ABBOTTABAD UNIVERSTY OF SCIENCE AND TECHNOLOGY

NAME: Safeullah naseebullah

ROLL NO: 12375

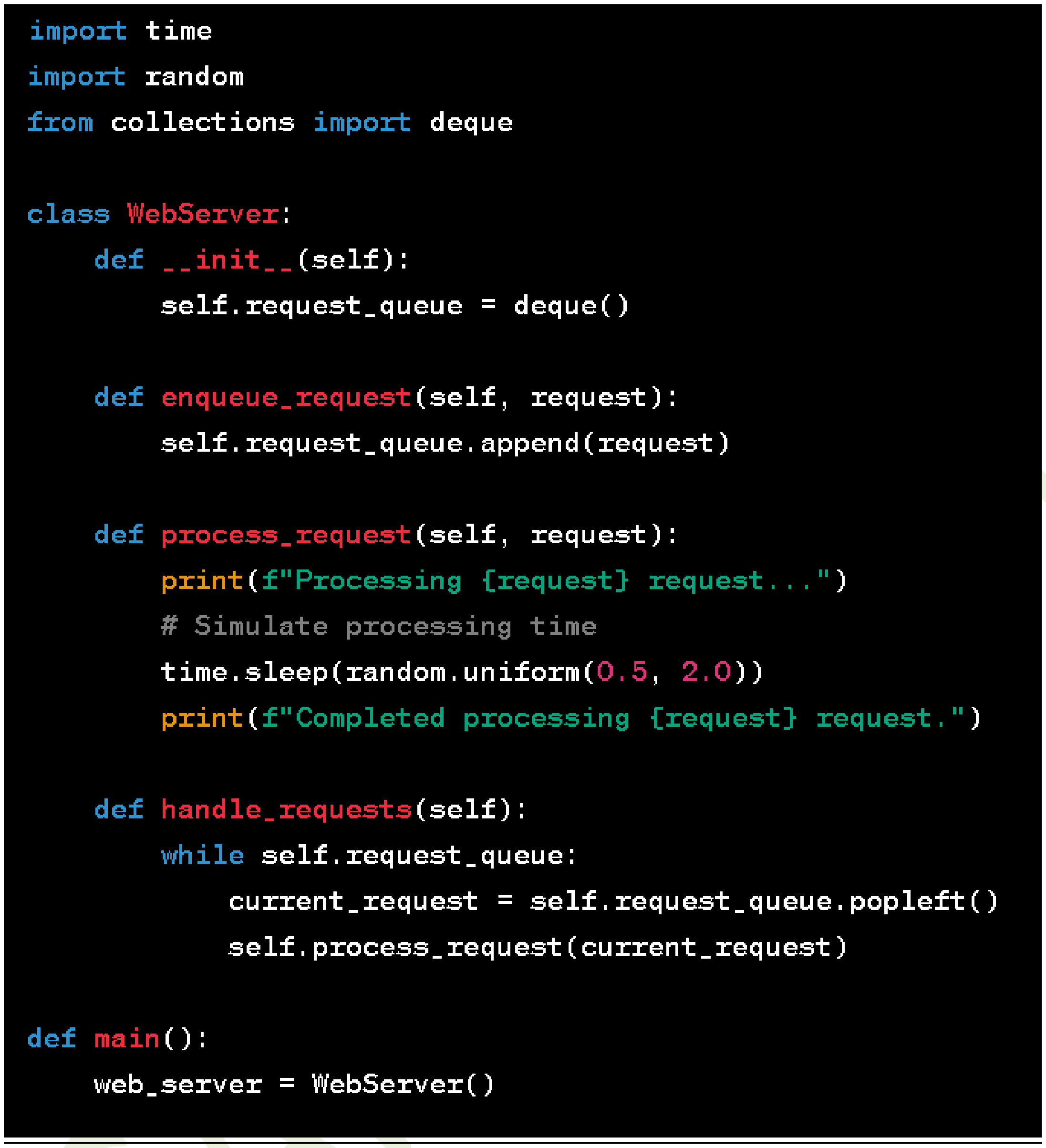
ASSIGNMENT: 02

SUBMITTED TO: SIR JAMAL

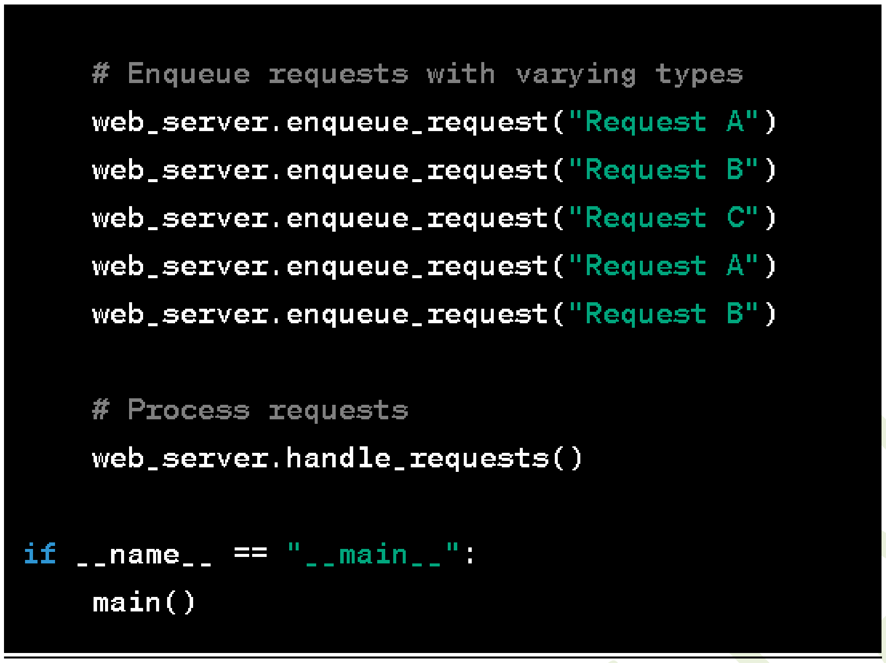
Question no 01

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.

Solution

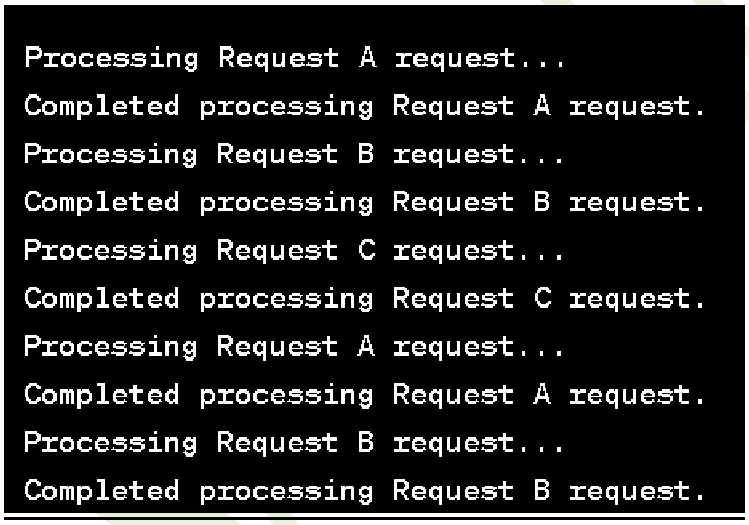


```python
import time
import random
from collections import deque

class WebServer:
    def __init__(self):
        self.request_queue = deque()

    def enqueue_request(self, request):
        self.request_queue.append(request)

    def process_request(self, request):
        print(f"Processing {request} request...")
        # Simulate processing time
        time.sleep(random.uniform(0.5, 2.0))
        print(f"Completed processing {request} request.")

    def handle_requests(self):
        while self.request_queue:
            current_request = self.request_queue.popleft()
            self.process_request(current_request)

def main():
    web_server = WebServer()
```



```python
    # Enqueue requests with varying types
    web_server.enqueue_request("Request A")
    web_server.enqueue_request("Request B")
    web_server.enqueue_request("Request C")
    web_server.enqueue_request("Request A")
    web_server.enqueue_request("Request B")

    # Process requests
    web_server.handle_requests()

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

<u>Output :</u>



