Reflection Essay - Assignment 3

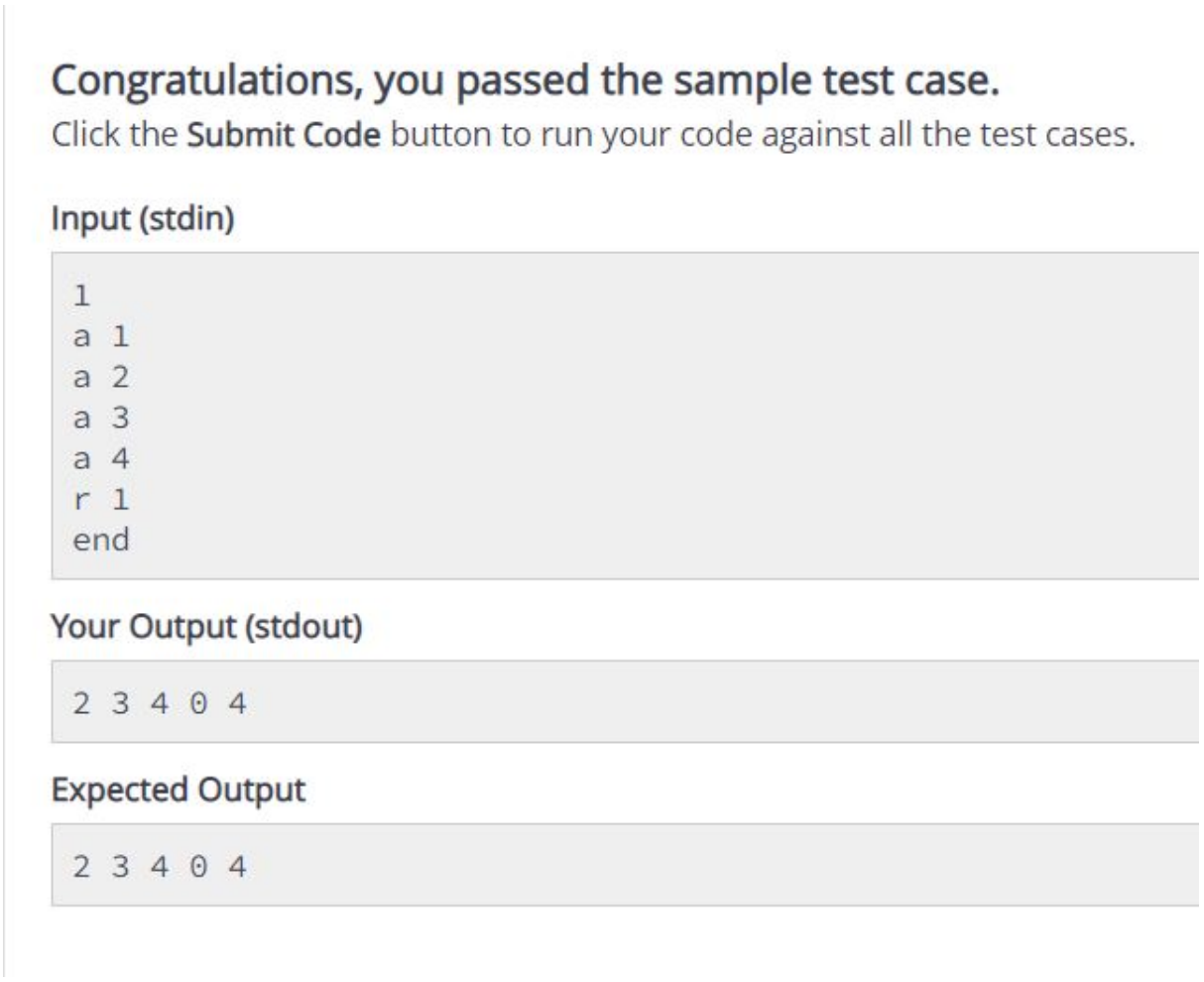
Prasann Viswanathan

Question 1: What grows up must shrink down!

My goal was to modify remove, removeFromEnd and shrinkArray functions in the least intrusive way possible in order to implement the shrinking procedure of the array.

I achieved my goals satisfactorily as all I had to do was to check if the updated number of elements on shrinking was half of the capacity of the array, and if so I would make an array of half the original capacity and store all the elements there. The condition checking and array resizing took very few additional instructions to implement.

