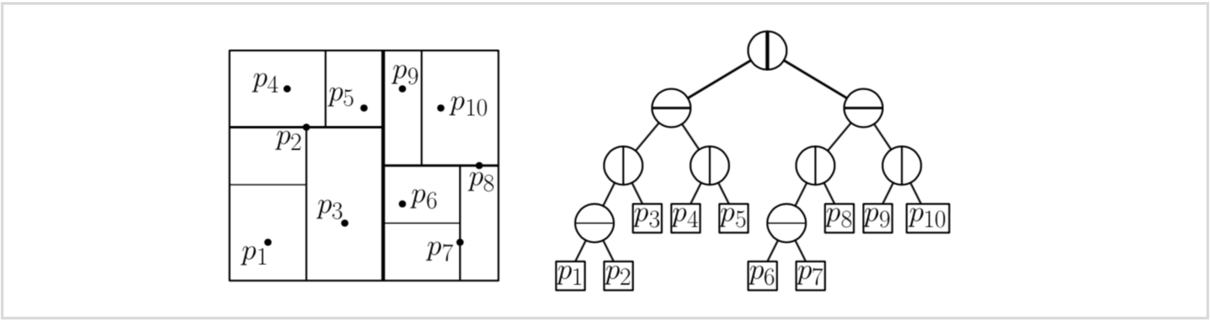
CS 6301.008.18S Lecture—February 1, 2017

Main topics are #range_searching and #kd-trees.

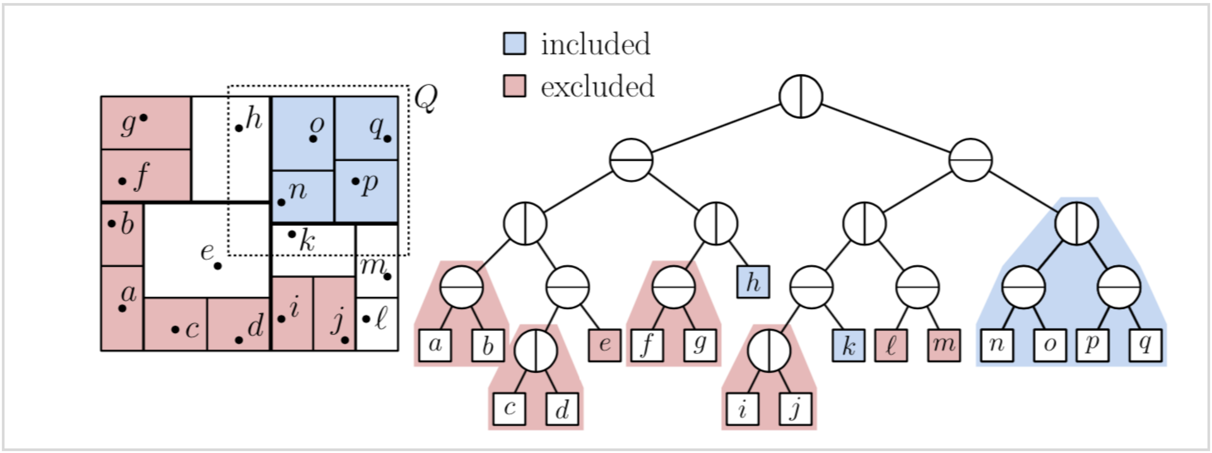
kd-trees

- Last time, we discussed the following problem: Given a point set P = {p_1, …, p_n} in 2D, build a data structure to help with *orthogonal rectangular range queries*: Given an axis-parallel rectangle Q, count or report every point of P lying in Q.
- We'll discussed a data structure called the kd-tree, designed by [Bentley '75].
- kd-trees are *partition trees*, meaning they store the points at the leaves and the canonical subset for a node contains the points in its leaves. At each node, we subdivide its point set by splitting them evenly based on their x-coordinate or y-coordinate.
- Each node t stores
  - t.cut-dim: which way we'll split the points (so 0 for x and 1 for y)
  - t.cut-val: at which x or y coordinate should we split the points
  - t.weight: the total weight or number of points in t's subtree
- Nodes with one point are the leaves. They store that point as t.point.



