COMP90077 Advanced Algorithms and Data Structures

Assignment 2

Part 1: Experimental Study on the Quake Heap

Report on Experimental Results

Name: Kechen Zhao

ID: 957398

1. Experimental Environment

This Experimental study on Quake Heap with competitors, the unsorted Dynamic Array and Binary Heap, is implemented and complied using Java and IntelliJ IDEA under macOS with 3.1 GHz Dual-Core Intel Core i5 processor and 8GB 2133 MHz memory. All result data are generated under this environment.

2. Data Generation

All required methods of the three data structures are implemented by using different Java classes and the input experimental data are strictly generated based on the following guidelines:

- For *Experiment 1: Time v.s. Number of Insertions*, the running time of 5 insertion-only sequences with length 0.1M, 0.2M, 0.5M, 0.8M and 1M on three data structures will be counted with $M = 10^6$;
- For *Experiment 2: Time v.s. Delete-Min Percentage,* the length of operation sequences are fixed to 1M, while 5 different percentages are adapted to determine the proportion of *delete-min* among insertion operations: 0.1%, 0.5%, 1%, 5%, 10%. Before each operation, equality between number of previous *insert* and *delete-min* will be checked to avoid any exception, then a random probability will be generated to decide whether a *delete-min* or *insert* should be performed;
- For *Experiment 3: Time v.s. Decrease-key Percentage,* same ratios will be used to set the proportion of *decrease-key* operations among 1M operations, random probabilities will be generated before actual operations to decide whether a *decrease-key* or *insert* should be performed;
- For *Experiment 4: Time v.s. Length of Mixed Operation Sequences,* 5 different length of operation sequences (same as for experiment 1) will be tested with 5% are *delete-min*, 5% are *decrease-key* and *insert* with the rest probability. Random probabilities will be generated before actual operations to decide whether a *decrease-key, delete-min* or *insert* should be performed;

3. Experiments

4 experiments as described above will be performed to examine the running time complexity of 3 data structures: Quake Heap, Unsorted Dynamic Array and Binary Heap. Time taken by running different lengths of sequences and various proportions of operations will be counted in millisecond to compare the efficiency of different data structures.

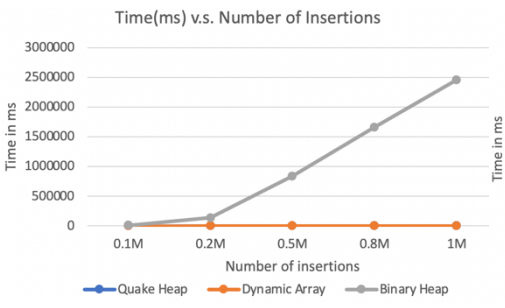
3.1 *Experiment 1: Time v.s. Number of Insertions*

This experiment analyzes the running time complexity of insert operations under Quake Heap, Unsorted Dynamic Array and Binary Heap. Algorithms will be running on operation sequences with lengths equal to 0.1M, 0.2M, 0.5M, 0.8M and 1M with $M = 10^6$.

3.1.1 Result Demonstration

The below table gives the running time of different numbers of insertion on three data structures, and the diagram plots the data and shows the comparison between them.

Time(ms) vs. Number of Insertions			
Length	Quake Heap	Dynamic Array	Binary Heap
0.1M	59	8	8906
0.2M	80	26	136614
0.5M	202	63	836929
0.8M	399	42	1660411
1M	487	81	2458678



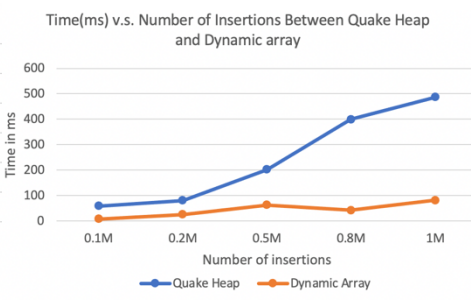


Table 1. Time(ms) vs. Number of Insertions. Diagram 1. Plot of Time(ms) vs. Number of Insertions I Diagram 2. Plot of Time(ms) vs. Number of Insertion II

It is obvious that larger sequences of insertion runs longer on all three data structures, while Diagram 1 & 2 reveal a more visible result that as number of insertions grows, Binary Heap has the rapidest increase. Moreover, Binary Heap also is the one that has the largest runtime complexity on all different lengths of operations, which is followed by Quake Heap. Overall, Dynamic Array has the best performance in this experiment and it has the highest efficiency under all operation sequences.

