Binary Heaps, Fibonacci Heaps and Dijkstras shortest path

Kristoffer Just Andersen, 20051234

Troels Leth Jensen, 20051234

Morten Krogh-Jespersen, 20022362

Project 1, Advanced Data Structures 2013, Computer Science
January 2014

Advisor: Gerth StÃ¸lting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Contents

Chapter 1

Introduction

This report details our investigation into the performance characteristics of the Red-Black trees and van Emde Boas trees. We will describe how Red-Black trees and van Emde Boas trees and their operations work and describe their time complexities. Furthermore, we will show how a Bit-Vector works and we will design a special construct of a combo tree that has van Emde Boas trees to a certain level and then Bit-Vectors for the leafs.

We will also design and perform specific tests for our trees and bit-vector. This requires us to think about how testing should be done since time-complexities for Red-Black tree are dominated by the number of elements in the structure while van Emde Boas trees and bit-vector is dominated by the universe of keys it supports. Our expectation is that one of our combo trees will perform the best overall.

This is the second project in the course and we will compare our tree implementations with the heaps from project one. This requires buildin an interface and rearranging the tests such that it makes sense to talk about elements inserted and the universe for van Emde Boas. We do not expect any of the trees to perform as well as the heaps.

Finally, we will argue a bit about specific implementation where we show that a naive implementation can be optimized to run quicker by thinking about how the upper and lower parts of the bits are calculated for the van Emde Boas tree.

Chapter 2

Bit-vector

A bit-vector or a bit array is a datastructure that supports INSERT, DELETE and MEMBER in constant time [**?**, p. 532]. The bit-vector consists of a set of bits, for which the $i'th$ bit indicates the presence of the $i'th$ key. Thus, in order to support queries over an universe of size u we must have a bit-vector with u bits.

To INSERT some key i into the bit-vector the $i'th$ bit is set to true, similarly to DELETE the same key i, the $i'th$ bit is cleared. The MEMBER test for key i is done by checking the $i'th$ bit. All these operations are supported easily through bit shifts.

Please note, that these bit operations assume that the universe size does not exceed the word size. If that is not the case, extra computation is needed to first find the correct block; this however is also constant time.

The bit-vector performs badly for MINIMUM, MAXIMUM, PREDECESSOR and SUCCESSOR because the entire array could be traversed in search for the answer.

Operation	Bit-Vector	Bit-Vector w. min & max
MEMBER	$\Theta(1)$	$\Theta(1)$
MINIMUM	$O(u)$	$\Theta(1)$
MAXIMUM	$O(u)$	$\Theta(1)$
PREDECESSOR	$O(u)$	$O(u)$
SUCCESSOR	$O(u)$	$O(u)$
INSERT	$\Theta(1)$	$\Theta(1)$
DELETE	$\Theta(1)$	$O(u)$

Alternatively, MINIMUM and MAXIMUM can be maintained during all changes to the bit vector, i.e. during INSERT and DELETE. With this approach the running time of DELETE would in worst case be completely dominated by the linear search for a new MINIMUM or MAXIMUM while querying for the MINIMUM or MAXIMUM would become constant time operations. For the remaining part of this report, we will refer to the Bit-Vector as the version that maintains MINIMUM and MAXIMUM.

Chapter 3

Red-Black Tree

The simplest search tree we know is the binary search tree (BST). Its invariant is simple, and both querying and updating is formulated easily, both iteratively and recursively. The runtime guarantees are, while equally easy to argue, not optimal.

The root of the problem is an issue of balancing. A plain BST maintains no notion of balance, so a pathological use can create 'flat trees' where all nodes end up along a single spine, giving worst-case linear time for most operations.

One solutions is a red-black tree (RBT), where the idea is to maintain additional structure in order to remain balanced.

The remainder of this section describes the structure of an RBT, and describes the operations which we have implemented, that has the following time complexities with compared with the Bit-Vector:

Operation	Bit-Vector	Red-Black tree
MEMBER	$\Theta(1)$	$O(\log n)$
MINIMUM	$\Theta(1)$	$O(\log n)$
MAXIMUM	$\Theta(1)$	$O(\log n)$
PREDECESSOR	$O(u)$	$O(\log n)$
SUCCESSOR	$O(u)$	$O(\log n)$
INSERT	$\Theta(1)$	$O(\log n)$
DELETE	$O(u)$	$O(\log n)$

3.1 Red-Black Trees

A red black tree is essentially an ordinary BST with an additional bit of information per node: the "color" of that node, black or red. Through the use of a handful of invariants, the color information is used to balance the RBT. So in addition to the usual fields of key, parent pointer, left and right child pointers, we also store a color bit.

The fact that an RBT is just an augmented BST means that the basic queries essentially need no changes to work on RBTs, and, indeed, Cormen et al. (REF) on which we base our implementation, defers to the section on simple BSTs.

The invariants that we place on an RBT, in addition to the basic search tree property, is as follows:

1. The root is black
2. All leaves are black
3. If a node is red, both its children are black
4. For each node, all paths from node to descendant leaves contain the same number of black nodes.

See Lemma 13.1 in Cormen et al (REF) for how these last two properties ensure balancing.

3.2 Operations

3.2.1 Queries

No queries exploit the RBT structure, and so work procedurally like the operations known from simple BSTs. The running times, however, benefit from the balancing guarantees of RBTs.

Minimum The minimum node of a search tree is its left-most node, if any. This is given by the basic search tree invariant. By virtue of being balanced, the minimum node lies at a maximum depth of $\log(n)$, where n is the number of nodes in the tree, and the worst-case running time is thus $O(\log(n))$.

Maximum Symmetric to Minimum.

Predecessor The Predecessor query asks, given a particular node in a search tree: "what is the greatest node smaller than this particular node?" In the case of an internal node, the answer can be constructed using Maximum, as we simply query the left subtree of the given node for its maximum node: running time proportional to the depth of the given node. In the case of a leaf, we must find the first ancestor such that the given node is in its right sub-tree. This will ensure $Successor \circ Predecessor = id$. The running time is again proportional to the depth of the given node, and so is always $O(\log(n))$.

Successor Symmetric to Predecessor.

Search The Search query asks, given a key and a search tree, "does the tree contain a node with the given key?", and relies entirely on the basic search tree invariant. If the node at the root of the given tree has the desired key, we have found what we are looking for. Otherwise, the desired key is either smaller or larger than the key at the root of the given tree, and we can thus recursively search either the left or right tree, respectively. We answer negatively upon hitting a leaf. The running time is proportional to the depth of the initial tree, and thus runs $O(\log(n))$ where n is the number of nodes in the tree.

3.2.2 Updates

Updating operations on RBTs are complicated by the need to maintain the RBT invariants described in section 3.1.

Insert To maintain the search-tree invariant, the RBT insertion procedure works operationally the same as the insertion procedure, and then does some additional work to reestablish the RBT invariants. The first phase of searching runs proportional to the depth of the tree, and the reestablishing runs back up the tree in the worst case, performing a constant amount of work at each node. This gives a grand total of $O(\log(n))$ work for insertions.

Delete Deletion is similar in argument to insertion, albeit complicated when the node to be removed is black. It does a search down, and a fix up, for a run-time of $O(\log(n))$.

Chapter 4

Van Emde Boas Tree

A van Emde Boas tree is a recursive data structure that supports finding the predecessor and successor in $\mathrm{O}(\log\log m)$ time where m is the size of the universe, in other words, the amount of distinct keys with a total ordering that the tree support [**?**, p. 545] [**?**]. Because van Emde Boas Trees allow universe sizes of any power of 2, we denote $\sqrt[\uparrow]{u} = 2^{\lceil(\log u)/2\rceil}$ and $\sqrt[\downarrow]{u} = 2^{\lfloor(\log u)/2\rfloor}$. Each level of the van Emde Boas Tree has a universe u and it contains $\sqrt[\uparrow]{u}$ clusters/bottom van Emde Boas trees of universe size $\sqrt[\downarrow]{u}$ and one auxilary van Emde Boas Tree of size $\sqrt[\uparrow]{u}$ we denote top. For each recursion the universe shrinks by $\sqrt{u}$.

For each tree we have two attributes that store the minimum and the maximum key. The minimum key cannot be found in any of the bottom trees, neither can the maximum unless it differs from the minimum, which happens if there are more than one element in the tree. The attributes helps reduce the number of recursive calls, because one can in constant time decide if a value lies within the range, without the need to recurse.

Van Emde Boas Trees utilizes that keys are represented as bits and if we view a key x as a $\log u$-bit binary integer, we can divide the bits up in a most significant and a least significant part. The most significant part of the bit vector identifies the cluster $\lfloor x/\sqrt{u}\rfloor$ where x will apear in position $x \mod \sqrt{u}$ in the beforementioned bottom tree. Let us denote $\mathsf{high}(x) = \lfloor x/\sqrt{u}\rfloor$ and $\mathsf{low}(x) = x \mod \sqrt{u}$ then we get the identity $x = \mathsf{high}(x)\sqrt{u} + \mathsf{low}(x)$.

