Master Thesis
Selection in a Heap

Kenn Daniel
20118457
kenn_daniel@hotmail.com

Casper Færgemand
20118354
shorttail@hotmail.com

Advisor: Gerth Stølting Brodal

June 13, 2016

Abstract

We test the algorithms presented by Frederickson [5] and investigate if they follow the theoretical bounds, with a focus on whether the theoretical linear time algorithm can be used in practice. We do this by implementing the algorithms (link in Table 0.1), and then measuring and comparing them on several parameters. Additionally we also reproduce the theoretical results to a higher degree of detail.

In Danish

Vi tester algoritmerne præsenteret af Frederickson [5] og undersøger om de følger de teoretiske grænser, med fokus på om algoritmen med teoretisk lineærtids kompleksitet kan bruges i praksis. Dette gjorde vi ved at implementere algoritmerne (link i Table 0.1), og derefter måle og sammenlige dem på adskillige parametre. Derudover efterprøver vi de teoretiske resultater med en højere grad af detalje.

`github.com/HeapSelection/Heap-Selection`

Table 0.1: Link to implementations

Contents

Contents **3**

1 Introduction **6**

2 Notation **8**

3 Naive Algorithm $O(k \cdot \log k)$ **10**

3.1 Introduction 10

3.2 The algorithm 10

3.3 Termination 11

3.4 Correctness 11

3.5 Theoretical bound 11

4 Framework **12**

4.1 Introduction 12

4.2 Partitioning 12

4.3 Clans 12

4.4 Recursion 13

4.5 Priority queue preservation 14

4.6 Category 14

5 Select **16**

5.1 Introduction 16

5.2 The algorithm 16

5.3 Further reading 18

6 Clans **19**

6.1 Clan 19

6.2 Clan-3 20

6.3 Clan-4 20

7 SEL1 $O(k \cdot \log\log k)$ **21**

7.1 Introduction 21

7.2 The algorithm 21

7.3 Termination 22
7.4 Correctness 22
7.5 Theoretical bound 22

8 SEL2 $O(k \cdot 3^{\log^*(k)})$ **24**
8.1 Introduction 24
8.2 The algorithm 24
8.3 Termination 25
8.4 Correctness 25
8.5 Theoretical bound 26

9 Priority queues **30**
9.1 Introduction 30
9.2 The data structure 30
9.3 Termination 31
9.4 Correctness 31
9.5 Theoretical bounds 32
9.6 Consequences of splitting 33

10 SEL3 $O(k \cdot 2^{\log^*(k)})$ **34**
10.1 Introduction 34
10.2 The algorithm 34
10.3 Termination 35
10.4 Correctness 36
10.5 Theoretical bound 36

11 SEL4 $O(k)$ **41**
11.1 Introduction 41
11.2 The algorithm 41
11.3 Termination 43
11.4 Correctness 43
11.5 Theoretical bound 45

12 Examples **52**
12.1 Naive 52
12.2 SEL1 and SEL2 54
12.3 SEL3 and SEL4 57

13 SEL4 parameter optimization **65**
13.1 Introduction 65
13.2 Choice of Parameters 65

14 Experiment heaps **68**
14.1 Worst case 68
14.2 Random case 68

14.3 Best case 69

15 Priority Queue Trouble **71**

16 Experimental results **74**
16.1 System specification 74
16.2 Runtime 74
16.3 Comparisons 81
16.4 Accesses 85
16.5 Conclusion 89

17 Valgrind Results **91**
17.1 Worst case 92
17.2 Random case 97
17.3 Best case 103
17.4 Conclusion 109

18 Conclusion **111**

19 Theorems **112**

Bibliography **114**

Chapter 1

Introduction

Given a binary min-heap of size $n \gg k$, we want to select the k smallest elements. Since $n \gg k$ we will view this binary min-heap as being infinitely large. Here a binary min-heap is a heap ordered binary tree, where every node contains a value, and has two children. The children of a node have values higher than or equal to that of their parent. The min-heap was introduced by Williams [6]. An example of our problem is seen in Figure 1.1, where we have selected the five smallest elements.

Frederickson [5] describes four algorithms with increasingly better upper bounds, with the final algorithm *SEL4* having the optimal theoretical bound of

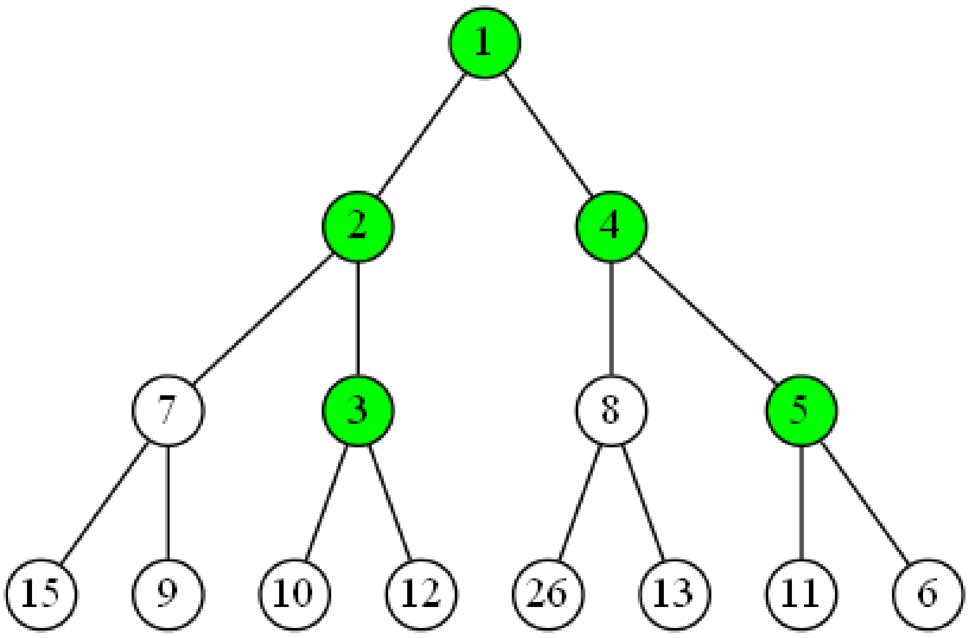

Figure 1.1: An infinitely large heap with the 5 smallest elements colored.

$O(k)$. In this thesis we have investigated the practicalities of these algorithms and assessed whether or not, especially the final optimal bound, works in practice. This we have done by implementing all the algorithms mentioned, and then measure their behaviours on different parameters such as, but not limited to: comparisons, accesses to the original heap and last level data cache misses.

Furthermore, because of how advanced some of the algorithms described by Frederickson in [5] are, another issue can be the understanding of them. Frederickson does provide some intuition by constructing the next algorithm through the addition of one or more paradigms to the previous. Thus building the algorithms iteratively. Despite Frederickson's efforts, understanding of the algorithms is not achieved without difficulty. We therefore try to give additional explanations, intuition and show how the algorithms work through examples.

Throughout this thesis we have also proven several unproven claims proposed by Frederickson [5], some claims we have not managed to prove, but for these we have tried to give some intuition into why they could hold.

Selection in a min-heap has 62 citations at the time of writing. Eppstein [4] uses it in an algorithm that finds the k shortest paths. Brodal [2] uses it in an algorithm that selects the k largest sums of subarrays in an array.

Chapter 2

Notation

Table 2.1: Table of Notation

$\mathcal{T}$	Our original infinitely large min-heap.
$\mathcal{H}_0$	The root of $\mathcal{T}$.
k	The number of small elements we are selecting.
r	Natural number, we use for representing subroutine sizes.
B	Clan size.
C	A clan.
C_i	Clan number i.
$os(C)$	The off-spring of clan C.
$pr(C)$	The poor-relation of clan C.
$\mathcal{H}$	A set of nodes from our infinite min-heap.
PQ	A priority queue.

Table 2.2: Table of Function Definitions

$$^{n}a = \underbrace{a^{a^{\cdot^{\cdot^{a}}}}}_{n}$$

$$\log^*(r) = \begin{cases} 1 & \text{if } r \leq 2 \\ 1 + \log^*(\lceil \log(r) \rceil) & \text{otherwise} \end{cases}$$

$$f(r) = \left\lfloor \left(\frac{\lceil \log(r) \rceil}{\log^*(r)}\right)^2 \right\rfloor$$

$$h_3(r) = \begin{cases} 1 & \text{if } r \leq 1 \\ h_3(\lfloor \log(r) \rfloor) \cdot \left\lceil \frac{r}{h_3(\lfloor \log(r) \rfloor)} \right\rceil & \text{otherwise} \end{cases}$$

$$h_4(r) = \begin{cases} 1 & \text{if } r \leq 1 \\ h_4(f(r)) \cdot \left\lceil \frac{r}{h_4(f(r))} \right\rceil & \text{otherwise} \end{cases}$$

$$A(r) = \prod_{i=1}^{2 \cdot \log^*(r)} (1 + \frac{4}{i^2})$$

$$B(r) = \frac{r}{3^{\log^*(r)} \cdot (r \cdot A(r) - \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot \lfloor \log(r) \rfloor \cdot A(\lfloor \log(r) \rfloor))}$$

$$C(r) = \frac{(\log^*(r))^2((2\log^*(r) - 1)^2 + 4)}{4\lceil \log^*(r) \rceil \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})}$$

Chapter 3

Naive Algorithm $O(k \cdot \log k)$

3.1 Introduction

The first and simplest algorithm for selecting the k smallest elements in a min heap uses a minimum priority queue holding heap nodes. It takes an integer k and a heap node $\mathcal{H}_0$ as arguments and returns the k smallest elements found in the tree rooted in $\mathcal{H}_0$. It is only briefly described in Frederickson's paper [5], but it is fairly easy to come up with. An example of how it works can be seen in Chapter 12. In algorithm SEL1, which improve the bound $O(k \cdot \log k)$ to $O(k \cdot \log \log k)$ and will be presented later, the Naive algorithm is used and is required to find the k smallest elements from a list of trees, represented by their roots. The single tree version simply calls the multiple tree version. In the multiple tree version we call the set of trees $\mathcal{H}_n$. It works as follows:

1. Initialize priority queue PQ with all elements from $\mathcal{H}_n$
2. Initialize result list R
3. k times do:
   a) $E_i := extractMin(PQ)$
   b) Add E_i to R
   c) Insert left and right child of E_i into PQ
4. Return R

3.2 The algorithm

We begin by initializing a priority queue PQ, where smaller elements have higher priority, and add the elements in $\mathcal{H}_n$ to PQ. $\mathcal{H}_n$ will only contain one element, namely $\mathcal{H}_0$ when called as the single tree version, but will maximally hold $O(k)$ elements when called through SEL1. Then we perform extractMin operations on PQ k times, and for each of these times we store the element we extracted as a part of the result. For every node extracted we then insert its children into the PQ. When k elements have been extracted we are done.

3.3 Termination

In order to argue that the algorithm terminates we first note that we make the assumption that we pick a priority queue for which the operations all terminate, given that the priority queue has a finite size. With this established our algorithm clearly terminates since we specifically call the extract-min operation k times. Since we only ever extract k elements, and every element has at most two children, the size of the priority queue will never exceed $O(k)$ elements, which means the size of the priority queue is indeed finite.

More precisely, if $\mathcal{H}_n$ holds s elements, then the maximum size of the priority queue will be $s + k$, because we k times extract one element and add two, effectively raising the queue size by one. Since $s = O(k)$, we have that the maximum priority queue size is $O(k)$.

3.4 Correctness

To achieve correctness the important invariant is that everything that will be inserted into our priority queue in the future has an ancestor in the priority queue, and every element extracted is smaller than all the elements we have not extracted. Since we start with root(s) of trees, it is easy to see that since we always insert both children of an extracted element we will not miss any nodes in the trees we are looking for minimum elements in. Additionally since we are working on a minimum priority queue it is clear that when we extract an element this element is the smallest in the priority queue at the time of extraction. Since we are selecting the smallest elements from a minimum heap, and we add the children of any extracted element, every time we extract an element this element will be the minimum element that was left. Which is why when he have extracted k elements we will have extracted the k smallest.

3.5 Theoretical bound

By the argument in Section 3.3 the priority queue will have size $O(k)$. This means the extractMin operation will achieve a bound of $O(\log(k))$ for a suitable choice of data structure to represent the priority queue (a minimum heap). Since we call extractMin k times we achieve a total bound of $O(k \cdot \log(k))$.

Chapter 4

Framework

4.1 Introduction

The four algorithms for finding the k smallest elements a min heap presented by Frederickson in [5] gradually improve upon the naive algorithm presented in Chapter 3. The paper presents multiple ideas that when used together provide the algorithm SEL4, which has an optimal linear theoretical bound.

These ideas also provide entry points for tweaking a practical implementation of several of the algorithms. In this chapter the ideas will be presented in the order they appear in [5], and some will be supplemented with a discussion of how the runtime is affected by changing parameters in the implementation.

4.2 Partitioning

Instead of directly finding the elements requested, we search for an element x with a rank that is greater than or equal to the rank of k'th smallest element. We then traverse the heap, adding all elements less than or equal to x to a list. This can result in more than k elements. To fix this the list can be partitioned using a standard partition algorithm, so that the k smallest elements are at the beginning of the list and the rest of the elements in the list are discarded.

See Chapter 5 for details. All four algorithms described by Frederickson in [5] use this technique, see Chapters 7, 8, 10, and 11.

4.3 Clans

An element of appropriate rank can be used to find the k smallest elements, see above section. This gives some freedom as to how elements are handled.

The four algorithms, SEL1, SEL2, SEL4, and SEL4, still use priority queues internally to keep track of which elements have been found. To reduce the time used by the priority queue we group elements into *clans*, and instead put the clans into the priority queues. For instance, if we let a clan contain $\sqrt{k}$

elements, we can reduce the number of extractMin and insert operations by a factor $\sqrt{k}$ and still obtain k elements.

The value of a clan is called a *representative*, which is used in the priority queue for ordering, and it is the largest member of the clan. Thus if the algorithm runs extractMin $\frac{k}{clansize}$ times, a total of k elements will have been extracted through the clans. Since the clans are ordered by representative and the lowest ranked clans are extracted first, the representative of the last clan must be larger than or equal to all other elements in extracted clans, and thus have a rank of at least k.

When a clan is extracted from the priority queue, we create new clans. The way this is done is relevant to the time bounds, but not to correctness. In regard to time bounds, if a clan is to be made with elements from a very large set of elements it follows that the smallest elements cannot be found fast: If we are to find the k smallest among 2^k unordered elements, we cannot hope to do so in $O(k)$ time. Therefore the four algorithms supply ways to split the sets of possible clan members into smaller sets. The method presented by SEL1 and SEL2 is considerably more complicated than the one presented by SEL3 and SEL4, but they both achieve the exact same thing: Splitting the set into a smaller size.

See Chapter 6 for details on clans and Chapters 7, 8, 10, and 11 for their use.

Tweaking

Clan size is relevant to time bounds, but not to correctness (within reasonable limits). If the clan size is set to 1, the algorithm essentially degenerates to the naive algorithm, with some overhead. Conversely, if the clan size is set to k, meaning we only need a single clan to get an element of at least rank k, we simply push the entire problem to whatever subroutine we're calling. For SEL1, this means simply asking the naive algorithm to solve the entire problem. For the recursive algorithms, continuously requesting clans of size k obviously breaks the termination proof (see reasonable limits above).

4.4 Recursion

We have changed the problem of finding the k smallest elements in $\mathcal{H}_0$ to finding the r smallest in some subset of $\mathcal{H}_0$ multiple times. If the time bounds can be improved by solving subproblems in the main algorithm, and the problem solved by the subroutine is very similar to the main problem, the next logical step is to apply our subroutine recursively: If the top layer is asked for k elements and it runs the subroutine asking for $f(k)$ elements for some definition of f, the next layer can call itself asking for $f(f(k))$ elements. We refer to the rank of the element requested at any level as r, with $r = k$ at the top level, and $r = 1$ as the base case for the recursion.

See SEL2, SEL3, and SEL4 in Chapters 8, 10, and 11 respectively for usage of recursion.

Tweaking

The recursion depth is an obvious place to tweak the algorithms. Essentially, the naive algorithm, SEL1, and SEL2 behave the same way. The naive algorithm has a recursion depth of 0, solving all of its work up front. SEL1 has a recursion depth of 1, cutting its work up once and then letting the naive algorithm solve it. SEL2 recurses until it reaches its base case of $r = 1$. Changing the base case to something larger than 1 could be a way to improve SEL2.

Unfortunately, in the case of SEL1 and SEL2, neither perform better than the naive algorithm in our runtime tests, so having a recursion depth of 0 is best. With a lot of extrapolation, SEL1 could possibly beat the naive algorithm if used on more data than we were able to generate. See Chapter 16 for details.

SEL4 has a chapter dedicated to tweaking the base case and will not be discussed here. See Chapter 13. SEL3 might benefit from some of the same optimizations, but since SEL4 is better than SEL3 in all measurements, it has not been examined.

4.5 Priority queue preservation

In the introduced recursion, elements are considered multiple times. For instance, if we are at layer ℓ and extract a clan from our priority queue, that clan was created at layer $\ell + 1$. When we return a clan from our level, we throw away our priority queue, and thus work done at this and lower levels will have to be redone if needed at a later time: The same elements that were already found previously might be found in subsequent calls.

The idea is then to try to reuse information from lower levels of the recursion. We want a guarantee that any element is extracted at most once per level.

This is done by adding the priority queue used at level $\ell + 1$ to the clan returned to level ℓ. When level ℓ later calls the subroutine for $\ell + 1$, it passes along the priority queue. This means work already done is essentially preserved between calls to the same level. An important property is that we need to split a priority queue in sublinear time to stay within time bounds.

See Chapter 9 for details on priority queue splitting.

4.6 Category

When we extract a clan from a priority queue, we create two new clans (see Chapters 6, and 9). In a rather obscure way of improving the time bounds, the last algorithm, SEL4, presents the idea of sometimes not splitting. It works as follows: When a clan C is first inserted into the priority queue, give it a category of 1. When C is extracted, if the category is below some threshold (based on r in SEL4), do not split its priority queue, but simply recurse on it. The resulting clan C' is given the category $cat(C) + 1$ and then inserted into our priority queue. If $cat(C) > threshold$, we split and recurse on both. The resulting clans are both given a category of 1. Since the threshold for splitting is proportional to r, deeper in the recursion we split more often.

Tweaking

Splitting threshold is relevant to time bounds, but not to correctness: Setting the threshold to 1, meaning we always split, we are no different from SEL3. The other extreme of never splitting also loses the time bound, but is also still correct: Never splitting means all layers of recursion will only ever have one clan in their priority queues, except the the layer where $r = 1$, where we get $O(k)$ clans, essentially the naive algorithm.

In our tests we found that changing the threshold had some effect, but changing the base case had a much greater effect, see Chapter 13.

Chapter 5

Select

5.1 Introduction

The four algorithms presented by Frederickson in [5] all use a selection algorithm for finding the k smallest elements in a list.

Blum presents an algorithm called PICK in [1], which finds the i'th smallest number in a list. In Chapter 9.3 in Corman [3], SELECT is described. SELECT builds on PICK and additionally partitions the list such that all elements before the k'th are less than or equal, and all later elements are greater than or equal. Select presented here is a variation of SELECT.

5.2 The algorithm

Select takes arguments k and A, k being the number of elements desired, A being the list of elements. It then looks at A in its entirety and calls the subroutine findMedian on A. It partitions A around the median. If the median is at a position different to k, repeat on a subset of A. The partitioning is done in place. It looks as follows:

```
1. $low := 0$
2. $high := length(A)$
3. While ($low < high$) do:
    a) $index := low$
    b) $findMedian(A, low, high)$
    c) $swap(A[low], A[high - 1])$
    d) $i := low - 1$
    e) $j := low$
    f) While ($j < high - 1$) do:
        i. If $(value(A[j]) \leq value(A[high-1])$:
            A. $i := i + 1$
            B. $swap(A[i], A[j])$
        ii. $j := j + 1$
    g) $swap(A[i+1), A[high-1]$
    h) $index := i + 1$
    i) If $(index < k)$:
        i. $low := index + 1$
    j) Else if $(k < index)$:
        i. $high := index$
    k) Else:
        i. Return
```

The findMedian subroutine takes A, a list of elements, and *low* and *high*, the range in which a median is to be found. It works by finding exact medians of fixed sized subsets of A, and combining these medians to create approximate medians of large parts of A, until eventually it condenses to a single value. This final value is not the exact median, but a good approximation, see [3].

In the following example the fixed size for exact medians is 9. The median it finds will be swapped to the beginning on the part of A it operates on. It looks as follows:

```
1. $remaining := high - low$
2. While $(remaining > 0)$ do:
    a) $index := low$
    b) $extra := remaining\%9$
    c) $i := 0$
    d) While $(i < remaining)$
        i. If $(i + 9 < remainder)$:
            A. $sort(A[i + low], A[i + low + 9])$
            B. $swap(A[index], A[low + i + 4])$
        ii. Else:
            A. $sort(A[i + low], A[low + remaining])$
            B. $swap(A[index], A[low + i + \frac{extra-1}{2}])$
        iii. $i := i + 9$
        iv. $index := index + 1$
    e) $remaining := \frac{remaining}{9}$
    f) If ($extra \neq 0$):
        i. $remaining := remaining + 1$
```

Note that our pseudo code finds the median of 9 elements, which is what the implementation uses. The algorithm described by Cormen in [3] finds the median of 5 elements, but is otherwise the same.

5.3 Further reading

The details of Select are not relevant to SEL1, SEL2, SEL3, and SEL4, and will not be discussed. The particular Select used in the implementation could be replaced by any other Select, so long as it runs in linear time. For a detailed overview of Select, see Chapter 9.3 in Cormen [3].

Chapter 6

Clans

6.1 Clan

For the algorithm in the next chapters we will use something Frederickson [5] calls clans. A clan C contains some amount of nodes from the original heap $\mathcal{T}$, a representative of the clan $rep(C)$, a set of nodes called the off-spring $os(C)$, and a set of nodes called the poor-relation $pr(C)$.

When we construct a clan C, we construct it from some forest A. Here the off-spring is then defined as the children of the nodes in C which are not themselves in C. The poor-relation is defined as the set of nodes in A, which are not in C. Lastly the representative of the clan C is defined as the largest value of any node in C.

The members of C are found from the roots in A, with each algorithm specifying its own way of doing so.

Off-spring and poor-relation are found in the following way, with the function createClan taking C and A:

1. All nodes in C and A start uncolored. Color all nodes in C.

2. For each node in children of nodes in C, if the node is uncolored, add it to $os(C)$.

3. For each node in A, if the node is uncolored, add it to $pr(C)$.

4. Remove color from nodes in C.

This ensures all nodes are uncolored when the algorithm terminates. Because all nodes in C are colored first, nodes in C cannot appear in $os(C)$ or $pr(C)$.

The construction of a clan given A and C is linear in the size of A and C, since the four sets are each iterated a constant number of times. This is asymptotically optimal.

6.2 Clan-3

Clan-3, used by SEL3, is a simpler construct than the previously defined clan. Clan-3 contains a representative and a heap as defined in chapter 9. The members of the clan are implicit in that they are smaller than or equal to the representative, and not included in the heap. The heap is a substitute for the previously used off-spring and poor-relation: the heap itself can be split into two, removing the need for a distinction between the different kinds of nodes not included in the clan.

6.3 Clan-4

Clan-4, used by SEL4, is identical to clan-3 defined above, except that it contains an additional integer value called category, which SEL4 uses.

Chapter 7

SEL1 $O(k \cdot \log\log k)$

7.1 Introduction

For our second algorithm we have implemented the first described algorithm by Frederickson [5]. We have illustrated how this algorithm works in Chapter 12.

7.2 The algorithm

SEL1 is an algorithm that uses clans of $\lfloor \log(k) \rfloor$ size to find an element with rank between k and $2k$. The problem of finding the clans in subtrees from $\mathcal{H}_0$ is solved by the naive algorithm. SEL1 takes two arguments, an integer k and $\mathcal{H}_0$, then calls Select (See Chapter 5) on the last representative found. It works as follows:

```
1. $clanSize := \lfloor \log(k) \rfloor$
2. $C_0 := createClan(naive(clanSize, \mathcal{H}_0))$
3. Initialize heap $PQ$ with $C_0$
4. $highest := -\inf$
5. $limit := \lceil \frac{k}{clanSize} \rceil$
6. $limit$ times do:
    a) $C_i := extractMin(PQ)$
    b) $highest := rep(C_i)$
    c) If $size(os(C_i)) > 0$:
        i. Insert $createClan(naive(clanSize, os(C_i)))$ into $PQ$
    d) If $size(pr(C_i)) > 0$:
```

i. Insert $createClan(naive(clanSize, pr(C_i)))$ into PQ

7. Do a breadth first search and Select (See 5) using *highest* according to chapter 5

7.3 Termination

This algorithm clearly terminates since we have already established that the Naive algorithm terminates, and we only do a fixed number of iterations. Like for the Naive algorithm the size of the priority queue PQ is finite, because we first add one clan made from the element the algorithm is called with. Then we *limit* times call extract and two clans made from the off-spring and poor-relation of the extracted clan. This means we *limit* times at most grow the size of the priority queue by one.

7.4 Correctness

For proof of correctness we first note that every clan is disjoint. Additionally since we extract $\left\lceil \frac{k}{\lfloor \log(k) \rfloor} \right\rceil$ clans from our priority queue, we have that the representative for the last extracted clan is larger than the values for all nodes in all our previously extracted clans, and since we have extracted $\left\lceil \frac{k}{\lfloor \log(k) \rfloor} \right\rceil$ clans of size $\lfloor \log(k) \rfloor$ we have that this last representative must be at least as large as the kth smallest element, and also it must be no larger than the $2 \cdot k$th smallest element, because for every clan we extract we make at most two new clans. Since we extract $\left\lceil \frac{k}{\lfloor \log(k) \rfloor} \right\rceil$ clans we make at most:

$$2 \cdot (\left\lceil \frac{k}{\lfloor \log(k) \rfloor} \right\rceil) \leq 2 \cdot (\frac{k}{\lfloor \log(k) \rfloor} + 1) \tag{7.1}$$

clans, and since every clan has size $\lfloor \log(k) \rfloor$, we have at most:

$$2 \cdot (\frac{k}{\lfloor \log(k) \rfloor} + 1) \cdot \lfloor \log(k) \rfloor = 2 \cdot k + 2 \cdot \lfloor \log(k) \rfloor \tag{7.2}$$

elements in all of these clans in total. Since our last extracted clan has a representative larger than or equal to all other elements in previously extracted clans we have that there can be at most $2 \cdot k$ of these elements less than or equal to our last representative. We remove the $2 \cdot \lfloor \log(k) \rfloor$ term because these are the elements created by our last extracted clan.

7.5 Theoretical bound

The bottleneck in our algorithm is the part where we call our Naive algorithm for every one of our $\left\lceil \frac{k}{\lfloor \log(k) \rfloor} \right\rceil$ iterations. Since our Naive algorithm runs in

$c' \cdot r \cdot \log(r)$, where r is the number of smallest elements we want to find and c' is some constant, we get in total. We will assume that $k > 1$, since for $k = 1$ the problem is easy, we get:

$$\begin{aligned}
c' \cdot \lfloor\log(k)\rfloor \cdot \log(\lfloor\log(k)\rfloor) \cdot \left\lceil \frac{k}{\lfloor\log(k)\rfloor} \right\rceil &\leq c' \cdot \lfloor\log(k)\rfloor \cdot \log(\lfloor\log(k)\rfloor) \cdot (\frac{k}{\lfloor\log(k)\rfloor} + 1) \\
&\leq c' \cdot \log(\lfloor\log(k)\rfloor) \cdot k + c' \cdot \log(\lfloor\log(k)\rfloor) \cdot \lfloor\log(k)\rfloor \\
&\leq 2c' \cdot \log(\lfloor\log(k)\rfloor) \cdot k \\
&\leq 2c' \cdot \log(\log(k)) \cdot k \\
&= O(k \cdot \log(\log(k)))
\end{aligned}$$

which is our asymptotic bound. Furthermore we need to show that we do not use too long in our last step of the algorithm where we call a k-select algorithm on lists. Referring to Chapter 5 we can see that this step will only take $O(k)$ time on a list of size maximum $2 \cdot k$. We get the asymptotic bound we were looking for.

Chapter 8

SEL2 $O(k \cdot 3^{\log^*(k)})$

8.1 Introduction

We have implemented the second algorithm described by Frederickson [5] called SEL2. We have illustrated how this algorithm works in Chapter 12.

8.2 The algorithm

SEL2 is a recursive algorithm that uses clans of $\lfloor \log(k) \rfloor$ size to find an element with rank between k and $2k$. In SEL1 the problem of finding the members of each clan was solved using the naive algorithm, while here we use SEL2 itself. SEL2 takes two arguments, r and $\mathcal{H}$.

Note that Frederickson [5] also mentions RSEL2, but since that and SEL2 do virtually the same, RSEL2 was ignored.

We've changed the name of k to r, as it also refers to the elements needed to solve sub problems. It works as follows:

1. If $r = 1$, return the smallest element in $\mathcal{H}$.
2. Partition $\mathcal{H}$ into subsets $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_s$ with $|\mathcal{H}_i| \leq 2\lfloor \log(r) \rfloor$.
3. Let $C_i := createClan(SEL2(\mathcal{H}_i, \lfloor \log(r) \rfloor), \mathcal{H}_i)$.
4. Initialize heap PQ with every C_i.
5. Let $limit := \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$.
6. $limit$ times do:
    a) Let $C_j := PQ.extractMin()$.
    b) Let $C_i := createClan(RSEL2(os(C_j), \lfloor \log(r) \rfloor), C_j)$.
    c) Let $C_{i+1} := createClan(RSEL2(pr(C_j), \lfloor \log(r) \rfloor), C_j)$.

   d) PQ.insert(C_i).
   e) PQ.insert(C_{i+1}).

7. Let C_j be the last extracted element from step 6.a.

8. Add the elements in $\mathcal{H}$ less than or equal to $rep(C_j)$ to list L.

9. Select the r smallest elements in L and return them.

8.3 Termination

The recursive call in 3, 6.b, and 6.c each happen a bounded number of times, and each recursive call has a smaller r with a base case of $r = 1$, so the algorithm terminates. Additionally by argumentation very similar to that of the Naive algorithm and SEL1, the priority queue is finite in size.

8.4 Correctness

The proof of correctness is an induction proof and we note that every clan is disjoint.

The base case is $r = 1$ in which the algorithm returns the smallest node in $\mathcal{H}$. This is trivially correct.

For the induction case we assume that the algorithm works for $r' < r$, which means that all the recursive calls are correct. The steps 7, 8, 9 require the rank of $r \leq rep(C_j)$, so the algorithm is correct if this is true.

Each clan extracted from PQ has size $\lfloor \log(r) \rfloor$ and in total $\left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$ clans are extracted. That means a minimum of

$$\lfloor \log(r) \rfloor \cdot \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \geq \lfloor \log(r) \rfloor \cdot \frac{r}{\lfloor \log(r) \rfloor} = r$$

nodes have been extracted, and the representative of the last clan extracted is greater than or equal to the nodes found in all the extracted clans. Thus the last representative has a rank of at least r.

The above requires that no clans have nodes in common. To prove that this is the case, we note that the clans give two sets of nodes from which new clans can be made, namely off-spring and poor-relation. Off-spring specifically does not include nodes also included in the clan it belongs to, and poor-relation is nodes in $\mathcal{H}$ not included in the clan. Neither off-spring nor poor-relation contain nodes that are ancestors of each other. This means that once a clan is created, its nodes will never be included in another clan.

