Roman Clark

U0944898

April 14, 2017

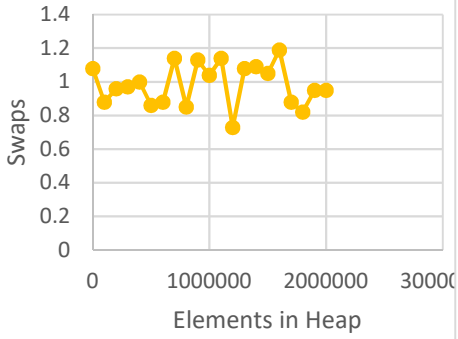
CS2420

Assignment 11 – Heaps

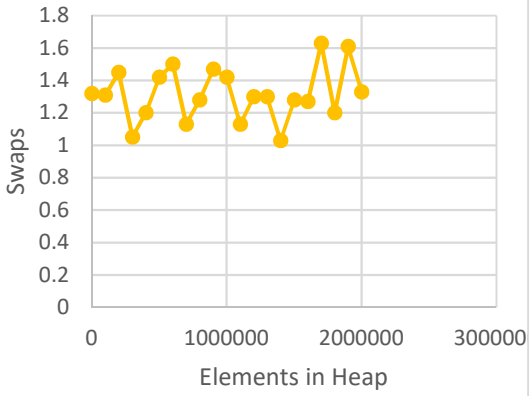
Partner: Mark Van der Merwe

The binary heap is an implementation of a priority queue. Each node in this min heap is smaller than its children. Heaps are complete, as every level of the tree is filled, except possibly the bottom row. This means that roughly half of the data, and exactly half of the data if the bottom row is filled, is found in the bottom row. This makes most operations more efficient on average than the worst-case scenario, because as we will prove in our timing analysis, you do not have to traverse through the entire tree with the majority of the data being closer to the bottom, at the point of initial insertion (Half of the data being in the bottom row, one-fourth of the data in the second-lowest row, and so on). Priority queues like binary heaps are useful in any situations where items in a list have different weights, and thus the highest priority item should be reported first (whether that's the greatest or smallest value). For example, in an emergency room, the degree of emergency would determine the weight of each patient waiting, and the patient with the most extreme emergency should be seen first (and would be at the top of the binary heap). A concrete use of the priority queue in computer science is in the upcoming assignment of Huffman encoding, where the higher frequency of each character in the string gives it higher priority, therefore creating shorter strings of bits for that character.

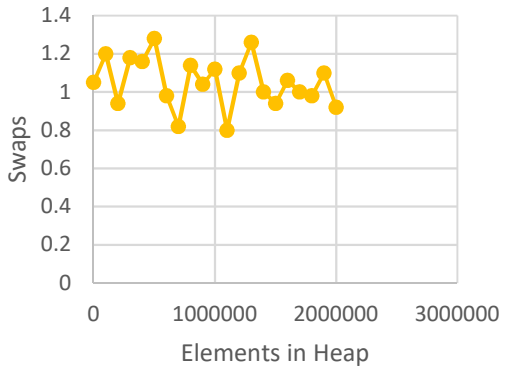
Inserting One Element into A Heap of N Sorted Elements - Swaps



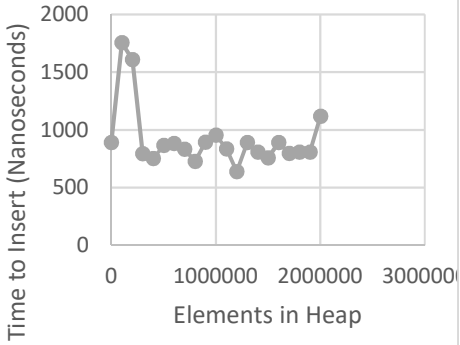
Inserting One Element from N Size Heap - Swaps



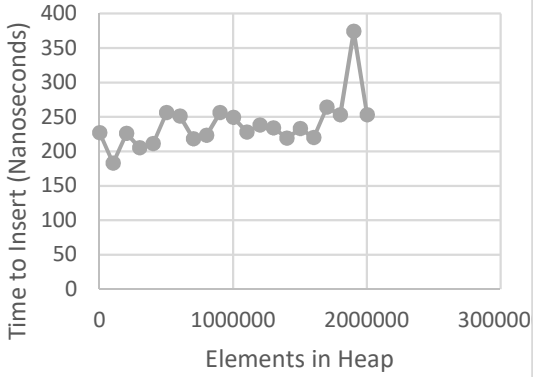
Inserting One Element into a Heap of N Elements - Swaps



