Optimizing Integer Set Operations and Exploring Practical Applications with the van Emde Boas Tree

DEVANSHU PANDYA

This paper examines the van Emde Boas tree (vEB), a high performance data structure that supports fundamental operations in $O(log(log(u)))$ time by using a fixed-universe recursive design. A custom implementation of the vEB tree is developed to investigate its actual performance across a range of input sizes. Empirical results are collected and compared with theoretical bounds to assess the viability of the data structure in practice. Findings demonstrate that while the structure's complexity may hinder widespread use, it remains a powerful tool in scenarios requiring rapid integer-based queries.

Word Count: 4158

INTRODUCTION

The design and analysis of efficient data structures remain central to the field of computer science. Although structures such as binary trees, hash tables, and heaps are staples in data structure courses and widely used in software systems, they typically operate under the assumption of comparison-based access patterns or unbounded universes. However, in certain applications such as real-time systems, devices that have a limited memory, and network routing, faster operations are required on data drawn from fixed-size integer domains.

A theoretically powerful yet often overlooked data structure suitable for such tasks would be the van Emde Boas (vEB) tree. First developed in 1975 by Peter van Emde Boas, the tree is a recursive, space-time efficient data structure designed for operations on fixed-universe integer keys. Unlike traditional comparison-based trees, the vEB tree achieves $O(log(log(U))$ time complexity for operations such as insertion, deletion, and predecessor/successor search, making it ideal in cases where the key universe is known and small.

STRUCTURE

The vEB tree works with integers in a set universe U with size 2^k for some integer k. Each vEB node manages a sub-universe of size $u = \sqrt{U}$ and is recursively structured with two key components: a summary node and an array of clusters. The summary is a vEB tree of size $\sqrt{u}$ and is used to track which clusters are nonempty, while each cluster is itself a vEB tree managing a universe of size $\sqrt{u}$. This recursive decomposition continues until the base case $u = 2$ is reached. Because the vEB tree maintains minimum and maximum values at each node, the operations findMin and findMax can be performed in constant $O(1)$ time. More complex operations such as insertion, deletion, and finding successor/predecessor, run in $O(log(log(U)))$ time.

Insertion

The insert(x) operation in a vEB tree begins by addressing trivial cases. If the tree is currently empty, both the minimum (min) and maximum (max) are simply set to x. If x is smaller than the current min, it is swapped with it. The original minimum, now larger, must still be inserted recursively as if it were x itself.

For trees where $u > 2$, the tree divides x into two components: high(x) and low(x). The high(x) operation computes the cluster number with $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$, whereas the low($x$) operation computes the element's index within the cluster with $x \mod \sqrt{u}$. As each cluster is represented as a smaller vEB tree, and the tree recursively inserts the low(x) element into the appropriate cluster identified by high(x). If the cluster is empty, the tree first inserts the high(x) index into the summary tree and then recursively inserts low(x) into the new cluster.

After the insertion, the maximum value of the tree may need to be updated if x is larger than the current maximum. This update is typically handled at the end of the operation. The constant updating of the min and max values for each insert operation ensures that calls to them will take constant $O(1)$ time. The insert(x) function itself takes $O(log(log(u)))$ time as a result of each recursive iteration reducing the problem size by a factor of $\sqrt{u}$.

Deletion

Similar to insert(x), the delete(x) operation begins by addressing trivial cases. If the tree contains only a singular element (e.g. $min == max$), the element is simply discarded, and min and max are set to $nullptr$ (or similar). In the event that the universe size is $u = 2$, we delete the corresponding node and set only the identifiers to $nullptr$.

In cases where the size of the universe is $u > 2$, there are several things to keep in mind. If the element to be deleted, x, is the current min value, the vEB tree will find the next smallest value from the non-empty cluster with the smallest index. This new value is then promoted to min and is deleted from its former location. x is deleted as well, and we have successfully promoted a new minimum value.

In the event that x is not the min value and that $u > 2$, the deletion proceeds directly to the appropriate cluster based on the high(x) and low(x) calculation of the x's value (same formula as above). If, after deletion, the target cluster becomes empty, its index must be removed from the summary tree to accurately reflect the updated structure. If x was also the maximum, the vEB tree determines whether the summary still holds any values; if it does, max is updated using the largest active cluster; else it is set to the current min value.

