4. Amortized Analysis

Just as you can work more efficiently if the items in your office are well organized, an algorithm can operate more efficiently if its data is organized well in memory. This leads us to the study of *data structures*, which we cover in extensive detail over the next six chapters. Expertise in data structures is a fundamental part of any successful algorithm designer's skill set.

Our study of data structures officially began back in Section 1.5, where we introduced several preliminary concepts, including arrays, linked lists, stacks, queues, and importantly, the distinction between the *specification* of a data structure in terms of the operations it should support, and its concrete *implementation*. There are often many ways to implement a particular type of data structure, each with its advantages and disadvantages — for example, an array or a linked list can serve as an implementation of an abstract mathematical sequence.

This chapter discusses *amortized analysis*, a useful method for analyzing data structures with non-uniform performance characteristics (of which there are many). For example, suppose each invocation of a data structure operation usually takes $\Theta(n)$ time, but once in a while it may take $\Theta(n^2)$ time to perform some periodic "housekeeping". Standard worst-case analysis (e.g., saying only "the worst-case running time is $\Theta(n^2)$ per invocation") does not serve our data structure well. It might scare away potential users, since it makes the data structure sound much slower than it actually is. Amortized analysis provides a much more accurate way to characterize performance by looking at worst-case running time averaged over a *sequence* of operations. This simplifies the description of the running time by smoothing away non-uniformity, and also does a much better job of faithfully capturing the true performance of the structure.

4.1 Example: Buffering Writes to Disk

Disk access involves much higher latency than memory access, since we typically need to wait for a spinning magnetic platter to rotate until our data physically lies underneath the read/write head of the disk. At this point, an entire block of data (e.g., 1K worth) is transferred to or from the disk all at once. In this setting, it makes sense to buffer, or "cache" disk accesses. For example, instead of writing a stream of data directly to disk, we write to a temporary buffer in memory that, when full, is sent to the disk with a single block transfer.

Suppose we have an array in memory that holds 100 elements of data. It takes 2000 units of time to write its contents to disk, clearing the array in the process. Our interface to this structure consists of a single operation, *write*, which inserts one element of data into the array. Each invocation of *write* takes only 1 unit of time, except every 100th invocation takes 2000 extra units of time (2001 in total).

Since worst-case invocations happen infrequently, we get a much more accurate understanding of performance by looking at worst-case running time amortized (i.e., averaged) over a sequence of invocations, rather than considering only the worst-case running time of a single, isolated invocation:

> The amortized running time of an operation is $f(n)$ if any sequence of invocations (say, k of them) requires at most $kf(n)$ time.

In our example above, *write* runs in 21 units of amortized time, since any sequence of k *write* operations involves $\lfloor k/100 \rfloor \leq k/100$ "expensive" invocations, and therefore takes at most $k + 2000(k/100) = 21k$ total time.

"Amortized" Versus "Worst-Case" Data Structures. Suppose we have a choice between structure A, supporting *write* in 21 amortized units of time, and structure B, supporting *write* in 21 units of time in the worst case. We typically call A an "amortized" data structure and B a "worst-case" data structure[1]. The worst-case data structure may look much more appealing, since it avoids the occasional slow invocation we might experience with an amortized structure. However, this difference would only be noticeable in a "real-time" setting where the response time of every single invocation is crucially important. For example, a data structure for searching a library catalog in response to queries over the web should answer every query as quickly as possible. However, from the perspective of the total running time of a larger algorithm using the *write* operation, there is *absolutely no difference* whether we use A or B. Since the algorithm makes a sequence of calls (say, k of them) to the *write* operation, it will spend at most $21k$ total units of time in either case. This is why amortized data structures are so useful. They are often far easier to design and implement than their worst-case counterparts, and they are every bit as good when serving as building blocks for a more sophisticated algorithm.

4.2 Methods for Amortized Analysis

The simplest way to perform amortized analysis, known as *aggregate analysis*, comes directly from the definition of amortized running time above. We first determine the worst-case total running time for an arbitrary sequence of k invocations, and then divide by k. Unfortunately, this technique generally only works well for very simple data structures, where it is easy to figure out the total running time for an arbitrary sequence of invocations. For most non-trivial data structures, it can be quite hard to make this calculation, especially when the sequence contains several

[1] Be careful not to let this nomenclature mislead you — remember that amortized bounds still give us worst-case guarantees, just over a sequence of invocations instead of for a single invocation.

different types of operations (e.g., *insert* and *delete*) interleaved in an arbitrary fashion.

4.2.1 Accounting Tricks with Running Time

As a financial term, amortization refers to paying off a large amount of money in small installments rather than as a single lump sum. If you need to pay $12,000 at the end of each year for a mortgage on a house, you might set aside $1,000 each month leading up to the due date of the payment, thereby amortizing the large payment into 12 smaller installments that are perhaps easier to manage from a bookkeeping standpoint. You still pay the same amount of money in the end. The only difference is that you account for some payments earlier than they are due (later is not allowed, since we would go into debt). Amortization in data structures is similar. We can view it as an accounting trick where we account for running time earlier than it actually happens, by mentally overcharging ourselves during earlier cheap invocations to generate a "credit" that is redeemed later to pay for expensive invocations.

In terms of the disk buffer example, we know that after 100 *write* operations we will need to make a "payment" of 2000 units of running time to flush the contents of the buffer to disk. In anticipation of this, let us charge each *write* operation 20 extra units of running time, so that by the 100th operation we will have built up 2000 units of credit, exactly enough to pay for the expensive invocation of *write*. We get the same 21-unit amortized cost here for *write* as we did with aggregate analysis, although our understanding of this cost is now based on management of credit through our accounting scheme. For a "cheap" invocation, 1 unit goes toward the actual immediate cost of the *write*, and 20 units is invested as a credit. The author likes to picture a $20 bill sitting on every element in our buffer that represents a prepaid credit of 20 units of work that we can draw upon when needed. When we reach an "expensive" invocation, we charge ourselves the same 21 units, and now we can redeem all the credit sitting in our data structure to pay off the 2001 units of work required for this invocation. By smoothing out its non-uniformity, we now get a much simpler description of the running time of *write*: 21 units of amortized cost for every invocation.

It is important to remember that *the data structure is not changing.* We are only playing accounting games with the analysis, accounting for some work slightly earlier than it happens. Just as in the financial example, we are only allowed to account for work earlier than it actually happens, never later (i.e., we are only allowed to build up a credit, not a debt). This ensures that for any sequence of invocations, the sum of the amortized running times we charge ourselves will properly bound the actual running time of the sequence. Our amortized running times may be "fictitious" numbers, but they still give legitimate worst-case performance bounds.

