ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY

Department of Computer Science
Assignment – 2

Submitted By : Sardar Muhammad Saad

Class : BSSE 3rd

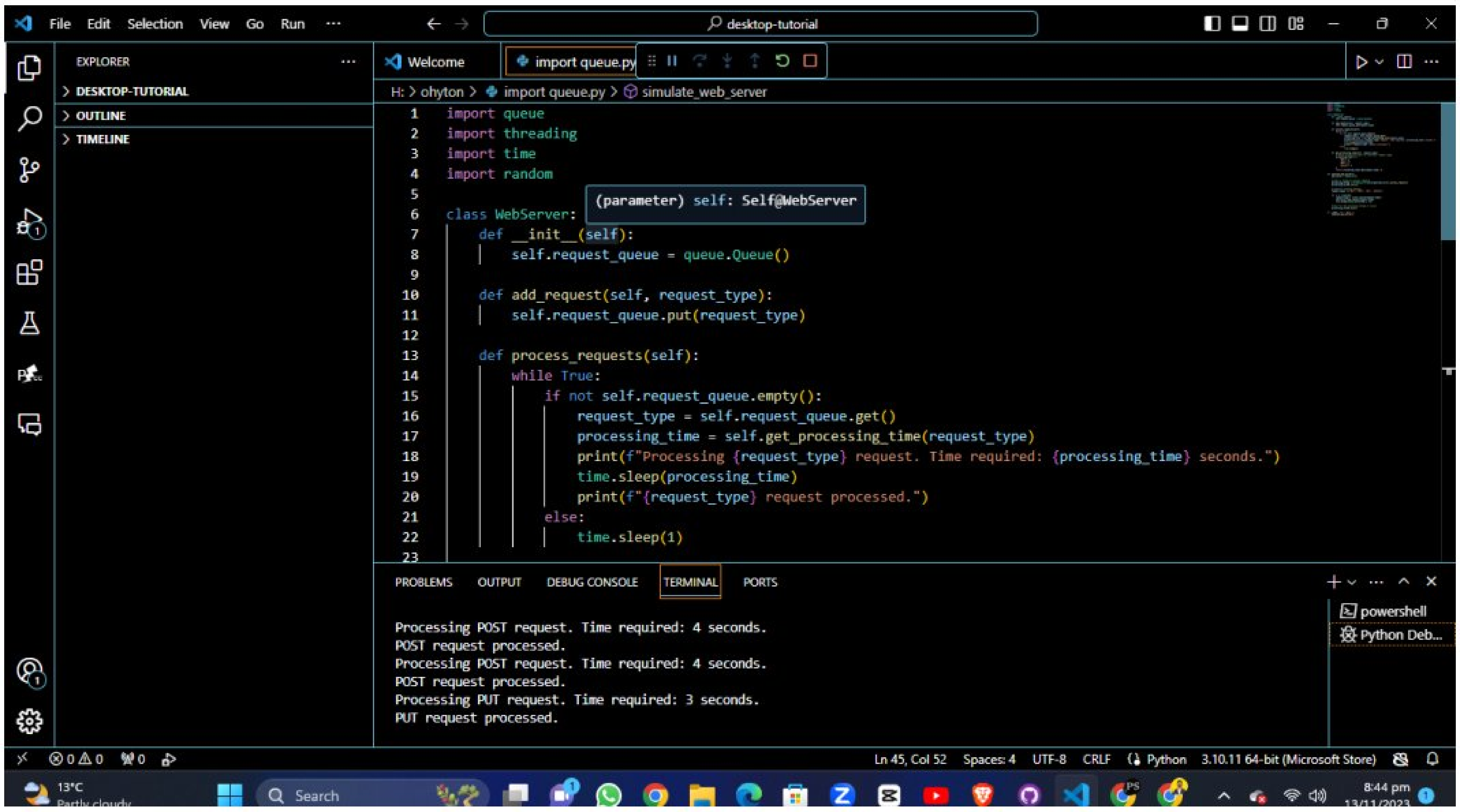
Section : C

Roll no :12386

Assignment : 02(DSA)

Submitted to : Sir Jammal Abdul Ahad

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.



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

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

    def add_request(self, request_type):
        self.request_queue.put(request_type)

    def process_requests(self):
        while True:
            if not self.request_queue.empty():
                request_type = self.request_queue.get()
                processing_time = self.get_processing_time(request_type)
                print(f"Processing {request_type} request. Time required: {processing_time} seconds.")
                time.sleep(processing_time)
                print(f"{request_type} request processed.")
            else:
                time.sleep(1)
```

