Lecture 5: Amortization

Amortized analysis is a powerful technique for data structure analysis, involving the total runtime of a sequence of operations, which is often what we really care about. This lecture covers:

- Different techniques of amortized analysis
  - aggregate method
  - accounting method
  - charging method
  - potential method
- Examples of amortized analysis
  - table doubling
  - binary counter
  - 2-3 tree and 2-5 tree

Table doubling

(Recall from 6.006) We want to store n elements in a table of size $m = \Theta(n)$. One idea is to double m whenever n becomes larger than m (due to insertions). The cost to double a table of size m is clearly $\Theta(m) = \Theta(n)$, which is also the worse case cost of an insertion.

But what is the total cost of n insertions? It is at most

$$2^0 + 2^1 + 2^2 + \cdots + 2^{\lceil \lg n \rceil} = \Theta(n).$$

In this case, we say each insertion has $\Theta(n)/n = \Theta(1)$ *amortized cost*.

Aggregate Method

The method we used in the above analysis is the aggregate method: just add up the cost of all the operations and then divide by the number of operations.

$$\text{amortized cost per operation } = \frac{\text{total cost of } k \text{ operations}}{k}$$

Aggregate method is the simplest method. Because it's simple, it may not be able to analyze more complicated algorithms.

Amortized Bound Definition

Amortized cost can be, but does not have to be, average cost. We can assign any amortized cost to each operation, as long as they "preserve the total cost", i.e., for any sequence of operations,

$$\sum \text{ amortized cost } \geq \sum \text{ actual cost}$$

where the sum is taken over all operations.

For example, we can say a 2-3 tree achieves $O(1)$ amortized cost per create, $O(\lg n^*)$ amortized cost per insert, and 0 amortized cost per delete, where n^* is the maximum size of the 2-3 tree during the entire sequence of operations. The reason we can claim this is that for any sequence of operations, suppose there are c creations, i insertions and $d \leq i$ deletions (cannot delete from an empty tree), the total amortized cost is asymptotically the same as the total actual cost:

$$O(c + i \lg n^* + 0d) = O(c + i \lg n^* + d \lg n^*)$$

Later, we will tighten the amortized cost per insert to $O(\lg n)$ where n is the *current* size.

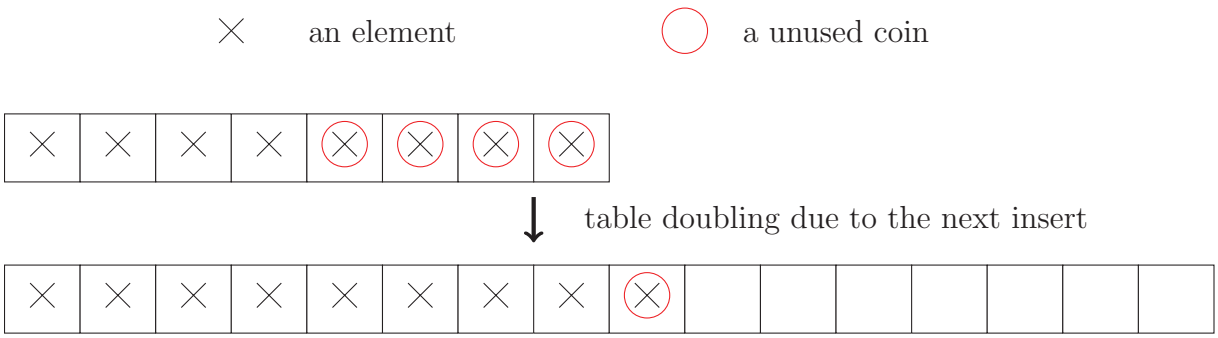
Accounting Method

This method allows an operation to store credit into a bank for future use, if its assigned amortized cost > its actual cost; it also allows an operation to pay for its extra actual cost using existing credit, if its assigned amortized cost < its actual cost.

Table doubling

For example, in table doubling:

- if an insertion does not trigger table doubling, store a coin represnting $c = O(1)$ work for future use.
- if an insertion does trigger table doubling, there must be $n/2$ elements that are inserted after the previous table doubling, whose coins have not been consumed. Use up these $n/2$ coins to pay for the $O(n)$ table doubling. See figure below.
- amortized cost for table doubling: $O(n) - c \cdot n/2 = 0$ for large enough c.
- amortized cost per insertion: $1 + c = O(1)$.



2-3 trees

Now let's try the accounting method on 2-3 trees. Our goal is to show that insert has $O(\lg n)$ amortized cost and delete has 0 amortized cost. Let's try a natural approach: save a $O(\lg n)$ coin for inserting an element, and use this coin when we delete this element later. However, we will run into a problem: by the time we delete the element, the size of the tree may have got bigger $n' > n$, and the coin we saved is not enough to pay for the $\lg n'$ actual cost of that delete operation! This problem can be solved using the charging method in the next section.

Charging Method

The charging method allows operations to charge cost retroactively to *past* operations.

$$\begin{aligned} \text{amortized cost of an operation} &= \text{actual cost of this operation} \\ &- \text{total cost charged to past operations} \\ &+ \text{total cost charged by future operations} \end{aligned}$$

Table doubling and halving

For example, in table doubling, when the table doubles from m to $2m$, we can charge $\Theta(m)$ cost to the $m/2$ insert operations since the last doubling. Each insert is charged by $\Theta(1)$, and will not be charged again. So the amortized cost per insert is $\Theta(1)$.

