Amortized Analysis Explained

by Rebecca Fiebrink
Princeton University

This is a version of a Wikipedia-style report submitted as part of COS 423 under Robert Tarjan in Spring 2007. My intention in posting this to the web is to benefit future students by offering a comprehensive discussion, selected in-depth examples accessible to students without prior knowledge of particular data structures, and a review of applications relevant to an algorithms course.

Contents

Introduction

Overview

Amortized analysis is a technique for analyzing an algorithm's running time. It is often appropriate when one is interested in understanding asymptotic behavior over *sequences* of operations. For example, one might be interested in reasoning about the running time for an arbitrary operation to insert an item into a binary search tree structure. In cases such as this, it might be straightforward to come up with an upper bound by, say, finding the worst-possible time required for any operation, then multiplying this by the number of operations in the sequence. However, many real data structures, such as splay trees, have the property that it is impossible for every operation in a sequence to take the worst-case time, so this approach can result in a horribly pessimistic bound! With a bit of clever reasoning about properties of the problem and data structures involved, amortized analysis allows a tighter bound that better reflects performance.

Key ideas

- Amortized analysis is an upper bound: it's the average performance of each operation *in the worst case*.
- Amortized analysis is concerned with the overall cost of a sequence of operations. It does not say anything about the cost of a specific operation in that sequence. For example, it is invalid to reason, “The amortized cost of insertion into a splay tree with n items is O(log n), so when I insert '45' into this tree, the cost will be O(log n).” In fact, inserting '45' might require O(n) operations! It is only appropriate to say, “When I insert m items into a tree, the average time for each operation will be O(log n).”
- Amortized analysis is concerned with the overall cost of arbitrary sequences. An amortized bound will hold regardless of the specific sequence; for example, if the amortized cost of insertion is O(log n), it is so regardless of whether you insert the sequence '10,' '160,' '2' or the sequence '2', '160', '10,' '399', etc.
- The two points above imply that both amortized and worst-case bounds should be understood when choosing an algorithm to use in practice. Because an amortized bound says nothing about the cost of individual operations, it may be possible that one operation in the sequence requires a huge cost. Practical systems in which it is important that all operations have low and/or comparable costs may require an algorithm with a worse amortized cost but a better worst-case per-operation bound.
- Amortized analysis can be understood to take advantage of the fact that some expensive operations may “pay” for future operations by somehow limiting the number or cost of expensive operations that can happen in the near future.
- If good amortized cost is a goal, an algorithm may be designed to explicitly perform this “clean-up” during expensive operations!
- The key to performing amortized analysis is picking a good “credit” or “potential” function that captures how operations with different actual costs affect a data structure and allows the desired bounds to be shown.

History

In finance, *amortization* refers to paying off a debt, such as a loan or mortgage, by smaller payments made over time. The paper introducing amortized analysis for algorithms as a general technique was published by Tarjan in 1985, though he recognizes prior application of the banker's method in work by Brown and Tarjan (1980) and Huddleston and Melhorn (1981, 1982) and credits Sleator with the concept of the potential method. Aggregate analysis is an older method (Aho et al. 1974) that can now be understood as a form of amortized analysis.

Early applications of amortized analysis included the "move-to-front" list updating heuristic (Sleator and Tarjan 1985a), self-adjusting and balanced binary trees (Tarjan 1985, Sleator and Tarjan 1985b), and path compression heuristics for the disjoint set union problem (Tarjan 1983, 1985). Amortization plays an important role in the analysis of many other standard algorithms and data structures, including maximum flow, Fibonacci heaps, and dynamic arrays. All these applications of amortized analysis are discussed below. Amortized analysis has recently been used in diverse applications including security (Mao et al. 2006), databases (Agarwal et al. 2006), and distributed computing (Fomitchev and Ruppert 2004), and it continues to be important to theoretical work on algorithms and data structures (e.g., Mendelson et al. 2006; Georgiadis et al. 2006).

Comparison to other analysis techniques

As mentioned above, worst-case analysis can give overly pessimistic bounds for sequences of operations, because such analysis ignores interactions among different operations on the same data structure (Tarjan 1985). Amortized analysis may lead to a more realistic worst-case bound by taking these interactions into account. Note that the bound offered by amortized analysis is, in fact, a worst-case bound on the average time per operation; a single operation in a sequence may have cost worse than this bound, but the average cost over all operations in any valid sequence will always perform within the bound.

Amortized analysis is similar to average-case analysis, in that it is concerned with the cost averaged over a sequence of operations. However, average-case analysis relies on probabilistic assumptions about the data structures and operations in order to compute an *expected* running time of an algorithm. Its applicability is therefore dependent on certain assumptions about probability distributions on algorithm inputs, which means the analysis is invalid if these assumptions do not hold (or that probabilistic analysis cannot be used at all, if input distributions cannot be described!) (Cormen et al. 2001, 92–3). Amortized analysis needs no such assumptions. Also, it offers an *upper-bound* on the worst case running time of a sequence of operations, and this bound will always hold. An average-case bound, on the other hand, does not preclude the possibility that one will get "unlucky" and encounter an input that requires much more than the expected computation time, even if the assumptions on the distribution of inputs are valid. These differences between probabilistic and amortized analysis therefore have important consequences for the interpretation and relevance of the resulting bounds.

Amortized analysis is closely related to competitive analysis, which involves comparing the worst-case performance of an online algorithm to the performance of an optimal off-line algorithm on the same data. Amortization is useful because competitive analysis's performance bounds must hold regardless of the particular input, which by definition is seen by the online algorithm in sequence rather than at the beginning of processing. Sleator and Tarjan (1985a) offer an example of using amortized analysis to perform competitive analysis.

