Geometric Algorithms

Kd-trees and range trees

Databases

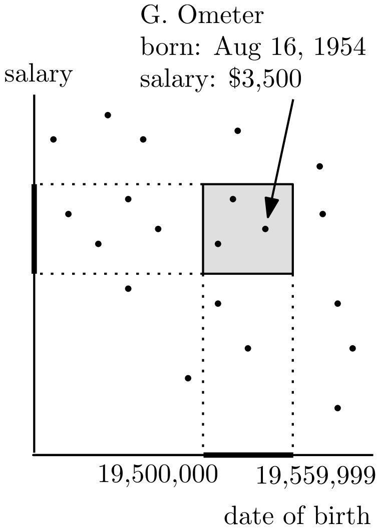
Databases store records or objects

Personnel database: Each employee has a name, id code, date of birth, function, salary, start date of employment, ...

Fields are textual or numerical

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



Database queries

When we see numerical fields of objects as coordinates, a database stores a point set in higher dimensions

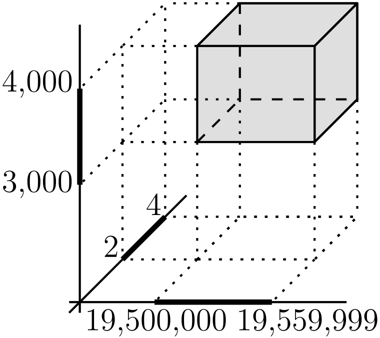
Exact match query: Asks for the objects whose coordinates match query coordinates exactly

Partial match query: Same but not all coordinates are specified

Range query: Asks for the objects whose coordinates lie in a specified query range (interval)

Database queries

Example of a 3-dimensional (orthogonal) range query: children in $[2, 4]$, salary in $[3000, 4000]$, date of birth in $[19{,}500{,}000 \, , \, 19{,}559{,}999]$



Data structures

Idea of data structures

- Representation of structure, for convenience (like DCEL)
- Preprocessing of data, to be able to solve future questions really fast (sub-linear time)

A (search) data structure has a storage requirement, a query time, and a construction time (and an update time)

1D range query problem

1D range query problem: Preprocess a set of n points on the real line such that the ones inside a 1D query range (interval) can be answered fast

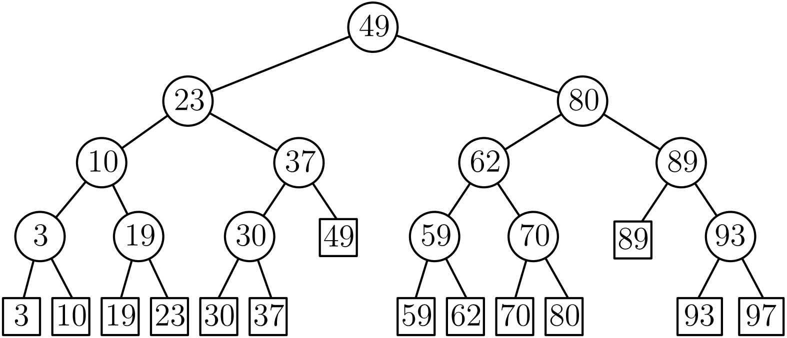
The points $p_1, \ldots, p_n$ are known beforehand, the query $[x, x']$ only later

A solution to a query problem is a data structure, a query algorithm, and a construction algorithm

Question: What are the most important factors for the *efficiency* of a solution?

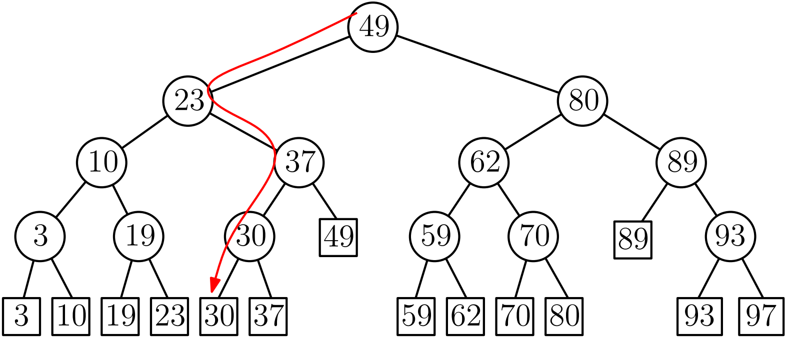
Balanced binary search trees

A balanced binary search tree with the points in the leaves



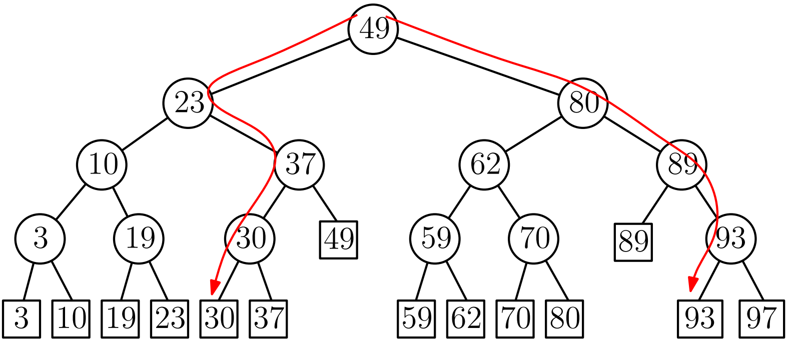
Balanced binary search trees

The search path for 25



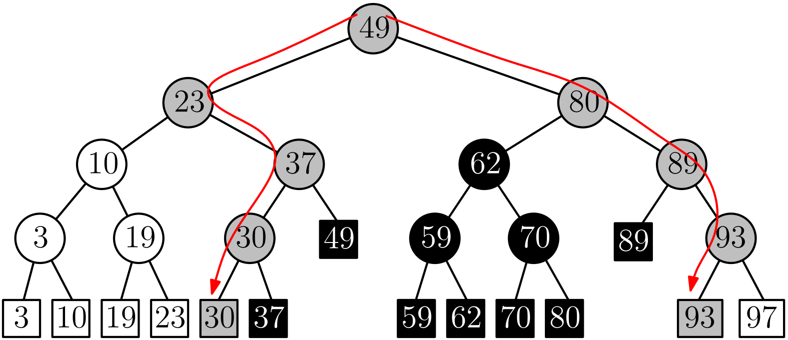
Balanced binary search trees

The search paths for 25 and for 90



Example 1D range query

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



Node types for a query

Three types of nodes *for a given query*:

- **White nodes:** never visited by the query
- **Grey nodes:** visited by the query, unclear if they lead to output
- **Black nodes:** visited by the query, whole subtree is output

Question: What query time do we hope for?

Node types for a query

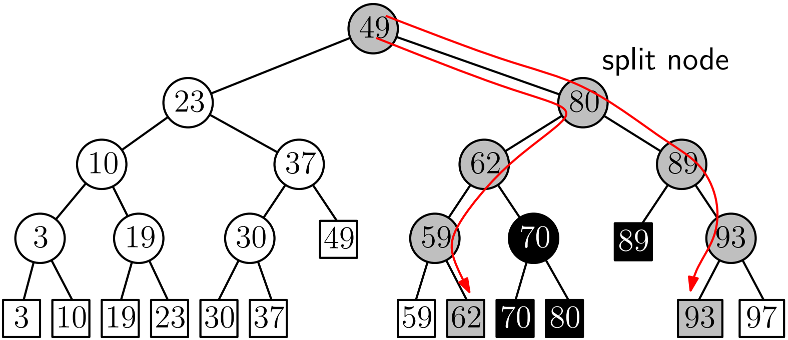
The query algorithm comes down to what we do at each type of node

Grey nodes: use query range to decide how to proceed: to not visit a subtree (pruning), to report a complete subtree, or just continue

Black nodes: traverse and enumerate all points in the leaves

Example 1D range query

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



1D range query algorithm

Algorithm $\textsc{1DRangeQuery}(\mathcal{T}, [x : x'])$

$\nu_{\text{split}} \leftarrow \textsc{FindSplitNode}(\mathcal{T}, x, x')$
if ν_{split} is a leaf
  then Check if the point in ν_{split} must be reported.
  else $\nu \leftarrow lc(\nu_{\text{split}})$
    while ν is not a leaf
      do if $x \leq x_\nu$
        then $\textsc{ReportSubtree}(rc(\nu))$
          $\nu \leftarrow lc(\nu)$
        else $\nu \leftarrow rc(\nu)$
    Check if the point stored in ν must be reported.
    Similarly, follow the path to x', and ...

