Computational Geometry

Orthogonal Range Queries: Fractional Cascading

Thomas Bläsius

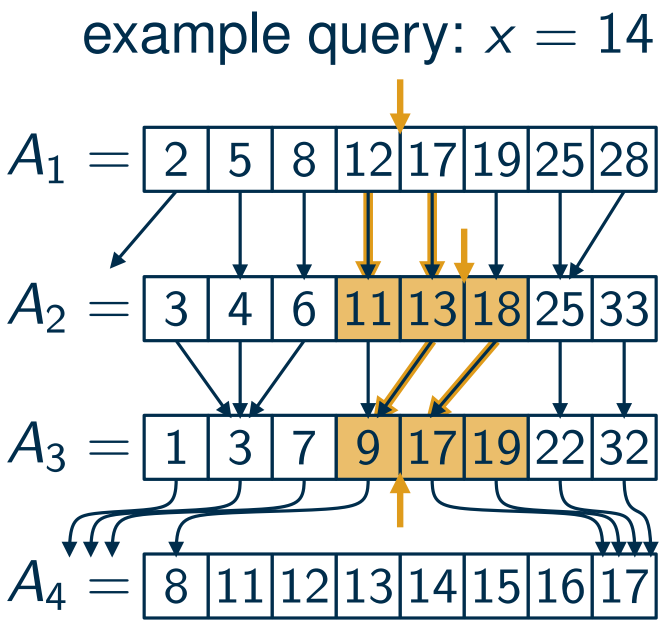
Searching In Many Arrays

Situation

- consider ℓ sorted arrays $A_1, \dots, A_\ell$ with $\leq n$ elements each
- find the position of x in all arrays
- obvious solution: $O(\ell \log n)$
- last lecture: $O(\ell + \log n)$ if $A_1 \supseteq A_2 \supseteq \cdots \supseteq A_\ell$

Is $\ell + \log n$ Possible In General?

- hope: search x in A_1, find x in $A_2, \dots, A_\ell$ via pointers
- problem: position of x in A_i may not help to find position in A_{i+1}



Observation

- $A_i \supseteq A_{i+1} \Rightarrow$ position in A_i determines position in A_{i+1}
- A_i contains many elements from $A_{i+1} \Rightarrow$ position in A_i roughly determines position in A_{i+1}
- idea: insert some elements from A_{i+1} into A_i

Fractional Cascading

Shared Elements

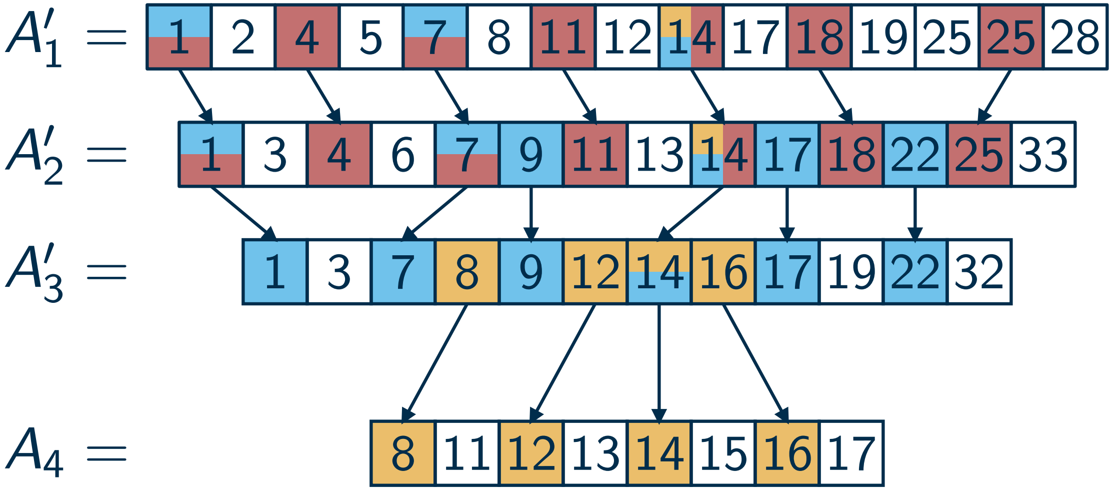
- new array A'_3: insert every other element from A_4 into A_3
- store pointers to copies
- pointers from $A'_3 \setminus A_4$ to prev / next element from A_4 $\Rightarrow$ position in A'_3 gives position in A_4 (± 1)
- pointers from elements in A_4 to prev / next in $A'_3 \setminus A_4 \Rightarrow$ position in A'_3 gives position in A_3
- cascade the process for all previous A_i

