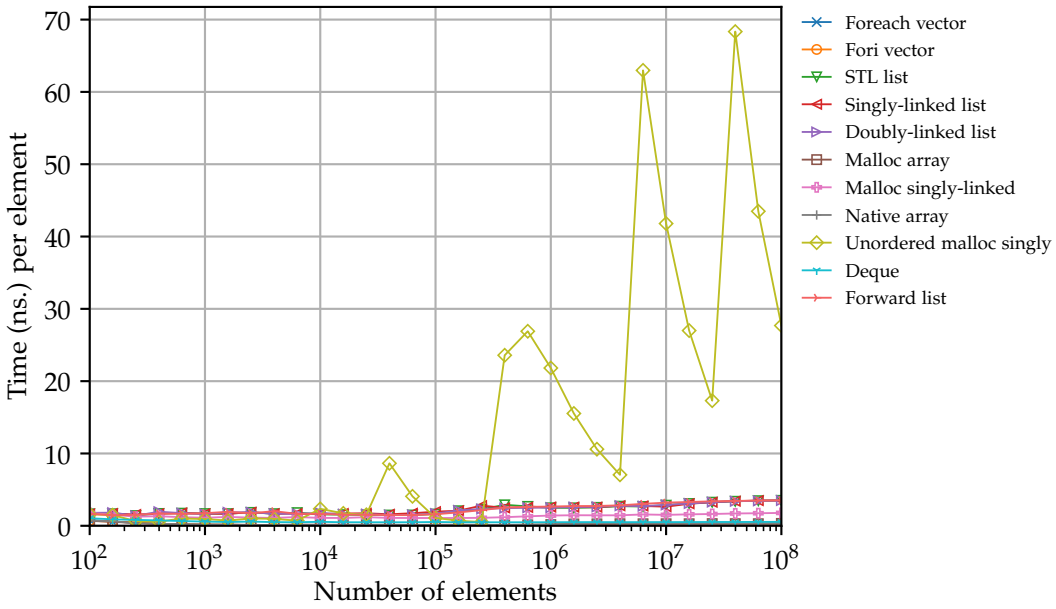


Figure 4.8: How long it takes per element to iterate all the elements in a list. Minimum result of 10 tests. Same as Figure 4.2a, but zoomed out to fit data points of the new unordered malloc singly-linked list implementation

Now that we have the memory we need to setup the nodes in random order. We get a random order by generating a new list L of unique integers $0, \ldots, n-1$ and then shuffle the list. After that we use the integers in the shuffled list to decide the order of the nodes in A. We read the value of the first integer in the list L_0. We then make $A[L_0]$'s next pointer point to the node of the second value $A[L_1]$. We also set the value of each struct to be equal to the index it is at, so we still get the same sum as the other implementations.

We can already now say that the shuffle is going to affect the performance. The tests are seeded, so it is the same test performed 10 times. We can therefore not conclude anything about how long we would expect it to perform in general, but we can see that the order can *heavily* affect the performance in a bad way. Further testing would be needed to know the full range of performance differences that can happen. Some of the downsides comes form the fact that we do not use the whole cache line when we load it. This means we might have to load a given cache line twice, or even more.

For us to see the full result we have to make a new graph since it dwarfs all the other data points. We see the results on Figure 4.8. All there really is to say is that the performance difference is massive and that the performance is very inconsistent. We see how it jumps from a very slow performance per element, to becoming multiple times faster, even as the number of elements increase.

If we made a graph that showed the relative performance we would see the exact same trends. It shows nothing new, but it is included on Figure A.2. The relative performance goes all the way up to a bit over 250 times slower than vector at some points.

There is no way around how much the unordered data affects the performance. It is clear the computer likes to read data that comes in order. This idea looks like it should

be avoided if the data is going to be unordered.

4.3.3 Own array based solutions

Just as we can make our own node based solution we can also make our own array based solutions. We will just cover static sized solutions and not dynamic solutions. Their performance when reading should not be different. We try what we call *native array*, which is doing:

```
int *testArray = new int[testSize];
```

And we also test what we call malloc array, which is where we use malloc to get a piece of memory and use that as an array:

```
int *testArray = static_cast<int *>(malloc(sizeof(int) * testSize));
```

We expect both of these to perform the same when looking at the iteration time. We also expect them to perform on par with vector since them being dynamic should not matter when just reading.

On Figure 4.2a we see both malloc array and native array follow the other array based solutions every step of the way. There is some minor difference which probably just boils down to some variance. This might be easier to see on Figure 4.2b where the relative performance is plotted. A zoomed version of it can be seen on Figure A.3.

4.3.4 Deque

Not all list implementations fit exactly into one of the two classes, node based and array based. Deque is an example of one that does not fit and is a data structure found in STL. Deque stand for Double-Ended QUEue and its implementation works like a hybrid solution of the node based and vector based classes.

Specific libraries may vary their implementations, so the following might not be completely true for all of them. In a deque the elements are being stored in chunks. The chunks work as arrays that stores elements consecutively. The chunks however are not guarantied to be store consecutively and are linked to by pointers. This means that accessing the next element by increasing the pointer by a specific amount might give you some random data and should therefore be avoided. This is not as bad as it might seem because deque does support random access in constant time, but it might be a tad slower than vector because it needs to do a couple of look-ups instead of just one.

Deque's positives include having the data packed densely inside the chunks. It also supports dynamic size, and does so without having these really bad cases where all the elements need to move that vector have. Once deque runs out of space it can allocate a new chunk and start using that. An extra positive is that the data structure supports inserting effectively in both ends.

Let us for the last time take a look at Figure 4.2a. We see how deque makes its own class between the array based solutions and the node based solutions. Intuitively, this also makes sense since it uses ideas from both. It does perform closer to the array based solution that the list solutions. The performance is actually really close to the performance we got when vectorization was disabled for vector. This makes sense since current GCC does not vectorize deque. Here vector took around 0.4 nanoseconds per element as we can see on Figure 4.6. Deque takes around 0.5 nanoseconds, so if we take

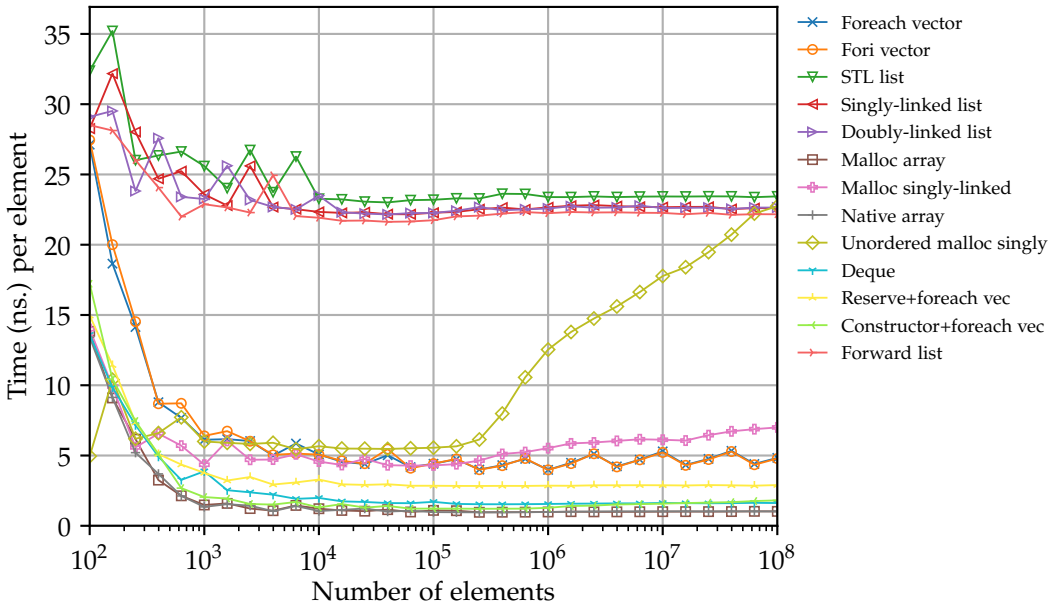


Figure 4.9: How long it takes per element to insert a given number of element in different list implementations. This is the minimum result of 10 tests

that it cannot use vectorization, and the fact that we need to do a little extra work for each look-up, then it makes sense.

We also see how both of them are not affected that much by running out of memory, giving us a very consistent performance across all input ranges. This is probably because the bottleneck is processing the data and not fetching the data. This also helps it to bring it relatively close to the performance of vector once vector runs out of last level cache, as can be seen on Figure A.3.

4.4 Performance of creating lists

It makes sense that when you have a list, you also want to be able to insert elements into it. To test insertion we have timed how long it takes to insert a given number of elements into all the implementations discussed in the previous test. We have also added two additional variants that relate to vector.

Theory told us we can insert into static sized array based solutions and node based solutions in constant time. For dynamic arrays we needed an amortized analysis to tell us that the performance is amortized constant per element. Alternatively, we could also define the size we want from the start.

4.4.1 Results

We have the results of these tests on Figure 4.9. This will be used for all the rest of the observations in this section. Like with the iterating test, we look at the per element time it takes to insert. The data points you see are the minimum of 10 executions. Compared to the iterating test the graph looks more like what we would expect from theory. Most of the lines looks pretty constant when we reach a certain point.

Non-malloc node implementation If we start by looking at the non-malloc node implementation we see that they perform about the same across all input sizes. They were also the ones that mostly performed the same in the iteration test. There also seems to be a small trend for the STL list to perform worse. STL list is setting more pointers than the forward list, so it makes sense that it is a bit slower when adding elements. The two implementations we have defined ourselves seem to perform right between the two implementations, and seem to not care about whether it is singly-linked or double-linked. Overall the difference is not that big compared to some of the differences we see for the other implementations.

Malloc implementation If we look at the test called "malloc singly-linked" where we allocate a big piece of memory and use that to store a singly-linked list, then we see a big improvement in performance. Here the elements are stored in memory in the order we will access them. We already discussed how it has an advantage because it is more dense compared to the other list implementations. It also has the advantage that we only allocate once compared to one for every single element which the node based structures we just talked about do. We see it starts to loss a bit of performance once we start to use more and more of the level three cache, but still performs much better than the node implementations we just mentioned.

Like with malloc singly-linked list we allocate our memory in a single call. However this time, to create the unordered list we randomly access the memory to place the elements. Theory would tell us this should not take extra time, but in practice it does. Looking at the performance we see that it performs close to the ordered version at low inputs, a bit slower. This is where the elements are still in the cache, and therefore the random writes are not yet affecting the running time much. We see the performance degrade once we run out of cache, as this greatly affect random writes more compared to all the other solutions. This means that once the cache is full it needs to load cache lines into cache multiple times. Since all the others write the elements in the order they appear in memory they never have to do this.

Vector Next let us take a look at the two vector implementations that we call for-each vector and for-i vector. These perform very well compared to the list implementations, and also slightly better than the ordered malloc implementation. We do see some consistent ups and downs. This is probably because the dynamic array needs to move all the elements when it is full, and it raises the average time a bit when this happens.

In we beginning the noted how we had two additional variants for vector. These variations both try to specify the size the array needs to be in order to save on the number of time the array needs to resize. The first way to do this is using the `reserve` function to reserve space for the number of elements we need from the start. This is called "reverse+foreach vec" on Figure 4.9. We can also see that this is some easy performance to gain if you know the array is going to be some size, and the good thing is that you can still expand again later.

We can also specify directly in the constructor how big the vector needs to be. This also gives quite the performance increase, even compared to reserve. A downside to this is that it actually adds that number of elements to the vector, meaning you now have to overwrite instead of push back. This can make it harder to use because you now need to track how much you use yourself.

It is interesting to see the difference that specifying the size in the constructor have compared to reserve. This was one of the things we expected to perform about that same as using reverse. We would also have expected both of these to be closer to the native array. It seems like quite a big change in performance if it is due to checking if it needs to expand on every insert.

Deque Next up is deque. Deque performs quite a bit better than vector and vector with reserve. We have to remember that this is without having to define a size. At smaller inputs it is beaten when we use the constructor to define a size for vector, but as the input size increase it seems like that solution's performance degrade a bit while deque stays pretty much constant.

Native array The last implementations we need to talk about are also the ones that perform the very best. The implementations are the native array which is when we use new on an array and the case where we use malloc to allocate memory for an array. Both of these solutions are super fast. We expected them to perform the best, because they are the most native list we have and they only do a single malloc.

4.4.2 Perspective

Once again we see the array based solutions dwarf the node based solutions. We have to remember that it mostly has to do with us testing the array based solution's strong cases. Once again we see that even though different implementations have constant time per element, then the performance difference can be immense. Once again it looks like vector should be your go-to implementation unless you are doing huge amount of the cases that node based solutions excel at.

An easy way to give a little perspective on the scale between inserting and iterating is to plot the total time it takes to insert and iterate once. The results of this can be seen on Figure 4.10, or the zoomed out version Figure A.4. The dominant part is mostly the time it takes to insert elements, this is also what we see on the figure.

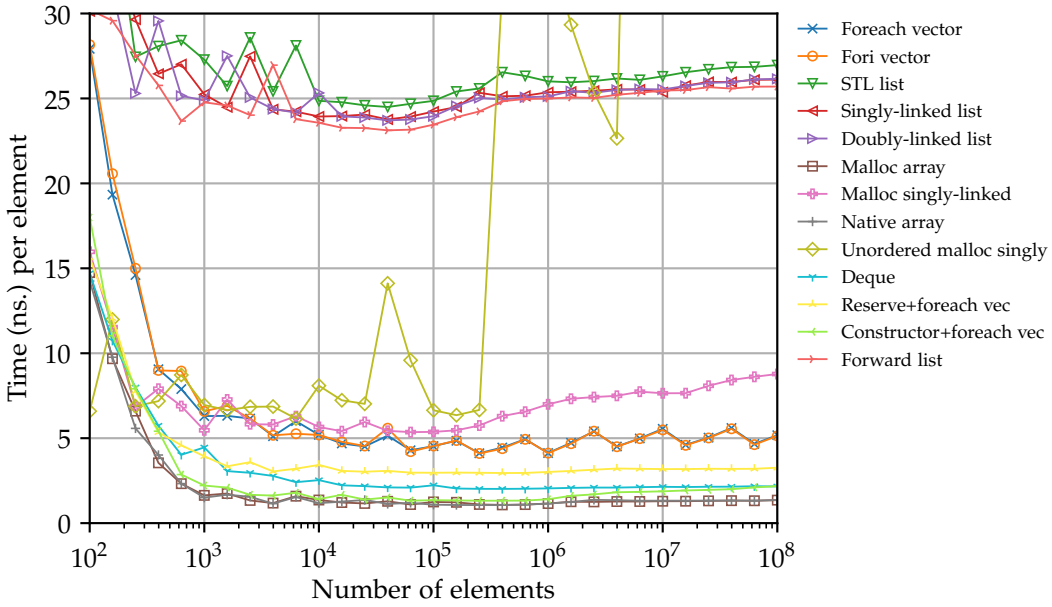


Figure 4.10: How long it takes per element to insert and then iterate a given number of element in different list implementations. This is the minimum result of 10 tests

5 Soft heap implementations

Here we will talk about the specific implementations of soft heaps we have made. All of the implementations are made in C++ and based on the papers we have read. For each of the implementation we will talk about how we translated the theoretical constructions into code, some of the choices we made and some of the problems we experienced.

