UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 9

# Queues

*Submitted by:*
Disomnong, Jalilah M.

*Instructor:*
Engr. Maria Rizette H. Sayo

10/11/2025

# 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

Below are the results of reconstructing the given source code by implementing Queues (FIFO) algorithm using an Array

[7]
✓ 0s

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

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

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

# Add item to the end of the queue (enqueue)
def enqueue(queue, item):
    queue.append(item)
    print("Enqueued Element: " + item)

# Remove item from the front of the queue (dequeue)
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 2 Screenshot of Queue sourcecode*

*link: [LAB 9 - Colab]*

```
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 1 Screenshot of Output*

*link: [LAB 9 - Colab]*

To answer question number 1, a stack follows the Last In, First Out (LIFO) principle, meaning the last element added is the first to be removed. This is like a stack of plates where you take the top

one off first. On the other hand, a queue operates on a First In, First Out (FIFO) basis, where the first element added is the first to be removed.

```
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))

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

*Figure 3 Screenshot of dequeue link: LAB 9 - Colab*

```
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
Dequeued Element: 2
The elements in the queue are: ['3', '4', '5']
```

*Figure 4 Output link: LAB 9 - Colab*

By answering question number 2, if we try to dequeue from an empty queue, Python will raise an IndexError because it tries to remove an item from an empty list. To prevent this, the operation is usually guarded with a check. A function like is_empty(queue) can be used before performing dequeue, and if the queue is empty, the code can return a message such as “The queue is empty” instead of proceeding with the removal.

While questions 3, if it changes the enqueue operation to add new items at the beginning of the queue instead of the end, it will no longer work like a queue. Instead, it will behave like a stack. That’s because both adding and removing would happen at the front, so the last item added would be the first one removed. This follows LIFO (Last In, First Out), which is how stacks work, not FIFO (First In, First Out) like queues.

In questions number 4, you can make queues using linked lists or arrays, but both have good and bad points. A linked list can grow bigger or smaller easily because it doesn’t have a fixed size. Adding and removing items is fast if done right, but it uses more memory and is a bit harder to write. An array (like a Python list) is easy to use and good for getting items by position. But taking items from the front is slow because everything behind has to move forward. Also, arrays can only hold a certain number of items unless you use special tricks to make them bigger.

Lastly, in number 5 questions, Queues are preferred in situations where the order of service matters, such as in food delivery. When customers place orders, the kitchen prepares and delivers them in the order they receive to ensure fairness. This way, the first order placed is the first to be completed, preventing any customer from being skipped or delayed. Queues work well here because they process requests on a first-come, first-served basis, unlike stacks which would serve the newest order first and could cause unfairness.

# IV. Conclusion

In this laboratory, we learned how queues work as a data structure that follows the First-In, First-Out (FIFO) rule. We practiced adding items to the back of the queue and removing them from the front using Python. This helps keep the order of items fair and organized.
Understanding queues is important because they are used in many real-world situations

# References

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

[2] Google. (2024). Google Colaboratory. Retrieved April 18, 2024, from https://colab.research.google.com/