$A_1 = [2, 5, 8, 12, 17, 19, 25, 28]$

$A_2 = [3, 4, 6, 11, 13, 18, 25, 33]$

$A_3 = [1, 3, 7, 9, 17, 19, 22, 32]$

$A_4 = [8, 11, 12, 13, 14, 15, 16, 17]$



Fractional Cascading – Running Time

Cost For The Search

- one search in $A'_1 \to O(\log(|A'_1|))$
- $O(1)$ for every subsequent array $\to O(\ell)$

$\}$ total: $O(\ell + \log(|A'_1|))$

How Large is A'_1?

(assumption: $|A_i| = n$ for all i)

- $|A'_{\ell-1}| = (\frac{1}{2} + 1)n$
- $|A'_{\ell-2}| = (\frac{1}{4} + \frac{1}{2} + 1)n$
- $|A'_{\ell-3}| = (\frac{1}{8} + \frac{1}{4} + \frac{1}{2} + 1)n$
- $|A'_1| \leq 2n$

$\Rightarrow$ search takes $O(\ell + \log n)$ time

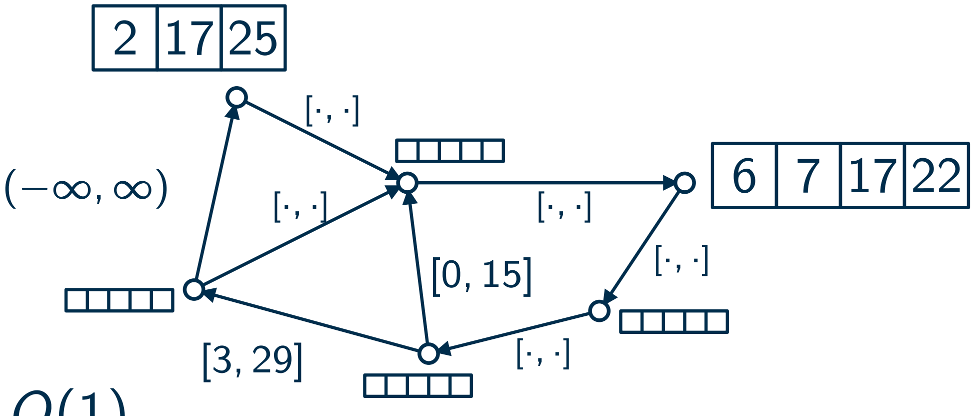
Memory Consumption

- only a constant factor overhead
- also true if not all arrays have the same size

Precomputation Time

- linear in the input

General Fractional Cascading



Now With A Directed Graph $G = (V, E)$

- sorted array A_v for every vertex v
- an interval I_e for every edge e
- for every number x and $u \in V$: $|\{uv \in E \mid x \in I_{uv}\}| \in O(1)$

How is this a generalization?

A Game Between Alice And Bob

- precomputes a data structure
- answers the question
- answers the question

- choose a number x and $u \in V$
- asks where x lies in A_u
- choose edge uv with $x \in I_{uv}$
- asks where x lies in A_v

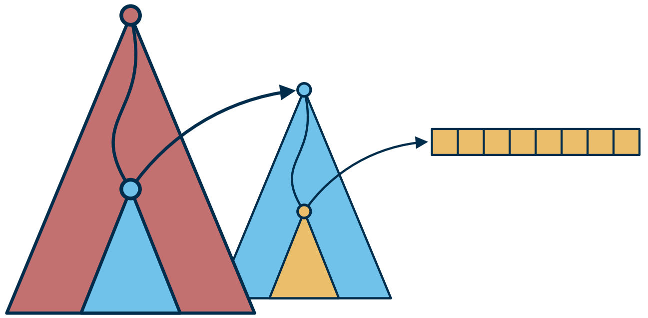
Similar Guarantee To The Path Setting (without proof)

(s = total array size)

- precomputation: $O(s)$ time and $O(s)$ space
- query: $O(\log s)$ for the first, then $O(1)$

Back To The Range Queries

query: $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$
direction: x y z



Query In 3D Range Tree (Simple Variant)

