Nickolas Komarnitsky u0717854

04/13/2017

CS2420 Assignment-11-Heaps

A heap is a balanced tree that has very quick operations. At best case it is O(N), worst is only log(n). Building a heap will always run at log(N) since it has to go through many more operations the bigger the input array. A priority queue can be used abstractly at an amusement park, club, etc. They have people without the special pass, who have the same priority, then those with the special pass who have a higher priority. In a computer system they are concretely used because they are based purely on the number that is in the system. They rely only on the numbers, the “bouncer” can’t just let someone in ahead of another. In a computer system, they enter at the back and then based on that priority they move ahead of the others and end up exactly where they are. There is no line skipping.

1. Timing Experiment

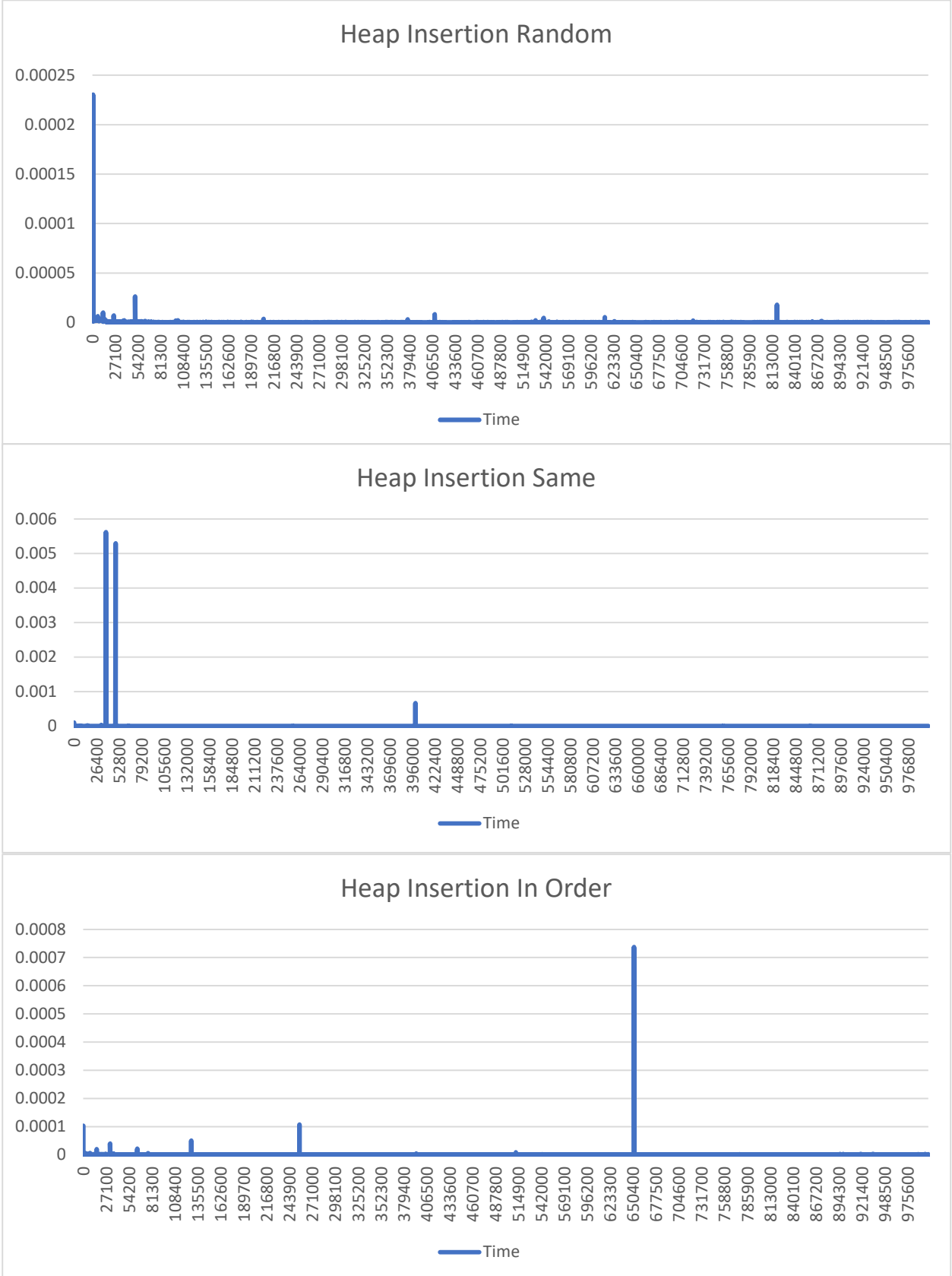
For my Timing Experiment I am trying to determine the runtime complexity of a min heap. The Mathematical analysis tells me that the heap should have an average complexity of O(N), with a worst-case complexity of log(N). The build from array function I expect to be log(N), taking much longer for a bigger given array.

