Lecture 20: Amortized Analysis

The claim that hash tables have *O*(1) expected performance for lookup and insert is based on the assumption that the number of elements stored in the table is comparable to the number of buckets. If a hash table has many more elements than buckets, the number of elements stored at each bucket will become large. For instance, with a constant number of buckets and *O*(*n*) elements, the lookup time is *O*(*n*) and not *O*(1).

The solution to this problem is to increase the size of the table when the number of elements in the table gets too large compared to the size of the table. If we let the ***load factor*** be the ratio of the number of elements to the number of buckets, when the load factor exceeds some small constant (typically 2 for chaining and .75 for linear probing), we allocate a new table, typically double the size of the old table, and rehash all the elements into the new table. This operation is not constant time, but rather linear in the number of elements at the time the table is grown.

The linear running time of a resizing operation is not as much of a problem as it might sound (although it can be an issue for some real-time computing systems). If the table is doubled in size every time it is needed, then the resizing operation occurs with exponentially decreasing frequency. As a consequence, the insertion of n elements into an empty array takes only $O(n)$ time in all, including the cost of resizing. We say that the insertion operation has $O(1)$ ***amortized run time*** because the time required to insert an element is $O(1)$ *on average*, even though some elements trigger a lengthy rehashing of all the elements of the hash table.

It is crucial that the array size grow geometrically (doubling). It might be tempting to grow the array by a fixed increment (e.g., 100 elements at time), but this results in asymptotic linear rather than constant amortized running time.

Now we turn to a more detailed description of amortized analysis.

Amortized Analysis

Amortized analysis is a worst-case analysis of a a *sequence* of operations — to obtain a tighter bound on the overall or average cost per operation in the sequence than is obtained by separately analyzing each operation in the sequence. For instance, when we considered the union and find operations for the disjoint set data abstraction earlier in the semester, we were able to bound the running time of individual operations by *O*(log *n*). However, for a sequence of *n* operations, it is possible to obtain tighter than an *O*(*n* log *n*) bound (although that analysis is more appropriate to 4820 than to this course). Here we will consider a simplified version of the hash table problem above, and show that a sequence of *n* insert operations has overall time *O*(*n*).

There are three main techniques used for amortized analysis:

- The ***aggregate method***, where the total running time for a sequence of operations is analyzed.
- The ***accounting*** (or ***banker's***) method, where we impose an extra charge on inexpensive operations and use it to pay for expensive operations later on.
- The ***potential*** (or ***physicist's***) method, in which we derive a *potential function* characterizing the amount of extra work we can do in each step. This potential either increases or decreases with each successive operation, but cannot be negative.

Consider an extensible array that can store an arbitrary number of integers, like an `ArrayList` or `Vector` in Java. These are implemented in terms of ordinary (non-extensible) arrays. Each `add` operation inserts a new element after all the elements previously inserted. If there are no empty cells left, a new array of double the size is allocated, and all the data from the old array is copied to the corresponding entries in the new array. For instance, consider the following sequence of insertions, starting with an array of length 1:

```
           +--+
Insert 11  |11|
           +--+
           +--+--+
Insert 12  |11|12|
           +--+--+
           +--+--+--+--+
Insert 13  |11|12|13|  |
           +--+--+--+--+
           +--+--+--+--+
Insert 14  |11|12|13|14|
           +--+--+--+--+
           +--+--+--+--+--+--+--+--+
Insert 15  |11|12|13|14|15|  |  |  |
           +--+--+--+--+--+--+--+--+
```

The table is doubled in the second, third, and fifth steps. As each insertion takes $O(n)$ time in the worst case, a simple analysis would yield a bound of $O(n^2)$ time for n insertions. But it is not this bad. Let's analyze a sequence of n operations using the three methods.

Aggregate Method

Let c_i be the cost of the i-th insertion:

```
c_i = i if i-1 is a power of 2
     1 otherwise
```

Let's consider the size of the table s_i and the cost c_i for the first few insertions in a sequence:

```
i    1  2  3  4  5  6  7  8  9 10
s_i  1  2  4  4  8  8  8  8 16 16
c_i  1  2  3  1  5  1  1  1  9  1
```

Alteratively we can see that c_i=1+d_i where d_i is the cost of doubling the table size. That is

```
d_i = i-1 if i-1 is a power of 2
     0 otherwise
```

Then summing over the entire sequence, all the 1's sum to $O(n)$, and all the d_i also sum to $O(n)$. That is,

$$\Sigma_{1\le i\le n}\ c_i \quad \le \quad n\ +\ \Sigma_{0\le j\le m}\ 2^{j-1}$$

where $m = \log(n - 1)$. Both terms on the right hand side of the inequality are $O(n)$, so the total running time of n insertions is $O(n)$.

Accounting (Banker's) Method

The aggregate method directly seeks a bound on the overall running time of a sequence of operations. In contrast, the accounting method seeks to find a *payment* of a number of extra time units charged to each individual operation such that the sum of the payments is an upper bound on the total actual cost. Intuitively, one can think of maintaining a bank account. Low-cost operations are charged a little bit more than their true cost, and the surplus is deposited into the bank account for later use. High-cost operations can then be charged less than their true cost, and the deficit is paid for by the savings in the bank account. In that way we spread the cost of high-cost operations over the entire sequence. The charges to each operation must be set large enough that the balance in the bank account always remains positive, but small enough that no one operation is charged significantly more than its actual cost.