3.1.2 Analysis

First we analyse the total conceptual running time for Quake Heap. For each single insertion operation, a new node will be added to the forest as an independent tournament tree, no *merge_tree* or *invariant_maintenance* will be performed, so the running time will be bounded by O(1). Therefore, for different numbers of insertions, the total runtime complexity is O(length).

Second, for the Unsorted Dynamic Array, the insertion is just to append an element to the end of the array list, which takes O(1), so the total running time for sequence of insertion equals to O(length).

Finally, based on theoretical analysis of Binary Heap, each insertion invoke a *bubble_up* operation which iteratively do the nodes' movement until the inserted element finds its correct position in the heap array. Consequently, the bound for a single insertion operation is O(1) (for insertion) + O(log(n)) (for bubble_up) = O(log(n)), where n is the total number of nodes and log(n) means the height of the heap. Thus, the total runtime complexity for each operation sequences will be O(log(n)*length).

Therefore, by calculating the overall runtime complexity for each data structures, it's clear to see that Binary Heap has the worst upper bound which is significantly huger than other two, and this conclusion is corresponded to the experimental data. Another interesting point is although based on theoretical bound, Quake Heap and Dynamic Array should have the similar experimental data, the actual running time of Quake Heap is generally longer than that of Dynamic Array. This may be explained by the hypothesis that due to the implementation of Quake Heap, both element and node and their corresponding features (e.g. key, priority, pointer, etc.) needs to be constructed during the insertion, which makes the running time slightly longer.

3.2 *Experiment 2: Time v.s. Delete-Min Percentage*

This experiment analyzes the runtime complexity of sequences that include both *insert* and *delete-min* operations under Quake Heap, Unsorted Dynamic Array and Binary Heap. Algorithms will be running on operation sequences with lengths equal to 10^6 and proportion of *delete-min* stands at 0.1%, 0.5%, 1%, 5%, 10%.

3.2.1 Result Demonstration

The below table gives the running time of different percentages of *delete-min* operation in the 10^6 sequences on three data structures, and the diagram plots the data and shows the comparison between them.

Time(ms) vs. Delete-Min Percentage			
%	Quake Heap	Dynamic Array	Binary Heap
0.1	10836	767	471
0.5	247425	13275	3131
1	1308093	74371	16978
5	8686251	556239	81904
10	21715628	2159831	310362

Table 2. Time(ms) vs. Delete-Min Percentage.