The basics

The three approaches to amortized analysis

There exist three main approaches to amortized analysis: aggregate analysis, the accounting method, and the potential method (Cormen et al. 2001, p.405). Aggregate analysis is a simple method that involves computing a bound on a sequence of operations, then dividing by the number of operations to obtain the amortized cost. All operations are assigned the same amortized cost, even if they are of different types (e.g., adding one item to a stack or popping multiple items from a stack) (p. 406–10).

The other two methods are somewhat more interesting and powerful, and by reasoning about particular local or global properties of the data structures, they allow for assigning different amortized costs to different types of operations in the same sequence.

The accounting method

Overview

The accounting method, or the “banker's view,” assigns charges to operations as if the computer were coin-operated. One can think of each operation always being accompanied by inserting one or more coins into the computer to pay for the operation, according to a pre-determined charge for that operation type (e.g., a stack push might have one charge, and a pop of several items from the stack might have another charge). The charge does not necessarily correspond to the actual time required for the particular operation; it is possible that the operation will complete in less time than the time charged, in which case some positive accumulation of credit is left after the time required for the operation to finish is subtracted from the operation payment. Or, it is possible that the operation will need more time than the time charged, in which case the operation can be paid for by previously accumulated credit.

In the accounting method, **the amount charged for each operation type is the amortized cost for that type**. As long as the charges are set so that it is impossible to go into debt (i.e., one can show that there will never be an operation whose actual cost is greater than the sum of its charge plus the previously accumulated credit), the amortized cost will be an upper bound on the actual cost for any sequence of operations. Therefore, the trick to successful amortized analysis with the accounting method is to pick appropriate charges and show that these charges are sufficient to allow payment for any sequence of operations.

More formally, denote the actual cost of the *i*th operation by c_i and the amortized cost (charge) of the *i*th operation by $\hat{c}_i$. If $\hat{c}_i > c_i$, the *i*th operation leaves some positive amount of credit, $credit_i = \hat{c}_i - c_i$ that can be used up by future operations. And as long as

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \quad , \quad (1)$$

the total available credit will always be non-negative, and the sum of amortized costs will be an upper bound on the actual cost (Cormen et al. 2001, p. 410).

It is common to keep track of stored credit by assigning it to a particular location in the data structure, for example some part of the structure that will be altered later by some expensive operation that was (partially) made possible by the operation that provided the credit. This location could be a node in a tree, a position in a stack, or something else. Different types of operations that provide credit might store their credit in different structures, but a single operation type will have a single, well-defined location to store its credit. Reasoning about when credit is produced, where it is stored, and how it is used up is key to showing that Equation 1 holds for all operation sequences. Finally, remember that amortization is an analytical device only! The credits “stored” in the data structure are imaginary tools we use to reason about the behavior of an algorithm or data structure, not something that is explicit or accessible in a data structure abstraction or its implementation code.

Example

This first example supplies a trivial illustration of the accounting method, adapted from Tarjan (1985) and Cormen et al. (2001, p.410). Suppose you have a simple stack of items, where you can push an item onto the stack or pop an item from the stack in constant time. The stack is constrained to naturally contain 0 or more items at all times. Suppose there is only one operation type defined for this stack, called `OP`, which involves applying zero or more pops followed by one push:

```
OP(n) {
        //requires at least n items are on the stack
        pop n items from stack
        push 1 item onto stack
}
```

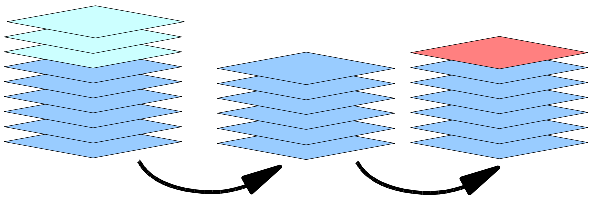

Figure 1: `OP(3)` *applied to a stack*

Also suppose that m such operations have been performed. The worst-case running time for an `OP` operation in the sequence is O(m), which occurs if operations 1 through $m - 1$ have each been `OP(0)` (each popping no items and pushing one), and the mth operation is `OP(m-1)` (popping *all* items from the stack, then pushing one). A simple worst-case analysis would consider m operations with a worst-case per-operation cost of O(m) to have an upper bound of O(m^2).

It should be intuitively obvious that not every single `OP` operation can have a cost of m, since previous operations have to push items onto the stack before they can be popped! Amortized analysis captures this intuition. Consider assigning each `OP` a charge of 2. The first `OP` must be `OP(0)`, popping nothing from the empty stack and pushing one item. The actual cost of this operation is 1, and the amortized cost is 2, so the credit remaining is 2 – 1 = 1. This credit of 1 is stored with the new item in the stack. If this item is popped from the stack in the future, the pop will be paid for by this stored 1 credit. In this way, any sequence of `OP`s maintains the invariant that there is one credit stored per stacked item. The algorithm will never run out of credit, because each pop is already pre-paid, and each push only uses up 1 of the two credits paid the `OP`. The amortized cost of each `OP` is 2, so the amortized cost of any sequence of m `OP`s is $2m$, which is O(m). This is quite an improvement over the O(m^2) analysis!

