COMS E6998-8: Advanced Data Structures (Spring'19) March 21, 2019

Lecture 8: Orthogonal Range Searching

Instructor: *Omri Weinstein* Scribes: *Evan Ziebart, Mark Dijkstra*

0.1 Plan for this Lecture

- Wrapup from last lecture
- Orthogonal Range Counting (ORC)
- Range Trees (RTs)
- Layered RTs
- Fractional Cascading

0.2 Wrapup $(1+\epsilon, r)$ANN

We have shown a trade-off between space and time complexity

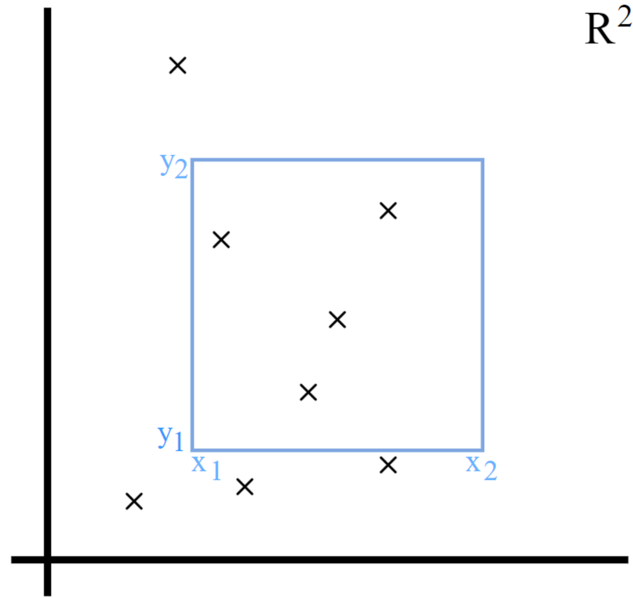
- can do $s = n^{O(1/\epsilon^2)}$ and $t = O(1)$ (optimal)
- can do $s = n^{1+\delta}$ and $t = n^{\delta}$ (optimality not proven)

In [PTW '10], a technique called "Cell Sampling" is used to show $s \geq n^{1+\Omega_\epsilon(1/t)}$. So, if $t = O(lgn)$, there is no improvement. The best we can prove is $t \geq \Omega(\frac{lg(n)}{lg(swd/n)}) \approx \Omega(\frac{lg(n)}{lg(lg(n))})$ (with $d = lg(n)$, $s = n$, $w = lg(n)$). More on cell sampling to come!

1 Orthogonal Range Searching (ORC)

1.1 Problem Statement

Goal of d-dimensional ORC ($dD - ORC$): Preprocess n points $x_1, \ldots, x_n \in \mathbb{R}^d$ such that given a *box*, $[x_1, y_1] \times [x_2, y_2]$ we can either filter, report, or count the number of points in this box. (For this lecture we focus on counting them)



The decision version of this problem is to return whether there exists a point in the box. Applications include computational geometry and spacial databases.

Recall that the fastest query time for NN is n^{δ} which is bad. Good news is that range queries are much easier than NN queries: $(lg^{\lambda}(n))$ and for $d \leq 10$ it is actually only poly-logarithmic.

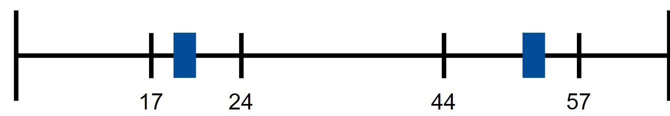
1.2 Sample Query

Find all female employees of Walmart who have been hired between 2006 and 2016 with salary between $77,000 and $230,000. This would help Walmart potentially estimate market size.

- Each axis represents a feature of the data (i.e. salary, year, etc)
- Query just specifies ranges on these axes (x-range, y-range, binary ranges, for example)

1.3 1D ORC

Given a 1D line of integers $x_1, \ldots, x_n$ and a range query $[a, b]$, count the number of points in the range.



Static DS: A first idea is to store with each element how many elements are less than it. Then, the query consists of finding the number of entries less than the successor of a and subtracting this from the number of entries less than the predecessor of b: $result = Pred(b).elt - Succ(a).elt$ (ie: we can use a Predecessor DS from earlier in the class). This is $s = O(n)$, $t = O(lg(lg(n)))$. However, this only works for a static data structure and does not generalize nicely for higher dimensions.