To make it easy to switch between the implementation and make it fair we want the two soft heaps to have the operations with the same method signatures.

Both implementations also supports any type of items using C++'s templates to implement a generic type. Just like you would initialize a list in C++ to hold a specific item, you initialize a soft heap to hold a specific type:

```
1 std::vector<int> exampleVector;
2 std::SoftHeap<int> exampleSoftHeap{threshold};
```

We will first cover the soft heap simplified, where we adapted the pseudocode implementation provide by Kaplan et al. [13] while also referring to the Python code he links to in the paper. This is followed by the soft sequence heap implementation, where we did a did more work because we discovered the performance was not inline with what we expected. For each of the soft heap implementation we will also do some isolated tests to test and improve the initial solution before moving on to later tests in the next chapters.

5.1 Soft Heaps Simplified

In this section we will talk about how we made the implementation for the simplified soft heap. The theory for the simplified soft heap can be found in Section 2.8 and in [13] where the algorithm was introduced. The paper contains a verbal description of the operations and pseudocode. In addition to the pseudocode they also provide a link to his own implementation. This link however is unfortunately dead. We managed to find an archive site that had a snapshot from when the website was still alive.[1] Looking

[1] https://web.archive.org/web/20180221053506/https://www.cs.tau.ac.il/~zwick/softheap.txt/

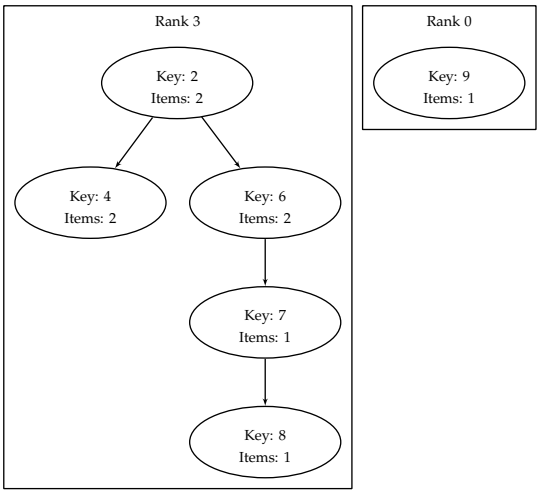


Figure 5.1: Example of a simplified soft heap with 9 items

at the code we can see that it is written in Python and follows the pseudocode very closely. The implementation of [13] is based around nodes with pointers. Translating this code into C++ does require some tweaks to make it work, which we will explain in this section. We also implemented such that you can output a given graph. An example can be seen on Figure 5.1

5.1.1 Interface

We want to match our interface where we can create a new soft heap with a given threshold, insert elements, delete the minimum element, find the minimum element, and meld two heaps together. Each of these operations then needs to be member functions to a class that represents the simplified soft heap. Doing it like this would let you do natural things like `heap.insert(i)` if `heap` were an instance of simplified soft heap. The method signatures for each of these can be seen here:

```
SimplifiedSoftHeap(float threshold);
void insert(T *e, kType key);
void deleteMin();
T *findMin();
T *extractMin(); // Does find and delete min in one.
void meld(SimplifiedSoftHeap<T> &H2);
```

Already now, we have a problem compared to the solution that Kaplan have made. All his implementations are static, meaning that they are not defined as member methods. This means you have to provide all the arguments every time, and take the return value every time. This works just fine, but does not fit the interface and does not utilize member functions which helps not just having every function defined in global scope. Here is an example of the difference in calling it makes:

```
H = insert(e, H)     // Kaplins et al.
H.insert(e)          // Ours
```

Here their H is a soft heap, which in his case means it is a pointer to the root of the root list. On the other hand, our H is an instance of a class. This class then have a member variable that have a pointer to the root of the root list named `rootroot`. To stay as true to the implementation as possible we define all of Kaplan et al.'s function

as private functions inside our `SimplifiedSoftHeap` class, here we can also make them static. Then we wrap his functions using our interface and store `rootroot` as a member variable. An example of how we wrap insert can be seen here.

```
template<class T>
class SimplifiedSoftHeap {
 private Node *rootroot;              // The first node in root list
 ...
 void insert(T *e, kType key) {       // Our exposed insert function
  // Insert is not called recursively
  // Therefore we can inline the code from Kaplan et al.
  // But we can see how we pass rootroot to the function and save it again
  // This never exposes the user to the root list
  rootroot = keySwap(meldableInsert(makeRoot(e, key), rankSwap(rootroot)));
 }
 ...
}
```

We do the same thing for all the other functions. This might decrease the performance slightly, but overall you should never be able to notice it. So many more things affect performance much more. The optimizer might also be able to inline this operation. Hiding the root list inside also makes it such that you cannot access the element outside the interface, which is what we want. It also makes it such that we can associate more values with a given soft heap, like e.g., the threshold, such that we are not forced to share this across all implementations.

5.1.2 General structure

One of the first things you have to decide when making a structure is how the different parts are going to be connected to each other and how the data is stored. In C++ we need to define these structs at run time and cannot change them later. In the Python implementation he does not need to do this. We also wanted our structure to be able to be used for any type of element, this means we turned it into a template class, as can be seen from the top line of the previous code snippet. This means that when you see the type `T` then it is the type we are holding.

The elements set The elements of type `T` that we are storing is stored inside objects of type `Item`. The reason for this is that `Item` is a node in a singly-linked list. This way of making a linked list is defined in [13] and is a quite simple way to make a list that supports all the operations we need in $\mathcal{O}(1)$ time. This includes the operations for concatenation, pop front, and push back. In the Python code Kaplan could add the next pointer directly to the object of type T. This however is not possible here, so we have to make this extra step. The `Item` struct is defined as follows:

```
struct Item {
    T *e;
    Item *next = this;
};
```

To make a list out of this we use the next pointer to point to the next element in the list. The list is cyclic, so the last element will have the first element in its next pointer. If there only is one element in the list then the next pointer points to itself. Now, if you accessed the list by a pointer to the first element then you would have a problem doing

pop front and push back in constant time. Instead we have a pointer to the last element. This might seem a bit weird, but as we will explain now, this will enable us to do all the operations we mentioned in $\mathcal{O}(1)$ time.

We are able to access the first element in constant time since the list is cyclic. To do this we just use the next value of the last element to get the first element in one step. Removing an element is easy since we just update the next pointer to have the value of the next node's next pointer, i.e., to delete the node after x we do `x.next = x.next.next` assuming x is not the only element in the list. Since we are in C++ we also have to remember to deallocate the item we just removed from the list if we are done with it. This means we can pop front in constant time. Extra code is needed to check if it is the last element.

Pushing to the back is also easy. Here we to make the last element point to the new item and the new item point to first item. Concatenation works much the same way. Here each list's first element will point to the other's last element, and the last element of each list will point to the first element. This way you have made them cyclic and you have a combined list.

So now we know how we store the elements in a list. These lists are then used to hold a group of items that have the same current key. This means that if there is more than one item in the list, then those items are corrupt.

Binary tree nodes The other structure that we use is called a `Node`. These are what the binary heap heaps in the simplified soft heaps are made of. Each node holds a value of `kType`, which stand for key type. This is going to be the representative for all the items found in the set. We also see here each node have a rank. We also find the left and right pointer we would expect since every node can have children. We also find a next pointer, this pointer is only used by notes in the root list, and defines the next element in the root list.

```
1  struct Node {
2      Item *set;
3      kType key;
4      int rank;
5      Node *left;
6      Node *right;
7      Node *next;
8
9      bool operator<(const Node &y) {
10         return this->key < y.key;
11     }
12 };
```

From the snippet we also see that we need these to be fully ordered. This is because it is these are going to be used to decide the order of the elements inside the heap. This is the only two structs we have in this implementation. Alternatively you could have two versions of node such that you would not have to store the next pointer for all the elements that are not root.

Null object

One of Kaplan's ideas in the original code is to create a null object that can be used instead of having null pointers. This null object would then be used in `Node`. The null

object would have the highest key possible and have all the pointers pointing to itself. Kaplan does this in his Python implementation. The motivation for doing this is that we can save a lot of null pointer checking in the code, since we would not need to check is everything is a null pointer before reading it. Instead we just need to check in the end if the are on the null object. The null object idea is good, but there is some problems.

The variable he creates in his solution is global, which is never something you would see when making e.g., an array. Furthermore, this would not even work for us since we use template class. So instead of making a global null object we made one for each of the soft heaps we created. This worked fine until you meld two different soft heaps together. Each of the melded structure have their own version of null object and these are not equal. This means that after the meld one structure have multiple different melds. This makes it such that the code can no longer check if it the correct null object. To solve this problem we sad to make the null object shared between instances of the soft heaps if they had the same template class. The solution we ended up with is this:

```
Node *getNullObject() {
    static Node *result = nullptr;

    if (result == nullptr) {
        result = new Node;
        result->set = nullptr;
        result->key = std::numeric_limits<kType>::max();
        result->rank = std::numeric_limits<kType>::max();
        result->left = result;
        result->right = result;
        result->next = result;
    }

    return result;
}
```

This function is then called in the constructor of the soft heap. This will create a static null object if it does not exist, that is shared across all instances of soft heap, that also have the same template class. If it exists, then it does not need to create a new one. This will make meld work.

One of the problems with this solution is that we never clean the null object up. However, we only create one single object for each type of soft heap you use. In most cases this would mean you only create a few null objects, which should not be a problem.

Memory management

In the original code he had pointed out that you needed to deallocate two places in the algorithm. The place is if empty a node in `delete-min` and have no children. The other place is in `fill` in the case where the node you have been filled from have no children. Our implementation requires additional deallocation because we have the list made of `Items` that hold the elements. The deallocation needs to take place inside `delete-min` too.

There are also other things with memory management you have to watch out for. We have to implement our own destructor, since a user might not extract all the element from the heap. This could leave memory leaks if we do not do something. An easy solution would be to all `delete-min` until no elements were left, but this would not be

very efficient. We ended up writing some code that would go through the structure and delete all the data efficiently.

The `meld` operation can also cause some problems with memory management. If we have two heaps that we have melded into one, then we have to make sure that once one goes out of scope then that does not ruin the other. We therefore have to not just move elements from one to the other, but also clean the other one up.

Lazy insert

There existed two versions of the simplified soft heap. The lazy insert version and the non-lazy insert version. The first one we implemented were the original, non-lazy one. There exists quite a number of ways to make it lazy here we explain the implementation we did.

To make this work we need to make some modifications. When an element is inserted into the soft heap, then we add it to a buffer instead of adding it directly to heap structure. The buffer is a simple STL list that just holds the element. We make sure the minimum element of the buffer is always in front, the order of all the other elements does not matter. The reason we do this is that we need `find-min` to return the correct value, and this needs to be the same value that `delete-min` will delete. We modify `find-min` such that it will find the minimum element on the heap and also check the first element in the list. It will return which ever is smaller. `delete-min` will also check if the first element is the lower than the lowest heap element. If this is the case then we remove it from the buffer. Next we add all the elements in the buffer to the heap. While doing this we need to track which elements get corrupted. Now, if the heap had the minimum element then we delete the minimum element from the heap. Lastly, we return a list of elements that have been corrupted.

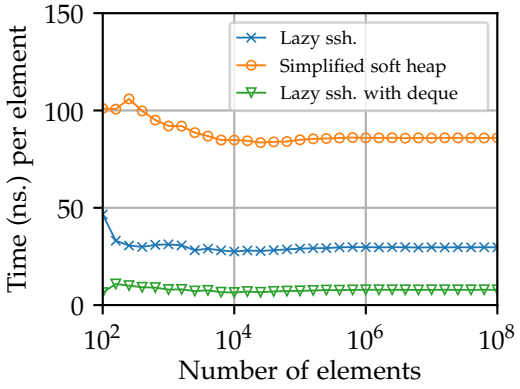
Down sides to this is that we do add some overhead, but we do not change the amortized time. It should also be clear why we can have a `delete-min` operation take a very long time. Before all of this time were spread out across many `insert` operations. Alternatively, we could insert right away and keep a list of corrupted items.

Performance of variants

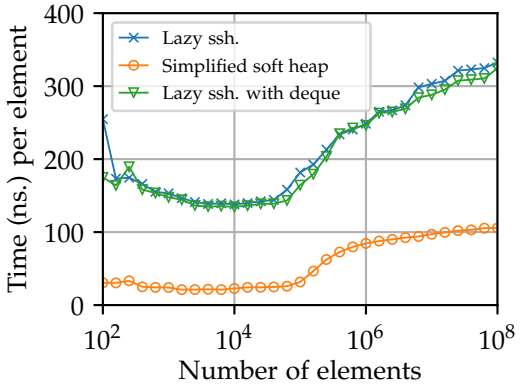
To check that the implementation we do some quick tests to see that they work. From this we can also see how bad the overhead is from lazy inserts. On Figure 5.2 we see the results on the blue and yellow series. We can see that the lazy implementation do add some overhead. We can try and apply some of the things we learned in experimenting with lists chapter. Here we learned that deque could have some advantages over vector, and we see that this indeed is the case. That is some cheap performance gain.

5.2 Soft Sequence Heaps

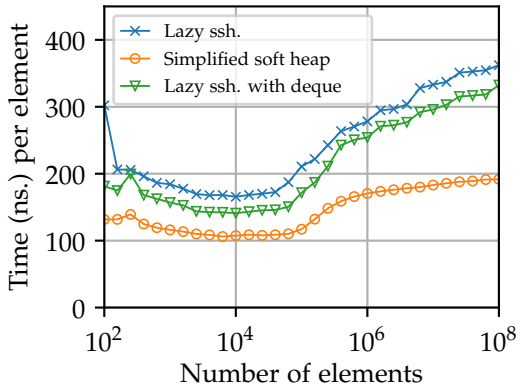
Before we implemented soft sequence heaps we implemented sequence heaps. We will not talk about that implementation since the code is just a simpler version of the soft sequence heaps code. Implementing soft sequence heaps has been a process where we have gone through different versions and different iterations of those versions. In this section we will mostly focus on the overall versions which we ended up having two of. The thing that sets the two overall versions apart is the list implementation they



(a) Time it takes to insert into the heap.



(b) Time it takes to extract from the heap.



(c) Time it takes to insert and extract all the elements.

Figure 5.2: Showing the performance of different instances of a simplified soft heap.

use for the outer and inner lists. The outer lists are the ones that are used to represent sequences. Each item in a sequence has a corruption- and witness-set which is what we call inner lists. The motivation for trying different list implementations was to see if it made a difference and if we could pinpoint what list implementation would give the fastest performing soft sequence heap.

In the paper that introduced soft sequence heaps, the functions are defined statically, e.g. instead of INSERT being called on an soft sequence heap object then it is given as an argument to INSERT. We elected to make a C++ class instead and thereby do it the object-oriented way. Performance-wise it does not make a difference so it is a matter of taste.

5.2.1 Outer and inner list as STL list

To begin with we chose the C++ STL container called list which is a doubly-linked list. We did that because it has a lot of utility out of the box and we therefore thought it would be useful for getting a quick initial implementation. For our first implementation we wanted to follow as much as possible the paper that introduced the soft sequence heap. Brodal [3, p. 5] specifies that a linked list should be used to represent the inner lists. This also aligned well with starting with STL list *for both the outer and inner lists*.