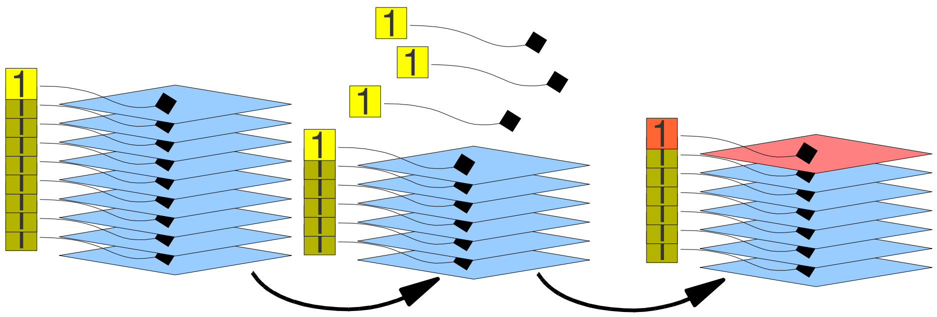

Figure 2: In the accounting method, one credit is stored with each stack item. `OP(3)` *pays for 3 pops using the popped items' credits, pays for the push with one of* `OP`*'s own 2 payments, and stores the other of its 2 payments as a credit tied to the new item.*

If a data structure possesses little or no stored credit, this is an indication that the "badness" resulting from a subsequent particular operation having actual cost that exceeds its amortized cost is limited, as the stored credit cannot fall below 0. On the other hand, if a data structure possesses much stored credit, it might be possible that a subsequent operation may have actual cost that far outweighs its amortized cost, as it may eat up the prepayments of the previous operations.

The potential method

Overview

The potential method, or "physicist's view," defines a function that maps a data structure onto a real-valued, non-negative "potential" (Tarjan 1985; Cormen et al. 2001, p.412). The potential stored in the data structure may be released to pay for future operations. In this way, the potential is similar to considering the total credit stored throughout a data structure using the accounting method. However, the potential method has a few other conceptual differences from the accounting method, as discussed below.

Let us refer to the data structure at time *i* (or alternatively, just after operation *i)*, as D_i. The potential function $\Phi(D_i)$ maps D_i onto a real value. Again, denote the actual cost of operation *i* as c_i .
In the potential method, the amortized cost of operation *i* is equal to the actual cost plus the increase in potential due to the operation:

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

That is, in performing analysis, the amortized cost and potential function must be defined so that this equivalence always holds. This equivalence implies that an operation whose actual cost is less than its amortized cost results in an overall increase in potential, and an operation whose actual cost is greater than its amortized cost requires a decrease in potential. A decrease in potential therefore "pays for" particularly expensive operations, just as a decrease in credits did for the accounting method.

Just as in the accounting method, we must enforce an additional condition to ensure that the total amortized cost of a sequence is always an upper bound on the total actual cost. By equation 2, we can derive a relationship between the total amortized and actual costs:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \qquad (3)$$

Note that all $\Phi(D_i)$ terms except $\Phi(D_0)$ and $\Phi(D_n)$ cancel out, leaving:

$$\sum_{i=1}^{n} \hat{c}_i = \left(\sum_{i=1}^{n} c_i \right) + \Phi(D_n) - \Phi(D_0) \qquad (4)$$

To ensure that the total amortized cost is an upper bound on the total actual cost, for a sequence of any length, we merely must ensure that $\Phi(D_i) \geq \Phi(D_0)$ for all *i*. Typically, $\Phi(D_0)$ is defined to be 0, and the potential is defined so that it is always non-negative (Cormen et al. 2001, p.413).

Example

Consider again the simple stack example introduced above. A natural potential function is $\Phi(D_i)$=*the number of items in the stack after operation* i. Consider performing `OP(k)` as operation *i*, which performs *k* pops and one push. `OP(k)` has an actual cost of $c = k + 1$. If the stack originally contained *n* items, the potential before the operation is $\Phi(D_{i-1}) = n$, and the potential after the operation is $\Phi(D_i) = (n - k + 1)$. The amortized cost of `OP(k)` is therefore:

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (k+1) + (n-k+1) - (n) = 2$. Furthermore, it is clear that the potential defined this way is never negative. Therefore, we can conclude that the amortized cost of `OP` is 2 (O(1)), and the total amortized cost for any sequence of `OP` operations is an upper bound on the actual cost of the sequence.

In practice, potential functions are often, unsurprisingly, a bit more complicated. For example, one might wish to define the potential based on more complicated property of the data structure. Also, it may be necessary to enumerate all the possible outcomes of an operation to show that equation 2 holds for each of them. Such slightly more complicated applications of the potential method appear in the in-depth examples below.

Comparing the accounting and potential methods

The accounting method charges each operation a certain amount according to its type (e.g., insert or delete), putting the focus on each operation's "prepayment" to offset future expensive operations. The potential method instead puts the focus on the effect of a particular operation at a particular point in time, based on the effects of the operation on the data structure and the corresponding change in possibility for future operations to incur expense. The accounting method "stores" credit in particular areas of a data structure, such as nodes of a tree or items in a stack, whereas the potential method generally considers properties of the entire data structure (such as how many nodes are in a tree, or how many items are in the stack) to compute the potential.

The accounting method and potential method are equivalent in terms of their applicability to particular problems and the bounds they provide (Tarjan 1985). However, one or the other may be easier to apply to a given problem. Tarjan suggests that the potential method may be more natural when fractional amounts of time must be considered, while the accounting method is less abstract.

In-depth examples

In this section, we take a step-by-step look at applying amortized analysis to some real problems. The reader is also encouraged to read Chapter 17 of Cormen et al. (2001), which provides additional simple examples.

