Amortized Analysis

- Not just consider one operation, but a sequence of operations on a given data structure.
- Average cost over a sequence of operations.
- Average case Analysis (Probabilistic Analysis):
  - Average case running time: average over all possible inputs for one algorithm (operation).
  - If using probability, called expected running time.
- Amortized analysis:
  - No involvement of probability
  - Average performance on a sequence of data structure operations, even some operations are expensive.
  - Guarantee average performance of each operation among the sequence in worst case.

Amotized Analysis

- We have a data structure
- We perform a sequence of operations
  - Operations may be of different types (e.g. Insert, delete)
  - Depending on the state of structure the actual cost of an operation may differ
- Just analysing the worst case time of a single operation may not say too much
- We want the average running time of an operation (but from the worst-case

Three Methods of Amortized Analysis

- Aggregate analysis:
  - All operations are treated equally
  - The worst case running time of a sequence of n operations is computed
  - Amortized cost = Total cost of *n* operations/*n*,
- Accounting method:
  - Assign each type of operation an (different) amortized cost
  - overcharge some operations,
  - store the overcharge as credit on specific objects,
  - then use the credit for compensation for some later operations.
- Potential method:
  - Same as accounting method
  - But store the credit as “potential energy” and as a whole.

Another example: increasing a binary counter

- Binary counter of length *k*, A[0..*k*-1] of bit array.
- INCREMENT(A)

$i \leftarrow 0$
while $i<k$ and A[i]=1
    do A[i] $\leftarrow$ 0 (flip, reset)
        $i \leftarrow i+1$
if $i<k$
then A[i] $\leftarrow$ 1 (flip, set)

Analysis of INCREMENT(A)

- Cursory analysis:
  - A single execution of INCREMENT takes $O(k)$ in the worst case (when A contains all 1s)
  - So a sequence of n executions takes $O(nk)$ in worst case (suppose initial counter is 0).
  - This bound is correct, but not tight.
- The tight bound is $O(n)$ for n executions.

Amortized (Aggregate) Analysis of INCREMENT(A)

Observation: The running time determined by #flips
but not all bits flip each time INCREMENT is called.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is never more than twice the total number of INCREMENT operations.

A[0] flips every time, total n times.
A[1] flips every other time, $n/2$ times.
A[2] flips every fourth time, $n/4$ times.
….
for i=0,1,…,k-1, A[i] flips $\lfloor n/2^i \rfloor$ times.

Thus total #flips is $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$

$$< n\sum_{i=0}^{\infty} 1/2^i$$

$$=2n.$$

Amortized Analysis of INCREMENT(A)

- Thus the worst case running time is $O(n)$ for a sequence of n INCREMENTs.
- So the amortized cost per operation is $O(1)$.

Amortized Analysis: Accounting Method

- Idea:
  - Assign different charges to different operations.
  - The amount of the charge is called amortized cost.
  - amortized cost is more or less than actual cost.
  - When amortized cost **>** actual cost, the difference is saved in specific objects as credits.
  - The credits can be used by later operations whose amortized cost **<** actual cost.
- As a comparison, in aggregate analysis, all operations have same amortized costs.

Accounting Method (cont.)

- Conditions:
  - suppose actual cost is c_i for the ith operation in the sequence, and amortized cost is c_i',
  - $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$ should hold.
    - since we want to show the average cost per operation is small using amortized cost, we need the total amortized cost is an upper bound of total actual cost.
    - holds for all sequences of operations.
  - Total credits is $\sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$, which should be nonnegative,
    - Moreover, $\sum_{i=1}^{t} c_i' - \sum_{i=1}^{t} c_i \geq 0$ for any $t>0$.

Accounting method: binary counter

- Let \$1 be the actual cost of flip of one bit).
- An amortized cost of \$2 is assigned for setting a bit to 1.
  - Amortized cost of resetting a bit is zero.
- Whenever a bit is set, use \$1 to pay the actual cost, and store another \$1 on the bit as credit.
- When a bit is reset, the stored \$1 pays the cost.
- At any point, a 1 in the counter stores \$1, the number of 1's is never negative
  - so total credit is never negative
- At most one bit is set in each operation, so the amortized cost of an operation is at most \$2.
- Thus, total amortized cost of n operations is $O(n)$, and average is $O(1)$.

