Lazy data structures

Kent Quanrud

March 11, 2021

1 As easy as 1, 2, 3

We have all coded the line

```
i++;
```

meaning, take an integer i, and increase it by 1. It's the simplest operation in the world, and understood to be very fast. Now consider the underlying bits as we repeatedly increment i, starting from 1.

`1, 10, 11, 100, 101...`

No big deal. But subtly, in the short list above, one of the increments takes a little bit longer than the others. To go from `11` (3) to `100` (4), we actually flipped *two bits*, rather than one. Of course two bits is no big deal. If we take the number $2^{32} - 1$, and add one;

`11111111111111111111111111111111, 10000000000000000000000000000000, ...`

we flip 32 bits![1] In general, in a k bit number, we may need to flip k bits. That's odd. The worst case performance of an increment is not uniform, and can be larger for large numbers. Yet we all have coded `i++` many times before and never really worried about this. Should we be worried?

When we count from 1 to n, how many total bits are flipped? Clearly the first (least significant) bit flips every time. The second bit flips every other time. The third bit flips every four times, and the fourth bit flips every 8 times. In general, the kth bit flips every 2^{k-1} times. So the number of bits flips is

$$\sum_{k=1}^{\infty} \left\lfloor \frac{2n}{2^k} \right\rfloor \leq n \sum_{k=1}^{\lceil \log n \rceil} \frac{1}{2^{k-1}} < 2n.$$

So *in the aggregate*, an increment acts like a constant time operation. Over n increments, we spend $O(n)$ time. *On average*, each increment takes $O(1)$ time. Unless we were timing each increment individually, we wouldn't notice the difference from a uniformly

[1] Of course, for modern computers, 32 bits is no big deal either. But that's besides the point.

constant running time. The technical term to express constant "average time" is to say that incrementing takes $O(1)$ **amortized time**.

In general, an operation takes T amortized time if any sequence of n invocations to the operation takes nT total time. We can easily generalize this to multiple operations. Suppose we have some data structure that has k different operations $\mathsf{op}_1, \ldots, \mathsf{op}_k$. Let $T_1, \ldots, T_k$ be k different values, one per operations. We say that each operation op_i takes T_i amortized time if, for the total running time of any sequence of operations $\mathsf{op}_{i_1}, \mathsf{op}_{i_2}, \ldots, \mathsf{op}_{i_n}$ is

$$O(t_{i_1} + t_{i_2} + \cdots + t_{i_n}).$$

2 Array Lists

We all know that an array is a sequence of entries $A[1..n]$ where each $A[i]$ can be accessed in constant time. The entries $A[1..n]$ are literally arranged in consecutive slots in memory. To retrieve $A[i]$, we really only need to know the location of $A[1]$, and then add $i - 1$ memory units[2] to compute the location of $A[i]$. We can then jump straight to the location of $A[i]$ in memory.

The downside to this convenience is that, after allocating $A[1..n]$, it is hard to adjust n. If we decide later to include an $(n + 1)$th element, there is no clear place to put it. One can use pointers and start making a more involved data structure, but then we are giving up the direct memory access that makes arrays so appealing. Alternatively, we could make a new array $B[1..n + 1]$ of size $n + 1$, copy over the n elements from A, and store the $(n + 1)$th item in the $(n + 1)$th slot. But of course this takes $O(n)$ time to do the copying. What happens on the $(n + 2)$ item? Do we copy everything over again?

A slight modification of this last strategy is assumed by the ubiquitous `ArrayList` data structure in `Java` and by lists in `Python`. When we want to add elements beyond the capacity of $A[1..n]$, we allocate an array $B[1..2n]$ of *twice* the capacity, and copy the n elements of A over to the first n slots of B. It seems like common sense to overallocate generously so that we don't have to copy as often. On the other hand, we certainly don't have $O(1)$ update time. To justify this approach, we turn to amortized analysis.

Theorem 1. *The array list supports constant time access to any element in the array, and takes $O(1)$ amortized time per insertion.*

Proof. Over k insertions, the total time spent by the data structure is equal to $O(k)$ plus the total time doubling and rebuilding the array. Thus the total time is

$$\begin{aligned} &O(k) + \sum_{i=1}^{\lceil \log k \rceil} \left(\text{time to double in size from } 2^i \text{ to } 2^{i+1} \right) \\ &= O(k) + \sum_{i=1}^{\lceil \log k \rceil} O\left(2^i\right) = O(k). \end{aligned}$$

So each insertion takes $O(1)$ amortized time. ∎

[2]For lack of a better word.