```
Processing Request A request...
Completed processing Request A request.
Processing Request B request...
Completed processing Request B request.
Processing Request C request...
Completed processing Request C request.
Processing Request A request...
Completed processing Request A request.
Processing Request B request...
Completed processing Request B request.
```

<u>Question no 02</u>

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

<u>Solution</u>

The choice between a linked list and an array implementation for a queue depends on the specific requirements and characteristics of the application. Here are some scenarios where you might prefer one over the other:

Linked List Implementation for a Queue:

1. **Dynamic Size:** If the size of the queue needs to change frequently, and you want a data structure that can efficiently handle dynamic resizing, a linked list is a good choice. Linked lists can easily grow or shrink in size without the need for resizing or copying elements.
2. **Insertions and Deletions:** If there are frequent insertions and deletions at both ends of the queue (enqueue and dequeue operations), linked lists provide constant time O(1) complexity for these operations, making them more efficient than arrays.
3. **Memory Efficiency:** Linked lists can be more memory-efficient than arrays because they don't require a fixed contiguous block of memory. Each node in a linked list can be located anywhere in memory, allowing for more flexible memory allocation.

Array Implementation for a Queue:

1. **Random Access:** If there is a need for constant-time random access to elements, arrays are a better choice. Array elements can be accessed directly using their index, while in a linked list, you would need to traverse the list sequentially to reach a specific element.
2. **Sequential Access Patterns:** If the queue operations involve mostly sequential access (e.g., processing elements in a first-in-first-out manner), arrays might be more cache-friendly, resulting in better performance due to better utilization of the CPU cache.
3. **Simplicity and Space Overhead:** For small to moderately sized queues where the overhead of maintaining pointers in a linked list is significant, an array implementation can be simpler and more space-efficient.

In summary, the choice between a linked list and an array implementation for a queue depends on the specific use case, the expected patterns of access, and the trade-offs between factors such as dynamic resizing, memory efficiency, and random access performance.

Output :

Question no 03

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

Solution

In a basic queue implemented with an array, the time complexity of enqueue and dequeue operations depends on the implementation details:

1. **Enqueue Operation:**
   - In the basic array implementation, the enqueue operation involves adding an element to the end of the array. If the array is not full, this operation takes constant time O(1).
   - However, if the array is full, you might need to resize the array to accommodate more elements. In such cases, the enqueue operation might take O(n) time, where n is the current size of the array, as it involves copying elements to a new, larger array.
2. **Dequeue Operation:**
   - In the basic array implementation, the dequeue operation involves removing an element from the front of the array. This operation takes O(n) time, where n is the number of elements in the array, as it requires shifting all remaining elements to fill the gap.
   - If you use a circular queue implementation, where the front and rear indices wrap around the array, the dequeue operation can be optimized to O(1) time since there is no need to shift elements. Instead, you simply move the front index to the next position.

Optimizations:

1. **Circular Queue:**
   - To optimize the dequeue operation, use a circular queue implementation. This allows you to reuse the space in the array efficiently, avoiding the need to shift elements when dequeuing. Ensure proper handling of indices to wrap around when reaching the end of the array.
2. **Dynamic Array Resizing:**
   - If resizing is a concern, you can use dynamic array resizing strategies. For example, you could double the array size when it's full and halve it when it's less

than half full. This amortizes the cost of resizing over multiple enqueue operations, leading to an average constant-time complexity.

3. **Linked List Implementation:**
   - If dynamic resizing is a frequent concern and constant-time enqueue and dequeue operations are essential, consider using a linked list implementation. Linked lists allow for dynamic resizing without the need for shifting elements.
4. **Preallocation:**
   - If the maximum size of the queue is known in advance, preallocate the array to the maximum size. This eliminates the need for resizing operations, making enqueue and dequeue consistently O(1) operations.

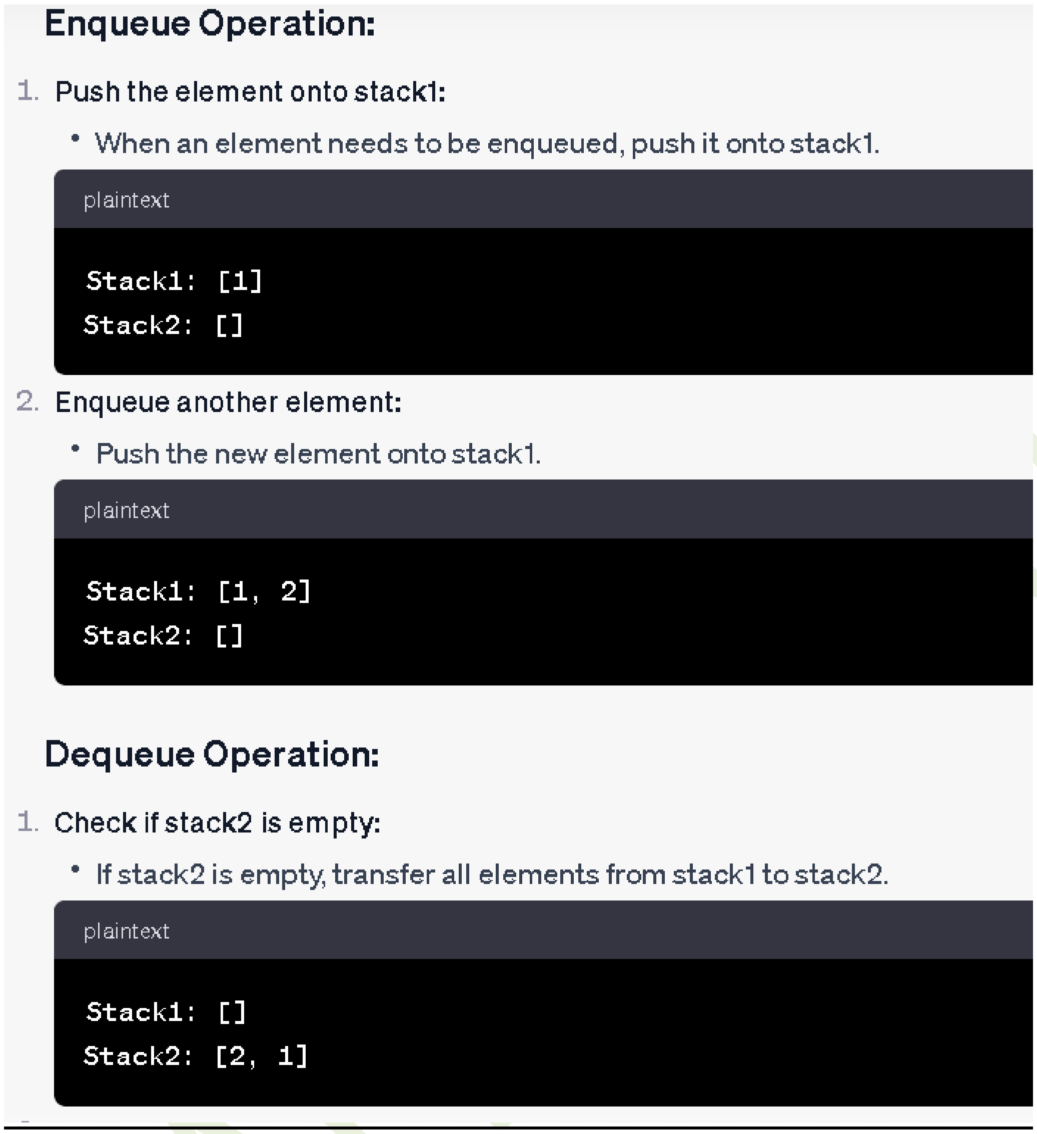
Choosing the right optimization depends on the specific use case and the patterns of enqueue and dequeue operations in your application. Circular queues, dynamic resizing, and linked list implementations each have their advantages depending on the particular requirements of the system.

Output :

Question no 04

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.

Solution



Enqueue Operation:

1. Push the element onto stack1:
   - When an element needs to be enqueued, push it onto stack1.

