Ben Lischin
CS 7280
Professor Prashant Pandey

Van Emde Boas Tree Performance Analysis

Design Considerations

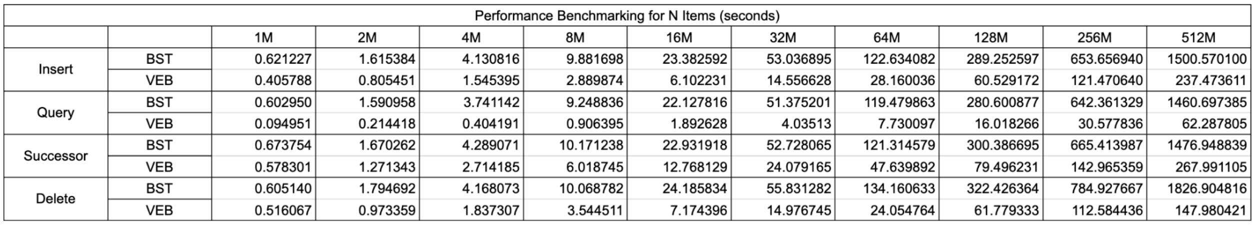
A fully recursive solution requires a base case universe size of 2. At this level, the base bit array can be extrapolated from the min and max values of the structure. An alternative to the expensive low-level recursion here is to break at a larger universe size and load a reasonably sized bit vector (i.e. 256 bits) into memory which we can use for efficient linear search or constant time manipulation at this scale. Beyond reducing the levels of recursion (and thus reducing the total amount of nested VEB objects in memory), this separate base class allows us to throw away many unnecessary fields within the object itself: min, max, and isEmpty can all be cheaply calculated in this context. Further information used for recursion (sqrtU, summary, clusters) can be dropped from this class as well.

To reduce upfront memory consumption, nested clusters are lazily allocated. A vector of known size is initialized with null pointers, deferring the creation of these nested structures until it is explicitly required during insertions, and memory is unallocated when deletions empty a cluster. This approach avoids pre-allocating potentially unnecessary memory at deeper levels of the tree that hold no crucial information; if a structure is empty, we do not need to store it. While this design adds the time cost of dynamic memory allocation during insertions and deletions, it offers a significantly more space-efficient implementation for smaller set representation within the same universe.

To minimize the cost of updating nested structures, min is lazily propagated, lagging one update behind the min of its parent. This staggering ensures that recursive updates are only done when absolutely necessary, when creating a new cluster (on average 1 in every $\sqrt{U}$ insertions), achieving amortized runtime *O(loglogU)* with just a single insertion. Deletions also benefit from this setup – when operating on the current min, we can find the next min replacement in *O(1)* while avoiding extra tree traversal.

