A Performance Analysis of Membership Data Structures for Integers in Java*

Marten Voorberg
University of Twente
martenvoorberg@gmail.com

ABSTRACT

We aim to implement a data structure that stores a set of elements in such a way that we can efficiently query whether some element is a member of that set. This research limits itself to sets containing integers in Java. The research examines the setup time, execution time, and memory usage of existing data structures and finds that bitmaps and hash tables offer the best performance. Additionally, we show that altering van Emde Boas trees by replacing the lowest layers with a bitmap improves their performance in the context of this paper. Furthermore, we propose a new data structure: the hash table bitmap. It combines the efficient memory usage of the hash table with the fast experiment time of the bitmap.

Keywords

Membership, Data Structure, Java, Performance Evaluation

1. INTRODUCTION

Determining whether some element is a member of some set is a common problem in computer science and as such, data structures that facilitate efficient membership queries have been around for many decades [10, 13, 32, 33]. The problem involves a subset S of the universal set U — this subset is stored in the membership data structure and allows the user to query whether an arbitrary element of U is also contained in the subset S. The formal definition of the problem can be found in Definition 1.1 [7].

Definition 1.1 (Membership Problem). *A universe $U = \{0, 1, 2, \ldots, u-1\} \subset \mathbb{N}$ and a set $S \subseteq U$ where $|S| = n$ are given; we want a data structure that can determine efficiently for an arbitrary $x \in U$ whether x is contained in S. We assume that $u \leq 2^b$, in which b is the CPU word size, permitting the usage of bit operations. In the **static membership problem**, the set S does not change. In the **dynamic membership problem**, we also require efficient insertion and deletion into and out of S.*

As stated above, many researchers have suggested data structures that can solve the membership problem in $O(1)$ time and $O(n)$ space [8, 32, 33]. These complexities are theoretical and therefore the size of constant factors is ignored. However, these constant factors can greatly influence real-world performance [18]. This research will therefore examine the performance of various membership data structures on real-world benchmarks. We will examine data structures such as hash tables, bitmaps, van Emde Boas trees, binary tries, and RedBlackTrees. Additionally the paper will propose two new data structure: The hash table bitmap and the van Emde Boas bitmap.

Related work on performance evaluation [14], statistics [17, 19, 20, 24] data structure implementation and optimization [1, 3, 4, 4, 6, 8–10, 12, 15, 18, 21, 22, 25, 33] and membership problem solution attempts [7, 9, 11, 13, 16, 23, 26, 32] is referenced throughout the paper whenever appropriate. Section 2 gives a detailed description of the *benchmarks* used to evaluate the performance of various data structures. Sections from 3 onwards, each contain *experiments* with a different focus. Section 3 aims to determine which kind, or family, of data structures is best suited to the membership problem. This is followed by Section 4 on a new data structure that combines the hash table and the bitmap. The following section investigates possible improvements to the vEB-tree through bitmaps. We conclude with a closer look at hash tables — more specifically, we look at perfect hash tables and different methods for collision resolution.

2. BENCHMARK METHODOLOGY

At the heart of our experiment lie benchmarks. A benchmarks evaluates some aspect of performance of a specific data structure (i.e. run-time or memory usage) under reproducible circumstances. To compare multiple data structures against each other, we will run the exact same benchmark for those data structures and analyze the results.

2.1 Layout of a Benchmark

Each benchmark consists of two phases: *The setup phase*, during which the data structure is filled with a generated set of integers, called the *setup set*. This is followed by the *experiment phase*. During this phase, we test the performance of a data structure on the static or dynamic membership problem. In the static case, we perform 2^{20} `isMember` queries. If we are instead interested in the performance for solving the dynamic problem, we perform $\lfloor 2^{20}/3 \rfloor$ `insert`, `isMember` and `delete` queries — these different kinds of queries are randomly interleaved. We perform these queries with the numbers contained in the so called *experiment set*. The choice of 2^{20} as a large number of experiments is not entirely arbitrary — through trial and error, we found this number as an amount that does expose the differences in performances between the

*The source code of the data structures and performance evaluation framework can be found on `https://github.com/marten-voorberg/membership-datastructures`.

35^{th} *Twente Student Conference on IT* 2 July 2021, Enschede, The Netherlands.

data structures but is not of such a size that running experiments becomes unpractical due to large run-times.

During a benchmark, we measure the run-time of both phases, respectively called the *setup time* and *experiment time*. Additionally, we measure the memory usage after the setup phase, as it does not change in the static problem, and is roughly constant in the dynamic problem as the number of insertions is the same as the number of deletions.

2.2 Generation of Setup and Experiment Sets

Each benchmark is based upon two pre-generated sets of integers. The first set, called the setup set, contains the integers used to fill the data structure during the setup phase. The second set, called the experiment set, contains tuples consisting of the integer to be queried and the type of query (i.e. `isMember`, `insert`, and `delete`). These two sets are generated independently in Python using the `scipy` [3]'package — some numbers in the setup set will by chance be in the experiment set and some won't. Data sets differ from each other in the number of elements inserted during the setup phase (n), whether we only perform membership queries or insertion and deletion queries too, and how the elements in the data set are distributed — the data can either be normally (according to some μ and σ) or uniformly distributed. These distributions were chosen as we expect most real-world sets to be distributed either uniformly or normally. For instance, a set of random user IDs is most likely uniformly distributed whilst a set of dates will oftentimes be clustered around some mean.

2.3 Performance Measurements

2.3.1 Setup and Experiment Time

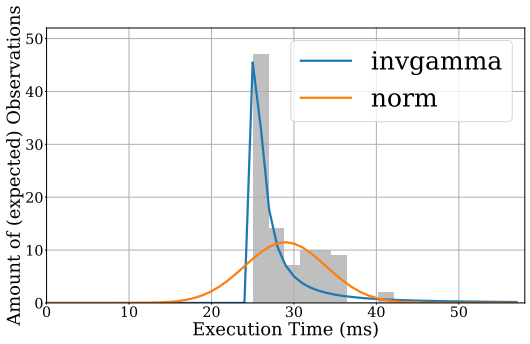
We measure the *execution and setup time* of a benchmark using the `System.currentTimeMillis` method in Java [5]. It is essential to consider the measured execution time is not only dependent on the used data structure but is influenced by a range of non-deterministic factors — namely, the scheduling of that specific thread by the operating system, garbage collection (GC), and Just In Time (JIT) compilation [14].

