Corbin Charpentier
Lab 10 Report

Introduction:

The following report contains efficiency analysis of Binomial Queue, Leftist Heap and Skew Heap abstract data types. The report starts with implementation, a description of how the experiment was quantified. After that, the actual experimental results and brief analysis are displayed. Finally, the report concludes with a deeper analysis of the results and a few comments .

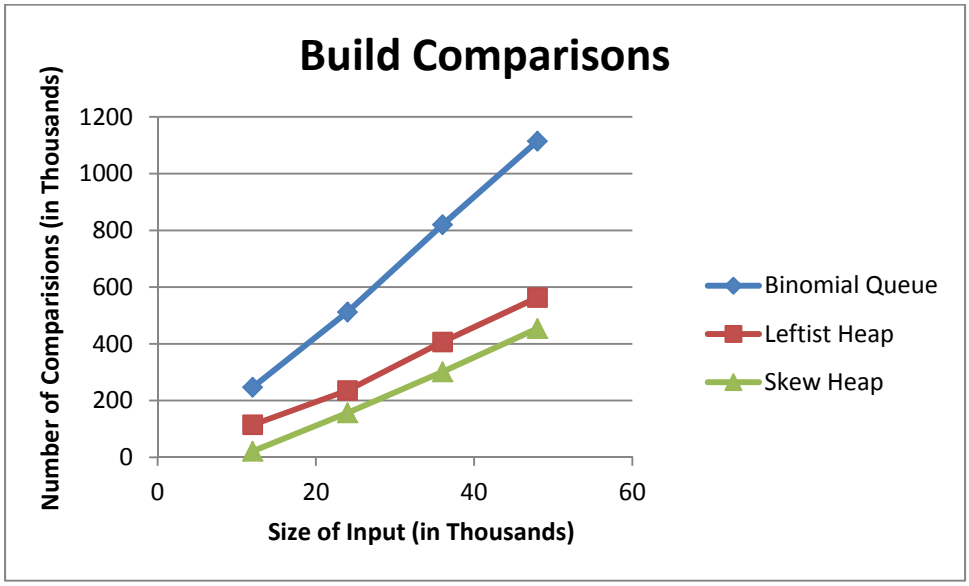
Implementation:

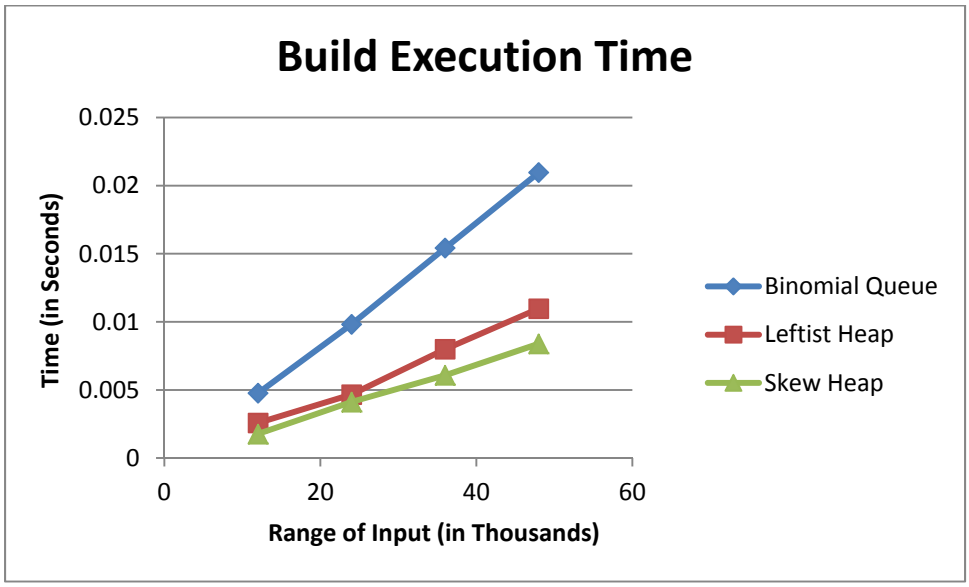
Timing – timing was implemented using the provided Timer class. A global instance of Timer was used record the time in seconds needed to execute the insertion and deletion operations for the Binomial Queue, Leftist Heap and Skew Heap. More technically, the timer began (Timer::start()) ticking when the forloop for each operation began executing and stopped (Timer::stop()) when each for-loop finished. The elapsed time, returned by Timer::stop(), was the total time required for each insertion and deletion operation, respectively.

