Data Structure and Algorithm

Laboratory Activity No. 9

Queues

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| **LastName, FirstName MI.** | **Engr. Maria Rizette H. Sayo** |

10, 11, 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?
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?

# Results

**CODE RESULT SNIPPET:**

**A screen shot of a computer

AI-generated content may be incorrect.**

1. The main difference between stack and queue implementations in terms of element removal is in the order of removal. Stack uses (LIFO - Last In, First Out) Elements are removed from the same end they were added to (the "top"), while Queue uses (FIFO - First In, First Out) Elements are removed from the opposite end they were added to (the "front").

2. If you try to dequeue from an empty queue, it generally results in an error or an exception because there's no element to return. In this code if the stack is empty, it returns the string "The stack is empty" instead of giving an error.

3. If you modify the enqueue operation to add elements at the beginning of a queue structure instead of the end, the queue behavior would reverse, turning the structure into a stack. This means the element that is just added at the beginning is the first one available for removal, changing the principle from First In, First Out (FIFO) to Last In, First Out (LIFO).

4. THE ADVANTAGES OF QUEUE IMPLEMENTATION:

Access to any element. Better cache locality (faster data access in memory). Simpler to implement. Dynamic size (can grow/shrink easily), enqueue (add to tail) and dequeue (remove from head) operations.

THE DISADVANTAGES OF QUEUE IMPLEMENTATION:

|  |
| --- |
| Fixed size (can overflow). Dequeuing from the front often requires shifting all other elements, which is an inefficient operation. Higher memory overhead due to storing pointers/references. Access time to any element (must traverse from the head). Poorer cache locality. |

5. Queues are preferred over stacks in real-world applications whenever processing must be done in the order that requests or data arrived. Some practical use includes, handling client requests. The first request received is the first one served to prevent some clients from being starved.

# Conclusion

This laboratory activity introduced the Queue data structure, a fundamental concept in Computer Science. We tell that a queue operates on the First-In, First-Out (FIFO) principle, meaning the element that has been in the queue the longest is the next one removed. The core operations, as part of the Queue Abstract Data Type, were identified as to add an element to the back, and to remove and return the element from the front.

**References**

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