- walk down the x-tree $\rightarrow O(\log n)$
- walk down in $O(\log n)$ y-trees $\rightarrow O(\log n \log n)$
- search in $O(\log n \log n)$ z-arrays $\rightarrow O(\log n \log n \log n)$

Last Lecture: Do z-Search Earlier

- walk down the x-tree $\rightarrow O(\log n)$ time
- search in z-arrays in roots of $O(\log n)$ y-trees $\rightarrow O(\log n \log n)$
- walk down in $O(\log n)$ y-trees (and follow z-array pointers) $\rightarrow O(\log n \log n)$

Idea: Do The z-Search Even Earlier

- search z-array in root of x-tree $\rightarrow O(\log n)$
- walk down the x-tree $\rightarrow O(\log n)$ time
- walk down in $O(\log n)$ y-trees $\rightarrow O(\log n \log n)$

Observation

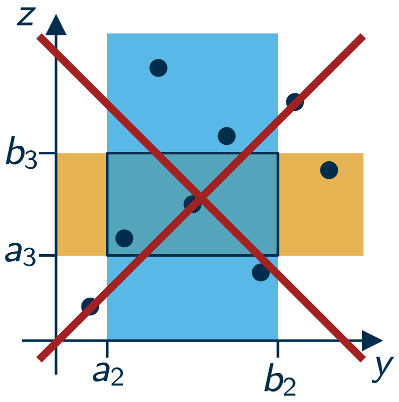
- getting rid of $\log n$ seems easy
- getting rid of $\log n$ seems hard
- goal: 2D DS with query time $O(\log n)$

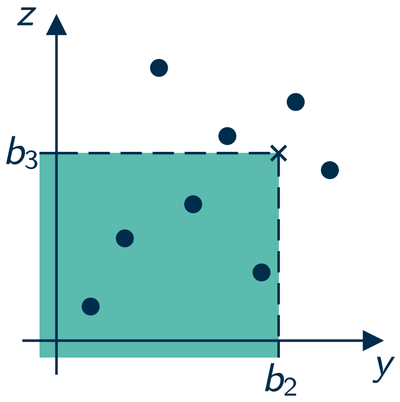
One-Sided 2D Range Queries

goal: 2D DS with query time $O(\log n)$

One-Sided Queries: Half The Sides, Half The Trouble

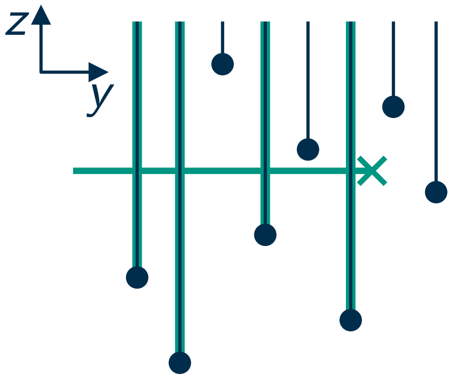
- goal: answer queries of the form $(-\infty, b_2] \times (-\infty, b_3]$ (instead of $[a_2, b_2] \times [a_3, b_3]$)





Alternative Perspective

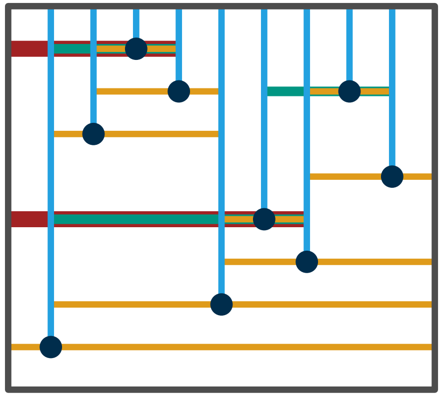
- shoot a ray from each point upwards
- ray from $\langle b_2, b_3 \rangle$ to the left
- intersecting rays yield desired points



Find All Intersecting Rays

- collect the intersecting rays from left to right
- we basically walk from cell to cell
- each cells knows its right neighbors sorted by $z \Rightarrow O(k \log n)$