Comparison Counting – this was implemented by adding a member variable (int comparisons) to the Binomial Queue, Leftist Heap and Skew Heap classes which keeps track of the number of comparisons .Whenever the member functions compare or iterate a loop, count was incremented by one.

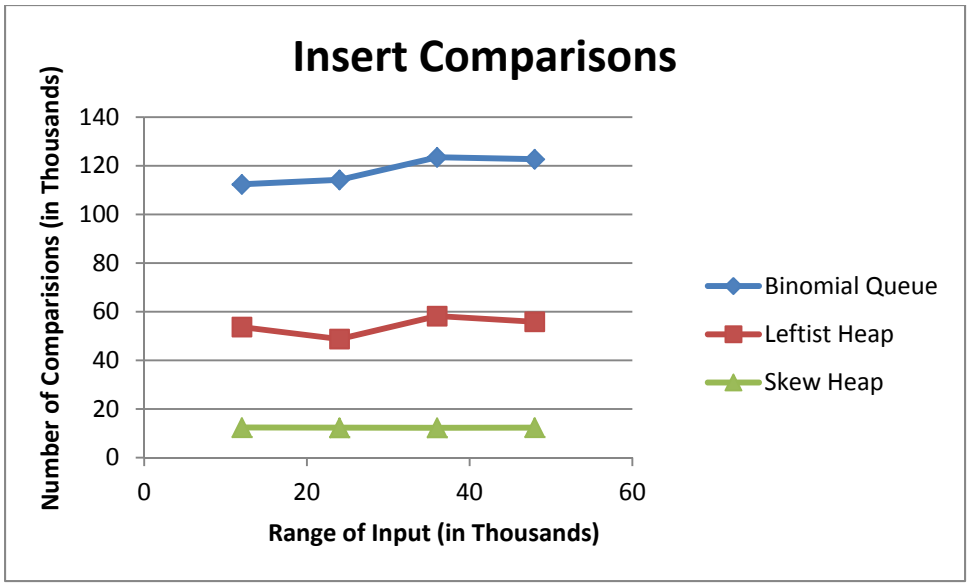
Random Data – random values were attained by calling int* generateRandomDataInRange(int n, int range), which took a size and a range, and returned a dynamically allocated array of size n, containing data from 0 to range.

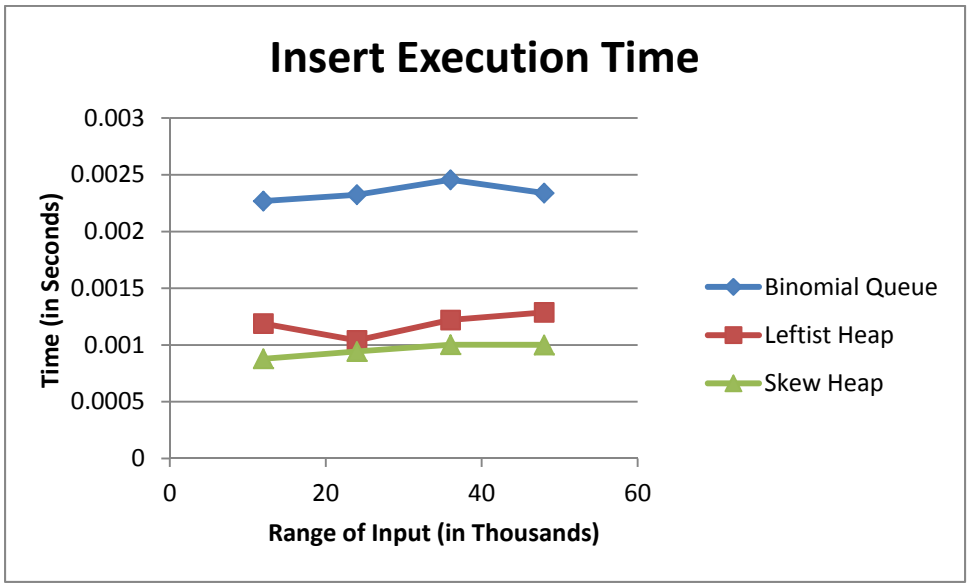
Experiment:





The data range is too small to draw any conclusions about big-O. However, in class learned that the Binomial Heap and Leftist heap have insertion efficiency of O(lgn), while Skew Heap inserts at worst case O(n). The results seem to contradict this since the Skew Heap actually out-performed the other two structures. Thus it becomes important to note that the skew heap's *amortized* insertion cost is O(lgn). While the Leftist Heap and Skew Heap share the same ceiling, the Skew Heap does not need to check rank at ever y level it inserts. This explains discrepancy between comparisons and execution time.