The approach above — leaving credits in certain places in our structure to pay for anticipated expensive operations — is sometimes known as the *accounting method.* In the simplest possible case, we might overcharge the operation of inserting an element into a data structure for all future work we will spend on that element. In slightly more complicated cases, we may leave behind small amounts of credit in areas of a data structure any time they are modified, in order to pay for expensive periodic "housekeeping" operations. Since we generally only need to perform expensive restructuring on areas of the data structure that have been heavily modified, these areas would presumably contain large amounts of credit.

4.2.2 Potential Functions

A more formal and mathematically rigorous way to perform amortized analysis is using a *potential function*, a function of the current state of our data structure that tells us the total amount of credit currently stored in the entire structure (this use of potential functions is similar to, but slightly more sophisticated than what we saw in Section 3.2.3, when we first introduced the use of potential functions to help compute running time). While the accounting method keeps track of credits on individual elements or in specific parts of a data structure, the potential function method lumps all of this into a single number. In our amortized buffer example, we could define a potential function $\phi = 20x$, where x denotes the current number of elements in the buffer[2]. The potential function ϕ expresses the same idea as with our earlier accounting method analysis, namely that each element in the data structure contributes 20 units of credit. However, we no longer think of leaving 20 units of credit on a particular element (e.g., a \$20 bill), but rather we think of paying 20 units towards the total potential of the structure (e.g., \$20 in the bank).

Amortized analysis is all about accounting for the cost of certain invocations earlier than they actually occur. This results in two types of invocations: "cheap" invocations that we overcharge in order to contribute credit into the data structure (so the amortized cost of the invocation is higher than its true immediate cost), and "expensive" invocations that we pay for using stored-up credit (so the amortized cost is lower than the true cost). Both of these ideas are captured in the following equation, which gives us an easy formulaic way to compute the amortized cost of each invocation of an operation once we have defined a potential function ϕ:

$$a_j = c_j + (\phi_j - \phi_{j-1}),$$

where a_j is the amortized cost of the jth invocation in some sequence of invocations, c_j is the true (immediate) cost of this invocation, and ϕ_j denotes the value of our potential function right after the jth invocation (so $\phi_j - \phi_{j-1}$ is the change resulting from the jth invocation).

To illustrate how easy this formula makes our analysis, consider the example of the amortized buffer. We can figure the amortized cost of *write* by considering two cases. For a "cheap" invocation, we have

$$a_j = c_j + (\phi_j - \phi_{j-1}) = 1 + (20) = 21,$$

since the buffer increases in size by one element, leading to a 20 unit increase in potential on top of the one unit of true cost we pay for the invocation. Here, we

[2]One should always be able to determine the potential associated with a data structure by inspecting its present state, knowing nothing about the history of how we reached this state. Potential functions should therefore be defined only in terms of the current state of the data structure, not in terms of the historical sequence of operations we called to reach this state. For example, we should avoid defining the potential function above as "20 times the number of write operations since the last time the buffer was flushed".

are overcharging the current operation so that we will have enough credit in our potential function to pay for future "expensive" invocations. When we reach such an "expensive" invocation, we then have

$$a_j = c_j + (\phi_j - \phi_{j-1}) = 2001 + (-1980) = 21,$$

since the buffer decreases in size by 99 elements, leading to a $99 \times 20 = 1980$ unit decrease in potential that offsets the high actual cost of emptying the buffer. In both cases, we obtain an amortized cost of 21, matching exactly the result we obtained using aggregate analysis or the accounting method. If we design our potential function correctly, our amortized costs usually balance out like this in the end.

The key to a successful potential function analysis is, not surprisingly, the design of a good potential function. Usually, we can do this by simply summing up the total credit we would get using the accounting method, but occasionally we will find a data structure for which the design of a good potential function requires a bit of divine inspiration (perhaps the best example being splay trees in Chapter 6). In addition to giving us the amortized costs we want (using the formula above), a potential function must always satisfy two important conditions: it must be non-negative (otherwise we would go into "debt", having accounted for work later than it actually happens), and it must be zero initially. If these two conditions hold, then the total amortized cost of any sequence of k invocations is given by

$$\sum_{j=1}^{k} a_j = \sum_{j=1}^{k} (c_j + \phi_j - \phi_{j-1}) = \left(\sum_{j=1}^{k} c_j\right) + \underbrace{\phi_k}_{\geq 0} - \underbrace{\phi_0}_{=0} \geq \sum_{j=1}^{k} c_j,$$

so our fictitious amortized running times do indeed give us a proper upper bound on the total running time actually spent by the sequence of invocations.

Now that we are familiar with how to perform amortized analysis, we will work through a number of increasingly sophisticated examples for the rest of the chapter.

4.3 Example: The Min-Queue

Suppose we wish to augment a standard FIFO queue so it supports the operation *find-min*, which identifies the element of minimum value in the queue. It is important that we can only locate this element but not remove it, since otherwise we would be dealing with a priority queue, a somewhat more ambitious undertaking (the entire next chapter is devoted to priority queues). This structure, which we call a *min-queue*, should ideally support the operations *enqueue*, *dequeue*, and *find-min* all in $O(1)$ time. While it is difficult (albeit possible) to accomplish this in a worst-case setting, it ends up being reasonably simple to implement a min-queue so that all operations run in $O(1)$ amortized time.

The min-queue is our first example of an amortized data structure where several different operations participate in the amortized analysis. In this case, the definition of amortized running time extends in a natural fashion. For example, with two operations A and B, we would say that A has amortized running time $f(n)$ and B has amortized running time $g(n)$ if any sequence of a invocations of A and b invocations of B requires at most $af(n) + bg(n)$ time. When multiple operations

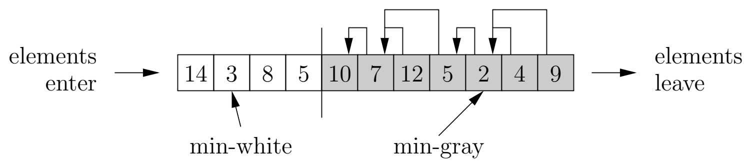

FIGURE 4.1: One way to implement a min-queue.

work together in an amortized data structure, our amortized analysis follows the same general idea as before: earlier cheap invocations of certain operations are overcharged to build up sufficient credit to pay for expensive invocations of other operations later on.

