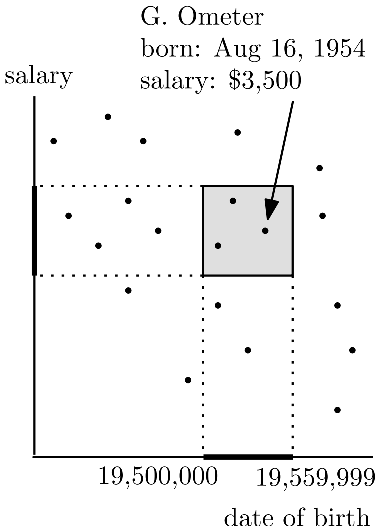
Range trees

Computational Geometry

Lecture 8: Range trees

Database queries

A database query may ask for all employees with age between a_1 and a_2, and salary between s_1 and s_2



Result

Theorem: A set of n points on the real line can be preprocessed in $O(n \log n)$ time into a data structure of $O(n)$ size so that any 1D range [counting] query can be answered in $O(\log n\ [+k])$ time

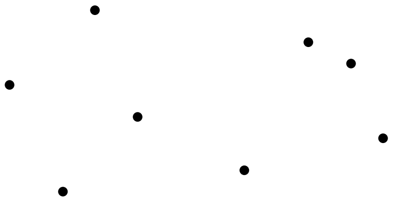
Result

Theorem: A set of n points in the plane can be preprocessed in $O(n \log n)$ time into a data structure of $O(n)$ size so that any 2D range query can be answered in $O(\sqrt{n}+k)$ time, where k is the number of answers reported

For range counting queries, we need $O(\sqrt{n})$ time

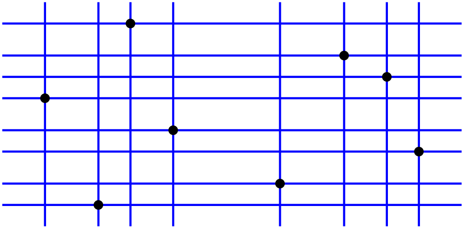
Faster queries

Can we achieve $O(\log n\ [+k])$ query time?

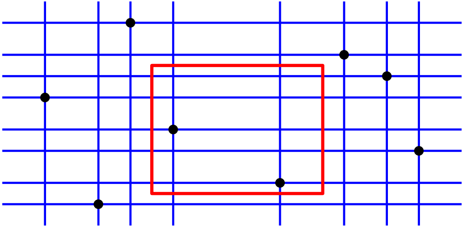


Faster queries

Can we achieve $O(\log n\ [+k])$ query time?



Faster queries

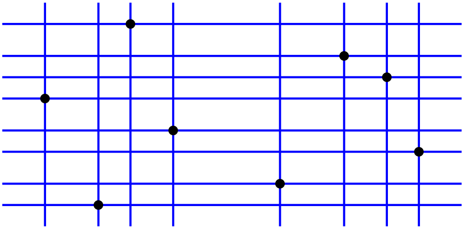


If the corners of the query rectangle fall in specific cells of the grid, the answer is fixed (even for lower left and upper right corner)

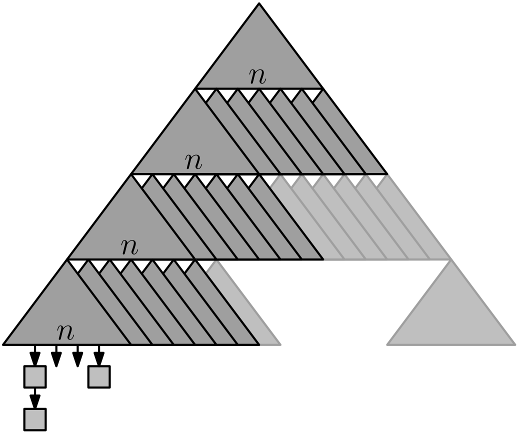
Faster queries

Build a tree so that the leaves correspond to the different possible query rectangle types (corners in same cells of grid), and with each leaf, store all answers (points) [or: the count]

Build a tree on the different x-coordinates (to search with left side of R), in each of the leaves, build a tree on the different x-coordinates (to search with the right side of R), in each of the leaves, ...



Faster queries



Faster queries

Question: What are the storage requirements of this structure, and what is the query time?

