
Reading: GT 1.5, p. 34–41 [up to 50 minutes]

It is most important to understand how Clearable Table is analyzed, how extensible array works, and how its analysis goes through. It is super useful to learn *The accounting method*, but you can safely skip the *Potential functions* on p.37–38, unless you get really excited by the topic.

Ingredients: Clearable Table, Extensible Array, Dynamic Hashing.

Rather than focusing on each operation separately, amortization considers the interaction between all the operations by studying the running time of series of these operations.

Clearable Table Data Structure

Operations

ADD(e) – Add the element e to the next available cell in the table.

CLEAR() – Empty the table by removing all its elements.

If we use an array of N cells, then clearing a table with n elements lasts $\Theta(n)$.

Consider a series of n operations on an initially empty clearable array. It is easy to see a bound of $O(n^2)$ on all these operations together, since the worst case of CLEAR is $O(n)$ and there may be n CLEAR operations in a series of n any operations in the worst-case.

But wait ... n clear operations only last $O(n)$ time in total. Why? In fact:

Theorem 1 (1.30 p. 34). *A series of n operations on an initially empty table implemented with an array takes $O(n)$ time in total.*

Proof. Let M_0, \dots, M_{n-1} be the n operations.

Let $M_{i_0}, \dots, M_{i_{k-1}}$ be the k CLEAR operations.

The j th CLEAR can only remove $i_j - i_{j-1} - 1$ elements so it runs in $O(i_j - i_{j-1})$.

The running time of all CLEAR calls in total is:

$$O(i_0 + \sum_{j=1}^{k-1} (i_j - i_{j-1})) = i_0 + i_1 - i_0 + i_2 - i_1 + \dots + i_{k-1} - i_{k-2} = i_{k-1} \quad (1)$$

(all terms, except for i_{k-1} cancel each other). But i_{k-1} is $O(n)$ in the worst case, so all CLEAR take at most linear time together.

Then all ADD also take linear time together (each takes $O(1)$ in the worst-case).

So all operations take $O(n)$ in total.

We say that each operations takes *amortized constant time* since $\frac{O(n)}{n} = O(1)$. \square

Extensible Array

Our table is that it only supports a fixed number of elements (N) :

To overcome this we would like to grow it, whenever $n = N$.

How do we “grow” an array A ?

1. Allocate a new array B of capacity $2N$.
2. For all $i = 0, \dots, N - 1$ copy $A[i]$ to $B[i]$.
3. $A \leftarrow B$ (whatever was in A previously is now to be garbage collected).

Performing a single growing step takes $\Theta(n)$ time. Why?

Intuitively this is not a lot, since it creates a space for another n elements.

Theorem 2 (1.31 p. 40). *Let A an extensible array A . The total time to perform a series of n ADD operations in an initially empty S is $O(n)$.*

Proof. Remember that we want to just count basic operations when we do complexity analysis.

In order to prove this theorem we need to argue that we only spend a constant amount of these operations per single element (or $O(n)$ operations in total).

Note that growing an array from k to size $2k$ requires $2k$ operations (for allocation, initialization and copying).

We will show that “we pay” only 3 basic operations per element, regardless of how many times the array needs to grow (and regardless of how large it is).

Let us just count the cost of growing the array.

Consider a situation just after doubling: $n = \frac{N}{2}$, so we have just spent $2n$ operations on doubling. But previous doubling happened at $\frac{n}{2}$ at cost n and so on. So the total cost of growing is:

$$\begin{aligned} 2n + n + \frac{n}{2} + \dots + 2 + 1 &= 2^{\log 2n} + 2^{\log n} + 2^{\log \frac{n}{2}} + \dots + 2^1 + 2^0 = \\ &= 2^{(\log n)+1} + 2^{\log n} + 2^{(\log n)-1} + \dots + 2^0 = \sum_{i=0}^{(\log n)+1} 2^i \stackrel{(1)}{=} \frac{1 - 2^{(\log n)+2}}{1 - 2} = \\ &= \frac{1 - 2^2 \cdot 2^{\log n}}{-1} = \frac{1 - 4 \cdot n}{-1} = 4n - 1 = O(n) . \quad (2) \end{aligned}$$

So we have spent only $O(n)$ time in total on growing the table up to n elements.

We have of course also spent $O(n)$ time on inserting these elements without growing (a constant time per each). So the total time per n ADD operations is $O(n)$.

Conclusion: each ADD operation runs in *amortized constant time*. □

Self study: Carefully study the argument in Section 1.5.2 p. 40. This is a different proof than I have presented, and it might be more understandable to you. The accounting technique that they use is general and a popular one.

Dynamic Hashing

If a hash table is full (for example $n = N$), how much time does not take to double it?

1. Allocate a new hash table T' of size $2N$
2. Rehash all the elements of T into T'
3. Deallocate T

This is the same asymptotic cost as for growing an extensible array. So the hash table can be grown cheaply, since the cost of growing is amortized.

In order to keep the number of collisions low, one doubles a hash-table at $n = N/2$ if memory is not an issue.

¹This sum is a sum of geometric series. Do you know that $\sum_{i=0}^k r^i = \frac{1-r^{k+1}}{1-r}$?