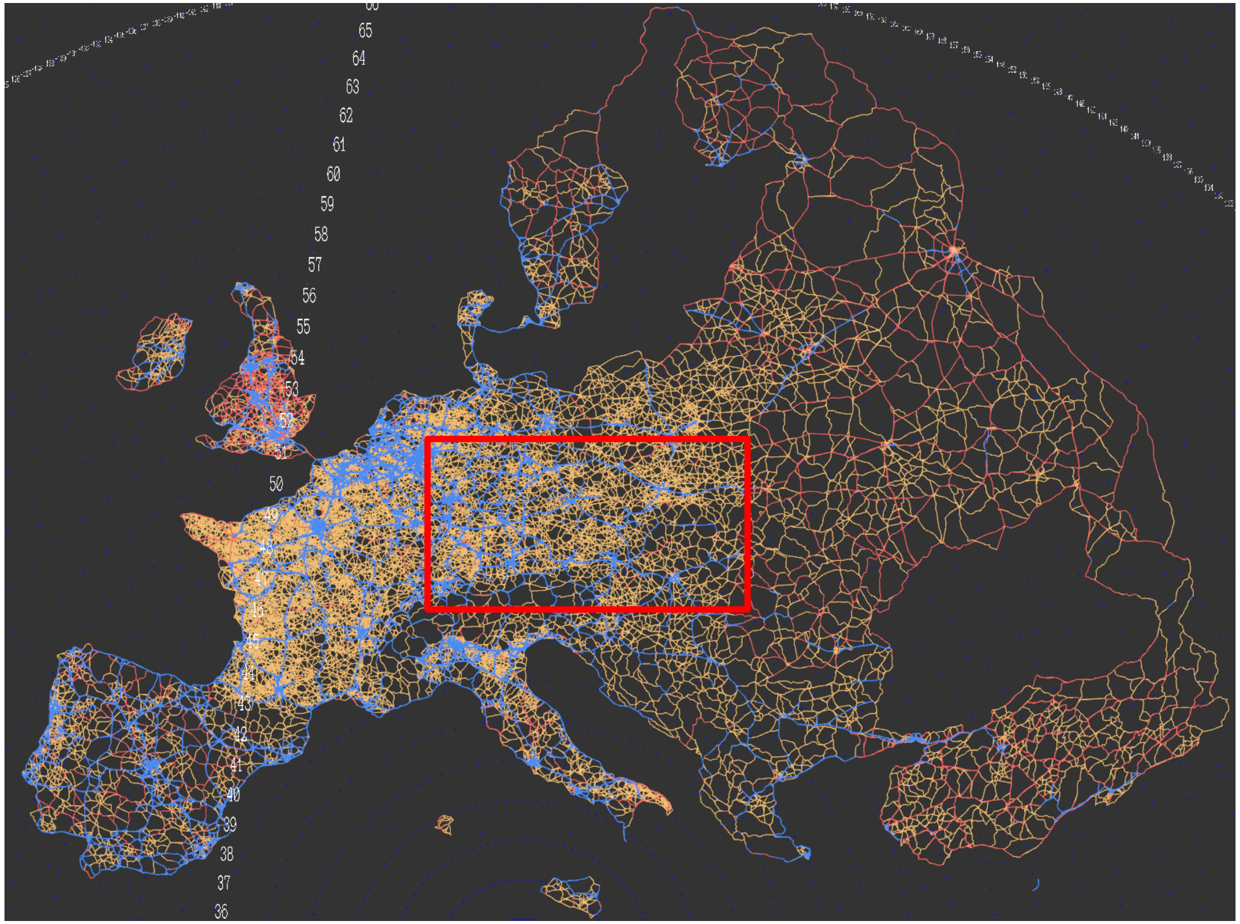
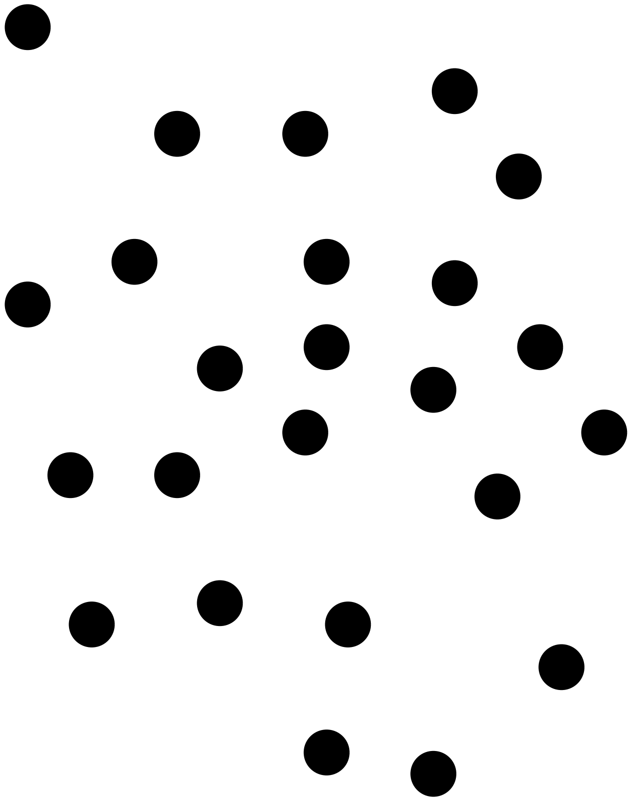
Prioritized Geographic Search

SPP subproject "Lightweight Acquisition and Large-Scale Mining of Trajectory Data"



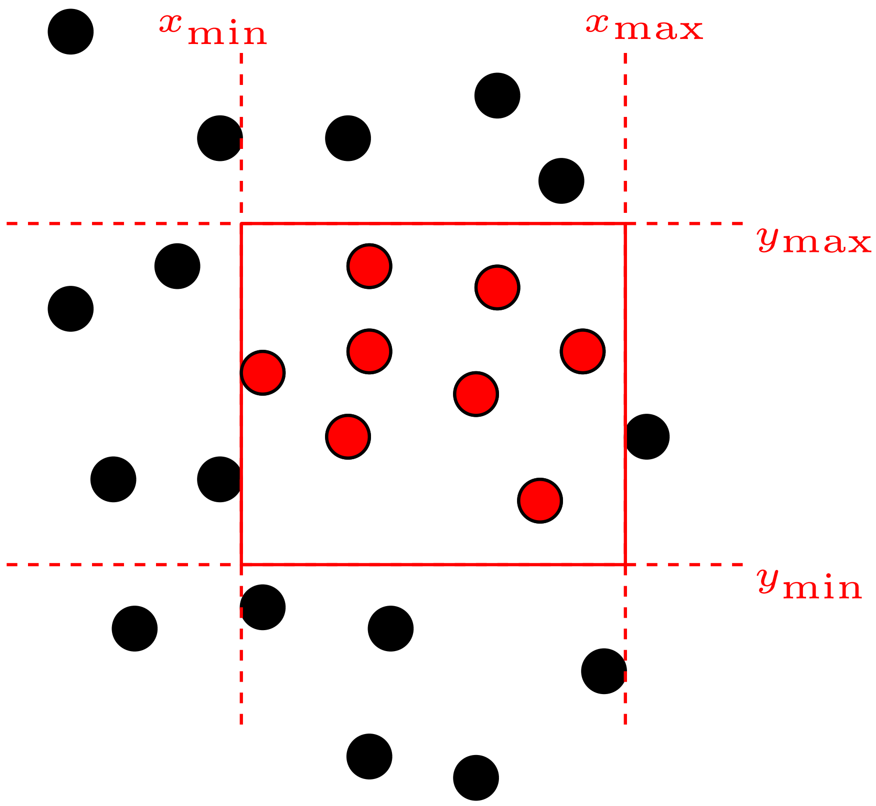
Prof. Stefan Funke
Institut f. Formale Methoden der Informatik
Abteilung Algorithmik
University of Stuttgart
Germany

Basic Geographic Range Search



Given: Set S of n points in $\mathbb{R}^2$

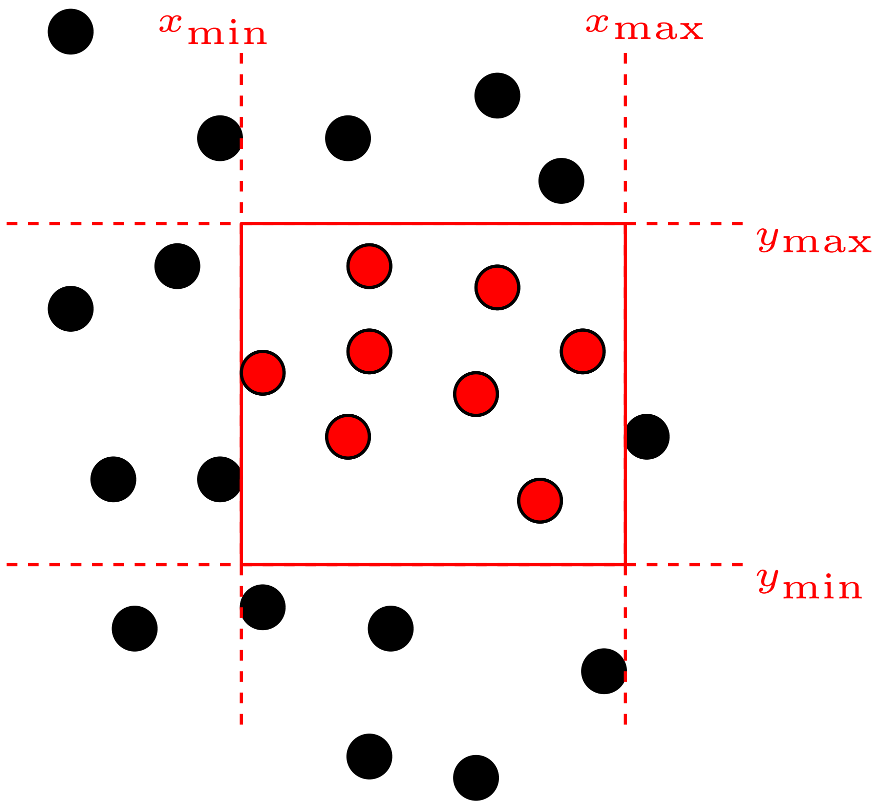
Basic Geographic Range Search



Given: Set S of n points in $\mathbb{R}^2$

Goal: Construct data structure $\mathcal{D}$ such that *range queries* specified by $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ can be answered efficiently.

Basic Geographic Range Search

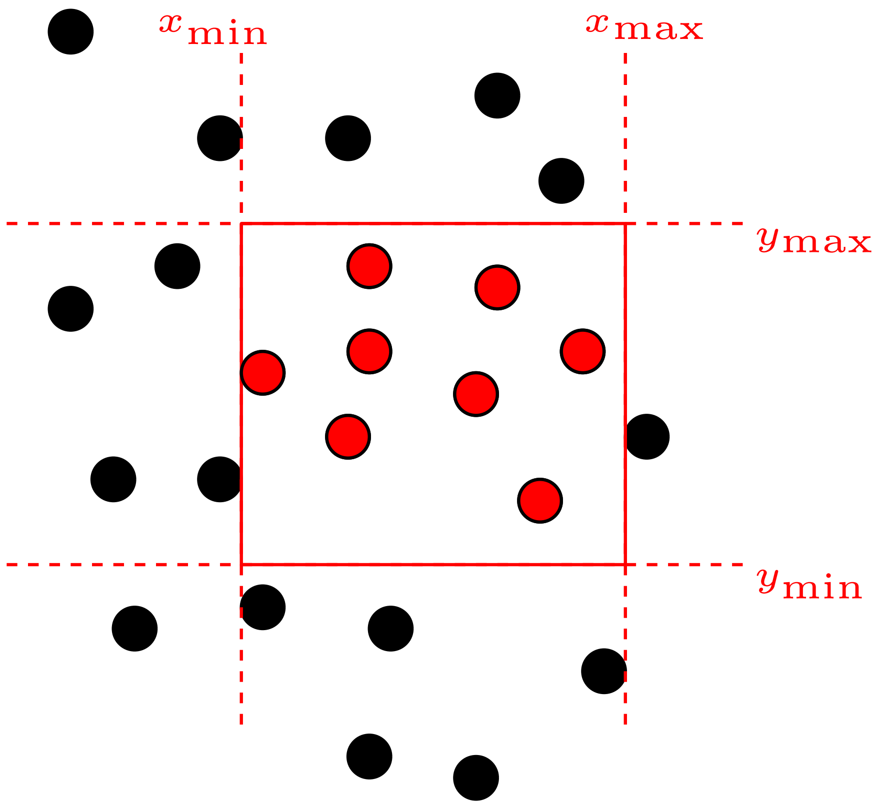


Given: Set S of n points in $\mathbb{R}^2$

Goal: Construct data structure $\mathcal{D}$ such that *range queries* specified by $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ can be answered efficiently.

Examples:
Output within a given area:
- all Greek restaurants
- locations of all mailboxes
- ...

Basic Geographic Range Search



Given: Set S of n points in $\mathbb{R}^2$

Goal: Construct data structure $\mathcal{D}$ such that *range queries* specified by $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ can be answered efficiently.

Examples:
Output within a given area:
- all Greek restaurants
- locations of all mailboxes
- ...