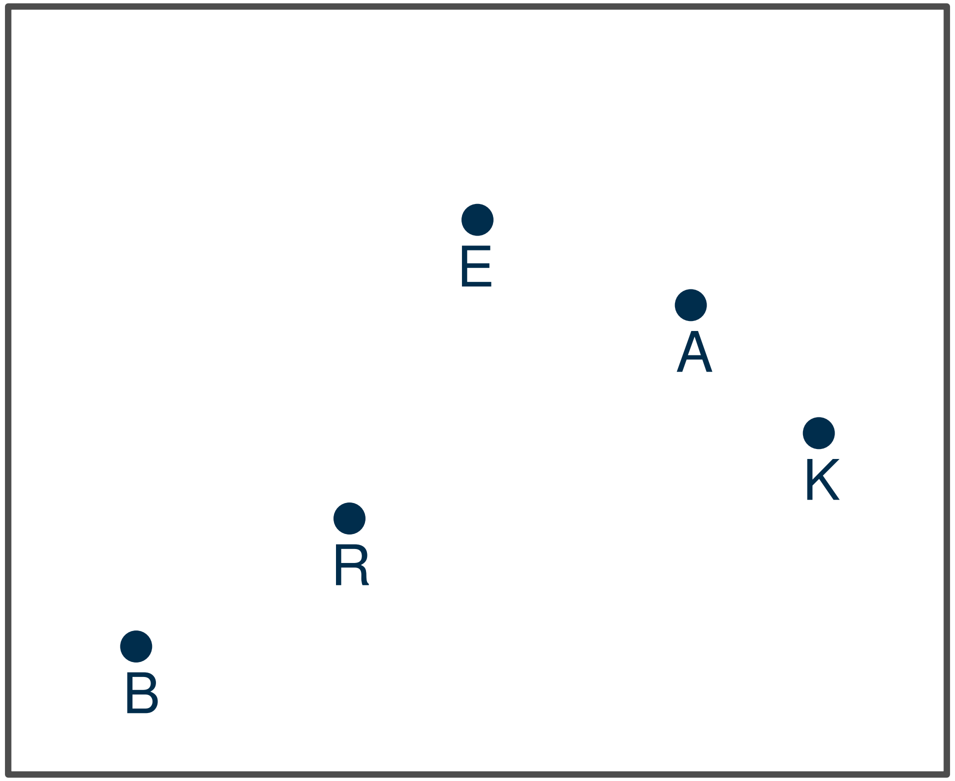
Can we do $\log n + k$?



Fractional Cascading!

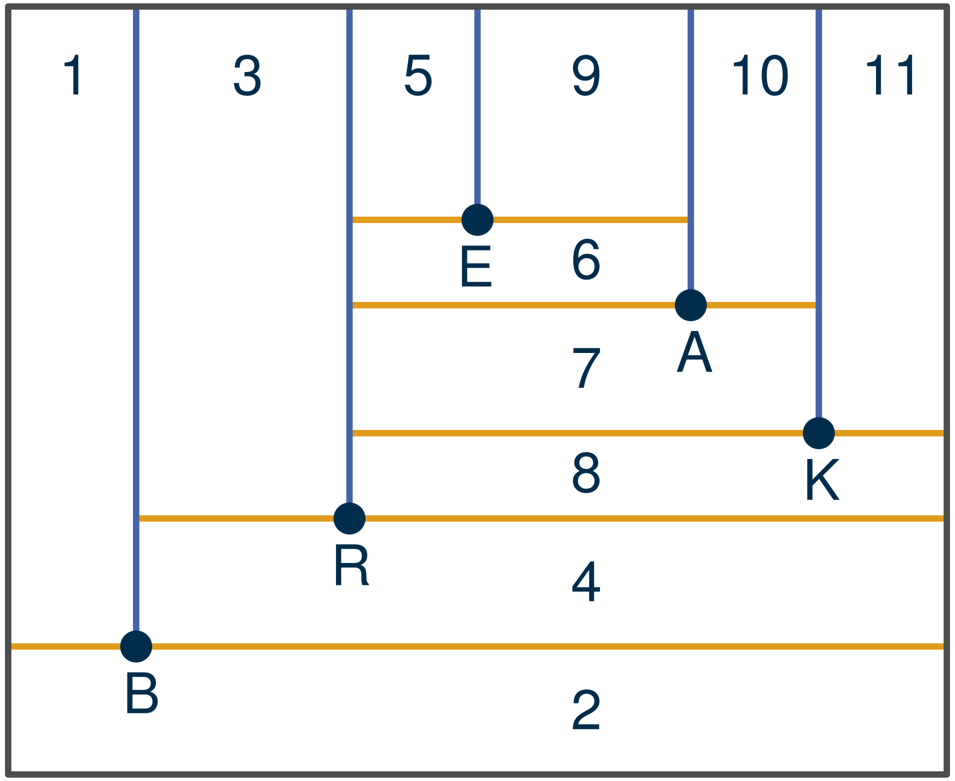
Count The Cells

How many cells do we get (with and without fractional cascading)?