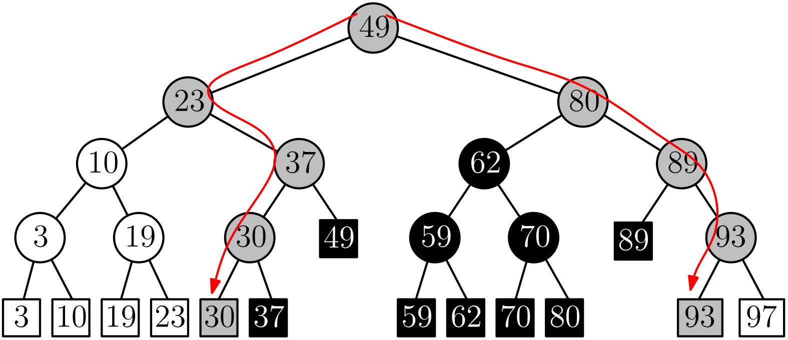
Faster queries

Recall the 1D range tree and range query:

- Two search paths (grey nodes)
- Subtrees in between have answers exclusively (black)

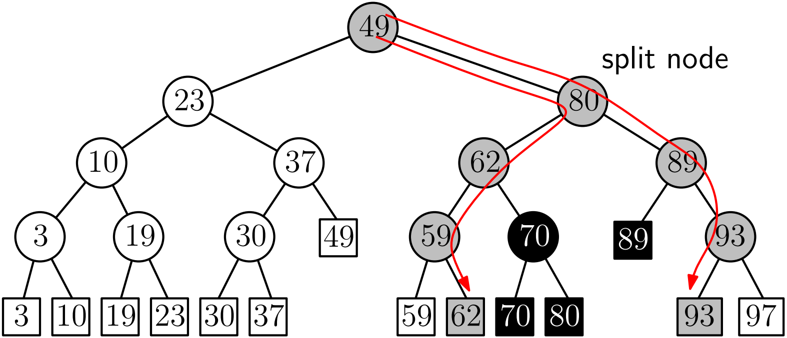
Example 1D range query

A 1-dimensional range query with $[25, 90]$



Example 1D range query

A 1-dimensional range query with $[61, 90]$



Examining 1D range queries

Observation: Ignoring the search path leaves, all answers are jointly represented by the highest nodes strictly between the two search paths

Question: How many highest nodes between the search paths can there be?

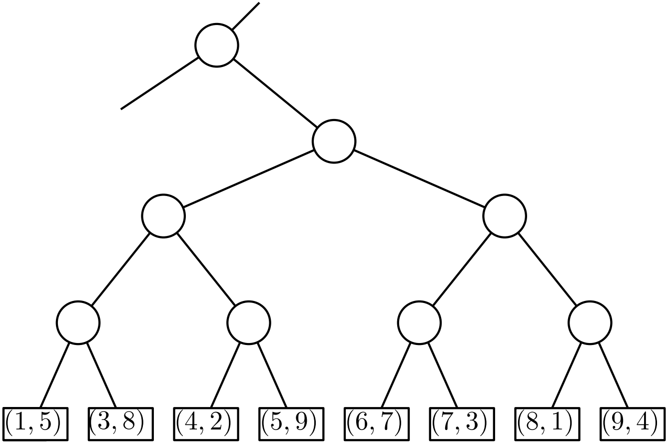
Examining 1D range queries

For any 1D range query, we can identify $O(\log n)$ nodes that together represent all answers to a 1D range query

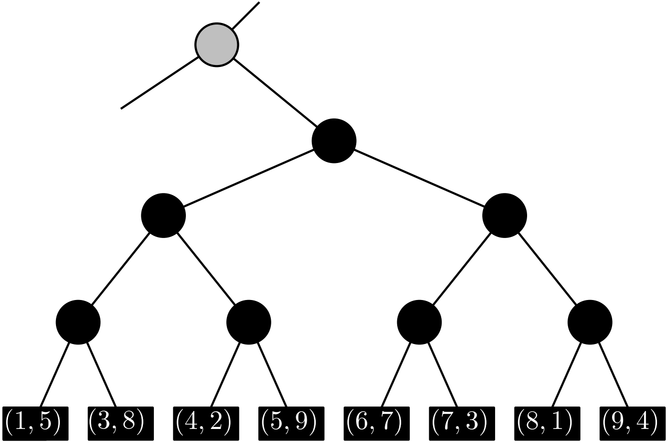
Toward 2D range queries

For any 2d range query, we can identify $O(\log n)$ nodes that together represent all points that have a correct first coordinate

