| Prob. 1 | Prob. 2 | Prob. 3 |
|---|---|---|
| | | |

## Problem 1.

Stack is capable of inserting and deleting at the end of a list but queue accepts elements from one end and elements are removed from the other end so they have quite different operational semantics. However, a queue can easily be simulated by two stacks, namely *inbox*, $S_i$ and *outbox*, $S_o$.

We will use $S_i$ for Insert-Queue and $S_o$ for Delete-Queue operation. Therefore, we push the element into $S_i$ if we want to insert a new element into queue. Delete operation contains two scenarios according to state of $S_o$. If it isn't empty, we pop its top element; otherwise, we transfer all elements from $S_i$ to $S_o$ respectively by popping from $S_i$ and pushing into $S_o$. Then, we have the required element at the top of $S_o$ so we just pop it.

For analysis, we define potential function: $\Phi = 2n_i$, where $n_i$ is the number of elements in stack $S_i$. There is a multiplier 2 since when *outbox* is empty, we transfer elements from *inbox* by pop and push, where each one has a cost of 1 unit, totally 2. Then, we have the following amortized cost for each Insert-Queue operation:

$$actual\ cost + \Delta\Phi = stack\ push + change\ of\ inbox = 1 + 2 \leq O(1)$$

For each Delete-Queue operation, we have to consider two cases. First, easier one, when $S_o$ is not empty, the amortized cost is:

$$actual\ cost + \Delta\Phi = stack\ pop + change\ of\ inbox = 1 + 0 \leq O(1)$$

When $S_o$ is empty, the amortized cost is:

$$\begin{aligned} actual\ cost + \Delta\Phi &= n\ stack\ pop + (n-1)\ stack\ push + change\ of\ inbox \\ &= (2n_i - 1) + (0 - 2n_i) \leq O(1) \end{aligned}$$

As a result, amortized cost of each Insert-Queue and Delete-Queue operations is constant.

Double-ended queue can also be implemented by two stack that are mounted each other as in the following figure.

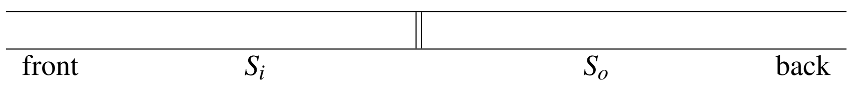

Table 1: Double-ended queue via two regular stacks

For each operation, we execute following steps:

- insert-front: push the element into $S_i$
- insert-back: push the element into $S_o$
- delete-back: check if there is an element in $S_o$. If exists, pop top element and return it. If $S_o$ is empty and $S_i$ is *not* empty, pop and push all elements from $S_i$ to $S_o$ one by one, and at the end pop top element of $S_o$ and return it. If both are empty, delete operation couldn't be completed, error is returned.
- delete-front: this operation is exactly opposite of delete-back operation. Firstly, check $S_i$, if it isn't empty, pop top element and return it. Otherwise, if $S_o$ isn't empty, transfer elements from $S_o$ to $S_i$ one by one, and at the end pop top element and return it. If both are empty, error is returned.

With above implementation, we can easily find a sequence whose amortized cost isn't $O(1)$. Sequence:

- $n$ insert-back
- $\frac{n}{2}$ delete-front and $\frac{n}{2}$ delete-back alternating between each other

We have done $2n$ operation totally and total cost is:

$$n+(2n+1)+[2(n-1)+1]+[2(n-2)+1]+\cdots+[2*1+1]=n^2+3n=\Theta(n^2)$$

In this equation, first term is the cost of $n$ insert-back operations and following terms are consecutive delete operations. Since the number of operations is the order of $\Theta(n)$ and cost is the order $\Theta(n^2)$, this is a counterexample to show that amortized cost of each four operation isn't the order of $O(1)$, constant. Costly operations are delete operations so one of them should be removed and in another way, we have used one insert type so removing it wouldn't help us to reduce our cost. However, one of the delete operations is removed, it can be shown that remaining three operations have constant amortized cost0. For analysis, let's remove delete-back operation and use potential function $\Theta = 2n_o$ where $n_o$ is the number of elements in $S_o$. Therefore, we have following amortized cost for insert-back:

$$actual\ cost + \Delta\Phi = 1+2 \leq O(1)$$

Insert-front:

$$actual\ cost + \Delta\Phi = 1+0 \leq O(1)$$

When $S_i$ isn't empty, delete-front:

$$actual\ cost + \Delta\Phi = 1+0 \leq O(1)$$

When $S_i$ is empty, delete-front:

$$actual\ cost + \Delta\Phi = (2n_o+1)+(0-2n_o) \leq O(1)$$

As seen above, each operation has constant time amortized cost.

