# Cross-Review Deep Comparison & Optimizations: Aida (Min-Heap) vs Tomiris (Max-Heap)

## 1. Executive Summary

This document extends the original cross-review by providing a deeper technical comparison, explicit optimization strategies for each heap implementation, and estimated outcomes (empirical impact and complexity consequences) after applying those optimizations. Baseline metrics referenced were taken from the partner reports (uploaded files).

## 2. Baseline (selected metrics from reports)

**Baseline CSV snapshot (from uploaded reports):**

| **N** | **Comparisons** | **Swaps** | **Array Accesses** | **Time (ms)** |
| --- | --- | --- | --- | --- |
| 100 | 1,478 | 102 | 1,528 | 1 |
| 1,000 | 24,712 | 1,204 | 22,540 | 2 |
| 10,000 | 347,652 | 12,788 | 294,742 | 5 |
| 100,000 | 4,475,868 | 128,614 | 3,615,588 | 13 |

(These baselines are reported in the partner cross-reviews and used below as the starting point.)

## 3. Deep Comparison (algorithmic & practical)

Asymptotics: both Min-Heap and Max-Heap implementations have identical asymptotic complexities: O(log n) per insert/extract and O(n log n) for naive sequential builds. Practically, differences are in constant factors driven by:

* • Comparator sign and branch outcomes — min vs max is semantically identical but can affect branch prediction and data-dependent comparisons.
* • Number of swaps and memory writes — implementations that swap per level pay higher write costs; 'hole' (shift) technique reduces writes.
* • Build strategy — sequential inserts (O(n log n)) vs bottom-up heapify (O(n)). Bottom-up frequently yields large constant-factor reductions for bulk loads.
* • Data layout and types — using primitive arrays (int[]) vs boxed objects (Integer/Comparable<T>) dramatically affects memory use, GC pressure, and speed.
* • Instrumentation & logging — detailed metrics collection (PerformanceTracker, println) adds runtime overhead; must be disabled or minimized for release benchmarks.

## 4. Optimization Plan — Aida (Min-Heap)

Summary of actionable optimizations for Aida's Min-Heap (ranked by expected impact):

1. 1) Bottom-up heap construction for bulk loads and merge().

• Change: When combining arrays (merge) or importing large datasets, allocate the combined array and run heapify-down from last non-leaf down to root (indices floor(n/2)-1 down to 0).

• Expected effect: Reduce build time from O(n log n) to O(n). For large n (e.g. 100k) expect comparisons to drop by an order of magnitude in build-dominated workloads (rough estimate: ~4.5M -> ~0.5M comparisons if the baseline is dominated by builds).

1. 2) Replace recursive heapifyDown/heapifyUp with iterative 'hole' (sift) approach.

• Change: Save the element to place (temp), then shift child elements up until the correct slot is found; write temp once.

• Expected effect: Fewer swaps/writes (30–60% fewer assignments depending on distribution), better cache behavior.

1. 3) Fix decreaseKey boundary check and precise comparison counting.

• Change: Correct if (index < 0 || index >= size) and only increment metric counters for logical comparisons (not bounds checks).

• Expected effect: Cleaner tests, slightly lower reported counts for comparisons (instrumentation accuracy improved).

1. 4) Preallocate for merge (avoid temporary doubling) & use bottom-up to rebuild in-place.

• Change: Allocate new array of exact n+m size, copy both arrays, run bottom-up heapify.

• Expected effect: Eliminates temporary memory spikes and extra copies — reduces memory footprint and GC overhead.

1. 5) Provide a primitive-backed specialized version (int/long) for performance-sensitive scenarios.

• Change: When heap values are primitive numeric types, implement an int[] version to avoid boxing/unboxing.

• Expected effect: Large improvements in throughput and reduced GC (typical speedups 20–4x depending on workload and GC pressure).

1. 6) Disable heavy logging during benchmarks; add a 'lightweight' instrumentation mode.

• Change: Toggle detailed metrics collection off for release runs or sample metrics (e.g., sample every k operations).

• Expected effect: Reduce instrumentation overhead (practical speedups 5–20% depending on how heavy tracking is).

## 5. Optimization Plan — Tomiris (Max-Heap)

Tailored optimizations for Tomiris's Max-Heap (ranked by expected impact):

1. 1) Bottom-up build for bulk operations (same rationale as above).

• Expected effect: Dramatic reduction in build comparisons for merge/bulk-load workloads.