The Potential Method

- Same as accounting method: something prepaid is used later.
- Different from accounting method
  - The prepaid work not as credit, but as “potential energy”, or “potential”.
  - The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

The Potential Method (cont.)

- Initial data structure D_0,
- n operations, resulting in D_0, D_1,…, D_n with costs c_1, c_2,…, c_n.
- A potential function Φ: $\{D_i\} \rightarrow$ R (real numbers)

∀ $\Phi(D_i)$ is called the potential of D_i.

- Amortized cost c_i' of the ith operation is:

  $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$. (actual cost + potential change)

  $\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$

- $= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$

The Potential Method (cont.)

- If $\Phi(D_n) \geq \Phi(D_0)$, then total amortized cost is an upper bound of total actual cost.
- But we do not know how many operations, so $\Phi(D_i) \geq \Phi(D_0)$ is required for any i.
- It is convenient to define $\Phi(D_0)=0$,and so $\Phi(D_i) \geq 0$, for all i.
- If the potential change is positive (i.e., $\Phi(D_i) - \Phi(D_{i-1})>0$), then c_i' is an overcharge (so store the increase as potential),
- otherwise, undercharge (discharge the potential to pay the actual cost).

Potential method: binary counter

- Define the potential of the counter after the *i*th INCREMENT is $\Phi(D_i) = b_i$, the number of 1's. clearly, $\Phi(D_i) \geq 0$.
- Let us compute amortized cost of an operation
  - Suppose the *i*th operation resets t_i bits.
  - Actual cost c_i of the operation is at most $t_i + 1$.
  - If $b_i=0$, then the *i*th operation resets all k bits, so $b_{i-1}=t_i=k$.
  - If $b_i>0$, then $b_i=b_{i-1}-t_i+1$
  - In either case, $b_i \leq b_{i-1}-t_i+1$.
  - So potential change is $\Phi(D_i) - \Phi(D_{i-1}) \leq b_{i-1}-t_i+1-b_{i-1}=1-t_i$.
  - So amortized cost is: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq t_i + 1 + 1 - t_i = 2$.
- The total amortized cost of n operations is $O(n)$.
- Thus worst case cost is $O(n)$.

Amortized analyses: dynamic table

- A nice use of amortized analysis
- Table-insertion, table-deletion.
- Scenario:
  - A table –maybe a hash table
  - Do not know how large in advance
  - May expend with insertion
  - May contract with deletion
  - Detailed implementation is not important
- Goal:
  - $O(1)$ amortized cost.
  - Unused space always ≤ constant fraction of allocated space.

Dynamic table

- ***Load factor*** α = *num*/*size*, where *num* = # items stored, *size* = allocated size.
- If *size* = 0, then *num* = 0. Call α = 1.
- Never allow α > 1.
- Keep α >a constant fraction → goal (2).

Dynamic table: expansion with insertion

- **Table expansion**
- Consider only insertion.
- When the table becomes full, double its size and reinsert all existing items.
- Guarantees that $\alpha \geq 1/2$.
- Each time we actually insert an item into the table, it's an ***elementary insertion***.

```
TABLE-INSERT(T, x)
if size[T] = 0
    then allocate table[T] with 1 slot
         size[T] ← 1
if num[T] = size[T]
    then allocate new-table with 2 · size[T] slots
         insert all items in table[T] into new-table      Num[t] ele. insertion
         free table[T]
         table[T] ← new-table
         size[T] ← 2 · size[T]
insert x into table[T]                                    1 ele. insertion
num[T] ← num[T] + 1
```

Initially, *num*[*T*] = *size*[*T*] = 0.

Aggregate analysis

- ***Running time:*** Charge 1 per elementary insertion. Count only elementary insertions,
- since all other costs together are constant per call.
- ci = actual cost of ith operation
  - If not full, $ci = 1$.
  - If full, have $i - 1$ items in the table at the start of the ith operation. Have to copy all $i - 1$ existing items, then insert ith item, $\Rightarrow ci = i$
- *Cursory analysis: n* operations $\Rightarrow ci = O(n) \Rightarrow O(n^2)$ time for n operations.
- Of course, we don't always expand:
  - $ci = \begin{cases} i & \text{if } i - 1 \text{ is exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$
- *So total cost* $= \sum_{i=1}^{n} ci \leq n + \sum_{i=0}^{\log(n)} 2^i \leq n + 2n = 3n$
- Therefore, **aggregate analysis** says amortized cost per operation = 3.

