CHAPTER 17

Amortized Analysis

- In an ***amortized analysis,*** the time required to perform a sequence of data structure operations is averaged over all the operations performed.
- It can be used to show that the average cost of an operation is small, if one averages over a sequence of operations.
- While a particular operation in sequence may be expensive, this operation may not occur often enough to make the average cost expensive.
- An amortized analysis guarantees the average performance of each operation in the worst case.
- It determines the average time without the use of probability.
- Three methods are covered in text. The main difference is the way the cost is assigned.
- **Aggregate Method Characteristics**
  - Computes the worst case time $T(n)$ for a sequence of n operations.
  - The amortized cost (the average cost) per operation is $T(n)/n$
  - Gives average performance of each operation in the worst case.
  - This method is less precise than other methods, as all operations are assigned the same cost.

Aggregate Method

- **An Aggregate Method Example:** (Stack Operations)
  - Assume the following three operations on a stack:
    - *push(S,x)* - pushes *x* onto stack *S*
    - *pop(S)* - pops & returns top of stack *S*
    - *multipop(S,k)* - pops and returns the top *min{k,|S|}* items of *S*.

- Worst case cost for *Multipop* is *O(n)*
  - *n* successive calls to *Multipop* would cost $O(n^2)$.

- Consider a sequence of *n push*, *pop* and *multipop* operations on an initially empty stack.
  - This $O(n^2)$ cost is unfair.
    - Each item can be popped only once for each time it is pushed.
  - In a sequence of *n* mixed operations, the most times *multipop* can be called is *n/2*.
  - Since the cost of push and pop is *O(1)*, the cost of *n* stack operations is *O(n)*.
  - Therefore, the average cost of each stack operation in this sequence is *O(n)/n* or *O(1)*.

Aggregate Method (a binary counter)

- We use an array *A[0,1,…,k-1]* of bits, where *length(A)=k*, as a counter.
- A binary number *x* that is stored in the counter has its lowest order bit in *A[0]* and highest-order bit in *A[k-1]*, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially *x=0* (*A[I]=0* for all *i*).
- To add *1* (modulo 2^k) to the value of the counter we use the following procedure.

```
INCREMENT(A)  {  i=0
                 while i<length[A] and A[i]=1 do { A[i]=0; i=i+1 }
                 if i<length[A] then A[i]=1                       }
```

- A single execution of *INCREMENT* takes *O(k)* in the worst case (if A contains all 1's).
- A sequence of *n* INCREMENT operations on an initially zero counter takes time *O(kn)* in the worst case. (??? Is this true?)
- We can tighten our analysis to yield a worst case cost of *O(n)* operations for a sequence of *n INCREMENT's* by observing that not all bits flip each time *INCREMENT* is called.
- For *i=0,1,…[log n],* bit *A[i]* flips $\lfloor n/2^i \rfloor$ times in a sequence of *n INCREMENT* operations on an initially *zero* counter. For *i>[log n]* bit *A[i]* never flips at all.
- The total number of flips in the sequence is thus $\sum_{i=0}^{\log n} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$
- Therefore, the worst-case time for a sequence of *n INCREMENT* operations on an initial *zero* counter is *O(n)*. The average cost of each operation, and therefore the amortized cost per operation , is *O(n)/n=O(1)*.

Accounting Method

- **The Accounting Method Characteristics**
  - Assign different (artificial) charges to different operations. The amount we charge an operation is called its *amortized cost*.
  - When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as *credit*.
  - Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost.
  - The balance in the bank account is not allowed to become negative.
  - The sum of the amortized costs for any sequence of operations must be an upper bound for the actual total cost of these operations.
  - The amortized cost of each operation must be chosen wisely in order to pay for each operation on or before the cost is incurred.
- **An Accounting Method Example: (stack operations)**
  - Recall the actual costs of these operations were
    - *push (S,x)* 1
    - *pop (S)* 1
    - *multipop(S,k)* $min(k,|S|)$ (complexity depends on k)
  - The amortized costs assigned are
    - *push* 2
    - *pop* 0
    - *multipop* 0
  - Observe that the amortized cost of each operation is $O(1)$.

