15-451 Algorithms, Spring 2016
Recitation #14 Worksheet

Segment Trees for Computational Geometry

Consider the following API which maintains a set of variables $a_0, a_1, \ldots, a_{n-1}$. ($n$ is fixed.)

addToRange(i, j, x):	Add x to each of $a_i, a_{i+1} \ldots, a_j$.
get(i):	Get te value of a_i.
sumRange(i, j):	Return $a_i + \cdots + a_j$.
maxRange(i, j):	Return $\max(a_i, \ldots, a_j)$.

We would like to support all these operations in $O(\log n)$ time and $O(n)$ space. Note that a naive array would give $O(1)$ insert, but all other operations would be $O(n)$ worst-case, if the range were large.

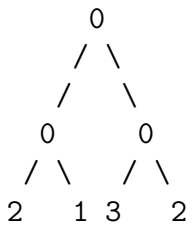
For the moment, lets discuss how to implement addToRange and get. We'll look at the others later.

Assume n is a power of two. If not, we can just round it up to the nearest one and leave the extra a_is at 0. We want to construct a balanced binary tree with n leaves.

This can be implemented as an array of $2N$ numbers, where the root is at index 1, and the leaves are at indices $N, \ldots, 2N-2$. The parent of node i at $\lfloor i/2 \rfloor$, and its children at $2i$ and $2i+1$.

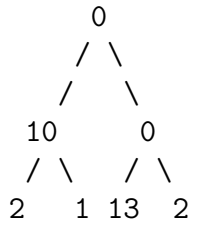
To solve the problem at hand, we store the values of the as in the leaves of the tree. In each internal node we store 0 in the beginning. We can call these values of the internal nodes "propagate" values, and they contain a number to add to all of the leaves which descend from them. Lets call these descendant leaves the "range" of the node.

For example, suppose that the $a[]$ values are $[2, 1, 3, 2]$. Then here is what is stored in the tree (a.k.a. array) initially:



In order to add a value c to a range $[i, j]$, we do the following recursive procedure, starting at the root: If the current node's range is fully contained by the $[i, j]$, add c to the propagate value of this node. If the node's range does not intersect the range at all, do nothing. If it partially intersects, recursively call this on the left and right children.

Adding 10 to the range $[0, 2]$ would cause the above tree to look like:



The recursive case can only happen at nodes on the paths down to either i or j, and there are $O(\log n)$ of these. Every case only does constant work, so we only do $O(\log n)$ work.

To query a value, we go down the path from the root to the child. This is $O(\log n)$ work. You can verify that this works on the tree above.

To implement sumRange and maxRange, we want to store, at each node, the sum and max of its range. This can be done similarly to how sizes are stored in augmented search trees: Whenever a node x has a child that is updated, we can set

```
x.sum = x.left.sum + x.right.sum + x.propogate*x.size
x.max = max(x.left.max + x.right.max) + x.propogate
```

And then, the parent would be updated. We can see that, since updates always go down the tree from the root, this will add only a constant factor of work.

In order query sumRange, we do the following recursive procedure, starting at the root: If the current node's range (the range of leaves below it) is fully subsumed by the range, return its sum. If the node's range does not intersect the range at all, return 0. If it partially intersects, return the sum of recursively calling this on the left and right children.

Exercise to the reader: how would you implement maxRange?

Segment Tree Algorithms

1. There is an apartment building with n floors. Every day, some kids throw a *really* big egg at the building, which makes some range $[i, j]$ of floors smell gross. At each day, the owners of the hotel want to know which floor has been hit by the most eggs, and how many eggs it was hit by, in order to write them a nice letter. On each day, the janitor will input the range into the computer. Your program to determine this should only run for $O(\log n)$ time each day.

2. You are given n vertical lines and n horizontal lines. Find how many intersections there are in $O(n \log n)$ time. Hint: sweepline.