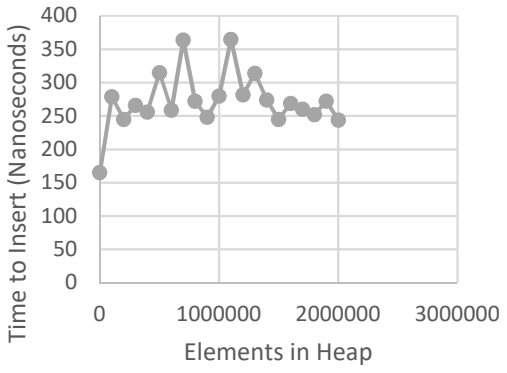
Inserting One Element into A Heap of N Sorted Elements - Time



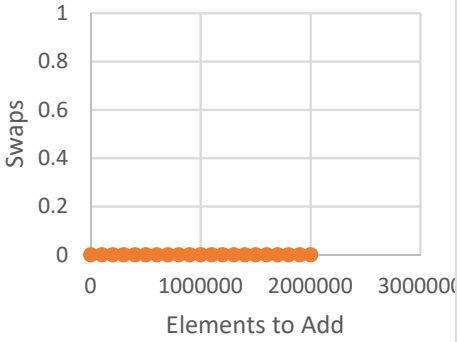
Inserting One Element from N Size Heap - Time



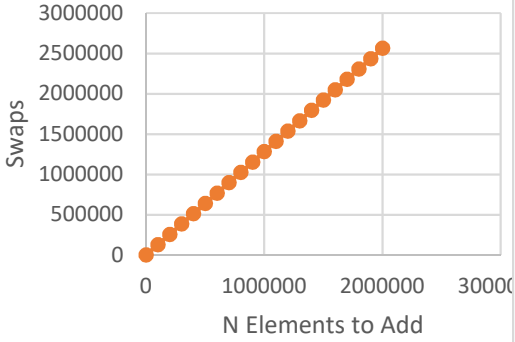
Inserting One Element into A Heap of N Elements - Time



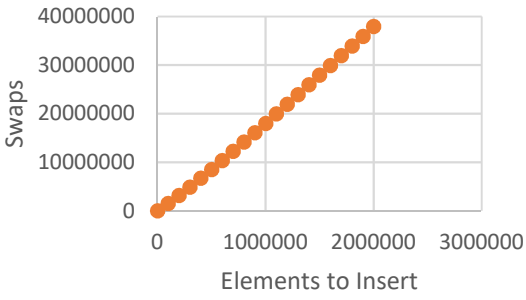
Adding N Sorted Elements to an Empty Heap - Swaps



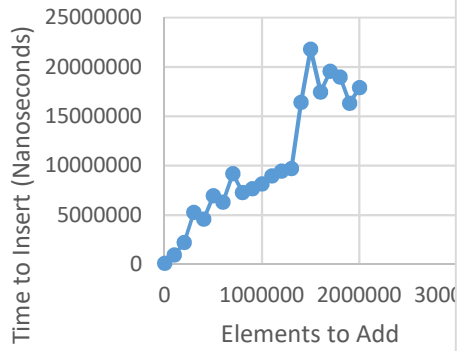
Inserting N Random Data Items into Empty Heap - Swaps



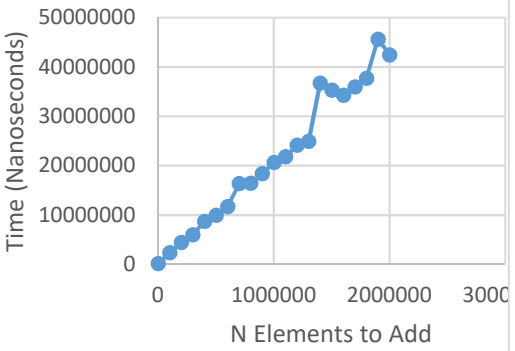
Inserting N Back Sorted Elements into an Empty Heap - Swaps



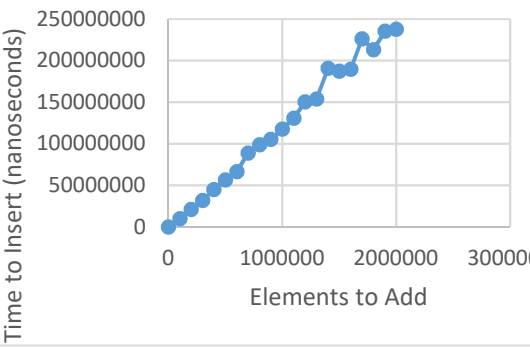
Adding N Sorted Elements to an Empty Heap - Time