We hypothesized that STL list would result in a heap implementation that is slower than using e.g., one's own doubly-linked list implementation because we expect that the memory management it does behind the scenes will impact performance as the input size grows.

We will now go through the two different ways we implemented heaps using STL list as the outer lists.

STL list iterator and hollow entries For our first implementation attempt we stored the rank objects[2] in a field variable `sequences` that was a STL list. Inside each rank object was a sequence struct containing a STL list called `sequence` and a pointer to a struct of the same type as this struct. Inside `sequence` were item structs that each contained key, value, corruption-set and witness-set.

The introducing paper left some implementation decisions unmentioned and one of them was how to maintain the different rank objects. In `insert` when the sequences of two rank objects are merged then we decided to just move one of the sequences to the other rank object while merging. Thereby leaving a "hollow" rank object in `sequences`. The purpose of this was to avoid using time on deleting and re-creating rank objects from `sequences`.

Since `sequences` was an STL list then we used C++ list iterators which is a pointer that is specialized in the sense that it only point to list entries. After many inserts a heap would contain a lot of hollow rank objects. To avoid iterating over them we needed pointers inside the STL list to be able to skip the hollow rank objects. These list iterators would need to be stored dynamically such that they would persist between calls to `insert`. Storing STL list iterators dynamically is not possible so we decided to abandon the idea of having a STL list field `sequences` that used list iterators. Instead we came up with the following approach.

Own doubly-linked list to store rank objects Our alternative to storing the rank objects in a STL list and using iterators was to use our own doubly-linked list implementation and simply have a pointer to its head and tail. Our motivation for this is also related to our hypothesis about STL list maybe being inefficient. More on that later. Our doubly-linked list is used by having a `dllNode` struct for each rank object. The `dllNode` struct can be seen on Listing 5.3. The way we represent an item can be seen on Listing 5.1 and Listing 5.2. We use two different item representations because most of the time it is convenient to store the item information which is a key and a value, together with the inner lists that are associated with that item. But the items inside the inner lists should not have their own inner lists so therefore we use the simpler representation here.

[2]On Figure 2.9 one can see that all the information that constitutes the outermost object of a soft sequence heap is bound to a rank. We call this outermost object the rank object and it is represented as a C++ struct.

```
struct itemCW {
    kType key{};
    T *value;
    std::list <item> cSet;
    std::list <item> wSet;
};
```

Listing 5.1: Item representation used for everything except what is mentioned in Listing 5.2.

```
struct item {
    kType key;
    T *value;
};
```

Listing 5.2: Item representation used for inner lists and when returning items.

```
struct dllNode {
    dllNode *prev;
    dllNode *next;
    int rank = 0;
    std::list <itemCW> sequence;
    dllNode *suffixMin;
};
```

Listing 5.3: Our final doubly-linked list implementation.

We decided to abandon the idea of having hollow rank objects because to make it efficient then we would need to maintain pointers to navigate past the stretch of hollow slots. This would increase the code complexity somewhat, so we elected to just remove and re(create) `dllNodes`. More specifically, when we merge two sequences then we do it by moving one of the sequences into the other, leaving one of them empty. The `dllNode` with an empty sequence, we remove. We (re)create a `dllNode` if the sequence produced inside `insert` has a different rank then the next existing sequence.

Since we had chosen to use our own doubly-linked list implementation instead of STL list then we needed to do our own memory management. This means calling the C++ functions `new` and `delete` explicitly. Doing this correctly was easy since there are few places in the code were an `dllNode` is either created or removed. As previously mentioned, our motivation for using our own doubly-linked list was that we expected that it would be at least as efficient if not more efficient then a STL list because structs are one of the simplest concepts in C++ and we control every memory call made. So we control what calls are being made on a very low level since `new` and `delete` are basically just wrappers for `malloc` and `free` calls. Memory management is an important, but tedious consideration that is left out of the original soft sequence heap paper [3] so we find that it is an angle worth investigating.

After having finished our first working soft sequence heap implementation, we plotted it `insert` time which can be seen on Figure B.2. We also know from soft sequence heap theory that `insert` takes amortized $\mathcal{O}(\lg \frac{1}{\varepsilon})$ amortized time so the series should not be growing, it should converge towards a constant. We therefore concluded that our implementation was flawed.

We discovered two separate types of inefficiencies in our code so we ended up making three iterations of the full STL list version[3] to be able to verify that changing the coded indeed made a noticeable time reduction. The three iterations are: original, reduce by reference, and dllNode.sequence as pointer. We will now explain the two improvements and how we found them. We were unsure of what part of the code was causing the slow `insert` time so we used the *record* mode of perf on our test that does

[3]Full STL list version means that is uses STL list for both the outer and inner lists

insertions and extractions and we got the flamegraph on Figure 5.4[4]. We noticed that the block which name ended with "::list" was the amount of time perf had seen the constructor call for STL list on the CPU. So we were creating STL lists a lot and this lead us to the inefficient code which we talk about in the next paragraph.

Reduce by reference The first inefficiency we spotted was that the argument to `reduce` was passed by value instead of reference. This could potentially be expensive if the argument required much space since a copy is created. This also allowed us to avoid making a couple of copy-assignment and copy-constructor calls. On Figure 5.3a we saw that there indeed is a noticeable time reduction between the first two series. The reason for this is that we often pass large sequences to `reduce` and they are represented as STL lists. Copying large lists is expensive, while just passing a reference is fast. We see that on the new flamegraph on Figure B.3 the STL list constructor block has completely disappeared. This does not mean we do not call the STL list constructor, but just that it was not on the CPU stack during one of perf's samples.

For the `extract-min` time we observe that reduce by reference outperforms the original iterations which is a bit odd since extract-min does not call `reduce`.

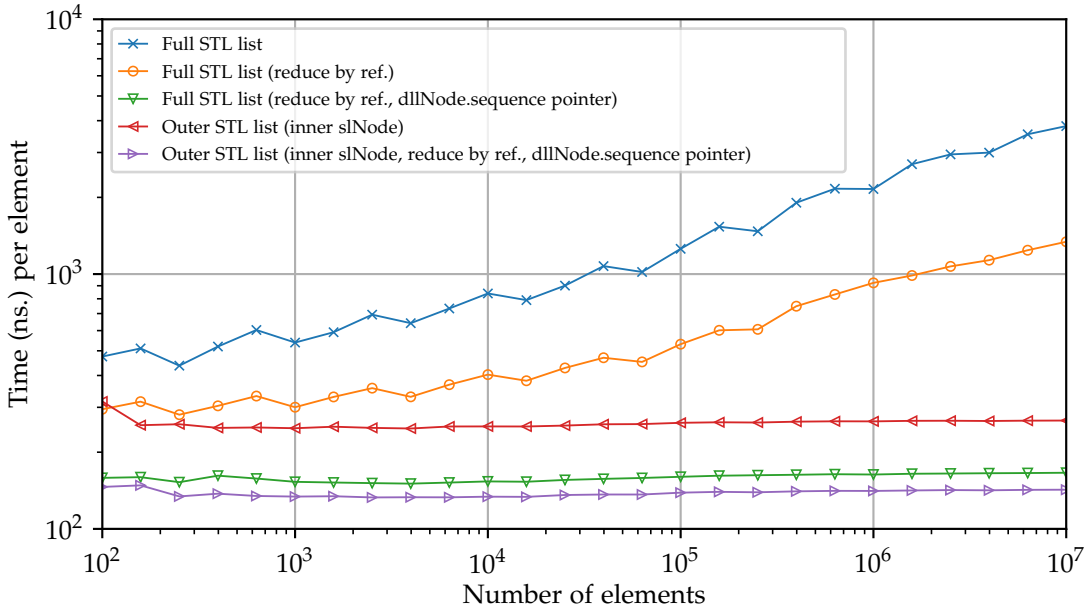
This improved iteration is still growing on Figure 5.3a so we again use perf record. On Figure B.3 we see that the block which name ends with "operator=" is the widest block. It is the function that is called when a STL list is assigned to a new variable which is the same as copying an STL list. With this information we believed that making `dllNode.sequence` to a pointer would avoid these copy-assignment calls which is the next thing we will talk about.

dllNode.sequence pointer On Listing 5.3 we see that `sequence` is now a pointer. The new flamegraph on Figure B.4 shows that making `dllNode.sequence` into a pointer has reduced the amount of "operator=" for STL list that perf now does not pick up on it in its samples. On Figure 5.3a we see that this new iteration is no longer growing and instead is converging towards a constant which lines up with the theoretical time complexity of soft sequence heap insert. It makes sense that turning `dllNode.sequence` into a pointer has made a big performance increase because the `dllNodes` need to be dynamically allocated and allocating a pointer is much faster than allocating an entire list. The cost of this is that `extract-min` becomes slower as seen on Figure 5.3b. The growth seems to accelerate near the end, but we are only talking about 250 nanoseconds so compared to insert then `extract-min` is still fast.

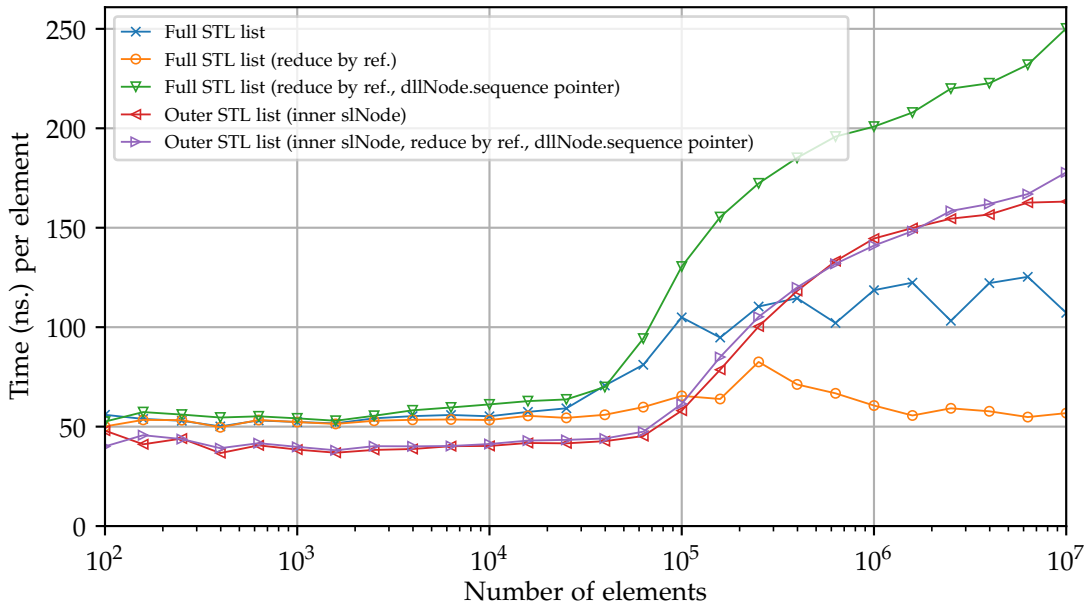
5.2.2 Outer STL list and inner linked list implemented ourselves

In the quest of finding the fastest performing soft sequence heap implementation we thought it would be a good idea to use our own implementation of a linked list instead of STL list. We liked the singly-linked list used in [13] so we implemented it as a struct called `slNode` which can be seen on Listing 5.4. We hypothesized that the largest performance gain would be gotten by changing the inner lists to `slNode`. So that is what we will talk about next.

[4]It is difficult to read the PDF version of the flamegraphs in the report. Interactable versions are available om `https://noobworld.dk/thesis/flameGraphs/`.



(a) `insert` time per element with logarithmic axes.



(b) `extract-min` time per element with logarithmic x-axis.

Figure 5.3: `insert` and `extract-min` time per element for all our soft sequence versions. The soft sequence heap is instantiated with threshold 0. The legends are ordered chronologically such that the topmost legend is our first heap implementation. Ext. STL list means that the outer list is an STL list. A plot combining the `insert` and `extract-min` time can be seen on Figure B.1.

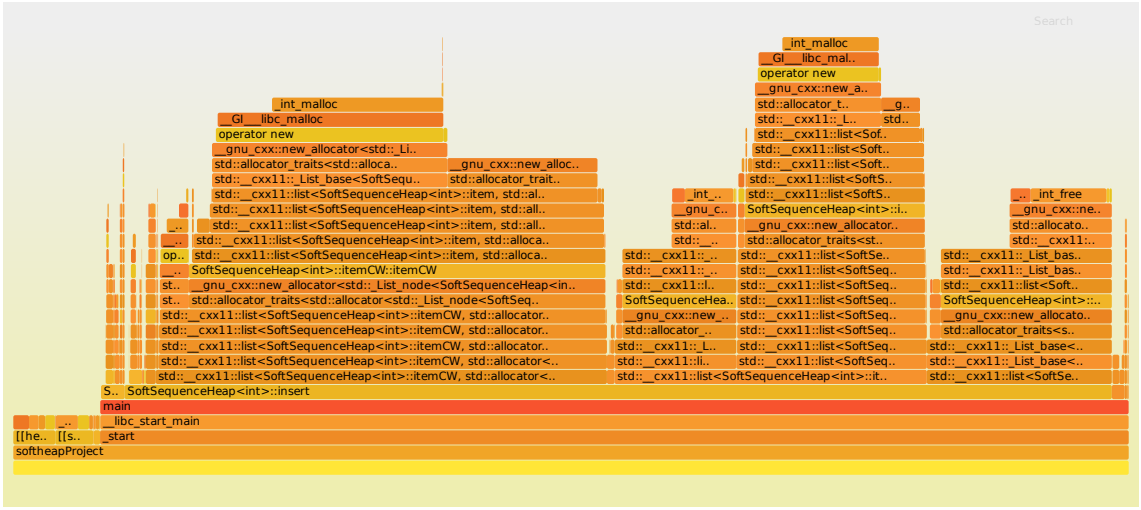


Figure 5.4: Flamegraph for full STL list version without improvements.

```
struct slNode {
    item slNodeItem;
    slNode *next = this;
};
```

Listing 5.4: Singly-linked

Inner slNode We had to change the code that used STL list functions to make `slNode` work as inner lists. For example to make `reduce` we had to implement our own splice function. Looking at Figure B.4 and Figure B.5 we observe that the chain of calls that `extract-min` does completely disappears. This can also be observed on Figure 5.3b since the inner `slNode` version beats the full STL list version. This inner `slNode` version does not beat the previous version in Figure 5.3a, but we have not yet applied the inefficiency fixes which we will do now.

Inner slNode with fixes We now apply the same inefficiency fixes that we talked about before to the inner `slNode` version. No big changes occurs from Figure B.5 to Figure B.6, but nonetheless a performance improvement is visible in Figure 5.3a and it is the best performing implementation so far. As the flamegraphs reveal then this is attributed by the fewer calls that our own implementation makes in contrast to the calls that STL list makes.

6 Experimenting with soft heaps

After having implemented the structures the next step is to do some experiments with them. The experiments in this chapter will focus on soft heaps i.e., they only use the basic soft heap operations of the different soft heap implementations. After this chapter we do soft heap experiments with focus on a given soft heap application. More on that in Chapter 7-8. We are also going to examine the real world *number of corruption*s to see if we are close to the corruption limit. Performing these experiments that only focuses on the soft heap implementations might give some knowledge that will improve one's understanding of soft heaps.