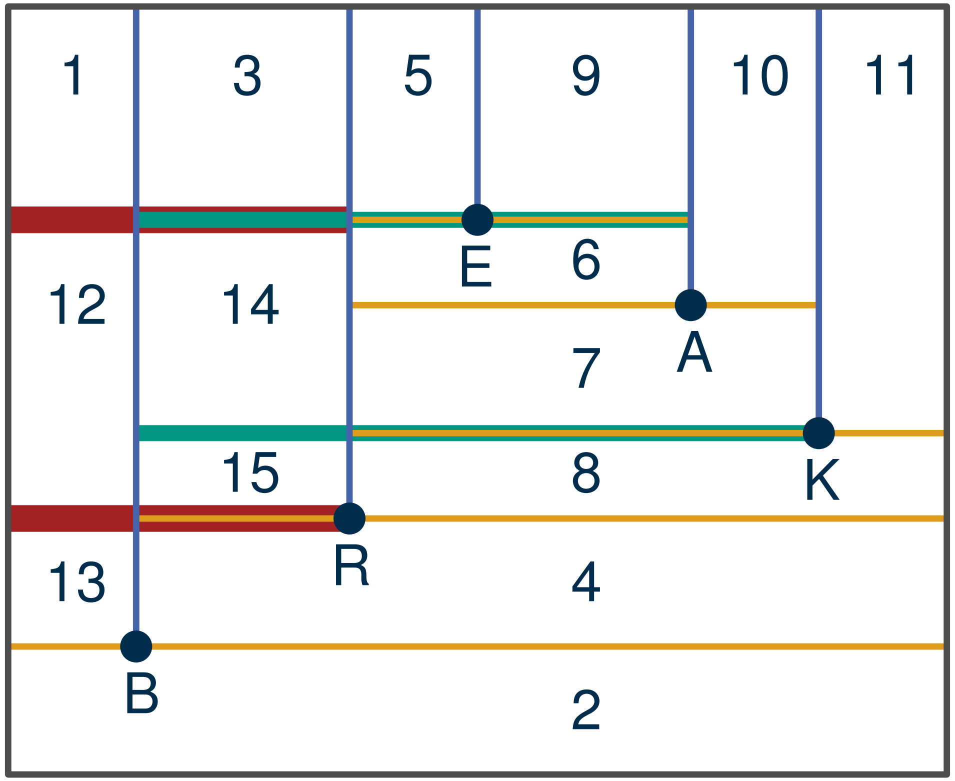
Count The Cells

How many cells do we get (with and without fractional cascading)?



Count The Cells

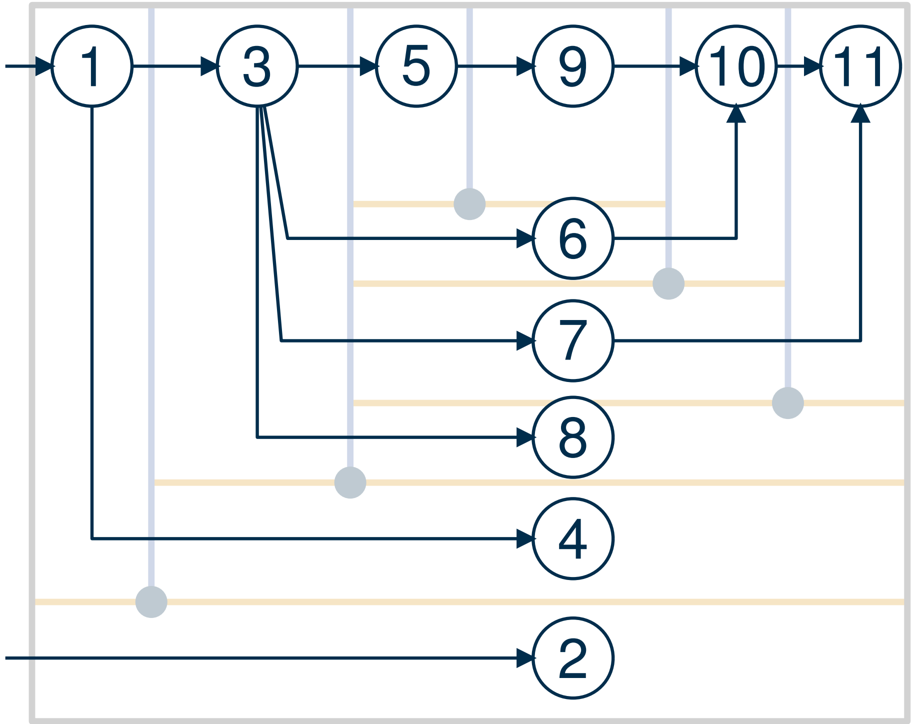
How many cells do we get (with and without fractional cascading)?

