Advanced Datastructures: Project 2
Van Emde Boas Trees

Jan H. Knudsen (20092926) Kris V. Ebbesen (20094539)
Roland L. Pedersen (20092817)

January 9, 2014

Contents

1 Introduction

Heaps are a very useful kind of data structure, and several algorithms depend one them. As seen in the previous project, choosing the right heap for the job at hand can prove to be quite an improvement in running times.

In this project, we will look at Van Emde Boas Trees, and how they can be used to create a quite efficient heap. Van Emde Boas trees are quite peculiar data structures, whose upper bound for the `Insert`, `Delete` `Member` and `Predecessor` are $O(\log(\log(U)))$, with U being the universe size. However, they have they drawback of not supporting using $O(U)$ memory, unless we have access to perfect hashing.

In this project we hope to show that by limiting the key type of the heap to be 24-bit integers and not supporting the `DecreaseKey` operation, we can construct a very efficient heap backed by a Van Emde Boas tree. Also, we wish to compare the performance of Van Emde Boas trees to more general trees supporting the `Predecessor` operation, specifically Red-Black trees, and hope to show that the Van Emde Boas trees are superior even on these simple operations.

The code can be found at:
`https://gist.github.com/Shne/31848ee3af3ab6e0d809`

2 Implementation Details

2.1 The Basic Van Emde Boas Tree

The Van Emde Boas Tree was implemented following the description in Cormon et al.[1]. This version of the tree is fairly basic, but provides the full features of the Van Emde Boas tree.

A few design choices of note are:

The Base Case When the Van Emde Boas Tree reaches 1 bit required for its universe (a 2-element universe), we skip most of the normal operations done in the tree, and maintain the tree fully in the `min` and `max` values of the tree.

Element Data Type Since we need to support integers of up to 24 bit, we store data values as unsigned 32 bit integers.

The `null` Element When we wish to signal that the `min` or `max` element is empty, we set its value to -1, which when converted to an unsigned integer will be the maximum value possible, and outside of the 24 bit universe.

Specifying Universe Size When the Van Emde Boas recieves the information about the required bits for its input, is does it by a templated variable. This ensures full compiler optimization of branches based on bit size, and saves storage space.

Splitting the Elements When elements are split into a high and low part, they are done so by selecting the high and low half of the bits. In case of an uneven amount of bits, the high part receives the most bits.

Input Validation When testing the tree, assertions check that we do not specify values that are outside of the universe. These assertions are disabled when doing performance testing.

2.2 Making a Heap based on the Van Emde Boas Tree

Implementing a heap on top of a Van Emde Boas Tree is fairly simple. Since we already know that a Van Emde Boas Tree requires storage space linear to the universe size, we simply allocate an array of the size of the universe, and use this for storage, while using the tree to store any keys.

Now most heap operations can be performed by doing a single operation on the Van Emde Boas Tree, and a single read or write on the array.

Note that DecreaseKeys are not supported, since they make little sense on a Van Emde Boas Tree. Instead, if one wishes to decrease a key, simply do an insert and a delete.

2.3 Small Universe Van Emde Boas Trees

When the universe size of a Van Emde Boas Tree falls below 6 bits, we can store the entire tree in a single 32 bit integer. Any operation on the tree can now be performed using a constant number of bitwise operations, assuming we can find the first and last bit set in constant time. Luckily, most processors support these operations.

This should save not only space, but also execution time.

Note that we store the 32-bit integer in the space normally holding the pointer to the top tree, to save memory. Also, since we specify the number of bits required for the universe, we get full compiler optimization for the branches determining whether or not to use integer instead of the normal Van Emde Boas tree operations.

2.4 Future Improvements

Our Van Emde Boas Tree lacks an important optimization often used in practical implementations of the data structure, which is the on-demand storage of the subtrees. Basically, the idea behind this optimization is to only store subtrees that actually contain elements. This is done by keep a map on the subtrees rather than an array, and adding or removing trees as needed.

This however requires HashMaps that grow and shrink as needed, which the unordered sets of the standard library does not. Also, this might make the worst-case performance of the tree worse, if the map does not do these operations in constant worst-case time.