Now let's extend the above example with table halving. The motivation is to save space when with deletes. If the table is down to 1/4 full, $n = m/4$, we shrink the table size from m to $m/2$ at $\Theta(m)$ cost. This way, the table is half full again after any resize (doubling or shrinking). Now each table doubling still has $\geq m/2$ insert operations to charge to, and each table halving has $\geq m/4$ delete operations to charge to. So the amortized cost per insert or delete is still $\Theta(1)$.

Free deletion in 2-3 trees

For another example, let's consider insertion and deletion in 2-3 trees. Again, our goal is to show that insert has $O(\lg n)$ amortized cost, where n is the size of the tree when that insert happens, and delete has 0 amortized cost.

Insert does not need to charge anything.

Delete will charge an insert operation. But we will not charge the insert of the element to be deleted, because we will run into the same problem as the accounting method. Instead, each delete operation will charge the insert operation that brought the tree to its current size n. Each insert is still charged at most once, because for the tree size to reach n again, another insert must happen.

Potential Method

This method defines a potential function Φ that maps a data structure (DS) configuration to a value. This function Φ is equivalent to the total unused credits stored up by all past operations (the bank account balance). Now

$$\text{amortized cost of an operation} \;=\; \text{actual cost of this operation} \;+ \Delta\Phi$$

and

$$\sum \text{amortized cost} \;= \sum \text{actual cost} \;+ \Phi(\text{final DS}) - \Phi(\text{initial DS}).$$

In order for the amortized bound to hold, Φ should never go below Φ(initial DS) at any point. If Φ(initial DS) $= 0$, which is usually the case, then Φ should never go negative (intuitively, we cannot "owe the bank").

Relation to accounting method

In accounting method, we specify $\Delta\Phi$, while in potential method, we specify Φ. One determines the other, so the two methods are equivalent. But sometimes one is more intuitive than the other.

Binary counter

Our first example of potential method is incrementing a binary counter. E.g.,

$$\begin{array}{rc} & 0011010111 \\ \text{increment} & \downarrow \\ & 0011011000 \end{array}$$

Cost of increment is $\Theta(1 + \#1)$, where $\#1$ represents the number of trailing 1 bits. So the intuition is that 1 bits are bad.

Define $\Phi = c \cdot \#1$. Then for large enough c,

$$\begin{aligned} \text{amortized cost} &= \text{actual cost} + \Delta\Phi \\ &= \Theta(1 + \#1) + c(-\#1 + 1) \\ &= \Theta(1) \end{aligned}$$

$\Phi(\text{initial DS}) = 0$ if the counter starts at $000\cdots0$. This is necessary for the above amortized analysis. Otherwise, Φ may become smaller than $\Phi(\text{initial DS})$.

Insert in 2-3 trees

Insert can cause $O(\lg n)$ splits in the worst case, but we can show it causes only $O(1)$ amortized splits. First consider what causes a split: insertion into a 3-node (a node with 3 children). In that case, the 3-node needs to split into two 2-nodes.

So 3-nodes are bad. We define Φ = the number of 3-nodes. Then $\Delta\Phi \leq 1 -$ the number of splits. Amortized number of splits = actual number of splits + $\Delta\Phi$ = 1. $\Phi(\text{initial DS}) = 0$ if the tree is empty initially.

The above analysis holds for any (a, b)-tree, if we define Φ to be the number of b-nodes.

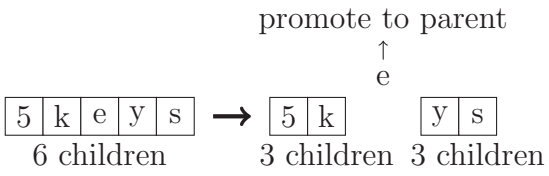
If we consider both insertion and deletion in 2-3 trees, can we claim both $O(1)$ splits for insert, and $O(1)$ merges for delete? The answer is no, because a split creates two 2-nodes, which are bad for merge. In the worse case, they may be merged by the next delete, and then need split again on the next insert, and so on.

What do we solve this problem? We need to prevent split and merge from creating 'bad' nodes.

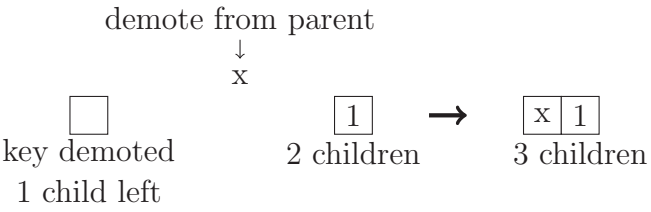
Insert and delete in 2-5 trees

We can claim $O(1)$ splits for insert, and $O(1)$ merges for delete in 2-5 trees.

In 2-5 trees, insertion into a 5-node (a node with 5 children) causes it to split into two 3-nodes.



Deletion from a 2-node causes it to merge with another 2-node to form a 3-node.



5-nodes and 2-nodes are bad. We define Φ = # of 5-nodes + # of 2-nodes. Amortized splits and merges = 1. Φ(initial DS) = 0 if the tree is empty initially.

The above analysis holds for any (a, b)-tree where $b > 2a$, because splits and merges do not produce bad nodes. We define Φ to be the number of b-nodes plus the number of a nodes.

Note: The potential examples could also be done with the accounting method by placing coins on 1s (binary counter) or 2/5-nodes ($(2, 5)$-trees).

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015