CS 561, Lecture 17

Jared Saia
University of New Mexico

Today's Outline

- Potential Method
- Dynamic Tables

Potential Method

- The most powerful method (and hardest to use)
- Builds on the idea from physics of potential energy
- Instead of associating taxes with particular operations, represent prepaid work as a *potential* that can be spent on later operations
- Potential is a function of the entire data structure

Potential Function

- Let D_i denote our data structure after i operations
- Let Φ_i denote the potential of D_i
- Let c_i denote the cost of the i-th operation (this changes D_{i-1} into D_i)
- Then the amortized cost of the i-th operation, a_i, is defined to be the actual cost plus the change in potential:

$$\boxed{a_i = c_i + \Phi_i - \Phi_{i-1}}$$

Potential Method

- So the *total* amortized cost of n operations is the actual cost plus the change in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

Potential Method

- Our task is to define a potential function so that
  1. $\Phi_0 = 0$
  2. $\Phi_i \geq 0$ for all i
- If we do this, the total *actual* cost of any sequence of operations will be less than the total amortized cost

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

Binary Counter Example

- For the binary counter, we can define the potential Φ_i after the i-th *Increment* operation to be the number of bits with value 1
- Initially all bits are 0 so $\Phi_0 = 0$, further $\Phi_i \geq 0$ for all $i > 0$, so this is a legal potential function

Binary Counter

- We can describe both the actual cost of an Increment and the change in potential in terms of the number of bits set to 1 and reset to 0.

$$c_i = \#\text{bits flipped from 0 to 1} + \#\text{bits flipped 1 to 0}$$
$$\Phi_i - \Phi_{i-1} = \#\text{bits flipped from 0 to 1} - \#\text{bits flipped 1 to 0}$$

- Thus, the amortized cost of the ith Increment is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits flipped from 0 to 1}$$

Binary Counter

- Since Increment only changes one bit from a 0 to a 1, the amortized cost of Increment is 2 (using this potential function)
- Recall that for a legal potential function, $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$ thus the total cost for n call to increment is no more than $2n$
- (Same as saying that the amortized cost is 2)

Potential Method Recipe

1. Define a potential function for the data structure that is 1) initially equal to zero and 2) is always nonnegative.
2. The amortized cost of an operation is its actual cost plus the change in potential.

Binary Counter Example

- For the binary counter, the potential was exactly the total unspent taxes paid using the taxation method
- So it gave us the same amortized bound
- In general, however, there may be no way of interpreting the potential as "taxes"

A Good Potential Function

- Different potential functions lead to different amortized time bounds
- Trick to using the method is to get the best possible potential function
- A good potential function goes up a little during any cheap/fast operation and goes down a lot during any expensive/slow operation
- Unfortunately, there's no general technique for doing this other than trying lots of possibilities

Stack Example

- Consider again a stack with Multipop
- Define the potential function Φ on the stack to be the number of objects on the stack
- This potential function is "legal" since $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all $i > 0$

Push

- Let's now compute the costs of the different stack operations on a stack with s items
- If the i-th operation on the stack is a push operation on a stack containing s objects, then

$$\Phi_i - \Phi_{i-1} = (s+1) - s = 1$$

- So $a_i = c_i + 1 = 2$

Multipop

- Let the i-th operation be Multipop(S,k) and let $k' = \min(k, s)$ be the number of objects popped off the stack. Then

$$\Phi_i - \Phi_{i-1} = (s - k') - s = -k'.$$

- Further $c_i = k'$.
- Thus,

$$a_i = -k' + k' = 0$$

- (We can show similarly that the amortized cost of a pop operation is 0)

Wrapup

- The amortized cost of each of these three operations is $O(1)$
- Thus the worst case cost of n operations is $O(n)$

Dynamic Tables

- Consider the situation where we do not know in advance the number of items that will be stored in a table, but we want constant time access
- We might allocate a fixed amount of space for the table only to find out later that this was not enough space
- In this case, we need to copy over all objects stored in the original table into a new larger table
- Similarly, if many objects are deleted, we might want to reduce the size of the table

