Abstract

I compared binary heap and binomial queue implementations using the batch-then-drain workload profile. Binary heap outperformed binomial queue by 4.8× at N = 2^20, which was the opposite of what I expected. I thought binomial queue's amortized O(1) insertions would give it the edge, but I was wrong.

1. Question and Hypothesis

Question: Which priority queue is most efficient for batch-then-drain?

Hypothesis: I expected binomial queue to win because it has O(1) amortized insertions while binary heap needs O(log N) for each insert. I figured this insertion advantage would make binomial queue faster overall. I did wonder if binary heap might be better at deletions, but I stuck with my prediction that binomial queue would come out on top.

2. Method

I followed the same approach as the Huffman profile experiments. The API uses insert, findMin, deleteMin, and extractMin operations with items (key, id) and comparator "key asc, then id asc."

I generated batch-then-drain traces following the Framework's Section 3 pattern: N inserts followed by N extractMin operations. Keys were uniform random integers from [1, N] (unlike Huffman which used a limited range to create duplicates). I tested sizes N = 13, 2^10 through 2^20, using seed 23 for reproducibility.

For timing, I used the Section 5 methodology: one untimed warm-up run, then 7 timed trials with the median selected. Only the replay loop was timed, not file I/O. I tested four implementations: binary_heap, binomial_queue, linear_base, and quadratic_oracle. I stopped quadratic_oracle at N = 2^15 because it got too slow.

3. Results

Figure 1 shows time vs N on a log-scaled x-axis for all four implementations. Binary heap beat binomial queue at every size, and the gap got bigger as N increased. At N = 2^10, binary heap was about 4.7× faster (0.33 ms vs 1.55 ms). By N = 2^20, it was 4.8× faster (643 ms vs 3061 ms). There was no crossover in speed binary heap stayed ahead the whole time.

Takeaway: Binary heap's cache-friendly array structure and simpler deletion algorithm make it the clear winner for batch-then-drain workloads, despite binomial queue's theoretical insertion advantage.

4. Interpretation

I was wrong about my hypothesis. I focused too much on insertions and didn't think enough about what batch-then-drain actually does: you insert everything first, then delete everything. That means deletions dominate the runtime.

Here is why binary heap wins.
Cache locality: Binary heap uses a contiguous array, so when you delete the minimum, you're moving through memory sequentially. The CPU can load chunks of the array into cache efficiently. Binomial queue uses a forest of trees with pointers everywhere, so you're jumping around memory and missing the cache more often.

Simpler deletion: Binary heap's deleteMin is straightforward swap the root with the last element, then sift it down through the array. Binomial queue has to scan all the root nodes to find the minimum, remove that tree, then merge all its children back into the forest. That's a lot more than binary heap has to do that's for sure.

Constant factors: Both are O(log N) for deletions, but binary heap's constant factors are way lower. The 4.8× difference at large N shows that even with the same big-O complexity, the actual implementation details matter a lot.

The graph shows both curves growing at roughly the same rate (both O(N log N)), but binary heap's line is always lower. The gap widens as N gets bigger because those constant factor differences add up over millions of operations.

Batch-then-drain is different from interleaved profiles like Huffman. In Huffman, you mix inserts and deletes together, which might favor structures with fast inserts. But batch-then-drain is all about deletion performance since you do all the deletions after the build phase. That's why binary heap shines here.

Artifacts

- Trace files: `traces/batch_then_drain/`
- CSV results: `csvs/batch_then_drain_profile.csv`
- Plot: `charts/pq_multi_impl_anchor_heap_tooltips.html`
  See README.md for commands to reproduce the results.