Queue Data Structure in Python

Introduction

A queue is a linear data structure that follows the First In First Out (FIFO) principle. In a queue, the first element added is the first one to be removed. This is analogous to a real-world queue, like a line of people waiting for their turn.

Queue Operations

1. **Enqueue:** Adding an element to the end of the queue.
2. **Dequeue:** Removing an element from the front of the queue.
3. **Peek/Front:** Retrieving the front element without removing it.
4. **IsEmpty:** Checking if the queue is empty.
5. **Size:** Getting the number of elements in the queue.

Implementing a Queue in Python

Using a List

We can implement a simple queue using a Python list. Here's how:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            return "Queue is empty"
```

```
    def peek(self):
        if not self.is_empty():
            return self.queue[0]
        else:
            return "Queue is empty"

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

    def size(self):
        return len(self.queue)
```

Example Usage

```
# Create a queue
q = Queue()

# Enqueue items
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

# Dequeue items
print(q.dequeue())  # Output: 10
print(q.peek())     # Output: 20
print(q.size())     # Output: 2
print(q.is_empty()) # Output: False

# Display remaining items
print(q.dequeue())  # Output: 20
print(q.dequeue())  # Output: 30
print(q.dequeue())  # Output: Queue is empty
```

Implementing a Queue Using Collections.deque

The `collections.deque` is a more efficient way to implement a queue because it is optimized for O(1) time complexity for append and pop operations from both ends.

```
from collections import deque

class Queue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()
        else:
            return "Queue is empty"

    def peek(self):
        if not self.is_empty():
            return self.queue[0]
        else:
            return "Queue is empty"

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

    def size(self):
        return len(self.queue)
```

Example Usage

```
# Create a queue
q = Queue()
```

```
# Enqueue items
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

# Dequeue items
print(q.dequeue())  # Output: 10
print(q.peek())     # Output: 20
print(q.size())     # Output: 2
print(q.is_empty()) # Output: False

# Display remaining items
print(q.dequeue())  # Output: 20
print(q.dequeue())  # Output: 30
print(q.dequeue())  # Output: Queue is empty
```

Implementing a Priority Queue

A priority queue is a type of queue where each element is associated with a priority and the element with the highest priority is served before the others.

We can implement a priority queue using Python's `heapq` module which provides an implementation of the heap queue algorithm.

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.queue = []
        self.index = 0

    def enqueue(self, item, priority):
        heapq.heappush(self.queue, (-priority, self.index, it
em))
        self.index += 1
```

```
    def dequeue(self):
        if not self.is_empty():
            return heapq.heappop(self.queue)[-1]
        else:
            return "Queue is empty"

    def peek(self):
        if not self.is_empty():
            return self.queue[0][-1]
        else:
            return "Queue is empty"

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

    def size(self):
        return len(self.queue)
```

Example Usage

```
# Create a priority queue
pq = PriorityQueue()

# Enqueue items with priorities
pq.enqueue("Task1", 1)
pq.enqueue("Task2", 3)
pq.enqueue("Task3", 2)

# Dequeue items
print(pq.dequeue())  # Output: Task2
print(pq.peek())     # Output: Task3
print(pq.size())     # Output: 2
print(pq.is_empty()) # Output: False

# Display remaining items
```

```
print(pq.dequeue())  # Output: Task3
print(pq.dequeue())  # Output: Task1
print(pq.dequeue())  # Output: Queue is empty
```

Summary

In this tutorial, we covered the basics of queue data structures and their implementations in Python using lists, `collections.deque`, and the `heapq` module for priority queues. Queues are fundamental data structures with numerous applications in real-world scenarios, such as task scheduling, managing resources, and breadth-first search algorithms.

By understanding and implementing these basic queue operations, you will have a strong foundation to tackle more complex problems that require queue-based solutions.

Homework Problem: Bank Queue Simulation with Priority

You are tasked with simulating a priority queue at a bank. The bank serves customers based on their priority and arrival order. Each customer has a unique ID, a specific amount of time they need to be served, and a priority level.

Task:

1. Implement a priority queue data structure to handle the customers.
2. Write functions to:
    - Add a new customer to the queue.
    - Serve the next customer in the queue.
    - Peek at the next customer.
    - Display the current queue status.
    - Calculate the total waiting time for all customers in the queue.

Requirements:

1. Create a class `Customer` with attributes `id` (string), `service_time` (integer), and `priority` (integer).
2. Create a class `BankQueue` with methods:

- `enqueue(customer)`: Adds a new customer to the queue based on their priority.
- `dequeue()`: Removes and serves the next customer in the queue.
- `peek()`: Displays the next customer without removing them.
- `display()`: Displays the current queue with each customer's ID, service time, and priority.
- `total_waiting_time()`: Calculates and returns the total waiting time for all customers in the queue.

Further explore by adding more features such as:

- A method to change a customer's priority.
- A method to find and display a specific customer's details.
- An option to handle tie-breaking based on arrival time for customers with the same priority.