

# Range searching and kd-trees

## Computational Geometry

### Lecture 7: Range searching and kd-trees

# Databases

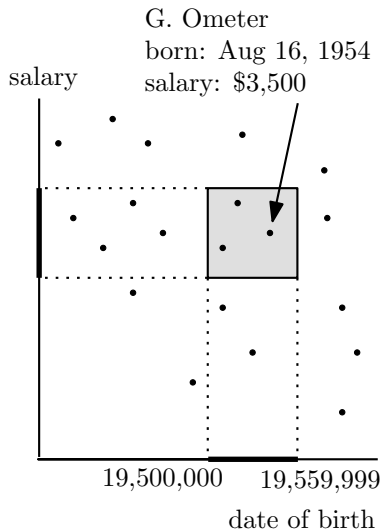
Databases store records or objects

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

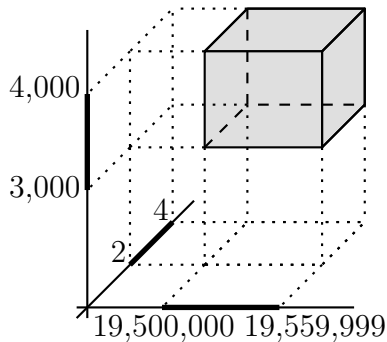
**Exact match query:** Asks for the objects whose coordinates match query coordinates exactly

**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 reported fast

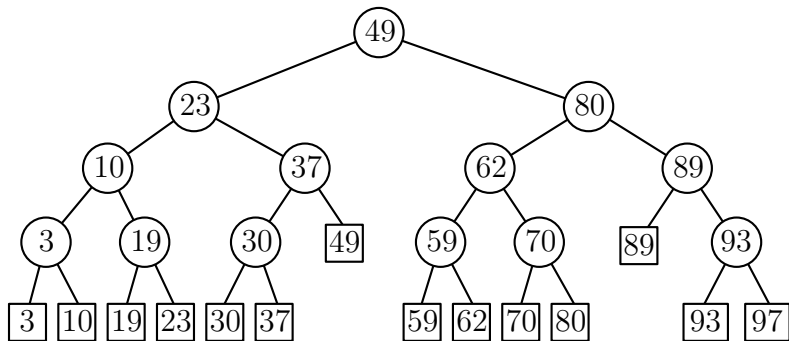
The points  $p_1, \dots, p_n$  are known beforehand, the query  $[x, x']$  only later

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

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

# Balanced binary search trees

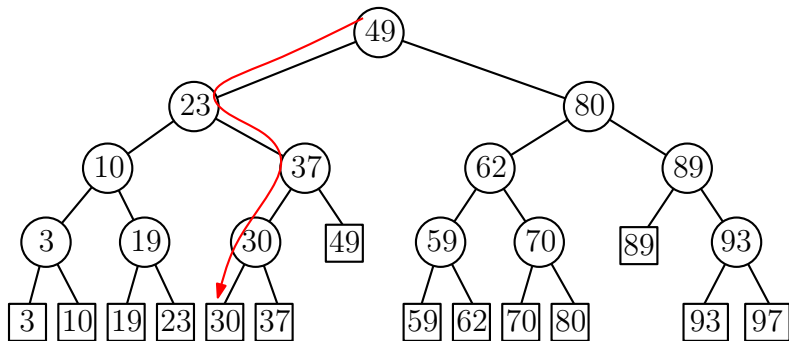
A balanced binary search tree with the points in the leaves