Move-to-front

The first example applies the potential method to the "move-to-front" (MTF) heuristic for list access and updating, adapted from Tarjan 1985 and Sleator and Tarjan 1985a (the reader is referred to the original sources for more information on the background of this problem, and a very detailed analysis and discussion of the algorithm). Aspects of this example are also drawn from the COS 423 Spring 2000 handout (Tarjan 2000).

The problem MTF addresses is this: Consider a linear list of items (such as a singly-linked list). To access the item in the i^{th} position requires time *i*. Also, any two contiguous items can be swapped in constant time (not including the time to access them in the list). The goal is to allow access to a sequence of *n* items in a minimal amount of time (one item may be accessed many times within a sequence), starting from some set initial list configuration. If the sequence of accesses is known in advance, one can design an optimal algorithm for swapping items to rearrange the list according to how often items are accessed, and when. However, if the sequence is not known in advance, a heuristic method for swapping items may be desirable. This problem is one formulation of the standard "dictionary problem" (Sleator and Tarjan 1985a, p.203).

The MTF is a heuristic that takes advantage of the fact that, for real problems such as paging, if item *i* is accessed at time *t*, it is likely to be accessed again soon after time *t* (i.e., there is locality of reference). MTF works by moving item *i* to the front of the list when it is accessed. This move is performed via successive swaps from position *i* all the way down to position 1. Therefore, when the *i*th item is accessed, the cost is *i* to access the item, plus *i* – 1 to move the item to the first position, for a total cost of 2*i* – 1 (Figure 3).

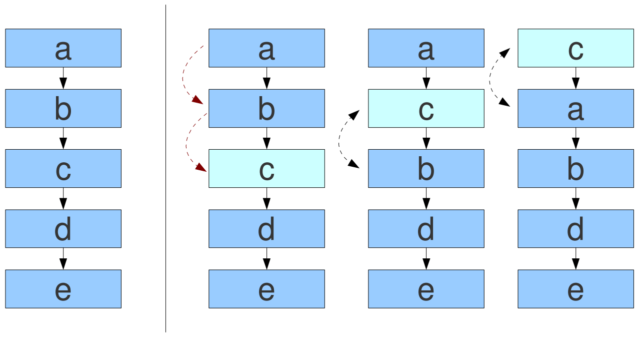

Figure 3: Move-to-front: To access 'c' in the original list (left), walk down from 'a', then move 'c' to front by swapping with 'b' then 'a' (right)

Amortized analysis can be used to show that MTF always performs within a factor of 4 of *any* algorithm – even an optimal algorithm that knows the access sequence in advance, and even without making assumptions about locality of reference. Here, to analyze with respect to an arbitrary (perhaps optimal) algorithm A, define the potential of MTF at time *t* as the two times number of pairs of items whose order in the MTF's list differs from their order in A's list at time *t*. For example, if MTF's list is ordered (*a, b, c, e, d*) and A's list is ordered (*a, b, c, d, e*), then the potential for MTF will be equal to 2, because one pair of items (*d* and *e*) differ in their ordering between A's list and MTF's list. The potential at *t*=0 is 0, as both algorithms begin with the same list by definition. Also, it is impossible for the potential to be negative. Therefore, the total amortized cost is an upper bound on the total actual cost of any access sequence.

Now, consider the analysis of accessing a single item, *x*. Let *x* be at position *k* in MTF's list and at position *i* in A's list. The cost to MTF of accessing *x* and moving it to the front is 2(*k* – 1). The cost for A to access *x* is *i*. Note that moving *x* to the front of the list reverses the ordering of all pairs including *x* and an item originally in location 1 to *k* – 1 (i.e., *k* – 1 pairs in total). The relative positions of all other pairs are unchanged by the move. In A's list, there are *i* – 1 items ahead of *x*; all of these will be behind *x* in MTF's list once *x* is moved to the front. Therefore, there are at most min{*k* – 1, *i* – 1} pair inversions (i.e., disagreements in pair order between MTF and A) that are *added* by the move to front of *x*. All other ordering reversals (at least *k* – 1 – min{*k* – 1, *i* – 1}) must result in pair inversion *removals* (i.e., the pair orderings now agree between MTF and A).

Therefore, the potential *change* incurred in this single access and move to front is bounded above by 2(min{*k* – 1, *i* – 1} – (*k* – 1 – min{*k* – 1, *i* – 1})) = 4min{*k* – 1, *i* – 1} – 2(*k* – 1). By equation 2, the amortized cost of this single access is bounded above as:

$$\hat{c} = c + \Delta\Phi \le 2(k-1) + 4\min\{k-1, i-1\} - 2(k-1) \le 4\min\{k-1, i-1\} \le 4i$$

So, the amortized cost of a single access and move-to-front by MTF is bounded above by four times the cost of the access by A.

One additional consideration is that A might independently perform swaps in response to a new access request, and this is not taken into account above. Say A does swap two items. This incurs no additional actual cost on the part of MTF, but it will increase or decrease the new potential by 2, and it will increase the cost of access for A by 1. The bound on MTF's amortized cost still holds, since the amortized cost is increased by at most 2, but the bound is increased by 4. This is true no matter how many swap operations A performs.

Splay trees

The next example applies amortized analysis to splay trees. This presentation of the example assumes no prior knowledge about splay trees, as they are quite straightforward structures whose essential behavior is explained below. The reader should have a basic understanding of binary search trees, however, and should understand the concept of a tree rotation. (If nothing else, remember that a rotation is a modification to some subtree of the search tree, which preserves the binary search tree order property but has the effect of moving some nodes towards the root and some nodes away from the root by 1. A single rotation can be performed in constant time using a standard binary tree implementation.)

