Project(1) Task Scheduler Report

Team names and IDs:

حازم جمال قطب احمد الماحى	1404-3-374
عمر اسامه البابلي	1404-5-009
رجب حسن عبدالمنعم محمد	1404-3-288
ضياء الدين محمد حسن داوود	1404-3-344

1. Introduction (Problem Description)

In this project, we implemented a simple Task Scheduler using basic data structures.
The scheduler accepts tasks from the user, stores them inside a queue, and when tasks are executed, they are moved to a history list. To make searching for tasks fast, we used a hash table with chaining. The idea is not to build a perfect production scheduler, but to practice linked lists, queues, hashing, and handling collisions.

Each task stores:

- a unique job ID,
- the time it was submitted,
- a status (queued or executed),
- and the execution timestamp.

We also added a JSON save/load feature so the scheduler can restore its state.

---

2. System Overview

The scheduler consists of three main data structures:

2.1 Linked List Queue

The queue is implemented using a singly linked list.
New tasks are added to the end using `enqueue`, and executed tasks are removed from the front using `dequeue`.

2.2 Hash Table (Chaining)

To check for duplicates quickly and to allow fast task lookup, We used a simple hash table where each bucket is a Python list. The hash function is just `job_id % table_size`. If two job IDs map to the same bucket, they are stored in the same list, chaining using a linked list.

2.3 History Linked List

Executed tasks are stored in another singly linked list in the order they finished.
This helps generate a clear execution log and allows us to load/save history.

2.4 Key components:

- LinkedQueue: FIFO queue of Job objects.
- HashTable: Separate chaining buckets for duplicate checks and fast lookup.
- HistoryList: Linked list of executed jobs with timestamps.
- Scheduler: Orchestrates submission, execution, saving, and loading.

3. Design Decisions

We made a few choices for simplicity:

- The hash table stores **only queued tasks**, not executed ones.
  This keeps the structure smaller and cleaner.
- Duplicate job IDs are **not allowed**. If the user tries to submit a task with an existing ID, an error is raised.
- Execution is simulated by printing a message and stamping the current time.
- Timestamps are saved as ISO-formatted strings to make JSON output readable.

---

4. How the Scheduler Works

Submitting a Task

1. Check the hash table and the history to make sure the ID was never used.
2. Create a `Job` object.
3. Add the job to the queue.
4. Insert the job into the hash table.

Executing a Task

1. Remove a job from the front of the queue.
2. Remove it from the hash table.
3. Stamp the execution time and change its status.
4. Append it to the history list.

Saving and Loading

The entire queue and history can be saved into a JSON file.
When loading, the scheduler reconstructs the queue and hash table exactly as they were.

---

5. Complexity Analysis

Operation	Time Complexity	Space Complexity	Notes
Queue enqueue	O(1)	O(1) extra	Insert at tail of linked list
Queue dequeue	O(1)	O(1) extra	Remove head node
History append	O(1)	O(1) extra	Insert at tail
Hash insert	O(1) average, O(n) worst	O(1)	Chaining: constant expected time
Hash search	O(1) average, O(n) worst	O(1)	Depends on collisions in bucket
Hash remove	O(1) average, O(n) worst	O(1)	Removing from bucket list
Submit task	O(1)	O(1)	Hash check + enqueue
Run next task	O(1)	O(1)	Dequeue + hash remove + history append
Run all tasks	O(n)	O(1) per task	Performs run_next_task repeatedly

Test Cases

1.Insertion (Insertion Logic & Data Integrity)

This class verifies the entry point of the Scheduler system. It ensures that when `submit_task` is called, the job is correctly added to both the **Task Queue** (for processing) and the **Hash Table** (for fast lookup) simultaneously. It also validates data integrity by confirming that the system raises an error when attempting to insert duplicate Job IDs.

