Topic 13: Amortized Analysis

(CLRS 17)

CPS 230, Fall 2001

1 Amortized Analysis

- After discussing algorithm design techniques (such as dynamic programming and greedy algorithms), we now return to data structures and discuss a new analysis method—*amortized analysis.*
- Until now we have seen a number of data structures and have analyzed the worst-case or average-case running time of each individual operation.
- Sometimes the cost of an operation varies widely, so that that worst-case running time is not really a good cost measure. On the other hand, average-case analysis is subject to probability assumptions about an input distribution or the randomness in the algorithm itself. There is no worst-case guaranteed bound on performance.
- Sometimes the cost of every single operation is not so important
  - For example, the total cost of a series of operations can be more important, as when using priority queue to sort.

$\Longrightarrow$ We may want to analyze the worst-case 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 thus we define *amortized time.*

If any sequence of n operations on a data structure takes $\leq T(n)$ time, then 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 $\leq n \cdot U(n)$ time.

- Again keep in mind: The average is over a worst-case sequence of n operations
  - *not* the average for some input distribution (as in quicksort)
  - *not* the 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 a linked list) such that Push and Pop each take $O(1)$ time per operation.
- 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 $\Longrightarrow$ $O(n^2)$ running time.
  - Better bound: Amortized running time of each operation is $O(1) \Longrightarrow O(n)$ running time.
    - Each element can be popped at most once each time it is pushed
      - Number of Pop operations (including the ones 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 the 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 { carry a 1 }
    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, plus 1. Example (Note: Bit furthest to the right is $A[0]$):

$$x = 47 \Longrightarrow A = \langle 0, \ldots, 0, 1, 0, 1, 1, 1, 1 \rangle$$
$$x = 48 \Longrightarrow A = \langle 0, \ldots, 0, 1, 1, 0, 0, 0, 0 \rangle$$
$$x = 49 \Longrightarrow A = \langle 0, \ldots, 0, 1, 1, 0, 0, 0, 1 \rangle$$

  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 $\log n \Longrightarrow n$ operations cost $O(n \log n)$.
  - Better: Amortized running time of Increment is $O(1)$ $\Longrightarrow$ $O(n)$ 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: $$T(n) = \sum_{i=0}^{\log n} \frac{n}{2^i} = n \sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^i = O(n)$$

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 the *aggregate method* of amortized analysis.
- In aggregate method 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* or *potential*, which can then help pay for expensive later operations.

- 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.
  - c_i is actual cost (time) of ith operation, $i = 1, 2, \ldots, n$.
    $\Longrightarrow$ 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 the actual cost (i.e., we use saved potential/credits)
  - If potential increases, the amortized cost is larger than the actual cost (i.e., we overcharge operation to save potential/credits).

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

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

  which implies that

$$\sum_{i=1}^{n} c_k \leq \sum_{i=1}^{n} \tilde{c}_i,$$

  if $\Phi(D_0) = 0$ and $\Phi(D_n) \geq 0$. Or actually all we need is for $\Phi(D_n) \geq \Phi(D_0)$.

- This relation is consistent (at least in one direction) with the intuition we were trying to capture at the beginning of this lecture note, assuming that $\Phi(D_n) \geq \Phi(D_0)$. If each $\tilde{c}_i$ is the same, then we have

$$\tilde{c}_i \geq \frac{1}{n}\sum_{i=1}^{n} c_k.$$

- The hard (tricky) part of amortized analysis is defining the potential function appropriately to give the desired amortized bounds. We want the potential function to be *large* when we are about to do an expensive operation. Immediately afterwards, we want the potential to be *small*, so that the decrease in potential pays for the bulk of that operation.

2.1 Example: Stack with multipop

- Define $\Phi(D_i)$ to be the size of stack $D_i \quad \Longrightarrow \quad \Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$
- Amortized costs:
  - PUSH:
$$\begin{aligned} \tilde{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 \\ &= O(1). \end{aligned}$$
  - POP:
$$\begin{aligned} \tilde{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (-1) \\ &= 0 \\ &= O(1). \end{aligned}$$
  - MULTIPOP(k):
$$\begin{aligned} \tilde{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k + (-k) \\ &= 0 \\ &= O(1). \end{aligned}$$
- 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 $\Phi(D_i) = \sum_{i \geq 0} A[i] \quad \Longrightarrow \quad \Phi(D_0) = 0$ 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: $A = \langle 0, 0, \cdots, 1, 1, 1, 1, \cdots, 1 \rangle$
  - In this case $c_i = k + 1$
  - $\Phi(D_i) - \Phi(D_{i-1}) = -k + 1$ since the first k positions of A are 0 after the increment and the $(k+1)$st position is changed to 1. (The other positions are unchanged.) $\Longrightarrow \tilde{c}_i = k + 1 + (-k + 1) = 2 = O(1)$.
- Total cost of n increments: $\sum_{i=1}^{n} c_k \leq \sum_{i=1}^{n} \tilde{c}_i = O(n)$.