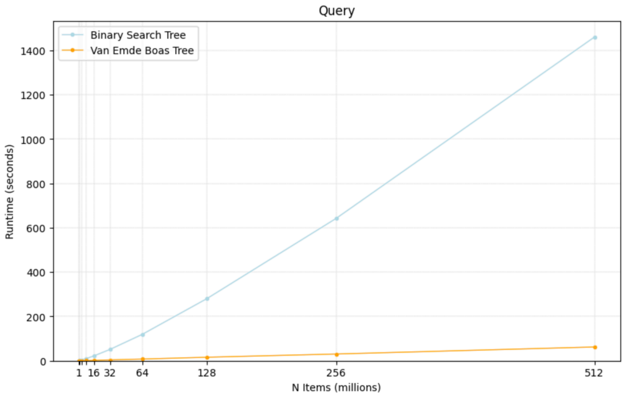
Performance Benchmarking

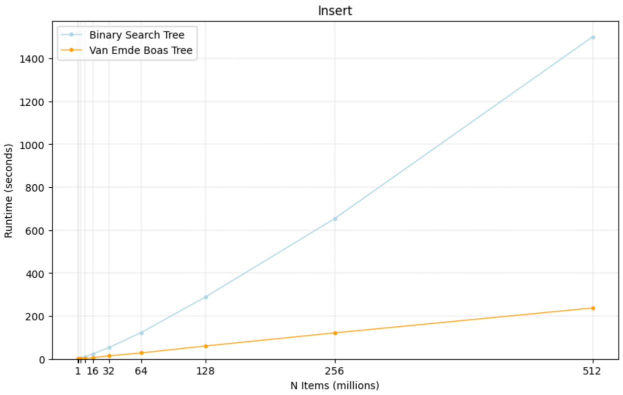
We will be comparing the performance of my Van Emde Boas Tree implementation against a Binary Search Tree implemented with the C++ std::set. Test operations were performed on uniformly random collections of unsigned 32-bit integers ($U = 2^{32}$).

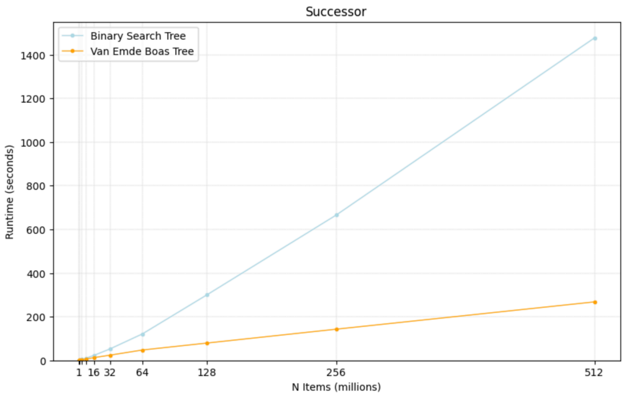


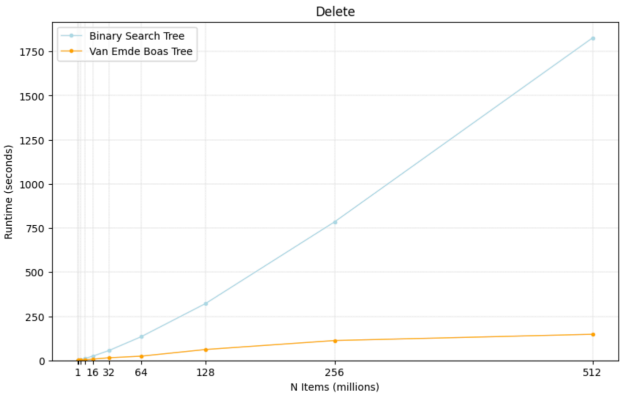
Performance Benchmarking for N Items (seconds)

		1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
Insert	BST	0.621227	1.615384	4.130816	9.881698	23.382592	53.036895	122.634082	289.252597	653.656940	1500.570100
	VEB	0.405788	0.805451	1.545395	2.889874	6.102231	14.556628	28.160036	60.529172	121.470640	237.473611
Query	BST	0.602950	1.590958	3.741142	9.248836	22.127816	51.375201	119.479863	280.600877	642.361329	1460.697385
	VEB	0.094951	0.214418	0.404191	0.906395	1.892628	4.03513	7.730097	16.018266	30.577836	62.287805
Successor	BST	0.673754	1.670262	4.289071	10.171238	22.931918	52.728065	121.314579	300.386695	665.413987	1476.948839
	VEB	0.578301	1.271343	2.714185	6.018745	12.768129	24.079165	47.639892	79.496231	142.965359	267.991105
Delete	BST	0.605140	1.794692	4.168073	10.068782	24.185834	55.831282	134.160633	322.426364	784.927667	1826.904816
	VEB	0.516067	0.973359	1.837307	3.544511	7.174396	14.976745	24.054764	61.779333	112.584436	147.980421