There are several ways to implement a min-queue. We describe one of them here, others[3] in Section 4.5 and problems 70, 71, and 133. Suppose we maintain the elements in our queue in either a circular array or a doubly-linked list, no different than with a standard queue. As shown in Figure 4.1, new elements entering the back of the queue (the left side) are colored white, and elements being removed from the front (the right side) are colored gray. In general, if we scan from the back of the queue forward, there will be a block of white elements followed by gray elements. We maintain a pointer to the current minimum white element and the current minimum gray element, so the *find-min* operation is easy to implement in $O(1)$ time by comparing these two elements. Finally, each gray element maintains a pointer to the smallest gray element behind it in the queue, so if we happen to remove the minimum gray element, we can use its associated pointer to find the new minimum gray element. The only problem with this approach is that eventually we might remove all the gray elements, leaving only white elements. If we then happen to remove the minimum white element, we have no efficient means of locating the new minimum white element, since white elements are not equipped with extra pointers like gray elements. Therefore, if a *dequeue* operation ever finds itself trying to remove a white element, it first executes a $\Theta(n)$ *recoloring* step, where it scans through the entire contents of the queue from back to front, coloring every element gray and initializing its extra pointer.

Every min-queue operation takes $O(1)$ time, except for the occasional slow invocation of *dequeue* that runs in $\Theta(n)$ time due to the need to recolor. However, since an element will participate in at most one recoloring during its lifetime in the structure, we can overcharge the *enqueue* operation an extra $O(1)$ worth of time to pay for this eventual cost in advance. This example illustrates the power and simplicity of the accounting method: we simply overcharge each *enqueue* operation by an extra $O(1)$ worth of time, so every white element will have an $O(1)$ credit associated with it. These credits are sufficient to pay for the eventual $\Theta(n)$ operation of recoloring the min-queue when it contains n white elements. It is quite straightforward to convert this idea into a more formal analysis using potential functions. [Further details]

[3]Some might consider it overkill to include discussion of five different techniques for a relatively minor data structure like a min-queue, but the author strongly believes that there is much insight to be gained from looking at the same problem or structure in multiple different ways.

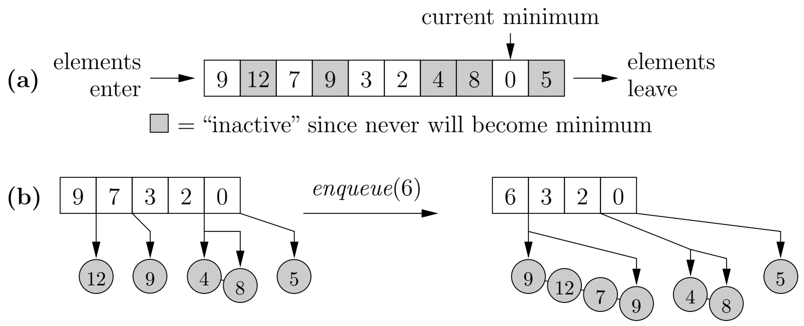

FIGURE 4.2: An alternative way to implement the min-queue. The logical contents of the queue are shown in (a), with an element shaded if it is "inactive", having no hope of it ever becoming the minimum (due to a smaller element being behind it). In (b), we see the actual representation of the min-queue in memory: an array/list of all active elements in descending order, where between each pair is attached a list of inactive elements. When a new element is enqueued (with value 6, in this example), it is added at the end of the array and the block of elements in front of it having larger keys (the 9 and 7 in this case) are deactivated.

Problem 70 (An Alternative Min-Queue). In this problem we analyze another simple way to implement the min-queue, shown in Figure 4.2. We store elements in a circular array or linked list, as with a standard queue. Every time we enqueue a new element e, we scan forward from e and remove all elements ahead of e that are larger than e (since, due to e's presence, these elements have no chance of ever becoming the minimum). As a result, the contents of the queue will be decreasing, making it easy to locate the minimum element (this also guarantees that the set of elements deleted during an *enqueue* operation will form a contiguous block). For example inserting 6 into a queue containing $9, 7, 3, 2, 0$ would "shortcut out" the 9 and 7, leaving $6, 3, 2, 0$. However, in order for *dequeue* to work properly, this process cannot just delete elements; rather, we "deactivate" them by moving them into linked lists between the active elements in our queue, so the FIFO order of all elements in the queue is preserved. Show that this implementation gives us an $O(1)$ worst-case running time for *dequeue* and *find-min*, and an $O(1)$ amortized running time for *enqueue*. [Solution]

Problem 71 (Building a Min-Queue From a Pair of Min-Stacks). Please show how to easily build a *min-stack* (a stack supporting *push*, *pop*, and *find-min*) with all operations taking $O(1)$ worst-case time. Then, show how to use a pair of "back to back" min-stacks to build a min-queue. Further, show that this approach can be extended to allow for insertions and removals in $O(1)$ amortized time at *both* ends of the queue, with *find-min* still taking $O(1)$ worst-case time. [Solution]

Sloppiness in Constants in Amortized Analysis. When we perform amortized analysis, we often temporarily suspend the use of $O(\cdot)$ notation for convenience. For example, in the min-queue we might say that a recoloring operation on n elements takes exactly n units of time, and that we overcharge each *enqueue* operation by 1 extra unit of time to pay for this cost. To be fully precise, we should really say that recoloring takes $\Theta(n)$ time and that the enqueue operation is overcharged by some constant c units of time, where c is the hidden constant in the $\Theta(n)$ time bound.

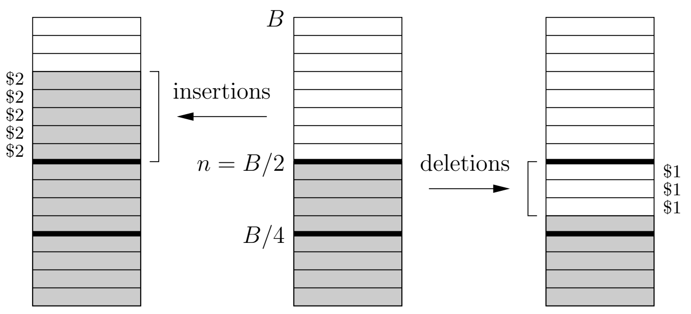

FIGURE 4.3: The middle block depicts the "equilibrium" state right after an expansion or a contraction, when $n = B/2$. Subsequent insertions (on the left) are overcharged by 2 units so they each contribute 2 units of credit (shown here as \$2 bills). When our memory block fills up, it will therefore contain at least B units of credit, which is sufficient to pay for the cost of transferring its B elements during the next expansion. Subsequent deletions (on the right) are each overcharged 1 unit (shown as \$1 bills placed on the slot left open by a deleted element), so that when n drops to $B/4$ we will have sufficient credit to pay the $B/4$ units required to transfer the contents of the block during the next contraction.