3 Computer Specs:

Model:	ASUS Zenbook UX32VD-R4013H
OS:	Ubuntu 13.10
CPU:	Intel Core i7-3517U, Quadcore, 1.90 GHz
Memory:	6 GB

4 Heap Tests

4.1 Experiment Details

The tests were run on a laptop using a few scripts to automate the testing and formatting of data. Valgrind with the Massif heap profiler was used to obtain data about the size of memory used during testing. This heap profiling was done in a seperate test run to not affect the running times of the main test. Each test was run with various amounts of operations, all powers of 2: $2^{18} = 262144$, $2^{19} = 524288$, $2^{20} = 1048576$, $2^{21} = 2097152$, $2^{22} = 4194304$, $2^{23} = 8388608$ and $2^{24} = 16777216$. The number of operations is given to each test as input parameter n.

4.1.1 Test Types

Three types of test was done:

TestInserts Testing the running time of `Insert` operations by measuring the time it takes to insert elements $n - 1$ to 0 in that order.

TestDeleteMin Testing the running time of `DeleteMin` operations by first inserting elements like in the TestInserts test without measuring the time and then running `DeleteMin` the same amount of times while measuring time.

TestInterleaved Testing the running time of interleaved `Insert` and `DeleteMin` operations by first inserting elements $n-1$ to $\frac{n-1}{2}$ elements without measuring time, and then interleaved calling `DeleteMin` and `Insert` on elements $\frac{n-1}{2}$ to 0 while measuring time.

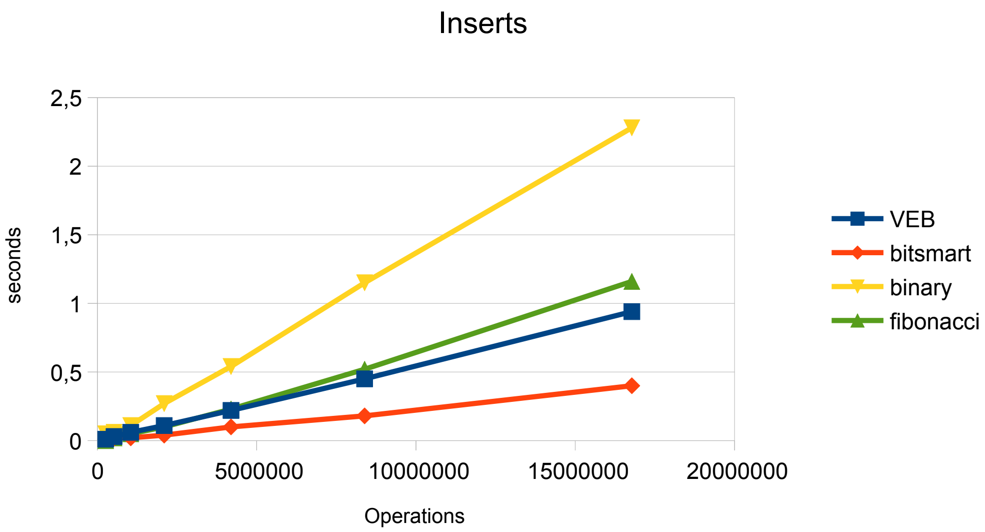
4.2 Expectations

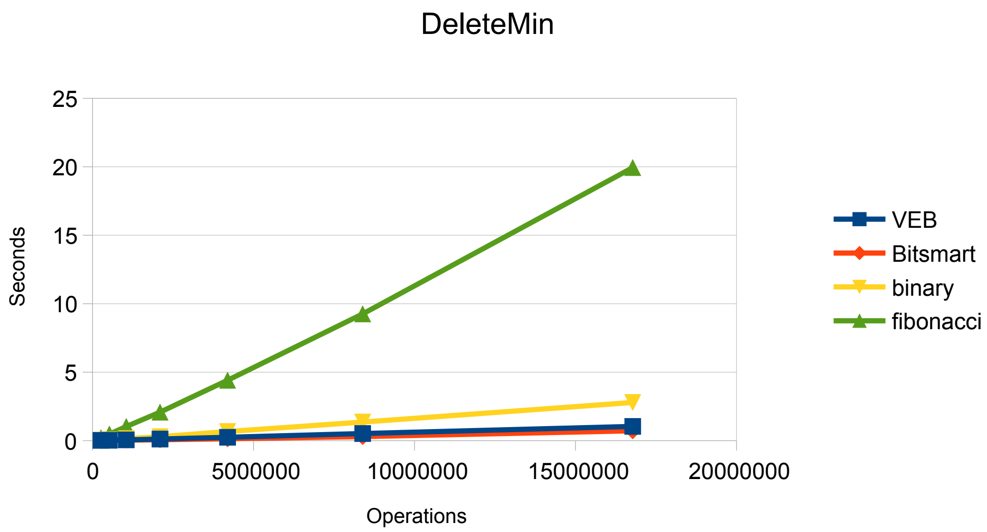
We expect the Van Emde Boas Trees to perform quite well in terms of speed. Their upper bound on comparisons and general operations of $O(\log(\log U))$, will most likely make them outperform the more general heaps, especially as data sizes grow.