Search

The search operation begins by checking if the element x matches the current min or max. If it does, the search is completed immediately as there is a constant $O(1)$ access to these continuously updated data members. If x does not match either value, the tree then recursively searches itself using high(x) and low(x). If x is not found in the cluster determined by low(x), the search continues in the summary tree which contains the minimal values of the non-empty clusters. The recursive search process reduces the problem size by a factor of $\sqrt{u}$ at each step, ensuring that the time complexity remains $O(\log \log u)$.

TABLE 1. Time Complexities on Various Tree Structures

Type	Insertion	Deletion	Search
AVL Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$
Binary Tree (unordered)	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$
B-Tree (order *m*)	$O(log_m(n))$	$O(log_m(n))$	$O(log_m(n))$
B+ Tree (order *m*)	$O(log_m(n))$	$O(log_m(n))$	$O(log_m(n))$
Heap (Binary)	$O(log(n))$	$O(log(n))$	$O(n)$
K-D Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$
Red-Black Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$
Segment Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$
Van Emde Boas Tree	$O(log(log(U)))$	$O(log(log(U)))$	$O(log(log(U)))$

Note: Notation n refers to the number of nodes in the tree. Instances of $log(n)$ may be replaced with h representing tree height instead.

CONSIDERATIONS

While the van Emde Boas tree provides impressive theoretical $O(log(log(U)))$ time for operations, these benefits come with important limitations. Chief among them is space usage: vEB trees allocate $O(U)$ space regardless of how many elements are actually stored. This means that even with only a few elements, the tree may allocate a large amount of memory if the universe size is large. For large universes where the number of elements is sparse, this can result in substantial memory overhead and make the structure infeasible in practice.

For instance, suppose that we want to hold 10 random integers using a std::vector<int> and a vEB tree and compare their space overheads. A std::vector stores data contiguously in memory, so holding 10 4-byte integers requires just 40 bytes, plus a small amount of metadata overhead. This brings the estimated total to around 50 to 60 bytes.

On the other hand, the vEB tree uses $O(U)$ space. If we assume that $U = 2^{16} = 65,536$, which is the smallest power-of-two universe large enough to represent all 16-bit integers (and is required as we will be choosing 10 random integers and thus need to ensure that our universe contains all possibilities), the vEB tree must allocate space for its recursive structure, including about 65,536 nodes. Assuming each node takes around 24-32 bytes (including the integers for min/max, and pointers to clusters/summary), the total memory used would be approximately 65536×24 bytes, or about 1.5 megabytes.

Ultimately, we see that even with a rough estimate calculation, the vEB tree is nowhere close to outperforming traditional std::vectors and arrays in terms of storage. vEB tree's main key advantage is its speed in processing information and modifying its structure, but it comes at a heavy storage cost.

APPLICATONS

Despite their drawbacks, van Emde Boas trees can still be highly useful in specific contexts where their theoretical speed advantages outweigh their memory costs. In particular, they are well-suited for applications involving dense sets drawn from a known and fixed universes. For instance, in network packet routing, priority queues for scheduling systems, or real-time event simulations — where the universe of possible keys (like timestamps, IP addresses, or priority levels) is known and bounded — vEB trees offer extremely fast operations that can ensure system responsiveness.

Additionally, vEB trees may outperform other structures when many operations need to be performed on a dynamic set. In situations where time constraints are strict but the available memory is sufficient (e.g. embedded systems with large but predictable key spaces), vEB trees strike a favorable balance.

However, contemporary processors are incredibly fast, and their large, sophisticated cache hierarchies reward data structures that maintain good spatial locality (something vEB trees inherently lack due to their recursive, pointer-heavy layout). In contrast, simpler and more memory-efficient structures such as red-black trees, hash tables, and B-trees are not only easier to implement, but often outperform vEB trees in real-world applications as they handle cache behavior and use memory more efficiently. Moreover, these alternatives avoid the excessive space overhead of $O(U)$ memory usage, making them far more scalable when the universe size is large but sparsely populated (a theme we have seen heavily up till now). For most practical applications today, the marginal speed advantage promised by the vEB tree does not outweigh its complexity and high memory cost. As a result, in modern software development, where both memory efficiency and implementation simplicity are highly valued, the van Emde Boas tree is rarely the structure of choice, adding them to a ever-growing list of theoretically elegant but practically underutilized data structures.

TESTING

To test the van Emde Boas Tree, we will first create the structure using C++. The source code for the tree will be separated from the test file to make the setup modular and maintainable. Once the environment is ready, basic functional tests should be written to verify the fundamental operations.