Query time analysis

The efficiency analysis is based on counting the numbers of nodes visited for each type

- **White nodes:** never visited by the query; no time spent
- **Grey nodes:** visited by the query, unclear if they lead to output; time determines dependency on n
- **Black nodes:** visited by the query, whole subtree is output; time determines dependency on k, the output size

Query time analysis

Grey nodes: they occur on only two paths in the tree, and since the tree is balanced, its depth is $O(\log n)$

Black nodes: a (sub)tree with m leaves has $m-1$ internal nodes; traversal visits $O(m)$ nodes and finds m points for the output

The time spent at each node is $O(1) \;\Rightarrow\; O(\log n + k)$ query time

Storage requirement and preprocessing

A (balanced) binary search tree storing n points uses $O(n)$ storage

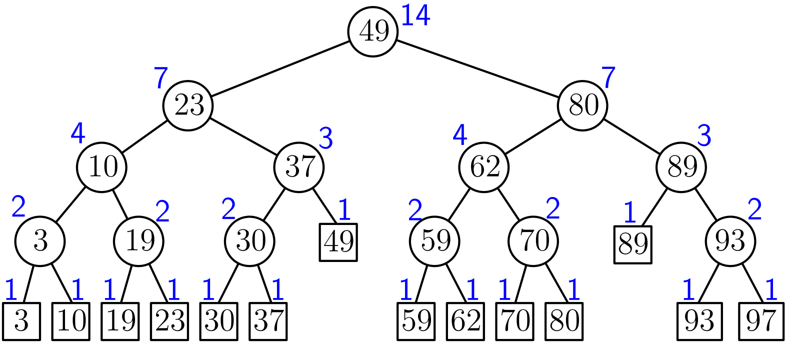
A balanced binary search tree storing n points can be built in $O(n)$ time after sorting

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 query can be answered in $O(\log n + k)$ time, where k is the number of answers reported

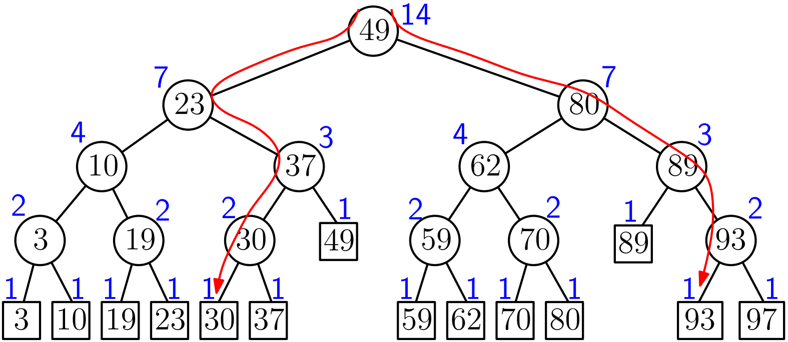
Example 1D range counting query

A 1-dimensional range tree for **range counting queries**



Example 1D range counting query

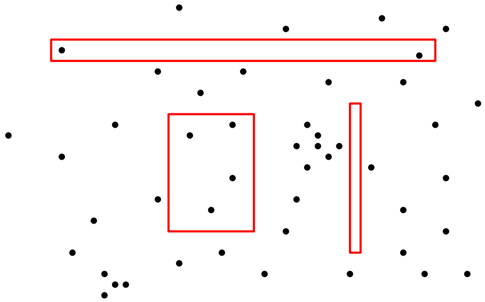
A 1-dimensional range counting query with $[25, 90]$



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 $O(\log n)$ time

Range queries in 2D



Range queries in 2D

Question: Why can't we simply use a balanced binary tree in x-coordinate?

Or, use one tree on x-coordinate and one on y-coordinate, and query the one where we think querying is more efficient?

Kd-trees

Kd-trees, the idea: Split the point set alternatingly by x-coordinate and by y-coordinate

split by x-coordinate: split by a vertical line that has half the points left and half right

split by y-coordinate: split by a horizontal line that has half the points below and half above

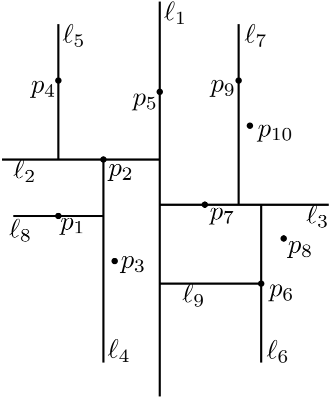
Kd-trees

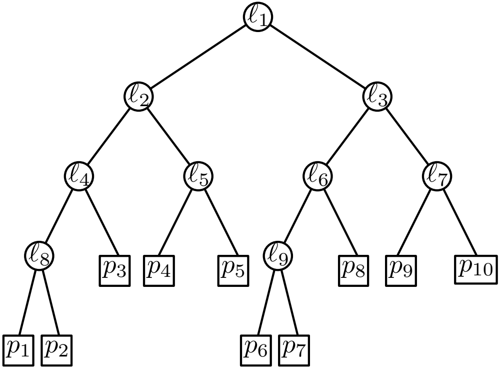
Kd-trees, the idea: Split the point set alternatingly by x-coordinate and by y-coordinate

split by x-coordinate: split by a vertical line that has half the points left or on, and half right

split by y-coordinate: split by a horizontal line that has half the points below or on, and half above

Kd-trees


ℓ_1
ℓ_5
ℓ_7
p_4
p_5
p_9
p_{10}
ℓ_2
p_2
p_7
ℓ_3
ℓ_8
p_1
p_8
p_3
ℓ_9
p_6
ℓ_4
ℓ_6


ℓ_1
ℓ_2
ℓ_3
ℓ_4
ℓ_5
ℓ_6
ℓ_7
ℓ_8
p_3
p_4
p_5
ℓ_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7

Kd-tree construction

Algorithm $\textsc{BuildKdTree}(P, \mathit{depth})$

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** depth is even
4. **then** Split P with a vertical line ℓ through the median x-coordinate into P_1 (left of or on ℓ) and P_2 (right of ℓ)
5. **else** Split P with a horizontal line ℓ through the median y-coordinate into P_1 (below or on ℓ) and P_2 (above ℓ)
6. $v_{\text{left}} \leftarrow \textsc{BuildKdTree}(P_1, \mathit{depth}+1)$
7. $v_{\text{right}} \leftarrow \textsc{BuildKdTree}(P_2, \mathit{depth}+1)$
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

Kd-tree construction

The median of a set of n values can be computed in $O(n)$ time (randomized: easy; worst case: much harder)

Let $T(n)$ be the time needed to build a kd-tree on n points

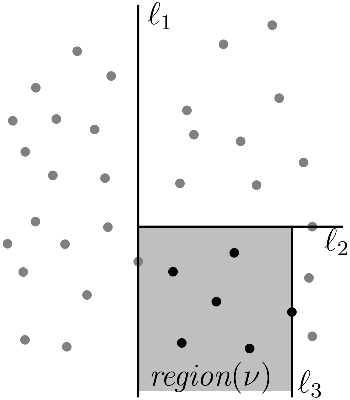
$$T(1) = O(1)$$

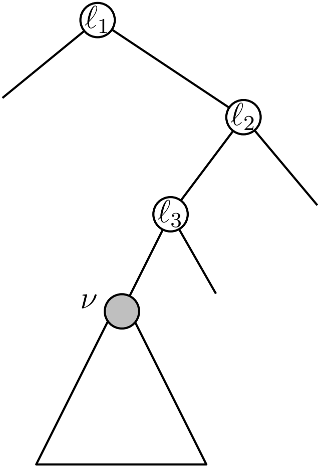
$$T(n) = 2 \cdot T(n/2) + O(n)$$

A kd-tree can be built in $O(n \log n)$ time

Question: What is the storage requirement?