The first thing we do to reduce this non-determinism is to run benchmarks many times. The idea is that the non-determinism factors will even out over the course of a lot of runs. To reduce the effect of JIT compilation on our benchmarks, we call each method we use inside the benchmark 1000 times without measuring the performance—the JIT compilation then occurs during these runs and not during the benchmark. GC also has an effect on the observed times, but we do not alter our experiments to mitigate this effect. In this experiment, we do not want to completely remove all influences of GC — after all, how a data structure performs in relation to GC is an important part of performance. This does leave us with GC related to the operations surrounding the experiment (i.e. writing and reading from files). It is impossible to prevent such garbage collection from occurring. Instead, we simply run our experiments so many times that each evaluated data structure is equally affected by this 'external' GC.

2.3.2 Memory Usage

The measurement of the amount of *memory used* by data structures is done using the `jcmd` utility [2] — the utility allows you to get a per class breakdown of the memory usage of your Java program. Running this utility has a notable impact on performance and as such a benchmark either measures memory usage or setup and execution time, not both at the same time.

Figure 1. Histogram of observed experiment times for a division based hash table (n=2^{19}), normally distributed.



For many data structures, the `jcmd` utility provides you with a very accurate memory usage estimate. For data structures that rely on classes defined in the Java standard library, the data structure class itself only stores a pointer to the standard library class instance. The actual memory usage is attributed to this standard library class. This complicates matters as this class may be used by other classes running in the background which we are not trying to benchmark. An example of this is the bitmap – the class itself stores just a pointer to an `int` array, which contains 2^{32} bits. Since the bitmap is not the only data structure using `int` arrays, the measurement provided by the utility is skewed. In these situations, we manually compute the memory usage of the data structure. For instance, we know that a bitmap storing a universe of size u requires $\frac{u}{8}$ bytes.

2.4 Statistical Analysis

The experiments conducted in this research usually compare two to six data structures for ten or more values of n. Traditional, hypothesis-based statistical tests would be impractical as the amount of time needed to conduct and evaluate the tests would be disproportionate to the time spent on the remainder of the research. Instead, we compute 95%-confidence intervals (CI) for each data structure at each value of n. A CI provides an upper and lower bound — for a 95% CI, we can state with 95% confidence that the actual variable we measure lies between the upper and lower bound [17, p.444]. In the Figures in the remaining sections, shaded areas represent CIs.

To compute a CI, we need to know the distribution of our data. Visual analysis of a histogram suggests our data did not adhere to a normal distribution and a Shapiro Wilk test confirms this hypothesis [24]. The histogram seen in Figure 1 clearly shows an asymmetric distribution with a high peak — it has very few values to the left of it, but quite some to the right of it. This shape makes the inverse gamma distribution a likely fit [19]. We then fit a distribution to our data using `scipy`'s fit function and conduct a KS test [20] — it showed that most experimental results are distributed according to an inverse gamma distribution. Unfortunately, the used fit function sometimes produces poor, statistically insignificant fits, and in those cases, the computed CI is needlessly large but still technically accurate. This does make some of the results slightly cluttered, but we can still draw conclusions from them.

3. PERFORMANCE OF DATA STRUCTURE FAMILIES

The first experiment aims to determine which family of data structures offers worthwhile performance for the dynamic and static membership problem. The data structures that perform well in this experiment will be examined more closely in the next sections.

3.1 Choice of Data Structures

The data structures we will evaluate are listed below,. Each data structure reprsents a 'data structure family'. The RedBlackTree represents the family of balanced search trees; the vEB-tree was chosen as a representative from the $O(\log \log u)$ query time data structures such as y-fast tries [32]; the hash table family is represented by a hash table that resolves collisions through chaining and uses a division based hash function. These act as probes: if a representative from a family does not perform well, it is highly unlikely for another data structure from the same family to perform significantly better.

1. RedBlackTree [6, 15] [8, p.308]. Binary search trees (BST) provide $O(n)$ space complexity and $O(h)$ time complexity, where h is the height of that tree. The RedBlackTree is one of many balanced BSTs. These *balance* the tree in such a way that the height h is limited to the order of $\log(n)$. Balanced BSTs therefore achieve $O(\log n)$ query time complexity and $O(n)$ space complexity.
2. BinaryTrie [12]. A binary trie is a type of tree. Each branch decision is made based upon a bit of the element. Since each element consists of b bits and by definition $2^b = u$, the binary trie achieves $O(\log u)$ query time.
3. Van Emde Boas Trees [8, 10] (vEB-trees) belong to the class of data structures that offer $O(\log \log u)$ query time. This query time is achieved by recursively splitting the value into chunks of $O(\sqrt{u})$.
4. Bitmap. Each element $x \in U$ belongs to a specific bit in the bitmap. This bit is a 1 if $x \in S$ and 0 otherwise. Through clever use of bit masking and shifting operations, you can perform queries in $O(1)$ time, but at the cost of $O(u)$ space complexity.
5. Division based, chained Hash Table [8, p.253]. Hash tables allow queries in $O(1)$ average time whilst only using $O(n)$ space complexity. Many variants of hash tables exist — we will firstly examine a division based hash table which uses chaining to handle collisions. We will give hash tables a size of $8n$ unless explicitly mentioned otherwise. This size is a trade-off between the likelihood of collisions and the memory usage and is more closely examined in §6.2.