8.5 Theoretical bound

First we note for a call to SEL2 with parameters r and $\mathcal{H}$, $\mathcal{H}$ will maximum have size $2 \cdot r$. This means we will maximally create $\frac{2 \cdot r}{2 \cdot \lfloor \log(r) \rfloor} = \frac{r}{\lfloor \log(r) \rfloor}$ clans in step 3. Furthermore we will at most create $2 \cdot \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$ clans inside step 6, since we $\left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$ times may create up to two clans. Which means we create at most $3 \cdot \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$ clans. Performing $\left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$ extract-min operations and $2 \cdot \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$ insert operations will take $O(r)$ time since the size of the priority queue is $O(r)$. Additionally step 8 and 9, where we find and select the r smallest elements in L and return them, also runs in time $O(r)$. This we can show by making an upper bound on the rank of the last extracted element. Since we maximally create $3 \cdot \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil$ clans as discussed earlier, the rank of the representative of our last extracted clan from the priority queue in step 6.a must be less than or equal to:

$$3 \cdot \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot \lfloor \log(r) \rfloor \leq 3 \cdot r + 3 \cdot \lfloor \log(r) \rfloor$$

Since we create two clans of size $\lfloor \log(r) \rfloor$ from the clan we have extracted as the last we can subtract $2 \cdot \lfloor \log(r) \rfloor$ from the result, and get that the maximum possible rank of our element is $3 \cdot r + \lfloor \log(r) \rfloor = O(r)$. Which means that step 9 can be solved in $O(r)$ time by [1]. All this together gives us the recursion:

$$\begin{aligned} T(1) &\leq c \\ T(r) &\leq cr + 3 \cdot \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot T(\lfloor \log(r) \rfloor) \end{aligned} \tag{8.1}$$

We would now like to prove that $T(r) \leq c' \cdot r \cdot 3^{\log^*(r)} \cdot A(r)$ using induction, where we let $A(r) = \prod_{i=1}^{2 \cdot \log^*(r)} (1 + \frac{4}{i^2})$. The choice of $A(r)$ is inspired by Frederickson's [5] proof of SEL4. Our base cases here are $r = 1$ and $r = 2$. For $r = 1$ we get:

$$\begin{aligned} c &\leq c' \cdot 1 \cdot 3^{\log^*(1)} \cdot \prod_{i=1}^{2 \cdot \log^*(1)} (1 + \frac{1}{i^2}) \\ &\leq c' \cdot 1 \cdot 3 \cdot 5 \cdot 2 \\ &\leq 30 \cdot c' \end{aligned}$$

So for the base case $r = 1$ we should set $c' \geq \frac{1}{30} \cdot c$. Now for $r = 2$. We start out by calculating the left side:

$$\begin{aligned} T(2) &\leq 2c + 3 \cdot \left\lceil \frac{2}{\lfloor \log(2) \rfloor} \right\rceil \cdot T(\lfloor \log(2) \rfloor) \\ &\leq 2c + 3 \cdot \left\lceil \frac{2}{1} \right\rceil \cdot c \\ &= 8c \end{aligned}$$

Now for the right side:

$$\begin{aligned} &c' \cdot 2 \cdot 3^{\log^*(2)} \cdot \prod_{i=1}^{2 \cdot \log^*(2)} (1 + \frac{4}{i^2}) \\ &= c' \cdot 2 \cdot 3 \cdot 10 \\ &= 60c' \end{aligned}$$

Which means we need $8c \leq 60c'$, which gives us that we should set $\frac{8}{60}c = \frac{2}{15}c \leq c'$ for the second base case to hold. Now that we have the two base cases we want to show the case $r > 2$ using induction. For the inductive case assume the claim is true for $r' < r$. We will here use $A(r)$, rather than substituting it with what it equals. We show:

$$\begin{aligned} T(r) &\leq cr + 3\left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot T(\lfloor \log(r) \rfloor) && \text{(Eq. 8.1)} \\ &\leq cr + 3\left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot c' \lfloor \log(r) \rfloor 3^{\log^*(\lfloor \log(r) \rfloor)} \cdot A(\lfloor \log(r) \rfloor) && \text{(Ind. hypo.)} \\ &\leq cr + \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot c' \lfloor \log(r) \rfloor 3^{\log^*(\lfloor \log(r) \rfloor)+1} \cdot A(\lfloor \log(r) \rfloor) \\ &\leq cr + \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot c' \lfloor \log(r) \rfloor 3^{\log^*(r)} \cdot A(\lfloor \log(r) \rfloor) && \text{(Def. } \log^*\text{)} \end{aligned}$$

We then need:

$$cr + \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot c' \lfloor \log(r) \rfloor 3^{\log^*(r)} \cdot A(\lfloor \log(r) \rfloor) \leq c' \cdot r \cdot 3^{\log^*(r)} \cdot A(r)$$

Moving around we get:

$$cr \leq c' \cdot 3^{\log^*(r)} \cdot (r \cdot A(r) - \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot \lfloor \log(r) \rfloor \cdot A(\lfloor \log(r) \rfloor))$$

Isolating c' we obtain:

$$\frac{r}{3^{\log^*(r)} \cdot (r \cdot A(r) - \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot \lfloor \log(r) \rfloor \cdot A(\lfloor \log(r) \rfloor))} \cdot c \leq c'$$

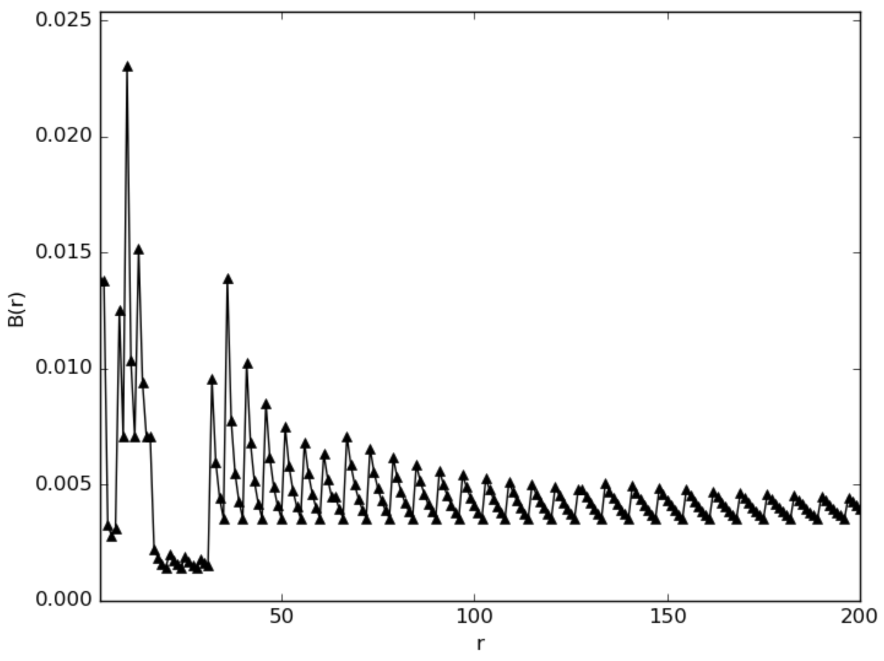

Figure 8.1: B(r) over r. With $r \in [3, 200]$.

If the expression $\frac{r}{3^{\log^*(r)} \cdot (r \cdot A(r) - \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot \lfloor \log(r) \rfloor)}$ converges we can pick c' such that our proof by induction holds. Which would mean we have shown that $T(r)$ is $O(r \cdot 3^{\log^*(r)} \cdot A(r))$, which for the $A(r)$ we have chosen is $O(r \cdot 3^{\log^*(r)})$, because our choice of $A(r)$ converges, which can be seen using Theorem 19.0.1 and Theorem 19.0.2. If we call our long expression for $B(r)$ we get:

$$B(r) = \frac{r}{3^{\log^*(r)} \cdot (r \cdot A(r) - \left\lceil \frac{r}{\lfloor \log(r) \rfloor} \right\rceil \cdot \lfloor \log(r) \rfloor \cdot A(\lfloor \log(r) \rfloor))}$$

And what remains is to show that this expression converges. We have tried for long to show this, but have been unable to. We should note that Frederickson refrains from showing this in [5], and just states that the recursion is bounded by $O(r \cdot 3^{\log^*(r)})$ on page 204. Looking at Figure 8.1 and Figure 8.2 there seems to be an indication that $B(r)$ does converge, and it looks like the value never exceeds $0.025 = \frac{1}{40}$. So if it does converge as indicated by the two figures, we could set $\frac{1}{40} \cdot c \leq c'$.

Lastly as argued earlier the representative of the last extracted clan will be $O(r)$, which means that given convergence of $B(r)$, we obtain an asymptotic bound of $O(r \cdot 3^{\log^*(r)})$.

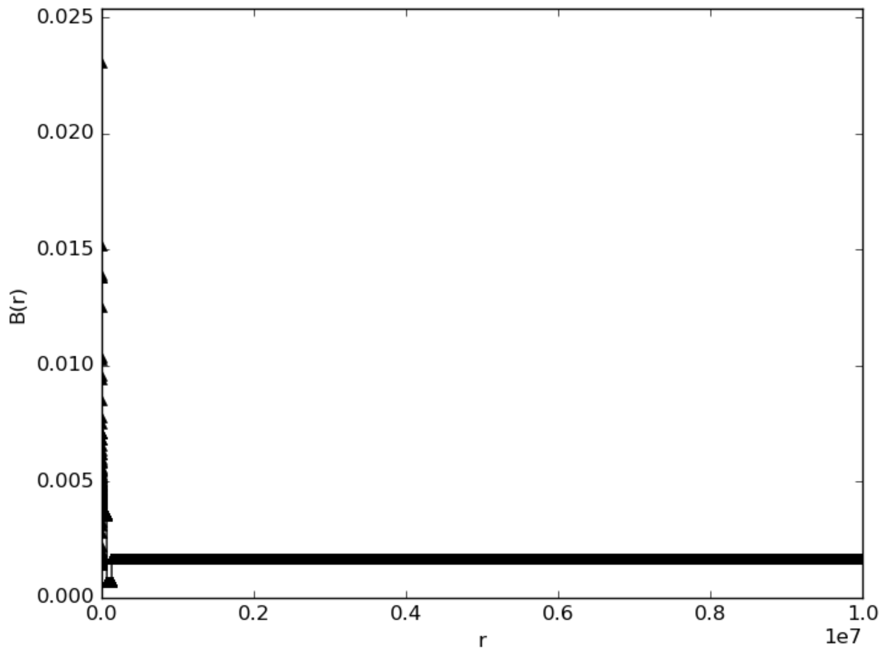

Figure 8.2: B(r) over r. With $r \in [3, 10000000]$.

Chapter 9

Priority queues

9.1 Introduction

In this chapter we describe the splittable priority queue used in SEL3 and SEL4. We will also prove they work and have a theoretical bound of $O(\log n)$ on insert and extract minimum, and $O(1)$ on split, which are the only operations we need to support for SEL3 and SEL4.

9.2 The data structure

The splittable priority queue has two priority queues internally, a and b, which are array based priority queues. The implementation uses `std::priority_queue` on top of `std::vector` from the Standard Template Library (STL).

Insert

Insert puts the element in the smallest of a and b. When the queue is initially empty, the balance between a and b can only be made at most one worse when they are the same size, and better in the case where one is smaller than the other.

Extract minimum

Extract minimum returns the minimum element between a and b. If one is empty then the minimum element of the other is returned. Since we can potentially have all the smallest elements a, calling extract minimum repeatedly can skew the balance between the priority queues.

Split

Split works by simply replacing b with an empty priority queue, and returning a new splittable priority queue containing the old b and an empty priority queue. After a split, both priority queues will have one empty internal priority queue.

9.3 Termination

Insert

Insert compares the size between a and b and inserts into the smallest of the two. Both operations are guaranteed to terminate in the implementation provided by the STL, so insert on the splittable priority queue terminate as well.

Extract minimum

Extract minimum looks at the smallest element in a and b and extracts the smallest between the two. Both looking at the smallest and extracting it are guaranteed to terminate in the STL.

Split

Split simply creates a new splittable priority queue and moves b to the new priority queue. It clearly terminates.

9.4 Correctness

To prove that the splittable priority queue functions correctly as a priority queue we'll look at the state of the priority queue before and after each operation. We'll show that if the priority queue is in a correct state before an operation, it will always be in a correct state after an operation.

Insert

Insert simply puts the element in one of the internal priority queues. Before insert the sum of the size of a and b is n. After we insert, one of the priority queues will be one larger, giving the sum the size of $n + 1$. a and b, being priority queues, will still have their smallest element at the top.

Extract minimum

Since we check which of the priority queues has the smallest element, we guarantee to return the smallest. When we extract the minimum element from the priority queue that has the smallest element, that priority queue will be one smaller, and still maintain heap order.

Split

The success criteria for split is that we end up with two splittable queues which don't contain the same elements. This is trivial, since we keep a in the splitting priority queue and give b to the new priority queue. No elements appear in both priority queues, since insert only ever inserts in one of a or b. There is no guarantee that one of the resulting priority queues won't contain all the elements.

9.5 Theoretical bounds

Before we show the complexity we note that the balance between a and b is irrelevant. For each operation we will also show why.

Insert

The worst case time complexity of insert into the internal priority queues is $O(\log n)$ with n being the size of the specific priority queue. The size of the priority queue can be checked in $O(1)$, so checking which of the two is smallest is also $O(1)$ and inserting into the smallest is $O(\log n)$, with n being the size of the smallest priority queue. So insert into the splittable priority queue is dominated by $O(\log n)$.

Extract minimum

Extract minimum includes looking at the top element of a and b, which is $O(1)$ and extracting the top element from the one containing the smallest, which is $O(\log n)$, with n being the size of that particular priority queue.

Note that one priority queue may contain all elements. However, this does not change the complexity, since in a splittable priority queue with m elements, if a and b are the same size, they will each contain $\frac{m}{2}$ elements. On the other hand, if one contains all elements, it will have m elements. Since the extract minimum takes logarithmic time, we find that the real difference between them is

$$\log(m) - \log(\frac{m}{2}) = \log(m) - (\log(m) - 1) = 1$$

1 is insignificant and thus extract minimum is dominated by $O(\log m)$.

Split

Split is trivially $O(1)$, as we just move the priority queue b from one splittable priority queue to another.

9.6 Consequences of splitting

Suppose we have a splittable priority queue SPQ with n elements in even portions in a and b. If we split it, we end up with two SPQ and SPQ', each with one internal priority queue containing $\frac{n}{2}$ elements. If we split SPQ again and obtain SPQ'', we'll get a splittable priority queue with 0 elements.

In the practical use of the splittable priority queue in SEL3 and SEL4, this is not a problem. In SEL3, when a priority queue with n elements is split, it is given to a subroutine that calls extract minimum $\frac{n}{2}$ times and insert n times. This pushes the size back up to n, and ensures that the internal priority queues are balanced and ready to be split again.

In SEL4, when an element is extracted, either one or two elements are inserted. This still pushes back the balance, however. If $\frac{n}{2}$ elements are in the queue to begin with, all of them in a, then extracting $\frac{n}{2}$ and inserting $\frac{n}{2}$ will give both a and b the size of $\frac{n}{4}$. When two elements are inserted, it looks the same way as for SEL3.

Chapter 10

SEL3 $O(k \cdot 2^{\log^*(k)})$

10.1 Introduction

For our fourth algorithm we have implemented the third discussed algorithm in [5]. We have illustrated how this algorithm works in Chapter 12.

10.2 The algorithm

We make use of *clan-3* (referred to simply as *clan* in this chapter) described in chapter 6.2, as well as the priority queues described in chapter 9.

SEL3 defines the recursive subroutine RSEL3, which takes $\mathcal{H}$ and r and returns C. $\mathcal{H}$ is a heap of clans, r is size of C, and C is the clan returned. Initially SEL3 calls RSEL3 with arguments nil and k. When RSEL3 returns a clan C, SEL3 extracts the representative of C and uses a breadth first search and the partition algorithm described in chapter 5 to find the k smallest elements.

RSEL3 works as follows:

```
1. If $r = 1$ :
    a) If $H = nil$ :
        i. $root := min(\mathcal{H}_0)$
        ii. Let $\mathcal{H}$ be a priority queue containing
            $leftChild(root), rightChild(root)$
        iii. return $createClan(root, \mathcal{H})$
    b) $x := extractMin(H)$
    c) Add children of $x$ to $\mathcal{H}$
    d) return $createClan(rep(x), \mathcal{H})$
2. if $H = nil$ :
    a) $C := RSEL3(\mathcal{H}, \lfloor \log(r) \rfloor)$
    b) Let $\mathcal{H}$ be a priority queue containing $C$
3. $limit := \left\lceil \frac{r}{h_3(\lfloor \log(r) \rfloor)} \right\rceil$
4. $limit$ times do:
    a) $C := extractMin(\mathcal{H})$
    b) Split $heap(C)$ into $\mathcal{H}_1$ and $\mathcal{H}_2$
    c) $C_1 := RSEL3(\mathcal{H}_1, \lfloor \log(r) \rfloor)$
    d) $C_2 := RSEL3(\mathcal{H}_2, \lfloor \log(r) \rfloor)$
    e) Add $C_1$ and $C_2$ to $\mathcal{H}$
5. return last $C$ extracted from $\mathcal{H}$
```

We should note that at step 3, [5] page 206 says we should set:

$$limit := \frac{h_3(r)}{h_3(\lfloor \log(r) \rfloor)} \tag{10.1}$$

But since we are in the case $r > 1$ and $h_3(r)$ is as it is in Table 2.2 we can use this to simplify the expression:

$$\frac{h_3(r)}{h_3(\lfloor \log(r) \rfloor)} = \frac{h_3(\lfloor \log(r) \rfloor) \cdot \left\lceil \frac{r}{h_3(\lfloor \log(r) \rfloor)} \right\rceil}{h_3(\lfloor \log(r) \rfloor)} = \left\lceil \frac{r}{h_3(\lfloor \log(r) \rfloor)} \right\rceil$$

So instead of setting $limit$ to $\frac{h_3(r)}{h(\lfloor \log(r) \rfloor)}$ as stated in [5] page 206, we set

$$limit := \left\lceil \frac{r}{h_3(\lfloor \log(r) \rfloor)} \right\rceil$$

which is a simpler equivalent in step 3. Additionally this also reassures us that $limit \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.

10.3 Termination

Having that *extractMin* and *Insert* terminates, it holds trivially that SEL3 terminates because we always work on smaller sizes in our recursive calls, and we run the for loop a fixed number of times.

10.4 Correctness

In order to argue the correctness of SEL3 it is easy to see that we just need to argue that the final element returned from RSEL3 needs to have a rank higher than or equal to the k we called it with. This is because the breadth first search and Select algorithm mentioned earlier (See Chapter 5) will then be able to find the kth smallest element.

First we note that any element extracted from a heap at level ℓ of our recursion is extracted at most once. This means each clan generated at level ℓ contains distinct elements, meaning any pair of clans at level ℓ are disjoint.

In order to argue correctness we will show that the rank of the returned element is $h_3(k)$. We do this using an inductive argument. For our base case we assume $r = 1$, which clearly holds. We then make the induction hypothesis that for $r' < r$ it holds that the returned element has rank $h_3(r')$. We now consider the case where $r > 1$. Since every clan created and extracted at this level of our recursion has size $h_3(\lfloor \log(r) \rfloor)$ due to the induction hypothesis, and that additionally these clans are all disjoint per the earlier argument and adding the fact that the rank of our last extracted clan C_{last} is higher than the rank of all the ranks of all the previous extracted clans at this level of our recursion we get the result:

$$rank(C_{last}) = limit \cdot h_3(\lfloor \log(r) \rfloor)$$

If we here use the definition of limit for RSEL3 from [5], which we see in equation 10.1 we then obtain:

$$rank(C_{last}) = \frac{h_3(r)}{h_3(\lfloor \log(r) \rfloor)} \cdot h_3(\lfloor \log(r) \rfloor) = h_3(r)$$

This shows that the rank of the clan we return is $h_3(r)$. Then since we have for $r > 1$:

$$h_3(r) = \left\lceil \frac{r}{h_3(\lfloor \log(r) \rfloor)} \right\rceil \cdot h_3(\lfloor \log(r) \rfloor) \geq \frac{r}{h_3(\lfloor \log(r) \rfloor)} \cdot h_3(\lfloor \log(r) \rfloor) = r \quad (10.2)$$

We obtain that the rank of our returned element from RSEL3 to SEL3 is $h_3(k) \geq k$, which is what we wanted.

10.5 Theoretical bound

In order to prove the theoretical bound of $O(k \cdot 2^{\log^*(k)})$ we will write up a recursion and solve it. Additionally we will argue that the rank of the last returned element will be that of the desired bound. Should the rank of the last returned element be $O(k \cdot 2^{\log^*(k)})$ then the last breadth first search and Select part will take $O(k \cdot 2^{\log^*(k)})$ time.

In order to analyze the asymptotic bound on time of SEL3, we will follow Frederickson's paper [5] losely, and include details which were omitted in the paper. Now consider a call to SEL3 with parameter $r > 1$. Here we perform:

$$limit := \frac{h_3(r)}{h_3(\lfloor \log(r) \rfloor)}$$

extractMin operations and $2 \cdot limit$ insert operations. Note that for the analysis we use the papers definition of the limit, and not how we changed it. Assuming the priority queue we use these operations on has size $O(r)$ (shown later), this will in total take $O(r)$ time. This means the operations will take $O(\log(r))$, and since $r \leq h_3(r)$ for $r > 1$ and by raising the constant slightly, we can say the operations take $O(h_3(\lfloor \log(r) \rfloor))$ yielding in total:

$$\frac{h_3(r)}{h_3(\lfloor \log(r) \rfloor)} \cdot c' \cdot h_3(\lfloor \log(r) \rfloor) = O(h_3(r))$$

This means we spend $O(h_3(r))$ time on one level of our recursion. Additionally since we make one recursive call for each insert operation, we get the recursion to describe our running time as:

$$\begin{aligned} &T(1) \leq c \\ &T(r) \leq ch_3(r) + 2 \cdot \frac{h_3(r)}{h_3(\lfloor \log(r) \rfloor)} \cdot T(\lfloor \log(r) \rfloor) \end{aligned} \tag{10.3}$$

for some suitable constant c. Now that we have this we would like to show that $T(r) \leq c' \cdot h_3(r) \cdot 2^{\log^*(r)} \cdot \prod_{i=1}^{2 \cdot \log^*(r)} (1 + \frac{4}{i^2})$. The reason for multiplying by this product, is because it is bounded by a constant, and we got the inspiration for it from Frederickson [5] proving the bound of SEL4. Now we will prove the bound by induction. Here it is trivial to see that the base $r = 1$ holds, additionally we will show the base case $r = 2$.

$$c \cdot h_3(2) + 2 \cdot \frac{h_3(2)}{h_3(\lfloor \log(2) \rfloor)} \cdot T(\lfloor \log(2) \rfloor) = 2c + 2 \cdot \frac{2}{1} \cdot c = 6c$$

This needs to be smaller than or equal to:

$$c' \cdot h_3(2) \cdot 2^{\log^*(2)} \cdot \prod_{i=1}^{2 \cdot \log^*(2)} (1 + \frac{4}{i^2}) = 2c' \cdot 2 \cdot (1+4) \cdot (1+1) = 40c'$$

Which means this holds if $c' \geq \frac{6}{40} \cdot c$

Now we would like to show the induction step, here we have the induction hypothesis that for some $r' < r$ it holds that $T(r') \leq c' \cdot h_3(r') \cdot 2^{\log^*(r')} \cdot$

$\prod_{i=1}^{2\cdot\log^*(r')} (1+\frac{4}{i^2})$. We show:

$$\begin{aligned}
T(r) &\le ch_3(r) + 2\cdot \frac{h_3(r)}{h_3(\lfloor \log(r)\rfloor)} \cdot T(\lfloor \log(r)\rfloor) && \text{(Recursion)}\\
&\le ch_3(r) + 2\cdot h_3(r)\cdot c' \cdot 2^{\log^*(\lfloor \log(r)\rfloor)} \cdot \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2}) && \text{(Ind. hypo.)}\\
&= ch_3(r) + c'\cdot h_3(r)\cdot 2^{1+\log^*(\lfloor \log(r)\rfloor)} \cdot \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2})\\
&\le ch_3(r) + c'\cdot h_3(r)\cdot 2^{1+\log^*(\lceil \log(r)\rceil)} \cdot \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2})\\
&= ch_3(r) + c'\cdot h_3(r)\cdot 2^{\log^*(r)} \cdot \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2}) && \text{(Def. } \log^*\text{)}
\end{aligned}$$

This expression needs to be smaller than or equal to what we are trying to prove, which means we need to select c' such that:

$$ch_3(r) + c'\cdot h_3(r)\cdot 2^{\log^*(r)} \cdot \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2}) \le c'\cdot h_3(r)\cdot 2^{\log^*(r)} \cdot \prod_{i=1}^{2\cdot\log^*(r)} (1+\frac{4}{i^2})$$

We divide on both sides by $h_3(r)$ and obtain:

$$c + c'\cdot 2^{\log^*(r)} \cdot \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2}) \le c'\cdot 2^{\log^*(r)} \cdot \prod_{i=1}^{2\cdot\log^*(r)} (1+\frac{4}{i^2})$$

If we move over the part containing c' on the left side, to the right side we obtain:

$$c \le c'\cdot 2^{\log^*(r)} \cdot (\prod_{i=1}^{2\cdot\log^*(r)} (1+\frac{4}{i^2}) - \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2}))$$

Isolating c' we can see how we need to set c':

$$\frac{1}{2^{\log^*(r)} \cdot (\prod_{i=1}^{2\cdot\log^*(r)} (1+\frac{4}{i^2}) - \prod_{i=1}^{2\cdot\log^*(\lfloor \log(r)\rfloor)} (1+\frac{4}{i^2}))} \cdot c \le c' \tag{10.4}$$

We note here that for $r > 2$, we have $\log^*(r) > \log^*(\lfloor \log(r)\rfloor)$ since:

$$\begin{aligned}
\log^*(r) &= 1 + \log^*(\lceil \log(r)\rceil) && \text{(Def. } \log^*\text{)}\\
&\ge 1 + \log^*(\lfloor \log(r)\rfloor)\\
&> \log^*(\lfloor \log(r)\rfloor)
\end{aligned}$$

Which means we will never get a 0 in the denominator of equation 10.4. The left side of equation 10.4 takes on its maximum when the denominator takes on its minimum. Since the denominator is a growing function it takes on its minimum when r is as small as possible, which in this case is when $r = 3$. For $r = 3$ we get equation 10.4 to be:

$$\frac{9}{290} \cdot c \leq c'$$

The claim then follows by induction with c' chosen to be larger than or equal to $\frac{9}{290} \cdot c$. Lastly we note that $\prod_{i=1}^{\infty}(1 + \frac{4}{i^2})$ is bounded by a constant by Theorem 19.0.1 and because:

$$\sum_{i=1}^{\infty} \frac{4}{i^2} = 4 \cdot \sum_{i=1}^{\infty} \frac{1}{i^2}$$

converges by Theorem 19.0.2, and because for $r > 1$ we have:

$$\begin{aligned} h_3(r) &= h_3(\lfloor \log(r) \rfloor) \cdot \left\lceil \frac{r}{h_3(\lfloor \log(r) \rfloor)} \right\rceil \\ &\leq h_3(\lfloor \log(r) \rfloor) \cdot (\frac{r}{h_3(\lfloor \log(r) \rfloor)} + 1) \\ &= r + h_3(\lfloor \log(r) \rfloor) \end{aligned}$$

we obtain that:

$$T(r) \leq c' \cdot h_3(r) \cdot 2^{\log^*(r)} \cdot \prod_{i=1}^{2 \cdot \log^*(r)} = O(r \cdot 2^{\log^*(r)})$$

Which is what we wanted to show.

Before we show that the rank of our final returned element is $O(k \cdot 2^{\log^*(k)})$ we will argue our assumption about the size of priority queues at our recursive levels.

We'll show that the priority queue $\mathcal{H}$ will get a size of at most $2 \cdot limit$ provided it is at most size *limit* to begin with. First, let's look at the case where $\mathcal{H} = nil$: We add a single element, then proceed to extract *limit* elements and insert $2 \cdot limit$ elements. This nets us a size of $limit + 1$.

If initially the size of $\mathcal{H}$ is *limit*, we extract *limit* elements and insert $2 \cdot limit$ elements. This gives us a final size of $2 \cdot limit$.

When we return a clan C containing $\mathcal{H}$ at the end of the recursive call, subsequent recursive calls starting from C will have $\mathcal{H}$ split in two even sized priority queues, $\mathcal{H}_1$ and $\mathcal{H}_2$. These will thus be at most size *limit*, meaning the recursive call will never get a priority queue of a larger size.

This means $\mathcal{H}$ will be at most size $O(r)$ like we wanted.

Now what remains is to show that the rank of the returned element from RSEL3 to SEL3 is $O(k \cdot 2^{\log^*(k)})$. This follows [5], page 207 exactly. Note that when [5] references the function h we called this h_3:

> *To complete the analysis, we show that the rank of the element returned by procedure RSEL3 to algorithm SEL3 is $O(k \cdot 2^{\log^*(k)})$. Clearly this is true if $k = 1$, so consider $k > 1$. First observe that any representative of a clan is smaller than the elements in the associated heap for the clan. Next, observe that any element x in H_0 that is not in any clan created by RSEL3 has an ancestor y that is not in a clan but whose parent z comprises a clan of size 1. Element z will be the representative of its clan, and thus be in an associated heap. A recursive application of the first observation establishes that z is larger than the element returned to algorithm SEL3. But z is smaller than x, which implies that any element not in a clan created by RSEL3 is larger than the element returned to algorithm SEL3. The number of elements placed in clans at all levels while finding a clan of size $h(r)$ is at most $T(r)$.*

From this quote and from our knowledge on the bound of $T(r)$ we can then see that the element returned to SEL3 has rank $O(k \cdot 2^{\log^*(k)})$. Which is what we wanted.

Chapter 11

SEL4 $O(k)$

11.1 Introduction

For our fifth algorithm we have implemented the fourth discussed algorithm in Frederickson's paper [5]. We have illustrated how this algorithm works in Chapter 12.

11.2 The algorithm

We make use of *clan-4* (referred to simply as *clan* in this chapter) described in chapter 6.3, as well as the priority queues described in chapter 9. The algorithm is almost identical to SEL3, but will be fully described regardless.

SEL4 defines the recursive subroutine RSEL4, which takes $\mathcal{H}$ and r and returns C. $\mathcal{H}$ is a priority queue of clans, r is the size of C, and C is the clan returned. RSEL4 also has access to $\mathcal{H}_0$. Initially SEL4 calls RSEL4 with arguments nil and k. When RSEL4 returns a clan C, SEL4 extracts the representative of C and uses a breadth first search and the Select algorithm described in chapter 5 to find the k smallest elements.

RSEL4 works as follows:

```
1. If $r = 1$ :
    a) If $\mathcal{H} = nil$ :
        i. $root := min(\mathcal{H}_0)$
        ii. Let $\mathcal{H}$ be a priority queue containing
            $leftChild(root), rightChild(root)$
        iii. return $createClan(root, \mathcal{H}, 1)$
    b) $x := extractMin(\mathcal{H})$
    c) Insert children of $x$ into $\mathcal{H}$
    d) Return $createClan(rep(x), \mathcal{H}, 1)$
```

```
2. If $\mathcal{H} = nil$ :
    a) $C := RSEL4(\mathcal{H}, f(r))$
    b) $cat(C) := 1$
    c) Let $\mathcal{H}$ be a priority queue containing $C$
3. $limit := \left\lceil \frac{r}{h_4(f(r))} \right\rceil$
4. $catLimit := \log^*(r)^2$
5. $limit$ times do:
    a) $C := extractMin(\mathcal{H})$
    b) If $cat(C) < catLimit$:
        i. $C_1 := RSEL4(priorityQueue(C), f(r))$
        ii. $cat(C_1) := cat(C) + 1$
        iii. Insert $C_1$ into $\mathcal{H}$
    c) Else:
        i. Split $priorityQueue(C)$ into $\mathcal{H}_1$ and $\mathcal{H}_2$
        ii. $C_1 := RSEL4(\mathcal{H}_1, f(r))$
        iii. $C_2 := RSEL4(\mathcal{H}_2, f(r))$
        iv. $cat(C_1) := cat(C_2) := 1$
        v. Insert $C_1$ and $C_2$ into $\mathcal{H}$
6. Return last $C$ extracted from $\mathcal{H}$
```

Just like in RSEL3 we can for RSEL4 assign *limit* in step 3 to something simpler than stated in [5]. Since we, like in RSEL3, are in the case where $r > 1$, we can apply the definition of our function h_4 here. Note that the definition of h is different for SEL3 and SEL4, though quite similar:

$$h_4(r) = \begin{cases} 1 & \text{if r = 1} \\ h_4(f(r)) \cdot \left\lceil \frac{r}{h_4(f(r))} \right\rceil & \text{otherwise} \end{cases}$$

We substitute $h_4(r)$ in $\frac{h_4(r)}{h_4(f(r))}$, which is what *limit* would be assigned to in Frederickson's paper [5], with the right side of the definition for $h_4(f(r))$ and, similarly to RSEL3, we obtain:

$$limit := h_4(f(r)) \cdot \left\lceil \frac{r}{h_4(f(r))} \right\rceil \cdot \frac{1}{h_4(f(r))} = \left\lceil \frac{r}{h_4(f(r))} \right\rceil$$

This is the reason we have set $limit := \left\lceil \frac{r}{h_4(f(r))} \right\rceil$ in step 3.

In comparison to RSEL3, RSEL4 differs in the number of clans retrieved from $\mathcal{H}$ as well when a clan is split. The former is pretty simple, while the latter is illustrated with these examples:

If $r = 82570$ then $catLimit = 25$ and $limit = 7507$. That means when the first clan is created and put into $\mathcal{H}$, it must be extracted and reinserted 24 times before it is split the first time. Since the clans made from the split heap get a category of 1, they again must be extracted 24 times individually before one is split. If we always extract the clan with the highest category, we get the maximum number of splits. We'll increase $\mathcal{H}$ with a size of roughly 312 after $7507 + \frac{7507}{24}$ inserts and 7507 extractions. If instead we always extract the clan with the smallest category, we can *bank* a category of 23 before forcing a split. This means we'll get a new clan every 47'th extraction (after creating two clans, extract one 23 time and the other 24 times) and we'll insert roughly 159 clans into the heap after $7507 + \frac{7507}{47}$ inserts and 7507 extractions.

If $r = 1024$ then $catLimit = 16$ and $limit = 171$. Every 15'th extraction of a single clan forces a split of its heap. With maximum splits we increase the size of $\mathcal{H}$ with roughly 11 after $171 + \frac{171}{15}$ inserts and 171 extractions. With minimum splits we increase the size with roughly 5 after $171 + \frac{171}{29}$ inserts and 171 extractions.

At the base case the heap grows on every extraction, since every element must have its two children accounted for.

11.3 Termination

Since our extractMin and Insert operations terminate, we can easily see that SEL4 terminates because we always work on smaller sizes in our recursive calls, and we run the for loop a fixed number of times. This means by an argument similar to the earlier algorithms that our priority queues will all have a finite size.