We care about:
- time $Q(n)$ to answer a single query
- space $S(n)$ of the data structure $\mathcal{D}$
- time $P(n)$ to construct $\mathcal{D}$

Basic Geographic Range Search – Challenges

Consider the query "all towns within the query rectangle"

Basic Geographic Range Search – Challenges

Consider the query "all towns within the query rectangle"

The result set is likely to contain several thousands of items . . .

Basic Geographic Range Search – Challenges

Consider the query "all towns within the query rectangle"

The result set is likely to contain several thousands of items . . .

Depending on the purpose of the query, it is pretty hard to make sense of this result set . . .

Analogy: Web Search $\approx$ Geographic Search

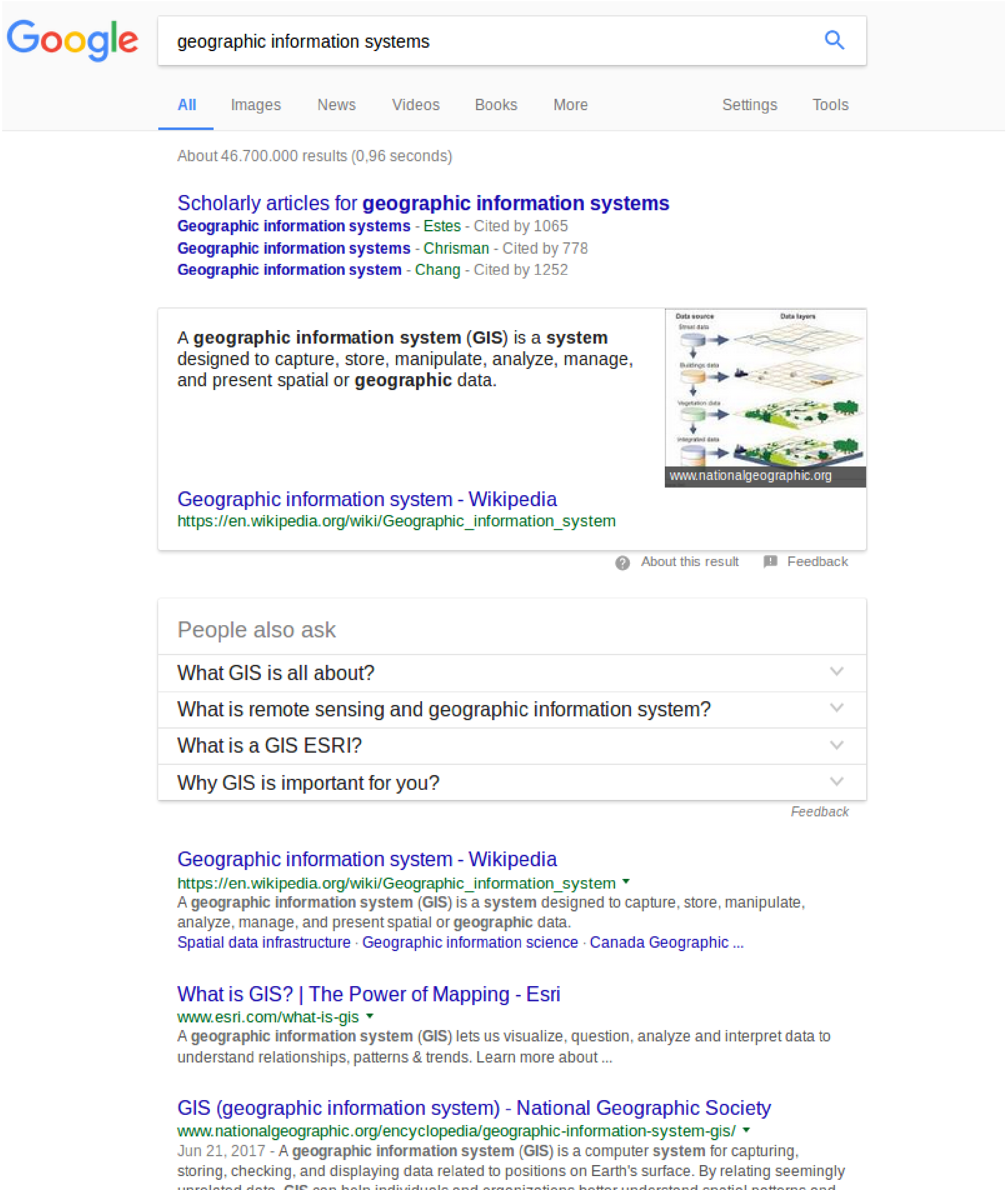
Web Search:

- Query for "geographic information systems" yields $\approx$ 46 Million results / webpages related to the search term
- yet, we can make sense of the outcome as Google *prioritizes* the outcome in a sensible manner
- good prioritization and prioritized querying main reason for Google prevailing over Yahoo, Altavista, ...

Analogy: Web Search ≈ Geographic Search

Web Search:

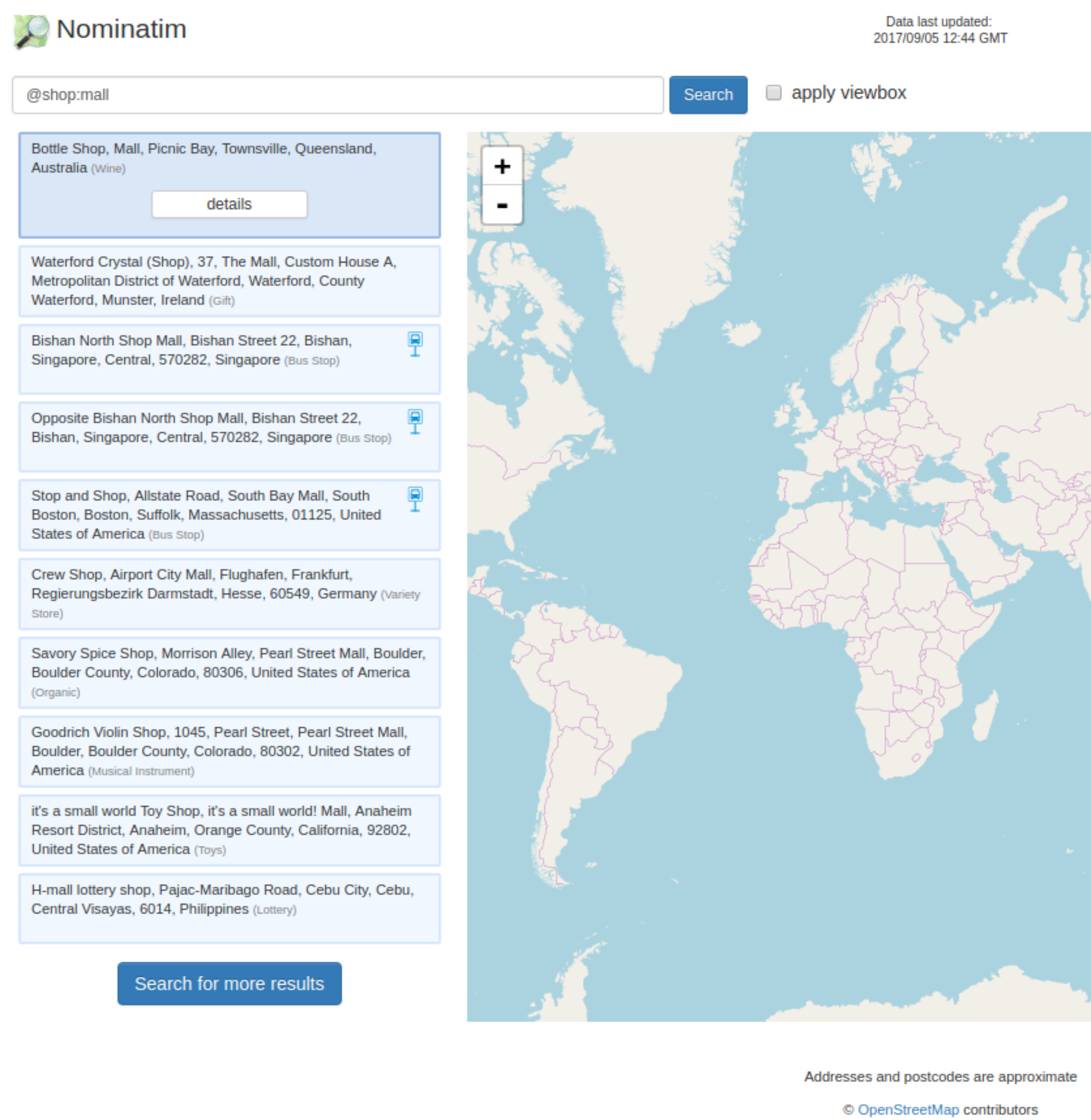
- Query for "geographic information systems" yields ≈ 46 Million results / webpages related to the search term
- yet, we can make sense of the outcome as Google *prioritizes* the outcome in a sensible manner
- good prioritization and prioritized querying main reason for Google prevailing over Yahoo, Altavista, ...



Analogy: Web Search ≈ Geographic Search

Geographic Search:

- the OpenStreetMap planet data set comprises more than 3 billion items
- an individual query might return hundreds of thousands of items
- again: *prioritization* key for reasonable use and interpretation of results



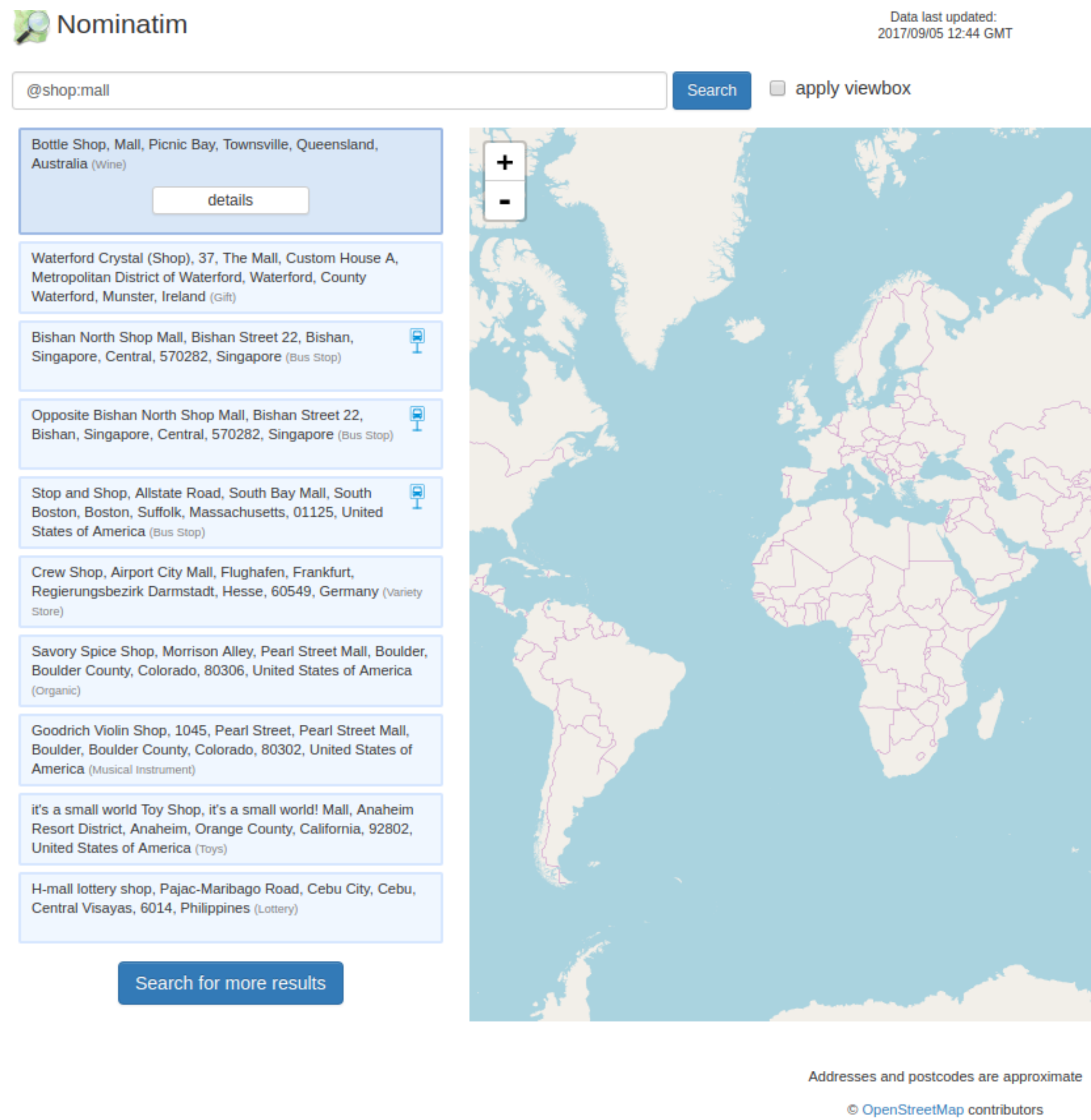
Analogy: Web Search ≈ Geographic Search

Geographic Search:

- the OpenStreetMap planet data set comprises more than 3 billion items
- an individual query might return hundreds of thousands of items
- again: *prioritization* key for reasonable use and interpretation of results

Query: @shop:mall

- ≈ 2,700 in Germany
- prioritization according to size seems very reasonable



Application of Prioritized Range Search: Map Rendering

When drawing a zoomed-out view of a map, only the most important (largest?) cities should be drawn . . .

