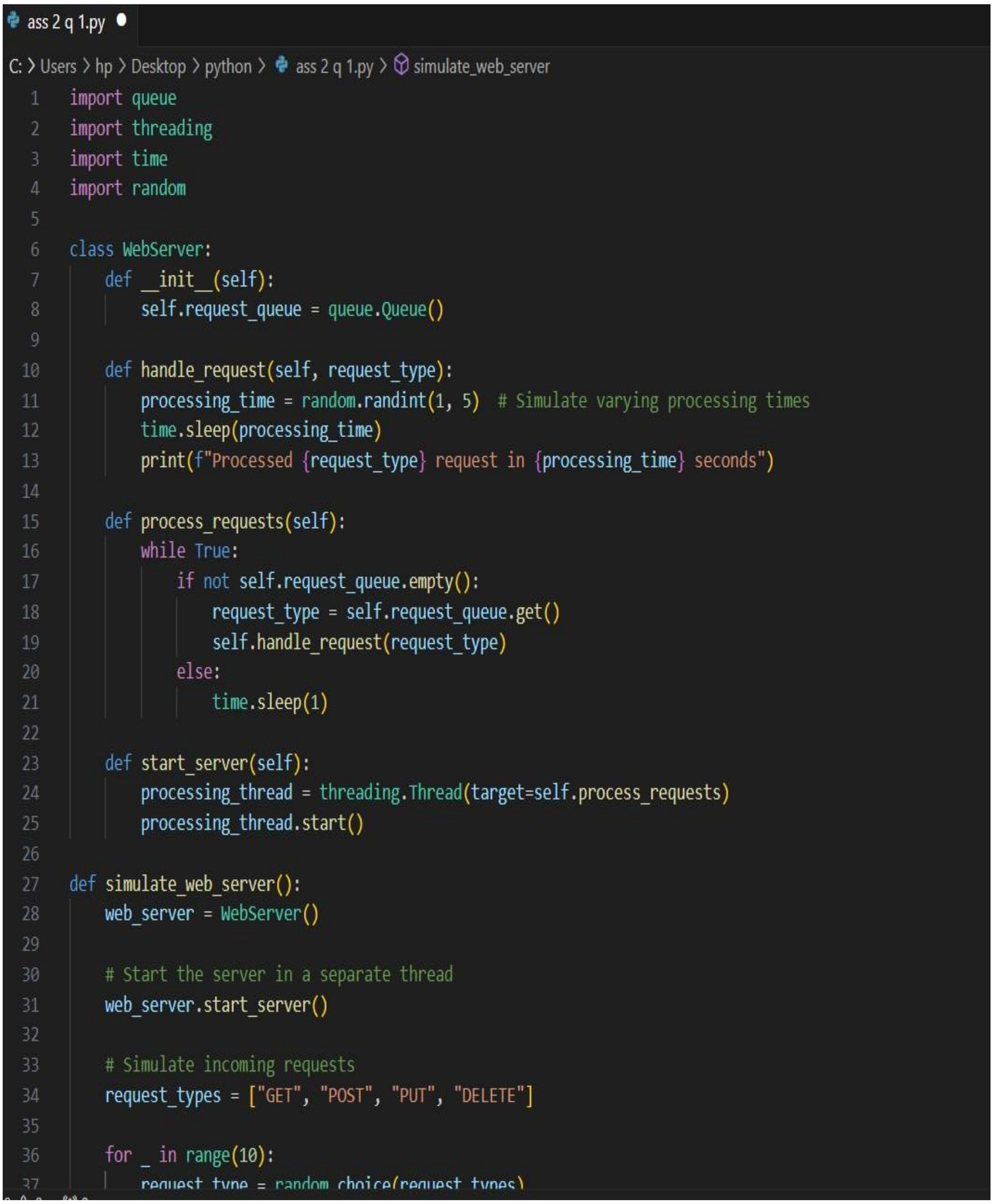
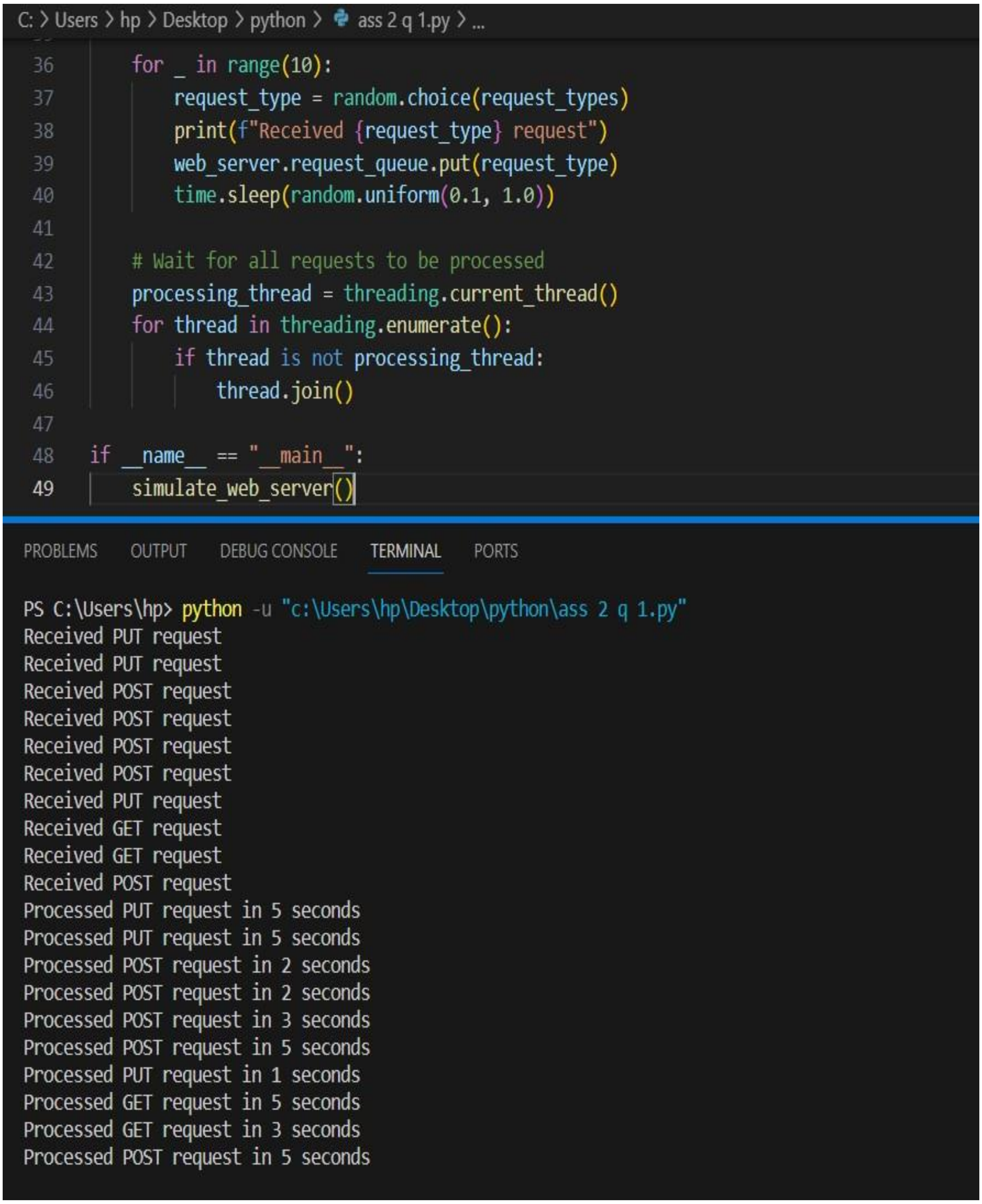
ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY

NAME	**SAAD SHAH**
ROLL NO	**12373**
SUBJECT	**DSA**
ASSIGNMENT NO	**02**
SUBMITTED TO	**MR JAMAL ABDUL AHAD**
DATED	**11-11-2023**

Q NO 1 : 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.


ass 2 q 1.py
C: > Users > hp > Desktop > python > ass 2 q 1.py > simulate_web_server

```python
import queue
import threading
import time
import random

class WebServer:
    def __init__(self):
        self.request_queue = queue.Queue()

    def handle_request(self, request_type):
        processing_time = random.randint(1, 5)  # Simulate varying processing times
        time.sleep(processing_time)
        print(f"Processed {request_type} request in {processing_time} seconds")

    def process_requests(self):
        while True:
            if not self.request_queue.empty():
                request_type = self.request_queue.get()
                self.handle_request(request_type)
            else:
                time.sleep(1)

    def start_server(self):
        processing_thread = threading.Thread(target=self.process_requests)
        processing_thread.start()

def simulate_web_server():
    web_server = WebServer()

    # Start the server in a separate thread
    web_server.start_server()

    # Simulate incoming requests
    request_types = ["GET", "POST", "PUT", "DELETE"]

    for _ in range(10):
        request_type = random.choice(request_types)
```


C: > Users > hp > Desktop > python > ass 2 q 1.py > ...

```python
    for _ in range(10):
        request_type = random.choice(request_types)
        print(f"Received {request_type} request")
        web_server.request_queue.put(request_type)
        time.sleep(random.uniform(0.1, 1.0))

    # Wait for all requests to be processed
    processing_thread = threading.current_thread()
    for thread in threading.enumerate():
        if thread is not processing_thread:
            thread.join()

if __name__ == "__main__":
    simulate_web_server()
```

```
PS C:\Users\hp> python -u "c:\Users\hp\Desktop\python\ass 2 q 1.py"
Received PUT request
Received PUT request
Received POST request
Received POST request
Received POST request
Received POST request
Received PUT request
Received GET request
Received GET request
Received POST request
Processed PUT request in 5 seconds
Processed PUT request in 5 seconds
Processed POST request in 2 seconds
Processed POST request in 2 seconds
Processed POST request in 3 seconds
Processed POST request in 5 seconds
Processed PUT request in 1 seconds
Processed GET request in 5 seconds
Processed GET request in 3 seconds
Processed POST request in 5 seconds
```

Q NO 2: In what scenarios would you choose a linked list implementation over an array implementation for a queue, and vice versa?

ANSWER:- The choice between a linked list implementation and an array implementation for a queue depends on the specific requirements and characteristics of the application. Here are some considerations for choosing between the two:

Linked List Implementation for a Queue:

1. Dynamic Size: If the size of the queue is not fixed and may change dynamically during the program's execution, a linked list implementation is often preferred. Linked lists allow for efficient dynamic memory allocation and deal location, accommodating a variable number of elements without the need for resizing.

2. Constant-Time Enqueue and Dequeue: Linked lists provide constant-time insertion and deletion at both the front and rear, making them suitable for scenarios where enqueue and dequeue operations are equally frequent and need to be fast.

3. Frequent Insertions and Deletions: If the queue involves frequent insertions and deletions, especially at the middle or beginning, a linked list's constant-time insertion and deletion are advantageous compared to arrays, where such operations may involve shifting elements.

Array Implementation for a Queue:

1. Random Access Requirements: If there is a need for random access to elements or if the queue size is fixed, an array implementation might be more suitable. Arrays provide constant-time random access to elements, allowing direct access to any element using its index.

2. Memory Efficiency: In scenarios where memory efficiency is a concern, an array may be preferred. Arrays generally have lower overhead per element compared to linked lists, which require additional memory for storing pointers.

3. Cache Locality: If the application benefits from good cache locality, an array may perform better. Arrays provide better spatial locality, which can result in improved cache performance compared to the scattered memory locations of a linked list.

Hybrid Approaches:

In some cases, hybrid approaches are used, such as using a dynamic array (resizable array) to combine the advantages of arrays and linked lists. Dynamic arrays can provide amortized constant-time insertions and deletions at the end while allowing for random access.