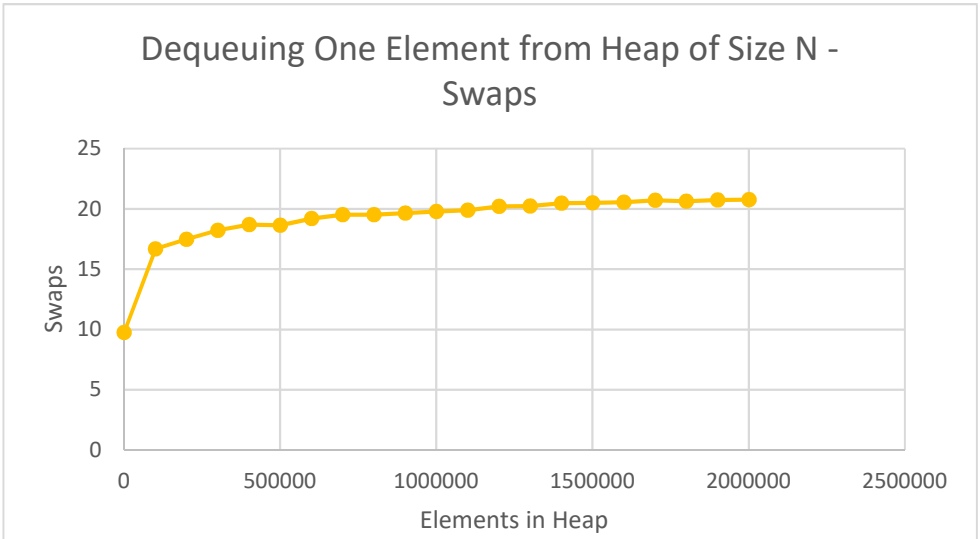


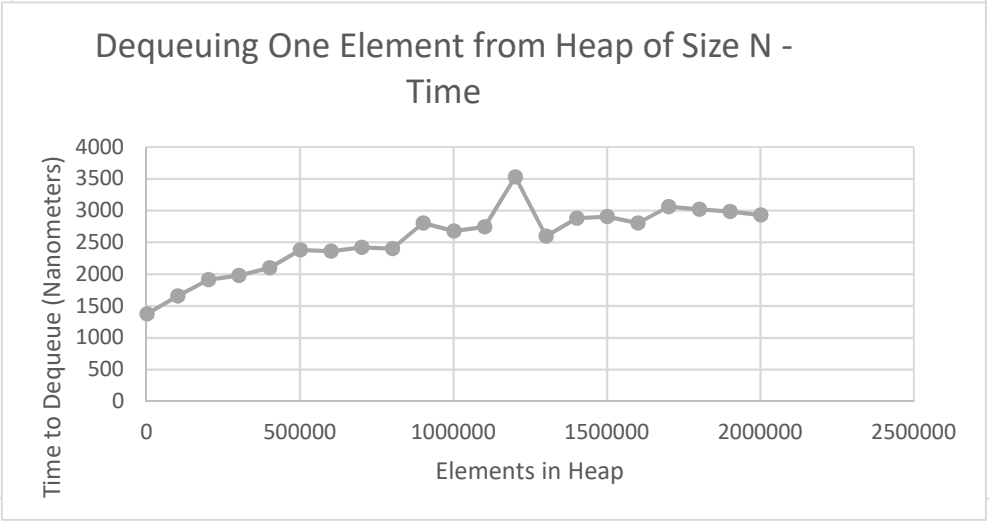
Inserting N Random Data Items into Empty Heap - Time

