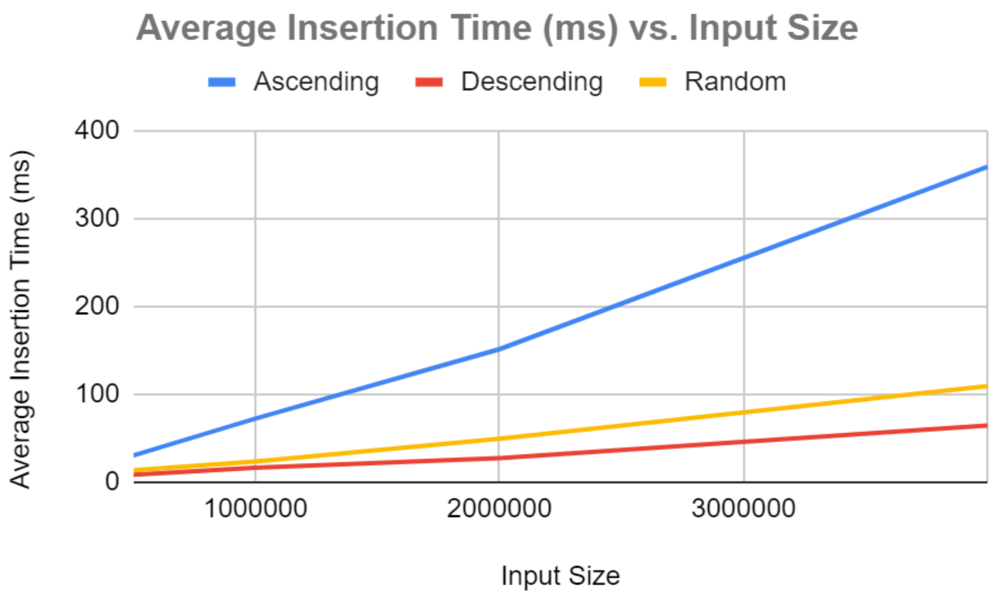
Name: Arjun Sawhney
PID: A15499408

dHeap Runtime Analysis

Test 1: Insertion to 4-ary max heaps with data of different data order

Input Size	Average Ascending Insertion Time (ms)	Average Descending Insertion Time (ms)	Average Random Insertion Time (ms)
500000	30	8	13
1000000	72	16	23
2000000	151	27	49
4000000	359	64	109

