Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [40]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 7

Copyright Luca de Alfaro, 2019-21. License: CC-BY-NC.

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form.
- This homework is due on **Thursday, 21 April 2022** by 11:59pm.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

`# YOUR CODE HERE`

and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  `assert x == 2`

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.
- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

About this homework

Two things are new about this homework assignment.

First, this homework is graded using `assert`, which is the standard way in Python of asking for something to be True. So, a cell succeeds if no error is raised; you no longer see "Success" printed out all the time.

Second, the hidden tests will make sure that your implementation works by comparing your implementation with a reference one. Precisely, we will be comparing the implementation of CountingQueue with that of Queue.

Stacks, Queues, and Their Counting Versions

A stack is a data structure with two operations: push, and pop. Picture it as a pile of dishes sitting on a counter. A push operation places a dish on top of the pile. A pop operation returns the dish on top of the pile, or None if the pile is empty, that is, contains no dishes. A "dish" can be any Python object.

A queue is a data structure with two operations: put, and get. Imagine it as a stack of books horizontally on a shelf. A put operation adds the book to the left end of the books on the shelf; a get operation gets the book from the right end of the shelf.

Thus, the difference between a stack and a queue is that the stack is FILO (First In, Last Out), whereas the queue is FIFO (First In, First Out). Elements in a stack are retrieved newest first. Elements in a queue are retrieved in the order they were put in, oldest first.

We will implement here these data structures, with a small twist: we will also introduce *counting* versions of them, which avoid keeping multiple identical copies of objects in a row.

Let us begin by implementing a plain vanilla stack.

In [41]:
```
class Stack(object):

    def __init__(self):
        self.stack = []

    def __repr__(self):
        """Defining a __repr__ function will enable us to print the
        stack contents, and facilitate debugging."""
```

```
        return repr(self.stack) # Good enough.

    def push(self, x):
        """The "top" of the stack is the end of the list."""
        self.stack.append(x)

    def pop(self):
        return self.stack.pop() if len(self.stack) > 0 else None

    def isempty(self):
        return len(self.stack) == 0

    def __len__(self):
        return len(self.stack)

    def __iter__(self):
        for el in self.stack:
            yield el

    def __getitem__(self, i):
        return self.stack[i]

    def __contains__(self, x):
        return x in self.stack
```

Let's see how this works.

In [42]:
```
s = Stack()
print(s.pop())
s.push('a')
s.push('b')
print(s.pop())
print(s.pop())
print(s.pop())
```

```
None
b
a
None
```

Ok! The definition of a queue is similar.

In [43]:
```
class Queue(object):

    def __init__(self):
        self.queue = []

    def __repr__(self):
        """Defining a __repr__ function will enable us to print the
        queue contents, and facilitate debugging."""
        return repr(self.queue) # Good enough.

    def add(self, x):
        self.queue.append(x)
```

```
    def get(self):
        # This is the only difference compared to the stack above.
        return self.queue.pop(0) if len(self.queue) > 0 else None

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

    def __len__(self):
        return len(self.queue)

    def __iter__(self):
        for el in self.queue:
            yield el

    def __getitem__(self, i):
        return self.queue[i]

    def __contains__(self, x):
        return x in self.queue
```

Let's see how it works.

In [44]:
```
s = Queue()
print(s.get())
s.add('a')
s.add('b')
print(s.get())
print(s.get())
print(s.get())
```

```
None
a
b
None
```

As you see, in a queue, the elements are retrieved in the same order in which they were added.

Python experts might note that, for a queue, we would do better by using the `collections.deque` class, rather than the list class, to make the `pop(0)` operation more efficient; in lists, it takes time proportional to the length of the list; in deques, it takes constant time. For small lists, however, the difference is negligible.

We now consider a use case in which we may need to put in the queue or stack many repeated copies of the same object. For instance, assume that the queue is used to store events, and assume that some event may end up being repeated many times in a row. As an example, the events can be "s", for the tick of a second, "m", when the minute advances, and "h", when the hour advances. There will be 60 consecutive "s" events between any two "m" events, and it seems a waste to store so many consecutive identical events. Storing many identical things in a row is akin to counting in unary notation, after all. We would be better

off storing the repeated elements only once, along with a count of the number of times they occur. Let's develop a queue using this idea (a stack can be done similarly).