6.1 Number of corruptions

So far we have talked about the corruption limit in theory, but now that we have the implementations we should quite easily be able to see how the number of corruptions changes over time in practice. To do this we create functions that can go in and traverse the soft heaps and figure out how many corruptions there are in a given structure. We can then call this function multiple times to see how the number of corruptions evolve over time. For the first test we chose $\varepsilon = \frac{1}{2}$. This means that the threshold is 3 for the simplified implementation and 1 for the sequence implementation. In the test we will insert 10^4 items and then extract them again. While doing this we will be calling our new corruption count function between each insert and extract.

We have the results of this test on Figure 6.1. Here we can see how adding the item slowly corrupts the simplified soft heap, whereas the lazy version and soft sequence heap only get real corruptions after we start to deleting items. We see that all implementations never get close to the corruption limit, and they all seem to top out at around $2,300$. Interestingly is also that both implementations top out very close to each other in this test. We can also see how item are slowly leaving the witness-sets of the sequence heap and turning corrupt. Since we know that the simplified soft heap implementation does its corruption on INSERT, we therefore know that no additional corruption is getting added. This therefore implies that the simplified soft heap seem to mostly return uncorrupted elements in the beginning. We note that from the point where the two

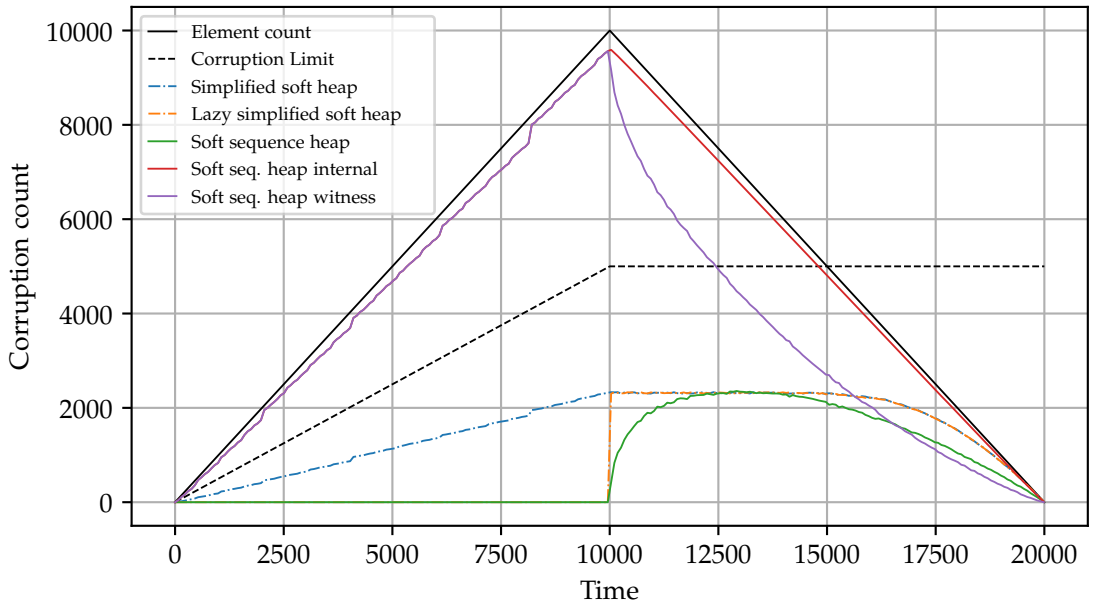


Figure 6.1: Here we count the number of corruptions over time as we insert 10,000 items and extract them again with $\varepsilon = \frac{1}{2}$. Each operation taking one time step. In addition to plotting the corruptions of the three implementation, we also plot the internal number of corrupted item (soft seq. heap internal) and the internal number of witnesses for the sequence implementation.

black lines intersect, which is 15,000 on the x-axis, soft heaps are not required to give any guaranties anymore. At this point the corruption limit is above the number of items left in the heap. However we see that both implementation are still not close to being fully corrupted.

6.2 Insertion and deletion speed

The whole reason for soft heaps is to improve the speed at a loss of precision, so let us compare soft heaps against other priority queues. In this section we test the performance of `insert` and `extract-min` for our best implementations of soft sequence heap and simplified soft heap. Since soft heaps are a type of priority queue then the performance of `insert` and `extract-min` is especially important since that is the core functionality of a priority queue.

Before one does experiments pitting soft sequence heaps against simplified soft heaps then one should consider that their formulas for the threshold t are different. This means that the t value does not change at the same time for the two sequences as can be seen on Figure 6.2. On the figure we see that t is mostly between 1 and 12, but it can also go higher. One could establish that without the figure, but it also illustrates that when using the same ε then t for simplified soft heaps is always 2 larger then soft sequence heaps. Looking at the figure we also see that for high ε values then large decreases are needed to get a change in t value compared to the lower ε values. In a way this shows that it really is the low ε values that one want to use most of the time,

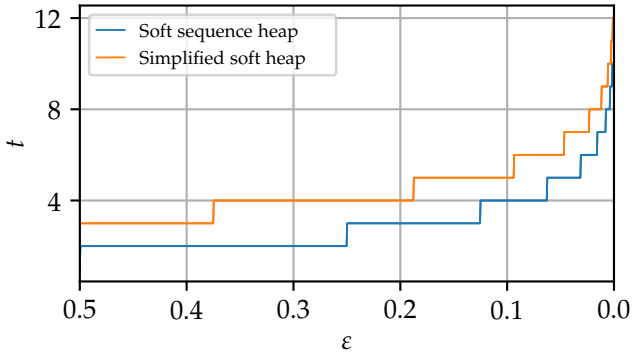


Figure 6.2: Showing how t changes as ε changes for soft sequence heaps and simplified soft heaps. Recall that the first uses $t = \lceil \lg \frac{1}{\varepsilon} \rceil$ and the latter uses $t = \lceil \lg \frac{3}{\varepsilon} \rceil$.

specifically values quite close to 0. Otherwise then you quite quickly get a low t value which leads to a lot of corruptions. Now knowing the effect of the different t formulas then one should also keep that in mind when seeing on Figure 6.1 that simplified soft heap makes less corruptions then soft sequence heap.

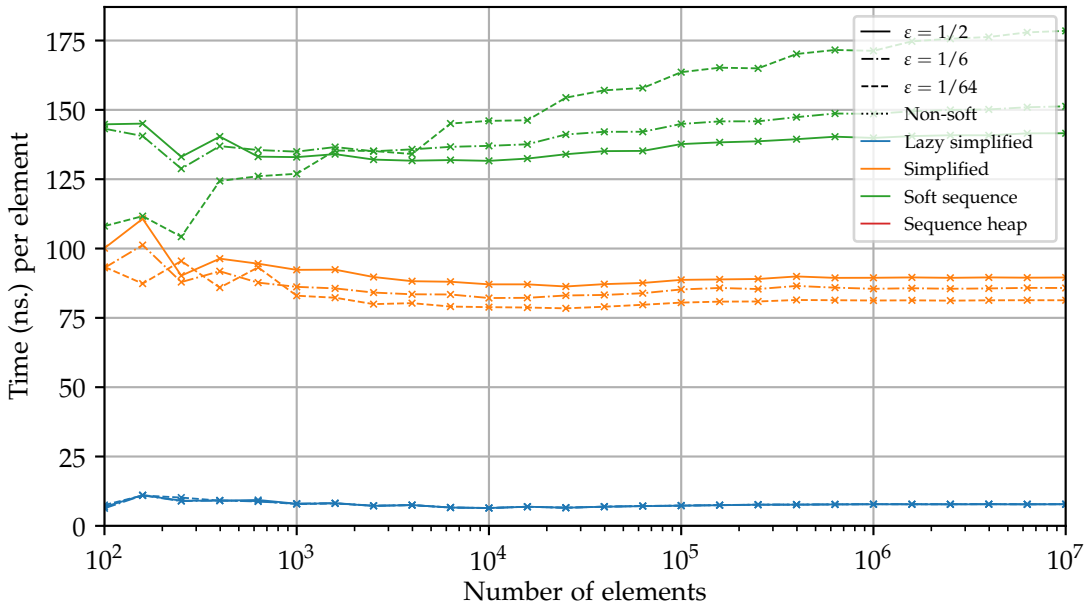
Different error rates As previously mentioned then soft sequence heaps trade-offs accuracy for performance. Therefore it is interesting to try different ε values and see how much it affects performance. You would expect that the more car-pooling you do, the faster the structure becomes, however you also have to remember that the car pooling can take extra time to do. We start with a high ε value of $\frac{1}{2}$ and from there try two smaller values. When we compare simplified and soft sequence based on them having the same ε value then it might not be completely fair since as previously mentioned then this does result in a bit lower threshold for soft sequence which should make it corrupt more items and thereby faster. `insert` performance can be seen on Figure 6.3a. As one would expect then lazy simplified is much faster than the others because it postpones work. We also see that for all three ε values then non-lazy simplified is faster then soft sequence heap. If looking at $\varepsilon = 1/2$ then simplified is only faster by about 30 nanoseconds. We are aware though that there are still improvements that can be made to our current best soft sequence heap implementation so it should be possible to close the gap even further. It is interesting that when going from high to low ε then the soft sequence gaps increases quite a bit unlike simplified which gap is primarily the same throughout and much tighter. This could be an indication of our soft sequence implementation being more expensive then simplified when running the data structures with many unpruned items. At low input values we see some zig-zag tendencies. This is probably because of clock noise, but looking at $\varepsilon = 1/64$ for soft sequence then it could also be a sign of car-pooling actually being expensive for small inputs since the higher ε values perform slower.

Unlike with `insert` we see on Figure 6.3b that soft sequence is fastest. Not by much in the beginning, but at the end it differentiates itself a bit from simplified. Another way in which this plot is the opposite of the insert plot is that here the gaps between the soft sequence series are small unlike simplified. Here we see the prize of doing lazy insertions since lazy simplified has a really steep time increase at the instead of it

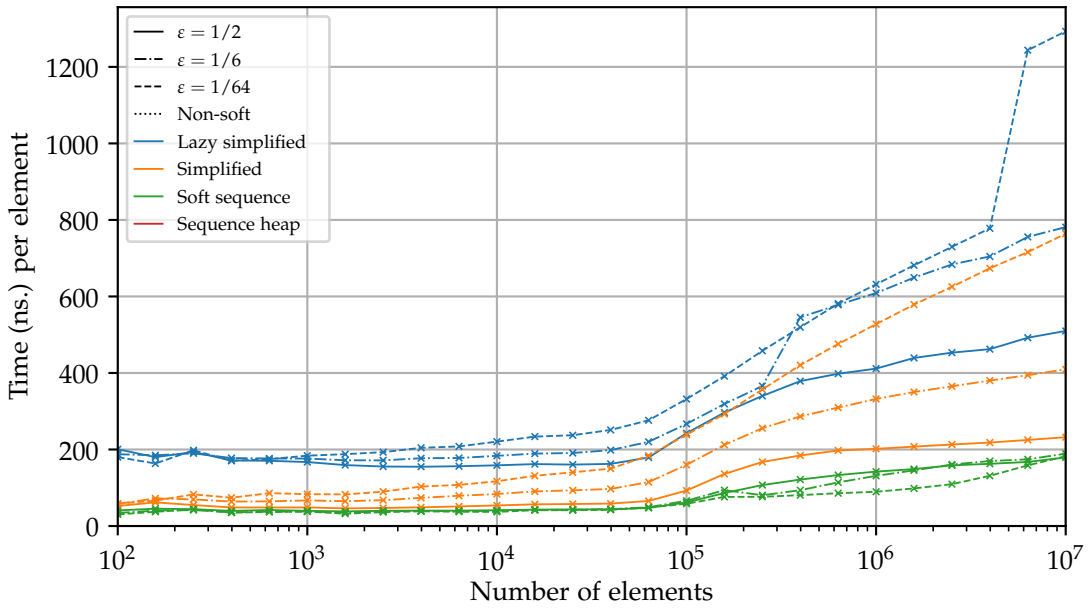
flattening out. This shows that its `extract-min` calls does not scale well which is quite unfortunate since as we see on the y-axis then `extract-min` calls are several times more expensive then `insert` calls. Only the highest ε for simplified has flattened out at the end so soft sequence's `extract-min` appear to scale better.

On Figure C.1 the time for `insert` and `extract-min` is added together. This plot is almost identical to the `extract-min` plot because as noted before then all the `extract-min` calls takes several times longer than the `insert` calls. Generally the series are closer together.

On Figure C.2 we have also included the container priority queue from STL and a solution where the minimum is extracted by sorting a vector. We see that vector sort is fastest which is expected. STL priority queue outperforms most solutions in the beginning, but ends up scaling quite poorly.



(a) `insert` time.



(b) `extract-min` time.

Figure 6.3: Soft heap implementations with ε set to $\frac{1}{2}, \frac{1}{6}, \frac{1}{64}$. This results in thresholds $t = 3, 5, 8$ for simplified and $t = 1, 3, 6$ for soft sequence. The combined plot can be seen on Figure C.1.

7 Application: Finding the k-smallest

In this section we will cover the practical tests around finding the k-th smallest element. In Subsection 2.10.1 we covered the theory of how to use a soft heap to solve this problem, and why it works. To get some perspective if it is worth it, we also included some alternative ways you might have solved this problem.

We will see that the soft heap implementations perform very poorly in this application, and cannot compete with the randomized select implementation. It does improve the pivot selection compared to random selection, but it is not enough to warrant the extra time spend.

7.1 Test setup

The test is going to first create a vector and fill it up with N integers of value 1 to N. After that we will scramble the vector and pass it to the function that needs to find the k-th smallest element. Timing starts just before we pass the vector to the function and stops as soon as we get the value back. We select k to be the median. The value of k should not matter for any of the algorithms, the only case where it would matter be a real advantage would be if you are lucky and hit the element in the first try. Since some algorithms are median approximating, this might have a slightly higher chance. However, at bigger input sizes this should not matter. Moreover, after just one pivot, the position should be irrelevant. Furthermore, if we hit the element on the first try, then we should easily be able to see it on the data.

This time we will be looking at the average instead of the minimum since this test can be very input dependent. We perform 10 tests for each implementation at each data size. We also made sure it is the same 10 scrambles that gets passed to each of the implementation. This should make it relatively fair, and make sure it is not just one good input for a given implementation.

We only test the non-lazy version of simplified since we do not need the stronger interface that it provides for this application. The non-lazy version should also perform better in all cases where the interface is not needed. We also did not test sorting as we talked about in theory. Random select should perform better than sort, size it is just

Quicksort where you only recurse in one part instead of two.

7.2 Implementation

Since we did not have an implementation chapter and only covered how the algorithms works in general, we want to add a few words about the implementations here. All the implementation are programmed to fit the same interface.

```
public:
    int Select(std::vector<int> &in, int index);
private:
    int Select(std::vector<int> &in, int left, int right, int index);
    int partition(std::vector<int> &data, int left, int right, int pivot);
```

The partition algorithm are shared between all the elements. We had to make a slight change to ensure decreasing argument, which helps when the inputs get very small. Doing this ensures that a recursive call will always contain fewer elements. This was done by splitting it into three groups instead of just two. The last one being elements that are equal.

