

# Computational Geometry Orthogonal Range Queries: Range-Trees

Thomas Bläsius

## Overview

#### **Basic Toolbox**

- convex hull
- line intersection
- triangulation
- plane intersection

### **Advanced Toolbox**

- Voronoi diagrams
- Delaunay triangulations
- randomized algorithms
- complexity



### **Geometric Data Structures**

- orthogonal range searching
- space partitioning
- point location





#### **Related Topics**

- What is geometry?
- hyperbolic geometry
- geometric graphs





## **Problem: Range Queries** Given a set of points $P \in \mathbb{R}^d$ and a box $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in $P \cap B$ .





### Problem: Range Queries

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### **Static Variant**

3

- point set P is fixed
- many different range queries





**Problem: Range Queries** 

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### **Static Variant**

- point set P is fixed
- many different range queries
- develop data structure based on P such that
  - each query is fast





#### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### **Static Variant**

- point set P is fixed
- many different range queries
- develop data structure based on P such that
  - each query is fast
  - data structure can be build efficiently



#### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### **Static Variant**

- point set P is fixed
- many different range queries
- develop data structure based on P such that
  - each query is fast
  - data structure can be build efficiently
  - data structure requires little space



#### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### **Static Variant**

- point set P is fixed
- many different range queries
- develop data structure based on P such that
  - each query is fast
  - data structure can be build efficiently
  - data structure requires little space

What are possible applications?





Problem: Range Queries

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### Simplest Case: d = 1

- the points are just numbers
- we look for all numbers in a given interval

### Problem: Range Queries

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### Simplest Case: d = 1

#### Solution 1

- the points are just numbers
- we look for all numbers in a given interval

### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

### Simplest Case: d = 1

- the points are just numbers
- we look for all numbers in a given interval

#### **Solution 1**

- data structure: sorted array
- query: binary search

#### 1 2 3 4 5 6 7 8

- $\rightarrow O(n \log n)$
- $ightarrow O(\log n + k)$ (k = output size)

#### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

### Simplest Case: d = 1

- the points are just numbers
- we look for all numbers in a given interval

#### **Solution 2**

- binary search tree with one leaf for each point
- query: search in the search tree

#### **Solution 1**

- data structure: sorted array
- query: binary search





- $\rightarrow O(n \log n)$
- $ightarrow O(\log n + k)$ (k = output size)



#### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

#### Simplest Case: d = 1

- the points are just numbers
- we look for all numbers in a given interval

#### **Solution 2**

- binary search tree with one leaf for each point
- query: search in the search tree

#### How does the search work?

#### **Solution 1**

- data structure: sorted array
- query: binary search



1 2 3 4 5 6 7 8





#### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

### Simplest Case: d = 1

- the points are just numbers
- we look for all numbers in a given interval

#### **Solution 2**

- binary search tree with one leaf for each point
- query: search in the search tree

How does the search work?

### **Solution 1**

- data structure: sorted array
- query: binary search



- $\rightarrow O(n \log n)$
- $\rightarrow O(\log n + k)$ (k = output size)



Which values do we store at the inner nodes?



#### **Problem: Range Queries**

Given a set of points  $P \in \mathbb{R}^d$  and a box  $B = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ , find all points in  $P \cap B$ .

### Simplest Case: d = 1

- the points are just numbers
- we look for all numbers in a given interval

#### **Solution 2**

- binary search tree with one leaf for each point
- query: search in the search tree

How does the search work?

### **Solution 1**

- data structure: sorted array
- query: binary search



- $\rightarrow O(n \log n)$
- $\rightarrow O(\log n + k)$ (k = output size)



Which values do we store at the inner nodes?







- first search in the first dimension (x)
- search in the second dimension (y) on the result





#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 





#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem





#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

#### **Problem**

y-search is done on a subset of points





#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

#### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

Why not?



#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

Why not?

#### Idea

store y-sorted array for few important subsets





#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

Why not?

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets





#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : 1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

Why not?

### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets



 $b_1$ 

- **b**2

 $a_2$ 

6

X

7 8

5.6

 $a_1$ 

3.8



1, 1

**6**, **2** 

2,3

4,4

8,5

<mark>5, 6</mark>

7,7

3,8

#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

## Why not?

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets







#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

## Why not?

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets







#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree





#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree







#### Idea

- first search in the first dimension (x)
- search in the second dimension (y) on the result

search for  $x \in [a_1, b_1]$ : **1,1 2,3 3,8 4,4 5,6 6,2 7,7 8,5** 

### Problem

- y-search is done on a subset of points
- we cannot store a y-sorted array for each possible subset

### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### Why not?







- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree





#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### Query $[a_1, b_1] \times [a_2, b_2]$





#### Idea

6

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### Query $[a_1, b_1] \times [a_2, b_2]$

• find the predecessor of  $a_1$  and the successor of  $b_1$  in the *x*-tree







#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

## Query $[a_1, b_1] \times [a_2, b_2]$

- find the predecessor of  $a_1$  and the successor of  $b_1$  in the x-tree
- for nodes directly below the path: binary search in the corresponding y-Array for [a<sub>2</sub>, b<sub>2</sub>] → output found points





#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### Query $[a_1, b_1] \times [a_2, b_2]$

- find the predecessor of  $a_1$  and the successor of  $b_1$  in the x-tree
- for nodes directly below the path: binary search in the corresponding y-Array for [a<sub>2</sub>, b<sub>2</sub>] → output found points

### **Running Time Of The Query**





#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### Query $[a_1, b_1] \times [a_2, b_2]$

- find the predecessor of  $a_1$  and the successor of  $b_1$  in the x-tree
- for nodes directly below the path: binary search in the corresponding y-Array for [a<sub>2</sub>, b<sub>2</sub>] → output found points

### **Running Time Of The Query**

search in the x-tree

```
\rightarrow O(\log n)
```







#### Idea

6

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### Query $[a_1, b_1] \times [a_2, b_2]$

- find the predecessor of  $a_1$  and the successor of  $b_1$  in the x-tree
- for nodes directly below the path: binary search in the corresponding y-Array for [a<sub>2</sub>, b<sub>2</sub>] → output found points

 $ightarrow O(\log n)$ 

### **Running Time Of The Query**

- search in the x-tree
- search in  $O(\log n)$  y-arrays  $\rightarrow O(\log^2 n)$



 $a_1$ 

 $b_1$
### Queries In A 2D Range Tree

#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### Query $[a_1, b_1] \times [a_2, b_2]$

- find the predecessor of  $a_1$  and the successor of  $b_1$  in the x-tree
- for nodes directly below the path: binary search in the corresponding y-Array for [a<sub>2</sub>, b<sub>2</sub>] → output found points

### **Running Time Of The Query**

search in the x-tree

- $ightarrow O(\log n)$
- search in  $O(\log n)$  y-arrays  $\rightarrow O(\log^2 n)$

(remember: +O(k) for the output size)







#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree
- **Computing The Data Structure**





#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

#### **Computing The Data Structure**

compute the x-tree



 $\rightarrow O(n \log n)$ 



#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

#### **Computing The Data Structure**

- compute the x-tree
- sort each of the y-arrays



 $\rightarrow O(n \log n)$ 



#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### **Computing The Data Structure**

- compute the x-tree
- sort each of the *y*-arrays:  $O(n \log n)$  per layer



 $ightarrow O(n \log n)$ ightarrow O(n \log^2 n)



#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### **Computing The Data Structure**

- compute the x-tree
- sort each of the *y*-arrays:  $O(n \log n)$  per layer
- improving the second step



 $ightarrow O(n \log n)$ ightarrow O(n \log^2 n)



#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### **Computing The Data Structure**

- compute the x-tree
- sort each of the *y*-arrays:  $O(n \log n)$  per layer
- improving the second step
  - sort all points once by y



 $\rightarrow O(n \log n) \\ \rightarrow O(n \log^2 n)$ 

 $\rightarrow O(n \log n)$ 



#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### **Computing The Data Structure**

- compute the x-tree
- sort each of the *y*-arrays:  $O(n \log n)$  per layer
- improving the second step
  - sort all points once by y
  - split sorted array to obtain sorted array for the children



 $\rightarrow O(n \log n) \\ \rightarrow O(n \log^2 n)$ 

 $\rightarrow O(n \log n)$ 



### 7 Thomas Bläsius – Computational Geometry

### Computing A 2D Range Tree

#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

#### **Computing The Data Structure**

- compute the x-tree
- sort each of the *y*-arrays:  $O(n \log n)$  per layer
- improving the second step
  - sort all points once by y
  - split sorted array to obtain sorted array for the children: O(n) per layer



