Carleton University
Department of Systems and Computer Engineering
SYSC 2100 - Algorithms and Data Structures - Winter 2021

Assignment 1 - Implementing a Deque Using a Circular Linked List

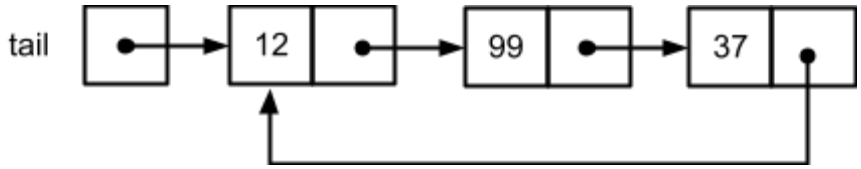
This assignment applies many of the key concepts from the first half of term (specification of abstract data types, using classes to implement ADTs, linked lists). Each method you'll write requires only a small number of lines of code, but it's important to visualize the linked list operations before you start coding; otherwise, it's easy to end up getting lost in a tangled web of nodes and pointers.

Background - Circular Linked Lists

Recall that each node in a singly linked list contains a link (pointer, reference) to the next node in the list. In the last node (tail node), an end-of-list value is stored in the link. In Python, this value is `None`, in C, this value is the `NULL` pointer.

One variation of the linked list is known as the *circular linked list*. The end-of-list value isn't used; instead, the link in the tail node points to the first node in the list (the head node). We also need a variable that stores the link to the tail node. There's no need for a variable that stores a link to the head node, because the tail node points to the head node.

We can visualize a circular singly-linked list this way:

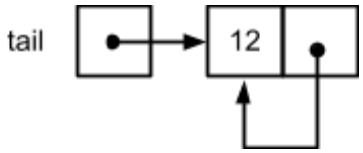


Assuming that the fields in the nodes are named `data` and `next`:

- `tail.data` is the value stored in the last node (tail node); that is, 12.
- `tail.next` is the link to the first node (head node).
- `tail.next.data` is the value stored in the first node; that is, 99.

Notice that accessing the last node and the first node are constant time operations. Accessing any other node requires us to traverse the nodes, starting with the tail node.

If a circular linked list has one node, that node points to itself:



If the circular linked list is empty, `tail` contains `None`.

ADT Deque - Design

A *deque* (pronounced "deck") is a generalization of a stack and a queue. The word is short for "double-ended queue". At a minimum, a deque provides operations to insert and retrieve items at both ends of the deque. (Some implementations provide operations to search a deque for a specific item, or insert, retrieve or remove items from anywhere in a deque. We won't consider these operations in this assignment.)

Earlier, we looked at an implementation of ADT Deque that used Python's built-in `list` type as the underlying data structure. In this lab, you'll implement a deque using a circular, singly-linked list. The tail node will represent the rear of the deque, while the head node will represent the front of the deque.

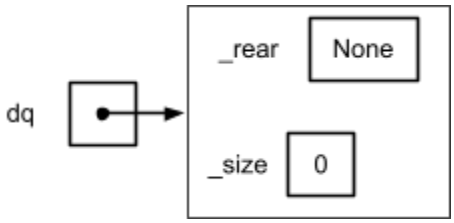
The linked list will be encapsulated in a class named `LinkedDeque`, which will have two instance variables:

- `_rear` refers to the node at the rear of the deque; that is, the last node in the linked list.
- `_size` keeps track of the number of items in the deque; that is, the number of nodes in the linked list.

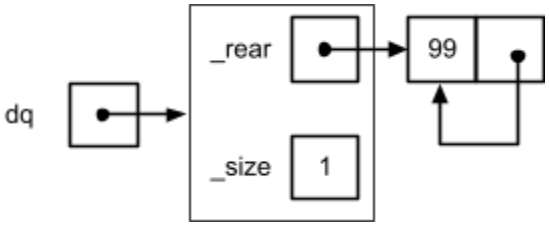
The *add-rear* operation adds an item at the rear of the deque. Suppose we create a new `LinkedDeque` object, then add four integers to the rear:

```
dq = new LinkedDeque()
dq.add_rear(99)
dq.add_rear(37)
dq.add_rear(12)
dq.add_rear(42)
```