Accounting Method (examples)

- We must now show that we can pay for any sequence of stack operations by charging the amortized cost (recall that we start with initially empty stack).
- The two unit costs associated with each *push* is used as follows:
  - 1 unit is used to pay the cost of the *push*.
  - 1 unit is collected in advance to pay for a potential future *pop*.
- For any sequence of *n* operations of *push*, *pop*, and *multipop*, the total amortized cost is an upper bound on the total actual cost.
- Since the total amortized cost is *O(n)*, so is the total actual cost.

- In **incrementing a binary counter**, we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we will use as our cost for this example.
- For the amortized analysis, let as charge an amortized cost of 2 dollars to set a bit to 1.
- When a bit is set, we use 1 dollar to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0.
- The amortized cost of an *INCREMENT* operation is at most 2 dollars

```
INCREMENT(A)  {  i=0
                 while i<length[A] and A[i]=1 do { A[i]=0; i=i+1 }
                 if i<length[A] then A[i]=1                       }
```

- Thus, for *n INCREMENT* operations, the total amortized cost is *O(n)*, which bounds the total actual cost.

The Potential Method

- This method stores prepayments as a *potential* to pay for future operations.
  - The potential stored is associated with the entire data structure rather than with a specific item in that data structure.
  - **Notation:**
    - D_o is the initial data structure (e.g., stack)
    - D_i is the data structure after the *i*th operation
    - c_i is the actual cost of the *i*th operation.
    - The potential function Φ (i.e., psi) maps each D_i to its potential value, $\Phi(D_i)$.
  - The amortized cost $\hat{c}_i$ of the *i*th operation is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

    - Note: $\hat{c}_i$ = (actual cost) + (change in potential)
  - The <u>total amortized cost</u> is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + \Phi(D_i) - \Phi(D_{i-1})]$$
$$= (\sum_{i=1}^{n} c_i) + \Phi(D_n) - \Phi(D_0)$$

– By requiring that $\Phi(D_i) \geq \Phi(D_0)$ for all *i*, we insure that the total amortized cost is an upper bound for the actual cost for any sequence (➔choose appropriate Φ).

The Potential Method (cont.)

- If $\Phi(D_i) - \Phi(D_{i-1}) > 0,$ then $\hat{c}_i$ is an overcharge for the *i*th operation and causes an increase in potential.
- Similarly, if $\Phi(D_i) - \Phi(D_{i-1}) < 0,$ then $\hat{c}_i$ is an undercharge and results in a decrease in potential.
- **A Potential Method Example:** (Stack Operations)
  - The data structure D is initially an empty stack.
  - Let $\Phi(D)$ be the number of items in the stack. Then

$$\Phi(D_0) = 0 \text{ and } \Phi(D_i) \geq 0 \ \forall i$$

  - *push* operation : If the *i*th operation on D is a push, then

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

  - The amortized cost for a *push* is

$$\hat{c}_i = c_i + [\Phi(D_i) - \Phi(D_{i-1})] = 1 + 1 = 2$$

  - *multipop* operation: If the *i*th operation is *multipop(D,k)* and $k' = min\{|D|, k\}$, then $c_i = k'$ and $\hat{c}_i = c_i + [\Phi(D_i) - \Phi(D_{i-1})] = k' - k' = 0$
  - *pop* operation: Hence, if pop is the *i*th operation, then $\hat{c}_i = 0$
  - COST ANALYSIS:
    - The amortized cost for each operation is *O(1)*.
    - The amortized cost of *n* operations is *O(n)*.
    - The upper bound for the total cost is *O(n)*.

Dynamic Table Problem

- **Problem:** Consider the cost of a sequence of TABLE-DELETE and TABLE-INSERT commands for a dynamic table
  - Normally, cost of a insertion or deletion is 1.
  - However cost is large if a table expansion or contraction is triggered by the add or delete.
  - Analysis given here is independent of the data structure used.
- **Restricted Dynamic Table Problem (Insertions)**
  - Only insertions are allowed.
  - Goal: Try to keep the table as small as possible.
  - Must enlarge table when too many items inserted
  - **Proposed Idea for Algorithm:**

1. Initialize table size to $m = 1$
2. Insert elements until the number n of elements satisfies $n > m$.
3. Generate a new table of size $2m$ and set $m \leftarrow 2m$
4. Re-insert old elements into the new table.
5. Go to step 2.