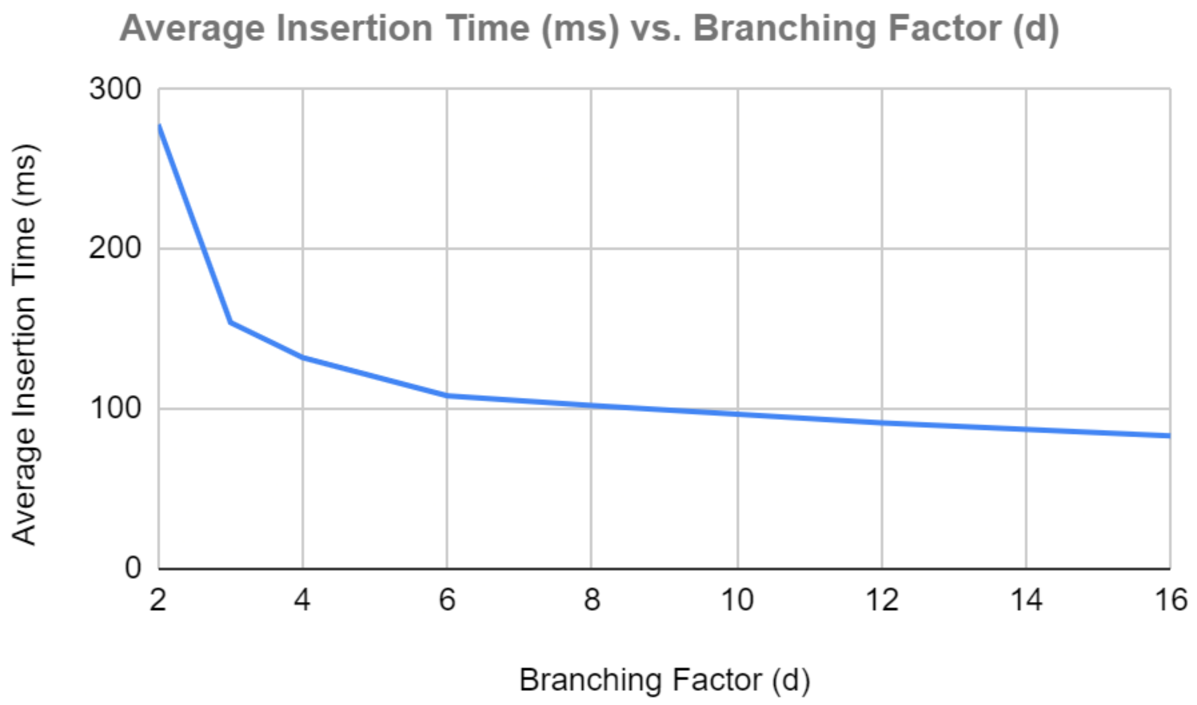


My dHeap implementation achieves the best runtime when the data is sorted in descending order, as observed in the table and graph above. Insertion takes $O(\log_4 n)$ time, but there is a difference in average run time as the data is inserted into a 4-ary **max** heap. This means that no elements had to be *bubbled-up* on insertion for descending data as elements were inserted at the correct level each time. In contrast, data sorted in ascending order requires every element to be *bubbled-up* upon insertion, resulting in extra swaps with parent nodes and longer runtimes. Data sorted in random order falls between the two curves, as not every insertion requires *bubbling-up*.

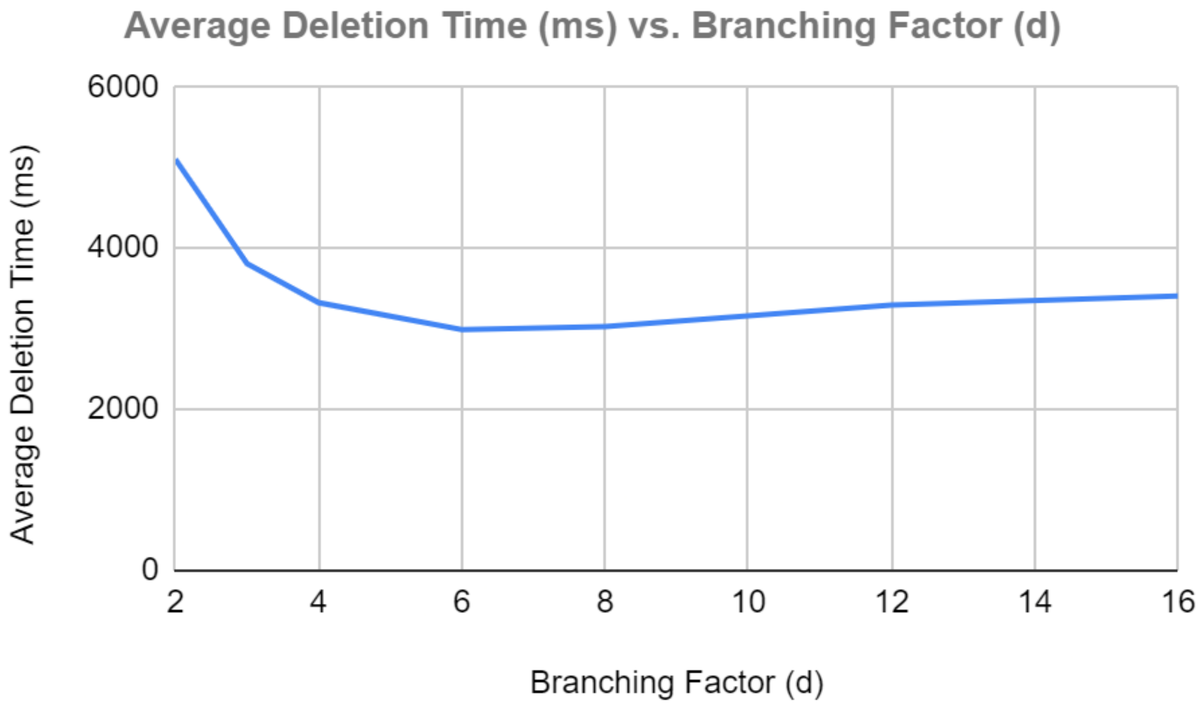
Name: Arjun Sawhney
PID: A15499408

Test 2: Insertion and deletion to d-Heaps with different branching factor (d)

Branching Factor (d)	Average Insertion Time (ms)	Average Deletion Time (ms)
2	278	5106
3	154	3808
4	132	3322
6	108	2986
8	102	3024
12	91	3292
16	83	3404