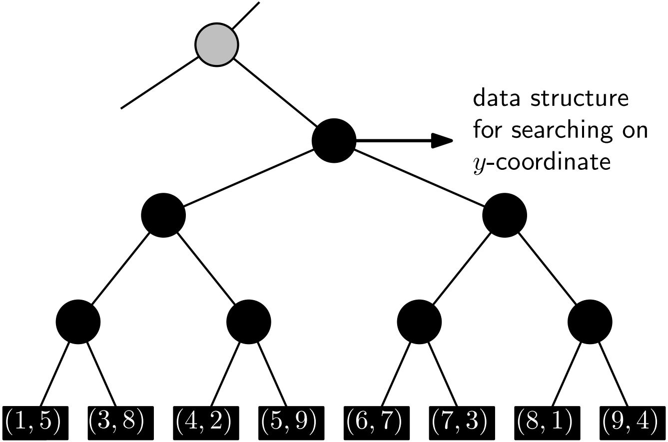
Toward 2D range queries



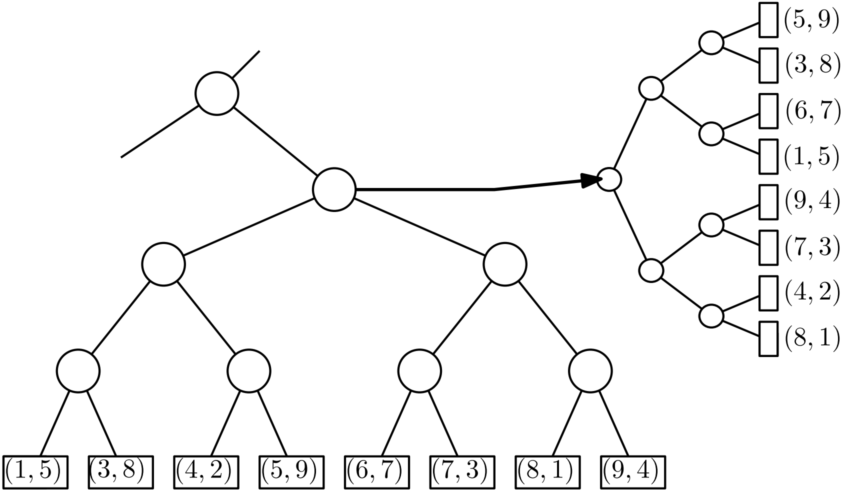
Toward 2D range queries



Toward 2D range queries

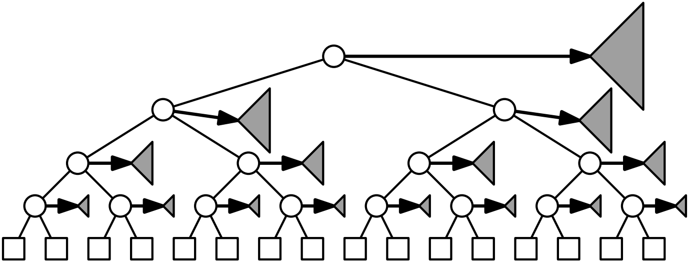


Toward 2D range queries



2D range trees

Every internal node stores a whole tree in an *associated structure*, on y-coordinate



Question: How much storage does this take?

Storage of2D range trees

To analyze storage, two arguments can be used:

- By level: On each level, any point is stored exactly once. So all associated trees on one level together have $O(n)$ size
- By point: For any point, it is stored in the associated structures of its search path. So it is stored in $O(\log n)$ of them

Construction algorithm

Algorithm $\text{BUILD2DRANGETREE}(P)$

1. Construct the associated structure: Build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates in P
2. **if** P contains only one point
3. **then** Create a leaf v storing this point, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v.
4. **else** Split P into P_{left} and P_{right}, the subsets $\leq$ and $>$ the median x-coordinate x_{mid}
5. $v_{\text{left}} \leftarrow \text{BUILD2DRANGETREE}(P_{\text{left}})$
6. $v_{\text{right}} \leftarrow \text{BUILD2DRANGETREE}(P_{\text{right}})$
7. Create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of v
8. **return** v

Efficiency of construction

The construction algorithm takes $O(n\log^2 n)$ time

$$T(1) = O(1)$$

$$T(n) = 2 \cdot T(n/2) + O(n\log n)$$

which solves to $O(n\log^2 n)$ time

Efficiency of construction

Suppose we pre-sort P on y-coordinate, and whenever we split P into P_{left} and P_{right}, we keep the y-order in both subsets