- If you zoom out a bit, here is the picture you see. Each node represents a rectangular region of space called a *cell*. The root's cell can be thought of as a big rectangle surrounding all the points. When you split a node's points, it's like you're splitting its cell into two smaller cells for that node's children.
- These nested cells are sometimes called a *hierarchical space decomposition*.
- Now, there's a few ways you can pick cut dimension and cut value. The standard way is to alternate between x and y as the cut dimension as I drew, and always set cut-val to be the median value so you split the points into two equal subsets.
- You can build this thing in O(n log n) time by first making two sorted lists of points by x-coordinate and y-coordinate. Then you can search for the median coordinate for each split and split up the lists to recursively build the two subtrees in time linear in the number of points in a subtree. You get a recurrence like T(n) = 2T(n/2) + n which solves to O(n log n).
- It's a balanced binary tree with O(n) leaves, so its size is O(n).
- We can do range counting with the following procedure:

- RangeCount(Q: the range, u: a node):
    - if u is a leaf
        - if u.point in Q, return u.weight
        - else return 0
    - else
        - if u.cell intersect Q = emptyset, return 0
        - else if u.cell subset Q, return u.weight
        - else, return RangeCount(Q, u.left) + RangeCount(Q, u.right)
- Basically, return the whole count if the whole cell lies in the range. Or return 0 if the whole cell lies outside the range. Otherwise, search deeper.



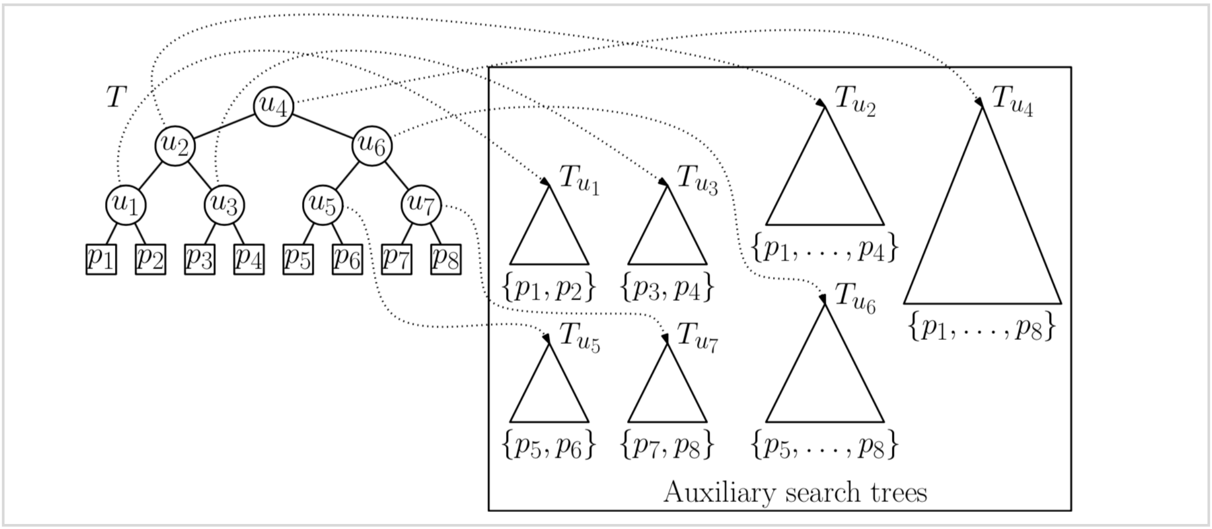
- If you want to report, then instead of returning the weight, you return a list of all the node's descendent leaves.
- How long does a query take? Call a node *expanded* if it and both children are visited by the recursive counting algorithm. Except for the root, each visited node has an expanded parent, so the running time is proportional to the number of expanded nodes.
- A cell is *stabbed* if it has a strict intersection with the range. There are more stabbed cells than expanded cells, so we'll just count those.
- Lemma: Any horizontal or vertical line stabs O(sqrt(n)) cells of the tree.
- Proof:
    - Suppose the line is vertical. Consider a node with cutting dimension x. The line stabs at most one of its children, and if it fails to stab a child, then it won't stab any descendent of that child.
    - However, a node with cutting dimension y may have both children stabbed.
    - Therefore, each node with cutting dimension x has at most two *grandchildren* stabbed. In general, the number of stabbed nodes increases by a factor of at most two every two levels of the tree.
    - Let S(n) be the maximum number of stabbed nodes by the vertical line. S(n) = 2 if n ≤ 4 and S(n) = 1 + 2T(n/4) otherwise.
    - This solves to S(n) = O(2^{log_4 n}) = O(n^(1/2)) = O(sqrt(n)).
- OK, for query range Q to stab a cell, that cell must be stabbed by at least one of the four

line segments bounding Q. But each of those stabs O(sqrt(n)) cells, so O(sqrt(n)) cells are stabbed or expanded. A counting query takes O(sqrt(n)) time. A reporting query takes O(sqrt(n) + k) time.