Below is the time-complexities of each operation listed for the corresponding data structure and let u denote the size of the universe and n the number of keys stored in the structure:

Operation	Bit-Vector	Red-Black Tree	van Emde Boas Tree
Member	$\Theta(1)$	$\mathrm{O}(\log n)$	$\mathrm{O}(\log\log u)$
Minimum	$\Theta(1)$	$\mathrm{O}(\log n)$	$\mathrm{O}(1)$
Maximum	$\Theta(1)$	$\mathrm{O}(\log n)$	$\mathrm{O}(1)$
Predecessor	$\mathrm{O}(u)$	$\mathrm{O}(\log n)$	$\mathrm{O}(\log\log u)$
Successor	$\mathrm{O}(u)$	$\mathrm{O}(\log n)$	$\mathrm{O}(\log\log u)$
Insert	$\Theta(1)$	$\mathrm{O}(\log n)$	$\mathrm{O}(\log\log u)$
Delete	$\mathrm{O}(u)$	$\mathrm{O}(\log n)$	$\mathrm{O}(\log\log u)$

4.1 Finding minimum or maximum key

This is a constant operation, since the van Emde Boas Tree directly stores the minimum and maximum element.

4.2 Finding a member

Finding out if a key is stored in a van Emde Boas Tree is simple. Either the key is the minimum or the maximum element or else recurse until we find the element. It is easy to figure out which tree to recurse on, since this is the $\mathsf{high}(x)$-th tree, and we just have to search in the smaller tree for $\mathsf{low}(x)$.

So how long does the search take in worst-case? The data structure can be described by the following recurrence function:

$$T(u) \leq T(\sqrt[\updownarrow]{u}) + \mathrm{O}\,(1)$$

If we let $m = \log u$ and realize that $\lceil m/2 \rceil \leq 2m/3$ for $m \geq 2$, which is the leaf size of the van Emde Boas Tree, we get:

$$\begin{aligned} T(u) &\leq T(\sqrt[\updownarrow]{u}) + \mathrm{O}\,(1) \\ &\Downarrow \\ T(2^m) &\leq T(2^{\lceil m/2 \rceil}) + \mathrm{O}\,(1) \\ &\Downarrow \\ T(2^m) &\leq T(2^{2m/3}) + \mathrm{O}\,(1) \\ &\Downarrow \\ S(m) &\leq S(2m/3) + \mathrm{O}\,(1) \end{aligned}$$

By the master thorem [**?**, p. 93], has the solution $S(m) = \mathrm{O}\,(\log m)$. Because $T(u) = T(2^m)$ we get $\mathrm{O}\,(\log m) = \mathrm{O}\,(\log \log u)$. Therefore the procedure takes $\mathrm{O}\,(\log \log u)$.

4.3 Finding a successor or a predecessor

As with finding a member, for the base case a successor or predecessor can be found in constant time. If not, we have to determine where to find the next element we are searching for. Finding successor and predecessor is completely analog thus we only describe finding the successor for an element x.

First, we check if the sucessor is in the bottom tree at index $\mathsf{high}(x)$. This check ban be done in constant time since it is a matter of checking maximum for that tree. If maximum exist and it is higher than $\mathsf{low}(x)$, we know we have to search inside that particular tree. This will take $\mathrm{O}\,(\log \log u)$ time.

If maximum does not exist or it is less than $\mathsf{low}(x)$ we use the top tree to search for the successor to $\mathsf{high}(x)$. If we can find such an element it will

gives us an index to a bottom tree b. Hereafter it is a constant lookup to find Minimum(b). This case also runs in $\mathrm{O}(\log\log u)$ time since we have on search in top and a constant lookup. Therefore, the total running time of successor and predecessor is $\mathrm{O}(\log\log u)$.

4.4 Insert

Insert is pretty simple. One of the following can happen:

- The list is empty, which can be discovered in constant time by checking the minimum attribute. If that is the case, insert just set min and max.
- If not in the base case, find out if the bottom of $\mathsf{high}(x)$ is empty. If that is the case, we can just set min and max as above and update the top structure. If not, we just call insert on the bottom tree for $\mathsf{low}(x)$.

Always remember to set max or swap with min (because min was not in the tree, but now has to be since it would no longer be min). Either, the insert function recurses on the top tree of size $\sqrt[\uparrow]{u}$ or on one of the bottom trees of size $\sqrt[\downarrow]{u}$ but not on both. Therefore, the running time is at most $\mathrm{O}(\log\log u)$.

4.5 Delete

If there is only one element or universe size is 2 is easy to perform delete in constant time. Otherwise, some more work has to be done. Consider what should be done if we try to delete the minimum element. Since min is not stored in the tree, we have to find the minimum element in the bottom trees and make sure we delete it so that it can be placed as the new min. Of course, it can happen that the the bottom tree becomes empty so the top structure has to be updated. It might also be that we remove max but updating max is a constant operation.

In the description above we actually could make two recursive calls; one to update the bottom tree and possibly one to update the top. But if we update the top then the bottom tree would only contain one element. But if that is the case the first recursive function takes constant time. Therefore delete runs in $\mathrm{O}(\log\log n)$.

Chapter 5

van Emde Boas Tree with Bit-Vector

The van Emde Boas tree is dominated by the size of the universe it supports for all of it operations except MINIMUM and MAXIMUM. For each recursion of the structure, the universe size shrinks by $\sqrt{u}$ and it might be beneficial, when u becomes sufficiantly small, to switch to another data-structure such as a bit-vector or a Red-Black tree. This is possible because of the recursive nature of vEB trees.

For example, to decide membership of an element over some universe is done by testing for inclusion in the first level. If it is not min or max we recurse to a descendant node that can either be a van Emde Boas tree or another structure that supports answering the same queries. The logic that goes into placing another structure on a specific level is not difficult to generalize such that the change can be implemented at an arbitrary level, where level is based on the size of the universe.

We have implemented a van Emde Boas tree that allow switching to a bit-vector when the universe falls below some constant and in our testing we will examine what the value of this constant means for the performance of our implementation. We will call a one-layered van Emde Boas Tree for van Emde Boas Combo 1, two layers for van Emde Boas Combo 2 and so forth.

Furthermore, where as the first van Emde Boas implementation is based on the [**?**] this implementation is based on Gudmunds article [**?**]. Furthermore, we are holding not just the min element outside of the data-structure but also the max element.

With some constant in mind, we can consider the total universe u_{total} ($0....2^{24} - 1$ for this handin) as the product of two different universes u_{vEB} and u_{bit} the universes decided exclusively by the vEB tree and bit vectors respectively. Now the van Emde Boas tree that utilized a bit vectors for a leaf does not, strictly speaking, decide on membership of its elements but it decide which bit-vector would contain the element if it is present.

Operation	Red-Black Tree	van Emde Boas Tree	Comb tree
MEMBER	$\mathrm{O}(\log n)$	$\mathrm{O}(\log \log u)$	$\mathrm{O}(\log \log u_{veb})$
MINIMUM	$\mathrm{O}(\log n)$	$\mathrm{O}(1)$	$\mathrm{O}(1)$
MAXIMUM	$\mathrm{O}(\log n)$	$\mathrm{O}(1)$	$\mathrm{O}(1)$
PREDECESSOR	$\mathrm{O}(\log n)$	$\mathrm{O}(\log \log u)$	$\mathrm{O}(u_{bit} + \log \log u_{veb})$
SUCCESSOR	$\mathrm{O}(\log n)$	$\mathrm{O}(\log \log u)$	$\mathrm{O}(u_{bit} + \log \log u_{veb})$
INSERT	$\mathrm{O}(\log n)$	$\mathrm{O}(\log \log u)$	$\mathrm{O}(\log \log u_{veb})$
DELETE	$\mathrm{O}(\log n)$	$\mathrm{O}(\log \log u)$	$\mathrm{O}(u_{bit} + \log \log u_{veb})$

Chapter 6

Testing trees

In this chapter we discuss the different tests we decided to implement plus the results of running said tests on our implemented trees.

6.1 Deciding on test-size

We were asked to implement van Emde Boas trees with a universe of size 2^{24}. The combo structure is also defined on this universe but in contrast the Red-Black Tree supports all numbers in the 32-bit integer-range.

To make matters more difficult all operations on the Red-Black tree is bounded by the height of the tree which is $\log n$ where n is the number of elements in the tree, where as the van Emde Boas Tree is bounded by the size of the recursive data-structure which is $\log \log u$ where u is the size of the universe. Therefore, we need to express n as an expression of u or u in terms of n. We chose the first approach and will refer to this as the fill rate.

This also forces us to rethink how testing is done. One could for example do a test of the INSERT operation by measuring the time of inserting the first 1000 elements, then 10.000 elements, 100.000 elements and so forth, which is one way to test the trees. Since the Red-Black tree time-complexity is bounded by the number of elements in it the van Emde Boas tree should take the same time no matter how many elements in the tree up to a constant, one could theorize that plotting in the results would yield two trendlines that would intersect for some x having the trendline for the Red-Black tree being below for small instances of x. What that measurement would give is then when, if one would do all inserts, it would be beneficial to use either in total time.

That is certainly a valid answer, but we do not believe one would just insert many nodes into the tree and then stop using it in practice. Therefore, we wanted to know when the time of doing one operation in a tree is most beneficial compared to a certain fill rate. We therefore divided the universe up in 1000 parts, which would give an interval size of 16.777 elements = EPP (EPP = Elements per promille). Then every test is performed by first filling the tree up to a fixed promille and then doing the same operation one promille times. For example, testing how long it takes to insert elements for fill-rate 50 %, we fill up the tree with $500 * EPP$ and then perform EPP inserts. In

our believes these tests makes more sense if the tree is mantained over a longer period, where the implementor knows roughly how many elements will be in the tree. As a sidenote, if the author has no intuition about the number of elements in the tree at all, it would probably be better to use the van Emde Boas Tree anyway as our results shows. All tests has been tested with fill-rates 1, 3, 5, 7, 9, 10, 30, 50, 70, 90, 100, 300, 500, 700, 900 EPP. PREDECESSOR is not tested since this is completely analog to successor.

