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.

In this lecture I want to turn tree based dynamic programming around into data structures. Recall that the main idea of tree based dynamic programming is that in a rooted tree, we can compute the dynamic program value of a node $p$ by merging together information of all its children.

For tree based dynamic programs, we used it for a states related to the problem, defined for an arbitrary tree given for the input. In this lecture, we will use a particular choice of tree, namely the complete binary tree / heap, to turn this into data structures for ranges of numbers.

To start, consider the dynamic version of the range sum problem:

**Problem 6.1.** Maintain an array $A[1 \ldots n]$ to support the operations

1. Modify $A[i]$,
2. Query $\text{Sum}(A[1 \ldots i])$

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

The way to set up tree base data structures is to use the value we want to record as the dynamic program state. Specifically, for a node $p$, we let $\text{Sum}[p]$ denote the sum of its entire subtree. That is, we get the DP transition

$$\text{Sum}[p] = \text{Sum}[p.left] + \text{Sum}[p.right].$$

Now consider what happens when we update an entry in the array: the only nodes whose sum change are those who subtrees contain $i$. In other words, the nodes that are ancestors of $i$ are the ones whose values needed to be updated, and the cost of that is the depth of $i$.

So the most important aspect of tree based data structures for tracking range statistics is that the user picks a short tree, namely the balanced binary search tree. That’s how the costs of all operations are kept to $O(\log n)$.

Then there is the query cost: for this, it’s easier to think about finding sum of $[1 \ldots i)$, that is, all the indices strictly before $i$. This is because we can simply take index $i$, and walk upward from it to root. Any time node is a right child, we know that its left sibling’s entire subtree is to the left of the path, and should thus be included in the sum. So conceptually, this takes $O(\log n)$ time as well.
There is then the question of how to implement such a tree. Here my favorite method (which is also what’s commonly used if you read others’ codes on these online auto judges) is the heap layout. That is, we set node $p$’s parent to be $\lfloor p/2 \rfloor$. This in turn gives:

```c
p.left = p * 2;
p.right = p * 2 + 1;
```

Then we put all the ‘actual’ vertices at the bottom-most layer of the tree. That is, if we set root to be 1, then a direct way to do this mapping is

$$i \rightarrow 2^d + i$$

where $d$ is the depth of the tree.

This convention takes a while to get used to, but once you get familiar with it, you will not want to code balanced binary search trees anymore. As an effort to convince you of how easy such statically balanced binary search trees are, here are the pseudo-codes for update:

```c
void Update(int i, int x) {
    int p = (1<<d) + i;
    tree[p] = x;
    while(p != 1) {
        p /= 2;
        tree[p] = tree[p * 2] + tree[p * 2 + 1];
    }
}
```

and querying prefix sums:

```c
int PrefixSum(i) {
    int p = (1<<d) + i;
    int s = tree[p];
    while(p != 1) {
        if(p % 2 == 1) { \is right child
            s += tree[p - 1]; \add value of left sibling
        }
        p /= 2;
    }
    return s;
}
```

Note that for sum, we can get the sum of $A[l \ldots r]$ via $A[1 \ldots r] - A[1 \ldots l - 1]$, so querying prefixes actually give us sums for arbitrary ranges.

For some other statistics, such as max, we need to get the ranges more directly. That is, we will need to do a top-down recursion instead starting from the root vertex 1. Here
the easiest way I know of is a 5-parameter recursion involving the range corresponding to the current node, and the query range. At each step we pick the mid point, and split the range accordingly. Code for this is below, but this is only something that I’m accustomed to: there are bottom-up ways of doing this (using indices similar to ones above) that are about 1.5x faster.

```c
int RangeMax(int p, int l_p, int r_p, int l_q, int r_q) {
  if(l_p == l_q && r_p == r_q) return tree[p];
  else {
    int mid = (l_p + r_p) / 2;
    if(r_q <= mid) return RangeMax(p * 2, l_p, mid, l_q, r_q);
    else if(mid < l_q) return RangeMax(p * 2 + 1, mid + 1, r_p, l_q, r_q);
    else return max(RangeMax(p * 2, l_p, mid, l_q, mid),
                     RangeMax(p * 2 + 1, mid + 1, r_p, mid + 1, r_q));
  }
}
```

The advantage of this kind of recursive queries is that we can also push flags downward the tree as we query/modify things (e.g. we want to incremental a whole range, while still querying range sums). This is something called ‘lazy propagation’ that we will cover next week.