2.3 Dynamic Tables

- We want to maintain a table (array) dynamically. Inserts and Deletes work as follows: When an insert causes the table to overflow, the table size is doubled (expansion). When a delete causes the table to become less than one-quarter full, the table size is cut in half (contraction).

- We want to show that the amortized cost for an insert or delete is $O(1)$, even though the worst-case time is $O(n)$.
- Define
$$\Phi(D) = \big|2 \times (\text{\# elements in table}) - (\text{size of table})\big|,$$
so that the potential is high when the table is near-full or relatively sparse, which is when the expensive operations of expansion or contraction are done. (The book uses a slightly more complicated potential function.)
- Let's say that the costs of expansion and contraction are each equal to the number of elements in the table at that time.
- Let's examine what happens when we insert an element.
  - If there is room in the table, the element is placed in the table; no expansion is done.
$$\begin{aligned}\tilde{c}_k &= c_k + (\Phi(D_{k+1} - \Phi(D_k)) \\ &\le 1 + 2 \\ &= 3\end{aligned}$$
  - If there is no room in the table, the table is doubled and the element is placed in the newly expanded table.
$$\begin{aligned}\tilde{c}_k &= c_k + (\Phi(D_{k+1} - \Phi(D_k)) \\ &= (\text{\# elements in table}) + 1 + (0 - (\text{\# elements in table})) \\ &= 1\end{aligned}$$
- Now let's examine what happens when we delete an element.
  - If the table remains at least one-quarter full after the deletion, there is no need to contract the table.
$$\begin{aligned}\tilde{c}_k &= c_k + (\Phi(D_{k+1} - \Phi(D_k)) \\ &\le 1 + 2 \\ &= 3\end{aligned}$$
  - If the deletion causes the table to become less than one-quarter full, then the table size is contracted to half its former size.
$$\begin{aligned}\tilde{c}_k &= c_k + (\Phi(D_{k+1} - \Phi(D_k)) \\ &\le (\text{\# elements in table}) + 1 + (2 - 2(\text{\# elements in table})) \\ &= 3 - (\text{\# elements in table}) \qquad (1) \\ &\le 3\end{aligned}$$
- The potential function in the book increases at only half the rate that ours does as the table occupancy decreases to one-quarter full, so the bound they get for (1) is simply 3.

- Let's look at how sensitive the analysis is to our decision to contract the table when it gets to less than one-quarter full. What if we decided to halve it when it got less than *one-half* full?

- Consider a sequence of operations that causes the table to double on the last operation. (The number of elements is 1 more than a power of 2.) Then consider the following subsequent sequence of operations applied to this data structure: two deletes followed by two inserts followed by two deletes followed by two inserts, and so on. The second of each pair of deletes will cause a contraction, and the second of each pair of inserts will cause an expansion

  $\Longrightarrow \quad \Omega(n)$ amortized time (not $O(1)$!!!).

- Therefore we have to be careful in our algorithm design in order to get a good amortized bound.

2.4 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. Amortized cost is 0.
  - Multipop: Use credits on removed elements to pay for the operation.

  In counter example:

  - Every 1 in A holds one credit.
  - Change from $1 \to 0$ paid using credit.
  - Change from $0 \to 1$ paid by Increment; pay one credit to do the flip and place another credit on the newly created 1.

    $\Longrightarrow$ Increment cost $O(1)$ amortized (at most one $0 \to 1$ 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*).

3 Summary

• Motivation for amortized analysis:

  - A way of expressing that even though the worst-case performance of an operation can be bad, the total performance of a sequence of operations cannot be bad.
  - One way of thinking of amortized time is as being an "average": If any sequence of n operations takes less than $T(n)$ time, the amortized time per operation is $T(n)/n$.

• We formally defined amortized time using the idea that we overcharge some operations and store the overcharge as credits or potential that can then help pay for later operations (*potential method*)

  - Consider performing n operations on an initial data structure D_0
  - D_i is data structure after ith operation.
  - c_i is actual cost (time) of ith operation.
  - Potential function: $\Phi : D_i \rightarrow R$
  - $\tilde{c}_i$ amortized cost of ith operation: $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Given $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$, we have $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \tilde{c}_i$

• We also discussed three examples of amortized analysis:

  - Stack with MULTIPOP ($O(n)$ worst-case time, $O(1)$ amortized time).
  - INCREMENT on binary counter ($O(\log n)$ worst-case time, $O(1)$ amortized time).
  - Dynamic table with inserts and deletes ($O(n)$ worst-case time, $O(1)$ amortized time).

  In each case, we could argue for $O(1)$ amortized performance without actually doing potential calculation—we just think about potential/credits as being distributed on certain parts of the data structure and let operations put and take credits while maintaining some invariant (*accounting method*).