Accounting analysis

- Charge \$3 per insertion of *x*.
  - \$1 pays for *x*'s insertion.
  - \$1 pays for *x* to be moved in the future.
  - \$1 pays for some other item to be moved.
- Suppose we've just expanded, *size* = *m* before next expansion, *size* = 2*m* after next expansion.
- Assume that the expansion used up all the credit, so that there's no credit stored after the expansion.
- Will expand again after another *m* insertions.
- Each insertion will put \$1 on one of the *m* items that were in the table just after expansion and will put \$1 on the item inserted.
- Have \$2*m* of credit by next expansion, when there are 2*m* items to move. Just enough to pay for the expansion, with no credit left over!

Potential method

- **Potential method**

∀ $\Phi(T) = 2 \cdot num[T] - size[T]$

- Initially, $num = size = 0 \Rightarrow \Phi = 0$.
- • Just after expansion, $size = 2 \cdot num \Rightarrow \Phi = 0$.
- Just before expansion, $size = num \Rightarrow \Phi = num \Rightarrow$ have enough potential to pay for moving all items.
- Need $\Phi \geq 0$, always.
- Always have
  - $size \geq num \geq ½\, size \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi \geq 0$.

Potential method

- ***Amortized cost of ith operation:***
  - num_i = *num* after *i*th operation ,
  - $size_i$ = *size* after *i*th operation ,

    $\Phi_i = \Phi$ after *i*th operation .
- If no expansion:
  - $size_i = size_{i-1}$,
  - $num_i = num_{i-1} + 1$,
  - $ci = 1$.
- Then we have
  - $C_i' = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 3.$
- If expansion:
  - $size_i = 2size_{i-1}$,
  - $size_{i-1} = num_{i-1} = num_i - 1$,
  - $c_i = num_{i-1} + 1 = num_i$.
- Then we have
- $C_i' = c_i + \Phi_i - \Phi_{i-1} = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3$

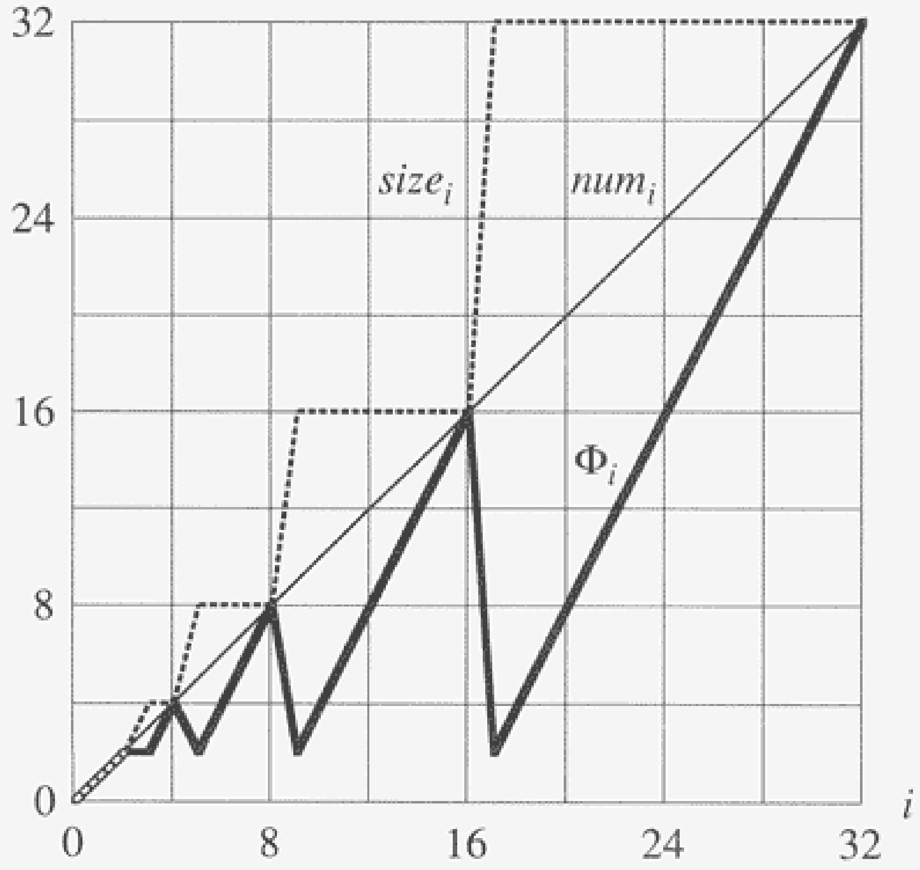

Figure 17.3 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential $\Phi_i = 2 \cdot num_i - size_i$, each being measured after the ith operation. The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i. Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterwards, the potential drops to 0, but it is immediately increased by 2 when the item that caused the expansion is inserted.