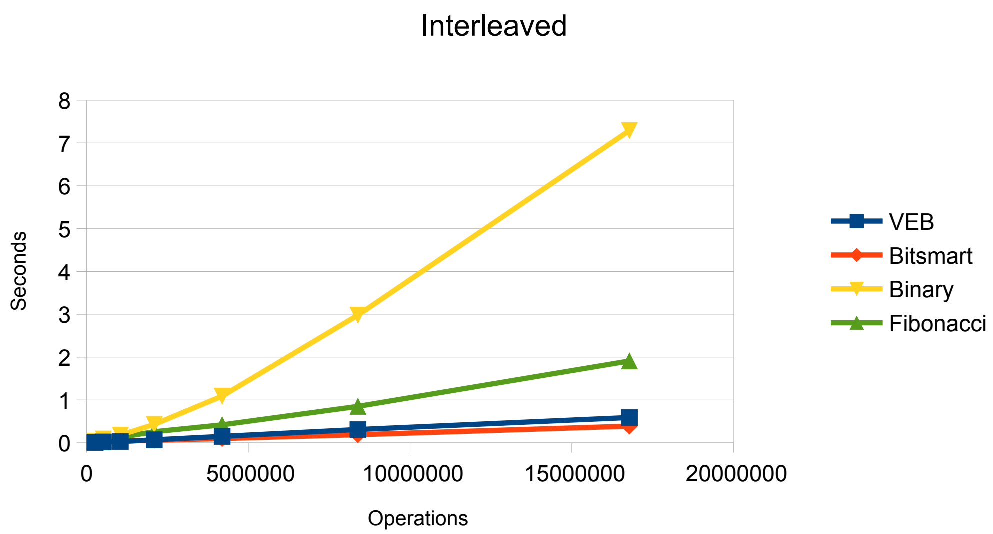
In terms of memory, we known that the Van Emde Boas Trees will use a large constant amount of memory, though we expect the Van Emde Boas tree with bit magic will use a smaller amount of memory than the other heaps as the universe grows full, due to not actually storing the entire value of the elements.