2.1 The potential method

In amortized analysis, we only really care about the total running time over a sequence of operations, and anyway to bound it is fine. For binary counters and array lists we were able to directly bound the sum. But other times the analysis gets trickier and there are some useful techniques that can help. One standard way that we will focus on is called the **potential method**. The idea is to invent a potential function, Φ, that is a function of the state of our data. The potential goes up and down and we pay for increase in the amortized running time. This means that sometimes we also benefit from a decrease, and can potentially offset large operations. Let us first illustrate the technique on the amortized array lists above, and then explain the general method after.

Proof by potential method. We define a potential

$$\Phi = \big(\#\text{ elements in the array that have never been copied}\big).$$

If a call to `append(`x`)` that does not invoke reallocation takes constant time and increases the potential by 1. Thus we claim it takes

$$O(1 + \Delta\Phi) = O(1) \text{ amortized time.}$$

Now consider a call to `append(`x`)` that does reallocate, say, n items. Then the reallocation takes $O(n)$ time but the potential also decreases by n. We claim that the operation takes

$$O(1) + O(n + \Delta\Phi) = O(1) + O(n - n) = O(1)$$

amortized time. Thus in either case, we claim that `append` takes $O(1)$ amortized time.

Now suppose we have k total insertions. For $i = 0, \ldots, k$, let Φ_i be the value of Φ after k insertions. Let t_i be the (real) time spent on the ith insertion, and let $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$ be the change in potential on the ith iteration. We have

$$\sum_{i=1}^{k}(t_i + \Delta\Phi_i) \overset{\text{(a)}}{=} \Phi_0 + \sum_{i=1}^{k}(t_i + \Delta\Phi_i) = (\text{total time}) + \Phi_k \overset{\text{(b)}}{\geq} (\text{total time}).$$

Here (a) is because $\Phi_0 = 0$. (b) is because Φ is always nonnegative. Above we argued that $t_i + \Phi_i \leq O(1)$ for each i. Thus, after k operations,

$$(\text{total time}) \leq \sum_{i=1}^{k}(t_i + \Delta\Phi_i) \leq O(k),$$

as desired. ■

Let us now explain the potential method more generally. Suppose we make k operations, where the ith operation takes t_i (real) time and increases the potential by $\Delta\Phi_i$. We have

$$\Phi_0 + \sum_{i=1}^{k}(t_i + \Delta\Phi_i) = (\text{total time}) + \left(\Phi_0 + \sum_{i=1}^{k}\Delta\Phi_i\right) = (\text{total time}) + \Phi_k.$$

Suppose we can show an upper bound α on $t_i + \Delta\Phi_i$ for all i. Then we would have

$$\text{(total time)} = \sum_{i=1}^{k} (t_i + \Delta\Phi_i) + \Phi_0 - \Phi_k \leq k\alpha + \Phi_0.$$

(Here we assume that Φ is nonnegative, without loss of generality.) Thus we can say that, after an initial cost of Φ_0 amortized time, each operation takes a amortized time. In the array list above, we have $\Phi_0 = 0$ and $\alpha = O(1)$, giving us constant amortized time (with no initial amortized cost).

3 Lazy search trees

The reader is also likely familiar with binary search trees. The idea is that we want to maintain a collection of ordered keys under insertions and deletions. Other auxiliary operations of interest include (a) retrieving the first key that comes before or after a query, (b) aggregate (e.g., sum) the values associated with all keys k in the range $a < k < b$, for given points a and b, and (c) list all the keys k in the range $a \leq k \leq b$, in order. We will assume the keys are distinct for simplicity though it is easy to adjust the ideas to accommodate duplicate keys.

Binary search trees arrange the data in a rooted tree. Each node contains one key and each key belongs to one node. For each subtree we maintain the following simple rule. Let k be the key at the root of the tree. Any key in the subtree that is $< k$ is placed in the left subtree. Any key in the subtree that is $> k$ is placed in the right subtree. This rule makes it easy to navigate the tree. Suppose we are looking for a particular key k. If the root contains k, then we are done. Otherwise we compare k to the key at the root and recurse on the left or right subtree appropriately.

It is easy to insert keys while maintaining the invariant. We search for the key would be, if it were in the tree. At the end we will find that the recursion takes us to an empty subtree that signals the key is not there. We place the key there as a leaf.

For both search and insertion, the worst case time to traverse a tree is entirely proportional to the height of the tree. Ideally, a tree would be *balanced*, where for each node, each subtree contains at most half of all the descendants. Given any static set of keys, it is easy to build a perfectly balanced binary search tree with height $O(\log n)$. But when the set of keys are changing, it is easy to insert keys in an order that leads to a completely unbalanced tree. Thus considerable attention is paid to maintaining a binary tree. Many ingenious schemes have been invented to achieve this. Red-black trees are one such data structure included in many data structure classes. Splay trees are another ingenious data structure that we will study later. Instead we will pursue a very simple and lazy approach that heavily leverages amortized analysis.

