UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 9

Queues

Submitted by:
Palmes, Lewis Clark L.

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?

Answer:
The main difference between stacks and queues lies in the order elements are removed. A stack uses the LIFO (Last-In-First-Out) principle, meaning the most recently added element is removed first using the pop() operation. On the other hand, a queue follows the FIFO (First-In-First-Out) principle, so the earliest added element is removed first using the dequeue() operation. This difference affects how data is processed and accessed in each structure.

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

Answer:
If we try to dequeue from an empty queue, there are no elements to remove, which would normally cause an error. In the code, this is handled safely by first checking if the queue is empty using the is_empty() function. If it is, the program returns a message like "The queue is empty" instead of attempting to remove an element, preventing runtime errors.

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

Answer:
If we modify the enqueue operation to add elements at the beginning of the queue instead of the end, the queue's behavior would reverse, it would start functioning like a stack. This is because the newest element would be at the front and would be removed first during a dequeue operation, changing the behavior from FIFO to LIFO.

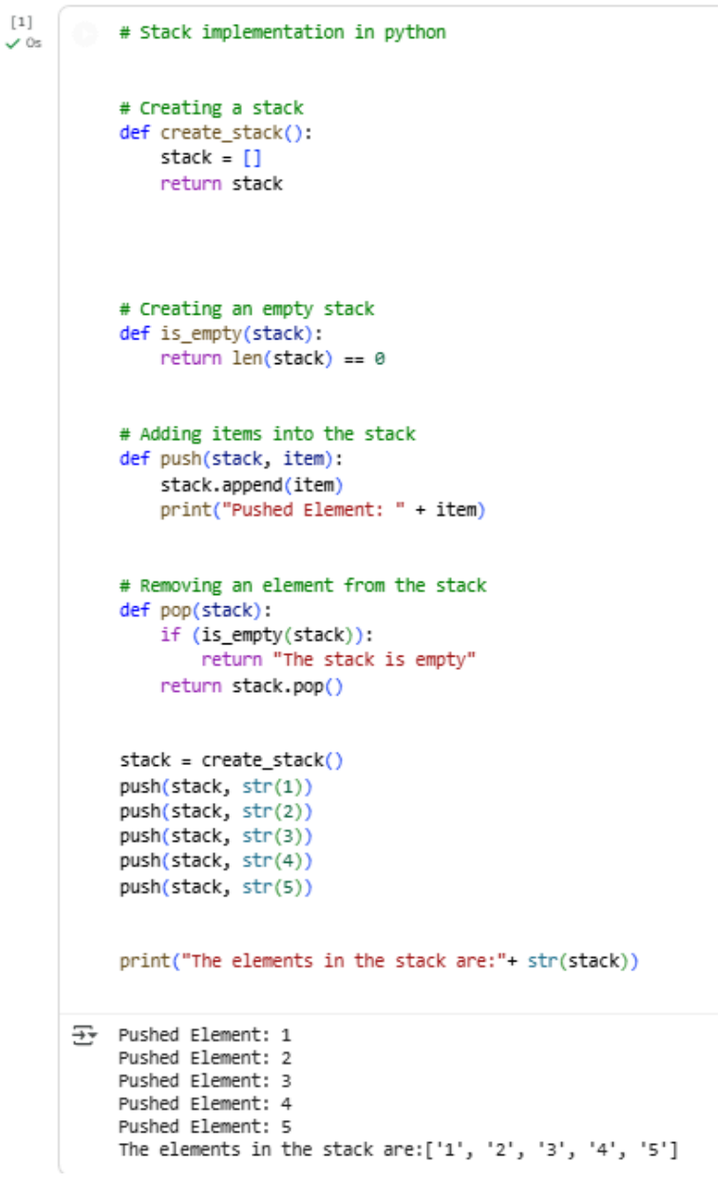
4 What are the advantages and disadvantages of implementing a queue using linked lists versus arrays?

Answer:
Using a linked list for a queue provides dynamic memory allocation, meaning the size can grow or shrink as needed without worrying about capacity limits. However, it requires more memory due to node pointers. On the other hand, an array-based queue is simpler and faster to access but may require resizing when it becomes full, which can be inefficient for large data sets.

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

Answer:
Queues are commonly used in situations where tasks must be handled in the exact order they arrive. Examples include printer task scheduling, CPU process management, customer service systems, network data packet handling, and message queues in software. These applications rely on the FIFO order to ensure fairness and proper sequence of operations.