In [45]:
```
class CountingQueue(object):

    def __init__(self):
        self.queue = []

    def __repr__(self):
        return repr(self.queue)

    def add(self, x, count=1):
        # If the element is the same as the last element, we simply
        # increment the count.  This assumes we can test equality of
        # elements.
        if len(self.queue) > 0:
            xx, cc = self.queue[-1]
            if xx == x:
                self.queue[-1] = (xx, cc + count)
            else:
                self.queue.append((x, count))
        else:
            self.queue = [(x, count)]

    def get(self):
        if len(self.queue) == 0:
            return None
        x, c = self.queue[0]
        if c == 1:
            self.queue.pop(0)
            return x
        else:
            self.queue[0] = (x, c - 1)
            return x

    def isempty(self):
        # Since the count of an element is never 0, we can just check
        # whether the queue is empty.
        return len(self.queue) == 0
```

Let's put this to the same test as before, printing the queue contents at each step to see what is going on.

In [46]:
```
q = CountingQueue()
q.add('a')
print(q)
q.add('b', count=5)
print(q)
q.add('c', count=2)
print(q)
while not q.isempty():
    print(q.get())
    print(q)
```

```
[('a', 1)]
[('a', 1), ('b', 5)]
[('a', 1), ('b', 5), ('c', 2)]
a
[('b', 5), ('c', 2)]
b
[('b', 4), ('c', 2)]
b
[('b', 3), ('c', 2)]
b
[('b', 2), ('c', 2)]
b
[('b', 1), ('c', 2)]
b
[('c', 2)]
c
[('c', 1)]
c
[]
```

It works! And notice that it works even if we add elements one by one.

In [47]:
```python
q = CountingQueue()
for i in range(10):
    q.add('a')
q.add('b')
for i in range(3):
    q.add('c', count=2)
print(q)
```

```
[('a', 10), ('b', 1), ('c', 6)]
```

The Homework Assignment

For this homework, you must implement the following methods for `CountingQueue`:

- `__len__`
- `__iter__`
- `__in__`
- `__getitem__`

Your goal is to have `CountingQueue` behave exactly like `Queue` to an outside user: the objects have to be different only due to their internal implementation. So for instance, `__len__` must return the number of elements, including repetitions; not the number of (element, count) pairs in `self.queue`.

Note that we are adding methods to a class that has already been defined, so our definition have the following somewhat unusual form:

In [48]:
```python
def counting_queue_peek(self):
    if len(self.queue) == 0:
        return None
```

```
    el, _ = self.queue[0]
    return el

CountingQueue.peek = counting_queue_peek
```

In other words, we first create a function (in this case `counting_queue_peek`) and then we assign it to the method `peek` of `CountingQueue`. It's a bit unusual, but it works, and it relieves us from the task of redefining the class each time we need a new method.

In [49]:
```
q = CountingQueue()
q.add("cat")
q.add("dog")
q.peek()
```

Out[49]:
```
'cat'
```

`__len__`

In [50]:
```
def counting_queue_len(self):
    # YOUR CODE HERE
    count = 0
    if len(self.queue) == 0:
        return 0
    for i in range(len(self.queue)):
        item, amount = self.queue[i]
        if amount >= 0:
            count += amount
    return count


CountingQueue.__len__ = counting_queue_len
```

In [51]:
```
# 5 points.  Simple tests

q = CountingQueue()
assert len(q) == 0
q.add("cat")
q.add("dog")
assert len(q) == 2
```

In [52]:
```
# 5 points.  More complicated tests.

q = CountingQueue()
assert len(q) == 0
q.add("cat")
q.add("cat")
assert len(q.queue) == 1
assert len(q) == 2
q.add("dog")
assert len(q) == 3
assert len(q.queue) == 2
q.add("dog")
assert len(q) == 4
```

```
assert len(q) == 4 # Hey, just in case you went for the quantum-mechanical solution
assert len(q) == 4
assert len(q.queue) == 2
```