For a sorted set, the associated structure can be built in linear time

Efficiency of construction

The adapted construction algorithm takes $O(n \log n)$ time

$$T(1) = O(1)$$

$$T(n) = 2 \cdot T(n/2) + O(n)$$

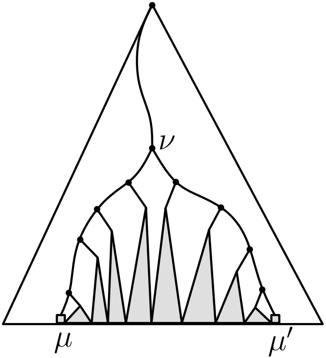
which solves to $O(n \log n)$ time

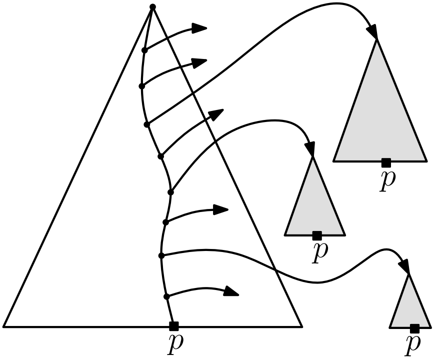
2D range queries

How are queries performed and why are they correct?

- Are we sure that each answer is found?
- Are we sure that the same point is found only once?

2D range queries





Query algorithm

Algorithm $\text{2DRANGEQUERY}(\mathcal{T}, [x : x'] \times [y : y'])$

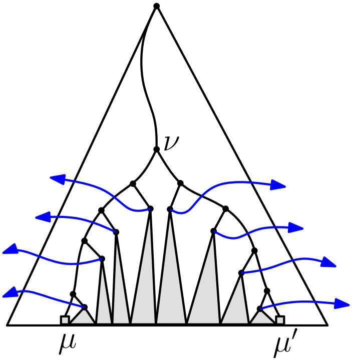
$\nu_{\text{split}} \leftarrow \text{FINDSPLITNODE}(\mathcal{T}, x, x')$
if ν_{split} is a leaf
  then report the point stored at ν_{split}, if an answer
  else $\nu \leftarrow lc(\nu_{\text{split}})$
    while ν is not a leaf
      do if $x \leq x_\nu$
        then $\text{1DRANGEQ}(\mathcal{T}_{\text{assoc}}(rc(\nu)), [y : y'])$
          $\nu \leftarrow lc(\nu)$
        else $\nu \leftarrow rc(\nu)$
    Check if the point stored at ν must be reported.
    Similarly, follow the path from $rc(\nu_{\text{split}})$ to x' ...

2D range query time

Question: How much time does a 2D range query take?

Subquestions: In how many associated structures do we search? How much time does each such search take?

2D range queries



2D range query efficiency

We search in $O(\log n)$ associated structures to perform a 1D range query; at most two per level of the main tree

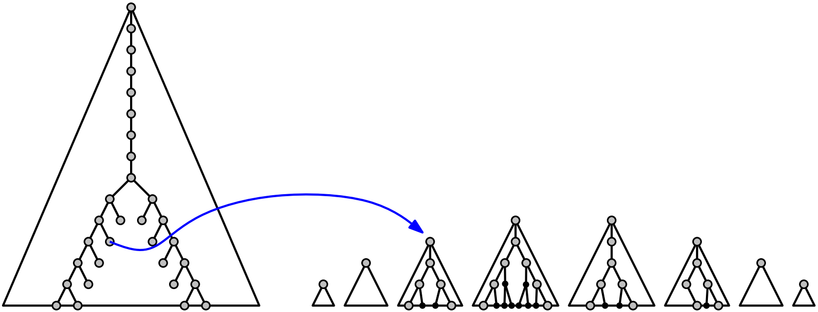
The query time is $O(\log n) \times O(\log m + k')$, or

$$\sum_{v} O(\log n_v + k_v)$$

where $\sum k_v = k$ the number of points reported

2D range query efficiency

Use the concept of grey and black nodes again:



2D range query efficiency

The number of grey nodes is $O(\log^2 n)$

The number of black nodes is $O(k)$ if k points are reported

The query time is $O(\log^2 n + k)$, where k is the size of the output

Result

Theorem: A set of n points in the plane can be preprocessed in $O(n \log n)$ time into a data structure of $O(n \log n)$ size so that any 2D range query can be answered in $O(\log^2 n + k)$ time, where k is the number of answers reported

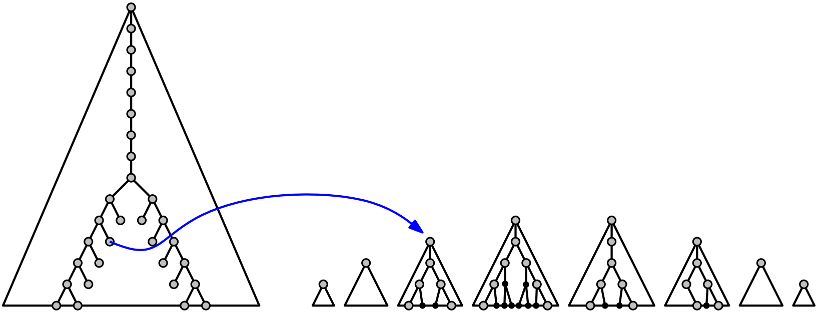
Recall that a kd-tree has $O(n)$ size and answers queries in $O(\sqrt{n} + k)$ time