 $ightarrow O(n \log n)$ ightarrow O(n \log^2 n)

 $ightarrow O(n \log n)$ ightarrow O(n \log n)



#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### **Computing The Data Structure**

- compute the x-tree
- sort each of the *y*-arrays:  $O(n \log n)$  per layer
- improving the second step
  - sort all points once by y
  - split sorted array to obtain sorted array for the children: O(n) per layer

 $\rightarrow O(n \log n)$  $\rightarrow O(n \log^2 n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$ 



#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### **Computing The Data Structure**

- compute the x-tree
- sort each of the y-arrays:  $O(n \log n)$  per layer
- improving the second step
  - sort all points once by y
  - split sorted array to obtain sorted array for the children: O(n) per layer

### **Memory Consumption**



 $\rightarrow O(n \log n)$  $\rightarrow O(n \log^2 n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$ 

#### 7 Thomas Bläsius – Computational Geometry

### Computing A 2D Range Tree

#### Idea

- store y-sorted array for few important subsets
- nodes in the x-tree define important subsets
- this data structure is called 2D range tree

### **Computing The Data Structure**

- compute the x-tree
- sort each of the *y*-arrays:  $O(n \log n)$  per layer
- improving the second step
  - sort all points once by y
  - split sorted array to obtain sorted array for the children: O(n) per layer

### **Memory Consumption**

O(n) per layer



 $\rightarrow O(n \log n)$  $\rightarrow O(n \log^2 n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$  $\rightarrow O(n \log n)$ 

 $ightarrow O(n \log n)$ 

What Is The Solution To The Query  $[4, 11] \times [3, 7]$ ?





### What Is The Solution To The Query $[4, 11] \times [3, 7]$ ?



#### Theorem

The range tree for *n* points in  $\mathbb{R}^2$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^2 n + k)$  time.



#### Theorem

The range tree for *n* points in  $\mathbb{R}^2$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^2 n + k)$  time.

Idea For Dimension d > 2



#### Theorem

The range tree for *n* points in  $\mathbb{R}^2$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^2 n + k)$  time.

Idea For Dimension d > 2binary search tree for dimension 1

#### Theorem

The range tree for *n* points in  $\mathbb{R}^2$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^2 n + k)$  time.

#### Idea For Dimension d > 2

- binary search tree for dimension 1
- every node stores a (d 1)-dim range tree for remaining dimensions of the points below this node



**Theorem** The range tree for *n* points in  $\mathbb{R}^{2^d}$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^{2^d} n + k)$  time.

#### Idea For Dimension d > 2

- binary search tree for dimension 1
- every node stores a (d 1)-dim range tree for remaining dimensions of the points below this node



**Theorem** The range tree for *n* points in  $\mathbb{R}^{\frac{2}{d}}$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^{\frac{2}{d}}n + k)$  time.

#### Idea For Dimension d > 2

- binary search tree for dimension 1
- every node stores a (d 1)-dim range tree for remaining dimensions of the points below this node

**Proof:** induction over *d* (base case d = 2 already done)



**Theorem** The range tree for *n* points in  $\mathbb{R}^{\frac{2}{d}}$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^{\frac{2}{d}}n + k)$  time.

### Idea For Dimension d > 2

- binary search tree for dimension 1
- every node stores a (d 1)-dim range tree for remaining dimensions of the points below this node



**Proof:** induction over *d* (base case d = 2 already done)

• building the binary search tree for dimension 1:  $O(n \log n)$  time and O(n) space

**Theorem** The range tree for *n* points in  $\mathbb{R}^{2^d}$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^{2^d} n + k)$  time.

### Idea For Dimension d > 2

- binary search tree for dimension 1
- every node stores a (d 1)-dim range tree for remaining dimensions of the points below this node



**Proof:** induction over *d* (base case d = 2 already done)

- building the binary search tree for dimension 1:  $O(n \log n)$  time and O(n) space
- per level: (d-1)-dim range trees for *n* points in total  $\Rightarrow O(n \log^{d-2} n)$  time and space per layer

**Theorem** The range tree for *n* points in  $\mathbb{R}^{2^d}$  can be computed in  $O(n \log n)$  time, requires  $O(n \log n)$  memory and enables range queries in  $O(\log^{2^d} n + k)$  time.