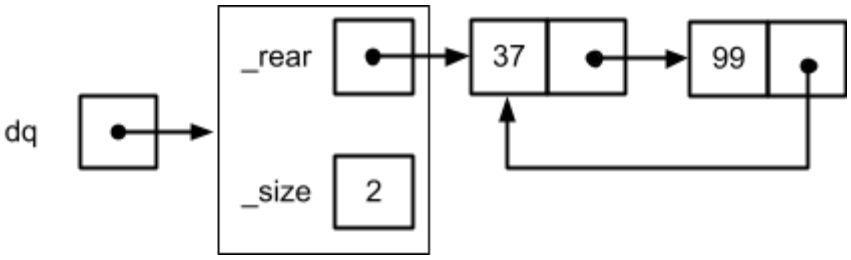
We can visualize the deque being built, starting with the new, empty `LinkedDeque` object:



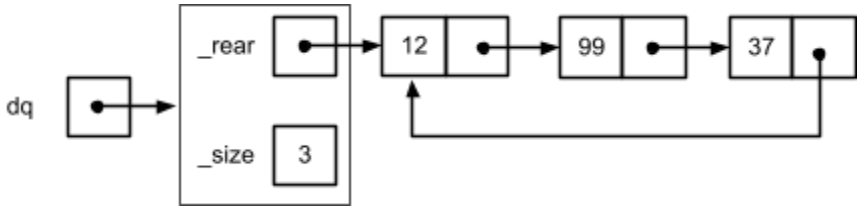
- After 99 is added to the rear of the deque:



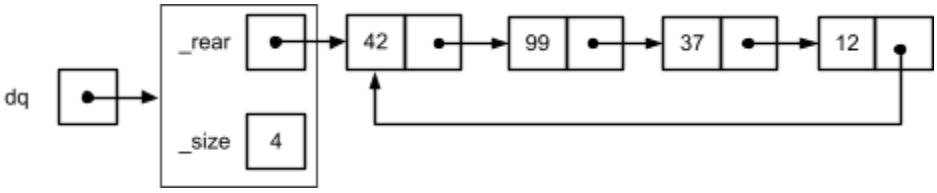
- After 37 is added to the rear of the deque:



- After 12 is added to the rear of the deque:



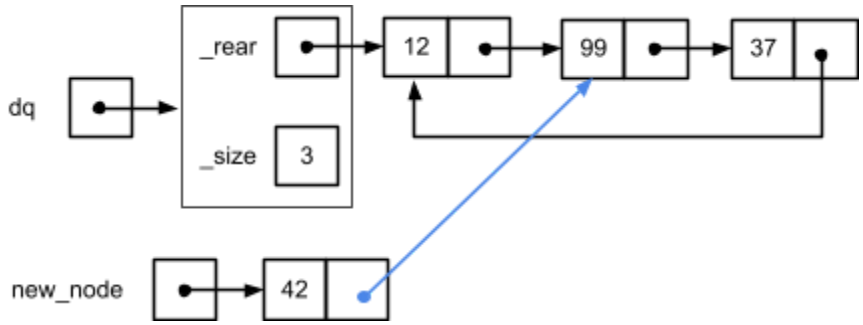
- After 42 is added to the rear of the deque:



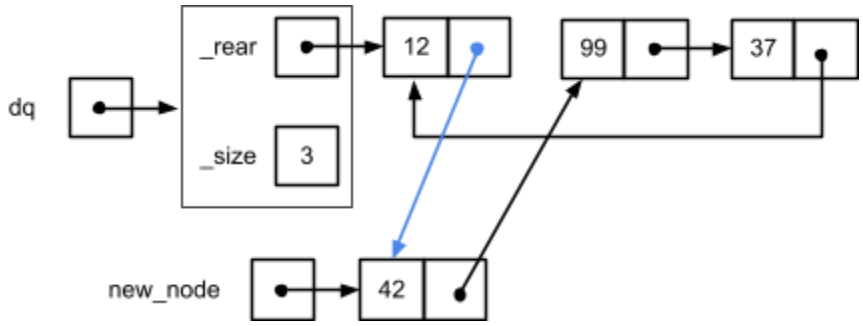
It might appear that adding a node requires a lot of pointer shuffling, but it doesn't. Adding a new node at the rear of a non-empty deque requires exactly three links to be updated (three assignment statements). For example, here are the three steps that add a node containing 42 to

the deque containing three items:

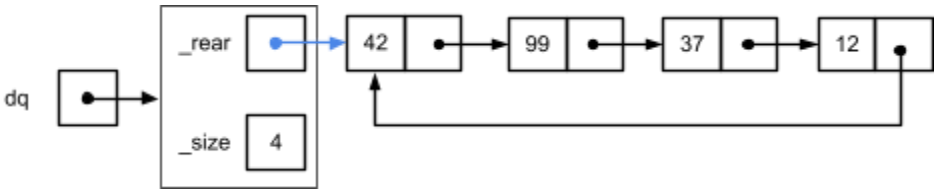
1. the `next` link in the new node is updated to point to the node at the front of the deque (i.e., the node containing 99);



2. the `next` link in the node at the rear of the deque is updated to point to the new node;



3. `_rear` is updated to point to the new node.

