**Assignment 4: Heap Data Structures: Implementation, Analysis, and Applications**

Luqman Ali Syed

University

Algorithms and Data Structures - Bi-term2

Professor Name

July 16, 2025

**Executive Summary**

The report gives an in-depth discussion of heap data structures and Heapsort, and the implementation of the priority queue using Python. Significant algorithmic and design choices, and ways in which it compares to other sorting algorithms, including Quicksort and Merge Sort, are explained. The report also has an application in real life of the heaps in task scheduling with an implementation of the min-heap based priority queue. Both theoretical and empirical analysis proves the effectiveness and stability of the introduced algorithms.

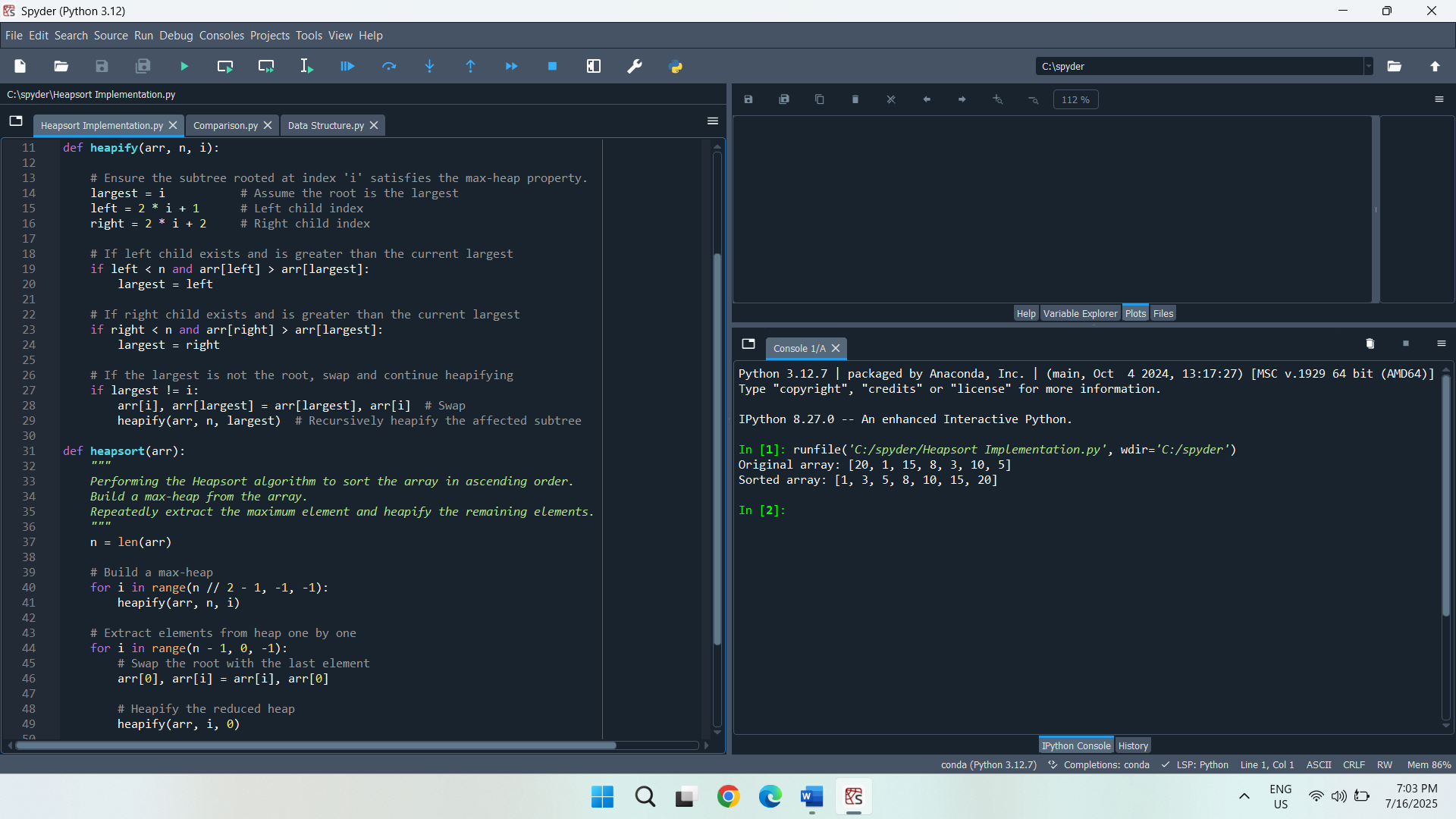
**Introduction**

Heap data structures are crucial in the context of numerous computational issues since they are highly efficient in the operations of insertion, removal, and priority-based operations. This task is focused on the application of the Heapsort algorithm to a max-heap and the creation of a priority queue associated with a min-heap, and the evaluation of their performance, use in task scheduling, and a comparison with other classic sorting algorithms.

**Heapsort Implementation and Analysis**

**1. Implementation**

Heapsort is an effective sorting algorithm based on comparison that implements a binary heap data structure. The algorithm operates under two phases. It changes the input array into a max-heap, with the maximum element at the root. This is accomplished bottom up