```
Processing POST request. Time required: 4 seconds.
POST request processed.
Processing POST request. Time required: 4 seconds.
POST request processed.
Processing PUT request. Time required: 3 seconds.
PUT request processed.
```

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

The choice between a linked list and an array implementation for a queue depends on the specific requirements and characteristics of the problem you are trying to solve. Here are some considerations for each:

Linked List Implementation:

1. **Dynamic Size:**

- Linked lists allow for dynamic memory allocation, making them suitable when the size of the queue is not known in advance or may change frequently.

2. **Insertions and Deletions:**

- Insertions and deletions are efficient in a linked list implementation. Adding elements at the end or removing elements from the front can be done in constant time.

3. **No Wasted Space:**

- Linked lists don't require contiguous memory, so there's no wasted space due to pre-allocation. This can be beneficial when memory efficiency is a concern.

4. **Frequent Enqueue and Dequeue Operations:**

- If your application involves frequent enqueue and dequeue operations, a linked list may be a good choice due to its constant-time insertion and deletion at both ends.

Array Implementation:

1. **Random Access Requirements:**

- If your application requires constant-time random access to elements (i.e., accessing elements by index), an array implementation may be more suitable. Arrays provide constant-time access to any element based on its index.

2. **Memory Efficiency:**

- Arrays can be more memory-efficient in certain scenarios where the overhead of maintaining pointers in a linked list is significant. If memory is a concern, and the size of the queue is fixed or can be estimated, an array might be a better choice.

3. **Cache Locality:**

- Arrays exhibit better cache locality compared to linked lists. If your application's performance depends on minimizing cache misses, an array may provide better performance.

4. **Predictable Time Complexity:**

- Arrays generally provide more predictable time complexity for access and storage. If your use case involves strict requirements on time complexity, an array might be preferable.

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

In a basic queue data structure, the enqueue and dequeue operations have different time complexities based on the underlying implementation.

Basic Queue Operations:

1. **Enqueue Operation:**

- Time Complexity: O(1)

- In a basic queue, the enqueue operation involves adding an element to the rear or end of the queue. This operation has a constant time complexity, O(1), because it can be done in a single step.

2. **Dequeue Operation:**

- Time Complexity: O(1)

- The dequeue operation removes an element from the front of the queue. In a basic queue, this operation also has a constant time complexity, O(1), because removing an element from the front can be done in constant time.

Optimization Strategies:

While the basic queue operations are already efficient, there are scenarios where you might want to optimize them further, especially in specific use cases.

1. **Optimizing for Memory Usage:**

- If memory usage is a critical concern, you might consider implementing a circular queue or a dynamic array-based queue. Circular queues can help reduce memory fragmentation and allow for more efficient use of space. Dynamic arrays can resize themselves dynamically to accommodate varying queue sizes, avoiding unnecessary memory allocation.

2. **Optimizing for Concurrent Access:**

- In scenarios where the queue is accessed concurrently by multiple threads, you might want to use thread-safe data structures or implement locking mechanisms to ensure that enqueue and dequeue operations are performed atomically. This can prevent race conditions and maintain data consistency.

3. **Batch Processing:**

- If your application allows for it, you can optimize enqueue and dequeue operations by processing elements in batches. Instead of performing individual operations for each element, process several elements at once. This can be especially beneficial when dealing with I/O operations or network communication.

4. **Priority Queues:**

- If your application requires elements to be processed in a specific order, consider using a priority queue. Priority queues allow you to assign a priority to each element and process them based on their priority. Priority queue implementations might have different time complexities for enqueue and dequeue operations, depending on the underlying data structure (e.g., binary heap).

5. **Double-ended Queues (Dequeues):**

- If your application requires efficient insertion and removal at both ends of the queue, consider using a double-ended queue (deque). Deques allow for O(1) operations at both ends, providing flexibility for various use cases.

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.

Using two stacks to implement a queue is a common approach. The idea is to simulate the behavior of a queue by using two stacks—one for enqueue operations and another for dequeue operations. Here's a step-by-step explanation of how you can implement the enqueue and dequeue operations:

Implementation Steps:

1. Initialize Two Stacks:

- Create two stacks, let's call them `stack_enqueue` and `stack_dequeue`.

```python
stack_enqueue = []  # Stack for enqueue operations

stack_dequeue = []  # Stack for dequeue operations
```

2. Enqueue Operation:

- When you want to enqueue an element (add it to the back of the queue):

  - Push the element onto `stack_enqueue`.

```python
def enqueue(element):
    stack_enqueue.append(element)
```

3. Dequeue Operation:

- When you want to dequeue an element (remove it from the front of the queue):

  - Check if `stack_dequeue` is empty. If it is, transfer elements from `stack_enqueue` to `stack_dequeue` to reverse their order.

  - Pop the top element from `stack_dequeue`, which now represents the front of the queue.

```python
def dequeue():
    if not stack_dequeue:
        # Transfer elements from stack_enqueue to stack_dequeue to reverse their order
        while stack_enqueue:
            stack_dequeue.append(stack_enqueue.pop())

    # Pop the top element from stack_dequeue
    if stack_dequeue:
        return stack_dequeue.pop()
```

    else:

        # Queue is empty

        return None

Example:

```python
# Initialize two stacks
stack_enqueue = []
stack_dequeue = []

# Enqueue some elements
enqueue(1)
enqueue(2)
enqueue(3)

# Dequeue elements
print(dequeue())  # Output: 1
print(dequeue())  # Output: 2

# Enqueue more elements
enqueue(4)
enqueue(5)


# Dequeue remaining elements

print(dequeue())  # Output: 3

print(dequeue())  # Output: 4

print(dequeue())  # Output: 5

print(dequeue())  # Output: None (queue is empty)
```

Explanation:

- Enqueue operations are directly performed on `stack_enqueue`.

- Dequeue operations are performed on `stack_dequeue`, and if it's empty, elements are transferred from `stack_enqueue` to `stack_dequeue` to reverse their order.

- This way, the front of the queue is always at the top of `stack_dequeue`.

This implementation ensures that enqueue and dequeue operations have an amortized time complexity of O(1). While individual operations may require O(n) time in the worst case (when transferring elements between stacks), this happens infrequently, and on average, the complexity remains O(1).