III. Results



```
[1]
✓ 0s
```

```python
# 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))
```

```
Pushed Element: 1
Pushed Element: 2
Pushed Element: 3
Pushed Element: 4
Pushed Element: 5
The elements in the stack are:['1', '2', '3', '4', '5']
```

Figure 1 Screenshot of program

This screenshot shows the original code that I reconstructed into a Queue implementation. The program here demonstrates a Stack (LIFO) structure where elements are added and removed from the same end, meaning the last element pushed is the first one removed. In the reconstructed version, I modified this logic to follow the Queue (FIFO) principle, where elements are added at the rear and removed from the front. Thus, this original stack code served as the foundation for creating the new queue implementation.

[7] ✓ 0s

```python
# Queue implementation in Python (FIFO)

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)

def display(queue):
    print("The elements in the queue are: " + str(queue))

queue = create_queue()
enqueue(queue, str(1))
enqueue(queue, str(2))
enqueue(queue, str(3))
enqueue(queue, str(4))
enqueue(queue, str(5))

display(queue)

print("Dequeued Element:", dequeue(queue))
print("Queue after dequeue:", queue)
```

```
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 dequeue: ['2', '3', '4', '5']
```

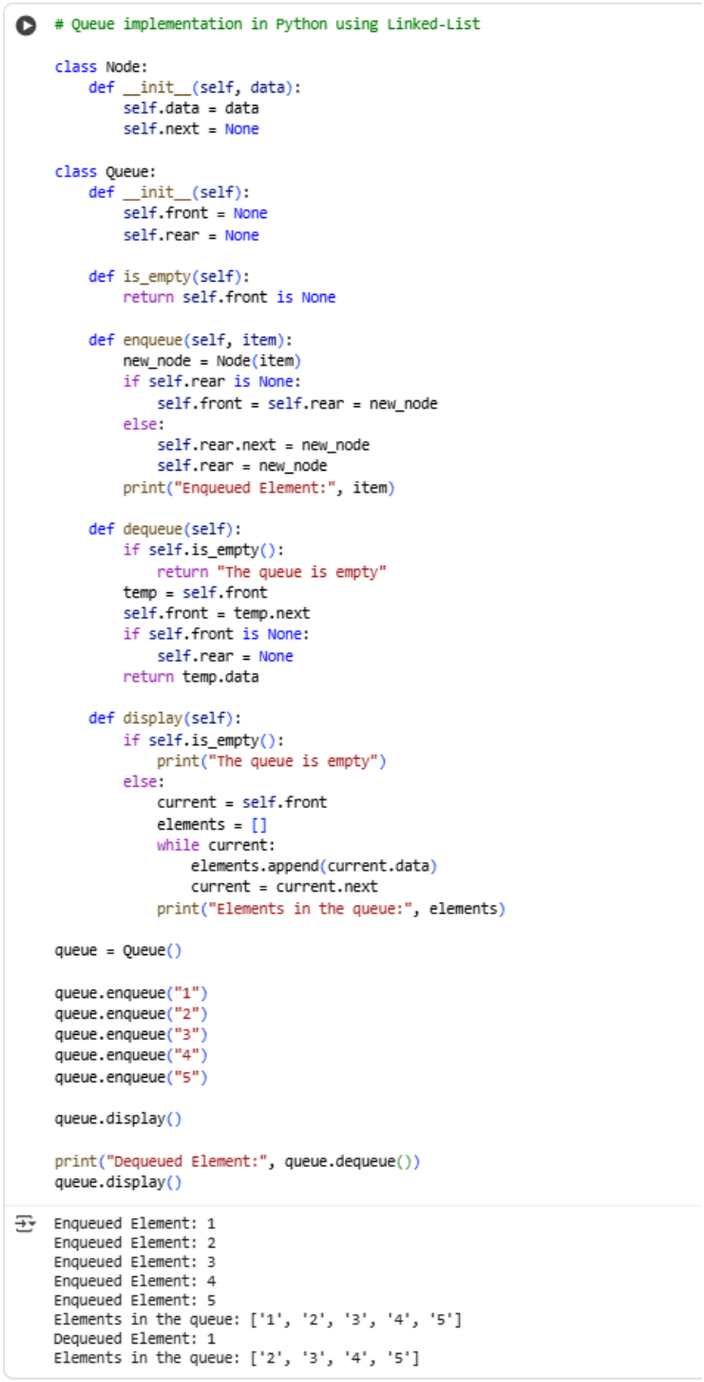
Figure 2 Screenshot of program

This screenshot shows the reconstructed version of the original stack code, now implemented as a Queue (FIFO) in Python. Unlike the stack, which removes the most recently added item first, this queue removes the oldest element first, following the First-In-First-Out principle. The enqueue() function adds new elements to the end of the list, while the dequeue() function removes elements from the front using pop(0). The output shows that after enqueueing elements 1 to 5, the first element (1) is dequeued, leaving ['2', '3', '4', '5']. This demonstrates how the queue processes data in the order it was added.

```
[6]
# Queue implementation in Python using Array

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)

def display(queue):
    print("Elements in the queue:", queue)

queue = create_queue()

enqueue(queue, str(1))
enqueue(queue, str(2))
enqueue(queue, str(3))
enqueue(queue, str(4))
enqueue(queue, str(5))

display(queue)

print("Dequeued Element:", dequeue(queue))
print("Queue after dequeue:", queue)

Enqueued Element: 1
Enqueued Element: 2
Enqueued Element: 3
Enqueued Element: 4
Enqueued Element: 5
Elements in the queue: ['1', '2', '3', '4', '5']
Dequeued Element: 1
Queue after dequeue: ['2', '3', '4', '5']
```

Figure 3 Screenshot of program

This screenshot shows the Queue implementation in Python using an Array (list). The program uses enqueue() to add elements to the end of the queue and dequeue() to remove elements from the front using pop(0). The output confirms that the first element added (1) is the first one removed during the dequeue operation, leaving the updated queue as ['2', '3', '4', '5']. This demonstrates the proper behavior of a queue, where elements are processed in the same order they were inserted.