We start from the standard (unbalanced) binary search tree. For each node, we also keep track of the number of nodes in the subtree rooted at that node. In particular, for each node, we are always aware of the difference in size of the two subtrees. Whenever we notice that, for some node n, one tree is more unbalanced than the other, we rebuild the

```
insert(node v, key k)
// insert k as usual
1. if v.key < k
   A. recurse on the left subtree
2. else recurse on the right subtree
// rebalance if needed
3. if one subtree of v has at least twice as many nodes as the other
   A. rebuild the entire subtree rooted at v
```

Lemma 3.1. *The tree always has height $O(\log n)$*

Proof. At any point in time, the number of nodes in one subtree is at most $2/3$ the number of nodes in the parent subtree. Thus there are at most $\log_{3/2}(n) = O(\log n)$ levels in the tree. ■

Theorem 2. *With lazy rebalancing, one can maintain a binary search tree of height $O(\log n)$ in $O(\log n)$ amortized time per insertion.*

Proof. For each node v in the tree, we define a potential Φ_v to be the absolute difference in size between its two subtrees. We define a global potential Φ to be the sum of the individual potentials,

$$\Phi = \sum_v \Phi_v.$$

Initially, $\Phi = 0$.

Claim 1. The part of insertion that comes before rebuilding $O(\log n)$ amortized time.

Because the tree has height $O(\log n)$, it takes $O(\log n)$ time to find the right leaf to place the key. The insertion increases the potential by at most 1 for each of the $O(\log n)$ nodes on the root to leaf paths.

Claim 2. Rebuilding at an imbalanced node v takes $O(1)$ amortized time.

Suppose v had k nodes. Then at v potential must have been at least $k/3$. Rebalancing takes $O(k)$ time and decreases the potential at v to 0, so the decrease in potential offsets the time spent to do the rebuilding.

Summing the two claims gives $O(\log n)$ amortized time overall. ■

3.1 Lazy deletions

Suppose now we wanted to support deletions. Here it is not entirely obvious what to do in a regular binary search tree. If we simply remove a node from the tree, unless it is a leaf, it creates a home, and we have to scramble to fill it with some other node. Here we will take a lazier alternative. We simply mark the node as deleted, and adjust the search routine to bypass keys that are marked for deletion.

`lazy-delete(`v`, `k`)`

1. If v's key equals k

   A. mark v for deletion

2. Else recurse appropriately on the left or right subtree of v

3. If v is the root and at least half the nodes in the entire tree are marked for deletion.

   A. Rebuild the entire tree.

Lemma 3.2. *At any point in time, less than half the nodes in the tree are marked for deletion.*

Lemma 3.3. *At any point in time, the tree has size $O(\log n)$, where n refers to the number of keys that have not been deleted.*

Proof. By the same argument as above, the tree has height logarithmic in the total number of nodes (marked or unmarked). But the total number of nodes is within a constant factor of the total number of unmarked vertices. ■

Theorem 3. *Lazy rebalancing and lazy deletions maintains a binary search tree with height $O(\log n)$ in $O(\log n)$ amortized time per insertion or deletion.*

Proof. In addition to the potential functions $\{\Phi_v : v \in V\}$ and $\Phi = \sum_v \Phi_v$, we introduce another potential function Φ that counts the total number of vertices marked for deletion. Each insertion takes $O(\log n)$ amortized time by the same argument as before. Each deletion first takes $O(\log n)$ time to find the key to mark for deletion (because the height is $O(\log n)$), and then increases Φ by 1. When we rebuild on account of deletions, Φ is at least $m/2$, where m is the total number of nodes in the tree. The rebuild takes $O(m)$ time and decreases Φ to 0. Thus the decrease in potential pays for the rebuild. ■

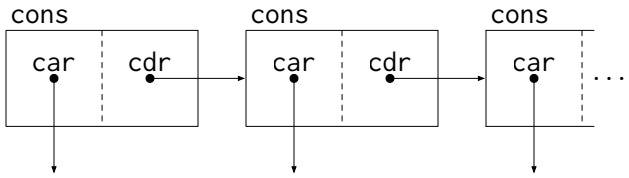
Remark 3.1. To support other tree operations, it is sometimes preferable to ensure that in every subtree, at most half the nodes are marked for deletion. To this end, we can keep a count of the number of nodes marked for deletion for every subtree, rebuilding whenever it represents half the subtree. This will still have $O(\log n)$ amortized time.

