DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

The recursive structure we discussed so far have the structure of each node tracking $B$ copies of some lower level structure, and interactions only happen between consecutive levels.

Last time, the monotonicity in $x$ condition led to the notion of ‘kinetic tournament trees’, where each comparison tracks how much increments need to happen to it until one of the decisions flip, and these flips are counted recursively. Note that this leads to recursive updates: when we update a node in bulk by increasing it by $\delta$, we need to check if $\delta$ exceeds the switching threshold of either of its children, and recursively apply to any point where the $\delta$ exceeds the threshold.

Today we discuss this notion of more complex recursion in more details. We first start with a version that has immediate applications.

Lemma 1.1. We can solve a sequence of $n$ operations of:

1. range add,
2. range sum/min/max,

in $O(n \log n)$ time.

A variant of this structure, that maintains min, is useful for computing the minimum cut that intersects two edges of some tree. It shows up because:

1. 3 vertex connectivity and planarity testing is closely related,
2. global min cut, with some reduction, becomes finding an edge that intersects up to two edges of a tree.

As a warmup, consider finding the minimum cut that intersects one edge of some tree. Think about what happens to the set of edges that cross some cut, as we move the cut around the tree. Say the edge we cut is from some $u$ to its parent $p(u)$, then the weight of the cut is the total weight of the subtree. There are two ways to deal with this:

1. Either maintain a (mergeable) priority queue of all the edges in each subtree, sorted by the depth of their ‘highest point’ (lowest common ancestor).
2. Or do a heavy-light decomposition to partition the trees into paths. Then move up each path, maintaining one priority queue for when the edges exit.
For the 2 edges being cut case, assume the 2 edges being cut belong to different pieces of a centroid decomposition. Then the thing to note is that an edge from the ‘lower’ tree incurs a cost equaling to its weight to everything in the ‘upper’ tree that’s not in the subtree of its other end point. So in the pre-order layout of the tree, this is a range increment of everything outside of a subtree. There is several layers of recursion involved, so the overall runtime comes out to $O(n \log^3 n)$ if done naively. I think [https://arxiv.org/abs/1911.01145](https://arxiv.org/abs/1911.01145) gets this all the way down to $O(m \log n)$.

For the rest of this lecture we focus on the data structure, which is a ‘standard’ recursive $B$-tree. Each node tracks the sum/minimum in its subtree, as well as how much ‘bulk’ has been updated to it. The key is to ‘scan’ downward, and push all the updated values downward. Then we can combine the min values of the next layer, upward. This technique in general is referred to as ‘lazy propagation’.

I claim it’s also useful to think about how lazy propagation works in a 2- and 3-level settings.

Things get far more complicated if I introduce a ‘SET$_\geq$/SET$_\leq$’ operations. That is, for everything in $A[i...r]$, I set $A[i]$ to be at least/at most $x$, which formally is

$$A[i] \leftarrow \max\{A[i], x\},$$

for SET$_\geq$, and for 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 $O(\log n)$. 