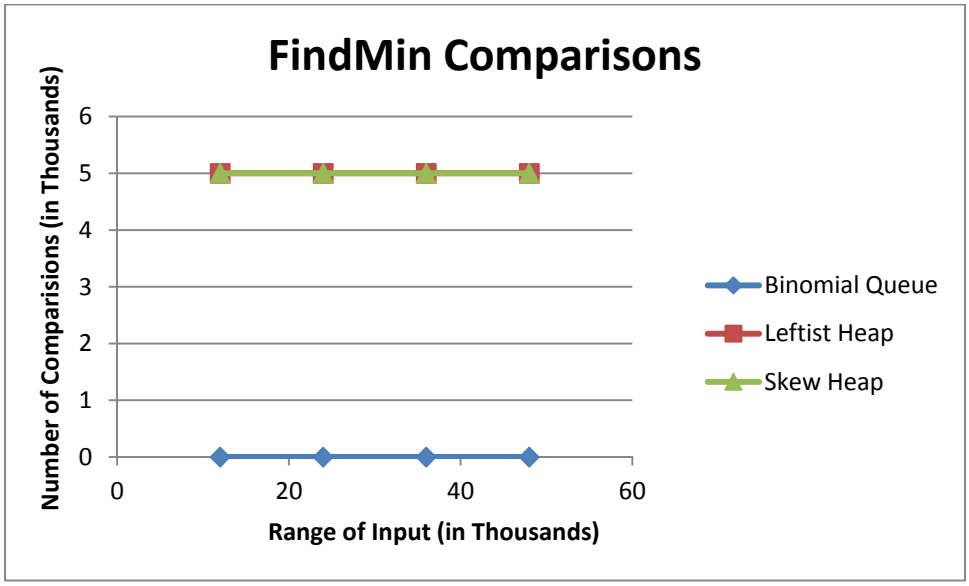


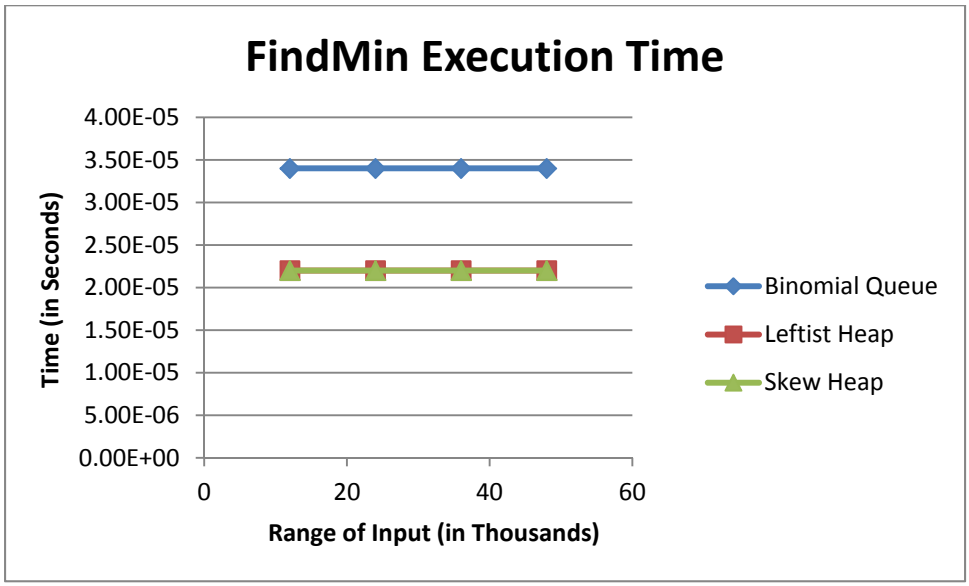
The trend continues. The trend lines look flat because the number of insertions remained constant at 5000. A larger structure (from build) will make the trees taller, but the increased data range somewhat nullifies it.

The Binomial Queue simply executes a lot more checks than the other two structures, allowing them to finish operations in a shorter amount of time. The Leftist Heap more than doubled the Skew Heaps comparison count (for the aforementioned reason), yet took about the same time to execute. This can probably be examined by observing that the Skew Heap must make up to lg(n) swaps at each insertion. For the Leftist Heap, the swapping is not necessary, but a check (comparison) is required. The data implies that the Leftist Heap's check and occasional swap was almost as fast as the Skew Heaps mandatory swap.