11.4 Correctness

For future references we would like to show that $f(r)$ is positive, that is $f(r) \geq 1$, when $r > 1$. To show this we need to show that $\lceil \log(r) \rceil \geq \log^*(r)$. Since these are both integers, this would mean $f(r) \geq 1$. First we show it holds for the case where $r = 2$:

$$\log^*(2) = 1$$
$$\lceil \log(2) \rceil = 1$$

Which clearly shows that $\lceil \log(r) \rceil \geq \log^*(r)$, when $r = 2$, now we assume $r > 2$, note that this means $\lceil \log(r) \rceil > 1$:

$$\begin{aligned} \log^*(r) &= 1 + \log^*(\lceil \log(r) \rceil) && \text{(Def. } \log^*\text{)} \\ &\leq 1 + \lceil \log(r) \rceil - 1 && (\log^*(a) < a) \\ &= \lceil \log(r) \rceil \end{aligned}$$

Which concludes showing that $\lceil \log(r) \rceil \geq \log^*(r)$ for $r > 1$, and thus we have:

$$f(r) \geq 1 \tag{11.1}$$

Just like for the previous algorithms, in order to argue correctness for SEL4, we just need to show that the element returned from RSEL4 to SEL4 has a rank greater than or equal to k.

Like for RSEL3 we first note that any element extracted from a priority queue at level ℓ of our recursion, is extracted at most once, this means each clan generated at level ℓ contains distinct elements, meaning any pair of clans at level ℓ are disjoint.

In order to then argue correctness we would like to show that a call to RSEL4 with argument $f(r)$ returns the $h_4(f(r))$ smallest elements. We argue this using induction: First consider the case where $r = 1$, which trivially holds. Then we make the induction hypothesis that for $r' < r$ it holds that RSEL4 with argument $f(r')$ returns the $h_4(f(r'))$ smallest elements. In order to show this we use the paper version of the limit:

$$limit := \frac{h_4(r)}{h_4(f(r))}$$

Since this is the number of clans we extract and every clan contains distinct elements by the earlier argument, we have from our induction hypothesis that these clans have size $h_4(f(r))$ giving us that the representative of the last extracted clan, C_{last} which will be larger than all previous representatives, since it was extracted last, and thereby the largest element of all the extracted clans. This gives us that the rank of this element will be:

$$rank(C_{last}) = limit \cdot h_4(f(r)) = h_4(r)$$

Since for $r > 1$ we have:

$$h_4(r) = h_4(f(r)) \cdot \left\lceil \frac{r}{h_4(f(r))} \right\rceil \geq h_4(f(r)) \cdot \frac{r}{h_4(f(r))} = r \tag{11.2}$$

We see that the rank of the element returned from RSEL4 to SEL4 is $h_4(k) \geq k$, which is what we needed.

11.5 Theoretical bound

Looking at the pseudo code it is easy to see that for $r = 1$, our algorithm spends constant time. Then for the case $r > 1$ we spend $\mathit{limit} \cdot \lceil \log(r) \rceil$ multiplied by some constant, since we have in the order of *limit* insertions and extractMin operations. The argument for the time of one of these operations being a constant times $\lceil \log(r) \rceil$ will follow later. We will use $\mathit{limit} = \frac{h_4(r)}{h_4(f(r))}$ like in [5]. Additionally we will have c_1 to be the constant it is multiplied by:

$$\begin{aligned}
c_1 \cdot \mathit{limit} \cdot \lceil \log(r) \rceil &= c_1 \cdot \frac{h_4(r)}{h_4(f(r))} \cdot \lceil \log(r) \rceil && \text{(Def. limit)} \\
&\leq c_1 \cdot \frac{h_4(r)}{f(r)} \cdot \lceil \log(r) \rceil && \text{(Eq. 11.2)} \\
&= c_1 \cdot h_4(r) \cdot \frac{1}{\left\lfloor (\frac{\lceil \log(r) \rceil}{\log^*(r)})^2 \right\rfloor} \cdot \lceil \log(r) \rceil && \text{(Def. f)} \\
&\leq c_1 \cdot h_4(r) \cdot \frac{2 \cdot (\log^*(r))^2}{\lceil \log(r) \rceil} && \text{(Eq. 11.1)} \\
&= 2 \cdot c_1 \cdot h_4(r) \cdot \frac{(\log^*(r))^2}{\lceil \log(r) \rceil} \\
&= O(h_4(r) \cdot \frac{(\log^*(r))^2}{\lceil \log(r) \rceil})
\end{aligned}$$

Additionally we make one recursive call every iteration except for the $\frac{1}{(\log^*(r))^2}$ time where we make two. Thus we get the recursion:

$$\begin{aligned}
&T(1) \leq c \\
&T(r) \leq c h_4(r) \cdot \frac{(\log^*(r))^2}{\lceil \log(r) \rceil} + (1 + \frac{1}{(\log^*(r))^2}) \cdot \mathit{limit} \cdot T(f(r))
\end{aligned} \tag{11.3}$$

We would now like to show that $T(r) \leq c' \cdot h_4(r) \cdot \prod_{i=1}^{2 \cdot \log^*(r)} (1 + \frac{4}{i^2})$. We will here follow Frederickson's work on page 211 in [5] very closely for the induction part. Frederickson [5] refrains from showing a lot of the inequalities needed to make the induction work. To show this we use induction on r. Here the basis is for $r \leq 2$. For $r = 1$, $T(1) \leq c$, so the claim is satisfied if $6c \leq c' \cdot 2(1 + \frac{4}{1})(1 + \frac{4}{4}) = 20 \cdot c'$. For $r = 2$, $T(2) \leq c \cdot 2 \cdot \frac{1}{1} + (1 + \frac{1}{1}) \cdot \frac{2}{1} \cdot c = 6c$, so our claim is satisfied if $6c \leq c'2 \cdot (1 + \frac{4}{1})(1 + \frac{4}{4}) = 20 \cdot c'$. For $r > 2$, we assume that the claim is true for $r' < r$. Before we start we show that $\log^*(r) > \log^*(f(r))$, when $r > 2$. First we will show that r grows faster than $f(r)$. If we know that $r > f(r)$ and that r grows faster than $f(r)$, then the difference between $\log^*(r)$ and $\log^*(f(r))$ will only grow as r grows. Since we

Table 11.1: r vs. f(r)

r	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0
f(r)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

have:

$$\begin{aligned} f(r) &= \left\lfloor \frac{\lceil \log(r) \rceil^2}{(\log^*(r))^2} \right\rfloor \\ &\leq \frac{\lceil \log(r) \rceil^2}{(\log^*(r))^2} \\ &\leq \lceil \log(r) \rceil^2 \\ &\leq (1+\log(r))^2 \end{aligned}$$

The growth of f is smaller than or equal to the growth of $(1+\log(r))^2$. To get how fast $(1+\log(r))^2$ grows, we take the derivative:

$$\frac{d}{dr}((1+\log(r))^2) = 2 \cdot (\log(r)+1) \cdot \frac{1}{r \cdot ln(2)}$$

We also take the derivative of r, but this is simple and we get $\frac{d}{dr}(r) = 1$. In order for it to hold that:

$$2 \cdot (\log(r)+1) \cdot \frac{1}{r \cdot ln(2)} \leq 1$$

We need to have $r \geq 12$. That means for $r \geq 12$, r grows faster than $f(r)$. What is missing then to show that r also grows faster than $f(r)$ for $r \in \{3,4,5,6,7,8,9,10,11\}$. We show them all in Table 11.1. We can see from Table 11.1 that r also grows faster than $f(r)$ below 12. Since we have established that r grows faster than $f(r)$ we only remain to show that for the smallest possible difference between r and $f(r)$, it holds that $\log^*(r) > \log^*(f(r))$. This smallest possible value is for $r = 3$, since we are only dealing with the case where $r > 2$. We obtain:

$$\begin{aligned} \log^*(3) &= 1 + \log^*(2) = 2 \\ \log^*(f(3)) &= \log^*(1) = 1 \end{aligned}$$

Since $2 > 1$, we obtain:

$$\log^*(r) > \log^*(f(r)) \tag{11.4}$$

Note that this result means that $\log^*(f(r)) \leq \log^*(r) - 1$. Now we will do the induction case of our proof. We have:

$$\begin{aligned}
T(r) &\leq ch_4(r) \cdot \frac{(\log^*(r))^2}{\lceil \log(r) \rceil} + (1 + \frac{1}{(\log^*(r))^2}) \cdot \frac{h_4(r)}{h_4(f(r))} && \text{(Eq. 11.3)} \\
&\leq ch_4(r)\frac{(\log^*(r))^2}{\lceil \log(r) \rceil} \\
&+ (1 + \frac{4}{(2\log^*(r))^2})\frac{h_4(r)}{h_4(f(r))}c'h_4(f(r)) \prod_{i=1}^{2\log^*(f(r))} (1 + \frac{4}{i^2}) && \text{(Ind. Hypo.)} \\
&\leq ch_4(r)\frac{(\log^*(r))^2}{\lceil \log(r) \rceil} + (1 + \frac{4}{(2\log^*(r))^2})c'h_4(r) \prod_{i=1}^{2\log^*(r)-2} (1 + \frac{4}{i^2}) && \text{(Eq. 11.4)} \\
&= ch_4(r)\frac{(\log^*(r))^2}{\lceil \log(r) \rceil} + \frac{c'h_4(r) \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})}{(1 + \frac{4}{(2\log^*(r)-1)^2})}
\end{aligned}$$

Using the last equation, we can see that it will hold if:

$$ch_4(r)\frac{(\log^*(r))^2}{\lceil \log(r) \rceil} + \frac{c'h_4(r) \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})}{(1 + \frac{4}{(2\log^*(r)-1)^2})} \leq c' \cdot h_4(r) \cdot \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})$$

Multiplying by $(1 + \frac{4}{(2\log^*(r)-1)^2})$ we obtain:

$$\begin{aligned}
ch_4(r)\frac{(\log^*(r))^2}{\lceil \log(r) \rceil}(1 + \frac{4}{(2\log^*(r)-1)^2}) + c'h_4(r) \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2}) \\
\leq (1 + \frac{4}{(2\log^*(r)-1)^2})c'h_4(r) \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})
\end{aligned}$$

The plus one on the right side multiplied with the $c'h_4(r) \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})$ on the right side, eats that same expression on the left side, giving us:

$$ch_4(r)\frac{(\log^*(r))^2}{\lceil \log(r) \rceil}(1 + \frac{4}{(2\log^*(r)-1)^2}) \leq (\frac{4}{(2\log^*(r)-1)^2})c'h_4(r) \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})$$

Which finally gives us:

$$\frac{\frac{(\log^*(r))^2}{\lceil \log(r) \rceil}(1 + \frac{4}{(2\log^*(r)-1)^2})}{(\frac{4}{(2\log^*(r)-1)^2}) \prod_{i=1}^{2\log^*(r)} (1 + \frac{4}{i^2})} \cdot c \leq c'$$

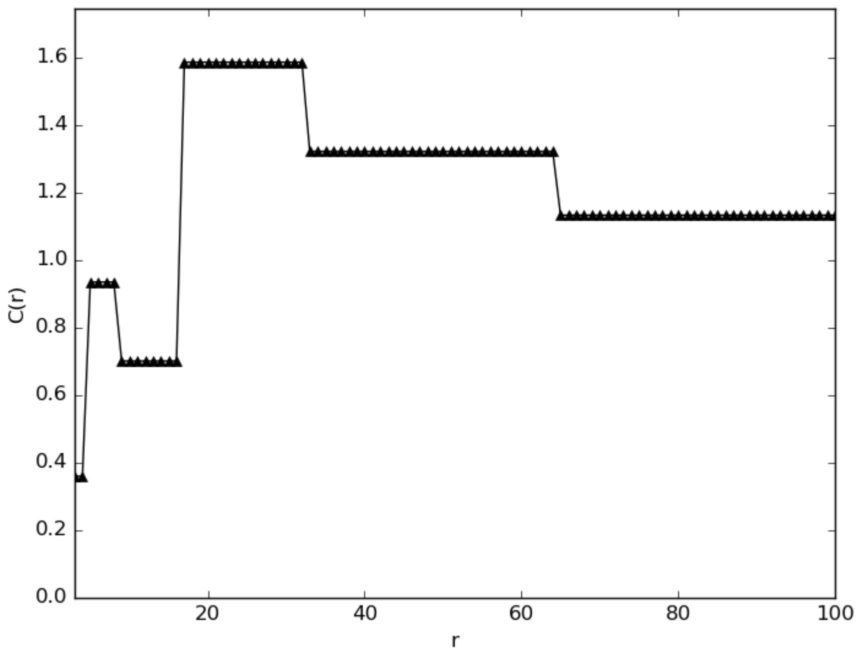

Figure 11.1: C(r) over r.

Which simplifies to:

$$\frac{(\log^*(r))^2((2\log^*(r)-1)^2+4)}{4\lceil\log^*(r)\rceil \prod\limits_{i=1}^{2\log^*(r)}(1+\frac{4}{i^2})} \cdot c \leq c'$$

If we call the expression on the left side $C(r)$, that is:

$$C(r) = \frac{(\log^*(r))^2((2\log^*(r)-1)^2+4)}{4\lceil\log^*(r)\rceil \prod\limits_{i=1}^{2\log^*(r)}(1+\frac{4}{i^2})}$$

Just like Frederickson says in [5], this function takes on its maximum in the range $r = 17$ to $r = 32$, as illustrated in Figure 11.1. We have attempted to prove that this is the case, but have been unable to do so, Frederickson also does not prove this in [5], but on page 212 just states that this is the case.

At the maximum in Figure 11.1 the value is 1.585, which means we get:

$$1.585 \cdot c \leq c'$$

The claim then follows by induction, with c' chosen to be $1.585 \cdot c$. From the claim it then follows that $T(r) = O(r)$, since for $r > 1$ we have:

$$\begin{aligned} h_4(r) &= h_4(f(r)) \cdot \left\lceil \frac{r}{h_4(f(r))} \right\rceil \\ &\leq h_4(f(r)) \cdot (1 + \frac{1}{h_4(f(r))}) \\ &\leq r + h_4(f(r)) \\ &\leq r + h_4(f(r)) \\ &= 2r \end{aligned}$$

and additionally we have from Theorem 19.0.1 and the fact that:

$$\sum_{i=1}^{\infty} \frac{4}{i^2} = 4 \cdot \sum_{i=1}^{\infty} \frac{1}{i^2}$$

converges by Theorem 19.0.2, that $\prod_{i=1}^{\infty}(1+\frac{4}{i^2})$ converges, and is thereby bounded by a constant.

What now remains to be shown is that the rank of the representative of our final extracted clan is $O(k)$. We argue this much like the earlier algorithms: Since we argued earlier that a call to RSEL4 with argument k returns the $h_4(k)$ smallest elements in a clan, represented by the representative of the clan, and since the representative is the largest element among them and also since we extract these from a priority queue, the rank of the representative of the last extracted clan will be:

$$\frac{h_4(k)}{h_4(f(k))} \cdot h_4(f(k)) = h_4(k)$$

Since we have, as argued earlier, that $h_4(k) \leq 2 \cdot k$, our algorithm runs in total in $O(k)$ time.

Priority queue size

To give some intuition about how large a priority queue PQ can get at any level, assume we start off with an empty PQ. We add t elements and then return PQ to the level above. The level above may choose to split or not to split PQ. Suppose we can at most choose not to split it b times. This means we can add a total of bt elements to PQ before splitting.

After a split PQ will still retain $\frac{bt}{2}$ elements. If we use the now split PQ another b times, it will have a total of $\frac{bt}{2} + bt$ elements. After another split and b uses, it will end up with $\frac{\frac{bt}{2}+bt}{2} + bt = \frac{bt}{4} + \frac{bt}{2} + bt$ elements.

The number of elements in PQ before a split is bounded by the sum $\sum_{i=0}^{\infty} \frac{bt}{2^i} = 2bt$.

If we then use the fact that we will at most add $\frac{h_4(f(r))}{h_4(f(f(r)))} \cdot \frac{1}{\log^*(f(r))^2}$ elements at the level beneath level r, and we will at most add additional elements corresponding to how often we split during our $limit = \frac{h_4(f(r))}{h_4(f(f(r)))}$ iterations. Which means we can maximally choose to not split it $\log^*(r)^2$ times we obtain the sum:

$$\sum_{i=0}^{\infty}(\frac{1}{2^i} \cdot \frac{h_4(f(r))}{h_4(f(f(r)))} \cdot \frac{1}{\log^*(f(r))^2} \cdot \log^*(r)^2)$$

Since everything except the $\frac{1}{2^i}$ does not contain i, we get:

$$\begin{aligned}
&\frac{h_4(f(r))}{h_4(f(f(r)))} \cdot \frac{1}{\log^*(f(r))^2} \cdot \log^*(r)^2 \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} \\
&= 2 \cdot \frac{h_4(f(r))}{h_4(f(f(r)))} \cdot \frac{1}{\log^*(f(r))^2} \cdot \log^*(r)^2 \\
&\leq 4 \cdot \frac{f(r)}{h_4(f(f(r)))} \cdot \log^*(r)^2 \cdot \frac{1}{\log^*(f(r))^2} && (h_4(r) \leq 2r) \\
&\leq 4 \cdot \frac{f(r)}{f(f(r))} \cdot \log^*(r)^2 \cdot \frac{1}{\log^*(f(r))^2} && (h_4(r) \geq r) \\
&= 4 \cdot \frac{f(r)}{\left\lfloor \frac{\lceil \log(f(r)) \rceil^2}{\log^*(f(r))^2} \right\rfloor} \cdot \log^*(r)^2 \cdot \frac{1}{\log^*(f(r))^2} && (\text{Def. f}) \\
&\leq 4 \cdot \frac{f(r)}{(\frac{\lceil \log(f(r)) \rceil^2}{\log^*(f(r))^2}) - 1} \cdot \log^*(r)^2 \cdot \frac{1}{\log^*(f(r))^2} \\
&= 4 \cdot \frac{\log^*(r)^2}{((\frac{\lceil \log(f(r)) \rceil^2}{\log^*(f(r))^2}) - 1) \cdot \log^*(f(r))^2} \cdot f(r) \\
&= 4 \cdot \frac{\log^*(r)^2}{\lceil \log(f(r)) \rceil^2) - \log^*(f(r))^2} \cdot f(r)
\end{aligned}$$

For this last expression we will look at the expression:

$$\frac{\log^*(r)^2}{\lceil \log(f(r)) \rceil^2 - \log^*(f(r))^2}$$

In this expression when $r \to \infty$ $\log(f(r))^2$ will dominate the other functions since $\log^*$ is such a slow growing one. This means the fraction will eventually converge. Which means we can replace it with some constant c (possibly large) and for the numbers until $\lceil \log(f()r) \rceil^2 > \log^*(f(r))^2$ we can just take the maximum value we would obtain from the original expression as a constant. We then get:

$$4 \cdot c \cdot f(r) = O(f(r))$$

Which means the size of our priority queue at the level below has size $O(f(r))$ which means it is linear in size. Since we did this for any random level it follows that our priority queue will never exceed linear size.

Chapter 12

Examples

12.1 Naive

The left side shows the heap and the right side the priority queue.

Figure 12.1 shows $\mathcal{T}$ on the left and the empty priority queue PQ on the right. Only discovered nodes are shown.

Figure 12.2 shows the first step in the naive algorithm. The root of $\mathcal{T}$ has been added to the priority queue.

Figure 12.3 shows node 1 having been extracted from PQ. Its children 2 and 4 have been inserted.

Figure 12.4 shows node 2 having been extracted from PQ. Its children 7 and 3 have been inserted.

Figure 12.5 shows node 3 having been extracted from PQ. Its children 10

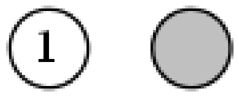

Figure 12.1: Naive step 0.

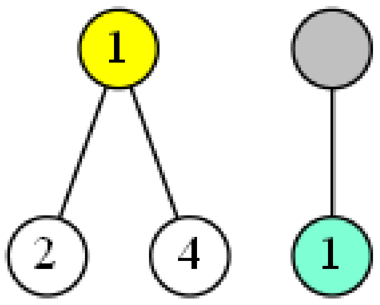

Figure 12.2: Naive step 1.

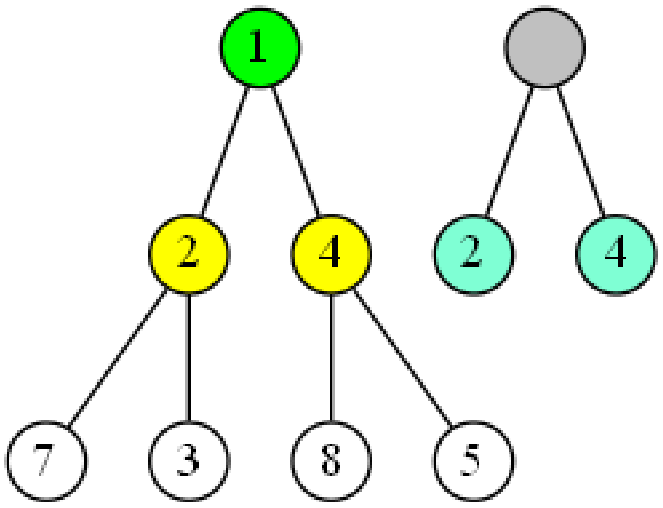

Figure 12.3: Naive step 2.

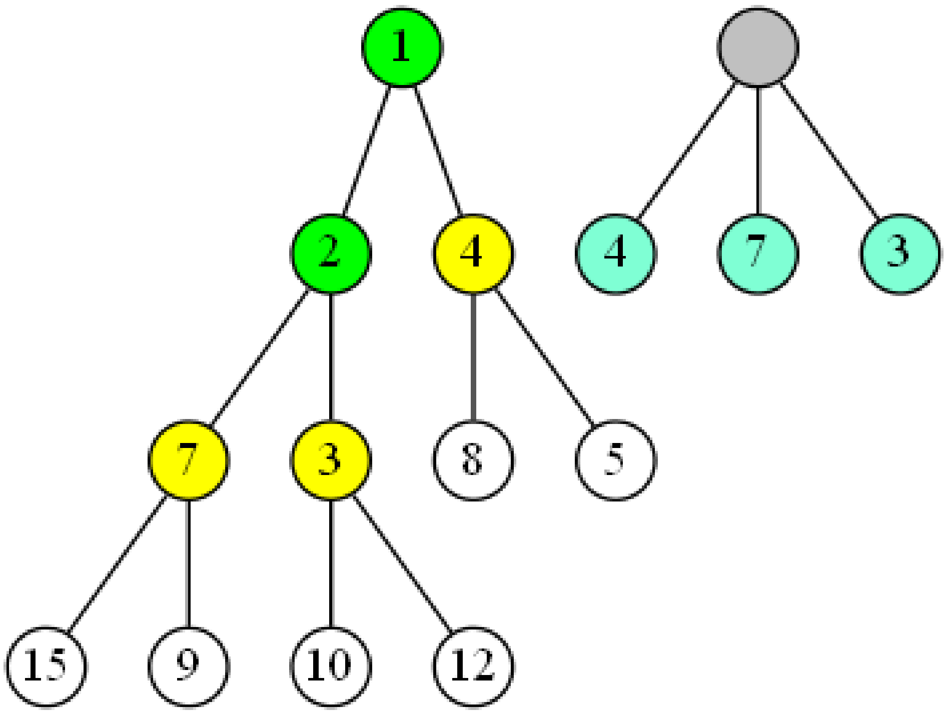

Figure 12.4: Naive step 3.

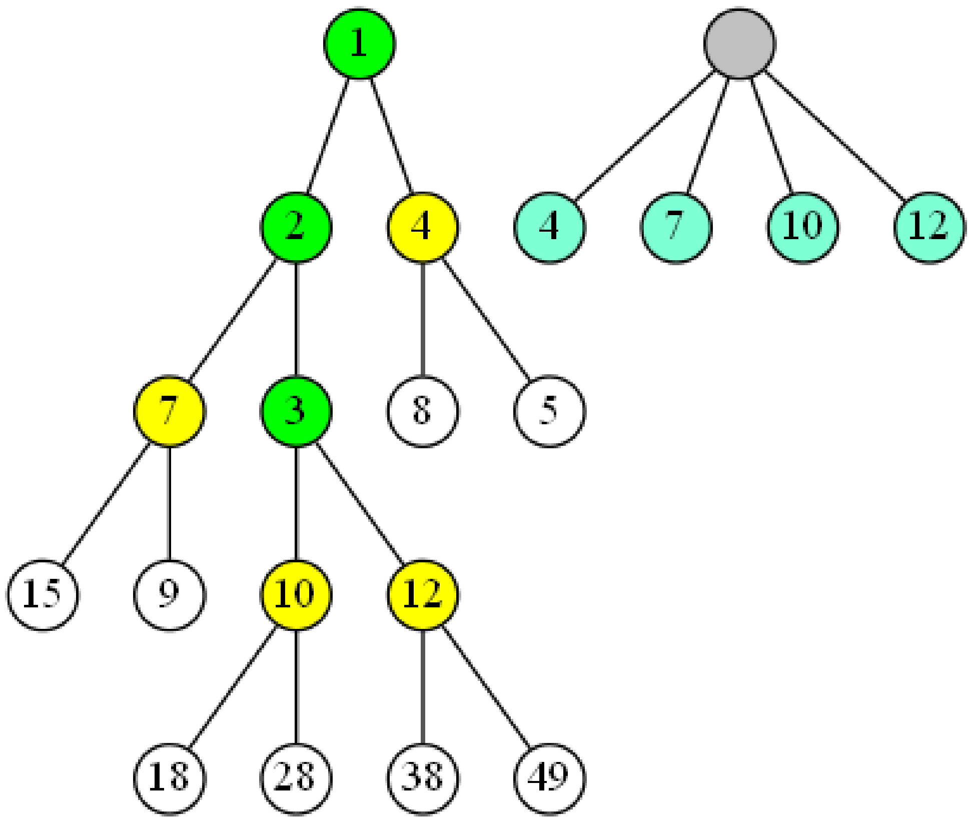

Figure 12.5: Naive step 4.

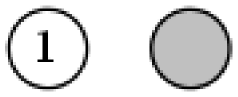

Figure 12.6: SEL1 step 0.

and 12 have been inserted.

The algorithm follows trivially from here, with 4, then 5 being the next nodes to be extracted.

12.2 SEL1 and SEL2

The left side shows the heap and the right side the priority queue.

Figure 12.6 shows $\mathcal{T}$ on the left and the empty priority queue PQ on the right. Only discovered nodes are shown.

Figure 12.7 shows the first clan consisting of the yellow nodes. The clan is found by a subroutine (naive for SEL1 and SEL2 itself for SEL2), which is not

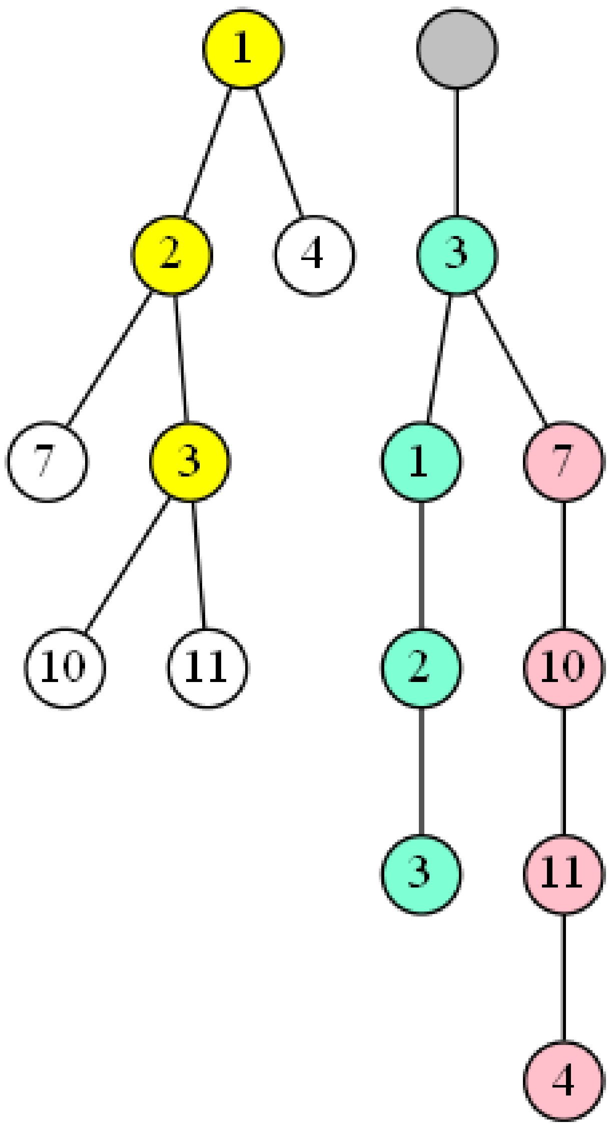

Figure 12.7: SEL1 step 1.

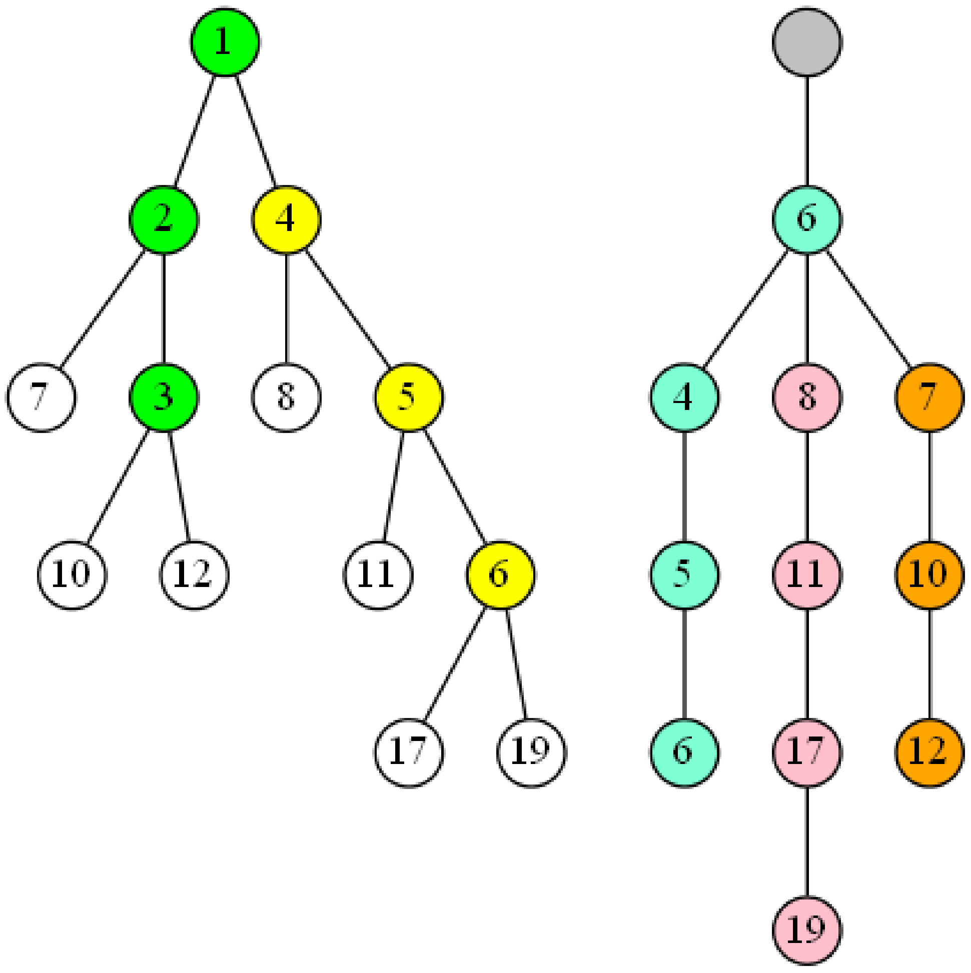

Figure 12.8: SEL1 step 2.

shown. The priority queue on the right side contains a single clan with the representative 3, members 1, 2, and 3, offspring 7, 10, 11, and 4, and no poor relation.

Figure 12.8 shows the first clan extracted from PQ, its nodes colored green in $\mathcal{T}$. From its offspring we've created a new clan represented by 6. This clan does have a poor relation, consisting of 7, 10, and 12.

Note that every node colored white in the heap to the left can be found in either offspring or poor relation to some clan in the priority queue to the right.

Figure 12.9 shows $\mathcal{T}$ after the clan represented by 6 has been extracted from PQ. Two new clans have been created, represented by 13 and 10.

Figure 12.10 shows $\mathcal{T}$ on the left and the empty priority queue PQ on the right. Only discovered nodes are shown.

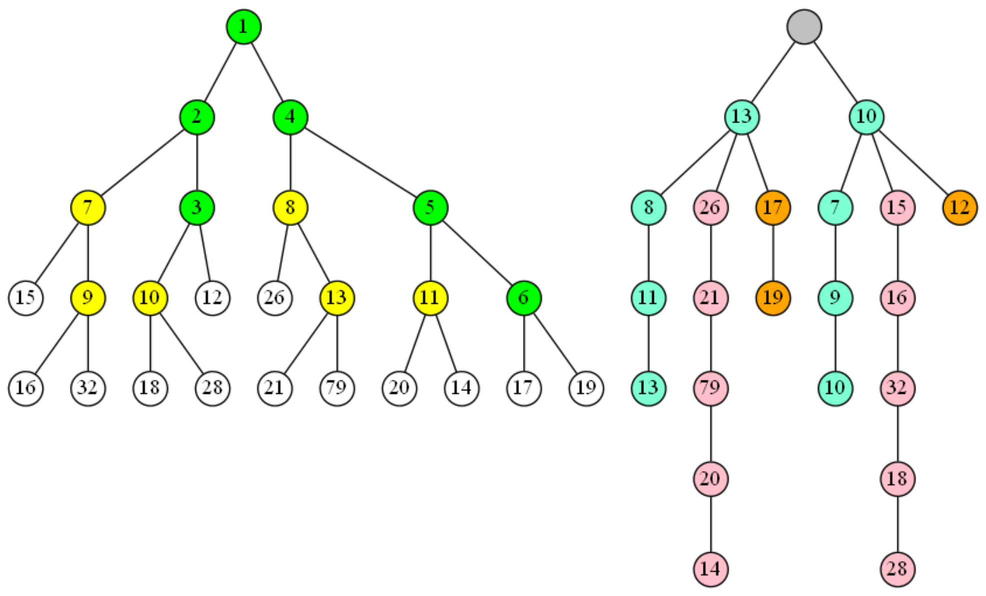

Figure 12.9: SEL1 step 3.