3.2 Experimental Design

We aim to determine the performance of the selected data structures for solving the dynamic and static membership problem and how this performance scales as the size of the set S increases. n will take on the values $2^{10}, 2^{11}, \ldots, 2^{20}$. For each of those values, we generate four data sets (static or dynamic; normally or uniformly distributed values) and conduct benchmarks as described in Section 2.

3.3 Results, Discussion, and Conclusion

The result of the experiment on uniformly distributed data for the dynamic membership problem can be found in Figure 2, the memory usage experiment can be found in Figure 3, and the other results are very similar [30]. It is immediately obvious that the RedBlackTree and the BinaryTrie have much larger setup and experiment times than the other data structures. The bitmap and hash table perform best, both in terms of setup and experiment time. The performance of the vEB-tree is close to that of the bitmap and hash table, but slightly worse. Memory usage is a different story: here the bitmap has extremely high memory usage; the vEB-tree and binary trie are slightly better, but still rather poor; the best memory usage is observed for the RedBlackTree and hash table.

The poor performance of the RedBlackTree in terms of setup and experiment time is because for large values of n, balanced BSTs that scale with $O(\log n)$ can simply not compete with the $O(1)$ scaling of bitmaps and hash tables. The RedBlackTree does have very good memory usage because exactly one node is needed for each element. The fact that the binary trie has to branch b times to determine membership causes it to be outperformed massively in terms of experiment time by the bitmaps and hash tables as they have to do much fewer instructions. The memory usage of the binary trie is also poor. This is because storing a single element, requires a node for each bit in that element. This may result in the adding of 32 nodes when one element is added. The experiments showed that the hash table and bitmap perform very well in terms of experiment and setup time. This was expected due to their $O(1)$ time complexities. Another experiment which examined values of n larger than 2^{20}, found the bitmap had an edge in terms of experiment and setup time. On the other hand, the hash table has much better memory usage than the bitmap — this is because the bitmap always has to store one bit for each element in U, whilst the memory usage of the hash table scales linearly with n.

4. HASH TABLE BITMAP

In Section 3, we concluded that the bitmap and the hash table outperformed all other membership data structures, both in the dynamic and the static case, across all distributions. The bitmap is the fastest of the two, but this comes at a hefty trade-off as its memory usage is extremely poor — the bitmap is worthwhile only when the set S is very dense. The idea of the Hash-Table-Bitmap (HTB), a simplified version of the data structure proposed by Brodnik et al. [7][1], is to partition the set S and store these partitions as bitmaps when they are very dense and otherwise in a hash table.

4.1 Semi-Static Membership

We will first discuss the HTB in the context of the semi-static membership problem — it is possible to perform `insert` and `delete` operations, but the HTB will not alter its structure after initialization. This means that if some part of the universe is very dense at initialization and stored as a bitmap, and this entire partition is subsequently deleted, it will still be stored as a bitmap despite being very memory inefficient. A HTB is defined in terms of some *partition-value* π. Based upon π the universe U is uniformly divided into 2^π partitions of size $\frac{u}{2^\pi}$, henceforth denoted as Π. We denote the lower and upper bound of a partition denoted by l_p and u_p, respectively. We then compute the sparseness r_p for each partition p using Equation 1. We store a partition p as a bitmap if r_p is larger

[1]The data structure described by Brodnik et al. [7] is impractical to implement as it relies on using less than b (CPU word size) per elements in a hash table. This technically possible using bit packing, but it would be very difficult to implement and it would likely harm performance.

Figure 2. Execution time of various membership data structures performing 2^{20} queries as n grows.

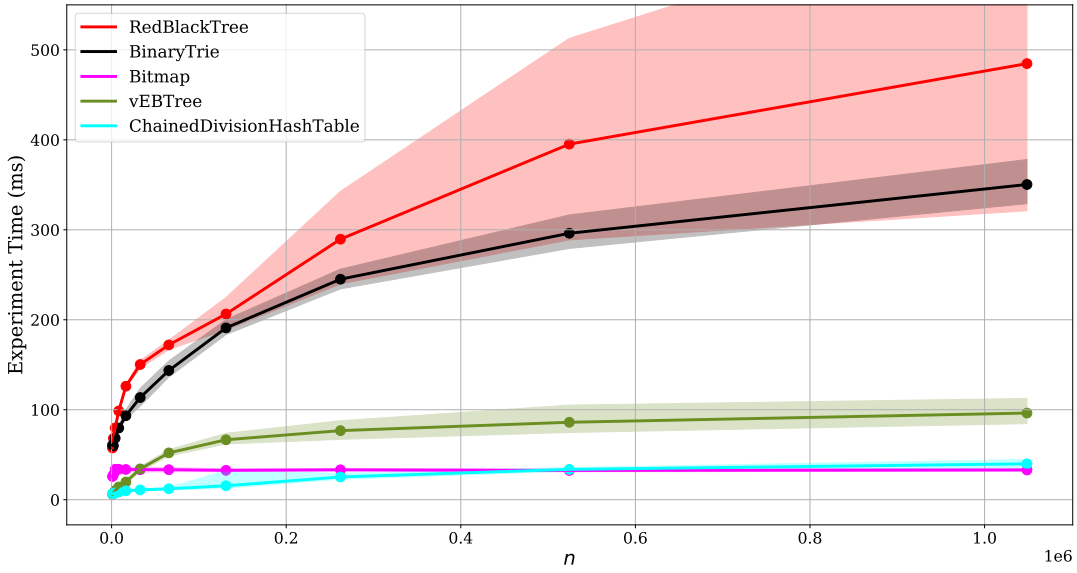
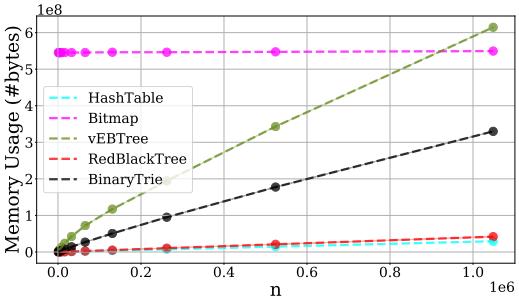