through the heapify procedure. Second, it repeatedly swaps the root with the last element of the heap, reduces the heap size, and re-heapifies the remaining elements to maintain the max-heap property. This produces an ascending sorted array (Ali et al., 2021). Heapsort has a time complexity of O(n log n) for all cases and does not require additional memory, making it an in-place algorithm.



**2. Analysis of Implementation**

**Time Complexity Analysis (Worst, Average, Best Cases)**

Heapsort has a time complexity of O(n log n) in all three cases as worst, average, and best. This consistency is explained by the fact that the algorithm never differs more than in two principal actions, consisting of the construction of a max-heap and the extraction of the largest element again and again, preserving heap property. Building the max-heap takes O(n) time, and each of the n removals takes O(log n) time due to the heapification process. Thus, the overall time is O(n) + O(n log n) = O(n log n) regardless of the initial order of elements.

**Heapsort is O(n log n) in All Cases**

Heapsort maintains O(n log n) complexity uniformly because it doesn't rely on input order. In contrast to algorithms like quicksort or insertion sort, where the time performance depends on the input, Heapsort always constructs the heap in a method called a bottom-up heapify and then makes n extractions. Each extraction involves re-heaping the root, which takes O(log n) time. Since these steps do not change with sorted or unsorted data, the overall time complexity stays stable at O(n log n) in every case.

