CSCI-SHU 210 Data Structures

Recitation 5 Stacks and Queues

You have a series of tasks in front of you. Complete them! Everyone should code on their own computer, but you are encouraged to talk to others, and seek help from each other and from the TA/LAs.

Important for this week:

- Understand the concept of ADT (Abstract Data Type)
  - Abstract Data Type: From the user's view
  - Actual implementation: From the developer's view

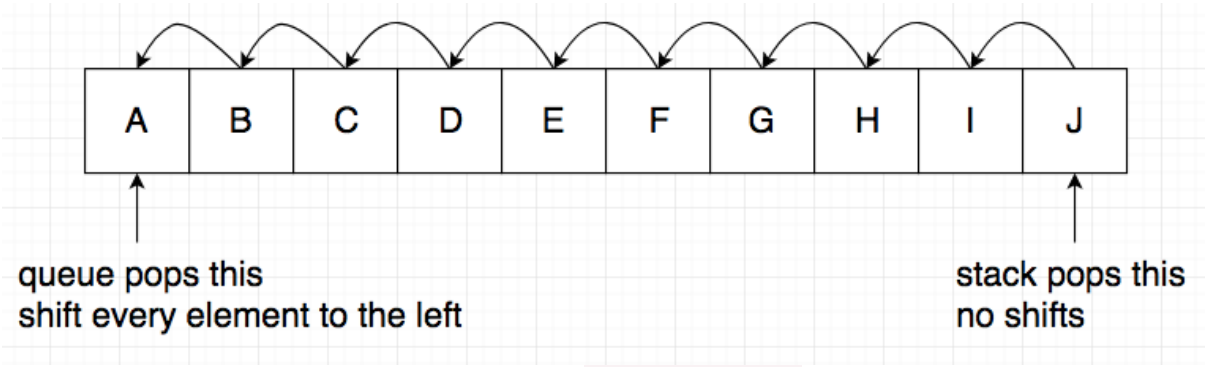
- Understand what is a stack (LIFO);
  - Understand why list append( ) / pop (-1) stack is perfectly fine.

- Understand what is a queue (FIFO);
  - Understand why circular queue is better than append( ) / pop(0) queue

- Can use stack & queue ADTs to solve some problems.

1. ArrayQueue

We are going to implement the `circular version queue`. There are many ways to implement a queue, one way we observed was using python list with `append()/pop()` operations. However, **stack we are popping index -1, queue we are popping index 0.**



Hence, implementing queue with python list `append()/pop()` is not that efficient. We are getting `O(1) enqueue` operation, `O(n) dequeue` operation. You can check badQueue.py file for details.

How can we do it better? We want O(1) dequeue operation instead of O(n)!
Solution: Make the queue circular.

> Your task 1: Implement class `ArrayQueue` in `Circular_Queue.py file` avoid using `append()/pop()` operations. So dequeue operation has constant runtime `O(1)`.
>
> How to make the queue circular? Answer: Use modulo (%) operation.

```
1.  class ArrayQueue():
2.      DEFAULT_CAPACITY = 10
3.
4.      def __init__(self):
5.          self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
6.          self._size = 0
7.          self._front = 0
8.
9.      def __len__(self):
10.         pass
11.
12.     def is_empty(self):
13.         pass
14.
15.     def first(self):
16.         pass
17.
18.     def dequeue(self):
19.         pass
20.
21.     def enqueue(self, e):
22.         pass
23.
24.     def __str__(self):
25.         pass
```

Code snippet 1: starting point for `ArrayQueue` task 1.

2. Computing Spans

Let's start from the definition: what is the span of an array?

Given an an array ***X***, the span ***S***[*i*] of ***X***[*i*] is the maximum number of consecutive elements ***X***[*j*] immediately preceding ***X***[*i*] and such that ***X***[*j*] ≤ ***X***[*i*]