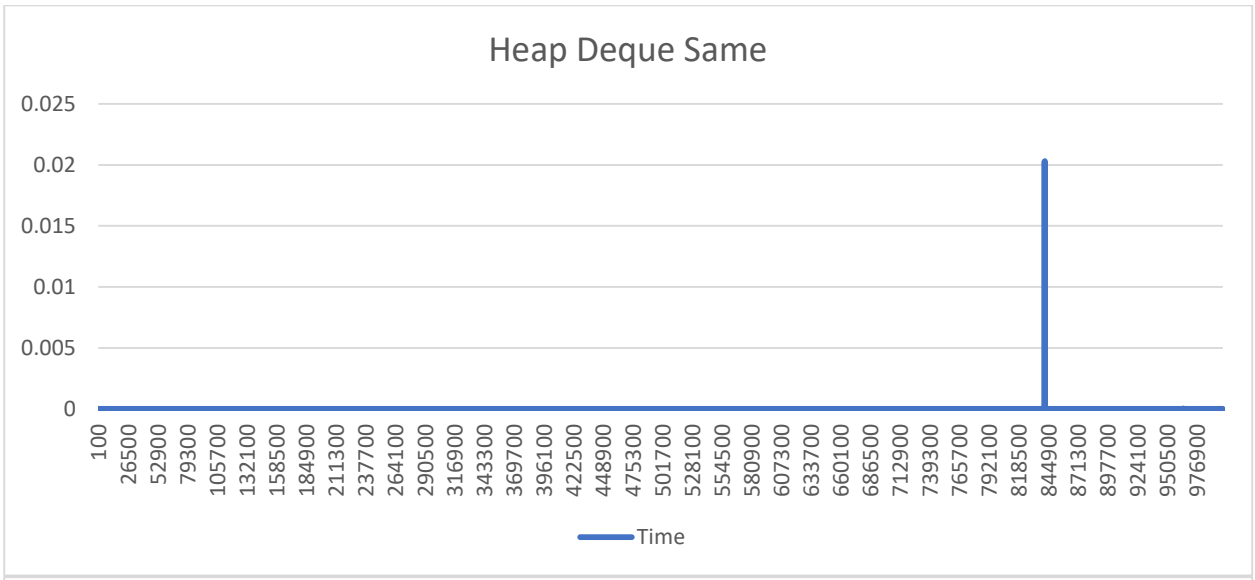
My testing was setup to determine as many cases as I could. My first test was on the insertion. My first test was to add items into an array in order from 1-1000000 incremented by 100, timing the add function each time. The second test was again to a size of 1000000, but adding in the same number each time. The third was to a size of 1000000 as well, but adding in random integers from 1 to 1000000.

