

CSCI 241 Data Structures

Project 3: Queue the Stacking of the Deque

In lecture, we have seen that a deque can serve both as a stack and as a queue depending on usage. We have also seen that a deque can be implemented using a linked list or an array to store its data. We will employ both of those features in this project.

Because we will use a deque as the storage medium for our queues and stacks, we will begin there. You will implement two deque classes, one using an array to store the contents and one using a linked list to store the contents. Notice that regardless of how you store the data, the six operations that define a deque must be present: two pushes, two pops, and two peeks. If something calls itself a deque, it must provide at least those six methods. The programming pattern that enforces this is called an *interface* or more generally an *abstract base class*. We provide a complete abstract base class called Deque in the file Deque.py. Notice that the file does not contain implementations; it only defines the functions that must be present. If you attempt to inherit from Deque, **the child class must contain the methods listed in the abstract base class**, or Python will refuse to construct the object and terminate. This is done via the special method `__subclasshook__`:

```
def __subclasshook__(child):
    required_methods = {'push_front', 'push_back', \
                        'peek_front', 'peek_back', \
                        'pop_front', 'pop_back'}
    if required_methods <= child.__dict__.keys():
        return True
    return False
```

Python will call this method to see if a child class is allowed to inherit from the Deque class—that is, are the six required methods a subset of the methods in the child's namespace? (In Python a set B is a subset of set A if and only if `A <= B` is True.) If all six methods are there, `__subclasshook__` will return True, indicating that child looks like a deque and instantiation can proceed. If any method is missing, `__subclasshook__` will return False, indicating that the child does not qualify as a deque and instantiation cannot proceed. This file is complete; **do not modify Deque.py**.

Linked_List_Deque and Array_Deque should both inherit from Deque. Linked_List_Deque's constructor will initialize an empty linked list for the data. Array_Deque's constructor will initialize an empty array for the data. The six required methods will have different implementations in each class because we interact with arrays and linked lists differently. Except for performance differences, the user of your deque should not be able to tell whether the implantation used a linked list or an array. Their functionality (and string representations) must be identical.

So far, we have always directly called a constructor/ `__init__` to build new objects. Here, we have two different classes that can create objects that do what we want. To support both of them, we introduce one more new design pattern: the *factory method*. The factory method is simply a function that returns various implementations of the abstract base class depending on some condition. Here, that condition is an optional parameter `deque_type`:

```
LL_DEQUE_TYPE = 0
ARR_DEQUE_TYPE = 1

def get_deque(deque_type=LL_DEQUE_TYPE):
    if deque_type == LL_DEQUE_TYPE:
        return Linked_List_Deque()
    elif deque_type == ARR_DEQUE_TYPE:
        return Array_Deque()
    raise NotImplementedError
```

If the user calls this method with a parameter value of `ARR_DEQUE_TYPE` (or 1), she will get an array-based deque. If she specifies `LL_DEQUE_TYPE` (or 0) or does not provide an argument, she will get a linked list-based deque. Once the object is returned, the user does not care about the implementation details; she only cares that she has a deque. You can imagine a larger software system where this decision is made based on the contents of a configuration/preferences file or some other setting. (If you have previously programmed in a language like C++ or Java, note that although you construct a `Linked_List_Deque` or `Array_Deque`, the factory method returns an object of type `Deque`, meaning that only methods defined in the `Deque` interface can be used on the returned object.) This is the beauty of polymorphism.

One important detail about the `Array_Deque` is that you must not limit its capacity. Initially, the array should have a capacity of just one single cell. Whenever the user pushes an item and no cells are available, first double the capacity of the contents array and copy the contents of the old array to the new one. We have seen this requirement of array-based implementations several times in lecture; now you must implement it. After resizing the array, you should have an available cell for the item being pushed. Notice that the `__grow` method skeleton in `Array_Deque` is private; there is no reason for the user to know about this logic as long as the item ultimately lands in the right spot of the deque.

Once your deque is implemented, use it as the storage medium in your stack and queue implementations by calling `get_deque` in their respective constructors. Consider carefully the performance implications of your design choices. What performance results from placing the top in different locations? Choose the best one. Do the same for the front and back of your deque.

Even though there are two deque implementations, there is only one implementation each of queue and stack. The queue and stack implementations must work correctly

regardless of which deque is used to store their contents. **Note that no exceptions are raised by any methods in Deque, Stack, or Queue.**

Now you have stack and queue implementations that work with either array-based or linked list-based deques, solve the following two problems using your stack implementation.