Note that this type of structure can also query for the global minimum in a range as we update: it has strictly more functionality than a heap! Furthermore, note that by varying the branching factor, one can get depth $d$ trees with branching factor $n^{1/d}$. This branching factor shows up in the cost of checking $n^{1/d}$ siblings during queries, while the gain comes from the smaller height of the tree. Formally, we can encapsulate this with the following problem:

**Problem 6.2.** Support entry modify in $d$ time, and queries in $O(d + n^{1/d})$ time.

Note for this type of things, big-O time complexity essentially breaks down, and probably for good reason. What one actually want to do is to set $n^{1/d}$ to be the size of a cache line, which for ints is somewhere between 64 and 256 ints. The more formal name for such storage is $B$-trees, and they are widely used in databases.

Anyways, enough theorycrafting. Here is a list of problems that can be solved using such data structures:

**Problem 6.3.** Given a sequence $s[1\ldots n]$, count the number of inversions, that is, $i < j$ with $s[i] > s[j]$, in $O(n \log n)$ time.

This problem is available at [https://dmoj.ca/problem/ccc05s5](https://dmoj.ca/problem/ccc05s5).

We go through the indices, as we encounter $s[i]$, we increment $A[s[i]]$. Then by the time we encounter $j$, the number of inversions involving is is

$$\text{SUM}(A[s[j] + 1, \ldots])$$

which is the sum of a suffix, and can be transformed into the sum of a prefix.
Problem 6.4. Find the longest increasing subsequence of a list of \(n\) numbers in \(O(n \log n)\) time.

For this, note that for \(DP[i]\), we are looking for

\[
\max_{j<i, A[j]<A[i]} DP[j],
\]

which is again a range search.

It can even more or less replace binary search trees.

Problem 6.5. Maintain a set of numbers in \([1, U]\) that supports insertion/deletion and predecessor/successor queries in \(O(\log U)\) time.

Note that \(Pred(i)\) is the max entry with value < \(i\), and can be answered using a range query. In case where \(U\) is really large (e.g. 64-bit integers), allocating the tree dynamically also leads to \(O(\log U)\) time, but also \(O(\log U)\) memory per call.

My general herusitic for when one really needs a binary search tree is only when interval flips happen: that is, you take an entire interval of numbers and reverse the order. In other words, almost never in the real world.

We can also take these problems to higher dimensions, namely checking whether there is a point that’s contained by another.

Problem 6.6. Given \(n\) 3-tuples \((a_i, b_i, c_i)\), for each \(i\) determine if there is some \(j\) such that \(j\) is better in all three aspects, aka.

\[a_i < a_j, b_i < b_j, c_i < c_j\]

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

First, note that we can sort the objects. We sort them in decreasing order of \(a\). Then for each \(i\), we need to find

\[
\max_{j>i, b_j>b_i} c_j,
\]

which is a range max query. Note that the set can also be maintained by sweeping downward in \(i\), while adding the \((b_i, c_i)\) pair to the tree.

We can also take this one step further to get

Problem 6.7. Given \(n\) 4-tuples \((a_i, b_i, c_i)\), for each \(i\) determine if there is some \(j\) such that \(j\) is better in all three aspects, aka.

\[a_i < a_j, b_i < b_j, c_i < c_j\]

in total time \(O(n \log^2 n)\).

This is still doable with just 1-D augmented search trees, but with an additional layer of divide-and-conquer.