12.3 SEL3 and SEL4

The image shows the same two structures for SEL3 and SEL4, but PQ is changed to reflect its recursive structure. For the example we let $k = 8$, such that for SEL3, $\log k = 3$ and $\log(\log k) = 1$. While only SEL3 is depicted, the general idea is the same for SEL4, with the only difference being how large r is at the different levels, how many iterations are made, and when a priority queue is split.

The images are colored and may be harder to follow in gray scale. First, to the right we will see three layers of priority queues. We'll name them after the r they're given, thus from top to bottom, the gray nodes represent PQ_8, PQ_3 and PQ_1. To the left, a white node has been added to a PQ_1. A yellow (darker in gray scale) node has been extracted from a PQ_1. A green node (darker than the yellow) has been extracted from PQ_3. A beige node (bright in gray scale) has been extracted from PQ_8.

Figure 12.11 shows $\mathcal{T}$ on the left and the empty priority queue PQ on the right.

Figure 12.12 shows a blank PQ_8. The recursive call has not returned yet, and there are no clans in it. For PQ_3 a clan represented by 1 has been added to the queue. 1's children, 2 and 4, are found in the PQ_1 belonging to 1. Since 1 has been extracted from a PQ_1, it has been colored yellow to the left.

Figure 12.13 shows the clan represented by 1 having been extracted from PQ_3. The PQ_1 the clan contained has been split into two and recursed on, leaving two new clans in PQ_3, represented by 2 and 4. To the left, 1 has been

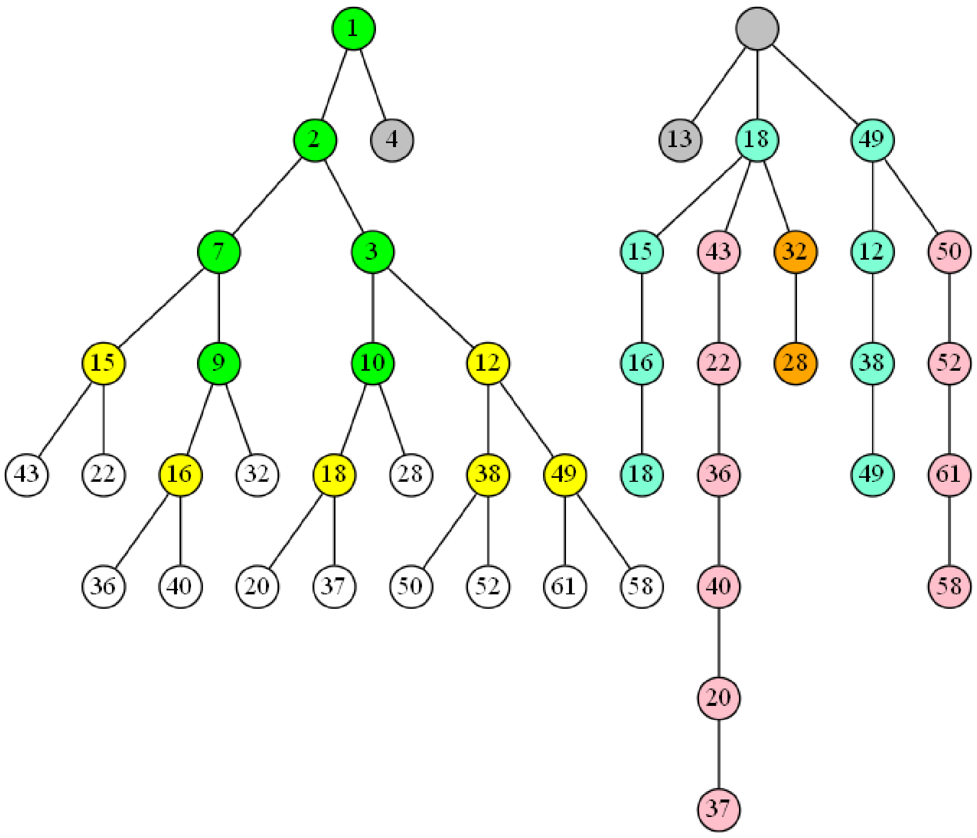

Figure 12.10: SEL1 step 4.

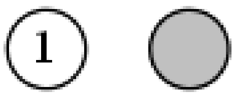

Figure 12.11: SEL3 step 0.

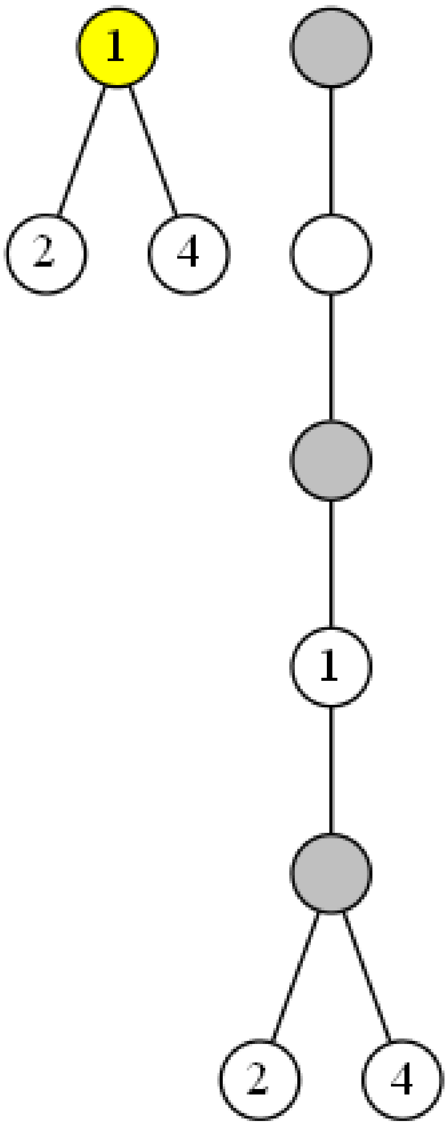

Figure 12.12: SEL3 step 1.

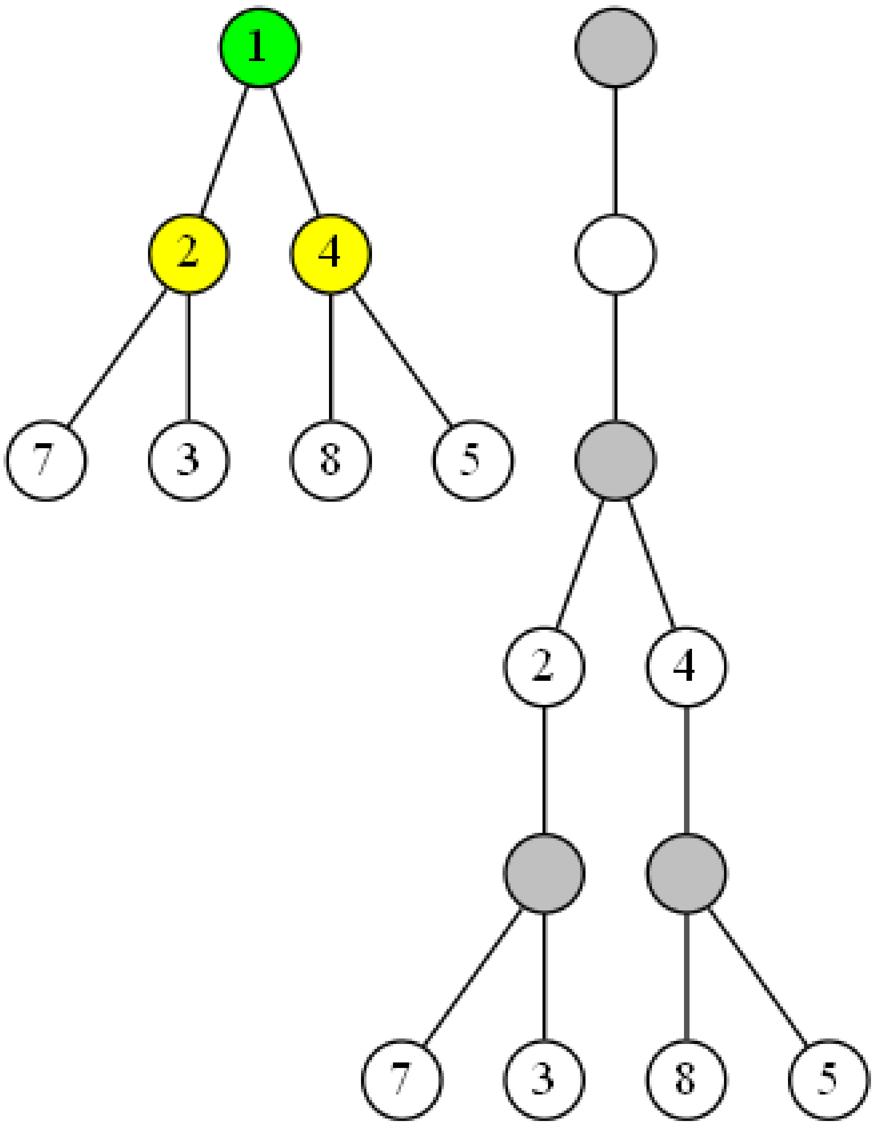

Figure 12.13: SEL3 step 2.

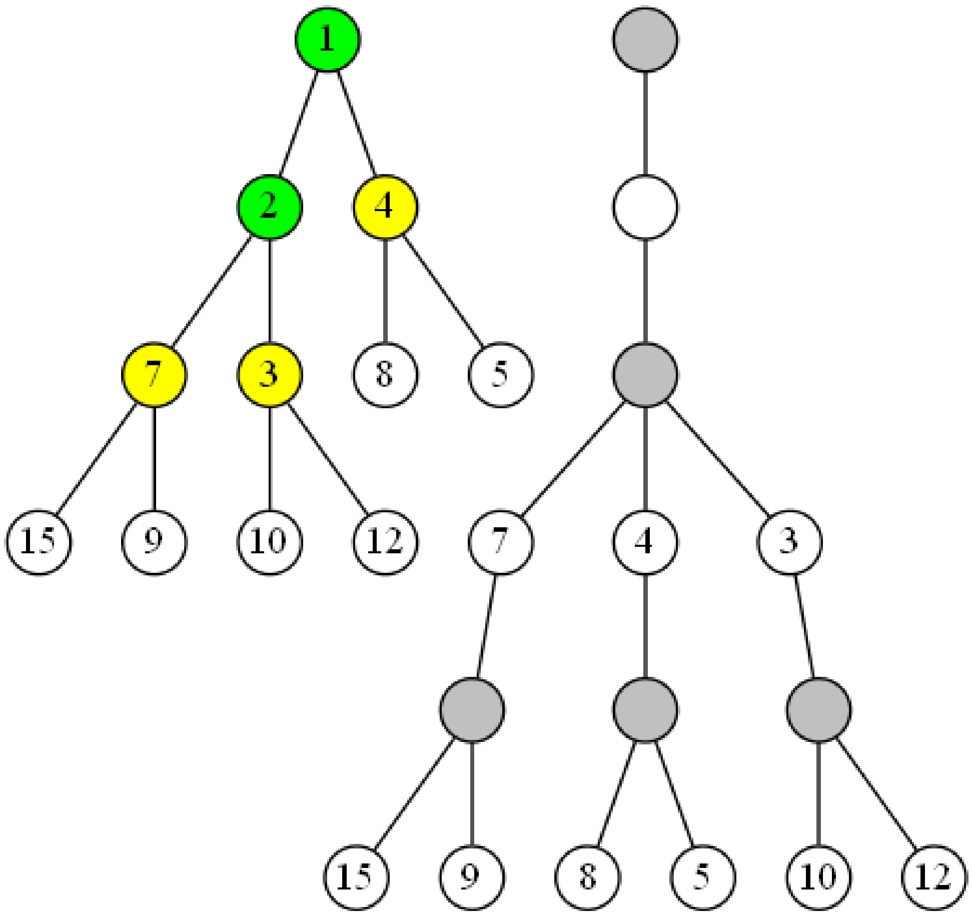

Figure 12.14: SEL3 step 3.

colored green after having been extracted from PQ_3 and 2 and 4 have been colored yellow after extraction from PQ_1.

Figure 12.14 shows 2 having been extracted from PQ_3 and its PQ_1 split into 7 and 3 and recursed on. This colors 2 green and 7 and 3 yellow.

Figure 12.15 shows 3 having been extracted from PQ_3 and its PQ_1 split into 10 and 12 and recursed on. This colors 2 green and 7 and 3 yellow. This also concludes the work done in the $r = 3$ layer, finally adding a clan to PQ_8 represented by 3.

Figure 12.16 shows 3 having been extracted from PQ_8 and its PQ_3 split into 7 and 4 on the left and 10 and 12 on the right. We color 3 beige in $\mathcal{T}$, to denote its extraction from PQ_8. Node that that level never sees the nodes 1 and 2, which are implicitly contained in the clan, thus we leave them green.

To the right, we now have two incomplete clans in PQ_8. The exact split of PQ_3 in the implementation might be different. We recurse on the incomplete

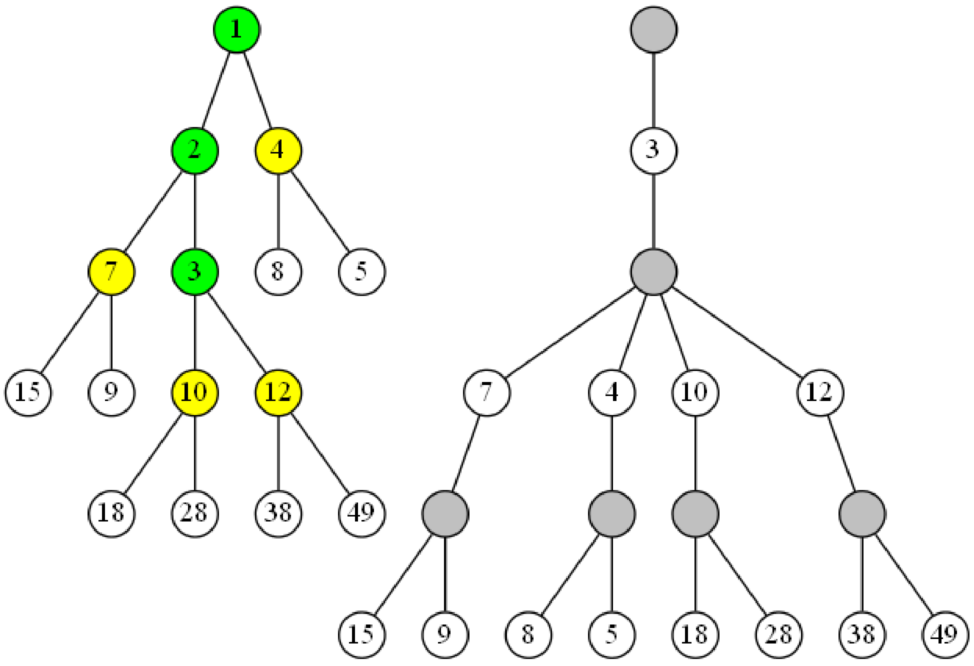

Figure 12.15: SEL3 step 4.

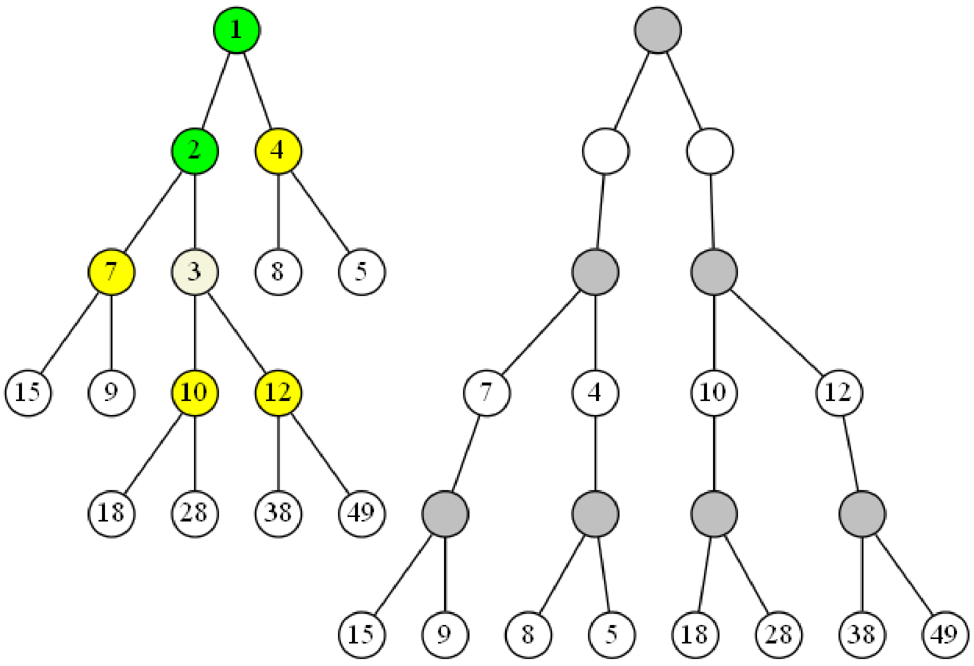

Figure 12.16: SEL3 step 5.

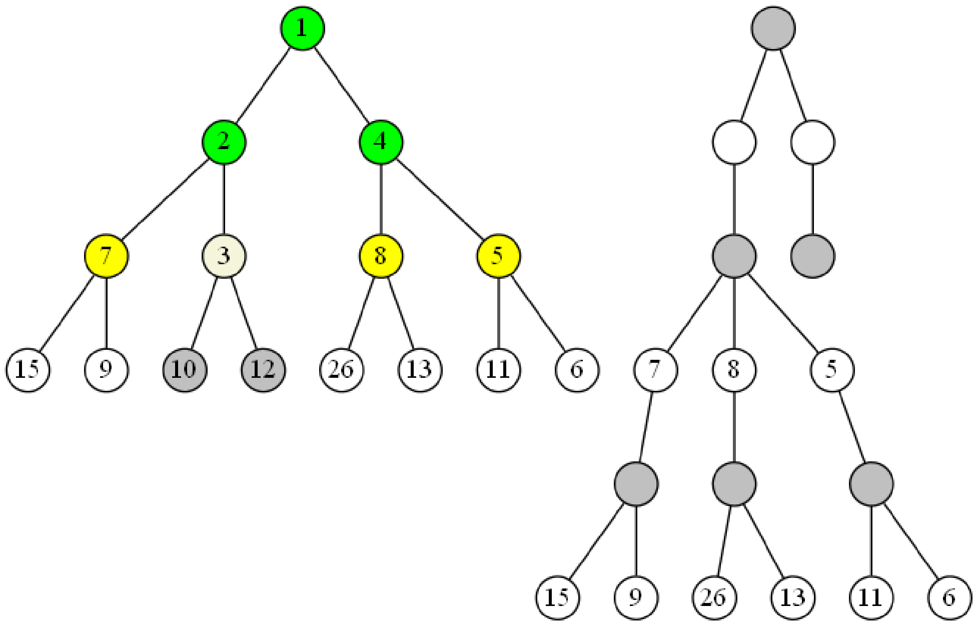

Figure 12.17: SEL3 step 6.

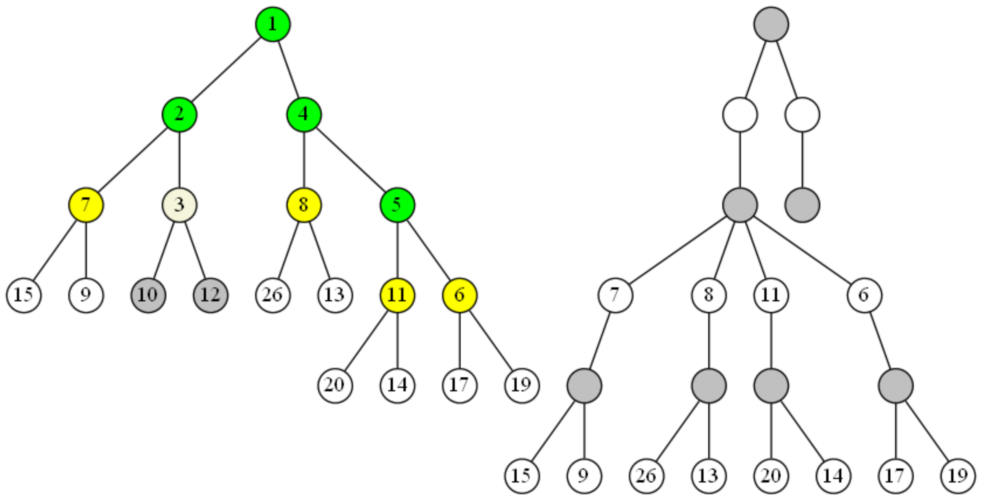

Figure 12.18: SEL3 step 7.

left clan in the next steps.

Figure 12.17 shows recursion on the left incomplete clan in PQ_8 (the right clan has been omitted to save space). This follows the same procedure as above, extracting 4 from PQ_3 and adding 8 and 5. This colors 4 green and 8 and 5 yellow.

Figure 12.18 shows 5 having been extracted from PQ_3 and 8 and 5 having

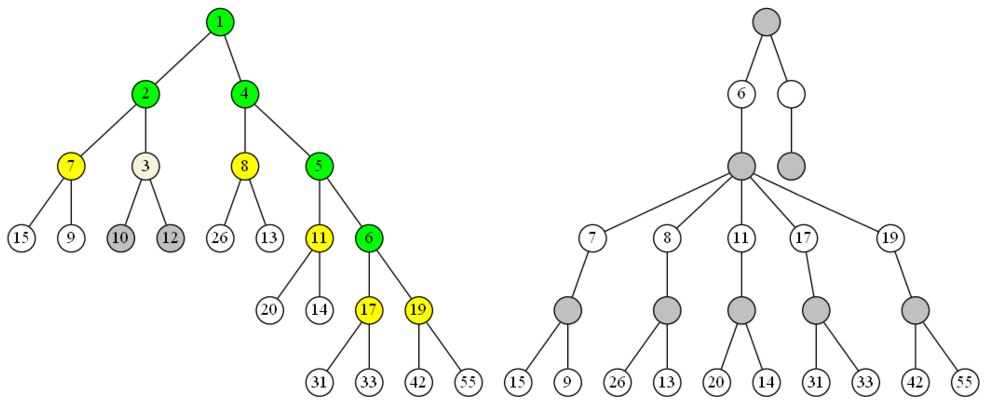

Figure 12.19: SEL3 step 8.

been added. This colors 5 green and 11 and 6 yellow.

Figure 12.19 shows 6 having been extracted from PQ_3 and 17 and 19 having been added. This colors 6 green and 17 and 19 yellow. This concludes the work done at level $r = 3$, adding a clan represented by 6 to PQ_8.

The rest of the example follows straight forward from here, with the right clan being completed next, followed by two more extractions from and insertions into PQ_8, after which the final representative is returned. This will not be shown.

Chapter 13

SEL4 parameter optimization

13.1 Introduction

In the algorithm SEL4 we have the boundary for when we enter the base case and the boundary for when we split our priority queues and make two recursive calls. These two boundaries are 1 and $\log^*(r)^2$ respectively from the theory. We might improve the running time of our algorithm by slightly changing these. To do this we have tried out a number of different choices for the boundary of the base case, we call this value $baseCase$ and the same we have tried for the other boundary which we call $catFactor$. Since the choice for $catFactor$ that optimizes our algorithm might not optimize if we change our choice of $baseCase$ we have run a cross evaluation with different values for both our boundaries. We have here chosen to focus on optimality in respect to the running time of the algorithm. We chose this because we do not know the ratio of significance between comparisons and accesses, and the optimal values might differ for these two.

13.2 Choice of Parameters

From Figure 13.1 we can see that around 16 and around 32 our values look to be minimal. Inspecting the data at these places in Table 13.1 we can see that our values are in fact minimal with $baseCase = 16$, $catFactor = 0.9$ and $baseCase = 32$, $catFactor = 1.0$. Since from Figure 13.1 the region around $baseCase = 32$ and $catFactor = 1.0$ looks less rapidly changing than around the other region, we have chosen these as the parameters for SEL4 in our experiments.

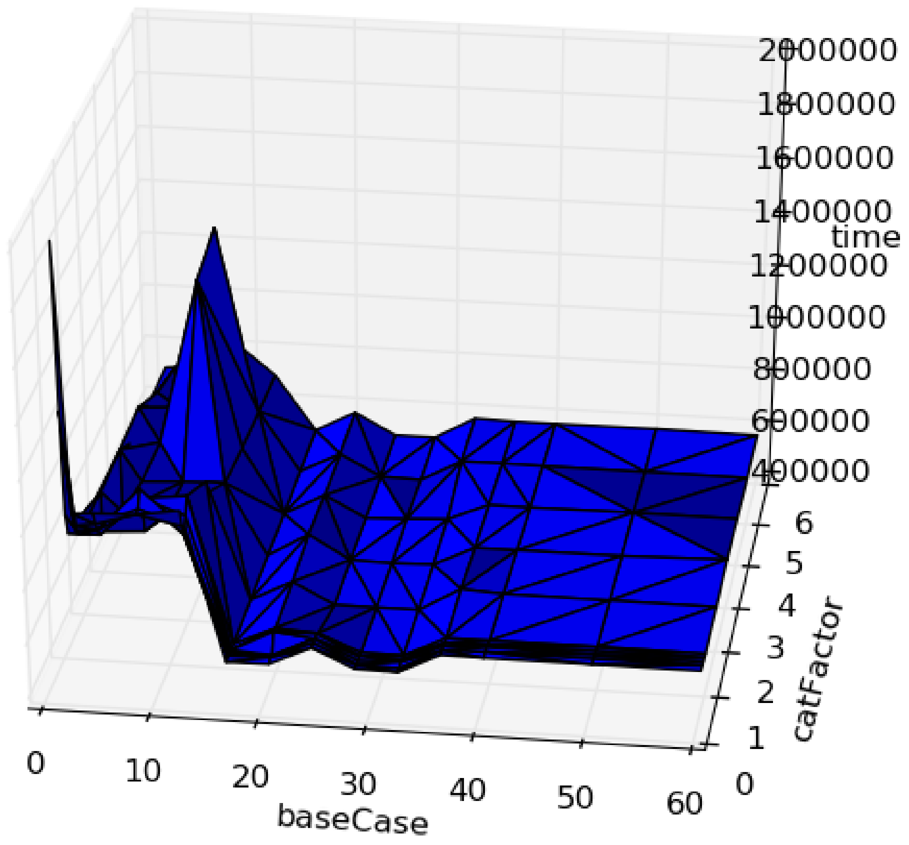

Figure 13.1: Graphical representation of the data in Table 13.1. Time is in microseconds.

Table 13.1: SEL4 parameter optimization data

	0.6	0.7	0.8	0.9	1.0	2.0	3.0	4.0	5.0	6.0
1	1948111	1333077	941053	857049	859050	770044	722041	706040	700040	698040
2	980056	921053	887051	857049	855049	837048	840048	847048	752043	702040
4	985057	932053	894051	859049	855049	768044	724041	718041	699040	694039
8	990057	994057	1051060	1002057	865050	758043	732042	1338076	1392080	813047
12	1013057	942054	893051	948055	1004057	898051	736043	713040	700040	691040
16	460026	466027	460026	455026	460027	461026	467027	468026	476028	483027
20	462027	470027	461026	534031	530030	539031	557032	549031	552032	561032
24	533030	534031	534030	535031	520030	461026	468027	469027	473027	481027
28	463027	461026	460026	460027	457026	460026	465027	475027	469027	481027
32	460027	458026	465026	456026	455026	460027	469027	483027	550032	564032
36	535031	535030	535031	530030	531031	549031	543031	547031	551032	560032
40	533030	539031	540031	533031	531030	540031	543031	546031	550032	560032
50	534030	537031	535030	532031	532030	539031	544031	549032	549031	562032
60	533031	536030	533031	531030	529031	538030	553032	548031	548032	559032

The numbers on the top are the numbers we multiplied our theoretical category with, and the numbers on the far left are the numbers that define our base case. Entries are runtime in microseconds on data size 1000000.

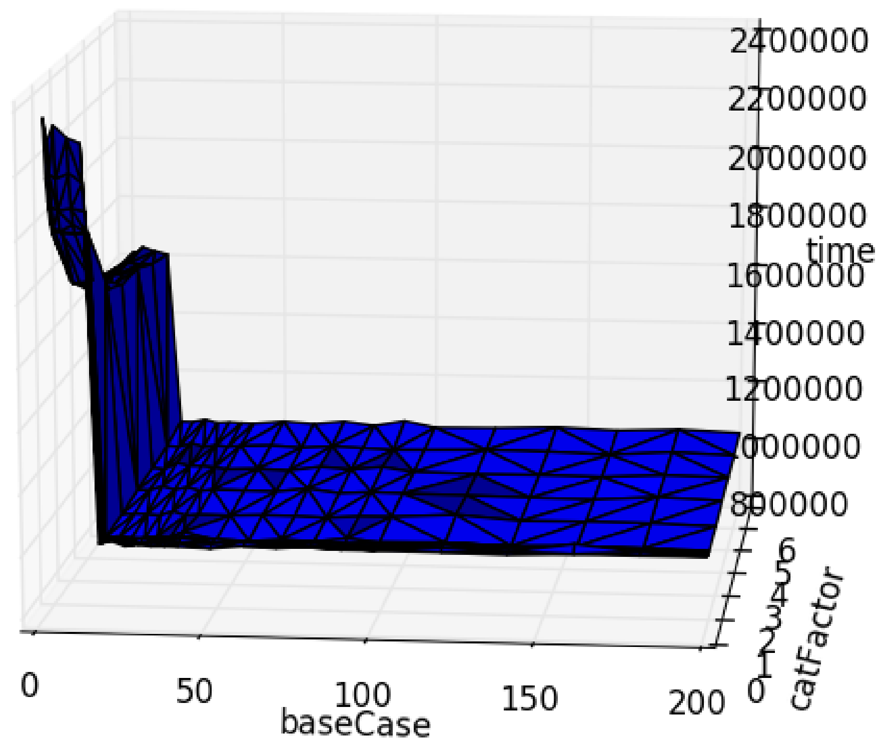

Figure 13.2: Graphical representation of the data in Table. Time is in microseconds. 13.1

Table 13.2: SEL4 parameter optimization data

	0.6	0.7	0.8	0.9	1.0	2.0	3.0	4.0	5.0	6.0
1	1001057	993057	989056	995057	990057	991057	1001057	1005057	1011058	1027059
2	1009058	1009057	999057	1001058	990056	995057	1001058	1001057	1008057	1027059
4	994057	996057	996057	989057	989056	994057	1015058	1024059	1022058	1036059
8	999058	997057	989056	997057	988057	991056	1004058	1014058	1010058	1025058
12	995057	999057	1000058	995057	997057	1006057	1014058	1015058	1016058	1028059
16	996057	1000057	991057	992057	996057	993057	1002057	1011058	1012058	1026058
20	995057	999057	1008058	1004057	1014058	997058	1008057	1006058	1016058	1025059
24	992056	1000058	996056	1001058	988056	996057	1007058	1010058	1020058	1035059
28	1002058	1009057	1002058	995057	988056	1000057	1006058	1006057	1014058	1028059
32	997057	998057	998057	999057	996057	999058	1018058	1021058	1021059	1039059
36	1005058	995056	1001058	1002057	992057	996057	1005057	1014058	1017058	1026059
40	997057	996057	996057	994057	1004057	1003058	1011058	1025058	1013058	1045060
50	997057	997057	994057	995057	994057	1004057	1009058	1010058	1015058	1024058
60	996057	1003058	1006057	1006058	997057	1015058	1007058	1007057	1020059	1027058

The numbers on the top are the numbers we multiplied our theoretical category with, and the numbers on the far left are the numbers that define our base case. Entries are runtime in microseconds on data size 1000000.

Chapter 14

Experiment heaps

14.1 Worst case

Image 14.1 shows the worst case test heap. The k smallest elements will form a right spine down the tree, and the left child of a node with a value of 1 will have a high value. The first left child will have a value higher than k (100 in the example), and its subtree will have the same value. The next left child of a 1 node will have one lower than the previous. This affects how the naive algorithm's internal priority queue shuffles around the nodes.

Looking at the tree, in the naive algorithm the first node we add to the priority queue PQ is the root. After we extract it, 100 and 1 are added. 1 is smaller than all other elements, so it is forced to traverse PQ all the way from the bottom to the top, swapping place with 100 in the internal array.

Next we extract 1, leaving only 100. We add 99, which is smaller than 100, and is thus moved to the front of the array. Then 1 is added, which again has to be moved all the way to the front.

Once we extract 1 again, we are left with 100 and 99. We first add 98 and then 1, both of which have to the moved all the way from the back to the front. This continues until k elements have been extracted and $2 \cdot k$ elements inserted, each insertion taking the worst case number of operations to complete.

14.2 Random case

The random case tree is an infinite tree where every node has a value equal to the value of their parent plus a random number in the range $[1, 100]$. These random numbers were generated using `std::uniform_int_distribution` from the Standard Template Library.