Kd-tree regions of nodes





Kd-tree regions of nodes

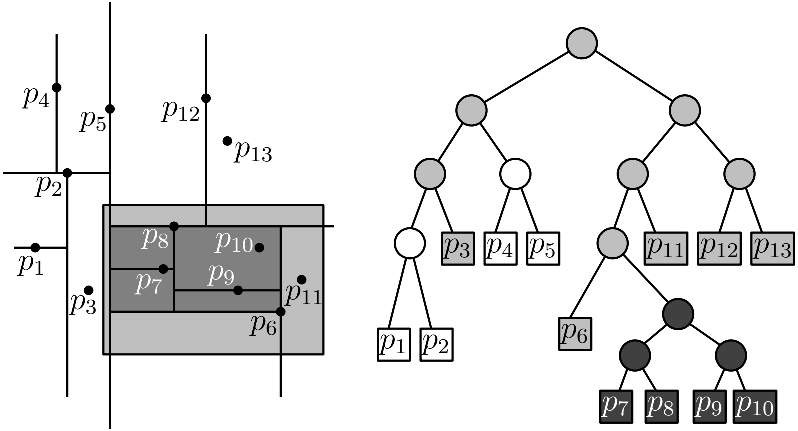
How do we know $region(v)$ when we are at a node v?

Option 1: store it explicitly with every node

Option 2: compute it on-the-fly, when going from the root to v

Question: What are reasons to choose one or the other option?

Kd-tree querying



Kd-tree querying

Algorithm $\textsc{SearchKdTree}(v, R)$
Input. The root of (a subtree of) a kd-tree, and a range R
Output. All points at leaves below v that lie in the range.

if v is a leaf
    then Report the point stored at v if it lies in R
    else **if** $region(lc(v))$ is fully contained in R
        then $\textsc{ReportSubtree}(lc(v))$
        else **if** $region(lc(v))$ intersects R
            then $\textsc{SearchKdTree}(lc(v), R)$
      if $region(rc(v))$ is fully contained in R
        then $\textsc{ReportSubtree}(rc(v))$
        else **if** $region(rc(v))$ intersects R
            then $\textsc{SearchKdTree}(rc(v), R)$

Kd-tree querying

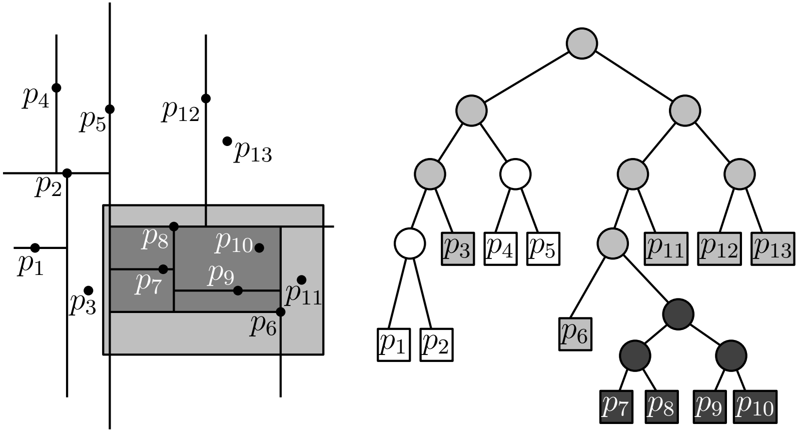
Question: How about a range *counting* query?
How should the code be adapted?

Kd-tree query time analysis

To analyze the query time of kd-trees, we use the concept of white, grey, and black nodes

- **White nodes:** never visited by the query; no time spent
- **Grey nodes:** visited by the query, unclear if they lead to output; time determines dependency on n
- **Black nodes:** visited by the query, whole subtree is output; time determines dependency on k, the output size

Kd-tree query time analysis

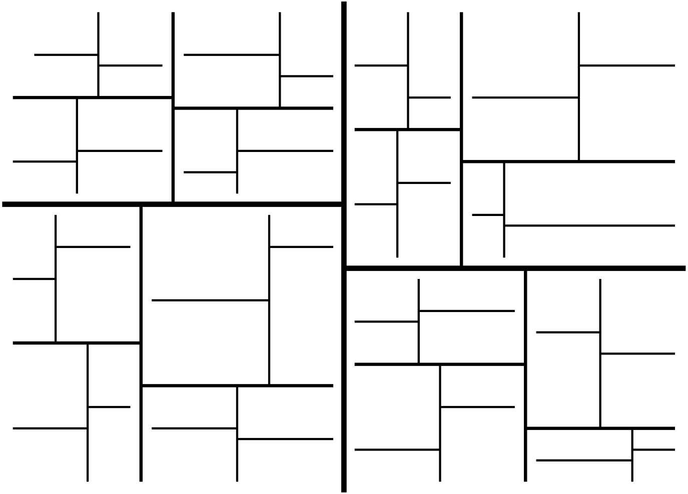


Kd-tree query time analysis

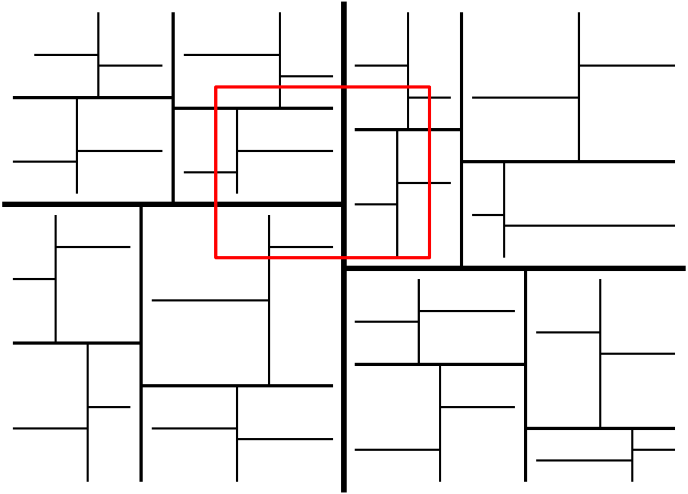
White, grey, and black nodes with respect to $region(v)$:

- **White node v:** R does not intersect $region(v)$
- **Grey node v:** R intersects $region(v)$, but $region(v) \not\subseteq R$
- **Black node v:** $region(v) \subseteq R$

Kd-tree query time analysis

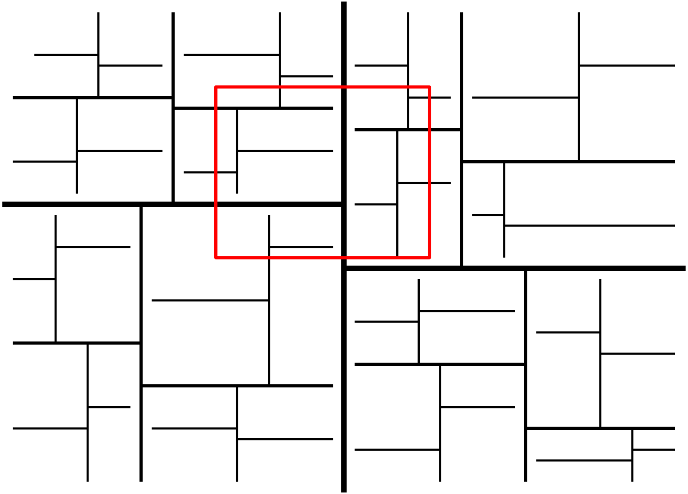


Kd-tree query time analysis



Question: How many grey and how many black *leaves*?