Unfortunately, this becomes awkward very quickly when we have many operations to consider (and hence many different constants to manage), so the simplest solution is typically just to pretend that every constant-time operation takes precisely 1 unit of time. The resulting bounds will still end up being asymptotically valid.

Being Lazy. Amortized data structures often gain simplicity by being "lazy", only updating their internal state in one efficient batch operation at the last possible minute rather than investing extra work keeping it perfectly up-to-date all the time. With the min-queue for example, we don't worry about maintaining extra augmented information for the white elements until this becomes absolutely necessary. This lazy outlook is quite common in data structure design.

4.4 Example: Dynamic Memory Allocation

We often don't know in advance the total number of elements that we will end up storing in a data structure. This is rarely a problem for "pointer-based" structures such as linked lists and binary search trees (Chapter 6), where each element of data lives in its own individually-allocated block of memory. The positions of these blocks within the memory as a whole are not so important, since they are linked together in an appropriate fashion using pointers. In contrast, "array-based" data structures such as hash tables (Chapter 7) and of course, arrays, are somewhat more difficult to manage. These data structures must fundamentally be stored in a single contiguous block of memory, and if this block fills up, it may not be possible to expand it since memory on either side may have already been allocated for other purposes. In this situation, we usually allocate a new, larger memory block (typically twice as large), and transfer the entire contents of our data structure into the new block. This is a costly operation to perform, but since it occurs infrequently, we shall see that it behaves well from an amortized perspective.

Memory Allocation. Before we proceed, let us say a few words on the subject of memory allocation. The underlying mechanism used by an operating system to allocate blocks of memory can be quite complicated, and it is not something we want to worry about too much as algorithm designers. For simplicity, we generally assume that all memory allocation requests require only $O(1)$ time (this assumption is perhaps a bit optimistic, but not unreasonable in practice). Therefore, when an algorithm requests a block of memory of some specified size B, it receives in $O(1)$ time a pointer to some contiguous block of B words of memory that are not initialized in any way. To initialize the block, we could spend $\Theta(B)$ time, or we could use the virtual initialization result of problem 2 to effectively reduce the initialization time to $O(1)$ (although this is rarely done in practice, due to the extra overhead required).

Block Expansion and Contraction. Suppose we are storing an n-element stack within an allocated block of memory of size B (a stack gives us a simple concrete example, but the technique here is quite general and applies to most array-based structures). We would like to maintain the property that $B = \Theta(n)$. In other words, we should always use only the amount of memory we actually need, at least to within some constant factor. The prototypical method for achieving this result is the following:

- If n grows as large as B due to insertions, we perform an *expansion* by transferring the structure into a newly-allocated block of size $2B$.
- If n drops down to $B/4$ due to deletions[4], we perform a *contraction* by transferring the structure into a newly-allocated block of size $B/2$.

It is easy to analyze the amortized performance of this scheme using the accounting method, as explained in Figure 4.3, and also to formalize this analysis using potential functions. [Details]

The expansion / contraction technique above contributes only an additional $O(1)$ amortized time to the running time of *insert* and *delete*, so as long as you are content with adding "amortized" quantifiers to your running times, it is generally safe to assume that any n-element array-based data structure can be modified so that its underlying block of memory expands and contracts as necessary to ensure that the data structure always occupies $\Theta(n)$ space. In fact, as we see in the next problem, we can sometimes even achieve this result in a worst-case setting.

Problem 72 (De-Amortization). In this problem we develop a systematic way to remove amortization from certain types of data structures, converting amortized bounds into worst-case bounds. Consider maintaining a queue on n elements, implemented using a circular array stored in a memory block that doubles or halves in size according to the

[4]One might initially suggest halving the block size when n drops to $B/2$ rather than $B/4$, but this does not work well. If $n = B - 1$, then the next insertion results in an expansion, after which $n = B/2$ and a deletion would cause a contraction, after which $n = B - 1$ and an insertion would cause an expansion, and so on. Each of the insertions and deletions in this alternating sequence takes $\Theta(n)$ time, which is far worse than the $O(1)$ amortized bound we are trying to obtain. Amortization is a wonderful technique, but it cannot magically reduce the running time of an operation that genuinely takes $\Theta(n)$ time per invocation.

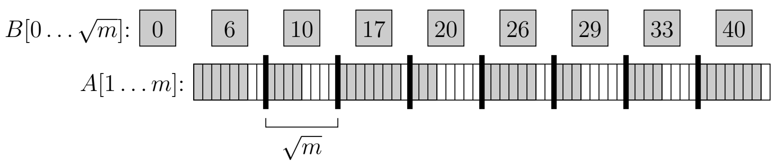

FIGURE 4.4: Illustration of our ordered file maintenance structure.

approach above, so *enqueue* and *dequeue* run in $O(1)$ amortized time. Can you show how to modify the structure so that these become $O(1)$ worst-case bounds? As a hint, keep two copies of the structure, one active and the other to help with rebuilding. You may want to consider initially only the "incremental" case (without the *dequeue* operation). [Solution]

Problem 73 (Ordered File Maintenance). In this section, we addressed the difficulties posed by arrays when they need to expand or contract. Here, we tackle another drawback of arrays — their inability to support efficient insertion or deletion of *interior* elements, as this involves shifting around large amounts of memory to keep the array contiguous. Perhaps the best resolution of this problem will arrive in Chapter 6, when we see how to achieve $O(\log n)$ time for *insert* and *delete* by encoding a sequence in a tree. However, as an interesting exercise, let us consider what happens if we relax the constraint that an array must be contiguous. This leads to what is sometimes called an *ordered file maintenance* structure where we leave periodic "gaps" in an array, much like your library leaves periodic gaps in its shelving so that new books can be added with minimal re-shelving of existing books. As shown in Figure 4.4, suppose we break a length-m array A into $\sqrt{m}$ consecutive blocks of size $\sqrt{m}$, each of which behaves like the expanding / contracting array outlined above, holding between $\sqrt{m}/4$ and $\sqrt{m}$ elements (ideally $\sqrt{m}/2$ elements). The only difference is that when one of these blocks becomes too full or too empty, we rebuild the *entire* structure into a new freshly allocated memory buffer, rebalancing the allocation of its n elements among the blocks by setting $m = 2n$ so that each block is half full. Observe that $m/4 \leq n \leq m$ always holds, so we are still space-efficient to within a constant factor ($m = \Theta(n)$). In an auxiliary array $B[0 \ldots \sqrt{m}]$, we maintain in $B[j]$ the total number of elements stored in blocks $1 \ldots j$ (so $B[j] - B[j-1]$ is the number of elements stored in just block j).