Figure 3. Memory usage of various membership data structures performing 2^{20} queries as n grows.



than some threshold value r^* — otherwise, we store it in the hash table.

$$r_p = \frac{|\{x \in S : l_p < x \leq u_p\}|}{\Pi} \tag{1}$$

The HTB is designed upon the principle that storing an element in the bitmap will perform better than storing that element in a hash table. With this idea in mind, we will pick π such that the total amount of elements to be stored inside bitmaps is maximized. Equation 2 shows that the maximum value π can take on is 5. This is because there is an overhead attached to each partition — by limiting π, we limit the number of partitions and subsequently the overhead.

$$\pi = \underset{\pi' \in \{2,3,4,5\}}{\arg\max} \sum_{p=0}^{2^{\pi'}-1} f(p) \tag{2}$$

$$f(p) = \begin{cases} |\{x \in S : l_p < x \leq u_p\}| & \text{if } r_p \geqslant r^* \\ 0 & \text{otherwise} \end{cases}$$

The value of r^* is also significant. If we make this value too small, we risk wasting memory due to storing very sparse partitions as bitmaps. Should we make this value too big, we lose performance as elements are more often stored in a hash table instead of a bitmap. We will firstly determine $\hat{r}$: the value at which the memory usage is the same for the bitmap and the hash table. A bitmap always stores Π elements, requiring that amount of bits. A hash table storing n_p elements needs to store the actual integer and a pointer to the next bucket, requiring 64 bits. Since $\Pi = 64n_p$, we get $\hat{r} = \frac{1}{64}$. We can conclude that $r_p \leqslant \hat{r}$, otherwise we would be wasting memory and have inferior query time compared to a hash table — the worst of both worlds. We could choose to make r_p even smaller than $\hat{r}$. In this case, our memory usage will see a minor increase compared to the hash table, but we will likely see better query times.

Algorithm 1 shows the algorithm for performing a query on the HTB data structure. As the used functions all have an average or worst-case time complexity of $O(1)$, it is self-evident that queries on the HTB have an average time complexity of $O(1)$ too.

Algorithm 1: HTB Query — Perform an arbitrary query (i.e. `isMember`, `insert`, or `delete`) on the HTB.

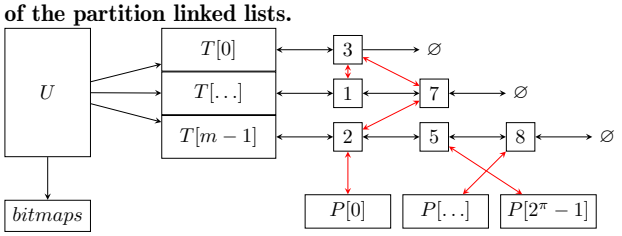
$p \leftarrow partition(x)$
if $bitmaps[p] \neq NIL$ **then**
  $x \leftarrow x - lowerBound(p)$
  return $bitmapQuery(bitmaps[p], x)$
else
  return $hashTableQuery(x)$
end

4.2 Dynamic Membership

To adapt the HTB to the dynamic membership problem, we have two additional requirements. (1) If we insert an element into the HTB and the size of the appropriate partition increases beyond some threshold, we want to convert that partition into a bitmap. (2) We also want the reverse: If we delete an element from a specific partition stored as a bitmap and the size of that partition decreases below some threshold, we want to remove the bitmap and insert its members into the hash table. Performing a membership query is identical for the dynamic and semi-static versions of the HTB.

To facilitate (1), we store a doubly-linked list for each partition (See Figure 4 for a schematic overview). This list contains all elements stored in the hash table belonging to a particular partition. We use a doubly-linked list here because it allows for deletion in $O(1)$ time provided we have access to the node we are deleting. If the threshold is exceeded, we iterate over the elements of the linked list for that partition, remove the elements from the hash table, and put them into the linked list instead (see Algorithm 2). Analysis of this algorithm proves an amortized time complexity of $O(1)$ [27].

Figure 4. A schematic overview of a dynamic HTB. The bi-directional black and red edges represent the hash table linked lists and partition linked lists respectively. T is an array storing the heads of the linked list used for collision resolution through chaining. P is an array storing the heads of the partition linked lists.



When, following many deletions, there are too few elements in the bitmap, its memory usage is too large to be worth the slight increase in query time. Therefore, we want to take all elements remaining in the bitmap and insert them into the hash table. To that end, we iterate over all bits in the bitmap, recompute its value, and insert it into the hash table. Since the bitmap contains Π bits, this deletion operation has a worst-case complexity of $O(\Pi)$.

Algorithm 2: Insertion into a dynamic HTB

```
p ← partition(x)
increment partitionCount[p]
if bitmaps[p] ≠ NIL and
 partitionCount[p] ≥ insertThreshold then
    bitmaps[p] ← new bitmap
    node ← partitionListHeads[p]
    while node ≠ NIL do
        bitmapInsert(bitmaps[p], node.value)
        node ← node.partitionNext
        /* Delete from bucket linked list      */
        node.buckNext.buckPrev ← node.buckPrev
        node.buckPrev.buckNext ← node.buckNext
    end
    bitmapInsert(bitmaps[p], x)
else if bitmaps[p] ≠ NIL then
    bitmapInsert(bitmaps[p], x)
else
    hashTableInsert(x)
end
```

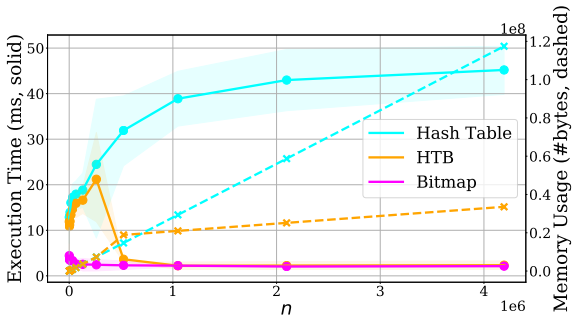
4.3 Experimental Design

To evaluate the (dynamic) HTB we use the benchmark methodology laid out in Section 2. In the experiments, we generate all data according to a normal distributed with $\mu = 2^{20}$ and $\sigma = 2^{18}$. We do not conduct an experiment on uniformly distributed data as there would be no dense partitions and the (dynamic) HTB would perform exactly like a hash table.