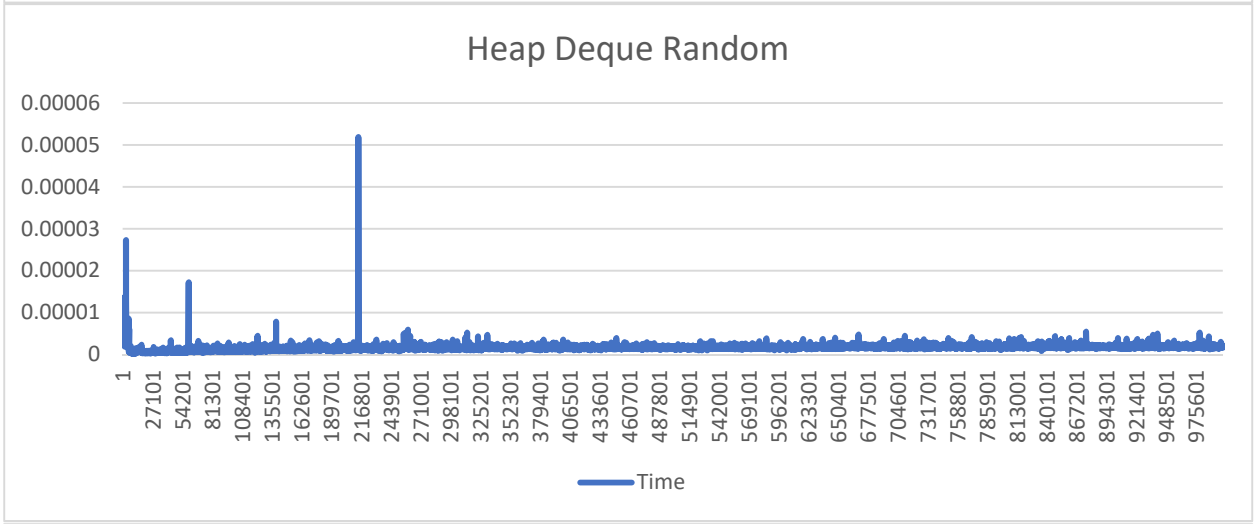
Then I tested the deque function. For the first test, I created an heap of size 1000000, with the items inserted in order, then timed the deque function until there was only 1 element left in the heap. The second test was with the same, but the same item was inserted into the heap beforehand. The final test I tested deque on an heap that had items inserted with random values. For this test, I generated an heap of the desired size with random integers being put in. For each size, I ran this 5 times, to make up for the randomness.

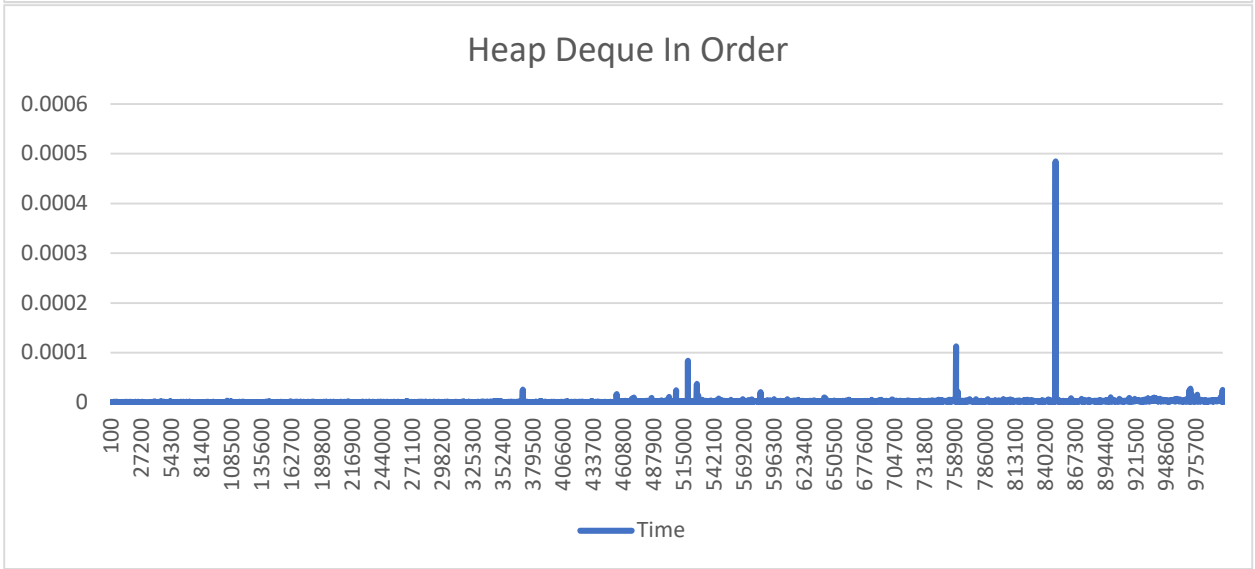
Then I got the heap sort and build from array done. I ran a test on build from array, with the given array being in order first. The second test was with the array having the same number in all spots. The final was an array with the numbers randomized. For all of these I timed the method.