Kd-tree query time analysis



Question: How many grey and how many black *nodes*?

Kd-tree query time analysis

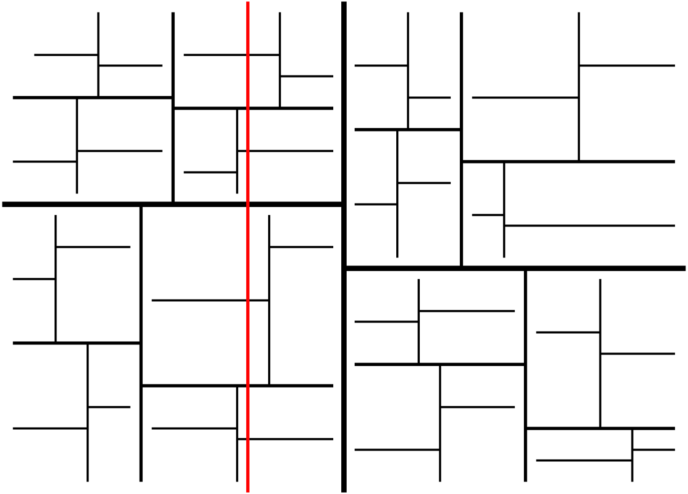
Grey node v: R intersects $region(v)$, but $region(v) \not\subseteq R$

It implies that the boundaries of R and $region(v)$ intersect

Advice: If you don't know what to do, simplify until you do

Instead of taking the boundary of R, let's analyze the number of grey nodes if the query is with a vertical line ℓ

Kd-tree query time analysis



Question: How many grey and how many black *leaves*?

Kd-tree query time analysis

We observe: At every vertical split, ℓ is only to one side, while at every horizontal split ℓ is to both sides

Let $G(n)$ be the number of grey nodes in a kd-tree with n points (leaves)

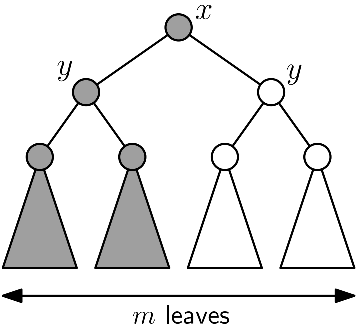
If a subtree has m leaves: $G(m) = 1 + G(n/2)$ at even depth

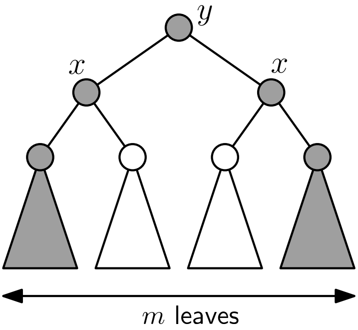
If a subtree has m leaves: $G(m) = 1 + 2 \cdot G(n/2)$ at odd depth

If we use *two levels at once*, we get:

$$G(m) = 2 + 2 \cdot G(m/4) \quad \text{or} \quad G(m) = 3 + 2 \cdot G(m/4)$$

Kd-tree query time analysis





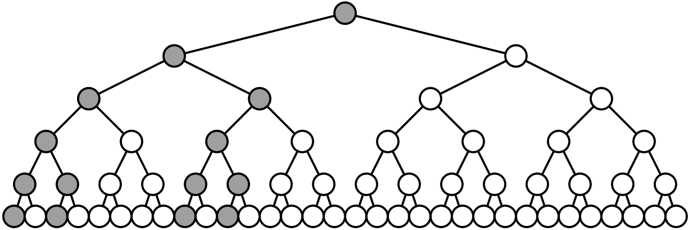
Kd-tree query time analysis

$$G(1) = 1$$

$$G(n) = 2 \cdot G(n/4) + O(1)$$

Question: What does this recurrence solve to?

Kd-tree query time analysis



The grey subtree has unary and binary nodes

Kd-tree query time analysis

The depth is $\log n$, so the binary depth is $\frac{1}{2} \cdot \log n$

Counting only binary nodes, there are
$2^{\frac{1}{2} \cdot \log n} = 2^{\log n^{1/2}} = n^{1/2} = \sqrt{n}$

Every unary grey node has a unique binary parent (except the root) $\cdots$

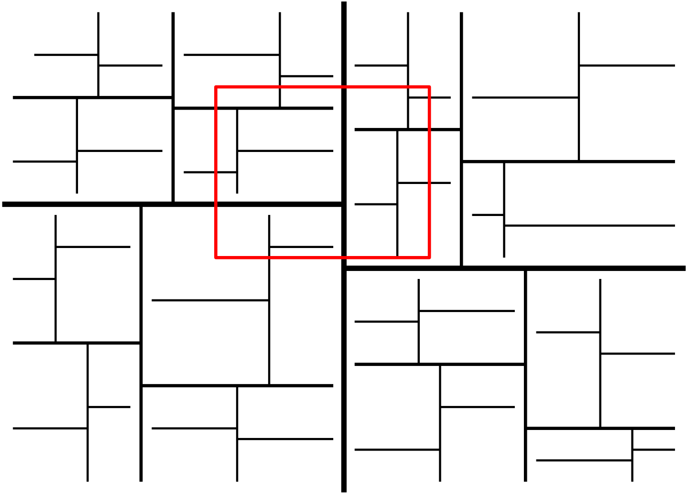
Kd-tree query time analysis

The number of grey nodes if the query were a vertical line is $O(\sqrt{n})$

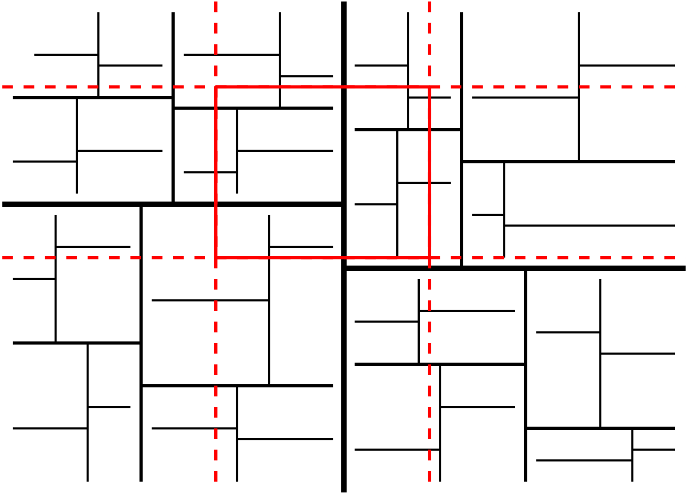
The same is true if the query were a horizontal line

How about a query rectangle?

Kd-tree query time analysis



Kd-tree query time analysis



Kd-tree query time analysis

The number of grey nodes for a query rectangle is at most the number of grey nodes for two vertical and two horizontal lines, so it is also $O(\sqrt{n})$!

