CSC110 Lecture 27: Queues and Priority Queues

Hisbaan Noorani

November 24, 2020

Contents

1 Exercise 1: Queue implementation and running time analysis

Consider the implementation of the Queue ADT from lecture:

```
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.

        Preconditions:
            - not self.is_empty()
        """
        return self._items.pop(0)
```

1. Complete the following table of running times for the operations of our Queue implementation. Your running times should be Theta expressions in terms of n, the number of items stored in the queue. Briefly justify each running time in the space below the table; no formal analysis necessary.

   (*Note*: equality comparison to an empty list is a constant-time operation.)

Method	Θ Runtime
`__init__`	$RT \in \Theta(1)$
`is_empty`	$RT \in \Theta(1)$
`enqueue`	$RT \in \Theta(1)$
`dequeue`	$RT \in \Theta(n)$

2. You should notice that at least one of these operations takes $\Theta(n)$ time—not great! Could we fix this by changing our implementation to use the *back* of the Python list to store the front of the queue?

```
class QueueReversed:
    """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.insert(0, item)

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

        Preconditions:
            - not self.is_empty()
        """
        return self._items.pop()
```

No, if we switched it to treat the back of the list as the front of the queue, then while the `dequeue` function would be $\Theta(1)$, the `enqueue` function would be $\Theta(n)$. Either way, one of the functions has to be $Theta(n)$.

2 Exercise 2: Priority Queues

1. Complete the following implementation of the Priority Queue ADT, which uses a private attribute that is an *unsorted list of tuples* (pairs of (priority, value)) to store the elements in the collection.

```
class PriorityQueueUnsorted:
    """A queue of items that can be dequeued in priority order.

    When removing an item from the queue, the highest-priority item is the one
    that is removed.

    >>> pq = PriorityQueueUnsorted()
    >>> pq.is_empty()
    True
    >>> pq.enqueue(1, 'hello')
    >>> pq.is_empty()
    False
    >>> pq.enqueue(5, 'goodbye')
    >>> pq.enqueue(2, 'hi')
    >>> pq.dequeue()
    'goodbye'
    """
    # Private Instance Attributes:
    #   - _items: A list of the items in this priority queue.
    #             Each element is a 2-element tuple where the first element is
    #             the priority and the second is the item.

    _items: List[Tuple[int, Any]]

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


    def is_empty(self) -> bool:
        """Return whether this priority queue contains no items.
        """
        return not self._items

    def enqueue(self, priority: int, item: Any) -> None:
        """Add the given item with the given priority to this priority queue.
        """
        self._items.append((priority, item))

    def dequeue(self) -> Any:
        """Return the element of this priority queue with the highest priority.

        Preconditions:
            - not self.is_empty()
        """
        greatest_priority = 0
        pop_index = 0
        for i in range len(self._items):
            if self._items[i][0] > greatest_priority:
                greatest_priority = self._items[i][0]
                pop_index = i

        return self._item.pop(pop_index)[1]
```

2. Complete the following table of running times for the operations of your `PriorityQueueUnsorted` implementation. Your running times should be Theta expressions in terms of n, the number of items stored in the priority queue. Briefly justify each running time in the space below the table; no formal analysis necessary.

Method	Θ Runtime
`__init__`	$RT \in \Theta(1)$
`is_empty`	$RT \in \Theta(1)$
`enqueue`	$RT \in \Theta(1)$
`dequeue`	$RT \in \Theta(n)$

3 Additional exercises

1. Implement a new version of the Priority Queue ADT called `PriorityQueueSorted`, which also stores a list of `(priority, item)` pairs, except it keeps the list sorted by priority.

   Your implementation should have a running time of $\Theta(1)$ for `dequeue` and a *worst-case* running time of $\Theta(n)$ for `enqueue` (but possibly with a faster running time depending on the item and priority being inserted).