- Let c_i be the cost of the ith insert. Then $c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2 \\ 1 & \text{otherwise} \end{cases}$

Aggregate Analysis

- Worst case is *O(n).*
- If repeated *n* times, is a worst case of $O(n^2)$ possible?
- Illustration:

Insertion	Size	Cost
1	1	1
2	2	1+1
3	4	1+2
4	4	1
5	8	1+4
6	8	1
7	8	1
8	8	1
9	16	1+8
10	16	1

- **Aggregate Analysis:**
  - *n* inserts cost $$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = n + \frac{2^{\lfloor \ln n \rfloor + 1} - 1}{2-1} \le n + (2n-1) < 3n.$$
  - Average cost of each operation
    = (Total Cost) / (Nr. of Operations)
    = 3
  - Asymptotically, cost is O(1) or the same as for a table of fixed size.

Accounting Analysis

- Avoids math if you can guess charges that work.
- Charge each operation 3 units for amortized cost:
  - use 1 to perform immediate insertion
  - store 2
- When table doubles,
  - use 1 unit to re-insert items added since last copy.
  - use 1 unit to re-insert items copied previously.

Potential Analysis

- Define $\Phi(T) = 2\, num(T) - size(T)$
- Immediately after an expansion but before a new item is inserted
  $num(T) = size(T) / 2$ which implies that $\Phi(T) = 0$
- Also, when the table is empty, $\Phi(T) = 0$.
- Since the table is always half-full $\Phi(T) \geq 0$
- Some useful definitions:
  num_i = number of elements after ith operation
  $size_i$ = table size after ith operation
  ϕ_i = potential after ith operation
- Since $num_0 = size_0$, it follows that $\phi_0 = 0$
- If ith insertion does not trigger an expansion, then $size_i = size_{i-1}$
  and $\hat{c}_i = c_i + \phi_i + \phi_{i-1} = 1 + (2\, num_i - size_i) - (2\, num_{i-1} - size_{i-1})$
  $= 1 + (2\, num_i - size_i) - 2\,(num_i - 1) + size_i = 3$

Potential Analysis (cont.)

– If the *i*th insertion causes an expansion, then $\frac{size_i}{2} = size_{i-1} = num_i - 1$

and $\hat{c}_i = c_i + \phi_i - \phi_{i-1} = num_i + (2\, num_i - size_i) - (2\, num_{i-1} - size_{i-1})$

$= num_i + 2\, num_i - 2\,(num_i - 1) - [2\,(num_i - 1) - (num_i - 1)]$

$= 3$

• • •

Definition: The load factor $\alpha(T)$ of a nonempty table T is $\alpha(T) = \frac{num(T)}{size(T)}$.

Observations:

1. $0 \le \alpha(T) \le 1$ and $size(T) * \alpha(T) = num(T)$,
2. If T is empty, then $num(T) = 0$.

– It is natural and useful to define for empty table

– $size(T) = 0$

– $\alpha(T) = 1$

– Note that these definitions preserve the equality in the first preceding observation.

Dynamic Tables with Insert and Delete

- ***Goal:*** **-** Keep the load factor of the dynamic table bound below by a positive constant.

  - Keep the amortized cost of each table operation bounded above by a constant.

- *Proposed Plan:* When the table usage drops below ½ , we could reduce the table to one-half its size.

  - *Problem:* This may cause thrashing. If a table is initially full, consider the following operations:

$$\mathrm{I}\uparrow\ \mathrm{DD}\downarrow\ \mathrm{I}\ \mathrm{I}\uparrow\ \mathrm{DD}\downarrow\ \mathrm{I}\ \mathrm{I}\uparrow\ \ldots$$

- A sequence of *n* insertions to fill the table, followed by a sequence of the above *n* operations has an average cost of *O(n)* per operation.

- *We avoid this problem* by waiting until the table is well below ½ full before contracting it.

- By contracting the table when it falls below ¼ full, we maintain the lower bound $\alpha(T) \geq 1/4$.

- If the table becomes empty, we cut its size to 0.

Analysis by the Potential Method

– ***Goal:*** We want the potential Φ(T) to satisfy Φ(T) = 0 immediately following a contraction/expansion (before any element is added/deleted).

