Solution Set 5

Solutions to problem set 4

Solution 1

The question asks us to work out the amortized cost of each operation for a total of n operations. An operation costs i if it is a power of k and costs 1 if it isn't a power of k.

For simplicity let's assume $k = 2$ and the total number of operations is a power of k. Let the total number of operations be 8. The operations and their cost is shown below.

Operation Number	1	2	3	4	5	6	7	8
Cost	1	2	1	4	1	1	1	8

In the above table, operation# 2, 4 and 8 are powers of $k = 2$ and are shown in blue. We need to calculate the total cost of all the operations and then divide it by n to get the amortized cost per operation.

We'll to compute two costs to get the total cost of n operations. One operations which are a power of $k = 2$ and two which aren't a power of k.

Operations power of k

We'll ignore the first operations as a power of k even though $k^0=1$. The cost in our example is thus $2 + 4 + 8 = 14$. The first question we need to answer is that given n operations, how many of those operations will be powers of k. The answer is k raised to what power would equal n? And as we have seen earlier,

it is $\boldsymbol{log_k n}$. Let's plug in the values for our example and see if the answer matches what we expect.

$$log_2 8 = 3$$

This is exactly what we see in our example, 2, 4 and 8 appear as powers of ***k*** = ***2***. Note, if we were counting the first operation as a power of 2 then we would have $\log_k n + 1$ powers of ***k*** occurring in ***n*** operations.

Next, we need to find out what these operations would sum to.You can deduce by inspection or if you are mathematically adept would already know that the sum of the first ***x*** powers of ***k*** is $\boldsymbol{k^{x+1} - 1}$. We can test it on our example.

$$2^{3+1} - 1 = 2^4 - 1 = 15 = 1 + 2 + 4 + 8$$

As you can see that this includes 1 as a power of ***k*** so we will need subtract an additional 1 for our example.

So now we can calculate the sum of all the operations in ***n*** operations that are powers of ***k***. There will be $\boldsymbol{log_k n}$ such operations and their sum is represented by

$$k^{(number\ of\ operations\ that\ are\ power\ of\ k+1)} - 1 - 1$$

$$= k^{(log_k n+1)} - 2$$

Operations not power of k

We have already calculated the number of operations that are power of ***k*** among the total ***n*** operations, which is $\boldsymbol{log_k n}$ so the number of operations not power of ***k*** are $\boldsymbol{n - log_k n}$.

Total Cost

The total cost is thus the sum of costs of operations that are power of k and those that are not power of k.

$$Total\ Cost = Cost\ of\ operations\ power\ of\ k + Cost\ of\ operations\ not\ power\ of\ k$$

$$Total\ Cost = (k^{(log_k n+1)} - 2) + 1 * (n - log_k n)$$

$$Total\ Cost = ((k^{log_k n} * k) - 2) + (n - log_k n)$$

$$Total\ Cost = ((n * k) - 2) + (n - log_k n)$$

$$Total\ Cost = nk - 2 + n - log_k n$$

We can plug in values for ***n*** and ***k*** from our example to verify that our expression correctly computes the total cost.

$$Total\ Cost = nk - 2 + n - log_k n$$

$$Total\ Cost = 8 * 2 - 2 + 8 - log_2 8$$

$$Total\ Cost = 16 - 2 + 8 - 3$$

$$Total\ Cost = 19$$

Amortized Cost

To get the amortized cost we'll simply divide the total cost by the total number of operations which is ***n***.

$$Amortized\ Cost = \frac{nk - 2 + n - log_k n}{n}$$

$$Amortized\ Cost = \frac{nk}{n} + \frac{n}{n} - \frac{log_k n}{n} - \frac{2}{n}$$

$$Amortized\ Cost = k + 1 - \frac{log_k n}{n} - \frac{2}{n}$$

As ***n*** becomes larger and larger $^{2}/_{n}$ becomes 0. Similarly as ***n*** increases the term $^{\log_k n}/_{n}$ also tends to zero because the denominators increases faster than the numerator. Finally, we can ignore the constant term 1 and are left with only ***k***. Hence the amortized cost of each operation is thus ***O(k)***.

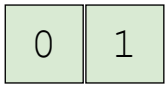
Solution 2

We need to work out the amortized cost of the dynamic array's insert operation using accounting method. We need to be mindful of the following costs

- Cost to insert the element the first time in the array
- Cost to copy the element whenever the array is doubled

The first cost is simple to account for. We can start with a cost of 1 to insert an element in the array. One may falsely argue that we can add an additional unit of cost to account for the copying on the doubling of the array for a total cost of 2. Let's see how that is false but gives us insight to reach the correct solution.

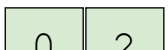
After adding the first element the array has a credit of 1 unit



Adding the second element, causes a doubling and the credit of 1 unit is utilized to copy the first element

The problem is apparent now, on adding the third element the dynamic array would double and the first element has no credit to copy itself over to the new array. The key here is to realize that whenever a doubling happens, the elements added after the initial array has doubled need to bring in credit to copy all the elements that existed before the doubling.

Let's take an example, say the array is of size two but only a single element has been added. When we add the second element, we assign a cost of 3 units to the insert operation. One unit is paid for the insert, one unit is left to copy the second element itself when the array doubles next time and the third unit is there to cover for charge to copy the first element. Below is the pictorial representation.

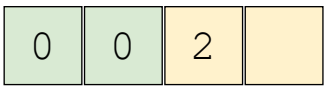


The second element holds a credit of two

Now when the third element gets inserted the array gets doubled.

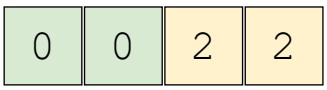
The credit is utilized to copy over the first two elements to the new array

Now, the third element is added with a credit of 3, one of which is utilized in inserting the third element itself, leaving us with 2 units of credit.



Third element with a credit of 2

Now when the fourth element gets added, it brings a credit of 2 units too.



Fourth element also comes in with a credit of 2

Now we have a total credit of four in the data-structure and four elements in all. Now when we double the array, we'll have enough credit to cover for the charge to copy the four elements to the new array.

Credit is consumed in copying the old half of the array when a doubling occurs

Now one can observe that when the fifth element is added, it'll bring with it the credit to copy the first element when a doubling is required. Similarly, the sixth element brings credit for copying the second element so on and so forth. Essentially, the new half of the array brings with it the credit to copy the old half of the array when the next doubling is required.

Hence the amortized cost of insertion is 3 units or constant.