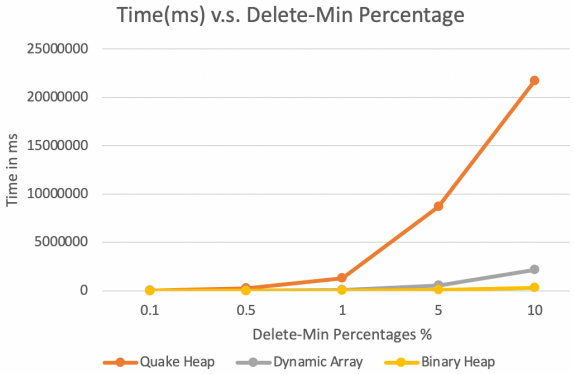


Diagram 3. Plot of Time(ms) vs. Delete-Min Percentage

Both table and plot illustrate that larger operation sequences takes longer to run, and higher proportion of *delete-min* operation will also cause higher time complexity under 3 data structures. To be more specific, Diagram 3 demonstrates that Quake Heap in this experiment has the highest running time regardless of the ratio of *delete-min,* and its growth of time with respect to the percentage increase also is the most dramatic compared to that of the Unsorted Dynamic Array and Binary Heap. Dynamic Array also has a large time complexity when running high ratios of *delete-min* operations, while its highest running time is only 10 percent of that of Quake Heap. In terms of Binary Heap, although its largest runtime complexity reaches 310362ms, this is the most efficient data structure for *delete-min* operation: it performs the best under all proportions of *delete-min*.

3.2.2 Analysis

The extremely high runtime complexity of Quake Heap can be demonstrated by studying the theoretical analysis of *delete-min* on Quake Heap. For each single operation, minimum node and its corresponding element (can be found in node's pointer) need to be searched by scanning each root in the forest, which takes O(number of trees), then the path which contains the minimum node need to cut and delete, which needs O(log(number of nodes in the tree)). After that, *merge_tree* and *invariant_maintenance* will be performed to link the trees that have the same height and maintain the invariant of it is violated. Due to the actual implementation, runtime complexity of *invariant_maintenance* may reaches O(n) in order to count the number of nodes in each level and that for *merge_tree* would reach O(number of root nodes^2) where n represents the total nodes in the heap. Therefore, since the upper bound for number of root nodes is n, the overall bound for the *delete-min* operation under my implementation is $O(n^2)$.

To perform the *delete-min* operation on the Unsorted Dynamic Array, each time the entire list need to be scanned to find the smallest element, then perform the deletion. Thus, the theoretical and also the actual bound for runtime is O(n), where n stands for the total number of elements in the array list.

Binary Heap is the most efficient data structure for *delete-min* operation, both theoretically and empirically. The smallest node is just the root so any searching process can be skipped. After interchanging the position of root and the last element in the heap array, only one *bubble-down* operation need to be executed to the new root to find its correct position, which takes O(log(n)), where log(n) represent the height of the heap. Consequently, the overall running time for a single *delete-min* operation on Binary Heap is bounded by O(log(n)).

In conclusion, the above theoretical analysis is matched with the experimental data: Binary Heap has the smallest bound that is O(log(n)), which is followed by the Unsorted Dynamic Array, O(n). Quake Heap has the worst efficiency under my implementation, which is $O(n^2)$.

3.3 *Experiment 3: Time v.s. Decrease-key Percentage*

This experiment analyzes the runtime complexity of sequences that include both *insert* and *decrease-key* operations under Quake Heap, Unsorted Dynamic Array and Binary Heap. Algorithms will be running on operation sequences with lengths equal to 10^6 and proportion of *decrease-key* stands at 0.1%, 0.5%, 1%, 5%, 10%.

3.3.1 Result Demonstration

The below table gives the running time of different percentages of *decrease-key* operation in the 10^6 sequences on three data structures, and the diagram plots the data and shows the comparison between them.

Time(ms) vs. Decrease-Key Percentage			
%	Quake Heap	Dynamic Array	Binary Heap
0.1	432	830	8063
0.5	629	2495	30546
1	982	4500	53908
5	1146	7760	68877
10	1673	11304	106730