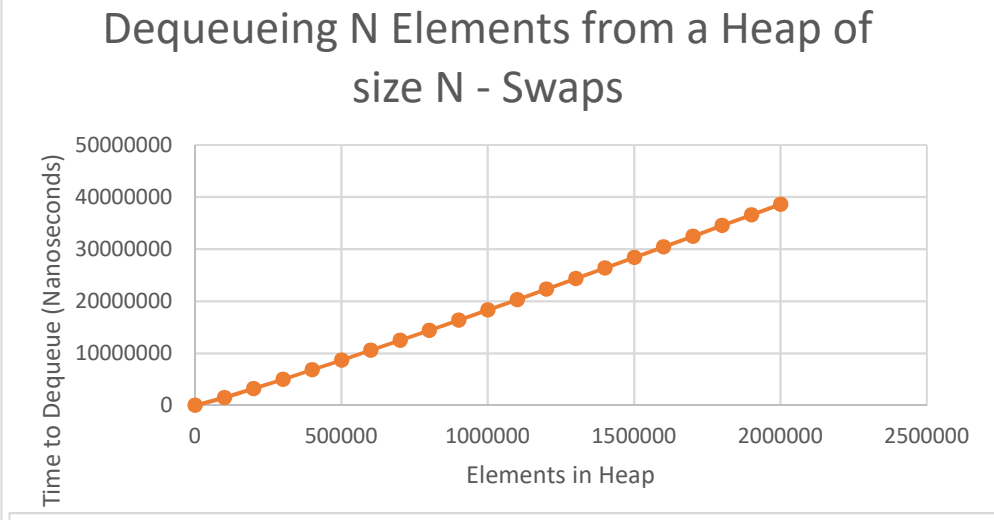


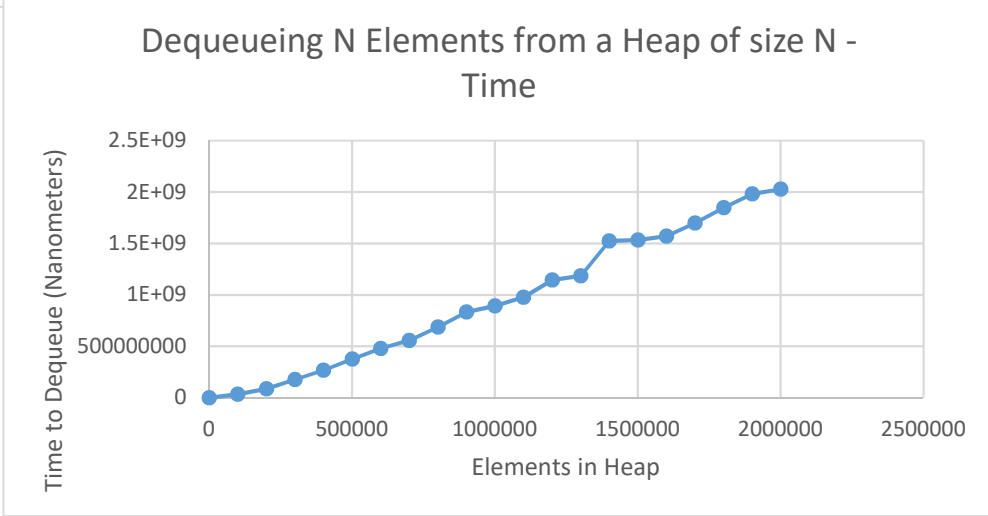
Inserting N Back Sorted Elements into an Empty Heap - Time