## Problem 2.

Fitstring representation are **not** unique because ”011” can always be written as a carry in the form of ”100”. Moreover, even if we prefer one over another and use it whenever possible, some numbers can be written in multiple forms.

$$1(0)\{2n+1,2n+1\} = 1(01)\{n,n\}$$

In the former representation ”1” is followed by $2n+1$ ”0” and in the latter, ”1” is followed $n$ ”01”.

*increase operation*:
First, how a carry is created:

- If last bit is ”0”, it is just flipped to ”1”.
  - $a0 \rightarrow a1$
- If last bit is ”1”, the previous bit is considered
  - if it is ”0”, the fitstring ends with ”01” and this ”01” is flipped to ”11”. Unlike binary, this works because last two bits represent value of 1.
    - $a01 \rightarrow a11$
  - if is is also ”1”, the fitstring ends with ”11” and this ”11” is flipped to ”01”. Then, a carry is created and propagated to more significant bits.
    - $a11 \rightarrow (a+1)01$

Secondly, general case at the position of $i$ with a carry:

- If it is ”0”, it is just flipped to ”1”.
  - $a(0+1)b \rightarrow a1b$
- If it is ”1”, the number increased to $2F_i$ and carry must be propagated. By using the property of fibonacci numbers, $2F_i = F_i + F_{i-2}$, carry can be propagated one up and two down. Then, following rules are defined:
  - $0(1+1)00 \rightarrow 1001$
  - $0(1+1)01a \rightarrow 1010(a+1)$
  - $1(1+1)00 \rightarrow (1+1)001$
  - $1(1+1)01a \rightarrow (1+1)010(a+1)$

As a result of these rules, the cost depends on propagation of the carry. However, each increment leaves alternating ”0” and ”1”s and at most one pair of adjacent ”1”.

*decrement operation*:

- If last bit is ”1”, it is flipped to ”0”
  - $a0 \rightarrow a1$
- If last is ”0” and previos bit is ”1”, then fitstring ends with ”10”, so ”10” is flipped to ”00” since last two bits represent value of 1.
  - $a10 \rightarrow a00$
- If last two bits are ”00” but third bit is ”1”, then fitstring ends with ”100”, so ”100” is flipped to ”010”.
  - $a100 \rightarrow a010$
- Otherwise, we need a borrow from more significant bits.
  - Search least significant ”1” in the fitstring and assume it is found at position $i$.
  - As explained above, this fit should have two zeros to its right. Therefore, ”100” is flipped to ”011”.
  - We have a ”1” fit at the position $i-2$ and if it has two zeros to its right, recursively the previous step is also applied this fit.
  - At the end, we hit the base cases and do the actual decrement:
    - $100000000\ldots00 \rightarrow 10101010\ldots10$
    - $100000000\ldots0 \rightarrow 10101010\ldots0$
  - Cost is depends on the number of consecutive 0s but leaves alternating 0 and 1s and decrement can create at most one pair adjacent zero pair.

Increment and decrement are costly in terms of trailing 1s and 0s, respectively but they transform the fitstring into alternation and have the next operation taken constant time.

Since cost depends on consecutive fits, we need to count them where alternation is the perfect case. Our potential function keeps track of the number of adjacent same fits:

$$\Phi(f) = \Phi(f_n f_{n-1} f_{n-2} \cdots f_1 f_0) = \sum_{i=1}^{n} f_{i-1} = f_i$$

If an operation has a real cost of $c$, then it must have $c$ identical trailing fits. Moreover, it should have arrived at least the potential of $c-1$ so that it contains $c$ identical fits but operation decreases potential by $c-1$ via transforming string into an alternation. However, each operation can create one pair identical fits in doing alternation but this cost is dominated by constant. As a result, the sum of total cost and potential change is bounded by a constant.

Problem 3.