Cross-Review Summary: Min-Heap vs Max-Heap Comparison

Pair 4: Heap Data Structures

Student A: Orazbek Ulan - Min-Heap Implementation

Student B: Zhetpyspaev Adilbek - Max-Heap Implementation

1. Algorithm Comparison Overview

Structural Comparison

Aspect	Min-Heap	Max-Heap
Root Property	Minimum element at root	Maximum element at root
Required Operation	decrease-key	increase-key
Additional Operation	merge	-
Primary Use Case	Priority queues (min priority)	Priority queues (max priority)

Both implementations use **array-based representation** with standard parent/child indexing:

- Parent: `(i-1)/2`
- Left child: `2i+1`
- Right child: `2i+2`

2. Time Complexity Comparison

Operation	Min-Heap	Max-Heap	Winner
Insert	Θ(log n)	Θ(log n)	Tie ✓

Extract	Θ(log n)	Θ(log n)	Tie ✓
Peek	Θ(1)	Θ(1)	Tie ✓
Decrease/Increase-Key	Θ(log n)	Θ(log n)	Tie ✓
Build-Heap	Θ(n)	Θ(n)	Tie ✓
Merge	Θ(n+m)	N/A	Min-Heap ✓

Conclusion: Theoretical complexities are **identical** for common operations.

3. Space Complexity Comparison

Component	Min-Heap	Max-Heap
Primary Storage	O(n)	O(n)
Position Map	O(n) HashMap	None
Total Space	**O(n)**	**O(n)**
Trade-off	More memory, faster key operations	Less memory, slower key lookup

Winner: Max-Heap uses less memory, but Min-Heap has better API usability.

4. Implementation Quality Comparison

Min-Heap Strengths

- **Value-based decrease-key** (user-friendly API)
- **Position tracking** with HashMap (O(1) contains check)
- **Dynamic capacity** with automatic resizing
- **Merge operation** implemented
- **Comprehensive benchmarking** (all operations)

- **Duplicate detection** for data integrity

Max-Heap Strengths

- **Lower memory footprint** (no HashMap)
- **Simpler implementation** (fewer data structures)
- **CSV export** for metrics
- **Clean, readable code**

Min-Heap Weaknesses

- Higher memory usage due to HashMap
- More complex code structure

Max-Heap Weaknesses

- **Index-based increase-key** (API design flaw)
- **Fixed capacity** (no dynamic resizing)
- **heapSort modifies original array** (side effects)
- **No duplicate detection**
- **Limited benchmarking** (only heapSort tested)

5. Performance Comparison (Empirical)

Benchmark Results

Test Configuration: Random data, Java 16, averaged over 5 runs

Size	Min-Heap Insert (ms)	Max-Heap Sort (ms)	Ratio
100	~0.05	0.127	2.5×
1,000	~0.30	0.248	0.8×
10,000	~4.50	1.366	0.3×
100,000	~60.00	9.975	0.17×

Note: Direct comparison difficult due to different operations tested.

Comparisons per Operation

Size	Min-Heap (insert)	Max-Heap (heapSort)
100	~6.5/op	~10.4/op
10,000	~13.2/op	~23.5/op
100,000	~16.8/op	~30.2/op

Analysis: Both follow O(log n) pattern, Max-Heap has higher constants due to heapSort overhead.

6. Optimization Recommendations

For Min-Heap:

1. **Optional**: Provide lightweight version without HashMap for memory-constrained scenarios
2. Improve documentation with complexity guarantees
3. Add more edge case tests

For Max-Heap (Critical):

1. **Implement value-based increase-key** with position tracking
2. **Add dynamic resizing** for scalability
3. **Fix heapSort side effects** (use array copy)
4. **Add comprehensive benchmarks** for all operations
5. Add duplicate detection

Conclusion

Both implementations demonstrate **solid understanding** of heap data structures with correct algorithmic complexity.

Min-Heap excels in **usability and feature completeness**, making it more suitable for production use.

Max-Heap excels in **simplicity and memory efficiency**, making it good for educational purposes and memory-constrained environments.

Recommendation: Combine best aspects - Min-Heap's API design with Max-Heap's simplicity.