CS261P- DATA STRUCTURES HW01

Name: Sameer Ganesh Shinde
UCI ID: 19644693 UCI NetID: sgshinde

1.
Consider a "superstack" data structure which supports four operations: create, push, pop, and superpop.
The four operations are implemented using an underlying standard stack as shown below.

```
def create(): s = stack.create()
def push(x): s.push(x)
def pop(): return s.pop()
def superpop(k, a): // k is an integer, a is an array with size >= k
i = 0
while i < k
        a[i] = s.pop()
        i = i + 1
```

show that each of these operations uses a constant amortized number of stack operations.
In your solution you should:
• define your potential function φ.
• state, for each operation, its actual time, the change in potential, and the amortized time.

Ans:

Here we can implement stack using LinkedList.

Potential function:
Φ = Number of elements in stack

def create(): s = stack.create()
Creating stack is same as creating a linked list. In that case we only need to initialize head, which will Cost 1.
Create():
O(1) actual time
$\Delta\Phi = 0$
O(1) amortized time

def push(x): s.push(x)
LinkedList can grow or shrink as per the elements adding or deletion. It will be automatically handled as we don't have to maintain a capacity of a LinkedList.

Push(x):
O(1+1) actual time (Time to push an element plus constant additional time).
$\Delta\Phi = \Phi(\text{new}) - \Phi(\text{old})$

If L is the old size of stack then,
$\Phi(\text{old}) = L$
$\Phi(\text{new}) = L + 1$

$\Delta\Phi = L+1 - L$
$= 1$

Amortized time = Actual time + $c \cdot \Delta\Phi$
$\leq c \cdot (2) + c \cdot (1)$
$= 3c$
$= O(1)$

```
def pop (): return s.pop()
```

Similarly, for pop operation

Pop():

O(1+1) actual time (time to pop an element plus constant additional time).

$\Delta\Phi = \Phi(\text{new}) - \Phi(\text{old})$

If L is the old size of stack then,

$\Phi(\text{old}) = L$

$\Phi(\text{new}) = L - 1$

$\Delta\Phi = L - 1 - L$
$= -1$

Amortized time = Actual time + $c \cdot \Delta\Phi$
$\leq c \cdot (2) - c \cdot (1)$
$= c$
$= O(1)$

```
def superpop(k, a):                // k is an integer, a is an array with size >= k
    i = 0
    while i < k
        a[i] = s.pop()
        i = i + 1
```

SuperPop():

O(k+1) actual time

$\Delta\Phi = \Phi(\text{new}) - \Phi(\text{old})$

If L is the old size of stack then,

$\Phi(\text{old}) = L$

$\Phi(\text{new}) = L - k$

$\Delta\Phi = L - k - L$
$= -k$

Amortized time = Actual time + $c \cdot \Delta\Phi$
$\leq c \cdot (k+1) - c \cdot (k)$
$= c$
$= O(1)$

2.

Suppose we add a superpush operation to the superstack from the previous problem, defined as follows:

```
def superpush(k,A): // k is an integer, A is an array with size >= k
    i = 0
    while i < k
        S.push(A[i])
        i = i + 1
```

Is it still true that each of the superstack operations uses a constant amortized number of stack operations? Answer YES or NO. If your answer is YES, give an amortized analysis as in the previous problem. (If you need to use a different potential function that is fine, just be sure to define it.) If your answer is NO, explain why.

The answer is No.

Explanation:

SuperPush(k, A):

O(k+1) actual time (Time to push k elements plus constant additional time).

$\Delta\Phi = \Phi(\text{new}) - \Phi(\text{old})$

If L is the old size of stack then,

$\Phi(\text{old}) = L$

$\Phi(\text{new}) = L + k$

$$\begin{aligned}\Delta\Phi &= L + k - L \\ &= k\end{aligned}$$

$$\begin{aligned}\text{Amortized time} &= \text{Actual time} + c \cdot \Delta\Phi \\ &\leq c \cdot (k+1) + c \cdot (k) \\ &= 2k \\ &= O(k)\end{aligned}$$

Superpush operation amortized time is O(k), which doesn't satisfy the constant amortized time.