

# Orthogonal range searching

## *Lecture 4*

Antoine Vigneron

antoine@jouy.inra.fr

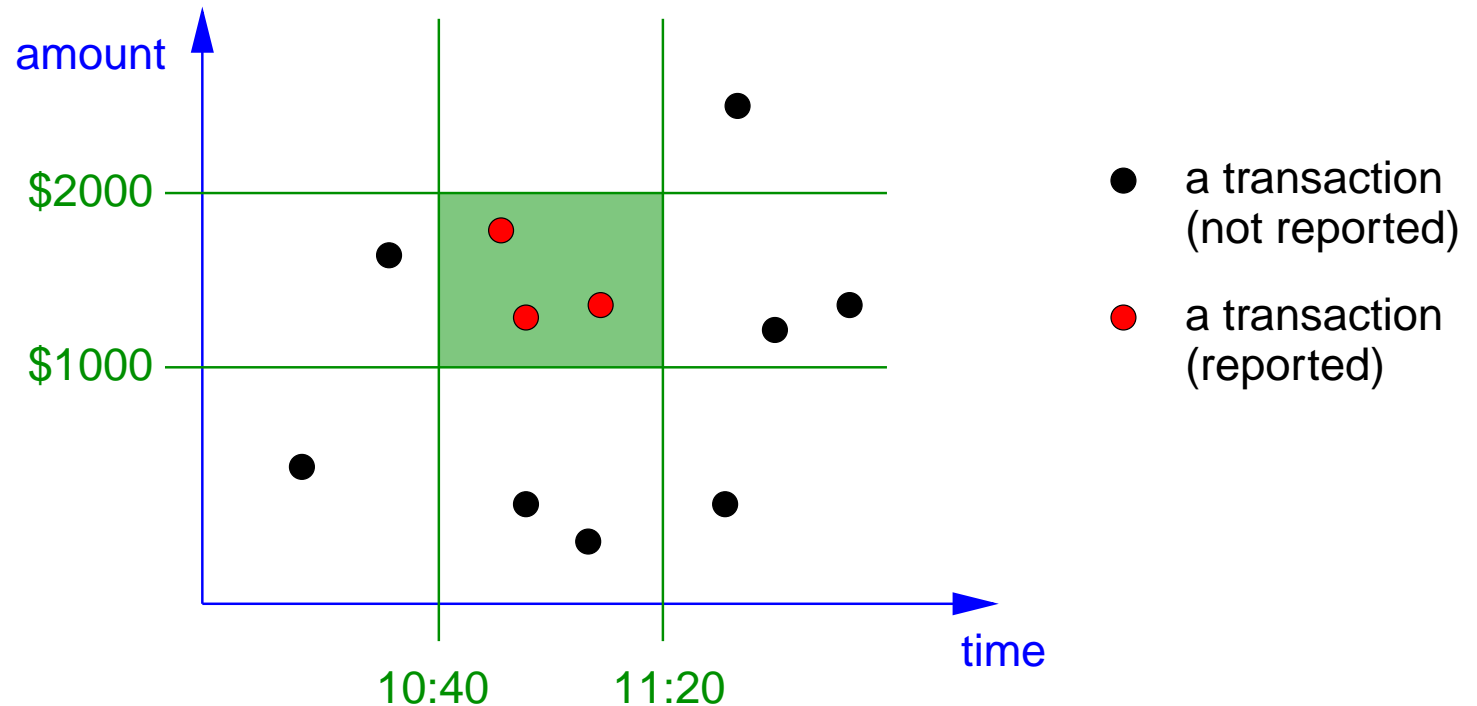
INRA

# Outline

- reference
  - textbook chapter 5
  - D. Mount Lectures 11 and 12
- problem: querying a database
- solution in one dimension
- data structure in  $\mathbb{R}^2$ : range trees
- extension to higher dimensions
- $\log n$  factor improvement