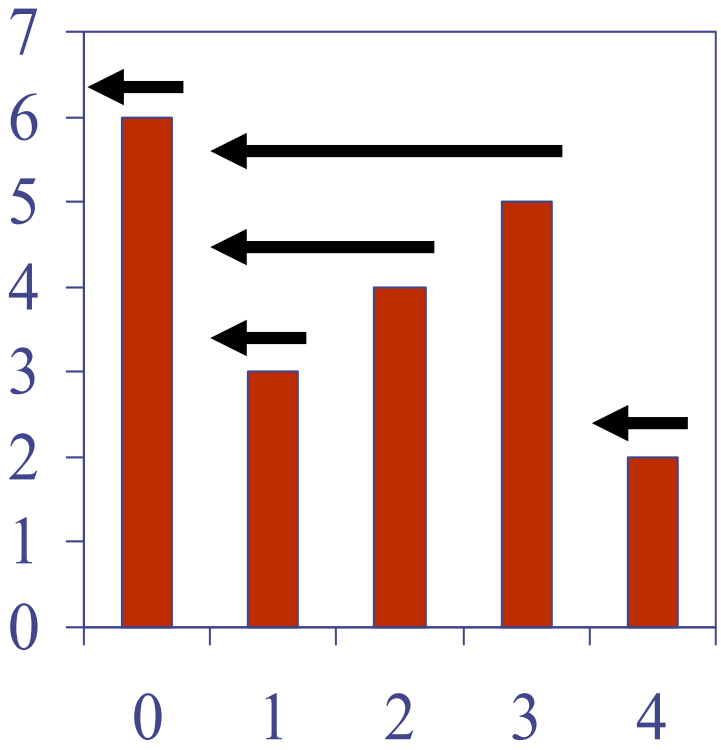


Figure 1. Graphical explanation for span of an array.

X[i]	6	3	4	5	2
Span	1	1	2	3	1

Chart 1. Corresponding span value for array X.

Your task 2: Implement `spans1(X)` function in Spans.py file, takes an array of integer X as input, compute and return the corresponding span array S.

No stack allowed. For each index, we look to the front of array until we found a value that is greater than this index's value.

Question 1: What is the runtime for task 2 algorithm?

Your task 3: Implement `spans2(X)` function in Spans.py file, takes an array of integer X as input, compute and return the corresponding span array S.

Use a stack. We use the stack to organize data inputs.

If the top of the stack is "smaller or equal" than the next data, top of the stack should be popped.

Question 2: What is the runtime for task 3 algorithm?

3. Double ended queue

Now let's implement a double ended queue that also runs O(1) on all enqueue(), dequeue() operations.

> Your task 4:Implement class `ArrayDeque` in `ArrayDeque.file`, so the queue can be inserted/popped from both sides.
>
> `add_first/add_last/delete_first/delete_last` operations should run `O(1)`.

```
class ArrayDeque:
    DEFAULT_CAPACITY = 10
    def __init__(self):
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0

    def __len__(self):
        pass

    def is_empty(self):
        pass

    def is_full(self):
        pass

    def first(self):
        pass

    def last(self):
        pass

    def delete_first(self):
        pass

    def add_first(self, e):
        pass

    def delete_last(self):
        pass

    def add_last(self, e):
        pass

    def __str__(self):
        pass
```

Code snippet 3: starting point for `ArrayDeque`.

4. Evaluation of arithmetic expressions.

```
Your task 5: Write a function evaluate(string) in Evaluation.py file which
evaluates infix arithmetic expression string. It evaluates an infix
expression string and return the value of the expression.
```

For the sake of simplicity, valid expressions respect the following rules:

- There is a space between each operand/operator, parsing becomes easy.
- Expression consists of operands (numbers), operators (+, -, *, /), and left (and right) parenthesis.

Hint. Use two separate stacks, one to push/pop operators and the other to push/pop values. Think about how your evaluation method should parse the arithmetic expression, and in particular about what should happen when it encounters a ')'.

```
Example 1:
>>> print(evaluate("9 + 8 * 7 / ( 6 + 5 ) - ( 4 + 3 ) * 2"))
0.0909090909
Example 2:
>>> print(evaluate("9 + 8 * 7 / ( ( 6 + 5 ) - ( 4 + 3 ) * 2 )"))
-9.66666666667
```

The following steps will compute the value of arithmetic expression string.

0. Preprocess the input.

1. While there are still tokens to be read in,
   1.1 Get the next token.
   1.2 If the token is:
      1.2.1 A number: push it onto the value stack.
      1.2.2 A left parenthesis: push it onto the operator stack.
      1.2.3 A right parenthesis:
        1 While the thing on top of the operator stack is not a
        left parenthesis,
          1 Pop the operator from the operator stack.
          2 Pop the value stack twice, get two operands.
          3 Apply the operator to the operands, in the correct order.
          4 Push the result onto the value stack.
        2 Pop the left parenthesis from the operator stack, and discard it.

1.2.5 An operator (call it thisOp):
1 While the operator stack is not empty, and the top thing on the operator stack has the same or greater precedence as thisOp,
1 Pop the operator from the operator stack.
2 Pop the value stack twice, getting two operands.
3 Apply the operator to the operands, in the correct order.
4 Push the result onto the value stack.
2 Push thisOp onto the operator stack.

2. While the operator stack is not empty,
1 Pop the operator from the operator stack.
2 Pop the value stack twice, getting two operands.
3 Apply the operator to the operands, in the correct order.
4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value stack should have only one value in it, which is the final result.