UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 9

Queues

Submitted by:
Enverzo, Kyle Andrey 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

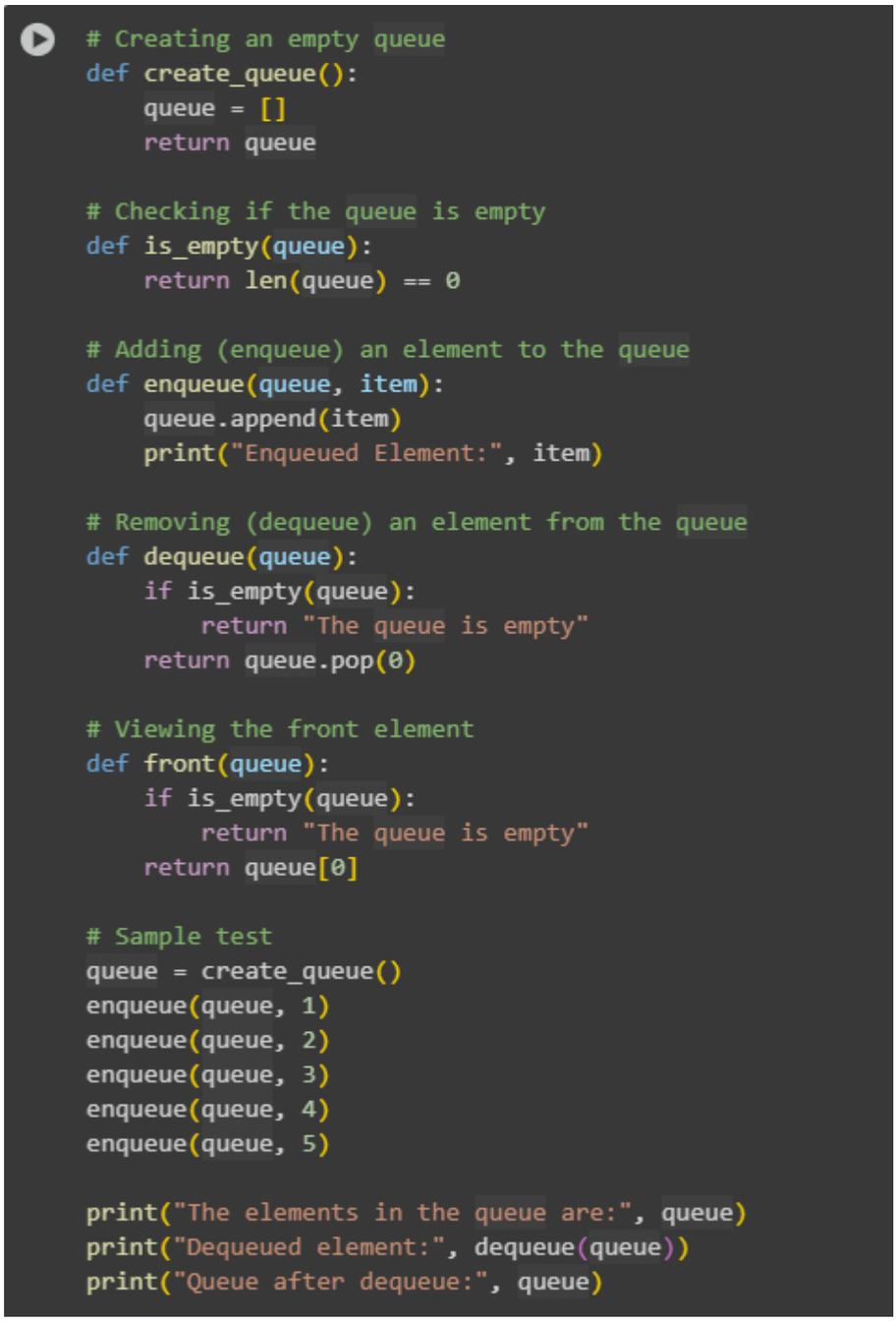
```
# Stack implementation in python

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

# Checking if the queue is empty
def is_empty(queue):
    return len(queue) == 0

# Adding (enqueue) an element to the queue
def enqueue(queue, item):
    queue.append(item)
    print("Enqueued Element:", item)

# Removing (dequeue) an element from the queue
def dequeue(queue):
    if is_empty(queue):
        return "The queue is empty"
    return queue.pop(0)

# Viewing the front element
def front(queue):
    if is_empty(queue):
        return "The queue is empty"
    return queue[0]

# Sample test
queue = create_queue()
enqueue(queue, 1)
enqueue(queue, 2)
enqueue(queue, 3)
enqueue(queue, 4)
enqueue(queue, 5)

print("The elements in the queue are:", queue)
print("Dequeued element:", dequeue(queue))
print("Queue after dequeue:", queue)
```