Application of Prioritized Range Search: Map Rendering

When drawing a zoomed-out view of a map, only the most important (largest?) cities should be drawn . . .

When zooming-in, also medium-sized towns should be on the rendering . . .

Application of Prioritized Range Search: Map Rendering

When drawing a zoomed-out view of a map, only the most important (largest?) cities should be drawn . . .

When zooming-in, also medium-sized towns should be on the rendering . . .

The further we zoom-in, the more smaller villages we want to see on the map . . .

Application of Prioritized Range Search: Map Rendering

When drawing a zoomed-out view of a map, only the most important (largest?) cities should be drawn . . .

When zooming-in, also medium-sized towns should be on the rendering . . .

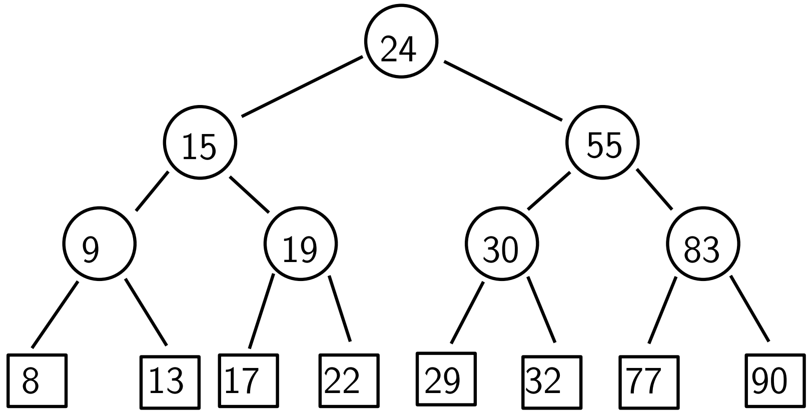
The further we zoom-in, the more smaller villages we want to see on the map . . .

(Almost) direct application of prioritized range queries!

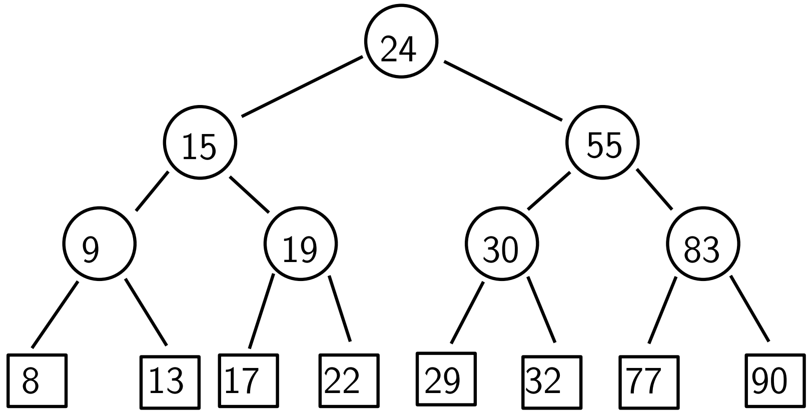
Two-dimensional Range Queries (no priorities yet)

1-dimensional range queries:

- use tree structure
- logarithmic depth
- space consumption for n elements: S(n)=O(n)
- query time for with result size k: Q(n)=O($\log n + k$)



Two-dimensional Range Queries (no priorities yet)



1-dimensional range queries:

- use tree structure
- logarithmic depth
- space consumption for n elements: S(n)=O(n)
- query time for with result size k: Q(n)=O($\log n + k$)

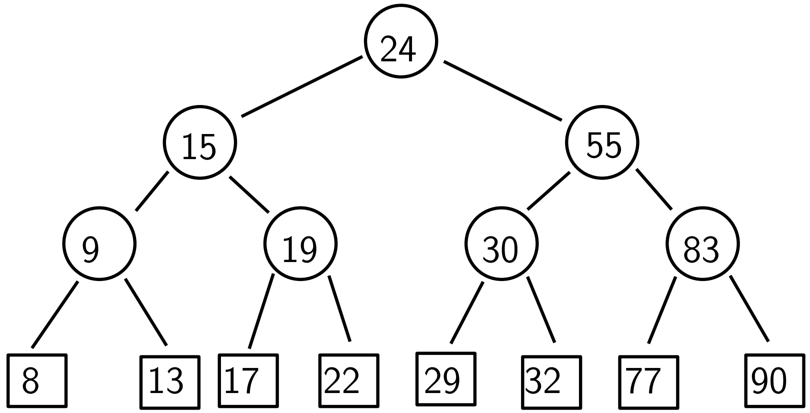
Naive generalization to 2 dimensions:

- build tree on x-coordinates
- build tree on y-coordinates
- S(n)=O(n)
- query x- and y-ranges separately, return intersection
- Q(n)=?

Two-dimensional Range Queries (no priorities yet)

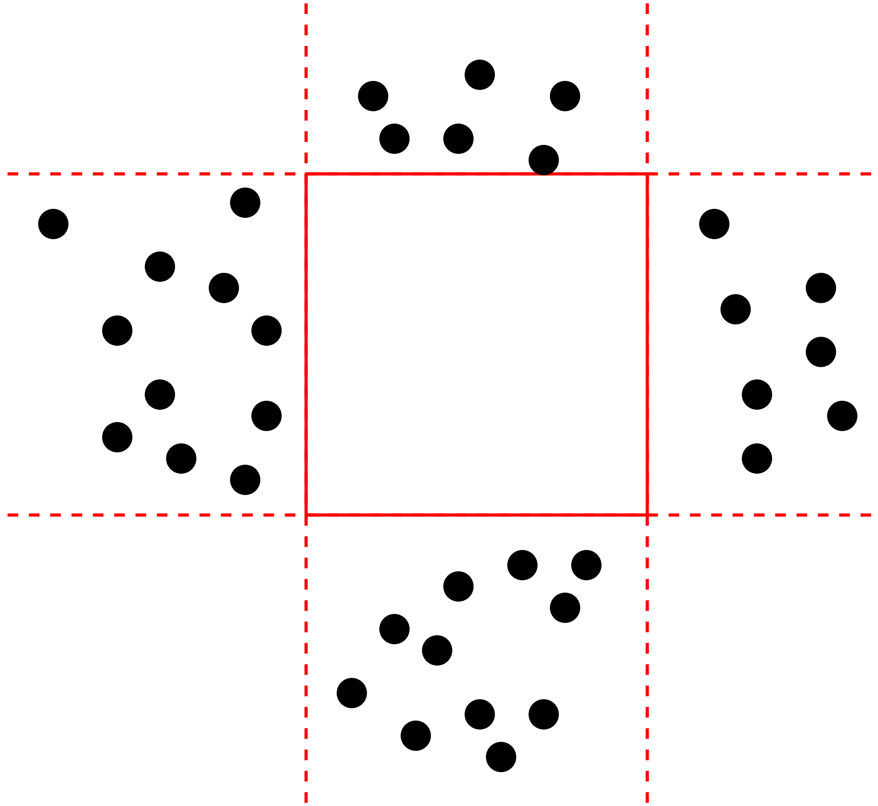
1-dimensional range queries:

- use tree structure
- logarithmic depth
- space consumption for n elements: $S(n)=O(n)$
- query time for with result size k: $Q(n)=O(\log n + k)$



Naive generalization to 2 dimensions:

- build tree on x-coordinates
- build tree on y-coordinates
- $S(n)=O(n)$
- query x- and y-ranges separately, return intersection
- $Q(n)=?$

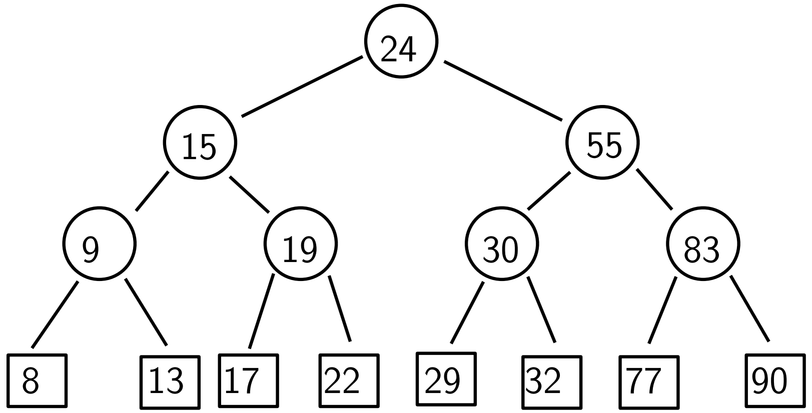


$Q(n)=\Theta(n)$ for a result size of 0!

1st Generalization Attempt: kd-Tree (no priorities yet)

1-dimensional tree construction:

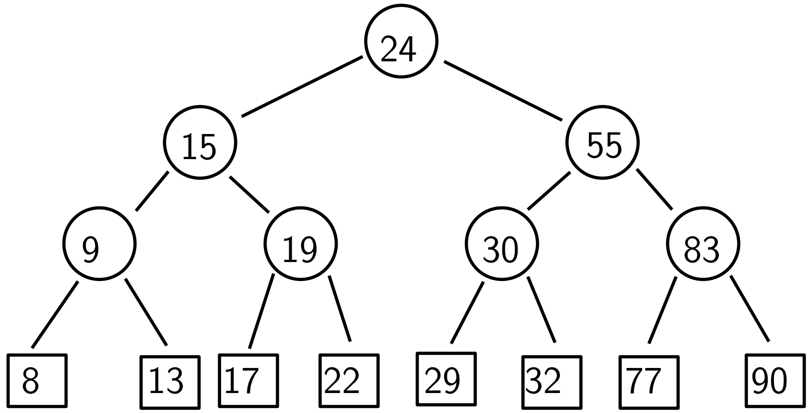
- recursively via median splitting
- construction time $P(n) = O(n \log n)$



1st Generalization Attempt: kd-Tree (no priorities yet)

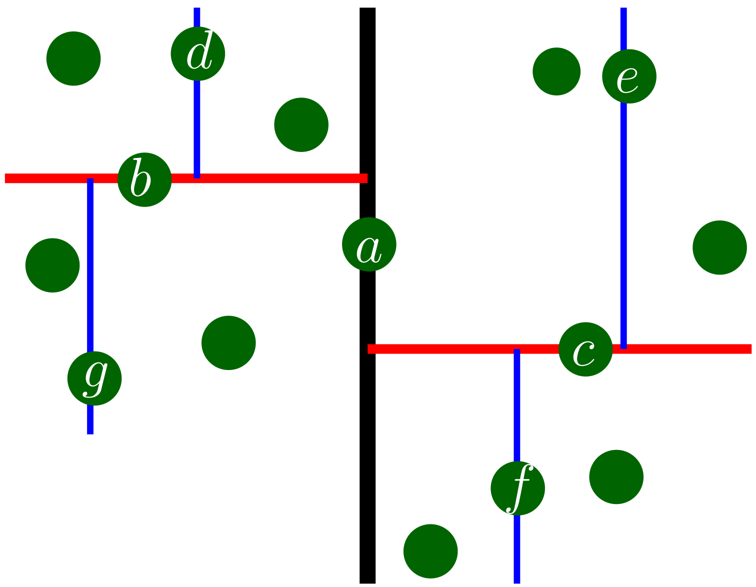
1-dimensional tree construction:

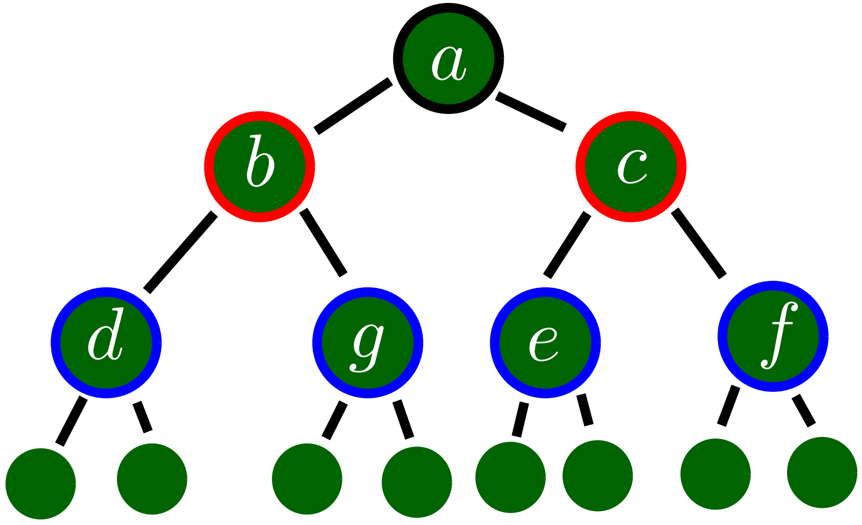
- recursively via median splitting
- construction time $P(n) = O(n \log n)$



Generalization:

- alternate between median splitting according to x- and y-coordinate
- construction time $O(n \log n)$
- space consumption $S(n) = O(n)$
- how to query? query time?