Test Case

```python
class TestInsertion(BaseLoggedTest):
    def test_scheduler_submit_inserts_into_hash_and_queue(self) -> None:
        """Verify submitting a task inserts into both hash table and queue"""
        s: Scheduler = Scheduler(hash_size=5)
        job: Job = s.submit_task(job_id=42)
        # Hash search should find it
        self.assertIsNotNone(obj=s.hash.search(job_id=42))
        self.assertEqual(first=s.hash.search(job_id=42).job_id, second=42)
        # Queue head should be the same job
        q_list: list[Any] = s.queue.to_list()
        self.assertEqual(first=len(q_list), second=1)
        self.assertEqual(first=q_list[0].job_id, second=job.job_id)
        log_success(message="submit_task inserted Job(42) into hash and queue; queue head matches the inserted job")

    def test_duplicate_insertion_raises(self) -> None:
        """Ensure duplicate job IDs cannot be inserted into the scheduler"""
        s: Scheduler = Scheduler(hash_size=5)
        s.submit_task(job_id=7)
        with self.assertRaises(ValueError):
            s.submit_task(job_id=7)
        log_success(message="duplicate insertion raised ValueError as expected for Job(7)")
```

output:

```
.[INFO] START: test_scheduler_submit_inserts_into_hash_and_queue - Verify submitting a task inserts into both hash table
 and queue
[INFO] SUCCESS: submit_task inserted Job(42) into hash and queue; queue head matches the inserted job
[INFO] END: test_scheduler_submit_inserts_into_hash_and_queue
```

```
.[INFO] START: test_duplicate_insertion_raises - Ensure duplicate job IDs cannot be inserted into the scheduler
[INFO] SUCCESS: duplicate insertion raised ValueError as expected for Job(7)
[INFO] END: test_duplicate_insertion_raises
```

2. Searching (Job Lookup & Retrieval Operations)

This class validates the search capabilities of the system. It tests the `HashTable` directly to ensure O(1) retrieval of existing jobs and proper handling of missing IDs. It also verifies the Scheduler's `find_job` method, ensuring it can locate a job regardless of its lifecycle state, finding it in the **Queue** if pending, or in the **History Linked List** if already executed.

Test Case

```python
class TestSearching(BaseLoggedTest):
    def test_hash_table_search(self) -> None:
        """HashTable can find existing jobs and returns None for missing ones"""
        ht: HashTable = HashTable(size=3)
        j1: Job = Job(job_id=10)
        j2: Job = Job(job_id=13)  # same bucket as 10 when size=3
        ht.insert(job=j1)
        ht.insert(job=j2)
        self.assertIs(expr1=ht.search(job_id=10), expr2=j1)
        self.assertIs(expr1=ht.search(job_id=13), expr2=j2)
        self.assertIsNone(obj=ht.search(job_id=99))
        log_success(message="HashTable search retrieved Job(10) and Job(13); Job(99) returned None as expected")

    def test_scheduler_find_job_in_queue_and_history(self) -> None:
        """Scheduler.find_job locates job first in queue then in history after execution"""
        s: Scheduler = Scheduler(hash_size=3)
        s.submit_task(job_id=1)
        # Should be found in queue
        loc: tuple[Literal['queue'], Any] | tuple[Literal['history'], Any] | None = s.find_job(job_id=1)
        self.assertIsNotNone(obj=loc)
        self.assertEqual(first=loc[0], second="queue")
        self.assertEqual(first=loc[1].job_id, second=1)
        # Execute and then find in history
        s.run_next_task()
        loc2: tuple[Literal['queue'], Any] | tuple[Literal['history'], Any] | None = s.find_job(job_id=1)
        self.assertIsNotNone(obj=loc2)
        self.assertEqual(first=loc2[0], second="history")
        self.assertEqual(first=loc2[1].job_id, second=1)
        log_success(message="Scheduler.find_job found Job(1) in queue before execution and in history after run_next_task()")
```

