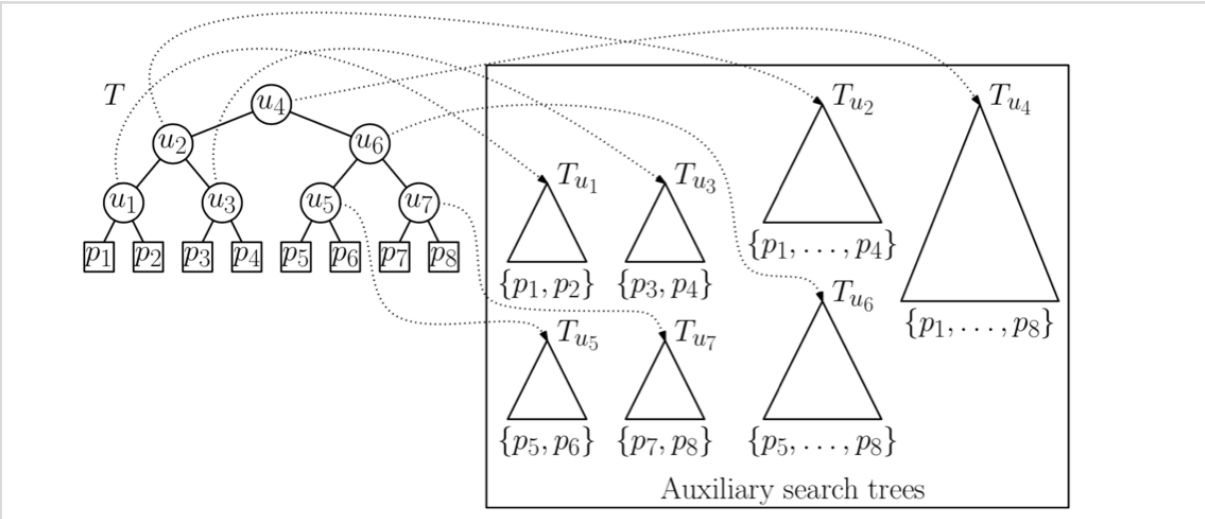
# CS 6301.002.20S Lecture 20—April 2, 2020

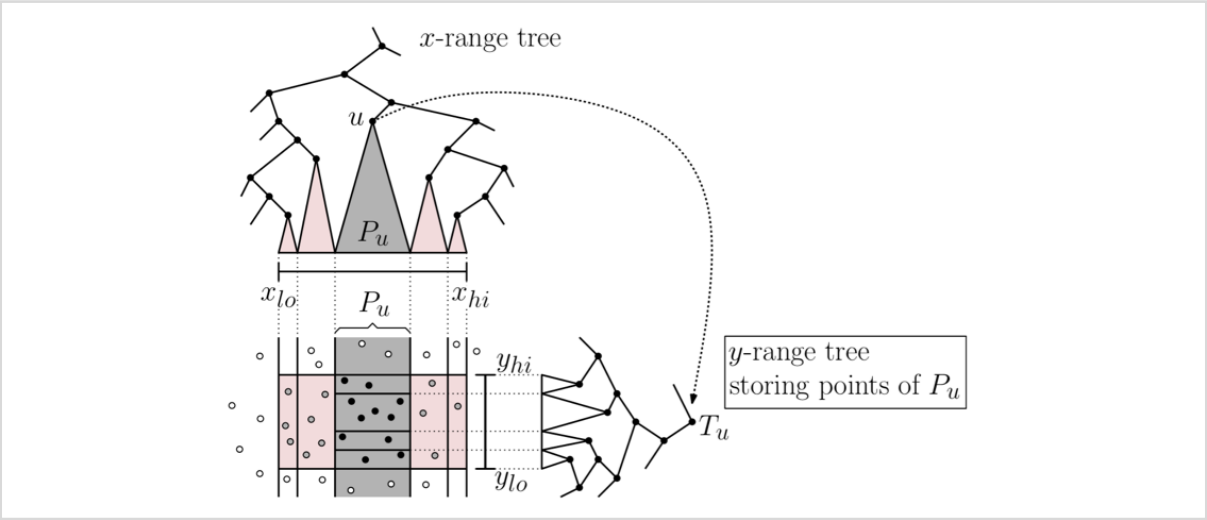
Main topics are #orthogonal_range_trees, #fractional_cascading, and #interval_trees.

