



UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 9

Queues

Submitted by:
Ruperto, April Anne A

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))

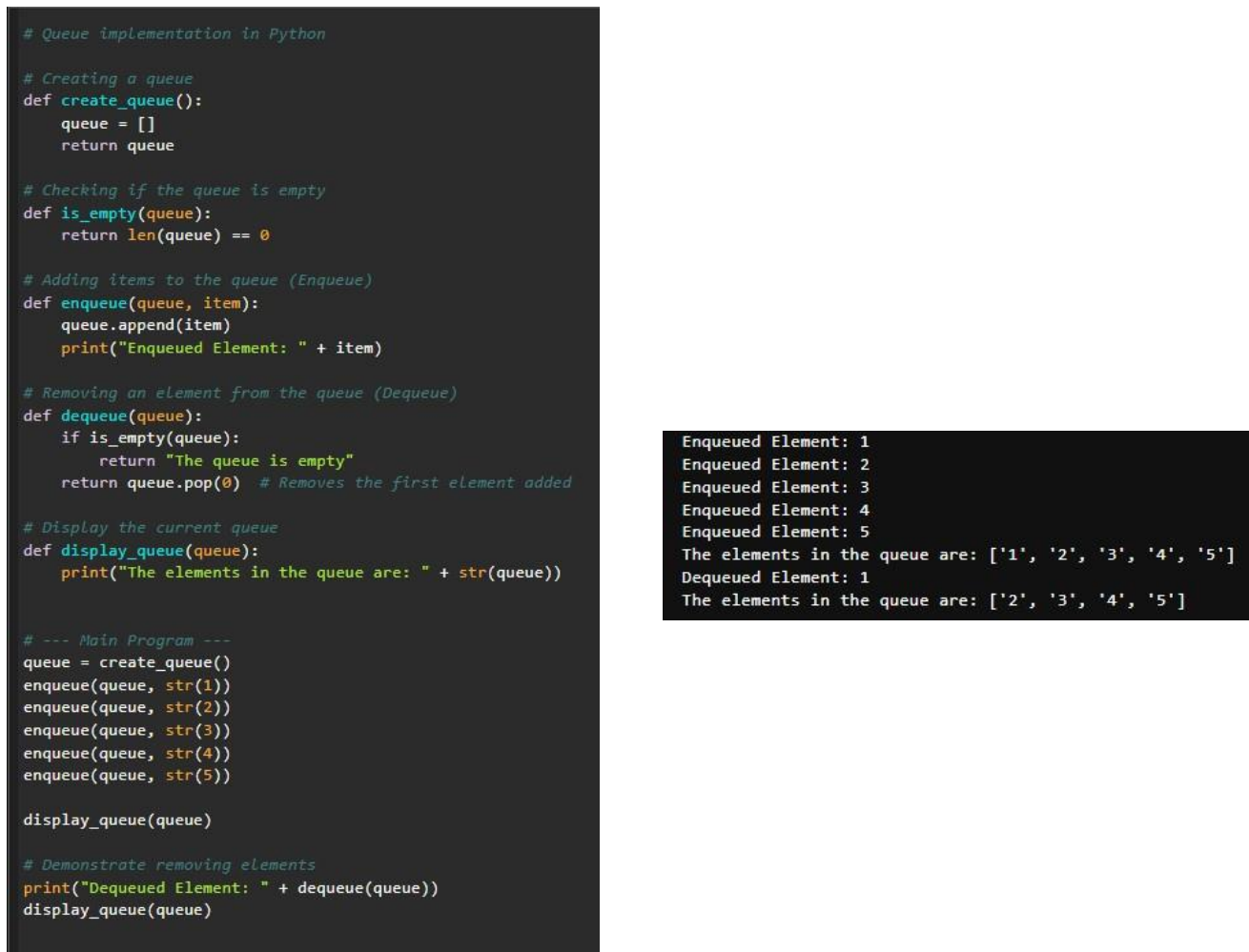
print("The elements in the stack are:" + str(stack))

```

Answer the following questions:

- 1 What is the main difference between the stack and queue implementations in terms of element removal?
- 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

The image shows a Python script for a queue implementation and its execution output. The script defines functions for creating a queue, checking if it's empty, enqueueing, dequeuing, and displaying the queue. The main program enqueues elements 1 through 5, displays the queue, dequeues the first element, and displays the queue again. The output shows the sequence of operations and the state of the queue at each step.

```
# Queue implementation in Python

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

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

# Adding items to the queue (Enqueue)
def enqueue(queue, item):
    queue.append(item)
    print("Enqueued Element: " + item)

# Removing an element from the queue (Dequeue)
def dequeue(queue):
    if is_empty(queue):
        return "The queue is empty"
    return queue.pop(0) # Removes the first element added

# Display the current queue
def display_queue(queue):
    print("The elements in the queue are: " + str(queue))

# --- Main Program ---
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(queue)

# Demonstrate removing elements
print("Dequeued Element: " + dequeue(queue))
display_queue(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
The elements in the queue are: ['2', '3', '4', '5']
```

Figure 1 & 2: Screenshot of the Program

Please follow this link: [Ruperto-April-Anne/CPE-201L-DSA-2/DSA_Laboratory_9](https://github.com/Ruperto-April-Anne/CPE-201L-DSA-2/DSA_Laboratory_9)

Answer/s:

1. In terms of element removal, stacks use the LIFO (Last-In-First-Out) principle, and with the `pop()` function, it will remove an element from the top of the stacks or the last element in the list. Queues use the FIFO (First-In-First-Out) principle and `dequeue()` to remove the first element in the list.
2. If we try to dequeue from an empty queue, it will normally cause an error since there's no element to remove. However, this situation can be handled properly using the `is_empty()` function. With this function, the program first checks if the queue is empty.
3. If we modify the enqueue operation to add elements at the beginning of the queue instead of the end, the queue will no longer follow the FIFO principle. But it will be possible if we use something like `queue.insert(0, item)` to insert new elements at the front.
4. When using a linked list, a major advantage is that the queue can grow or shrink dynamically without worrying about a fixed size, since allocated memory is needed, but it requires extra memory for storing pointers and is a bit more complex to implement and manage compared to arrays. In arrays, the implementation of queues is simpler and easier to understand, and accessing elements by index is straightforward. However, when using the `pop()` function

for dequeuing, all remaining elements need to be shifted one position forward, and since arrays also have a fixed size, we might run out of space if the queue grows too large unless we use dynamic resizing.

5. In real-world applications, queues are preferred over stacks where tasks or data need to be processed in the same order they arrive, following the FIFO principle. An example of this is printer job management, where print requests are queued so that the first document sent to the computer is the first one printed. Another example is customer service systems, ensuring that the earliest customers are assisted first. Overall, queues are ideal for any scenario that requires orderly and sequential processing of tasks or data.

IV. Conclusion

In conclusion, stacks and queues are both essential data structures that serve different purposes depending on how data needs to be processed. The operations in stacks are based on the LIFO (Last-In-First-Out) principle thus making them appropriate for situations that require reversing or backtracking, on the other hand, queues are based on the FIFO (First-In-First-Out) principle which makes them the correct choice for handling operations in the order that they come. The knowledge of each structure's way of managing the addition and removal of elements can be very helpful in selecting the most suitable one for a certain application. Queues are indispensable in the implementation of the systems in the real world such as task scheduling, printer management, and customer service operations that are the source of fairness and efficiency in the processing of tasks. In the end, the decision of whether to use stacks or queues is determined by whether one wants to access the most recent or the earliest element first.

References

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