To test the insertion functionality of the van Emde Boas tree, we will start by inserting values one by one and verifying that they can be correctly found using the search function. After each insertion, we will check that the min and max values are updated as expected. Specifically, if a value smaller than the current min is inserted, it should become the new min, and similarly, if a value larger than the current max is inserted, it should become the new max. We will also test inserting values into an initially empty tree and ensure that the tree correctly adjusts its structure when moving from an empty state to containing several elements. Edge cases such as inserting into a tree with only one element or inserting duplicate values will also be tested to ensure proper handling of these scenarios.

The deletion operation will be tested by removing the min, max, and other arbitrary elements from the tree. For each deletion, we will verify that the min and max fields are updated correctly, especially in cases where the deleted element was the current min or max. When deleting the min value, we will ensure that the new min is correctly promoted from the non-empty clusters, and the internal structure is reorganized properly. Deleting arbitrary elements will also be tested, and we will check that the tree correctly handles the removal and adjusts its internal structure without leaving any inconsistencies.

To test the search functionality of the van Emde Boas tree, we will verify that the search operation returns the correct results for various queries. Initially, we will test by searching for values that have been inserted into the tree, ensuring that the correct element is found each time. This includes searching for the minimum, maximum, and middle values to ensure that the tree maintains its structural integrity after multiple insertions. We will also test searching for elements that are not present in the tree, verifying that the search operation correctly identifies the absence of the element.

If time permits, a recursive printing function may also be created so that we may visually see the changes occurring to the vEB tree per operation. If not, a simply printing function will suffice to ensure that the tree performs as expected.

To validate the theoretical performance of the van Emde Boas tree, especially its $O(log(log(U)))$

time complexity, performance tests will be conducted using C++ and the <chrono> timing library (or other depending on feasibility). Randomized sequences of integers will be inserted and searched in vEB trees initialized with varying universe sizes $u = 2^k$, where k will range from approximately 8 to 18 (once again, depending on the implementation, time, and memory usage, the range of k may be shifted). For each size, 10 operations will be timed individually, with the average of them being recorded as a result. These range-of-k measurements will be plotted against the graph of $O(log(log(U)))$ to verify the expected scaling behavior. The timing will help determine how the tree's performance changes as the universe expands, providing clear insights into how the data structure behaves in practice.

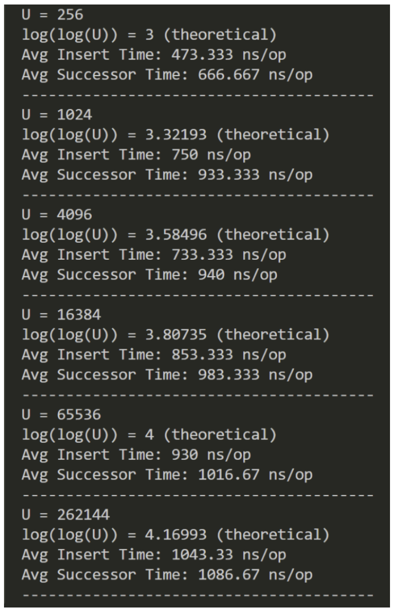
CODE RESULTS

The code for creating and testing the vEB tree was organized into two separate files to improve modularity and maintainability. The primary implementation, along with all associated unit tests, was written in a C++ file. This file included the complete definition of the vEB tree class, as well as functions to insert, delete, and search for elements while tracking operation timings. To improve data analysis, visualization, and presentation of performance trends, a companion Python script was used generate plots. This script utilized libraries such as matplotlib visualize key metrics like operation time complexity and space usage across various input sizes, enabling a clearer understanding of how the vEB tree behaves in practice. Ultimately, the terminal outputs and the graphs generated provide a good understanding of how efficient our implementation really is.

The code files can be found here: `https://github.com/DPandaman/van-Emde-Boas-Tree-Tests`. Now onto the results:

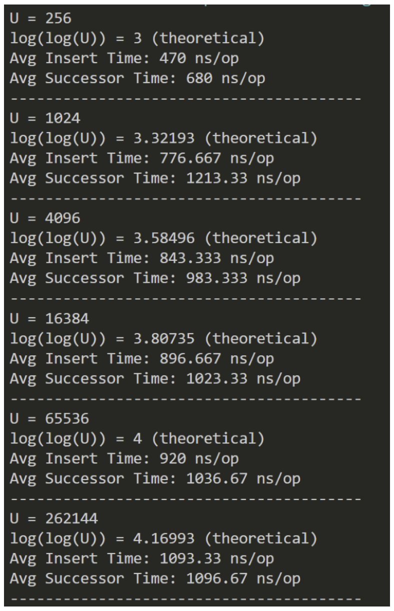
Three main trials were performed. For each trial, universes with k values $k \in \{\, k \in \mathbb{Z} \mid 8 \leq k \leq 18,\ k \equiv 0 \pmod 2 \,\}$ were chosen, leading to the final set $U = \{256, 1024, 4096, 16384, 65536, 262144\}$. To ensure consistency across the various universe sizes, the member functions of the tree were tested with the relatively small set $N = 3, 5, 2, 8, 12, 15$. Each test trial measured the average runtime of both insert and successor operations. For every universe size, the full set N was inserted and then successor queries were executed for each element. The time per operation was recorded in nanoseconds and averaged over several runs to reduce noise due to hardware fluctuations, OS

FIGURE 1. Numerical Outputs to Three Runs



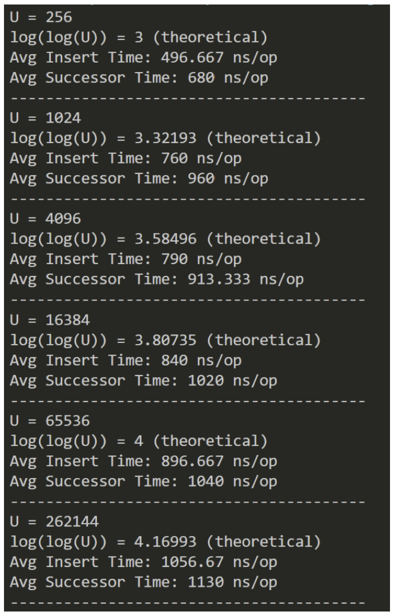
```
U = 256
log(log(U)) = 3 (theoretical)
Avg Insert Time: 473.333 ns/op
Avg Successor Time: 666.667 ns/op
--------------------------------------
U = 1024
log(log(U)) = 3.32193 (theoretical)
Avg Insert Time: 750 ns/op
Avg Successor Time: 933.333 ns/op
--------------------------------------
U = 4096
log(log(U)) = 3.58496 (theoretical)
Avg Insert Time: 733.333 ns/op
Avg Successor Time: 940 ns/op
--------------------------------------
U = 16384
log(log(U)) = 3.80735 (theoretical)
Avg Insert Time: 853.333 ns/op
Avg Successor Time: 983.333 ns/op
--------------------------------------
U = 65536
log(log(U)) = 4 (theoretical)
Avg Insert Time: 930 ns/op
Avg Successor Time: 1016.67 ns/op
--------------------------------------
U = 262144
log(log(U)) = 4.16993 (theoretical)
Avg Insert Time: 1043.33 ns/op
Avg Successor Time: 1086.67 ns/op
--------------------------------------
```

(a) Run #1



```
U = 256
log(log(U)) = 3 (theoretical)
Avg Insert Time: 470 ns/op
Avg Successor Time: 680 ns/op
--------------------------------------
U = 1024
log(log(U)) = 3.32193 (theoretical)
Avg Insert Time: 776.667 ns/op
Avg Successor Time: 1213.33 ns/op
--------------------------------------
U = 4096
log(log(U)) = 3.58496 (theoretical)
Avg Insert Time: 843.333 ns/op
Avg Successor Time: 983.333 ns/op
--------------------------------------
U = 16384
log(log(U)) = 3.80735 (theoretical)
Avg Insert Time: 896.667 ns/op
Avg Successor Time: 1023.33 ns/op
--------------------------------------
U = 65536
log(log(U)) = 4 (theoretical)
Avg Insert Time: 920 ns/op
Avg Successor Time: 1036.67 ns/op
--------------------------------------
U = 262144
log(log(U)) = 4.16993 (theoretical)
Avg Insert Time: 1093.33 ns/op
Avg Successor Time: 1096.67 ns/op
--------------------------------------
```