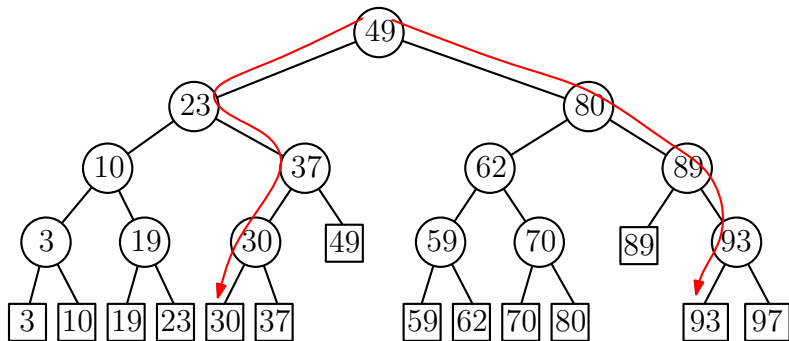
## Balanced binary search trees

The search path for 25

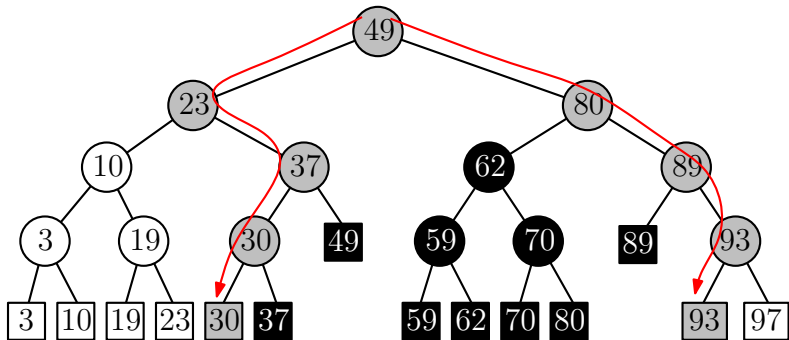


# Balanced binary search trees

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?

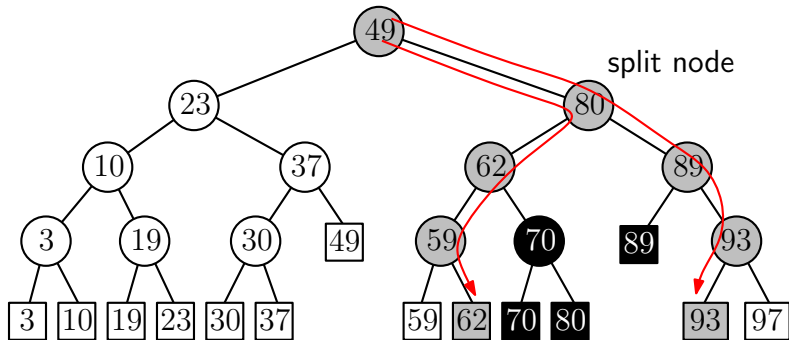
## Node types for a query

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** 1DRANGEQUERY( $\mathcal{T}, [x : x']$ )

1.  $v_{\text{split}} \leftarrow \text{FINDSPLITNODE}(\mathcal{T}, x, x')$
2. **if**  $v_{\text{split}}$  is a leaf
3.     **then** Check if the point in  $v_{\text{split}}$  must be reported.
4.     **else**  $v \leftarrow lc(v_{\text{split}})$
5.         **while**  $v$  is not a leaf
6.             **do if**  $x \leq x_v$
7.                 **then** REPORTSUBTREE( $rc(v)$ )
8.                      $v \leftarrow lc(v)$
9.                 **else**  $v \leftarrow rc(v)$
10.     Check if the point stored in  $v$  must be reported.
11.      $v \leftarrow rc(v_{\text{split}})$
12.     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

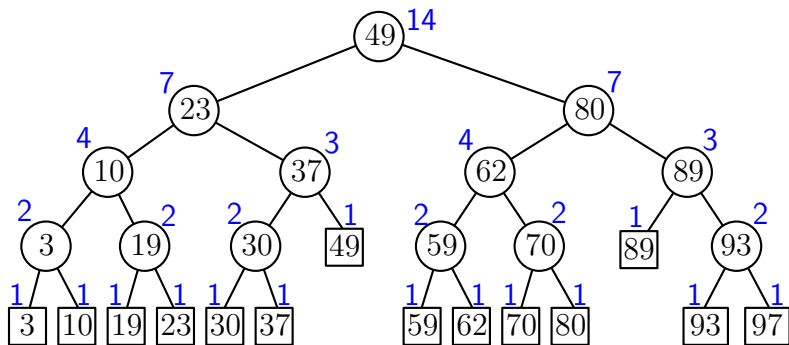
A balanced binary search tree storing  $n$  points can be built in  $O(n)$  time after sorting, so in  $O(n \log n)$  time overall (or by repeated insertion in  $O(n \log n)$  time)