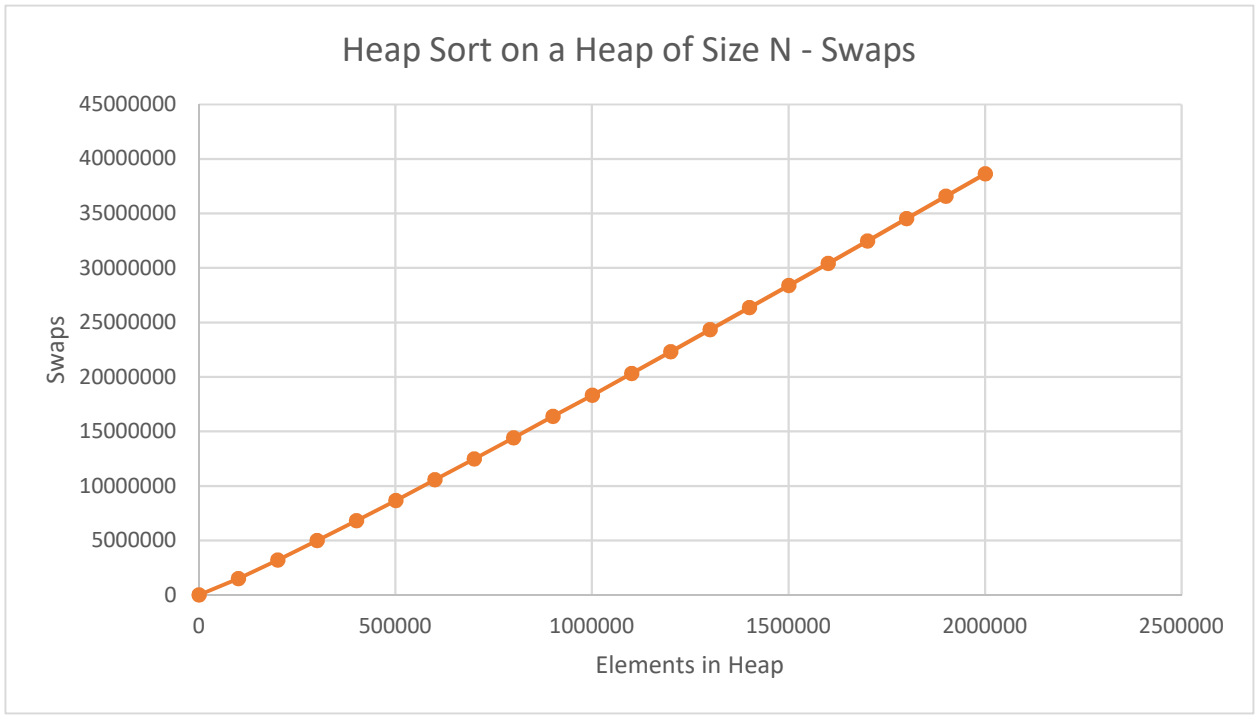


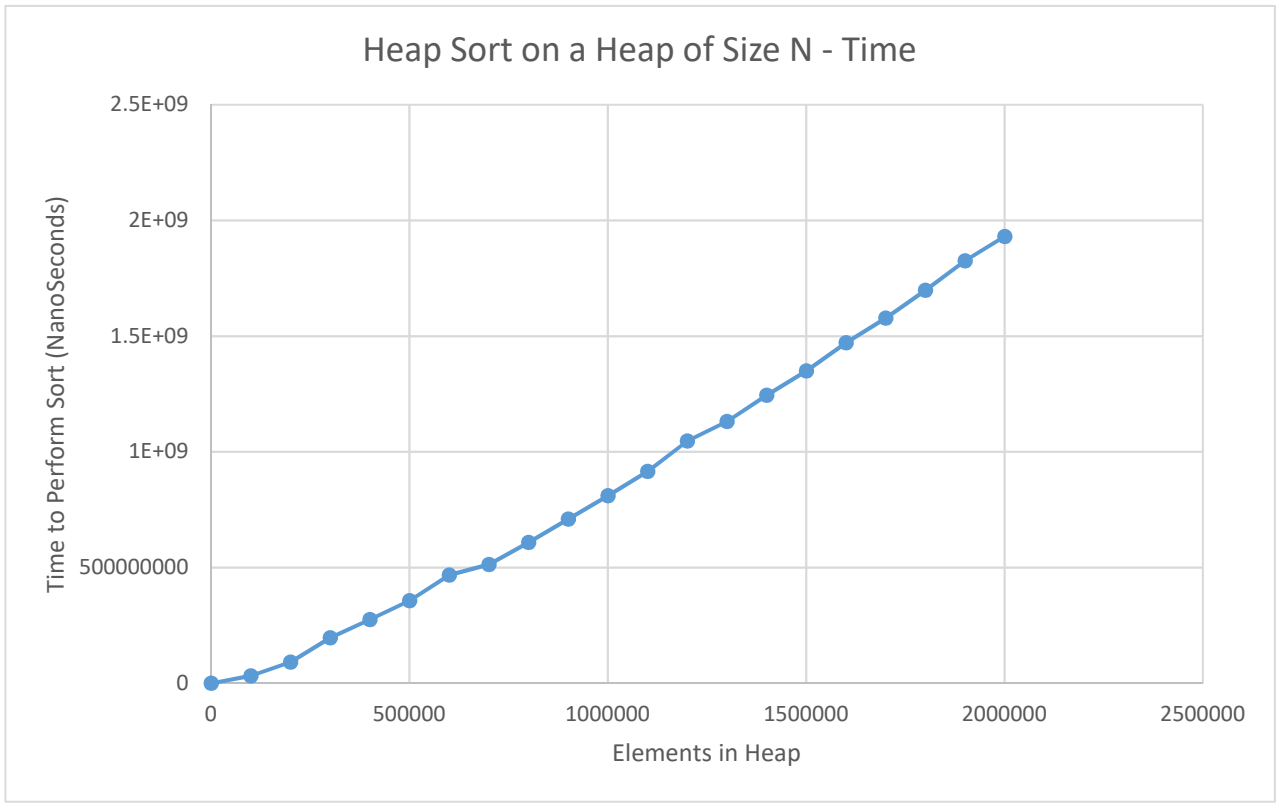

Dequeuing One Element from Heap of Size N - Swaps
Swaps
25
20
15
10
5
0
0
500000
1000000
1500000
2000000
2500000
Elements in Heap


Dequeuing One Element from Heap of Size N - Time
Time to Dequeue (Nanometers)
4000
3500
3000
2500
2000
1500
1000
500
0
0
500000
1000000
1500000
2000000
2500000
Elements in Heap

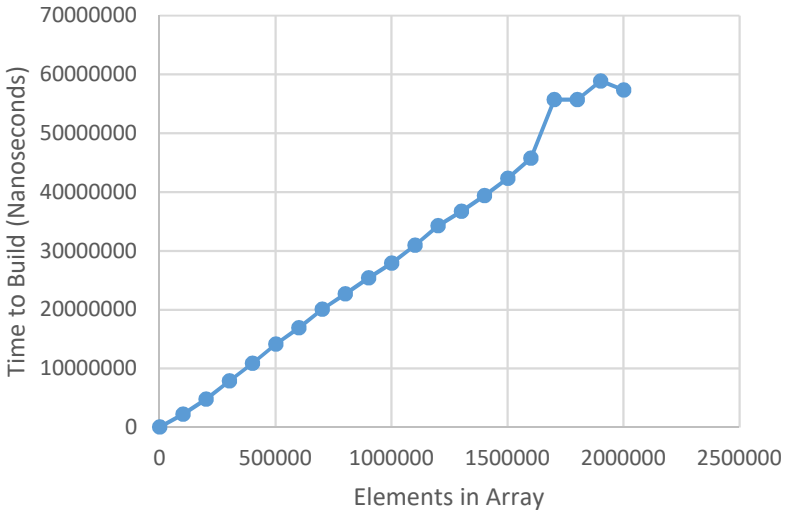

Dequeueing N Elements from a Heap of size N - Swaps
Time to Dequeue (Nanoseconds)
50000000
40000000
30000000
20000000
10000000
0
0
500000
1000000
1500000
2000000
2500000
Elements in Heap


