Lab 11 Report

EECS 560

4/29/19

Evan Brown

Organization of Experimental Profiling

This lab had a very large number of files compared to previous labs, 39, due to significant complexity of the problem. In addition to writing and testing binary search trees, min-heaps, and max-heaps, we also needed the random number generator and timer used to perform profiling in lab 4, an input file reader and parser for testing purposes, and a user interactivity monitor. Similar to in lab 4, once profiling was started, the random number generator was used to generate a large sample of numbers to insert into the data structure being tested, and each insert was timed using the timer library. Then, a number of deletes of both the maximum and minimum numbers in the data structure were timed.

Generating Random Input Data

This lab used the random number generator library that was written for lab 4. The most useful functions in the library are *seedTime*, which uses the current system time to seed the random number generator and ensure that the numbers given are as close to random as possible in order to obtain representative test results, and *getFromOneTo*, which takes a maximum upper bound and returns a number between 1 and the upper bound, inclusive.

CPU Time Recording in C++

This lab also used the timer library that was written for lab 4. This library has simple *start*, *stop*, *reset*, and *get* functions, and can also return values in milliseconds. In order to calculate meaningful values, the library divides timing values by the constant CLOCKS_PER_SECOND and multiplies by 1000 to convert CPU cycle counts into milliseconds.

Data Recording and Analysis

In this experiment, m = 1,000,000

For each data structure, the following process was performed:

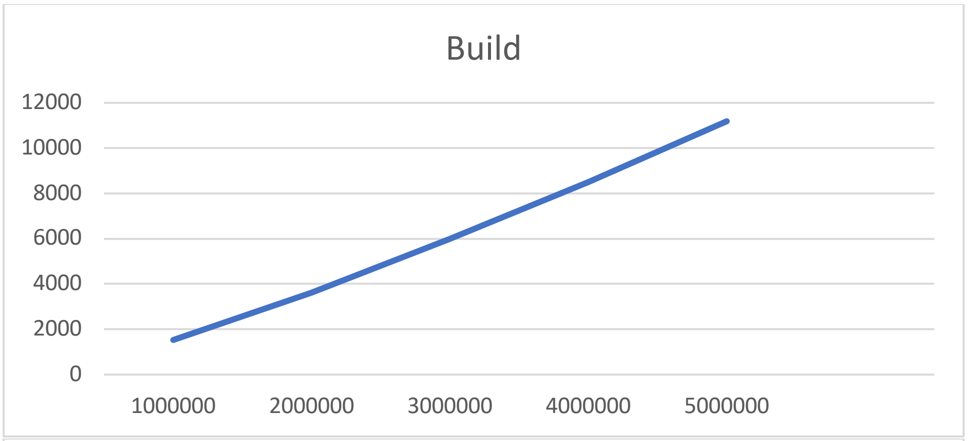
1. Create an initially empty data structure (heaps used an initial array size of 10,000,000)
2. Start the timer
3. Insert a random number between 1 and 5m into the data structure
4. Stop the timer
5. Add the time taken to the total build time
6. Repeat steps #2 to #5 m times
7. Start the timer
8. Delete the minimum number in the data structure
9. Stop the timer
10. Add the time taken to the total deleteMin time
11. Repeat steps $7 to #10 0.001m times
12. Start the timer
13. Delete the maximum number in the data structure
14. Stop the timer
15. Add the time taken to the total deleteMax time
16. Repeat steps #12 to #15 0.001m times
17. Repeat steps #1 to #16 4 more times, and record the average total build, deleteMin, and deleteMax times for the 5 trials.
18. Repeat steps #1 to #17 4 more times, but replace m in steps #6 and 0.001m in steps #11 and #16 with 2m & 0.002m, 3m & 0.003m, 4m & 0.004m, and 5m & 0.005m