Table 3. Time(ms) vs. Decrease-Key Percentage

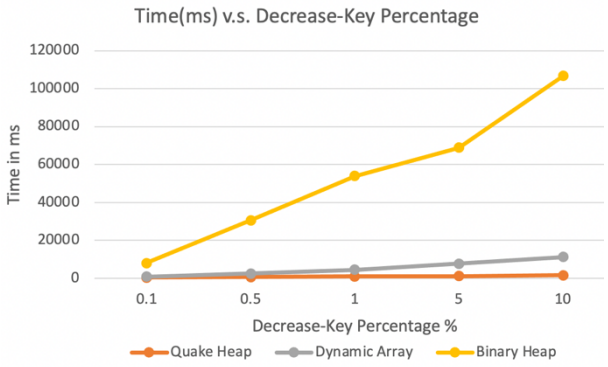


Diagram 4. Plot of Time(ms) vs. Decrease-Key Percentage

From the above plot and data, Quake Heap are the most efficient data structure for decrease-key operation, and it has the lowest running time regardless of percentages of decrease-key in the operation sequences. Dynamic array is the second efficient one, and the time growth with respect to the percentages are not so significant in the plot. Binary Heap has the worst performance when running decrease-key operations, it has the highest complexity in all cases, with the longest runtime equals to 106730ms when there are 10% of the operations are decrease-key.

3.3.2 Analysis

Decrease-key operation supported by Quake Heap can be achieved with O(1) complexity due to the pointer of each node which points to the corresponding element. During each operation, the key value of the given element will be modified, and the nodes which set it as the pointer will also changes their key values. After that, any cut that is

necessary well be performed in O(1) time. Conceptually, the running time of Quake Heap in all cases should be approximately equal to 487ms as we get in *Experiment 1,* and when percentage is 0.1%, they are indeed roughly equal. However, as percentage increases, time complexity grows. This may be explained that the implementation manually loops all nodes that need to reset the pointer to change their key values, therefore the actual running time is higher than 487ms. Nonetheless, Quake Heap still has the best performance with respect to other competitors.

The performance of the Unsorted Dynamic Array is slightly worse than the Quake Heap, and this is due to the scanning of the array list in order to find the element that need to be processed. The theoretical bound is O(n), which is the same as we analyzed in *Experiment 2*. However, the actual running time is this experiment is better than that in *Experiment 2,* and the reason is that to perform the *decrease-key* operation, only part of the array list need to be scanned, once the target element is reached, the loop stops. Thus, although the bound for *decrease-key* supported by the Unsorted Dynamic Array is O(n), the actual performance will be better.

The theoretical bound for *update-key* by using Binary Heap is O(log(n)), where log(n) stands for the height of the heap. This bound is the direct result of *bubble-up* and *bubble-down* involved in both *decrease-key* and *insert* operations once the min-heap property is violated.

Therefore, although the Unsorted Dynamic Array has the highest bound, it is reasonable that the actual performance is better than the theoretical bound and better than the performance of Binary Heap. As a result, the experimental data is agreed with the conceptual analysis.

3.4 *Experiment 4: Time v.s. Length of Mixed Operation Sequences*

This experiment analyzes the running time complexity of mixed operations under Quake Heap, Unsorted Dynamic Array and Binary Heap. Algorithms will be running on operation sequences with lengths equal to 0.1M, 0.2M, 0.5M, 0.8M and 1M with $M = 10^6$, and the percentages of *delete-min* and *decrease-key* will be set to 5% of the length of the sequences respectively, with the rest are insertion operations.

3.4.1 Result Demonstration

The below table gives the running time of different length of mixed operation sequences on three data structures, and the diagram plots the data and shows the comparison between them.

Time(ms) vs. Length of Mixed Operations			
Length	Quake Heap	Dynamic Array	Binary Heap
0.1M	9476	550	7149
0.2M	83321	5576	106356
0.5M	681446	50243	520980
0.8M	2198829	147983	1265887
1M	4346753	337190	1989929

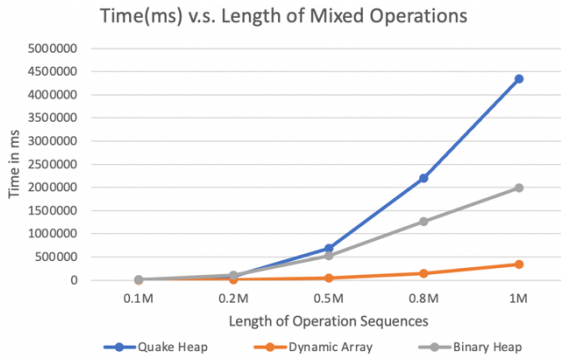


Table 3. Time(ms) vs. Length of Operation Sequences Diagram 4. Plot of Time(ms) vs. Length of Operation Sequences

From the above data, all three data structures have high runtime complexities compare to the previous three experiments, with the Quake Heap takes the longest time to run on the different lengths of mixed operation sequences, which is followed by Binary Heap. The Unsorted Dynamic Array is the most efficient data structures for mixed operations, which has the lowest running time regardless of the lengths of the operation sequences.