(a) Since A contains gaps, it is no longer possible to perform random access — jumping directly to the ith non-gap element — in constant time. However, show that using the information in B, we can still locate the ith non-gap element in $O(\log n)$ time. Further, show that if we maintain the elements in A in sorted order, we can still "binary search" for the element of a specific value in $O(\log n)$ time. [Solution]

(b) Show how to *insert* and *delete* in only $O(\sqrt{n})$ amortized time[5] (using the result from (a) to find the desired location for the insertion/deletion). [Solution]

We can regard this structure, with $O(\sqrt{n})$ blocks of size $O(\sqrt{n})$, as a 2-level hierarchical decomposition of our array. By increasing the number of levels of hierarchy to the point where we essentially have a binary tree built on top of our array, a fancier generalization of the gap rebalancing approach above leads to $O(\log^2 n)$ amortized running times for *insert* and *delete*; see the endnotes for further details.

[5]Here is a subtle point, as this is the first non-constant amortized time bound we have seen: "n" usually denotes the size or number of elements in a data structure. An $O(n)$ amortized bound might seem somewhat ambiguous, since it refers to an average over a sequence of invocations during which n may change. To remedy this, in non-constant amortized bounds, we usually take n to represent an upper bound on the structure's size during the period of analysis.

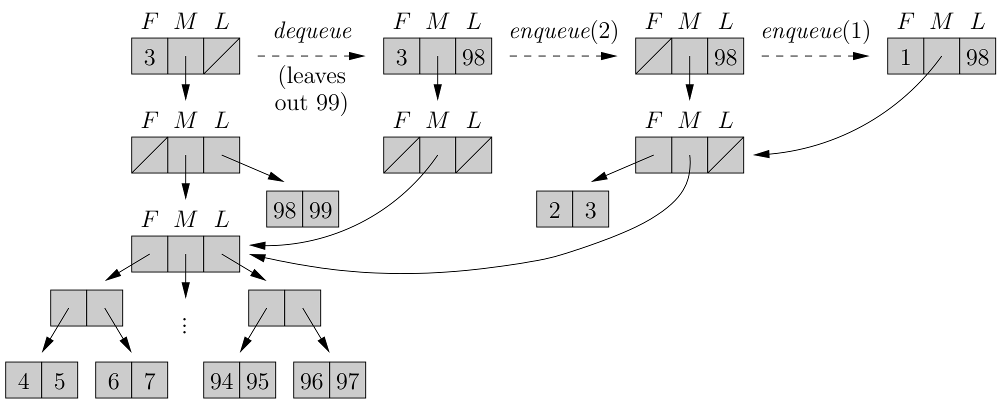

FIGURE 4.5: Storing the numbers 3 through 99 in sequence in a functional queue (on the left), after which we process several operations, leaving a queue storing the numbers 1 through 98 (on the far right). The innermost part of the queue, holding 8 . . . 93, is omitted from the picture. Note that parts of the final queue on the right still refer to objects from earlier version of the queue that remained unchanged — this is typical in functional data structures.

4.5 Example: Functional Queues

Functional programming, in languages like Lisp, Haskell, or OCaml, differs substantially from *imperative* programming in languages like C or Java. In a functional setting, objects in memory typically cannot be modified once created, and consequently we get the nice property that a function called with the same values always yields the same result (by contrast, a C function depending on external global variables might return different results if the global state changes). Functional algorithm design involves a much greater use of recursion, since iteration usually requires reassigning a loop variable again and again. The reader is highly encouraged to give functional programming a try, since this provides a very different mental perspective on algorithmic thinking that can dramatically sharpen your skills at recursion.

Rather than using arrays and random access, functional data structures are almost exclusively "pointer-based", often built from nothing more than pairs of objects[6]. A list can be built using a series of nested pairs (first element, (second element, (third element, ...))). This representation as a pair of (first element, rest of list) is ideal for recursive manipulation. For example, if $f(L)$ is a function summing the contents of list L, then f can just return first$(L) + f($second$(L))$, where first and second refer to the first and second elements of the pair stored in L. If S points to a list representing a stack, we can push a new element onto the stack by returning (new element, S), and we can pop an element by returning second(S). Note that it is simple to work with k-tuples of elements as well as pairs, since these can be built from composition of a small number of pairs.

[6]More precisely, we should say pairs of *pointers* to objects. Since objects don't change after creation, a pointer is perfectly suitable as a stable reference to an object. It is much cheaper in terms of memory to store multiple references to the same object as pointers; making explicit copies of an object is never necessary, since the original object in memory will never change.

While the (first element, rest of structure) schema works quite well for "one-sided" data structures like stacks, we need to be more careful with "two-sided" structures like queues, where interaction occurs at both ends. For example, we might consider storing a queue as a 3-tuple $(F,\ M,\ L)$, where F and L are the first and last elements, and M points to the middle contents of the queue, recursively stored as another of these 3-tuples. This gives a simple recursive method to *enqueue* (insert on the left) a new element x, by returning $(x,\ enqueue(F,\ M),\ L)$. Unfortunately, this takes $\Theta(n)$ time, since the recursion propagates through the entire structure.

We achieve better performance by introducing a small amount of "slack", allowing F and L to hold zero or one elements. We also make M a recursive queue of *pairs* of elements (so its inner structure is a queue of pairs of pairs of elements, and so on). To enqueue x, we now return (x, M, L) if F is empty, or otherwise if F already contains an element (say, y), then we return (empty, $enqueue((x, y),M)$, L), propagating the operation so that we now recursively enqueue the pair (x, y) into M. The *dequeue* (delete from right) operation is symmetric, possibly causing a recursive dequeue from M. Pictures of these operations are shown in Figure 4.5.

Although recursion might propagate quite far into the structure as before, our added slack causes lengthy propagation to happen infrequently, making *enqueue* and *dequeue* both run in only $O(1)$ amortized time [Details]. By maintaining an additional "current minimum" field along with each (F, M, L) triple, we get yet another means of building a min-queue with constant amortized performance bounds.