# Example

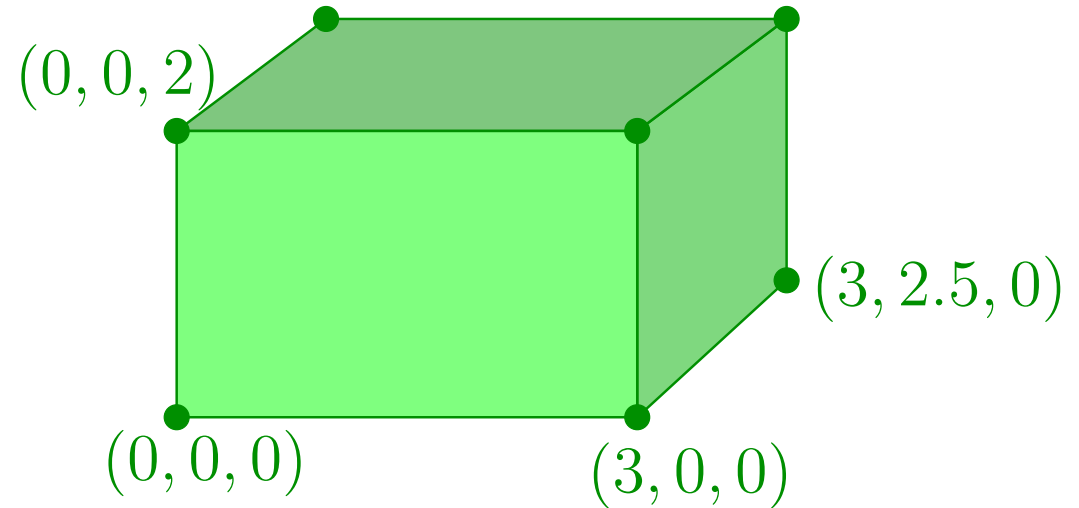
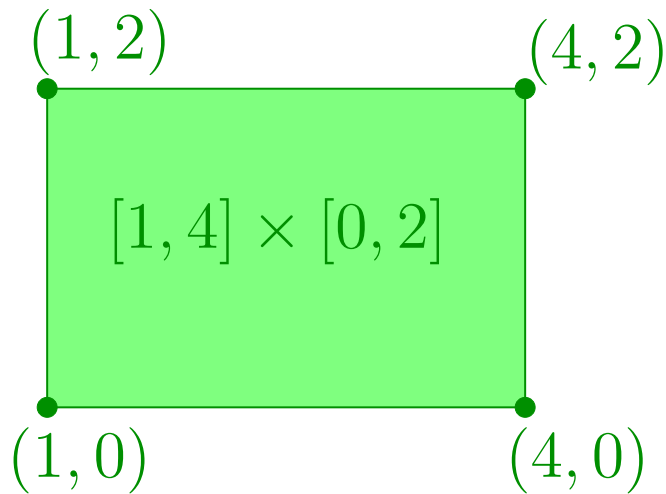
- a database in a bank records transactions
- a query: find all the transactions such that
  - the amount is between \$ 1000 and \$ 2000
  - it happened between 10:40am and 11:20am
- geometric interpretation



# Query problems

- assume  $n$  is the total number of transactions in the database
- we will show how to build a data structure in  $O(n \log n)$  time that allows to perform this type of queries in  $O(k + \log n)$  time where  $k$  is the size of the output (the number of transactions that are reported)
- the data structure is built only once, then a large number of queries can be answered quickly
- $O(n \log n)$  is the *preprocessing time*
- $O(k + \log n)$  is the *query time*

# Boxes



- a 2d–box
- also known as rectangle
- parallel to coordinate axis

- the 3d–box  
 $[0, 3] \times [0, 2.5] \times [0, 2]$
- generalize to any dimension
- algorithmic problems with boxes are relatively easy