4.3 Data

4.3.1 Running Times

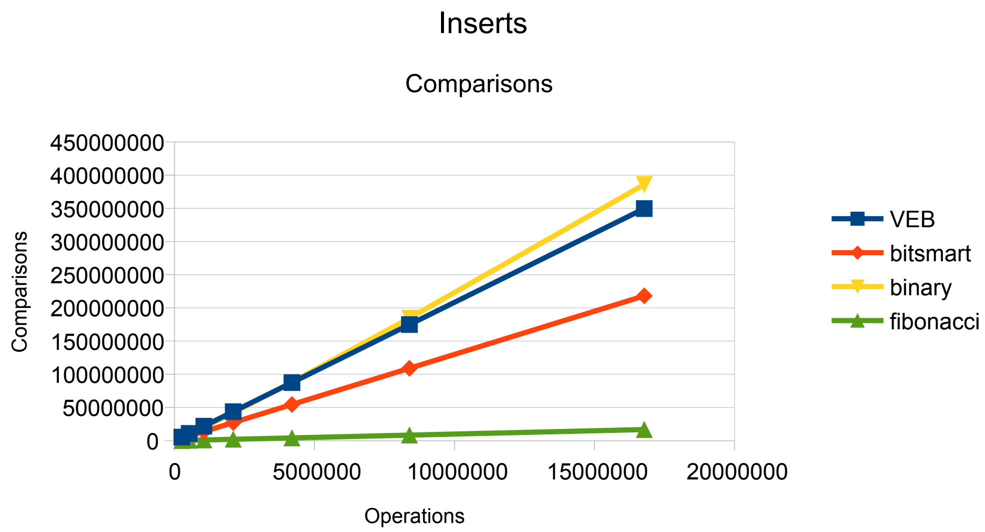


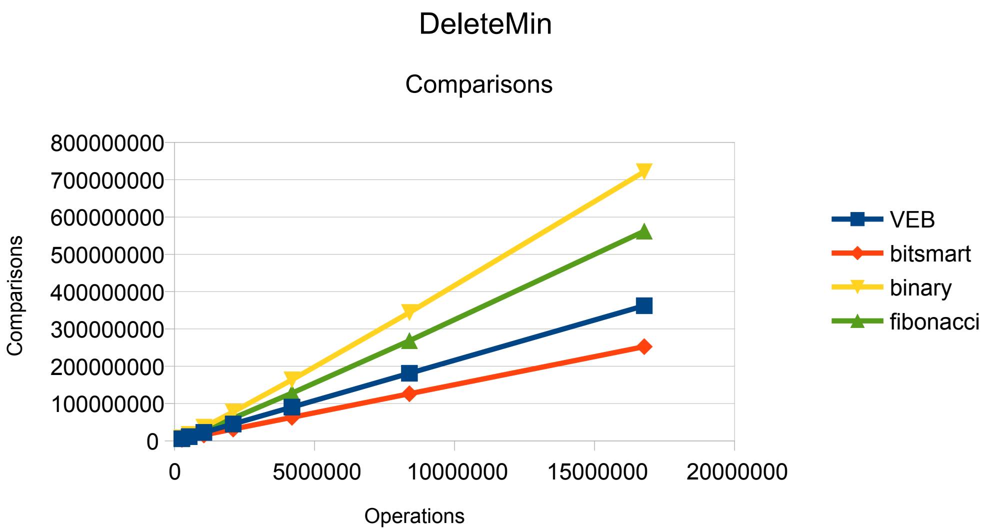


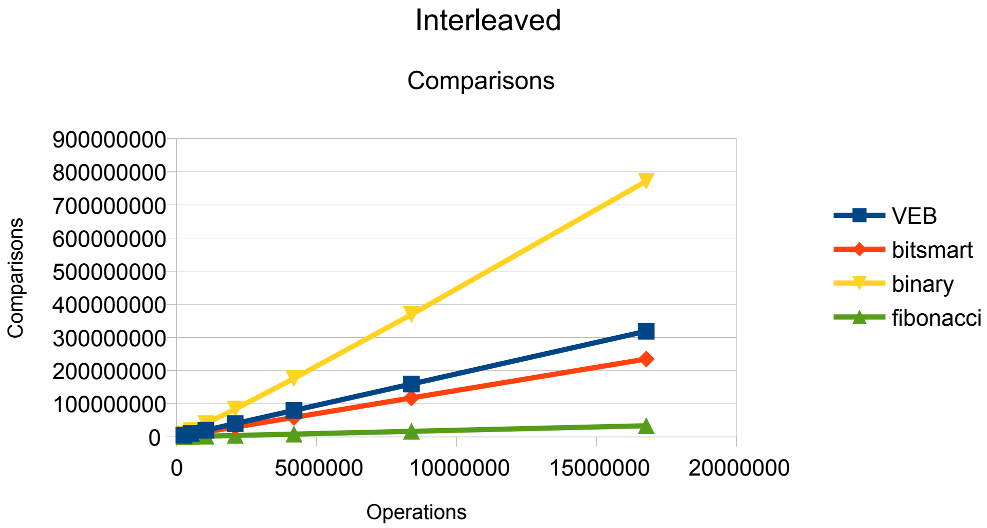


4.3.2 Comparisons

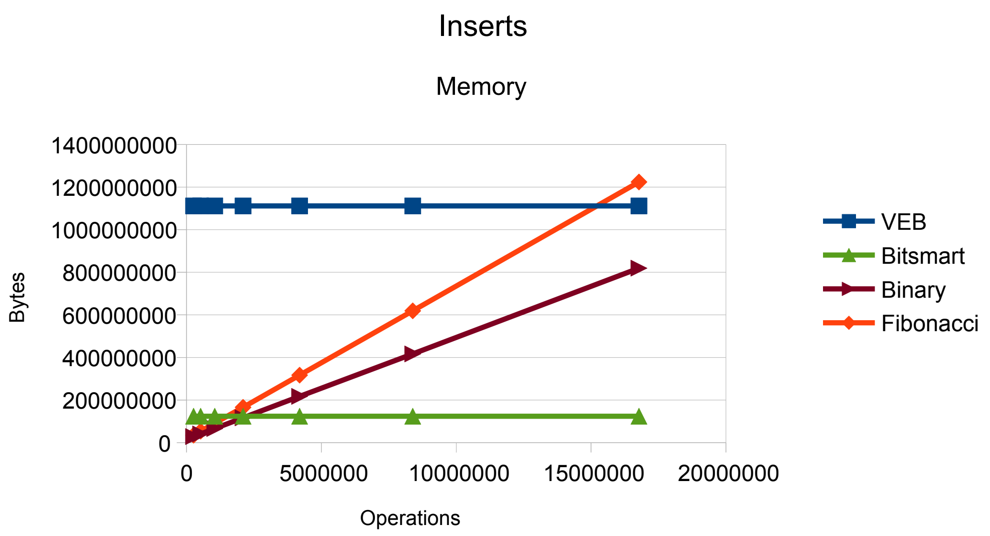
We have been slightly more aggressive in counting comparisons in our Van Emde Boas tree implementation, so we might be under counting the comparisons used in the binary and fibonacci heaps compared to the Van Emde Boas heaps.

