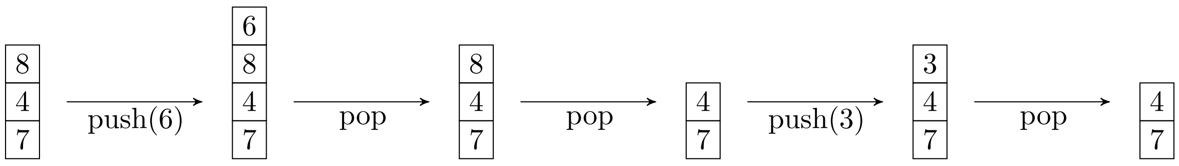
Chapter 7

Stacks and Queues

Here are described two structures used for storage of elements. The structures will provide two operations: *push* (inserting the new element to the structure) and *pop* (removing some element from the structure).

7.1. Stack

The stack is a basic data structure in which the insertion of new elements takes place at the top and deletion of elements also takes place from the top. The idea of the stack can be illustrated by plates stacked on top of one another. Each new plate is placed on top of the stack of plates (operation push), and plates can only be taken off the top of the stack (operation pop).



The stack can be represented by an array for storing the elements. Apart of the array, we should also remember the size of the stack and we must be sure to declare sufficient space for the array (in the following implementation we can store N elements).

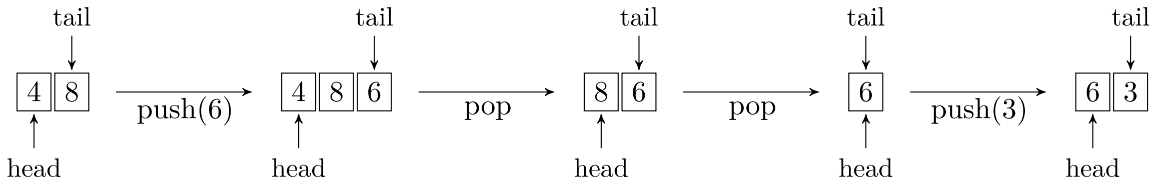
7.1: Push / pop function — $O(1)$.

```
stack = [0] * N
size = 0
def push(x):
    global size
    stack[size] = x
    size += 1
def pop():
    global size
    size -= 1
    return stack[size]
```

The *push* function adds an element to the stack. The *pop* function removes and returns the most recently pushed element from the stack. We shouldn't perform a pop operation on an empty stack.

7.2. Queue

The queue is a basic data structure in which new elements are inserted at the back but old elements are removed from the front. The idea of the queue can be illustrated by a line of customers in a grocery store. New people join the back of the queue and the next person to be served is the first one in the line.



The queue can be represented by an array for storing the elements. Apart of the array, we should also remember the front (head) and back (tail) of the queue. We must be sure to declare sufficient space for the array (in the following implementation we can store $N - 1$ elements).

7.2: Push / pop / size / empty function — $O(1)$.

```
queue = [0] * N
head, tail = 0, 0
def push(x):
    global tail
    tail = (tail + 1) % N
    queue[tail] = x
def pop():
    global head
    head = (head + 1) % N
    return queue[head]
def size():
    return (tail - head + N) % N
def empty():
    return head == tail
```

Notice that in the above implementation we used cyclic buffer (you can read about it more at http://en.wikipedia.org/wiki/Circular_buffer).

The *push* function adds an element to the queue. The *pop* function removes and returns an element from the front of the queue (we shouldn't perform a pop operation on an empty queue). The *empty* function check whether the queue is empty and the size function returns the number of elements in the queue.

7.3. Exercises

Problem: You are given a zero-indexed array A consisting of n integers: $a_0, a_1, \ldots, a_{n-1}$. Array A represents a scenario in a grocery store, and contains only 0s and/or 1s:

- 0 represents the action of a new person joining the line in the grocery store,
- 1 represents the action of the person at the front of the queue being served and leaving the line.

The goal is to count the minimum number of people who should have been in the line before the above scenario, so that the scenario is possible (it is not possible to serve a person if the line is empty).

Solution $O(n)$: We should remember the size of the queue and carry out a simulation of people arriving at and leaving the grocery store. If the size of the queue becomes a negative number then that sets the lower limit for the number of people who had to stand in the line previously. We should find the smallest negative number to determine the size of the queue during the whole simulation.

7.3: Model solution — $O(n)$.

```python
def grocery_store(A):
    n = len(A)
    size, result = 0, 0
    for i in xrange(n):
        if A[i] == 0:
            size += 1
        else:
            size -= 1
            result = max(result, -size)
    return result
```

The total time complexity of the above algorithm is $O(n)$. The space complexity is $O(1)$ because we don't store people in the array, but only remember the size of the queue.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.