For black nodes, reporting a whole subtree with k leaves takes $O(k)$ time (there are $k-1$ internal black nodes)

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

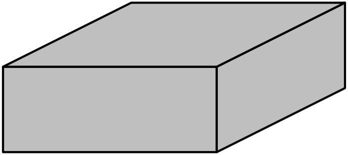
Efficiency

n	$\log n$	$\sqrt{n}$
4	2	2
16	4	4
64	6	8
256	8	16
1024	10	32
4096	12	64
1.000.000	20	1000

Higher dimensions

A 3-dimensional kd-tree alternates splits on x-, y-, and z-coordinate

A 3D range query is performed with a box



Higher dimensions

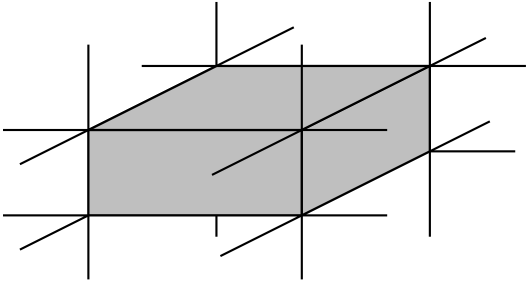
The construction of a 3D kd-tree is a trivial adaptation of the 2D version

The 3D range query algorithm is exactly the same as the 2D version

The 3D kd-tree still requires $O(n)$ storage if it stores n points

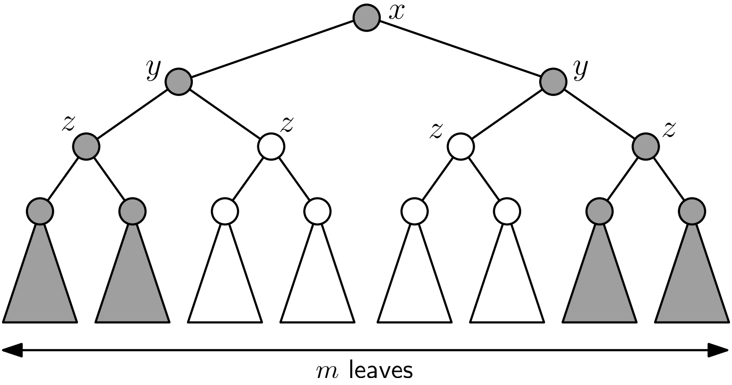
Higher dimensions

How does the query time analysis change?



Intersection of B and $region(\nu)$ depends on intersection of facets of B $\Rightarrow$ analyze by axes-parallel planes (B has no more grey nodes than six planes)

Higher dimensions



Kd-tree query time analysis

Let $G_3(n)$ be the number of grey nodes for a query with an axes-parallel plane in a 3D kd-tree

$G_3(1) = 1$

$G_3(n) = 4 \cdot G_3(n/8) + O(1)$

Question: What does this recurrence solve to?

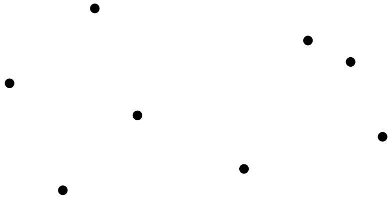
Question: How many leaves does a perfectly balanced binary search tree with depth $\frac{2}{3}\log n$ have?

Result

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

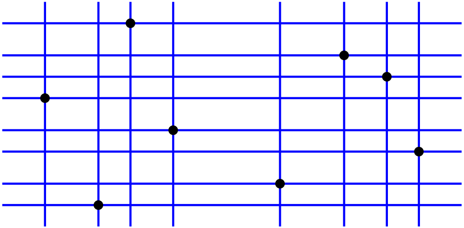
Faster queries

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

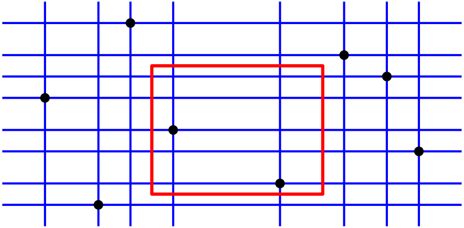


Faster queries

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



Faster queries

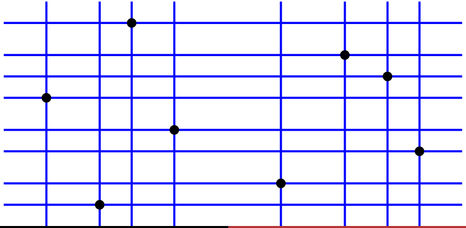


If the corners of the query rectangle fall in specific cells of the grid, the answer is fixed (even for lower left and upper right corner)

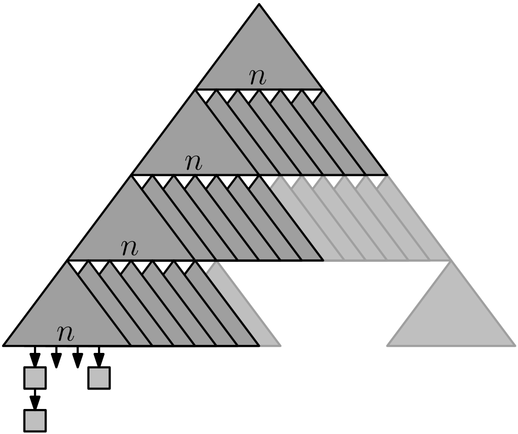
Faster queries

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?

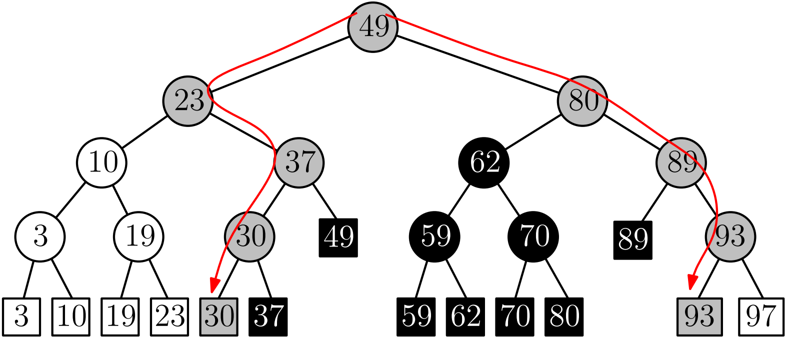
Faster queries

Recall the 1D range tree and range query:

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

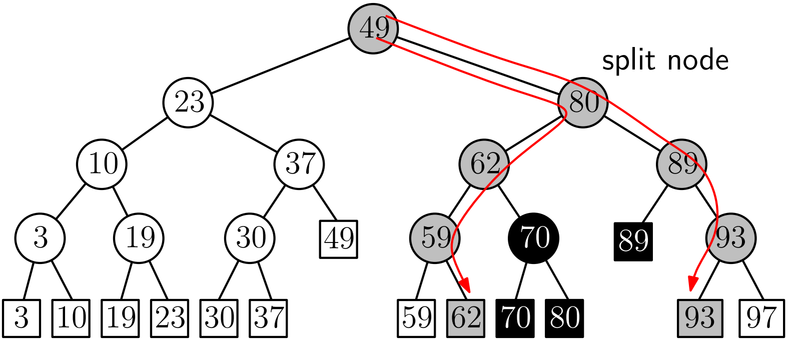
Example 1D range query

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?