We expect the Random Case to be a slightly worse case than the Best Case, but not by a lot. The reason we have this expectation is because inserting random elements into a priority queue will by expectation not move the elements around a lot.

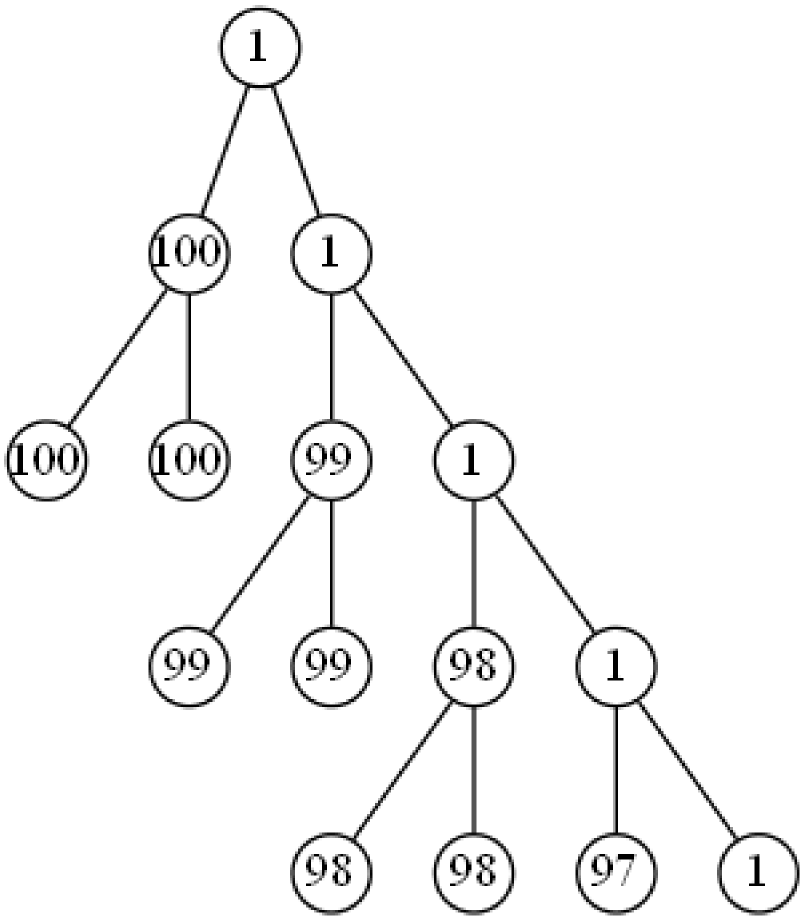

Figure 14.1: Worst case heap.

For our testing of the Random Case we should note that we have left out algorithm SEL2, because we disabled the Selection part of the algorithms, and without this SEL2 does not work at all. The reason for this is that since we are using random numbers the Selection part will vary by a lot on how much time it spends on this part, and therefore our results would be hard to explain. Additionally our experimental testing of the Selection algorithm indicated that we could not necessarily trust that it was linear time.

14.3 Best case

The best case tree is one where all nodes have the same value. Insertion and extraction of nodes never cause any swapping of nodes. In the implementation all nodes have the value 1.

This makes the naive algorithm linear, as insertions and extractions are now constant time operations.

It did not work as expected, see Chapter 15.

Chapter 15

Priority Queue Trouble

In our priority queues we have relied on `std::priority_queue` as mentioned earlier. We expected this implementation on insertions to insert the inserted element into the bottom right most part of the tree, and then bubble the element up through the tree until it has found its correct position. For extractMin (called `pop`), we expected it to remove the root, then insert the bottom right most element into the root and bubble that element down until it found its correct position.

It is with this in mind we created our test cases. That is why when inserting only elements with the same value we would expect the insert operation to be constant time, since we will not need to bubble any elements around. We expected the same for extraction. Essentially, the bubble up and bubble down procedures should be constant time.

Looking at Figure 15.1 we see no indication that the insert operation is not constant as expected. We get two jumps, one at $k = 2^{13}$ and one at $k = 2^{20}$. The machine that ran the test had 32 KB L1 data cache. The elements inserted in the test each take up 4 B, leaving room for:

$$\frac{32 \cdot 1024}{4} = 2^{13}$$

The machine also had 4 MB L3 data cache, leaving room for:

$$\frac{4 \cdot 1024 \cdot 1024}{4} = 2^{20}$$

Incidentally, the two most noticeable jumps are at $k = 2^{13}$ and $k = 2^{20}$ in Figure 15.1. Before, between, and after these peaks it looks reasonably constant.

If we look at Figure 15.2, the extractMin operation does not look very constant. Outside of the $k = 2^{20}$ L3 data cache jump, it looks like a growing linear function. This could be an indication of a $O(n \cdot \log n)$ bound, where n is the number of elements in the priority queue.

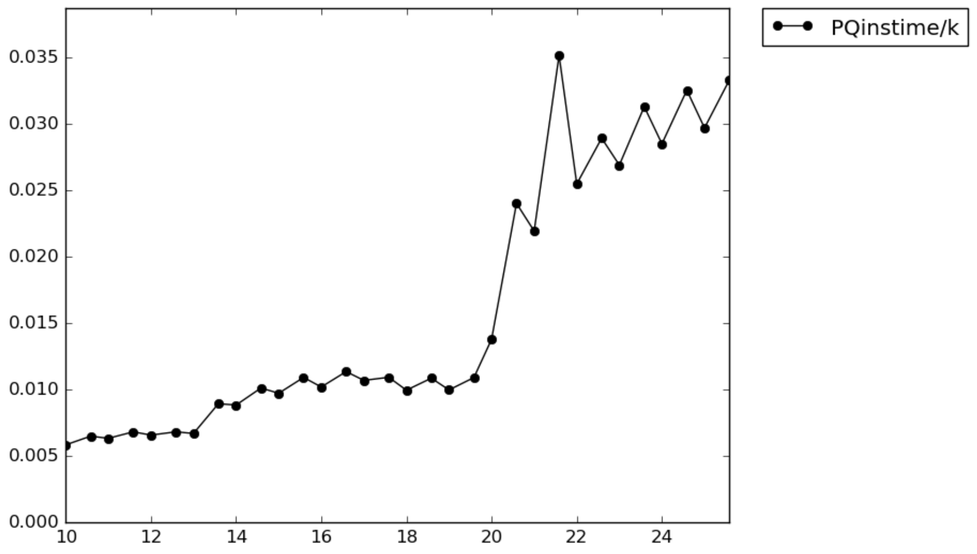

Figure 15.1: Insertions into `std::priority_queue` divided by k over $\log k$. Every element inserted had value 1.

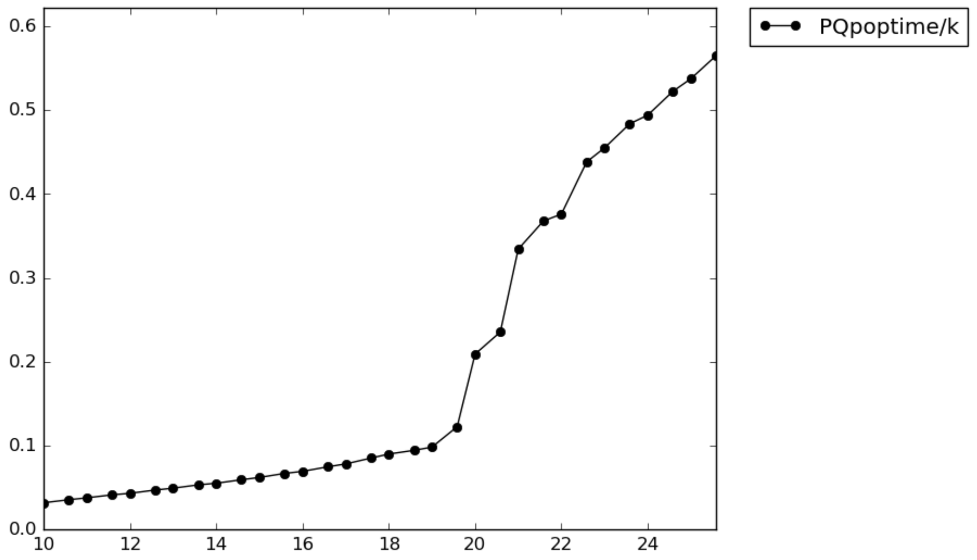

Figure 15.2: Pops from `std::priority_queue` divided by k over $\log k$. Every popped element had value 1.

This seem to indicate that the pop operation does not work as expected. Additionally we had a look into the GCC (GNU Compiler Collection, gcc.gnu.org) implementation, which is the one we use, and while we do not fully understand how it works, it definitely does not work according to our original expectation.

This insight came late during the project, and unfortunately we did not have the time to write our own version, insert it everywhere, and rerun all the tests.

Chapter 16

Experimental results

16.1 System specification

The system used to run the tests was rented on Amazon Web Services (aws.amazon.com) where it was named *m4.xlarge*. It had the following specs:

- Architecture: x86_64
- Clock speed: 2394 MHz
- CPUs: 8
- RAM: 16 GB
- L1 instruction cache: 32KB
- L1 data cache: 32KB
- L2 data cache: 256KB
- L3 data cache: 30 MB
- Operating system: Ubuntu 14.04 LTS

16.2 Runtime

Runtime was measured in microseconds using `std::chrono::high_resolution_clock` from the Standard Template Library. Runtimes that measured below 100,000 microseconds were run again as follows:

1. $begin := now()$
2. $rounds := 1$
3. $completed := 0$

```
4. $time := 0$
5. Loop:
    a) Run algorithm.
    b) $completed := completed + rounds$
    c) $rounds := rounds * 2$
    d) $time := now() - begin$
    e) If $time > 100000$:
        i. $time := \frac{time}{completed}$
        ii. Break.
```

Essentially, the benchmark continuously doubled the number of times an algorithm was run until it took at least 100,000 microseconds, then divided the total time by the total number of runs. This was done to scale up the fastest benchmarks and make them measurable.

The number of extra runs grows exponentially, to reduce the overhead of checking if enough time had passed.

For the runtime tests, the fastest of five runs was kept. The rationale is that many things can contribute noise to the runtime result, but almost none of that noise make the runtime faster. The ticking clock can have a significant effect, if for instance the start and end measurements happen to be the same, the total time is 0. However, this affects the shorter runtimes the most and was mitigated with the extra runs described above.

Worst case

Figure 16.1 shows the runtime of all five algorithms in microseconds divided by k over $\log k$. Just like before, SEL2 and partly SEL3 are the slowest. A jump is visible in both at 2^{16}, where an extra layer of recursion is added. Both are otherwise horizontal, which implies a linear growth, aside from when $\log^*$ changes value. This is because a time complexity of $O(k \cdot 3^{\log^* k})$ is practically linear. $\log^*$ increases in increments of one at the values ${}^{2}2 = 2^2 = 4$, ${}^{3}2 = 2^4 = 16$, and ${}^{4}2 = 2^{16} = 65536$. The next increment will be at ${}^{5}2 = 2^{65536}$, which is way outside of practical reach.

Figure 16.2 shows the runtime of the three fastest algorithms in microseconds divided by k over $\log k$. With the noise of the slower algorithms cut out, we can see both SEL1 and SEL4 become horizontal as k grows. SEL1's time complexity is $O(k \cdot \log\log k)$, which is practically indistinguishable from $O(k)$: For input size below 2^{32}, $\log\log k$ never exceeds 5. If we increase the input size to 2^{64}, we only increase $\log\log k$ to 6.

Naive is not horizontal due to $\log k$ not being insignificant in the data sizes we're able to test. The test design ensures that the internal priority queue

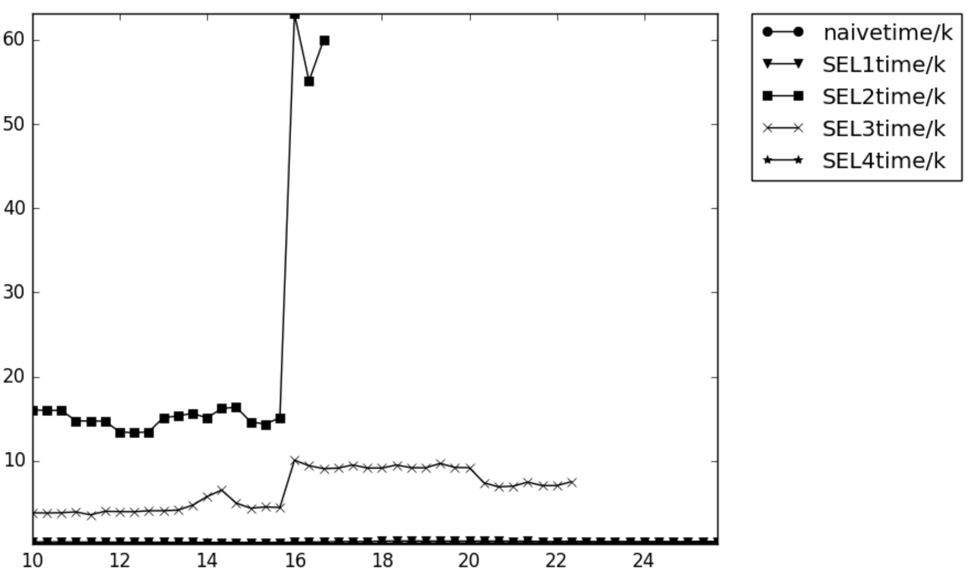

Figure 16.1: Runtime in microseconds divided by k over $\log k$. Worst case tree.

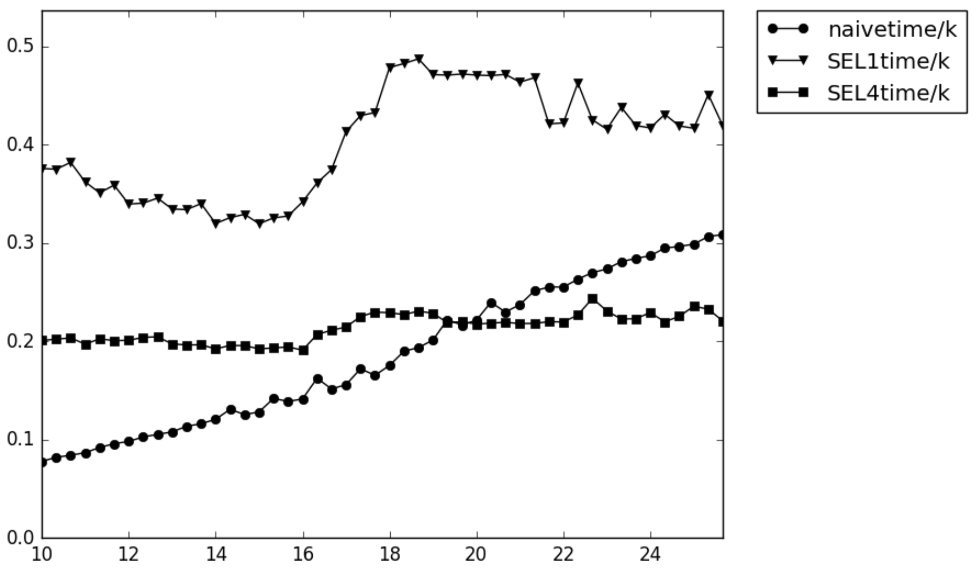

Figure 16.2: Runtime in microseconds divided by k over $\log k$. Worst case tree.

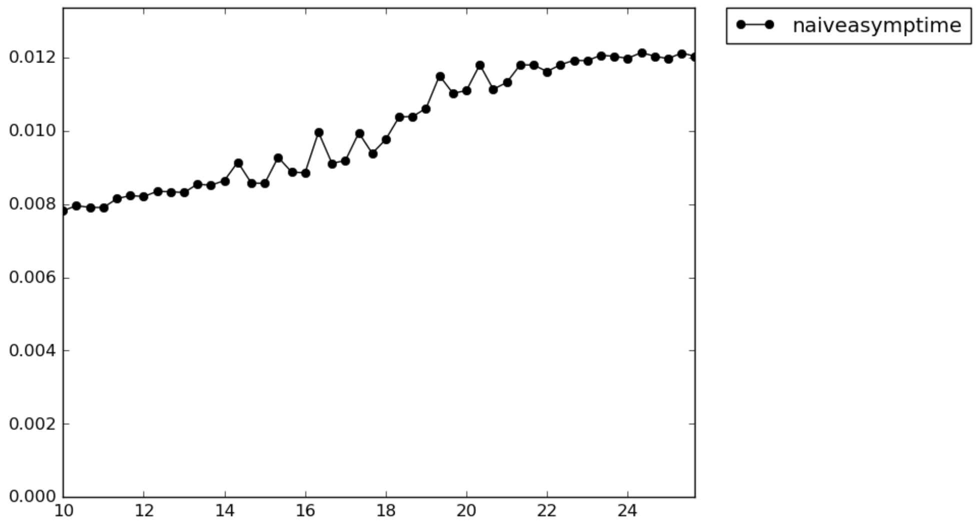

Figure 16.3: Runtime in microseconds divided by $k \cdot \log k$ over $\log k$. Worst case tree.

used by the naive algorithm always exhibits its worst case time complexity, see chapter 14. At $k = 2^{20}$ SEL4, with its linear complexity, starts beating the naive algorithm in execution time.

Figure 16.3 shows the naive algorithm with runtime in microseconds divided by the theoretical bound of $O(k \cdot \log k)$. It does not form a horizontal line. Figure 17.10 shows the L3 data cache misses for the naive algorithm, which stabilizes at around $k = 2^{20}$. The runtime graph also stabilizes at that point, and it is likely that cache misses are to blame for the increase in runtime before $k = 2^{20}$. This supports that the theoretical bound holds for the implementation.

Random case

As mentioned in 14, SEL2 is not run in any of the tests with the random case graph.

Figure 16.4 shows the runtime of the four algorithms in microseconds divided by k over $\log k$. Like the worst case tree, SEL3 is the slowest, but otherwise appears linear outside of $k = 2^{16}$.

Figure 16.5 shows the runtime of the three fastest algorithms in microseconds divided by k over $\log k$. In comparison to the worst case tree in Figure 16.2, SEL4 climbs from 2^{16} to 2^{24}. Some of this can be explained by more L3 data cache misses, which can be seen in Figure 17.21. The test worst case tree has fewer L3 data cache misses, while the random tree has more. The naive algorithm also performs better on random data compared to a tree that was created to get worst case performance on all operations the naive algorithm relies on.

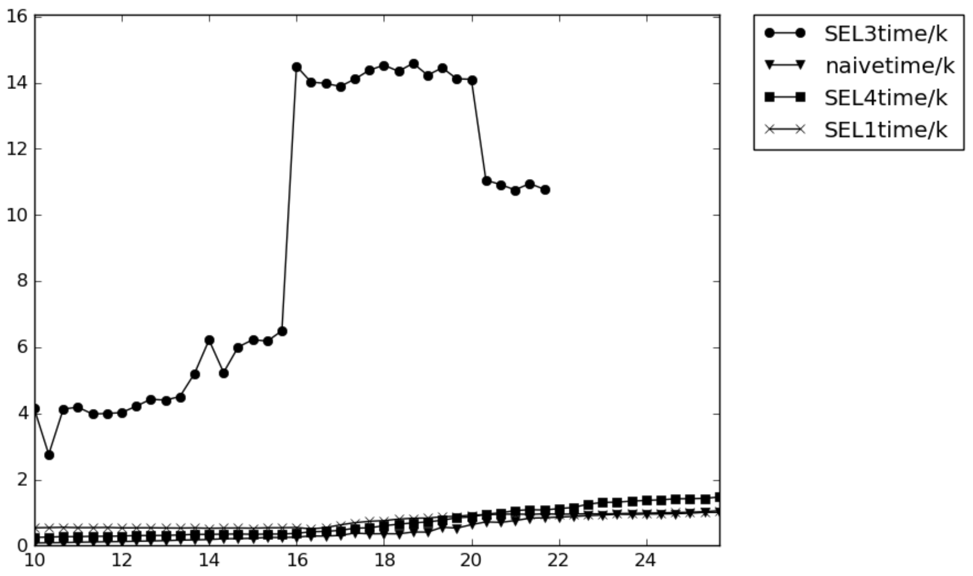

Figure 16.4: Runtime in microseconds divided by k over $\log k$. Random tree.

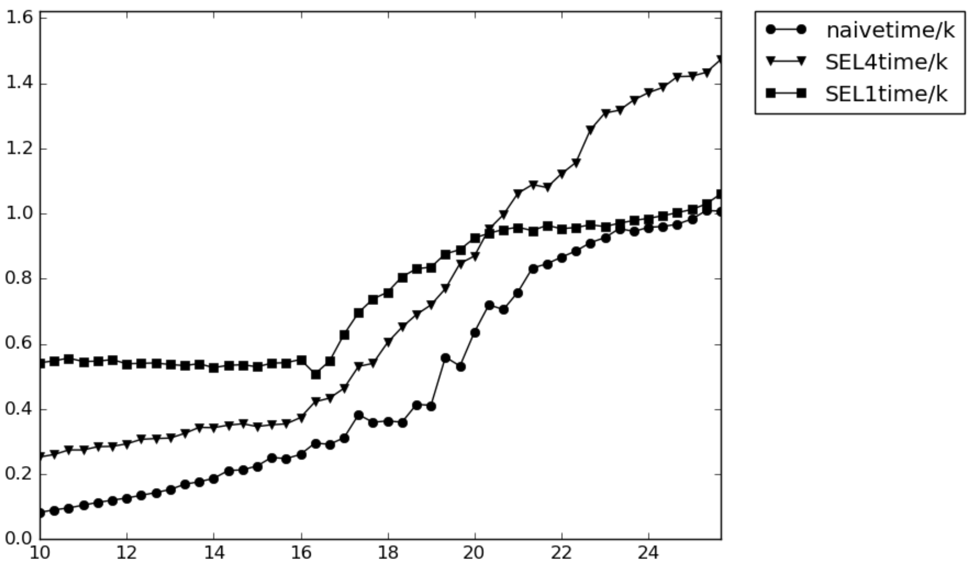

Figure 16.5: Runtime in microseconds divided by k over $\log k$. Random tree.

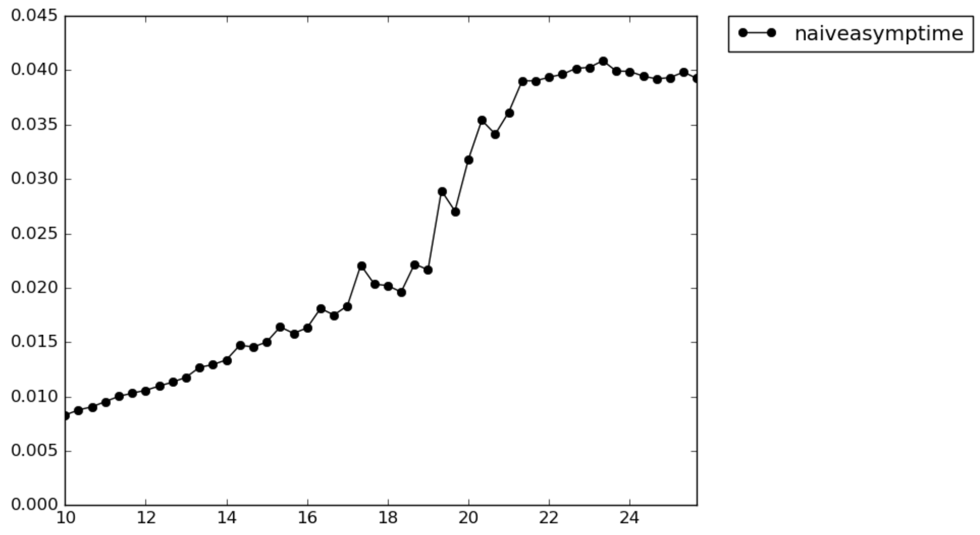

Figure 16.6: Runtime in microseconds divided by $k \cdot \log k$ over $\log k$. Random tree.

Figure 16.6 shows the naive algorithm with runtime in microseconds divided by the theoretical bound of $O(k \cdot \log k)$. Like the worst case graph, it does not form a horizontal line before $k = 2^{21}$. Figure 17.21 shows the L3 data cache misses for the naive algorithm, which starts to flatten at $k = 2^{22}$. From this we can attribute some of the growth before $k = 2^{21}$ to increasing L1 and L3 data cache misses.

Best case

Figure 16.7 shows the runtime of all five algorithms in microseconds divided by k over $\log k$. Like the worst case tree, SEL2 and SEL3 are the slowest, and appear otherwise linear outside of $k = 2^{16}$.

Figure 16.8 shows the runtime of the three fastest algorithms in microseconds divided by k over $\log k$. In comparison to the worst case tree in Figure 16.2, SEL4 has some climb from 2^{15} to 2^{17}, but remains constant otherwise.

SEL1 climbs from 2^{15} to 2^{19} and again climbs around 2^{22}. Outside of those regions it remains mostly constant.

The naive algorithm does not appear constant. An array based priority queue can be inserted into and extracted from in constant time when all elements are equal. Something affects the runtime, but unfortunately we do not know what.

The naive algorithm is faster than SEL4 for all data points we collected. The missing data points for SEL1 and SEL4 are due to the machine running out of memory on the best case tree earlier than on the worst case tree.

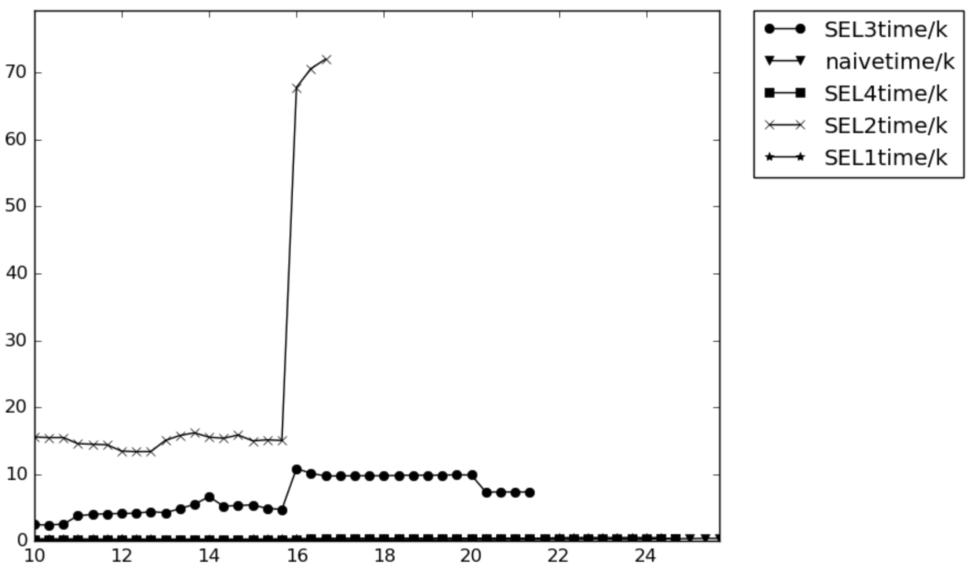

Figure 16.7: Runtime in microseconds divided by k over $\log k$. Worst case tree.

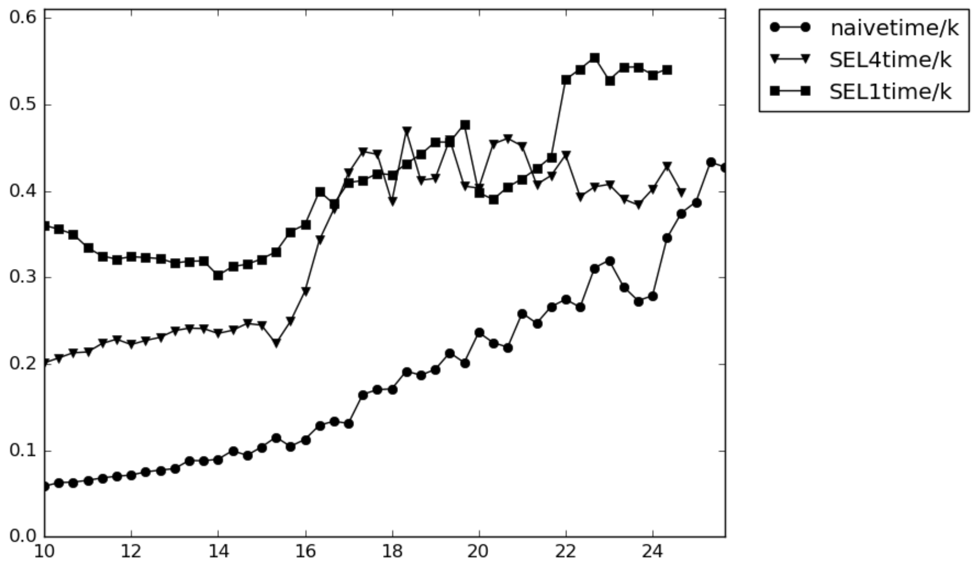

Figure 16.8: Runtime in microseconds divided by k over $\log k$. Best case tree.

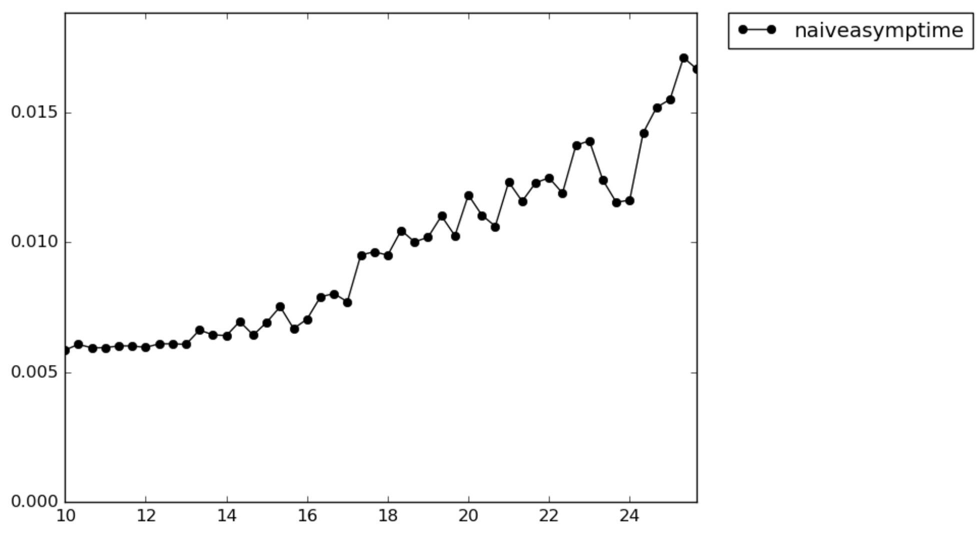

Figure 16.9: Runtime in microseconds divided by $k \cdot \log k$ over $\log k$. Best case tree.

Figure 16.9 shows the naive algorithm with runtime in microseconds divided by the theoretical bound of $O(k \cdot \log k)$. Like the worst case graph, it does not form a horizontal line. Figure 17.32 shows the L3 data cache misses for the naive algorithm, but it does not help explain the growth.

16.3 Comparisons

Comparisons were measured only in a single run, since any random numbers were created from a fixed seed and the algorithms are otherwise deterministic.

Worst case

Figure 16.10 shows horizontal lines for SEL2 and SEL3, except for $k = 2^{16}$.

Figure 16.11 shows SEL1 and SEL4 are less horizontal than SEL2 and SEL3. It's hard to say if they converge. Naive clearly does not converge on the graph, which supports its $O(k \cdot \log k)$ bound.

Random case

Figure 16.12 shows horizontal line for SEL3, except for $k = 2^{16}$.

Figure 16.13 shows SEL1 and SEL4 are very close to horizontal lines. Compared to Figure 16.11, all three algorithms appear to have a lower overhead. This is likely caused by the operation of inserting a random element into a priority queue not always being log of the size of the queue.

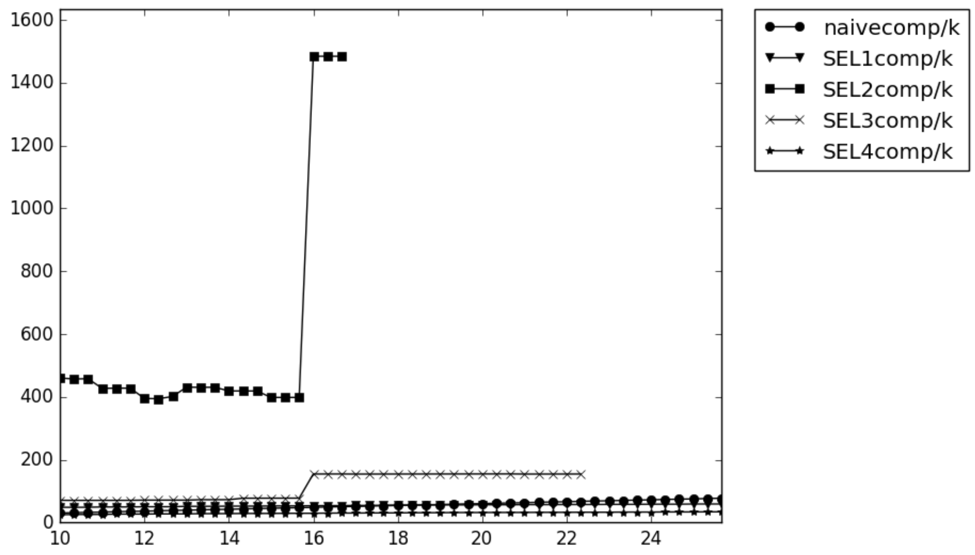

Figure 16.10: Comparisons divided by k over $\log k$. Worst case tree.

