Lecture 16: Amortized Analysis

(CLRS 17.1-17.3)

June 10th, 2002

1 Amortized Analysis

- Until now we have seen a number of data structures and analyzed the worst-case running time of each individual operation.
- Sometimes the cost of an operation vary widely, so that that worst-case running time is not really a good cost measure.
- Similarly, sometimes the cost of every single operation is not so important
  - the total cost of a series of operations are more important (e.g when using priority queue to sort)

$\Downarrow$

- We want to analyze running time of one single operation averaged over a sequence of operations
  - Note: We are not interested in an average case analyses that depends on some input distribution or random choices made by algorithm.
- To capture this we define *amortized time*.

If any sequence of n operations on a data structure takes $\leq T(n)$ time, the amortized time per operation is $T(n)/n$

  - Equivalently, if the amortized time of one operation is $U(n)$, then any sequence of n operations takes $n \cdot U(n)$ time.
- Again keep in mind: "Average" is over a sequence of operations for *any* sequence
  - *not* average for some input distribution (as in quick-sort)
  - *not* average over random choices made by algorithm (as in skip-lists)

1.1 Example: Stack with Multipop

- As we know, a normal stack is a data structure with operations
  - Push: Insert new element at top of stack
  - Pop: Delete top element from stack
- A stack can easily be implemented (using linked list) such that Push and Pop takes $O(1)$ time.
- Consider the addition of another operation:
  - Multipop(k): Pop k elements off the stack.
- Analysis of a sequence of n operations:
  - One Multipop can take $O(n)$ time $\Rightarrow O(n^2)$ running time.
  - Amortized running time of each operation is $O(1) \Rightarrow O(n)$ running time.
    - Each element can be popped at most once each time it is pushed
      - Number of Pop operations (including the one done by Multipop) is bounded by n
      - Total cost of n operations is $O(n)$
      - Amortized cost of one operation is $O(n)/n = O(1)$.

1.2 Example: Binary counter

- Consider the following (somewhat artificial) data structure problem: Maintain a binary counter under n Increment operations (assuming that the counter value is initially 0)
  - Data structure consists of an (infinite) array A of bits such that $A[i]$ is either 0 or 1.
  - $A[0]$ is lowest order bit, so value of counter is $x = \sum_{i \geq 0} A[i] \cdot 2^i$
  - Increment operation:

```
A[0] = A[0] + 1
i = 0
WHILE A[i] = 2 DO
    A[i + 1] = A[i + 1] + 1
    A[i] = 0
    i = i + 1
OD
```

- The running time of Increment is the number of iterations of while loop +1.

  Example (Note: Bit furthest to the right is $A[0]$):

  $x = 47 \Rightarrow A =< 0, \ldots, 0, 1, 0, 1, 1, 1, 1 >$

  $x = 48 \Rightarrow A =< 0, \ldots, 0, 1, 1, 0, 0, 0, 0 >$

  $x = 49 \Rightarrow A =< 0, \ldots, 0, 1, 1, 0, 0, 0, 1 >$

  Increment from $x = 47$ to $x = 48$ has cost 5

  Increment from $x = 48$ to $x = 49$ has cost 1

- Analysis of a sequence of n INCREMENTS
  - Number of bits in representation of n is $\log n \Rightarrow n$ operations cost $O(n \log n)$.
  - Amortized running time of INCREMENT is $O(1) \Rightarrow O(n)$ running time:
    - $A[0]$ flips on each increment (n times in total)
    - $A[1]$ flips on every second increment ($n/2$ times in total)
    - $A[2]$ flips on every fourth increment ($n/4$ times in total)

      $\vdots$
    - $A[i]$ flips on every 2^ith increment ($n/2^i$ times in total)

    $\Downarrow$

    Total running time:
$$\begin{aligned} T(n) &= \textstyle\sum_{i=0}^{\log n} \frac{n}{2^i} \\ &\leq n \cdot \textstyle\sum_{i=0}^{\log n} (\frac{1}{2})^i \\ &= O(n) \end{aligned}$$

2 Potential Method

- In the two previous examples we basically just did a careful analysis to get $O(n)$ bounds leading to $O(1)$ amortized bounds.
  - book calls this *aggregate analysis*.
- In aggregate analysis, all operations have the same amortized cost (total cost divided by n)
  - other and more sophisticated amortized analysis methods allow different operations to have different amortized costs.
- *Potential method*:
  - Idea is to *overcharge* some operations and store the overcharge as *credits/potential* which can then help pay for later operations (making them cheaper).
  - Leads to equivalent but slightly different definition of amortized time.
- Consider performing n operations on an initial data structure D_0
  - D_i is data structure after ith operation, $i = 1, 2, \ldots, n$.
  - c_i is actual cost (time) of ith operation, $i = 1, 2, \ldots, n$.

