A BRIEF(ISH) RECAP OF AUGTABLES

by anna •

let's take a step back and revisit a simpler data structure... tables!

so, what are tables? tables are (unordered) mappings of keys to values

for example, we can have a table that maps the letters in "table" to their rank in the alphabet:

key	t	a	b	l	e
value	20	1	2	12	5

(if it helps, you can think of these like python dicts!)

we denote tables by writing out their key-value pairs as $\{ \text{key} \mapsto \text{value} \}$, so the above example would be written $\{ t \mapsto 20, a \mapsto 1, b \mapsto 2, l \mapsto 12, e \mapsto 5 \}$

(we'll assume that the keys in a table are unique, i.e. no duplicates, unless stated otherwise, but we make no assumptions about the values)

since we make no assumptions about the keys, we can only compare them for equality. this means that we have a fairly limited set of things we can do with them:

- we can create them (empty, singleton, tabulate)
- we can lookup and modify key/value associations (find, insert, delete)
- we can combine two tables (intersection, union, difference)

note that all the operations will be fairly expensive since we know nothing about the ordering of the keys so we need to look at all the keys to find the one we want

having to look at all the keys kinda sucks :(can we add constraints to make our lives easier? yes :)

this strengthened table is called an ordered table. ordered tables are tables whose keys have a total ordering. the example table above would be an ordered table since letters inherently have a total ordering. however, a table that maps fruits to their color $\{ 🍓 \mapsto 🔴, 🍊 \mapsto 🟠, 🍋 \mapsto 🟡, 🍏 \mapsto 🟢, 🍇 \mapsto 🟣 \}$ would not inherently be an ordered table. (we can make it an ordered table if we ordered the fruits, for example 🍓 < 🍊 < 🍋 < 🍏 < 🍇) (if you're reading this, what would your ordering of the fruits be?)

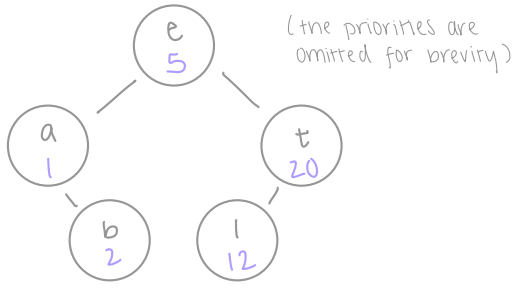
with an ordered table, we can do all the operations of a table, but quicker, and more since our keys are ordered:

- any operation the requires finding a key in the table can be done faster since we can compare/order keys, thus we know "approximately" where it would be in the table
- we can do additional operations on the table based on the rank of the keys (first, last, previous, next, rank, select)
- we can do additional operations based on the keys' relative ranking to each other (split, getRange, splitRank)

let's take a small detour and consider how ordered tables are implemented. (keep in mind that the way tables are implemented only impacts the work/span bounds of the functions, not the result of calling the functions.) in 210, we implement ordered tables using treaps. We can do this since the keys are ordered and thus the BST/order invariant can be enforced, and we can assign random priorities to the keys to enforce the heap invariant (though you'll never need to consider the priorities when working with tables since they're not relevant to tables).

since tables are implemented using treaps, oftentimes we'll draw out a tree representation of the tree we're discussing. so, for example, our example from before could be drawn like so:

key	t	a	b	l	e
value	20	1	2	12	5



(the priorities are omitted for brevity)

okay, so I know things about the keys. what if I want to know things about my values? we can augment our table with our ideal function to make it extra stronk. augmented tables are ordered tables that also store the result of applying a function to all the values in it (essentially the result of calling reduce f on it).

