CSCI-SHU 210 Data Structures

Assignment 5 Stack and Queue

Problem 1: Leaky Stack

Your task is to solve P-6.35:

The introduction of Section 6.1 notes that stacks are often used to provide "undo" support in applications like a Web browser or text editor. While support for undo can be implemented with an unbounded stack, many applications provide only limited support for such an undo history, with a fixed-capacity stack. When push is invoked with the stack at full capacity, rather than throwing a Full exception (as described in Exercise C-6.16), a more typical semantic is to accept the pushed element at the top while "leaking" the oldest element from the bottom of the stack to make room. Give an implementation of such a LeakyStack abstraction, using a circular array with appropriate storage capacity.

More info:

You should make a class `LeakyStack`. It should support `__init__(self,maxsize)`, `push(self,x)`, `pop(self)`, `__len__(self)`, `is_empty(self)` and `__str__(self)`.

Please start from the attached skeleton code in Problem1_LeakyStack.py file.

The constructor takes a parameter which specifies the maximum number of items the stack can hold. If the stack reaches this size, and a push is performed, the **oldest** item in the stack is removed and forgotten. `__str__` is for debugging, code something reasonable.

Write some code that demonstrates your stack and its leaky features.

Important:

- All operations except __str__ should run in O(1) time (assuming append on lists takes O(1) time).
- The leaky stack does not resize. (In other words, static array, never call `append`( ))
- If the stack is full, oldest element gets lost.
- No additional self variables allowed in this class. In other words, don't modify `__init__` function.

Problem 2: Boost Queue

You are to code a class called `BoostQueue`. A `boostQueue` is like a regular queue, but it also supports a special `boost` operation, that moves the element currently at the back of the queue a specified number of steps forward.

Class `BoostQueue` is a Queue. Which means you only need to code one function:

- `boost(self,k)` **:** moves the element from the back of the queue k steps forward. If the queue is empty an exception is raised. If k is too big (greater or equal to the number of elements in the queue) the last element will become the first. No return value.

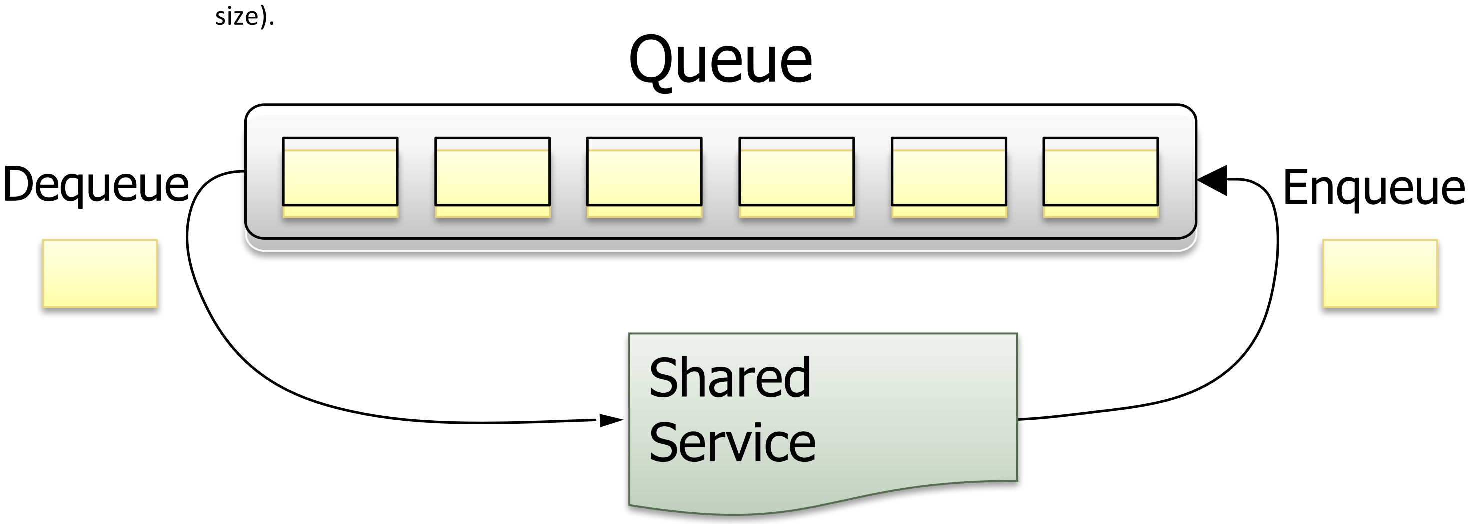
Please start from the attached skeleton code in Problem2_BoostQueue.py file.

Important:

- `boost(self,k)` **:** should run in worst case O(k) time.
- The `BoostQueue` is implemented using Static Array. (Python List with no append( ))

Problem 3: Round Robin Schedulers

In certain applications of the queue ADT, it is common to repeatedly dequeue an element, process it in some way, and then immediately enqueue the same element. Modify the ArrayQueue implementation to include a **rotate( )** method that has semantics identical to the combination, **Q.enqueue(Q.dequeue( ))**. However, your implementation should be more efficient than making two separate calls (for example, because there is no need to modify size).



Please start from the attached skeleton code in Problem3_RoundRobinSchedulersUsingQueueDynamic.py file and modify rotate() function.

Problem 4: Shared memory

Write a `class DoubleStack` to provide two stacks (`stack1` and `stack2`) that share the same list. The list has fixed size, no data item can be pushed in either stack when the list is full, but each stack can grow independently.

Hint. There is a hard way and an easy way of coding this exercise. The easy way positions each stack cleverly in the array so that pushing an element in a stack has no effect on the elements of the other stack.

Please start from the attached skeleton code in Problem4_Sharedmemory.py file.

Important:

- For simplicity, you don't have to handle `Full/Empty Exceptions`. I will only test with valid push/pop operations.
- No additional self variables allowed in this class. In other words, don't modify `__init__` function.
- Never call self.array.`append`(value). The size of the list is fixed.

Problem 5: Infix to postfix

`Infix notation` is easy to read for *humans*, whereas `postfix notation` is easier to parse for a machine. The big advantage in `postfix notation` is that there never arise any questions like operator precedence.

Infix Example: (3 + 2) / 4 + (3 * 2 + 4)
Postfix Example: 3 2 + 4 / 3 2 * 4 + +

- Implement function `infix_to_postfix(string)`, takes `infix notation string` as parameter, prints corresponding `postfix notation` on the screen.

Please start from the attached skeleton code in Problem5_Infix_to_postfix.py file. Complete the to do part to print postfix expressions.

You can also use this link to convert infix to postfix:

http://www.mathblog.dk/tools/infix-postfix-converter/

The following steps will print a string of infix notation in postfix order.

Algorithm
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.
…..**3.2** Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop and output from the stack until an '(' is encountered.

6. Repeat steps 2-6 until infix expression is scanned.
7. Pop and print from the stack until it is empty.

Important:

- Input infix string contains spaces between each operand/operator.
- Use a stack!
- You may encounter 6 operators like **+ - * / ( )**
- I will only test with valid inputs.
- For simplicity, no ^ operator because a ^ b ^ c evaluates b ^ c first.