# Problem statement

- let  $P$  be a set of  $n$  points in  $\mathbb{R}^d$
- we assume  $d = O(1)$
- preprocess  $P$  so as to answer queries of the type
  - input:  $(a_1, b_1, a_2, b_2, \dots, a_d, b_d)$
  - output:  $P \cap ([a_1, b_1] \times [a_2, b_2] \dots \times [a_d, b_d])$
- we denote  $k = |P \cap ([a_1, b_1] \times [a_2, b_2] \dots \times [a_d, b_d])|$

# One dimensional case ( $d=1$ )

# Problem

- $P$  is a set of real numbers
- queries: find all the points in  $P$  that are between  $a$  and  $b$
- data structure:
  - Balanced Binary Search Tree
  - see CS1102 or CS3230
  - preprocessing time:  $\Theta(n \log n)$  time to build a BBST
  - space usage:  $\Theta(n)$
- query time:  $\Theta(k + \log n)$  time. How?

# Answering a query

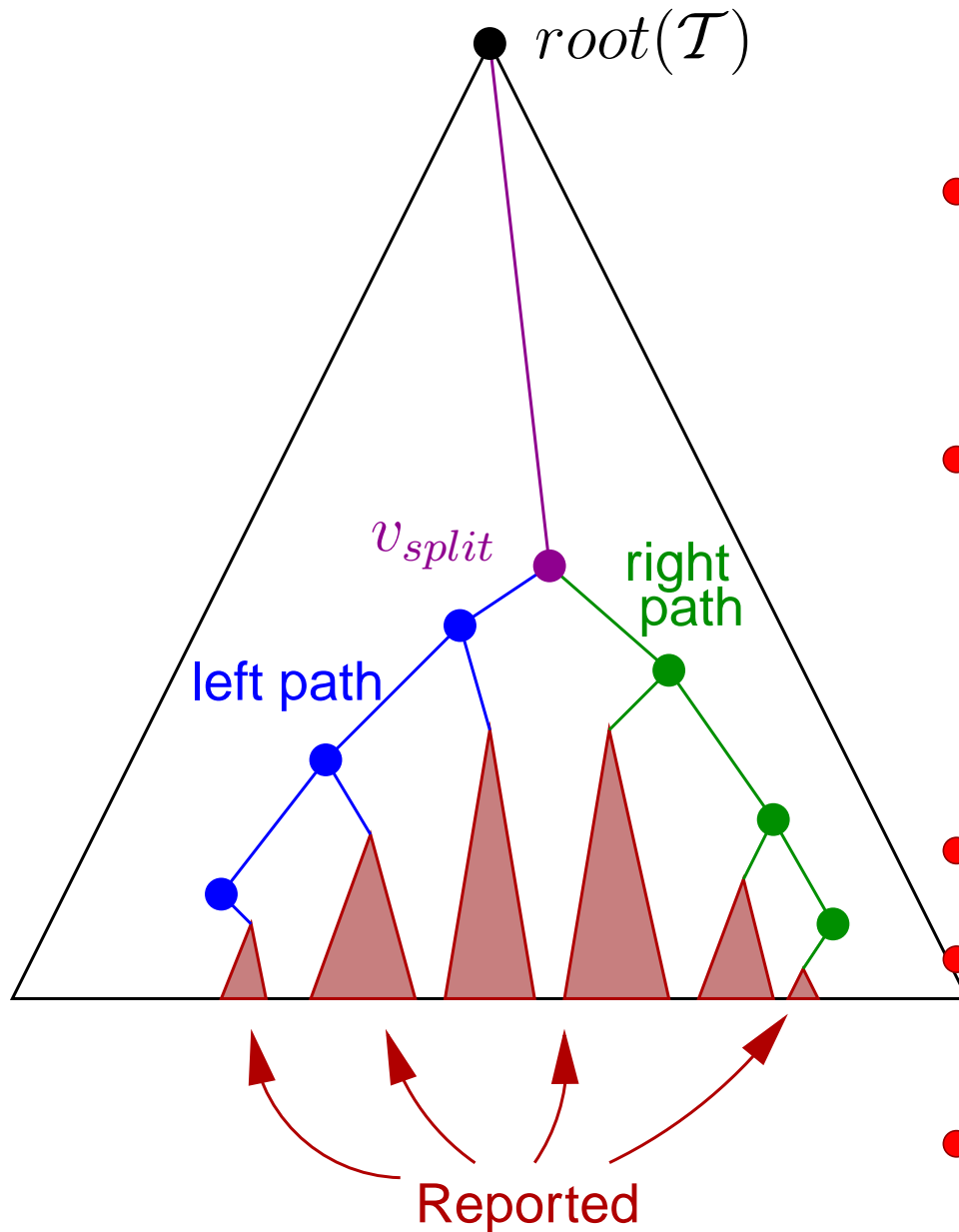
**Algorithm**  $Report(\mathcal{T}, a, b)$

