Binary Heap Priority Queue Conjectures:

Adding n entries to the initially empty queue:

Worst case for inserting requires traversal up the entire height of the tree. Given that each node has a max of two children, the height is Log(n).

O(n) = n \* Log(n)

Amortized running time of m >= n operations chosen at random between ExtractMax, Insert, and IncreaseKey

ExtractMax removes the first value from the array and replaces it with the last item in the array, then swapping each greater child with it until it possibly reaches the bottom. Worst case requires traversal of entire tree of height Log(n). Given m runs of ExtractMax, O(n) = m \* Log(n)

Insert inserts another value to the end of the list, swapping with lesser parents until it reaches its maximum height in the heap. Worst case requires traversal of entire tree of height Log(n). Given m runs of Insert, O(n) = m \* Log(n)

IncreaseKey simply increases the priority to whatever given value is provided. Worst case involved increasing a key at the bottom of tree to the new max, requiring traversal of entire tree of height Log(n). Given m runs of Increase key, O(n) = m \* Log(n)

Each method provides an asymptotic runtime of O(n) = m \* Log(n). As there are m run operations, the amortized analysis of each runtime narrows to O(n) = Log(n).

Van Emde Boas Tree Priority Queue Conjectures:

Adding n entries to the initially empty queue:

Worst case for inserting requires updating lowest clusters in tree. Given that each sub cluster holds a possible universe size of usub = Sqrt(u), where u is the smallest value of U(n) = 2^(2^k) such that u >= n, the height is Log(Log(u)).

O(n) = n \* Log(u)

Amortized running time of m >= n operations chosen at random between ExtractMax, Insert, and IncreaseKey

ExtractMax removes the maximum value from the tree and requires updating of clusters up to the maximum depth of the tree in the worst case, where the height of Log(Log(u)). Given m runs of ExtractMax, O(n) = m \* Log(Log(u))

Insert places a new value in the list. Worst case requires traversal of entire structure to maximum depth of height Log(Log(u)). Given m runs of Insert, O(n) = m \* Log(u)

IncreaseKey increases the priority of the given value to whatever is greater than the current max in the tree. Worst case follows the worst case of an deletion and insert, which are equal in runtime at O(n) = Log(Log(u)). Given m runs of Increase key, O(n) = m \* Log(Log(u))

Each method provides an asymptotic runtime of O(n) = m \* Log(Log(u)). As there are m run operations, the amortized analysis of each runtime narrows to O(n) = Log(Log(u)).

1. Runtime values

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Construction time | N = 100 | N = 1000 | N = 10000 | N = 100000 | N = 1000000 |
| Binary Heap | 3532500ns | 1486500ns | 7017000ns | 286914500ns | 33591447000ns |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| M operations | N = 100 | N = 1000 | N = 10000 | N = 100000 | N = 1000000 |
| Binary Heap | 548800ns | 2249800ns | 7221700ns | 20615500ns | 284333100ns |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Construction time | N = 100 | N = 1000 | N = 10000 | N = 100000 | N = 1000000 |
| Van Emde Boas Tree | 2801600 ns | 2104100 ns | 18663700 ns | 417541800 ns | 35846722500 ns |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| M operations | N = 100 | N = 1000 | N = 10000 | N = 100000 | N = 1000000 |
| Van Emde Boas Tree | 415500 ns | 2139200 ns | 17681800 ns | 156961300 ns | 1528002900 ns |

1. Beginning with the Binary heap, the construction runtimes (thus looking at data initialization and insert only) of the Binary Heap mismatch the expected O(N) = NLog(N) runtimes by factors of 354x, 74x, 26x, 42x, and 842x. The runtime of the M operations (assuming M = 2N, which is its worst case in my test cases) compares to its projected O(N) = MLog(N) by factors of 413x, 112x, 193x, 6x, and 7x, respective to N. There is clearly a trend towards the expected runtimes, for which I might say the larger runtimes (especially seen in the early few constructions of the data structure) are subject to the whims of Java’s memory allocation and implementation of the ArrayList, an issue which seems to intensify when dealing with data sets larger than N = 1000000, where the actual runtime output exceeds the expected by a factor of 842x. I put the strike on Java’s implementation as the runtimes for the M operations seems to stabilize with larger data sets, with the final runtime with N = 100000 holding a runtime only 7 times larger than the projected O(N) = MLog(N) runtime. When first running these experiments, my initial implementation did not include an overloaded constructor for the Binary Heap wherein the Arraylist would be passed an initial size, which resulted in runtimes exceeding 4000x the expected output. Though I would have preferred to implement an array for the structure as to avoid unnecessary overhead, I did not want to handle overflows with inserts and array size. Furthermore, I’m not entirely sure as to what else could be done to improve this specific implementation of the priority queue.

Turning to the Van Emde Boas Tree, we inconsistently find this same phenomenon which has occurred in the Binary Heap implementation. Here however, the asymptotic running time of each of M operations is O(N) = Mlog(log(u)), with a projected construction time of O(N) = Nlog(log(u)), where M >= N. This value would be understandably lower than the runtimes of the Binary Heap due to its additional application of Log upon u, which seems to apply in every case of runtime. In comparing directly, we find the construction times for the Van Emde Boas Tree to relate to the construction times for the Binary Heap by factors of .79x, .706x, .376x, .687x, and 1.067x. Except for the final value, we see the runtime values consistently lower than the Binary Heap, which supports the structure’s conjecture. The two structure’s M operation runtimes compare by factors reported thusly: .757x, .951x, 2.52x, .547x, and .045x. Again, aside from one value, we find an expected decrease in runtime. Of course, the last value is diluted due to the problematic final runtimes of the Binary Heap Priority Queue, but the minimal flip-flopping in larger runtimes presents good evidence for the VEB tree running similar operations at speeds faster than O(N) = NLog(N) or O(N) = MLog(N).

Strangely, the runtimes mismatch the projected runtimes to much greater degrees than the Binary Heap implementation. The construction runtimes exceed the theoretical runtimes—bounded by O(N) = NLog(Log(u))—by factors of 9338x, 526x, 466x, 835x, and 7169x. The M operations exceed the theoretical runtimes—bounded by O(N) = MLog(Log(u)), where M = 2N in the worst case—by factors of 692x, 267x, 221x, 156x, and 152x. These numbers are admittedly quite alarming, and are likely due to some fundamental flaw in my own implementation. In my first attempts analyzing the runtimes in comparison to their theoretical limits, I would like to add that lackluster runtimes found in previous trials prompted heavy reiteration and reimplementation of the VEB structure in order to find faster times than the Binary Heap. The biggest issue was in the creation of the VEB tree, wherein I would initialize each VEB structure recursively until completion. With the trial where N=1000000, the runtimes were astronomical. Worse than that, most of the three operations at this N size would run to completion but provide incorrect outputs. I found that a large issue exists in storing the universe size being an Integer, as the universe size when N=1000000 is 4294967296, which exceeds the Integer’s max size (and thus messes with equational functions like high(x), low(x), and index(x)). To solve the first issue, I changed it so that each VEBPQ struct would only be initialized as it was visited recursively during the input. This prevented the previous issue of memory overflows, as I did not require 65536 subclusters (only 3907 were required for the 1000000 elements) fully initialized. The second issue I solved by adding a new variable to the structure which effectively was a duplicate of the universe integer, rather it was a double so that it could hold greater values to facilitate proper functionality where the universe exceeded maximum integer values. However, aside from these solutions I’ve generated, I can’t confidently express any further reasons as to why my runtimes greatly mismatch the theoretical runtimes to such great a factor.