Name: Arjun Sawhney
PID: A15499408

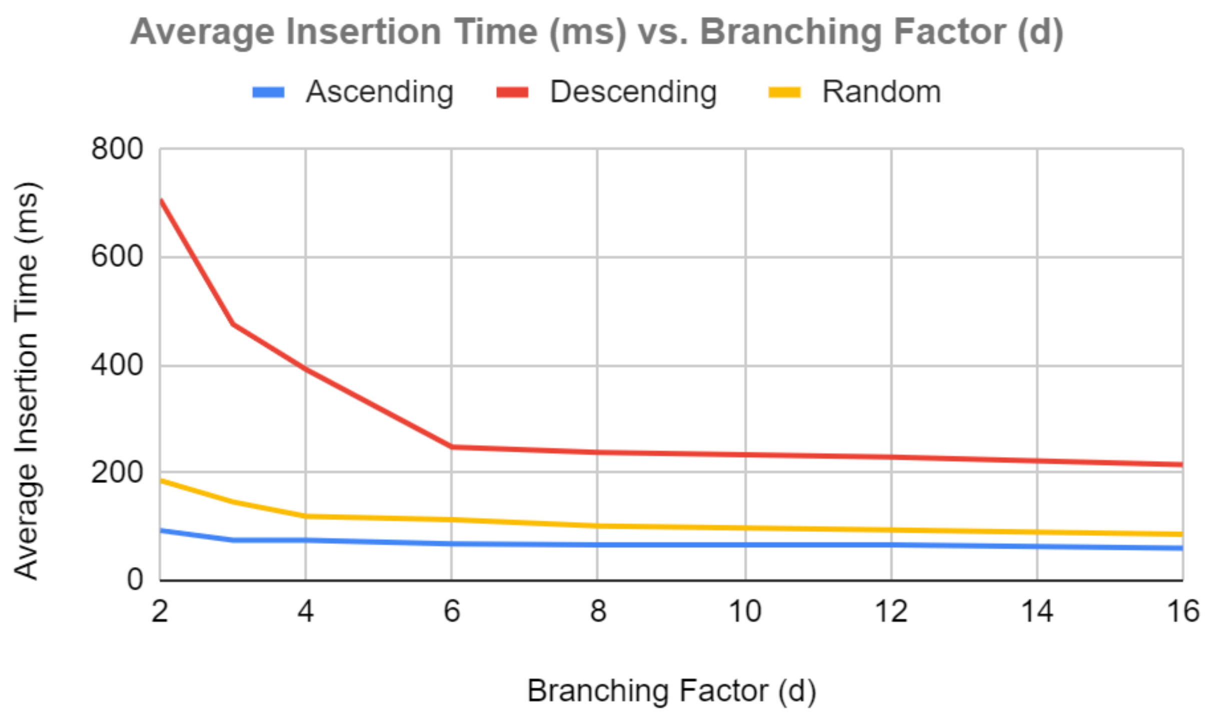


a. Branching factor d = 16 gives me the best performance for insertion while branching factor d = 6 gives me the best performance for deletion in the **min** heap as observed in the table and graphs above. Insertion time decreases with an increase in branching factor, as explained in Part 2.1. We know that insertion in d-heaps takes $O(\log_d n)$ time as the height of a d-heap is $\lceil \log_d n \rceil$. The value $\lceil \log_d n \rceil$ decreases as d increases. The number of comparisons and swaps (*bubbling-up*) depend only on the height during insertion. Therefore, when d is max (=16), it has the fastest insertion time. Although the same logic applies to deletion time, the tradeoff in having a larger branching factor is having to perform extra comparisons finding the min child among children nodes when *trickling-down*. The sweet spot for this trade-off appears to be at d = 6, where the time to find the min child is lowest compared to the runtime benefit of having a smaller tree height.
b. There does not appear to be any discrepancy between the theoretical time complexity described in Part 2.1 and real empirical runtime **for insertion**. However, although the deletion time does increase somewhat as d grows to a large number, the runtime for d > 2 appears to be lower than d = 2. This can be observed in the graph above where the growth in average runtime is not much from 6 < d < 16, but the dip after d = 2 is substantial. This shows that the benefit from having a smaller tree height outweighs the cost of finding the minimum child when d > 2.