**Input:** a BBST  $\mathcal{T}$  storing  $P$ , an interval  $[a, b]$

**Output:**  $P \cap [a, b]$

1. **if**  $\mathcal{T} = NULL$
2.     **then return**
3.  $x \leftarrow$  value stored at the root of  $\mathcal{T}$
4. **if**  $a < x$
5.     **then**  $Report(\mathcal{T}.left, a, b)$
6. **if**  $a \leq x \leq b$
7.     **then output**  $x$
8. **if**  $x < b$
9.     **then**  $Report(\mathcal{T}.right, a, b)$

# Analysis



- report left path, right path,  $v_{split}$  and subtrees in between.
- length of
  - path from root to  $v_{split}$
  - left path
  - right path
- all lengths are  $O(\log n)$
- sum of the sizes of red subtrees:  $\leq k$
- query time:  $O(k + \log n)$

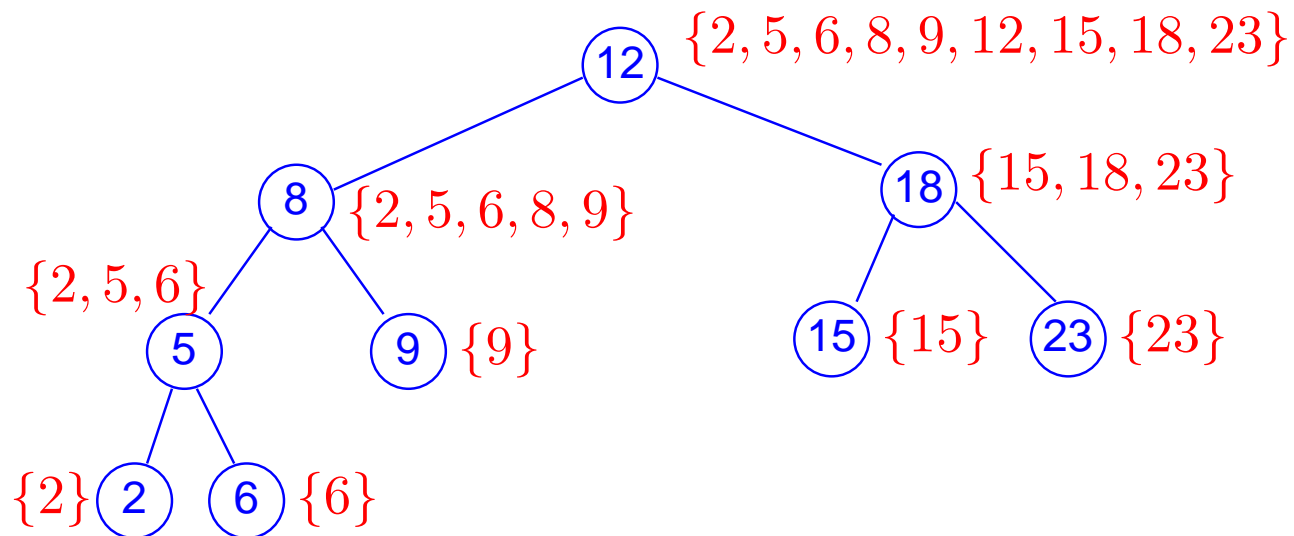
# Planar case ( $d=2$ ) using range trees

