Project Phase 3 Deliverable 3: Optimization, Scaling, and Final Evaluation

Bereket Gebremariam, Sachin Karki

Dr. Satish Penmatsa

2025 Fall - Algorithms and Data Structures (MSCS-532-M80)

University of the Cumberland

## **Introduction**

Phase 2 of the TaskScheduler used a heap to order tasks by due date and urgency and a dictionary for O(1) lookups. Although functional, it became slower for large task sets due to repeated string date comparisons and inefficient cleanup after task completion.  
  
Phase 3 focuses on optimization, scaling, and performance testing. The main changes include replacing string dates with numeric timestamps, implementing a lazy deletion flag, and adding robust input validation. Benchmarks show that it handles 1,000 tasks in under 0.01 seconds and uses ~1 MB of memory — a 5–10× speed improvement over Phase 2. The system is now more scalable and stable under stress tests.

## **Optimization Techniques**

Problems Identified in Phase 2:

• Date comparisons used strings like '2025-10-28', which are lexicographically slow.  
• Removing completed tasks left junk heap entries that accumulated over time.

Key Optimizations:

1. Numeric Dates (Timestamps): Converted dates to Unix timestamps for faster numeric comparisons and lower memory footprint.



**Fig.** Adding Task

2. Lazy Deletion: Introduced an is\_deleted flag instead of direct heap removal, skipping invalid entries when accessed.  
3. Input Validation: Blocked duplicate IDs, invalid dates, and negative urgency values.  
4. Update Feature: Allowed updates to task urgency or deadline without rebuilding the heap.  
  
**Result**: All core operations maintain O(log n) time. Average execution for basic operations dropped from ~0.005 s to 0.002 s (≈ 60% improvement).

A computer screen shot of a program

AI-generated content may be incorrect.

**Fig.** Basic Testing

## **Scaling Strategy**

**Challenges with Large Inputs:**• String dates added incremental delays as n grew.  
• Deleting many tasks quickly caused heap fragmentation.  
  
**Solutions:**  
• Timestamps cut date comparison time by ~50%.  
• Lazy Deletion reduced heap cleanup overhead.  
• O(n) Memory: The heap and hash table together use linear space (~1 MB for 1,000 tasks). Scaling to 10,000 tasks used ~11 MB and took ~0.08 s.

|  |  |  |
| --- | --- | --- |
| **Tasks** | **Time (s)** | **Memory (MB)** |
| 100 | 0.0009 | ~0.1 |
| 1,000 | 0.0083 | ~1.13 |
| 10,000 | 0.0800 | ~11.0 |

## **Testing and Validation**

Testing covered basic operations, edge cases, and stress tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Type** | **Description** | **Result** | **Time (s)** | **Memory (MB)** |
| Basic | Add/find/complete 3 tasks | Correct priority selection | 0.0020 | 0.0000 |
| Edge Cases | Bad date, duplicate ID, negative urgency | Caught errors, no crash | 0.0002 | 0.0000 |
| Stress 100 | Random dates/urgency | All completed successfully | 0.0009 | 0.0000 |
| Stress 1000 | Larger dataset | Scales linearly | 0.0083 | 1.1328 |
| Stress 10000 | Extreme load test | Stable, no memory leaks | 0.080 | 11.0 |

A screenshot of a computer program

AI-generated content may be incorrect.

**Fig.** Output from Stress Testing

A graph of a line and a line

AI-generated content may be incorrect.

**Fig.** Performance Metric for Stress

**Observation**: Execution time grows logarithmically with task count while memory scales linearly.

## **Performance Analysis**

Compared to Phase 2, Phase 3 achieves 5–10× faster operation times for all task management functions.  
• Basic operations: 0.002 s vs 0.005 s  
• 100 tasks: 0.0009 s vs 0.02 s  
• 1,000 tasks: 0.008 s vs 0.05 s  
  
The heap maintains O(log n) efficiency for insertion and retrieval, while lazy deletion minimizes the cost of removing obsolete tasks.  
  
**Trade-offs:**  
• Slight space overhead from retaining old entries.  
• Small timestamp conversion cost on insert, offset by faster runtime.  
  
Performance Plot: Runtime vs. task count follows a logarithmic trend, confirming good scalability for medium data sizes.

## **Final Evaluation**

Strengths:  
• Executes 1,000 tasks in < 0.01 s with minimal memory.  
• Stable against invalid inputs and duplicate entries.  
• Scales smoothly for moderate workloads.  
  
**Limitations:**  
• Performance may decrease if tasks are rapidly added and deleted in bursts.  
• No persistent storage or multi-user support yet.

References

Birler, T., Mühlbauer, P., & Kaden, M. (2023). Simple, Efficient, and Robust Hash Tables for Join Processing. *Technische Universität München*. Retrieved from <https://db.in.tum.de/~birler/papers/hashtable.pdf>.

Cai, X. (2022). Concurrent Extensible Hash Table (Project Report). *GitHub Pages*. Retrieved from <https://xinzhu-cai.github.io/418project.github.io/Report.pdf>.

Li, Y., & Li, R. (2024). Performance Evaluation of Separate Chaining for Concurrent Hash Maps. *MDPI*, *13*(17), 2820. <https://www.mdpi.com/2227-7390/13/17/2820>.

Srivastava, A., & Gupta, P. (2025). The Bathroom Model: A Realistic Approach to Hash Table Algorithm Optimization. *arXiv*. Retrieved from <https://arxiv.org/html/2502.10977v1>.