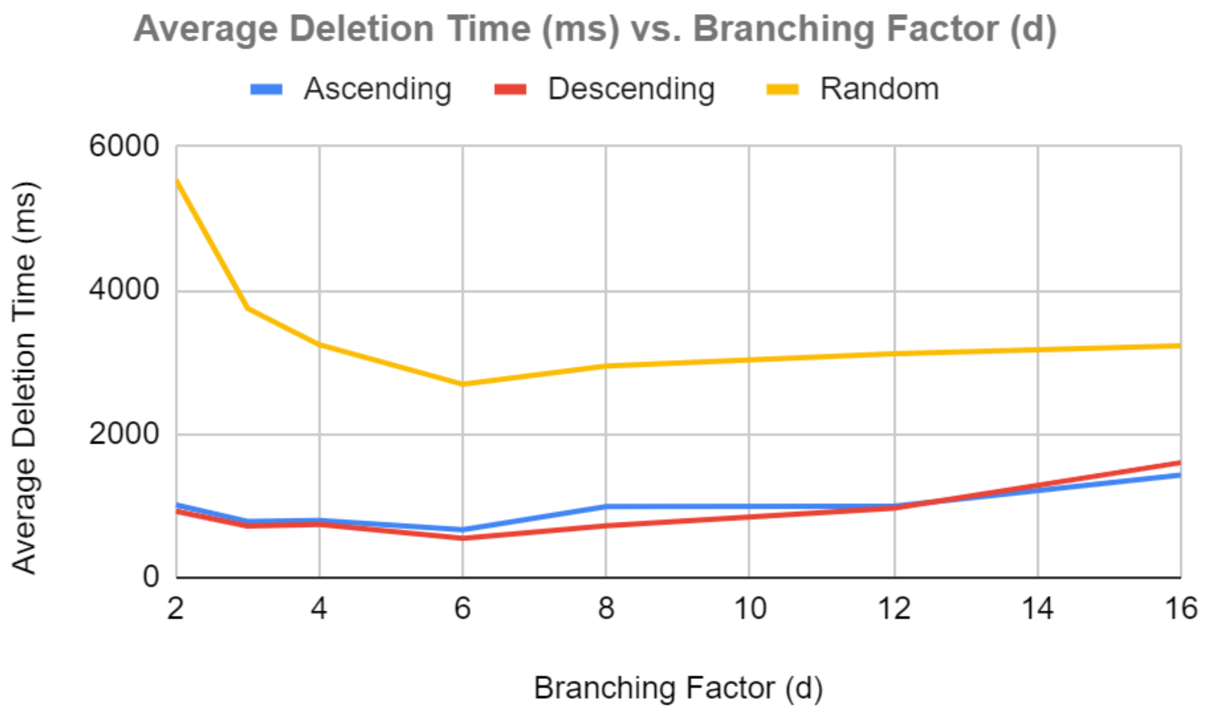
Name: Arjun Sawhney
PID: A15499408

Extra Credit: Insertion and deletion to d-Heaps with different branching factor (d) and data of different data order

Branching Factor (d)	Average Ascending Insertion Time (ms)	Average Descending Insertion Time (ms)	Average Random Insertion Time (ms)	Average Ascending Deletion Time (ms)	Average Descending Deletion Time (ms)	Average Random Deletion Time (ms)
2	93	708	186	1014	925	5540
3	75	476	146	783	718	3750
4	75	392	119	798	740	3244
6	68	248	113	666	548	2692
8	66	238	101	990	722	2947
12	66	229	94	993	966	3118
16	60	215	86	1427	1602	3230



Name: Arjun Sawhney
PID: A15499408



a. The factors are the same as in Test 2 for ascending and descending order. They are d = 16 for best insertion runtime and d = 6 for best deletion runtime. The reasoning for this is the same described in Test 2 and does not change when the data is sorted in different orders.
b. Insertion takes the most time when the data is sorted in descending order as we are inserting into a d-ary **min** heap. This means that *bubbling-up* requires extra comparisons and swaps with every insertion. In contrast, when the data is sorted in ascending order, it is inserted to the correct level with each insertion and no *bubbling-up* is required. Randomly sorted data falls between the two run-times as expected, as not every insertion requires *bubbling-up*. Deletion takes the most time when the data is sorted randomly and follows similar runtimes when the data is sorted in ascending order or descending order. This is because trickling down is more expensive when the data is not sorted in any order, requiring several comparisons and swaps on average when searching for the min child among children. In contrast, sorted data does not take time to find the min child for data sorted in ascending and descending order.