

1 Announcements

- Start reading Chapter 2. Section 2.6 is only necessary for CSC401 students.
- Remember Homework 1 and Practicum Problem 1 have been assigned.
- CSC401 accounts have been created, so try logging in as soon as you can to see if they are there. We will cover (quickly) submission instructions on Thursday.

2 Amortization

There will be many tricks we shall learn to more effectively and accurately evaluate the asymptotic running times of our algorithms. One scheme is *amortization* whereby we spread the cost of an action over multiple (previous) executions. The idea is to show that the total running time (cost) at the end is below some bounded value.

The best way to see this is with a simple example. Let us therefore look at the simple Clearable Table Data Structure. Here we have two main functions:

- `add(e)`: Add e to end of the vector.
- `clear()`: Remove (and delete) all elements from the vector.

Let us for now assume that the array can hold some maximum large amount N , so we don't worry about resizing the array. If we implement this as an array with an index pointer, then one should be able to see that the `add()` operation takes $\Theta(1)$, constant, time, and that the `clear()` operation can take up to $\Theta(m)$, linear, time where m is the number of elements currently in the array. The linear time is because not only must we empty the table but we should also free (delete) the elements inside the table, or possibly do something with them.

Let us now assume we have a sequence of n operations on an empty table. We don't know what these operations are, clear or add, so from a worst-case point of view they could nearly all be the costly clear operation. Since each clear operation takes $O(n)$ time and there are $O(n)$ clear operations, the time would be $O(n^2)$. This is certainly a correct analysis. The running time is $O(n^2)$ but it is far from the tightest possible bound we can achieve. If we really look into this problem we will see that the total running time is in fact $\Theta(n)$!

Let M_i represent the i -th operation, 0 to $n - 1$. Let M_{i_j} represent the j -th clear operation, 0 to $k - 1$. Thus we have the sequence:

M_0	M_1	M_2	M_{i_0}	M_4	M_5	M_{i_1}	M_7	M_8	M_9	M_{i_2}	M_{11}	M_{12}
a	a	a	c	a	a	c	a	a	a	c	a	a

Notice that the running time of a clear operation is not (in this case) $\Theta(n)$ but is actually the number of elements in the *table* at the time. What is this value? For operation M_{i_j} , the size is $i_j - i_{j-1} - 1$, thus the cost of a clear operation is actually $O(i_j - i_{j-1})$. We now compute the sum over all clear operations and get a telescoping sum (a math trick!), so T_{clear} is

$$\sum_{j=0}^{k-1} O(i_j - i_{j-1}) = O(i_{k-1} - i_{-1}) = O(n).$$

Notice that $i_{-1} = 0$, as a special case. Think about the cost to do the first operation M_{i_0} . The remaining time T_{add} is the number of add operations which is also $O(n)$, since there are $n - k$ add operations and each one takes $O(1)$ time. Thus, the total time $T = T_{add} + T_{clear}$ is $O(n)$. This means that the *average* time per operation is $O(1)$, even though some operations may take as much as $O(n)$. This important observation leads to a better (though initially confusing) analysis scheme. There are actually several techniques available, but I'll demonstrate my favorite (the accounting method) as well as the other main one from the book (potential functions).

The **amortized running time** of a single operation within a series of operations is the worst-case running time of the series divided by the number of operations. That is, it is the *average* running time per operation.

2.1 Accounting Method

Here we treat operations as currency costing some amount of cyber-dollars. Here are some things to consider when using accounting to do amortization:

- The dollars are stored in (one or more) accounts, scattered in the data structure. For example, each entry in our array can store cyber dollars.
- Execution of (a constant number of) primitive operations **costs** a constant number of cyber dollars. Usually we say just one, since constants generally do not matter. For example, inserting one element into, accessing an element in, or deleting an element from the array each cost one cyber dollar.
- Execution of an amortized operation, the ones that we want to compute the average time, **deposit** cyber dollars into the account. They deposit into the account by being **charged** cyber dollars.
- When an operation is performed it is done by either charging another operation or withdrawing money from an account, thus paying for the operation.
- We never want our accounts to go below zero, in the red, but we may finish with extra cash, in the black.
- The amortized cost of each operation is the maximum number of dollars charged to any amortized operation.