Splay trees are a self-adjusting binary search tree introduced by Sleator and Tarjan (1985b). The values of the nodes are ordered symmetrically, meaning that the left subtree of node *x* contains nodes whose key values are strictly less than *x*, and the right subtree contains nodes whose values are greater than *x*. Splay trees are not balanced trees like red-black trees; their structure may be arbitrarily "bad," (e.g., Figure 7), meaning that an access on a tree with *n* nodes may require *n* operations. However, much like the move-to-front heuristic described above, splay trees move an element to the root of the tree through a series of "splay" steps (rotations). This move has the effect of local "clean-up" along the path from the accessed node to the root, which tends to reduce the depth of the tree, and it is what makes splay trees "self-adjusting." Splay trees have some nice properties in comparison to balanced trees such as red-black trees, for example simpler implementation and analysis. Using amortized analysis, it is possible to show that splay trees are just as efficient as balanced trees for the standard operations such as access, insert, and delete (Sleator and Tarjan 1985b). The following example shows this in a step-by-step analysis, which is compiled from Sleator and Tarjan (1985b) and Tarjan (1985), as well as course materials available online by Wayne (2001), Tarjan and Mulzer (2007), and Karger and Zhang (2005) The reader is encouraged to consult these sources for additional discussion and clarification.

First, let us introduce the `SPLAY(x, S)` operation on some element x and a splay tree, S. This operation has the effect of rotating *x* up to the root of *S* if *x* is in *S*, or else rotating the next biggest or next smallest element to *x* up to the root (depending on the implementation). `SPLAY` works by repeatedly performing rotations. At each iteration of the while loop, two rotations (cases 2a-d) are performed to replace *x*'s grandparent with *x*, or one rotation (cases 1a-b) is performed to put *x* at the root, if *x*'s parent is the root. Figures 4 through 6 illustrate the rotations for cases 2a, 2c, and 1a; cases 2b, 2d, and 1b are symmetric.

```
SPLAY(x, S) {
   while x is not root of S {
      if x's parent is root {
         if x is a left child, do a right rotation (case 1a)
         else x is a left child, so do a left rotation (case 2a)
         //x is now root
      } else {
         if x is a left child and its parent, y, is a left child, do 2 right rotations (case 2a)
         else if x is a right child and y is a right child, do 2 left rotations (case 2b)
         else x is a right child and y is a left child, so do 1 left then 1 right rotation (case 2c)
          else if x is a left child and y is a right child, do 1 right then 1 left rotation (case 2d)
      }
   }
}
```

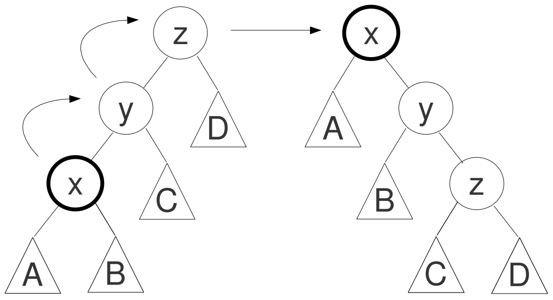

Figure 4: `SPLAY` *case 2a*

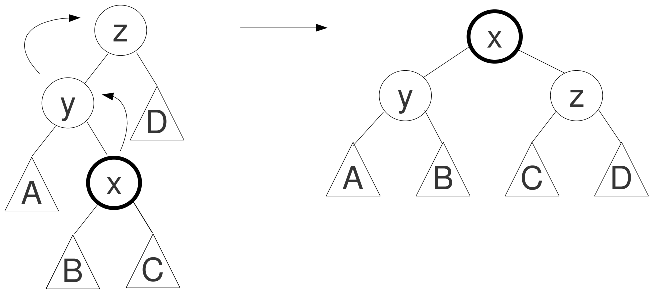

Figure 5: `SPLAY` *case 2c*

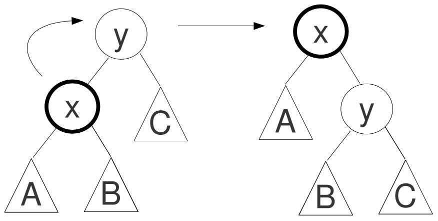

Figure 6: `SPLAY` *case 1a*

`SPLAY` moves *x* up one level in the tree each time a rotation is done. This could result in a total of 0 rotations, in the case where *x* is already the root, or it could be *n* – 1 rotations, when the tree contains *n* nodes in a single chain and *x* is the lowest in the chain (e.g., item 1 in Figure 7). Therefore, a worst-case bound on the `SPLAY` operation is O(*n*) for *n* nodes.

Figure 7 shows the sequence of tree configurations that occur over the while-loop in `SPLAY(1,S)`. Indeed, 9 rotations are performed on the tree with 10 nodes. However, notice that the tree is shallower after this access and splay; there is no way to do another access that will result in `SPLAY` taking 9 operations. In fact, consider the operation `SPLAY(2,S)` on this new tree (Figure 8). This splay only required 5 rotations, and it left the tree even more balanced. Amortized analysis is useful for reasoning about the fact that performing a very expensive operation, such as `SPLAY(1,S)` on the original tree in Figure 7, performs some sort of clean-up that limits the expense of future operations.