6.2 Measuring time

For this project we decided to change the way we measured time when running the tests. For one, we did not want the construction time of the van Emde Boas Tree taking into account since this is just a constant. This required us to measure time from within the program.

Using the normal clock gives different results depending on the platform. On Ubuntu the result is rounded to the nearest 1000 clock-cycles but in OS X there is no rounding. We chose to measure time by using clock_gettime which is inputd in time.h in Ubuntu with the argument CLOCK_PROCESS_CPUTIME which gives us the time the process has spent being active in nanoseconds.

6.3 Results

In this section we present our results for each test. As a reminder, all times are in nanoseconds and a fill-rate of 100 translates to $100 * EPP = 2^{24}/10$.

6.3.1 Results for testing Insert

Testing INSERT just fills the tree with random numbers from the entire univers and insert EPP elements.

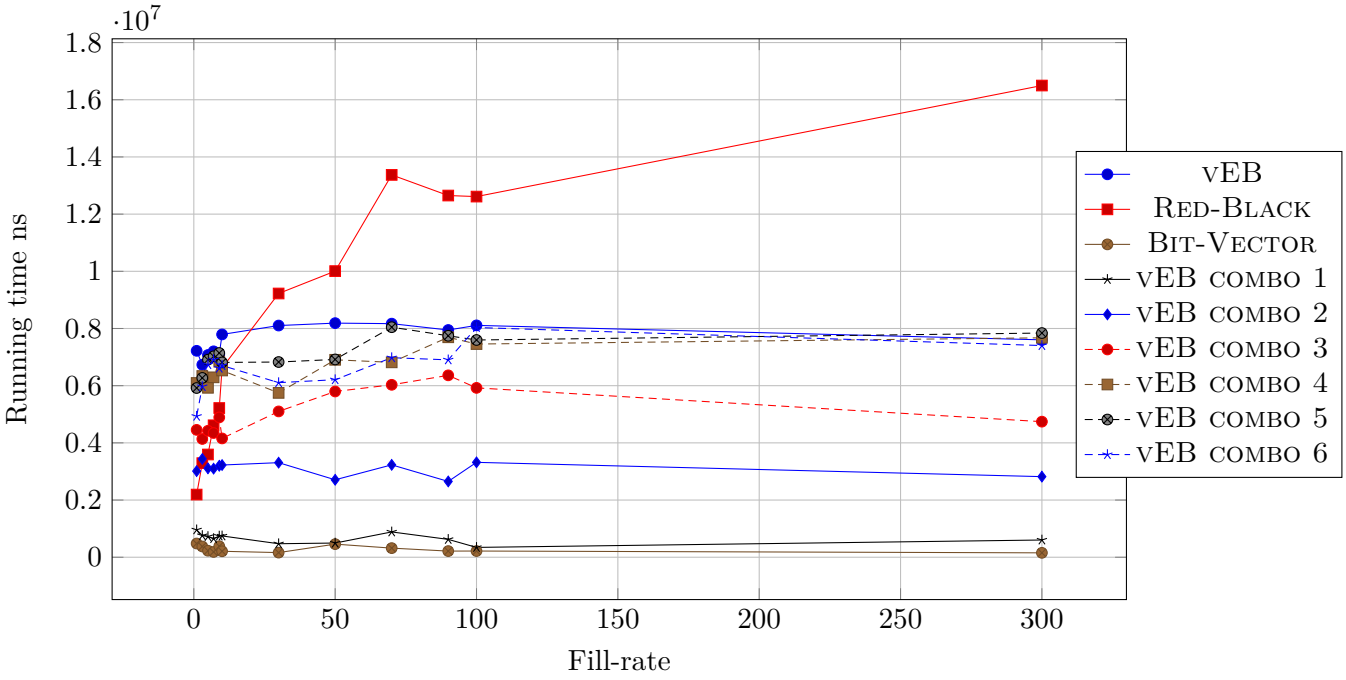

Figure 6.1: TITEL

The first test is testing inserts, and we can see that Bit-Vector is the fastest. This comes as no surpise since this can be done in constant time. All of the vEB trees insert in time independently of the fill-rate whereas the Red-Black tree takes longer time depending on how many elements that are in the tree.

What is suprising though is that one layer and even two layers of vEB with bit-vectors as leafes performs very well. What is also suprising is that it only takes around 20 promille before the Red-Black tree is slower than all other tested data structures.

6.3.2 Results for testing Delete

Testing Delete just fills the tree with random numbers from the entire universe and deletes EPP elements. The graph is adjusted to only shwo the fill-rates for 1 to 50, which is no more 5 percent of 2^{24} number of elements.

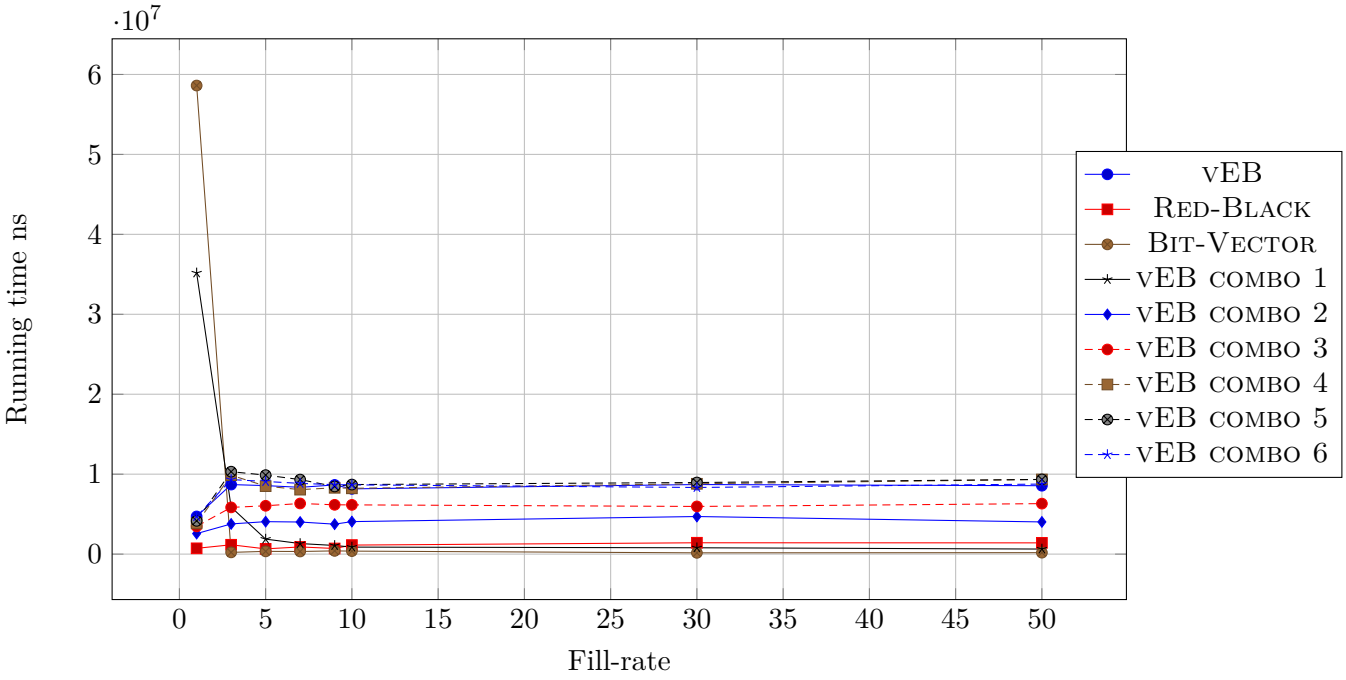

Figure 6.2: TITEL

It is no suprise that the van Emde Boas Trees are constant with regards to the number of elements in the data structure, however it is suprisingly that DELETE for the Red-Black tree almost is constant too - even for large number of elements. The explanation must be that we are asking to delete elements that exist in the tree such that the amount of misses is kept low and thereby not forcing the algorithm to of DELETE to traverse all the way to the bottom too many times.

DELETE for the Bit-Vector may have to update min or max by using a SUCCESSOR or PREDECESSOR search. However, this is linear worst-case and with a small number of elements in the vector this performs very badly. However, after just 5 promille fill it the most effecient.

6.3.3 Results for testing Member with guaranteed find

Testing MEMBER by filling the tree with random numbers and then takes the first EPP numbers in that sequence and ask for membership. This test is designed so that we are certain the member exists.