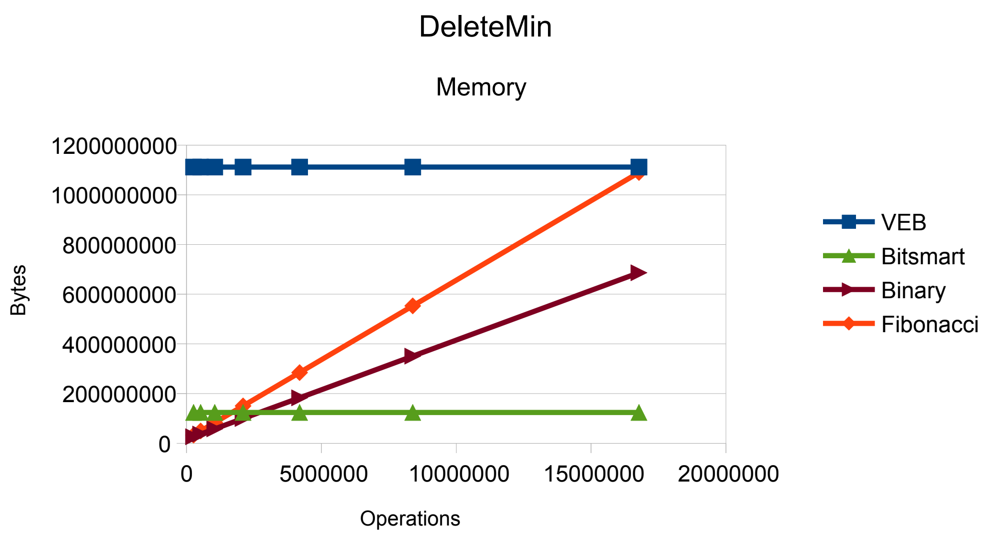

Inserts
Comparisons
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
Comparisons
0
5000000
10000000
15000000
20000000
Operations
VEB
bitsmart
binary
fibonacci

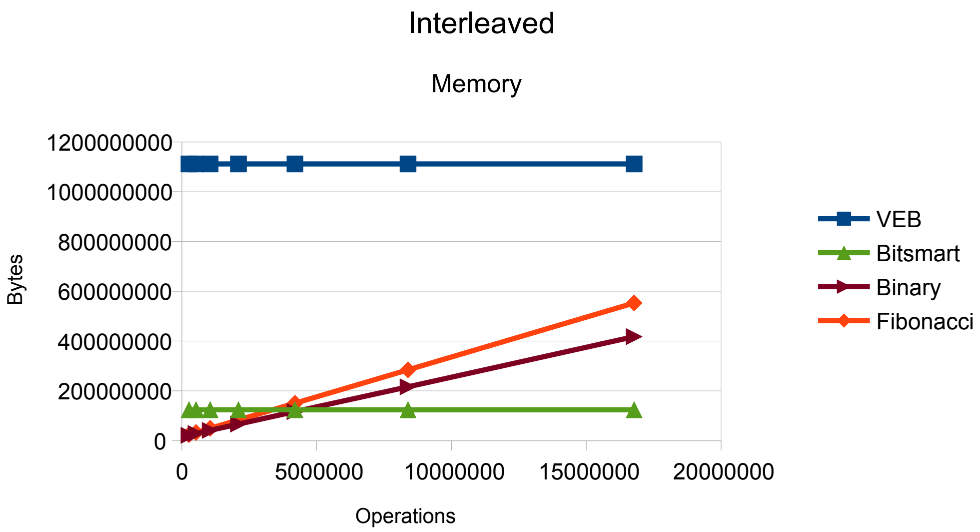

DeleteMin
Comparisons
800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
Comparisons
0
5000000
10000000
15000000
20000000
Operations
VEB
bitsmart
binary
fibonacci