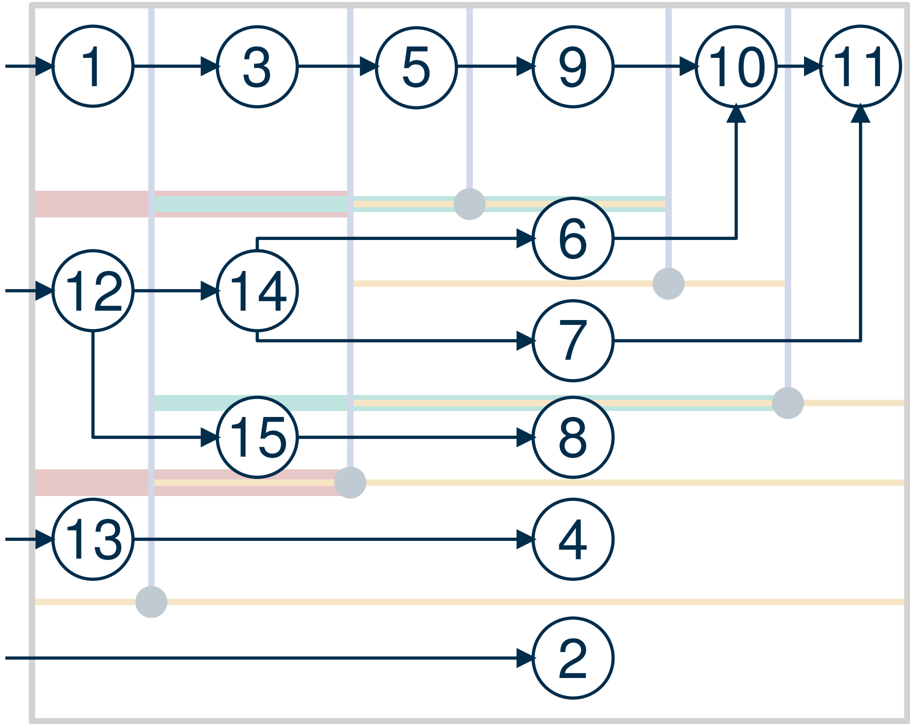


General Framework vs. Specific Situation

Useful Way Of Thinking

- mental shortcut: multiple searches for the same number $\rightarrow$ fractional cascading probably helps
- specific situation: problem-specific argument often easier than pressing it into the framework





One-Sided 3D Range Queries

Seen So Far: queries $(-\infty, b_2] \times (-\infty, b_3]$ can be answered in $O(\log n + k)$ **(DS1)**

one search in z direction

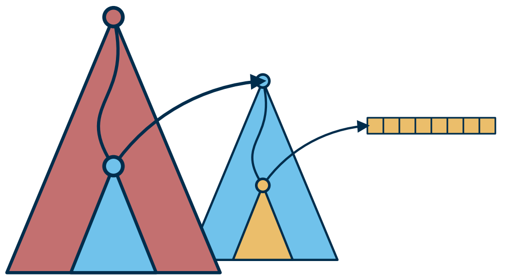
output size

New Goal: answer queries of the form $[a_1, b_1] \times (-\infty, b_2] \times (-\infty, b_3]$

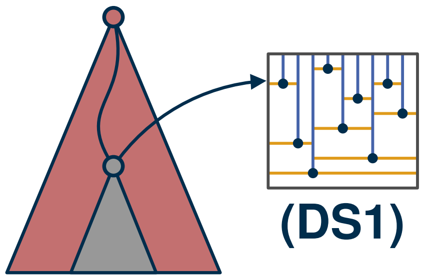
We Already Know How To Do This ...

