Viswesh Uppalapati
DSC 30
05/18/20

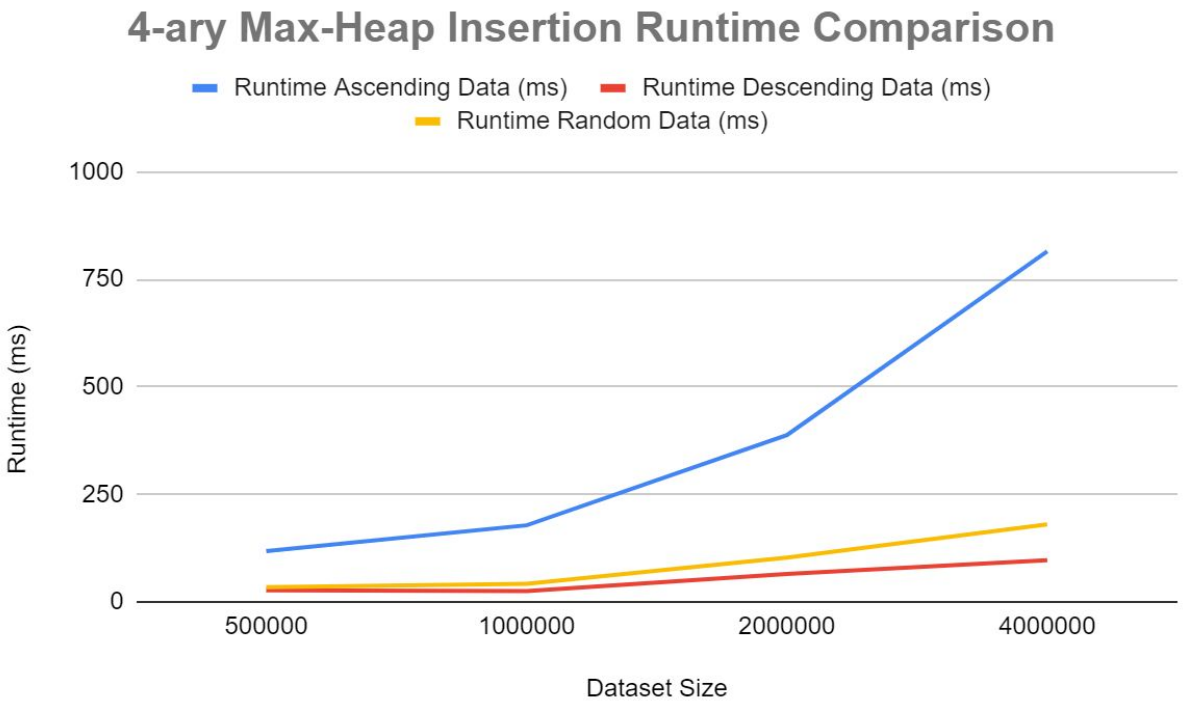
d-Heap Runtime Analysis

Test 1: Insertion with data of different data order

1)

Dataset Size (# of elements)	Runtime Ascending Data (ms)	Runtime Descending Data (ms)	Runtime Random Data (ms)
500000	118	27	34
1000000	178	25	42
2000000	388	65	103
4000000	815	97	180

2)



3)

