COS 226 Precept 8 Class Notes Fall 2023 Revision 1

Robert E. Tarjan

September 20 2023

1 September 16: Persistence, Extendable Arrays, Amortization

1.1 Persistence

Informally speaking, a *data structure*, also called an *abstract data type*, is a collection of data items and relations among them, on which some specified set of operations can be performed. Such operations are of two types: *access* operations, which return certain information about the state of the data structure, and *update* operations, which change the state of the data structure, by adding or deleting items and/or by altering the relations among them. Typically the allowed operations are simple, but we want to perform many of them, one after another. We may even want to do several operations at the same time, a possibility we shall ignore, at least for the moment.

So far in lectures we have seen several examples of data structures. One is the *stack*, which has operations *push* (add a given new item to the top of the stack) and *pop*(remove and return the item on top of the stack). Both of these operations are updates, because they change the stack contents. We might want to support a third operation, *top*, which merely returns the top item on the stack. This is a query, not an update, since it does not change the stack contents.

In many uses of data structures, we only need access to the current version; when doing an update, we are free to alter or destroy the old version. We call a data structure that supports operations only on the current version *emphemeral*. In contrast, some situations require *persistent* data structures, which support accesses and possibly updates not only to the most recent version but to some or all previous versions. This idea raises the question of whether it is possible to make a given emphemeral data structure persistent, and how much this costs in extra time and storage space per operation.

There are many flavors of persistence, but we focus on two: *partial persistence*, in which queries to old versions are possible, but all updates are to the current version (that is, previous versions are frozen in time), and *full persistence*, in which updates as well as queries are possible on any version.

A persistent structure needs to support access to all its versions. A simple way to support such access is to use an extendible array *structure*, whose positions hold pointers (references) to successive versions. That is, *structure*(i) is a pointer to version i, the version resulting from the first i updates; *structure*(0) is a pointer to the initial version of the structure (which in many but not all situations will be the empty structure, indicated by the null pointer). We also need to maintain t, the total number of updates done so far. To do an update, we increment t, do the update, and store a pointer to the new version of the structure in *structure*(t). We use array resizing as needed to extend *structure* as t grows.

Remark 1.1. *Here and throughout these notes I use parentheses instead of brackets to denote array positions. An array is just a map from an interval of integers to some range, and there is no need to distinguish between functions and arrays, at least when doing high-level algorithm design.*

In precept I posed the problem of implementing a fully persistent stack on which each operation takes O(1) time worst-case. We concluded that the standard singly-lnked list implementation of a stack is naturally fully persistent, if we implement a pop merely by setting the top-of-stack pointer for the new stack to indicate the second item on the old stack, without changing the top-of stack pointer for the old stack. The overall structure becomes a forest of rooted trees, with $x.next$ being the parent of node x in the forest. (If some pop operation creates an empty stack and a later operation pushes an item on this empty stack, the result is a new one-node tree, which can grow by later pushes.) Observe that this representation of a fully persistent

stack allows any version to be iterable (Why?) Observe also that the array-based representation of a stack does not lend itself to making the stack fully persistent, at least not in any straightforward way. (Why?)

Problem for discussion in precept on September 22: Design a partially persistent queue that supports updates enqueue and dequeue; queries first, which returns the first item on the queue, and last, which returns the last item on the queue; and can be made iterable. Hint: This is easier than making a stack fully persistent. Both the list representation and the array representation of a queue can be made fully persistent in a straightforward way.

As Pedro discussed in precept on September 8, the rooted tree data structure for representing disjoint sets can be made fully persistent merely by adding time stamps to the links (or, better, adding a time stamp to each child indicating the sequence number of the unite that made it a child). To do a find in a given version, follow parent pointers until reaching a node that is a root in the given version, which is true if the node points to itself, or its time stamp is later than that of the version being accessed. This simple method works with any linking rule, but with path compression it fails. (Why?)

These are just some of the simplest examples of making ephemeral data structures persistent. Much more is known - if you are interested ping me. We may revisit the question of persistence when we study search trees.

1.2 Extendable Arrays

An *extendable array* A of size n is a map from the integers in the interval $[0, n-1]$ into some universe U. We call $i \in [0, n-1]$ an *array index* and $A(i)$ for any index i an *array entry*. Initially all array entries are null. An extendable array supports the operations *read*, *write*, *grow*, and *shrink*. A read operation is given an index i and returns the corresponding entry $A(i)$. A write operation is given an index i and a value $x \in U$ and sets the entry $A(i)$ equal to x. A grow operation increases n by 1 and initializes $A(n)$ to null. A shrink operation decreases n by 1. A shrink reduces the set of allowed indices, making the array entry indicated by the disallowed index (n after the shrink) unreachable.