Problem 74 (Functional Double-Ended Queues). Suppose we want to be able to insert and delete at *both* ends of our queue. Show that the structure above no longer maintains its constant amortized performance bounds when used in this context, but that if we allow F and L to hold between zero and *two* elements, then we once again can use the structure in a way that provides $O(1)$ amortized guarantees. [Solution]

Recursive Slowdown. We can think of many data structures as being built in "levels" (in the example above, from the outermost level inward), where operations start at the first level and occasionally propagate to higher levels. As long as we can limit the rate of propagation between successive levels, amortization generally works. For example, note how our added slack causes the *enqueue* operation to propagate to the next level only once in every two invocations. This is effectively like passing along only 1/2 a unit of work to level $x + 1$ for every unit of work processed at level x, and since $1/2 + 1/4 + \ldots = 1$, each unit of original work therefore propagates to at most 2 total units of work across all levels. This notion is sometimes known as *recursive slowdown*, and it is a common feature of many amortized structures[7].

Prototypical examples of recursive slowdown are shown in the following problems, where we repeatedly increment a binary counter or a permutation. Each increment causes a change to the least-significant digits, and these changes propagate but with increasingly slower frequency to more significant digits.

[7]A mechanical analog of this idea appears in a fun kinetic sculpture by artist Arthur Ganson, where a motor turns the first of a series of twelve gears that each turn progressively slower, the final gear being embedded in a concrete block. As in our data structures, plenty of work is happening on one end, but very little of it propagates to the other side.

Problem 75 (Enumerating Subsets by Incrementing a Binary Counter). Suppose we store the digits of an n-bit binary number in an array A of length n. We wish to increment our number from 0 to $2^n - 1$, updating A accordingly. For example, if $n = 3$, we start with $A = (0, 0, 0)$, then we toggle the last entry to obtain $A = (0, 0, 1)$, eventually reaching $A = (1, 1, 1)$ after several increments. Please describe how to implement an *increment* operation on A that runs in only $O(1)$ amortized time[8] (assuming the entries in A all start out at zero). One motivation for this problem is to quickly enumerate all 2^n subsets of an n-element set $S = \{1, 2, \ldots, n\}$. We can represent each subset with a length-n array of zeros and ones (known as an *incidence vector*), where a one corresponds to an element present in the set and a zero corresponds to an element absent from the set. By incrementing our binary counter, we can enumerate through all successive subsets of S in $O(1)$ amortized time per subset. [Solution]

Problem 76 (Enumerating Permutations). Consider a length-n array $A = (1, 2, 3, \ldots, n)$. We would like to step through all $n!$ permutations of A, updating the array as we go to represent each subsequent permutation. Permutations should be generated in lexicographic order; for example, with $n = 3$ we start with $A = (1, 2, 3)$, then move to $A = (1, 3, 2)$, $A = (2, 1, 3)$, $A = (2, 3, 1)$, $A = (3, 1, 2)$, and finally $A = (3, 2, 1)$. Please describe how to implement an operation *next-perm* with $O(1)$ amortized running time that modifies A to produce the next permutation in lexicographic order[9]. [Solution]

4.6 Example: Disjoint Sets

Disjoint set data structures (also sometimes called *union/find* structures) play an important role in a surprising number of algorithms, perhaps most notably Kruskal's algorithm for the minimum spanning tree problem (Section **??**). They maintain a collection of n elements partitioned into non-overlapping sets (so each element belongs to precisely one set), and support the operations:

- *Make-Set*(e). Add a new element e, belonging to a single-element set $\{e\}$.
- *Find-Set*(e). Return a value identifying the set containing element e. This is usually a pointer to some "canonical" element in the set. Typically, we use *find-set* to determine if two elements e and e' are in the same set by checking if *find-set*(e) = *find-set*(e').
- *Union*(s_i, s_j). Join together sets s_i and s_j, so all elements formerly belonging to these two sets now belong to a single larger set. The sets s_i and s_j are usually specified by providing their canonical elements.

4.6.1 List-Based Implementations

There are several simple ways to implement a disjoint set data structure. For example, if we store all the elements in one large array or list, we can augment each

[8] *Gray codes* (problem **??**) give a method for generating all 2^n binary numbers in an $O(1)$ worst-case setting, where each successive number differs from the former in a single digit. We also mention in the endnotes an amusing way to obtain an $O(1)$ worst-case mechanism via the introduction of a fictitious "2" digit, which while "cheating" in some sense, still has useful applications.

[9] *Heap's algorithm* gives a method for generating all $n!$ permutations in an $O(1)$ worst-case setting, where each successive permutation differs from the former by a single swap. See the endnotes for references.

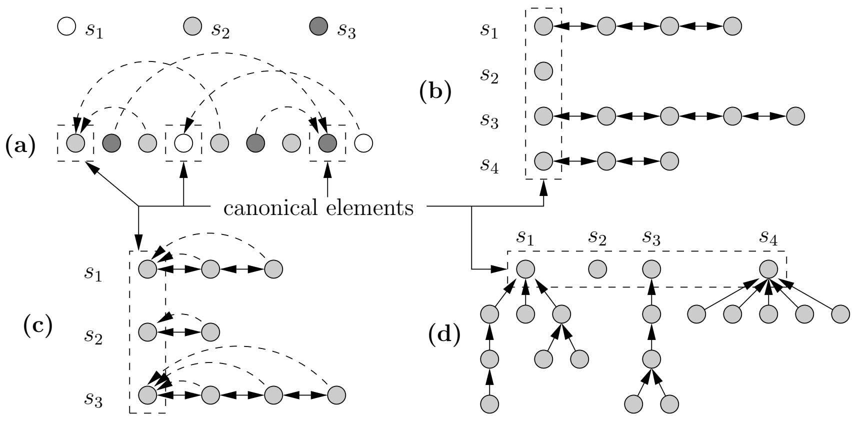

FIGURE 4.6: Disjoint set implementations: (a) a single large array or list of all elements, each pointing to the canonical element of its current set, (b) each set maintained as a doubly-linked list of elements, with the canonical element defined as the first element in each list, (c) each set maintained as a doubly-linked list of elements, with each element also pointing to the canonical element in its set, and (d) each set maintained as a tree, where the root is the canonical element.

element with a pointer to the canonical element in its set, as shown in Figure 4.6(a). This allows *make-set* and *find-set* to run in $O(1)$ time, but *union*(s_i, s_j) takes $\Theta(n)$ time since we need to scan all the elements, updating any pointer to the canonical element of s_i so that it now points instead to the canonical element of s_j (or vice versa). In Figure 4.6(b), we maintain the elements in each set in a doubly-linked list, with the canonical element of each set being its first element. It is easy to join two such lists in $O(1)$ time (assuming we also maintain a pointer to the end of each list), so *make-set* and *union* take $O(1)$ time. However, now *find-set* takes $\Theta(n)$ time in the worst case, since it involves scanning from an element backward to the canonical element at the beginning of its list.

