Algorithm Design and Analysis

Lecture 3

Amortized Analysis

Amortized analysis

- Main goals of the lecture:
  - *to understand what is **amortized analysis**, when is it used, and how it differs from the average-case analysis;*
  - *to be able to apply the techniques of the **aggregate analysis**, the **accounting method**, and the **potential method** to analyze operations on simple data structures.*

Beyond Worst Case Analysis

- Worst-case analysis.
  - Analyze running time as function of worst input of a given size.
- Average case analysis.
  - Analyze average running time over some distribution of inputs.
  - Ex: quicksort.
- Amortized analysis.
  - Worst-case bound on sequence of operations.
  - Ex: splay trees, union-find, Fibonacci heap.
- Competitive analysis.
  - Make quantitative statements about online algorithms.
  - Ex: paging, load balancing.

Sequence of operations

- *The problem*:
  - 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 the structure the actual cost of an operation may differ (e.g., *inserting into a sorted array*)
  - Just analyzing 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 sequence of operations*!).

Example for amortized analysis

- Stack operations:
  - PUSH(S,x), $O(1)$
  - POP(S), $O(1)$
  - MULTIPOP(S,k), min(s,k)

```
while not STACK-EMPTY(S) and $k>0$
    do POP(S)
        $k=k-1$
```

- Let us consider a sequence of n PUSH, POP, MULTIPOP.
  - The worst case cost for MULTIPOP in the sequence is $O(n)$, since the stack size is at most n.
  - thus the cost of the sequence is $O(n^2)$. Correct, but not tight.

Aggregate Analysis

- In fact, a sequence of *n* operations on an initially empty stack cost at most $O(n)$. Why?

Each object can be POP only once (including in MULTIPOP) for each time it is PUSHed. #POPs is at most #PUSHs, which is at most n.

Thus the average cost of an operation is $O(n)/n = O(1)$.

Amortized cost in aggregate analysis is defined to be average cost.

Binary counter example

- *Example data structure: a binary counter*
    - Operation: *Increment*
    - Implementation: An array of bits $A[0..k-1]$

```
Increment(A)
1 i ← 0
2 while i < k and A[i] = 1 do
3     A[i] ← 0
4     i ← i + 1
5 if i < k then A[i] ← 1
```

- *How many bit assignments do we have to do in the **worst-case** to perform Increment(A)*?
    - But usually we do much less bit assignments!

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.

Analysis of binary counter

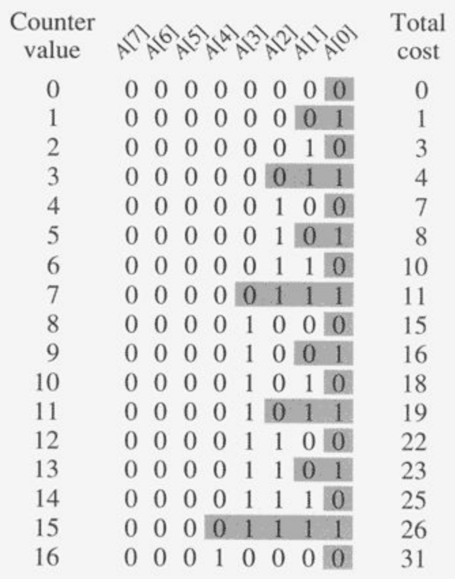
- *How many bit-assignments do we do on average*?
  - Let's consider a sequence of *n Increment's*
  - Let's compute the sum of bit assignments:
    - *A*[0] assigned on each operation: *n* assignments
    - *A*[1] assigned every two operations: *n*/2 assignments
    - *A*[2] assigned every four ops: n/4 assignments
    - *A*[*i*] assigned every 2^i ops: n/2^i assignments

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < 2n$$

- Thus, a single operation takes 2n/n = 2 = O(1) time **amortized** time

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

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

Aggregate analysis

- ***Aggregate*** *analysis – a simple way to do amortized analysis*
  - Treat all operations equally
  - Compute the *worst-case* running time of a sequence of *n* operations.
  - Divide by *n* to get an amortized running time

Three Methods of Amortized Analysis

- Aggregate analysis:
  - 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.

Amortized Analysis

The Accounting Method.

We assign different charges to different operations - sometimes more than appropriate, sometimes less.

The amount we charge = **amortized cost**.

When charged more than the actual cost, an operation will save some credit; when charged less, it will have to draw down some of the accumulated credit.

The Stack Example.

Operation	Charge
Push	2
Pop	0
Multipop	0

Amortized Analysis

- When a Push is executed, we put 2 on the plate and take 1 to pay for the operation.
- When Pop is executed, we put nothing on the plate (charge 0 to the operation) and take 1 - 0 if the stack is empty.
- When Multipop is executed, we put nothing on the plate and take 1 for each Pop operation actually executed.

Total cost over *n* operations: *O(n)*.

Each operation has cost *O(1)*.