Dequeueing N Elements from a Heap of size N - Time
Time to Dequeue (Nanometers)
2.5E+09
2E+09
1.5E+09
1E+09
500000000
0
0
500000
1000000
1500000
2000000
2500000
Elements in Heap

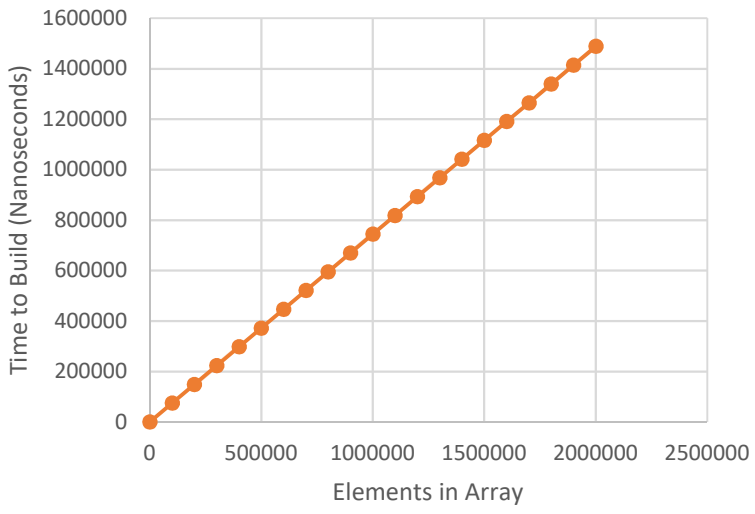

Heap Sort on a Heap of Size N - Swaps
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Swaps
0
500000
1000000
1500000
2000000
2500000
Elements in Heap


Heap Sort on a Heap of Size N - Time
2.5E+09
2E+09
1.5E+09
1E+09
500000000
0
Time to Perform Sort (NanoSeconds)
0
500000
1000000
1500000
2000000
2500000
Elements in Heap

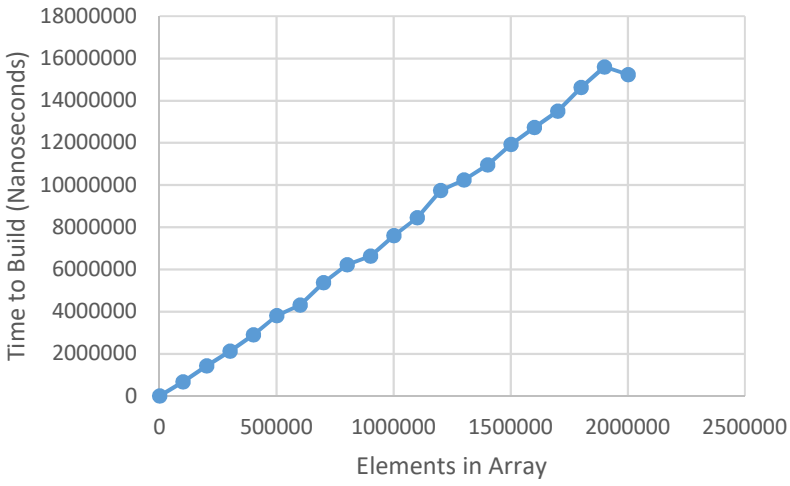
Building Heap from Array of N Random Values - Time



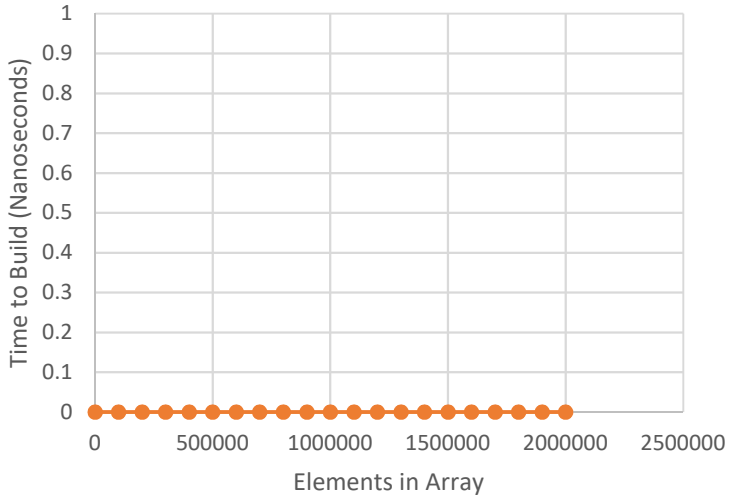
Building Heap from Array of N Random Values - Swaps