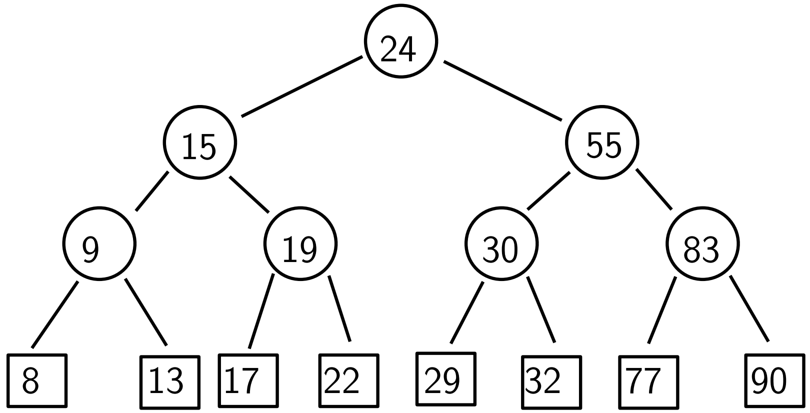




1st Generalization Attempt: kd-Tree (no priorities yet)

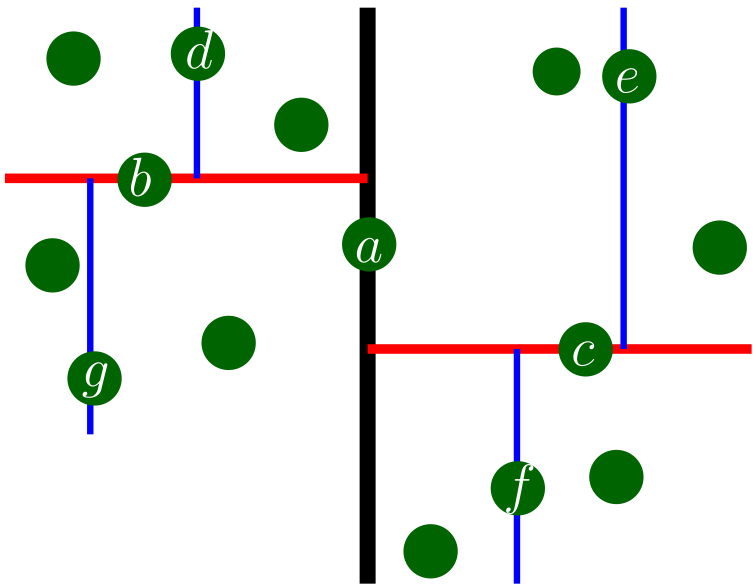
1-dimensional tree construction:

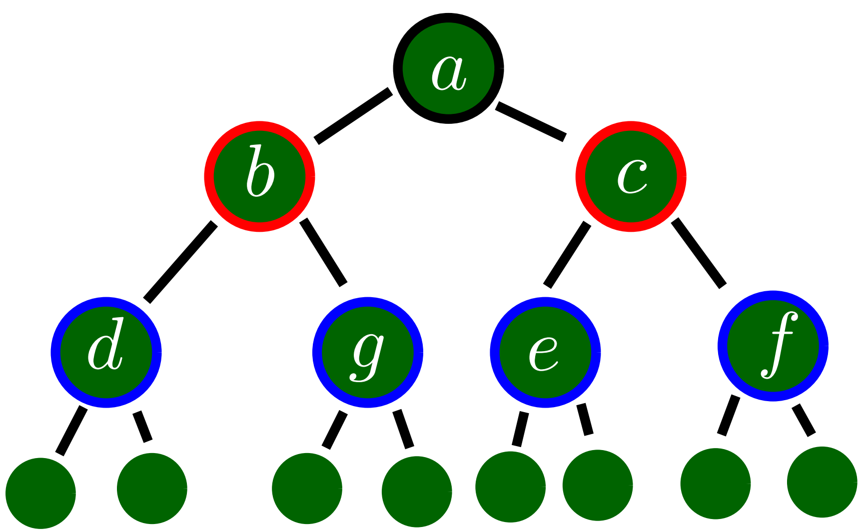
- recursively via median splitting
- construction time $P(n) = O(n \log n)$



Generalization:

- alternate between median splitting according to x- and y-coordinate
- construction time $O(n \log n)$
- space consumption $S(n) = O(n)$
- how to query? query time?



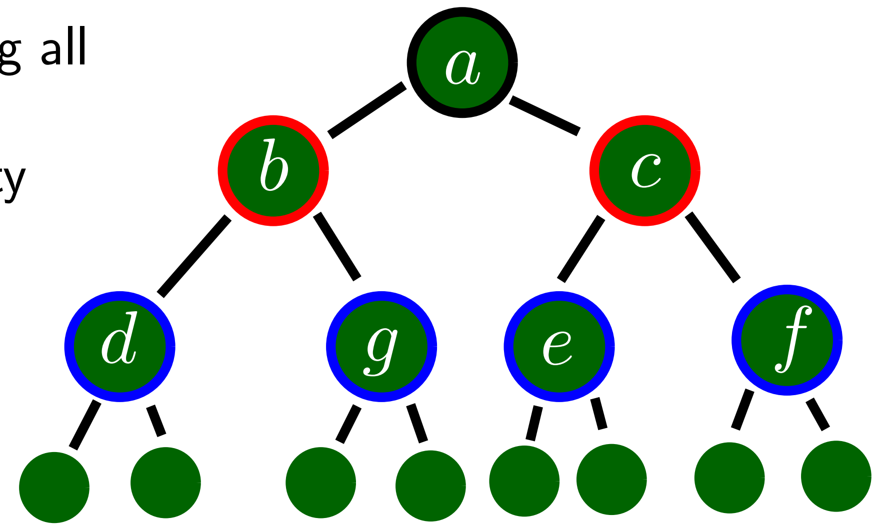


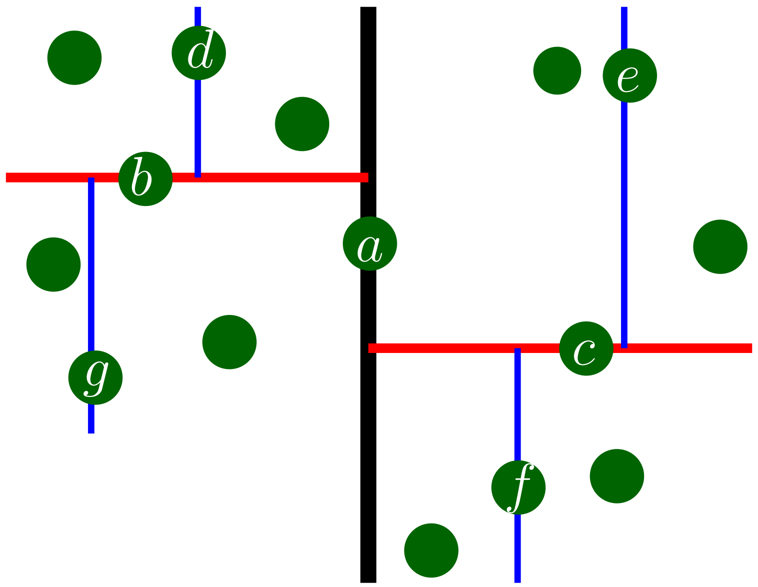
kd-Tree

1st Generalization Attempt: kd-Tree (no priorities yet)

Query:

- associate with each node rectangle containing all points of the subtree
- recursively traverse rectangles with non-empty intersection with query rectangle
- query time $Q(n) = ?$

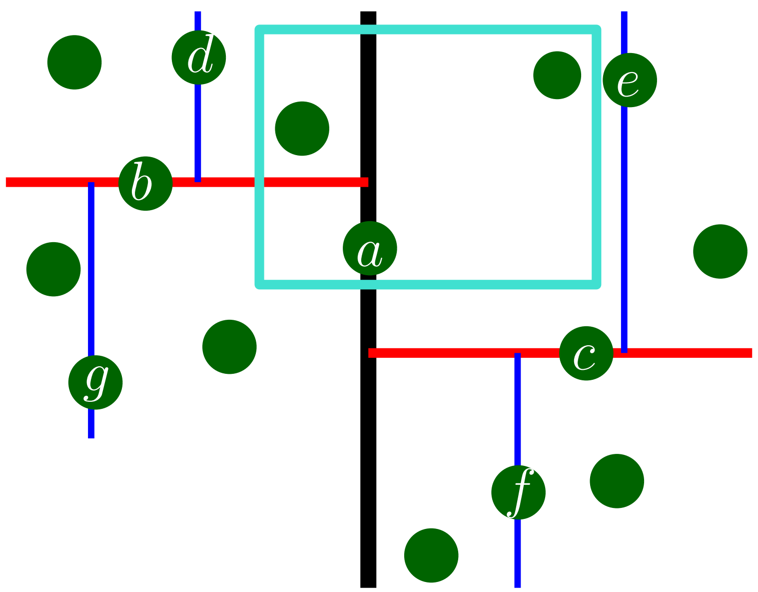


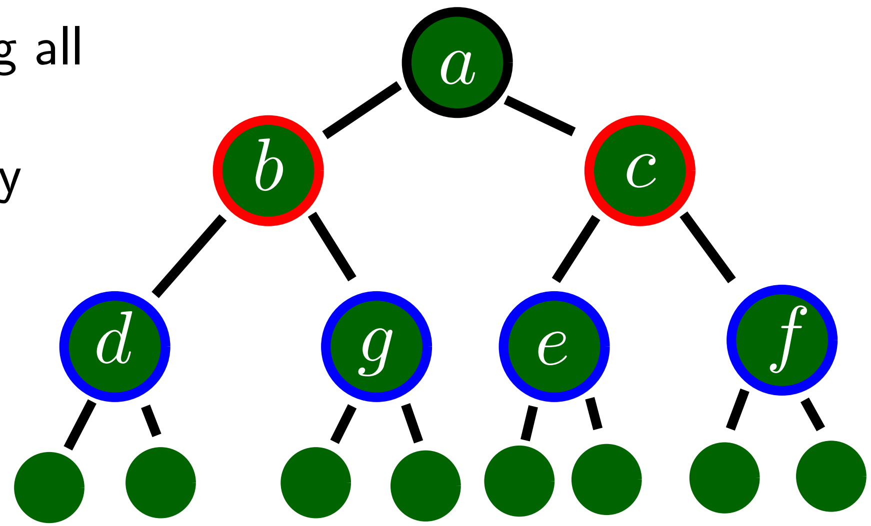


1st Generalization Attempt: kd-Tree (no priorities yet)

Query:

- associate with each node rectangle containing all points of the subtree
- recursively traverse rectangles with non-empty intersection with query rectangle
- query time $Q(n) = ?$

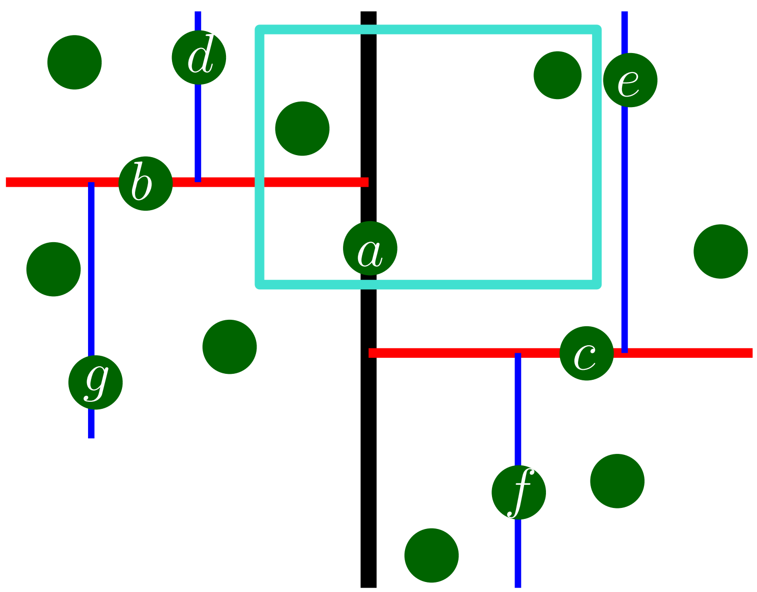


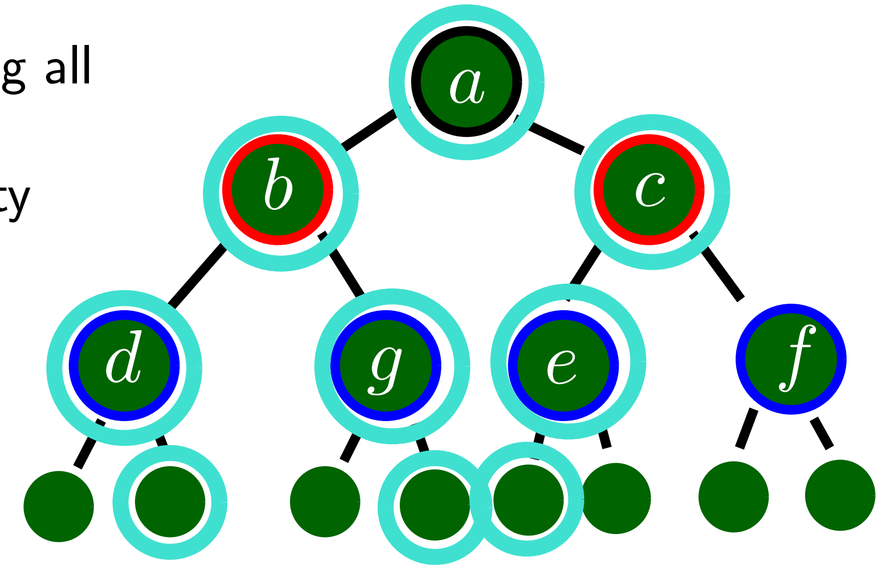


1st Generalization Attempt: kd-Tree (no priorities yet)