- binary search tree for x-direction
- every node stores **(DS1)** for the corresponding points

Last Lecture



Now



Don't We Have To Search In $O(\log n)$ Many (DS1)?

- yes, but ... **Fractional Cascading!**
- search once in z-direction in the root of the x-tree
- follow pointers for the z-positions while walking down the x-tree
- save the first search in **(DS1)** $\Rightarrow$ total running time $O(\log n + k)$

One-Sided $\to$ Two-Sided

Lemma **(DS2)**
For n points in $\mathbb{R}^3$, we can answer queries of the form $[a_1, b_1] \times (-\infty, b_2] \times (-\infty, b_3]$ in $O(\log n + k)$ time after $O(n \log n)$ preprocessing with $O(n \log n)$ memory.

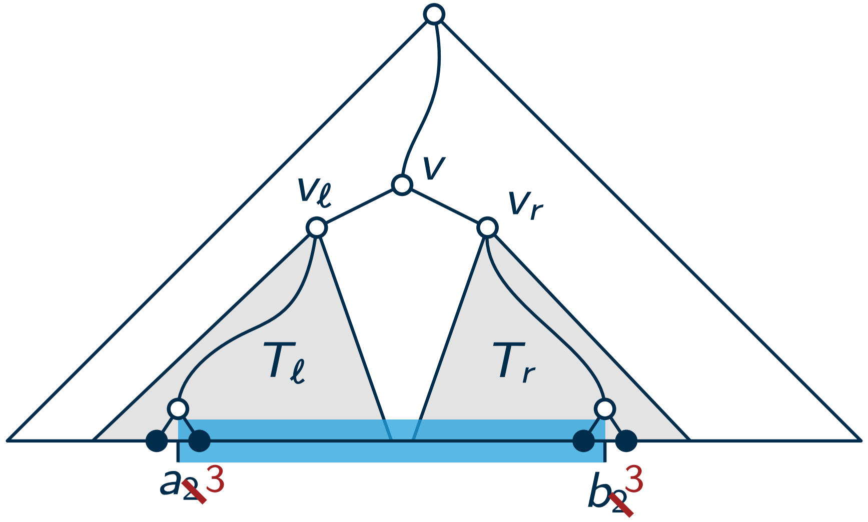
Plan
- use **(DS2)** as black box
- y-inverted variant $\to$ $[a_2, \infty)$ queries
- query $[a_2, \infty)$ and $(-\infty, b_2]$ to get $[a_2, b_2]$

Why can't we just use the intersection of two queries?

$$\begin{array}{ccccccc} [a_1, b_1] & \times & [a_2, b_2] & \times & [a_3, b_3] & = \\ [a_1, b_1] & \times & (-\infty, b_2] & \times & (-\infty, b_3] & \cap \\ [a_1, b_1] & \times & [a_2, \infty) & \times & [a_3, \infty) & \end{array}$$

Theorem **(DS4)**
For n point in $\mathbb{R}^3$, we can answer queries of the form $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$ in $O(\log n + k)$ time after $O(n \log^3 n)$ preprocessing with $O(n \log^3 n)$ memory.

Two-Sided Query In ~~y~~ z-Direction



Simplified Perspective: Ignore x And ~~z~~ y-Direction

- **(DS~~2~~3)** allows $[a_{\cancel{2}3}, \infty)$ and $(-\infty, b_{\cancel{2}3}]$ queries
- goal: build data structure, that allows $[a_{\cancel{2}3}, b_{\cancel{2}3}]$ queries

Binary Search Tree In ~~y~~ z-Direction

- search for $a_{\cancel{2}3}$ and $b_{\cancel{2}3}$ splits at v to v_ℓ and v_r
- queries in instances of **(DS~~2~~3)**: $[a_{\cancel{2}3}, \infty)$ on points in T_ℓ and $(-\infty, b_{\cancel{2}3}]$ on points in T_r
- running time: $O(\log n)$ for search in ~~y~~ z-tree plus $O(\log n + k)$ for two queries in **(DS~~2~~3)**
- memory: $O(\log n) \cdot$ (memory for **DS~~2~~3**)

~~Lemma~~ Theorem **(DS~~3~~4)**
For n point in $\mathbb{R}^3$, we can answer queries of the form $[a_1, b_1] \times [a_2, b_2] \times (\cancel{-\infty}\ a_3, b_3]$ in $O(\log n + k)$ time after $O(n \log^{\cancel{2}3} n)$ preprocessing with $O(n \log^{\cancel{2}3} n)$ memory.