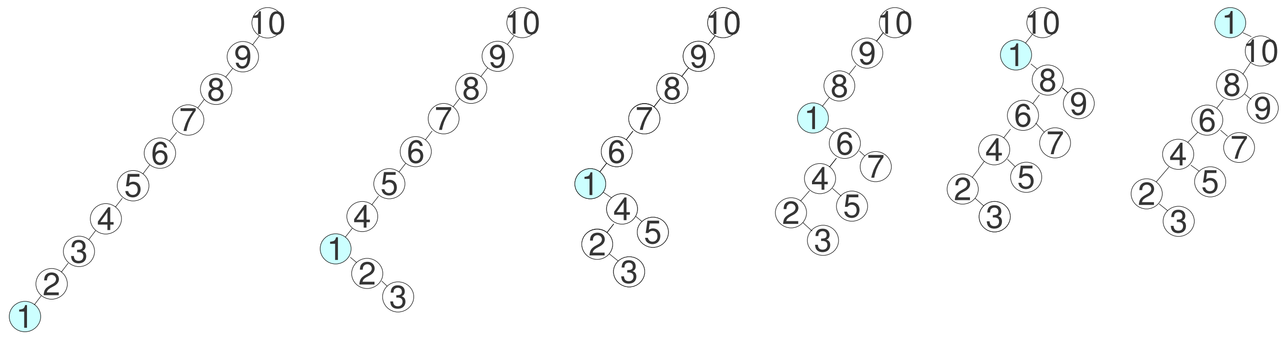
Figure 7: `SPLAY(1,S)` *on a "bad" tree (from Wayne 2001)*

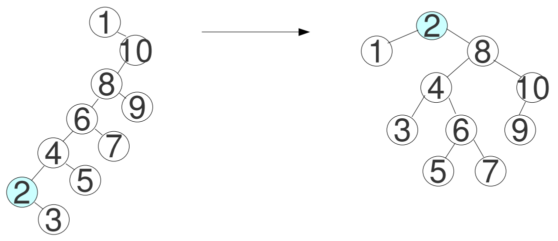
Figure 8: `SPLAY(1,S)` *on tree from Figure 7*

The `SPLAY` operation is called once for each Insert, Delete, Join, and Find operation. To Find element *x* in tree *S*, for example, we first call `SPLAY(x,S)`, to make *x* the root of *S* if *x* is in the tree. The cost of the `SPLAY` operation is therefore crucial to analyzing all other operations.

Now, we define a potential function in order to perform amortized analysis. Let $S_i(x)$ denote the subtree of *S* rooted at node *x* at time *i* (specifically, after the *i*th iteration of the `SPLAY` while-loop, for example), and $|S_i(x)|$ denote the number of nodes in that subtree at that time. Define the "rank" function to be $r_i(x)=\lfloor \log|S_i(x)|\rfloor$. Let $r_i(x)$ indicate the "local" potential of node *x* in the tree *S* at time *i*. Finally, let our overall potential function at time *i* be $\Phi_i=\sum_{x\in tree} r_i(x)$.

It can be shown that the amortized cost of any splay step on node *x* is O(log *n*). To provide this bound, we can enumerate all the possible changes to the tree that a single iteration of the `SPLAY` while-loop might cause. Then, for each case, we compute the potential for the entire tree before and after the loop iteration, and the actual cost required to perform the iteration. This gives us a bound on the amortized cost of a single loop iteration, which we can then sum over all iterations of the loop to obtain a bound on the amortized cost of the entire `SPLAY` operation.

Choose case 2a from `SPLAY` to start with. Figure 4 shows the configuration before and after the two right rotations. The total actual cost of the 2 rotations is 2. The change in potential introduced by these two rotations is the sum of the changes in potential at each node:

$$\Phi_i-\Phi_{i-1}=\sum_{x\in tree} r_i(x)-\sum_{x\in tree} r_{i-1}(x)=\sum_{x\in tree} r_i(x)-r_{i-1}(x)$$

Note that case 2a only changes the potential of nodes *x*, *y*, and *z*: the sizes of all other nodes' subtrees remain unchanged. Furthermore, observe that $r_i(x)=r_{i-1}(z)$, $r_{i-1}(y)\geq r_{i-1}(x)$, and $r_i(y)\leq r_i(x)$. By equation 2, the amortized cost of this single case is:

$$\begin{aligned}\hat{c}_i &= c_i+\Phi_i-\Phi_{i-1} = 2+r_i(x)-r_{i-1}(x)+r_i(y)-r_{i-1}(y)+r_i(z)-r_{i-1}(z)\\ &= 2+(r_i(x)-r_{i-1}(z))+r_i(y)+r_i(z)-r_{i-1}(x)-r_{i-1}(y)\\ &\leq 2+0+r_i(x)+r_i(z)-r_{i-1}(x)-r_{i-1}(x) = 2+r_i(x)+r_i(z)-2r_{i-1}(x)\ . \qquad (5)\end{aligned}$$

We would like to get the dependence on *z* out of this expression somehow. We can accomplish this by taking advantage of the following property of logarithms:

$$\frac{\log a+\log b}{2}\leq\log\left(\frac{a+b}{2}\right)$$

So,

$$r_{i-1}(x)+r_i(z) \;=\; \log S_{i-1}(x)+\log S_i(z) \;\leq\; 2\log\left(\frac{S_{i-1}(x)+S_i(z)}{2}\right)$$

Using the property that $S_{i-1}(x)+S_i(z)\leq S_i(x)$,

$$r_{i-1}(x)+r_i(z) \;\leq\; 2\log\left(\frac{S_i(x)}{2}\right) \;=\; 2\log(S_i(x))-2\log 2 \;\leq\; 2r_i(x)-2$$

so, $r_i(z) \;\leq\; 2r_i(x)-r_{i-1}(x)-2$.