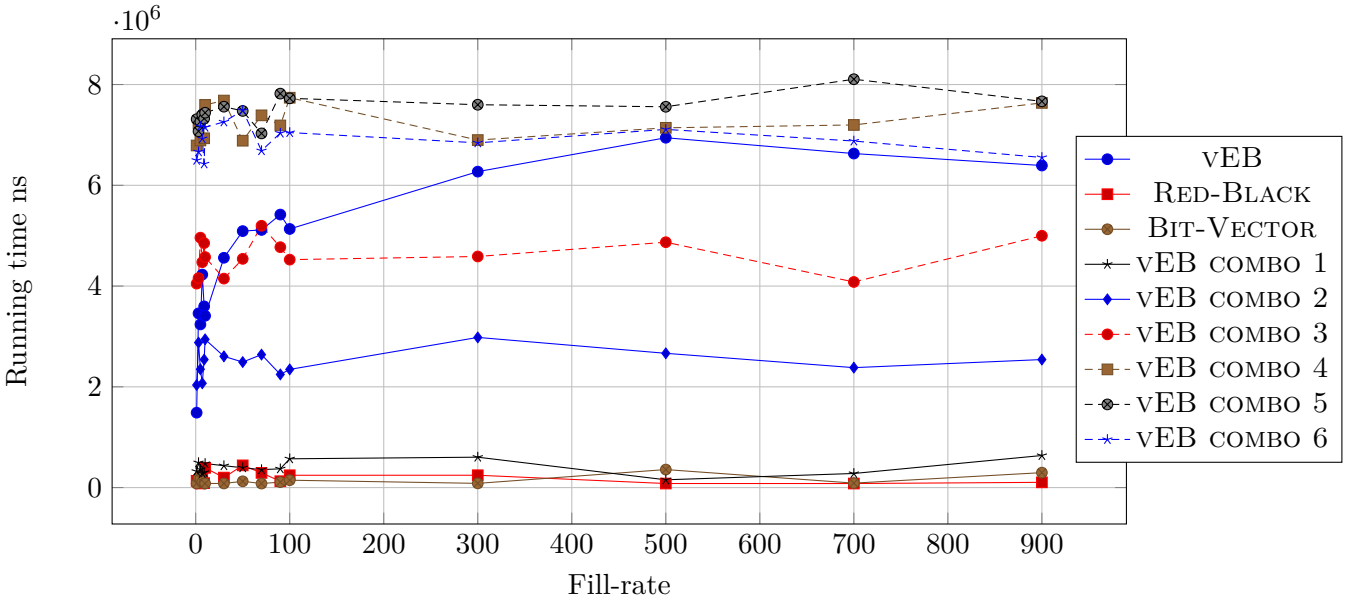

Figure 6.3: TITEL

In this test it is very clear that the more levels of van Emde Boas trees the slower the data structure will be performing Member search. We also see that the van Emde Boas tree implemented from [?] increases do not look constant. This is propably due to how the arithmetic calculations are implemented and we will discuss this later in the report.

6.3.4 Results for testing Member with un-guaranteed find

Almost the same as above but again we have another sequence of random numbers that we search for. In this way the probability of the element existing in the data-structure increases as the fill-rate increases.

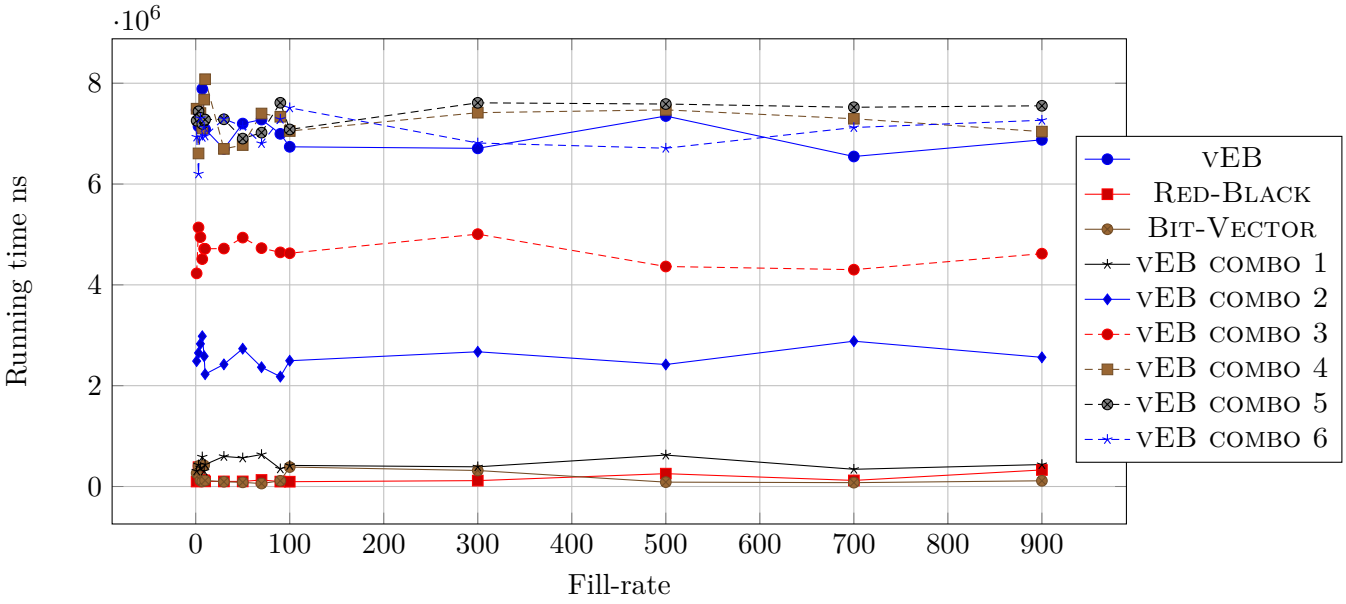

Figure 6.4: TITEL

The results are pretty similar to the results above but but the lines seems more straight. This is probably due to the number of misses is higher such that searching in the interval is more consistent.

6.3.5 Results for testing Successor with guaranteed find

Testing Successor by filling the tree with random numbers and then takes the first EPP numbers in that sequence and ask for their successor. This test is designed so that we are certain the successor exists.

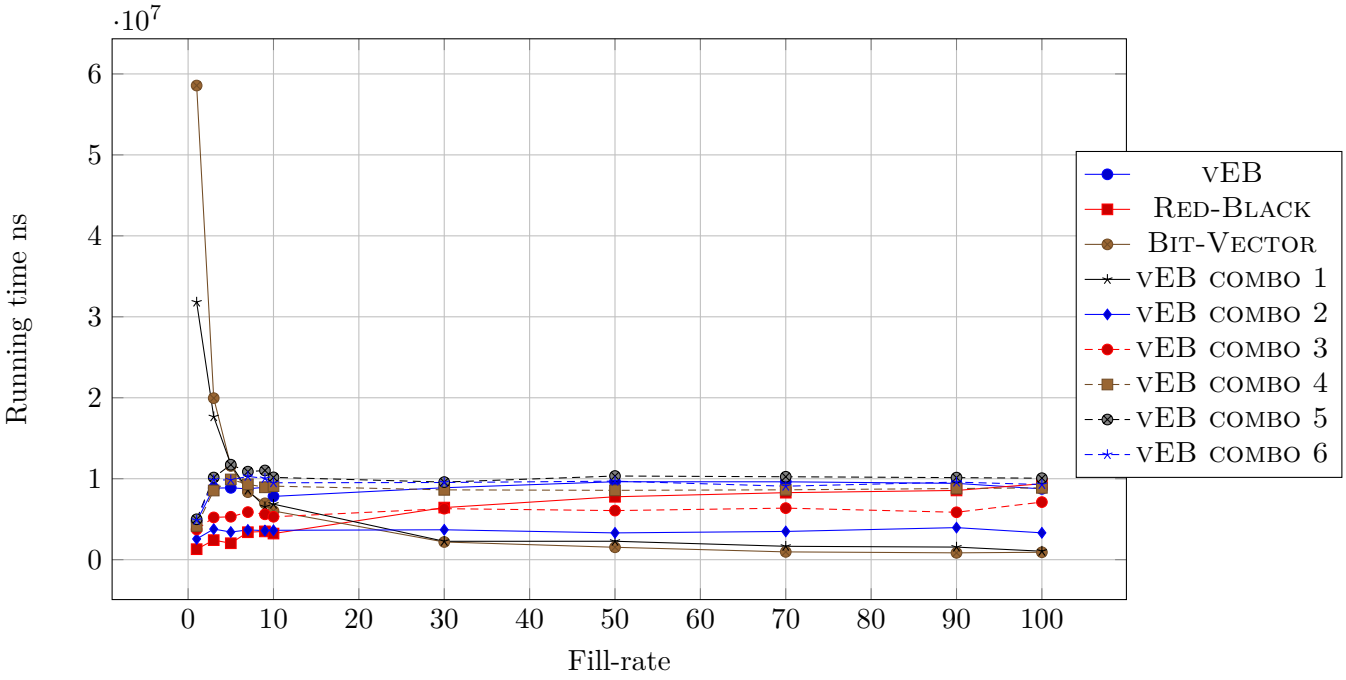

Figure 6.5: TITEL

Here we have adjusted the graph to just show a fill-rate between 1 and 100. It is pretty obvious that the Red-Black Tree will continue to grow and take longer and longer time where the van Emde Boas tree is approaching constant time. The full graph can be found in the appendix.

Again is is facinating that it only takes around 0.5 percent before the Bit-Vector and the van Emde Boas tree with one level has competitative running times.

6.3.6 Results for testing Successor with un-guaranteed find

Almost the same as above but we then generate another sequence of random numbers that we try to find the successor for. In this way we have no idea how easy it is to find the successor and might resemble a more general use-case.