# Introduction

- a set  $P$  of  $n$  points in  $\mathbb{R}^2$
- queries: given  $(a_1, b_1, a_2, b_2)$ , find all the points  $(x, y)$  in  $P$  such that  $x \in [a_1, b_1]$  and  $y \in [a_2, b_2]$ .
- results presented in this section
  - $\Theta(n \log n)$  preprocessing time
  - $\Theta(n \log n)$  space usage
  - $\Theta(k + \log^2 n)$  query time
- query time will be slightly improved in the last section

# Canonical sets

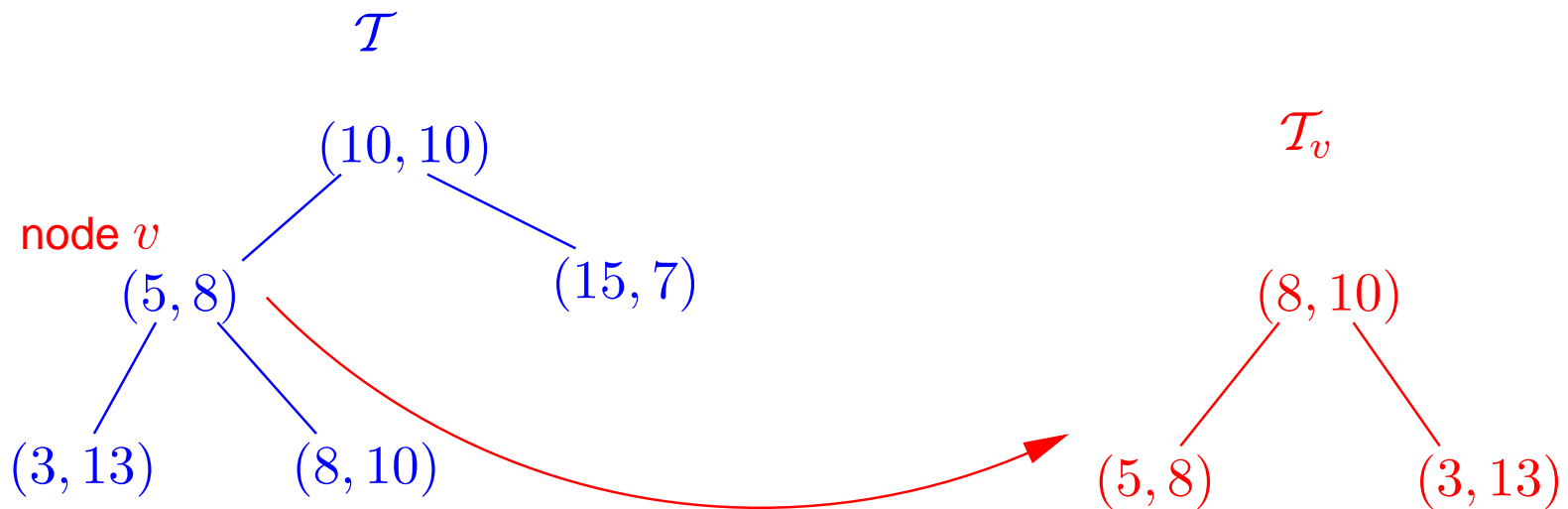
- we first store  $\mathcal{T}$  in a BBST using the  $x$ -coordinates as keys
- each node  $v$  of  $\mathcal{T}$  is associated with a *canonical set*  $C_v$ , which is the set of all the points in  $P$  that are stored in the subtree rooted at  $v$