## Orthogonal Range Trees

- Last time, we discussed data structures for performing *orthogonal rectangular range queries*. Given a set of n points P, build a data structure so one can quickly count or report all points lying within any arbitrary query rectangle Q.
- We saw a 1D partition tree that used O(n) space. Counting queries take O(log n) time and reporting queries take O(log n + k) time where k is the number of points to report.
- For R^d, we saw the kd-tree. It also uses O(n) space. Counting queries take O(n^{1 - 1 / d}) time and reporting takes O(n^{1 - 1 / d} + k) time.
- Today, we'll use a bit more space to substantially speed up those queries.
- An *orthogonal range tree* uses O(n log^{d-1} n) space and performs counting queries in O(log^{d-1} n) time.
- We'll start with a slightly simplified version for the plane that uses O(n log n) space and has O(log^2 n) query time. Later, we'll see how to get the query time down to O(log n).
- The key idea is not to mix sorting by x and y value anymore, but to separate them using what is called a *multi-level search tree*.
- Say we have a query rectangle Q = [x_lo, x_hi] X [y_lo, y_hi]. Let Q_1 = [x_lo, x_hi] X R be a vertical strip and Q_2 = [y_lo, y_hi] X R be a horizontal strip. Query range Q is simply their intersection.
- We'll try searching for points in Q_1 first, and then we'll dig deeper to find points in the intersection.
- We already saw a way to search Q_1, we can use a partition tree for one-dimensional range queries.



- In the tree, each node u was associated with a canonical subset P_u. A search consists of finding a set of nodes with disjoint canonical subsets making up Q_1 intersect P.
- But since every point of each canonical subset lies in Q_1, we can safely report all of its points that lie in Q_2 as well by restricting our search to each canonical subset of Q_1 independently.
- And to make that efficient, we'll store an *auxiliary search tree* for the canonical subset P_u of each node u. The canonical subsets are disjoint, so results from searching these auxiliary search trees will be disjoint as well and we can safely add or merge them.
- So in summary, a *2-dimensional range tree* consists of a two levels: an x-range tree T where each node u points to an auxiliary y-range tree T_u over its canonical subset P_u.
- For a query, we find a collection of O(log n) nodes whose canonical subsets have the correct x-values for our range. For each P_u, we do a second search in T_u to count or report the points of P_u with the correct y-values.

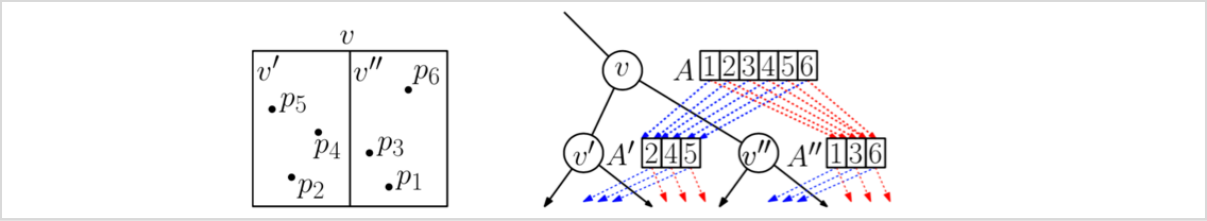



- There are O(log n) searches in auxiliary trees, each taking O(log n) time, so the query time is for counting is O(log^2 n). For reporting, we can report the leaves from each auxiliary search tree in time proportional to the number of points in it, so reporting k points total takes O(log^2 n + k) time.
- But how much space do we use? Each point appears at most once in each auxiliary search tree, and it appears in one auxiliary search tree for each ancestor of its leaf in T. So each point appears O(log n) times for a total space usage of O(n log n).
- We can build the data structure in O(n log n) time as well by doing so in a bottom-up manner. Whenever we want to build T_u for a node u, we look at the auxiliary structures for its two children. They already have their points sorted by y-value, so we can merge them and build T_u in time O(|P_u|). So everything ends up taking time linear in the size of the data structure.
- In d-dimensions, you have a top level tree for the first coordinates, and then auxiliary (d-1)-dimensional range trees at each node to take care of the remaining coordinates. You get a structure of size O(n log^{d-1} n) with O(log^d n) query time—that's one log factor per

level of the search tree.