The public SELECT function is what you can all to start the algorithm. This will call the internal SELECT that with the index left and right set to 0 and N. These are used so we do not have to transfer all the elements to a new vector and serves as bounds. We also see that the array is passed by reference to give good performance.

We have select a threshold value of $\varepsilon = \frac{1}{3}$ for the soft heap implementations. This means that we will extract $\frac{1}{3}$ of the elements and take the highest we have seen, as described in the theory section. This means a threshold of 3 for the simplified implementaitons and 1 for the sequence implementations.

7.3 Results

On Figure 7.1 we see the results of our testing. Here it is clear right away that soft heaps for this application performs worse across all input spaces compared to the non-soft solutions. This would suggest that the overhead for finding a median becomes a big part of the running time for the soft heap implementations as the pivoting algorithm are the same. It also suggests that the median we find might not be that much better than what the other solutions get, we will check this in a bit. A positive thing is that we can see that while the input is relatively small, then the soft heap implementations behave linearly in the number of elements, as the theory would suggest.

We can also see that the simplified implementation seems to perform better than the sequence implementation. From Section 6.2 we learned that the simplified version is better than simplified soft heap when it comes to inserting element, we also learned that the soft sequence is better at extracting elements, and that extracting elements is the more heavy operation. But, we have to remember that we only extract a third of the elements while we still insert all of them. So the relative performance between the two soft heap implementations follows the results that we have already seen and there are no surplices here.

Choosing the fastest soft heap implementation is one thing, but if the problem can be solved much faster using a non-soft implementation, then it is probably a good idea

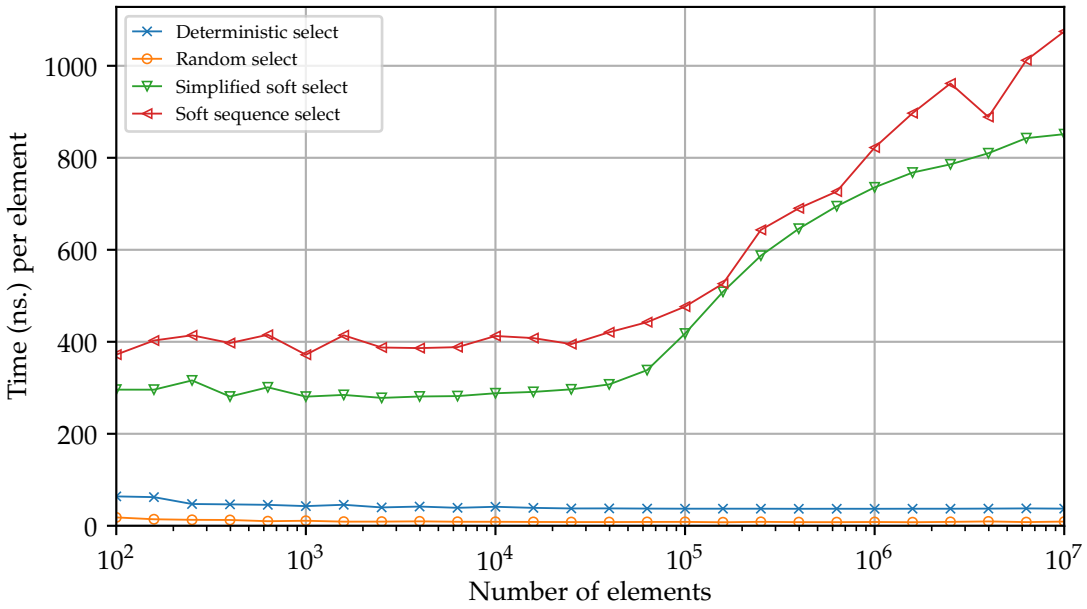


Figure 7.1: Time it takes to find the median

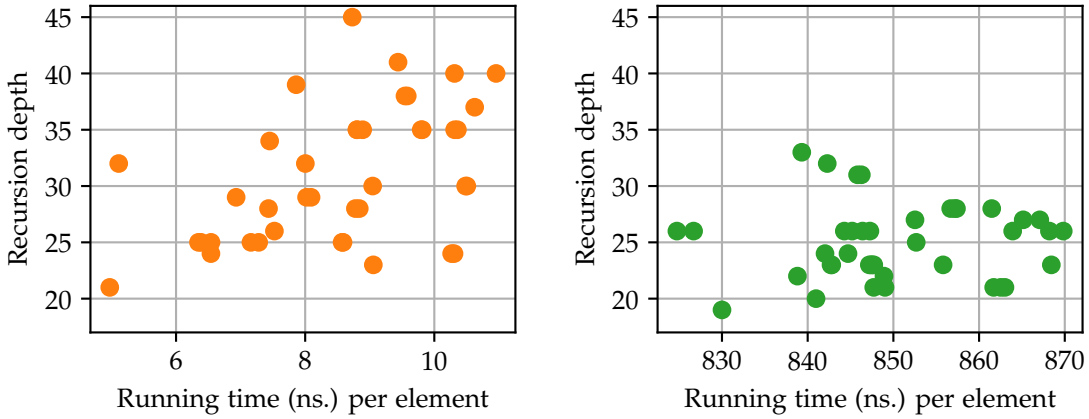
to use that instead. That was the motivation for also checking the soft heap implementations against other implementation. Sadly we can see that the soft heap implementations simply cannot keep up with the non-soft implementation. The randomized select completly wins the race. We also have to remember that even through the worst case time of random select is $\mathcal{O}(N^2)$, the expected time is still $\mathcal{O}(N)$. Deterministic select trails a bit behind, but still much better than the soft implementation.

Compared to the non-soft implementations the soft heaps cannot work on the data inplace. This means that for each iteration the data have to be transferred into a new soft heap; costing a lot of time.

Picking a pivot So soft heaps try to pick a better median, but we have to remember that picking a point that is far away from the median does not inherently make it bad, it is all about what part of group that the element we are looking for is going to be in. The median is the point where we are sure to remove the most elements no matter which group the element we are looking for resides in. We also have to be careful not to spend too much time on picking a good median, because we might be able to do multiple bad pivots by the time we have picked a median.

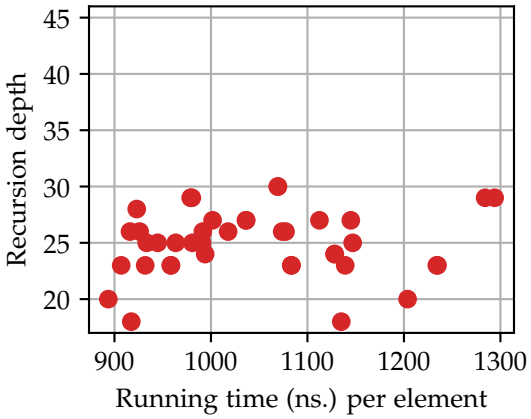
To explore this we run the tests again, but this time we count the number of recursions that gets made. The result of this can be seen on Figure 7.2 where we plot the number of calls to Select, which is going to correspond to the recursion depth. Since the deterministic select uses select for more than just the recursive call it have been omitted. We have also made it such that the ranges on the y-axis are the same for easier comparison.

An important thing to note before we comment on the results is that fewer recursion does not necessarily mean less work. A good pivot in the beginning can save a tremendous amount of work compared to a later one since the amount of data left is small.



(a) Random select

(b) Simplified soft select



(c) Soft sequence select

Figure 7.2: How 40 runs varies in time and varies in recursion depth for the different implementations. The input size is 10^7 and the soft implementations uses $\varepsilon = \frac{1}{3}$

The first thing we want to point out is if we were able to select the median every time, then we would expect to do $\lg 10^7 \approx 23.253$ recursions. Generally we see both of the soft heap produce a lower amount of recursions compared to the random select, which we would hope or else we are just doing extra work. They seem to produce around the same number of recursions. However, the variation in running on simplified soft is both lower than soft sequence heap, but also a lot more consistent. We see how the range of simplified goes from 820 to 870 nanoseconds, compared to soft sequence's 900 to 1300 nanoseconds.

Random select is super fast as we already know. We can also see that it varies a lot more in the number of recursions and generally takes more recursions, but is still faster.

7.4 Recap

Soft heaps are able to improve the selected pivot point compared to simply picking one at random, but with the current implementations the cost is simply too high compared to other solution. One of the big downsides is that the soft heaps cannot work on the elements in place, and have to use a lot of time creating the soft heap too.

Other solution, because of their higher density, also uses up less cache and therefore degrade later in performance. We have already seen in Chapter 4 how important a dense structure is, and this seems to be one of the main reasons random select performs so well. Furthermore, you might be able to beat random select by finding a balance between total random selection of the pivot and finding the median.

8 Application: Heap selection

In this section we will cover the selection problem called heap selection. Recall from Subsection 2.10.2 where we covered the corresponding theory, that the problem is finding the k smallest elements in a heap. We test both a non-soft implementation and two soft implementations. We need the strengthened interface for this algorithm, so we will be using the lazy version of the simplified soft heap. We will be using our sequence heap as the non-soft implementation since it performed better than the built in priority queue in the test in Section 6.2.

In the tests we see the soft heap implementation perform a bit worse than the non-soft implementation. We see that the selection part of the soft heap implementation is very fast. We also see that increasing ε actually decreases the performance of the heap, contrary to what we would expect. The size candidate set follows what we would expect based on ε.

8.1 Test setup

For this application the first step is to create a heap. We create a root node and a recursive function that can take a node and recursively fill the children of the node. The function takes a depth parameter and will create a tree until it reaches the desired depth. When it picks a value for a given node it will look at its parent and then pick a random value and add it to the value of the parent. This makes it such that the values in the trees are not predictable and that it satisfies the heap constraint.

The root node we create have a value of 0. We call the recursive function to create our tree. We set the depth to be 22, this gives us a perfectly balanced tree with $4,194,303$ nodes. Normally we would run each test in isolation, but because creating a tree can take quite some time we only created one tree and then run all the tests. This works since none of the functions are destructive. We are aware that since we run each test after each other, the cache performance might be affected. However, since the tree just have been created anyway it could have a cache advantage anyways. Furthermore, we run all the tests for each implementation one after the other, so any advantage should apply to all. We also still use the minimum of 10 runs, so even the first should also be

fine.

We are going to test how the different implementations works at many different sizes of k. As noted the tree size will be fixed. This means that for the small tests k will only be a very small part of the total amount of items. There will be a performance difference when the algorithm starts to reach the bottom of the tree, but as long as it does not then it does not matter how many nodes there are left. The unused nodes will just lay in RAM.

8.2 Implementation

For the non-soft implementation we follow the pseudo code (Algorithm 2) we provided in Subsection 2.10.2. We use sequence heap because that one was the fastest in our testing at high input sizes. In Figure C.2 we saw that for small inputs it might be a tad slower, but for bigger inputs it seems to win by quite a bit. For the array we use a vector, this should provide good speed when inserting in the back.

For the soft implementations the code is a bit more complicated. The pseudo code can be seen on Algorithm 3 and it originates from [14]. We use the soft heap to generate the candidate set S. We use a vector for the array S. This should also help make the final select in the algorithm fast. To select we use the randomized select algorithm that we learned was super fast in the previous test in Chapter 7. The rest mostly matches the pseudo code. We test the soft sequence heap and lazy version of simplified soft heap.

8.3 Results

The first thing we was interested in was the time it took for them to complete. We ended up testing it for three different values of ε. We used $\varepsilon = \frac{1}{4}$ as we had in our pseudo code and as it was in [14], and we also used $\frac{1}{2}$ and $\frac{1}{64}$. The results can be seen on Figure 8.1. We see that none of the current soft implementations can beat the non-soft implementation. However, compared to the results in Chapter 7 this is much closer.

We initially did $\varepsilon = \frac{1}{4}$ and did further break down of the soft heap running times to see what takes time. The plot for this can be seen on Figure 8.2. Here we saw how the majority of the time was taken up making the array of candidates. From this we thought that we would up the amount of allowed corruption to speed up the part that build S. The part that would select from S seemed like it could handle some more work. The idea was to find balance between. But as can be seen on the figures, setting $\varepsilon = \frac{1}{2}$ does not improve the running time, it actually makes it worse.

The main reason the assumption did not hold most likely have to do with the fact that the heaps can be sparse. Compared to the tests we have made in the previous chapters, this one uses the dynamic features of the soft heaps. That implies that elements are removed constantly and this can cause some sequences to becomes quite sparse. The main speed up from car-pooling comes from the fact that sequences get car-pooled to reduce the number of elements, but if we also reduce the number of elements from extracting, then car-pooling might not have a big effect or it may even hamper the performance by increasing the work load.

To be complete we also tried a larger ε of $\frac{1}{64}$. Looking at Figure 8.1 we can see that for the lazy implementation this seems to slightly increase the performance at the lower

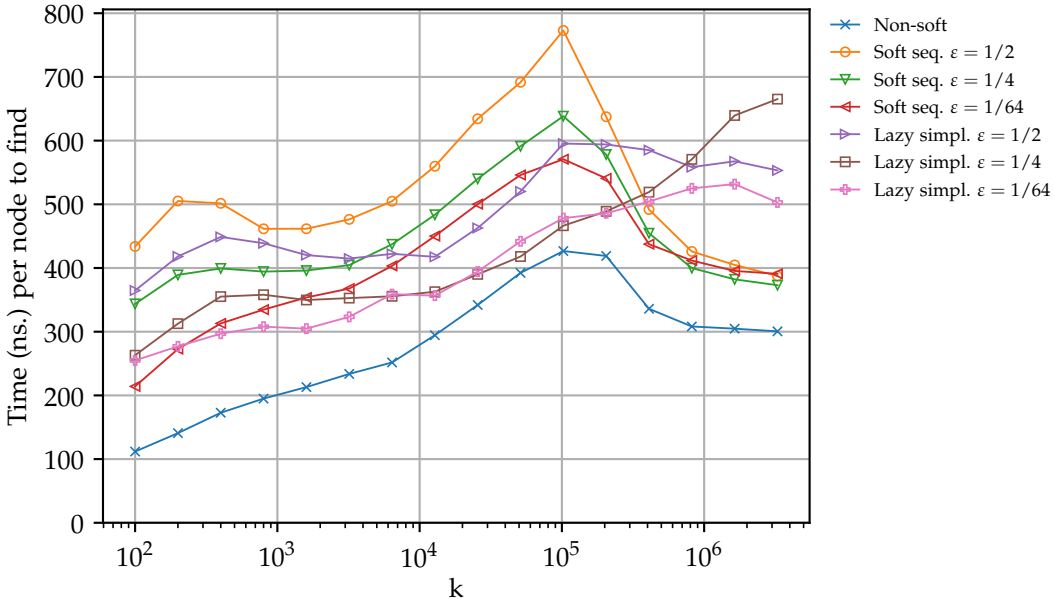


Figure 8.1: Heap selection time for non-soft, soft sequence, and lazy simplified soft heap using different ε values.

