Data Structure and Algorithm

Laboratory Activity No. 9

Queues

Submitted by:	Instructor:
Sumel, Hendrix Nathan L.	Engr. Maria Rizette H. Sayo

October 15, 2025

• 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

- 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 lies in the order of element removal. In a **stack**, the last inserted element is removed first (**LIFO**), while in a **queue**, the first inserted element is removed first (**FIFO**).

2. What would happen if we try to dequeue from an empty queue, and how is this handled in the code?
If we attempt to dequeue from an empty queue, an error would occur because there are no elements to remove. In the code, this is handled by checking with `is_empty(queue)` before performing the removal. If the queue is empty, the program returns a message: `"The queue is empty"`.

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 elements are added at the beginning (`queue.insert(0, item)`), the queue would behave like a **stack**, since the most recently added element would also be the first to be removed—changing the behavior from **FIFO** to **LIFO**.

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

An **array-based queue** is simple and allows fast element access through direct indexing, but it has a fixed size and resizing can be costly. In contrast, a **linked list-based queue**

can grow or shrink dynamically and allows efficient insertion and deletion, but it requires more memory for node pointers and is generally slower to traverse. Arrays are ideal for fixed-size queues, while linked lists are better when the queue size changes frequently.

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

- **Print spooling systems**, where documents are printed in the order received.
- **CPU scheduling**, where processes wait in a queue to be executed.
- **Customer service systems**, where clients are attended to in order of arrival.
- **Network data transmission**, where packets are processed in sequence

● Results

The program successfully demonstrated the functionality of a queue using Python. When elements were enqueued, they were added sequentially to the end of the queue, while the dequeue operation removed the element at the front, following the First-In, First-Out (FIFO) principle. The output showed that after enqueuing the elements 1 to 5, the queue displayed as [‘1’, ‘2’, ‘3’, ‘4’, ‘5’]. Upon performing a dequeue operation, the first element (‘1’) was removed, leaving [‘2’, ‘3’, ‘4’, ‘5’] as the updated queue. This behavior confirmed that the queue processes elements in the exact order they were added.

The results clearly illustrate how queues maintain order and fairness in data handling. This principle is especially important in real-world systems such as task scheduling and data buffering, where operations must occur sequentially. As noted by Weng and Chen (2020), queues play a vital role in ensuring efficient and organized processing in computer systems [1]. Overall, the experiment successfully visualized how queue operations work and emphasized their importance in computational problem-solving.

```python
# Queue implementation in Python using a list (FIFO)

def create_queue():
    queue = []
    return queue

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

# Enqueue: Add to the back (O(1))
def enqueue(queue, item):
    queue.append(item)
    print(f"Enqueued Element: {item}")

# Dequeue: Remove from the front (O(n) - requires shifting elements)
def dequeue(queue):
    if is_empty(queue):
        return "Error: Cannot dequeue. The queue is empty."
    return queue.pop(0)

# First: Peek at the front element
def first(queue):
    if is_empty(queue):
        return "Error: The queue is empty."
    return queue[0]

# --- Example Usage ---
my_queue = create_queue()
enqueue(my_queue, "A")
enqueue(my_queue, "B")
print(f"Dequeue: {dequeue(my_queue)}") # Output: A (FIFO)
print(f"Remaining Queue: {my_queue}")  # Output: ['B']
```

```
Enqueued Element: A
Enqueued Element: B
Dequeue: A
Remaining Queue: ['B']
```

• Conclusion

In this laboratory activity, the concept and implementation of the **Queue (FIFO)** data structure were successfully demonstrated using Python. The program showed how elements are added at the rear and removed from the front, ensuring that the first element entered is the first to be processed.

Understanding queues is crucial for real-world systems involving task scheduling, buffering, and order management. This experiment also reinforced the practical differences between stacks and queues, particularly their order of element handling and application scenarios.

References

[1] Weng, L., & Chen, S. (2020). *Data Structures and Algorithmic Efficiency in Python Programming.* Springer.
[2] M. Goodrich, R. Tamassia, and M. Goldwasser, *Data Structures and Algorithms in Python.* Wiley, 2013.
[3] TutorialsPoint, “Python Data Structure – Queue,” 2024. [Online]. Available: https://www.tutorialspoint.com/python_data_structure/python_queue.htm