CSCI 6057 Advanced Data Structures - Assignment 1

Yansong Li
B00755354
yansong.li@dal.ca

Question 1.

Suppose it is an initially empty stack with n operations applied on it, which means that number of elements in the stack is at most n.

First, from worst-case analysis we can obtain that the worst-case cost of $pop(S)$ is $O(1)$ and $push(S,x)$ is $O(n+1) = O(n)$ (i.e., we are pushing an element that is smaller than the element at the bottom of the stack) since the sorted stack is implemented as a linked list. Therefore, the worst-case cost of a sequence of n operations is $O(n^2)$, which gives us an amortized cost of $O(n^2)/\text{n} = O(n)$. However, it is not tight since we are considering each operation individually.

Next, we use aggregate method to calculate the amortized cost again. The $push(S,x)$ operation of the sorted stack is practically a $multipop(S,k)$ operation plus a $push(x)$ operation on a regular stack where k is number of elements in S that is smaller than x. We can make the same observation that we made on regular stack, which is each object can be popped at most once for each time it is pushed. Under the sorted stack scenario, the total number of pops is the sum of regular $pop(S)$ plus the number of elements that have been popped ($multipop(S,k)$ parts) during $push(S,x)$ operations, which cannot exceed the number of elements that have been pushed during $push(S,x)$ operation. Since we can only push one element into the sorted stack for each $push(S,x)$ operation; hence, the number of pushed elements is at most n with n operations. Therefore, the total cost of n $pop(S)$ or $push(S,x)$ is $O(n)$, which gives us an amortized cost of both $pop(S)$ and $push(S,x)$ is $O(n)/n = O(1)$.

Question 2.

First, if we perform a series of c insertions into a resizable array. In the best-case scenario, there are $c-1$ insertions take $\Omega(1)$ time since the array is not full, and the last insertion would take $\Omega(c)$ time since the array is full and a new memory block need to be allocated, then the content of the old memory block will be copied to the new one with the last element inserted in the new block as well. Overall, this series of c insertions takes $\Omega(c+c)$ time.

For a series of n insertions, we divide them into block with size c; hence, there will be $\frac{n}{c}$

blocks. Therefore, the total time of preforming a series of n insertion is

$$(c+c)+(2c+c)+(3c+c)+\cdots+\left(\frac{n}{c}*c+c\right) \leftarrow \frac{n}{c} \text{ parts}$$

$$=c(\frac{n}{c})+(c+2c+3c+\cdots+\frac{n}{c}*c)$$

$$=n+c(1+2+3+\cdots+\frac{n}{c})$$

$$=n+c*\frac{\frac{n}{c}\left(1+\frac{n}{c}\right)}{2}$$

$$=n+\frac{n\left(1+\frac{n}{c}\right)}{2}$$

$$=n+\frac{\left(n+\frac{n^2}{c}\right)}{2}$$

$$=n+\frac{n}{2}+\frac{n^2}{2c}$$

$$=\frac{n^2}{2c}+\frac{3n}{2}$$

$$=\Omega(n^2)$$

Therefore, we have proved performing a series of n insertions into an initially empty resizable array takes $\Omega(n^2)$ time.

Question 3.

(i) Pseudocode of this INCREMENT.

```
INCREMENT (A [0…k-1])
    i ← 0
    while i < k and A [i] = 2
        do   A [i] ← 0
             i ← i + 1
    if i < k
        then  A [i] ← A [i] + 1
```

(ii) Amortized analysis of this INCREMENT.

Aggregate method is applied, we let $k = 4$, then write out the first 16 increments into a table.

Value	$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0	0

1	0	0	0	1
2	0	0	0	2
3	0	0	1	0
4	0	0	1	1
5	0	0	1	2
6	0	0	2	0
7	0	0	2	1
8	0	0	2	2
9	0	1	0	0
10	0	1	0	1
11	0	1	0	2
12	0	1	1	0
13	0	1	1	1
14	0	1	1	2
15	0	1	2	0

Observation: $A[0]$ flips every time (3^0) INCREMENT is called
$A[1]$ flips every three times (3^1) INCRMENT is called
$A[2]$ flips every nine times (3^2) INCRMENT is called
$A[3]$ never flips during the first 16 increments.

Therefore, it can be concluded that for $i = 0, 1, 2, \lfloor \log_3 n \rfloor$ bit $A[i]$ flips $\lfloor n/3^i \rfloor$ times in a sequence of n INCREMENT operations on an initially empty array. For bit $i > \lfloor \log_3 n \rfloor$, bit $A[i]$ never flips. Hence, the total number of flips in the sequence is

$$C(n) = \sum_{i=0}^{\lfloor \log_3 n \rfloor} \left\lfloor \frac{n}{3^i} \right\rfloor \leq \sum_{i=0}^{\infty} \frac{n}{3^i} = n \sum_{i=0}^{\infty} \frac{1}{3^i} = \frac{3}{2} n$$

Hence, we have proved that $C(n) \leq 3/2n$.

(iii) Prove $c \geq 3/2$.

We prove it by contradiction, which is showing for each $c < \frac{3}{2}$ there exist an n such that $C(n) > cn$. Next, we introduce a variable a, which represent the difference between c and $\frac{3}{2}$ $(a > 0)$. Therefore, we are showing that for each $c = \frac{3}{2} - a$ there exist and n such that $C(n) > (c - a)\, n$.

