Assignment 3: Simple Priority Queue

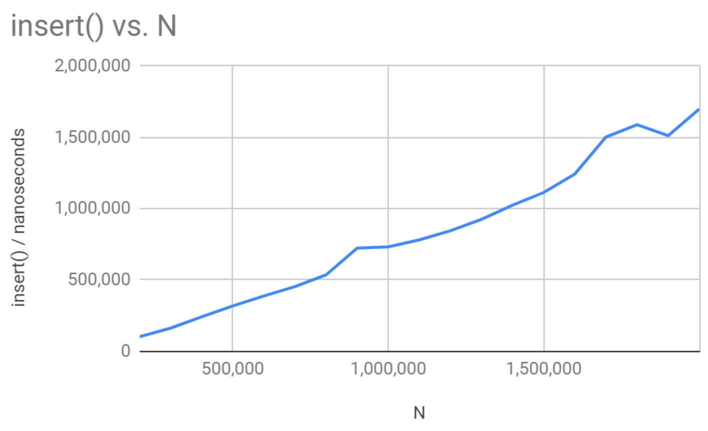
Analysis Document by Qianlang Chen (u1172983)

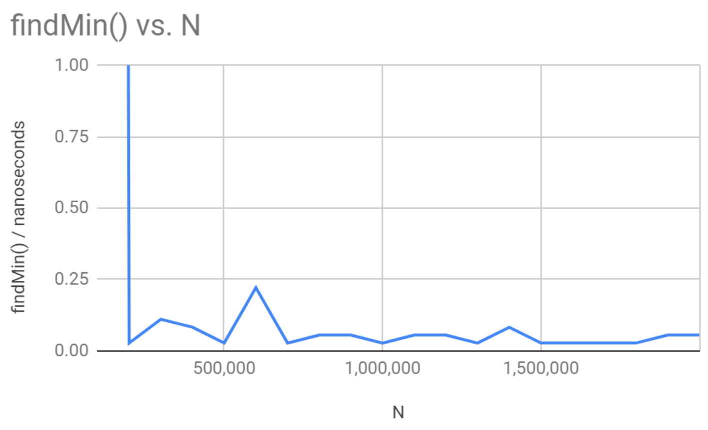
In this assignment, I gladly had Brandon Walton as my programming partner who was also my partner on the previous assignment. Throughout the completion of this assignment, we have only switched our programming roles exactly 3 times. We figured that switching roles too often could potentially mess up our thought process to building the whole project and would lead to inconsistent design and/or code formatting, because of the difference in our coding preferences. What we planned doing instead is we have a person to be the driver most of the time for one assignment, and for each new assignment/project we switch roles. By doing this, each of us will have equal opportunity to learn and show off for the other to learn. Since we have this little plan in mind, and we have gotten into each other to work more efficiently, along with his solid understanding and skills to programming, I will choose to work with this partner again.

The requirements of the assignment pointed out that we could not use Java's *ArrayList* class as our data holder. I did not like this requirement as a start, but after finishing the assignment, I realized that using Java's premade classes would not necessarily make our class run more efficiently. Using a premade class would surely decrease our development time because most of our functions would have existed for us to use. However, we did not use a premade class, so we could better control how our class works rather than letting another class handle it, since we knew exactly what our class does but had no clue what a premade class does. Moreover, calling functions does have its cost, and it is necessary when we use a helper class.

To analyze and understand the behavior of our algorithms in the class, we tested and times the runtime for two of our most important methods in the class: *insert()* and *findMin()*. We created a timer class to run our methods with different data sizes, and here are our results:

N	findMin()	insert()
100,000	38.000	52,126
200,000	0.028	102,763
300,000	0.111	163,195
400,000	0.083	242,030
500,000	0.028	318,151
600,000	0.222	388,285
700,000	0.028	454,329
800,000	0.055	536,268
900,000	0.056	723,449
1,000,000	0.028	732,066
1,100,000	0.055	781,597
1,200,000	0.055	844,812
1,300,000	0.028	925,444
1,400,000	0.083	1,024,294
1,500,000	0.028	1,113,191
1,600,000	0.028	1,242,919
1,700,000	0.028	1,501,474
1,800,000	0.028	1,587,680
1,900,000	0.055	1,510,858
2,000,000	0.055	1,697,605





The results show that the behavior of our *findMin()* and *insert()* methods are **constant** and **linear** consecutively, and they match our predictions well. Our *insert()* method uses a binary search to find the new index which, we know, has a behavior of log(N), but the method also has to copy and/or shift the elements around, and this process has a linear behavior. Therefore, the *insert()* method has an overall **linear** behavior due to its dominance as N gets large. Our *findMin()* method took advantage of the fact that our queue is sorted so that it returns the last element regardless how big our queue size N is (except 0, of course), and this means that its running time stays **constant**.