CSCI-SHU 210 Data Structures

Assignment 5 Stack and Queue

Problem 1: Leaky Queue

You are going to code a class called `LeakyQueue`. A `LeakyQueue` is like a regular queue, but it has a fixed capacity. When the `LeakyQueue` is full, the "oldest" element in the `LeakyQueue` (front of the queue) is lost. The front of the queue then updates and becomes the second oldest element in the queue.

For example, your capacity 5 queue should behave like the following:

```
Front →1, 2, 3, 4, 5          enqueue(6)
Front →2, 3, 4, 5, 6          enqueue(7)
Front →3, 4, 5, 6, 7          enqueue(8)
Front →4, 5, 6, 7, 8

while not q.is_empty( ):
  print(q.dequeue( ))

4
5
6
7
8
```

More info:

You should implement class `LeakyQueue`. It should support `__init__(self, maxsize)`, `enqueue(self, e)`, `dequeue(self)`, `__len__(self)`, `is_empty(self)` and `__str__(self)`.

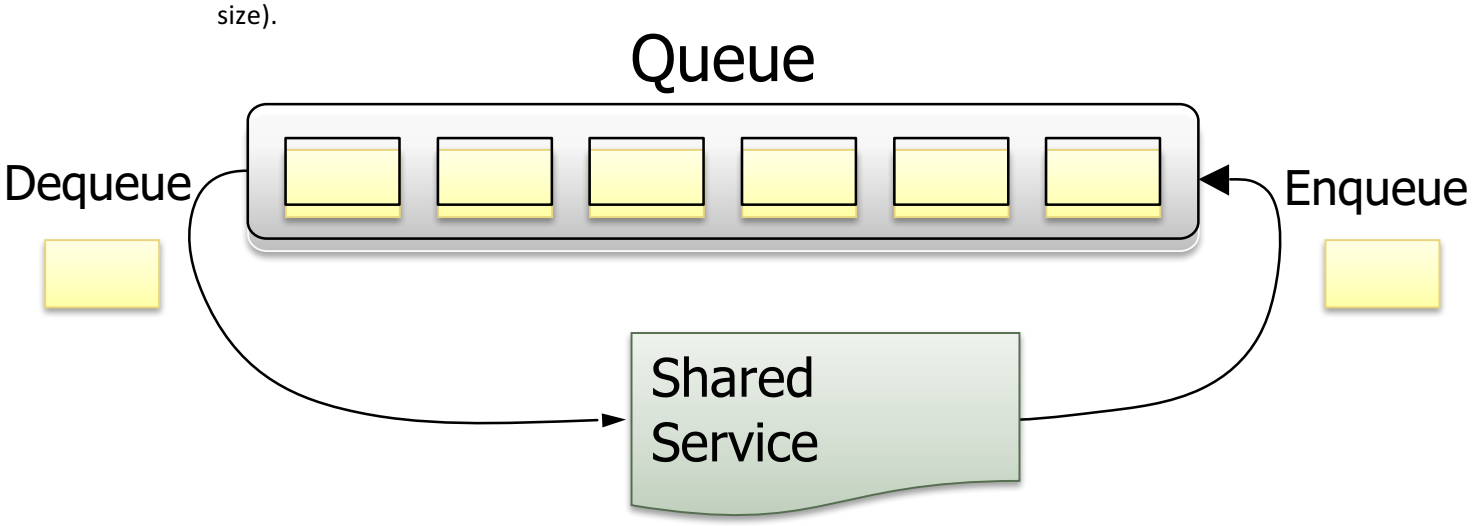
Use the attached skeleton code in Problem2_LeakyQueue.py file.

Important:

- All operations except __str__ should run in O(1) time. ### 10pts
- The leaky queue does not resize. (In other words, it is a fixed size array, never call `list.append`())
- If the queue is full, oldest element gets lost.
- No additional self variable is allowed in this class. In other words, don't modify `__init__` function.

Problem 2: 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).



Use the attached skeleton code in Problem3_RoundRobin.py file and modify rotate( ) function.

Important:

- Again, in this question, the Queue ADT is implemented with circular array.
- Your `rotate( )` function should do exactly the same job as **Q.enqueue(Q.dequeue( ))**, but more efficiently.

Problem 3: 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)
Corresponding Postfix Example: 3 2 + 4 / 3 2 * 4 + +

- Implement function `infix_to_postfix(string)`, takes `infix notation string` as parameter, returns corresponding `postfix notation string`.

The following steps will return 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, add it to the returning string.
3. Else if the scanned token is '(', add '(' to the stack.
4. Else if the scanned token is ')', pop and append tokens from the stack, until a '(' is popped.
5. Else, the scanned token is in **{+ - * / }**,
5.1 While the stack is not empty and, the top of the stack token has higher or equal precedence than the scanned token, pop and append tokens from the stack.
5.2 Now the stack is either empty, or top of the stack has lower precedence than the scanned token. Push the scanned token to the stack.
6. While the stack is not empty, pop and append tokens from the stack.
7. Return the answer.

Use the attached skeleton code in Problem5_Infix_to_postfix.py file. Complete the to do part to return the postfix expression string.

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.

Problem 4: Token checker

Implement function, `check_tokens(filename)`, Your function should return True if for the given file, all the “[“, “]”, “{“, “}”, “(“, “)” are matching correctly. Correct means that for each left token (“[“, “{“, “(“), we are able to find a matching right token (“)”, “}”, “]”), there should be no extra right tokens, and there’s no left alone “[“, “{“, “(“ at the end.

For example, we have a file named, “test.c”

```c
int main()
{
    int i, sum = 0;
    int n[10];

    for ( i = 0; i < 10; i++ ) {
        n[ i ] = i + 100;
    }

    for ( i = 1; i <= LAST; i++ ) {
        sum += i;
    }
    printf("sum = %d\n", sum);

    return 0;
}
```

```
>>> check_tokens(“test.c”)
True
```

Important:

- You don’t need to check syntax for the input file.
- You only check if all the “[“, “{“, “(“, “)”, "}”, “]” are matching correctly.
- You should also check there’s no extra “[“, “{“, “(“ at the end.
- Use a stack!

Use the attached skeleton code in Problem6_TokenChecker.py file.

Problem 5: Queue using Stacks

Implement a Queue using only two Stacks. In the __init__ method, you can only use two stacks. You need to implement a Queue (enqueue, dequeue, front, len, is_empty method). You can use ArrayStack class and it's methods (push(e), pop(),top, len, is_empty) to implement a Queue (FIFO).

Use the attached skeleton code in Problem5_Queue_using_stacks.py file.