Dynamic Tables

- The data structure that we want is a Dynamic Table (aka Dynamic Array)
- We can show using amortized analysis that the amortized cost of an insertion and deletion into a Dynamic Table is $O(1)$ even though worst case cost may be much larger

Load Factor

- For a nonempty table T, we define the "load factor" of T, $\alpha(T)$, to be the number of items stored in the table divided by the size (number of slots) of the table
- We assign an empty table (one with no items) size 0 and load factor of 1
- Note that the load factor of any table is always between 0 and 1
- Further if we can say that the load factor of a table is always at least some constant c, then the unused space in the table is never more than $1 - c$

Table Expansion

- Assume that the table is allocated as an array
- A table is full when all slots are used i.e. when the load factor is 1
- When an insert occurs when the table is full, we need to expand the table
- The way we will do this is to allocate an array which is twice the size of the old array and then copy all the elements of the old array into this new larger array
- If only insertions are performed, this ensures that the load factor is always at least 1/2

Pseudocode

```
Table-Insert(T,x){
  if (T.size == 0){allocate T with 1 slot;T.size=1}
  if (T.num == T.size){
    allocate newTable with 2*T.size slots;
    insert all items in T.table into newTable;
    T.table = newTable;
    T.size = 2*T.size
    }
  T.table[T.num] = x;
  T.num++
}
```


Amortized Analysis

- Note that usually Table-Insert just does an "elementary" insert into the array
- However very occasionally it will do an "expansion". We will say that the cost of an expansion is equal to the size before the expansio occurs
- (This is the cost of moving over all the old elements to the larger table)

Aggregate Analysis

- Let c_i be the cost of the i-th call to Table-Insert.
- If $i-1$ is an exact power of 2, then we'll need to do an expansion and so $c_i = i$
- Otherwise, $c_i = 1$
- The total cost of n Table-Insert operations is thus

$$\begin{aligned}
\sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j && (1)\\
&< n + 2^{\lfloor \log n \rfloor + 1} && (2)\\
&= n + 2 * 2^{\lfloor \log n \rfloor} && (3)\\
&\leq n + 2n && (4)\\
&= 3n && (5)
\end{aligned}$$

- Thus the amortized cost of a single operation is 3

Taxation method

- Every time Table-Insert is called, we tax the operation 3 dollars
- Intuitively, the item inserted pays for:
  1. its insertion
  2. moving itself when the table is eventually expanded
  3. moving some other item that has already been moved once when the table is expanded

Taxation Method

- Suppose that the size of the table is m right after an expansion
- Then the number of items in the table is $m/2$
- Each time Table-Insert is called, we tax the operation 3 dollars:
  1. One dollar is used immediately to pay for the elementary insert
  2. Another dollar is stored with the item that is inserted
  3. The third dollar is placed as credit on one of the $m/2$ items already in the table

Taxation Method

- Filling the table again requires $m/2$ total calls to Table-Insert
- Thus by the time the table is full and we do another expansion, each item will have one dollar of credit on it
- This dollar of credit can be used to pay for the movement of that item during the expansion

Potential Method

- Let's now analyze Table-Insert using the potential method
- Let num_i be the num value for the i-th call to Table-Insert
- Let $size_i$ be the size value for the i-th call to Table-Insert
- Then let

$$\Phi_i = 2 * num_i - size_i$$

In Class Exercise

Recall that $a_i = c_i + \Phi_i - \Phi_{i-1}$

- Show that this potential function is 0 initially and always nonnegative
- Compute a_i for the case where Table-Insert does not trigger an expansion
- Compute a_i for the case where Table-Insert does trigger an expansion (note that $num_{i-1} = num_i - 1$, $size_{i-1} = num_i - 1$, $size_i = 2 * (num_i - 1)$)