**Hash Table (Dictionary) PoC Implementation**

The hash table will be contained within a TaskScheduler class, serving as the task\_metadata repository.

**1. Partial Implementation of Data Structure**

The Hash Table must handle the core operations: **Insert (add\_task)**, **Lookup (find\_task)**, and **Delete (complete\_task)** .

**Python Implementation Snippet: Task Metadata Storage**

Task Metadata Storage Implementation

The core component for fast task detail access is the Hash Table, implemented in Python as the dictionary self.task\_metadata within the TaskScheduler class. This structure ensures $O(1)$ constant-time performance for all primary metadata operations111. The dictionary uses the unique task\_id as the key and stores the full task metadata (deadline, urgency, description) as the value, fulfilling the project's requirement to store all necessary task details2. Essential methods are included for this proof of concept: add\_task\_metadata handles the $O(1)$ insertion of new tasks; find\_task\_metadata performs the $O(1)$ lookup to retrieve task information; and complete\_task\_metadata executes the $O(1)$ deletion of a task's record 333. This design makes the Hash Table the primary source of truth for the latest task information, crucial for the later implementation of the Lazy Deletion strategy4.

**2. Demonstration of Key Operations**

A simple script will demonstrate the $O(1)$ operations—Insert, Lookup, and Delete—and confirm the hash table's integrity.

**Test Script: Basic Operations and Edge Cases**

Demonstration Script and Edge Cases

The accompanying test script provides a practical demonstration of the Hash Table's functionality. The script first performs basic insertion (Demo 1) for tasks like T101 and T102, confirming the successful storage of metadata. It then showcases the lookup operation (Demo 2), instantly retrieving details for existing tasks, which is the key benefit of the hash table's $O(1)$ access5. The script also includes necessary edge case testing: it attempts a lookup for a non-existent task (T999) and a double deletion attempt (T101) to validate the included error handling mechanisms and confirm the robustness of the insertion, lookup, and deletion methods. The clean execution of these tests validates that the Hash Table is operating correctly as a fast, reliable task metadata repository.

**Expected Output**

The expected output shows that the task details are correctly stored, instantly retrieved, and cleanly removed.

--- DEMO 1: Task Insertion (add\_task) ---

Metadata added for Task ID: T101

Metadata added for Task ID: T102

Metadata added for Task ID: T103

Current tasks in Hash Table: ['T101', 'T102', 'T103']

--- DEMO 2: Task Lookup (find\_task) ---

Details for T102: {'deadline': 5, 'urgency': 100, 'description': 'Fix critical bug'}

Details for T999: Error: Task ID 'T999' not found.

--- DEMO 3: Task Deletion (complete\_task) ---

Metadata deleted for Task ID: T101

Current tasks in Hash Table after T101 deletion: ['T102', 'T103']

Error: Task ID 'T101' not found.

**3. Documentation of Implementation Process**

**Challenges and Solutions**

| **Challenge** | **Solution** |
| --- | --- |
| **Data Synchronization** | The Hash Table is implemented to be the **sole source of truth** for the latest task metadata, including deadlines and urgency4. This is crucial for the **Lazy Deletion** strategy, where the Hash Table's current data is used to validate potentially stale entries from the Min-Heap5. |
| **Data Structuring** | The value in the hash map was defined as a dictionary ({'deadline': ..., 'urgency': ..., 'description': ...}) to be **self-documenting** and easily extensible, adhering to the requirement for storing task title, deadline, priority, and complexity6. |
| **Error Handling** | Simple checks were added for missing or duplicate task\_id during lookups, insertions, and deletions to ensure robustness and prevent unexpected application failures7. |

**Next Steps for Full Implementation**

1. **Integrate Min-Heap**: Implement the Priority Queue (using heapq) and integrate its operations (add\_task to the heap, get\_next\_task) with the existing Hash Table8888.
2. **Implement Lazy Deletion**: Code the logic for update\_task (which re-inserts into the heap and updates the Hash Table) and the validation check during get\_next\_task extraction9999.
3. **Implement Multi-Attribute Priority Key**: Build the complex tuple key (deadline, -urgency, task\_id) for the Min-Heap to ensure robust and deterministic prioritization10101010.
4. **Complete CLI**: Flesh out the command-line interface logic to parse user input and call the class methods111111.

**4. Code Quality and Best Practices**

The code uses a **class-based, modular structure** (TaskScheduler) 12 and incorporates:

* **Meaningful Variable Names** (e.g., task\_metadata, deadline).
* **Docstrings/Comments** to explain the purpose of methods and link them to the project's complexity analysis (e.g., $O(1)$)13131313.
* **Simple Error Handling** to make the code more robust and user-friendly by catching common issues like trying to delete a non-existent task14.