Interleaved
Comparisons
900000000
800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
Comparisons
0
5000000
10000000
15000000
20000000
Operations
VEB
bitsmart
binary
fibonacci

4.3.3 Memory


Inserts
Memory
1400000000
1200000000
1000000000
800000000
600000000
400000000
200000000
0
Bytes
0
5000000
10000000
15000000
20000000
Operations
VEB
Bitsmart
Binary
Fibonacci


DeleteMin
Memory
1200000000
1000000000
800000000
600000000
400000000
200000000
0
Bytes
0
5000000
10000000
15000000
20000000
Operations
VEB
Bitsmart
Binary
Fibonacci



4.4 Results

We see that our experiments behave much as we expected. The Van Emde Boas Tree based heap are much faster than the ordinary heaps, and using bit magic gives a small, yet significant speed boost. They even outperform Fibonacci Heaps on inserts, which is quite impressive.

In terms of memory we can see that the ordinary Van Emde Boas Tree uses a huge amount of memory, only being surpassed by the Fibonacci Heap at a full universe. The bit magic Van Emde boas tree on the other hand uses a quite acceptable amount of memory, being roughly equal with the other heaps at about 2 million elements.

5 Search Tree Tests

5.1 Red-Black Tree Implementation

The c++ standard library contains a *set* datatype as `std::set` which is implemented using a red-black tree so to ensure no errors in the implementation and to save some time, we decided to use that for our tests. This will also give the red-black tree a better chance at competing with our Van Emde Boas tree based search tree as we expect the standard library to be well optimized. We wrote a small class implementing Insert, Remove and Predecessor using a `std::set`. We also wrote a small class implementing the same, using either our Van Emde Boas tree or bitsmart Van Emde Boas tree as well as the `min` function to handle the case when there is no predecessor for our query in the interleaved test.

5.2 Experiment Details

Setup was much like in the previous heap tests, though the test themselves differed slightly.

5.2.1 Test Types

Four types of tests was done, where n is the number of operations:

TestInserts Tests the running time of `Insert` operations by measuring the time it takes to insert the n elements 0 to $n-1$.

TestRemove Tests the running time of `Remove` operations by measuring the time it takes to remove the n elemens 0 to $n-1$.

TestPredecessor Tests the running time of `Predecessor` operations by first filling the search tree with $\frac{n}{4}$ random [1], yet unique, elements and then calling the predecessor function on elements 0 to $n-1$.

TestInterleaved Tests a somewhat closer to realistic scenario by first filling the search tree like in the TestPredecessor test and then n times querying for the predecessor of a random element, removing it from the datastructure and inserting it again.

