Introduction to Amortized Analysis

Amortized analysis is a method for analyzing a sequence of operations in an algorithm and gives the average time or space taken by the algorithm over the complete sequence.
It is useful when a few operations in the sequence are very slow, and others are faster to execute.
Amortized time complexity in data structures analysis will give the average time taken per operation, which will be better than worst-case time complexity.

it is often the case that an algorithm will on occasion perform a very expensive operation, but most of the time the operations are cheap. Amortized analysis is the name given to the technique of analyzing not just the worst case running time of an operation but the average case running time of an operation. This will allow us to balance the expensive-but-rare operations against their cheap-and-frequent peers.

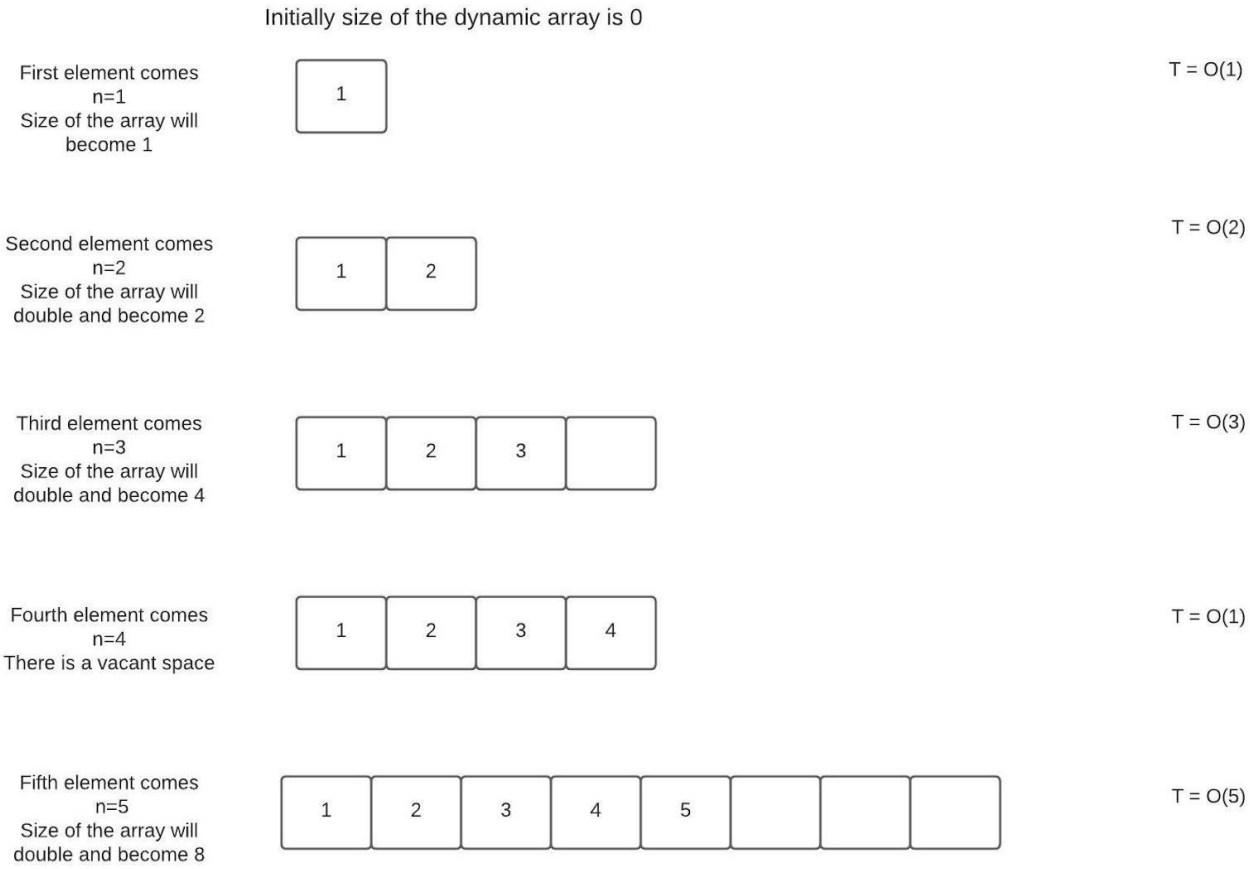
Aggregate Method

The aggregate method calculates the average time complexity of operations in the amortized analysis by using the following mathematical formula:

$$\text{Amortized Time Complexity} = \frac{\text{Time Complexity (N operations)}}{N} = \frac{\text{Time Complexity (Normal operations)+ Time Complexity (Slower operations)}}{N}$$

Example of Amortized Analysis on Dynamic Array

Suppose we have to store elements and we don’t know the number of elements in advance. So, we can use a dynamic array to store the elements. If we declare a large size of the array initially, then we have to compromise with the space complexity. So, we start with a zero-size dynamic array and start inserting elements. Whenever a new element comes, and we have a vacant space in the array, it takes O(1) time to insert a new element into the array. But whenever there is not a vacant space in the array, the size of the array is doubled. So, if there is no vacant space in the array at the time of nth element insertion, it takes O(n) time for inserting that element into the dynamic array.



we can observe that when (n-1) is a power of 2, we come to a case when we don't have a vacant space in the dynamic array and we need to double its size to accommodate the new element. In this case, the time complexity is O(n), else the time complexity to insert the new element is O(1).

$$\text{Amortized cost} = \frac{\overbrace{O(1 + 1 + 1 + ..)}^{N \text{ terms}} + \overbrace{O(1 + 2 + 4 + ...)}^{|\log_2 N| + 1 \text{ terms}}}{N}$$

$$= \frac{O(N) + O(2N)}{N}$$

$$= O(1)$$

What is the Potential method?

According to computational complexity theory, the potential method is defined as:

A method implemented to analyze the amortized time and space complexity of a data structure, a measure of its performance over sequences of operations that eliminates the cost of infrequent but expensive operations.

We then define the amortized time of an operation as

***c + Φ(a') − Φ(a)**, where **c** is the original cost of the operation and **a** and **a'** are the states of the data structure before and after the operation, respectively.*

Analysis of potential method with example

Amortized cost of Push: According to potential function.
Amortised cost(C_i) = Actual cost(c_i) + change in potential
$C_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+n-(n-1) = 2$
So, C_i is also **O(1)**

Amortized cost of Pop: According to potential function:
Amortised cost(Ci) = Actual cost(ci) + change in potential
$Ci = ci + \Phi(D_{i-1}) - \Phi(D_i) = 1+n-1-(n) = 0$
So, C_i is also **O(1)**
Amortized cost of MultiPop: According to potential function:
Amortised cost(Ci) = Actual cost(ci) + change in potential
$Ci = ci + \Phi(D_i) - \Phi(D_{i-1}) = k + s – k – s = 0$
So, C_i is also **O(1)**