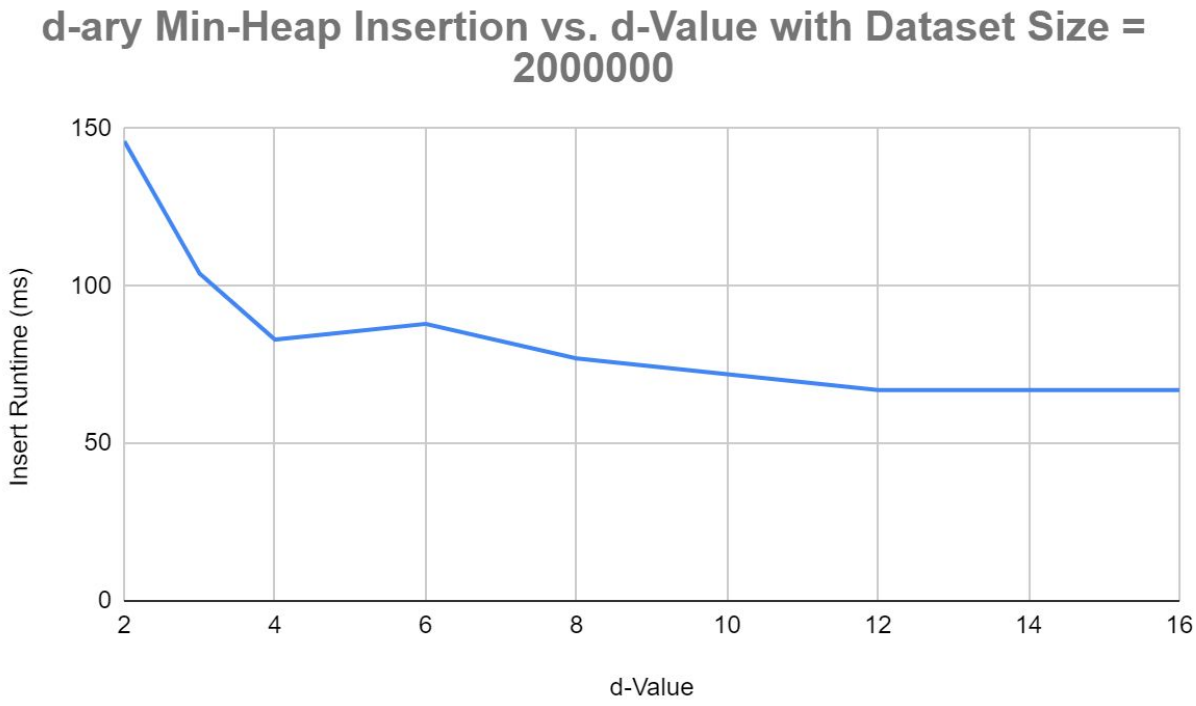
a) The best performance occurs when the dataset is in descending data as seen on the graph (red line), which indicates the lowest runtime in milliseconds. This is because since this test was run using a max-heap, we know that the largest element will be accessed first, so the heap orders them toward the top. If the data is in descending order already, the data will be inserted into the heap without "bubbling" up the data as inserting the data in descending order maintains the max heap definition. A d-ary heap will have a runtime of $O(\log_d n)$ in the worst case, but in the case of a descending ordered insertion of data, it will be best case each time as each data that is added is directly added to its correct spot in the heap without the need for swapping elements through bubbleUp. Therefore, the descending ordered data has the best runtime.

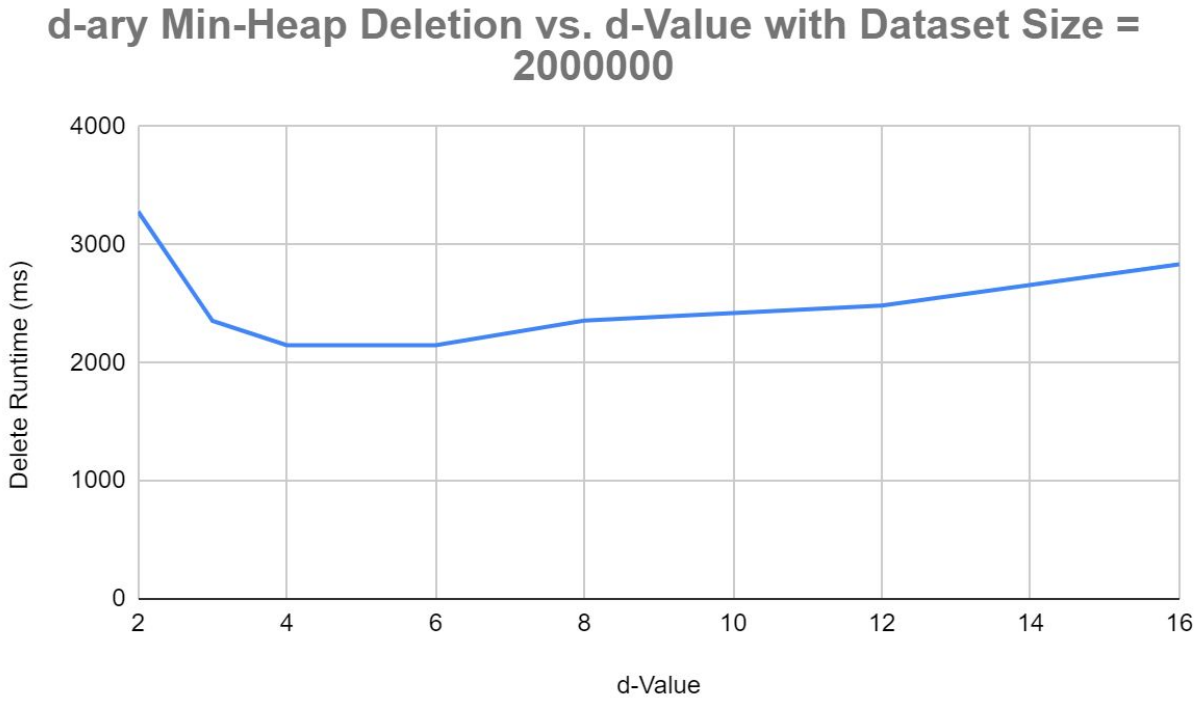
Test 2: Insertion and deletion to d-Heaps with different branching factor (d)