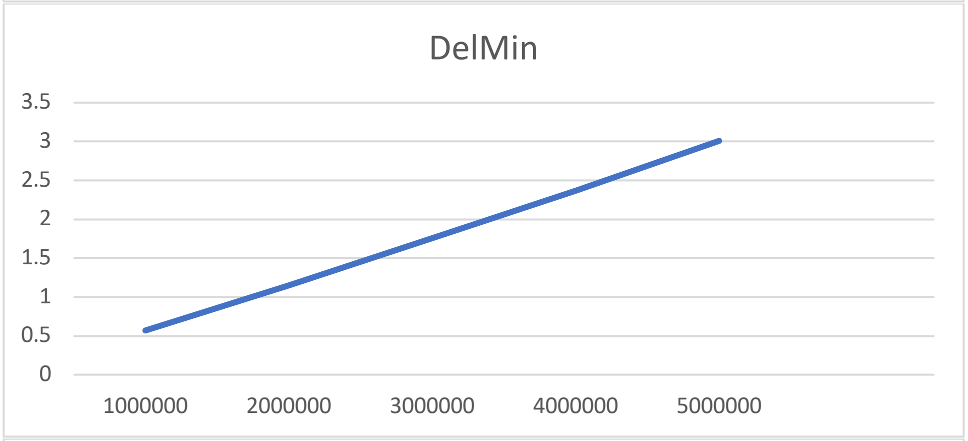
Performance Comparison & Summary

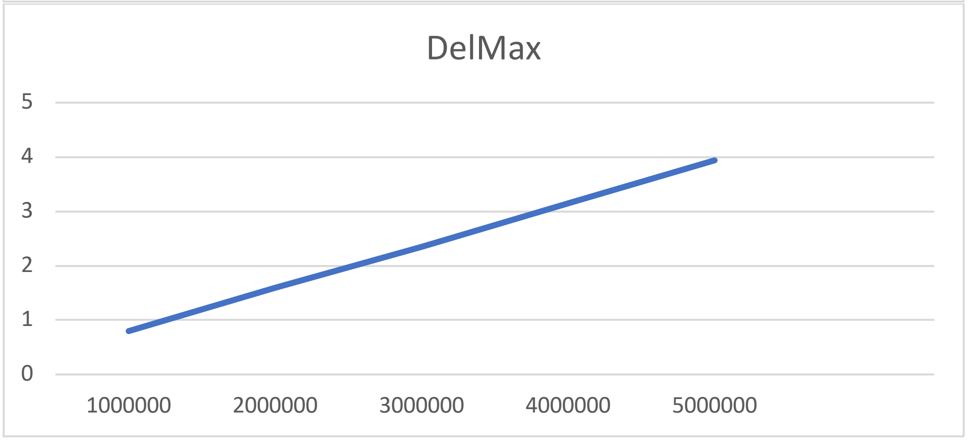
The recorded data was as follows:
(All times are in milliseconds)

Binary Search Tree:

# elements	**1000000**	**2000000**	**3000000**	**4000000**	**5000000**
Build	1528.91	3630.36	5982.54	8514.71	11186.9
DelMin	0.5684	1.1464	1.7582	2.367	3.008
DelMax	0.7972	1.5926	2.3496	3.1502	3.939