4.3.1 Static Performance

We compare the HTB to the bitmap and hash table on the static membership problem. In our experiment, n takes on the values 2^{10}, 2^{11}, $\cdots$, 2^{22}. Since we are only interested in the static membership problem, we do not consider the setup time, but instead only focus on the execution time and the memory usage.

Figure 5. Memory usage and execution time of the HTB, bitmap and hash table on the static membership problem. For the sake of clarity the memory usage of the bitmap is excluded—memory usage is approximately 500 million bytes for any n.



4.3.2 Dynamic Performance

In order to evaluate the performance of the Dynamic Hash Table Bitmap (DHTB), we perform two experiments. In the first, we start with an empty data structure and fill it up to n. We do this for the usual values of n from 2^{10} to 2^{20}. In the last experiment, we first fill it up to n and subsequently delete those n elements, and we only measure the time taken for deletion. We compare the performance of the DHTB, the bitmap, and the division based, linked list chained, hash table.

4.4 Results, Discussion, and Conclusion

4.4.1 Static Performance

Figure 5 shows the result of the static performance experiment. We can see that the performance of the HTB is solid compared to the hash table. Its memory usage and query times are the same as the hash table for smaller n. Once n grows larger, the memory usage of the HTB becomes much better. The query time is also becomes much better for larger values of n. This happens because the elements in certain partitions start to get stored inside a bitmap. In short, on normally distributed data, for large values of n, the HTB provides query times just as fast as the bitmap but uses even less memory than the hash table to achieve this. For smaller values of n, the performance is akin to that of the hash table.

4.4.2 Dynamic Performance

Figure 6. Performance of DHTB compared to the bitmap and hash table on normally distributed data.

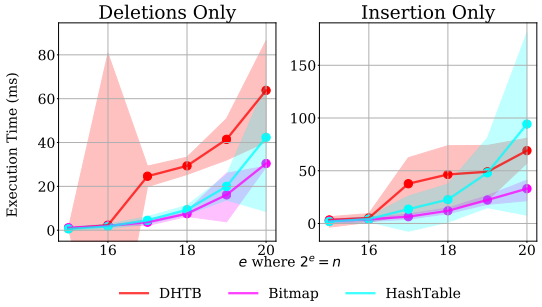


Figure 5 shows the results of the experiment in which only insertions. The results show that for e smaller than 16, where $n = 2^e$, all data structures perform roughly the same [28]. At $e = 16$, the DHTB converts a partition to a bitmap, and at that point, the HTBs performance is the worst out of the three. This is the case because we perform an expensive operation to optimize future insertions, but there are few future insertions. As e increases beyond 16, we can see that the DHTB starts to outperform the hash table. This occurs as there are many insertion operations into a bitmap instead of into a hash table, causing an increase in performance. The results of the deletion experiment show that for values of e smaller than 16, all data structures again perform identically [28]. We clearly see that the DHTB performs the worst out of the three as e grows beyond 16. This is because, at this point, a bitmap must be converted to a hash table, and the massive cost ($O(\Pi)$) causes it to perform worse than both the hash table and the bitmap.

5. VAN EMDE BOAS TREE VARIANTS

In Section 3 we concluded that vEB-trees outperform balanced search tries and binary tries but hash tables and bitmaps, in turn, outperform vEB-trees. This section will investigate alterations to the traditional vEB-tree and see whether these improve query time or memory usage for solving the membership problem.

5.1 van Emde Boas Bitmap

A vEB-tree V contains an attribute u storing the universe size of the tree, *min* and *max* storing the smallest and largest number stored in the tree, and *children* which is an array of vEB-trees. In order to determine the membership of x in a traditional vEB-tree, x is compared against the explicitly stored minimum and maximum, and if unequal, the correct child is recursively queried [8, p.550]. The recursion ends either when explicitly stored minimum or maximum value matches x or when the universe size of the queried vEB-tree is equal to 2. A traditional vEB-tree that stores a universe of 16 elements has up to 12 descendants and, in the worst case, needs to perform two recursive calls to determine the membership of x.

We propose a variant to the vEB-tree: The van Emde Boas Bitmap (vEBB) alters the traditional bitmap — it does not store the *min* or *max*, but instead stores an integer named *bitmap*. A vEBB where $u = 16$ requires no further recursive calls to perform membership queries and has no descendants. Algorithm 3 shows the `isMember` algorithm — insertion and deletion are defined analogously, with different bit operations. We expect that the elimination of the descendants will improve the memory usage of the vEBB, and the elimination of the recursive calls will improve the query time a little. The vEBB does not improve upon the theoretical space and time complexities — it merely aims to improve the constant factors.

Algorithm 3: vEBB membership query.

Input: vEBB V, integer x
if $v.u = 16$ **then**
  return $(V.bitmap \wedge (1 \ll x)) \neq 0$
else
  return $isMember(V.children[high(x)], low(x))$
end

5.2 Experimental Design

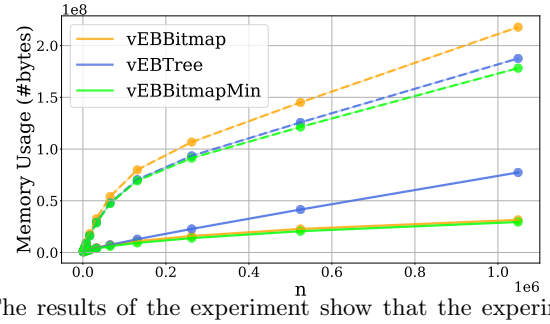
This section outlines the experimental design used to compare execution time, setup time, and memory usage, of vEBB to those of the traditional vEB-tree. To evaluate the memory usage, we consider the amount of memory stored for various values of n ranging from 2^{10} up to and including 2^{20}, both when this data is normally and uniformly distributed. We conduct these experiments according to the methodology outlined in Section 2.

