In particular we may want to know the total worst case of sequence of operations and it may be some of those are cheap , while only few of them are expensive. We analyze these sequence of operation and predict a worst case average time which is lower than the worst case of a particular expensive operation.

Ex – amortized analysis can be used on Hash tables , disjoint sets and splay trees.

**Amortized Cost** : cost(n operations ) / n .

Consider a scenario where we have a dynamically allocated array . Insertion cost of new element is O(1) . But if the array is full then we have to create new array of double the size and store the previous element( O(n) operation) and then add new element to the new array which is O(1) operation . Lets analyze this situation.

**Aggregate method :** total running time for a sequence of operations is analyzed. Use amortized cost to evaluate.

note : generally the size of new array is double the size of old array

let ci be the cost of ith insertion : -

ci = i if i-1 is power of 2

= 1 otherwise

Alternatively , ci  = 1 +di where di is the cost of doubling the array size -

di = i-1 , i-1 is the power of 2 --- size = capacity

= 0 otherwise

Summig them up , all 1s sum to O(n) and all di sum to O(n) :

∑ 1≤ i ≤ n ci  ≤ n + ∑ 0≤ j ≤ m 2j

where m = └log(n-1) ┘ .

we will divide above equation by n to get amortized cost ---

above equation /n = O(n)/n = O(1)

**Accounting ( Banker’s) Method –** we use token system or saving account system .

Idea – cheap operations are charged little bit more than their true cost , and extra is deposited in the savings account . And we charge little less for expensive operations and the extra amount is paid from savings account .

In our example of dynamic array – insertion is cheap(O(1)) operation so we will charge it extra and save it as token . And when array is full then we have to create new array (which is expensive – O(n)) then we use this extra token.

--- > here we will charge 3 token – 1 will be used immediately as element is inserted in array(2 left) .

--- > 1 token will be given to that element so that when array will be full and thus during reallocation that token will be used(similar to prepaid)

--- > 1 last token will be assigned to capacity/2 element prior to that element.

**Potential ( Physicist’s ) Method -** define a potential function ɸ (phi) which maps the state of the data structure to the integer.

* ɸ(h0) = 0 , So intial state of data structure has potential 0.
* ɸ(ht) ≥ 0

Amortized cost for operation t : ct + ɸ(ht) - ɸ(ht-1) , where c sub t (is true cost) + change in potential after doing the operation and before doing the operation

Choose phi such that –

* if c sub t is small , the potential increases (to save some potential for later work)
* if c sub t is large , decrease the potential ( rest potential is paid by the saved one )

the sum of n operation (sum of true cost) : I

the sum of all amortized cost : i +ɸ(hi) -ɸ(hi-1))

=c1 + ɸ(h1) - ɸ(h0) +…

=c2 + ɸ(h2) - ɸ(h1) +…

=cn  + ɸ(hn) - ɸ(hn-1)

= I  + ɸ(hn) - ɸ(h0) = I  + ɸ(hn) ≥ I (ɸ(h0) = 0 , ɸ(ht) ≥ 0)

We got arelation between sum of amortized cost and sum of true cost . Also we come up with a lower bound on the sum of amortized cost which is the sum of the true cost. So if we want to look at cost of a entire sequence of of operations , we know its atleast sum of the true cost.

Let for dynamic array, do n calls pushback

Left , ɸ(h) = 2 \* size – capacity

Verify => ɸ(h0) = 2\*0 -0 and ɸ(hi) = 2\* size – capacity > 0 since size > capacity/2

Amortized cost of adding elements (without resizing) :

ci +ɸ(hi) -ɸ(hi-1)

= 1 + 2\*sizei – capi –(2 \* sizei-1 – capi-1)

Since we are not resizing hence our capacity is not changing and it cancel out.

=1 + 2(sizei – sizei-1)

Difference in size is 2 because we added one element –

=1+2

=3

The same number we used in bankers method .

Amortized cost of adding elements (with resizing) :

Let k = sizei-1 = capi-1  , because we are about to resize the array and as the array is full hence size = capacity

ɸ(hi-1) = 2 \* sizei-1 - capi-1  = 2k – k =k .So after resizing -

ɸ(hi) = 2 \* sizei - capi = 2 \*(k+1) - 2\*k = 2 , size sub I value is k+1 because we moved k elements and then we added one.

So amortized cost is –

ci +ɸ(hi) -ɸ(hi-1)

= sizei  + 2 –k , c sub i is same as size sub I as we move i-1 elements and then add 1 and cost is same as size.

= k+1 +2 –k = 3