## Fractional Cascading

- Can we do better? It turns out we can speed up the queries by eliminating some redundant work using a technique called *fractional cascading*.
- The problem is that we're doing a lot of separate O(log n) time searches even though each of them uses the same pair of search keys, y_lo and y_hi.
- The main idea is that once we know where the lowest point higher than y_lo lies in one canonical subset P_u, we should immediately know the lowest such point in the canonical subsets for the children of u.
- Let's assume the auxiliary structures aren't trees, but instead arrays sorted by y-value which we'll call *auxiliary lists*. We'll also stick to reporting queries. If you can find the least point in the array for P_u above y_lo, you can easily find all the other points of P_u in the range by just walking forward along the array and stopping as soon as you've gone too far.
- Say we have a node v in the top level tree with two children nodes v′ and v″. For each element in the auxiliary list for v, we'll add a pointer to the least element in the list for v′ which has higher y-value. We'll also add one to the least element in the list for v″ which has higher y-value.

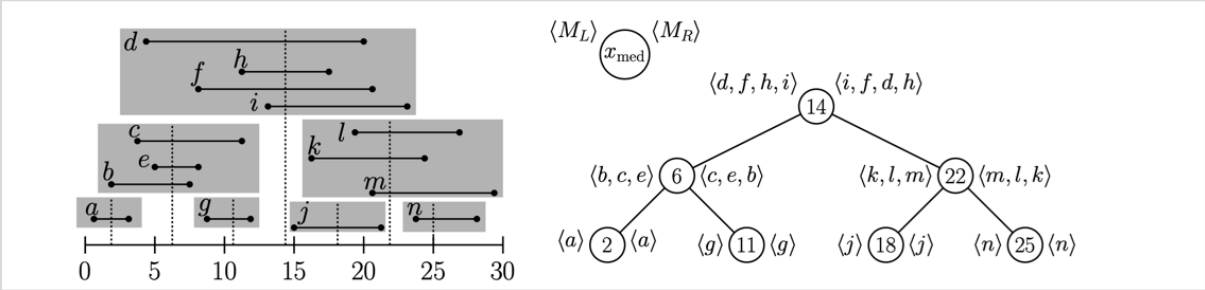



- For a query, we'll start by searching the root's node's list for the lowest point higher than y_lo.
- Now, as we do our search for points in Q_1, we'll be able to in only O(1) time per node follow those pointers we added and learn the lowest point higher than y_lo for every auxiliary list for every node we touch.
- Meaning we spend no additional time doing searches in auxiliary lists once we know the canonical subsets for Q_1.
- Reporting now takes O(log n + k) time: we find the starting positions for all the auxiliary lists in O(log n) time total and then walk along them in O(k) time total to return the points.
- Counting as I defined it can be done in O(log n) time, but you have to be a bit more clever since we're working with arrays instead of trees for the auxiliary structures. In short, store the prefix sum at every element in the lists and do some subtractions to get your counts for relevant members of the lists. Basically, the index trick a student mentioned on Tuesday.
- Unfortunately, fractional cascading only works at the lowest level of a d-dimensional range tree, because it crucially depends upon you only needing to search for a constant number of things in the auxiliary structure. We don't have time to find a bunch of canonical subsets

for which to do things recursively. Query times in d dimensions go down to O(log^{d-1} n).

## Interval Trees

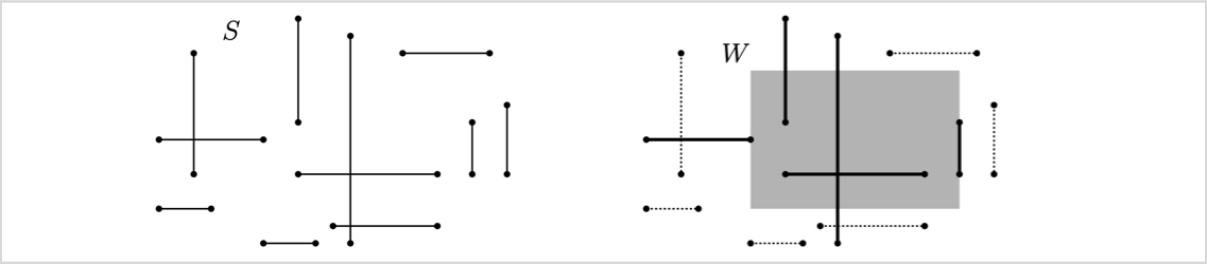
- We have a little extra time, so let's discuss one more data structure. It's used as part of a more complicated data structure, but we likely don't have much time to discuss it since we lost a week.
- We're given a set I = {[x_1, x'_1], [x_2, x'_2], …, [x_n, x'_n]} of n *intervals* in 1D that we want to preprocess. Given a query point q, we want to quickly report which of the line segments contains q. This is sometimes called a *stabbing query*. Let's just focus on reporting for today and not counting.
- We'll build a data structure called an *interval tree*.
- Again, we'll try to base it on a balanced binary tree. Except now, the intervals overlap so it's hard to give them a strict ordering.
- Instead, we'll start by sorting the endpoints. Let x_med be the median of the 2n endpoints.
- Let L be the intervals strictly to the left of x_med, R be those strictly to the right, and M be those that contain x_med. Any query point q will miss every interval in R or L if it is to the left or right of x_med, respectively. Therefore, we can recursively build interval trees for L and R and make them the children subtrees of the root node. During a query, we'll recursively do reporting in exactly one of those subtrees.
- But which intervals of M do we report? Suppose q ≤ x_med. Each interval [x_i, x'_i] of M passes through x_med and will therefore contain q if and only if x_i ≤ q.
- So let's store all intervals of M twice: once as a list M_L sorted by left endpoint and once as a list M_R sorted by right endpoint. We can quickly scan one of those lists during a query.