Amortized Analysis: Accounting Method

- Idea:
  - Assign differing 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 $t>0$.

Another look at binary counter

- *Another way of looking at it (proving the amortized time)*:
  - To assign a bit, I have to use one dollar
  - When I assign "1", I use one dollar, plus I put one dollar in my "savings account" associated with that bit.
  - When I assign "0", I can do it using a dollar from the savings account on that bit
  - *How much do I have to pay for the* Increment(*A*) *for this scheme to work*?
    - Only one assignment of "1" in the algorithm. Obviously, two dollars will always pay for the operation

Amortized Analysis

The Increment Example.

Operation	Cost
Set a bit to 1	2
Set a bit to 0	0

When setting a bit to 1, put two units on the plate and withdraw one; when setting a bit to 0, withdraw a unit from the plate.

Cost of *n* operations is *O(n)*. Cost per operation *O(1)*.

Three Methods of Amortized Analysis

- Aggregate analysis:
  - 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.

Potential method

- *We can have one account associated with the whole structure*:
  - We call it a **potential**
  - It’s a function that maps a state of the data structure after operation *i* to a number: $\Phi(D_i)$

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

  - The main step of this method is defining the potential function
    - Requirement: $\Phi(D_n) - \Phi(D_0) \geq 0$
  - Once we have Φ, we can compute the amortized costs of operations

Binary counter example

- *How do we define the potential function for the binary counter?*
  - Potential of *A*: b_i – a number of "1"s
  - *What is $\Phi(D_i) - \Phi(D_{i-1})$, if the number of bits set to 0 in operation i is t_i?*
  - *What is the amortized cost of* Increment(*A*)?
    - We showed that $\Phi(D_i) - \Phi(D_{i-1}) \leq 1 - t_i$
    - Real cost $c_i = t_i + 1$
    - Thus,

$$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$

Potential method

- *We can analyze the counter even if it does not start at 0 using potential method*:
  - Let's say we start with b_0 and end with b_n "1"s
  - Observe that: $$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} c'_i - \Phi(D_n) + \Phi(D_0)$$
  - We have that: $c'_i \leq 2$
  - This means that: $\sum_{i=1}^{n} c_i \leq 2n - b_n + b_0$
  - Note that $b_0 \leq k$. This means that, if $k = O(n)$ then the total actual cost is $O(n)$.

Dynamic table

- *It is often useful to have a dynamic table:*
  - The table that expands and contracts as necessary when new elements are added or deleted.
    - **Expands** when insertion is done and the table is already full
    - **Contracts** when deletion is done and there is “too much” free space
  - Contracting or expanding involves **relocating**
    - Allocate new memory space of the new size
    - Copy all elements from the table into the new space
    - Free the old space
  - Worst-case time for insertions and deltions:
    - Without relocation: $O(1)$
    - With relocation: $O(m)$, where m – the number of elements in the table

Three Methods of Amortized Analysis

- Aggregate analysis:
  - 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.

Requirements

- Load factor
  - *num* – current number of elements in the table
  - *size* – the total number of elements that can be stored in the allocated memory
  - *Load factor* α = *num/size*
- It would be nice to have these two properties:
  - Amortized cost of insert and delete is constant
  - The load factor is always above some constant
    - That is the table is not too empty

Naïve insertions

- *Let's look only at insertions*: *Why not expand the table by some constant when it overflows?*
  - *What is the amortized cost of an insertion?*
  - *Does it satisfy the second requirement?*

Aggregate analysis

- *The "right" way to expand – double the size of the table*
  - Let's do an aggregate analysis
  - The cost of i-th insertion is:
    - i, if $i-1$ is an exact power of 2
    - 1, otherwise
  - Let's sum up…
  - The total cost of n insertions is then $< 3n$
  - Accounting method gives the intuition:
    - Pay \$1 for inserting the element
    - Put \$1 into element's account for reallocating it later
    - Put \$1 into the account of another element to pay for a later relocation of that element

Potential function

- *What potential function do we want to have?*
  - $\Phi_i = 2num_i - size_i$
  - It is always non-negative
  - Amortized cost of insertion:
    - Insertion triggers an expansion
    - Insertion does not trigger an expansion
  - Both cases: 3

Deletions

- *Deletions*: *What if we contract whenever the table is about to get less than half full*?
  - *Would the amortized running times of a sequence of insertions and deletions be constant*?
  - Problem: we want to avoid doing reallocations often without having accumulated "the money" to pay for that!

Deletions

- *Idea: delay contraction!*
  - Contract only when *num* = *size*/4
  - Second requirement still satisfied: $\alpha \geq 1/4$
- *How do we define the potential function?*

$$\Phi = \begin{cases} 2 \cdot num - size & \text{if } \alpha \geq 1/2 \\ size/2 - num & \text{if } \alpha < 1/2 \end{cases}$$

- It is always non-negative
- Let’s compute the amortized running time of deletions:
  - $\alpha < 1/2$ (with contraction, without contraction)