The Big Picture

(DS4)

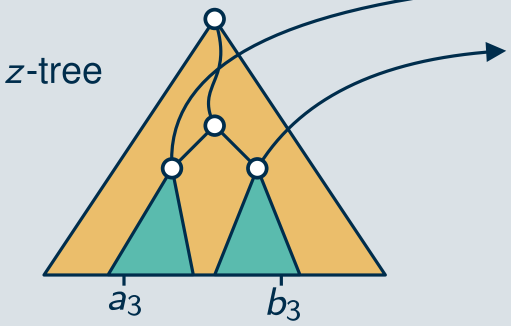
$[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$

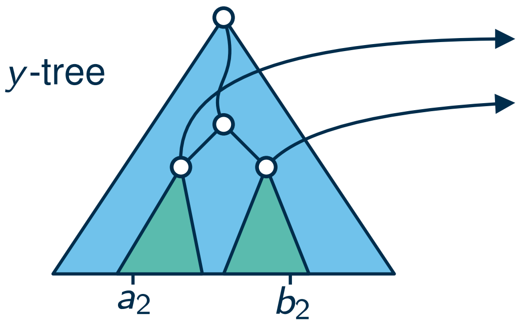
precomp: $O(n \log^3 n)$

memory: $O(n \log^3 n)$

query: $O(\log n + k)$

- search for a_3, b_3 in z-tree
- split → two **(DS3)** queries





(DS3)

$[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$

precomp: $O(n \log^2 n)$

memory: $O(n \log^2 n)$

query: $O(\log n + k)$

- search for a_2, b_2 in y-tree
- splits → two **(DS2)** queries

(DS2)

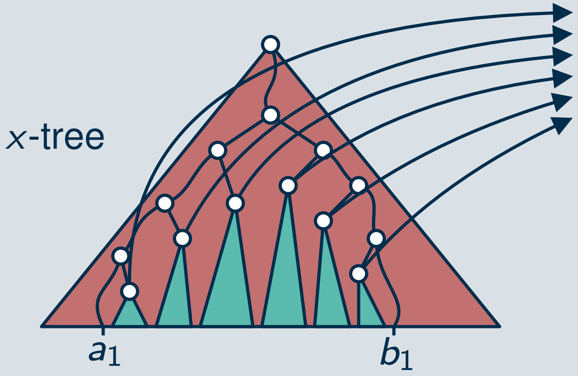
$[a_1, b_1] \times (-\infty, b_2] \times (-\infty, b_3]$

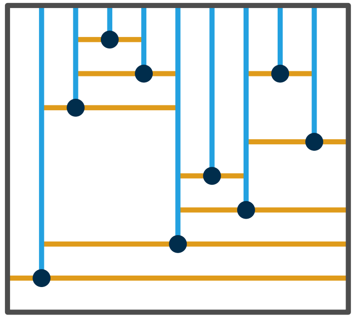
precomp: $O(n \log n)$

memory: $O(n \log n)$

query: $O(\log n + k)$

- search for z in the root
- walk down x-tree
- follow z-pointers
- each x-subtree: **(DS1)** query (with initial z-pos)





(DS1)

$(-\infty, b_2] \times (-\infty, b_3]$

query: $O(k)$

precomp: $O(n)$ (if points are sorted)

memory: $O(n)$

- initial z-position known
- walk in y-direction
- decide for cell by z-query
- output intersected rays

Wrap-Up

What Have We Learned Today?

- fractional cascading: search only once and then follow pointers
- clever geometric solution for simplified queries $(-\infty, b_2] \times (-\infty, b_3]$
- transformation from $(-\infty, b]$ to $[a, b]$

Theorem **(DS4)**
For n point in $\mathbb{R}^3$, we can answer queries of the form $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$ in $O(\log n + k)$ time after $O(n \log^3 n)$ preprocessing with $O(n \log^3 n)$ memory.

What Else Is There?

- many applications of fractional cascading
- dynamic range queries: inserting and deleting points
- $O(\log n \cdot (\log n / \log \log n)^{d-3} + k)$ queries with $O(n \cdot (\log n / \log \log n)^{d-3})$ memory
- even better results with some bit-hacking in the word RAM model