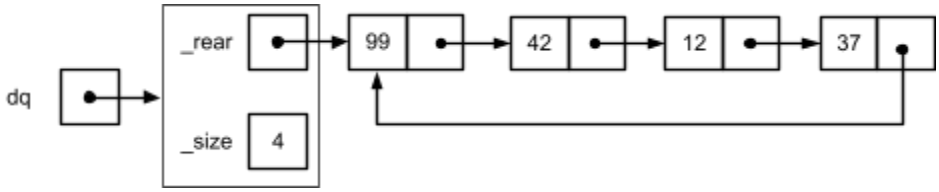


So, *add-rear* is an $O(1)$ operation.

The *add-front* operation adds an item to the front of the deque. This operation always requires two links to be updated (two assignment statements), so it's a $O(1)$ operation.

Here's a picture of a deque after four integers are added to the front a new `LinkedDeque`:

```
dq = new LinkedDeque()
dq.add_front(99)
dq.add_front(37)
dq.add_front(12)
dq.add_front(42)
```



(Drawing the pictures that depict the deque after the first, second and third values were added to the front of the initially empty deque is left as an exercise for you. What links are updated as a new node is added? What are the two assignment statements that do this?)

The *remove-front* operation removes (and returns) the item at the front of the deque. It's also a $O(1)$ operation.

The *remove-rear* operation removes (and returns) the item at the rear of the deque. This is an $O(n)$ operation.

Design Exercise

- draw diagrams that depict the deque after 99, 37, 12 and 42 are added to the front of an initially empty deque(one diagram for each value). Your last diagram should look like the one immediately above.
- draw diagrams that depict the deque after the *remove-front* operation is repeatedly performed on a deque that initially contains 4 values, until the deque is empty. Use the diagram to determine how many link updates are required when the deque is empty and when the deque has at least one item. Make sure you understand why this operation is $O(1)$.
- draw diagrams that depict the deque after the *remove-rear* operation is repeatedly performed on a deque that initially contains 4 values, until the deque is empty. Use the diagram to determine how many link updates are required when the deque is empty and when the deque has at least one item. Make sure you understand why this operation is $O(n)$.

ADT Deque - Implementation

Download linked_deque.py from the *Lab Materials* section of the main cuLearn course page.

Class `LinkedDeque` uses a circular linked list of `Node` objects as the underlying data structure.

This class has a complete implementation of `__init__`. You can create a new, empty deque this way:

```
dq = LinkedDeque()
```

This method takes an optional argument, which must be an *iterable*. (An iterable is an object that is capable of returning its members one at a time. Instances of Python's `list` and `tuple` types

and `range` objects, are three examples of iterables.) The contents of the iterable are added one-by-one to the **rear** of the `LinkedDeque` object, so creating a `LinkedDeque` this way:

```
dq = LinkedDeque([99, 37, 12, 42])
```

is equivalent to this code:

```
dq = new LinkedDeque()
dq.add_rear(99)
dq.add_rear(37)
dq.add_rear(12)
dq.add_rear(42)
```

Note: `__init__` calls `add_rear`, which you have to write, so you won't be able to create a deque from an iterable until that task is finished.

The class also has a complete implementation of `__str__`. Here's an example of the strings produced by this method:

```
>>> dq = new LinkedDeque()
>>> dq.add_rear(99)
>>> dq.add_rear(37)
>>> dq.add_rear(12)
>>> dq.add_rear(42)
>>> str(dq)
'99 -> 37 -> 12 -> 42'
```

Important: the leftmost item in the string is the value at the **front** of the deque, and the rightmost item in the string is the value at the **rear** of the deque. You'll have to remember that instance variable `_rear` refers to the node containing the rightmost item. Also, the link from the rear node to the front node isn't shown in the string representation.

"Stub" implementations have been provided for methods `__repr__`, __len__, `add_rear`, `add_front`, `remove_front`, `remove_rear`, `peek_front` and `peek_rear`. If you call any of these methods on a `LinkedDeque` object, Python will throw a `NotImplementedError` exception.

For each method, read the docstring and replace the `raise` statement with a correct implementation of the method. If you've prepared a complete set of design diagrams for the add and remove operations, this shouldn't take much time.

I recommend that you use an incremental, iterative approach: as you write each method, test it before you move on to the next one. Use the shell or write a short script to test the methods. If you write a script, please put it in a separate module (file). We don't want test scripts in the linked_deque.py file you submit to cuLearn.