Read is the only query operation on an extendable array, since write, grow, and shrink all change the array.

As we have seen in lecture, an extendable array is one way to implement a stack, but in addition to the standard stack operations it supports direct access to all the items on the stack.

Our goal is to implement an extendable array so that every read or write takes $O(1)$ time worst case, the total time for m grow and shrink operations is $O(m)$ ($O(1)$ per grow and shrink operation), and the total space used is $O(n+1)$ at all times. For now we shall assume that $n = 0$ initially; that is, the initial array has no indices and no entries.

Our solution is to represent an extendable array by a single fixed (non-extendable) array of N positions, with positions 0 through $n-1$ of this array corresponding to those of the extendable array. For this to work we need $n \leq N$. When the fixed array is full ($n = N$) and a grow operation occurs, we increase N, allocate a new fixed array of size N, copy the contents of the old fixed array into the new one, and replace the old fixed array by the new one.

For the implementation to be efficient, we cannot allocate a new array too often. When a grow operation on a full array occurs, we set the size of the new array to be cn, where c is some constant greater than 1. In lecture c was chosen to be 2, but we shall treat c as a parameter to be chosen later.

We also need to deal with the array becoming underfull. A succession of shrink steps can cause the extendable array to occupy a diminishing fraction of the fixed array, violating our requirement that the space used is always $O(n)$. We define the fixed array to be *underfull* when $n < fN$, where f is some constant such that $0 < f \leq< 1/c$. When a shrink step would make the fixed array underfull, we reduce N and proceed as in a grow step: We allocate a new fixed array of size N, copy the contents of the old fixed array into the new one, and replace the old fixed array by the new one.

When reducing N, we do not set N equal to n, since if the next operation is a grow step it would immediately trigger another array allocation. Instead we set N as in a grow step, equal to cn, so that the array density after immediately after an array allocation is same whether the allocation was caused by a grow or shrink step. Similarly, we do not allow $f = 1/c$, since if the next operation after an array allocation is a shrink step, it would immediately trigger another array allocation.

This method maintains the invariant that $fN \leq n \leq N$ except in the middle of a grow or shrink step, so it satisfies our space bound.

Exercise: Prove this. Note that the allowed space bound is $O(n+1)$, allowing some slack to deal with boundary cases.

Let us study the time efficiency of the method. We make the (conservative) assumption that allocating a fixed array of size N takes $\Theta(N)$ time. In our calculations we shall assume that the constant factor is 1; that is, allocating a fixed array takes 1 unit of time per position.

Suppose a new array is allocated. Let n take its value just after this allocation. If the next allocation is the result of a grow step, the time required for this allocation is the size of the new array, which is at most $c(cn+1)$. But there must have been at least $(c-1)n+1$ grow steps since the previous allocation. We charge the time for the new allocation to these grow steps, at most $c^2/(c-1) = O(1)$ to each (ignoring an extra c, which we can charge to the grow step causing the allocation.

Similarly, if the next allocation is the result of a shrink step, the time required for this allocation is at most fc^2n, but there have been at least $(1-fc)n$ shrink steps since the previous allocation. We charge the time for the next allocation to these shrink steps, at most $fc^2/(1-fc) = O(1)$ to each.

Summing over all the grow and shrink steps, we conclude that if there are m of these in total, they take $O(m)$ time: their total time is at most a constant factor times the total number of these steps, where the constant factor depends on the choice of c and f.

Exercise: Redo this analysis assuming that the cost of allocating an array is the number of values copied into it rather than its size. This number is $cn+1$ in the case of overflow and fcn in the case of underflow.

It is natural to ask whether we really need a constant factor extra space to make a fixed array extendable. That is, can we avoid the time/space tradeoff in the solution we have developed. I leave this as a problem for discussion in precept on September 22: Design an implementation of extendable arrays in which the space required for an extendable array of n positions is at most that of a single fixed array of n positions plus extra space that is sublinear in n at all times. The time to access an array position should be $O(1)$ worst case; the time for a total of m grow and shrink operations should be $O(m)$. If no such implementation is possible, prove it.