Efficiency

n	$\log n$	$\log^2 n$	$\sqrt{n}$
16	4	16	4
64	6	36	8
256	8	64	16
1024	10	100	32
4096	12	144	64
16384	14	196	128
65536	16	256	256
1M	20	400	1K
16M	24	576	4K

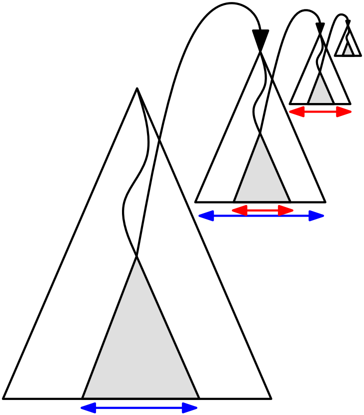
2D range query efficiency

Question: How about range *counting* queries?



Higher dimensional range trees

A d-dimensional range tree has a main tree which is a one-dimensional balanced binary search tree on the first coordinate, where every node has a pointer to an associated structure that is a $(d-1)$-dimensional range tree on the other coordinates



Storage

$$S_1(n) = O(n) \quad \text{for all } n$$

$$S_d(1) = O(1) \quad \text{for all } d$$

$$S_d(n) \leq 2 \cdot S_d(n/2) + S_{d-1}(n) \quad \text{for } d \geq 2$$

This solves to $S_d(n) = O(n \log^d n)$

Query time

The number of grey nodes $G_d(n)$ satisfies:

$$G_1(n) = O(\log n) \quad \text{for all } n$$

$$G_d(1) = O(1) \quad \text{for all } d$$

$$G_d(n) \leq 2 \cdot \log n + 2 \cdot \log n \cdot G_{d-1}(n) \quad \text{for } d \geq 2$$

This solves to $G_d(n) = O(\log^d n)$

Result

Theorem: A set of n points in the plane can be preprocessed in $O(n \log^{d-1} n)$ time into a data structure of $O(n \log^{d-1} n)$ size so that any d-dimensional range query can be answered in $O(\log^d n + k)$ time, where k is the number of answers reported

Recall that a kd-tree has $O(n)$ size and answers queries in $O(n^{1-1/d} + k)$ time

Comparison for $d = 4$

n	$\log n$	$\log^4 n$	$n^{3/4}$
1024	10	10,000	181
65,536	16	65,536	4096
1M	20	160,000	32,768
1G	30	810,000	5,931,641
1T	40	2,560,000	1G

Improving the query time

We can improve the query time of a 2D range tree from $O(\log^2 n)$ to $O(\log n)$ by a technique called fractional cascading

This automatically lowers the query time in d dimensions to $O(\log^{d-1} n)$ time

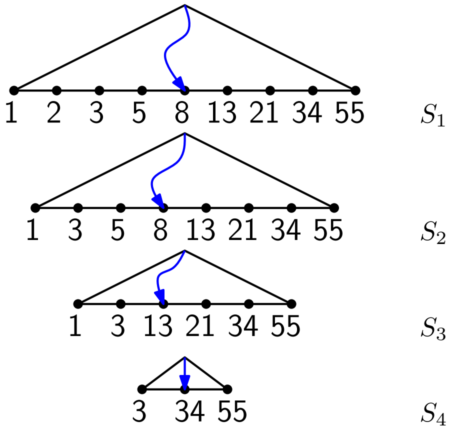
Improving the query time

The idea illustrated best by a *different* query problem:

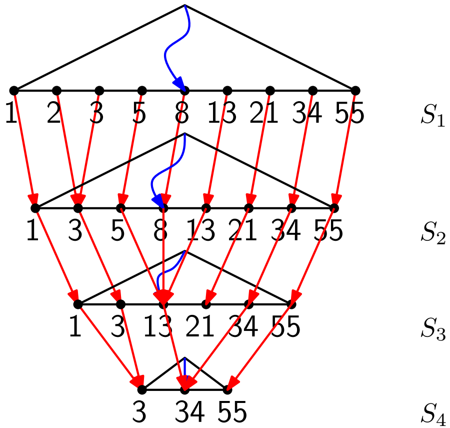
Suppose that we have a collection of sets $S_1, \ldots, S_m$, where $|S_1| = n$ and where $S_{i+1} \subseteq S_i$