Output:

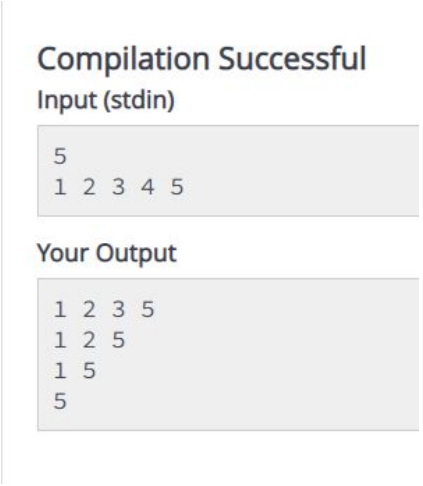


References: NA (100% my work)

Question 2: Deleting a node in a Singly Linked List

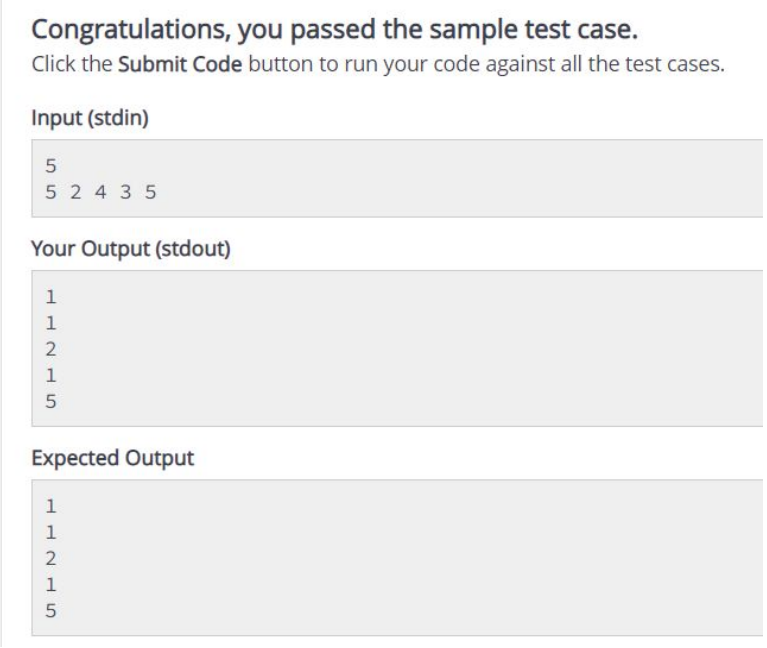
The crux of this problem is in observing the subtle condition that says ‘It is guaranteed that the node to be deleted will not be the last node.’ With this is mind the implementation is very simple. Make the node to be deleted a copy of the node following it, and delete the node following it. This preserves the structure of the linked list desired and makes deletion of a node, an O(n) operation, O(1) and successfully passes all test cases.

Output:



References:
https://stackoverflow.com/questions/793950/algorithm-for-deleting-one-element-in-an-single-linked-list-with-o1-complexity
(I had come up with the same idea before finding the reference so 70% my work 30% push provided by reference)

Question 3: Span
The simplest algorithm would be O(n^2) involving counting backwards from each index X[i] till we reach the first element X[j] such that X[j]>X[i] thus, S[i]=i-j
However we need a linear time algorithm. So the trick would be to keep track of the largest element's position preceding each element and storing them on a stack. Suppose this position is given by idx(i), then S[i]=i-idx(i) and if X[i] is the largest encountered yet, S[i]=i+1. Implementing this we get a performance of O(n) as the maximum iterations in i-th step is size of stack and i+1th step is stack size after i-th step. Observing this condition holds for all i, our worst case is 2 per iteration giving us O(2n) \equiv O(n).
Output:



References: https://www.geeksforgeeks.org/the-stock-span-problem/

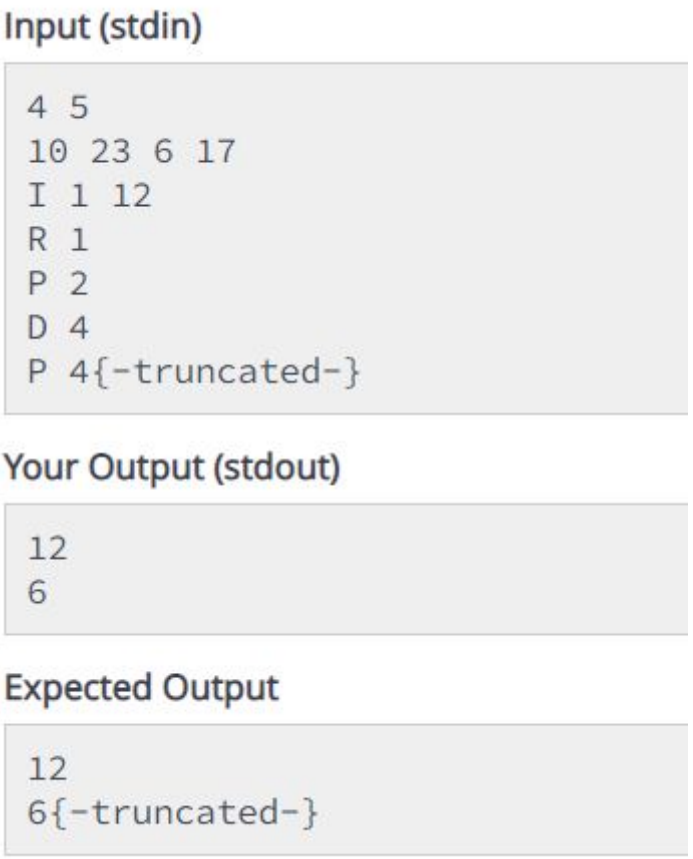
(I was able to pass all test cases but one without any reference, this reference helped me pass the remaining one case. 80% my work, 20% reference)

Question 4: Too many Queries

I implemented a double ended queue using a doubly linked list thinking that rotation would be O(1) and other instructions can be performed iterating from the front or rear based on what idx was closer to. This would give an average of n/8 iterations of per operation (print, delete, insert). Even this didn't work within the time limit for many cases.

My final implementation isn't something I am very proud of because it makes use of all pre-existing standard vector functions, which seems like cheating to me. But this passed all test cases. I don't understand why my original implementation was too slow as even in this, rotate, insert and delete are O(n) operations while print is O(1). This was the fastest implementation I could think of.

Output:



References:

https://www.geeksforgeeks.org/doubly-linked-list/
https://www.geeksforgeeks.org/implementation-deque-using-doubly-linked-list/
https://www.geeksforgeeks.org/vector-in-cpp-stl/

(Earlier implementation was made with regular reference from the doubly linked list template so 50% writing and understanding effort 50% geeks for geeks
Vector implementation was a necessity to pass all cases, 100% self effort except for syntax googling)

Question 5:
There are two ways of interpreting this question:

1 - Each time we make a copy of the stack we do not intend to maintain the copy and original be the same after further operations.
In such a case the amortized time complexity is O(1): simply given by

$1/p(p+\lfloor p/n \rfloor \times n) = 2$ (Assuming p is a multiple of n) => O(1)

2 - We maintain each copy to be identical to the original.

Here after n operations we will need to do 2n (One for stack and one for first copy) then 3n and so on uptil $\lfloor p/n \rfloor$ times.

So our operations will be:

$(n+2n+3n+...+\lfloor p/n \rfloor n) = n \lfloor p/n \rfloor (\lfloor p/n \rfloor +1)/2$ ($1+2+...+\lfloor p/n \rfloor = \lfloor p/n \rfloor (\lfloor p/n \rfloor +1)/2$)

So the amortized complexity is given by

$1/p(n \times \lfloor p/n \rfloor (\lfloor p/n \rfloor +1)/2)$

Assume n | p perfectly allowing us to get rid of floors

We have O(p/n) which one would expect to be the case.

(100% own solution)

Question 6:

Solution for part 1:
Background: The XOR operation (Denoted by ^ from now) has the following properties
x^x = 0 and x^0 = x and x^y = y^x
This basically means if we XOR many items together, if each of the items occurs and even number of times except one, the result will be the one item that occurs an odd number of times. With that trick in mind here is the algorithm which is time complexity O(n)

```
ADT data;
int XOR = 0;
for(int i=0; i<=N; i++){
        XOR ^= data.f(i);
}
for(int i=1; i<=N; i++){
        XOR ^= i;
}
for(int i=1; i<=N; i++){
        if(XOR == data.f(i))
                cout<<i<<'\n';
```

}
cout<<XOR<<'\n';
After the two XOR manipulation loops, all Y values \in {1, …, N} will occur twice except the one repeated value. Thus we have got the Y that is mapped to by X1 and X2

Then we loop again over x values till we find the two values x_1 and x_2 that map to y
Our program prints both these values.

Solution for part 2:

Didn't understand the truncated linked list part.
Assuming we have the function f that gives corresponding y value, we can use above algorithm here as well with only XOR (1 variable) being the required data
Once again O(n) time complexity O(1) space. No need for a truncated list at all.

(100% own solution, XOR trick was because of my EE background in digital systems)