# Result

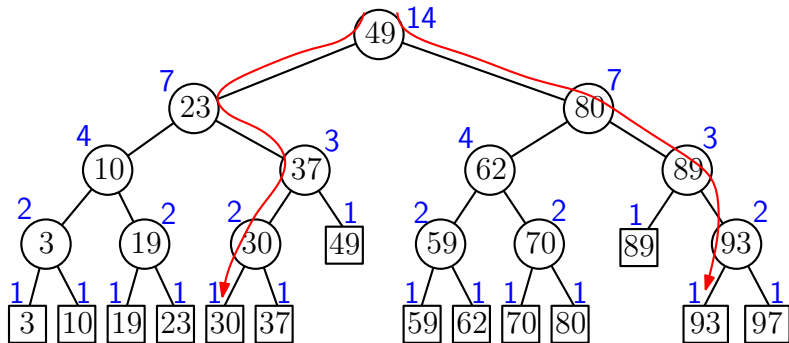
**Theorem:** A set of  $n$  points on the real line can be preprocessed in  $O(n \log n)$  time into a data structure of  $O(n)$  size so that any 1D range query can be answered in  $O(\log n + k)$  time, where  $k$  is the number of answers reported

## Example 1D range counting query

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



## Example 1D range counting query

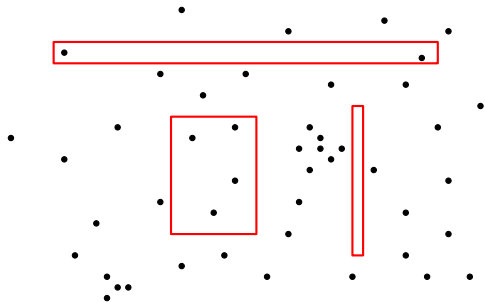
A 1-dimensional range counting query with  $[25, 90]$ 

# 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

**Note:** The number of points does not influence the output size so it should not show up in the query 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

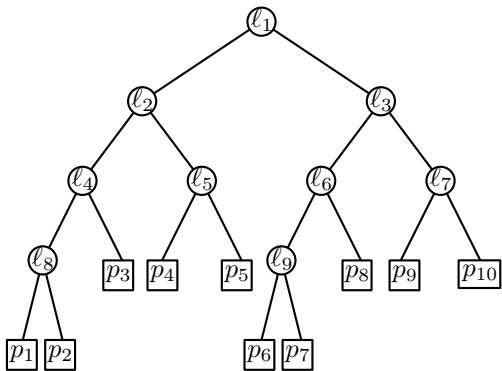
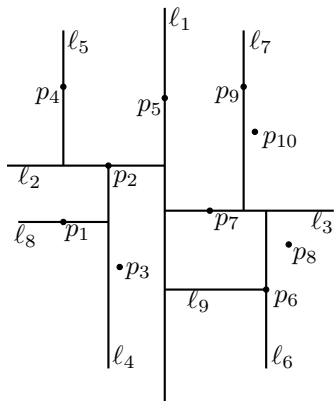
# Kd-trees

**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



# Kd-tree construction

**Algorithm** BUILDKDTREE( $P, depth$ )

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  $\ell$  through the median  $x$ -coordinate into  $P_1$  (left of or on  $\ell$ ) and  $P_2$  (right of  $\ell$ )
5.         **else** Split  $P$  with a horizontal line  $\ell$  through the median  $y$ -coordinate into  $P_1$  (below or on  $\ell$ ) and  $P_2$  (above  $\ell$ )
6.          $v_{\text{left}} \leftarrow \text{BUILDKDTREE}(P_1, depth + 1)$
7.          $v_{\text{right}} \leftarrow \text{BUILDKDTREE}(P_2, depth + 1)$
8.         Create a node  $v$  storing  $\ell$ , make  $v_{\text{left}}$  the left child of  $v$ , and make  $v_{\text{right}}$  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