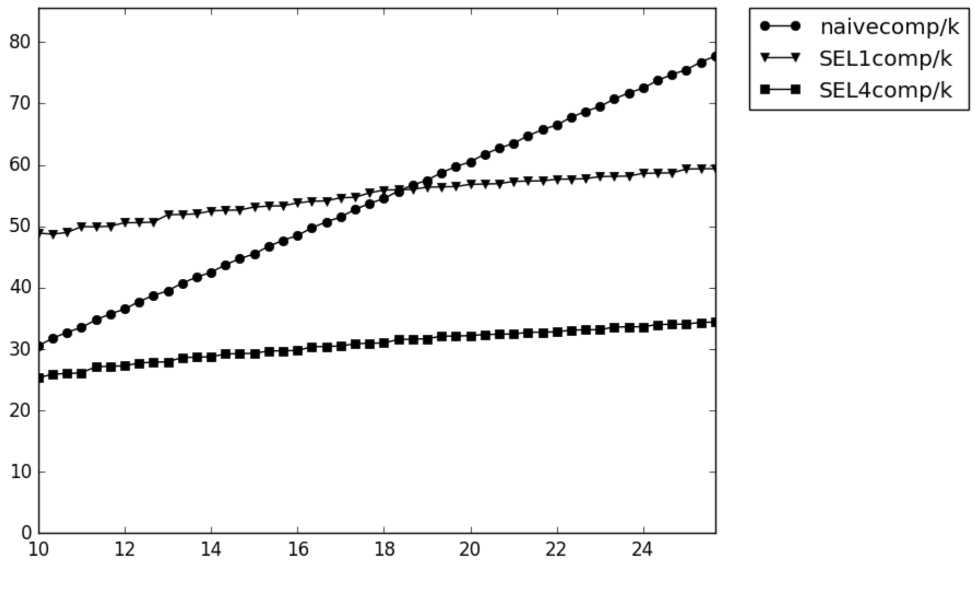

Figure 16.11: Comparisons divided by k over $\log k$. Worst case tree.

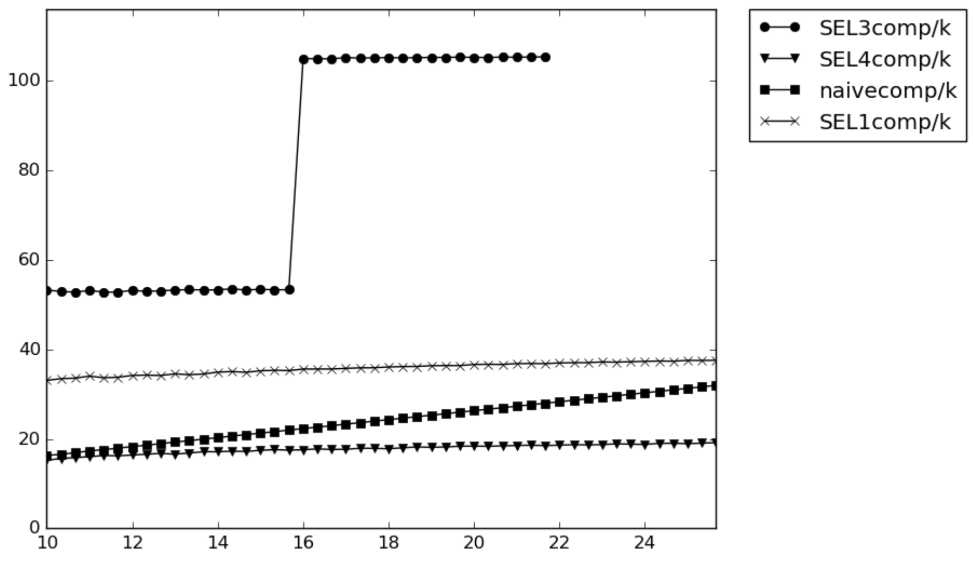

Figure 16.12: Comparisons divided by k over $\log k$. Random tree.

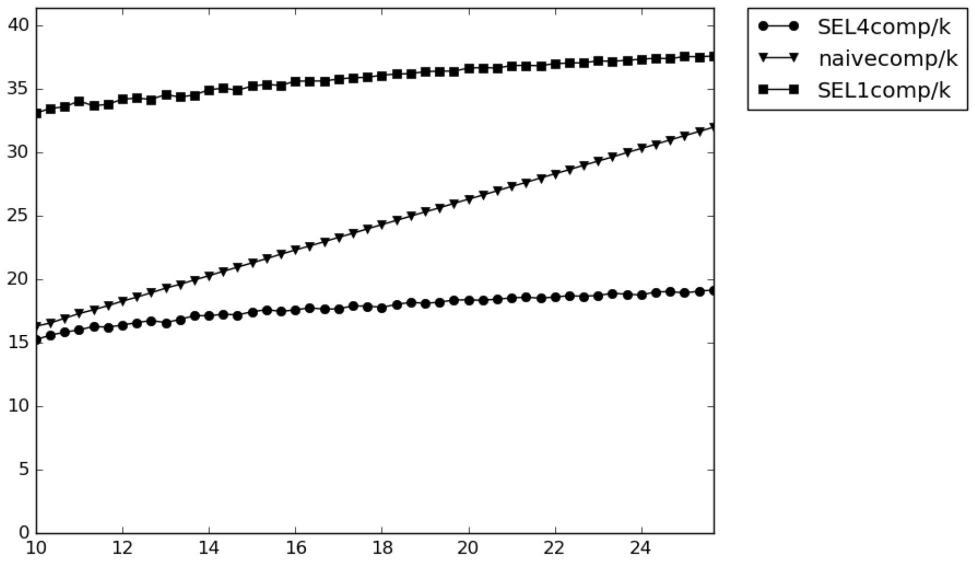

Figure 16.13: Comparisons divided by k over $\log k$. Random tree.

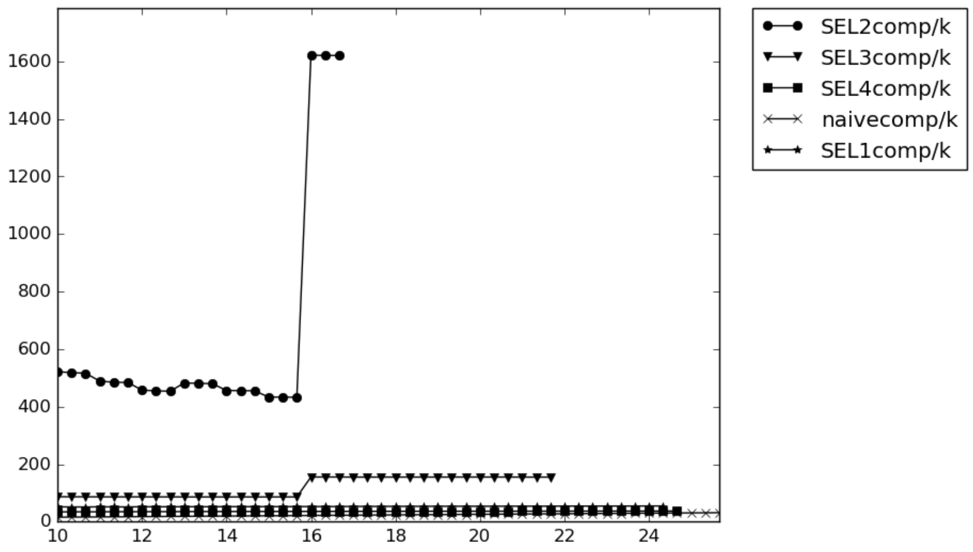

Figure 16.14: Comparisons divided by k over $\log k$. Best case tree.

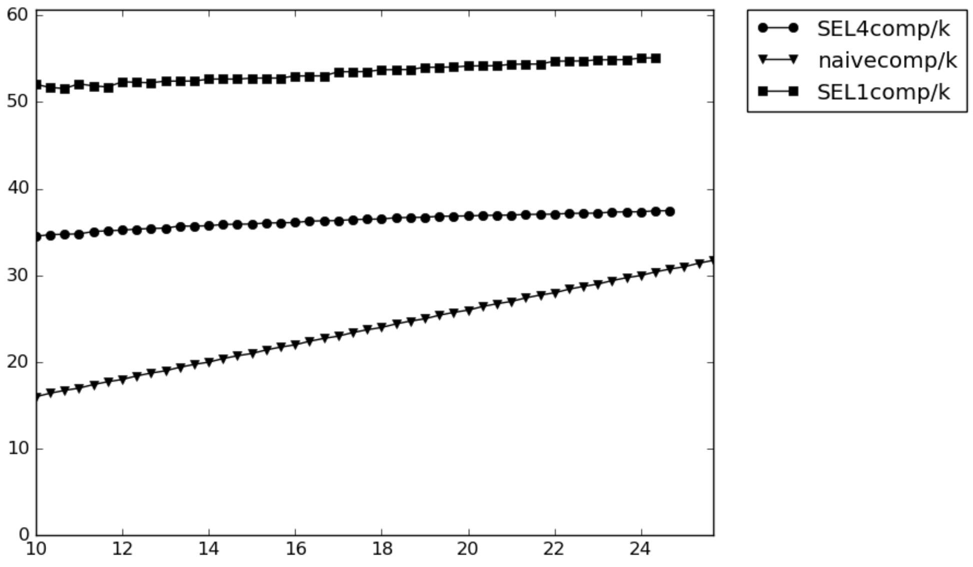

Figure 16.15: Comparisons divided by k over $\log k$. Best case tree.

Best case

Figure 16.14 shows horizontal lines for SEL2 and SEL3, except for $k = 2^{16}$.

Figure 16.15 shows SEL1 and SEL4 are considerably closer to horizontal lines than in Figure 16.11. The naive algorithm is still not horizontal, but it has much smaller values than in the worst case tree. SEL4 makes more

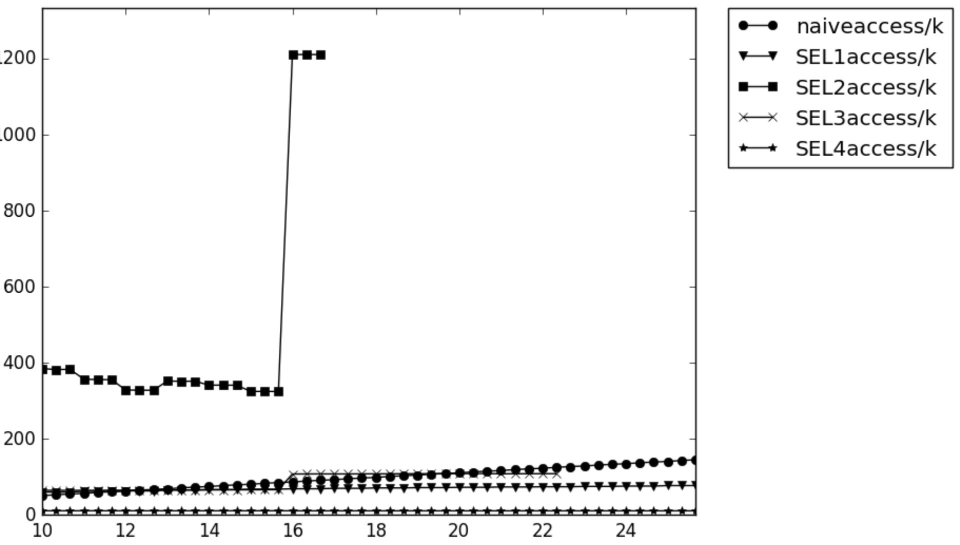

Figure 16.16: Accesses divided by k over $\log k$. Worst case tree.

comparisons than in the worst case tree, while SEL1 remains the same.

16.4 Accesses

Accesses were measured only in a single run, since any random numbers were created from a fixed seed and the algorithms are otherwise deterministic.

Worst case

Figure 16.16 shows SEL2 being close to linear with respect to the number of accesses, again except for $k = 2^{16}$.

Figure 16.17 shows the number of accesses done by the naive algorithm, SEL1, SEL3, and SEL4 divided by k. SEL3 is horizontal outside of $k = 2^{16}$ and SEL4 is horizontal all the way. It's hard to say if SEL1 converges, while the naive clearly does not.

Figure 16.18 shows the naive algorithm with accesses and comparisons divided by the theoretical bound of $O(k \cdot \log k)$. Both comparisons and accesses seem to be constant, supporting the theoretical bound.

Random case

Figure 16.19 shows horizontal lines for SEL3, except for $k = 2^{16}$. SEL1 and SEL4 also seem mostly horizontal. Note that compared to Figure 16.17, the overhead of all but SEL4 are much lower. This is likely part of the reason the

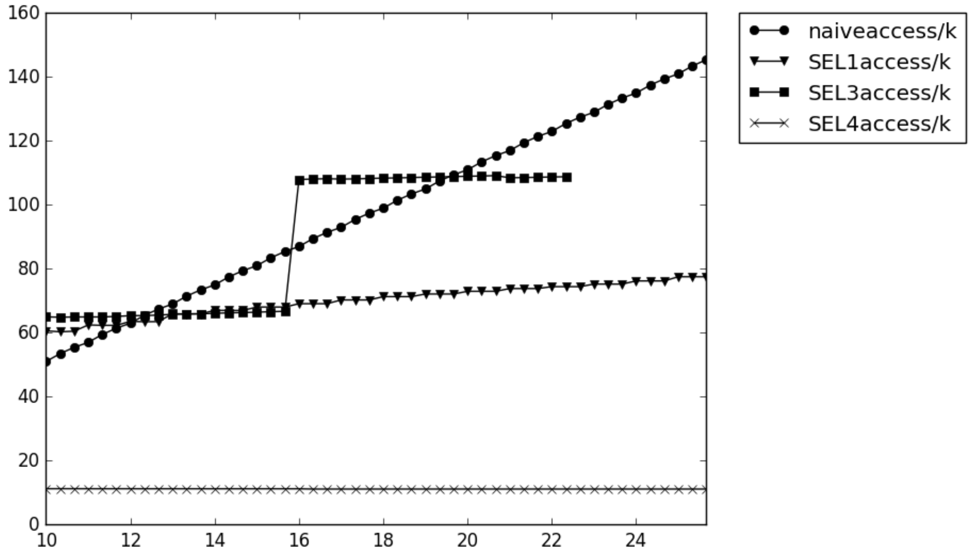

Figure 16.17: Accesses divided by k over $\log k$. Worst case tree.

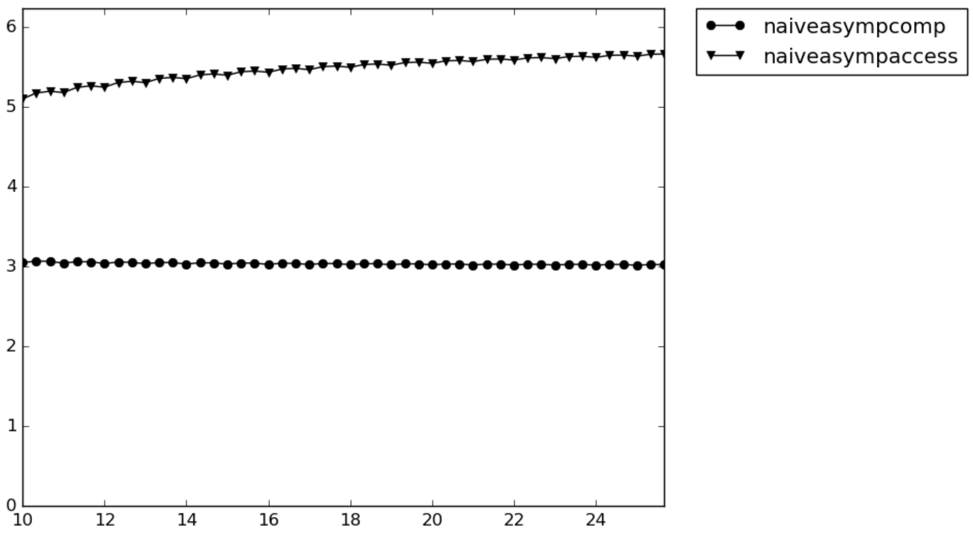

Figure 16.18: Accesses and comparisons divided by $k \cdot \log k$ over $\log k$. Worst case tree.

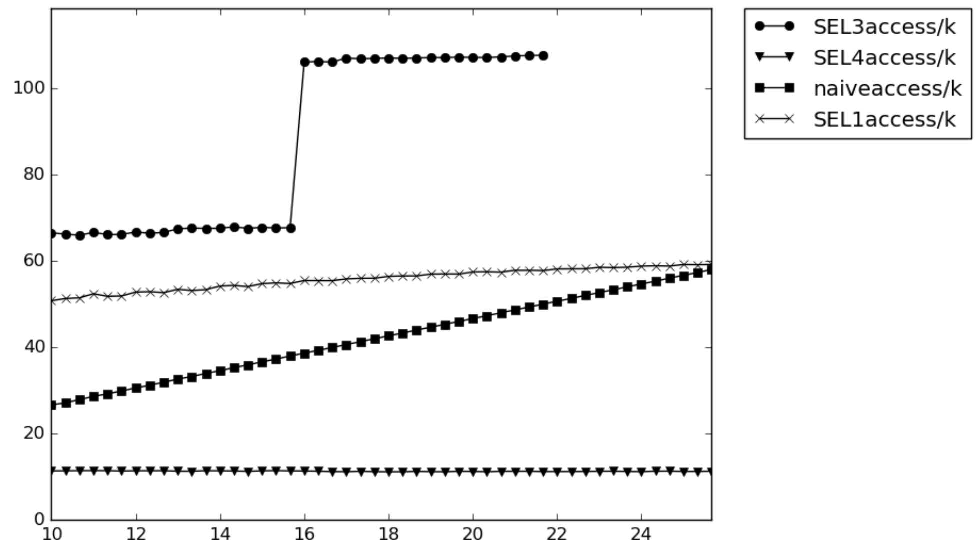

Figure 16.19: Accesses divided by k over $\log k$. Random tree.

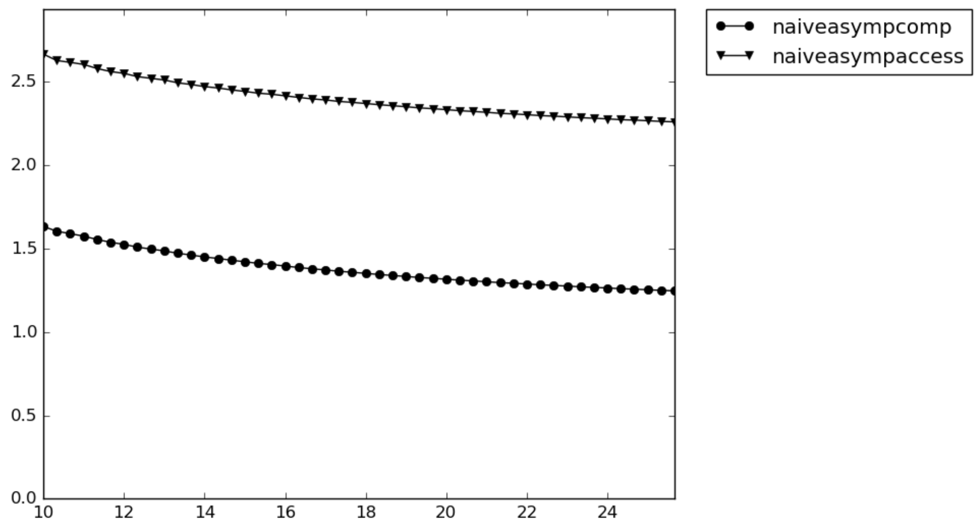

Figure 16.20: Accesses and comparisons divided by $k \cdot \log k$ over $\log k$. Random tree.

SEL4 does not get faster than the naive algorithm in the tests. See Figure 16.5.

Figure 16.20 shows the naive algorithm with accesses and comparisons divided by the theoretical bound of $O(k \cdot \log k)$. Like the worst case graph, both comparisons and accesses seem to be constant, supporting the theoretical bound.

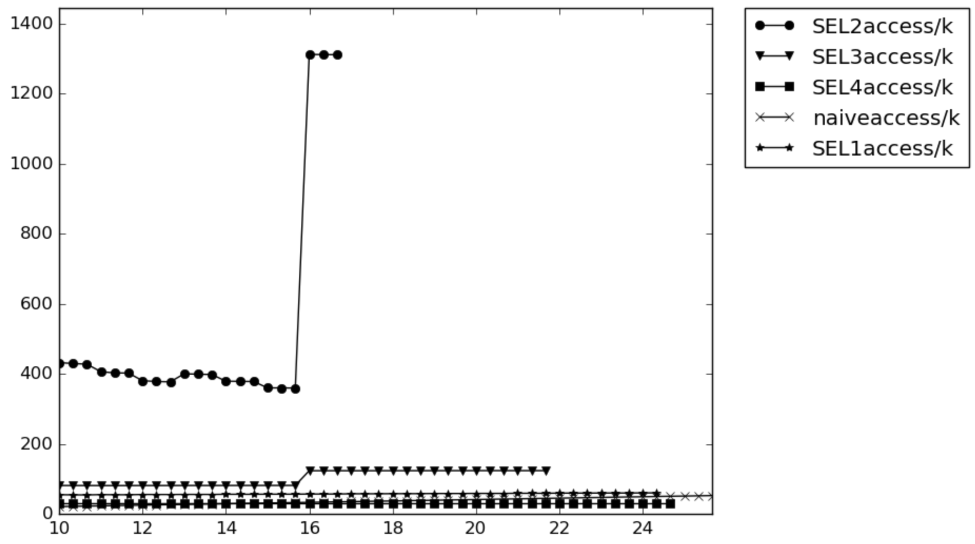

Figure 16.21: Accesses divided by k over $\log k$. Best case tree.

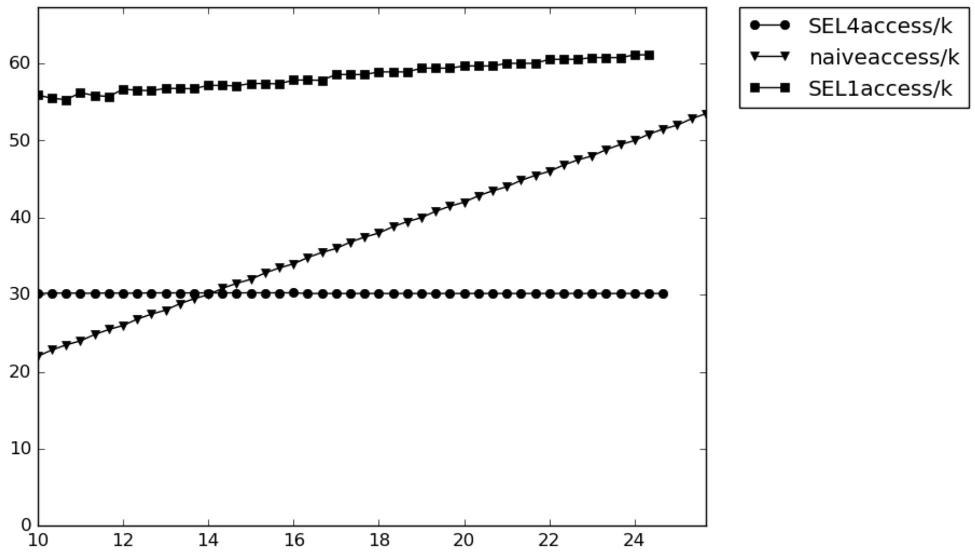

Figure 16.22: Accesses divided by k over $\log k$. Best case tree.

See Figure 16.6 for the timing.

Best case

Figure 16.21 shows horizontal lines for SEL2 and SEL3, except for $k = 2^{16}$.

Figure 16.22 shows a horizontal line for SEL4, and a near horizontal line for SEL1. The naive algorithm appears to do more than a linear number of

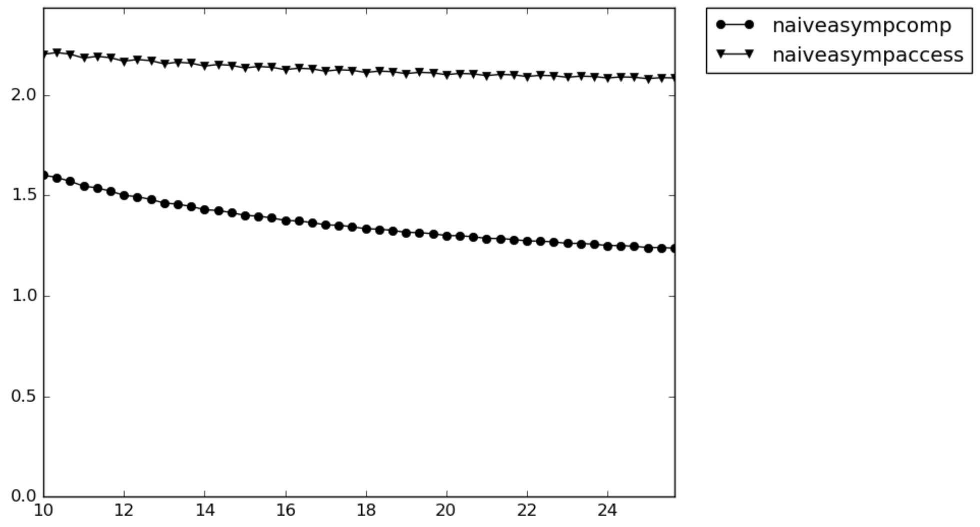

Figure 16.23: Accesses and comparisons divided by $k \cdot \log k$ over $\log k$. Best case tree.

accesses, despite the shape of the best case tree. We can not give a reasonable cause for that, however.

Compared to Figure 16.17, SEL4 does almost three times as many accesses and the naive algorithm does starts lower and grows at a slower pace.

Figure 16.23 shows the naive algorithm with accesses and comparisons divided by the theoretical bound of $O(k \cdot \log k)$. Like the worst case graph, both comparisons and accesses seem to be constant, supporting the theoretical bound.

16.5 Conclusion

Worst case

SEL4 is the fastest algorithm when k gets large enough. It also has the lowest number of accesses and comparisons. The naive algorithm is the second fastest, with SEL1 following closely behind.

SEL2 and SEL3 both seem to follow their theoretical time complexity very closely: SEL2 with the bound of $O(k \cdot 3^{\log^*(k)})$ seem to increase its number of accesses, comparisons, and its runtime by a factor of 3 at $k = 2^{16}$, which is where $\log^*$ jumps from 4 to 5. Similarly SEL3, with its theoretical bound of $O(k \cdot 2^{\log^*(k)})$ seem to double its accesses, comparisons, and runtime at the same point.

Random case

SEL4 gets performs poorly and never catches up to SEL1 or the naive algorithm. Some of the issues can be attributed to data cache misses being more frequent than in the worst case tree tests.

The naive algorithm performs well on random data. SEL1 starts off worse, climbs with its L3 data cache misses, and ends up being as fast as the naive algorithm at the end.

Best case

The naive algorithm is the fastest, though it looks like if more data points existed for SEL4, they might have been on par at $k = 2^{25}$. This was expected, though the naive algorithm was also expected to be linear, which it is not on the runtime graphs. This is explained in Chapter 15.

SEL1, SEL2, SEL3, and SEL4 behave largely the same way as in the worst case.

Chapter 17

Valgrind Results

The Linux tool Valgrind was used to profile the implementations in terms of the number of instructions used, the number of memory lookups, and the number of L1 and L3 data cache misses. Valgrind is also able to profile branches and branch mispredicts. However, due to time constraints the tests have not been instrumented with a branch simulator. Branch simulation is very time consuming, and without it some tests already ran for over 20 hours.

Valgrind is available at `valgrind.org`

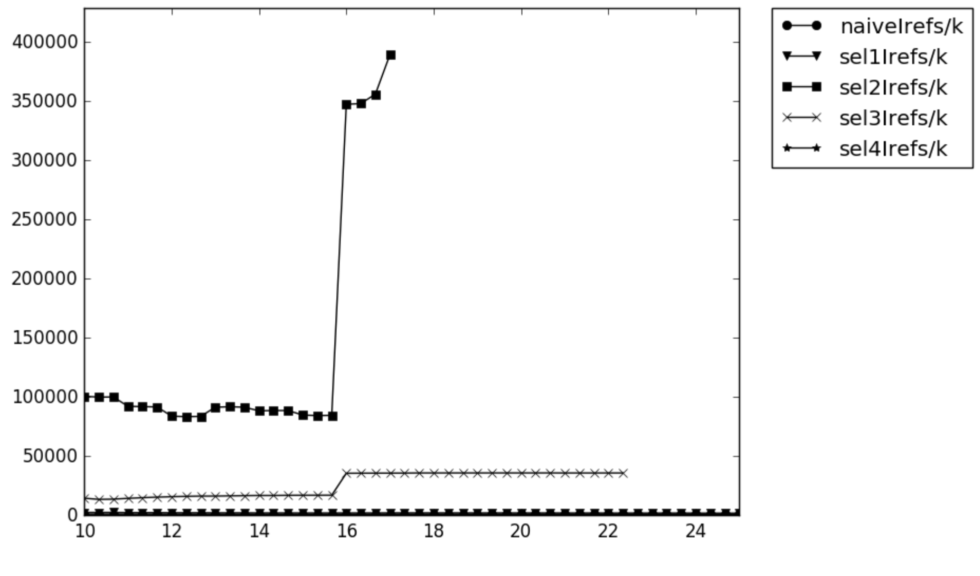

Figure 17.1: Instruction references divided by k over $\log k$. Worst case tree.

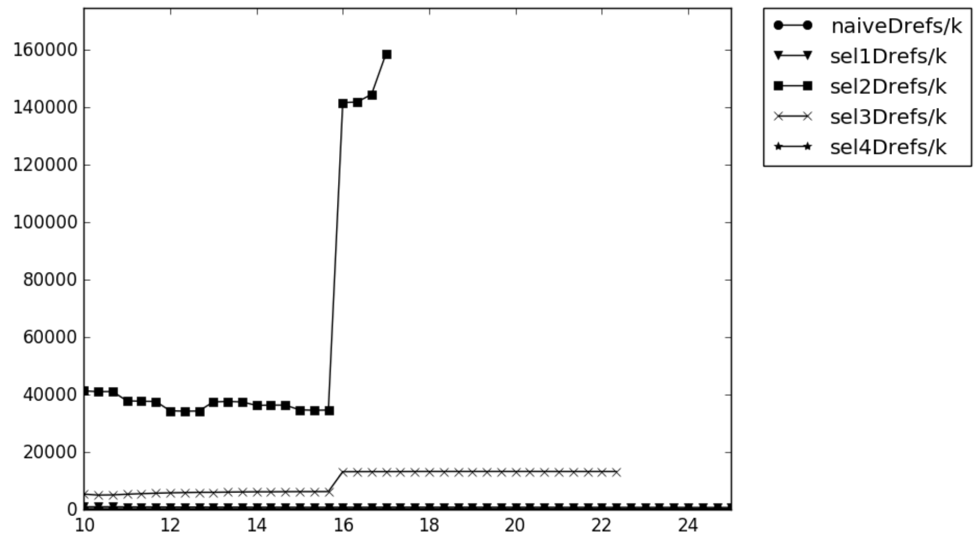

Figure 17.2: Data references divided by k over $\log k$. Worst case tree.

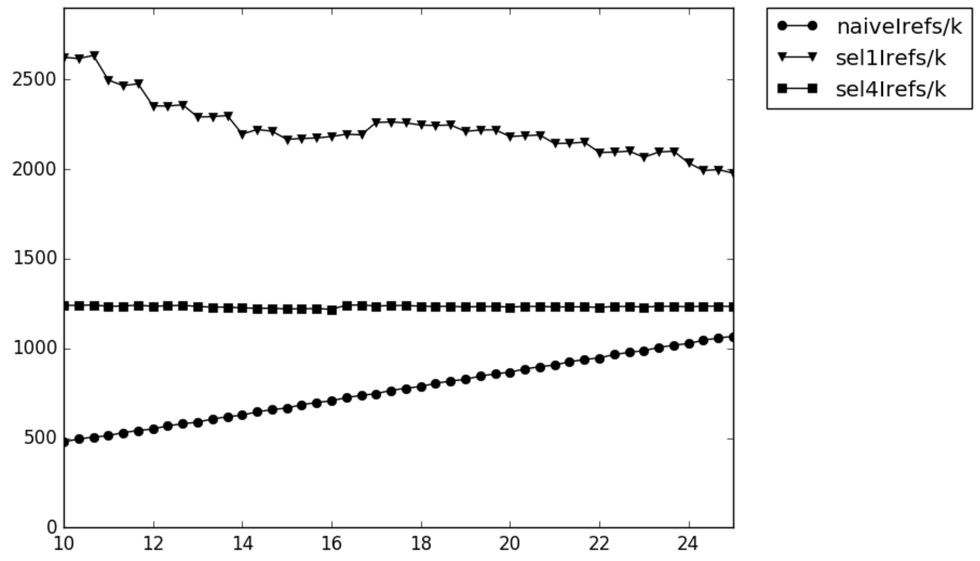

Figure 17.3: Instruction references divided by k over $\log k$. Worst case tree.

17.1 Worst case

Figure 17.1 shows instruction references over $\log k$. Figure 17.2 shows data references over $\log k$. Both are very similar graphs to Figure 16.1 and show that runtime of SEL2 and SEL3 are mostly determined by the number of instruction they use and the number of data references they make.