The Towers of Hanoi

In the ancient game Towers of Hanoi, there are three pegs on which discs of varying size can be stacked. The rules are simple:

- Each disc must be smaller than every disc underneath of it.
- Only one disc may be moved at a time.

Initially, all discs are stacked on the first peg. The goal is to move all discs to the third peg without violating either of the above rules. Iterative solutions to this problem are not intuitive (though they exist). Consider the problem recursively. If the goal is to move n discs, find a solution that involves moving $n - 1$ discs and combining that movement somehow with the single disc left behind. Take advantage of the auxiliary peg, and consider carefully which pegs should be used as the source, auxiliary, and destinations pegs at each step. Remember that parameters are just temporary local variables that get aliased to whatever is passed in on each function call.

Once you have your Stack implemented, use it to solve the Towers of Hanoi problem, for which I have provided a skeleton file. In this implementation, discs will be represented by integers that correspond to their diameters with 0 being the smallest disc. You must solve this recursively. Fill in the #TODO section in the recursive call to implement the movement from one stack to another. My solution is six lines of code including the if and else lines (so really only four lines of actual code). I have included some print lines in the recursive function so that you can see the behavior as the algorithm progresses. Observe the timings and discuss your conclusions about this algorithm's performance in your writeup. For 3 discs, my solution prints:

(page break to keep sample output together)

two structures? Make an argument that those test cases are complete; do they cover all expected possibilities when a user programs with your implementation?

2. What is the worst-case performance of every method in Stack and Queue? Include in your analysis discussions of both Array_Deque-based Stacks and Queues and Linked_List_Deque-based Stacks and Queues. What is your opinion of our design decision not to raise exceptions in any methods? Does this limit functionality in any way? Present a reasoned argument to justify your opinion. Which test cases are designed to test these two structures? Make an argument that those test cases are complete; do they cover all expected possibilities when a user programs with your implementation?

3. As you observe the timings for Hanoi, what do you notice? Is this a performance class we have seen before or a different one? How would you describe the rate at which the runtime of Hanoi is increasing? No formal analysis is necessary here; just describe what you see and what is causing it. Including Delimiter_Check, we have now seen several examples of Python programs that accept arguments/parameters when they are invoked on the command line. Describe how you could employ this, together with factory method infrastructure provided by Deque_Generator, to allow a deque, stack, or queue program to use either an array-based deque or linked list-based deque depending on how it is invoked from the command line. Finally, describe the process through which you tested Hanoi and Delimiter_Check to ensure that they function correctly.

SUBMISSION EXPECTATIONS

Linked_List.py This should be your **corrected** Linked_List implementation from project 2. If your Linked_List still fails any test cases, an 11% penalty will be applied to your project score. It is not possible to earn an A on this project if problems remain in your Linked_List implementation.

Linked_List_Deque.py, Array_Deque.py, Stack.py, and Queue.py Your **completed** implementation of the skeleton Deque.py, Stack.py, and Queue.py files attached to this project assignment. Be mindful of **performance considerations** in your method implementations. Note that solutions that perform in linear time will not receive full credit if a constant time solution is possible. **Do not modify the definitions of any functions or change file or class names.**

Hanoi.py Your completed recursive Hanoi implementation. **Do not modify the contents of this file beyond replacing #TODO** with your minimal implementation. You may also change the initial number of discs in the main section for testing purposes.

Delimiter_Check.py Your completed standalone program that takes a file name as an argument and prints exactly one of the following

messages (without quotes): "The file contains balanced delimiters." or "The file contains IMBALANCED DELIMITERS."

DSQ_Test.py Your unit tests for all of your class implementations, combined. The setup method initializes objects of all three classes, so you have access to a deque object called `self._deque`, a stack object called `self._stack`, and a queue object called `self._queue` in each test method. You should not modify anything that is provided in this file. **Simply replace #TODO with your test functions.** There should be one function for each discrete test that you design, and all test function names should begin with `test_`. Use the test case file from the linked list project as an example. It is not appropriate to use the unit testing approach for this implementation of Hanoi or delimiter checking. Use unit tests to verify the functionality of your deque, stack, and queue implementations. For Hanoi and delimiter checking, examine the printed output manually.

DequePerformanceAndTests.pdf A one-page prose document addressing writeup prompt 1.

StackQueuePerformanceAndTests.pdf A one-page prose document addressing writeup prompt 2.

AppsPerformanceAndTests.pdf A one-page prose document addressing writeup prompt 3.

Do not submit `Deque.py`, `Deque_Generator.py`, or `Linked_List_Test.py`. We will use the same versions of these files that we uploaded to Blackboard.