Example of canonical sets

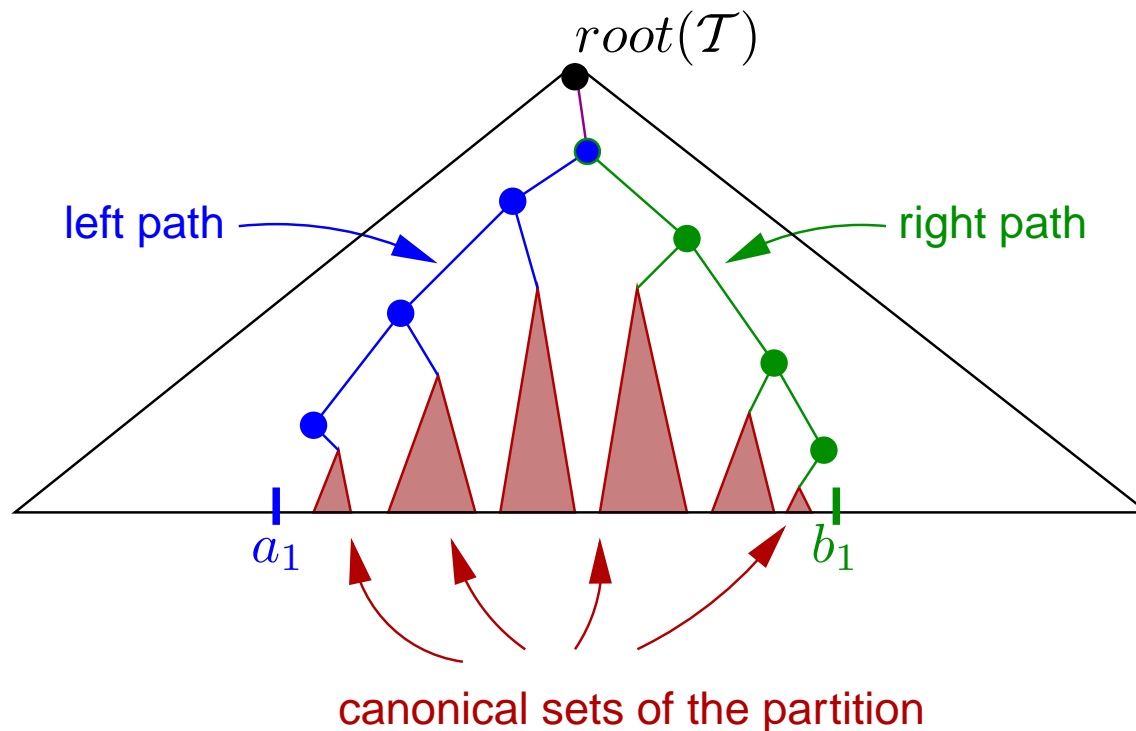
# Range trees in $\mathbb{R}^2$

- the canonical set of each node  $v$  of  $\mathcal{T}$  is stored in a BBST  $\mathcal{T}_v$  using the  $y$ -coordinates as keys



# Querying a range tree

- let  $P' = P \cap ([a_1, b_1] \times (-\infty, \infty))$
- let  $P''$  be the set of points on the right path and the left path (when searching for  $a_1$  and  $b_1$ )
- we partition  $P' \setminus P''$  into canonical subsets
  - $P' = P'' \cup C_1 \cup C_2 \cup \dots \cup C_c$



# Partitioning $P'$

- find the left path and the right path, which gives  $P''$
- pick each canonical set that is stored in the right child of a node of the left path
- pick each canonical set that is stored in the left child of a node of the right path
- it takes  $O(\log n)$  time (height of the BBST)
- there are  $c = O(\log n)$  canonical sets in our partition