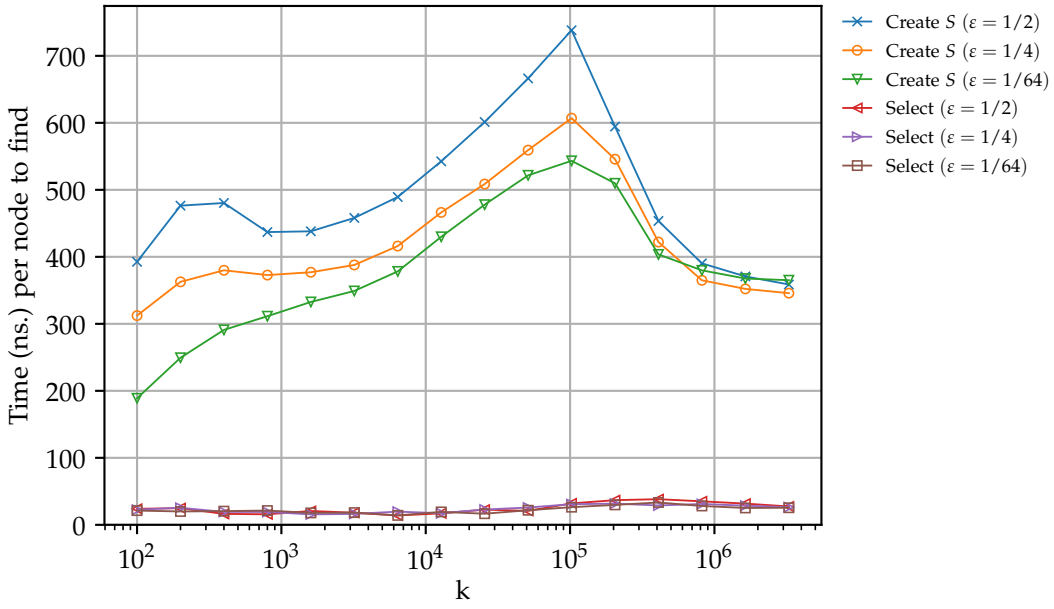
and high sizes of k, but not by much. For soft sequence heap we see it perform slight better from the small to medium sizes of k only be caught up to when k gets large.

Size of S Even though the time it takes to select from S became irrelevant as we could not decrease the time of soft heaps, it can be interesting to look at how much corruption was introduced into S. From the size of S we can calculate the current number of corrupt currently in the heap. This is done by the simple formula $|S| - k$. This is not something you could do without the strengthened interface and show how this interface can give much stronger information than the simple corruption bound.

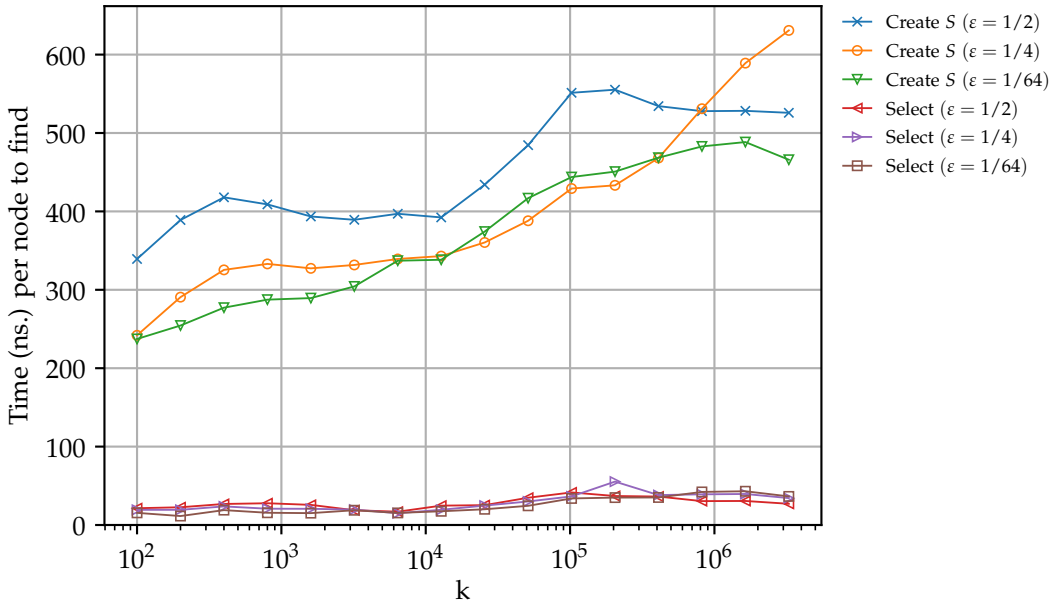
The results of the test can be seen on Figure 8.3. Soft sequence heap creates smaller candidate sets compared to the lazy simplified version at a given ε value. We want $|S|$ to be as low as possible, and therefore the final select should be fast in the soft sequence cases. It is can also be seen that as k reaches the number of elements in the heap at around $k = 10^5$. At that point then the size of S gets smaller compared to k. This happens because there are limited number of elements left that can be corrupt. We can see it creates the same trend as we saw on Figure 6.1 from Chapter 6 and gets very close to k. We also see that a lower ε gives less corruption, as one would expect even though we are not explicit using the corruption bound.

8.4 Recap

Soft heaps cannot beat our non-soft implementation in this test. However, compared to the find k-smallest test in Chapter 7, the soft heap implementations puts up more of a fight in this test. Soft heaps also plays a much larger role than just finding a candidate for a pivot point. The performance of the soft heaps do vary, but overall they follow



(a) Soft sequence heap



(b) Lazy simplified soft heap

Figure 8.2: Time it takes to make S and time it take to `select` k-smallest elements from S. The tree has a depth of 22.

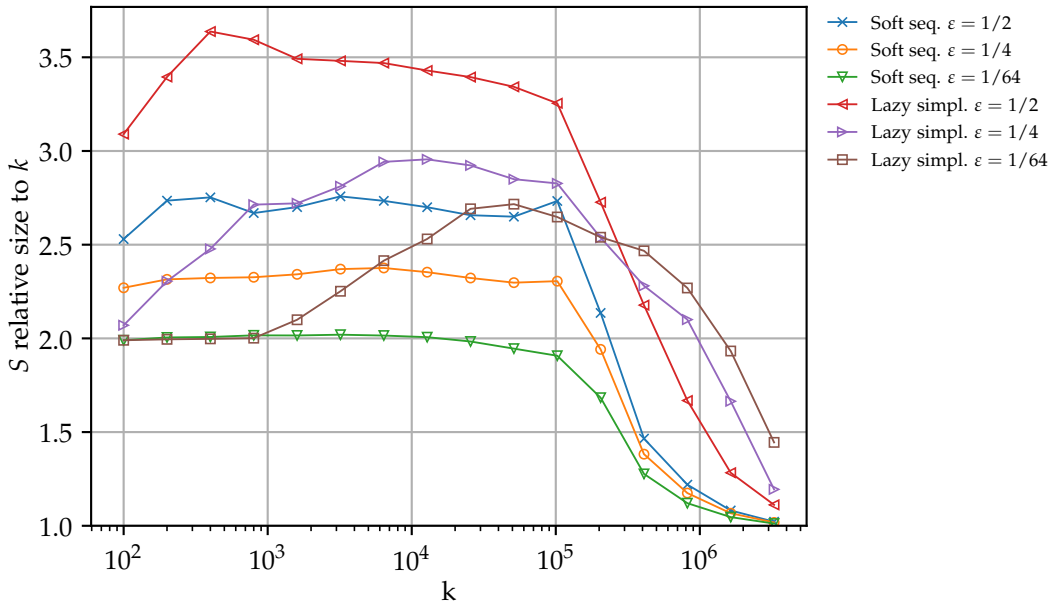


Figure 8.3: The size of S at the end of heap selection using soft heaps, divided by k.

the same trend. From Figure 8.3 we saw that it was able to bring down the number of elements to a constant factor of k. The chose of ε also seems to have a lesser effect on the total running time of the whole thing.

Another thing is show is how the new interface have some great properties that can. The new interface can help when used in a dynamic setting. Without this, we would end up losing your guarantees because you would delete elements while not lowering the corruption bound. With further optimizations to the soft heaps implementations, you probably would be able to beat the current non-soft implementation.

9 Conclusion

We have covered the theory around soft heaps and related concepts. We have gone into depth with why soft sequence heaps and simplified soft heaps comply with the corruption limit of εN. We also covered the theory of how the running time for soft sequence heaps are $\mathcal{O}(\lg \frac{1}{\varepsilon})$ for insert and $\mathcal{O}(1)$ for all other operations, and how soft heap simplified have $\mathcal{O}(\lg \frac{1}{\varepsilon})$ for extracting the minimum $\mathcal{O}(1)$ for all other operations. The theory about applications that uses soft heaps and lists which is an important component of soft heaps have also been explained.

In the lists experiments we test a number of different list implementations. We saw the difference that could be in the performance of two different algorithms, even if they had the same $\mathcal{O}$-notation. Here we also saw how some hardware components would favor certain implementations. We have shown that often we want a data structure to be dense, which a node base structure inherently is not.

The initial tests of soft heaps showed that the real number of corruptions were way better than the corruption limit that it needs to adhere to. We also saw that even after we had no guarantee from the corruption limit, the heaps would still have a reasonably amount of non-corrupted elements. We also saw that most of the elements inside the soft sequence heap are actually internally corrupt.

We learned a number of things in the test where we insert and extract a large number of elements in a soft heap. Here we saw that soft sequence heaps performed better when inserting and simplified soft heaps performed better when extracting elements. We see that soft sequence heaps only have minor performance differences when changing ε. Simplified soft heaps on the other hand are greatly affected. With $\varepsilon = \frac{1}{2}$ it performs about the same as soft sequence heaps. We also saw that the soft heap implementation perform better than C++'s priority queue at large inputs.

In the k-th smallest application test we saw that finding a good median simply was not worth spending extra time on, when the pivoting part of the algorithm could utilize the hardware well. We did however show that it did improve the quality of the median compared to just selecting a random median. A problem is also that we use a lot of time adding elements and the never use them.

From our find the k smallest elements in a heap experiments, we found that the non-

soft priority queue version beat the soft heap versions consistently. Another interesting result was that the soft versions with large thresholds resulted in slower performance than the ones with small thresholds. This is contrary to what the theory tells us. The test results from heap selection are of particular interest to us since they actually used the soft heaps in a dynamic way instead of just inserting back to back like some of our other tests.

Now that we have done all the tests we have seen that $\mathcal{O}$-notation does not capture all the performance characteristics there are, but in many cases it still provided a relatively correct estimate and is easy to use compared to other models.

In our testing soft heaps seems to have problems beating most non-soft heap implementations. However we saw in the list experiments that the implementation details can be very important for the performance. This means that some tuning to make the soft heap implementation more hardware friendly could probably make soft heap perform quite well in some cases. Our hypothesis from Chapter 1 was that soft heaps with low thresholds would outperform non-soft heaps in some areas. We find this to be mostly false, at least for the specific experiments that we do and using our implementation that could be better tuned. With that said then it did outperform C++'s priority queue for large input sizes. In the future work section we mention a number of things that might be able to help the performance of soft heaps.

9.1 Future Work

There are a number of things we could try to improve the performance of the different soft heap implementations. For the corruption set in both soft implementations there are a number of things you could try. Ideally they could be in some sort of vector or queue. The problem is that we want to be able to concatenate them in constant time. A solution to this could be to be inspired by C++'s deque and how it put things into chunks. We could make a structure that would move elements into chunks when there are few element, with a chunk size of some factor of the cache line. It would then start linking them with pointers when there are more elements. Linking could be done in constant time. This could improve the time when extracting elements because the element could be more dense, and that they would be in the same cache line. It would however also complicate it quite a bit about what to do with half filled chunks and how to track what is used and what is not.

Dynamic setting If you were in a dynamic setting you might want to look at how you might not want to make stuff overly sparse. One of our tests suggested that sparse soft heaps might not respond as we would expect when we change ε. There are a number of things you could try. One could be to downgrade the rank once it losses enough elements, or skipping the reduce step if the amount of elements are low.

Simplified soft heap We could try and move the next pointer out of the main node struct. This would bring it from 42 bytes down to 32 bytes. This would allow two elements per cache line. There is no need to go lower since we learned that the minimum allocation size is 32.

There are a number of other ways you could implement lazy insert on, that might give better performance. You could do the inserts right away and then lazy return the

corrupted set instead of doing the lazy insert.

Soft sequence heap We could make the sequences into arrays or queues. This would increase the density. It could also improve the merging speed, since the items we would be merging would be together in memory compared to a node based solution. We do lose the fact that we can stop merging when one of the lists are empty, but we think it will pay off based on the list experiments.

Another speed up could be to allocate a fixed space for the roots and suffix-min. This could help speed up updating suffix-min. The space needed here is not a lot, since we only need one for each rank. If we just made an array of size 32 then you would have other problems before you would run out of ranks slots.

Applications There exists many more applications that could be implemented. The application paper [14], which we already have used quite a bit contains multiple other application that can be tested. There are also the minimum spanning tree algorithm that motivated the soft heaps in the beginning. Testing of melding and applications that use melding is also something that we have not explored. There is also a number of things that could be tested for the more dynamic solution, like heap select.

We find the application that use the stronger interface interesting. Relaying on the returned elements that can be corrupt is a very interesting concept, compared to only using the corruption bound.

Bibliography

[1] Alka Bhushan and Sajith Gopalan. External memory soft heap, and hard heap, a meldable priority queue. In Joachim Gudmundsson, Julián Mestre, and Taso Viglas, editors, *Computing and combinatorics. 18th annual international conference, COCOON 2012, Sydney, Australia, August 20–22, 2012. Proceedings*, volume 7434 of *Lecture Notes in Computer Science*, pages 360–371. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32241-9. doi:10.1007/978-3-642-32241-9_31 .

[2] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448 – 461, 1973. ISSN 0022-0000. doi:10.1016/S0022-0000(73)80033-9 .

[3] Gerth Stølting Brodal. Soft sequence heaps. Unpublished, 2020.

[4] Bernard Chazelle. Car-pooling as a data structuring device: The soft heap. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *Algorithms — ESA' 98*, pages 35–42, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-68530-2. doi:10.1007/3-540-68530-8_3 .

[5] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, November 2000. ISSN 0004-5411. doi:10.1145/355541.355562 .

[6] Bernard Chazelle. The soft heap: An approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, November 2000. ISSN 0004-5411. doi:10.1145/355541.355554 .

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[8] Dorit Dor and Uri Zwick. Selecting the median. *SIAM J. Comput.*, 28(5):1722–1758, May 1999. ISSN 0097-5397. doi:10.1137/S0097539795288611 .

[9] Dorit Dor and Uri Zwick. Median selection requires $(2 + \epsilon)n$ comparisons. *SIAM Journal on Discrete Mathematics*, 14(3):312–325, 2001. doi:10.1137/S0895480199353895 .

[10] G.N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197 – 214, 1993. ISSN 0890-5401. doi:https://doi.org/10.1006/inco.1993.1030 .

[11] Tomasz Jurkiewicz and Kurt Mehlhorn. The cost of address translation. *CoRR*, abs/1212.0703, 2012. URL http://arxiv.org/abs/1212.0703.

[12] Haim Kaplan and Uri Zwick. A simpler implementation and analysis of chazelle's soft heaps. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, page 477–485, USA, 2009. Society for Industrial and Applied Mathematics.