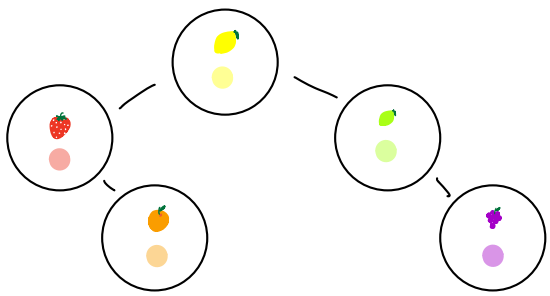
since we didn't add any additional constraints to the keys, augtables have the same key-based functions as ordered tables.

then, why are augtables cool? (why do we care about them?) augtables are cool because they compute the result of applying a function over all values in the table as the table is constructed / modified, which we can then retrieve later by calling reduceVal on the table. this might seem inconsequential since we can also just call reduce on the table to get that value, but the cost of doing so could be very different, which we'll explore in a bit.

for example, consider the fruit table from before: (assume we've ordered these fruits) {🍓 ↦ 🔴, 🍊 ↦ 🟠, 🍋 ↦ 🟡, 🍏 ↦ 🟢, 🍇 ↦ 🟣}. if we wanted to know what the colors of the fruits mixed together would look like, we can augment the table with a color mixing function. then, when we call reduceVal on it, it'll give the result of 🔴 + 🟠 + 🟡 + 🟢 + 🟣 = 🟤.

that's cool, but i'm not sure how it works.

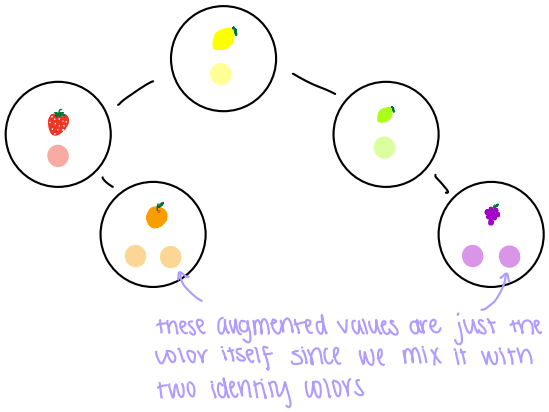
let's consider a treap representation of our table:



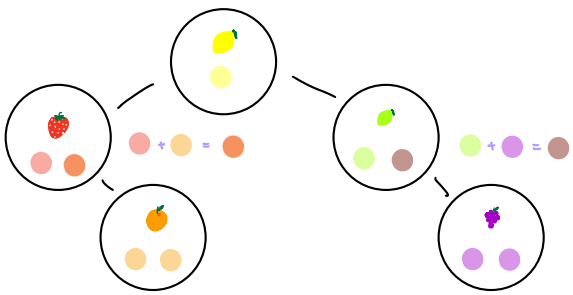
we can then augment our table by applying the function starting at the bottom of the treap (essentially, we're computing the value after we finish creating the children of each node).

we first compute the value of the nodes with empty leaves as children:

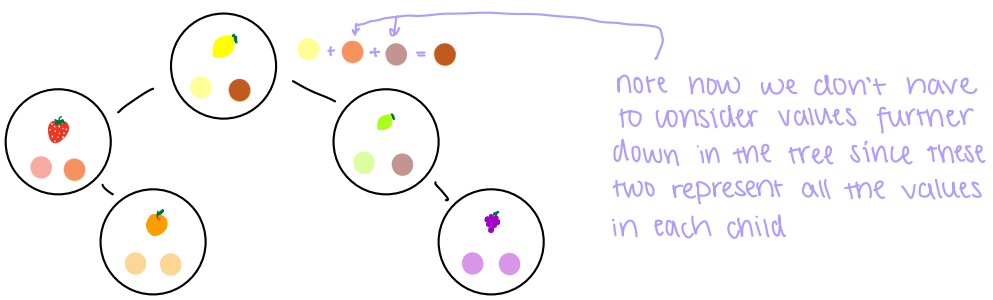
(we'll assume we have some identity color for color mixing)



then we work our way back up the tree and mix their parents:



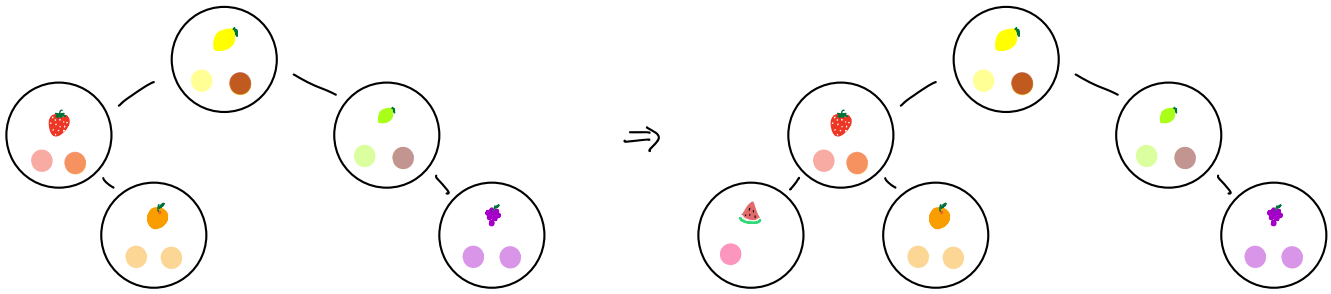
we continue to the next level up...



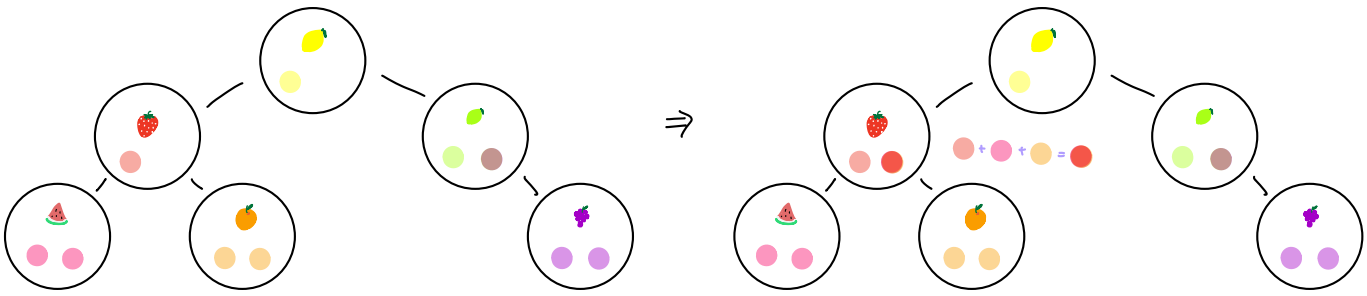
we then have computed the reduceVal for the entire table and have our augmented table.

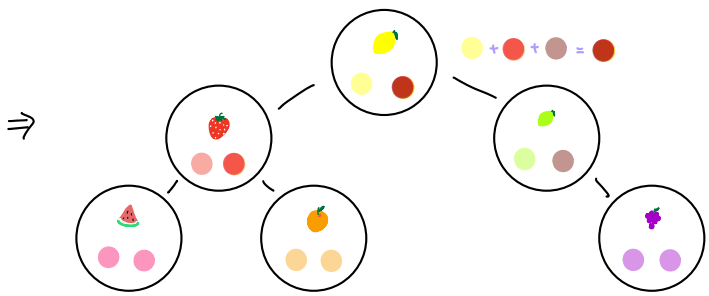
* since the augmenting function works like reduce, it has the same requirements:
  - ✓ it must be associative : $f(x, f(y,z)) = f(f(x,y),z)$
  - ✓ it must have an identity : $f(x, I) = x = f(I, x)$

insert / delete / update work similarly: consider inserting 🍉 ↦ 🍎 into our table, where 🍉 < 🍓



now, we need to update the reduceVal of the table:





notice that we only need to update the reduceVal of the ancestors of our updated node. then, the cost of updating tree and getting its new reduceVal is simply the cost of updating the tree. this is where augtables are useful - it is cheaper to update the reduceVal as the table is updated than it is to update the table and then call reduce on it.