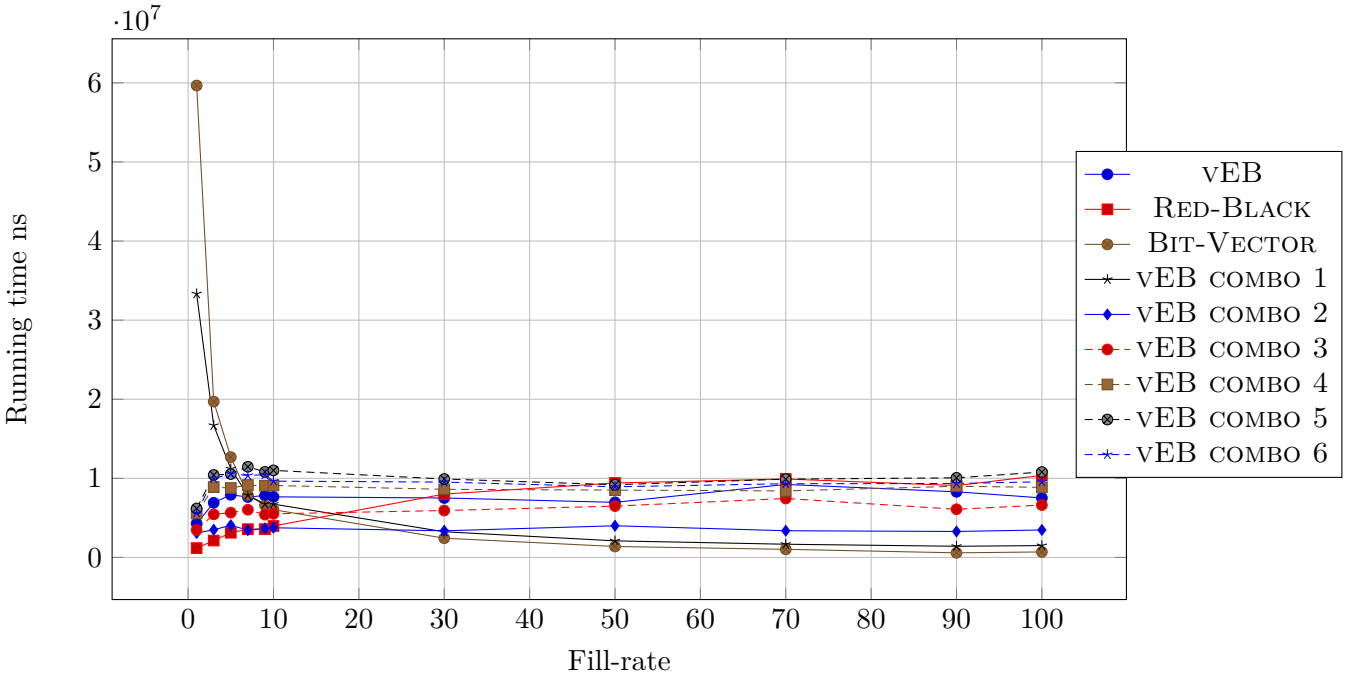

Figure 6.6: TITEL

Compared to the results before where we had a guaranteed member there is almost no change for the Bit-Vectors and the 1 level van Emde Boas Tree, because the check that the Bit-Vector makes is the same no matter if an element is in the vector or not.

It is a bit disappointing not seeing a larger change for the Red-Black tree around 100 fill-rate which is about 1.6 million elements.

6.3.7 Results for testing Minimum

Testing Minimum by calling Minimum EPP times. The default implementation of a Red-Black Tree do not maintain min and max, so we expect the Red-Black tree to perform much worse than the other data structures when the amount of elements increases.

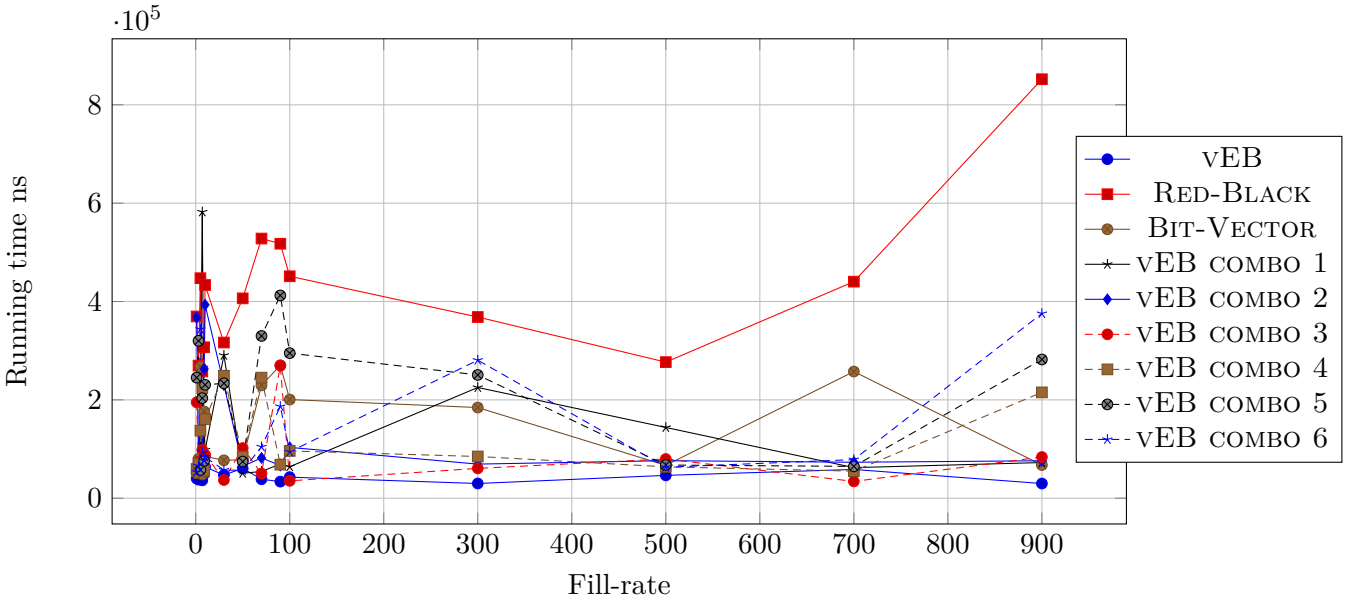

Figure 6.7: TITEL

As we can see, the Red-Black tree is slow, even for very small fill-rates. But the results do not entirely match what we anticipated. From fill-rate 100 to 500 the Red-Black Tree actually decreases before it finally rises again. The lowest point in this trendline is actually when the tree is filled with half of the universe.

Every van Emde Boas tree also has something that looks like a similar pattern in the time usage when the tree is half-full. This seems strange since the, however we should note that the difference is around 0.1 of a millisecond.

6.4 Summary on search tree results

The combo tree with two levels of van Emde Boas seems to be the best overall choice considering running time and fill-rates. The bit-vector and the one-layered combo tree suffers from very high query times for successor and predecessor when the fill-rate are below 0.5 0.7 percent which roughly translates to about 100.000 elements, so the combo-tree with two layers looks like a perfect candidate when considering the results of our tests.

Chapter 7

Testing trees as if they where heaps

In this chapter we will see how our search trees and Bit-Vector performs compared to our heap implementations from a previous project in the course. We will first briefly describe the priority queues we implemented and their running times, then describe our manufactured tests and finally show how the structures performs on said tests.

7.1 Priority Queues from Project 1

In our last project we implemented four different heaps - A binary heap that used an array as storage, a binary heap that used nodes and pointers and two Fibonacci Heaps.

Operation	Binary heap	Fibonacci heap v1 (amortized)	Fibonacci heap v2 (amortized)
MakeHeap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$\Theta(1)$	$O\left(l(\log(\frac{n}{l})+1)\right)$
Insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
DeleteMin	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$O(1)$	$O(1)$
Delete	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
Meld	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

7.2 Designing tests

We made a series of tests in the previous project one being inserting a number of elements and deleting them again. The test had the following results:

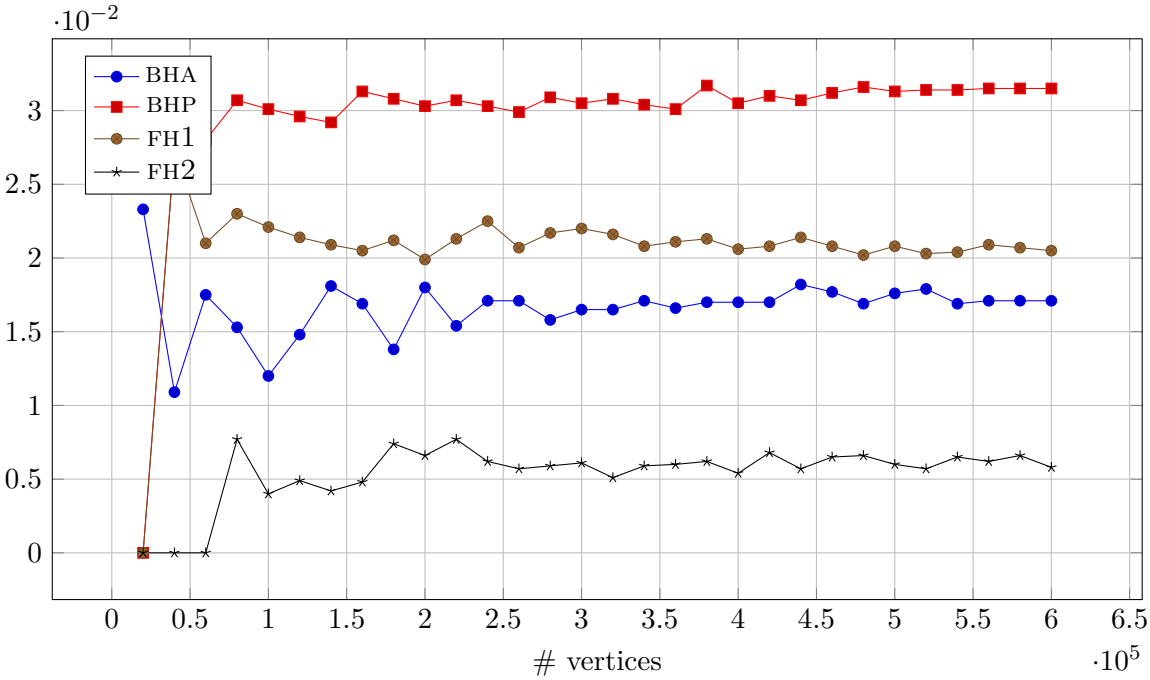

Figure 7.1: Running time divided by *nlogn*

In general the Fibonacci v2 and Binary Heap with arrays performed the best but as with testing the Red-Black search tree against the van Emde Boas Tree, we have to adjust our test sizes to be expressed in terms of the universe size.