Figure 1 Screenshot of program

The following Python code implements a queue using a list and functions to interact with the queue. The queue is initialized with the function create_queue(), and the is_empty() test whether there are any elements on it. the enqueue() works to add elements to the ends of the queue and dequeue() works to remove from an element, following First In, First Out (FIFO). The front() function prints the first element without removing. The main part of the program collects a few elements, displays them and removes one to illustrate the workings of a queue in sequence.Conclusion

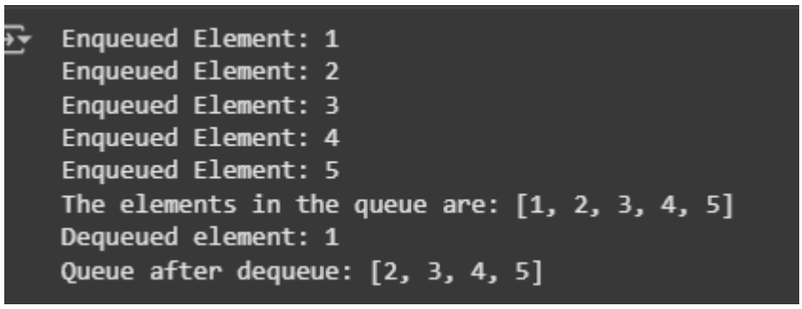


Figure 2 Screenshot of output

The output is a demonstration of the elements added in the Queue by calling enqueue() function. After pushing the numbers 1 -> 5, the program will print out the queue [1,2,3,4,5]. When dequeue( ) is called, the first element (1) comes out of the queue. And then The program also shows that after removing one element, the new list becomes [2, 3, 4, 5], which confirms our claim that the first item inserted is indeed being removed.The example explains to us how the concept of queue is implemented in Python and demonstrates a simple FIFO (First In First Out).

Answer:

1. The main difference between the stack and queue implementations in terms of element removal is that a stack follows the *Last In, First Out (LIFO)* principle, while a queue follows the *First In, First Out (FIFO)* principle, meaning that in a queue, the first element added is the first one to be removed.

2. If we try to dequeue from an empty queue, the program checks whether the queue is empty using the is_empty() function and displays the message “The queue is empty” instead of causing an error.

3. If the enqueue operation is modified to add elements at the beginning instead of the end, the queue will behave like a stack, since the most recently added element will be removed first.

4. Implementing a queue using a linked list has the advantage of allowing dynamic memory allocation and easy insertion or deletion, but it requires more memory due to pointer storage; on the other hand, using an array is simpler and provides faster access, but it has a fixed size and may require resizing when full.

5. Queues are preferred in real-world applications such as print job scheduling, customer service systems, task scheduling in operating systems, breadth-first search (BFS) algorithms, and network packet management, where processing must occur in the same order that tasks arrive.

Conclusion

In summary, the program is able to effectively illustrate the idea and functionality of a queue using Python. The program illustrates how elements are inserted and deleted in a First In, First Out (FIFO) fashion. With the help of functions such as enqueue(), dequeue(), and front(), the program is able to well depict how queues handle data in an orderly fashion. This exercise served to learn how queues function and the significance of queues in actual applications including task scheduling, data handling, and service systems.

References

[1] W. Stallings, *Data and Computer Communications*, 10th ed. Upper Saddle River, NJ, USA: Pearson Education, 2013.

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

[3] M. Lutz, *Learning Python*, 5th ed. Sebastopol, CA, USA: O'Reilly Media, 2013.

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

[5] Python Software Foundation, "Python Data Structures — list, queue, and stack operations," *Python.org*, 2024. [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html