We have been discussing the following type of recursive structure: each node tracks $B$ copies of some lower level structure, and aggregates its information from them.

Today we discuss how time can factor into this. The simplest form is persistence: anytime we access a data structure, we can make copies of pointers to create a ‘new’ copy of the node. This means we can jump to the data structure at any particular point of time.

The natural extension of this is optimize over time. Consider supporting:

1. Range increment (by possibly negative number).
2. Range historical max (max that some number in a range has ever been)

We start with a two level structure with $B = n^{1/2}$. Here note that if we bulk update a range, the historical min is the min in the range (before we last updated somewhere in there), plus the min prefix sum of the increments to the range. More formally, we use the ability to track the best prefix min of a sequence of numbers, even as that sequence of numbers is given online.

Now consider a 3 level of this. Note that if we have two separate histories over a node (the $B$ and $B^2$ sized parent blocks), it’s difficult to ‘merge’ the information together. Instead, we need to use the fact that updates to the $B$ sized parent take place BEFORE all the updates to the $B^2$ sized ancestor. Specifically, whenever we go from the $B^2$ sized node $p_2$ to the $B$ sized node $p_1$, we do:

\[ p1.minincrement \leftarrow \min \{ p1.minincrement, p1.sum + p2.minincrement \} \quad (1) \]
\[ p1.sum \leftarrow p1.sum + p2.sum \quad (2) \]

(and similarly to the other child of $p_2$), after which we CLEAR all the value from $p_2$.

This ensures the following type of property:

**Claim 1.1.** In any root to leaf path, the time interval in which the bulk stored updates happen do not intersect.

Next, I want to introduce a ‘SET$_{\geq}$/SET$_{\leq}$’ operations. That is, for everything in $A[i \ldots r]$, I set $A[i]$ to be at least/at most $x$, which formally is

\[ A[i] \leftarrow \max \{ A[i], x \} , \]
for $\text{SET}_\geq$, and for $\text{SET}_\leq$:

$$A[i] \leftarrow \min \{A[i], x\}.$$ 

Consider the set at least operation first. Note that this value does propagate downward nicely by itself: if a parent node is set to at least $x_1$, and this node is set to at least $x_2$, then everything below should be set to at least $\max\{x_1, x_2\}$.

The issue comes at combining with range increment: note that

$$\text{SET}_\geq (3, 5) + 3 \leq \text{SET}_\geq (3 + 3, 5),$$ 

so we cannot reorder these operations arbitrarily, or in other words, turn a long update sequence into a short one.

Instead, we need to handle the ‘set at least’ operations aggressively, and make progress by noting that they **decrease the number of distinct elements**.

That is, on each node, we track:

1. value of the minimum,
2. how many times the minimum appears,
3. value of the second minimum.

If the set max value is more than minimum, less than second minimum, we just increase the minimum. Otherwise, we recursively get rid of the minimums, and find a new second minimum. These recursions are affordable because range increments can only add to the number of distinct elements in partially touched regions, which there are only $O(\log n)$.

This stuff is most epic in combination with range historical min/max operations. That is, not can I detect the min/max of a range, I can also detect what’s the min/max it has ever been. This may seem of dubious value, but in PS12 you’ll see it comes in reasonably naturally with range distinct element queries.