To illustrate this. We charge two cyber-dollars to each of our two main operations, **add** and **clear**. Notice that the add operation actually only requires one cyber dollar to execute whereas the clear operation may require many more than two cyber dollars to execute. So, how do we pay for the actual operations in clear?

When an add operation is performed, it is charged two cyber-dollars, one to pay for the cost of performing the (constant number of) primitive operations it performs. The second dollar is deposited into the account at the entry created. When a clear operation is performed, the two cyber dollars charged are used to pay for the act of checking if there is anything to delete (which actually only costs one cyber dollar). Then for each succeeding deletion, we use the cyber dollar stored in each entry's account to pay for the actual deletion of the entry. Since each add operation placed a dollar in each entry's account and once an entry's account is spent, the entry is removed, we never go below zero and we are able to pay for each clear operation. Thus, the amortized cost of each operation is $O(1)$, two cyber-dollars.

Note: It is very important to understand that this is an analysis technique. We are actually *not* altering the data structure itself. We are simply analyzing the performance by using a hypothetical account associated with the data structure.

2.2 Potential Function

The potential function is just a more advanced means of using the telescoping (collapsing) sum property. In this method, we view the data structure as storing energy (potential) Φ in a system. The important points to remember about the potential function approach are:

- Φ represents the potential energy of the entire system.
- Φ_i represents the potential energy after operation i ,
- We set $\Phi_0 \geq 0$ as the initial energy in the system.
- Each operation (e.g. add/clear) contributes some energy to the system, this is the amortized cost of the operation, as well as removing some energy from the system to compensate for the energy used to perform the operation itself.
- We determine this cost by analyzing the difference in energy in the system from time $i - 1$ to i . That is, we are concerned with $\Phi_i - \Phi_{i-1}$.
- Let t_i represent the actual time taken on an operation (add/clear).
- Let t'_i represent the *amortized* time for an operation (add/clear)
- $t'_i = t_i + \Phi_i - \Phi_{i-1}$.
- Let $T = \sum_{i=1}^n t$ be the actual time taken by all n operations.
- Let $T' = \sum_{i=1}^n t'_i$ be the amortized time taken by all n operations.
- Using telescoping sums we see that $T = T' + \Phi_0 - \Phi_n$.
- Want to be sure $\Phi_n \geq \Phi_0$, so that $T \leq T'$. The true running time is less than the amortized running time.

To illustrate, we reuse the example.

- Potential Φ is the number of elements in the table.
- $\Phi_0 = 0$ and $0 \leq \Phi_n \leq n$ (at most that many could be in array).
- Claim: $t'_i = 2$
- **add:**
 - $\Phi_i = \Phi_{i-1} + 1$, since table increases by one.
 - Time taken t_i is 1.
 - $t'_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 1 = 2$.
- **clear:**
 - $\Phi_i = 0$, since table becomes empty.
 - Time taken t_i is size of table plus a few (2) more units of time:
 $t_i = \Phi_{i-1} + 2$.
 Note, the size of the table is the potential energy. We could use any constant actually, but two is just cleaner in the end.
 - $t'_i = t_i + \Phi_i - \Phi_{i-1} = \Phi_{i-1} + 2 + 0 - \Phi_{i-1} = 2$.
- Since $\Phi_n \geq \Phi_0$, we know $T \leq T'$ which is $O(n)$.

2.3 Important Reading!

The book gives another analysis using the extendable array. An array that grows as the number of elements exceeds its bounds. It illustrates cyber-dollars again as well as a lower bound proof for why only adding a constant amount per step fails to do well.