1. 2) Growth factor tuning: 1.5× instead of 2× (memory vs resize trade-off).

• Change: Use growthFactor = 3/2 to reduce average unused capacity.

• Expected effect: Reduce peak memory by ~20–30% at the cost of slightly more resizes and array copy work (net runtime impact typically small).

1. 3) Avoid redundant child comparisons & pre-check child existence.

• Change: When heapifyDown compares children, check right child index first for existence and only compare when present.

• Expected effect: Small constant reduction in comparisons (≈3–8%).

1. 4) Iterative heapify + hole method to reduce swaps and call overhead.

• Expected effect: Reduce assignments and eliminate recursion overhead improving throughput by ≈5–15%.

1. 5) Conditional instrumentation & JMH microbench harness for accurate measurement.

• Expected effect: More accurate baselines and reliable before/after comparisons; avoid skew from JVM warmup and GC.

## 6. Estimated Outcomes (quantitative examples)

Using the baseline row N=100,000 as a reference point (Comparisons=4,475,868; Swaps=128,614; ArrayAccesses=3,615,588; Time=13 ms) we estimate the following plausible outcomes when applying the listed optimizations. These are approximate ranges — actual results depend on workload composition (build-dominated vs many mixed ops).

Baseline row source: partner reports (uploaded).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Optimization | Metric(s) Impacted | Estimated % Improvement (range) | Example New Value (100k) - low | Example New Value (100k) - high |
| Bottom-up build (replace sequential inserts) | Comparisons, Time | 70% - 90% | ≈ 450k comps Time ≈ 3–5 ms | ≈ 1,300k comps Time ≈ 6–9 ms |
| Hole (sift) iterative heapify | Swaps, Array writes, Time | 30% - 60% | Swaps ≈ 51k ArrayAccesses ≈ 2.0M | Swaps ≈ 90k ArrayAccesses ≈ 2.8M |
| Iterative vs recursive (call overhead) | Time | 5% - 15% | Time ≈ 11 ms | Time ≈ 12.5 ms |
| Growth factor → 1.5× | Memory (peak), GC | 20% - 30% memory reduction | Peak mem −20% | Peak mem −30% |
| Disable heavy instrumentation | Time | 5% - 25% | Time ≈ 9.8 ms | Time ≈ 12.4 ms |
| Primitive-backed specialized heap | Time, Memory | 20% - 400% (workload dep.) | Time ≈ 3–8 ms | Time ≈ 8–12 ms |

Notes:

* • These estimates assume the run is dominated by the operations optimized (e.g., build-heavy workloads for bottom-up gains). For mixed workloads the improvements scale proportionally to how much time is spent in those operations.
* • Primitive-backed gains are highly workload dependent (boxed → unboxed transitions and GC behavior can dominate).

## 7. Microbenchmark & Validation Plan (recommended)

To validate the improvements empirically, use the following protocol:

1. 1) Use JMH for microbenchmarks (forks, warmup iterations, measurement iterations).
2. 2) Test distributions: random uniform, sorted, reverse-sorted, nearly-sorted, many duplicates.
3. 3) Input sizes: n = 100, 1k, 10k, 100k (same as baseline) and larger if feasible.
4. 4) For each configuration record: time (ms), comparisons, swaps, array accesses, GC logs, and memory peak.
5. 5) Re-run with instrumentation disabled to measure 'production' performance and compare to instrumented results to quantify instrumentation overhead.
6. 6) Save CSVs and plot time vs n (log-scale), comparisons vs n, swaps vs n, and present side-by-side before/after plots.

## 8. Expanded Summary & Recommendations

* 1) Priority list for implementation (quick wins first):

• Fix correctness & counting issues (decreaseKey bounds and comparison counting).

• Replace sequential bulk-load with bottom-up build for merge() and large imports.

• Convert recursive heapify to iterative hole-based implementation.

• Provide primitive-type specializations for performance-critical builds.

• Tune growth factor and disable heavy logging for benchmarks.

2) Expected outcome: After applying these optimizations both implementations should retain identical asymptotic complexity but show substantial practical speedups, reduced memory usage, and more accurate instrumentation metrics. Build-heavy workloads will benefit the most (order-of-magnitude reductions are possible for build passes).

3) Deliverables you can produce now: updated implementation branches, microbenchmark CSVs, before/after plots, and a short 1-page appendix summarizing measured improvements.