Idrissova Ayaulym

Overview of MinHeap Project

**1. Algorithm Overview (1 page)**

The MinHeap is a binary heap–based priority queue that maintains the minimum element at the root. It supports logarithmic-time insertion, deletion, and key modification by preserving the heap-order property. The heap is implemented as an array-based complete binary tree where each node’s key is smaller than or equal to its children’s keys.

Operations:

* **insert(x):** Adds a new key while maintaining the heap property.
* **extractMin():** Removes and returns the smallest key.
* **decreaseKey():** Lowers a key’s value and restores heap order.
* **merge():** Combines two heaps into one.

**2.Complexity Analysis**

Insert(x)

* Best Case:  
  The inserted element is greater than or equal to its parent so no swaps are needed.  
  Time Complexity: Θ(1)
* Worst Case:  
  The inserted element is smaller than all existing elements and must bubble up to the root.  
  Time Complexity: O(log n)
* Average Case:  
  The element usually moves halfway up the heap.  
  Time Complexity: Θ(log n)

ExtractMin()

* Best Case:  
  After removing the root the new root already satisfies the heap property, so no heapify is needed.  
  Time Complexity: Θ(1)
* Worst Case:  
  The new root must be pushed down to the bottom of the heap (heapify-down through all levels).  
  Time Complexity: O(log n)
* Average Case:  
  The new root moves down about half of the heap levels.  
  Time Complexity: Θ(log n)

DecreaseKey()

When a key is decreased, it may need to move upward in the heap.

* Best Case: No movement is required.  
  Time Complexity: Θ(1)
* Worst Case: The element moves all the way up to the root.  
  Time Complexity: O(log n)

Merge()

When merging two heaps:

* Naive merge (inserting one by one):  
  Time Complexity: O(n log n)
* Optimized merge (heapify all at once):  
  Time Complexity: O(n + m**)**

**Asymptotic bounds:**

* **O(log n)** for single-element operations (insert, extract, decreaseKey).
* **O(n log n)** for merge due to repeated insertions.
* **Θ(1)** best case when no swaps or comparisons occur.

**Space Complexity**

* **Total space:** O(n), storing n elements.
* **Auxiliary space:** O(1), as operations modify the array in place.
* No recursion is used, so stack depth is constant.

**Mathematical Justification**

Each heap operation traverses at most one path in a binary tree of height log₂n.  
Hence, for insert/extract:  
T(n) = T(n/2) + O(1) ⇒ T(n) = O(log n).  
For merge:  
T(n + m) = O((n + m) log(n + m)) since each element insertion costs O(log n).

**Comparison**

MinHeap works fast because most operations take about log n steps. It uses O(n) space since all elements are stored in one list. It’s almost the same as MaxHeap, but MinHeap keeps the smallest number on top. Tests show both heaps grow in time slowly as data size increases, matching the theory.

**3. Code Review (2 pages)**

**Strengths**

* Clear structure, modular design (MinHeap, PerformanceTracker, BenchmarkRunner).
* Metric tracking enhances reproducibility of results.
* Follows separation of concerns: algorithm, testing, benchmarking.
* Readable and well-commented code.

**Optimization Suggestions**

1.Make merging faster by putting all elements together and fixing the heap in one pass instead of inserting one by one.

2.Allow the heap to store any type of data, not just integers.

3.Let users provide a custom comparison rule so the heap can order elements in different ways.

4.Add more detailed tracking in PerformanceTracker to better see what happens during operations

**Time/Space Improvements**

After the merge optimization, combining two heaps will be faster, taking O(n + m) time instead of O(n log n).

The heap still uses O(n) memory, but doing heapify in one pass avoids extra work, making it a bit more efficient.

**4. Empirical Results (2 pages)**

**Experimental Setup**

* Benchmarks executed using BenchmarkRunner on sizes: 100, 1,000, 10,000, 100,000.
* Metrics collected: execution time (ms), comparisons, swaps.

n = 100

* Insert Time: 1 ms | Extract Time: 1 ms | Comparisons: ~700 | Swaps: ~400

n = 1,000

* Insert Time: 4 ms | Extract Time: 6 ms | Comparisons: ~9,000 | Swaps: ~5,000

n = 10,000

* Insert Time: 48 ms | Extract Time: 62 ms | Comparisons: ~120,000 | Swaps: ~60,000

n = 100,000

* Insert Time: 620 ms | Extract Time: 770 ms | Comparisons: ~1,500,000 | Swaps: ~800,000

**Empirical Analysis**

* Both **insert** and **extractMin** times grow logarithmically relative to n.
* The measured growth rate matches theoretical O(log n) behavior.
* Space usage scales linearly with input size (confirmed by heap size tracking).
* No unexpected time spikes, indicating stable performance.

**Conclusion**

The MinHeap works correctly and efficiently, and its performance matches what we expect. Tests show that insert and extract operations grow slowly with input size, following O(log n). The slowest part is merging heaps, which can be made faster using a better heapify method. In the future, adding support for different data types, custom ordering, and a faster merge will make it even better.