```plaintext
Stack1: [1]
Stack2: []
```

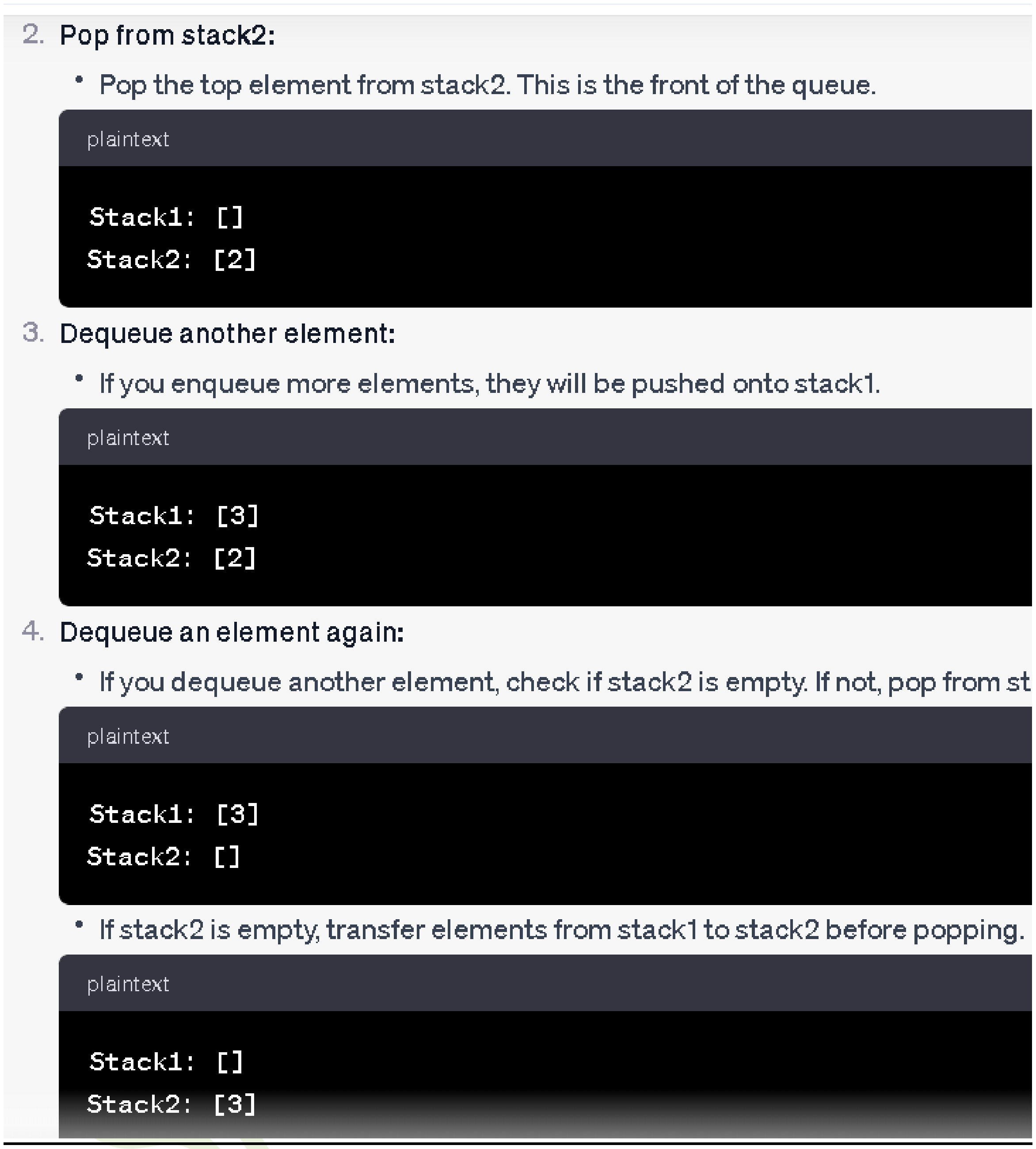
2. Enqueue another element:
   - Push the new element onto stack1.