Union by Rank. We benefit in terms of efficiency by combining the two ideas above, shown in Figure 4.6(c). Sets are stored as doubly-linked lists, and each element also maintains a pointer to the canonical element in its set, allowing *find-set* to run in $O(1)$ time. Computing the union of two sets requires $O(1)$ time to physically join their lists, plus the time to scan one set and update its canonical element pointers so they point to the canonical element in the other set. In order to re-route as few pointers as possible, we use a natural heuristic called *union by rank*, where we re-assign the pointers in the smaller set, since this obviously involves less work (this typically requires keeping track of the sizes of our sets[10], which we store as annotations attached to their canonical elements).

Union by rank causes at most $\log n$ pointer reassignments to any particular element during its lifetime in the structure, since any time an element's pointer changes,

[10]We can use the following clever trick to avoid maintaining augmented size information: during *union*(s_i, s_j), we scan s_i and s_j *simultaneously*, stopping when we reach the end of the smaller set. This takes time proportional to the size of the smaller set, which we will be spending anyways during subsequent pointer re-assignment.

the set containing that element must at least double in size (since its set is being merged with a larger set). More than $\log n$ such doublings are impossible, since this would result in a set larger than n elements. By charging an element up front for all its pointer re-assignments, we get amortized running time bounds of $O(\log n)$ for *make-set* and $O(1)$ for *union*.

4.6.2 Tree-Based Implementations

Another elegant and popular approach involves storing the elements in each set as a rooted tree, as shown in Figure 4.6(d), with the root being the canonical element. Every element maintains only a single pointer to its parent (no child pointers are needed), so we can easily take the union of two trees in $O(1)$ time by making the root of one point to the root of the other as its new parent. The only potentially slow operation is *find-set*(e), which requires scanning upward from e until we reach the root of its tree. In the degenerate case where e is at the bottom of a tree shaped like a long path, this could take $\Theta(n)$ time.

Union by Rank. We can improve the worst-case running time of *find-set* to $O(\log n)$ by keeping the height of each tree bounded at $O(\log n)$, using a variant of *union by rank*. We augment each root with a number we call its *rank*, initially zero for a tree consisting of a single element. When linking together two trees, we make the smaller-rank tree a child of the root of the larger-rank tree. We can link in either direction in the case of a tie, in which case we increment the rank of the root of the resulting tree; this is the only situation where a rank ever increases. Note that the rank of a tree corresponds exactly to its height, so all we need to do is show that ranks cannot exceed $\log n$. This is easily shown by arguing that the a tree of rank r must contain at least 2^r elements (so a tree of rank larger than $\log n$ cannot exist, since it would contain more than $2^{\log n} = n$ elements). Observe that the only way to create a tree of rank r is to link two trees of rank $r - 1$, which by induction on r both must contain at least 2^{r-1} elements, so the resulting tree must contain at least 2^r elements. After being created, a tree of rank r can only grow, if smaller-rank trees are linked into it; it can never shrink.

Path Compression. Our tree-based implementation also gains efficiency when we employ *path compression*. Observe that there is no advantage whatsoever for our trees to have large depth. In fact, the ideal tree shape is that of set s_4 in Figure 4.6(d), where every non-root element links directly to the root. Path compression brings a tree closer to this perfect shape: after executing *find-set*(e) and identifying the root of e's tree, we make a second pass upward from e, linking e and all its ancestors directly to the root. This does not change the asymptotic running time of *find-set*, and it makes future *find-set* operations potentially much faster.

Problem 77 (Potential Function Analysis of Path Compression). Please show that path compression (by itself, without union by rank) gives $O(\log n)$ amortized bounds for *find-set* and *union*. In your analysis, use the potential function $\phi = \sum_e \log s(e)$, where $s(e)$ denotes the number of elements in the subtree rooted at element e. As a hint: consider separately "big" versus "small" steps during the path compression process, where a big step from e up to its parent p occurs when $s(p) \geq 2s(e)$. Note that there are at most $\log n$ big steps during any call to *find-set*, so paying for these directly is fine; since there can be many more small steps, these will need to be paid using potential. [Solution]

Combining Union by Rank with Path Compression. Something remarkable happens when we use *both* union by rank[11] and path compression: the amortized running time of every disjoint set operation drops to only $O(\alpha(n))$, with $\alpha(n)$ being the extremely slowly-growing inverse Ackermann function we defined back in Section 2.1.3 (e.g., $\alpha(n) \leq 4$ if n is the number of atoms in the observable universe). In fact, using the two-term variant of the inverse Ackermann function (also defined in Section 2.1.3) we can say something slightly stronger — that any sequence of $m \geq n$ operations takes $O(m\alpha(m, n))$ time on a disjoint set structure with n elements. In particular, $O(\alpha(m, n)) = O(1)$ if m is ever-so-slightly larger than n in an asymptotic sense. For example, even if $m \geq n \log^{*\cdots*} n$ (with any constant number of stars), then $O(\alpha(m, n)) = O(1)$ and we get a linear performance guarantee of just $\Theta(m)$ for our m disjoint set operations. Surprisingly, despite the apparent mathematical complexity of the inverse Ackermann function, the amortized analysis of this structure is still relatively painless. [Details]

The original 1975 paper by Robert Tarjan describing this result (and also showing that the bounds above are tight) remarks that "This is probably the first and maybe the only existing example of a simple algorithm with a very complicated running time". Indeed, even today, decades later, it is hard to find any other algorithm with comparable simplicity whose running time has such a complex mathematical characterization. Any time you encounter an $O(\alpha(n))$ running time in the world of algorithms, there is a good chance it originates from this data structure.

4.7 Example: Persistent Data Structures

Most data structures are *ephemeral*, maintaining only their present state. In this section, we show an elegant method for automatically transforming certain types of ephemeral data structures into *persistent* data structures, which allow interaction with historical versions of the structure. We focus on making a structure *partially persistent*, supporting updates to its present state as well as read-only queries against any past version of the structure (*full persistence*, where we also allow modification to past versions, is usually somewhat more involved).

