COMPSCI 330 Nov. 2nd, 2017

Lecture 17: Amortized Analysis

Lecturer: Rong Ge *Scribe: Xingyu Chen*

1 Overview

When designing algorithms, there might be operations that are repeated multiple times. In certain situations this operation might be very fast, and in other situations the operation might be very slow.

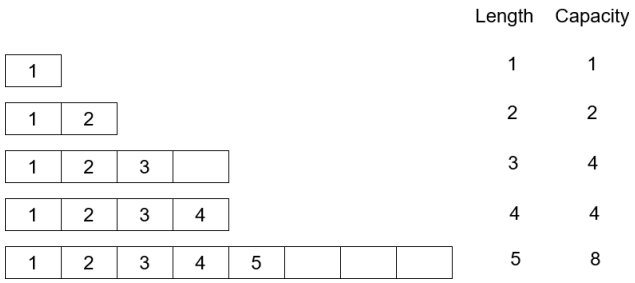
In such cases, it is often too crude to estimate the total time spend by these operations using the number of operations multiplied by the worst-case running time for one operation. This is because maybe only a small fraction of the operations take a long time.

Amortized analysis tries to analyze the cost of the operations more carefully. In this lecture we will look at an example of Dynamic Array.

2 Dynamic Array

2.1 Definition

A dynamic array is a random access, variable-size list data structure that allows elements to be added to the end of the array. At any time, the amount of memory should be proportional to the length of the array.
The initial maximum capacity of the array is 1. When the number of elements added is over its maximum capacity, the capacity of the array must be doubled. See the following Figure for graphical explanation.



2.2 Operation

The dynamic array needs to support one operation ”Add”, which takes a number and add this number to the dynamic array.

However, this ”Add” operation can cost different running time.

Definition 1.

- *Define the operation as ”Heavy” if before this ”Add” operation, the number of elements inside the array reaches its maximum capacity.*

- *Define the operation as "Light" if before this "Add" operation, the number of elements inside the array doesn't reach its maximum capacity.*

If the operation is light, it only costs O(1) time since we just need to put this element into an empty slot.

If the operation is heavy, it costs 2^{k+1} time assuming the current maximum capacity is of size 2^k. This is because it needs to allocate an array of size 2^{k+1}, copy first 2^k elements into the new array, and put the 2^k+1th element into the new array.

3 Aggregate Method

The idea of the aggregate method is to compute the total cost of n operations, then divide the total cost by n. This is the analysis technique used to analyze DFS, Dynamic Array and various other problems.

3.1 Analysis

Let t_i be the running time of the ith add operation.

$$\begin{aligned}
\text{Total Running Time} &= \sum_{i=1}^{n} t_i \\
&= \sum_{i=1, i\neq 2^k+1}^{n} t_i + \sum_{k, 2^k+1\leq n} t_{2^k+1} \\
&= \sum_{i=1}^{n} 1 + \sum_{k, 2^k+1\leq n} 2^{k+1} \\
&\leq n + (2+4+8+...+2^{l+1}) \\
&= n + 2^{l+2} - 2 \\
&\leq n + 4n = 5n
\end{aligned}$$

where l is the largest number such that $2^l + 1 \leq n$.
The last inequality is because

$$\begin{aligned}
2^l + 1 &\leq n \\
2 \cdot (2^l + 1) &\leq 2n \\
2^{l+1} &\leq 2n
\end{aligned}$$

So the amortized time is equal to the total time divided by n. Then the amortized time for "Add" operation is 5, which is equal to $O(1)$.

4 Charging Argument

Charging Method is also called Accounting Method. Image you have a bank account (of running time) which saves time in order to pay for the more expensive operations

The major step is to design a way of charging the expensive operations to the normal operations and make sure the bank account always have money to pay in the future.

4.1 Analysis

Between two heavy operations 2^k+1th operation and $2^{k+1}+1$th operation, we have $2^{k+1}+1-(2^k+1)-1=2^k-1$ light operations. The cost for the $2^{k+1}+1$th operation is 2^{k+2}. So the average time we need to save is

$$\frac{2^{k+2}}{2^k-1} \approx 4$$

What we should to is to save 4 units of time for each light operation. Then when $(2^{k+1}+1)th$ operation happens, I have saved $(2^k-1)\cdot 4 = 2^{k+2}-4$ units of time in my account. So the additional time I need to pay now is just 4. In summary,

- For Light operations: Pay 1, Save 4 (In total, I cost 1+4 =5)
- For Heavy operations: Use all savings, Pay 4

In total, in each step, I won't pay more than 5. So the amortized cost for "Add" operation is $O(1)$.