NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Fall 2017

Homework #5
Due by Friday 10/27, 11:55pm

Submission instructions:
1. You should submit your homework in the NYU Classes system.
2. For this assignment you should turn in 5 files. One '.py' file for each question. Name your files 'YourNetID_hw5_q1.py', 'YourNetID_hw5_q2.py', 'YourNetID_hw5_q3.py', etc.

Question 1:
Give an alternative implementation for the *Queue* ADT.

Implementation Requirements:
1. For the representation of `Queue` objects, your data members should be:
   - **Two** Stacks – of type `ArrayStack`
   - Additional $\theta(1)$ space - for additional data members, if needed

2. Any sequence of n `enqueue` and `dequeue` operations (starting with an empty queue) should run in worst-case of $\theta(n)$ altogether.

Question 2:
Give a Python implementation for the *MidStack* ADT. The *MidStack* ADT supports the following operations:
- `MidStack()`: initializes an empty MidStack object
- `midS.is_empty()`: returns `True` if `S` does not contain any elements, or `False` otherwise.
- `len(midS)`: Returns the number of elements `midS`
- `midS.push(e)`: adds element `e` to the top of `midS`.
- `midS.top()`: returns a reference to the top element of `midS`, without removing it; an exception is raised if `S` is empty.
- `midS.pop()`: removes and returns the top element from `midS`; an exception is raised if `midS` is empty.
- `midS.mid_push(e)`: adds element `e` in the middle of `midS`.
  That is, assuming there are n elements in S: In the case n is even, `e` would go exactly in the middle. If n is odd, `e` will go after the $\frac{n+1}{2}th$ element.

For example, your implementation should follow the behavior as demonstrated in the two execution examples below:

```
>>> midS = MidStack()
>>> midS.push(2)
>>> midS.push(4)
>>> midS.push(6)
>>> midS.push(8)
>>> midS.mid_push(10)
>>> midS.pop()
8
>>> midS.pop()
6
>>> midS.pop()
10
>>> midS.pop()
4
>>> midS.pop()
2
```

```
>>> midS = MidStack()
>>> midS.push(2)
>>> midS.push(4)
>>> midS.push(6)
>>> midS.push(8)
>>> midS.push(10)
>>> midS.mid_push(12)
>>> midS.pop()
10
>>> midS.pop()
8
>>> midS.pop()
12
>>> midS.pop()
6
>>> midS.pop()
4
>>> midS.pop()
2
```

Implementation Requirements:

1. For the representation of `MidStack` objects, your data members should be:
   - A Stack – of type `ArrayStack`
   - A double ended queue – of type `ArrayDeque`
   - Additional $\theta(1)$ space - for additional data members, if needed

2. Your implementation should support the `mid_push` operation in $\theta(1)$ amortized time. For all other Stack operation, the running time should remain as it was in the original implementation (That is, $\theta(1)$ amortized for `push` and `pop`, and $\theta(1)$ worst-case for `top`, `len` and `is_empty`).

<u>Question 3:</u>
Implement an interpreter-like **postfix calculator**. Your program should repeatedly:

- Print a prompt to the user. The prompt should be: '-->'
- Read an expression from the user
- Evaluate that expression
- Print the result

Your calculator should support 2 kinds of expressions:

1. **<u>Arithmetic expressions</u>** – are given in postfix notation. The tokens of the expression are separated by a space.
2. **<u>Assignment expressions</u>** – are expression of the form:

   variable_name = *arithmetic_expression*

   When evaluated, it first evaluates the *arithmetic_expression* (given in postfix notation), and then it associates that value with *variable_name* (in a data structure of your choice).

   <u>Notes</u>:
   - The value of an assignment expression, is the name of the variable being assigned.
   - Assume that the *variable_name*, the '=' symbol, and the *arithmetic_expression* are separated by a space.

<u>Notes</u>:

1. Arithmetic expressions can contain variable names, for referencing to values associated with variables that were defined before.
2. You may assume that the input the user enters, is valid. That is, the arithmetic expressions are legal; all variables used in an expression were already defined; etc.
3. The program should keep reading, and evaluating expressions until the user types 'done( )'.

Your program should interact with the user **exactly** as demonstrated in the example below:

```
--> 4
4
--> 5 1 -
4
--> x = 5 1 -
x
--> x
4
--> x x +
8
--> y = 1 x + 3 4 * - 2 /
y
--> y
-3.5
--> done()
```

Question 4:

Give a Python implementation for the *MaxStack* ADT. The *MaxStack* ADT supports the following operations:

- `MaxStack()`: initializes an empty MaxStack object
- `maxS.is_empty()`: returns `True` if `maxS` does not contain any elements, or `False` otherwise.
- `len(maxS)`: Returns the number of elements in `maxS`
- `maxS.push(e)`: adds element `e` to the top of `maxS`.
- `maxS.top()`: returns a reference to the top element of `maxS`, without removing it; an exception is raised if `maxS` is empty.
- `maxS.pop()`: removes and return the top element from `maxS`; an exception is raised if `maxS` is empty.
- `maxS.max()`: returns the element in `maxS` with the largest value, without removing it; an exception is raised if `maxS` is empty.

Note: Assume that the user inserts only integers to this stack (so they could be compared to one another, and a maximum data is well defined).

For example, your implementation should follow the behavior below:

```
>>> maxS = MaxStack()
>>> maxS.push(3)
>>> maxS.push(1)
>>> maxS.push(6)
>>> maxS.push(4)
>>> maxS.max()
6
>>> maxS.pop()
4
>>> maxS.pop()
6
>>> maxS.max()
3
```

Implementation Requirements:

1. For the representation of `MaxStack` objects, your data members should be:
   - A Stack – of type `ArrayStack`
   - Additional $\theta(1)$ space - for additional data members, if needed
2. Your implementation should support the `max` operation in $\theta(1)$ worst-case time. For all other Stack operation, the running time should remain as it was in the original implementation.

Hint: You may want to store a tuple, as elements of the `ArrayStack`. That is, to attach to every “real” data in this stack some additional information.

Question 5:

Implement the following function:

`def permutations(lst)`

The function is given a list `lst` of integers, and returns a list containing all the different permutations of the elements in `lst`. Each such permutation should be represented as a list.

For example, if `lst=[1, 2, 3]`, the call `permutations(lst)` could return `[[1, 2, 3], [2, 1, 3], [1, 3, 2], [3, 2, 1], [3, 1, 2], [2, 3, 1]]`

Implementation Requirements:

1. Your implementation should be **non-recursive**.
2. Your implementation is allowed to use a Stack, a Queue, and $\theta(1)$ additional space.

Hint: Use the stack to store the elements yet to be used to generate the permutations, and use the queue to store the (partial) collection of permutations generated so far.