(b) Run #2



```
U = 256
log(log(U)) = 3 (theoretical)
Avg Insert Time: 496.667 ns/op
Avg Successor Time: 680 ns/op
--------------------------------------
U = 1024
log(log(U)) = 3.32193 (theoretical)
Avg Insert Time: 760 ns/op
Avg Successor Time: 960 ns/op
--------------------------------------
U = 4096
log(log(U)) = 3.58496 (theoretical)
Avg Insert Time: 790 ns/op
Avg Successor Time: 913.333 ns/op
--------------------------------------
U = 16384
log(log(U)) = 3.80735 (theoretical)
Avg Insert Time: 840 ns/op
Avg Successor Time: 1020 ns/op
--------------------------------------
U = 65536
log(log(U)) = 4 (theoretical)
Avg Insert Time: 896.667 ns/op
Avg Successor Time: 1040 ns/op
--------------------------------------
U = 262144
log(log(U)) = 4.16993 (theoretical)
Avg Insert Time: 1056.67 ns/op
Avg Successor Time: 1130 ns/op
--------------------------------------
```

(c) Run #3

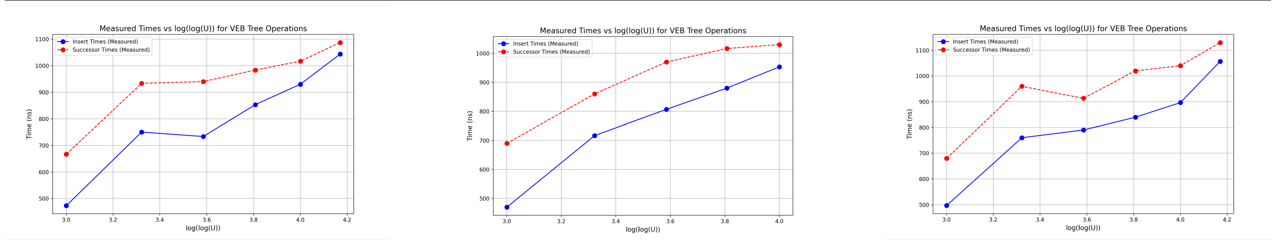
scheduling, and caching effects. This is done under the assumption that the insert/remove functions and the successor/predecessor functions will have the same run times.

What is clear between the three trials is that while the insert(x) and successor(x) operations increase slowly, successor(x) consistently takes slightly longer than insert(x). This discrepancy is surprising at first, since both are expected to run in $O(log(log(U)))$ time and use similar recursive structures. However, in practice, the specific control flow and conditional branching involved in successor may lead to longer paths through the recursive structure. For example, the insert operation often terminates early when inserting into an empty subtree, which is a common case in sparse universes (such as the one that N occupies), whereas the successor may have to search deeper to find the next occupied slot.

A possible explanation is that successor(x) tends to go deeper into recursion and more conditional branching, especially when it needs to search across clusters to find the next element. This can lead to more irregular memory access and potential cache misses compared to insert(x), which often terminates early in sparse universes. These differences can make successor(x) slightly slower in practice.

Figure 2 shows the Python-generated graphs corresponding to each trial run presented in Figure 1. Each subplot corresponds to one of the three trials and plots the average operation time for insert(x) and successor(x) against increasing universe sizes. The x-axis represents $log(log(U))$, reflecting the

FIGURE 2. Comparison Charts for Three Runs



(a) Run #1 **(b) Run #2** **(c) Run #3**

theoretical time complexity, while the y-axis shows the average time per operation in nanoseconds. As the universe size U increases from 256 to 262,144, $log(log(U))$ gradually increases from 3 to approximately 4.17. Correspondingly, the average operation times also increase slowly, rising from around 500–700 ns to about 1000–1100 ns.