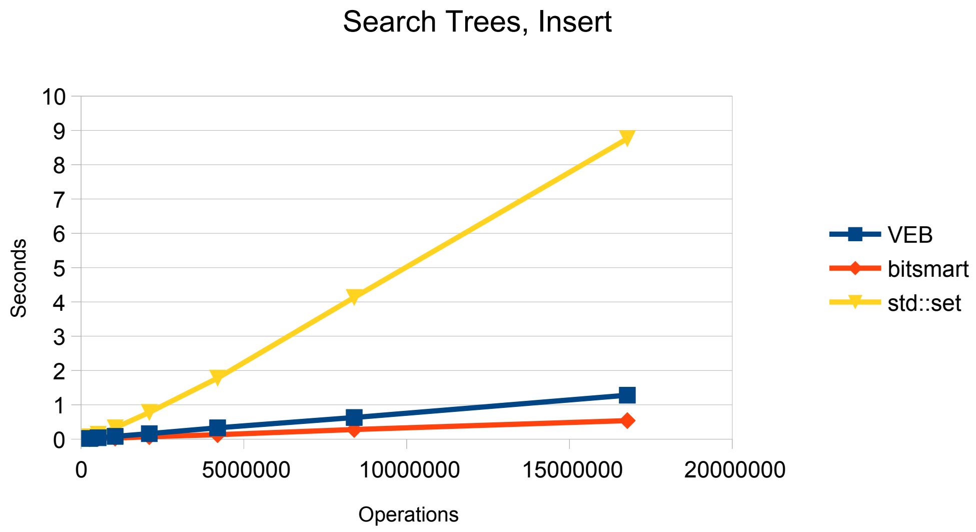
5.3 Expectations

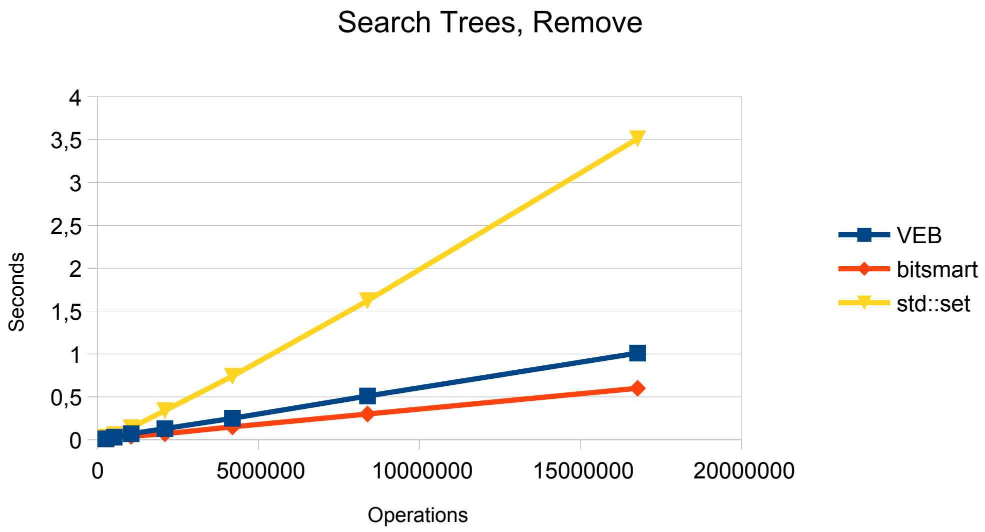
In this second test we expect to see similar trends as in the first, with the Van Emde Boas trees being significantly faster than the Red-Black tree. In terms of memory, we expect the normal Van Emde Boas tree to use a very large chunk of memory, but hope that our bitsmart Van Emde Boas tree will use a constant *small* amount of memory, hopefully less than the Red-Black tree at large data sizes.

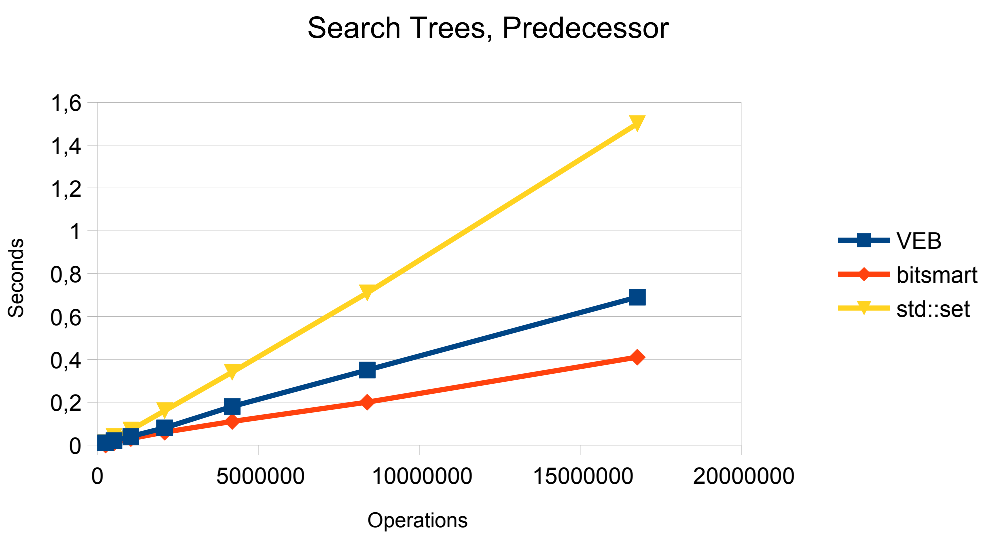
[1] because we use c++'s rand() with default seed, each test runs with the same input elements