Our search trees do not support the DecreaseKey operation, so this has been cruedly implemented as a Delete operation followed by an Insert operation. Also, we will not test melding or member search since this is not something both types of data structures support.

Another thing to notice is that our heaps require direct indexing for the Delete operation. For each item we pass to any of the heaps a constant lookup will give the node (position for the array implementation). This can be achievede for the Red-Black tree, which we have done, but not for The van Emde Boas because we only support storing keys from the universe. We do not expect any of the trees to perform better than the heaps for any of the operations besides FindMin which is a constant operation for all but Fibonacci v2.

Since the Fibonacci heaps have a somewhat lazy approach to when work has to be done, we have decided that we will test the heaps by forcing all remaining work to be completed, which generally amounts to calling DeleteMin and FindMin. Measering and fill-rates are the same as for testing trees.

7.3 Results

In this section we present the test and the results accompanying them.

7.3.1 Results for testing Insert

The test for INSERT inserts a certain promille of the entire universe into each data structure followed by a single DELETEMIN and a single FINDMIN.

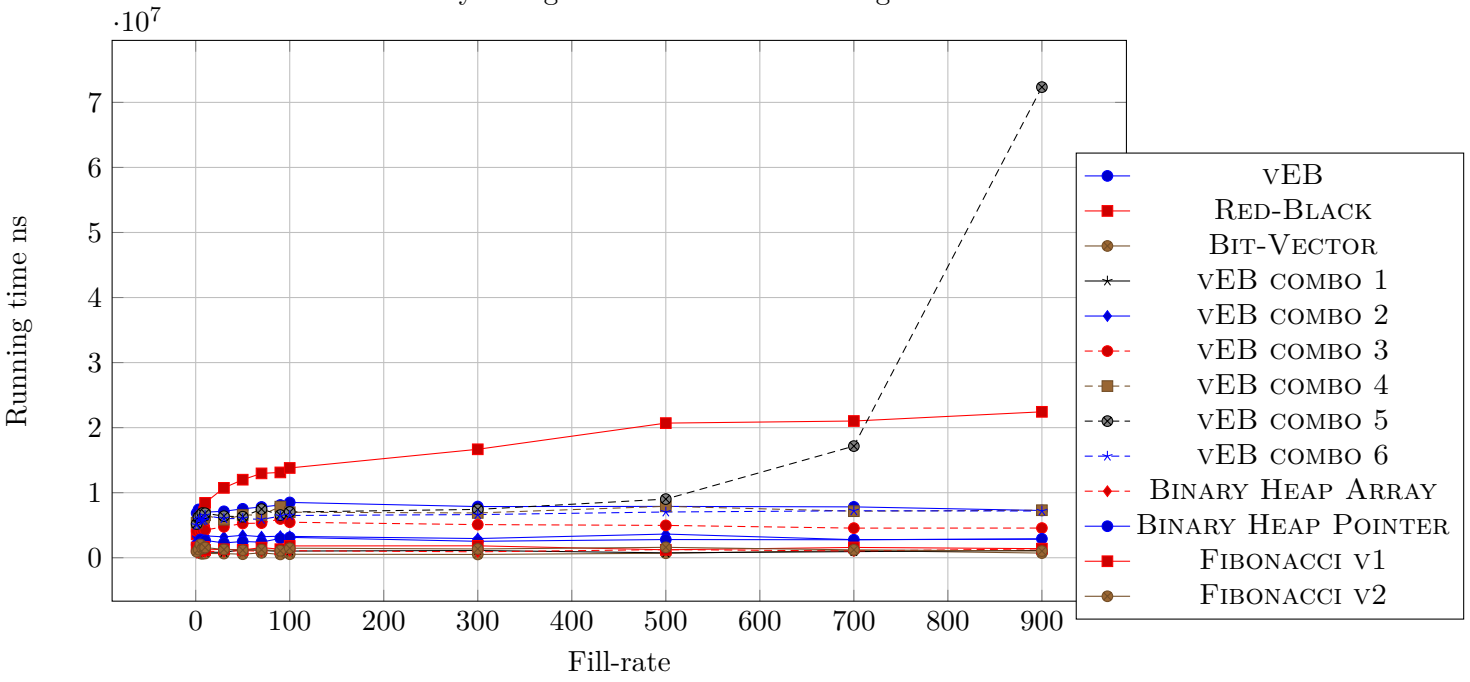

Figure 7.2: TITEL

As we can see the Red-Black tree is taking the longest except for one layer of van Emde Boas combined with a bit-vector. The Red-Black tree must wastes some effort being continously balanced so this comes as no surprise, however, it is strange that the one layer of van Emde Boas is performing that bad, but we figure it must be the overhead of bringing such large vectors into memory without giving the computer any possibility of caching. It is a bit hard to see what transpires when the fill-rate is small, so the following figure shows the change from 1 to 50 promille:

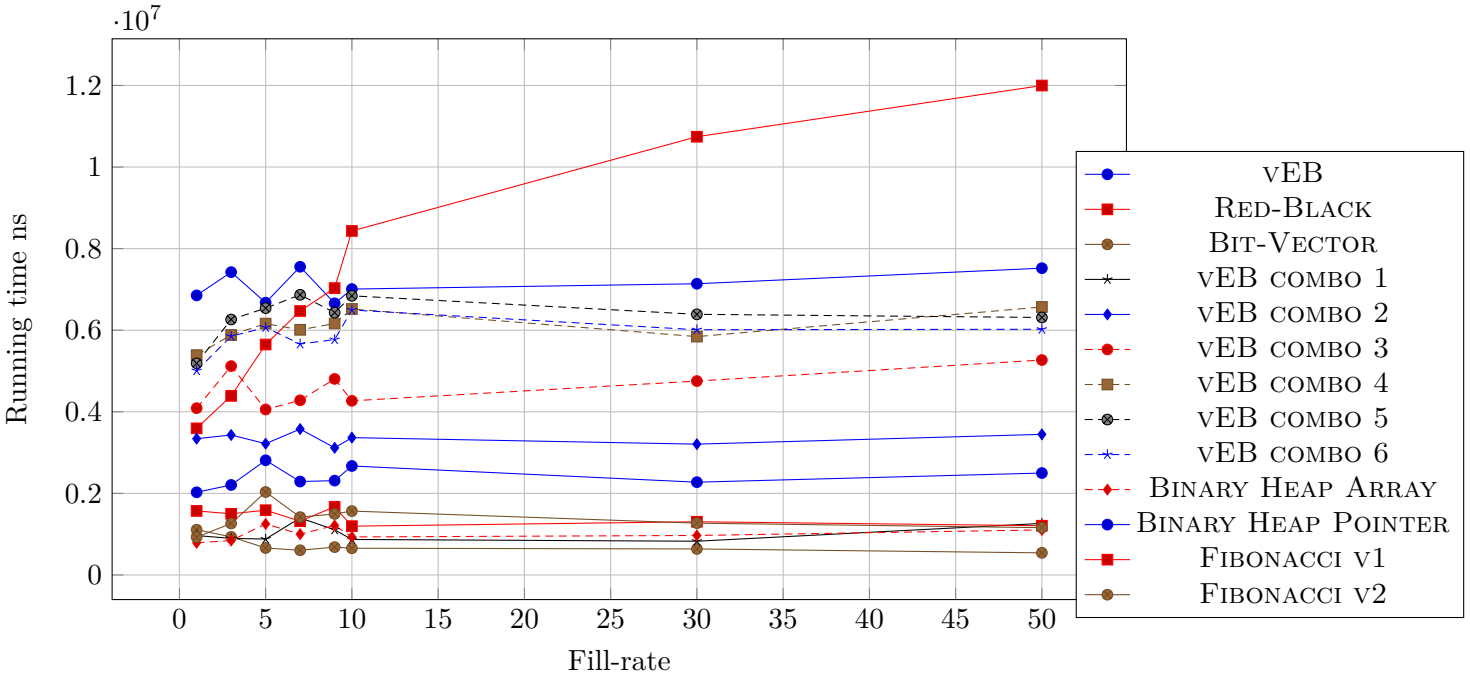

Figure 7.3: TITEL

Here we can cleary see that the heaps do perform the best, only interupted by the Bit-Vector which ofcourse has extremely fast inserts. A bit suprising is it to see that van Emde Boas Combo 2, that has two layers, almost matches the binary heap with pointers.