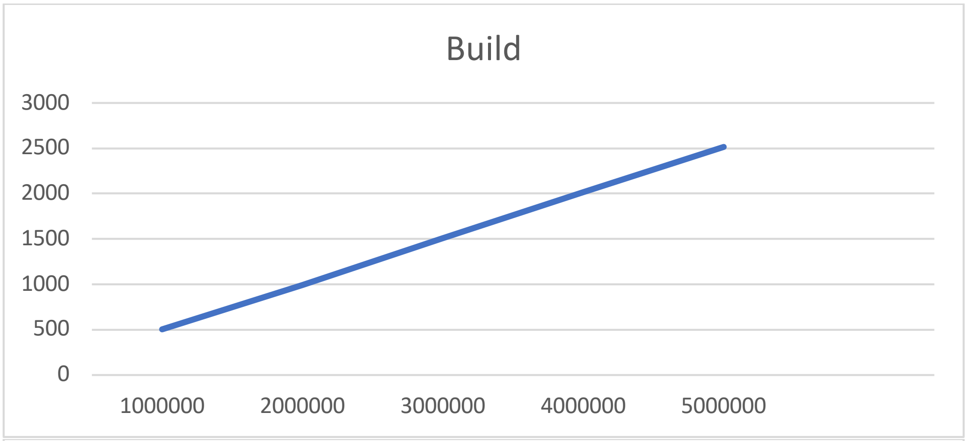


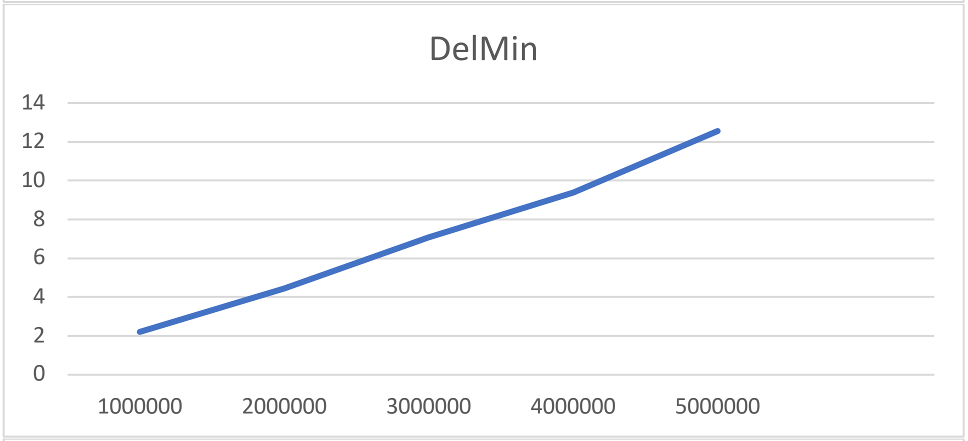


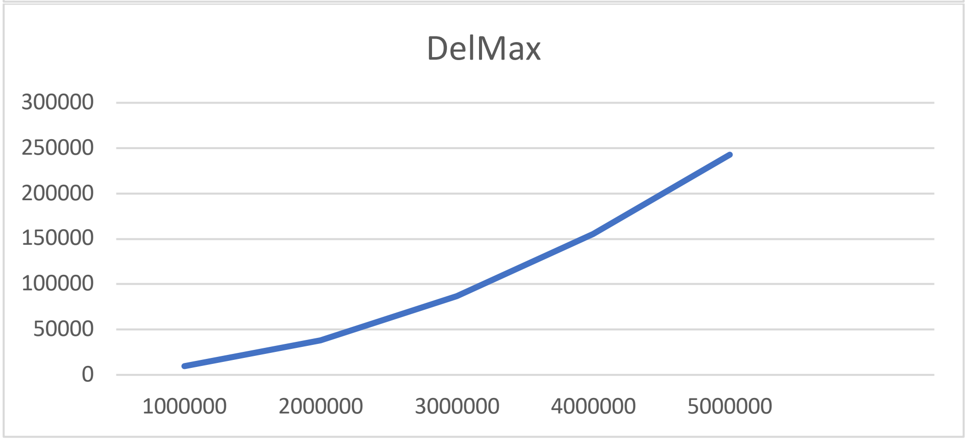


Min 5-Heap

# elements	**1000000**	**2000000**	**3000000**	**4000000**	**5000000**
Build	503.895	994.664	1511.32	2017.93	2515.29
DelMin	2.209	4.4428	7.0656	9.3776	12.5562
DelMax	9602.88	38372.9	87148.4	155376	242797

