**DEPARTMENT OF SOFTWARE ENGINEERING**

**NAME # WAQAS AHMAD**

**ROLL\_NO # 12409**

**SEMESTER (SECTION) # 3rd Semester (c)**

**SUBJECT # DSA (Assignment:02)**

**SUBMITTED TO # Mr.Abdul Ahad**

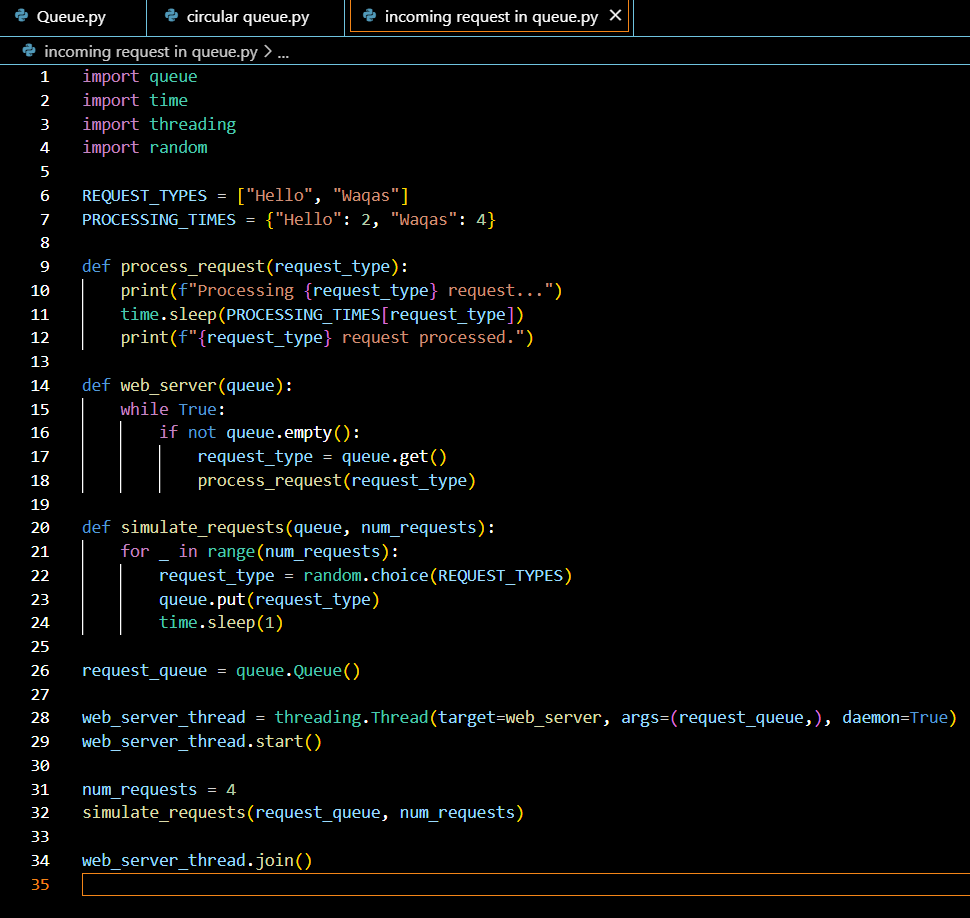
**DATE # 13/11/2023**

**AUST**

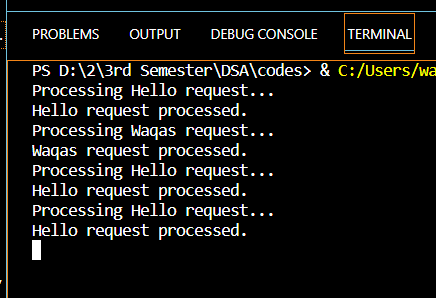
**ABBOTTABAD UNIVERSITY OF SCIENCE & TECHNOLOGY**



**Q1: Design a Python program that simulates a web server handling incoming requests using a queue. Model different types of requests with varying processing times and simulate their processing order.**



**Output:**



**Q2: In what scenarios would you choose a linked list implementation over an array implementation for a queue, and vice versa?**

**Linked List Implementation:**

* Use when you don't know how many elements you'll have in advance or if the number of elements can change.
* Good for frequently adding or removing elements from both ends of the queue.
* Uses memory more flexibly.

**Array Implementation:**

* Use when you know the maximum number of elements beforehand or if it's fixed.
* Suitable if you need to access elements randomly by their position.
* Can be simpler and more memory-efficient for small queues.

In essence, choose a linked list for flexibility and dynamic sizing, and choose an array for simplicity, known size, and efficient random access.

**Q3: Discuss the time complexity of enqueue and dequeue operations in a basic queue. How can you optimize these operations for specific use cases?**

**Time Complexity of Basic Queue Operations**:

**Enqueue Operation:**

**Time Complexity: O(1)**

Enqueue adds an element to the end of the queue. In a basic queue implementation, this operation involves inserting an element at the rear of the queue, and it can be done in constant time since we just need to update the rear pointer and assign the new element to the rear.

**Dequeue Operation:**

**Time Complexity: O(1)**

Explanation: Dequeue removes an element from the front of the queue. In a basic queue, this operation is also a constant time operation since we only need to update the front pointer to the next element in the queue.

**Optimizations for specific use cases:**

**1. Frequent Enqueue/Dequeue:**

Optimization: Use a double-ended queue (deque) for quick additions and removals from both ends.

**2. Limited Memory:**

Optimization: Choose a circular queue to efficiently use memory and prevent waste.

**3. Priority Handling:**

Optimization: Opt for a priority queue to quickly access the highest-priority element.

**4. Concurrency:**

Optimization: For multiple operations happening at once, use a concurrent queue or implement locks to avoid conflicts.

**5. Real-time Requirements:**

Optimization: Select a data structure that ensures quick access and modification for real-time applications.

**6. Batch Processing:**

Optimization: Design queue operations for efficient batch processing to minimize individual operations.

**7. Minimizing Wasted Resources:**

Optimization: Choose a dynamic data structure that adjusts its size, like a dynamic array-based queue.

**8. Time Complexity Overhead:**

Optimization: Prioritize data structures with consistent constant-time complexity for enqueue and dequeue operations, such as a well-implemented circular buffer.

Q4: How can you use two stacks to implement a queue? Provide a step-by-step explanation of the enqueue and dequeue operations in this scenario.

Implementing a queue using two **stacks** involves utilizing the **LIFO (Last In, First Out)** nature of **stacks** to simulate the **FIFO (First In, First Out)** order of a queue.

**Enqueue Operation:**

1. **Push the element to be enqueued onto stack1.** This maintains the order of elements to be dequeued.
2. **If stack2 is not empty, transfer all elements from stack2 to stack1.** This ensures that the oldest element remains at the bottom of **stack1**.
3. **Set stack2 as the empty stack.** This prepares **stack2** to receive the elements from **stack1**.
4. **Transfer all elements from stack1 to stack2.** This reverses the order of elements, making the oldest element the top element of **stack2**.
5. **Push the element enqueued in step 1 onto stack2.** This places the new element at the front of the queue.

**Dequeue Operation:**

1. **If stack2 is empty, transfer all elements from stack1 to stack2.** This ensures that the oldest element is at the top of **stack2**.
2. **Pop the top element from stack2.** This removes the oldest element from the queue.
3. **Return the popped element.** This completes the dequeue operation.