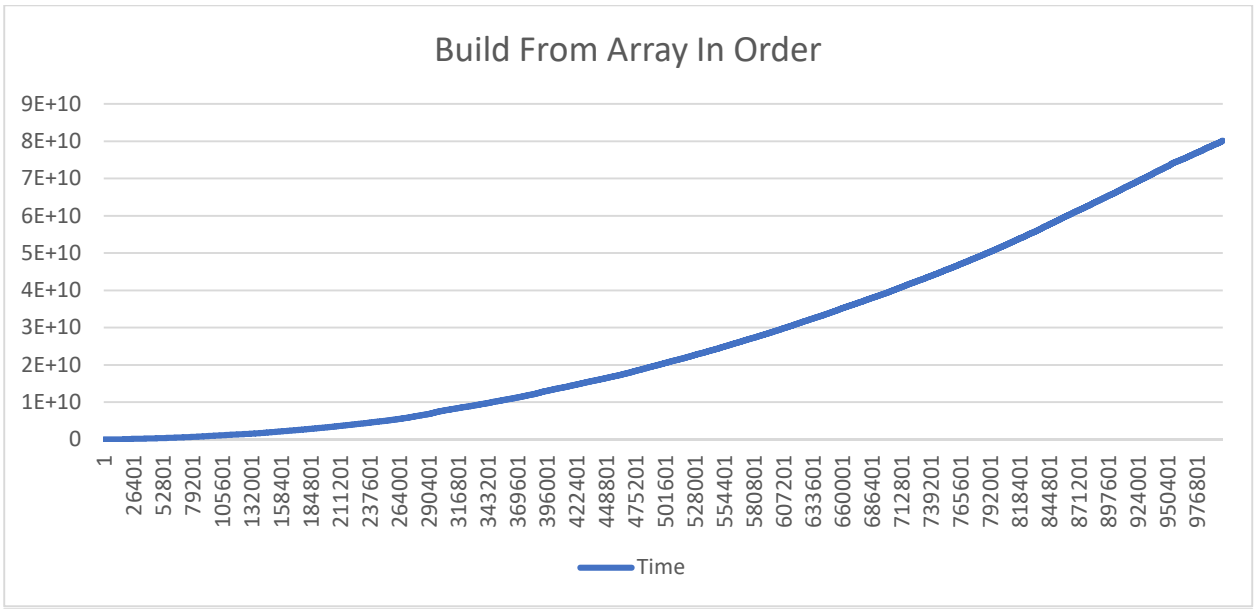
The graphs below show time to size, with the size on the x-axis and time on the y-axis. For the insertion tests, almost all of them came out with a constant time, with a few outliers in the data. They showed a run time of O(c) for all insertions. The deque was the same, taking O(c) for each test. The Build from array function was log(N) for each test, with it taking longer the larger the given array.