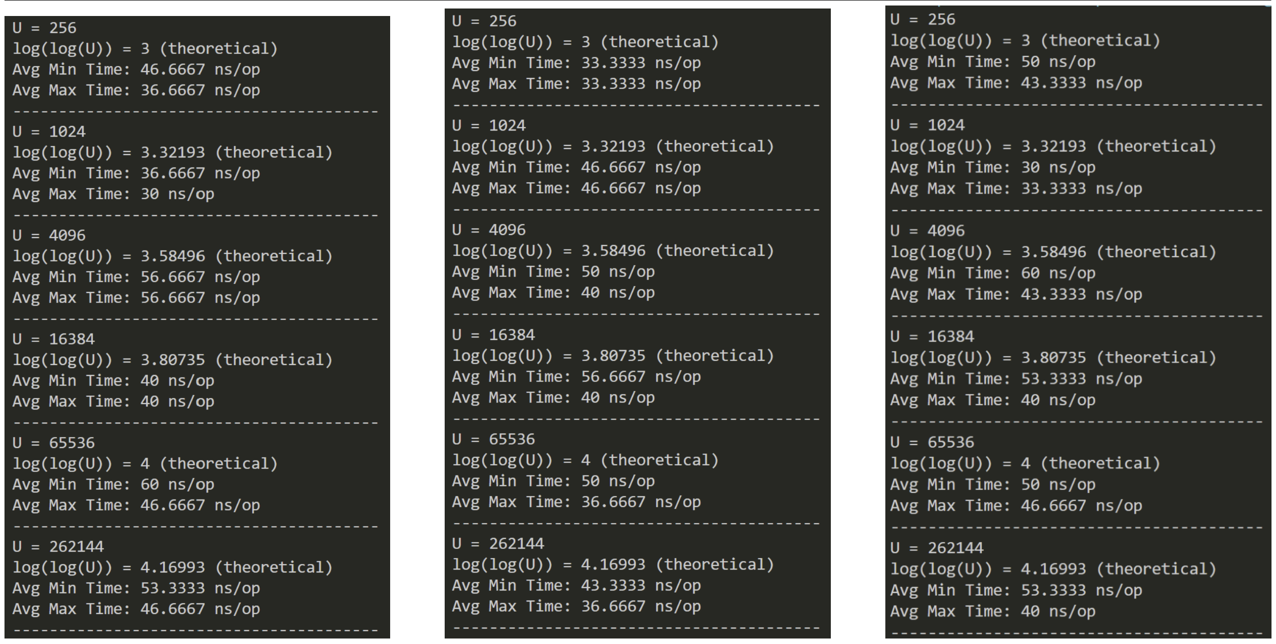
Ultimately, these graphs illustrate the slow growth in runtime as U increases, confirming the theoretical $O(log(log(U)))$ expectation. They also highlight the consistently higher cost of successor(x) compared to insert(x) across all universe sizes and trials, suggesting additional overhead in its recursive search even though both share the same theoretical complexity.

Out of the three inital trial runs, Run #2 seems to be the most interesting. As is evident with the graphs, Run #1 and #3 follow a similar shape of a steep initial incline, followed by a very slight drop, and finally by consistent, steady, slow increase. Run #2, however, never has a slight dip in the middle and instead continiously increases with smaller increments as $O(log(log(U)))$ gets larger.

The slight dip observed in Runs #1 and #3 could be due to random fluctuations in the data or specific artifacts of the particular system state during those runs. For example, if certain cache optimizations or CPU scheduling factors played a role, they could have caused temporary speed-ups at certain points. In Run #2, however, these fluctuations might have been absent, leading to a smoother, continuous increase. This could have been something as simple as not clearing the terminal before re-running, or something far more complex.

The second part of the tests focused on testing the minimum() and maximum() functions to check whether or not they work in $O(1)$ constant time. Once again, to ensure consistency, the implementation, set N, and the k values for the universe were kept exactly the same. Only the main() function was modified to test for the values.

FIGURE 3. Minimum and Maximum Trial Runs



```
U = 256
log(log(U)) = 3 (theoretical)
Avg Min Time: 46.6667 ns/op
Avg Max Time: 36.6667 ns/op
--------------------------------------
U = 1024
log(log(U)) = 3.32193 (theoretical)
Avg Min Time: 36.6667 ns/op
Avg Max Time: 30 ns/op
--------------------------------------
U = 4096
log(log(U)) = 3.58496 (theoretical)
Avg Min Time: 56.6667 ns/op
Avg Max Time: 56.6667 ns/op
--------------------------------------
U = 16384
log(log(U)) = 3.80735 (theoretical)
Avg Min Time: 40 ns/op
Avg Max Time: 40 ns/op
--------------------------------------
U = 65536
log(log(U)) = 4 (theoretical)
Avg Min Time: 60 ns/op
Avg Max Time: 46.6667 ns/op
--------------------------------------
U = 262144
log(log(U)) = 4.16993 (theoretical)
Avg Min Time: 53.3333 ns/op
Avg Max Time: 46.6667 ns/op
--------------------------------------
```

