Pedro Hernandez Carranza
Student ID: 008806974
Repository Link: https://github.com/PedroHernandezC/Project-5-Part-1

Empirical Comparison of huffman coding and batch then Drain for a Priority-Queue API

Abstract
We compare binomial queue and binary heap across two workload profiles—batch-then-drain, and huffman coding. The huffman profile represents a use case where insertions and extractions both happen frequently whereas the batch then drain profile highlights the implementations insert and delete methods when not intertwined. Traces are generated with fixed seeds, checked against an oracle, and timed with a harness that measures only the replay loop. The results showed that both the binomial queue and the binary heap have the expected NlogN curve with the binomial queue being pushed upward due to its longer search and deallocation of memory.

Question / Hypotheses
How do the different implementations of the priority queue react under different usage profiles. Under both profiles we expect to see a NlogN. They should both be fairly close to one another with them being closer in the huffman coding due to the mixture of insertion and deletions. The binary queue should be faster for the batch then drain profile as its vector implementation is cache friendly.

Method (one paragraph)
First trace generators for the huffman profile and batch then drain profile are run. These traces are then used by the harness to determine what operation to execute from insert, findMin, deleteMin, and extractMin. The harness also begins with a throw away loop, only the replay loop is timed, trials use medians for the reported time, and seeds are fixed for reproducibility reasons.The generators also create randomized values for interactions for the given number of trials. After running the harness csvs are created that can then be used with the pq_multi_impl_anchor_heap_tooltips.html to display the profiles data in a graph.

Data

Figure 1

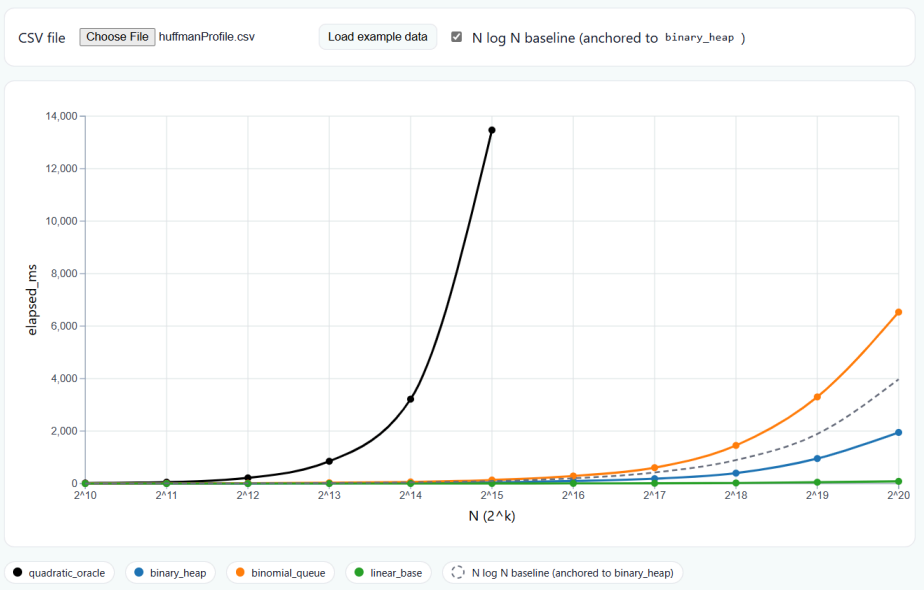

Figure 1 shows the results of running harness with the huffman coding profile. From here we can see that all the graphs have the expected NlogN shape. The binomial queue (orange) grows faster with N then the binary queue due to its slower deletions. Similarly the binary queue grows slower due it fast cache friendly deletions despite the slower insertions.

Figure 2

Figure 2 shows the results for the batch then drain profile. Here the binary queue very closely follows the shape of the NlogN line (dotted). The binomial queue on the other

hand greatly increases as N becomes larger. This is likely due to the slow deletion process.

Discussion

The batch then drain emphasized the priority queues deletion and insertion operations. The binomial has fast insertions by allocating memory for new does to be added to its internal list, but deleting them requires traversal of the list and then deallocation of the memory. The binary queue uses a vector that requires elements to sometimes be shifted or for more memory to be given to the vector leading to slower insertion times. The benefit is that the vector is indexable and is stored in continuous memory. The binary queue's linear memory makes serialized deletions faster.

Limits / Scope

Single-threaded, in-memory; deleteMin(void) with extraction as F→D; item = (key, id [, payload]); comparator ignores payload. Results may shift with string comparators, different allocators, or hardware.

Conclusion

The queues are generally the same for N < 2^16. Past that point the binary queue tended to be faster for both the human coding and batch then drain profiles. In both trials the implementation of the queues all had the expected NlogN shape but the binary queue tended to be a closer fit to the theoretical run time complexity.