UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 9

Queues

Submitted by:
Luminario, Venice Lou Gabrielle M.

Instructor:
Engr. Maria Rizette H. Sayo

Oct 22, 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?

The main difference between the stack and queue implementations in terms of element removal lies in the order and position from which elements are removed, which reflects their data structure principles, the Stack (LIFO - Last In, First Out), and the Queue (FIFO - First In, First Out).

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

Attempting to dequeue from an empty queue triggers the is_empty check, returning "The queue is empty" without removal and preventing errors like IndexError.

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

If you tweak the enqueue to insert elements at the front (like using queue.insert(0, item)), it flips the queue's FIFO nature upside down, making it act more like a stack (LIFO)—the newest item added would jump to the front and get dequeued first.

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

Arrays rock for queues with fast random access and low memory waste, but front removals drag (O(n) shifts) and resizing can overflow or underuse space—like cramming into fixed spots.

Linked lists adapt dynamically with O(1) end ops, no shifting needed, but they hog pointer memory, slow peeks, and code up messier.

arrays (or deque) for steady, zippy needs; linked lists for fluid, edge-focused flows.

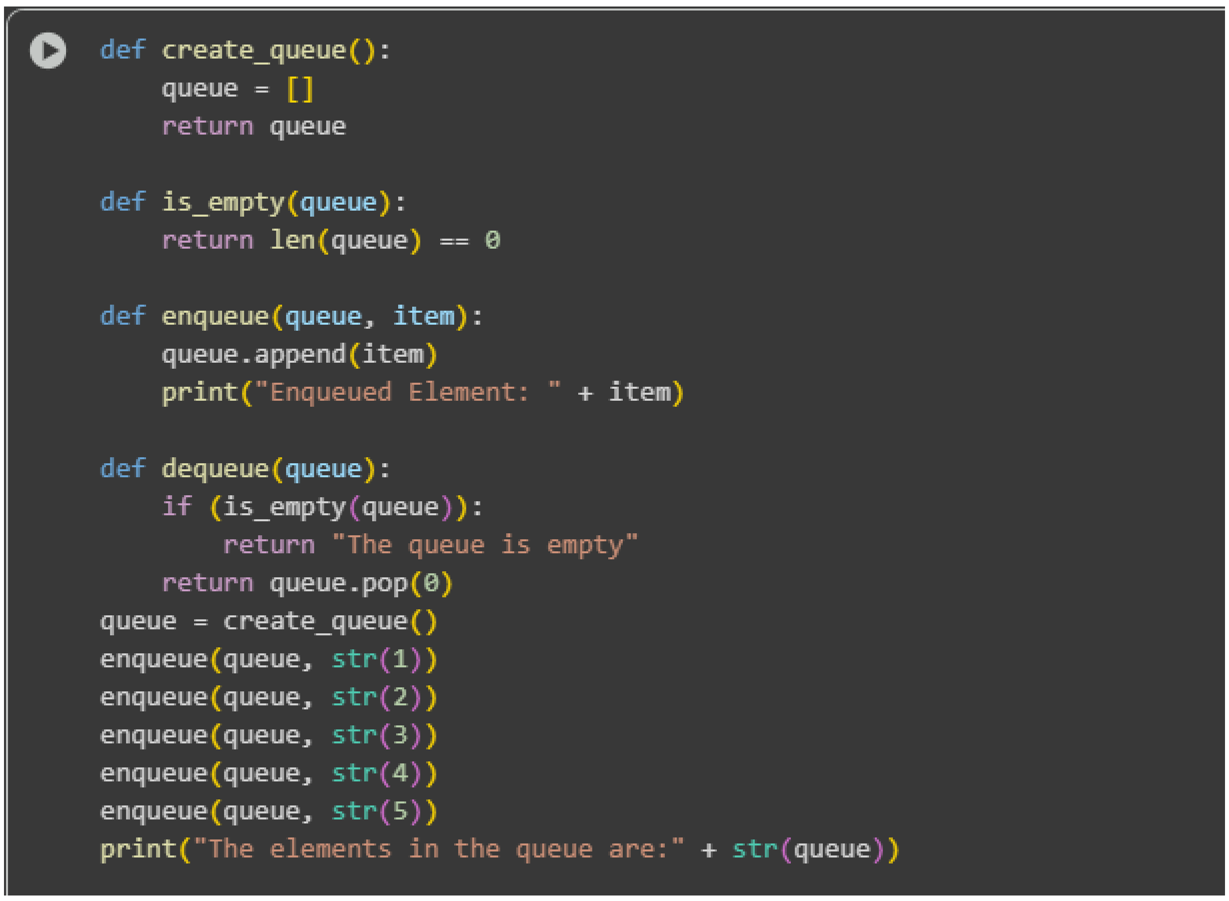
5 In real-world applications, what are some practical use cases where queues are preferred over stacks?

Queues outshine stacks in real-world spots needing "first in, first out" fairness like OS task scheduling, print jobs, network buffering for smooth streams, customer service lines, graph BFS for shortest paths, or simulating wait times whenever order of arrival matters more than recency, dodging the LIFO chaos.

III. Results

Present the visualized procedures done. Also present the results with corresponding data visualizations such as graphs, charts, tables, or image . Please provide insights, commentaries, or explanations regarding the data. If an explanation requires the support of literature such as academic journals, books, magazines, reports, or web articles please cite and reference them using the IEEE format.

Please take note of the styles on the style ribbon as these would serve as the style format of this laboratory report. The body style is Times New Roman size 12, line spacing: 1.5. Body text should be in Justified alignment, while captions should be center-aligned. Images should be readable and include captions. Please refer to the sample below:



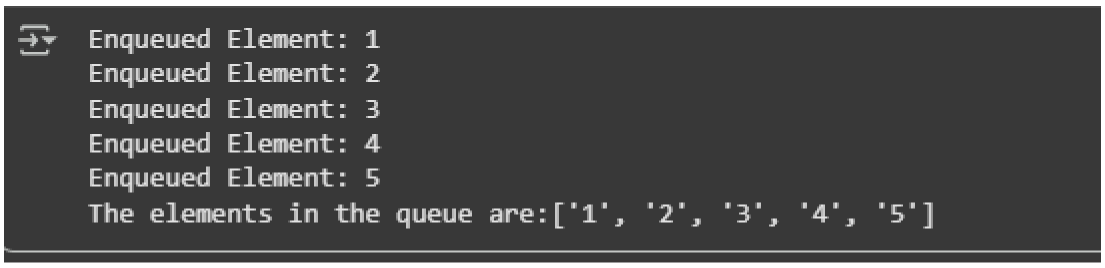
```python
def create_queue():
    queue = []
    return queue

def is_empty(queue):
    return len(queue) == 0

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()
enqueue(queue, str(1))
enqueue(queue, str(2))
enqueue(queue, str(3))
enqueue(queue, str(4))
enqueue(queue, str(5))
print("The elements in the queue are:" + str(queue))
```

Figure 1 Screenshot of program



```
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']
```

Figure 2 Results

If an image is taken from another literature or intellectual property, please cite them accordingly in the caption. Always keep in mind the Honor Code [1] of our course to prevent failure due to academic dishonesty.

IV. Conclusion

The conclusion expresses the summary of the whole laboratory report as perceived by the authors of the report.

I learned in this python queue code captures FIFO essentials with smooth enqueues at the end and dequeues from the front, smartly dodging errors on empty queues—great for simple apps, but level up to collections.deque for real speed without the shifty slowdowns.

References

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