[13] Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Soft heaps simplified. *SIAM Journal on Computing*, 42(4):1660–1673, 2013. doi:10.1137/120880185 .

[14] Haim Kaplan, László Kozma, Or Zamir, and Uri Zwick. Selection from Heaps, Row-Sorted Matrices, and X+Y Using Soft Heaps. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *2nd Symposium on Simplicity in Algorithms (SOSA 2019)*, volume 69 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:21, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-099-6. doi:10.4230/OASIcs.SOSA.2019.5 .

[15] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, March 1995. ISSN 0004-5411. doi:10.1145/201019.201022 . URL https://doi.org/10.1145/201019.201022.

[16] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming*, pages 49–60, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45022-1. doi:10.1007/3-540-45022-X_6 .

[17] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, January 2002. ISSN 0004-5411. doi:10.1145/505241.505243 .

[18] Peter Sanders. Fast priority queues for cached memory. *J. Exp. Algorithmics*, 5:7–es, December 2001. ISSN 1084-6654. doi:10.1145/351827.384249 .

[19] Arnold Schönhage, Michael S. Paterson, and Nicholas Pippenger. Finding the median. Unpublished, April 1975. URL http://wrap.warwick.ac.uk/59399/.

[20] Douglas Stinson. *Cryptography: Theory and Practice,Second Edition*. CRC/C&H, 2nd edition, 2002. ISBN 1584882069.

[21] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[22] Mikkel Thorup, Or Zamir, and Uri Zwick. Dynamic Ordered Sets with Approximate Queries, Approximate Heaps and Soft Heaps. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 95:1–95:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-109-2. doi:10.4230/LIPIcs.ICALP.2019.95 . URL http://drops.dagstuhl.de/opus/volltexte/2019/10671.

Curriculum

Here we list our curriculum.

Paper	**Curriculum**
Kaplan et al. [13]	All of it, excluding section 6 and 7 (extensions and variants).
Brodal [3]	All of it, excluding section 4 (ternary tree).
Chazelle [6]	Abstract and section 1, 2, and 3.
Tarjan [21]	Abstract and section 1 and 2.
Kaplan et al. [14]	Section 2-3.1.
Cormen et al. [7]	Section 9.2 and 9.3.
Stinson [20]	Page 55.
Jurkiewicz and Mehlhorn [11]	Abstract and section 1.
Kaplan and Zwick [12]	Abstract.
Chazelle [5]	Abstract.
Chazelle [4]	Abstract.
Bhushan and Gopalan [1]	Abstract.
Schönhage et al. [19]	Abstract.
Dor and Zwick [8]	Abstract.
Frederickson [10]	Abstract.
Sanders [18]	Abstract.
Pettie and Ramachandran [16]	Abstract.
Pettie and Ramachandran [17]	Abstract.
Dor and Zwick [9]	Abstract.
Thorup et al. [22]	Abstract.
Karger et al. [15]	Abstract.

A
List performance

A.1 Clock measurement code for STL vector

```
auto testVector(int testSize) {
    auto startOfFunction = std::chrono::steady_clock::now();
    std::vector<int> testArray;

    for (int j = 0; j < testSize; ++j) {
        testArray.push_back(j);
    }

    auto startOfSum = std::chrono::steady_clock::now();
    int sum = 0;
    for (auto i : testArray) {
        sum += i;
    }
    auto stop = std::chrono::steady_clock::now();

    std::cout << sum << ",";
    std::cout << std::chrono::duration_cast<std::chrono::nanoseconds>(
    startOfSum - startOfFunction).count() << ",";
    std::cout << std::chrono::duration_cast<std::chrono::nanoseconds>(stop
    - startOfSum).count() << ",";
    return std::chrono::duration_cast<std::chrono::nanoseconds>(stop -
    startOfFunction);
}
```

A.2 Extra plots

Here are some extra plots from Chapter 4.

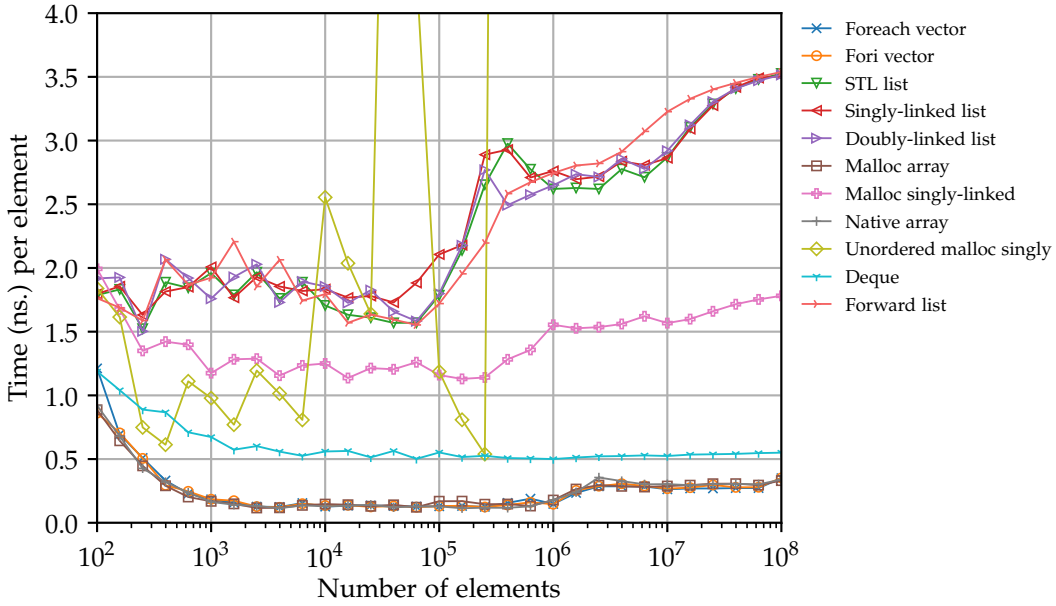


Figure A.1: Absolute performance of iterating through a list while summing the values. This is the *average* of 10 runs

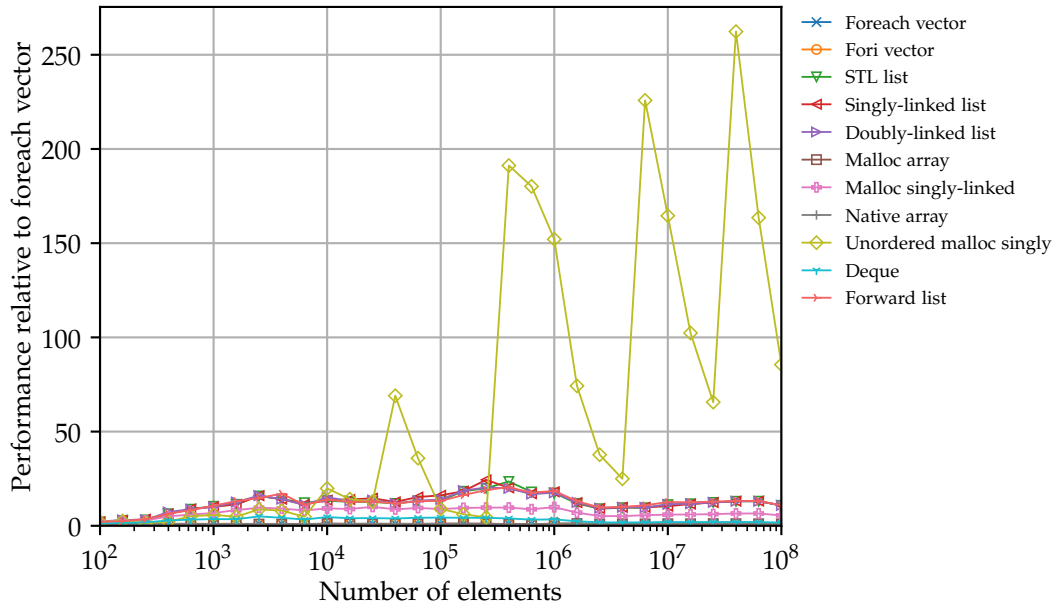


Figure A.2: Same as Figure 4.2b but zoomed out to fit data points of the new unordered malloc singly-linked list implementation

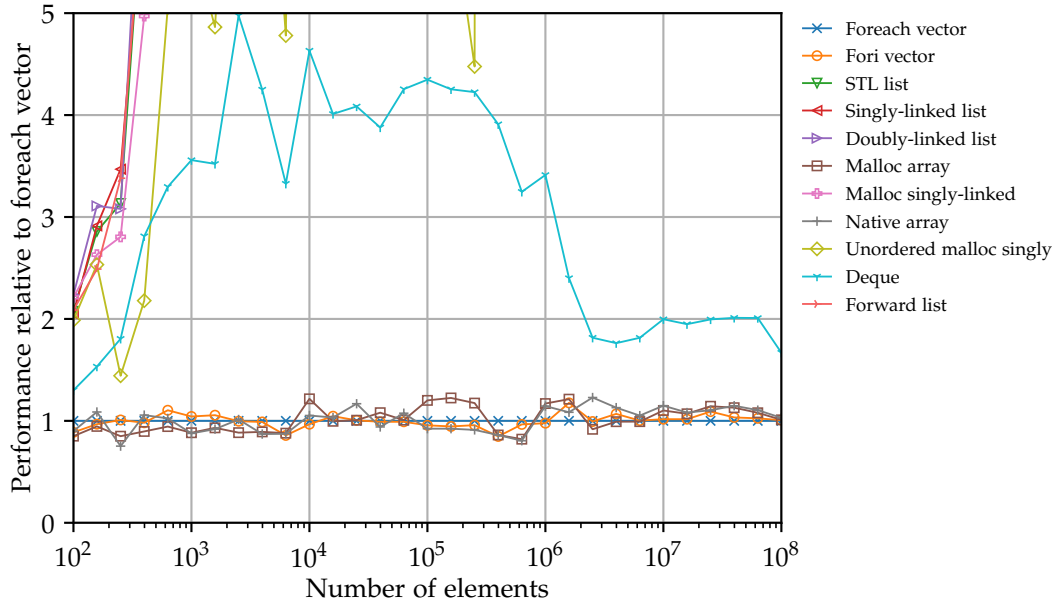


Figure A.3: Same as Figure 4.2b, but zoomed it to better see how the algorithms close to relative performance 1.

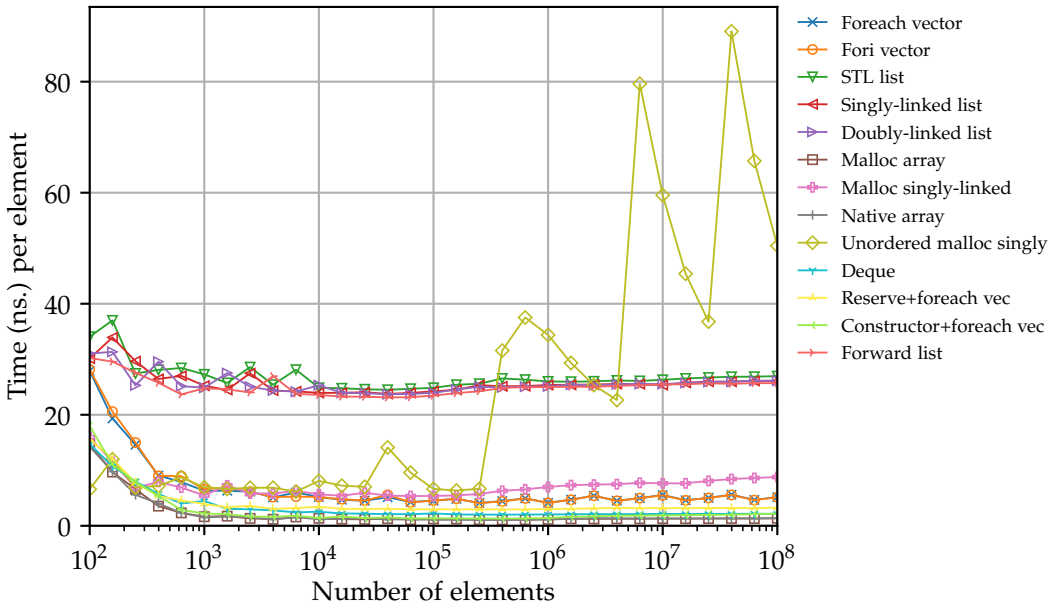


Figure A.4: Zoomed out version of Figure 4.10

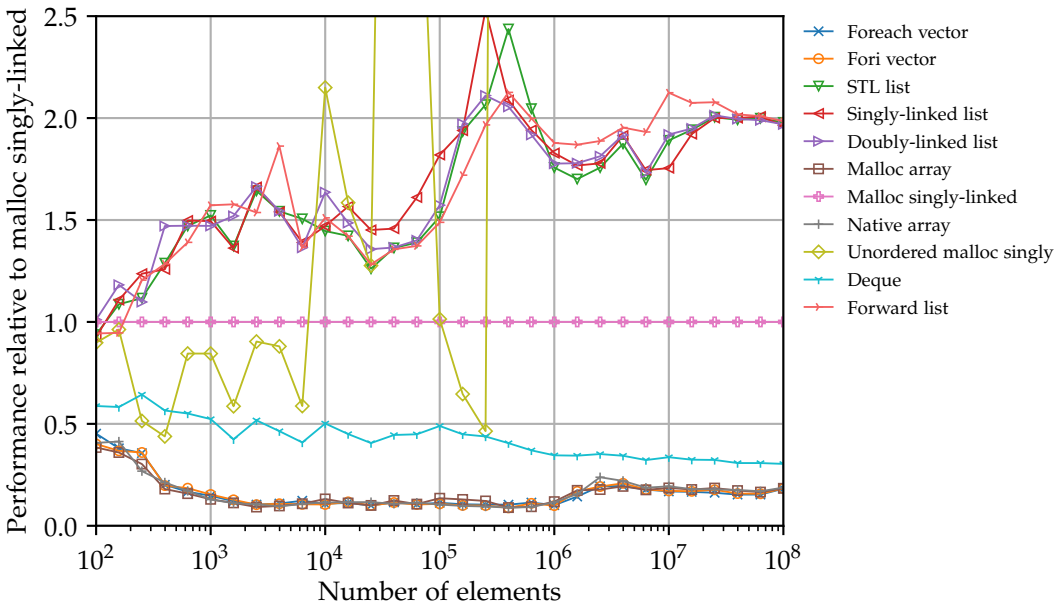


Figure A.5: Running time performance normalized to malloc singly-linked list

B
Soft sequence heap results

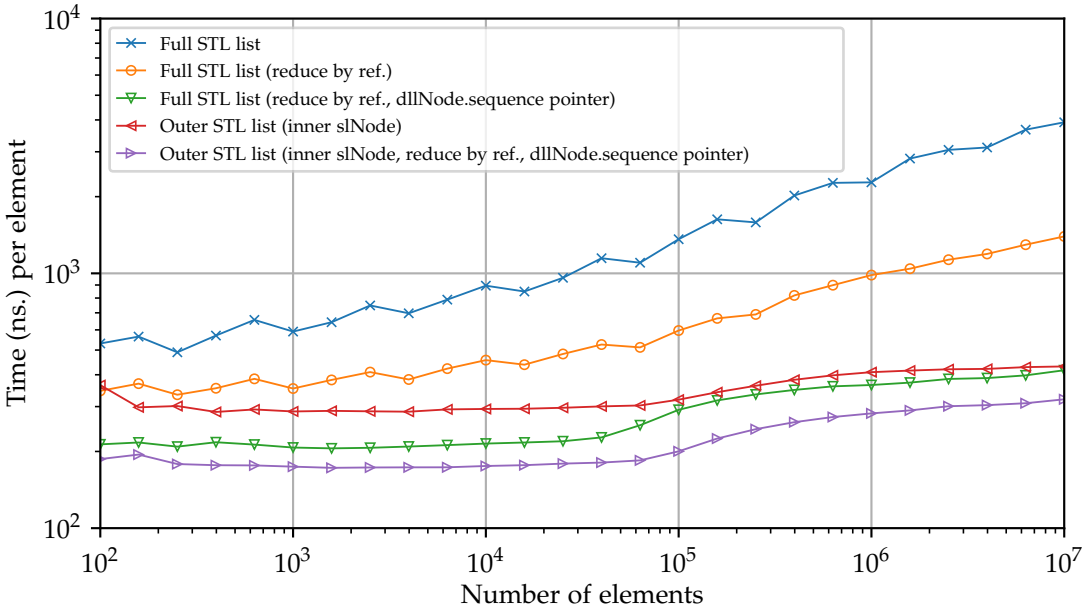


Figure B.1: Like Figure 5.3, but time for insert and extract-min is combined.

