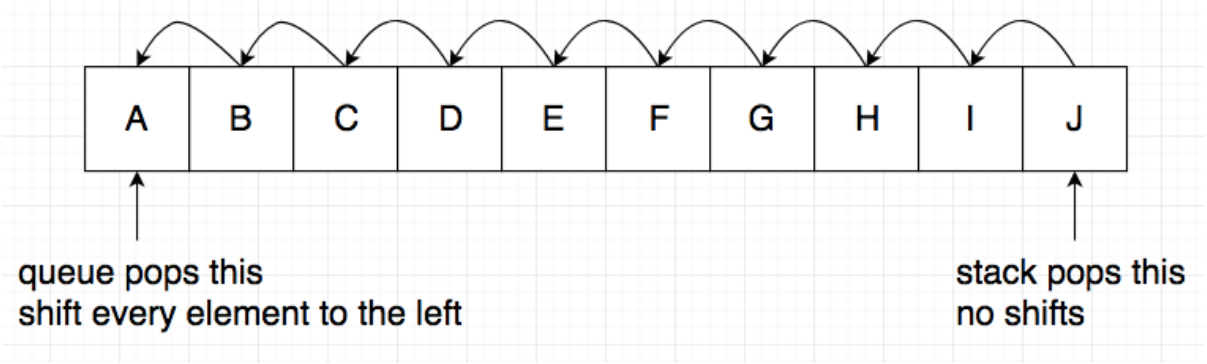
CSCI-SHU 210 Data Structures

Recitation 6 Stack and Queue

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.

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, avoid 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

Starting from 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*]

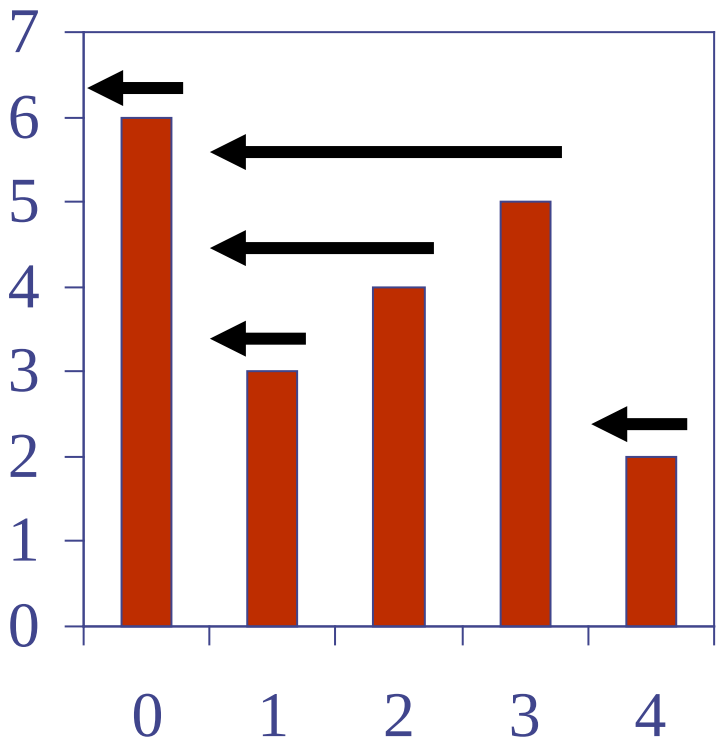


Figure 1. Graphical explanation for span of an array.

X	6	3	4	6	2
S	1	1	2	3	1

Chart 1. Corresponding span value for array X.

Your task 1: Implement `spans1(X)` function, 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 1 algorithm?

Your task 2: Implement `spans2(X)` function, takes an array of integer X as input, compute and return the corresponding span array S.

Use a stack. We push values into the stack until we encounter, a value is smaller than the top of the stack.

Keep track the index of the oldest value within the stack. When we encounter a value smaller than top of the stack, we can perform index subtraction to get span value.

Question 2: What is the runtime for task 2 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: Implement class `ArrayDeque`, so the queue can be inserted/popped from both sides.
>
> `Enqueue/Dequeue/peek` operations should run `O(1)`.

```
1.  class ArrayDeque:
2.      DEFAULT_CAPACITY = 10
3.      def __init__(self):
4.          self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
5.          self._size = 0
6.          self._front = 0
7.
8.      def __len__(self):
9.          pass
10.
11.     def is_empty(self):
12.         pass
13.
14.     def is_full(self):
15.         pass
16.
17.     def first(self):
18.         pass
19.
20.     def last(self):
21.         pass
22.
23.     def delete_first(self):
24.         pass
25.
26.     def add_first(self, e):
27.         pass
28.
29.     def delete_last(self):
30.         pass
31.
32.     def add_last(self, e):
33.         pass
34.
35.     def __str__(self):
36.         pass
```

Code snippet 3: starting point for `ArrayDeque`.

4. Evaluation of arithmetic expressions

```
Your task: Write a function evaluate(string) which evaluates infix arithmetic
expressions. 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.
- No power operator (^). Because a^b^c = a^(b^c), very complicated.

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

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

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, getting 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.
    1.2.4 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.

5. Infix to postfix (If we have time, extra practice)

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 + +

```
Your task: Implement function infix_to_postfix(string), takes infix notation
string as parameter, prints or return corresponding postfix notation on
the screen.
```

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 not empty.

Important:

- Input infix string contains spaces between each operand/operator.
- Use a stack!
- **+ - * / ( )** you may encounter 6 operators.
- Assume inputs are valid.
- For simplicity, no ^ operator because a ^ b ^ c evaluates b ^ c first.