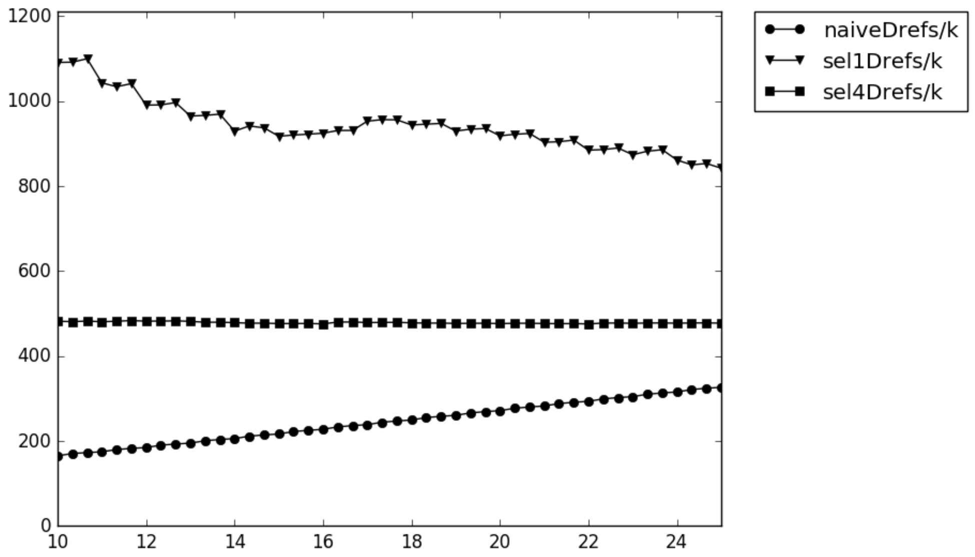

Figure 17.4: Data references divided by k over $\log k$. Worst case tree.

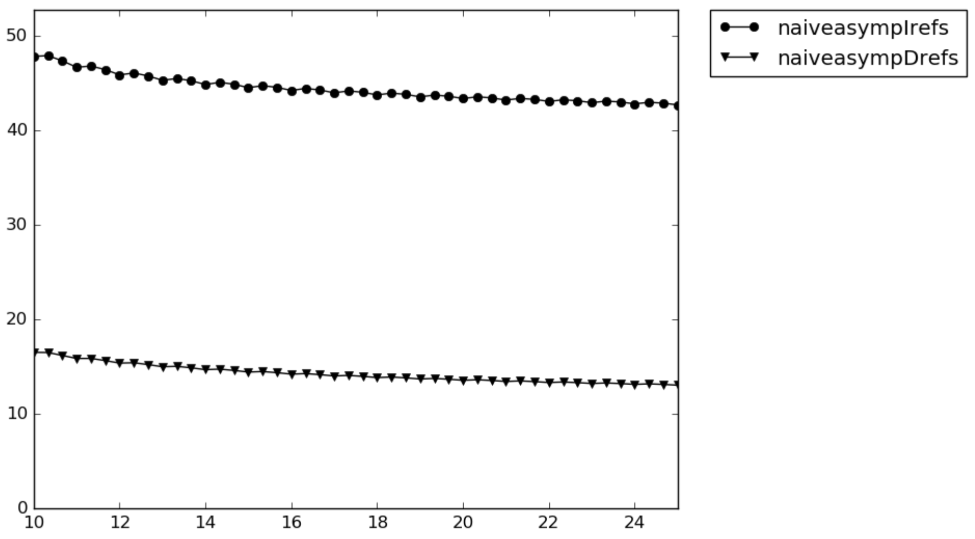

Figure 17.5: Instructions and data references divided by $k \cdot \log k$ over $\log k$. Worst case tree.

Figure 17.3 shows instruction references over $\log k$ for the fastest three algorithms. Figure 17.4 shows data references over $\log k$ for the same algorithms. Both are similar. SEL1 and SEL4 appear mostly constant in both graphs. The naive algorithm appears to climb, though it is uses less instructions and references memory less than SEL4 does for all data points.

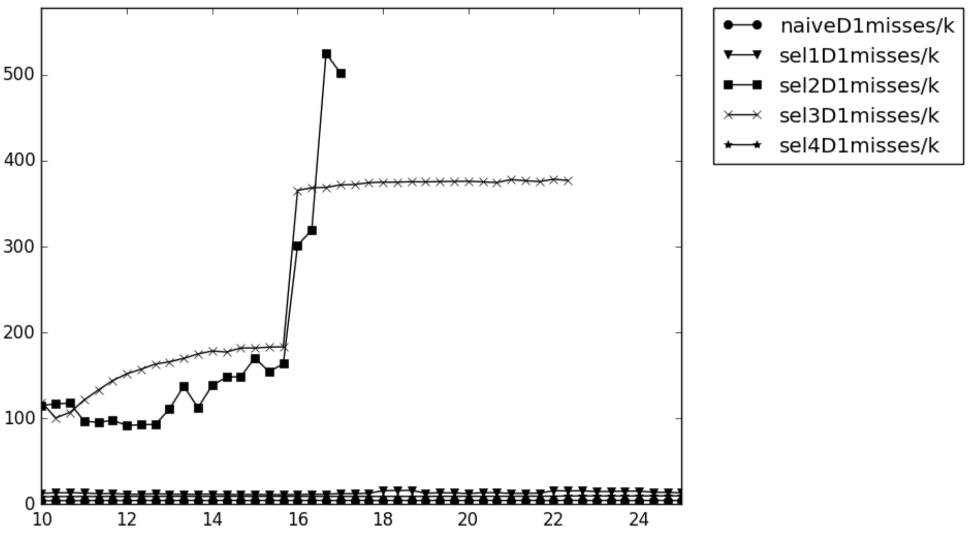

Figure 17.6: L1 data cache misses divided by k over $\log k$. Worst case tree.

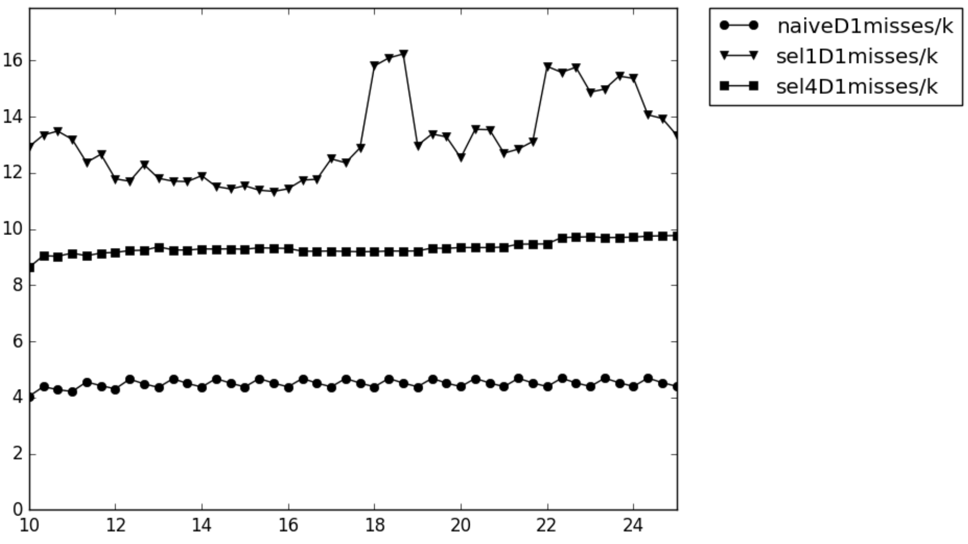

Figure 17.7: L1 data cache misses divided by k over $\log k$. Worst case tree.

Figure 17.5 shows the number of instructions and data references used by the naive algorithm divided by $k \cdot \log k$. They both seem to converge and support that the implementation follows the theoretical bound.

Figure 17.6 shows L1 data cache misses divided by k for all five algorithms. SEL3 appears mostly constant after it settles after $k = 2^{16}$. SEL2 does not have enough data points to support any conclusions.

Figure 17.7 shows the same for the naive algorithm, SEL1, and SEL4. All

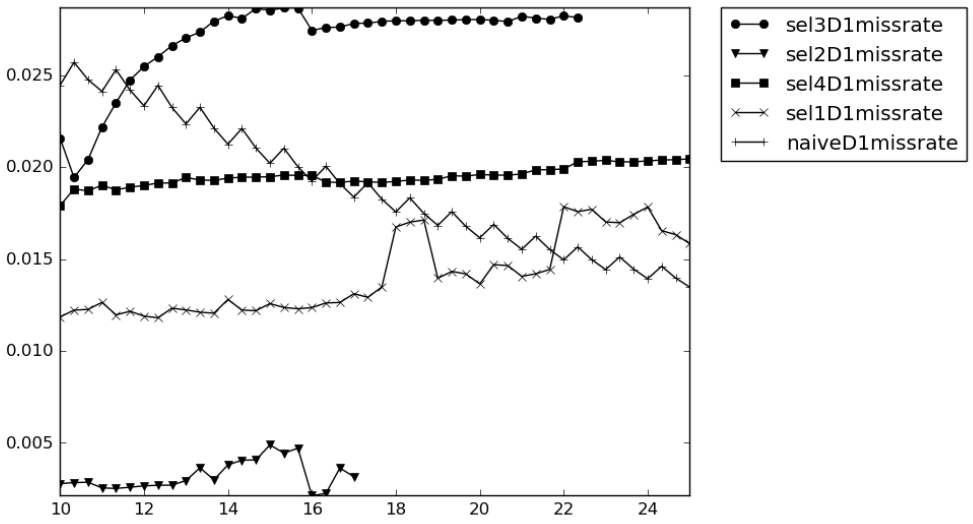

Figure 17.8: L1 data cache missrate over $\log k$. Worst case tree.

three appear mostly constant, and again the naive algorithm has less L1 data cache misses than SEL4.

Figure 17.8 shows the L1 data cache missrate for all five algorithms. SEL2 has a noteworthy small missrate. This can be explained by SEL2 computing the same clans many times over, where the cache is already filled with the data needed for the next calculations.

At the top of the graph, SEL3 has the highest L1 missrate. SEL3 guarantees clans are returned at most once from any level of RSEL3, making it less likely for data it needs to be in the L1 cache. Still the missrate is below 3% for all five algorithms.

The naive algorithm seems to steadily drop its missrate from 2.5% at $k = 2^{10}$ to below 1.5% at $k = 2^{25}$. This can be explained by the worst case tree forcing the array based priority queue of the naive algorithm to perform the maximum number of swaps for every insertion and extraction. In an array based priority queue, most of these swaps will involve nodes at the top, meaning at the front of the array, which will be in cache. Early in the data set, where k is small, the $O(k \cdot \log k)$ operations performed by the priority queue is not much larger than the $O(k)$ lookups into the tree $\mathcal{H}_0$. Later, as the $\log k$ factor starts to set in, more time will be spent in the priority queue than getting elements from $\mathcal{H}_0$.

SEL1 and SEL4 appear mostly constant.

Figure 17.9 shows L3 data cache misses divided by k for all five algorithms. SEL2 and SEL3 around $k = 2^{14}$ and $k = 2^{16}$. After that SEL3 appears constant, while SEL2 does not have enough data points to support any conclusions.

Figure 17.10 shows the same for the naive algorithm, SEL1, and SEL4. All

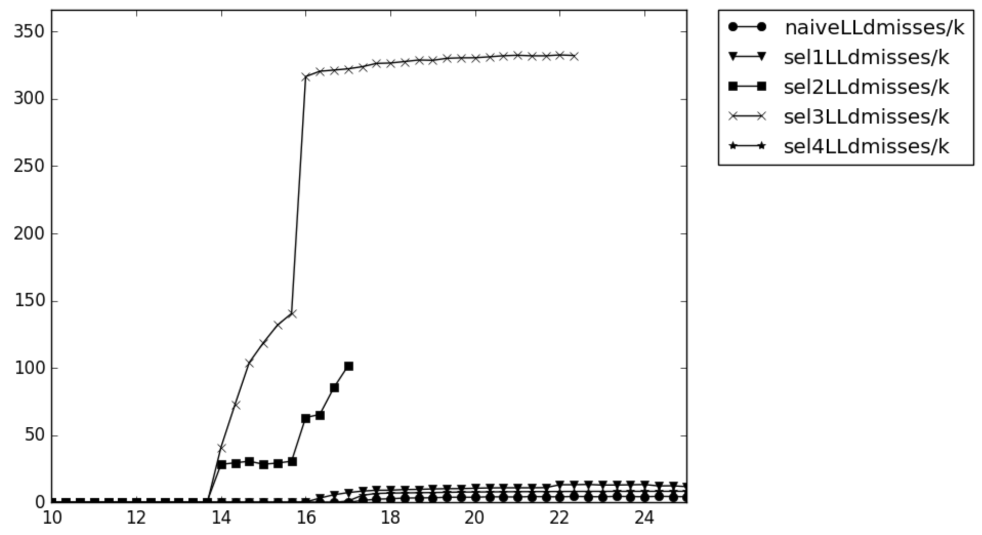

Figure 17.9: L3 data cache misses divided by k over $\log k$. Worst case tree.

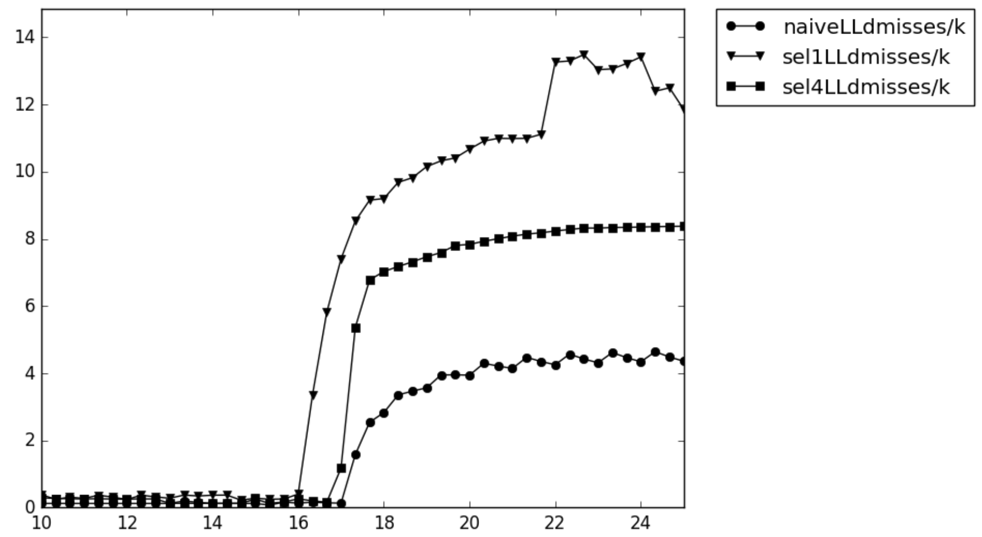

Figure 17.10: L3 data cache misses divided by k over $\log k$. Worst case tree.

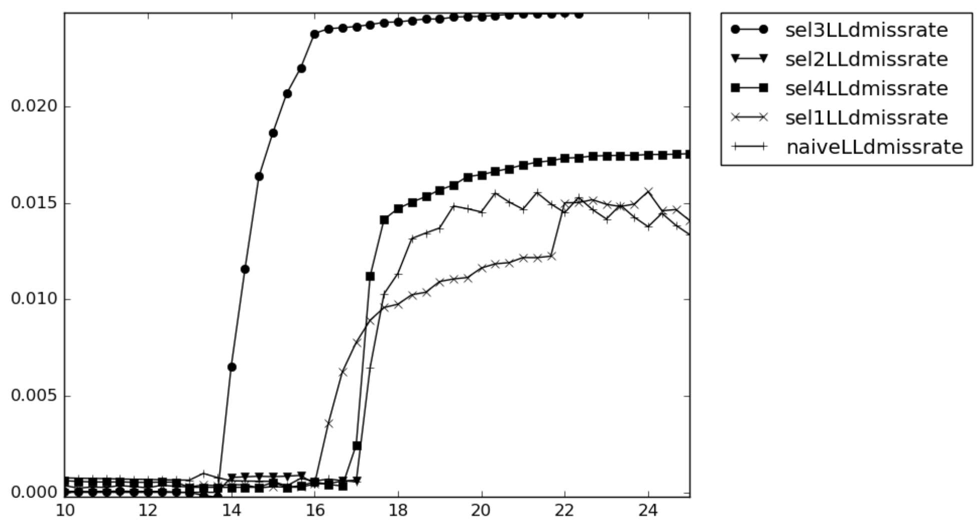

Figure 17.11: L3 data cache missrate over $\log k$. Worst case tree.

three appear mostly constant, and again the naive algorithm has less L3 data cache misses than SEL4, just like it did with L1 data cache misses.

Figure 17.11 shows the L3 data cache missrate over $\log k$ for all five algorithms. Like for L1 misses, SEL3 is the highest and starts spiking at $k = 2^{14}$. This conforms with the high memory use of SEL3, making it fill up the L3 cache earlier than the other algorithms.

SEL2 is barely visible at the bottom. The naive algorithm, SEL1, and SEL4 seem to converge. The naive algorithm again has a lower missrate than SEL4.

17.2 Random case

As mentioned in 14, SEL2 is not run in any of the tests with the random case graph.

Figure 17.12 shows instruction references over $\log k$. Figure 17.13 shows data references over $\log k$. Both are very similar graphs to Figure 16.1 and show that runtime of SEL3 is likely determined by the number of instruction it uses use and the number of data references it makes.

Figure 17.14 shows instruction references over $\log k$ for the fastest three algorithms. Figure 17.15 shows data references over $\log k$ for the same algorithms. They are similar. SEL4 appears mostly constant in both graphs, while SEL1 appears to converge. The naive algorithm climbs, though at a much slower rate than seen in Figure 17.3. Random numbers inserted into an array based priority queue is expected to be better than the worst case, and it shows in the slower climb.

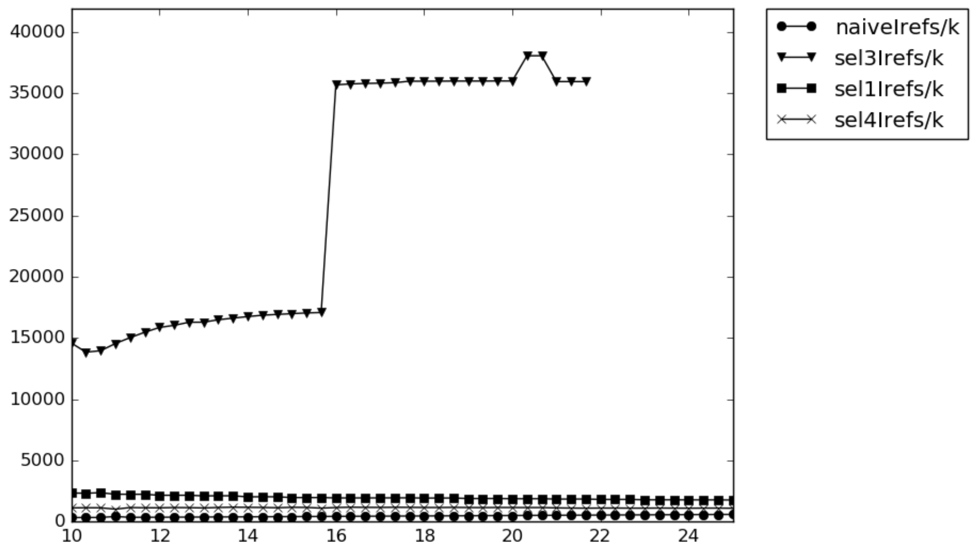

Figure 17.12: Instruction references divided by k over $\log k$. Random tree.

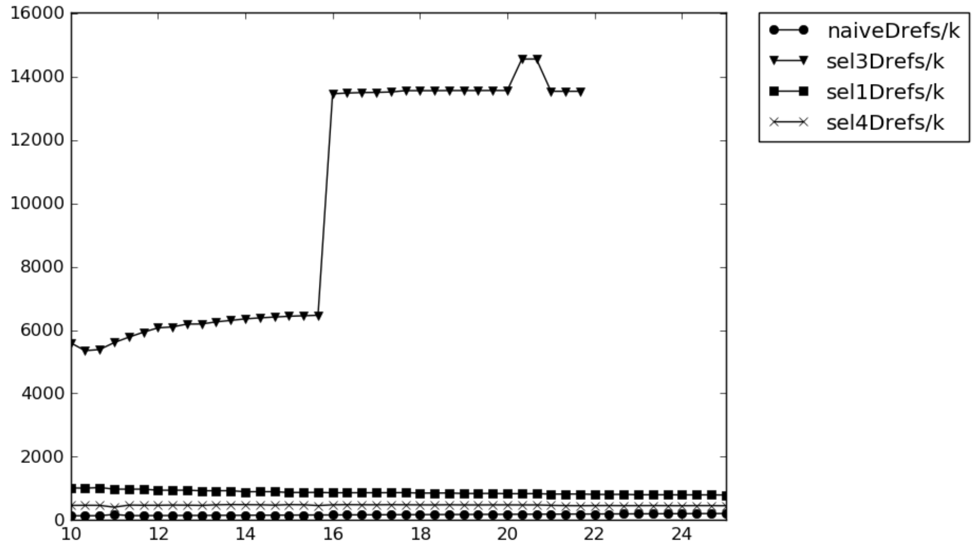

Figure 17.13: Data references divided by k over $\log k$. Random tree.

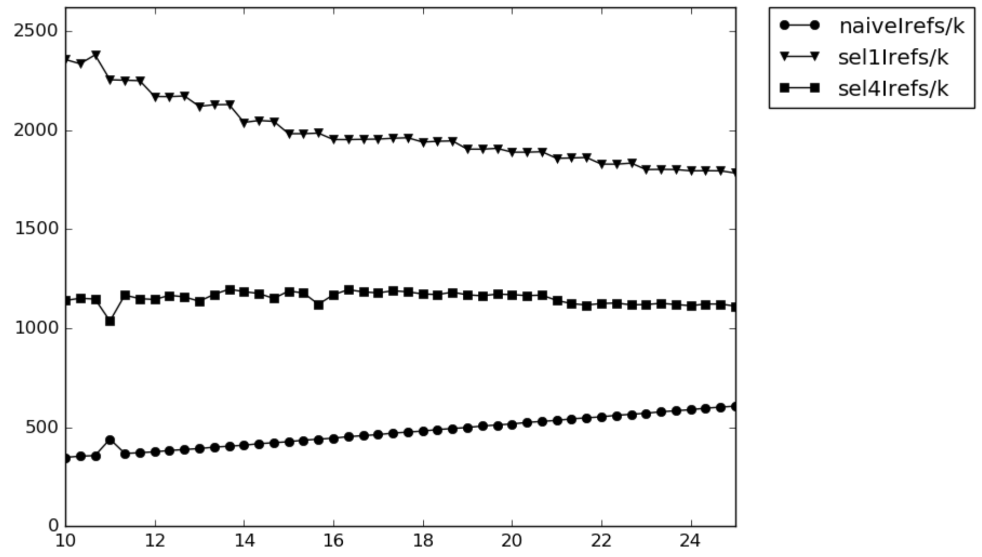

Figure 17.14: Instruction references divided by k over $\log k$. Random tree.

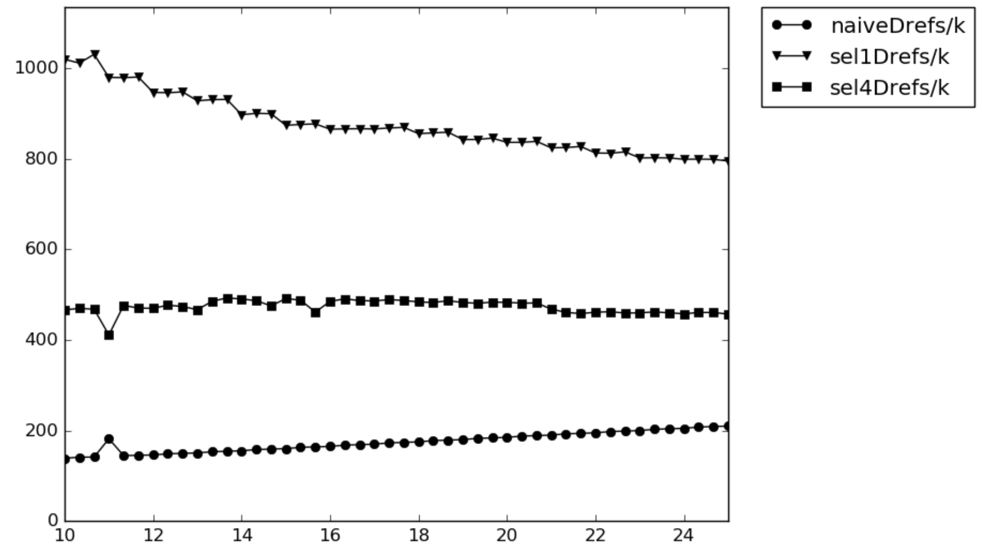

Figure 17.15: Data references divided by k over $\log k$. Random tree.

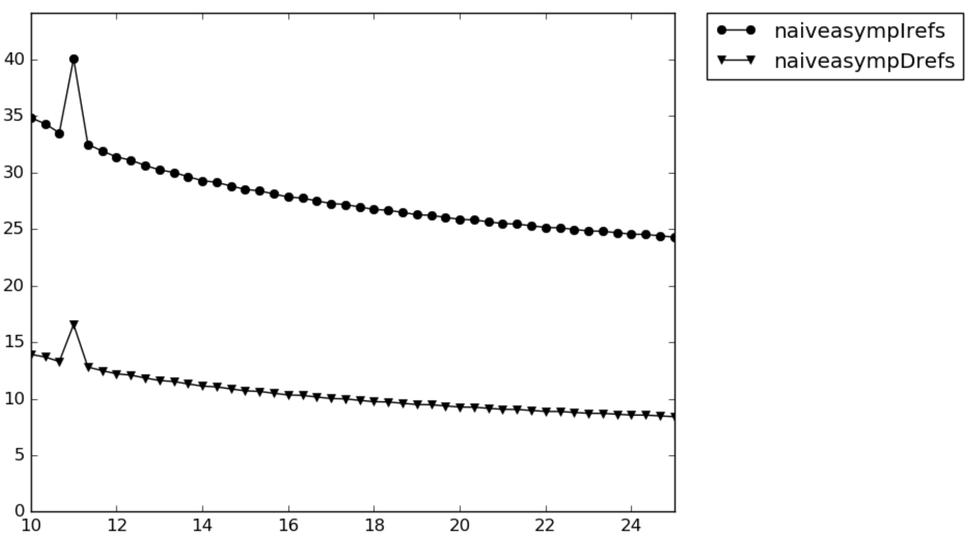

Figure 17.16: Instructions and data references divided by $k \cdot \log k$ over $\log k$. Random tree.

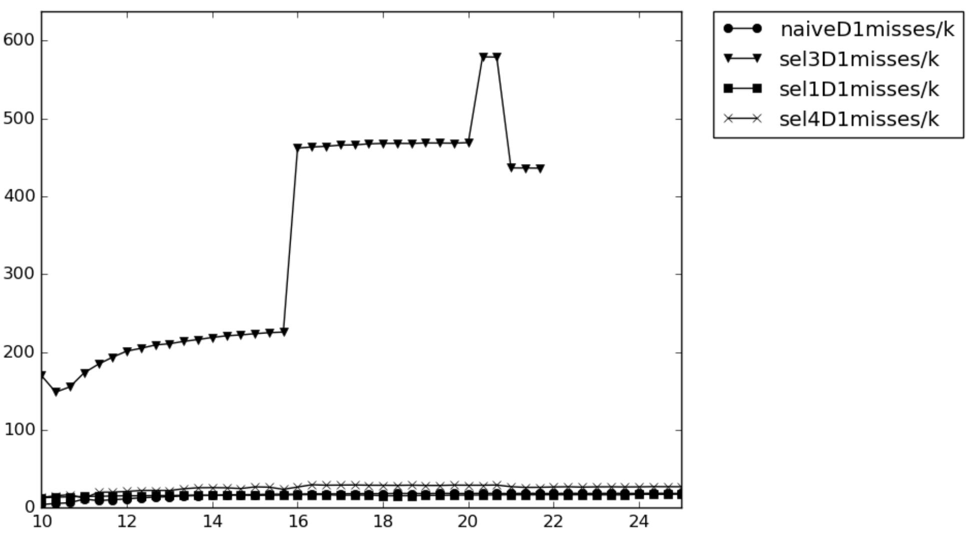

Figure 17.17: L1 data cache misses divided by k over $\log k$. Random tree.

Figure 17.16 shows the number of instructions and data references used by the naive algorithm divided by $k \cdot \log k$. Like the worst case graph, they both seem to converge and support that the implementation follows the theoretical bound.

Figure 17.17 shows L1 data cache misses divided by k for all five algorithms. SEL3 appears mostly constant after it settles after $k = 2^{16}$.

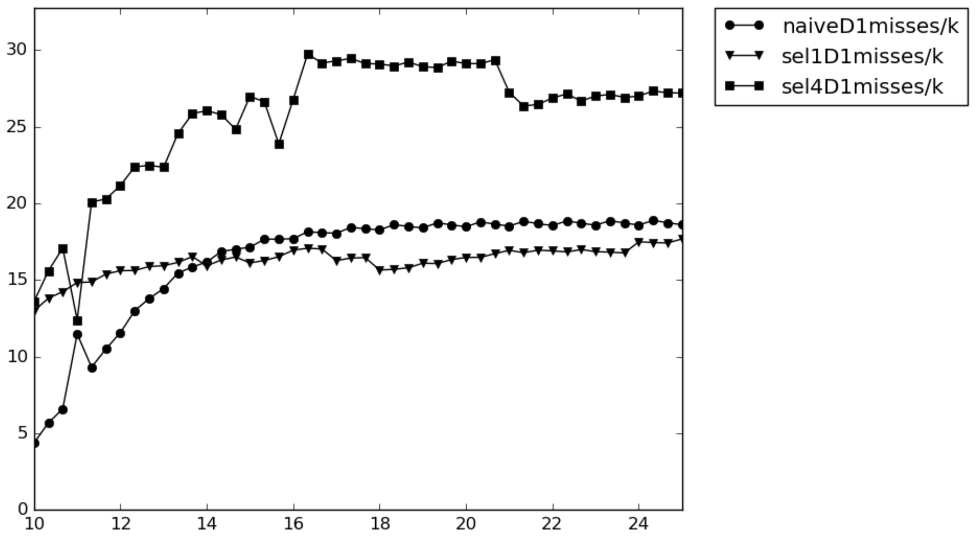

Figure 17.18: L1 data cache misses divided by k over $\log k$. Random tree.

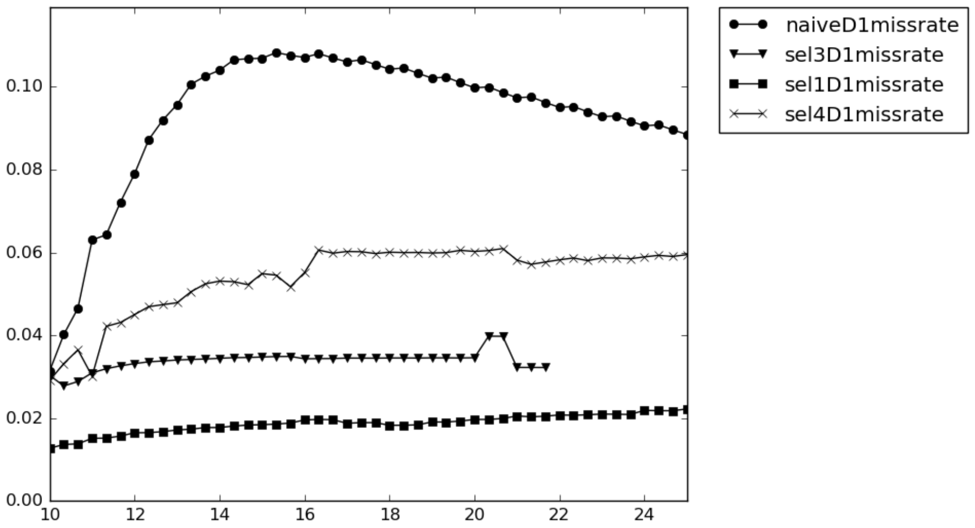

Figure 17.19: L1 data cache missrate over $\log k$. Random tree.

Figure 17.18 shows the same for the naive algorithm, SEL1, and SEL4. All three appear to converge, with SEL4 having about 50% more L1 data cache misses than the naive algorithm and SEL1.

Figure 17.17 shows the L1 data cache missrate of the four algorithms. The naive algorithm tops at around $k = 2^{15}$ and then declines. The other three remain constant throughout.

Note that while the naive algorithm has a higher L1 data cache missrate

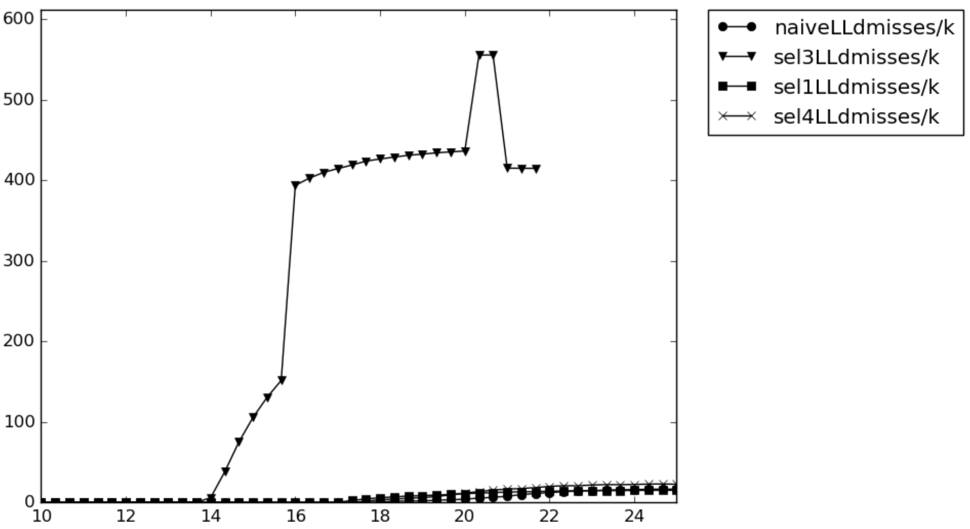

Figure 17.20: L3 data cache misses divided by k over $\log k$. Random tree.