7.3.2 Results for testing Delete

In this test we fill the data-structures with elements and then remove one promille (16.777) from the data-structure. The deleted keys are guaranteed to be in the structure since our heaps require an existing item to use for deletion. The most action happens inside 50 EPP:

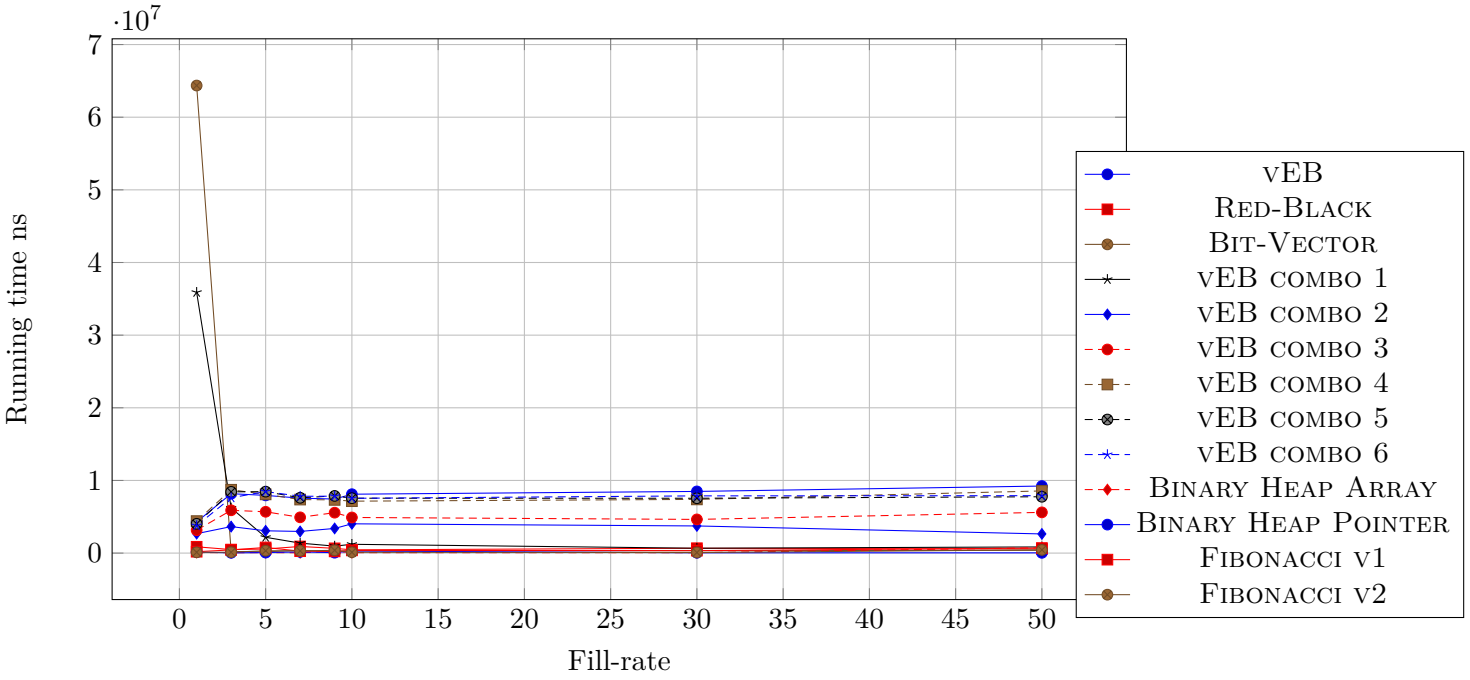

Figure 7.4: TITEL

As before in the previous tests the Bit-Vector and one layer performs bad for small fill-rates because that requires to search longer in the array for a minimum and a maximum if one is to be delete.

We see the four heaps performing the best but again followed by the two-layered van Emde Boas Tree, especially for large fill-rates.

7.3.3 Results for testing DeleteMin

In this test we fill the data-structures with elements and the deletes the minimum element EPP times.

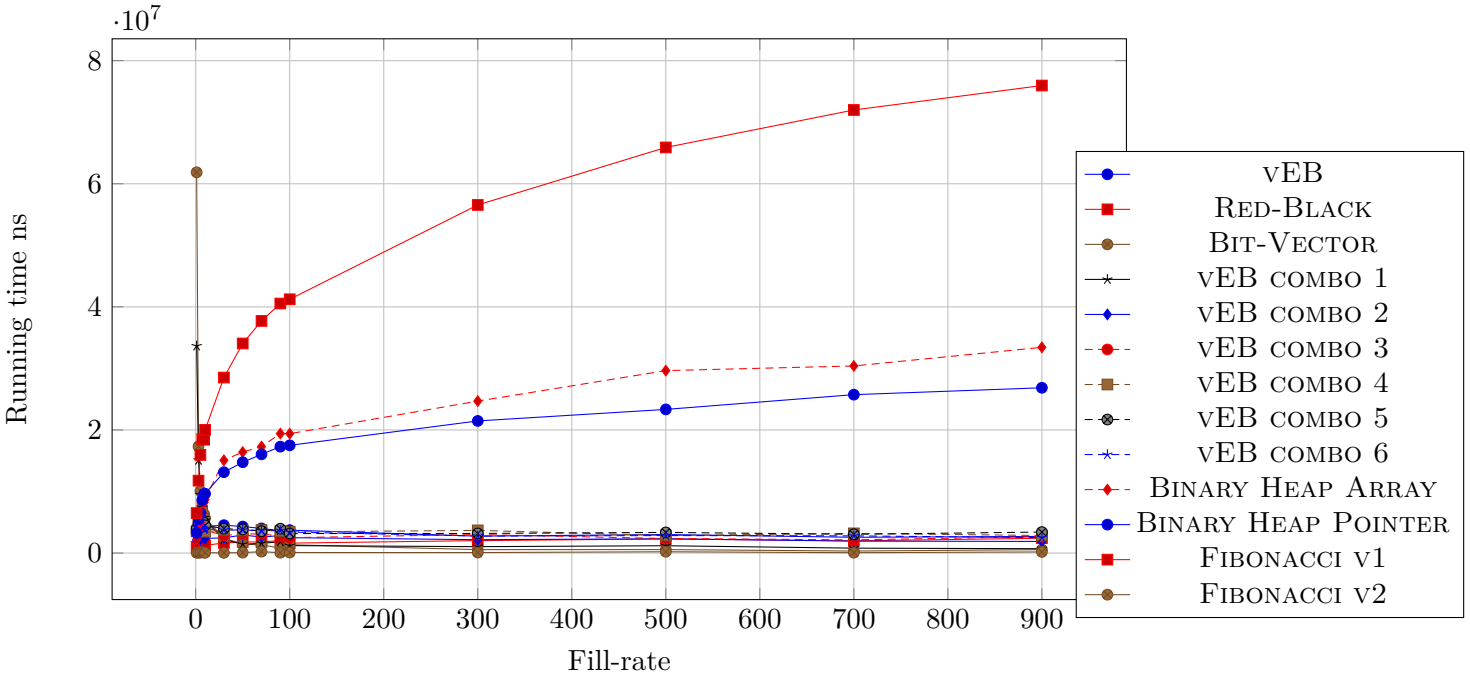

Figure 7.5: TITEL

We already knew the Red-Black tree would perform rather bad when the fill-rate rose, and we also know that the Bit-Vector and 1-layer van Emde Boas would have a hugh penaly for very small fill-rates. We did not anticipate the two binary trees performing this badly and we did not expect the remaining implementations to be inside 1 millisecond from each other for fill-rates 50 promille and up.

7.3.4 Results for testing DecreaseKey

In this test we fill the data-structures with elements and the decreases EPP keys by a random number in the interval of 0 and the value of the current key such that no negative keys can occur.