Query:

- associate with each node rectangle containing all points of the subtree
- recursively traverse rectangles with non-empty intersection with query rectangle
- query time $Q(n) = ?$

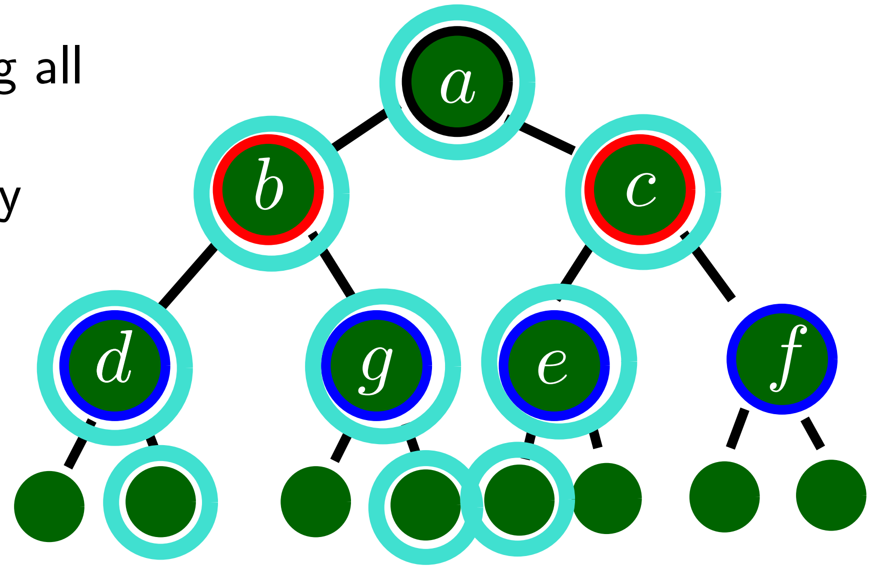


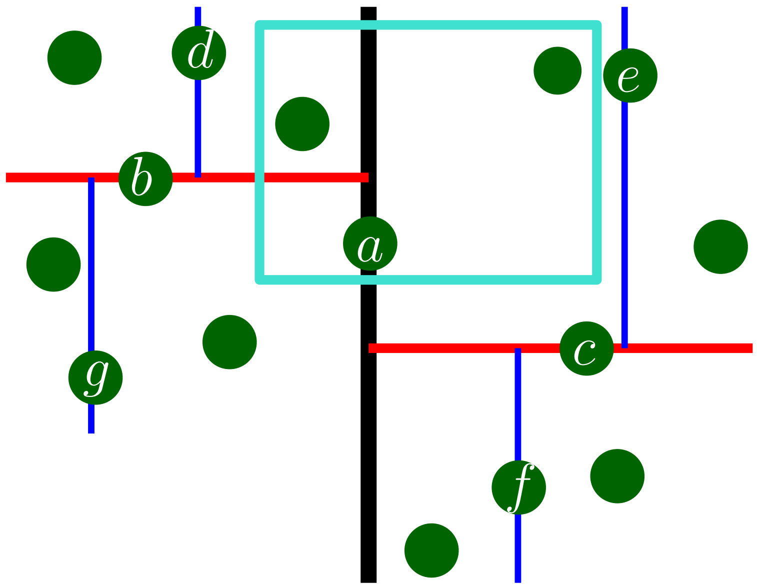


1st Generalization Attempt: kd-Tree (no priorities yet)

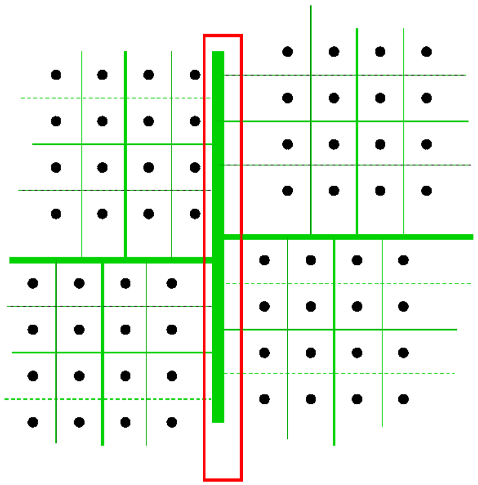
Query:

- associate with each node rectangle containing all points of the subtree
- recursively traverse rectangles with non-empty intersection with query rectangle
- query time $Q(n) =?$





- Query time can be $Q(n) = \Omega(\sqrt{n})$ with 0 points reported :(
- but it cannot be worse $Q(n) = O(\sqrt{n} + k)$



1st Generalization Attempt: kd-Tree - Including Priorities

- every point now also bears a priority
- query is of the form

$$(x_{\min}, x_{\max}, y_{\min}, y_{\max}, prio_{\min})$$

1st Generalization Attempt: kd-Tree - Including Priorities

- every point now also bears a priority
- query is of the form

$$(x_{\min}, x_{\max}, y_{\min}, y_{\max}, prio_{\min})$$

We could treat priority as 3rd dimension
(allowing also max-priorities)

- size $S(n) = O(n)$
- construction time $P(n) = O(n \log n)$
- query time $Q(n) = O(n^{2/3} + k)$

1st Generalization Attempt: kd-Tree - Including Priorities

- every point now also bears a priority
- query is of the form

$$(x_{\min}, x_{\max}, y_{\min}, y_{\max}, prio_{\min})$$

We could treat priority as 3rd dimension
(allowing also max-priorities)

- size $S(n) = O(n)$
- construction time $P(n) = O(n \log n)$
- query time $Q(n) = O(n^{2/3} + k)$

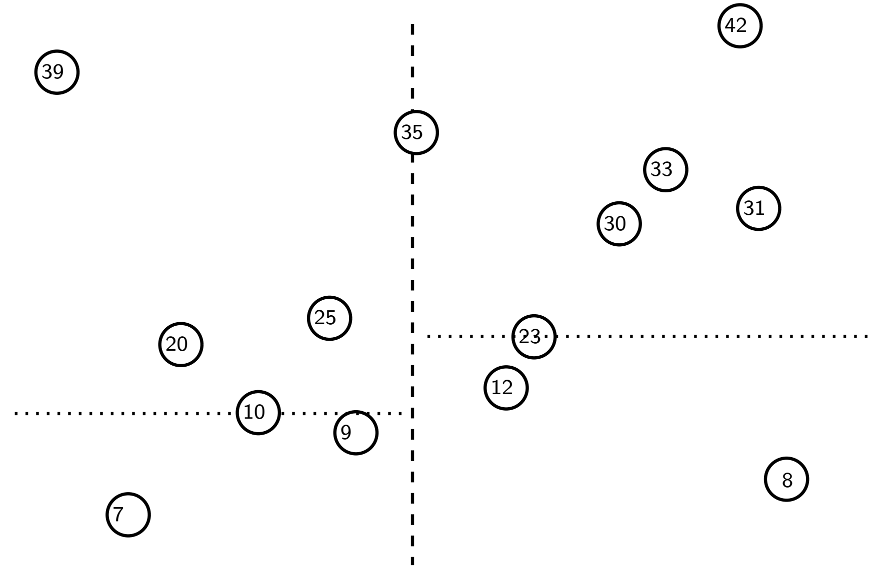
Better alternative:

- Augmented kd-prio tree:
  - before median extraction, extract max-priority point and store within node
- space $S(n) = O(n \log n)$
- construction time $P(n) = O(n \log n)$
- query time $Q(n) = O(\sqrt{n} + k)$
- abort subtree inspection as soon as priority is below $prio_{\min}$

1st Generalization Attempt: kd-Tree - Including Priorities

- every point now also bears a priority
- query is of the form

$$(x_{\min}, x_{\max}, y_{\min}, y_{\max}, prio_{\min})$$

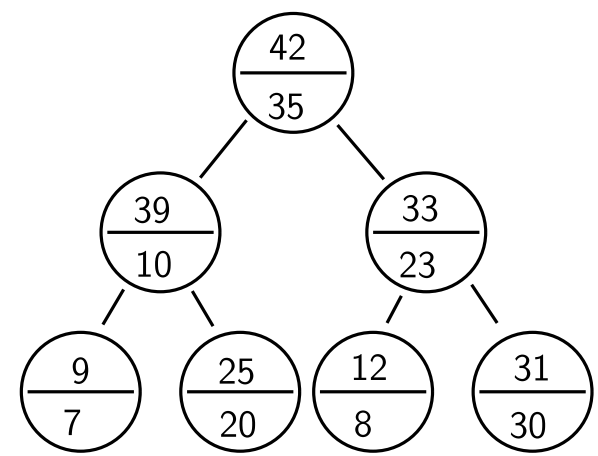


We could treat priority as 3rd dimension (allowing also max-priorities)

- size $S(n) = O(n)$
- construction time $P(n) = O(n \log n)$
- query time $Q(n) = O(n^{2/3} + k)$

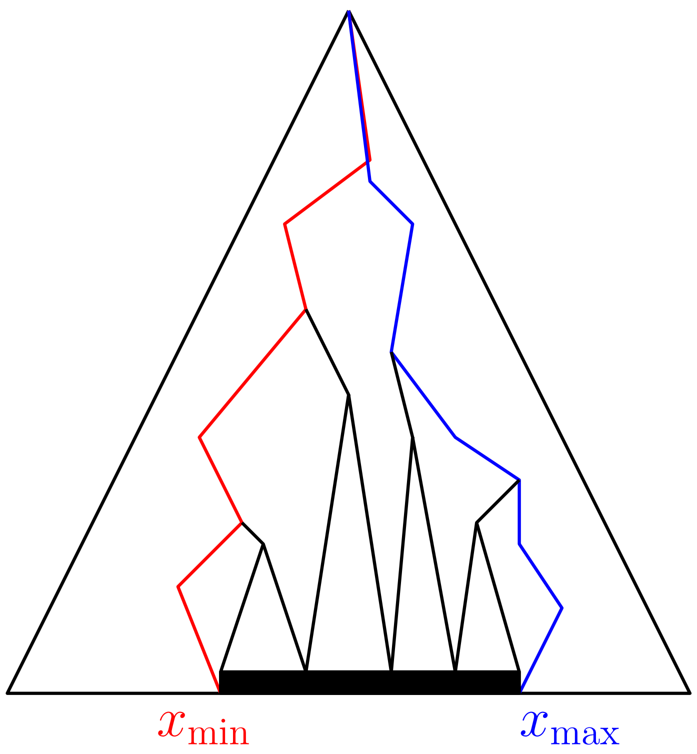
Better alternative:

- Augmented kd-prio tree:
  - before median extraction, extract max-priority point and store within node
- space $S(n) = O(n \log n)$
- construction time $P(n) = O(n \log n)$
- query time $Q(n) = O(\sqrt{n} + k)$
- abort subtree inspection as soon as priority is below $prio_{\min}$



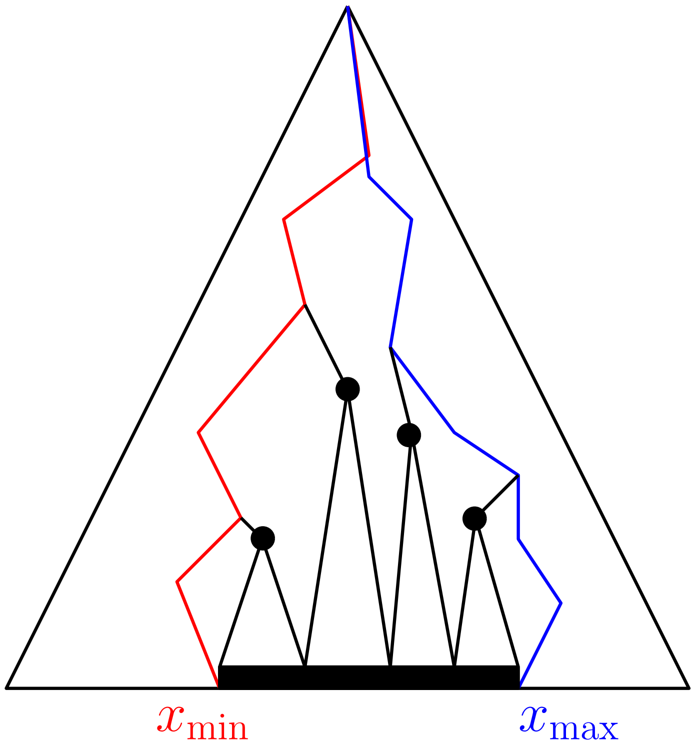
2nd Generalization Attempt: nested RangeTrees