5.3 Results, Discussion, and Conclusion

Figure 7 shows the memory usage of the vEBB, the vEB-tree, and the vEBB with a cached minimum for uniformly and normally distributed data. Interestingly, the vEBB uses more memory than the vEB-tree for uniformly distributed data. We hypothesized this is due to min-caching in the vEB-tree — the minimum is explicitly stored and not inserted fully into the appropriate subtree. When the data is uniformly distributed, many branches in the vEB-tree only contain one element. In that situation, the vEB-tree can store that element explicitly as the minimum, but the vEBB has to generate the entire branch and store the element in the bitmap. Figure 7 supports this hypothesis — by adding an explicit minimum to the vEBB, its memory becomes better than that of the vEB-tree.

On normally distributed data, we can see the vEBB using much less memory than the vEB-tree. This reduction occurs as most elements belong to just a few branches if the tree is storing normally distributed data. Therefore min-caching does not cause a significant reduction in memory usage. The elimination of the last two layers of the vEB-tree through a bitmap does improve memory usage here.

Figure 7. Memory usage of various vEB-tree variants as n increases. The dashed and solid line show uniformly and normally distributed data respectively.



The results of the experiment show that the experiment time of the vEBB and the vEB-tree is very similar for smaller values of n [29]. As n increases, we can see that the vEBB outperforms the traditional vEB-tree, both on uniformly but especially on normally distributed data, for solving the static and dynamic membership problem. The first reason is that the vEBB has to do fewer recursive calls due to replacing the last two layers with a bitmap. Additionally, we likely see the vEBB perform better for the dynamic problem due to the much-simplified insertion and deletion algorithms.

Based upon these experiments, we can conclude that the vEBB outperforms the traditional vEB-tree for solving both the dynamic and the static membership problem. It provides better query time for both the dynamic and the static problem, both for uniformly and normally distributed data. Its memory usage is much better than that of the vEB-tree if the elements are normally distributed. However, if the elements are uniformly distributed, the vEBB uses slightly more memory than the vEB-tree.

6. IMPROVING HASH TABLES

It became clear in Section 3 that hash tables offer excellent performance for solving the membership problem. Due to this performance and their widespread use, hash tables have been the subject of extensive research. We will examine the performance of perfect hash tables for solving the static membership and compare the performance of various collision handling methods.

6.1 Perfect Hash Tables

Perfect hash tables earn their name because of their stellar space and time complexity, namely $O(n)$ and $O(1)$ worst-case, respectively [8, p.278]. The usage of traditional perfect hash tables is limited to the static dictionary problem, and thus to the static membership problem[2]. The idea behind perfect hashing is to first use some hash function h, but instead of resolving collisions by a traditional method such as a linked list, we use a smaller hash table S_j with its own hash function h_j. However, this in and of itself does not guarantee our stellar $O(n)$ and $O(1)$ complexities. To achieve those, we first pick the first hash function h uniformly at random from a universal hash family $\mathcal{H}$. We then compute the memory usage, and if it is too big, we simply choose another h uniformly at random, until we find one that satisfies our space complexity. We take a similar approach for each smaller S_j: We pick $h_j \in \mathcal{H}$ uniformly at random until we find a h_j that does not have any collisions. All of this may seem like a lot of randomization but if we pick our hash function family well, both theoretical analysis and real-world experiments show this approach to be feasible.

6.1.1 *Experimental Design*

The performance of the perfect hash table (PHT) will be compared against that of Bitmap, ChainedDivisionHashTable, ABHashTable. To get an idea of the performance difference between non-perfect and perfect hashing we include the ABHashTable and the ChainedDivisionHashTable. The bitmap, the data structure with the best experiment times, is included as a baseline.The PHT and ABHashTable will use hash functions from the universal hash family $\mathcal{H}_{pm} = \{h_{ab} | a \in \mathbb{Z}_p, b \in \mathbb{Z}_p\}$ where $h_{ab}(k) = ((ak + b) \mod p) \mod m$, p is prime, and m is the size of the hash table [8, p.269].

We compare the memory usage and experiment time for two data sets of size 2^e where $e \in \{10, 11, \cdots, 20\}$ — one set's values are normally distributed ($\mu = 2^{20}$, $\sigma = 2^{18}$) and the other is uniformly distributed. We conduct these experiments in accordance with the methodology outlined in Section 2.

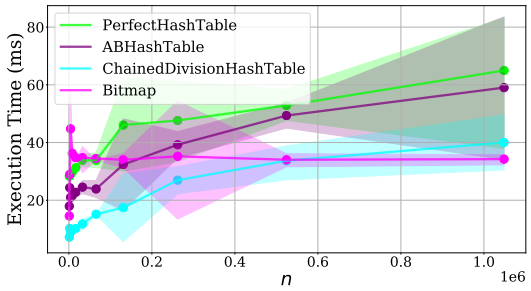
6.1.2 *Results, Discussion, and Conclusion*

In Figure 8 we can see the experiment times for the uniformly distributed data. We can clearly see that the PHT performs equally as well, or worse than the other data structures, despite having no collisions. The same results were seen on normally distributed data [31]. This is probably because the improvements you would expect to see in execution time due to a lack of collisions are balanced by bigger constant factors — namely, when computing the first hashed value, retrieving the second hash function and table, and then computing the second hashed value. Additionally, the PHT has much higher setup times which is obviously due to the randomization in the setup phase [31].

[2]Dietzfelbinger et al. [9] did create a data structure called dynamic perfect hashing that has the same complexities except that the insert and delete complexities are amortized.