4 Immutable data structures

In purely functional programming languages such as Haskell[3], all data is required to be *immutable*. Immutability is a fancy name for one simple rule: once something is written down, it can never be changed. This might seem restrictive, but assuming immutability can greatly simplify the reasoning in your programs. If you hand off the data to some function, you are promised that the function won't mess with your data. This is especially helpful in concurrent programming, when it is otherwise hard to keep track of all the changes made by other threads. Immutability also allows the compiler to be more aggressive when producing optimized lower level code.

[3] `http://learnyouahaskell.com/chapters` - have fun!

A (singly) linked list is inherently immutable. A linked list consists of `nodes`, each of which contain two pointers. One pointer is to the next node in the list, or `null` if it is the end of a list. The other pointer is to some piece of data that we would want to associate with the node. In `lisp`, a node is called a `cons` cell, the pointer to the data is called the `car`, and the pointer to the next `cons` is called the `cdr`.



To insert (`push` / `cons`) an element x to the beginning of a linked list, we create a new node that points to x and to the first node in the list. Henceforth the new node is treated at the beginning of the list. Observe that no changes were made in the existing list. In particular, any other pointers to the top of the list will not notice any changes.

Likewise we can immutably remove (`pop`) elements from the beginning of a linked list. Given a pointer to the beginning of a list, we just follow the pointer to the second element in the list, and use that as the top of the pointer to the list. Note that we didn't make any change to the data structure from an external point of view. We simply updated our own reference to the data structure so that the data structure is changed only from our own perspective.

Immutability means that when we want to change a small part of an object, we instead have to make an entire copy of that object that incorporates the small small change. This might seem wasteful. But often it is not so burdensome in many object-oriented systems that then do be composed of individually small objects with pointers to many others. Suppose we want to update an object that has some links to other objects. When we make a new version of that copy, we don't have to follow the links and recursively copy those objects as well. We simply copy over the pointers to the objects in the new version.

As a concrete example, consider the lazy search trees in Section **??**. They can easily be made mutable. The primary difference is that, when we insert a key at the bottom, we have to rebuild the nodes along the path from the root to the leaf. Note that we don't have to rebuild the subtrees hanging off the paths along the way. (Unless the algorithm calls for us to rebuild the entire tree, which we account for separately.) Fortunately for us, the lazy search trees ensure that the height of the tree is at most $O(\log n)$. So we only create $O(\log n)$ nodes. Each node is constant size, so this all takes $O(\log n)$ time.

A free benefit of immutable data structures is that you automatically produce snapshots of all previous states of the data structure. That is, we get an "undo" operation for free - simply replace the current pointer to the data structure with the previous pointer to the previous version of the data structure.

4.1 Stacks and Queues

Recall that a **stack** is a data structure that allows us to insert and delete items strictly in **first-in-last-out** order. Namely, there are two operations, `push` and `pop`. `push` inserts an object. `pop` removes the most recently `push`'d object that had not yet been popped. Imagine

a stack of plates. The linked list discussed above gives an immutable implementation of a stack. What about a **queue**? A **queue** inserts and removes objects in **first-in-first-out** order. A singly linked-list will not suffice because we are inserting and deleting from opposite ends of a list. In fact it seems difficult to achieve constant (worst-case) update time. But we will relax our requirements and allow for constant *amortized* update time. We challenge the reader to try to figure how to implement an immutable queue with $O(1)$ update time before proceeding below the fold.

It turns out that there is a clever way to achieve this, using only stacks (which are already immutable). Here are the ingredients.

1. Maintain two (immutable) stacks, called the insert and output stacks.
2. Insert into the input stack.
3. Remove from the output stack.
4. If the output stack is empty, then replace the output stack with the input stack reversed, and make the input stack empty.

Theorem 4. *The two-stack queue implements insertion and deletion in $O(1)$ amortized time.*

Proof. We maintain a potential function Φ equal to the number of elements in the input stack. Each insertion takes $O(1)$ (real) time and then increases the potential by 1, so it takes $O(1)$ amortized time. Each deletion, excluding the part where we rebuild the output stack, takes $O(1)$ real time. Rebuilding the list takes $O(\Phi)$ time, where Φ is the number of elements in the input stack. Meanwhile the potential Φ decreases to 0. ■

If we allow for amortized running time, then many data structures can be made immutable while retaining the same asymptotic performance. We refer the reader to [2] for many more examples.

References

[1] *Functional Pearls*. Wiki article on `haskell.org`. URL: `https://wiki.haskell.org/Research_papers/Functional_pearls`.

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.