Hint: Consider representing the extendable array by a collection of non-extendable arrays, rather than by just one.

Another natural question is whether initializing a fixed array of size N needs to take $\Theta(N)$ time. In JAVA one is stuck with this bound, but not if one has more control over memory allocation. Specifically, can you design a way to allocate a fixed array and initialize all its entries to null in $O(1)$ time, while preserving an $O(1)$ time bound for reads and writes?

Hint: It is easy to designate a block of memory to hold the array, but this block may be full of unknown contents. We require a means to indicate which positions have actually been read and written: When an unwritten position is read, its value should be returned as null. You have all the tools you need to solve this problem.

1.3 Amortization

Our study of the running time of the extendable array implementation is an example of *amortized analysis*: We obtained a bound on the total time for a sequence of operations by showing that the slow operations are infrequent enough that the time they take is balanced by the time taken by fast but frequent operations. Such an analysis is appropriate when we have a sequence of operations of varying cost. We can get a bound on the total time of the sequence by estimating the worst-case time per operation and multiplying by the number of operations, but this may give a gross overestimate. Instead we find a way to make the fast operations pay for the slow ones, resulting in a much smaller overall estimate. We call this "amortization," adapting the common meaning of the word. "Amortize" comes from a Latin word meaning "to reduce to the point of death." To "amortize" a loan means to pay it off with regular small payments.

The *Algorithms* textbook devotes one page to this concept, but it is fundamental in the analysis of data structures and algorithms, so let us spend a bit more time on it. It is useful to have a more formal way to do such analyses. One such way is the *banker's view* of amortization. Imagine that our computer is coin-operated: One coin, which we shall call a *credit*, will pay for a constant amount of computer time (or space, or whatever resource is of interest). To perform a sequence of operations, we allocate a certain number

of credits to each one. To perform an operation, we start spending the credits allocated to it. It may happen that the operation finishes just as we spend the last credit. Better, we may end up not spending all the credits allocated to the operation, in which case we can save the unused credits and spend them on future operations. Worse, we may run out of credits before the operation finishes, in which case we need to spend saved credits, or else borrow from the bank. We can keep track of bank borrowings using debits: One debit is created for each borrowed credit; we can pay off one debit with one credit.

If we can complete the sequence of operations by spending only the credits allocated to the operations (paying off any debits incurred in the process), we call the number of credits assigned to each operation its *amortized cost*: The sum of the amortized costs of the operations is an upper bound on the actual total cost.

Of course there are many ways of allocating credits to operations. We seek a way that produces the smallest overall cost bound for all sequences of operations. Choosing the right allocation of credits, and keeping track of saved credits and accumulated debits (if any) is the heart of such an analysis.

Sometimes, as in the case of extendable arrays, the allocation is straightforward, almost mechanical: To each grow/shrink step we allocate two credits. We spend one immediately do do the step, whether or not it does an array allocation. We save the other. By the time an array allocation occurs, we have saved enough credits to pay for it.

In such an analysis it is often useful to associate the saved credits (or accumulated debits) with the state of the data structure (or algorithm). In the case of extendable arrays, there is one saved credit per grow/shrink step since the last array allocation.

An alternative view of amortization is the *physicist's view*. This view flips the banker's view on its head and views the state of the data structure as primary. The analogy here is to potential and kinetic energy: A big round stone that is stationary at the top of a hill has large potential energy; if we give it a nudge, it rolls down the hill, converting its potential energy into kinetic energy. Using this idea as a metaphor, we assign to each state of the data structure a real-valued . We define the *amortized cost* of an operation to be the sum of its actual cost and the net potential increase caused by the operation (the potential after the operation minus the potential before the operation. Summing over a sequence of operations, the total amortized cost equals the total actual cost plus the final potential minus the initial potential. If the initial potential is 0 (corresponding to an initially empty data structure) and the final potential is non-negative, then the sum of the amortized costs of the operations is an upper bound on the sum of the actual costs.

The potential in the physicist's view of amortization corresponds to the number of saved credits in the banker's view. A non-negative final potential in the physicist's view corresponds to having no unpaid debits in the banker's view. The physicist's view of amortization is a bit more abstract than the banker's view and a bit harder to work with, but they are entirely equivalent. We shall mostly use the banker's view in COS 226 and save the physicist's view for COS 423.

Exercise: Recast the analysis of extendable arrays in the physicist's view.