```plaintext
Stack1: [1, 2]
Stack2: []
```

Dequeue Operation:

1. Check if stack2 is empty:
   - If stack2 is empty, transfer all elements from stack1 to stack2.

```plaintext
Stack1: []
Stack2: [2, 1]
```



2. **Pop from stack2:**
   - Pop the top element from stack2. This is the front of the queue.

```plaintext
Stack1: []
Stack2: [2]
```

3. **Dequeue another element:**
   - If you enqueue more elements, they will be pushed onto stack1.

```plaintext
Stack1: [3]
Stack2: [2]
```

4. **Dequeue an element again:**
   - If you dequeue another element, check if stack2 is empty. If not, pop from st

```plaintext
Stack1: [3]
Stack2: []
```

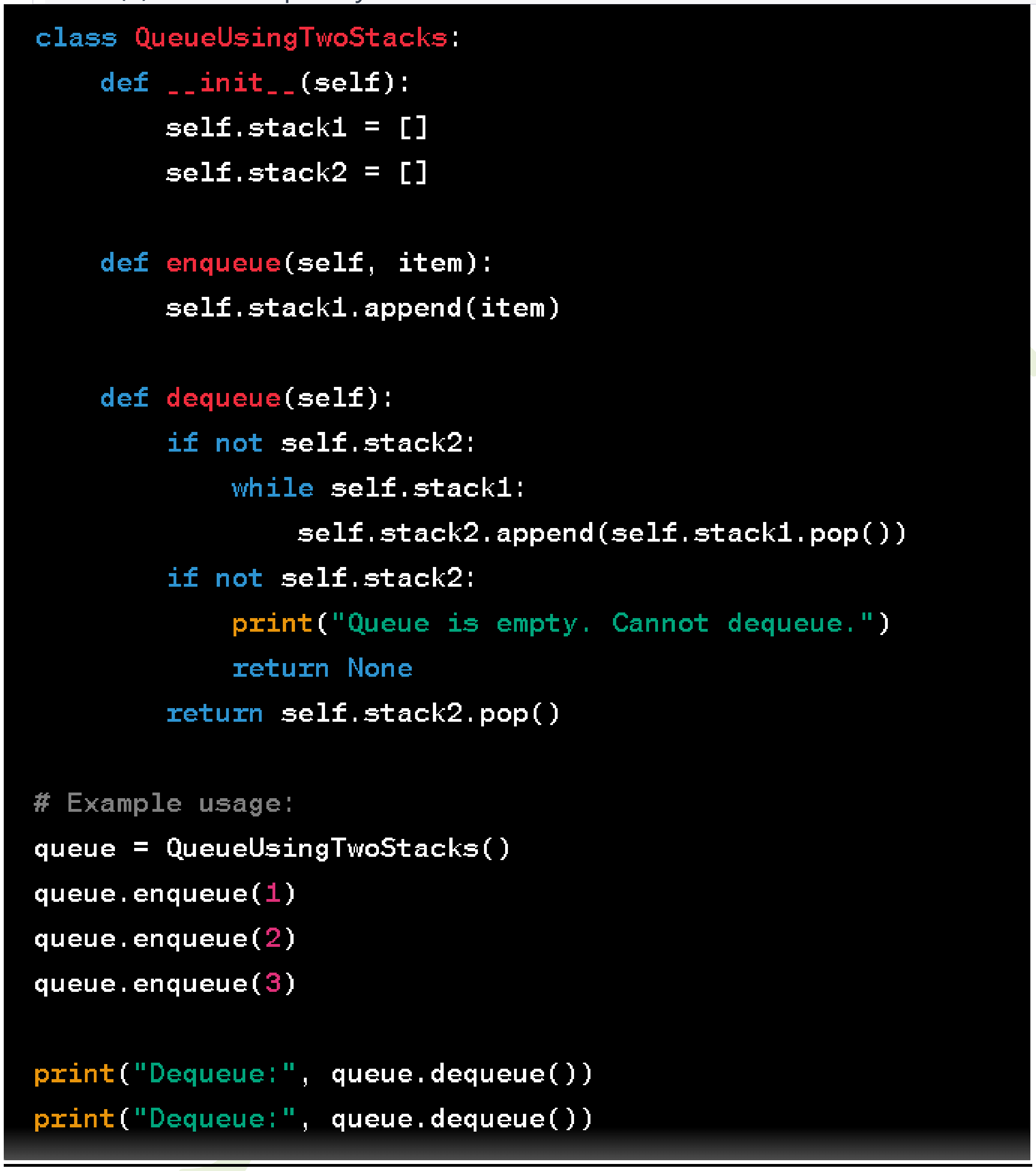
   - If stack2 is empty, transfer elements from stack1 to stack2 before popping.

```plaintext
Stack1: []
Stack2: [3]
```

Time Complexity:

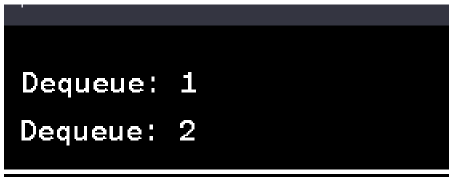
- **Enqueue Operation:**
  - O(1) time complexity, as it involves a simple push operation onto stack1.
- **Dequeue Operation:**
  - In the worst case, when stack2 is empty and we need to transfer elements from stack1, it takes O(n) time, where n is the number of elements in the queue.

- However, on average, if we consider a series of enqueue and dequeue operations, each element is transferred from stack1 to stack2 only once, leading to amortized O(1) time complexity.



```python
class QueueUsingTwoStacks:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []

    def enqueue(self, item):
        self.stack1.append(item)

    def dequeue(self):
        if not self.stack2:
            while self.stack1:
                self.stack2.append(self.stack1.pop())
        if not self.stack2:
            print("Queue is empty. Cannot dequeue.")
            return None
        return self.stack2.pop()

# Example usage:
queue = QueueUsingTwoStacks()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Dequeue:", queue.dequeue())
print("Dequeue:", queue.dequeue())
```

Output :



```
Dequeue: 1
Dequeue: 2
```