to compute the cost of this insert, we can use the fact that treaps are roughly balanced with high probability to get the recurrence

↓ we only need to update one of the two children

$$W(n) = W(\tfrac{n}{2}) + W_f$$

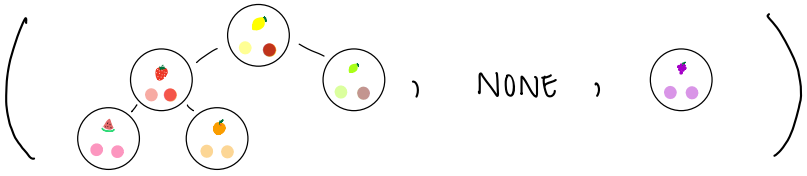
so, if we can mix colors in $O(1)$, then our insert is $W(n) = W(\frac{n}{2}) + O(1) = O(\lg n)$

if we need $O(\lg n)$ to mix colors, our insert is $W(n) = W(\frac{n}{2}) + O(\lg n) = O(\lg^2 n)$

split is a really helpful function to use in algorithm design :) especially when you want to consider subsets of your data. split returns a tuple that contains a table with all the key/value mappings whose keys are strictly less than the key you're splitting at, the value associated with the key you're splitting at if there is one, and a table with all the key/value mappings whose keys are strictly greater than the key you're splitting at.

if we split our updated fruit table at 🍊, we'll get

if we split our fruit table at 🫐, where 🍏 < 🫐 < 🍇, we'll get

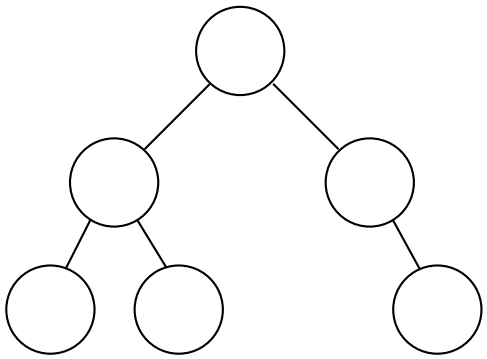


the following pages are an actually brief summary of BSTs / treaps / tables from 522

BSTs

hopefully you know what a binary tree is :)

however, if you need a refresher, here is what a binary tree looks like:



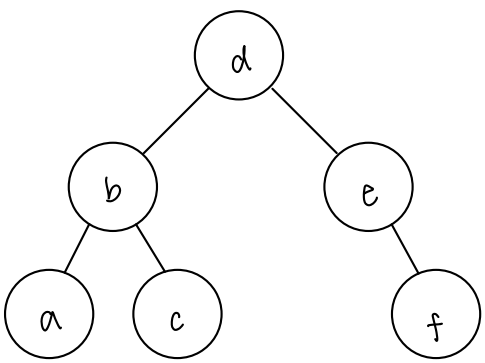
* each node has ≤ 2 children

a binary search tree (BST) is a binary tree with an order invariant:

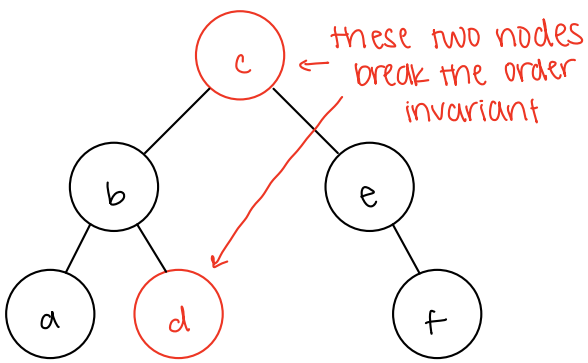
for each node 

, all keys in the left subtree L are less than K and all keys in the right subtree R are greater than K

here is an example of a valid BST:



however this is not a valid BST:



this order invariant allows us to search for keys in the tree quickly (i.e. in O(height of the tree))