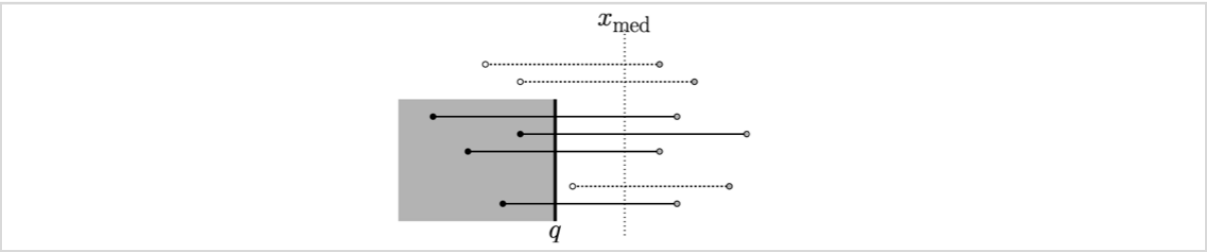

- There will be at most n leaves, and each interval is stored in an M_L or M_R list once. The total size of the interval tree is O(n).
- Here's how we do a query q: Suppose q ≤ x_med. We scan the root's M_L list, reporting each interval [x_i, x'_i] such that x_i ≤ q in time proportional to the number of intervals in M we report. Then we recursively do a stabbing query for L. If q > x_med, you instead scan M_R and then recurse in R.
- Sets L and R are at most half the size of I. So the depth of the tree is O(log n). We touch O(log n) nodes and report k intervals, so a stabbing query takes O(log n + k) time total.

## Window Queries for Orthogonal Segments

- Both 3Marks and Mount describe (modified) interval trees as one small part of a bigger data structure.
- We're given n axis-aligned line segments S in the plane that we want to preprocess. We want to perform *window queries* where we're given a closed axis-aligned query rectangle W, and we want to report all intervals that intersect W.




- Some of these segments are easy to repot. Suppose an interval has an endpoint inside the query window. We can quickly find all such intervals by building a 2-dimensional range tree over the interval endpoints. The range tree may report both endpoints of an interval completely contained W, but we can avoid double reporting by marking intersected intervals we have already reported.
- But how will we handle the remaining intervals that pass all the way through W? These horizontal segments of S pass through the left side of W and these vertical segments of S pass through the bottom side of W.
- What we need is a way to perform *vertical* and *horizontal segment stabbing queries*. We'll focus just on stabbing horizontal segments of S with a query vertical segment q = (x_q, y_lo) (x_q, y_hi).
- We *almost* solved this problem earlier using interval trees. We can create two sets of horizontal segments L and R as before, based on whether they lie to the left or right of a splitting line x = x_med.
- But it's not enough to handle M by considering only x-coordinates, because q only uses a subset of y-coordinates.
- Instead, we have the following observation: if x_q ≤ x_med, then a segment of M with left endpoint (x, y) intersects q if and only if x ≤ x_q and y_lo ≤ y ≤ y_hi.




- But that's just an orthogonal rectangular range query (with the left side of the rectangle at -infinity).
- Instead of sorted lists, we'll use 2-dimensional range trees for M_L and M_R. Each interval is still involved in a constant number of range trees, so the total space for our data

structure will be O(n log n). During a windowing query, we search up to O(log n) range trees, so the total query time will be O(log^2 n + k), assuming we do fractional cascading.