Input size	100,000	200,000	300,000	400,000	500,000
BuildHeap	0.0176	0.0217	0.0338	0.0442	0.0559
FindMin	0.00022	0.00044	0.00066	0.00087	0.0011
FindMax	0.0022	0.0043	0.0065	0.0085	0.011
DeleteMin	0.0102	0.0185	0.0276	0.04	0.0510
DeleteMax	0.0137	0.0285	0.0412	0.0527	0.0623

Q. Observe your output for the min-max heap operations for the build, find (min and max) and delete (min and max) operations.

a. For each operation, compare the worst case complexities of your experimental profiling to the worst case complexity of the theoretical results.

- BuildHeap - my results are just slightly superlinear, which makes sense because using a top-down insert approach it is theoretically O(nlogn). My inserts are O(logn) and during a build we insert n times, so O(nlogn) makes perfect sense and the growth of the times reflects that.
- FindMin and FindMax - the growth in these experimental times scales precisely with n. The FindMin and FindMax are constant (O(1)), regardless of heap size, and when we do it n times we see a linear growth in these experimental times. If we were to divide our times above by the input size, we'd get a roughly constant time for each size.
- DeleteMin and DeleteMax - these are theoretically O(logn) because we remove the root (or the max, which is one of the root's children) which is O(1) and do an O(logn) insert to replace the node we just removed. We see this growth is also slightly superlinear.

b. Justify the worst case complexities of each of your experimental profiling results.

- O(nlogn) top-down builds makes sense. Again, we are doing O(n) inserts which are O(logn) each.
- O(1) FindMin and FindMax - we have to do 1 transform on these experimental results, which is normalizing by the input size. I wrapped the clock timer around the loops which had FindMin and FindMax inside it, which is why we see the time slowly increasing linearly. If we divide by input size, in both cases, we see constant times.
- O(nlogn) DeleteMins and DeleteMaxs - the slight growth here makes sense. We are doing an O(1) delete and a potentially O(logn) re-insert. The bigger the tree, the larger the O(logn) insert.