The PHT also has higher memory usage for normally and uniformly distributed data [31]. This is due to the need to explicitly store the used hash function h_j for each S_j. We can conclude that PHTs, despite their perfect theoretical complexities, do not perform better, and sometimes perform worse, than standard hash tables in experiment time, setup time, and memory usage.

Figure 8. The performance of the PHT compared to other data structures on uniformly distributed data.



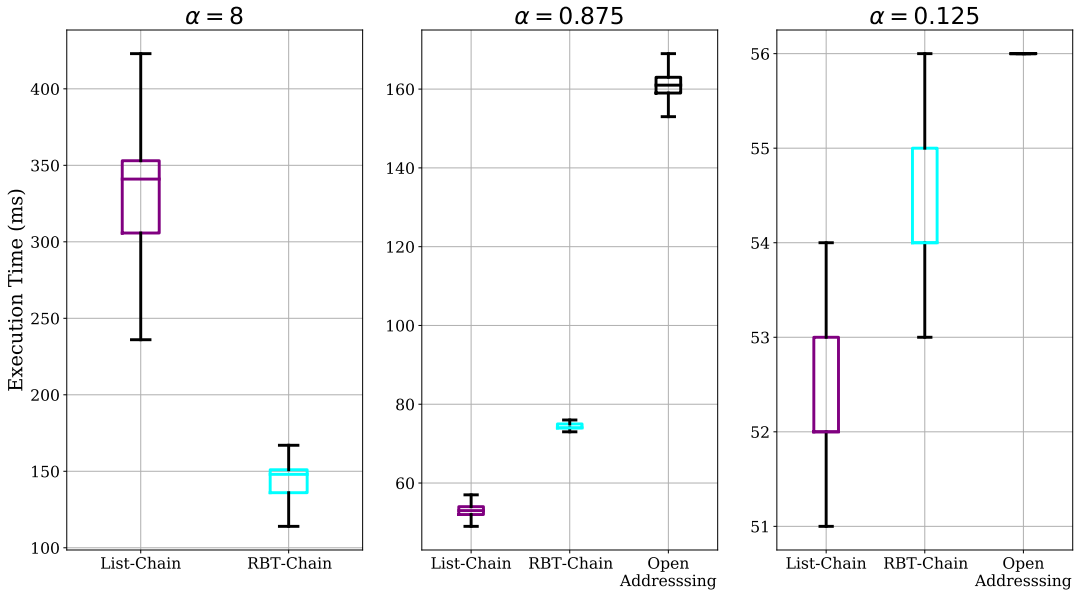
6.2 Improved Collision Resolution

When for two different values x_1 and x_2 in our hash table, we have $h(x_1) = h(x_2)$, we have a collision. For hash tables to be correct, we must resolve this collision. The most basic method of resolving collisions is linked list chaining: We store a linked list for each slot j and we will denote the length of this list as l_j. The performance of this method is good as long as this list is not too long. If due to a poorly chosen hash function or a small table size, this list starts growing, the performance of your hash table will become a lot worse. We could improve this scalability by using some form of a balanced binary search tree, such as a RedBlackTree. A balanced BST would offer a $O(\log l_j)$ query times, and your hash table's performance would scale much better than a linked list in the case that l_j becomes large. A slightly more complicated version of this approach is used in the OpenJDK `HashMap` implementation [22, line 158].

There are also methods of resolving collisions that avoid chaining altogether. In open-addressing, each element in the set is stored directly in the table, instead of in some other data structure [8, p.269]. We extend the hash function to take an additional parameter: the probe number. Insertion, membership checking, and deletion happen by computing $h(k, 0)$, $h(k, 1)$, $\cdots$ until we either find an empty slot or the desired element. We will only consider linear probing, in which $h(k, i) = (h'(k)+i) \mod m$ where m is the size of the hash table [8, p.273]. We limit ourselves to the collision resolution methods outlined above, though it must be stated that there are many methods for collision resolution, making this subsection easily expandable into another paper.

So far, we have not given too much thought to the size of the hash table, m. So far we have usually set it to $m = 8n$ — this has been possible as we always know the value n in our experiments. For reasoning about the size of the table, we usually use the *load factor* $\alpha = n/m$: the average amount of elements stored in a table slot [8, p.258]. Since open-addressing stores each element directly in the table, α can not exceed 1. When using chaining, α can exceed 1 — in such cases, the choice of chaining data structure (in our case, a RedBlackTree or linked list) starts to become a significant part of the performance. We do not consider the possibility of resizing the hash table in this paper.

Figure 9. Box-plots of experiment times, solving 2^{20} membership queries where $n = 2^{20}$, for various collision resolution methods, for various load factors.



6.2.1 *Experimental Design*

We will compare three kinds of collision resolution: open-addressing, chaining with a linked list, and chaining with a RedBlackTree. Each of these hash tables will use the same hash function. We will fix $n = 2^{20}$, examine $\alpha \in \{0.125, 0.875, 8\}$, and perform 2^{20} queries on the static problem. The open addressing collision resolution method does not work for $\alpha > 1$ and is thus excluded from the experiment where $\alpha = 8$.

6.2.2 *Results, Discussion, and Conclusion*

The results of the experiments can be seen in Figure 9. For $\alpha = 8$, there will be chains of considerable size. The hash table that stores chains as a RedBlackTree then greatly outperforms the linked list based chaining due to the improved scaling ($O(\log l_j)$ versus $O(l_j)$). For $\alpha = 0.875$ we can see that the list-based chaining performs best. The list probably performs better than the RBT-chain as l_j is usually rather small and the smaller constant factors of the linked list matter more than the better scalability of the RedBlackTree. Open addressing performs very poorly since 7/8 positions are taken. When using open addressing, all operations end once an empty address is found — it then makes sense that when there are very few empty addresses, these operations take significantly longer. For $\alpha = 0.125$, all three solutions perform similarly. Although the data would suggest that the list-based chaining works best, the observed times are so similar that should not be considered more than a mere suggestion.