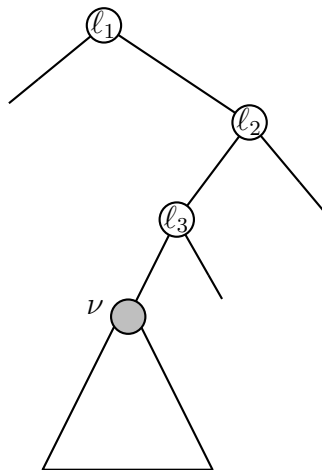
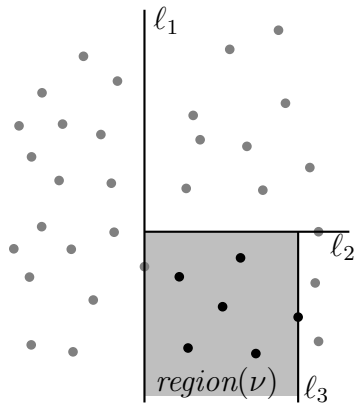
$$T(1) = O(1)$$

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

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

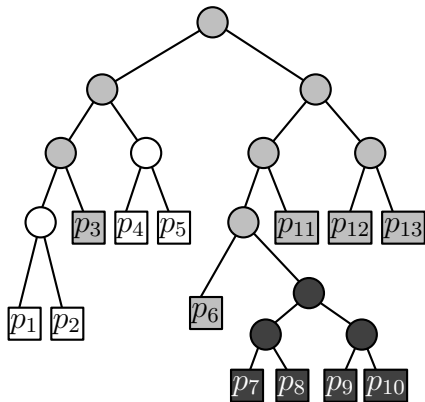
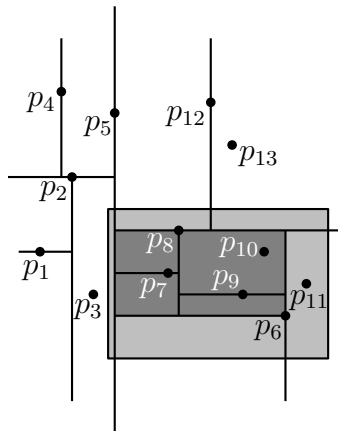
How do we know  $region(v)$  when we are at a node  $v$ ?

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** 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.

1. **if**  $v$  is a leaf
2.     **then** Report the point stored at  $v$  if it lies in  $R$
3.     **else if**  $region(lc(v))$  is fully contained in  $R$
4.         **then** REPORTSUBTREE( $lc(v)$ )
5.         **else if**  $region(lc(v))$  intersects  $R$
6.             **then** SEARCHKDTREE( $lc(v), R$ )
7.     **if**  $region(rc(v))$  is fully contained in  $R$
8.         **then** REPORTSUBTREE( $rc(v)$ )
9.         **else if**  $region(rc(v))$  intersects  $R$
10.             **then** SEARCHKDTREE( $rc(v), R$ )

# Kd-tree querying

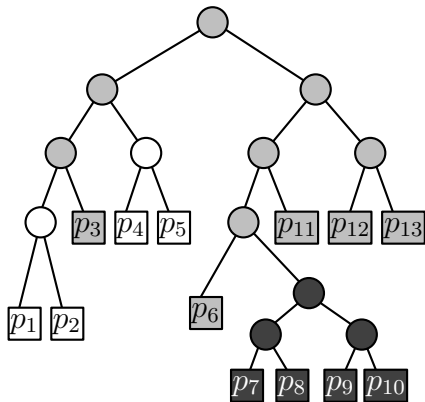
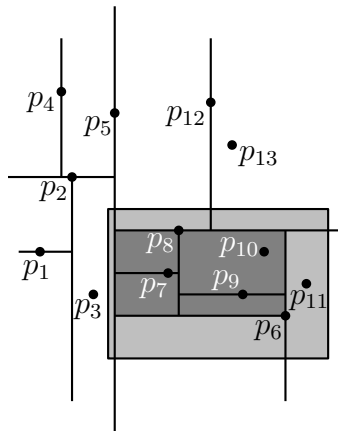
**Question:** How about a range *counting* query?  
How should the code be adapted?