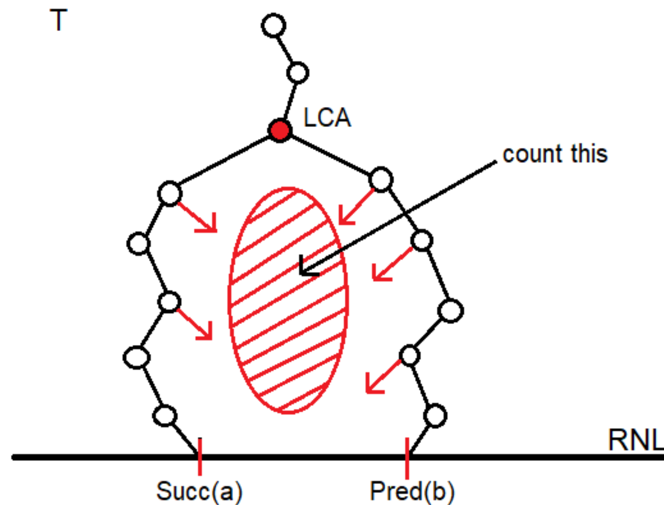
Dynamize with BSTs: This gives $s = O(n)$, $t = lg(n)$ and is fully dynamic. Augment the non-leaf nodes to store the range sums. Need to traverse a single path to leaf on query and update. However, this still doesn't generalize well to higher dimensions. For that, we will introduce a new DS called a Range Tree.

2 Range Trees (RTs)

We will show the 1D case, then talk about the 2D case to see how it can generalize to higher dimensions.

2.1 1D Version

Place all keys into a BST. Augment each internal node $v \in T$ to store the min, max, and number of nodes in left and right subtrees. To query the DS, run a predecessor search on b and a successor search on a, both starting at the root. The node where the searches diverge is the LCA of the desired subrange. To get the range itself, continue the searches down the tree. Anytime the predecessor search on b follows a right child, count the nodes in the left subtree. Symmetrically, anytime the successor search on a follows a left child, count the nodes in the right subtree. Also count both leaves in the searches.

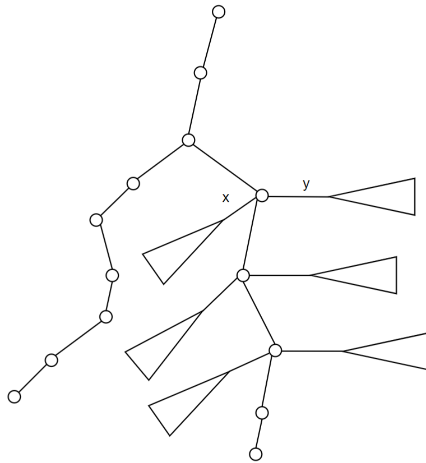


A Range Tree built on top of the 1-d dimensional problem.

2.2 2D Version

We will take advantage of the Cartesian product structure of higher dimensional spaces.

1. Project X coordinates onto the axis, then run a range tree on the projection of x. We count everything within the first dimension query, so this means there are no items outside of the x-range in our result; however, there may be items outside of our y-range.
2. Project y coordinates of the resultant trees from the set constructed above. Now we will do a range search on the y coordinates. We do this by building a new range tree for all nodes of the previously built range tree for the x dimensions.



Thus: Build 1-D range tree on x-coordinate of point set $P = \{(x, y)\} \in \mathbb{R}^2$
For every internal node $v \in \mathcal{T}_x$ (corresponds to x-range $[v_1, v_2]$) build a 1-d range tree on the $\{P_y|(x, y) \leftarrow P, x \in [v_1, v_2]\}$
We build a y-range tree on top of all nodes that correspond to an x-range we are interested in. This will give us the box that we are interested in querying.
query time $= O(lg^2 n)$
$S = O(n\, lgn)$ (each tree appears in at least lg(n) trees).

In higher dimensions
$t = O(lg^d(n))$
$S = O(n * lg^{d-1}(n))$
These can be shown inductively.

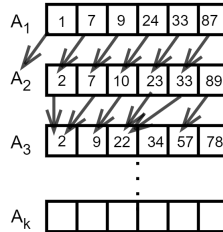
Is this optimal?
Intuitively there is room for improvement, because we perform the same predecessor query on y $\forall$ y-Range trees along the search path for all of the internal nodes in the x-Range tree. Thus, we hope to be able to re-use searches.

The solution is *cross linking*, which is a special case of *fractional cascading*.