* if the tree is balanced and its keys are unique, then we can search/update in O(lg n)

BSTs are also really nice since we can do stuff to them in parallel (we can work on the left and right subtrees in parallel)

TREAPS

when talking about O(lg n) search/update for BSTs, one big assumption that we make is that the tree is balanced, but we don't have this guarantee with arbitrary BSTs

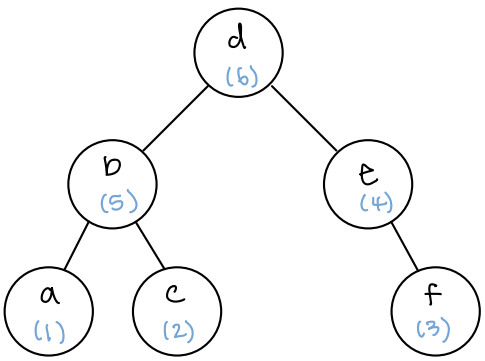
* treaps allow us to make this assumption :) (they are O(lg n) height whp!)

treaps are BSTs with a heap invariant:

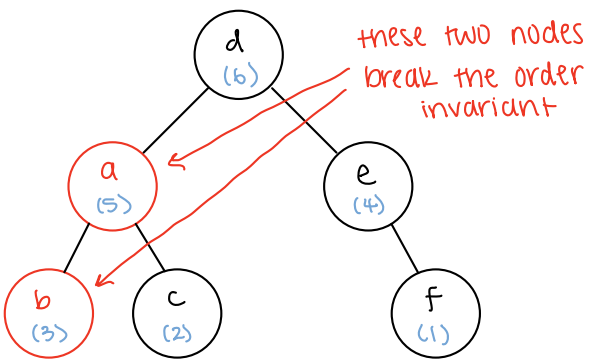
for each node with priority p, p is greater than the priorities of all the nodes in its left and right subtrees

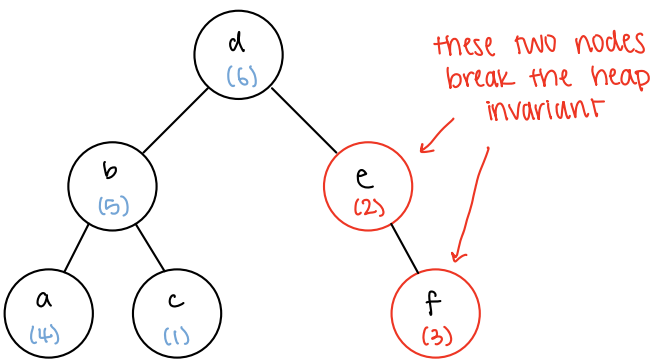
* each node has a key and a priority (these are not necesarily the same!)

here is an example of a valid treap:



but these are invalid treaps:





an interesting consequence of these two invariants is that, if the priorities are unique, then there is a unique tree structure for the treap

(you can prove this using induction on the size of the tree)

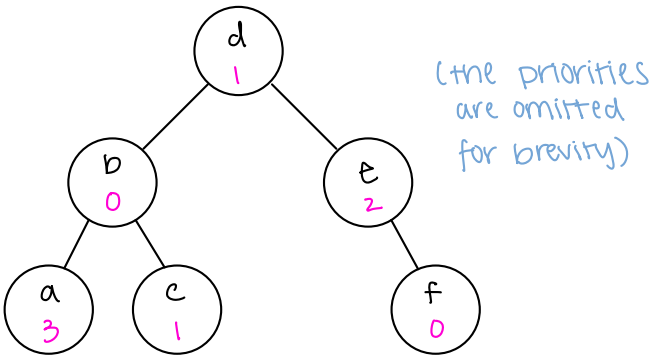
* since priorities are assumed to be uniformly random, a lot of treap analysis is probability based!