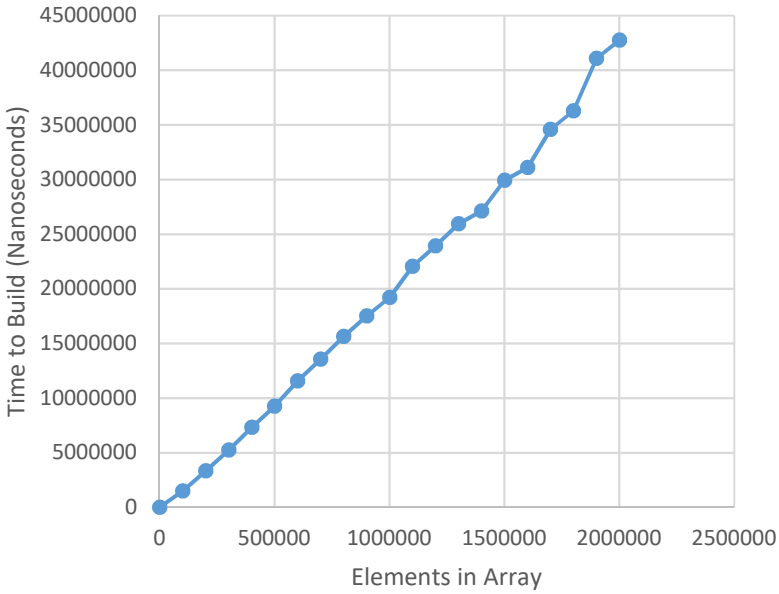
Building a Heap from an Array of N In Order Elements - Time



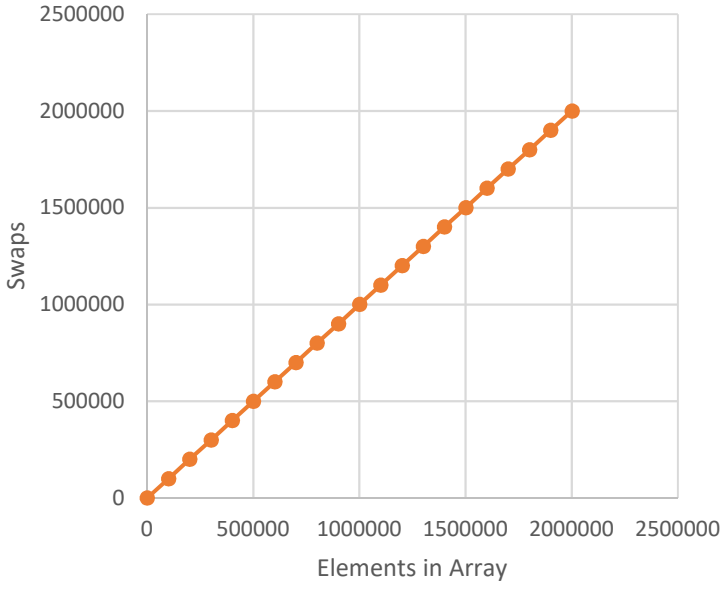
Building a Heap from an Array of N In Order Elements - Swap



Build Heap from an Array of N Back Ordered Elements - Time



Build Heap from an Array of N Back Ordered Elements - Swaps



Timing Experiment

We did extensive timing analysis of the Heap algorithm. Every test considered Heaps of 1,000 elements to 2,000,000 elements at increments of 100,000 for 20 data points. At each increment of N, the test was done 100 times and the average results were taken. For all experiments, the objects in the Heaps were Integers. As described in the following analysis, we timed performance of insertion, dequeuing, heap sort, and building heap from arrays for a variety of data.