- What about higher dimensions? For those, the kd-tree splits along each of the d dimensions, the 0th, the 1st, the 2nd, and so on then back to repeat the list. It still uses O(n) space and can be built in O(n log n) time if d is a constant.
- However, the running time of a counting query increases to O(n^{1-1/d}).
- And even in the plane, you get O(sqrt(n)) which is a fair bit higher than the O(log n) we got on the line. Can we do better?

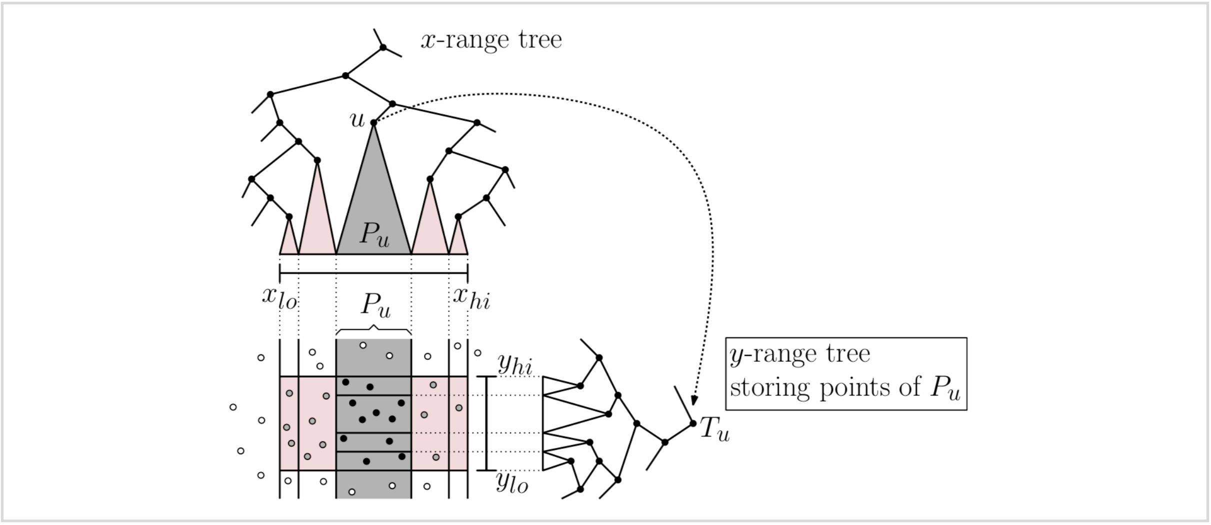
Orthogonal Range Trees

- It turns out we can do a lot better if we increase our space usage by a small amount.
- 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 focus on a slightly simplified version for the plane that uses O(n log n) space and has O(log^2 n) query time. Time permitting, 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.
- So why don't we 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.

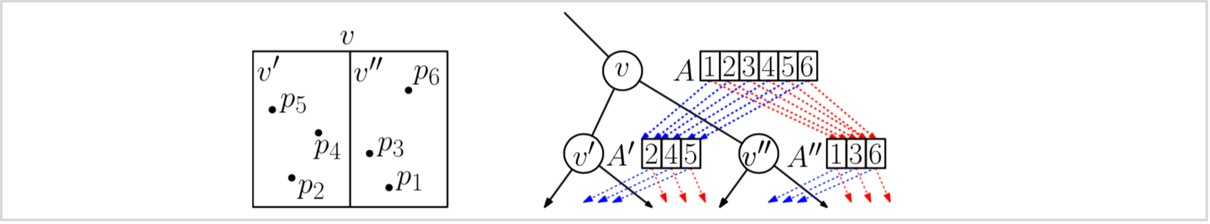
- But since every point of each canonical subset lies in Q_1, we can safely report all of its points that lie in Q_2 as well by restricting our search to each canonical subset of Q_1 independently.
- And to make that efficient, we'll store an *auxiliary search tree* for the canonical subset P_u of each 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 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 in time proportional to the number of points, 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 an 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 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 build 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.
- The problem is that we're doing a lot of O(log n) time searches, but strangely, 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.
- To keep things clean, 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 with two children nodes v' and v''. For each element in v's auxiliary list, 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 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.
- Also, you can only play this trick at the lowest level, since 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. So query times go down to O(log^{d-1} n).