```python
# Queue implementation in Python using Linked-List

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None

    def is_empty(self):
        return self.front is None

    def enqueue(self, item):
        new_node = Node(item)
        if self.rear is None:
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
        print("Enqueued Element:", item)

    def dequeue(self):
        if self.is_empty():
            return "The queue is empty"
        temp = self.front
        self.front = temp.next
        if self.front is None:
            self.rear = None
        return temp.data

    def display(self):
        if self.is_empty():
            print("The queue is empty")
        else:
            current = self.front
            elements = []
            while current:
                elements.append(current.data)
                current = current.next
            print("Elements in the queue:", elements)

queue = Queue()

queue.enqueue("1")
queue.enqueue("2")
queue.enqueue("3")
queue.enqueue("4")
queue.enqueue("5")

queue.display()

print("Dequeued Element:", queue.dequeue())
queue.display()
```

```
Enqueued Element: 1
Enqueued Element: 2
Enqueued Element: 3
Enqueued Element: 4
Enqueued Element: 5
Elements in the queue: ['1', '2', '3', '4', '5']
Dequeued Element: 1
Elements in the queue: ['2', '3', '4', '5']
```

Figure 4 Screenshot of program

This screenshot shows the Queue implementation in Python using a Linked List. In this version, each element is represented by a Node that stores data and a pointer to the next node. The Queue class manages two pointers, front (the first node) and rear (the last node). The enqueue() method adds new nodes at the rear, while the dequeue() method removes nodes from the front, maintaining the FIFO (First-In-First-Out) order. The output shows elements being enqueued from 1 to 5 and then dequeued from the front, leaving ['2', '3', '4', '5']. This implementation efficiently handles dynamic data without needing to resize an array, making it ideal for queues that grow or shrink frequently.

IV. Conclusion

Since we already tackled queues before, I actually found this activity pretty easy to apply using both arrays and linked lists. I already understand how the FIFO (First In, First Out) process works, so it felt more like a review and practice for me. Using arrays made it simple to visualize the order of elements, while using linked lists helped me see how data can move dynamically. Overall, it was a smooth and enjoyable task that helped me strengthen what I already know about queues.

References

[1]“Google Colab.”

https://colab.research.google.com/drive/1nxDSy70yp7siy24ZmIA-6nozYXTGc387#scrollTo=nlmo4JJtpdoY

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

[3] Lewis-Clark-Palmes, “CPE-201L-DSA-2-A/LAB-9.ipynb at main · Lewis-Clark-Palmes/CPE-201L-DSA-2-A,” *GitHub*. https://github.com/Lewis-Clark-Palmes/CPE-201L-DSA-2-A/blob/main/LAB-9.ipynb

[4] GeeksforGeeks, “Queue data structure,” *GeeksforGeeks*, Jul. 23, 2025. https://www.geeksforgeeks.org/dsa/queue-data-structure/