# 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

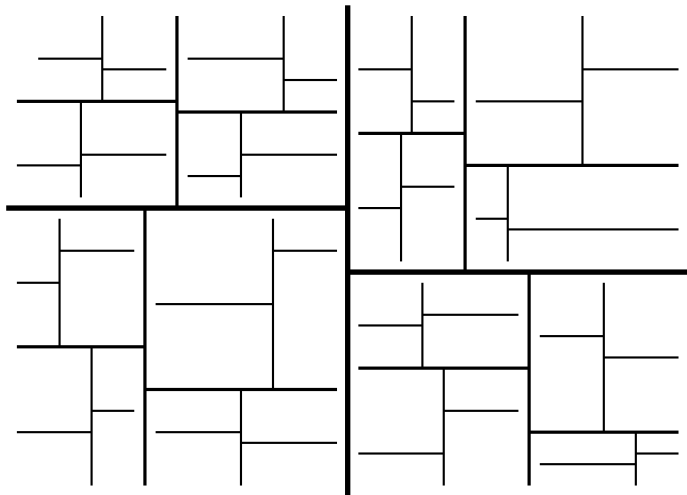


# Kd-tree query time analysis

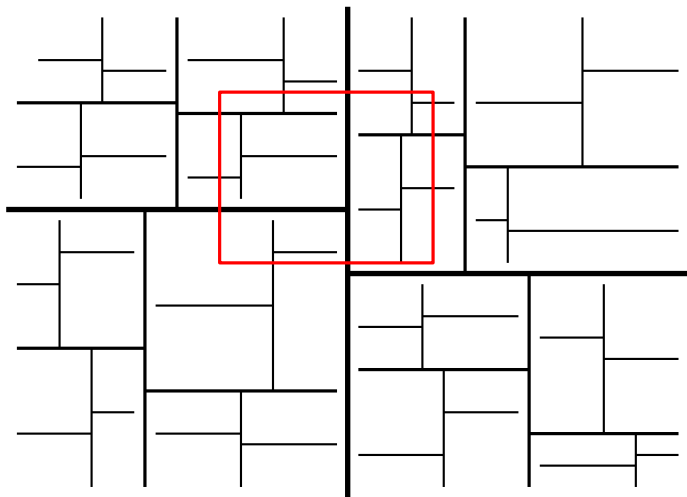
White, grey, and black nodes with respect to  $region(\mathbf{v})$ :