We want a data structure that can report for a query number x, the smallest value $\geq x$ in all sets $S_1, \ldots, S_m$

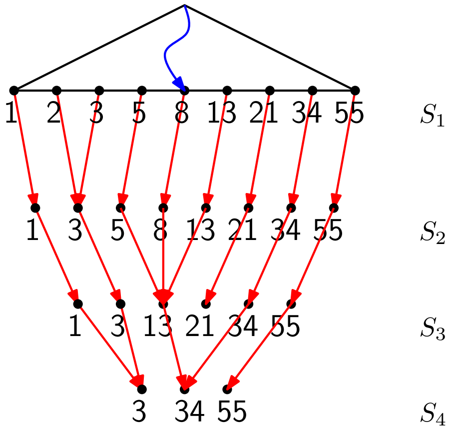
Improving the query time



Improving the query time



Improving the query time



Improving the query time

Suppose that we have a collection of sets $S_1, \ldots, S_m$, where $|S_1| = n$ and where $S_{i+1} \subseteq S_i$

We want a data structure that can report for a query number x, the smallest value $\geq x$ in all sets $S_1, \ldots, S_m$

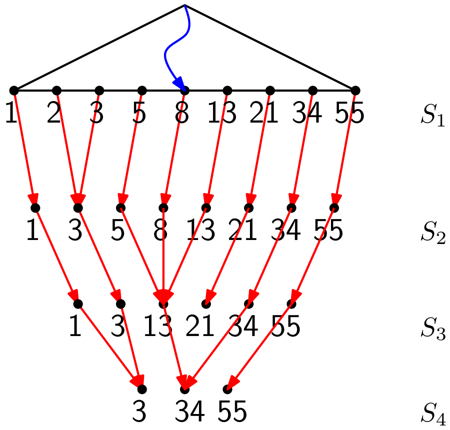
This query problem can be solved in $O(\log n + m)$ time instead of $O(m \cdot \log n)$ time

Improving the query time

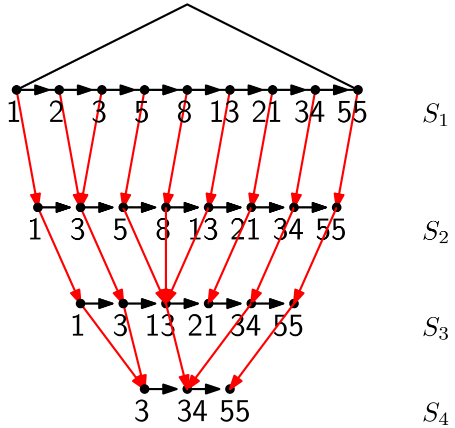
Can we do something similar for m 1-dimensional range queries on m sets $S_1, \ldots, S_m$?

We hope to get a query time of $O(\log n + m + k)$ with k the total number of points reported

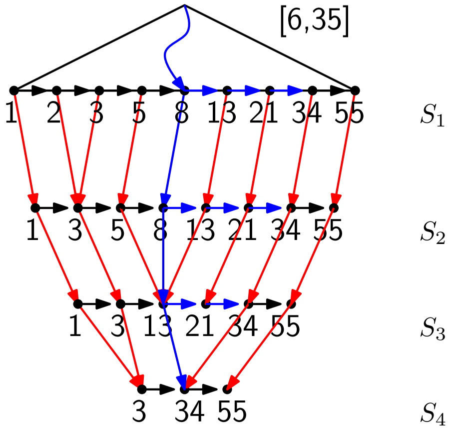
Improving the query time



Improving the query time



Improving the query time



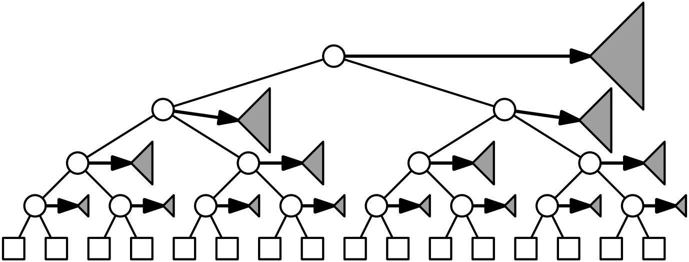
Fractional cascading

Now we do "the same" on the associated structures of a 2-dimensional range tree

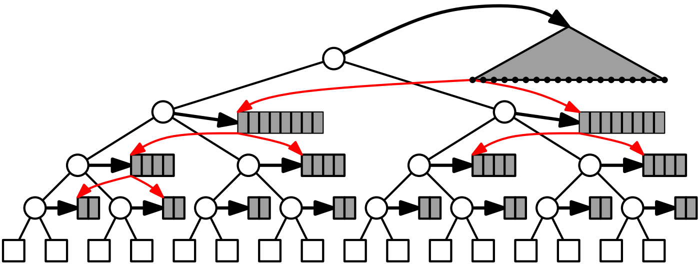
Note that in every associated structure, we search with the same values y and y'