In summary, the choice between a linked list and an array for a queue depends on the specific requirements of the application, including factors such as dynamic sizing, frequent insertions/deletions, random access needs, and memory efficiency considerations.

Q NO 3: Discuss the time complexity of enqueue and dequeue operations in a basic queue. How can you optimize these operations for specific use cases?

ANSWER:-

In a basic queue implemented using an array or a linked list, the time complexity of enqueue and dequeue operations depends on the underlying data structure and the specific implementation.

Basic Queue:

1. Enqueue Operation:

- Array Implementation: In an array implementation, adding an element to the end of the array (enqueue) takes constant time on average, assuming that the array has available space. However, if the array needs to be resized to accommodate the new element, the operation may take O(n) time in the worst case, where n is the number of elements in the array.

- Linked List Implementation: In a linked list, adding an element to the end (enqueue) always takes constant time, as it involves updating the next pointer of the last node to point to the new node.

2. Dequeue Operation:

- Array Implementation: Removing an element from the front of an array (dequeue) takes O(n) time in the worst case because it requires shifting all remaining elements to fill the gap left by the removed element. On average, it is a constant-time operation.

- Linked List Implementation: Removing an element from the front (dequeue) takes constant time in a linked list, as it involves updating the head pointer to the next node.

Optimization Strategies:

1. Circular Buffer for Array Implementation:

To avoid the need for frequent resizing of the array, a circular buffer can be used. This involves maintaining two pointers, one for the front and one for the rear, and using modulo arithmetic to wrap around the array. This way, when the rear pointer reaches the end of the array, it can wrap around to the beginning.

2. Dynamic Array (Resizable Array):

Use a dynamic array that can resize itself when needed. This involves doubling the array size and copying elements when it becomes full. While this resizing operation has a time complexity of O(n), it occurs infrequently, and on average, the enqueue operation remains amortized constant time.

3. Double-Ended Queue (Deque):

If both enqueue and dequeue operations need to be efficient at both ends, a double-ended queue (deque) might be a better choice. Deques allow constant-time insertions and deletions at both the front and rear, providing flexibility for various use cases.

4. Optimizing for Specific Scenarios:

Consider the specific use case requirements. For example, if enqueue operations are more frequent than dequeues, optimization efforts can focus on minimizing the cost of enqueue operations, and vice versa.

In summary, the time complexity of enqueue and dequeue operations in a basic queue depends on the implementation details, and optimization strategies can be applied based on specific use cases to improve the efficiency of these operations.

Q NO 4: 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.

ANSWER: You can implement a queue using two stacks. The basic idea is to use one stack for the enqueue operation (adding elements to the back of the queue) and the other stack for the dequeue operation (removing elements from the front of the queue). The process involves transferring elements between the two stacks when necessary. Below is a step-by-step explanation of how to perform enqueue and dequeue operations in this scenario.

Enqueue Operation:

1. Push onto Stack 1 (s1):

When you want to enqueue an element, push it onto Stack 1 (s1).

Enqueue(1):

s1: [1]

2. Push onto Stack 1 (s1):

Continue pushing elements onto Stack 1 for each enqueue operation.

Enqueue(2):

s1: [1, 2]

3. Dequeue Operation:

When a dequeue operation is needed, check if Stack 2 (s2) is empty. If it's empty, transfer all elements from Stack 1 to Stack 2.

Enqueue(3):

s1: [1, 2, 3]

Dequeue:

Transfer elements from s1 to s2.

s1: []

s2: [3, 2, 1]

4. Dequeue from Stack 2 (s2):

Pop the top element from Stack 2. This is the front of the queue.

Dequeue:

s1: []

s2: [3, 2]

Result: 1 (front of the queue)

Repeat Operations:

- Continue repeating the enqueue and dequeue operations as needed.

Summary:

Enqueue operation: Push elements onto Stack 1 (s1).

Dequeue operation:

If Stack 2 (s2) is not empty, pop from Stack 2.

- If Stack 2 is empty, transfer all elements from Stack 1 to Stack 2, then pop from Stack 2.

THE END