- organize points in 1-dimensional search tree according x-coordinate
- for query $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$:
  - search for $x_{\min}$ and $x_{\max}$ in tree
  - observation: all points with matching x-coordinate 'enclosed' by search paths
- **store in each internal node all points of subtree in a tree on the y-coordinates**
- space $S(n) = O(n \log n)$
- query time $Q(n) = O(log^2 n + k)$



2nd Generalization Attempt: nested RangeTrees

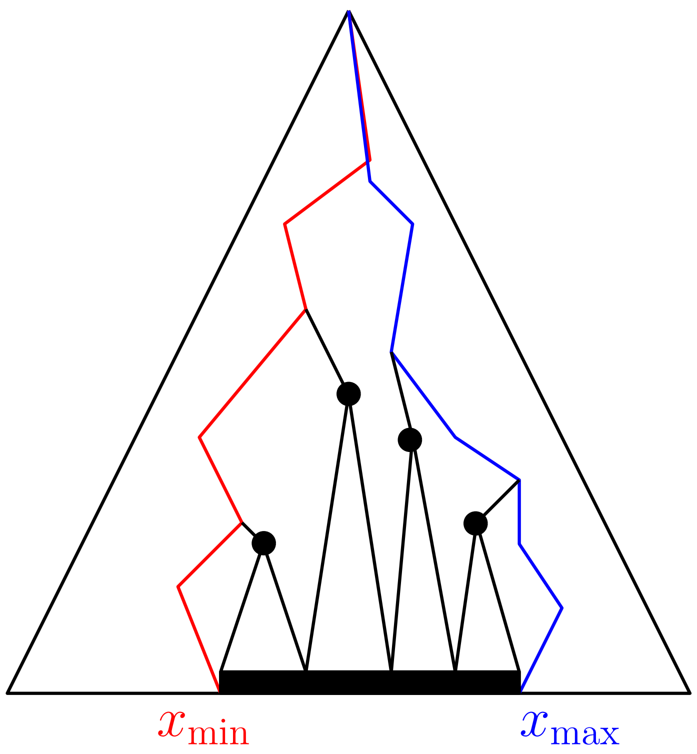
- organize points in 1-dimensional search tree according x-coordinate
- for query $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$:
  - search for $x_{\min}$ and $x_{\max}$ in tree
  - observation: all points with matching x-coordinate 'enclosed' by search paths
- **store in each internal node all points of subtree in a tree on the y-coordinates**
- space $S(n) = O(n \log n)$
- query time $Q(n) = O(log^2 n + k)$



(a $\log n$ factor can be shaved off the query time using the technique of *fractional cascading*)

2nd Generalization Attempt: nested RangeTrees

- organize points in 1-dimensional search tree according x-coordinate
- for query $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$:
  - search for $x_{\min}$ and $x_{\max}$ in tree
  - observation: all points with matching x-coordinate 'enclosed' by search paths
- **store in each internal node all points of subtree in a tree on the y-coordinates**
- space $S(n) = O(n \log n)$
- query time $Q(n) = O(log^2 n + k)$



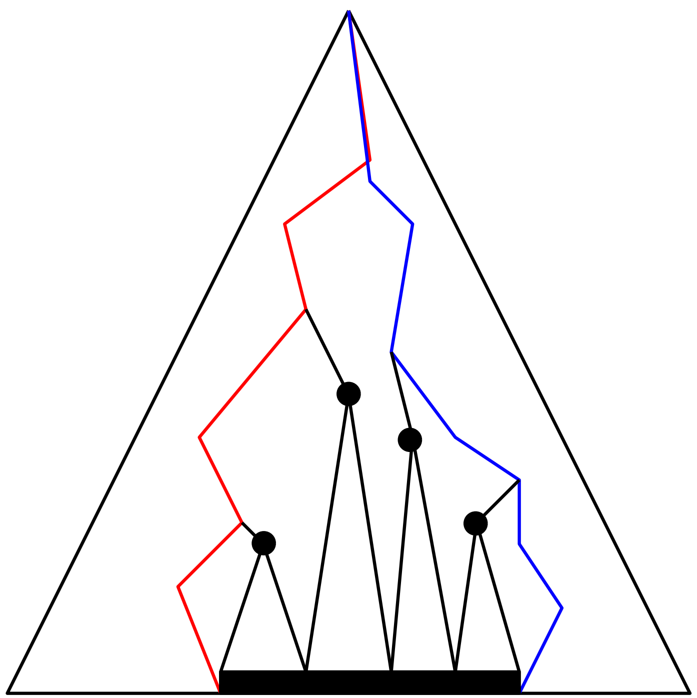
Priorities can be incorporated as 3rd dimension:

- $S(n) = O(n \log^2 n)$
- $Q(n) = O(log^3 n + k)$

(a $\log n$ factor can be shaved off the query time using the technique of *fractional cascading*)

2nd Generalization Attempt: nested RangeTrees

- organize points in 1-dimensional search tree according x-coordinate
- for query $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$:
  - search for $x_{\min}$ and $x_{\max}$ in tree
  - observation: all points with matching x-coordinate 'enclosed' by search paths
- **store in each internal node all points of subtree in a tree on the y-coordinates**
- space $S(n) = O(n \log n)$
- query time $Q(n) = O(log^2 n + k)$



Priorities can be incorporated as 3rd dimension:

- $S(n) = O(n \log^2 n)$
- $Q(n) = O(log^3 n + k)$

We can do better by using another secondary structure . . .

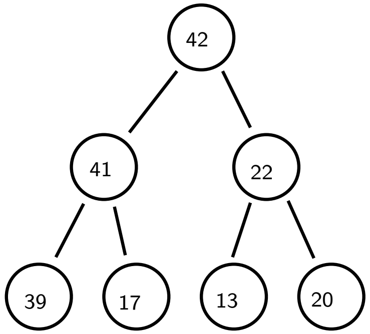
(a $\log n$ factor can be shaved off the query time using the technique of *fractional cascading*)

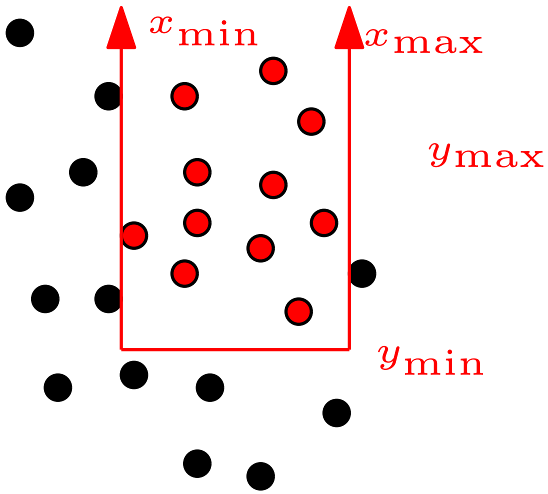
2nd improved Generalization Attempt: Range Tree+Treap

- A *treap* can answer queries of the type

$$[x_{\min}, x_{\max}] \times [y_{\min}, \infty]$$

- *treap*$\equiv$ *tree* and *heap* at the same time



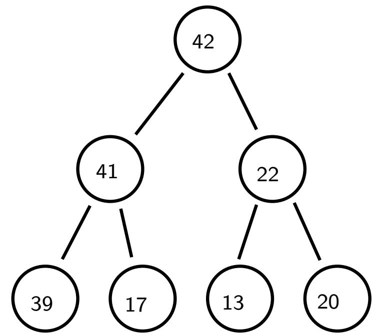


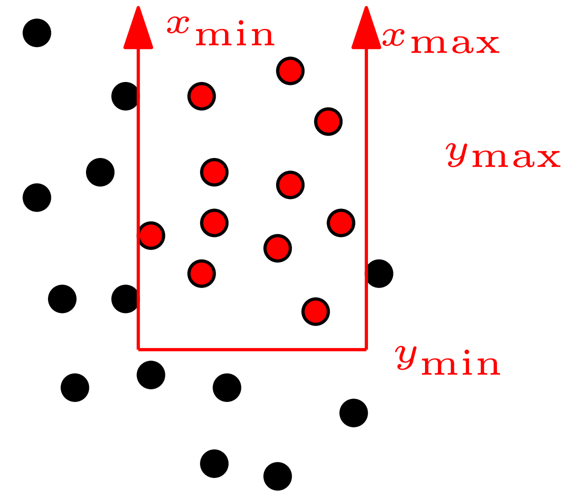
2nd improved Generalization Attempt: Range Tree+Treap

- A *treap* can answer queries of the type

$$[x_{\min}, x_{\max}] \times [y_{\min}, \infty]$$

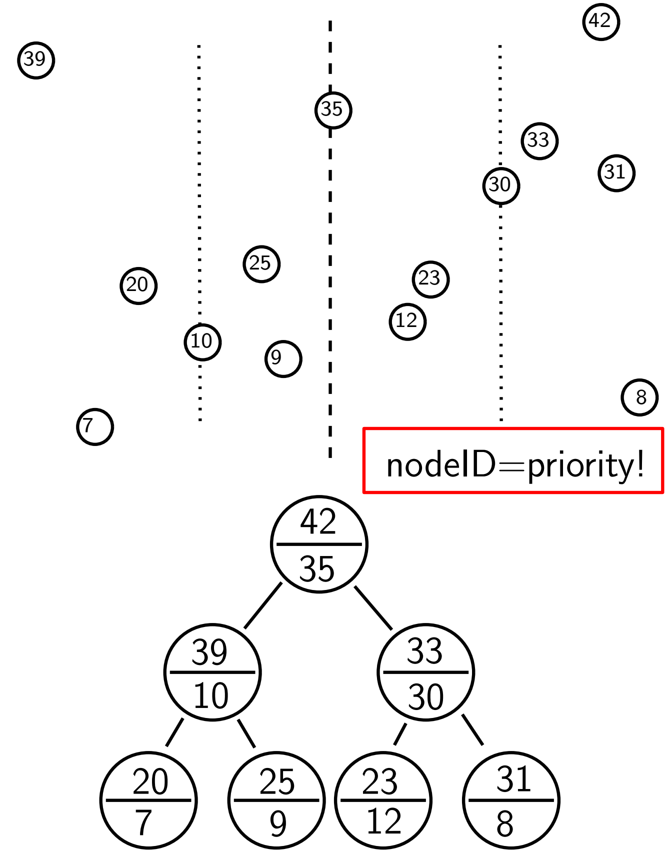
- *treap*$\equiv$ *tree* and *heap* at the same time





Construction:

- extract max-prio point and x-median point
- recursively construct left and right subtrees
- time $P(n) = O(n \log n)$
- space $S(n) = O(n)$

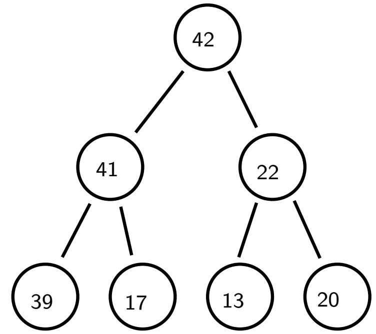


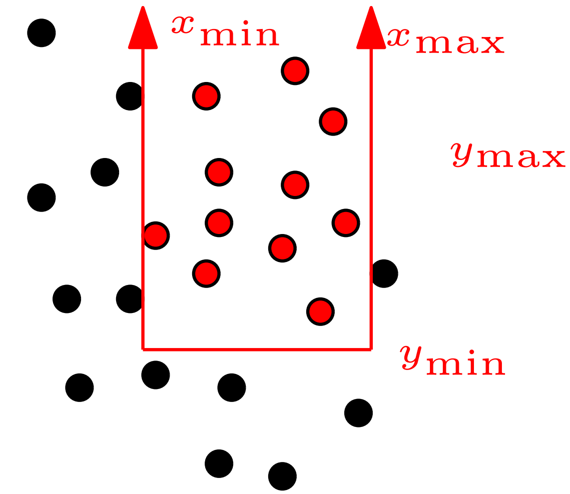
2nd improved Generalization Attempt: Range Tree+Treap

- A *treap* can answer queries of the type

$$[x_{\min}, x_{\max}] \times [y_{\min}, \infty]$$

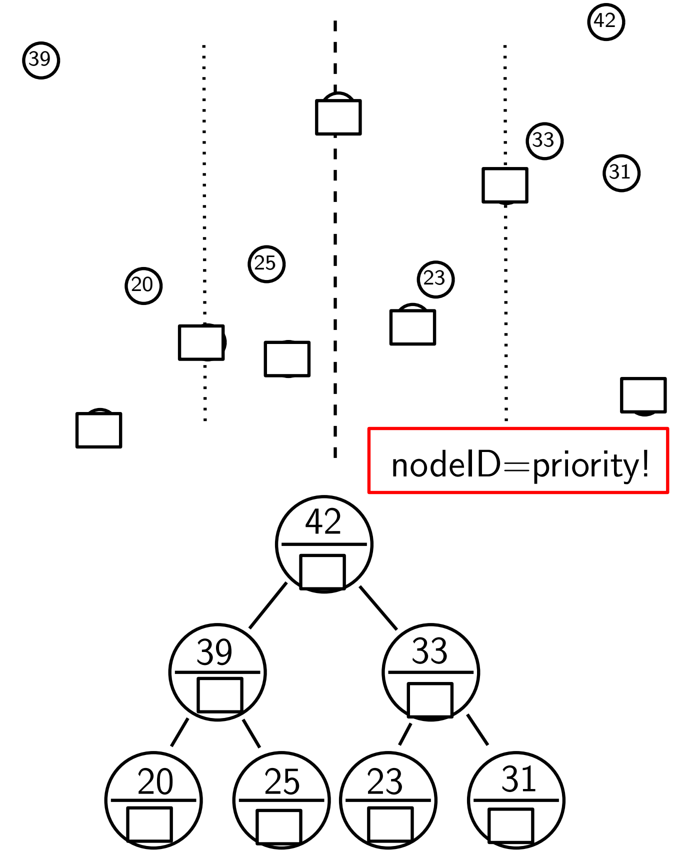
- *treap*$\equiv$ *tree* and *heap* at the same time





Construction:

- extract max-prio point and x-median point
- recursively construct left and right subtrees
- time $P(n) = O(n \log n)$
- space $S(n) = O(n)$
- ignoring median points, we have a heap!