Substituting this back into equation 5, we see that

$$\hat{c}_i \;\leq\; 2+r_i(x)+2r_i(x)-r_{i-1}(x)-2-2r_{(i-1)}(x) \;=\; 3r_i(x)-3r_{i-1}(x) \;=\; 3(r_i(x)-r_{i-1}(x)) \quad (6).$$

The analysis of cases 2b, 2c, and 2d are nearly identical. For cases 1a and 1b, the reasoning is similar, but the actual cost c_i is only 1, not 2, so the bound is $\hat{c}_i \;\leq\; 3(r_i(x)-r_{i-1}(x))+1$.

Now, we have an upper bound on the amortized cost for any individual iteration of the while-loop of SPLAY. The total amortized cost of an entire `SPLAY` operation is $\hat{c}=\sum_{i=1}^{m}\hat{c}_i$ if the while-loop executes m times. Using equation 6, and taking advantage of the telescoping property and the fact that either case 1a or 1b will only be executed once, we get:

$$\hat{c} \;=\; \sum_{i=1}^{m}\hat{c}_i \;\leq\; \sum_{i=1}^{m}(3(r_i(x)-r_{i-1}(x)))+1=3(r_m(x)-r_0(x))+1 \quad .$$

Note that $r_m(x) = n$ because, at time m, x is the root, so all n nodes are in its subtree. Note again that $r_0(x)$ may be as small as 1, if x is at a leaf. Therefore, our amortized bound for one `SPLAY` operation, $\hat{c}$, is O(log n).

Finally, this bound on `SPLAY` can be used to easily show amortized bound on all operations of splay trees. Trivially, the Find(x) operation requires one `SPLAY`, so its cost is also O(log n). Other operations are slightly more involved; the reader is directed to Karger and Zhang (2005), Sleator and Tarjan (1985b), or Tarjan (1983, p.53–6) for more details.

More examples

The examples above require little background knowledge of the data structures or problem area. This section provides brief outlines of the use of amortized analysis in several other algorithms and data structures that are somewhat more involved. This section is provided so that readers familiar with these topics may appreciate the role amortization has played in their analysis and/or design, and so that readers comfortable with the examples presented thus far may consult sources for the following topics in order to learn more.

Red-black trees

Red-black trees are a type of balanced binary search tree that enforces balance by coloring each node in the tree either red or black, then re-adjusting the tree after operations such as insert and delete, according to the colors of the nodes (Cormen et al. 2001, Chapter 13). In simple (unbalanced) binary search trees, basic operations such as insert and delete can run in O(n) time on a tree of size n. For red-black trees, these operations run in O(log n) time. However, there is increased overhead involved in the structural modifications performed to maintain tree balance. In the worst case, any operation can cause O(log n) modifications. However, amortized analysis can be used to show that any *sequence* of m insert and delete operations causes O(m) structural modifications in the worst case (Cormen et al. p.428). It can similarly be shown that the total time for m consecutive insertions in a tree of n nodes is O(n + m) (Tarjan 1985).

Tarjan (1985) uses the accounting method to show this second bound. The analysis shows that the invariant is maintained that every black node in the tree contains 0, 1, or 2 credits, depending on whether it has one red child, no red children, or two red children, respectively. Initially, O(n) credits are added to the tree. The analysis takes the following form: all possible local effects of a rebalancing operation following an insert operation are enumerated. The invariant is shown to hold for all of these cases, where the terminating cases require payment of O(1) credits, and nonterminating cases (which require one or more additional rebalancing operations) use up available credit in the tree to perform their operations. Because each operation includes only one terminating case by definition, each insert operation must pay only a constant amount of credit. Therefore, the overall amortized cost of a single insert operation is O(1), and so the amortized cost of any sequence of m insert operations is O(n (to insert the initial n credits) + m).

Fibonacci heaps
Fibonacci heaps are a heap data structure represented by a forest of heap-ordered trees (Fredman and Tarjan 1987). They allow deletion from a heap with n items in O(log n) amortized time and support other heap operations (insert, find min, meld, decrease key) in O(1) amortized time. Fibonacci heaps have been used as underlying data structures to achieve better theoretical bounds on minimum spanning trees and shortest path problems.

Cormen et al. (2001, Chapter 20) employ the potential method to analyze the running time of Fibonacci heap operations. The potential function used is $\Phi(H) = t(H) + 2m(H)$ where t is the number of trees in the root list of heap H and $m(H)$ is the number of marked nodes in H. With this potential function, it is fairly easy to show O(1) amortized time for each non-delete operation individually, and O(lg n) time for extract-min and delete (see the book for details).

Disjoint sets
Disjoint set data structures maintain a collection of disjoint dynamic sets (Cormen et al. 2001, Chapter 21) and support operations to make a set, joint (union) two sets, and find the set that contains a particular element. The straightforward union algorithm for a disjoint set forest implementation takes O(n) time for n elements, using both amortized and worst-case analysis. However, using the union-by-rank and path compression heuristics, one can show that any sequence of m make, union, and find operations runs in O(m α(n)) time, where α is the slowly-growing Ackermann function. Tarjan performs this analysis using the accounting method (1983, p.24–30), and Cormen et al. (2001, p.513–8) use a potential function. Both analyses require understanding of the details of path compression that lie outside the scope of this paper, but the curious reader is directed to the Tarjan and Cormen textbooks for more detailed treatment.

Maximum flow
The goal of maximum flow problems is to find the maximum flow through a graph, where one node is a source, one node is a sink, and each edge in the graph has a known capacity (Cormen et al. 2001,

Chapter 26). This well-studied problem has several solution approaches. One such approach, called "push-relabel," includes many of the asymptotically fastest solutions to the maximum flow problem (Cormen et al., p.609). An analysis of push-relabel hinges on finding a bound on the number of nonsaturating pushes performed by the algorithm. This is done using a potential function dependent on the heights of the vertices in the graph (Cormen et al., p.678–9).

