Amortized Analysis

We use amortized analysis to compute the average time of algorithms that have occasional operations that have very slow operations, but we know that the average time is guaranteed to be faster than the worst case time: **Hash table, disjoint sets, display trees.**

Generally, when during amortized analysis computation, we start with an empty data structure, and get average time from the very beginning. This ensures that you don't start right before an expensive operation, which may skew your average.

1 Averaging method:

We are using a hash table example in this computation.

Suppose newly created hash table only has one bucket N buckets and n items; suppose insert doubles size of table (before adding new item) if n = N. This keeps the load factor (load factor is n/N) $\leq$1.

Notes: **Resizing** takes$\leq$ 2n seconds (creates an array that is double the size of the current array, n)

Our current situation:

Construct an empty hash table perform i insert operations:
n (# items in table) = i
N (# of buckets) is smallest power of 2 that is $\leq$ n
S (Total seconds for all resizing ops) is: 2 + 4 + 8 + ... + N/4 + N/2 + N = 2N – 2
Total cost of inserts is $\leq$ i (total # of inserts) + 2N – 2 (resizing time)

> Note: We do this to put upper bound on N. We know that N < 2n (which = 2i), so total cost after substituting N is i + 2(2i) – 2 = O(5i-2) time. After we get rid of the constants, this becomes O(i) time

The **average** time of insert O(i)/i = O(1)
The **amortized** running time of insert ϵ of O(1)
worst case time ϵ Θ(n)

2 Accounting Method

The accounting method is more powerful than averaging method. Suppose we have a hash table where we are able to insert() and remove(). The averaging method won't work if our hash table has deletions (resize the hash table downwards to save memory).

Unit of time: dollar (unit of time to execute slowest constant-time computation)

Each operation has:

* an amortized cost: # of dollars that are "charged" to do the operation
* actual cost: actual # of O(1) – time ops that operation performs

Invariant: never overdraw your account. The bank balance remain > 0.

> Notes: Actual costs vary a lot (may have few rare extremely expensive operations... use amortized cost to smooth out costs over a large sequence of operations

During the operation:

* If amortized cost $>$ actual cost, our extra dollars stored in "bank" to be spent on later (more expensive) operations
* If actual cost $>$ amortized cost, dollars withdrawn from bank to pay for it

In our hash table model:

Every operation costs \$1 of actual time unless the hash table is resized

\$5 to Insert(): doubles size if the load factor > 1

\$5 to Remove(): halves the size if load factor would be $<.25$.

\$1 to Find() (no resizing necessary)

Consider hash table that is new, or just resized. We know that N buckets will be between N/2 - 1 and N/2 + 1 items

* 3 cases:

- new table: N=1, n=0
- just doubled: N/2 +1 items
- just halved: N/2 - 1 items

At any moment in time, we can look at how many buckets and items there are, and figure out a lower bound of the money that we have in our bank.

We know that every insert/remove costs \$1, and puts \$4 in the bank.

Insert: Depending on where we are, we know that there must be >= 4(|n - N/2| - 1) dollars saved...since we haven't resized within this bound. **We only resize if n reaches N+1**. At this time, there is \$2N in the bank, so we can affoard resizing, which costs \$2N.

Remove: Every remove costs \$1, and puts \$4 in the bank; we know that remove resizes the table if n **drops to N/4 - 1.** Let's say that it costs \$N (upper bound, so this is okay) to resize N to N/2 buckets. When n is N/4 -1,

we have $N in the bank, so this is also affordable.

Bottom Line —-> Bank balance never drops below 0. All of the operations (insert, remove, find)'s amoritized costs are constants. This means that the amortized running times are ϵ O(1) for all of the operations. Note that the amortized analysis only tells the **upper bound (big-oh)** bo