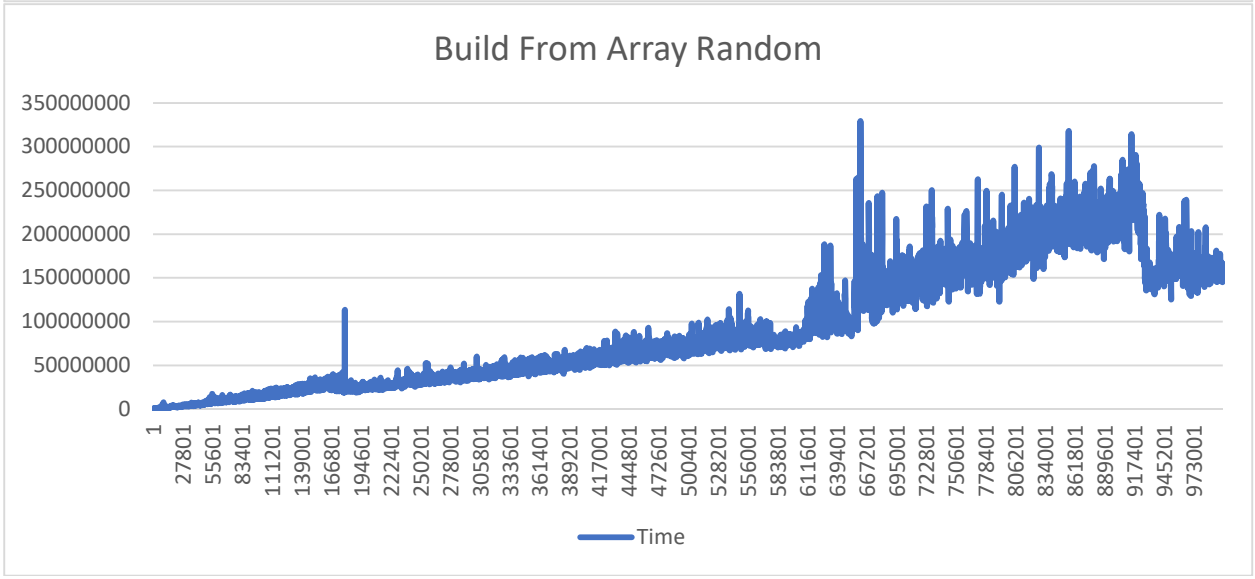

Heap Insertion Random
0.00025
0.0002
0.00015
0.0001
0.00005
0
0
27100
54200
81300
108400
135500
162600
189700
216800
243900
271000
298100
325200
352300
379400
406500
433600
460700
487800
514900
542000
569100
596200
623300
650400
677500
704600
731700
758800
785900
813000
840100
867200
894300
921400
948500
975600
Time
Heap Insertion Same
0.006
0.005
0.004
0.003
0.002
0.001
0
0
26400
52800
79200
105600
132000
158400
184800
211200
237600
264000
290400
316800
343200
369600
396000
422400
448800
475200
501600
528000
554400
580800
607200
633600
660000
686400
712800
739200
765600
792000
818400
844800
871200
897600
924000
950400
976800
Time
Heap Insertion In Order
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
0
27100
54200
81300
108400
135500
162600
189700
216800
243900
271000
298100
325200
352300
379400
406500
433600
460700
487800
514900
542000
569100
596200
623300
650400
677500
704600
731700
758800
785900
813000
840100
867200
894300
921400
948500
975600
Time