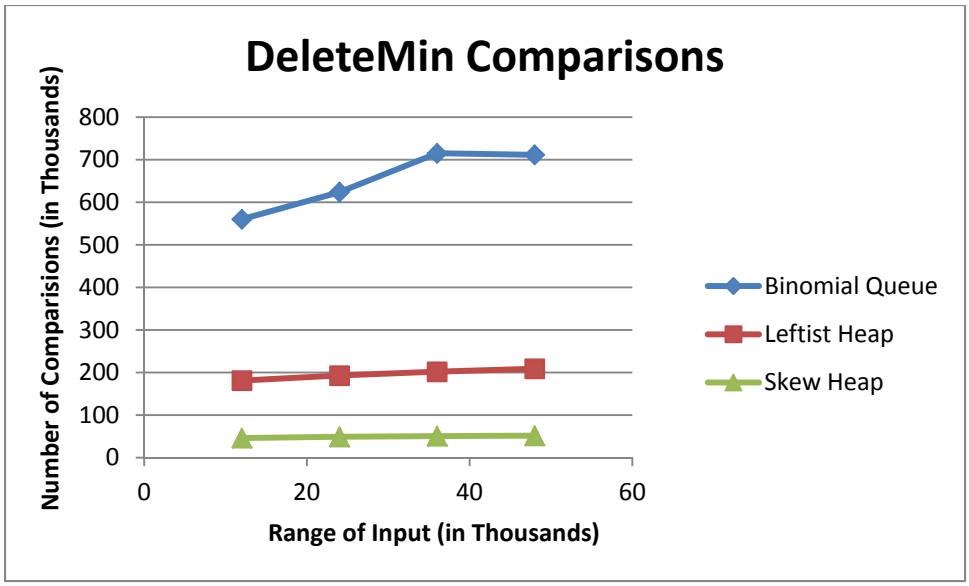
The data structures insert in O(lgn)* time.

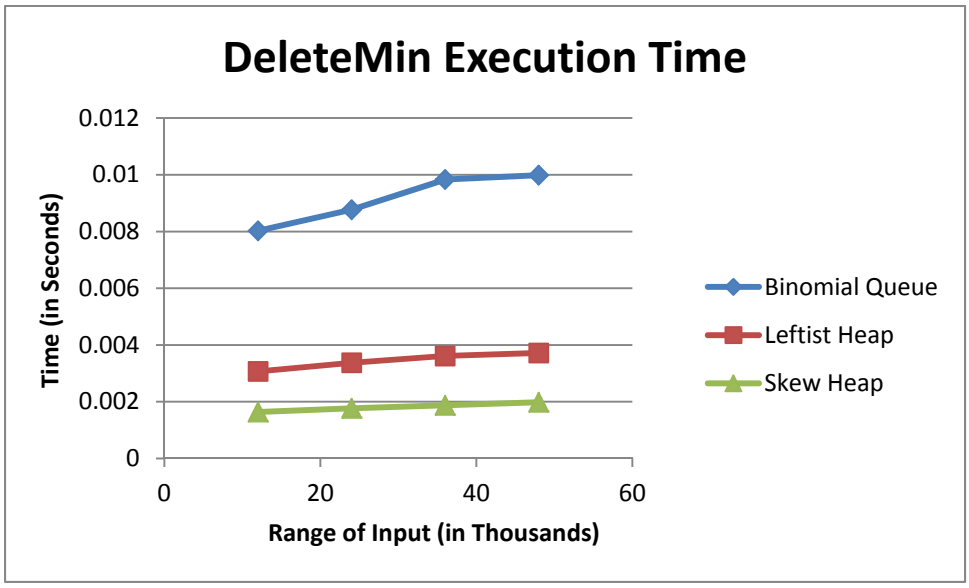
*Amortized Skew Heap





These results are as expected. Since the Binomial Heap was implemented with a min pointer and the Leftist Heap and Skew Heap's min is always at root, findMin was a constant (O(1)) operation for all three data structures. The Leftist Heap and Skew Heap took 5000 comparisons because they each checked if the heap was empty before returning the first index. In retrospect, the Binomial Queue should have checked for NULL as well.





All three data structures showed correlation between number of records and execution time. All three data structures deleteMin at O(lgn)* efficiency. Explanation for the relation in time between the three structures was explained in the insert section.

*Amortized for Skew Heap

Conclusion:

It's interesting to observe that, while the worst case for Skew Heap is O(n), the Skew Heap outperformed two data structures with unamortized O(lg(n)) operations. The Binomial Queue was dominated by the Leftist and Skew Heaps an all non-constant operations, always at least doubling their execution time. It is difficult to conceive specific explanations for this because the two structures operate much differently. When the Binomial Queue inserts, it checks if there is another tree of degree one (O(lgn)) and then it potentially must perform lg(n) merges up the queue. There Binomial Queue simply must perform many more checks than its counterparts.