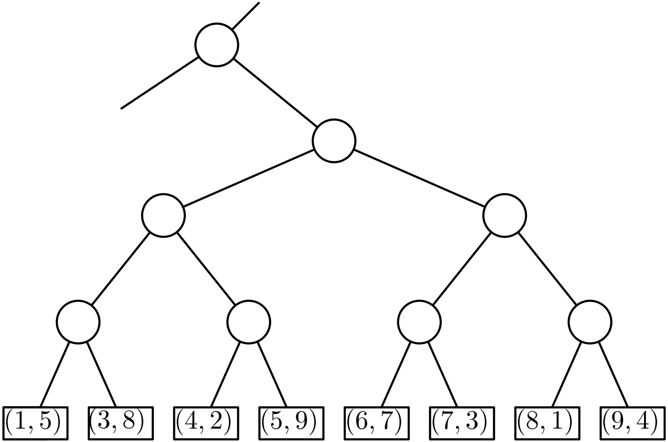
Examining 1D range queries

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

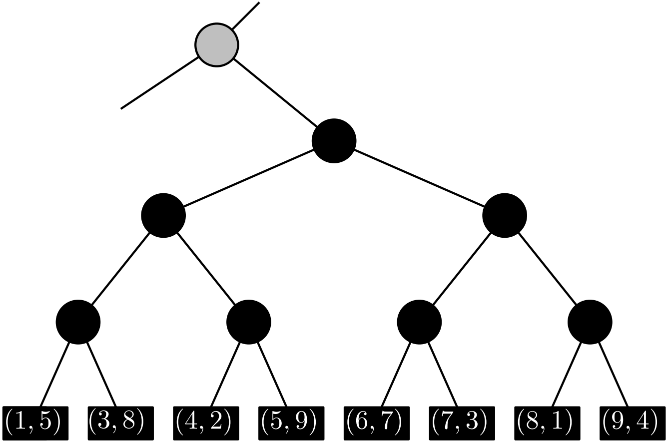
Toward 2D range queries

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

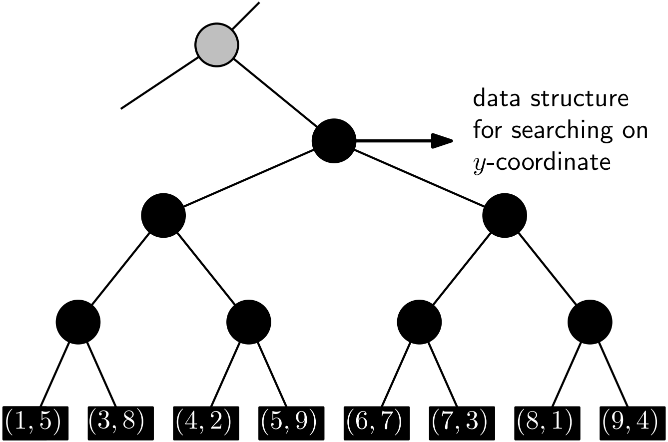
Toward 2D range queries



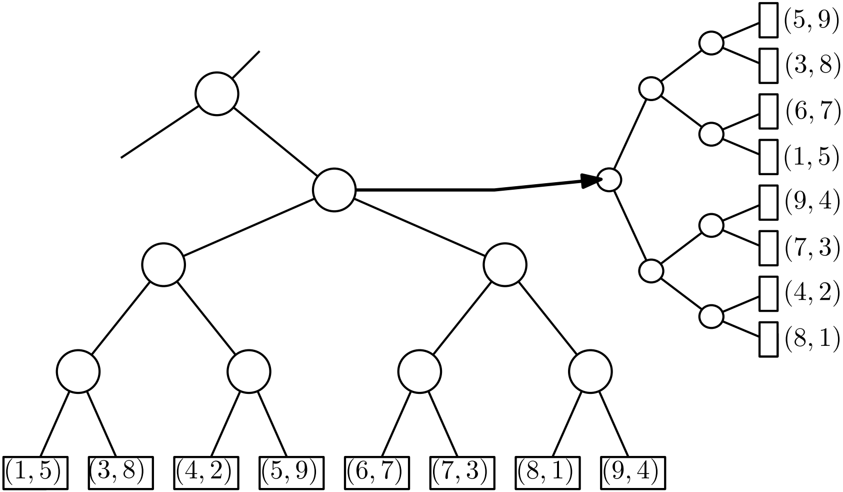
Toward 2D range queries



Toward 2D range queries

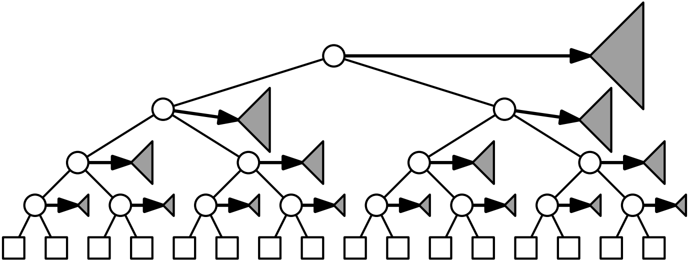


Toward 2D range queries



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 of 2D 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 $\textsc{Build2DRangeTree}(P)$

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 ν storing this point, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of ν.
4. **else** Split P into P_{left} and P_{right}, the subsets $\leq$ and $>$ the median x-coordinate x_{mid}
5. $\nu_{\text{left}} \leftarrow \textsc{Build2DRangeTree}(P_{\text{left}})$
6. $\nu_{\text{right}} \leftarrow \textsc{Build2DRangeTree}(P_{\text{right}})$
7. Create a node ν storing x_{mid}, make ν_{left} the left child of ν, make ν_{right} the right child of ν, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of ν
8. **return** ν

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

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

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

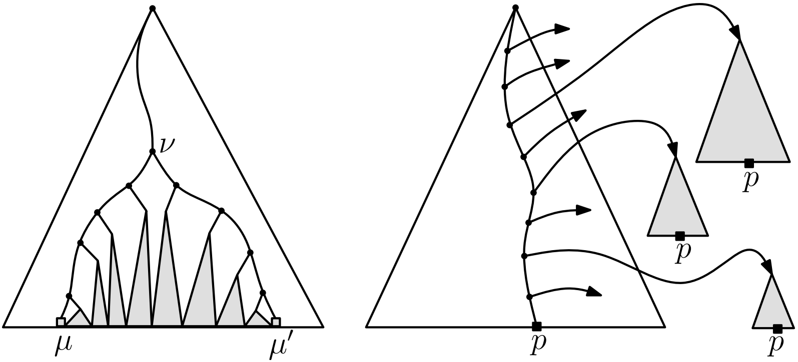
which solves to $O(n \log n)$ time

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 $2\text{DR}{\small\text{ANGE}}\text{Q}{\small\text{UERY}}(\mathcal{T}, [x : x'] \times [y : y'])$

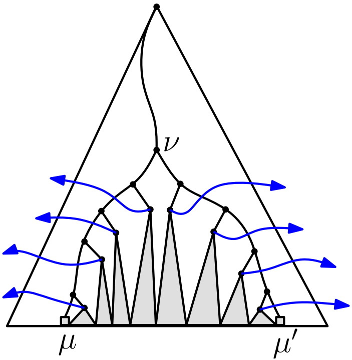
$\nu_{\text{split}} \leftarrow \text{F}{\small\text{IND}}\text{S}{\small\text{PLIT}}\text{N}{\small\text{ODE}}(\mathcal{T}, x, x')$
if ν_{split} is a leaf
  then report the point stored at ν_{split}, if an answer
  else $\nu \leftarrow lc(\nu_{\text{split}})$
    while ν is not a leaf
      do if $x \leq x_\nu$
        then $1\text{DR}{\small\text{ANGE}}\text{Q}(\mathcal{T}_{\text{assoc}}(rc(\nu)), [y : y'])$
          $\nu \leftarrow lc(\nu)$
        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

