

The first piece of data that was tested was using the buildHeap() method with sorted data. Using buildHeap() with sorted data had the lowest average runtime out of all of the other tests ran, which was 41.82 milliseconds. Keep in mind that heaps have an order property, so this is undoubtedly due to the fact that the data was already sorted which means the heap does not have to sort the data while it is being entered. Depending on the number of items, this could become costly. The range of numbers entered were between 40,000-10,000,000. While running the initial tests I found that using too low of numbers resulted in the time being 0 which is essentially useless when comparing runtimes. 40,000 provided a good starting point for consistently getting numbers greater than 0 to average out and compare. Also, it is important to note that the program was ran a total of ten times. It was decided that the program needed to be ran a total of 10 times to gather a substantial amount of data to be compared and analyzed. Referring to the table, you can see that the higher the input gets the higher the average runtimes become because buildHeap() has an O(n) runtime.



The next piece of data that was tested was using the insert() method with sorted data. The total average runtime was 166.275 milliseconds. So this method was obviously considerably slower than buildHeap() which will remain consistent for all data types throughout the analysis. This is due to the cost of having to loop through the data and insert one by one which is a lot more costly than the buildHeap() method. For the most part as the number of items being added to the heap becomes larger so do the average runtimes.



The next piece of data analyzed is using the buildHeap() method with reversed data. This will become a little more costly due to the ordered property of heaps because now the data needs to be sorted. The total average runtime was 93.42. This is a little higher than the sorted data average runtime of 41.82 but not much. This is still pretty efficient. The trend of the average runtimes for each input growing larger as the input grows larger continues to be present.



When comparing insert() with buildHeap() with reversed data we really see a huge jump in average runtime. The total average runtime is 616.66 milliseconds compared to buildHeap()’s 93.42. The insert() method is now doing even more additional steps to sort the data which slows this method down severely. It is important to note that when the input size is low there is not a huge difference between insert() and buildHeap() but when the input ranges between 1,000,000 and 10,000,000 there is a more significant difference. So when using a small input the difference between the two would probably be unrecognizable. On the other hand, when using a large input like 10,000,000 the average runtime of buildHeap() is 320.1 milliseconds compared to insert()’s 2,355.6 millisecond runtime.



The runtimes for random input varied because of the fact the input is being added in a random order so the difference between insert() and buildHeap() was not huge. For the buildHeap() method the size average runtimes grew consistently as the size of the input grew. The total average runtime of 104.27 milliseconds is still pretty low.



As stated earlier the insert() method is still less efficient than buildHeap(). The total average runtime of insert() with random input is 242.145 milliseconds. The randomness of the input seems a little more prevalent in the average runtimes with insert(). For example, the average runtime for 700,00 is 151.8 and periodically from test to test the runtime jumps from around 35 milliseconds to 400 milliseconds but when the input size is 1,000,000 the average runtime is 54.2 milliseconds and stays more consistent from test to test.