Expansion and contraction

- **Expansion and contraction**
- When α drops too low, contract the table.
  - Allocate a new, smaller one.
  - Copy all items.
- Still want
  - α bounded from below by a constant,
  - amortized cost per operation = $O(1)$.
- Measure cost in terms of elementary insertions and deletions.

Obvious strategy

- Double size when inserting into a full table (when $\alpha = 1$, so that after insertion α would become <1).
- Halve size when deletion would make table less than half full (when $\alpha = 1/2$, so that after deletion α would become >= 1/2).
- Then always have $1/2 \le \alpha \le 1$.
- Suppose we fill table.
  - Then insert $\Rightarrow$double
  - 2 deletes $\Rightarrow$halve
  - 2 inserts $\Rightarrow$double
  - 2 deletes $\Rightarrow$halve
  - . . .
  - Cost of each expansion or contraction is $\Theta(n)$, so total n operation will be $\Theta(n^2)$.
- Problem is that: Not performing enough operations after expansion or contraction to pay for the next one.

Simple solution

- Double as before: when inserting with $\alpha = 1 \Rightarrow$ after doubling, $\alpha = 1/2$.
- Halve size when deleting with $\alpha = 1/4 \Rightarrow$ after halving, $\alpha = 1/2$.
- Thus, immediately after either expansion or contraction, have $\alpha = 1/2$.
- Always have $1/4 \le \alpha \le 1$.
- ***Intuition:***
- Want to make sure that we perform enough operations between consecutive expansions/contractions to pay for the change in table size.
- Need to delete half the items before contraction.
- Need to double number of items before expansion.
- Either way, number of operations between expansions/contractions is at least a constant fraction of number of items copied.

Potential function

$\forall$ $\Phi(T) =$ $2num[T] - size[T]$ if $\alpha \geq ½$

$size[T]/2 - num[T]$ if $\alpha < ½$.

- T empty $\Rightarrow \Phi = 0$.
- $\alpha \geq 1/2 \Rightarrow num \geq 1/2size \Rightarrow 2num \geq size \Rightarrow \Phi \geq 0$.
- $\alpha < 1/2 \Rightarrow num < 1/2size \Rightarrow \Phi \geq 0$.

intuition

- measures how far from $\alpha = 1/2$ we are.
  - $\alpha = 1/2 \Rightarrow \Phi = 2num - 2num = 0$.
  - $\alpha = 1 \Rightarrow \Phi = 2num - num = num$.
  - $\alpha = 1/4 \Rightarrow \Phi = size/2 - num = 4num/2 - num = num$.
- Therefore, when we double or halve, have enough potential to pay for moving all *num* items.
- Potential increases linearly between $\alpha = 1/2$ and $\alpha = 1$, and it also increases linearly between $\alpha = 1/2$ and $\alpha = 1/4$.
- Since α has different distances to go to get to 1 or 1/4, starting from 1/2, rate of increase differs.
- For α to go from 1/2 to 1, *num* increases from *size* /2 to *size*, for a total increase of *size* /2. Φ increases from 0 to *size*. Thus, Φ needs to increase by 2 for each item inserted. That's why there's a coefficient of 2 on the $num[T]$ term in the formula for when $\alpha \geq 1/2$.
- For α to go from 1/2 to 1/4, *num* decreases from *size* /2 to *size* /4, for a total decrease of *size* /4. Φ increases from 0 to *size* /4. Thus, Φ needs to increase by 1 for each item deleted. That's why there's a coefficient of −1 on the $num[T]$ term in the formula for when $\alpha < 1/2$.

Amortized cost for each operation

- Amortized costs: more cases
  - insert, delete
  - $\alpha \geq 1/2$, $\alpha < 1/2$ (use α_i, since α can vary a lot)
  - *size* does/doesn't change

Summary

- Amortized analysis
  - Different from probabilistic analysis
- Three methods and their differences
- how to analyze