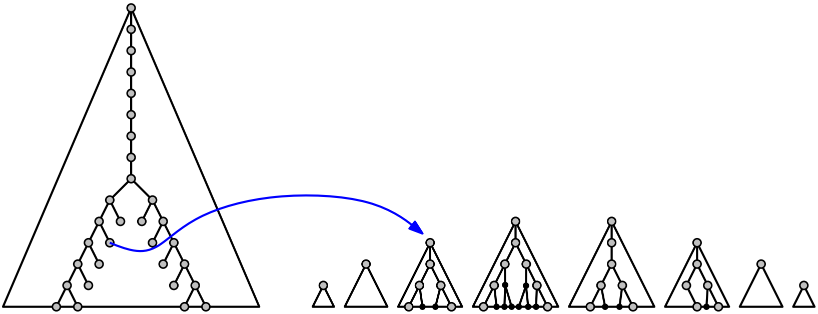
The query time is $O(\log n) \times O(\log m + k')$, or

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

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

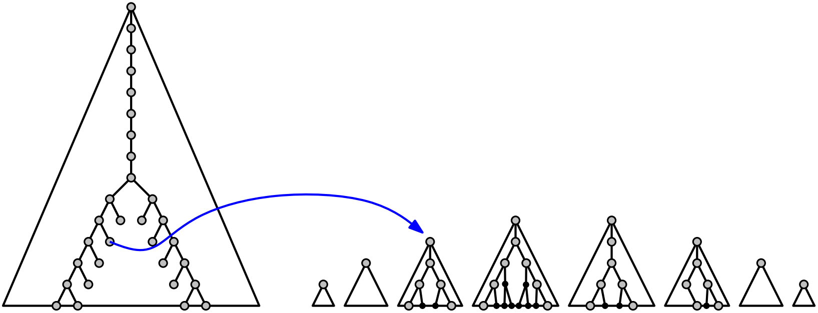
Recall that a kd-tree has $O(n)$ size and answers queries in $O(\sqrt{n} + k)$ time

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

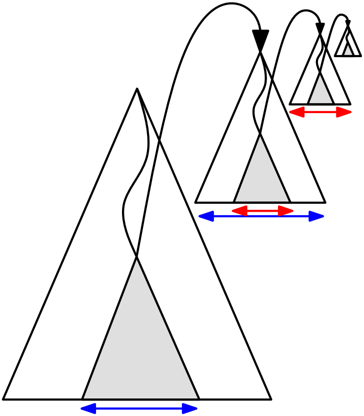
2D range query efficiency

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

The size $S_d(n)$ of a d-dimensional range tree satisfies:

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

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

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

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 d-dimensional space 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

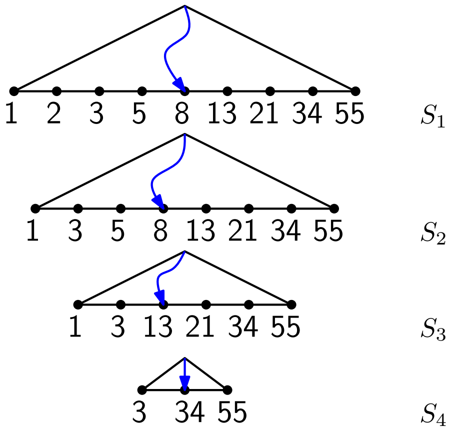
Improving the query time

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

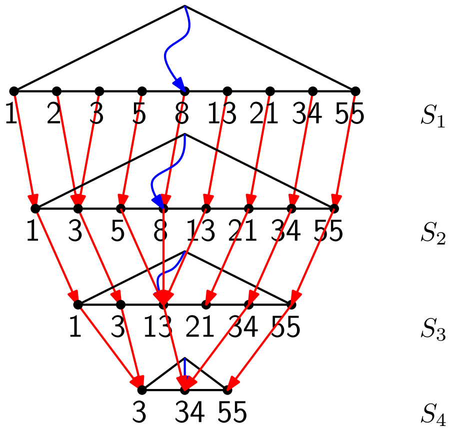
Suppose that we have a collection of sets $S_1, \ldots, S_m$, where $|S_1| = n$ and where $S_{i+1} \subseteq S_i$

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

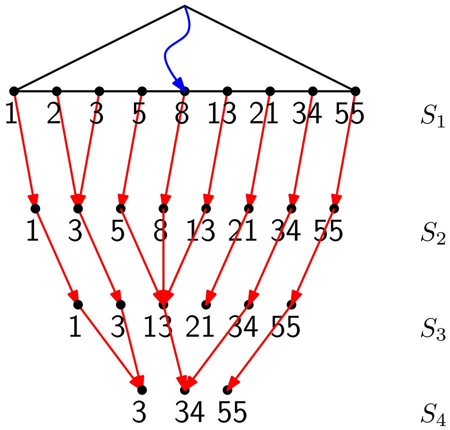
Improving the query time



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 $S_1, \ldots, S_m$

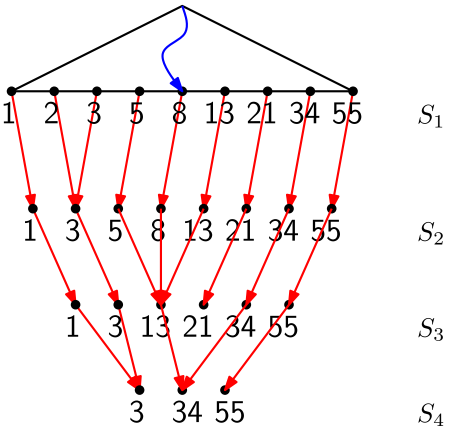
This query problem can be solved in $O(\log n + m)$ time instead of $O(m \cdot \log n)$ time

Improving the query time

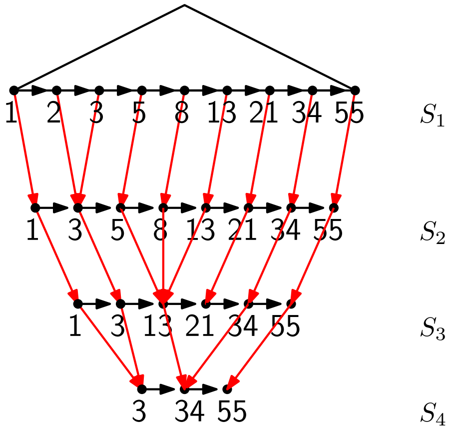
Can we do something similar for m 1-dimensional range queries on m sets $S_1, \ldots, S_m$?

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

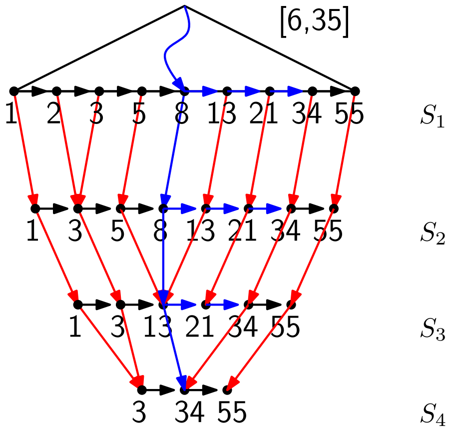
Improving the query time



Improving the query time



Improving the query time



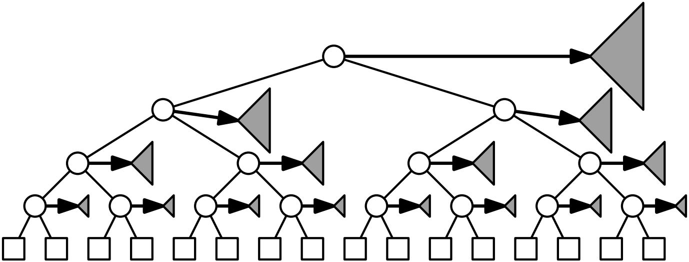
Fractional cascading

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

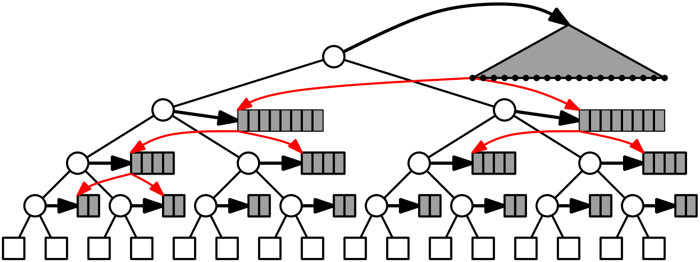
Note that in every associated structure, we search with the same values y and y'