– Φ(T) builds to pay for a change as the load factor approaches ¼ or 1.

Definition:

$$\Phi(T) = \begin{cases} 2num(T) - size(T) & if \;\; \alpha(T) \geq 1/2 \\ size(T)/2 - num(T) & if \;\; \alpha(T) < 1/2 \end{cases}.$$

– ***Properties:***

– Both branches of formula agree (and equal 0) at their switchover point, $\alpha(T) = \frac{1}{2}$.

– If $\alpha(T) = 1$, then $num(T) = size(T)$, $\Phi(T) = num(T)$.

So the potential can pay to move each item.

– If $\alpha(T) = 1/4$, then $4num(T) = size(T)$,

and $\Phi(T) = 4num(T)/2 - num(T) = num(T)$.

So the potential can pay to move each item.

– Φ(T) is zero immediately after each table expansion or contraction.

– *Exercise:* Consider graph of Φ(T) over [8,32] immediately after *num* increases from 16 to 17.

Analysis by the Potential Method (INSERT)

– ***Notation:*** The subscript *i* in

$$\hat{c}_i, c_i, num_i, size_i, \alpha_i, \Phi_i$$

will denote their values after the *i*th operation.

– ***Observation:*** Recall that initially

$$num_0 = size_0 = \Phi_0 = 0;\ \alpha_0 = 1.$$

– ***Case 1:*** Suppose the *i*th operation is INSERT and $\alpha_{i-1}(T) \geq 1/2$. Then the analysis is same as for a table allowing only INSERT and $\hat{c}_i = 3$.

– ***Case 2:*** Suppose the *i*th operation is INSERT and $\alpha_{i-1}(T) < 1/2$, $\alpha_i(T) < 1/2$.

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1} = 1 + (\tfrac{1}{2}\, size_i - num_i) - (\tfrac{1}{2}\, size_{i-1} - num_{i-1})$$

$$= 1 + \tfrac{1}{2}\, size_i - num_i - \tfrac{1}{2}\, size_i + num_i - 1 = 0.$$

– ***Case 3:*** Suppose the *i*th operation is INSERT and $\alpha_{i-1}(T) < 1/2$, $\alpha_i(T) \geq 1/2$.

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1} = 1 + (2\, num_i - size_i) - (\tfrac{1}{2}\, size_{i-1} - num_{i-1})$$

$$= 1 + (2\,(num_{i-1}+1) - size_{i-1}) - (\tfrac{1}{2}\, size_{i-1} - num_{i-1})$$

$$= 3\, num_{i-1} - \tfrac{1}{2}(3\, size_{i-1}) + 3 = 3\, \alpha_{i-1} size_{i-1} - \tfrac{1}{2}(3\, size_{i-1}) + 3$$

$$< 3/2\, size_{i-1} - 3/2\, size_{i-1}) + 3 = 3.$$

- Thus, for an INSERT, the amortized cost is $\hat{c}_i \leq 3$.

Analysis by the Potential Method (DELETE)

– ***Case 4:*** Suppose the *i*th operation is DELETE and $\alpha_{i-1}(T) < 1/2$, but a deletion does not cause a contraction. An easy calculation (in text) shows that $\hat{c}_i = 2$.

– ***Case 5:*** Suppose $\alpha_{i-1}(T) < 1/2$, but deletion causes a contraction.

– one item is deleted, leaving num_i items to move, so $c_i = 1 + num_i$.

– note that $size_i = ½\ size_{i-1} = 2\ num_{i-1} = 2(num_i + 1)$.

– Then, $\hat{c}_i = c_i + \phi_i - \phi_{i-1} = 1 + num_i + (½\ size_i - num_i) - (½\ size_{i-1} - num_{i-1})$

$= 1 + num_i + num_i + 1 - num_i - 2\ num_i - 2 + num_i + 1 = 1.$

– ***Case 6:*** $\alpha_{i-1}(T) \geq 1/2$ and *i*th operation is a deletion.

– By Exercise 17.4-2, the amortized cost $\hat{c}_i$ with respect to the potential function is bounded above by a constant.

– Exercise 17.4-2 is assigned as homework.

– In all cases, the amortized cost for each operation is bounded above by a constant.

– Thus, the actual time for *n* insert and delete operations on a dynamic table is *O(n)*.