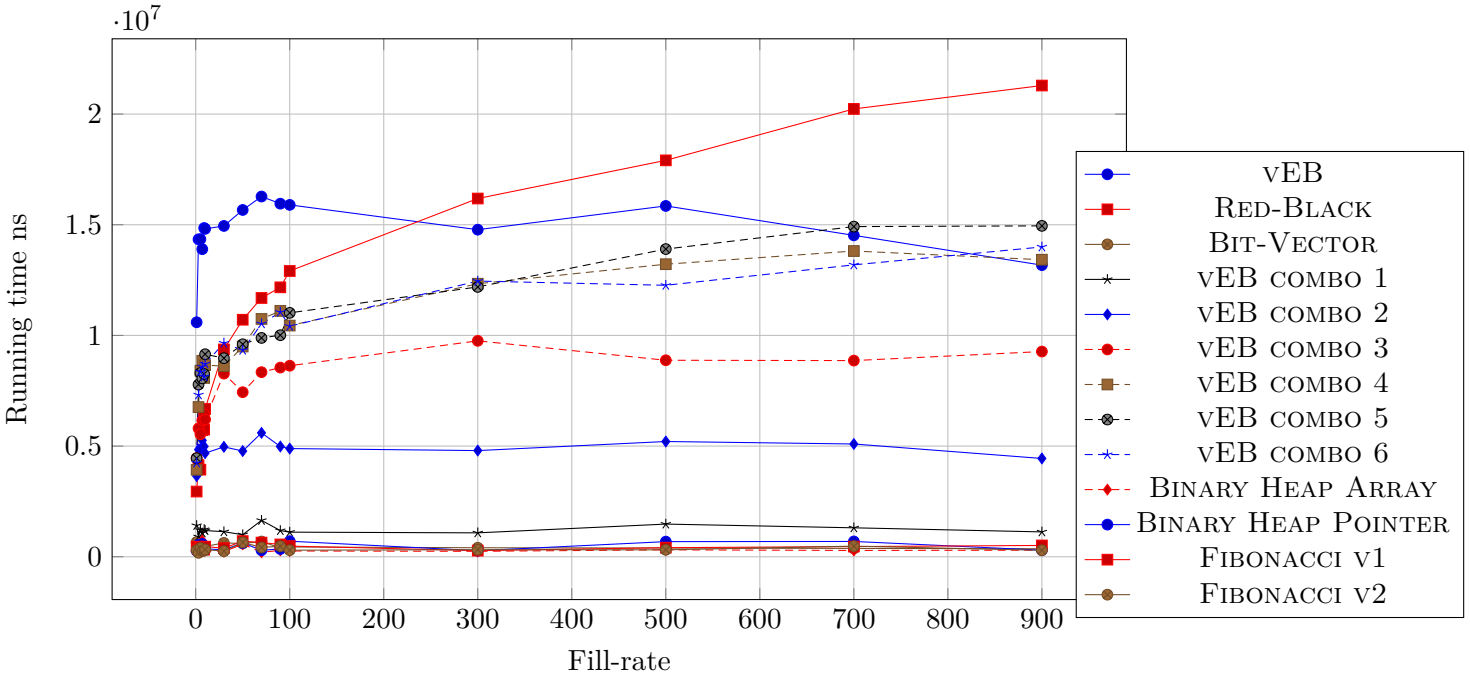

Figure 7.6: TITEL

The four heaps are staying at the bottom and the Red-Black tree again shows that it behaves the wors for large fill-rates. Interestingly, both the Bit-Vector and the 1-layer van Emde Boas has almost the same running times. Each additional layer of van Emde Boas almost has the same difference between them.

7.3.5 Results for testing FindMin

In this test we fill the data-structures with data and then try to find the minimum element EPP times. This should not be unfair to Fibonacci v2 since the test is constructed such that the entire data-structure has no remaining work.

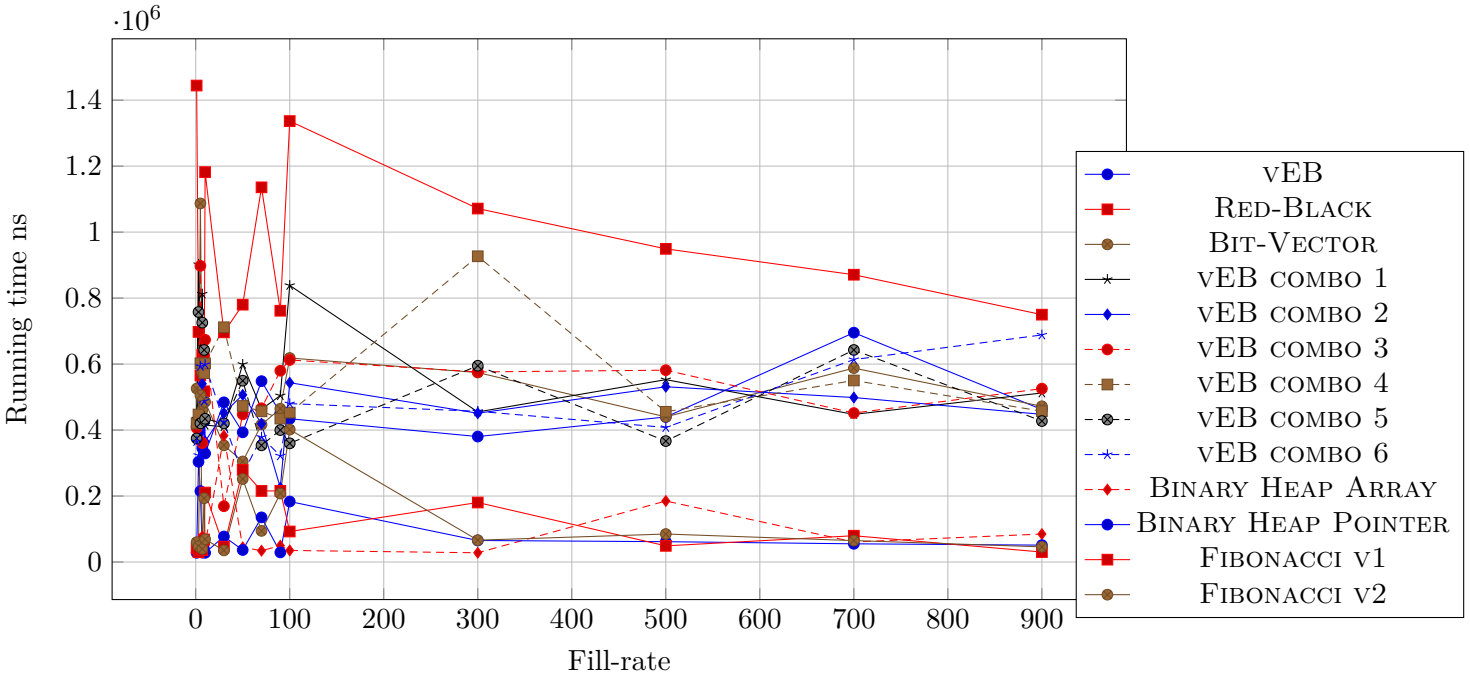

Figure 7.7: TITEL

The results are a bit more spread out this time and there are of odd points from 0 to 100 in fill-rate. However, one should notice that the tics on the y-axis is 200 microseconds, so a context-switch can have a lot of impact. We see the Red-Black tree perform the worst, which makes sense since it has no constant time lookup for Minimum. The Bit-Vectors and van Emde Boas Trees are placed nicely int he middle and then we have the heaps at the bottom.

7.4 Summary on priority queues vs search trees results

We must admit we were rather suprised by the outcome of the tests. We had anticipated that the search trees perform much worse than they had and the only test we thought it would tie, FindMin it wound up loosing. The performance of the van Emde Boas Tree with two layers and then Bit-Vectors seems as a fair compromise if one would not put in the effort to build a heap.

Chapter 8

Arithmetic operations

Everytime we recurse in our van Emde Boas trees we perform the same arithmetic operations to figure out where to divide the bits in an upper and lower half. As a result of the combo tree implementation, it was more natural to keep the universe in size of bits than having the universe in integer representation.

This lead us to investigate if there was a difference in how the computer calculated the operations. Corman et al writes in their implementation that higher can be found as $\uparrow\sqrt{u} = 2^{\lceil (\log u)/2 \rceil}$. A naive implementation might be something like:

```
pow(2, (int)ceil((log2 (u))/2));
```

So this would make a few different calls to the math package being log2, ceil and power. log2 can be found by MSB which we have seen a constant time operation for, but we are trying to bring down the number of instructions and measure the total time. There is also a cast in there. It would be fun to see if we could do it faster. All tests has been run with a Clang compiler with 3rd level of optimization, so we do not expect to get results that are much higher . This is the different ways we calculate $\uparrow\sqrt{u}$:

```
1: pow(2, (int)ceil((log2 (u))/2));

2: result = 1<<(int)ceil((log2 (u))/2);

3: uint32_t b = log2 (u);
   uint32_t h = (b >> 1) + (b & 1);
   result = 1<<h;

4: result = 1<<(int)ceil((log2_x86 (u))/2);

5: uint32_t b = log2_x86 (u);
   uint32_t h = (b >> 1) + (b & 1);
   result = 1<<h;

6: uint32_t b = log2_table (u);
   uint32_t h = (b >> 1) + (b & 1);
```

```
result = 1<<h;
```

log2_x86 is just the MSB instruction that can be found on x86 architectures with the assembly instruction bsr. log2_table is a computed table that gives the result by a lookup. We are performing 100.000 calculations and then divide the total time by the same amount. We have compiled all the tests the Clang compiler without any optimizations and the Clang compiler with the most aggresive optimatization. We will use 1 as the measurement for how well the other test performs:

Method	Clang no opt	Clang with opt	% faster no opt	% faster opt
1	288	185	0	0
2	179	171	38	8
3	178	170	38	8
4	173	163	40	12
4	174	161	40	13
6	182	164	37	11

We must assume that the most people would compile their van Emde Boas trees with optimization turned on, if not, a better arithmetic approach could yield a 40 % quicker tree. But still, if optimization is turned on, the crucial arithmetic used for seperating bits can still be made 13 % quicker.

Chapter 9

Conlusion

We have shown a total of 9 different data-structures that each perform the duties described by a search tree. We have shown that the best candidate when considering run times is a combo tree with two layers of van Emde Boas trees and a bit-vector of size 64????? at every leaf.

We also showed that the search structures all performs worse than our heap implementations from a previous project, however, the combo tree with 2 layers performs the best of the trees and is very close to the binary heap with pointers.

When running our tests on the data structures we took interest in how to find the top and the bottom part of the bits effectively. We showed that the calculation for upper could be done 40 % faster if no optimization was used when compiling and 13 % faster with optimization being turned on compared to a straight forward naive approach.