1)

d-Value	Insert Runtime (ms)	Delete Runtime (ms)
2	146	3281
3	104	2355
4	83	2149
6	88	2149
8	77	2358
12	67	2484
16	67	2833

2)


d-ary Min-Heap Insertion vs. d-Value with Dataset Size = 2000000
Insert Runtime (ms)
150
100
50
0
2
4
6
8
10
12
14
16
d-Value


d-ary Min-Heap Deletion vs. d-Value with Dataset Size = 2000000
Delete Runtime (ms)
4000
3000
2000
1000
0
2
4
6
8
10
12
14
16
d-Value

3)

a) According to my table and graphs presented, The largest d-values of 12 and 16 had the fastest insertion runtime. This is because when d is larger, the height of the tree also decreases as each parent can have up to d children. The lower the height of the tree, the lower the runtime as now, bubbleUp will make less comparisons as it makes them based on the tree's height. Therefore, the lower the height, the fewer the comparisons, the faster the runtime. The middle d values had the lowest deletion runtime as seen on the graph and tables, the lowest being at d = 6. This is because as d increases, the number of comparisons drastically increases in each step of trickle down as d comparisons will be made to first find the lowest child (in the case of a min heap), and $\log_d n$ comparisons are made "trickle" down the node based on the height of the tree. Therefore, the larger d-values would have a longer runtime. That is why the middle d-values have the fastest deletion runtime as they are a medium between the most optimized insertion and deletion runtimes.

b) The best average runtime for insertion happened to be when d = 12 and d = 16, and it was 67 ms. The best average runtime for deletion happened at d = 6 and d = 4, and it was 2149 ms. The time complexities for insert and remove are $O(\log_d n)$ and $O(d \log_d n)$, and they are predicted to be $\log_{16} (2000000) = 5.23$ and $6(\log_6 (2000000)) = 6(8.10) = 48.6$ for the above values. And these numbers are the average number of comparisons made per insertion and deletion. The table values and the runtime complexity values differ.

Extra Credit:

a) For me, the fastest insertion and deletion runtime for ascending data occured at d = 6 with an average of 37ms for insertion and 552 ms for deletion. But, most of my upper d values were within a close range of the above values. The fastest deletion for descending data happened at d = 6 with an average of 489 ms and fastest insertion was at d = 16 with an average of 217 ms. These values are not the same because for ascending data, the data is already ordered so that the heap will not have to perform any additional comparisons because it is a min heap and the lowest elements are towards the top for insertion. That is why d = 16 has the fastest insertion for descending data when data needs to be reordered. For deletion, d = 6 posed as a good medium between the low and high d values because high d-values result in too many comparisons made in trickleDown.

b) For insertion, there is a significant difference between the insertion of descending data and the insertion of ascending and random data. This is because when the data is in

ascending order the heap does not need to make any additional comparisons in bubbleUp in order to insert as all the data is properly ordered. For random data, the runtime is on average a little higher than that of the ascending data set as the dataset is a random combination of ordered and unordered data. Lastly, the descending has the largest and most significantly different runtime for this min-Heap test because the data is entered opposite to the definition of a min heap, resulting in many comparisons being made in bubbleUp at each insert, which causes larger runtime compared to the other two datasets. For deletion, in general the random data set had a much larger runtime than the two ordered ones. I believe that this is because the heap when created is in a random order, therefore needing multiple comparisons while trickling down. For the ordered sets, however, when the tree is made they end up ordering the data in the heap so that the heap array is just an in order representation of the data. Therefore, while deleting, trickling down needs fewer comparisons for the data to be reordered correctly.