# 

# 

# 

# 

# 

# 

# 

# Project Phase 1 Deliverable 1: Data Structure Design and Implementation

# Bereket Gebremariam, Sachin Karki

# Dr. Satish Penmatsa

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

# University of the Cumberland

# October 24th, 2025

# 

# 

# 

# 

Project: **Task Scheduler Program**

**Objective**

The primary objective of this project is to design and implement a task scheduling system. The input data includes the task title, task deadline, task priority, and task complexity. The system efficiently manages and retrieves task priorities based on priority factors used and leverages optimized data structures in Python.

This project aims to demonstrate how theoretical knowledge of data structures and algorithmic efficiency can be applied to a realistic, command-line-based scheduling scenario.

**Application Context**

In today's world, we are overwhelmed with multiple tasks. It will be challenging to manage all the tasks simultaneously. Our job is to ensure we prioritize tasks correctly, whether at work, school, or in our personal lives, for an efficient workflow. A Scrum team or any project can utilize this, such as building a house or completing a project. We can divide smaller tasks and prioritize them according to the project milestones. If there are 100's of functions, then our job is to prioritize them to achieve ultimate efficiency.

**We believe an efficient scheduler must:**

● Prioritize urgent or high-value tasks first.

● Support dynamic updates for deadline changes in the functions (since anything can change at any moment).

● Help streamline task completion one-by-one.

**Goals**

● Implement a priority queue (min-heap) for efficiently ranking and retrieving the most urgent or highest-value tasks.

● Implement a hash table for fast O(1) lookups and updates of task details. Much more efficient.

● Design a simple command-line interface (CLI) for task addition, retrieval, and removal.

● Evaluate optimization strategies and trade-offs in terms of space and time complexity.

**Key Data Structures**

| **Data Structure** | **Purpose** | **Time Complexity** | **Description** |
| --- | --- | --- | --- |
| Priority Queue (Heap) | To maintain tasks sort by priority | Insert: O(log n) Remove: O(log n) | Built using Python’s inbuilt data structure to ensure efficient scheduling based on deadline. |
| Hash Table (Dictionary) | To store and retrieve tasks by ID | Lookup: O(1) Insert/Delete: O(1) | Used to store metadata (task ID, deadline, urgency). |

### 

### **Design Rationale**

Heap-based Priority Queue: Enables efficient retrieval of the highest-priority task (smallest deadline or highest bid) in O(\log n) time.

**Hash Table:** Provides instant access to task details, essential for quick command-line lookups.

**Tie-breaking Rule:** When multiple tasks share the same deadline, priority is determined by bid amount and, if necessary, task ID for determinism.

**Implementation Overview**

The project uses Python’s built-in heapq for priority operations and a dictionary for hash-based storage.

**CLI Commands:**

● add\_task <id> <deadline> <urgency> <description>

● get\_next\_task – retrieves the highest-priority task

● find\_task <id> – displays task info

● complete\_task <id> – marks a task as completed

### **Challenges and Limitations**

The hybrid design offers excellent performance for core operations but introduces specific complexities in dynamic data handling and inherent memory overhead.

#### 

#### **Dynamic Priority Updates and Lazy Deletion**

Updating an existing task’s priority (e.g., changing bids or deadlines) is a non-trivial operation because Python’s native heapq implementation lacks an efficient decrease-key or increase-key function, which would allow the heap property to be locally restored in $O(\log n)$ time (Cormen et al., 2022). More complex heap structures designed for dynamic updates are known, but add implementation complexity (Ioannou & Katevenis, 2012; Sintoni et al., 2014).

Therefore, the system must rely on the Lazy Deletion workaround:

1. When a task's priority changes, the old, now-outdated entry remains in the heap. The hash table is immediately updated with the new priority values.
2. The updated task is then reinserted into the Min-Heap with its new priority key in $O(\log n)$ time.
3. During the get\_next\_task extraction phase, outdated (or "stale") entries are efficiently skipped using a validation check against the Hash Table.

This approach maintains the system's correctness and ensures an amortized $O(\log n)$ performance for updates and retrieval, but it temporarily increases the size of the heap and introduces minor, cyclical extraction overhead due to the required hash table validation.

#### **Deterministic Tie-Handling**

The system's multi-attribute priority key ((deadline, -bid\_amount, task\_id)) is crucial for ensuring predictable scheduling. Tasks with identical deadlines and bids require a final, stable sorting mechanism to guarantee a deterministic extraction order. The inclusion of the unique task\_id as the final element serves as this deterministic tiebreaker. This is vital, as arbitrary ordering in real-time or critical systems can lead to unpredictable behavior. Ensuring determinism in complex sorting scenarios is a recognized requirement in algorithmic design (Sedgewick & Wayne, 2011).

#### **Space-Time Trade-off and Memory Consumption**

The hybrid model necessitates the duplication of key data (Task ID, deadline, bid) across both core structures—metadata in the Hash Table and the priority information in the Min-Heap. This increases the overall memory footprint compared to a single, monolithic data structure. However, this is a conscious space-time trade-off that is entirely justified. The constant-time $O(1)$ access provided by the Hash Table for lookups (Knuth, 1998) and the $O(\log n)$ efficiency of the heap for priority updates (Cormen et al., 2022) ensures responsiveness. This trade-off is often favored in real-time or dynamic systems where immediate responsiveness is paramount.

### **Expected Outcomes**

The successful completion of this project is defined by the fulfillment of both functional and analytical objectives, resulting in a fully justified and operational command-line scheduler.

#### **Functional Deliverables**

● Operational CLI: A fully functional, text-based command-line interface supporting all core task management commands: add\_task, get\_next\_task, find\_task, and complete\_task.

● Hybrid Data Integrity: The final Python class (TaskScheduler) must demonstrate synchronized, concurrent management, effectively linking task metadata in the Hash Table with priority keys in the Min-Heap.

● Robust Prioritization: The system must consistently deliver accurate task ordering based on the multi-level logic: Earliest Deadline First $\rightarrow$ Highest Bid $\rightarrow$ Unique Task ID.

#### **Analytical Deliverables**

● Performance Analysis: An empirical comparison of the heap-based approach versus a list-based sorting approach for varying task volumes (e.g., 100, 1000 tasks). This analysis will provide practical validation of the theoretical $O(\log n)$ complexity for the priority queue operations.

● Trade-off Documentation: A clear written discussion documenting the precise nature of the space-time efficiency balance, justifying the chosen memory overhead against the guaranteed constant-time access benefits of the Hash Table.

● Implementation Details: Comprehensive inclusion of pseudocode for all core operations and annotated Python snippets detailing the construction of the complex heap key logic, the deterministic tie-breaking mechanism, and the implementation of the lazy deletion strategy.

### 

### **References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Ioannou, A., & Katevenis, M. (2012). Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. IEEE/ACM Transactions on Networking (TON), 20(4).

Knuth, D. E. (1998). The Art of Computer Programming, Vol. 3: Sorting and Searching (2nd ed.). Addison-Wesley Professional.

Liu, C. L., & Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM (JACM), 20(1), 46–61.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.

Sintoni, M., Bonfà, E., & Davoli, F. (2014). An Efficient Data Structure for Dynamic Priority Queue Management. 2014 IEEE 10th International Conference on Network and Service Management (CNSM).