- 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

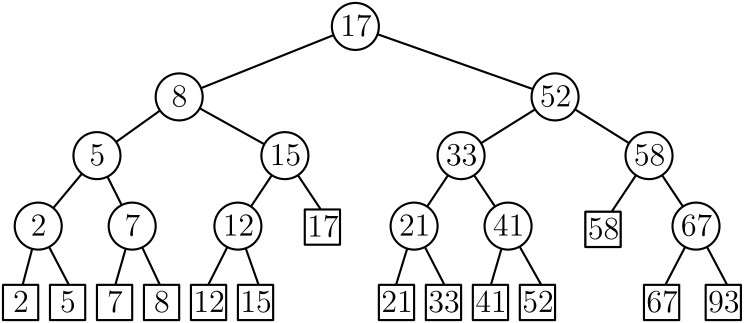
Fractional cascading



Fractional cascading



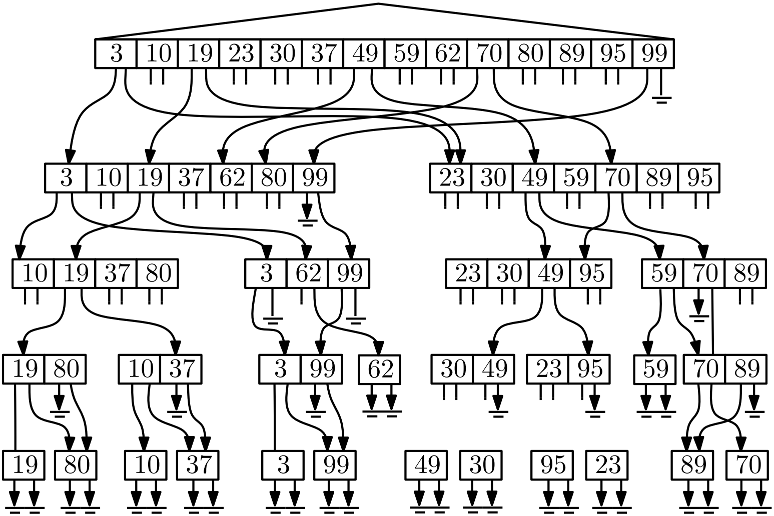
Fractional cascading



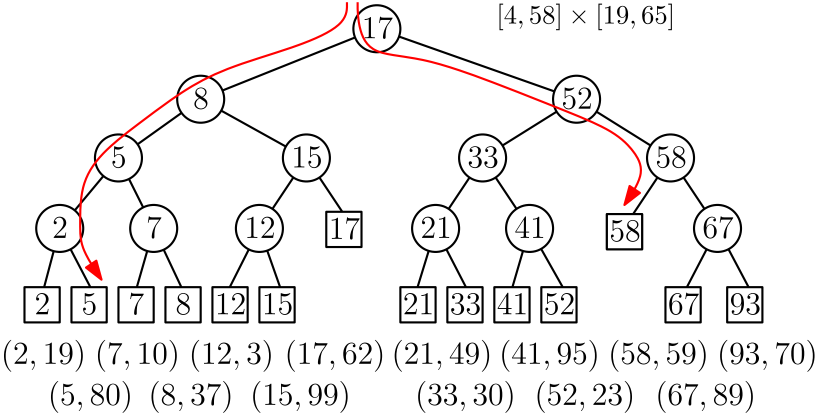
$(2, 19)$ $(7, 10)$ $(12, 3)$ $(17, 62)$ $(21, 49)$ $(41, 95)$ $(58, 59)$ $(93, 70)$

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

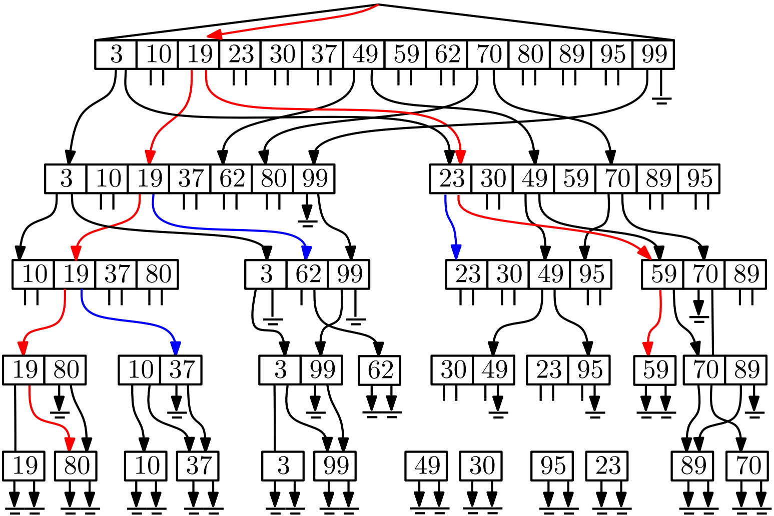
Fractional cascading



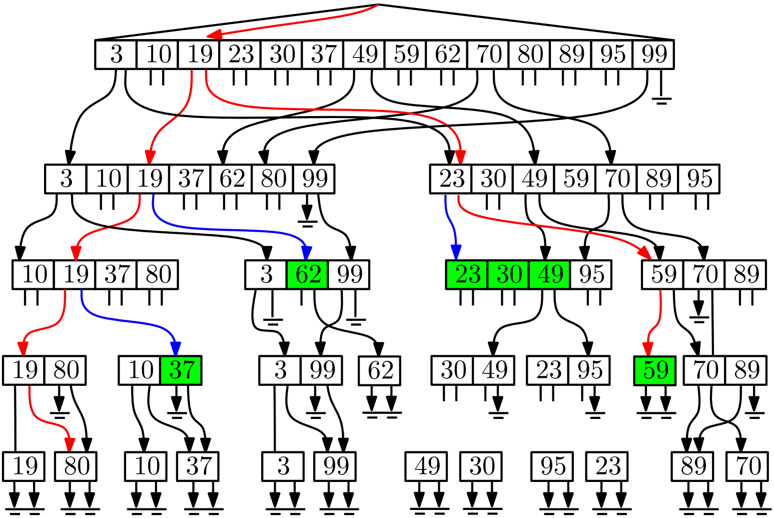
Fractional cascading



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 $O(\log n)$

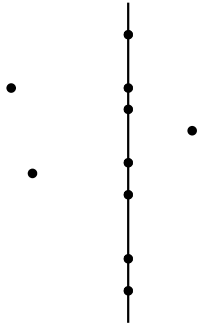
Result

Theorem: A set of n points in d-dimensional space 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

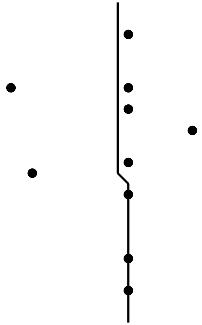
Degenerate cases

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

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