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

# Kd-tree query time analysis

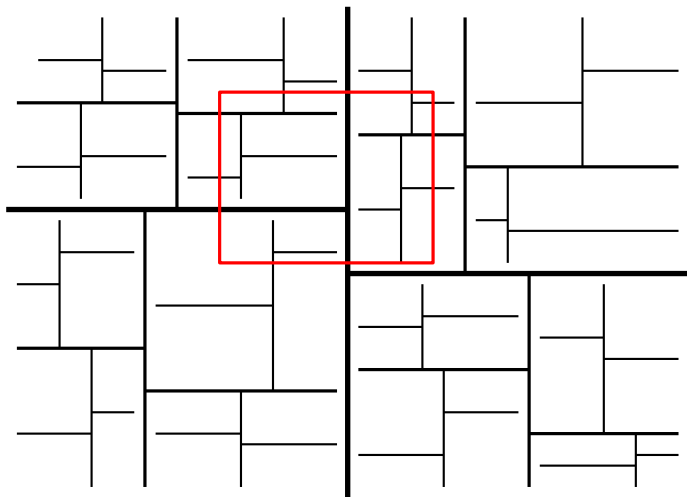


## Kd-tree query time analysis



**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

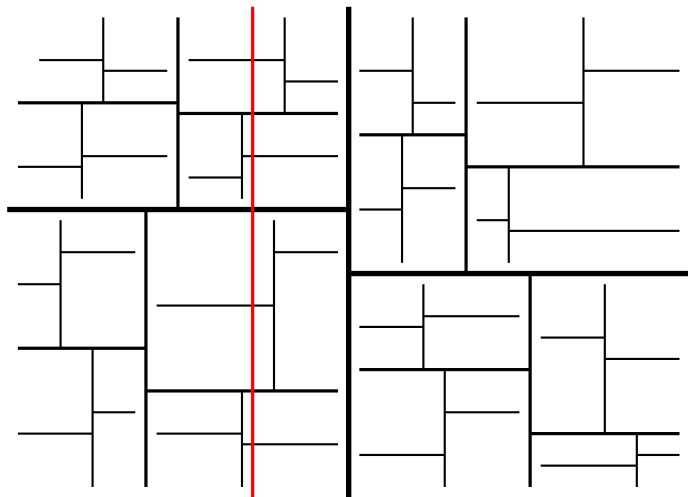
Grey node  $v$ :  $R$  intersects  $region(v)$ , but  $region(v) \not\subseteq R$

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  $\ell$

## 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,  $\ell$  is only to one side, while at every horizontal split  $\ell$  is to both sides

Let  $G(n)$  be the number of grey nodes in a kd-tree with  $n$  points (leaves). Then  $G(1) = 1$  and:

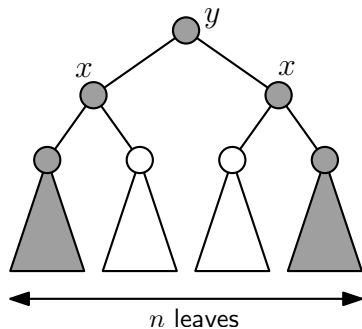
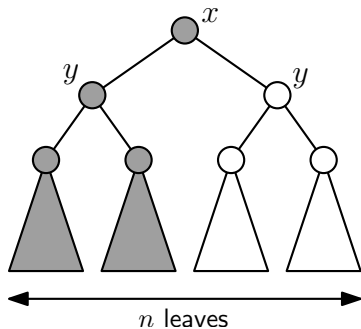
If a subtree has  $n$  leaves:  $G(n) = 1 + G(n/2)$  at even depth

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

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

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

## Kd-tree query time analysis



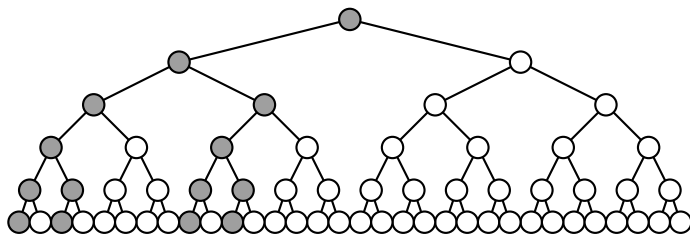
# Kd-tree query time analysis

$$G(1) = 1$$

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

**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  $\frac{1}{2} \cdot \log n$   
Important: The logarithm is base-2

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), so there are at most twice as many unary nodes as binary nodes, plus 1