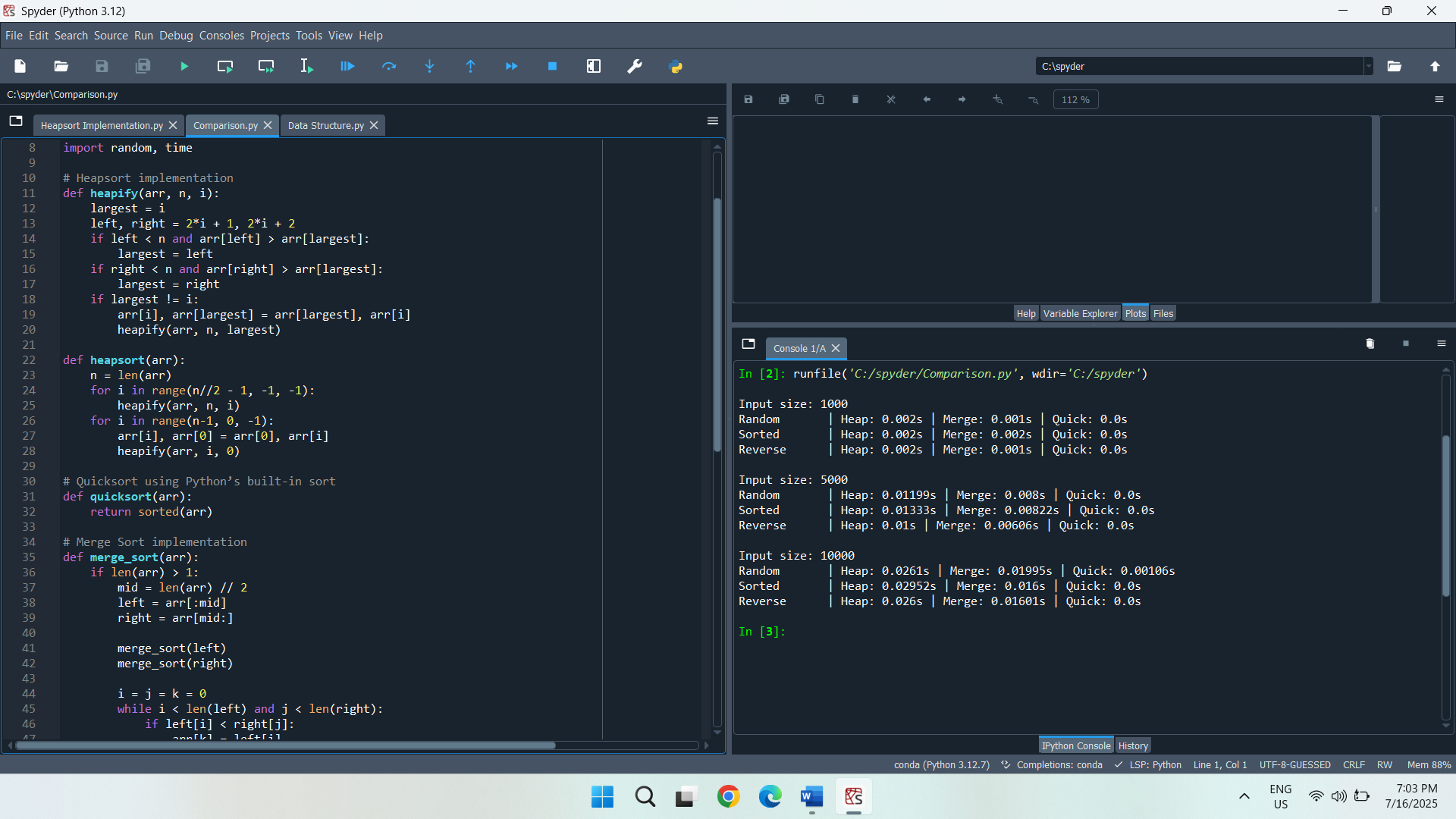
**Space Complexity and Additional Overheads**

