10.7 Queues

Picture a lineup at a fast food restaurant. The first person in line is the first one served, then the next person in line, and so forth. As new people join the line, they join at the back, so that everyone who joined before them are served before them. This is the exact opposite of a stack: in this lineup situation, people leave the line in the *same* order they joined it.

In this section, we'll introduce a new abstract data type to represent this type of collection, see how to implement it in Python, and analyze the running time of our implementation's operations.

The Queue ADT

A **queue** is another collection of data that, like a stack, adds and removes items in a fixed order. Unlike a stack, items come out of a queue in the order in which they entered. We call this behaviour *First-In-First-Out (FIFO)*.

- **Queue**
  - Data: a collection of items
  - Operations: determine whether the queue is empty, add an item (*enqueue*), remove the least recently-added item (*dequeue*)

In code:

```
from typing import Any


class Queue:
    """A first-in-first-out (FIFO) queue of items.

    Stores data in a first-in, first-out order. When removing an item from the
    queue, the most recently-added item is the one that is removed.

    >>> q = Queue()
    >>> q.is_empty()
    True
    >>> q.enqueue('hello')
    >>> q.is_empty()
    False
    >>> q.enqueue('goodbye')
    >>> q.dequeue()
    'hello'
    >>> q.dequeue()
    'goodbye'
    >>> q.is_empty()
    True
    """

    def __init__(self) -> None:
        """Initialize a new empty queue."""

    def is_empty(self) -> bool:
        """Return whether this queue contains no items.
        """

    def enqueue(self, item: Any) -> None:
        """Add <item> to the back of this queue.
        """

    def dequeue(self) -> Any:
        """Remove and return the item at the front of this queue.

        Raise an EmptyQueueError if this queue is empty.
        """


class EmptyQueueError(Exception):
    """Exception raised when calling dequeue on an empty queue."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'dequeue may not be called on an empty queue'
```

Much like a stack, we can also picture implementing this with a Python list. And, once again, we need to decide which end of the list is considered the front. Unlike the stack, we will see that there is a trade-off in choosing which end of the list is considered a front. Before reading the rest of the section, try to informally reason with yourself why this might be, taking into account that a queue is a FIFO.

List-based implementation of the Queue ADT

In the following implementation, we use a Python list that is hidden from the client. We have decided that the beginning of the list (i.e., index 0) is the front of the queue. This means that new items that are enqueued will be added at the end of the list, and items that are dequeued are removed from the beginning of the list.

```
class Queue:
    """A first-in-first-out (FIFO) queue of items.

    Stores data in a first-in, first-out order. When removing an item from the
    queue, the most recently-added item is the one that is removed.

    >>> q = Queue()
    >>> q.is_empty()
    True
    >>> q.enqueue('hello')
    >>> q.is_empty()
    False
    >>> q.enqueue('goodbye')
    >>> q.dequeue()
    'hello'
    >>> q.dequeue()
    'goodbye'
    >>> q.is_empty()
    True
    """
    # Private Instance Attributes:
    #   - _items: The items stored in this queue. The front of the list represents
    #             the front of the queue.
    _items: list

    def __init__(self) -> None:
        """Initialize a new empty queue."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this queue contains no items.
        """
        return self._items == []

    def enqueue(self, item: Any) -> None:
        """Add <item> to the back of this queue.
        """
        self._items.append(item)

    def dequeue(self) -> Any:
        """Remove and return the item at the front of this queue.

        Raise an EmptyQueueError if this queue is empty.
        """
        if self.is_empty():
            raise EmptyQueueError
        else:
            return self._items.pop(0)
```

Implementation efficiency

Our `Queue.enqueue` implementation calls `list.append`, which we know takes constant ($\Theta(1)$) time. However, the `Queue.dequeue` calls `self._items.pop(0)`, which takes $\Theta(n)$ time (where n is the number of items stored in the queue). If we changed things around so that the front of the queue is the end of the list (rather than the beginning), we simply swap these running times. This presents a trade-off; using an array-based list, we can *either* have an efficient enqueue or an efficient dequeue operation.

Is there, perhaps, another data structure we can use instead of a list to improve efficiency? Unfortunately, both `dict` and `set` are unordered data structures, but queues need to maintain (and remember) a very specific order. One interesting programming challenge is to implement a queue using two stacks, which can be done correctly but is not always more efficient. Eventually you will learn about even more interesting data structures, and it may be a good idea to revisit the Queue ADT and see how to use your new arsenal of data structures instead of a Python list. And because of abstraction (i.e., `_items` is a private attribute), you can modify your `Queue` implementation however you like without having to change any client code that uses it!