- Replace all associated structure except for the root by a linked list
- For every list element (and leaf of the associated structure of the root), store **two** pointers to the appropriate list elements in the lists of the left child and of the right child

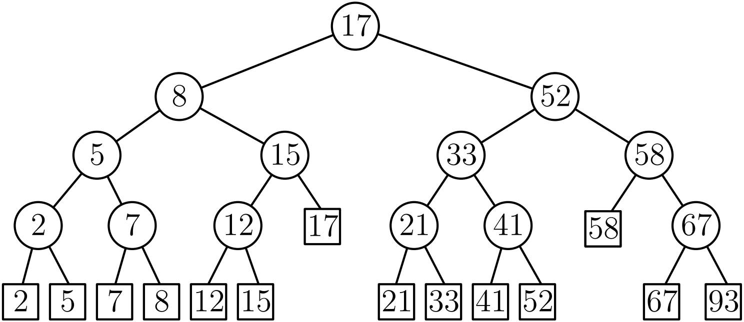
Fractional cascading



Fractional cascading



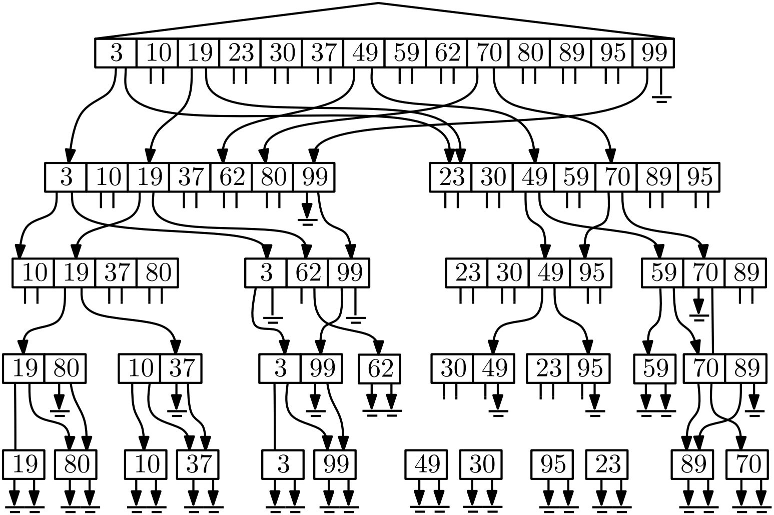
Fractional cascading



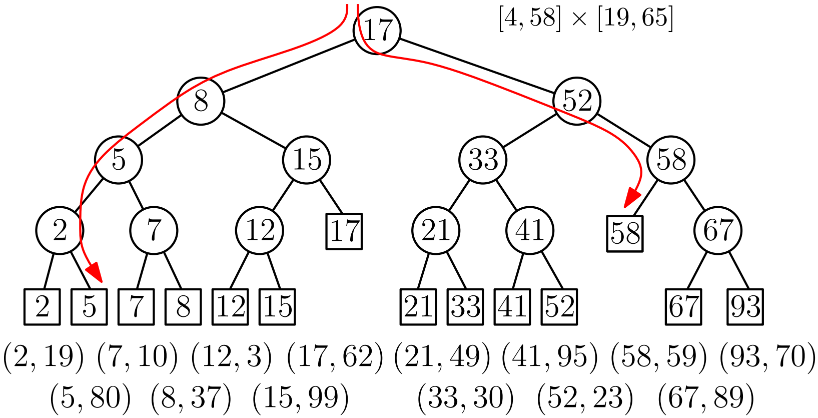
(2, 19) (7, 10) (12, 3) (17, 62) (21, 49) (41, 95) (58, 59) (93, 70)

(5, 80) (8, 37) (15, 99) (33, 30) (52, 23) (67, 89)