3 Layered Range Trees

3.1 Cross-Linking

Consider:

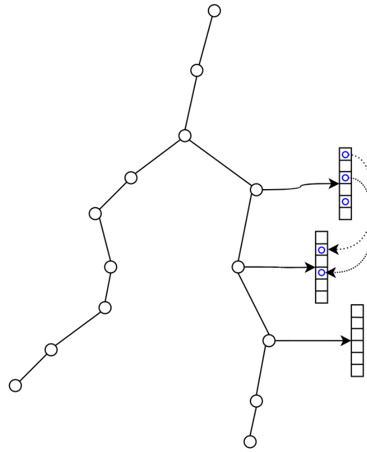


We have k arrays, and want to perform predecessor query q on each of them.

A naive method has us calling $Pred(q)$ on each $(t = O(k \cdot lg(n)))$. Better is to store pointers in each element to its predecessor in the array below. Now, just perform the query on the first array, then follow predecessor pointers through the rest of the arrays. This improves the runtime to $t = O(k + lg(n))$. This technique of adding pointers to the elements in the arrays is called *cross linking*.

There is 1 bug in the above scheme: it only works when the range of each array is a superset of the range of the array below it ($A_{i+1} \subseteq A_i \forall i$). We will show that this is always satisfied in the construction of a range tree. Later, we will show the method by which generalized *fractional cascading* gets around this bug.

3.2 Constructing LRTs



Layered Range Tree

Cross-linking makes the LRT work. If we have 2 dimensions x and y, store a list of corresponding y-coords at each range node in a RT over x. Because of the structure of the tree, it is guaranteed that a lower array will be a subset of a higher one. This means the arrays along a path from root to leaf can be cross-linked. Moreover, this will generalize to higher dimensional spaces.

Analysis

- 2D: $t = O(lg(n))$ and $s = O(n \cdot lg(n))$
- dD LRT: $t = O(lg^{d-1}n)$ and $P = s = O(n \cdot lg^{d-1}n)$
- In [Chazelle '95], arithmetic model where this D.S. is optimal, $S = O(n \cdot lg^{d-1}n)$ and $t = \Omega\big(\big(\frac{lg(n)}{lglg(n)}\big)^{d-1}\big)$

Open Question: Do there exist higher lower bounds for truly linear space. $s = O(n)$?

3.3 Dynamization

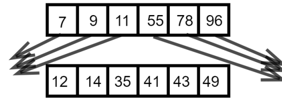
This seems hard at first because of all the pointers and preprocessing necessary to build a LRT. However, we can achieve a good amortized time by "occasionally" reconstructing the DS.

Observation: Range counting/ORC is a decomposable search problem. Recall a search problem is *decomposable* if $P(x_1 \cup x_2, q = g(P(x_1, q), P(x_2, q))$. This means we can use the Segment Tree from lecture 6 to dynamize the LRT. In general, if $\exists$ a static DS P with preprocessing time $p(n)$ and query time $\leq t(n)$, then we can construct a dynamic DS (for insertions only) with $\hat{t_u} \leq O(\frac{p(n)}{n} \cdot lg(n))$. For a LRT, this means $\hat{t_u} \leq O(\frac{n \cdot lg^{d-1}(n)}{n} \cdot lg(n)) = O(lg^d(n))$, and $\hat{t_q} = O(lg^{d-1}(n))$.

Next Lecture, we will show for a 2D ORC dynamic DS, $max\{t_q, t_u\} \geq \Omega(lg_w^2(n))$.

4 General Fractional Cascading

Note that cross linking is not enough in the following case:



The problem is that all the elements in the top array are between two of the elements in the bottom array. This means a search on the top array gives no information about the bottom array. One possibility is to merge the two arrays, but this is slow and expensive: $t = lg(nk)$ and $s = O(nk^2)$

Alternately, merge every second element. We say recursively $A_{i,next} := A_{i,prev} \cup \frac{1}{2}A_{i-1,next}$. Then, $|A_{i,next}| = n + \frac{1}{2}|A_{i-1,next}| = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots \leq 2n$. this gives $S = O(nk)$ and $t = O(k + lgn)$.

Main benefit: Keep every second element means you can be off by at most one search via a standard predecessor search; that is, you may not get the correct query off of the initial predecessor query, but it only takes 1 look over to the right and then you can answer the predecessor query question.

Finally, note that this idea can be generalized to graph DSs as well.