First, we let $b = -\lceil \log_3 a \rceil$; hence, $b \geq -\log_3 a$ and $-b \leq \log_3 a$. Therefore, $3^{-b} = 3^{\lceil \log_3 a \rceil} \leq a$. Therefore, $(c - a)\, n \leq (c - 3^{-b})n$

We let $n = 3^b$, when we are incrementing until n, the last bit will flip 3^0 time, the last second bit will flip 3^1 time, until the first bit will flip 3^b. Therefore,

$$C(n) = \sum_{k=0}^{b} 3^k = \sum_{k=0}^{b} 3^{b+1} - 1$$

From previous we have proved that $(c-a)n \leq (c-3^{-b})n$, then if we let $c = \frac{3}{2}$ and $n = 3^b$, we can obtain that $(c-a)n \leq \left(\frac{3}{2} - 3^{-b}\right)3^b$, and $\left(\frac{3}{2} - 3^{-b}\right)3^b = \frac{3*3^{-b}}{2} - \frac{3^b}{3^b} = \frac{3^{b+1}}{2} - 1$. $\frac{3^{b+1}}{2} - 1$ is clearly less than $3^{b+1} - 1$, which is $C(n)$. Overall, $(c-a)n < C(n)$.

Hence, we have showed that $(c-a)n < C(n)$ for each $c = \frac{3}{2} - a$. Hence, for each $c < \frac{3}{2}$ there exist an n such that $C(n) > cn$, which indicates the inequality $c \geq \frac{3}{2}$ much holds.

Question 4.

(i) Worst-case running times.

$MultiPopX(k)$: When $k = n$ (pop all the elements in X), we pop the total number of n elements out of stack X and each pop takes $O(1)$; hence, the worse-case running time of $MultiPopX(k)$ is $\boldsymbol{O(n)}$.

$MultiPopY(k)$: Similarly, when $k = m$ (pop all the elements in Y), we pop the total number of m elements out of stack X and each pop takes $O(1)$; hence, the worse-case running time is $O(m)$. Also, since $O(c * m) = O(n)$where c is a constant. Therefore, the worse-case running time of $MultiPopY(k)$ is $\boldsymbol{O(n)}$.

$Move(k)$: When $k = n$, we pop the total number of n elements out of stack X and each pop takes $O(1)$, then we push n elements onto stack Y and each push takes $O(1)$, which takes $O(2n) = O(n)$ in total; hence, the worst-case running time of $Move(k)$ is $\boldsymbol{O(n)}$.

(ii) All five operations' amortized cost are $\boldsymbol{O(1)}$.

We analyze the amortized costs of the five operations by the potential method. We define the potential function $\emptyset$ on two initially empty stacks to be the two times the number of elements in stack X plus the number of elements in stack Y minus two times the number of elements moves, which is $\boldsymbol{2n + m - 2u}$ (u represents the number of elements moves). Therefore, for two empty stacks D_0that we start from with no move yet, $\emptyset(D_0) = 0$. Since the number of

objects in two stacks cannot be negative and the number of moves cannot exceed the number of elements in X, $\emptyset(D_i) \geq 0$.

If the ith operation is $\boldsymbol{PushX(a)}$ on stack X with total $S = 2n + m - 2u$ elements in two stacks minus two times elements moves in $\emptyset(D_i - 1)$, then

$$\begin{aligned} a_i &= C_i + \emptyset(D_i) - \emptyset(D_i - 1) \\ &= 1 + (S + 2) - S \\ &= 3 \end{aligned}$$

Each element that are pushed onto stack X can be either popped or moved once, so we overcharge the $PushX(a)$ operation by 2, which means the overcharges are enough to cover the undercharges cause by either $MultiPopX(k)$ or $Move(k)$.

If the ith operation is $\boldsymbol{PushY(a)}$ on stack Y with total S elements in two stacks plus elements moves in $\emptyset(D_i - 1)$, then

$$\begin{aligned} a_i &= C_i + \emptyset(D_i) - \emptyset(D_i - 1) \\ &= 1 + (S + 1) - S \\ &= 2 \end{aligned}$$

Each element that are pushed onto stack Y can be popped only once, so we overcharge the $PushX(a)$ operation by 1, which means the overcharges are enough to cover the undercharges cause by $MultiPopX(k)$.

If the ith operation is $\boldsymbol{MultiPopX(k)}$ on stack X, which causes $k' = (k, n)$ elements out of two stacks but the number of elements moves remain the same, then

$$\begin{aligned} a_i &= C_i + \emptyset(D_i) - \emptyset(D_i - 1) \\ &= k' + (S - k') - S \\ &= 0 \end{aligned}$$

If the ith operation is $\boldsymbol{MultiPopY(k)}$ on stack X, which causes $k' = min(k, m)$ elements out of two stacks but the number of elements moves remain the same, then

$$\begin{aligned} a_i &= C_i + \emptyset(D_i) - \emptyset(D_i - 1) \\ &= k' + (S - k') - S \\ &= 0 \end{aligned}$$

If the ith operation is $\boldsymbol{Move(k)}$ on two stacks, which causes 0 elements out of two stacks but $k' = min(k, n)$ new element moves, then

$$\begin{aligned} a_i &= C_i + \emptyset(D_i) - \emptyset(D_i - 1) \\ &= 2k' + (S - 2k') - S \\ &= 0 \end{aligned}$$

Hence, we have proved that all the amortized cost of all five operations are all constants, which means they are all $\boldsymbol{O(1)}$.