CS-UY 1134: Data Structures and Algorithms
Lab 6: Stacks and Queues

Friday, July 5, 2019

Instructions

This lab will focus on recursion, with a little bit of practice with sorting.

Try to solve all of the problems by the end of the lab time. As added practice, it is a good idea to try writing your code with pen and paper first, as you will have to do this on the exam.

When you type up your code, write some test cases. Try multiple test cases for each function you have to implement, and consider edge cases carefully.

As a reminder, you *may* (and are encouraged to!) discuss these problems with your classmates during lab and help each other solve them. However, make sure the code you submit is your own.

What to submit

If you believe you have solved all the problems, let a TA or the professor know and they will check your work. This is a good chance to get feedback on problems similar to what you might see on the exam.

If you don't finish by the end of the lab, you may submit something on NYU Classes by Sunday July 7, 11:55 PM.

Either when a TA has checked your solutions, or by the Sunday submission deadline, submit your solutions to the "Lab 6: Stacks and Queues" assignment on NYU Classes.

1 Double-Ended Queues

A **double-ended queue**, or **deque** (pronounced like "deck"), is similar to a queue. However, in a deque, elements can be inserted to the front or the back of the deque. Similarly, elements can be removed from either the front or the back.

For this problem, implement an *array-based implementation* of a Deque, as `class ArrayDeque`. All the operations should run in $O(1)$ amortized time. Your implementation should include the following methods and look something like:

```
class ArrayDeque:
    def __init__(self):
        '''Initialize an empty Deque'''

    def __len__(self):
        '''Return the number of elements in the Deque'''

    def is_empty(self):
        '''Returns True if the Deque is empty'''
    def first(self):
        '''Returns (without removing) the element at the front of the Deque
        or raise an Exception if the Deque is empty'''

    def last(self):
        '''Returns (without removing) the element at the back of the Deque
        or raise an Exception if the Deque is empty'''

    def enqueue_first(self, elem):
        '''Adds elem to the front of the Deque'''

    def enqueue_last(self, elem):
        '''Adds elem to the back of the Deque'''

    def dequeue_first(self):
        '''Remove and return the element at the front of the Deque
        or raise an Exception if the Deque is empty'''

    def dequeue_last(self):
        '''Remove and return the element at the back of the Deque
        or raise an Exception if the Deque is empty'''
```

Hint: Your implementation will be very similar to the implementation of `ArrayQueue` we did in class. The code is available on Classes under "Resources." Refer to this code when implementing `ArrayDeque`.

2 Checking for Balanced Parantheses

Implement a function `def balanced_expression(str_input)`. This function will take a string as an argument, and return True if it is a *balanced expression*, meaning for every "(", there is a corresponding ")" in the correct nested position. In addition, for every "[", there should be a corresponding "]", and for every "{", there should be a corresponding "}". As a brief example, the strings `"()"` and `"[()]"` are balanced while `"("` and `"([)]"` are not.

Examples: A call to `balanced_expression("{{([])}}([])")` should return True. On the other hand, a call to `balanced_expression("{{[(])}}")` should return False.

Similarly, a call to `balanced_expression("([]{{[]}())}")` should return False, and a function call to `balanced_expression("(([]{{[]}})")` should also return False. On the other hand, calling the function `balanced_expression("([]{{[]}()})")` should return True.

Hint: You should use a stack for this. Scan through the string and push each left parenthesis to the stack. When you see a right parenthesis, its type (")", "]", or "}") should match the type of the left parenthesis at the top of the stack. If the stack is empty, then there was no matching left parenthesis. When you've finished scanning the string expression, if the stack is empty then the string was a balanced expression. If the stack is not empty at the end, this means there was a left parenthesis with no matching right parenthesis.

Work with pen/pencil and paper first. Scan through the string and simulate this approach with a stack. See how it works. Then try implementing it.

3 Balancing HTML Tags

In HTML, portions of the document are delimited by HTML *tags*. A simple html tag has the form: `<name>`. The corresponding closing tag then has the form: `</name>`. An HTML document should have matching tags:

every tag should have a corresponding closing tag. An example of an HTML document with tags is provided with the lab on Classes. Write a function `def is_matched_html(html_str)` that takes a string `input_str` containing HTML tags and returns True if the HTML tags are matched and nested properly.

Here is what you should do:

1. Write a generator `def get_tags(html_str)` that yields, one by one, the HTML tags in `html_str`. The tag will start with a `<` symbol and end with a `>` symbol. You may assume the symbols `<` and `>` do not appear anywhere in the text except around tags. You may want to use the `find` method for strings. Note that the find method can optionally take an extra parameter indicating the index in the string to begin searching. You may refer to the following sample code, which yields parts of a string (called tokens) between commas.

    ```
    def get_tokens(input_str):
        start_ind = 0
        end_ind = input_str.find(",", start_ind)
        while end_ind != -1:
            yield input_str[start_ind:end_ind]
            start_ind = end_ind+1
            end_ind = input_str.find(",", start_ind)
        yield input_str[start_ind:]
    ```

    For example, with the code above, calling `get_tokens("one,two,three,four")` will yield the strings `"one"`, then `"two"`, then `"three"`, and then `"four"`.

    Read this code and understand it (note how it makes use of the optional parameter to `find`). Then, write a similar generator which will yield HTML tags (you will need to search for a `<` symbol and the following `>` symbol to get the full tag).

2. Once you have this generator written, write the function `is_mathced_html(html_str)` to check whether the HTML tags in the given string are properly matched. You will use the same approach as with the previous problem (checking if parentheses are properly matched), only this time with opening and closing HTML tags rather than left and right parentheses.