Based upon these experiments, we give some guidelines for the choice of collision resolution depending on how accurately α can be estimated. If there is a possibility of α taking on values larger than 1, it is impossible to use open addressing. In such cases, it may be wise to use collision resolution through chaining with a balanced binary search tree, such as the RedBlackTree, due to improved scalability if the chains get very large. If you do not expect values much larger than 1 and you trust your hash function to evenly distribute the elements, it is impossible to go wrong with chaining through a linked list. This approach is very easy to implement and performed well in our experiments.

7. CONCLUSION

In Section 3, we have shown that the data structures whose complexities suggested would perform the best — namely the Bitmap and the HashTable — also performed best in our benchmarks. The bitmap had a slight edge in terms of experiment and setup time, but its memory usage is very poor. The findings of this section, and the work of Brodnik et al. [7], served as inspiration for the creation of a new data structure: The Hash Table Bitmap (HTB) — it is a normal hash table except that really dense partitions, if there are any, are stored in a bitmap. The experiments showed that the hash table bitmap combines the best aspects of the bitmap and the hash table — it performs similarly to the hash table, except on dense, normally distributed data. For such data, the HTB takes on the excellent performance of the bitmap, whilst using less memory than the bitmap or the hash table.

We also found that the performance of vEB-trees can be improved for the membership problem by ending the recursion in a bitmap. The vEBB has better setup and experiment time than the vEB-tree for solving both the dynamic and the static problem, both on normally and uniformly distributed data. Its memory usage on normally distributed data is better than the vEB-tree, though its memory usage for uniformly distributed data is slightly worse.

Lastly, we looked into possible improvements to hash tables — namely perfect hash tables and other methods of collision resolution. Suprisingly, we found that the performance of perfect hash tables was worse across the board than their imperfect counter parts. Additionally we showed that collision resolution by chaining using a linked-list, performed very well for load factors below 1. Once load factors got a lot larger than 1, chaining through a RedBlackTree performed very well compared to linked list based chaining.

ACKNOWLEDGEMENTS

I would like to extend special thanks to my supervisor, Vadim Zaytsev, for his support in this research project. Additionally, my thanks go out to all who have proofread my paper.

REFERENCES

[1] HashSet Implementation in OpenJDK. https://github.com/openjdk/jdk/blob/891f72fe6ea545cdf845d26157a64315a93f9a06/src/java.base/share/classes/java/util/HashSet.java#L90. [Online; accessed 23-april-2021; line 107].

[2] The jcmd utility. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr006.html. Accessed: 2021-05-26.

[3] Scipy Statistical Functions. https://docs.scipy.org/doc/scipy/reference/stats.html. [Online; accessed 15-June-2021].

[4] Set Implementations. hhttps://docs.oracle.com/javase/tutorial/collections/implementations/set.html. [Online; accessed 23-april-2021].

[5] Documentation on System.currentTimeMillis. System (Java Platform SE 7), https://docs.oracle.com/javase/7/docs/api/java/lang/System.html, Jun 2020.

[6] R. Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, dec 1972.

[7] A. Brodnik and J. I. Munro. Membership in Constant Time and Almost-Minimum Space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[9] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. Auf Der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, jul 1994.

[10] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[11] F. Fich and P. B. Miltersen. Tables should be sorted (On random access machines). In *Lecture Notes in Computer Science*, volume 955, pages 482–493. Springer Verlag, 1995.

[12] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[13] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with 0(1) Worst Case Access Time. *Journal of the ACM (JACM)*, 31(3):538–544, jun 1984.

[14] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, oct 2007.

[15] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, volume 1978-October, pages 8–21. IEEE Computer Society, 1978.

[16] J. Iacono and M. Pătraşcu. Using hashing to solve the dictionary problem (In external memory). In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 570–582. Association for Computing Machinery, 2012.

[17] B. Illowsky and S. Dean. *Introductory statistics*. 12th Media Services, 2017.

[18] N. D. Jones. Constant time factors do matter. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of Computing*, pages 602–611, 1993.

[19] A. Llera and C. Beckmann. Estimating an inverse gamma distribution. *arXiv preprint arXiv:1605.01019*, 2016.

[20] F. J. Massey Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[21] S. Nilsson and M. Tikkanen. Implementing a dynamic compressed trie. In *Proceedings of WAE*, pages 1–12.

[22] OpenJDK. Openjdk hashmap source code. https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/HashMap.java.

[23] J. Radhakrishnan, V. Raman, and S. Srinivasa Rao. Explicit deterministic constructions for membership in the bitprobe model. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2161, pages 290–299. Springer Verlag, 2001.

[24] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[25] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM (JACM)*, 32(3):652–686, jul 1985.

[26] R. E. Tarjan and A. C. C. Yao. Storing a Sparse Table. *Communications of the ACM*, 22(11):606–611, nov 1979.

[27] M. Voorberg. Amortized Analysis of Dynamic HTB Insertion. DOI: 10.6084/m9.figshare.14785083.v1, Jun 2021.

[28] M. Voorberg. Experiment time of dynamic hash table bitmap compared to bitmap and hash table. DOI: 10.6084/m9.figshare.14790270.v1, Jun 2021.

[29] M. Voorberg. Experiment time of various vEB-tree variants. DOI: 10.6084/m9.figshare.14790372.v1, Jun 2021.

[30] M. Voorberg. Performance of 5 membership data structures. DOI: 10.6084/m9.figshare.14797656.v2, Jun 2021.

[31] M. Voorberg. Performance of Perfect Hash Tables. DOI: 10.6084/m9.figshare.14779128.v3, Jun 2021.

[32] D. E. Willard. Log-logarithmic Worst-Case Range Queries are Possible in Space Θ(N). *Information Processing Letters*, 17(2):81–84, aug 1983.

[33] A. C.-C. Yao. Should Tables Be Sorted? *Journal of the ACM*, 28(3):615–628, July 1981.