For the insertion algorithm, we tested 4 variables for 3 types of data being inserted into a Heap, for a total of 12 graphs, seen on the first page of graphs. Data being inserted was either inserted in order (smallest to greatest), randomly generated, or inserted in reverse order (greatest to smallest). The first two tests found the time it took to insert one element into a heap of size N, and the number of position swaps done in that insertion. Although for each data type this fluctuated for varying N, both the time taken and the number of swaps were relatively constant. This makes sense because we took the average time of 100 insertions, so even though the complexity of one insertion is O(log N), since most of the data is on the bottom row, only 1-2 swaps were necessary, so the cost of one insertion is constant on average, evidently at about 250 nanoseconds. The next test was adding N elements to an initially empty heap and keeping track of time taken, and the number of swaps necessary to do so. If the data is already in order, you can see on the graph that no swaps are necessary, because it is automatically placed in the correct place, so it was also the fastest of the three datatypes, and of course had O(N) complexity, since the cost was just placing N elements into an array. As previously discussed, the average cost of one insertion is constant for random data, so the expected average complexity of inserting N random elements is O(N), and the general linearity of the time taken to insert N random elements supports this. For random data, the time taken per insertion was about 20 nanoseconds, and the number of swaps per insertion was almost exactly 1.28 for every value of N, meaning the constant for insertion is ~1.28*2=2.56 (# of swaps multiplied by 2 due to relocation of 2 different elements), which matches the expected bubbling up complexity of O(2.6). Although the reverse sorted graph looks similar, the time taken and number of swaps increases. You can also see a slight curve in the number of swaps graph, and this is because it is no longer O(N), but instead O(N log N). Because reverse sorted data means having to percolate up through the entire tree, a height of log N, rather than just one or two levels of tree.

On the next page of graphs, we tested the performance of dequeuing one element from a Heap of size N, and then dequeuing N elements from a heap of size N. The expected complexity of dequeuing an element is O(log N), as it is expected that after replacing the root of the Heap with an element at the bottom of the Heap, this new root will have to percolate down the entire tree to get in the correct position, a height of log N. Our tests support this, as the number of swaps and the time taken to dequeue one element follows a log N pattern, and the time and swaps for dequeuing N elements follows N log N. After dividing the average number of swaps by N log N, we get a constant work of about .92 swaps, so the constant for dequeuing is ~.92*2=1.84. (multiplied by N log N for total complexity).

The next timing analysis was on Heap Sort. Because the algorithm of Heap Sort is to simply dequeue all elements in the Heap, we would expect that a Heap Sort follows a complexity of O(N log N), ignoring the cost of initially building the heap. Again, our data supports this expectation. Although the graphs for the time taken and the number of swaps look mostly linear, there are again slight curves suggesting N log N behavior. We can divide the average number of swaps per N and divide by log N to

get a constant work of .92, for a constant of 1.84 again, which as expected, is the same constant of work as dequeuing, since we are essentially dequeuing N elements.

On the fourth and final experiment, we compared the performance of building a heap from an array for three types of arrays: random data, sorted data, and reverse sorted data. The algorithm for building a heap from an array requires starting halfway up the tree (second to lowest row) and percolating every element down. Since the amount of data decreases by a factor of 2 every time you move up the heap, the further an element would potentially have to percolate down, the fewer elements are at that level, so we have an average case scenario of O(N), with few swaps necessary for each element. With reverse sorted data, we still get O(N) behavior, but the number of average swaps increases, and of course, sorted data would require 0 swaps, and would perform the fastest. Our data backs this, as the time for building a heap for all arrays was linear, with .74 * N swaps done on average for random arrays, N swaps done for reverse sorted arrays (the worst-case scenario), and 0 swaps for the sorted array. As expected, sorted array was the fastest, and reverse sorted array was the slowest.

Software Engineering

The assignment was one of the shorter ones, although our timing analysis took some time to organize. The actual coding portion couldn't have taken more than 4 or 5 hours. We did not run into significant problems.

The compare method allows the user to define the parameters on which two elements are compared in creating a heap. If the user does define a comparator, likely for non-Comparable types, the Heap class will use this given comparator to define two elements to determine which has higher priority. Otherwise, if comparator is left null, we can assume the Type in the Heap is a Comparable, so we use the compareTo method already defined by this Comparable Type's comparator.

Java doesn't allow generic arrays, but we want to make the Heap generic. To do this, we create an array of Objects, and then cast it with "Type[]". This cast is an unchecked cast to a generic, so to remove this warning, we use the "@SuppressWarnings("unchecked"), as this is intentional.

The best string representation of the Heap is the same info as the DOT file, and writing this in the toString method allows us to use the string output in the console, and then simply call the method in the generateDotFile method. If it was the other way around, we couldn't call the generateDotFile method in the toString method without creating a file. Additionally, if we want to quickly check or test the creation of the Heap by just looking at the console output rather than creating a file, it is better to have this output in the toString method.

The "test_lots_of_insertions_deletions_peeks" is a powerful testing tool because it allows you to tests many cases of the Heap. Starting from an empty array, it tests the insertion into an empty Heap, tests insertion of a single value Heap, and tests insertion on a Heap of many values. It also ensures that the Heap is automatically resizing and that this resizing process does not mess up future insertions. On top of this, the loop created for dequeuing allows you to check that at every dequeue that you're actually getting the minimum element in the heap, meaning dequeue is working correctly, and like insertion, tests multiple cases of Heaps, like a Heap with many elements, a Heap with a few elements, and a single element Heap.