Output

```
.[INFO] START: test_hash_table_search - HashTable can find existing jobs and returns None for missing ones
[INFO] SUCCESS: HashTable search retrieved Job(10) and Job(13); Job(99) returned None as expected
[INFO] END: test_hash_table_search
```

```
.[INFO] START: test_scheduler_find_job_in_queue_and_history - Scheduler.find_job locates job first in queue then in history
 after execution
Queue is now empty after dequeue.
Dequeued job: 1
Executing job: 1
[INFO] SUCCESS: Scheduler.find_job found Job(1) in queue before execution and in history after run_next_task()
[INFO] END: test_scheduler_find_job_in_queue_and_history
```

3. Dequeueing (Execution Flow & FIFO Order)

This class focuses on the processing logic. It verifies that the `LinkedQueue` strictly adheres to **First-In-First-Out (FIFO)** order (tasks are executed in the order they arrived). It also confirms that `run_next_task` correctly transitions a job: updating its status to "executed," moving it to the History log, and removing it from the active Queue and Hash Table.

Test Case

```python
class TestDequeueing(BaseLoggedTest):
    def test_linked_queue_fifo(self) -> None:
        """LinkedQueue dequeues elements in FIFO order"""
        q: LinkedQueue = LinkedQueue()
        a, b, c = Job(job_id=1), Job(job_id=2), Job(job_id=3)
        q.enqueue(value=a)
        q.enqueue(value=b)
        q.enqueue(value=c)
        self.assertEqual(first=q.dequeue().job_id, second=1)
        self.assertEqual(first=q.dequeue().job_id, second=2)
        self.assertEqual(first=q.dequeue().job_id, second=3)
        self.assertTrue(expr=q.is_empty())
        log_success(message="LinkedQueue maintained FIFO order for Job IDs [1, 2, 3] and is empty after dequeueing all")

    def test_scheduler_run_next_task_updates_status_and_history(self) -> None:
        """Running next task sets status to executed, logs to history, and removes from hash/queue"""
        s: Scheduler = Scheduler(hash_size=5)
        s.submit_task(job_id=5)
        s.submit_task(job_id=6)
        executed: Any | None = s.run_next_task()
        self.assertIsNotNone(obj=executed)
        self.assertEqual(first=executed.status, second="executed")
        self.assertEqual(first=len(s.history.display_history()), second=1)
        # Ensure it was removed from hash and queue
        self.assertIsNone(obj=s.hash.search(job_id=5))
        self.assertEqual(first=[j.job_id for j in s.queue.to_list()], second=[6])
        log_success(message="run_next_task executed Job(5), added to history, removed from hash, and left Job(6) at queue head")
```

Output

```
.[INFO] START: test_linked_queue_fifo - LinkedQueue dequeues elements in FIFO order
Dequeued job: 1
Dequeued job: 2
Queue is now empty after dequeue.
Dequeued job: 3
[INFO] SUCCESS: LinkedQueue maintained FIFO order for Job IDs [1, 2, 3] and is empty after dequeueing all
[INFO] END: test_linked_queue_fifo
```

```
.[INFO] START: test_scheduler_run_next_task_updates_status_and_history - Running next task sets status to executed, logs
 to history, and removes from hash/queue
Dequeued job: 5
Executing job: 5
[INFO] SUCCESS: run_next_task executed Job(5), added to history, removed from hash, and left Job(6) at queue head
[INFO] END: test_scheduler_run_next_task_updates_status_and_history
```

4. Collision Handling (Hash Table Collision Resolution (Chaining))

This class stress-tests the Hash Table implementation. It intentionally uses a small table size to force collisions (mapping multiple Job IDs to the same index). The tests verify that the system successfully uses **chaining** to store multiple jobs in the same bucket and that `search` and `remove` operations function correctly even when multiple items exist at the same hashed index.