    $\Downarrow$

    Total cost of n operations is $\sum_{i=0}^{n} c_k$.
- We define *potential function* mapping D_i to R. ($\Phi : D_i \rightarrow R$)
  - $\Phi(D_i)$ is potential associated with D_i
- We define *amortized cost* $\tilde{c}_i$ of ith operation as $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - $\tilde{c}_i$ is sum of real cost and *increase* in potential

    $\Downarrow$
  - If potential decreases the amortized cost is lower than actual cost (we use saved potential/credits)
  - If potential increases the amortized cost is larger than actual cost (we overcharge operation to save potential/credits).

- Key is that, as previously, we can bound total cost of all the n operations by the total amortized cost of all n operations:

$$\begin{array}{rcl} \sum_{i=1}^{n} c_k & = & \sum_{i=1}^{n}(\tilde{c}_i + \Phi(D_{i-1}) - \Phi(D_i)) \\ & = & \Phi(D_0) - \Phi(D_n) + \sum_{i=1}^{n} \tilde{c}_i \end{array}$$

$\Downarrow$

$\sum_{i=1}^{n} c_k \leq \sum_{i=1}^{n} \tilde{c}_i$ if $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i (or even if just $\Phi(D_n) \geq \Phi(D_0)$)

2.1 Example: Stack with multipop

- Define $\Phi(D_i)$ to be the size of stack $D_i \Rightarrow \Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$
- Amortized costs:
  - PUSH:
  $$\begin{array}{rcl} \tilde{c}_i & = & c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ & = & 1 + 1 \\ & = & 2 \\ & = & O(1). \end{array}$$
  - POP:
  $$\begin{array}{rcl} \tilde{c}_i & = & c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ & = & 1 + (-1) \\ & = & 0 \\ & = & O(1). \end{array}$$
  - MULTIPOP(k):
  $$\begin{array}{rcl} \tilde{c}_i & = & c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ & = & k + (-k) \\ & = & 0 \\ & = & O(1). \end{array}$$
- Total cost of n operations: $\sum_{i=1}^{n} c_k \leq \sum_{i=1}^{n} \tilde{c}_i = O(n)$.

2.2 Example: Binary counter

- Define $\Phi(D_i) = \sum_{i \geq 0} A[i] \Rightarrow \Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$
  - $\Phi(D_i)$ is the number of ones in counter.
- Amortized cost of ith operation: $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Consider the case where first k positions in A are 1 $A =< 0, 0, \cdots, 1, 1, 1, 1, \cdots, 1 >$
  - In this case $c_i = k + 1$
  - $\Phi(D_i) - \Phi(D_{i-1})$ is $-k + 1$ since the first k positions of A are 0 after the increment and the $k + 1$th position is changed to 1 (all other positions are unchanged)

    $\Downarrow$
  - $\tilde{c}_i = k + 1 - k + 1 = 2 = O(1)$
- Total cost of n increments: $\sum_{i=1}^{n} c_k \leq \sum_{i=1}^{n} \tilde{c}_i = O(n)$.

2.3 Notes on amortized cost

- Amortized cost depends on choice of Φ
- Different operations can have different amortized costs.
- Often we think about potential/credits as being distributed on certain parts of data structure.

  In multipop example:

  - Every element holds one credit.
  - Push: Pay for operation (cost 1) and for placing one credit on new element (cost 1).
  - Pop: Use credit of removed element to pay for the operation.
  - Multipop: Use credits on removed elements to pay for the operation.

  In counter example:

  - Every 1 in A holds one credit.
  - Change from $1 \rightarrow 0$ payed using credit.
  - Change from $0 \rightarrow 1$ payed by Increment; pay one credit to do the flip and place one credit on new 1.

    $\Downarrow$

    Increment cost $O(1)$ amortized (at most one $0 \rightarrow 1$ change).
- Book calls this the *accounting method*
  - Note: Credits only used for analysis and is not part of data structure
- Hard part of amortized analysis is often to come up with potential function Φ
  - Some people prefer using potential function (*potential method*), some prefer thinking about placing credits on data structure (*Accounting method*)
  - Accounting method often good for relatively easy examples.
- Next time we will discuss an elegant "self-adjusting" search tree data structure with amortized $O(\log n)$ bonds for all operations (*splay trees*).