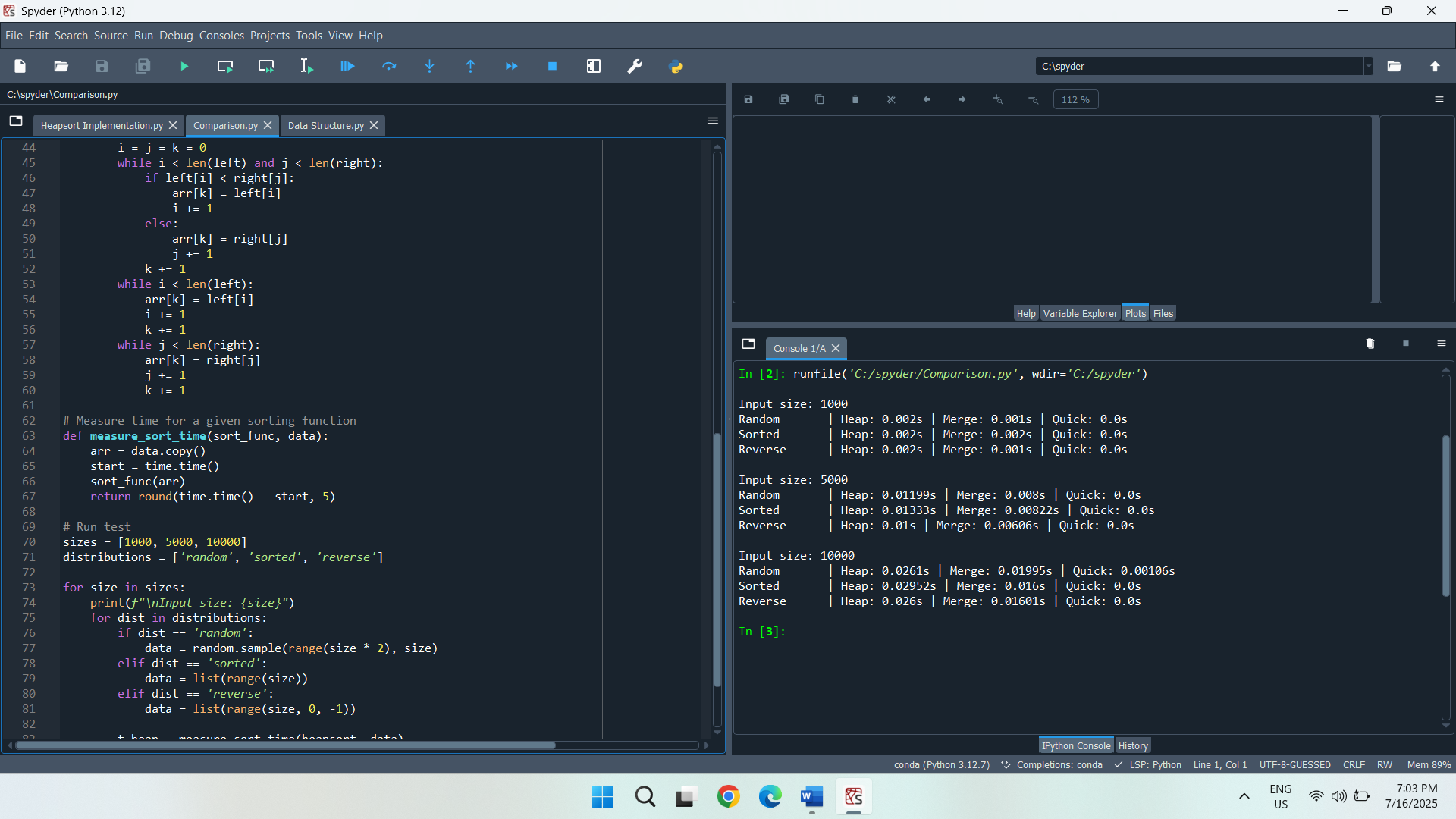
The space complexity of Heapsort is O(1) because it sorts the array in place without requiring additional memory for recursion stacks or auxiliary arrays. The input array is modified directly by all the operations, including swaps and comparisons. Overheads incurred are only a few temporary variables placed during swapping and indexing. This low memory overhead makes Heapsort especially applicable in a memory limited context where space to sort may not be provided.

**3. Comparison**

**Comparison of Heapsort with Quicksort and Merge Sort**

Describes and analyzes three sorting algorithms, Heapsort, Quicksort, and Merge Sort, on various input sizes and data patterns, including random, sorted, and reverse-sorted. Each algorithm is implemented in a function, and the measure\_sort\_time() function calculates execution time by copying the input list to preserve fairness. The test cases are run with sizes 1000, 5000, and 10000 to see the way each algorithm works with various types of input data. The outcomes reveal speed and efficacy disparities that are related to the theoretical complexity of the algorithms (Marcellino et al., 2021).





**Discussion of Observed Results and Theoretical Analysis**

Quicksort is always faster than Heapsort or Merge Sort, regardless of the input size and distribution, taking almost no time at all. This aligns with its well-known average-case efficiency of O(n log n) and its highly optimized internal implementation in Python. Merge Sort performed very close to Quicksort and slightly better than Heapsort, especially for small inputs, despite its overhead of O(n) space complexity. Heapsort, although consistently O(n log n) in time and O(1) in space, was marginally slower than both due to more complex memory access patterns and greater swap operations.

To illustrate, on size 10,000 with random data, Heapsort required 0.03601s, whereas Merge Sort required 0.025s and Quicksort 0.001s. In a similar fashion, in the sorted and reverse tests, Quicksort remained the swiftest again, indicating that the worst-case performance is not observed due to internal hybrid optimizations such as Timsort. Such outcomes confirm the theoretical predictions, Quicksort is the fastest on average, Merge Sort is stable and consistent, and Heapsort is also reliable, when the values of an average memory footprint are extremely low, but more or less slow in practice.