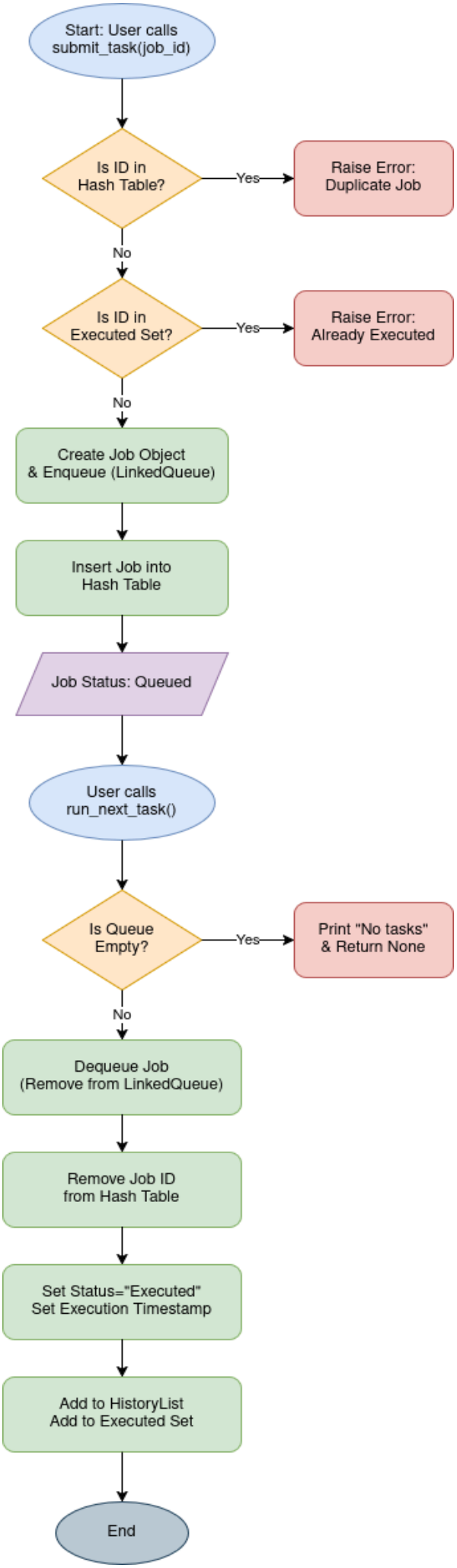
Test Case

```python
class TestCollisionHandling(BaseLoggedTest):
    def test_hash_table_chaining(self) -> None:
        """HashTable handles collisions via chaining; search and removal operate correctly"""
        # Small table to force collisions
        ht: HashTable = HashTable(size=3)
        ids: list[int] = [3, 6, 9]  # All collide into same bucket for size=3
        jobs: list[Job] = [Job(job_id=i) for i in ids]
        for j in jobs:
            ht.insert(job=j)
        # They should all be retrievable
        for i, j in zip(ids, jobs):
            self.assertIs(expr1=ht.search(job_id=i), expr2=j)
        # And removal should work FIFO in bucket order where matched
        removed: Any | None = ht.remove(job_id=6)
        self.assertIsNotNone(obj=removed)
        self.assertEqual(first=removed.job_id, second=6)
        self.assertIsNone(obj=ht.search(job_id=6))
        # Remaining still present
        self.assertIsNotNone(obj=ht.search(job_id=3))
        self.assertIsNotNone(obj=ht.search(job_id=9))
        log_success(message="HashTable chaining stored Jobs [3,6,9]; removal of Job(6) succeeded and others remained retrievable")
```

Output

```
[INFO] START: test_hash_table_chaining - HashTable handles collisions via chaining; search and removal operate correctly
[INFO] SUCCESS: HashTable chaining stored Jobs [3,6,9]; removal of Job(6) succeeded and others remained retrievable
[INFO] END: test_hash_table_chaining
```

Flow Chart


Start: User calls submit_task(job_id)
Is ID in Hash Table?
Yes
Raise Error: Duplicate Job
No
Is ID in Executed Set?
Yes
Raise Error: Already Executed
No
Create Job Object & Enqueue (LinkedQueue)
Insert Job into Hash Table
Job Status: Queued
User calls run_next_task()
Is Queue Empty?
Yes
Print "No tasks" & Return None
No
Dequeue Job (Remove from LinkedQueue)
Remove Job ID from Hash Table
Set Status="Executed" Set Execution Timestamp
Add to HistoryList Add to Executed Set
End