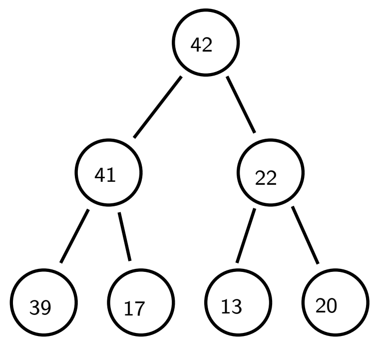


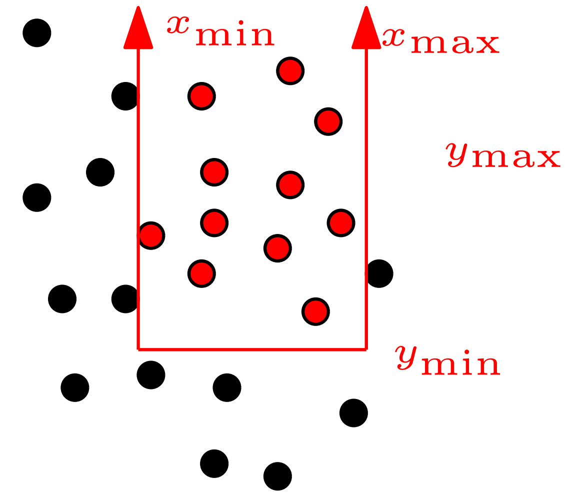
2nd improved Generalization Attempt: Range Tree+Treap

- A *treap* can answer queries of the type

$$[x_{\min}, x_{\max}] \times [y_{\min}, \infty]$$

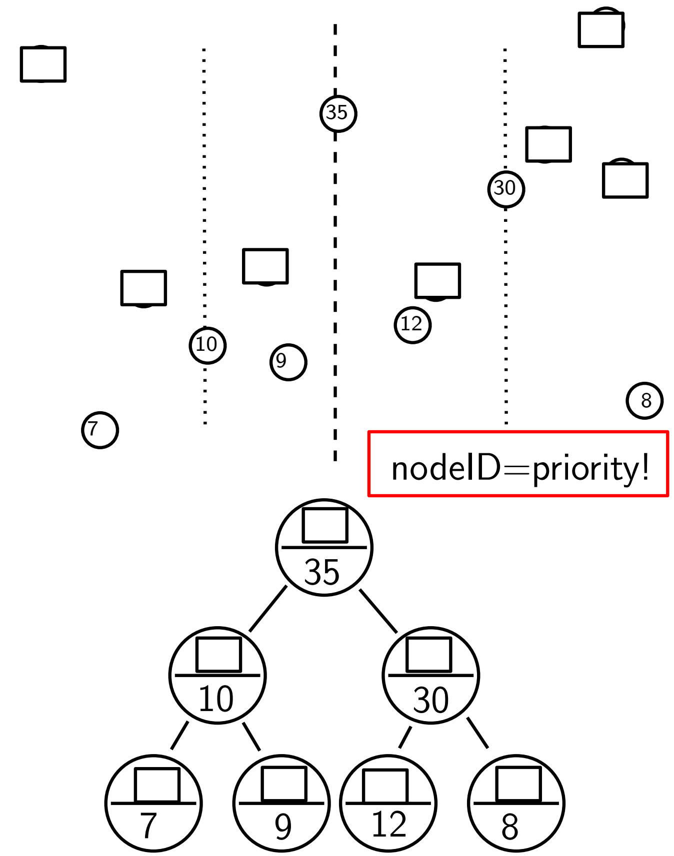
- *treap*$\equiv$ *tree* and *heap* at the same time





Construction:

- extract max-prio point and x-median point
- recursively construct left and right subtrees
- time $P(n) = O(n \log n)$
- space $S(n) = O(n)$
- ignoring prio points, we have a search tree!

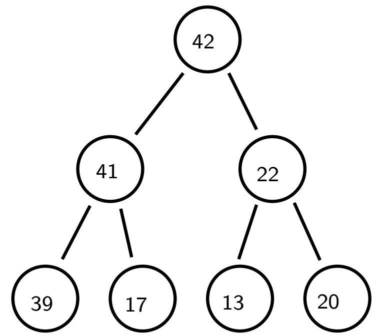


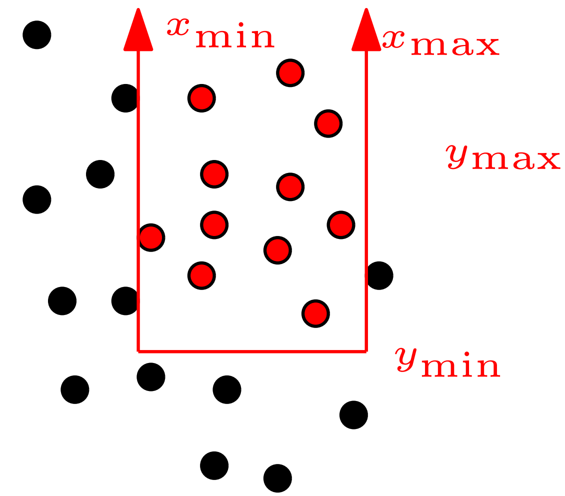
2nd improved Generalization Attempt: Range Tree+Treap

- A *treap* can answer queries of the type

$$[x_{\min}, x_{\max}] \times [y_{\min}, \infty]$$

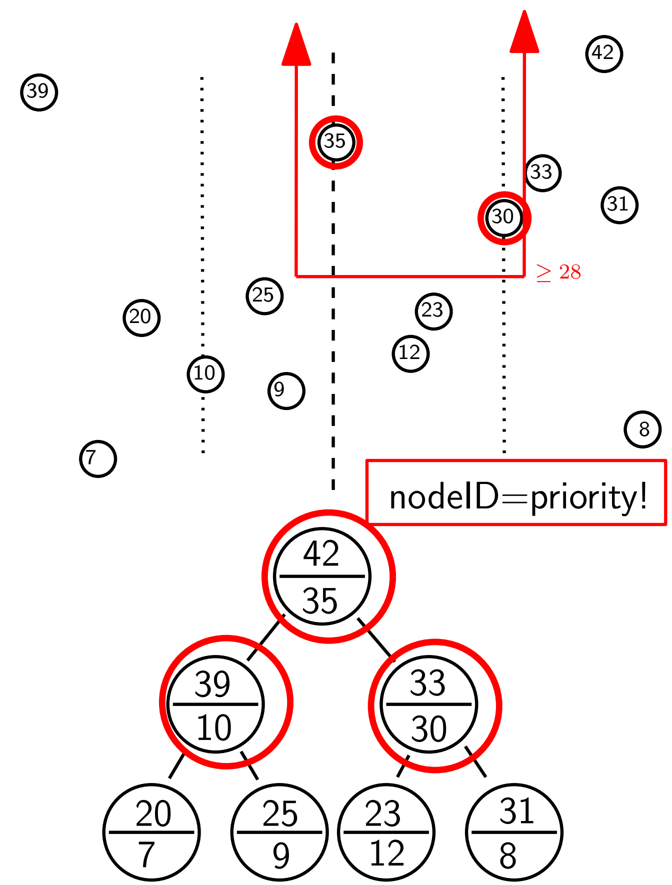
- *treap* $\equiv$ *tree* and *heap* at the same time





Construction:
- extract max-prio point and x-median point
- recursively construct left and right subtrees
- time $P(n) = O(n \log n)$
- space $S(n) = O(n)$

Query:
- . . . the obvious way

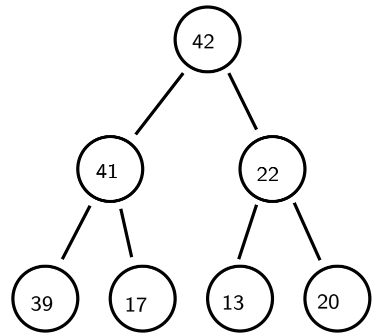


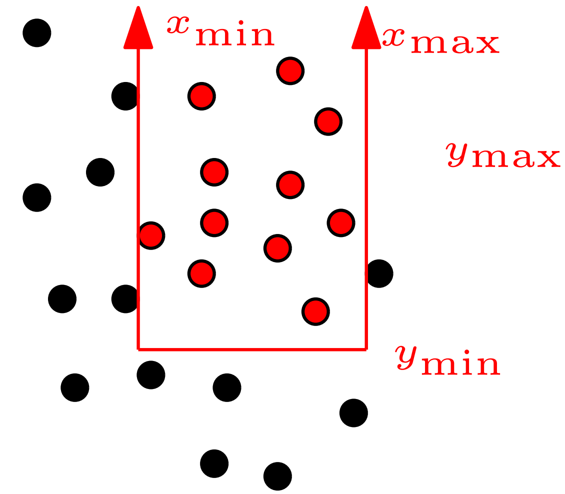
2nd improved Generalization Attempt: Range Tree+Treap

- A *treap* can answer queries of the type

$$[x_{\min}, x_{\max}] \times [y_{\min}, \infty]$$

- *treap*$\equiv$ *tree* and *heap* at the same time



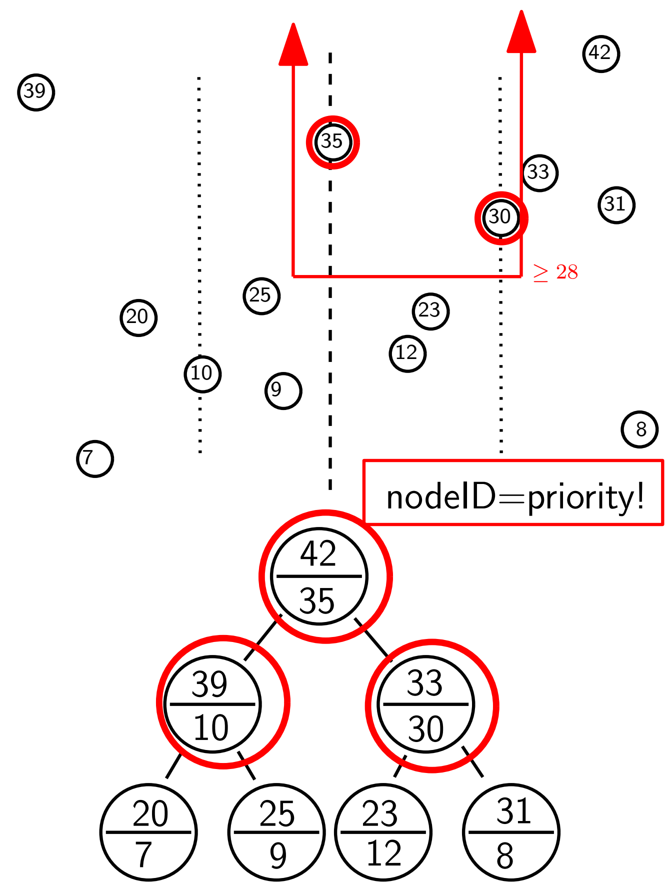


Construction:

- extract max-prio point and x-median point
- recursively construct left and right subtrees
- time $P(n) = O(n \log n)$
- space $S(n) = O(n)$

Query:

- ... the obvious way



Use treap as secondary structure in a 1-dimensional range tree:

- $S(n) = O(n \log n)$
- $Q(n) = O(log^2 n + k)$

Things to consider for 'Practical' Solutions on large Data Sets

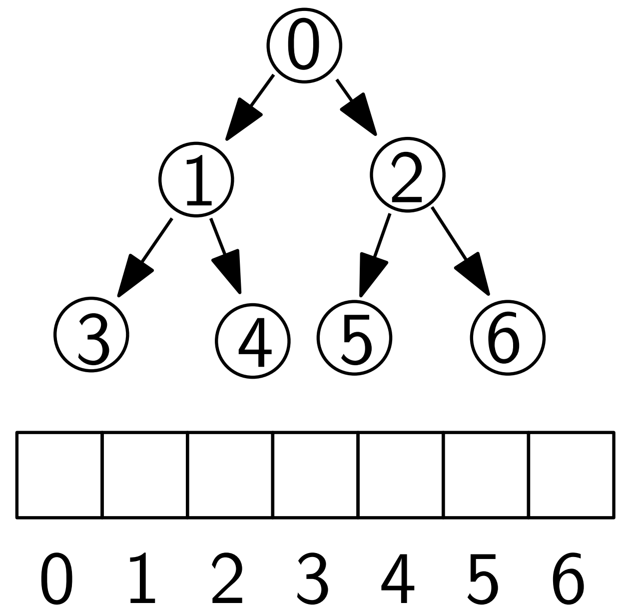
- use only space *linear* in the number of data items

Example:

- $n = 2^{29}$ (around 500 million)
- $n \log n = 29 \cdot n$, that is, we need 29 times more space than just to store the data!