Among its applications, partial persistence can be useful for transforming solutions to "offline" problems, where we know queries up front, so they work for "online" problems, where queries arrive later. If we know queries up front, an algorithm can scan its input, pausing at appropriate locations to answer queries. However, with persistence, we don't need to know the queries in advance. We can go ahead and scan the input, and later when a query arrives, we can back up to the appropriate location in time to answer it. In Section **??**, we apply this method to several prominent geometric "sweep line" algorithms.

The technique we describe below applies to any pointer-based data structure (e.g.,

[11] Since the height of a tree can now shrink due to path compression, the rank of a root element no longer corresponds to the height of a tree. However, we still perform union by rank the same was as before: associate with every element a rank, and always link the smaller-rank root as a child of the larger-rank root when taking the union of two trees. In case of a tie, we can link in either direction, after which we increment the rank of the resulting root.

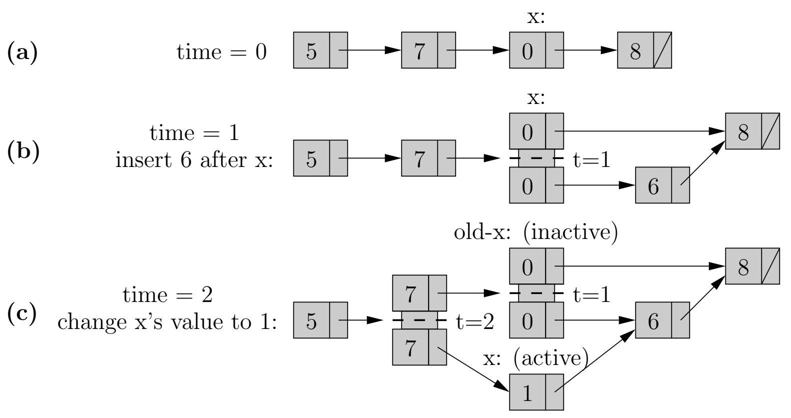

FIGURE 4.7: Making a simple linked list persistent. Each node has room to hold a single modification (that is, each node has a modification list of length $k+1=2$, since it has $k=1$ incoming pointers). In (b), we modify node x so that after time $t=1$ (indicated by the lower half of node x) it points to a newly created node with value 6, whereas before time $t=1$ (indicated by the upper half of node x) it points to its original successor with value 8. In (c), we again modify x by changing its value to 1. Since x's modification list is already full, we split x into two nodes, thereby causing a modification to x's predecessor.

a linked list or binary tree), where each node in the structure uses $O(1)$ memory and is pointed at by at most $k = O(1)$ other nodes[12]. All access to the structure starts at a designated "root" node, such as the first node in a linked list.

Suppose our original structure supports queries in $O(Q)$ time. We would like to add partial persistence without substantially slowing down update operations, while also keeping the query time close to $O(Q)$. To get started, here are several simple ways to achieve partial persistence that are not quite ideal in terms of efficiency:

- **Wholesale Copying.** Suppose after each update that we make a fresh copy of the *entire* structure, where the copies are organized by maintaining a time-sorted array of pointers to their roots. Although update time is absolutely terrible, query time is relatively fast at $O(Q + \log T)$, where T denotes the number of historical versions we are tracking (i.e., the number of updates), and the additive $O(\log T)$ term comes from binary searching the length-T array of root pointers to look up the root of the correct historical version.

- **Modification History in Nodes.** Let us store updates to nodes in the nodes themselves, so each node maintains its own private modification list in the form of a time-sorted array. Any time a node is updated, we add an entry to the end of its modification list, which stays sorted since modifications are conveniently made in increasing order of time. By storing these arrays using

[12]This requirement of constant indegree is satisfied by most pointer-based structures, such as linked lists ($k=1$), doubly-linked lists ($k=2$), and binary trees ($k=3$ if we maintain a pointer to each node from its left child, right child, and parent). A notable exception is the class of disjoint set data structures from the preceding section, since their canonical elements often have large numbers of incoming pointers.

the amortized doubling technique from Section 4.4, each node modification (which previously took $O(1)$ time) now requires only $O(1)$ additional amortized time, thereby only adding an "amortized" quantifier to our total update time. However, query time slows down a bit, to $O(Q \log T)$, since each node examination requires not $O(1)$ time as before, but $O(\log T)$ time in order to binary search for the appropriate historical record in its modification list.

By using aspects of both approaches, we can achieve a highly effective solution in which updates gain an "amortized" quantifier, and queries take $O(Q + \log T)$ time. To do this, we maintain a modification list at each node capable of holding at most $k+1$ entries, where $k = O(1)$ is the maximum number of incoming pointers to any node. If we make k or fewer modifications to some node i, we can therefore store the result within i itself, as shown in Figure 4.7(b). If we modify i again, then since i's modification list is full, we leave this modification list in place and split off a new copy of node i with only a single entry in its modification list. As shown in Figure 4.7(c), this effectively moves the active version of node i to a new location in memory, so we need to make updates in all the nodes pointing at i so they now (as of this point in time) point to the new location. If any of these nodes also have full modification lists, then we split them as well, potentially resulting in a sequence of cascading splits that could be quite extensive in the worst case. However, since these happen infrequently, we can show that they contribute only $O(1)$ additional amortized time to each original update operation. The only node we do not copy is the root node, whose modification list is allowed to grow without bound. The need to binary search this list at the beginning of each query is what contributes the additive $O(\log T)$ to our query time. Aside from this, queries run in the same time as in the original structure. Later in problem 106, we will see how to achieve this same result using randomization instead of amortization. [Full amortized analysis]

This is another good example of the "recursive slowdown" idea we mentioned when discussing functional queues in Section 4.5, since the short modification lists attached to our nodes serve to buffer updates and hence slow their propagation backward to other nodes by an appropriate factor. In fact, functional data structures have the advantage of being naturally persistent, since historical versions of the structure are never changed. All we need to do is maintain a list of historical roots to be able to access any past version of the structure.

It is usually ill-advised to apply persistence to an already "amortized" data structure. For example, right before a periodic housekeeping operation, queries in an amortized structure might run quite slowly. This is usually not a problem, since the structure will speed up after housekeeping. With persistence, however, we can continue to issue historical queries to the data structure as of this inefficient state. Please therefore exercise caution when mixing amortization and persistence. In addition, the persistence technique above only works on data structures for which query operations are read-only. A data structure that re-configures itself on queries (e.g., the splay tree in Chapter 6) cannot do so in the read-only context of a historical query.