## Kd-tree query time analysis

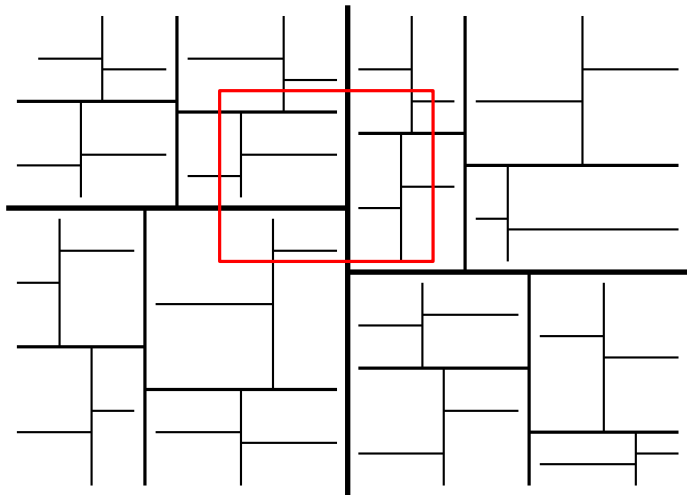
The number of grey nodes if the query were a vertical line is  $O(\sqrt{n})$

The same is true if the query were a horizontal line

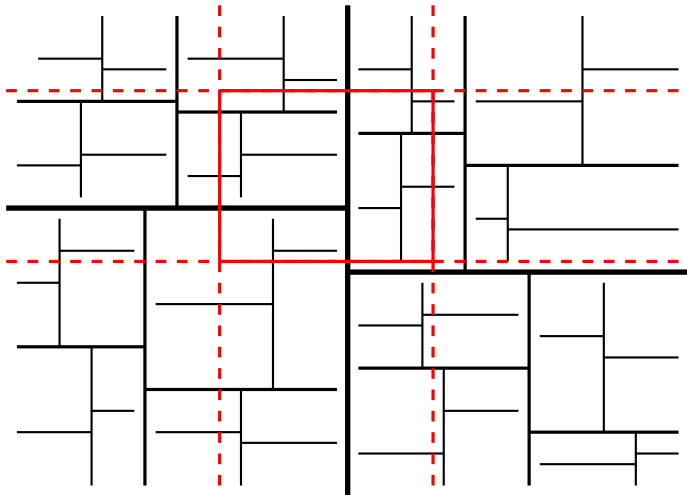
How about a query rectangle?



# 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 at most  $4 \cdot O(\sqrt{n}) = O(\sqrt{n})$  !

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

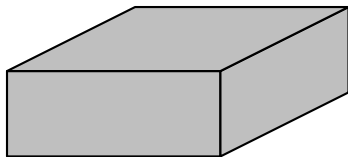
## Efficiency

$n$	$\log n$	$\sqrt{n}$
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

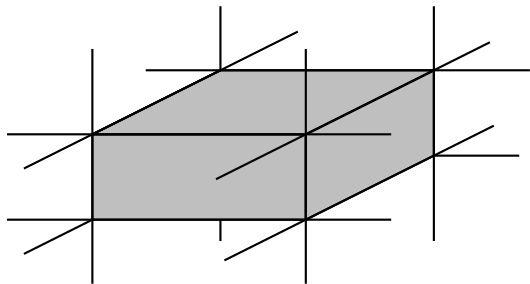
The construction of a 3D kd-tree is a trivial adaptation of the 2D version

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

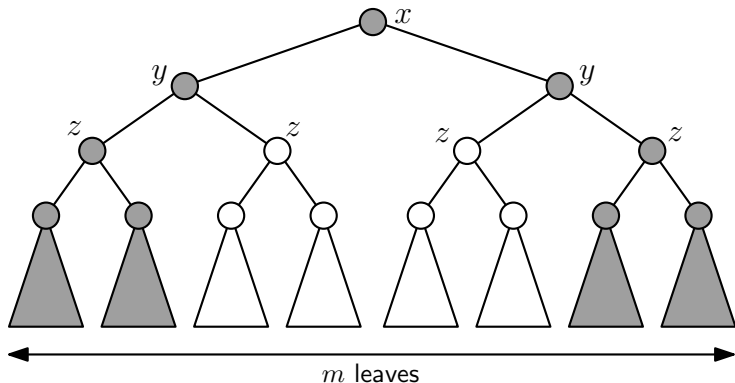
How does the query time analysis change?



Intersection of  $B$  and  $region(v)$  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  $O(n^{1-1/d} + k)$  time, where  $k$  is the number of answers reported