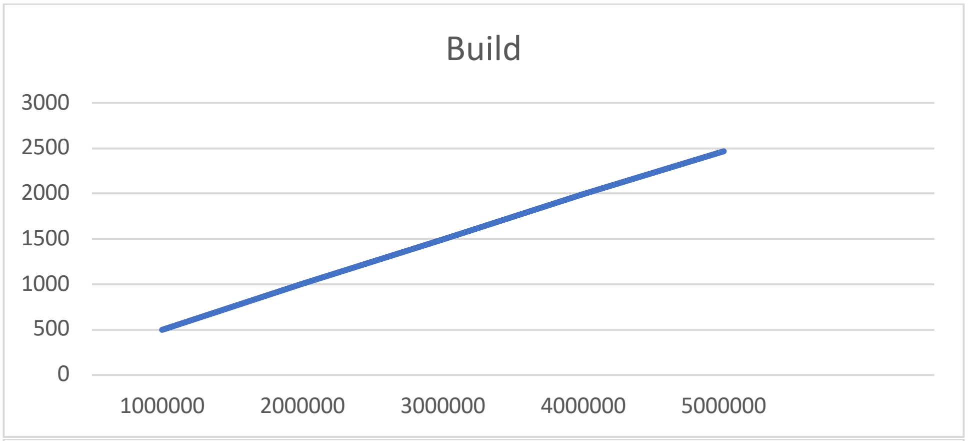


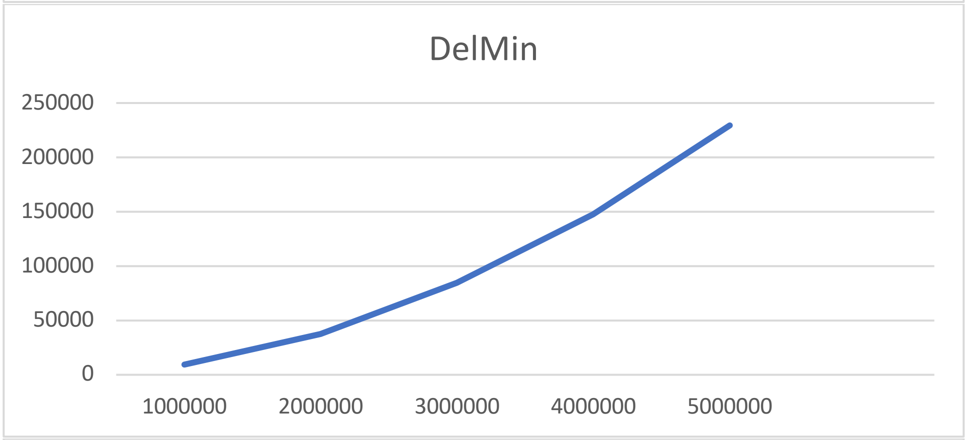


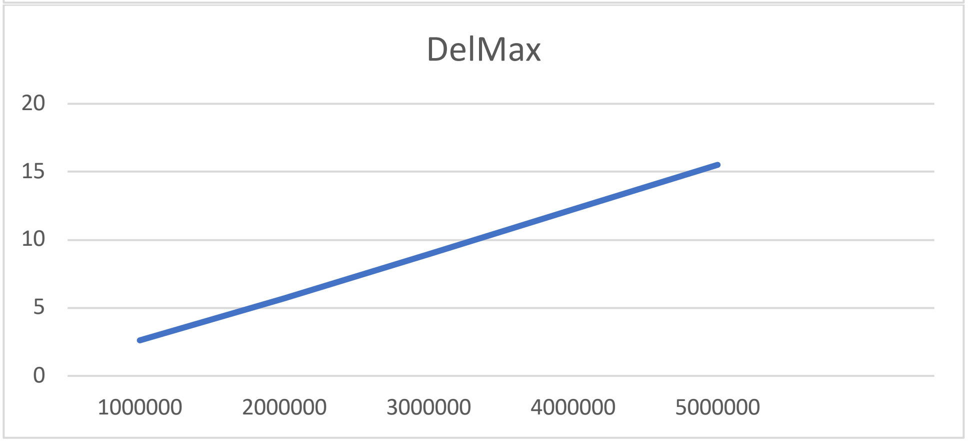


Max 5-Heap

# elements	**1000000**	**2000000**	**3000000**	**4000000**	**5000000**
Build	498.296	1010.1	1491.84	1998.74	2467.13
DelMin	9383.02	37348.6	84401.6	147510	229410
DelMax	2.6114	5.6842	8.9626	12.2608	15.5138







Let's find the complexity for each of these operations.

For a binary search tree, we can expect that for a random sample of data of large enough size as used in this lab, the tree will be roughly balanced. Therefore the tree will have approximately log(n) levels, where n is the number of items in the tree, so for an insert operation which will make one comparison per level, the complexity will be log(n).

Doing m inserts will then require $C * \sum_{i=1}^{m} \log{(i)}$ time. We can find the big-O complexity of this operation as such:

$$C * \sum_{i=1}^{m} \log(i) = Clog(m!) = O(nlog(n))$$

This observation seems to agree with the observed data, although it is certainly not proven without a doubt by the data. Fitting a linear trendline to the data yields an R^2 value of 0.9979, while fitting a nlog(n) trendline gives an R^2 value of 0.9989—only slightly better. We would need many more data points to conclusively show that the complexity of this operation is nlog(n).