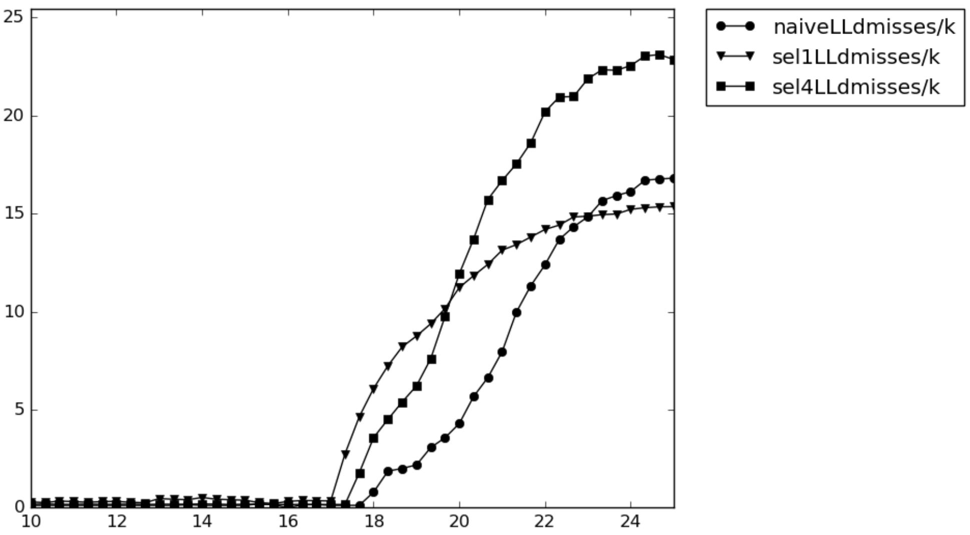

Figure 17.21: L3 data cache misses divided by k over $\log k$. Random tree.

than SEL4, SEL4 has more misses since it has more data references. Similarly SEL1 has the lowest missrate, but the highest number of data references amongst the three.

Figure 17.20 shows the L3 data cache misses divided by k of the four algorithms. SEL3 converges after the jump.

Figure 17.21 shows the L3 data cache misses divided by k of the three fastest algorithms. All three perform similar climbs, but SEL4 ends up having

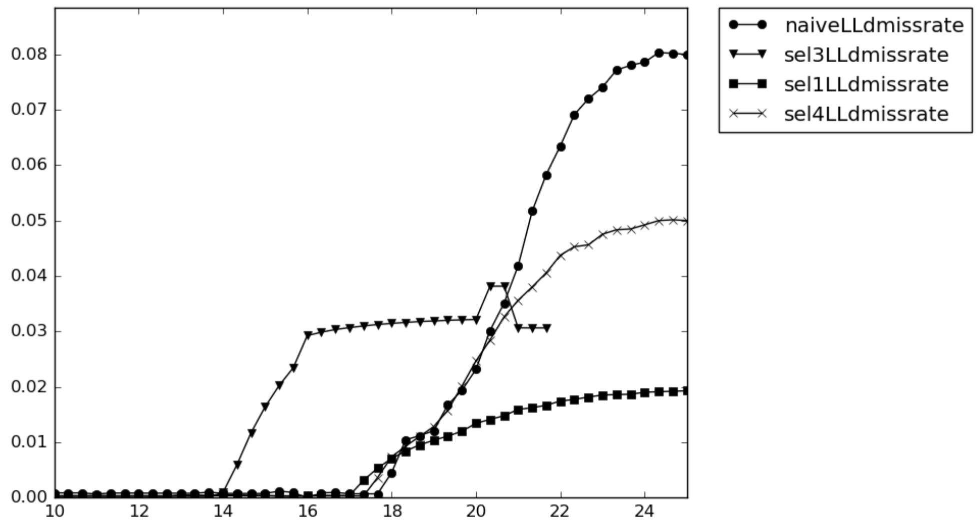

Figure 17.22: L3 data cache missrate over $\log k$. Random tree.

around one third more L3 data cache misses than the naive algorithm and SEL1.

Figure 17.22 shows the L3 data cache missrate for all four algorithms. SEL3 spikes much earlier than the other three due to its higher memory usage. The naive algorithm has the highest misrate at 8%, but since SEL4 has many more data references, it also has more L3 data cache misses, as seen in Figure 17.21.

In general, the naive algorithm's superior performance can be attributed to its priority queue performing much better at random data than worst case data. The worst case tree also had an L3 data cache missrate that converged much earlier, beginning at $k = 2^{18}$, see Figure 17.11. In comparison, the same test on the random tree doesn't begin to converge before $k = 2^{23}$, giving us very little data to work with.

17.3 Best case

Figure 17.23 shows instruction references over $\log k$. Figure 17.24 shows data references over $\log k$. Both are very similar graphs to Figure 16.1 and show that runtime of SEL2 and SEL3 are mostly determined by the number of instruction they use and the number of data references they make.

Figure 17.25 shows instruction references over $\log k$ for the fastest three algorithms. Figure 17.26 shows data references over $\log k$ for the same algorithms. They are similar. SEL1 and SEL4 appear mostly constant in both graphs. Interestingly, the naive algorithm does not appear constant in instruction count, likely because the priority queue it uses does not behave as expected. This is discussed further in Chapter 15.

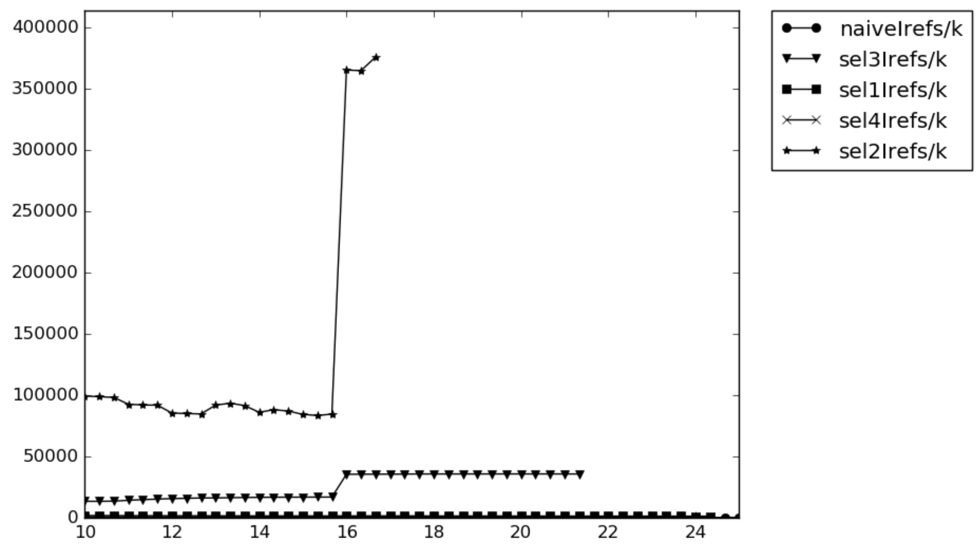

Figure 17.23: Instruction references divided by k over $\log k$. Best case tree.

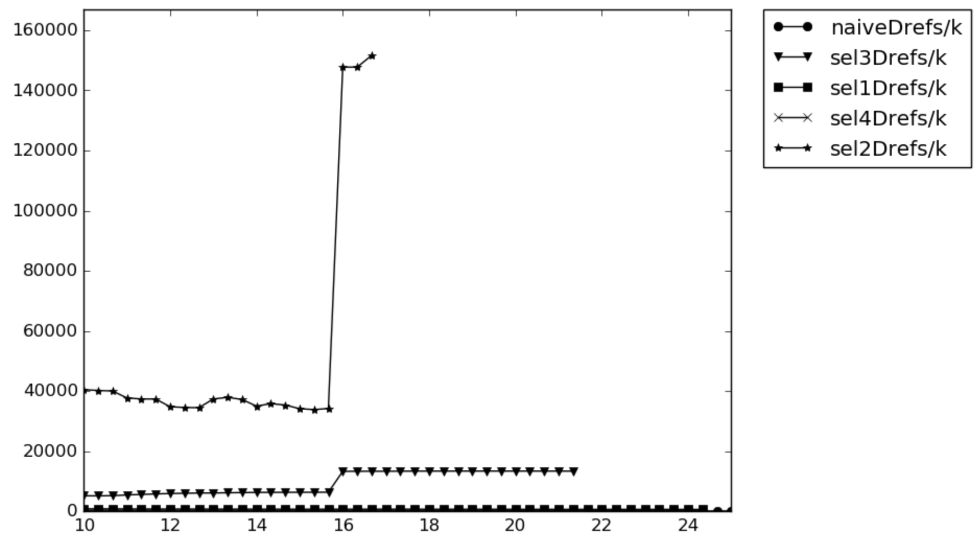

Figure 17.24: Data references divided by k over $\log k$. Best case tree.

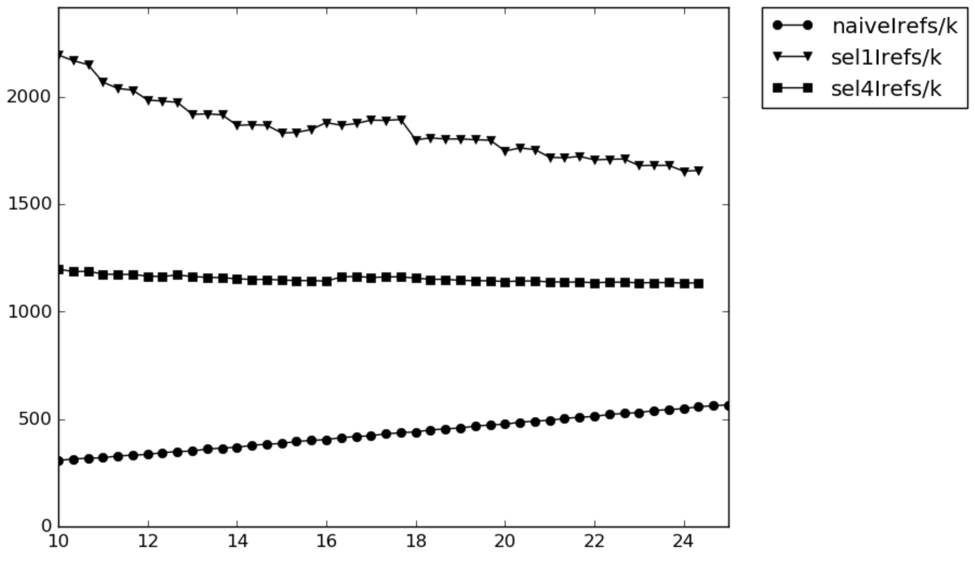

Figure 17.25: Instruction references divided by k over $\log k$. Best case tree.

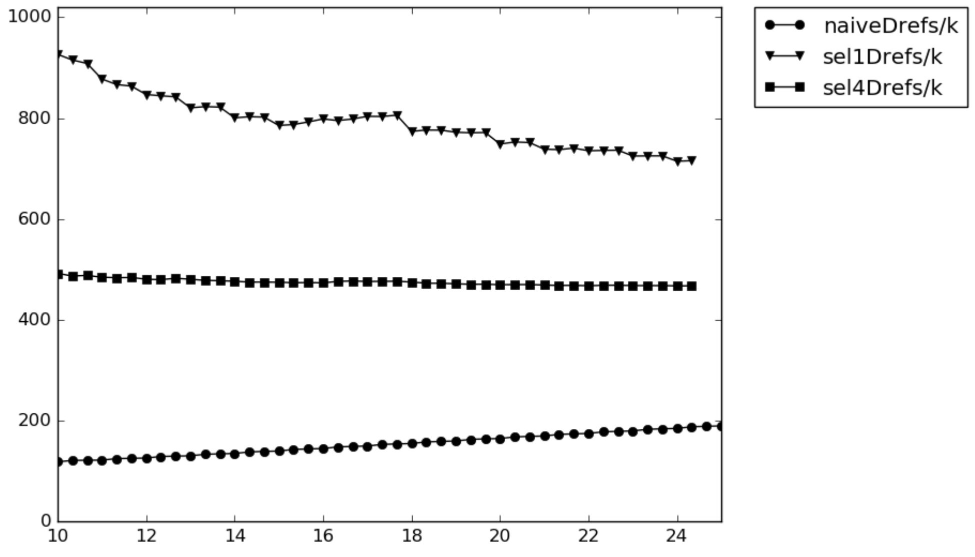

Figure 17.26: Data references divided by k over $\log k$. Best case tree.

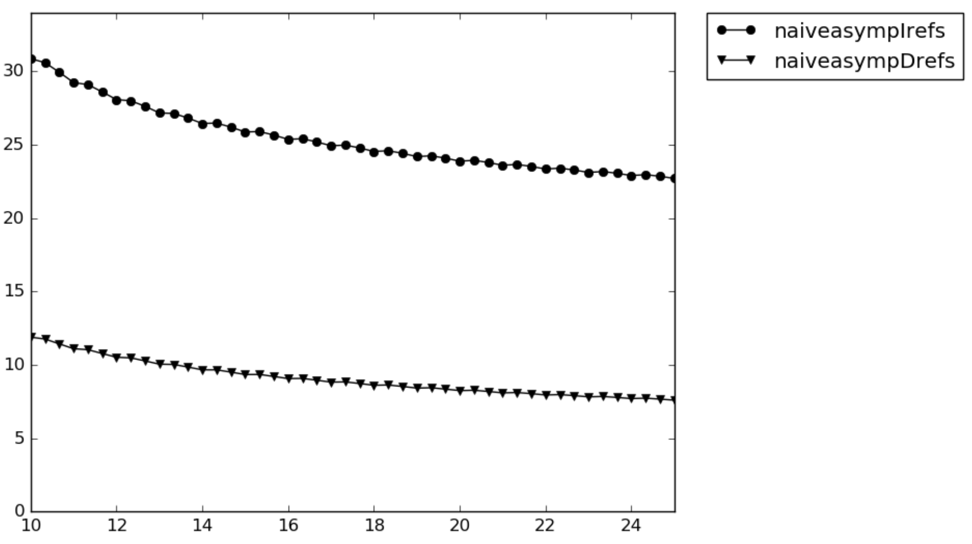

Figure 17.27: Instructions and data references divided by $k \cdot \log k$ over $\log k$. Best case tree.

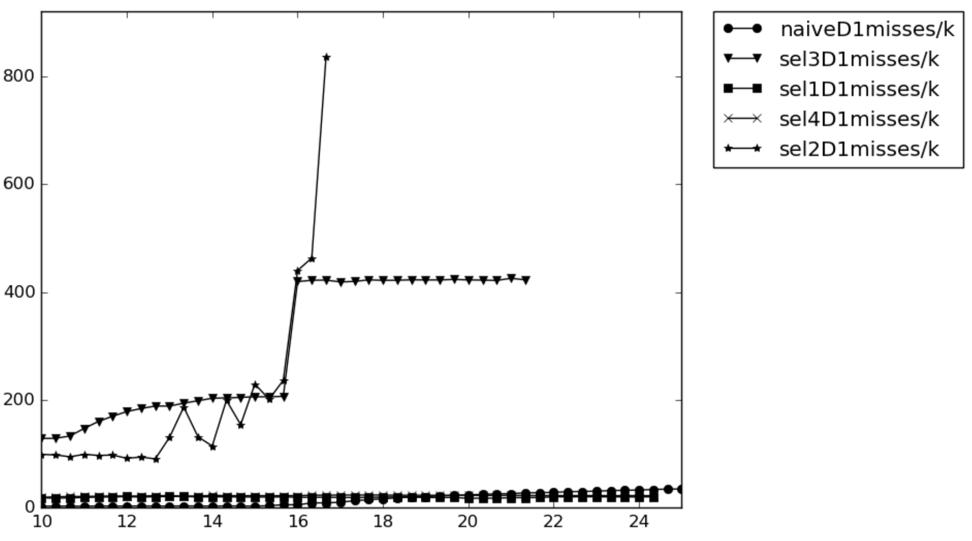

Figure 17.28: L1 data cache misses divided by k over $\log k$. Best case tree.

Figure 17.27 shows the number of instructions and data references used by the naive algorithm divided by $k \cdot \log k$. Like the worst case graph, they both seem to converge and support that the implementation follows the theoretical bound.

Figure 17.28 shows L1 data cache misses divided by k for all five algorithms. SEL3 appears mostly constant after it settles after $k = 2^{16}$. SEL2 is too hard

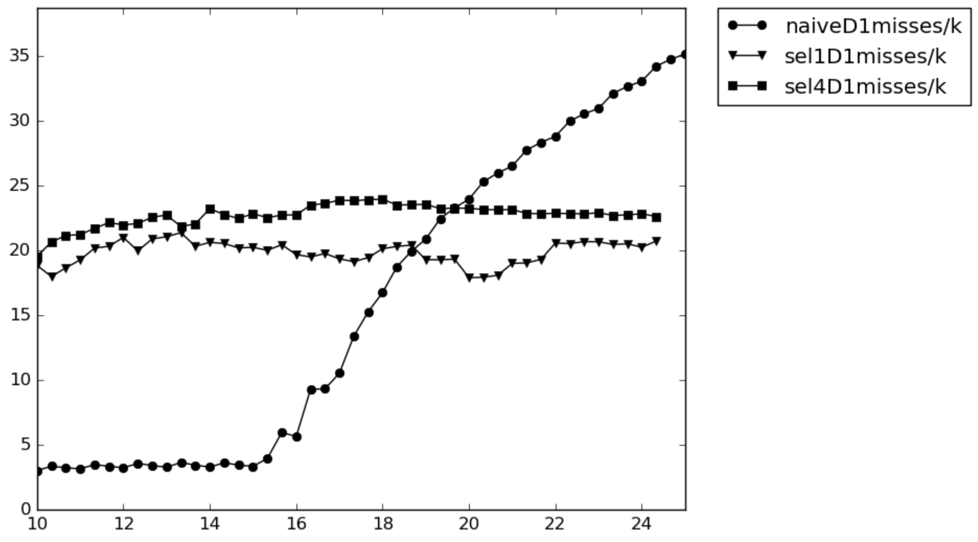

Figure 17.29: L1 data cache misses divided by k over $\log k$. Best case tree.

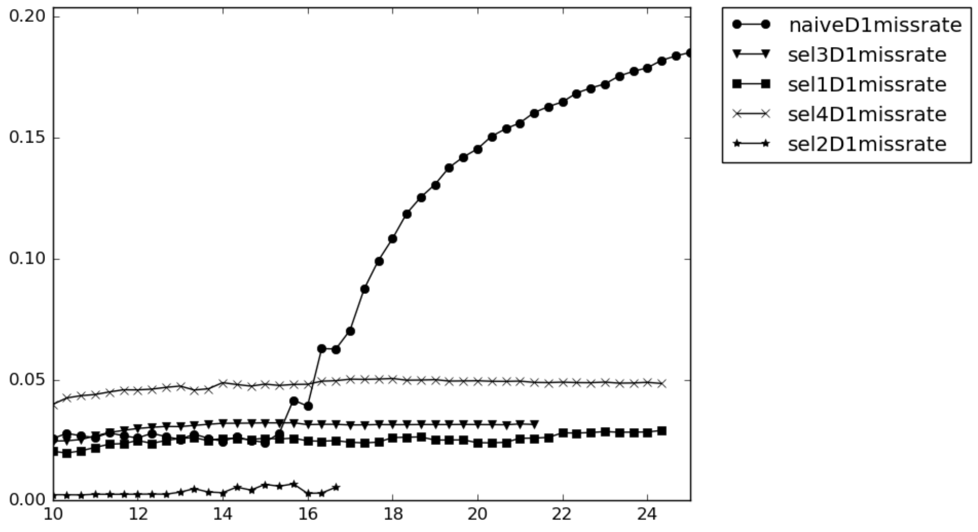

Figure 17.30: L1 data cache missrate over $\log k$. Best case tree.

to make sense of with the data available.

Figure 17.29 shows the same for the naive algorithm, SEL1, and SEL4. SEL1 and SEL4 appear mostly constant, while the naive algorithm climbs after $k = 2^{16}$. See Chapter 15 for an explanation.

Figure 17.28 shows the L1 data cache missrate of the five algorithms. Unlike the worst case results from Figure 17.28, the naive algorithm deviates from a constant around $k = 2^{16}$, and climbs to a 18% L1 missrate. SEL4 jumps from

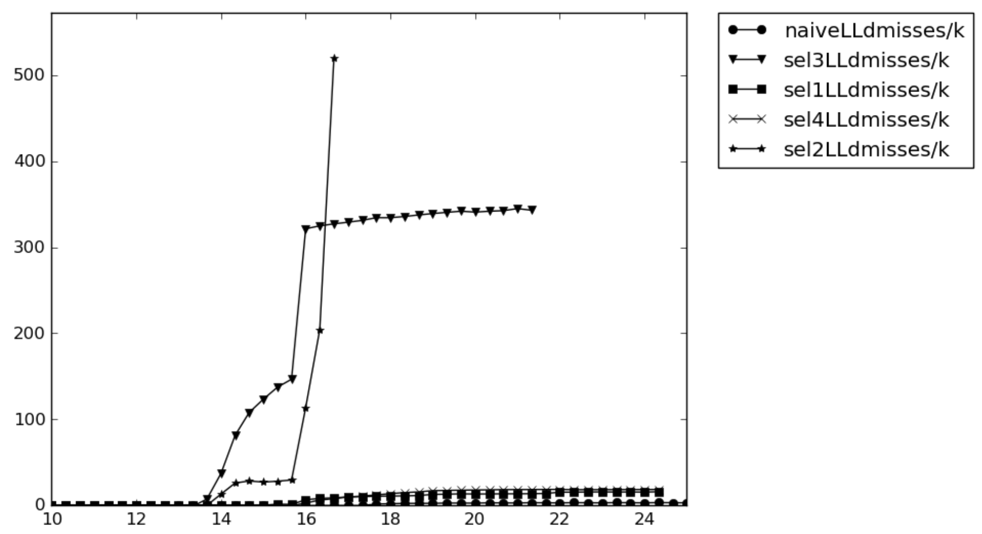

Figure 17.31: L3 data cache misses divided by k over $\log k$. Best case tree.

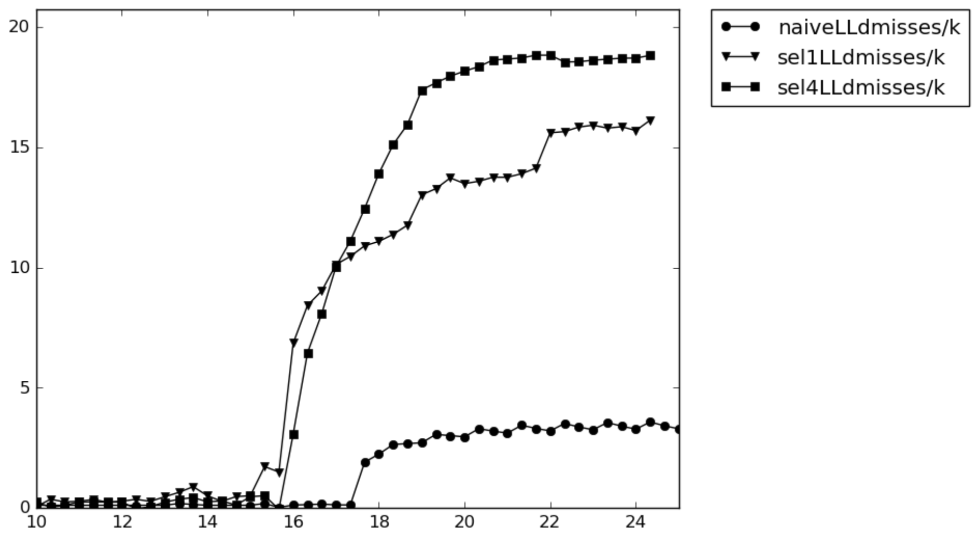

Figure 17.32: L3 data cache misses divided by k over $\log k$. Best case tree.

2% in the worst case graph to 5% here. The remaining algorithms perform as before.

Both the naive algorithm and SEL4 performing unexpected is explained in Chapter 15.

Figure 17.31 shows a constant SEL3 after the jump and a SEL2 that does not come with enough data for analysis.

Figure 17.32 shows SEL1 being close to the same as in Figure 17.32, in

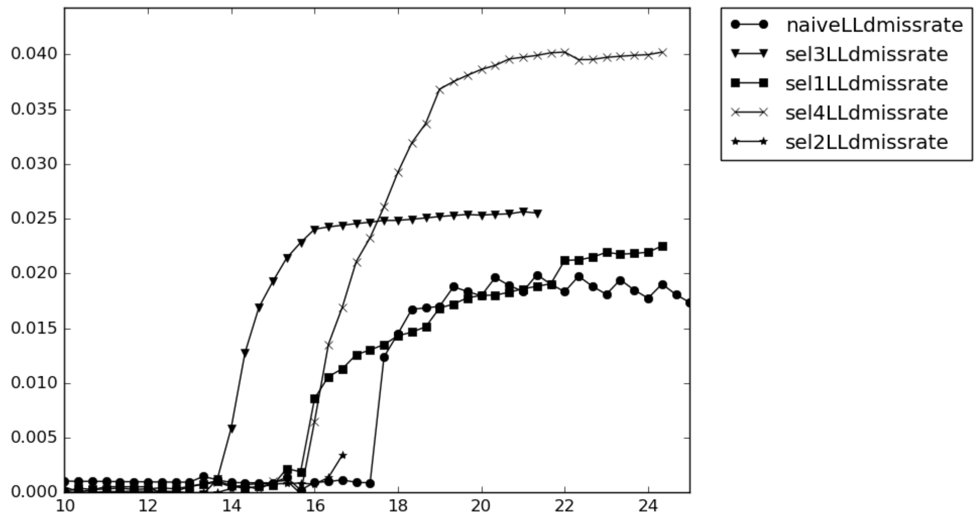

Figure 17.33: L3 data cache missrate over $\log k$. Best case tree.

the worst case graph. SEL4 has more L3 cache misses than in the worst case graph, and the naive algorithm has much fewer.

Figure 17.33 shows the L3 data cache missrate for all five algorithms, the same as Figure 17.11 shows for the worst case graph. The only difference is that SEL4 have a much higher missrate. See Chapter 15.

17.4 Conclusion

Worst case

Runtime results for all five algorithms from Chapter 16 largely correlate with instruction, data reference, and cache data found with Valgrind. SEL2 and SEL3 show linear number of used instructions, data references, and L1 and L3 data cache misses, outside of their jump at $k = 2^{16}$, which is where an extra layer of recursion is added.

SEL1 and SEL4 are largely linear in most in most graphs. The naive algorithm shows linear tendencies in cache related measures, and superlinear tendencies in terms of instructions used and data references made.

Random case

The random case tree favors the naive algorithm to the extend that it is the fastest in the tests. It is possible it would eventually lose due to its higher complexity, but that is speculation.

Best case

The best case graphs shows mostly the same as the worst case graphs, due to issues explained in Chapter 15. Some algorithms have switched positions in comparison to one another, with especially SEL4 performing worse. SEL2 is mostly unaccounted for due to lack of data.

Chapter 18

Conclusion

The optimal $O(k)$ algorithm SEL4 presented by Frederickson [5] and described in Chapter 11 can be faster than the simpler naive algorithm described in Chapter 3. SEL2 and SEL3 generally have terrible performance, while SEL1 has comparable performance to SEL4 and the naive algorithm.

For the test heap created to give the naive algorithm the worst case performance, SEL4 hinted at a practical linear complexity, becoming faster than the naive algorithm for $k \geq 2^{20}$. SEL1 is slower than both SEL4 and the naive algorithm.

When tested with a random heap, the naive algorithm performed better than SEL4. SEL4 was mostly slowed down by cache misses, which only started to stabilize at $k = 2^{23}$. The machine running the tests could not handle much larger than $k = 2^{25}$, so data showing a possible linear performance of SEL4 could not be obtained. SEL1 managed to tie with the naive algorithm at the last data points.

For the best case heap the naive algorithm was faster than SEL4, though the naive algorithm was not linear as expected. The reason for that is explained in Chapter 15. Judging by the time graph, it is likely SEL4 would be as fast as the naive with a few more data points. SEL1 performed on par with SEL4 for some time, then got slower at $k \geq 2^{22}$.

Chapter 19

Theorems

Theorem 19.0.1 (Product convergence) *Let a_n be a sequence of positive numbers for which $a_n \to 0$ for $n \to \infty$, then the product:*

$$\prod_{n=1}^{\infty}(1+a_n)$$

converges if and only if, the sum:

$$\sum_{n=1}^{\infty}a_n$$

converges.
Proof:
For this section whenever we write log *we use the natural logarithm. First we show that:*

$$\lim_{x\to 0}\frac{\log(1+x)}{x}=1$$

Since when $x=0$, we have $\log(1)=0$ the fraction will be $\frac{0}{0}$ which is indeterminate, we can use L'Hôpital's rule:

$$\begin{aligned}\lim_{x\to 0}\frac{\log(1+x)}{x}&=\lim_{x\to 0}\frac{\frac{d}{dx}(\log(1+x))}{\frac{d}{dx}(x)}\\&=\lim_{x\to 0}\frac{\frac{1}{1+x}}{1}\\&=\lim_{x\to 0}\frac{1}{1+x}\\&=1\end{aligned}$$

Now we have this we apply the natural logarithm to our product:

$$\log(\prod_{n=1}^{\infty}(1+a_n))=\sum_{n=1}^{\infty}\log(1+a_n)$$

Remember we have $a_n \to 0$ for $n \to \infty$, we obtain:

$$\lim_{a_n \to \infty} \frac{\log(1 + a_n)}{a_n} = 1$$

By the limit comparison test we obtain, that $\sum_{n=1}^{\infty} \log(1 + a_n)$ converges if and only if $\sum_{n=1}^{\infty} a_n$ converges.

Theorem 19.0.2 (Sum convergence) *The sum:*

$$\sum_{i=1}^{\infty} \frac{1}{i^2} \tag{19.1}$$

converges.
Proof:
If instead of starting the sum of Equation 19.1 in 1 we start it in 2 we obtain:

$$\sum i = 2^{\infty} \frac{1}{i^2}$$

Which clearly will also converge if Equation 19.1 does. We then make the observation that $0 \leq \frac{1}{i^2} \leq \frac{1}{i^2 - i}$, since subtracting below from the denominator will make the expression bigger. Additionally we show that, for $i \geq 2$:

$$\frac{1}{i-1} - \frac{1}{i} = \frac{i - (i-1)}{i \cdot (i-1)} = \frac{1}{i^2 - i}$$

Using this we can now look at the sum:

$$\sum_{i=2}^{\infty} \left(\frac{1}{i-1} - \frac{1}{i}\right)$$

First we will look at the function if rather than going to infinity it goes to some N:

$$\begin{aligned}\sum_{i=2}^{N}\left(\frac{1}{i-1} - \frac{1}{i}\right) &= \left(\frac{1}{1} - \frac{1}{2}\right) + \left(\frac{1}{2} - \frac{1}{3}\right) + \left(\frac{1}{3} - \frac{1}{4}\right) + \ldots + \left(\frac{1}{N-2} - \frac{1}{N-1}\right) + \left(\frac{1}{N-1} - \frac{1}{N}\right) \\ &= \frac{1}{1} - \frac{1}{N}\end{aligned}$$

We get this because for every part except the first and last there is both a positive and negative occurrence, and these cancel out. If we now let $N \to \infty$ and look at the limit we get:

$$\lim_{N \to \infty} \sum_{i=2}^{N} \left(\frac{1}{i-1} - \frac{1}{i}\right) = \lim_{N \to \infty} \left(1 - \frac{1}{N}\right) = 1$$

Which means it converges, and thus our original expression of Equation 19.1 converges.

Bibliography

[1] Manuel Blum et al. “Time bounds for selection”. In: *Journal of Computer and System Sciences* 7.4 (Aug. 1973), pp. 448–461. ISSN: 00220000. DOI: `10.1016/S0022-0000(73)80033-9`. URL: `http://www.sciencedirect.com/science/article/pii/S0022000073800339`.

[2] Gerth Stølting Brodal and Allan Grønlund Jørgensen. “Selecting Sums in Arrays”. In: *Algorithms and Computation: 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings.* Ed. by Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 100–111. ISBN: 978-3-540-92182-0. DOI: `10.1007/978-3-540-92182-0_12`. URL: `http://dx.doi.org/10.1007/978-3-540-92182-0_12`.

[3] Thomas H Cormen et al. *Introduction to Algorithms, Third Edition.* 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[4] David Eppstein. “Finding the k Shortest Paths”. In: *SIAM Journal on Computing* 28.2 (1998), pp. 652–673. DOI: `10.1137/S0097539795290477`. eprint: `http://dx.doi.org/10.1137/S0097539795290477`. URL: `http://dx.doi.org/10.1137/S0097539795290477`.

[5] G.N. Frederickson. “An Optimal Algorithm for Selection in a Min-Heap”. In: *Information and Computation* 104.2 (June 1993), pp. 197–214. ISSN: 08905401. DOI: `10.1006/inco.1993.1030`. URL: `http://www.sciencedirect.com/science/article/pii/S0890540183710308`.

[6] J. W. J. Williams. “Algorithm 232 - Heapsort”. In: *Communications of the ACM* 7.6 (1964), pp. 347–348.