SETs / TABLES / AUGTABLES

treaps have some great properties, but what can we use them for? implementing tables!

abstractly, tables are a mapping of keys to values (we assume that the keys are unique, but we don't make any assumptions about the values)

for example, if we wanted a table of how many 210 TAs start with certain letters, we could represent it with a treap:



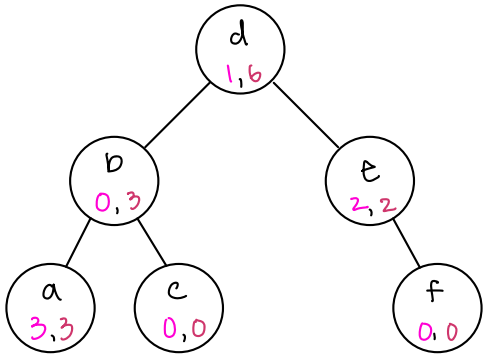
in general, you can think about/use tables as simply a mapping of keys → values! the specifics of how they're implemented only affects cost bounds

tables are a great tool when we want to "quickly" lookup or modify keys and/or their values

* for example, sets are implemented as unit tables since we only care about fast lookup of keys

sometimes we also want to do computations over all the values in a table, which is where augmented tables become very useful

for example, if we wanted to know how many 210 TAs there are, we can augment our table with addition:



NOTE: the augmenting function must be associative and have an identity

here are a few important functions in the augtable interface:

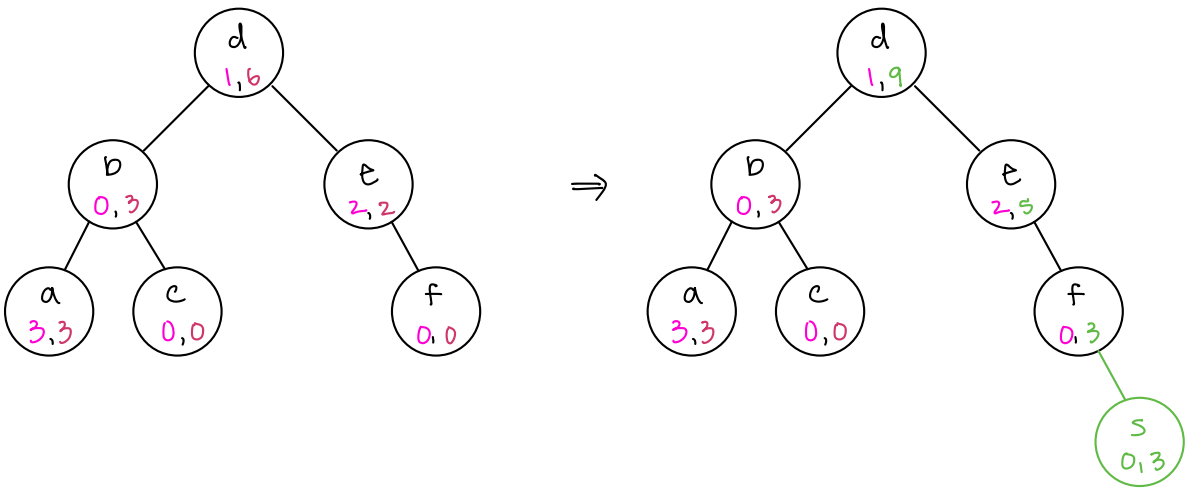
✓ reduceVal O(1) w/s

- instead of computing the result of reducing over the values in the table each time we need it, we simply store it and update it whenever the values in the table change
- reduceVal simply returns this stored value

✓ insert wst depends on the w/s of the augmenting function

- it inserts the new node into the table
- in a treap implementation, it then recomputes the reduceVal of all of its ancestors

ex: inserting $s \rightarrow 3$ into the 210 TA table



* the augmenting function must be O(1) for insert to be O(lg n)

✓ split wst depends on the w/s of the augmenting function

- split is really helpful for algorithm design, especially when you want the result of doing computation on a subset of your data

ex: if we want to know how many TAs have names that start with letters a-d, we can split the table at e and take the reduceVal of the left tree