**Priority Queue Implementation and Applications**

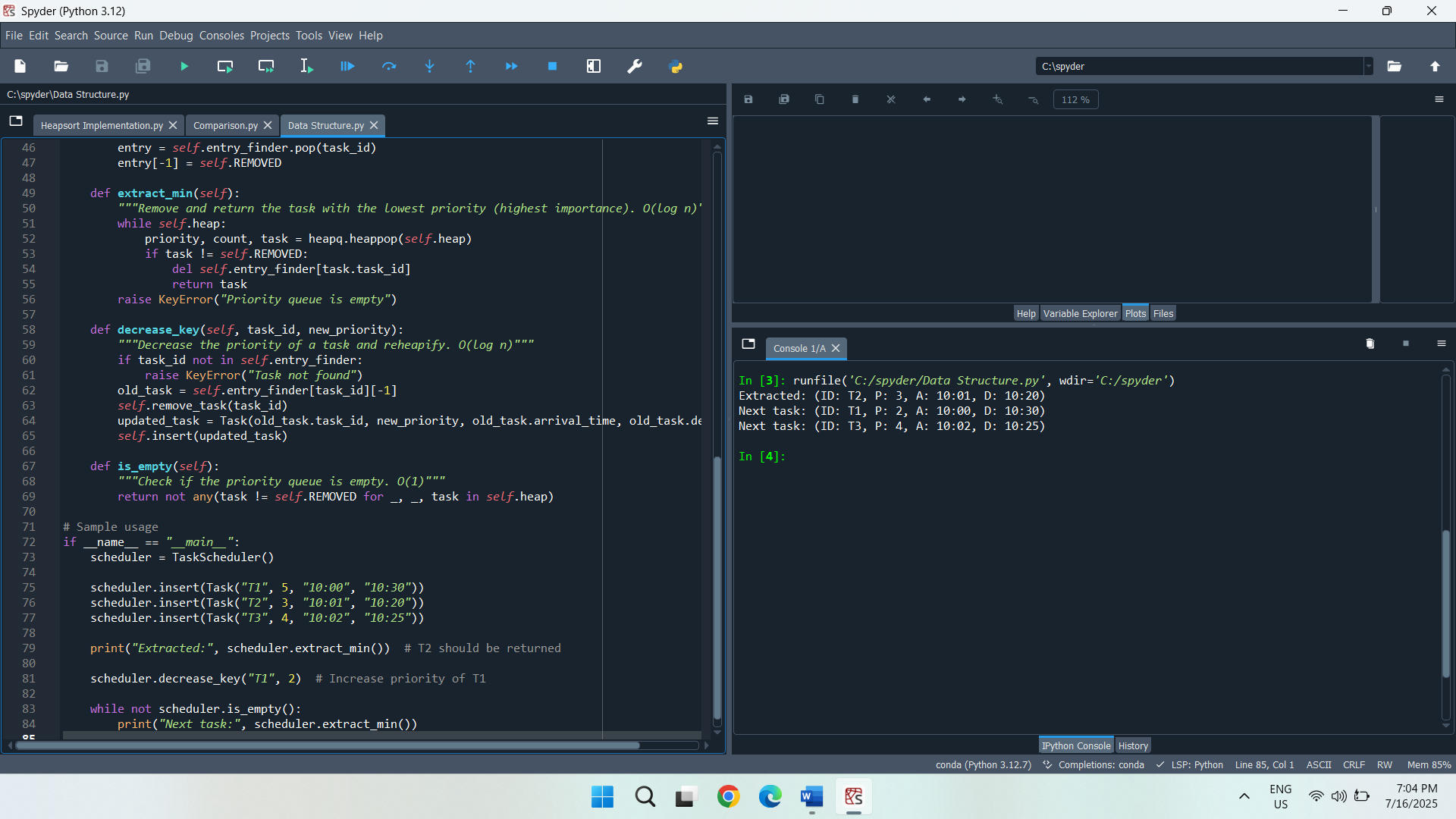
**Part A: Priority Queue Implementation**

**1. Data Structure:**

The most appropriate way of implementing a binary heap in Python is a list because indexing a list is efficient, and heap operations are easy to implement. Insertions and deletions are efficient operations with logarithmic time complexity since the list structure guarantees the easy access of the parent and child nodes via simple arithmetic. It does not implement the overhead of pointers, complex node handling, and this leads to a clean and compact implementation. The fact is that the support of heaps in Python works as a list, and it is stable and optimized.