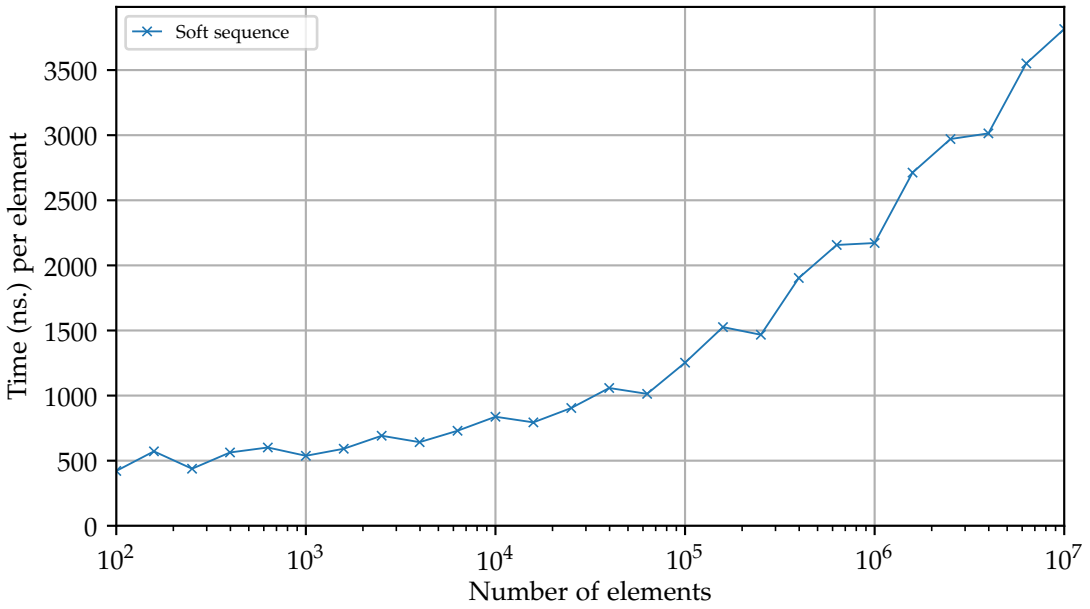


Figure B.2: Insert time per element for our first working soft sequence heap implementation and other priority queues. The first working implementation used STL list for every list need. The soft heaps are instantiated with threshold 1.

B.1 Flamegraphs

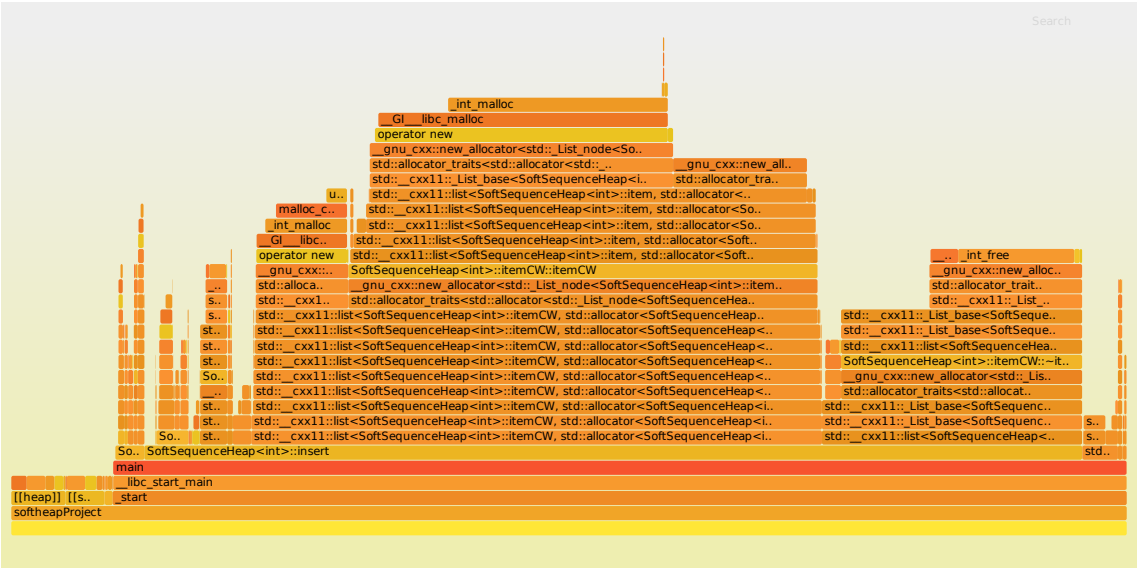


Figure B.3: Flamegraph for full STL list version where reduce function is pass-by-reference.

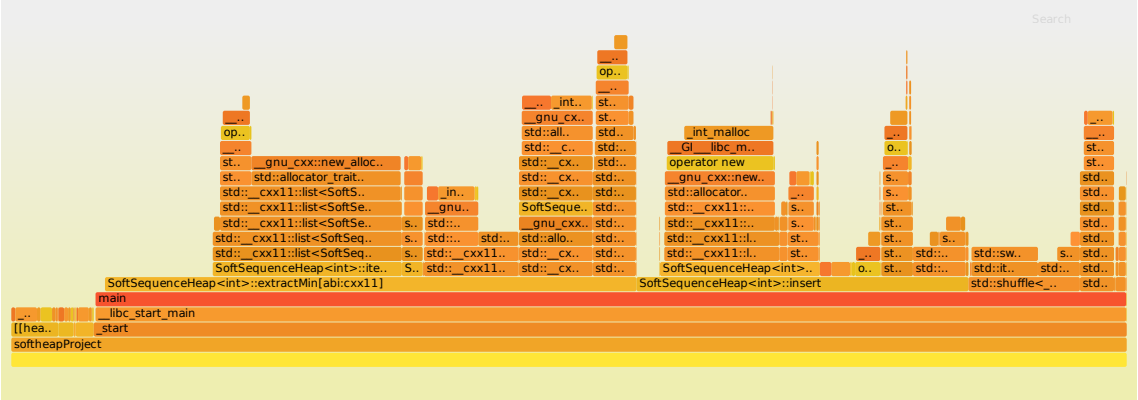


Figure B.4: Flamegraph for full STL list version where reduce function is pass-by-reference and `dllNode.sequence` is a pointer.

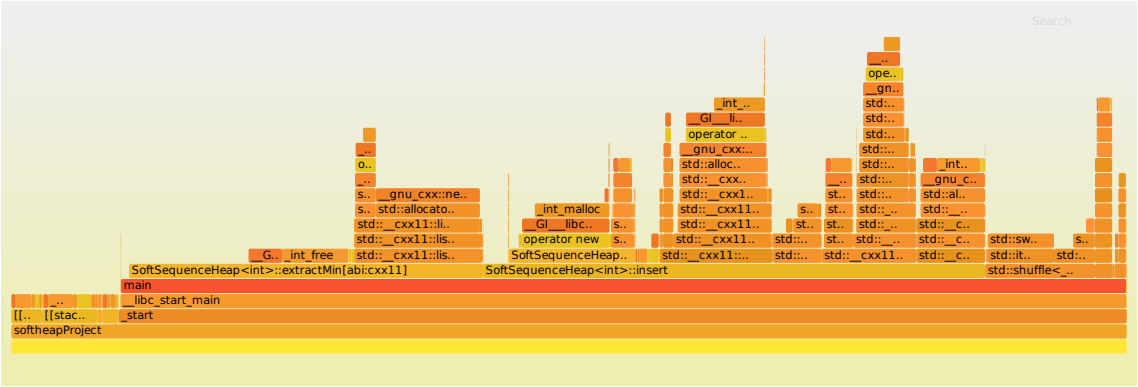


Figure B.5: Flamegraph for soft sequence heap using STL list as outer list and slNode as inner list.

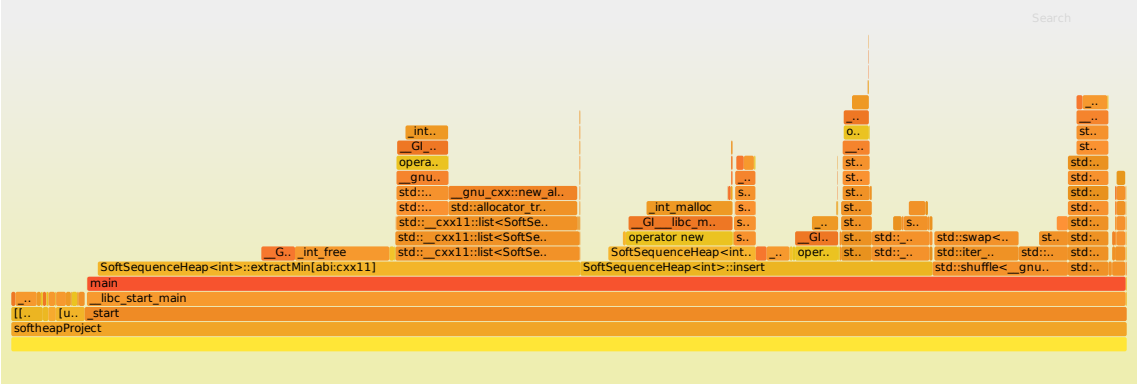


Figure B.6: Flamegraph for soft sequence heap using STL list as outer list, slNode as inner list, and same improvements as Figure B.4.

C
Soft heap experiments results

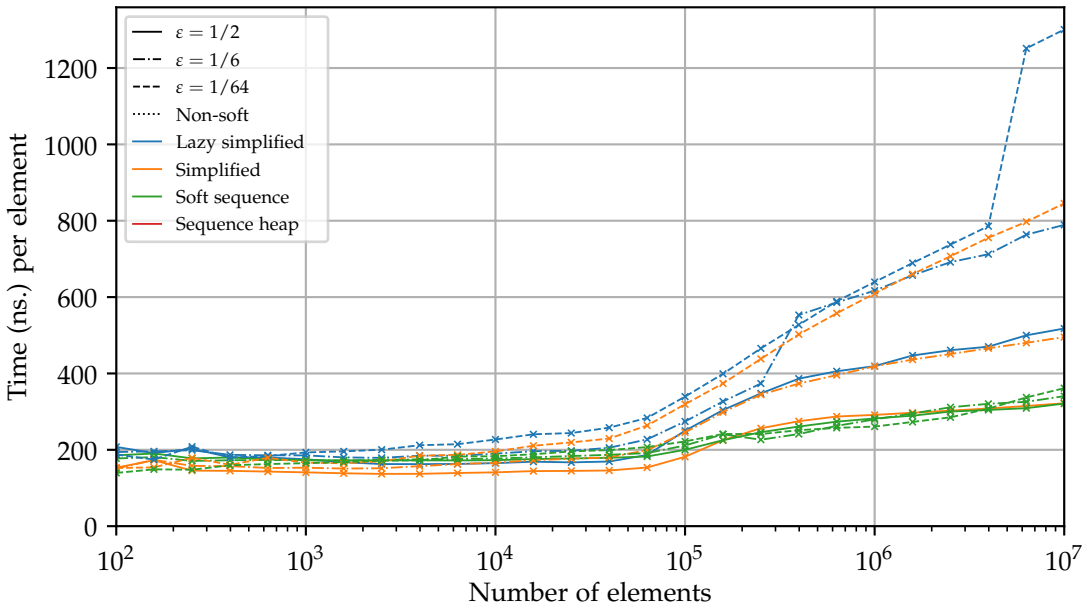


Figure C.1: `insert` and `extract-min` time combined for soft heap implementations with ε set to $\frac{1}{2}$, $\frac{1}{6}$, and $\frac{1}{64}$. This results in thresholds $t = 3, 5, 8$ for simplified and $t = 1, 3, 6$ for soft sequence.

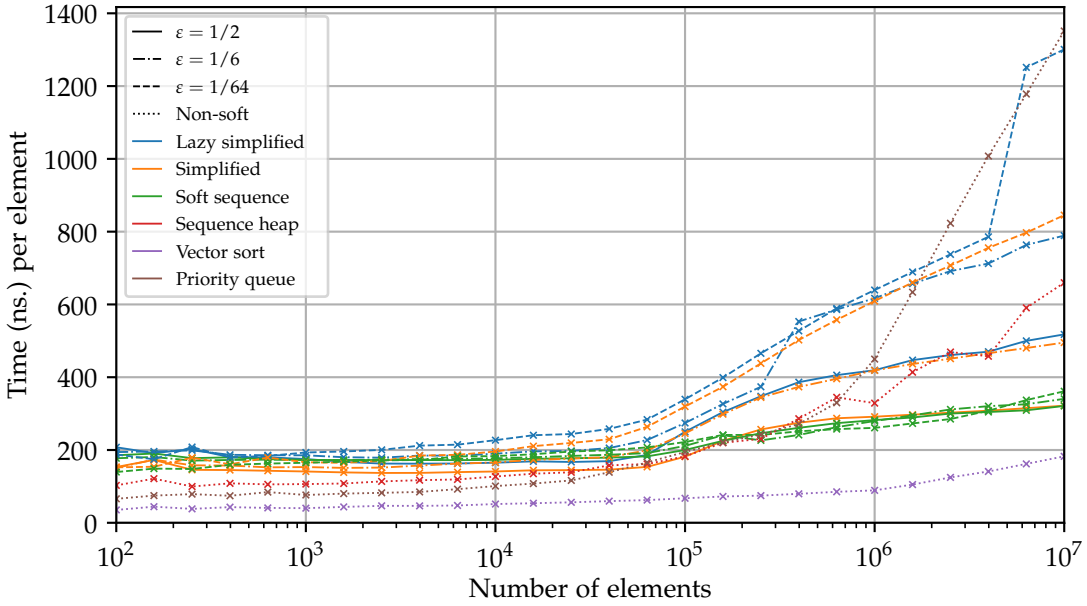


Figure C.2: Same as Figure C.1, but sequence heap, vector sort, and priority queue are also included.