UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 9

Queues

Submitted by:
Adoracion, Jerick Dave D.

Instructor:
Engr. Maria Rizette H. Sayo

October 11, 2025

I. Objectives

Introduction

Another fundamental data structure is the queue. It is a close “the same” of the stack, as a queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle. That is, elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed.

The Queue Abstract Data Type

Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the first element in the queue, and element insertion is restricted to the back of the sequence. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in, first-out (FIFO) principle. The queue abstract data type (ADT) supports the following two fundamental methods for a queue Q:

Q.enqueue(e): Add element e to the back of queue Q.

Q.dequeue( ): Remove and return the first element from queue Q;
an error occurs if the queue is empty.

The queue ADT also includes the following supporting methods (with first being analogous to the stack’s top method):

Q.first(): Return a reference to the element at the front of queue Q, without removing it;
an error occurs if the queue is empty.

Q.is empty( ): Return True if queue Q does not contain any elements.

len(Q): Return the number of elements in queue Q; in Python, we implement this with the special method len .

This laboratory activity aims to implement the principles and techniques in:

- Writing Python program using Queues

Writing a Python program that will implement Queues operations

II. Methods

Instruction: Type the python codes below in your Colab. Reconstruct them by implementing Queues (FIFO) algorithm. Hint: You may use Array or Linked List

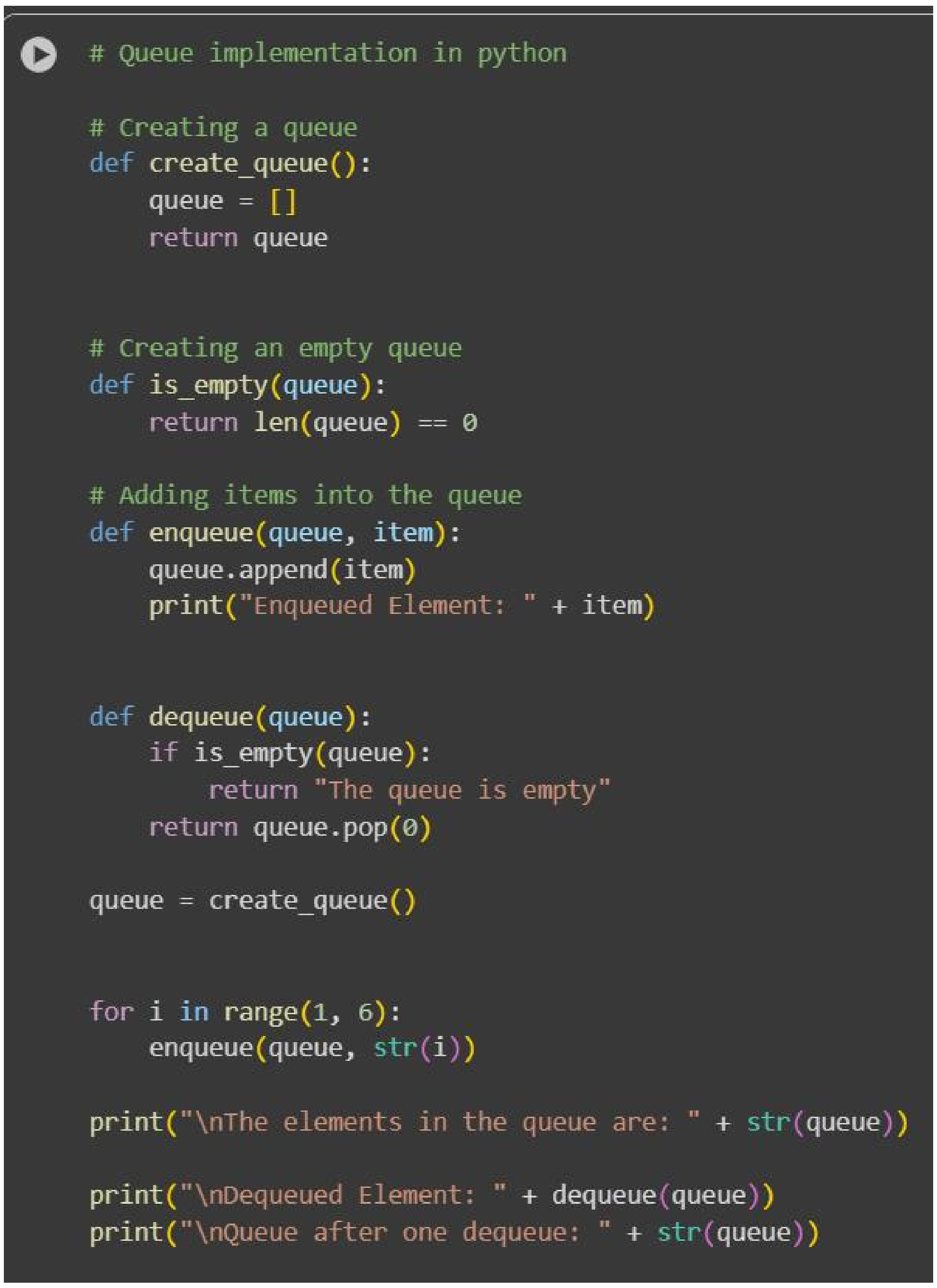
```
# Stack implementation in python

# Creating a stack
def create_stack():
    stack = []
    return stack
```

```
# Creating an empty stack
def is_empty(stack):
    return len(stack) == 0

# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("Pushed Element: " + item)

# Removing an element from the stack
def pop(stack):
    if (is_empty(stack)):
        return "The stack is empty"
    return stack.pop()

stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
push(stack, str(5))

print("The elements in the stack are:"+ str(stack))
```

Answer the following questions:

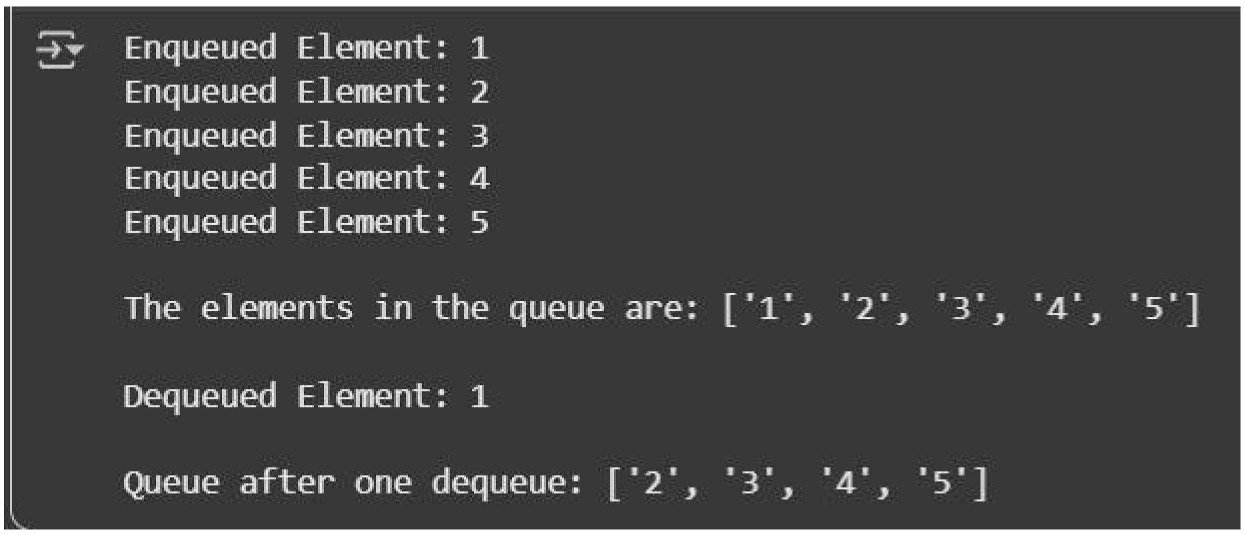
1. What is the main difference between the stack and queue implementations in terms of element removal?
2. What would happen if we try to dequeue from an empty queue, and how is this handled in the code?
3. If we modify the enqueue operation to add elements at the beginning instead of the end, how would that change the queue behavior?
4. What are the advantages and disadvantages of implementing a queue using linked lists versus arrays?
5. In real-world applications, what are some practical use cases where queues are preferred over stacks?

III. Results



```python
# Queue implementation in python

# Creating a queue
def create_queue():
    queue = []
    return queue


# Creating an empty queue
def is_empty(queue):
    return len(queue) == 0

# Adding items into the queue
def enqueue(queue, item):
    queue.append(item)
    print("Enqueued Element: " + item)


def dequeue(queue):
    if is_empty(queue):
        return "The queue is empty"
    return queue.pop(0)

queue = create_queue()


for i in range(1, 6):
    enqueue(queue, str(i))

print("\nThe elements in the queue are: " + str(queue))

print("\nDequeued Element: " + dequeue(queue))
print("\nQueue after one dequeue: " + str(queue))
```

Figure 1: Source Code



```
Enqueued Element: 1
Enqueued Element: 2
Enqueued Element: 3
Enqueued Element: 4
Enqueued Element: 5

The elements in the queue are: ['1', '2', '3', '4', '5']

Dequeued Element: 1

Queue after one dequeue: ['2', '3', '4', '5']
```

Figure 2: Output

1. **What is the main difference between the stack and queue implementations in terms of element removal?**

In stack, it uses LIFO (Last In, First Out), while in Queue, it uses First In, Fast Out (FIFO). In terms of element removal, in stack, the top/ end is removed first. While in queues, the front/ start is the first to be removed.

2. **What would happen if we try to dequeue from an empty queue, and how is this handled in the code?**

We can't dequeue an empty queue. If we were to run it, it would return as "The queue is empy".

3. **If we modify the enqueue operation to add elements at the beginning instead of the end, how would that change the queue behavior?**

It won't follow the usual First In, First Out pattern anymore. As it will now act as a stack because the most recently enqueued item will be the first one dequeued.

4. **What are the advantages and disadvantages of implementing a queue using linked lists versus arrays?**

A linked list–based queue is more efficient for frequent insertions and deletions because elements can be added or removed in constant time without shifting. It also grows dynamically, but requires extra memory for pointers and is more complex to implement. In contrast, an array-based queue is simpler and allows direct element access, but removing elements from the front is slower) since it involves shifting, and the queue size is fixed unless dynamically resized.

5. **In real-world applications, what are some practical use cases where queues are preferred over stacks?**

If we were to apply it in real-world applications, I think, one of the best examples of Queue is printing documents and files in a printer wherein the pages are arranged on a queue (FIFO) order. It is better to implement than stack because it helps the user to determine and arrange the order of papers for better use.

IV. Conclusion

In conclusion, this laboratory activity helped me understand the difference between stacks and queues, especially how their operations affect data order and performance. By implementing both structures in Python, I learned how LIFO and FIFO principles work in practice and how different data structures like arrays and linked lists impact efficiency. This activity strengthened my understanding of fundamental data handling, which is essential in computer engineering applications such as memory management, scheduling, and algorithm design.

References

[1] Co Arthur O.. “University of Caloocan City Computer Engineering Department Honor Code,” UCC-CpE Departmental Policies, 2020.

[2] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[4] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1996.

[5] Python Software Foundation, “Python Data Structures — Python 3 Documentation,” 2025. [Online].