The deleteMin and deleteMax operation on a binary search tree will have the same complexity, since they delete the leftmost and rightmost nodes of the tree, respectively, and are therefore just mirrored operations across the y-axis. A single deleteMin/Max operation will have a complexity of log(n), making the same assumptions as for insertion. Doing 0.001m deletions from a tree with m elements will then require $C * \sum_{i=0.999m}^{m} \log{(i)}$ time. We can find the big-O complexity of this operation as such:

$$C * \sum_{i=0.999m}^{m} \log(i) = C\big(\log(m!) - \log\big((0.999m)!\big)\big) = Clog\left(\frac{m!}{(0.999m)!}\right) = O\big(nlog(n)\big)$$

Again, there is not really enough data to conclude that this expected complexity is supported by the experimental data or not. The trendlines for both deleteMin and deleteMax all have an R^2 greater than or equal to 0.9997 for both linear and nlog(n) fits.

Min and max heaps have identical logic—their only difference is the replacement of a > with a <. Therefore their build complexities will be the same, and deleteMin for one type of heap will have the same complexity as deleteMax for the other type.

For a heap with n elements, an insert operation has complexity log(n), since a heap has log(n) layers and a newly inserted element will on average be pushed up log(n)/2 layers. We can find the big-O complexity of doing m inserts as follows:

$$C * \sum_{i=1}^{m} \log(i) = Clog(m!) = O(nlog(n))$$

We can see that heaps have the same build complexity as binary search trees. Again, the trendlines for the build operations have R^2 values very close to 1 for both linear and nlog(n) fits, so no conclusions can be drawn from the data without more data points.

DeleteMin in a min-heap or deleteMax in a max-heap is also a log(n) operation, since the element replacing the root will on average be pushed down log(n)/2 layers. Thus the big-O complexity for deleting 0.001m elements is:

$$C * \sum_{i=0.999m}^{m} \log(i) = C\big(\log(m!) - \log((0.999m)!)\big) = Clog\left(\frac{m!}{(0.999m)!}\right) = O\big(nlog(n)\big)$$

Again, the R^2 values are virtually identical and very close to 1 for both linear and nlog(n) fits, and no conclusions can really be drawn from the data.

DeleteMax in a min-heap or deleteMin in a max-heap are more interesting cases. First the min/max element must be found, then it must be removed from the heap. The min/max element will always be in a leaf node, which a heap with n elements has n/2 of. Searching these elements for the min/max requires n/2 comparisons; therefore the complexity is O(n). Actually deleting the element still has log(n) complexity, which will be drowned out by the O(n) search, so the overall complexity is O(n).

Deleting 0.001m elements will therefore have complexity:

$$C * \sum_{i=0.999m}^{m} i = C(0.0009995m^2 + 0.9995m) = O(n^2)$$

This conclusion is strongly supported by the data. The plots for deleteMin in a max-heap and deleteMax in a min-heap both have an $R^2 = 1$ with a quadratic fit.

Conclusion

This lab usefully demonstrated some of the strong suits and shortcomings of binary search trees and heaps. Both of these data structures support insertion in O(log(n)) time, but binary search trees can delete both their minimum and maximum in O(log(n)) time, while heaps can only delete one of the two, the other taking O(n) time. Of course, the most important benefit of heaps was not tested in this lab: min-heaps can find their minimum and max-heaps can find their maximum in O(1) time, while binary search trees take O(log(n)) time to do either.

There were a few conclusions that could be drawn from this lab. The expected time complexity for building either of the data structures in O(nlog(n)) time was not confirmed or refuted; neither were deleteMin for BSTs and min-heaps or deleteMax for BSTs and max-heaps, also in O(nlog(n)) time. The main reason for this was too small of sample size; the graphs for O(n) and O(nlog(n)) are so similar for only 5 data points that it was not possible to draw conclusions. However, the poor performance of deleteMin in max-heaps and deleteMax in min-heaps in O(n^2) time was experimentally verified successfully. Overall, this lab was a success, but it could have been improved by running more tests.