We emphasize that the extra time charged to an operation does not mean that the operation really takes that much time. It is just a method of accounting that makes the analysis easier.

If we let c'_i be the charge for the i-th operation and c_i be the true cost, then we would like

$$\Sigma_{1≤i≤n}\, c_i ≤ \Sigma_{1≤i≤n}\, c'_i$$

for all n, which says that the ***amortized time*** $\Sigma_{1≤i≤n}\, c'_i$ for that sequence of n operations is a bound on the true time $\Sigma_{1≤i≤n}\, c_i$.

Back to the example of the extensible array. Say it costs 1 unit to insert an element and 1 unit to move it when the table is doubled. Clearly a charge of 1 unit per insertion is not enough, because there is nothing left over to pay for the moving. A charge of 2 units per insertion again is not enough, but a charge of 3 appears to be:

i	1	2	3	4	5	6	7	8	9	10
s_i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
c'_i	3	3	3	3	3	3	3	3	3	3
b_i	2	3	3	5	3	5	7	9	3	4

where b_i is the balance after the i-th insertion.

In fact, this is enough in general. Let m refer to the m-th element inserted. The three units charged to m are spent as follows:

- One unit is used to insert m immediately into the table.
- One unit is used to move m the first time the table is grown after m is inserted.
- One unit is donated to element $m − 2^k$, where 2^k is the largest power of 2 not exceeding m, and is used to move that element the first time the table is grown after m is inserted.

Now whenever an element is moved, the move is already paid for. The first time an element is moved, it is paid for by one of its own time units that was charged to it when it was inserted; and all subsequent moves are paid for by donations from elements inserted later.

In fact, we can do slightly better, by charging just 1 for the first insertion and then 3 for each insertion after that, because for the first insertion there are no elements to copy. This will yield a zero balance after the first insertion and then a positive one thereafter.

Potential (Physicist's) Method

Above we saw the aggregate method and the banker's method for dealing with extensible arrays. Now let us look at the physicist's method.

Suppose we can define a ***potential function*** Φ (read "Phi") on states of a data structure with the following properties:

- $\Phi(h_0) = 0$, where h_0 is the initial state of the data structure.
- $\Phi(h_t) \geq 0$ for all states h_t of the data structure occurring during the course of the computation.

Intuitively, the potential function will keep track of the precharged time at any point in the computation. It measures how much saved-up time is available to pay for expensive operations. It is analogous to the bank balance in the banker's method. But interestingly, it depends only on the current state of the data structure, irrespective of the history of the computation that got it into that state.

We then define the ***amortized time*** of an operation as

$$c + \Phi(h') - \Phi(h),$$

where c is the actual cost of the operation and h and h' are the states of the data structure before and after the operation, respectively. Thus the amortized time is the actual time plus the change in potential. Ideally, Φ should be defined so that the amortized time of each operation is small. Thus the change in potential should be positive for low-cost operations and negative for high-cost operations.

Now consider a sequence of n operations taking actual times $c_0, c_1, c_2, ..., c_{n-1}$ and producing data structures $h_1, h_2, ..., h_n$ starting from h_0. The total amortized time is the sum of the individual amortized times:

$$\begin{aligned}&(c_0 + \Phi(h_1) - \Phi(h_0)) + (c_1 + \Phi(h_2) - \Phi(h_1)) + ... + (c_{n-1} + \Phi(h_n) - \Phi(h_{n-1}))\\&= c_0 + c_1 + ... + c_{n-1} + \Phi(h_n) - \Phi(h_0)\\&= c_0 + c_1 + ... + c_{n-1} + \Phi(h_n).\end{aligned}$$

Therefore the amortized time for a sequence of operations overestimates of the actual time by $\Phi(h_n)$, which by assumption is always positive. Thus the total amortized time is always an upper bound on the actual time.

For dynamically resizable arrays with resizing by doubling, we can use the potential function

$$\Phi(h) = 2n - m,$$

where n is the current number of elements and m is the current length of the array. If we start with an array of length 0 and allocate an array of length 1 when the first element is added, and thereafter double the array size whenever we need more space, we have $\Phi(h_0) = 0$ and $\Phi(h_t) \geq 0$ for all t. The latter inequality holds because the number of elements is always at least half the size of the array.

Now we would like to show that adding an element takes amortized constant time. There are two cases.

- If $n < m$, then the actual cost is 1, n increases by 1, and m does not change. Then the potential increases by 2, so the amortized time is 1 + 2 = 3.
- If $n = m$, then the array is doubled, so the actual time is $n + 1$. But the potential drops from n to 2, so amortized time is $n + 1 + (2 - n) = 3$.

In both cases, the amortized time is O(1).

The key to amortized analysis with the physicist's method is to define the right potential function. The potential function needs to save up enough time to be used later when it is needed. But it cannot save so much time that it causes the amortized time of the current operation to be too high.