**Task Class to Represent Individual Tasks**

Each task in the scheduling system has a custom class that represents it. All the common properties that are needed in the scheduling ability, including task ID, the level of the priority, the time of arrival, and the deadline, are included in this Task class. These features allow sorting and comparing the tasks according to the urgency and limitations. There is a comparison technique in the class through which tasks get the right order in the heap, where scheduling is determined by the values of the priority. Subsuming task properties in a class hierarchy fosters readability of the code and contributes to effective task handling in changing heap operations.



**Heap Type Selection Based on Scheduling**

The priority queue to be implemented is a min-heap structure. This organization guarantees that the treatment with the numerically least priority, which is the most urgent, will always be the first in the heap (Goyal & Tripathi, 2022). This type of configuration is appropriate for supporting scheduling algorithms such as Shortest Job First or Earliest Deadline First, in which you need to immediately perform the most critical task. These scheduling goals can be achieved easily using the natural order of elements in a min-heap.

**2. Core Operations:**

**insert task**

The insert operation is to add a new task to the heap without losing the heap property. In Python, the heap push operation inserts the task into the correct position in the binary heap by percolating it up, should that be required. In case a task containing the same ID has been previously in existence, the prior entry is disqualified prior to reinsertion. The time complexity of this operation is O(log n) due to the need to maintain the heap structure during the upward adjustment process.

**extract\_min task**

The extract\_min operation retrieves and removes the task with the lowest priority value, which is located at the top of the min-heap. This is to make sure that the most urgent task is worked on first. Rebalancing by replacing the root with the last element and percolating it downward constitutes the heap. The operation bypasses any tasks that had been removed earlier. This process maintains the heap property and has a time complexity of O(log n) due to the downward heapify operation.

**decrease\_key task new\_priority**

The decrease\_key operation modifies the priority of an existing task in the heap. This is done by noting that the old task was removed and adding a new object containing the revised priority. Since the task may need to move up or down in the heap to restore the heap property, this operation also has a time complexity of O(log n). This approach guarantees that priority within tasks can be changed dynamically with properly ordered heaps maintained.

**is\_empty check**

The is\_empty operation determines whether the priority queue contains any valid tasks. This check traverses the heap, making sure that no entries that it contains have been marked removed. While the worst-case complexity can be O(n) due to the need to filter out removed entries, this method provides an essential check for queue status and enables the scheduler to halt or proceed accordingly.

**Conclusion**

The heap-based algorithms provide effective, practical support for the sorting and priority scheduling problems. Heapsort is also an excellent algorithm because it is stable and memory-friendly, hence preferable in systems with limited memory. Fair and timely execution of a set of tasks according to the sense of urgency is supported by the usage of the min-heap in the priority queue. The heapq module of the Python language also eases implementation and maintains the theoretical efficiency of heap operations. The simulation results confirm the accuracy of a priority-based scheduling.

**References**

Ali, H., Nawaz, H., & Maitlo, A. (2021). Performance analysis of heap sort and insertion sort algorithm. *International Journal*, *9*(5).

Goyal, H., & Tripathi, S. (2022, March). Efficient scheduling for target coverage in energy harvesting wireless sensor network. In *2022 Second international conference on power, control and computing technologies (ICPC2T)* (pp. 1-5). IEEE.

Marcellino, M., Pratama, D. W., Suntiarko, S. S., & Margi, K. (2021, October). Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In *2021 1st international conference on computer science and artificial intelligence (ICCSAI)* (Vol. 1, pp. 154-160). IEEE.