Dynamic arrays / hash tables
A dynamic table is expanded or contracted as data is added to or removed from it. It is therefore necessary to design and reason about algorithms for managing table size. Using one simple algorithm described in Cormen et al. (2001, p.416–24), a sequence of m operations can be shown to take O(m) amortized time.

Scapegoat trees
Scapegoat trees are another type of balanced binary search tree (Galperin and Rivest 1993). Unlike red-black trees, they do not store additional information at each node. Amortization using a potential function based on the sum of local node potentials yields O(log n) insertion and deletion.

Conclusions

Amortized analysis is a useful tool that complements other techniques such as worst-case and average-case analysis. It has been applied to a variety of problems, and it is also crucial to appreciating structures such as splay trees that have been designed to have good amortized bounds.

To understand the application of amortized analysis to common problems, it is essential to know the basics of both the accounting method and the potential method. The resources presented here supply many examples of both methods applied to real problems.

To perform an amortized analysis, one should choose either the accounting method or the potential method. The approaches yield equivalent results, but one might be more intuitively appropriate to the problem under consideration. There is no magic formula for arriving at a potential function or accounting credit scheme that will always work; the method used depends on the desired bounds and the desired complexity of the analysis. Some strategies that sometimes work, however, include enumerating the ways in which an algorithm might operate on a data structure, then performing an analysis for each case; computing the potential of a data structure as a sum of "local" potentials so that one can reason about the effects of local changes while ignoring irrelevant and unchanging components of the structure; designing the potential method around the desired form of the result (e.g., relating the potential to the log of the subtree size for splay trees, as the desired outcome is a logarithmic bound); and reasoning about each type of operation in a sequence individually before coming up with a bound on an arbitrary sequence of operations.

An understanding of amortized analysis is essential to success in an algorithms course, to understanding the implication of theoretical bounds on real-world performance, and to thoroughly appreciating the design and purpose of certain data structures. The reader is therefore again urged to consult any of the sources mentioned here to improve his or her understanding of amortized analysis and to explore these algorithms in greater depth.

References

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1974. *The design and analysis of computer algorithms*. Addison-Wesley.

Agarwal, P. K, J. Xie, J. Yang, and H. Yu. 2006. "Scalable continuous query processing by tracking hotspots." *Proceedings of the 32nd International Conference on Very Large Databases* 32: 31–42.

Brown, M. R., and R. E. Tarjan. 1980. "Design and analysis of a data structure for representing sorted lists." *SIAM Journal on Computing* 9(3): 594–614.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. 2001. *Introduction to algorithms*, 2nd ed. MIT Press.

Fomitchev, M., and E. Ruppert. 2004. "Lock-free linked lists and skip lists." *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, 50–9.

Fredman, M. L., and R. E. Tarjan. 1987. "Fibonacci heaps and their uses in improved network optimization algorithms." *Journal of the ACM* 34(3): 596–615.

Galperin, I., and R. L. Rivest. 1993. "Scapegoat trees." *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, 165–74.

Georgiadis, L., R. E. Tarjan, and R. F. Werneck. 2006. "Design of data structures for mergeable trees." *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, 394–403.

Huddleston, S., and K. Melhorn. 1981. "Robust balancing in B-trees." 5th GI-Conference on Theoretical Computer Science: *Lecture Notes in Computer Science* 104. New York: Springer-Verlag. 234–44.

Huddleston, S., and K. Melhorn. 1982. "A new data structure for representing sorted lists." *Acta Informatica*, 17: 157–84.

Karger, D., and X. Zhang. 2005. "Splay trees." Scribe notes for September 12, 6.854 Advance Algorithms, MIT. Available: http://courses.csail.mit.edu/6.854/05/scribe/dzhang-splaytrees.ps.

Mao, Y., Y. Sun, M. Wu, and K. J. R. Liu. 2006. "JET: Dynamic join-exit-tree amortization and scheduling for contributory key management." *IEEE/ACM Transactions on Networking* 14(5): 1128–40.

Mendelson, R., R. E. Tarjan, M. Thorup, and U. Zwick. 2006. "Melding priority queues." *ACM Transactions on Algorithms* 2(4): 535–56.

Sleator, D. D., and R. E. Tarjan. 1985a. "Amortized efficiency of list update and paging rules." *Communications of the ACM* 28(2): 202–8.

Sleator, D. D., and R. E. Tarjan. 1985b. “Self-adjusting binary search trees.” *Journal of the ACM* 32(3): 652–86.

Tarjan, R. E. 1983. *Data structures and network algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics.

Tarjan, R. E. 1985. “Amortized computational complexity.” *SIAM Journal on Algebraic and Discrete Methods* 6(2): 306–18.

Tarjan, R. E. 2000. “Amortized analysis of move-to-front (MF) list rearrangement.” Course handout for COS 423, Theory of Algorithms, Princeton University. Available: http://www.cs.princeton.edu/courses/archive/spr00/cs423/handout2.pdf.

Tarjan, R. E., and W. Mulzer. 2007. Scribe notes for February 17, COS 423, Theory of Algorithms, Princeton University. Available: http://www.cs.princeton.edu/courses/archive/spring07/cos423/lectures/lecture4.pdf.

Wayne, K. 2001. “Amortized analysis.” Course slides for COS 423, Theory of Algorithms, Princeton University. Available: http://www.cs.princeton.edu/~wayne/cs423/lectures/amortized-4up.pdf.