# Querying a range tree

- $\forall p \in P''$  check if  $p \in [a_1, b_1] \times [a_2, b_2]$ , and report it if it the case
- for all  $i$  perform a one dimensional range searching query in  $C_i$  with the interval  $[a_2, b_2]$  (using the appropriate tree  $\mathcal{T}_{v_i}$ )
- the union of all these results gives  $P \cap ([a_1, b_1] \times [a_2, b_2])$
- let  $k_i$  be the number of points reported in  $C_i$
- $\sum_i k_i < k$
- query time:

$$\sum_{i=1}^c O(k_i + \log n) = O(k) + c \log n = O(k + \log^2 n)$$

# Space usage

- for all node  $v$ , let  $C_v$  denote the canonical set at node  $v$
- a point  $p$  belongs to all the canonical sets in the path from the vertex of  $\mathcal{T}$  that stores  $p$  to the root (and only these canonical sets)
- hence it belongs to  $O(\log n)$  canonical sets

- so

$$\sum_{v \in \mathcal{T}} |C_v| = O(n \log n)$$

- the space usage is  $O(n \log n)$
- it is actually  $\Theta(n \log n)$ 
  - why?

# Preprocessing time

- $T_v$  can be build in  $O(|C_v| \log |C_v|)$  time
- hence the range tree can be build in time

$$\sum_v |C_v| \log |C_v| \leq \log n \cdot \sum_v |C_v| = \log n \cdot O(n \log n) = O(n \log^2 n)$$

- we can do better
  - compute the  $T_v$ 's from leaves to root
  - computing  $T_v$  is merging two sorted sequences
  - it takes  $O(|C_v|)$  time
  - overall, we can build the range tree in time

$$\sum_v |C_v| = \Theta(n \log n)$$

# Range trees in higher dimension

# Idea

- we want to perform range searching in  $\mathbb{R}^d$
- we still build  $\mathcal{T}$  with respect to the  $x_1$ -coordinate
- for each canonical set of  $\mathcal{T}$  we build a  $(d - 1)$ -dimensional range searching data structure using coordinates  $(x_2, x_3, \dots, x_d)$
- to answer a  $d$ -dimensional query
  - find the canonical sets of  $\mathcal{T}$  associated with  $[a_1, b_1]$
  - make a  $d - 1$ -dimensional query on each canonical set recursively, using  $[a_2, b_2] \times [a_3, b_3] \times \dots \times [a_d, b_d]$

# Analysis

- we assume  $d > 1$  and  $d = O(1)$
- query time:  $O(k + \log^d n)$
- space usage:  $\Theta(n \log^{d-1} n)$
- preprocessing time:  $\Theta(n \log^{d-1} n)$
- proof
  - by induction on  $d$