3.4.2 Analysis

Based on the theoretical analysis from the previous three experiments, we have known that the runtime complexities of a single operation under my implementation are (with n represents the total number of nodes/elements in the data structure):

- The Quake Heap
  - *Insert*: O(1)

    - *Delete-min*: $O(n^2)$
    - *Decrease-ke*y: O(1)
    - Total runtime complexity of mixed operations
      = O(1*length)*95%+O(n^2*length)*5%+O(1*length)*5%
      Which may be dominated by O(n^2*length)
- The Unsorted Dynamic Array
    - *Insert*: O(1)
    - *Delete-min*: O(n)
    - *Decrease-key*: O(n)
    - Total runtime complexity of mixed operations
      = O(1*length)*95%+O(n*length)*5%+O(n*length)*5%
      Which may be smaller than O(n*length)
- The Binary Heap
    - *Insert*: O(log(n))
    - *Delete-min*: O(log(n))
    - *Update-key*: O(log(n))
    - Total runtime complexity of mixed operations
      = O(log(n)*length)*95%+O(log(n)*length)*5%+O(log(n)*length)*5%
      = O(log(n)*length)

Therefore, when running the sequences of mixed operations, it's sensible that Quake Heap will have the highest runtime complexity since $O(n^2)$ is extremely high when n goes to large numbers, and that will dominate the overall running time. It is also obvious that Binary Heap will also reach a large runtime complexity because of the O(log(n)) bound for each single operations. For the Unsorted Dynamic Array, it's not easy to make the conclusion that it is the most efficient data structure for mixed operations, although this is true based on the experimental data under my implementation. As analyzed in *Experiment 3*, the actual runtime for *decrease-key* supported by the Unsorted Dynamic Array is significantly smaller than *delete-min*, while they have the same upper bound. Therefore, by combining all the runtime for mixed operations, the Unsorted Dynamic Array may reaches a much smaller bound than O(n). This result also can be tested by adding the experimental data from the previous three experiments to get an approximation for Experiment 4: if we choose the data for 0.8M from Experiment 1, percentage = 5% in both Experiment 2 and 3, we will get:

- Quake Heap: 399ms + 8686251ms + 1146ms = 8687796ms
- The Unsorted Dynamic Array: 42ms + 556239ms + 7760ms = 564041ms
- Binary Heap: 1660411ms + 81904ms + 68877ms = 1811192ms

This also indicates that the Unsorted Dynamic Array is the most efficient data structure when running mixed operation sequences under my implementation.

3.5 *Conclusion*

To sum up, all three data structures have their own advantages when running different operations under my implementation: the Unsorted Dynamic Array is the most efficient one for *insert* operation; Binary Heap performed the best when operating *delete-min* due to its min-heap property; Quake Heap has the shortest runtime for *decrease-key* operation which is theoretical bounded by O(1). Generally, based on the data given by *Experiment 4*, and also the overall result given by the first three experiments, the Unsorted Dynamic Array is the most suitable data structure for mixed operation sequences under my implementation.

However, different implementations may give different results, and one of the drawbacks of mine is the high runtime complexities for *merge_tree* and *invariant_maintenance* methods implemented in Quake Heap class. The $O(n^2)$ upper bound of them enormously lower the overall performance of Quake Heap in *Experiment 2 and 4*, which make it to be the worst data structure when executing mixed operations. Different methods could be tried in the future to increase the actual performance to achieve different results.