Algorithm Project — Pair 4: Heap Data Structures

Student A: MinHeap Implementation (decrease-key, merge)
Student B: MaxHeap Implementation (increase-key, extract-max)

Peer Analysis Report — MinHeap (by Student A)

Reviewer: Student B
Analyzed file: MinHeap.java
Supporting class: PerformanceTracker.java

---

1. Algorithm Overview

The implementation represents a Min-Heap data structure based on a complete binary tree, where each parent node is smaller than its children. The minimum element is always located at the root (heap[0]).

Implemented operations:

- insert(int key)
- extractMin()
- decreaseKey(int i, int newVal)
- merge(MinHeap other)

Helper methods include heapifyUp, heapifyDown, swap, and printHeap.

---

2. Complexity Analysis

Operation	Best Case	Average Case	Worst Case	Description
insert()	Θ(1)	O(log n)	O(log n)	Inserts an element and restores heap property using percolate-up.
extractMin()	Θ(1)	O(log n)	O(log n)	Removes the smallest element and restores heap property.
decreaseKey()	Θ(1)	O(log n)	O(log n)	Decreases the value and repositions the element upward.
merge()	O(n + m)	O(n + m)	O(n + m)	Sequentially inserts all elements from another heap.

Space Complexity: O(n) total, with an optional O(log n) recursive call depth in heapifyDown.

---

3. Code Quality Evaluation

Strengths:

- The implementation is well-structured, modular, and adheres to object-oriented principles.
- Proper error handling is included (IllegalArgumentException, IllegalStateException).
- Integration with PerformanceTracker provides detailed operation metrics.
- Naming conventions and documentation are consistent and clear.

Weaknesses and Recommendations:

1. The merge() method is implemented using sequential insertions, which results in O((n + m) log(n + m)) complexity instead of O(n + m). A more efficient approach would rebuild the heap in a single pass.
2. The addMemoryAllocation() method is invoked multiple times unnecessarily. It should only be called during actual memory expansion.
3. The heapifyDown() method is recursive and could be rewritten iteratively to avoid stack overhead.
4. The PerformanceTracker method addArrayAccess() is not utilized and should be integrated for more precise tracking.
5. Additional unit tests are recommended for edge cases, such as empty heaps and duplicate elements.

---

4. Empirical Results (Expected Behavior)

Input Size (n)	Average Execution Time (ms)	Comparisons	Swaps	Memory Allocations
100	~0	~350	~180	~100
1,000	~1	~4,000	~2,100	~1,000
10,000	~3	~47,000	~24,500	~10,000

The time complexity follows the expected logarithmic trend O(n log n).
The number of swaps and comparisons scales predictably, while memory usage grows linearly with input size.

---

5. Comparison with MaxHeap

Aspect	MinHeap (Student A)	MaxHeap (Student B)
Heap Property	Parent < Child	Parent > Child
Key Update Operation	decreaseKey()	increaseKey()

Aspect	MinHeap (Student A)	MaxHeap (Student B)
Merge Function	Implemented	Not required
Main Operation	extractMin()	extractMax()

Both implementations share identical asymptotic complexities and structural logic.
The MinHeap implementation provides an additional merge() feature, which introduces an opportunity for further optimization.

---

6. Conclusion

The MinHeap implementation is correct, logically consistent, and well-documented.
It demonstrates a solid understanding of heap operations and fulfills the project's structural and analytical requirements.

Evaluation Summary:

- **Correctness:** 10 / 10
- **Readability:** 9 / 10
- **Performance:** 8 / 10
- **Testing Coverage:** 7.5 / 10
- **Overall Quality:** 8.6 / 10

Prepared by: Student B
Report Title: Peer Review of Student A's MinHeap Implementation
Project: Algorithmic Analysis — Pair 4 (Heap Data Structures)