# Improved range trees

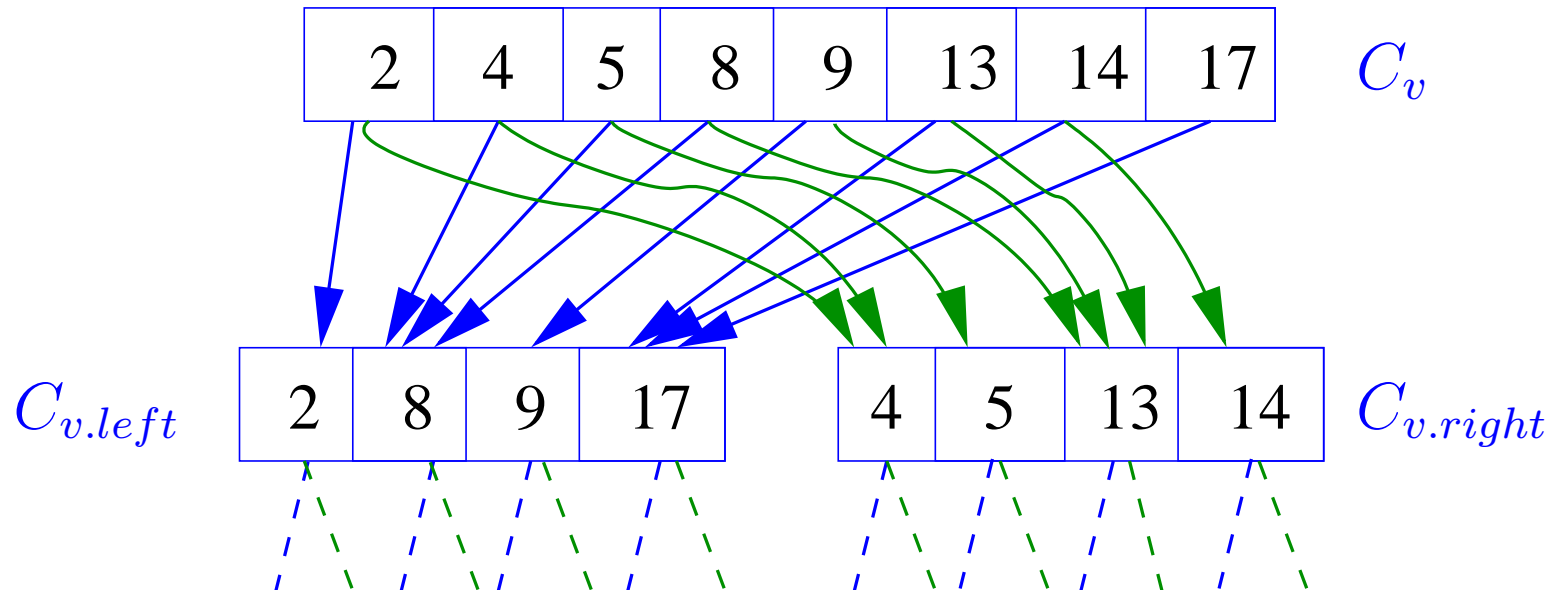
# Motivation

- in  $\mathbb{R}^2$  the query time of range trees is  $\Theta(k + \log^2 n)$
- for comparison based algorithms,  $\Omega(k + \log n)$  is a lower bound
- can we do better?
- yes, in this section we obtain  $\Theta(k + \log n)$  query time

# Idea

- when processing a query  $(a_1, b_1, a_2, b_2)$ , we search some trees  $T_v$ , always with the key  $a_2$  or  $b_2$
- for each such tree we spend  $O(\log n)$  time
- $C_{v.left}$  and  $C_{v.right}$  are subsets of  $C_v$
- we will keep pointers between nodes of  $T_v$  and nodes of  $T_{v.left}$  and  $T_{v.right}$  that keep the same key, or the next smallest key.
- after performing a search in  $T_v$  this will allow to perform a search in  $T_{v.left}$  and  $T_{v.right}$  in  $O(1)$  time.

# Data structure



- we first make a search in  $C_{root}$
- it takes  $O(\log n)$  time
- we follow these links when finding the canonical sets  $(C_1, C_2 \dots C_c)$
- a search in  $C_i$  can thus be performed in  $O(1)$  time

# Consequences

- this technique is known as fractional cascading
- by induction, it also improves by a factor  $O(\log n)$  the results in  $d > 2$
- range trees with fractional cascading in  $d \geq 2$  yield
  - query time:  $\Theta(k + \log^{d-1} n)$
  - space usage:  $\Theta(n \log^{d-1} n)$
  - preprocessing time:  $\Theta(n \log^{d-1} n)$
- in  $d = 2$ , query time and preprocessing time are optimal, but space usage is not
- $O(n \log n / \log \log n)$  space is possible in  $d = 2$  with same query time, and this is optimal (not covered in CS4235)

# Concluding remarks

- range trees:
  - simple
  - nearly optimal
- spatial databases mainly use  $R$ -trees
  - not covered in CS4235
  - good in practice with usual datasets
  - but no performance guarantee (no good worst case bound on the query time)