In [53]:
```
# 5 points.  Works same as Queue.

import random

for k in range(100):
    q0 = Queue()
    q1 = CountingQueue()
    for _ in range(100):
        el = random.choice(["a", "b", "c"])
        q0.add(el)
        q1.add(el)
        assert len(q0) == len(q1)
        assert len(q0.queue) >= len(q1.queue)
```

__iter__

In [54]:
```
def counting_queue_iter(self):
    # YOUR CODE HERE
    for i in range(len(self.queue)):
        item, amount = self.queue[i]
        count = amount
        for j in range(count):
            yield item



CountingQueue.__iter__ = counting_queue_iter
```

In [55]:
```
# 5 points.  Simple tests.

q = CountingQueue()
q.add("cat", count=2)
q.add("dog", count=3)
assert [x for x in q] == ["cat"] * 2 + ["dog"] * 3
```

In [56]:
```
# 5 points.  Works the same as queue.

for k in range(100):
    q0 = Queue()
    q1 = CountingQueue()
    for _ in range(100):
        el = random.choice(["a", "b", "c"])
        q0.add(el)
        q1.add(el)
        assert [x for x in q0] == [x for x in q1]
```

__in__

In [57]:
```
def counting_queue_in(self, el):
    # YOUR CODE HERE
    return el in self.queue

CountingQueue.__in__ = counting_queue_in
```

In [58]:
```
# 0 points.  Simple cases.

q = CountingQueue()
assert "cat" not in q
q.add("cat", count=2)
assert "cat" in q
assert "dog" not in q
q.add("dog")
assert "cat" in q
assert "dog" in q
q.get()
assert "cat" in q
assert "dog" in q
q.get()
assert "cat" not in q
assert "dog" in q
q.get()
assert "cat" not in q
assert "dog" not in q
```

In [59]:
```
# 0 points.  Behaves the same as Queue.

elements = range(5)
for k in range(100):
    q0 = Queue()
    q1 = CountingQueue()
    for _ in range(20):
        el = random.choice(elements)
        q0.add(el)
        q1.add(el)
        for x in elements:
            assert (x in q0) == (x in q1)
```

__getitem__

In [60]:
```
def counting_queue_getitem(self, n):
    # YOUR CODE HERE
    count = 0
    for i in range(len(self.queue)):
        item, amount = self.queue[i]
        for j in range(amount):
            if n == count:
                return item
            count += 1
    return self.queue[n]

CountingQueue.__getitem__ = counting_queue_getitem
```

In [61]:
```
# 5 points: simple tests.

q = CountingQueue()
q.add("cat", count=2)
q.add("dog", count=3)
q.add("bird", count=4)
els = [q[i] for i in range(9)]
assert els == ['cat'] * 2 + ['dog'] * 3 + ['bird'] * 4
# Let's do it again.
els = [q[i] for i in range(9)]
assert els == ['cat'] * 2 + ['dog'] * 3 + ['bird'] * 4
```

In [62]:
```
# 10 points: you raise IndexError when accessing elements out of bounds.

q = CountingQueue()
q.add("cat", count=2)
q.add("dog", count=3)

# Raise IndexError when it's too high...
try:
    q[5]
    assert False, "Failed to raise IndexError"
except IndexError:
    pass
assert q[4] == "dog"
try:
    q[5]
    assert False, "Failed to raise IndexError"
except IndexError:
    pass

# And also when it's too low.
try:
    q[-10]
    assert False, "Failed to raise IndexError"
except IndexError:
    pass

# or too large:
try:
    q[10]
    assert False, "Failed to raise IndexError"
```

```
except IndexError:
    pass

# And raise TypeError if you try to index with a non-integer.
try:
    q["hello"]
    assert False, "Failed to raise TypeError"
except TypeError:
    pass
```

In [63]:
```
# 5 points.  Behaves the same as Queue.

elements = range(3)
for k in range(100):
    q0 = Queue()
    q1 = CountingQueue()
    for m in range(40):
        el = random.choice(elements)
        q0.add(el)
        q1.add(el)
        for i in range(m):
            assert q0[i] == q1[i]
```