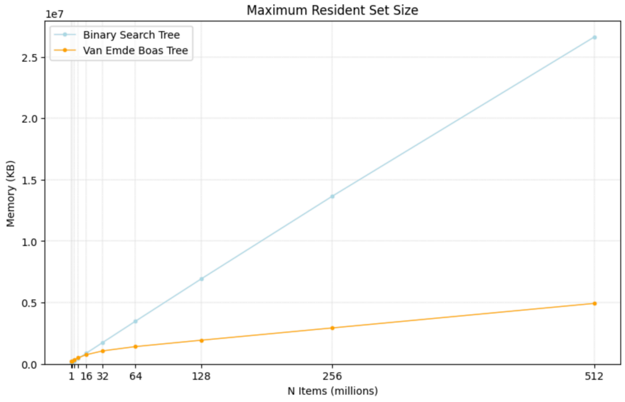




Space Complexity

Maximum resident set size measures the largest amount of memory held in RAM by the running benchmark process. This measures the space including the tree data structure at its largest point, the arrays in_numbers and out_numbers, as well as other minor local variables. We will use this to compare the memory usage of each tree structure.

Maximum Resident Set Size (KB)										
	1M	2M	4M	8M	16M	32M	64M	128M	256M	512M
BST	60,344	115,128	224,488	443,012	879,392	1,750,124	3,483,548	6,917,208	13,654,232	26,630,016
VEB	196,980	247,696	341,476	504,692	752,844	1,064,684	1,415,360	1,932,344	2,932,944	4,931,804



Discussion

Considering the objective of maintaining a collection n of integers within a fixed universe u, we compare the performance of two data structures: Van Emde Boas Tree and Binary Search Tree.

The power of a Van Emde Boas Tree comes from its hierarchy of $\sqrt{U}$ cluster size at every next level. Inserts perform 2 recursive calls in the worst case, producing a recurrence $T(U) = 2T(\sqrt{U}) + O(1)$ for runtime $O(logU)$. One of these recursive calls is for updating the summary struct which only needs to happen when insert into an empty cluster, on average ~1 in every $\sqrt{U}$ insertions. Amortized, we have $T(U) = T(\sqrt{U}) + O(1)$ for runtime $O(loglogU)$. Similarly, the same two recursive calls occur in deletes – one being a summary update which only happens after emptying a cluster. On a populated tree, this only happens ~1 in every $\sqrt{U}$ deletions, giving amortized runtime $O(loglogU)$. Queries and successor searches both make a single recursive call into a nested cluster for runtime $O(loglogU)$. Once any of these methods hit the base case struct, all operations can be brute forced in $O(1)$ given fixed array size 256. VEB space complexity is $O(U)$ as it allocates memory for all possible elements of the universe u, not just those in n. Both the count and size of nested trees are determined by the current universe size ($\sim\sqrt{U}$). Since my implementation supports lazy loading of these nested clusters, the space may approach $O(N)$ for smaller, more concentrated sets. In the worst case, though, a larger or more broadly distributed collection will bring the space closer to $O(M)$ as more unique clusters are populated in memory.

A Binary Search Tree implemented with the C++ standard library's ordered set guarantees balanced tree properties. That is, the height of the left and right subtrees of any node differ by no more than 1, ensuring $O(logN)$ tree height. With this, all operations can be done in $O(logN)$. For benchmarking methods with this structure we have inserts (std::set::insert), queries (std::set::find), successor searches (std::set::lower_bound), and deletes (std::set::erase). Space is $O(N)$ as a BST only maintains the nodes of elements in n, with $O(1)$ information at each node.

The empirical performance visualized in the table and graphs above support these theoretical guarantees. BST runtime grew significantly as N increased, while that had less of an effect on the VEB performance as those methods are bounded by a function of the fixed universe size U. Insert was fast as the tree is relatively empty given the scale of the universe. So the amortized runtime is a good estimate as many insertions may be into new clusters. Deletions at this scale of n are more often performed slower than our amortized bound, since the tree in the context of 32-bit universe is not close to full, so it tends to empty clusters with a delete more often than the theoretical approximation. Queries contain very few computations at each level – this small constant factor makes the operation not just asymptotically fast within the big O bound, but highly efficient in practice. It is the most performant of all the VEB methods.