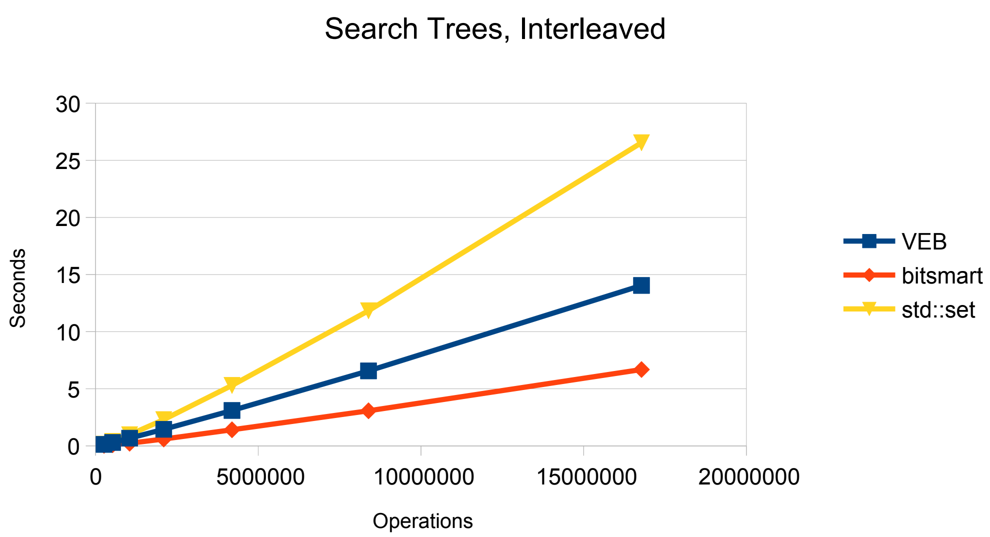
5.4 Data

5.4.1 Running Times

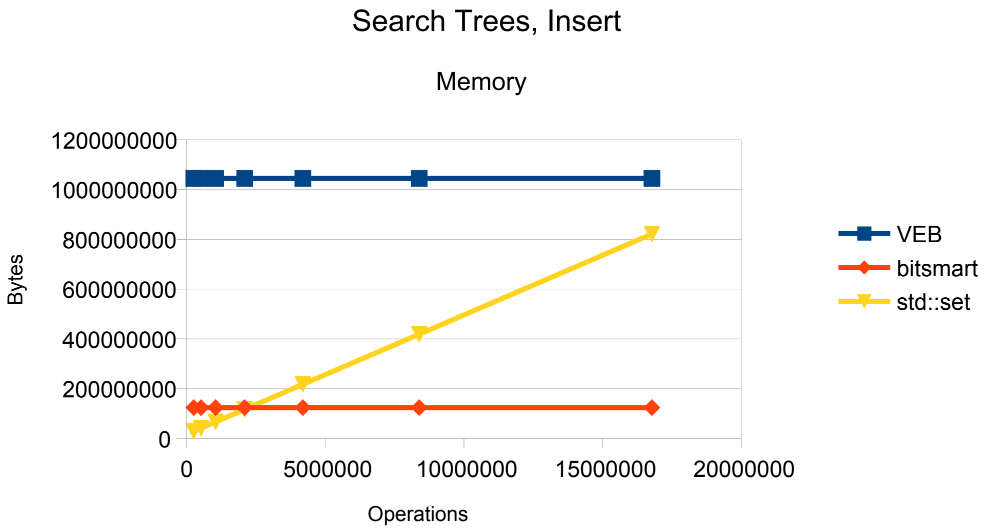





Search Trees, Predecessor
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Seconds
0
5000000
10000000
15000000
20000000
Operations
VEB
bitsmart
std::set


Search Trees, Interleaved
30
25
20
15
10
5
0
Seconds
0
5000000
10000000
15000000
20000000
Operations
VEB
bitsmart
std::set

5.4.2 Memory



5.5 Results

Much as we had expected, we see both our Van Emde Boas trees run faster than `std::set` on all test cases, and again we see the bitsmart Van Emde Boas tree being somewhat faster than its unoptimized variant.

In terms of memory, we see the same trend as in test 1. The Van Emde Boas trees use a constant significant chunk of memory, with the basic Van Emde Boas using about 8 times as much as the optimized version. Again we see the memory efficient variant of the Van Emde Boas tree outperform the reference data structure in terms of memory at about 2 million elements.

6 Conclusion

In this project, we have managed to implement Van Emde Boas trees in a way that turned out to give very fast running times.

We saw a much better running time for heaps and search trees when using the Van Emde Boas tree as a backing data structure, and when using a special bit twiddling optimization, we even managed to bring the memory usage down below below that of standard heap or search tree structures when we insert a significant portion of the universe.

References

[1] Cormen et al. *Introduction To Algorithms.* MIT Press, 2009, Third edition.