Deliverable 2: Proof of Concept Implementation

Bereket Gebremariam, Sachin Karki

Dr. Satish Penmatsa

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

University of the Cumberland

**Hash Table (Dictionary) Proof-of-Concept Implementation**

This section provides a clear explanation of how the hash table is used within the TaskScheduler class to store and manage task metadata efficiently. The hash table serves as the primary repository for task information, playing a crucial role in ensuring rapid access and updates to task data.

The core purpose of the hash table is to support three fundamental operations: inserting new tasks, retrieving task information, and removing completed tasks. These operations are implemented through the add\_task, find\_task, and complete\_task methods in the TaskScheduler class. Because the hash table is implemented using Python’s dictionary data structure, each of these operations executes in constant time, or O(1), on average. This performance consistency is crucial, especially when managing numerous tasks in real-time environments.

In the hash table, the unique task\_id is used as the key. In contrast, the value associated with each key is a dictionary containing metadata about the task, including the deadline, urgency level, and a brief description. This design ensures that the stored metadata is well-structured, easy to understand, and simple to extend should additional attributes need to be included later.

To demonstrate the effectiveness of the hash table, a test script was used to showcase insertion, lookup, and deletion operations, including the handling of edge cases. For example, inserting tasks such as T101 and T102 confirms that data is stored correctly. A lookup for T102 returns its details instantly, highlighting the speed of the hash table. Attempts to access a non-existent task, such as T999, confirm that appropriate error messages are displayed. Additionally, deleting a task and attempting to delete it again verifies that duplicate deletions are managed gracefully.

A computer screen shot of text

AI-generated content may be incorrect.  
**Fig.** Hash Table Implementation Output

The hash table also plays a vital role in the overall design of the task scheduling system, particularly when integrating future enhancements. As the system expands to include components such as a priority queue implemented using a min-heap, the hash table will serve as the single source of truth for determining whether task entries in other structures are valid or outdated. This capability is essential for supporting a lazy deletion strategy, where tasks in other structures may not always be immediately removed.

From a development perspective, several important considerations guided this implementation. Ensuring data consistency required designating the hash table as the authoritative source for all task metadata. Structuring the stored metadata as a descriptive dictionary made the implementation more intuitive while supporting future growth. Error handling was also incorporated to help maintain system stability and provide users with meaningful feedback.

To complete the full system implementation, the following steps include integrating the hash table with a min-heap to support priority-based scheduling, implementing lazy deletion logic to manage outdated records efficiently, constructing a multi-attribute priority key to ensure accurate task ordering, and developing a command-line interface to support user interaction. By following these steps, the task management system will evolve into a robust, efficient, and user-friendly application.

**Priority Queue (Heap) Integration**

Following the initial hash table proof-of-concept, a priority queue was integrated using Python’s built-in heapq module. This enhancement enables the system to manage task prioritization efficiently based on deadlines and urgency levels. The heap ensures that the task with the earliest deadline and highest urgency is always accessible in O(1) amortized time, while insertion and removal operations maintain O(log n) performance.

Each task is represented in the heap as a tuple: (deadline, -urgency, task\_id). This composite key enforces the priority rules: earliest deadline first, highest urgency next, and finally, task ID for stability in the event of a tie.

A screenshot of a computer

AI-generated content may be incorrect.

**Fig.** Priority Queue (Heap) and Heap TableImplementation Output

**Data Structure Summary**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Structure** | **Purpose** | **Insert** | **Remove** | **Lookup** |
| Priority Queue (Heap) | Maintain tasks sorted by priority | O(log n) | O(log n) | O(1) (peek) |
| Hash Table (Dictionary) | Store and retrieve task metadata | O(1) | O(1) | O(1) |

Together, these data structures create a balanced and efficient scheduling system. The hash table provides direct access to metadata, while the heap maintains order by task priority. A lazy deletion strategy ensures that heap entries referencing deleted tasks are ignored automatically during subsequent operations.

**Implementation Challenges and Next Steps**

During the integration process, several challenges were encountered. Managing synchronization between the hash table and priority queue required careful coordination to maintain data consistency. This was resolved using a lazy deletion mechanism, where outdated heap entries are cleaned up only when encountered.

Another challenge involved defining a robust multi-attribute sort key. The chosen structure (deadline, urgency, task\_id) ensures consistent ordering even under complex scenarios with identical deadlines or urgencies. Error handling was also reinforced to manage edge cases, such as empty queues and invalid task lookups.

Next steps involve extending this implementation to support updating the priorities of existing tasks, integrating a user interface, and implementing persistent storage for long-term task tracking. The project will also add more extensive testing coverage to ensure reliability across different workloads.

**Demonstration and Testing**

The test script demonstrates all core functionalities of the TaskScheduler class. It includes five structured test cases that validate the insertion, lookup, prioritization, and deletion mechanisms:

1. Adding multiple tasks and confirming successful storage.

2. Retrieving the highest-priority task (earliest deadline and highest urgency).

3. Finding tasks by ID, including error handling for invalid IDs.

4. Completing tasks sequentially based on their calculated priority order.

5. Handling edge cases, such as attempts to retrieve tasks when the queue is empty.

The observed output confirmed that the system performed as expected, correctly ordering tasks and handling errors gracefully. Each operation’s performance matched theoretical time complexities, validating the efficiency of the underlying data structures.

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). The MIT Press.

Ioannou, A., & Katevenis, M. (2006). Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking*, *14*(3), 540–553.

Knuth, D. E. (1998). *The art of computer programming, volume 3: Sorting and searching* (2nd ed.). Addison-Wesley Professional.

Pagh, R., & Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms*, *51*(2), 122–144.