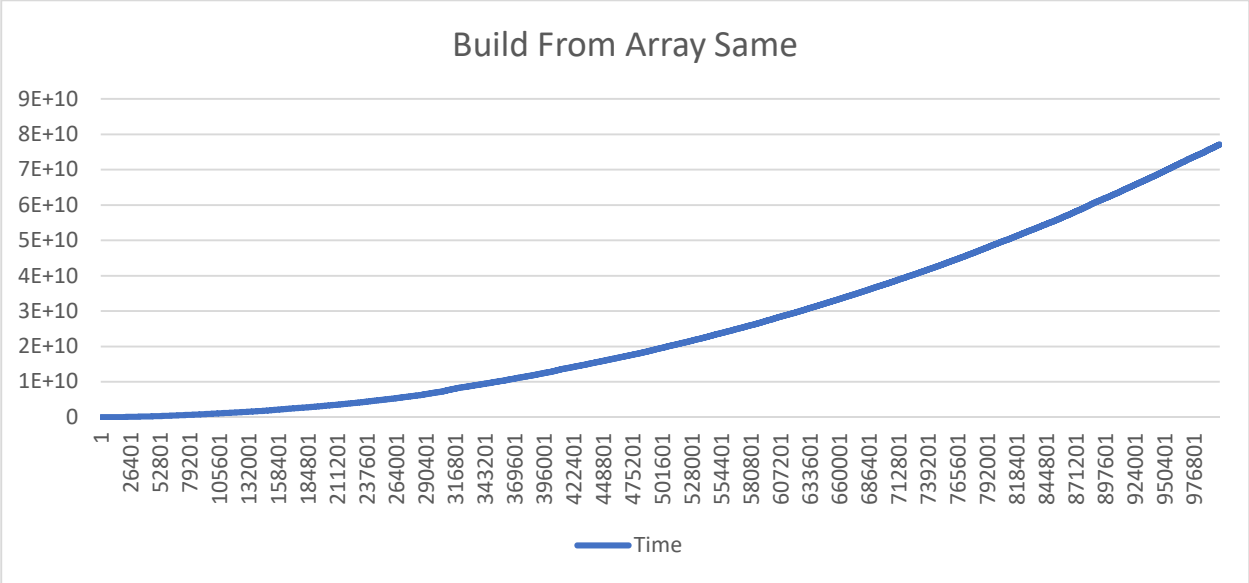

Heap Deque Same
0.025
0.02
0.015
0.01
0.005
0
100
26500
52900
79300
105700
132100
158500
184900
211300
237700
264100
290500
316900
343300
369700
396100
422500
448900
475300
501700
528100
554500
580900
607300
633700
660100
686500
712900
739300
765700
792100
818500
844900
871300
897700
924100
950500
976900
Time


Heap Deque Random
0.00006
0.00005
0.00004
0.00003
0.00002
0.00001
0
1
27101
54201
81301
108401
135501
162601
189701
216801
243901
271001
298101
325201
352301
379401
406501
433601
460701
487801
514901
542001
569101
596201
623301
650401
677501
704601
731701
758801
785901
813001
840101
867201
894301
921401
948501
975601
Time


Heap Deque In Order
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
100
27200
54300
81400
108500
135600
162700
189800
216900
244000
271100
298200
325300
352400
379500
406600
433700
460800
487900
515000
542100
569200
596300
623400
650500
677600
704700
731800
758900
786000
813100
840200
867300
894400
921500
948600
975700
Time


Build From Array In Order
9E+10
8E+10
7E+10
6E+10
5E+10
4E+10
3E+10
2E+10
1E+10
0
1
26401
52801
79201
105601
132001
158401
184801
211201
237601
264001
290401
316801
343201
369601
396001
422401
448801
475201
501601
528001
554401
580801
607201
633601
660001
686401
712801
739201
765601
792001
818401
844801
871201
897601
924001
950401
976801
Time


Build From Array Random
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
1
27801
55601
83401
111201
139001
166801
194601
222401
250201
278001
305801
333601
361401
389201
417001
444801
472601
500401
528201
556001
583801
611601
639401
667201
695001
722801
750601
778401
806201
834001
861801
889601
917401
945201
973001
Time



2. Software Engineering

This assignment took me about 10 hours, although a lot of that was just writing the timing tests. Non-heap wise, I figured out multithreading in javaFX much better and even got a decent GUI made that can show exactly how far along my tests are. This GUI can also have as many tests as needed added into it. The compare method is to determine what child is smaller. This allows for you to move an item up the heap as needed. The heap returns an array of Objects instead of Type because it is safer, as object has more functionality than type does. We use SupressWarnings because the editor will highlight warnings on type. We generate it in the toString method because it creates the heap, not the array so having it be made in a dot readable style makes the most sense. The test_lots_of_insertions_deletions_peeks is a powerfull tool because it allows you to make sure that your code will accept a lot of data inserted and removed and that this is occurring properly.