(a) Run #1

```
U = 256
log(log(U)) = 3 (theoretical)
Avg Min Time: 33.3333 ns/op
Avg Max Time: 33.3333 ns/op
--------------------------------------
U = 1024
log(log(U)) = 3.32193 (theoretical)
Avg Min Time: 46.6667 ns/op
Avg Max Time: 46.6667 ns/op
--------------------------------------
U = 4096
log(log(U)) = 3.58496 (theoretical)
Avg Min Time: 50 ns/op
Avg Max Time: 40 ns/op
--------------------------------------
U = 16384
log(log(U)) = 3.80735 (theoretical)
Avg Min Time: 56.6667 ns/op
Avg Max Time: 40 ns/op
--------------------------------------
U = 65536
log(log(U)) = 4 (theoretical)
Avg Min Time: 50 ns/op
Avg Max Time: 36.6667 ns/op
--------------------------------------
U = 262144
log(log(U)) = 4.16993 (theoretical)
Avg Min Time: 43.3333 ns/op
Avg Max Time: 36.6667 ns/op
--------------------------------------
```

(b) Run #2

```
U = 256
log(log(U)) = 3 (theoretical)
Avg Min Time: 50 ns/op
Avg Max Time: 43.3333 ns/op
--------------------------------------
U = 1024
log(log(U)) = 3.32193 (theoretical)
Avg Min Time: 30 ns/op
Avg Max Time: 33.3333 ns/op
--------------------------------------
U = 4096
log(log(U)) = 3.58496 (theoretical)
Avg Min Time: 60 ns/op
Avg Max Time: 43.3333 ns/op
--------------------------------------
U = 16384
log(log(U)) = 3.80735 (theoretical)
Avg Min Time: 53.3333 ns/op
Avg Max Time: 40 ns/op
--------------------------------------
U = 65536
log(log(U)) = 4 (theoretical)
Avg Min Time: 50 ns/op
Avg Max Time: 46.6667 ns/op
--------------------------------------
U = 262144
log(log(U)) = 4.16993 (theoretical)
Avg Min Time: 53.3333 ns/op
Avg Max Time: 40 ns/op
--------------------------------------
```

(c) Run #3

After running three trials, it is pretty clear that our implementation is capable of performing the operations in constant O(1) time. Run #1 yielded an average of 39.4 ns for the minimum time and an average of 42.8 ns for the maximum time, Run #2 yielded an average of 46.7 ns for the minimum time and an average of 38.9 ns for the maximum time, and Run #3 yielded an average of 49.4 ns for the minimum time and an average of 41.1 ns for the maximum time. When compared to Figure 1, Figure 2 demonstrates that increasing k values have very little impact on the runtime. Despite these slight fluctuations in timings, the times remain very close to each other, reinforcing the fact that both operations consistently perform in constant time.

These results indicate that the execution time of the minimum() and maximum() functions is largely independent of the universe size U, with the average operation times remaining relatively stable across trials. The small differences between the runs are likely due to minor fluctuations in system performance such as varying CPU load or other environmental factors during each test. However, these variations are negligible compared to the expected behavior of O(1) operations. These consistent timings further validate the efficiency of the van Emde Boas tree implementation for these operations.

IMPROVEMENTS

While the model roughly demonstrated the efficiency and capability of a vEB tree, especially for operations such as insertion, successor, and minimum/maximum queries, there are several areas where the tests could be improved to enhance the accuracy and reliability of the findings.

There are several factors that could have impacted the precision and reliability of our results. One potential source of error in our testing process was the hardware and environmental conditions under which the experiments were conducted. CPU load, background processes, and other activities could have affected the timing of individual operations. For instance, fluctuations in system performance could explain the slight variations in operation times across different runs. While these variations were small, a more controlled testing environment or the use of performance profiling tools could help ensure more consistent and reproducible results.

Moreover, we focused primarily on the performance of the vEB tree in a sparse universe, but the performance in denser universes could differ. Our set N contained only six elements, though while enough to test several operations, is limits how extensively the tree can be tested. Sparse universes often lead to earlier terminations of recursive calls, which can result in faster operations due to fewer elements being processed. In contrast, a denser universe might result in deeper recursion and slower performance. Future tests should explore this aspect by using different configurations of input data, such as both sparse and dense universes, to provide a fuller picture of the vEB tree's performance across a variety of cases.

Lastly, it is worth noting that memory usage was not directly tested in this project, primarily because evaluating the vEB tree's space efficiency would have required allocating massive amounts of memory—especially for large universe sizes like $U = 2^{32}$ or higher. Attempting to initialize trees of that scale would be impractical on most machines, as the recursive nature of the data structure leads to high memory overhead. Each vEB tree contains multiple levels of subtrees and summary trees, many of which may be allocated even if they remain unused. As a result, any meaningful analysis of memory performance would require specialized tools, hardware, or alternative methods such as tracking allocation counts or using profiling software. While the tests successfully demonstrated the time efficiency of the structure, a more thorough investigation into memory usage remains a potential area for future work.