Things to consider for 'Practical' Solutions on large Data Sets

- use only space *linear* in the number of data items
- try to avoid explicit storage of connectivity information



Example:

- data item has 10 bytes (lat/lon/prio)
- every node in the tree must have incoming pointer (8 Bytes)
- connectivity information almost *doubles* space
- Better: implicit connectivity e.g. via array representation:
  - leftchild(i)$=2 \cdot i + 1$
  - rightchild(i)$=2 \cdot i + 2$
  - parent(i)$=i/2$

Challenge:

- need 'almost complete' search tree
- requires more deliberate splitting choice than median

Things to consider for 'Practical' Solutions on large Data Sets

- use only space *linear* in the number of data items
- try to avoid explicit storage of connectivity information
- try to keep things in memory consecutively

More concrete:

- storing and scanning elements linearly in an array is surprisingly fast (cache effects)
- tree search issues rather dispersed memory accesses
- avoid many individual 'objects' allocated on the heap
- often a combination of a 'real' data structure with linear search very effective

A simple, yet practical solution: Grid + Linear Scan

Linear Scan:

- store (lat,lon,prio) consecutively in an array
- $Q(n) = \Theta(n)$ (always!)

A simple, yet practical solution: Grid + Linear Scan

Linear Scan:

- store (lat,lon,prio) consecutively in an array
- $Q(n) = \Theta(n)$ (always!)

Prioritized Linear Scan:

- store (lat,lon,prio) consecutively in an array **sorted according to prio**
- $Q(n) = \Theta(n')$ where n' is # of elements with high enough priority (overall)
- not bad if querying mainly large regions
- inefficient for very small regions

A simple, yet practical solution: Grid + Linear Scan

Linear Scan:

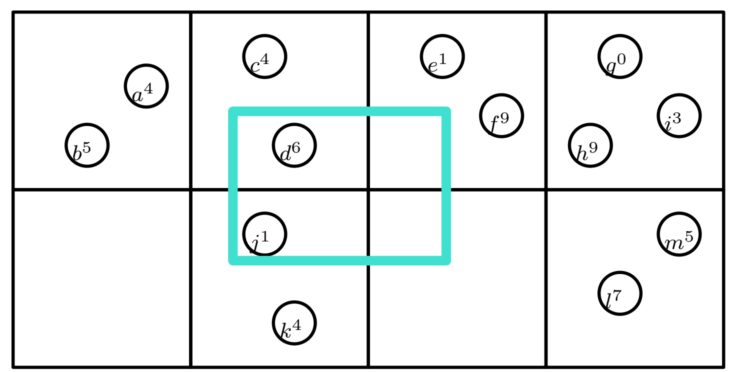
- store (lat,lon,prio) consecutively in an array
- $Q(n) = \Theta(n)$ (always!)

Prioritized Linear Scan:

- store (lat,lon,prio) consecutively in an array **sorted according to prio**
- $Q(n) = \Theta(n')$ where n' is # of elements with high enough priority (overall)
- not bad if querying mainly large regions
- inefficient for very small regions

Grid + Prioritized Linear Scans

- create grid
- store grid cells consecutively in memory
- within each grid cell store items sorted according to prio
- for query:
  - determine grid cells to be inspected
  - perform prioritized linear scan in each of them
- grid: limits effort if query region small
- prio sort: limits effort if min priority high
- parameter to choose: cell size
- good, if queries to be expected are somewhat uniform, otherwise kd-prio-tree might pay off



$$b^5 \quad a^4 \quad d^6 \quad c^4 \quad f^9 \quad e^1 \quad h^9 \quad i^3 \quad g^0 \quad k^4 \quad j^1 \quad l^7 \quad m^5$$

A simple, yet practical solution: Grid + Linear Scan

Linear Scan:

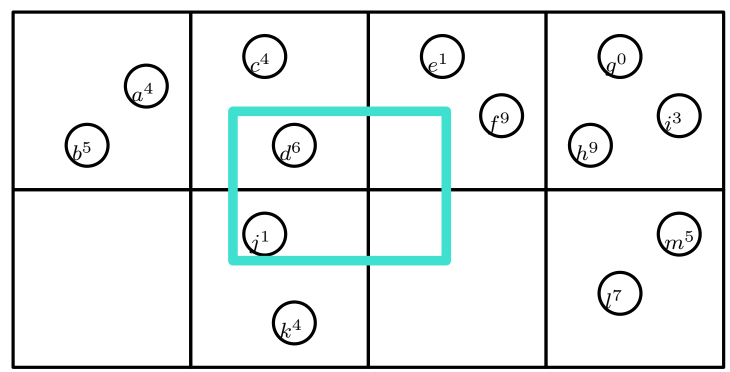
- store (lat,lon,prio) consecutively in an array
- $Q(n) = \Theta(n)$ (always!)

Prioritized Linear Scan:

- store (lat,lon,prio) consecutively in an array **sorted according to prio**
- $Q(n) = \Theta(n')$ where n' is # of elements with high enough priority (overall)
- not bad if querying mainly large regions
- inefficient for very small regions

Grid + Prioritized Linear Scans

- create grid
- store grid cells consecutively in memory
- within each grid cell store items sorted according to prio
- for query:
  - determine grid cells to be inspected
  - perform prioritized linear scan in each of them
- grid: limits effort if query region small
- prio sort: limits effort if min priority high
- parameter to choose: cell size
- good, if queries to be expected are somewhat uniform, otherwise kd-prio-tree might pay off



$$b^5 \quad a^4 \quad d^6 \quad c^4 \quad f^9 \quad e^1 \quad h^9 \quad i^3 \quad g^0 \quad k^4 \quad j^1 \quad l^7 \quad m^5$$

You will implement this in the exercises!

Thank you

... for your attention!

Questions?

Applications? ($\rightarrow$ also talk to Filip!)

See you in the exercise session . . .

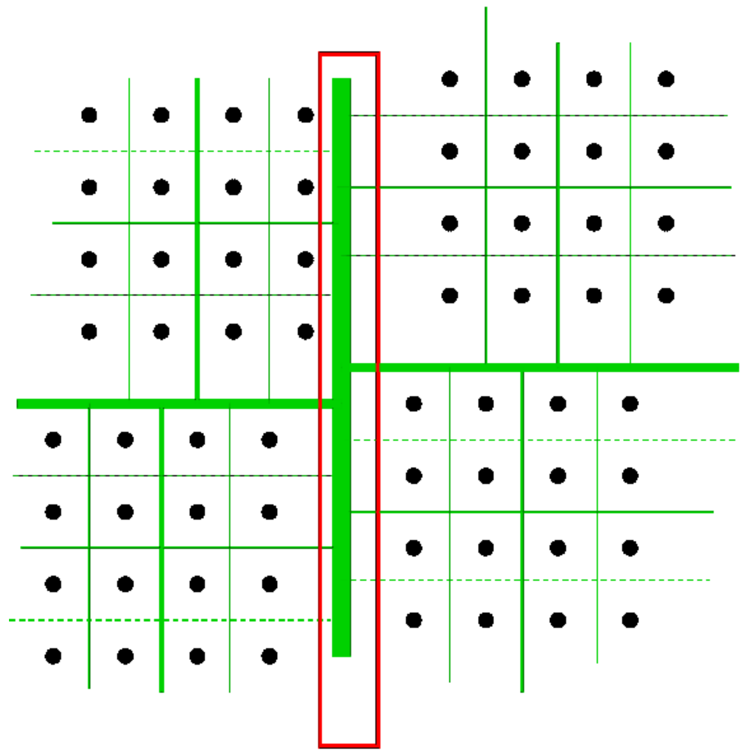
In Detail: Bounds for kd-Tree Query Time

How bad can kd-Tree
query time become?

In Detail: Bounds for kd-Tree Query Time

How bad can kd-Tree query time become?

. . . in fact pretty bad:



In this perturbed 8×8 grid we have $n = 64$ and the query box intersects $\Omega(\sqrt{n})$ many cells without reporting anything.

$$\Rightarrow S(n) = \Omega(\sqrt{n} + k)$$

In Detail: Bounds for kd-Tree Query Time

Can it become worse?

In Detail: Bounds for kd-Tree Query Time

Can it become worse?

Fortunately, NO!
We can prove a bound of

$$Q(n) = O(\sqrt{n} + k)$$

due to the following observations:

- rectangles fully contained in query are already accounted for in the output size k
- only need to count rectangles properly *intersected* by query rectangle
- even simpler: consider the number of rectangles intersected by a vertical line

In Detail: Bounds for kd-Tree Query Time

Can it become worse?

Fortunately, NO!
We can prove a bound of

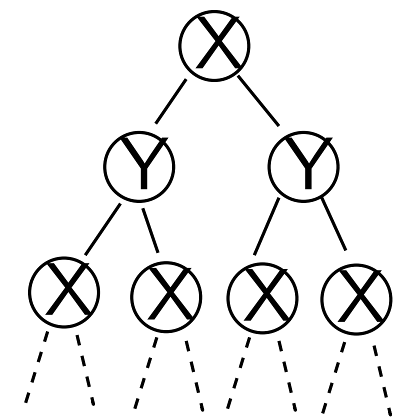
$$Q(n) = O(\sqrt{n} + k)$$

due to the following observations:

- rectangles fully contained in query are already accounted for in the output size k
- only need to count rectangles properly *intersected* by query rectangle
- even simpler: consider the number of rectangles intersected by a vertical line

How many cells can be intersected by one vertical line?

- assume all nodes on even levels correspond to an X-split, on odd levels to an Y-split; root has level 0
- let $T(n)$ be the number of cells intersected below an X-split node with n nodes in its subtree



The following holds:

- $T(n) = 1 + 1 + 2 \cdot T(n/4)$
- $T(1) = 1$

In Detail: Bounds for kd-Tree Query Time

Can it become worse?

Fortunately, NO!
We can prove a bound of

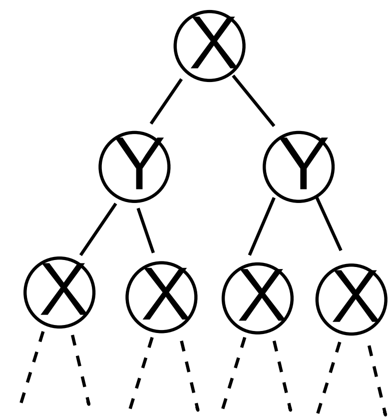
$$Q(n) = O(\sqrt{n} + k)$$

due to the following observations:

- rectangles fully contained in query are already accounted for in the output size k
- only need to count rectangles properly *intersected* by query rectangle
- even simpler: consider the number of rectangles intersected by a vertical line

How many cells can be intersected by one vertical line?

- assume all nodes on even levels correspond to an X-split, on odd levels to an Y-split; root has level 0
- let $T(n)$ be the number of cells intersected below an X-split node with n nodes in its subtree



The following holds:
- $T(n) = 1 + 1 + 2 \cdot T(n/4)$
- $T(1) = 1$

This yields the following sum with $(\log n)/2$ summands:

$$2 + 4 + 8 + \cdots = \sum_{i=1}^{(\log n)/2} 2^i$$

$$\leq 2^{(\log n)/2+1} = 2 \cdot \sqrt{n}$$

In Detail: Bounds for kd-Tree Query Time

Can it become worse?

Fortunately, NO!
We can prove a bound of

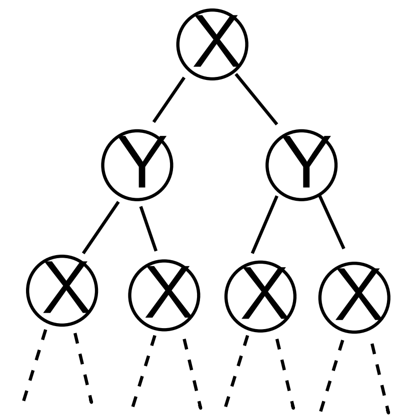
$$Q(n) = O(\sqrt{n} + k)$$

due to the following observations:

- rectangles fully contained in query are already accounted for in the output size k
- only need to count rectangles properly *intersected* by query rectangle
- even simpler: consider the number of rectangles intersected by a vertical line

How many cells can be intersected by one vertical line?

- assume all nodes on even levels correspond to an X-split, on odd levels to an Y-split; root has level 0
- let $T(n)$ be the number of cells intersected below an X-split node with n nodes in its subtree



The following holds:
- $T(n) = 1 + 1 + 2 \cdot T(n/4)$
- $T(1) = 1$

This yields the following sum with $(\log n)/2$ summands:

$$2 + 4 + 8 + \cdots = \sum_{i=1}^{(\log n)/2} 2^i$$

$$\leq 2^{(\log n)/2+1} = 2 \cdot \sqrt{n}$$

$$\boxed{\Rightarrow Q(n) = O(\sqrt{n} + k)}$$