I suggest you implement the methods in this order.

- `add_rear`. This method should be *O*(1). After testing `add_rear`, verify that you can create a deque by passing `__init__` an iterable.
- `__len__`. This method should be *O*(1).
- `__repr__`. Notice that `__repr__` will return an expression that would create an instance of `LinkedDeque` that is identical to the object on which `__repr__` is called. Hint: feel free to "borrow" the code from `__str__`, but note that there will be some important differences between the two methods.
- `remove_front`. This method should be *O*(1). At this point, you should be able to use a deque as a FIFO queue (`add_rear` enqueues an item, `remove_front` dequeues an item). Test this.
- `peek_front` and `peek_rear`. These methods should be *O*(1).
- `add_front`. This method should be *O*(1). At this point, you should be able to use a deque as a LIFO stack (`add_front` pushes an item, `remove_front` pops an item). Test this.
- `remove_rear`. This method should be *O*(*n*). Refer to your design diagrams, and make sure you understand why it can't be *O*(1).

Programming Style - Code Formatting

Before submitting your code, please run it through Wing 101's source reformatter, as described in following paragraphs.

Wing 101 can easily be reconfigured to reformat your code so that it adheres to some of the formatting conventions described in the "official" Python coding conventions document, PEP 8 - *Style Guide for Python Code*.[1]

To configure reformatting, follow the instructions in Section 3.2 of *Installing Python 3.9.1 and Wing 101 7.2.7*, which is posted on cuLearn. (The instructions apply to both the Windows 10 and macOS versions of Wing 101.)

If you prefer, you manually reformat your files even if you've disabled automatic reformatting. This is described in Section 3.3. Note that you still have to open the Auto-formatting window to configure which PEP 8 conventions are followed, as described in Section 3.2.

[1] https://www.python.org/dev/peps/pep-0008/

Manually reformatting an open file is easy:

- From the menu bar, select Source > Reformatting
- From the pop-up menu, select Reformat File for PEP 8

Wing 101 7.2.8 fixes a bug in release 7.2.7: the Reformat Selection for PEP 8 command now works.

Programming Style - Writing Readable Code

Focus on writing code that is readable and maintainable, and not just code "that works". Using a code reformatter can improve the layout of your code, but it can't enforce every coding convention. For example, a reformatter won't change your variable names if they aren't descriptive or follow the "official" Python convention (that is, variables names are lowercase, with words separated by underscores).

Here are some things to consider:[2]

"The ease by which other *people* can read and understand a program (often called "readability" in software engineering) is perhaps the most important quality of a program. Readable programs are used and extended by others, sometimes for decades. For this reason, we often say that programs are written to be read by humans, and only incidentally to be interpreted by computers.

A program is composed well if it is concise, well-named, understandable, and easy to follow.

Excellent composition does not mean adhering strictly to prescribed style conventions. There are many ways to program well, just as there are many styles of effective communication. However, the following guiding principles universally lead to better composition of programs:

- **Names**. To a computer, names are arbitrary symbols: "xegyawebpi" and "foo" are just as meaningful as "tally" and "denominator". To humans, comprehensible names aid immensely in comprehending programs. Choose names for your functions and variables that indicate their use, purpose, and meaning.

- **Functions**. Functions are our primary mechanism for abstraction, and so each function should ideally have a single job that can be used throughout a program. When given the choice between calling a function or copying and pasting its body, strive to call the function and maintain abstraction in your program.

- **Purpose**. Each line of code in a program should have a purpose. Statements should be removed if they no longer have any effect (perhaps because they were useful for a previous version of the program, but are no longer needed). Large blocks of unused code,

[2] Adapted from a style guide used by students in an introductory programming course at the University of California, Berkeley.

even when turned into comments, are confusing to readers. Feel free to keep your old implementations in a separate file for your own use, but don't turn them in as your finished product.

- **Brevity**. An idea expressed in four lines of code is often clearer than the same idea expressed in forty. You do not need to try to minimize the length of your program, but look for opportunities to reduce the size of your program substantially by reusing functions you have already defined."

Wrap Up

Please read *Important Considerations When Submitting Files to cuLearn*, on the last page of the course outline.

The submission deadline for this assignment is Wednesday, March 10, 23:55 (Ottawa time), for all lab sections.

To submit your lab work, go to the cuLearn page **for your lab section** (not the main course page). Submit linked_deque.py. Ensure you submit the version of the file that contains your solutions, and not the unmodified file you downloaded from cuLearn! You are permitted to make changes to your solutions and resubmit the file as many times as you want, up to the deadline. Only the most recent submission is saved by cuLearn.

Last edited: February 27, 2021