Despite these drawbacks, one of the key strengths of our implementation was its ability to perform operations in expected time bounds, particularly the minimum() and maximum() functions, which showed stable constant-time performance across all three runs. The results for these operations were consistent, with only minor fluctuations in timing across different trials, which were likely caused by environmental factors or minor issues with the implementation. This indicates that the fundamental design of the vEB tree and its operations, particularly for finding the minimum and maximum values, works as intended and conforms to the expected $O(1)$ time complexity.

The second notable strength was the overall scalability of the data structure, especially with the insert(x) and successor(x) operations. Although the expected time complexity for these operations is $O(log(log(U)))$, the actual observed increase in execution time remained relatively slow and predictable as the universe size U grew. This proves that the vEB tree's performance is efficient even for large inputs, confirming the utility of this data structure in managing large sets efficiently.

REFLECTIONS

Ultimately, the van Emde Boas tree is an incredibly powerful data structure for its niche applications. Though there is a quite steep learning curve, it is not too difficult to get used to its unique properties. Perhaps the most difficult aspect of the process was the initial set up for the recursive structure, and its lengthy runtime with the tests themselves. Debugging issues related to recursive insertions, empty clusters, or incorrect successor results often meant tracing through multiple layers of calls and inspecting deeply nested tree states, which was mentally demanding. Moreover, although the data structure itself may be quick at each operation, it was observed that the above implementation took several tens of seconds when working with universes of sizes $U = 262,144+$.

Additionally, it was worth noting that the test cases and timed outputs grew larger when the device was removed from charging. Though this was never tested beyond some sample iterations and comparisons, it seems as if the device would slow its performance down depending on how much battery power it had. If this is not a fluke and indeed an example of active control, then it is possible to speed up/down the tree by changing the hardware.

CONCLUSION

In conclusion, the van Emde Boas (vEB) tree is a theoretically powerful yet practically nuanced data structure. Through the process of implementation and testing, we validated many of the structure's claimed performance characteristics. In particular, we observed that operations such as insert(x) and successor(x) scale in factor with the expected $O(log(log(U)))$ time complexity, though with some variance due to recursion overhead and control flow complexity. However, functions like minimum() and maximum() consistently executed in constant time across multiple trials, reinforcing the structure's efficiency for simple queries. These results highlight the vEB tree's unique suitability for scenarios requiring fast dynamic set operations within a bounded universe.

However, it was also revealed the limitations and challenges that arise in practice. While theoretically efficient, the recursive nature of the vEB tree introduces non-negligible runtime and memory overhead when implemented naively. In our tests, initializing and operating on large universe sizes resulted in slow test execution, not because the operations themselves were inefficient, but due to the exponential growth in structure depth and the resulting memory allocation and function call overhead. Furthermore, the memory demands of the structure made it impractical to measure or benchmark memory usage on large instances.

Despite its complexity, the van Emde Boas tree served as a valuable case study in advanced data structure design and recursive algorithm implementation. The process of building and testing the structure provided deeper insights into how theoretical performance translates to real-world behavior. Minor implementation details, particularly around memory management and recursion, were found to significantly affect practical performance. Future iterations could benefit from optimized memory allocation, more efficient handling of base cases, and the use of hybrid structures that employ simpler models for small universes.

CITATIONS

: Peter van Emde Boas: Preserving order in a forest in less than logarithmic time (Proceedings of the 16th Annual Symposium on Foundations of Computer Science 10: 75-84, 1975)

: Pandey, Sandeep. "Van Emde Boas Trees." *CS7280: Advanced Data Structures, Northeastern University*, https://www.khoury.northeastern.edu/home/pandey/courses/cs7280/spring25/papers/veb.pdf.

: "Lecture 14: Van Emde Boas Trees." *CS166: Data Structures, Stanford University*, 2025. https://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/14/Slides14.pdf. Accessed 8 May 2025.

: "Fusion Tree". OpenGenus IQ: Computing Expertise and Legacy. 4 April 2019.

: Ammer, Thomas; Lammich, Peter (23 November 2021). "van Emde Boas Trees". Archive of Formal Proofs.

: "chrono - C++ Reference." *cplusplus.com*. https://cplusplus.com/reference/chrono/.

: "ctime - C++ Reference." *cplusplus.com*. https://cplusplus.com/reference/ctime/.