COL106 A2 (Bonus Part BY➔ Kabir Nagpal, Entry Number ➔ 2022EE31743)

Question How can we make the push, pop operation in part b better than worst case O(n**2) or better than amortized O(n). This can be done using Hysteresis (Class notes).

Solution-

How is it O(n**2) in worst case for normal implementation?

Ans=> Size doubles on adding an element when size == capacity and size decreases when we pop an element at size==capacity/2+1 the capacity of the array is made half. The complexity of each of these push and pop when the capacity is changed is O(n) because we have to copy the entire array to a new array which will have double or half the size respectively.

Now for worst case of push and pop operation the number of elements in array (size)== capacity of the array. It works as follows➔

Lets say we add an element to the array at size==capacity ➔ the array size doubles and the complexity for adding SINGLE element comes out to be order O(n) (otherwise O(1))(because we are doubling the size of the array which takes O(n)) and then if we pop the element again then condition for reduction of size is also met in pop operation so for this pop as well we get complexity of O(n). So for each (Push, Pop) we get a complexity of O(n). IE for worst case n push pop we get complexity(n**O(n)==O(n**2)) and average case (per case)complexity of (O(n)) (amortized).

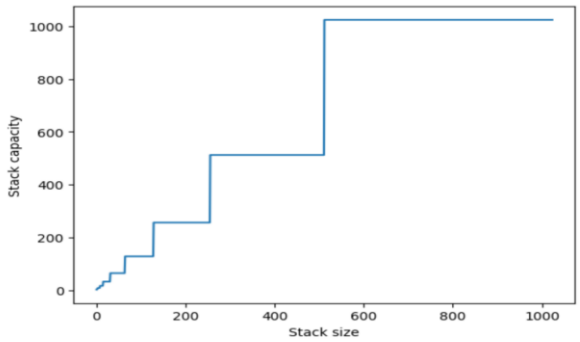
Hence ➔ O(n) + O(n)

(Class note pics)

Dynamic Stacks: Worst case

- In general, consider 2^k+1 push() operations
- Followed by 2^{k-1} push()+pop() operations
- Total number of operations
  $2^k+1 + 2 * 2^{k-1} = 2^k + 2^k + 1 = 2^{k+1}+1 = N$
- Time taken by 2^k+1 push() operations
  $O(2^k+1) = O(N/2) = O(N)$
- Time taken by each push()+pop() operation
  $2^{k+1} + 2^k + 2^k = 2^{k+2} = 2N-2$
- Total time taken by 2^{k-1} [= (N-1)/4] push()+pop() operations
  $((N-1)/4) * (2N - 2) = O(N^2)$

Analysis of Dynamic Stacks



Consider a sequence of 513 push() operations followed by 511 (pop() + push()) operation.

Solution ➔Hysteresis. So basically in hysteresis we change the rate at which we resize when we are popin our elements. As explained in class we grow our stack same as before but when we pop we resize when (size<= capacity/(c**2)) and not (size<= capacity/c). By this we get amortized O(1) and worst case O(N). Hysteresis in itself means delay / lag (from ell :-)). So basically our pop operation gets delayed and gets distributed over the number of push and pop operations we do, resulting in an overall constant amortized cost for push operations, which then implies a constant amortized time complexity for the entire sequence of push and pop operations.

Cost analysis ➔ (worst case)

For push with resize and pop ➔ O(n) + O(1)

For pop with resize and push ➔ O(n) + O(1)

Averaged push pop ➔ it will have worst case O(n)

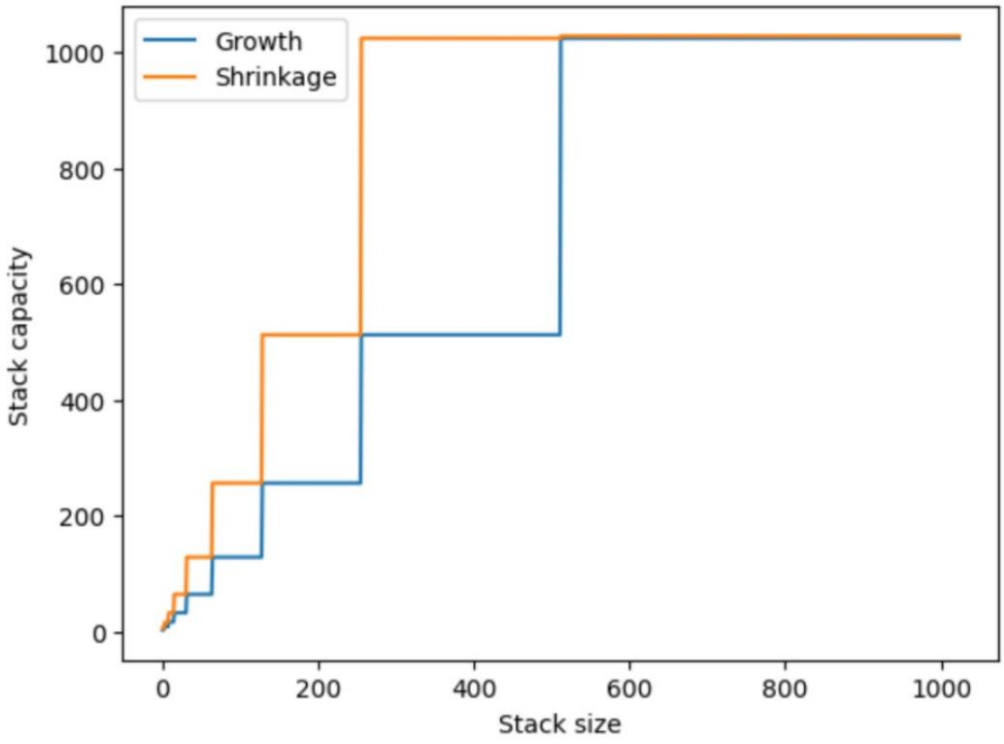
Cost analysis ➔ (amortized)

For Push and pop ➔ O(1) +O(1) (As average case hence resizing considered) ➔ overall O(1)

Despite the expense associated with resizing and copying the array, the occurrences of this operation remain relatively infrequent when compared to the total number of insertions. This is where the utility of amortized analysis becomes evident. While certain individual insertions might exhibit higher costs, the overall average cost across the entire sequence remains significantly lower.

The amortized analysis provides a means to demonstrate that the average cost per insertion is notably lower than the potentially worst-case scenario. Occasional individual insertions experience longer durations due to array resizing.

Class notes reference➔



Mathematical proof uses potential function (out of scope as of now)