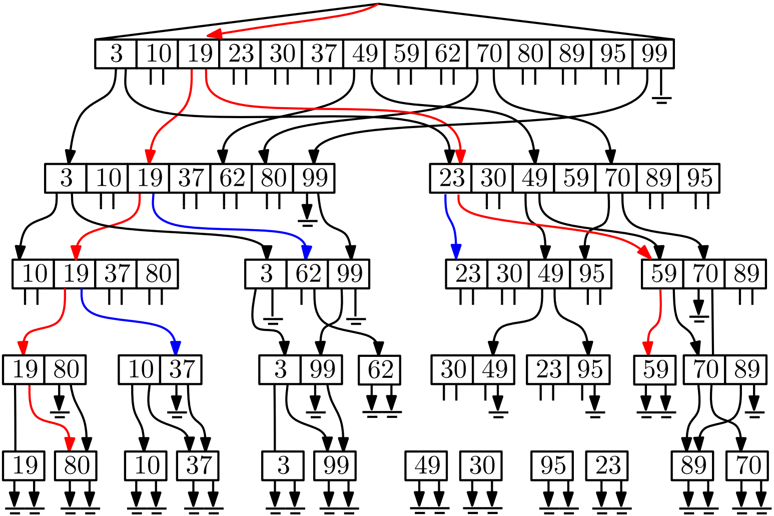
Fractional cascading



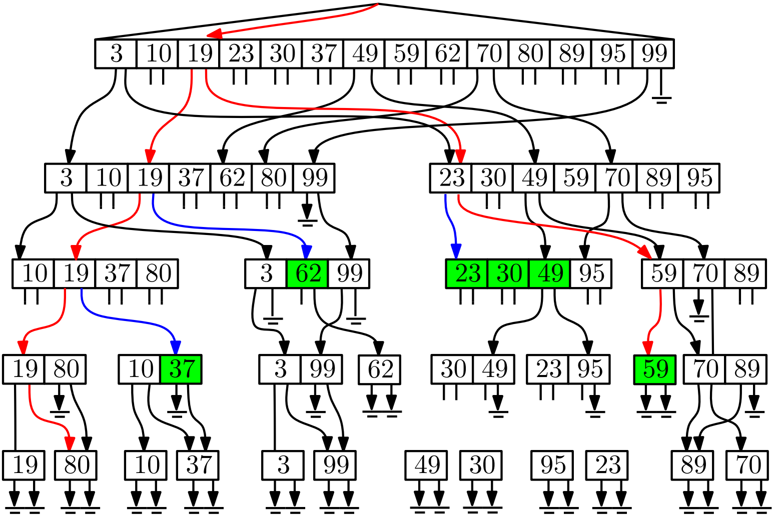
Fractional cascading



Fractional cascading



Fractional cascading



Fractional cascading

Instead of doing a 1D range query on the associated structure of some node v, we find the leaf where the search to y would end in $O(1)$ time via the direct pointer in the associated structure in the parent of v

The number of grey nodes reduces to $O(\log n)$

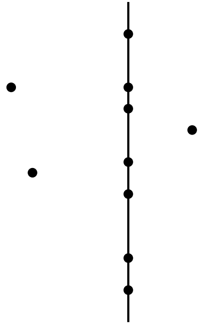
Result

Theorem: A set of n points in the plane can be preprocessed in $O(n\log^{d-1} n)$ time into a data structure of $O(n\log^{d-1} n)$ size so that any d-dimensional range query can be answered in $O(\log^{d-1} n+k)$ time, where k is the number of answers reported

Recall that a kd-tree has $O(n)$ size and answers queries in $O(n^{1-1/d}+k)$ time

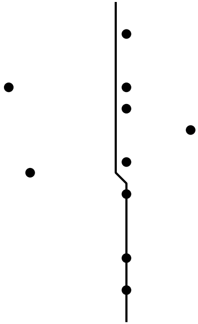
Degenerate cases

Both for kd-trees and for range trees we have to take care of multiple points with the same x- or y-coordinate



Degenerate cases

Both for kd-trees and for range trees we have to take care of multiple points with the same x- or y-coordinate



Degenerate cases

Treat a point $p = (p_x, p_y)$ with two reals as coordinates as a point with two composite numbers as coordinates

A composite number is a pair of reals, denoted $(a|b)$

We let $(a|b) < (c|d)$ iff $a < c$ or ($a = c$ and $b < d$); this defines a total order on composite numbers

Degenerate cases

The point $p = (p_x, p_y)$ becomes $(\,(p_x|p_y)\,,\,(p_y|p_x)\,)$. Then no two points have the same first or second coordinate

The median x-coordinate or y-coordinate is a composite number

The query range $[x : x'] \times [y : y']$ becomes

$$[(x|-\infty) : (x'|+\infty)] \times [(y|-\infty) : (y'|+\infty)]$$

We have $(p_x, p_y) \in [x : x'] \times [y : y']$ iff
$(\,(p_x|p_y)\,,\,(p_y|p_x)\,) \in [(x|-\infty) : (x'|+\infty)] \times [(y|-\infty) : (y'|+\infty)]$