### Idea For Dimension d > 2

- binary search tree for dimension 1
- every node stores a (d 1)-dim range tree for remaining dimensions of the points below this node



- **Proof:** induction over *d* (base case d = 2 already done)
- building the binary search tree for dimension 1:  $O(n \log n)$  time and O(n) space
- per level: (d-1)-dim range trees for *n* points in total  $\Rightarrow O(n \log^{d-2} n)$  time and space per layer
- query:  $O(\log n)$  for first search plus  $O(\log n)$  queries in (d-1)-dim range trees (with disjoint output!)

#### **Current State**

 $d = 1 \qquad d = 2 \qquad d > 2$ range query  $\log n + k \quad \log^2 n + k \quad \log^d n + k$ precomputation  $n \log n \quad n \log n \quad n \log^{d-1} n$ memory  $n \quad n \log n \quad n \log^{d-1} n$ 

#### **Current State**

	d = 1	d = 2	<i>d</i> > 2
range query	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
precomputation	n log n	n log n	$n \log^{d-1} n$
memory	п	n log n	$n \log^{d-1} n$

• for each dimension, we lose a log *n* factor



#### **Current State**

 $d = 1 \qquad d = 2 \qquad d > 2$ range query  $\log n + k \quad \log^2 n + k \quad \log^d n + k$ precomputation  $n \log n \quad n \log n \quad n \log^{d-1} n$ memory  $n \quad n \log n \quad n \log^{d-1} n$ 

• for each dimension, we lose a log *n* factor

• if we improve d = 2, we also improve d > 2

#### **Current State**

d = 1d = 2d > 2range query $\log n + k$  $\log^2 n + k$  $\log^d n + k$ precomputation $n \log n$  $n \log n$  $n \log^{d-1} n$ memoryn $n \log n$  $n \log^{d-1} n$ 

- for each dimension, we lose a log *n* factor
- if we improve d = 2, we also improve d > 2
- from d = 1 to d = 2, we already saved log *n* in precomputation

(the trick with sorting only once and then splitting the sorted array)

#### **Current State**

	d = 1	d = 2	<i>d</i> > 2
range query	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
precomputation	n log n	n log n	$n \log^{d-1} n$
memory	п	n log n	$n \log^{d-1} n$

### for each dimension, we lose a log n factor

- if we improve d = 2, we also improve d > 2
- from d = 1 to d = 2, we already saved log n in precomputation (the trick with sorting only once and then splitting the sorted array)

#### Today

- save log *n* query time for d = 2
- also saves a log n factor for all higher dimensions

#### **Current State**

	d = 1	d = 2	<i>d</i> > 2
range query	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
precomputation	n log n	n log n	$n \log^{d-1} n$
memory	п	n log n	$n \log^{d-1} n$

- for each dimension, we lose a log *n* factor
- if we improve d = 2, we also improve d > 2
- from d = 1 to d = 2, we already saved log n in precomputation (the trick with sorting only once and then splitting the sorted array)

### Today

- save  $\log n$  query time for d = 2
- also saves a log n factor for all higher dimensions

#### **Next Lecture**

- save another  $\log n$  factor in query time for d = 3
- pay for this with an additional log n factor in precomputation time and memory



### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees



### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$





### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$



Actually...

• we only search on  $\leq n$  numbers in total



### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$

# 

#### Actually...

• we only search on  $\leq n$  numbers in total

 $b_1$ 

we always search for the same numbers



### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$

### Actually...

• we only search on  $\leq n$  numbers in total

 $b_1$ 

 $a_1$ 

- we always search for the same numbers
- we only need so long as the numbers are split into subsets



### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$

#### Actually...

• we only search on  $\leq n$  numbers in total

 $b_1$ 

 $a_1$ 

- we always search for the same numbers
- we only need so long as the numbers are split into subsets
- searching on all *n* numbers would be faster

### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$

## 

#### Actually...

- we only search on  $\leq n$  numbers in total
- we always search for the same numbers
- we only need so long as the numbers are split into subsets
- searching on all *n* numbers would be faster

**Idea:** search only once in a superset of the relevant points


### Why Is It So Expensive?

### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$

# $\begin{array}{c} & & & \\ & & & & \\ & & & \\ & & & & \\ & & & & \\ & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\$

#### Actually...

- we only search on  $\leq n$  numbers in total
- we always search for the same numbers
- we only need so long as the numbers are split into subsets
- searching on all *n* numbers would be faster

**Idea:** search only once in a **superset** of the relevant points **Problem:** result potentially contains points not in the *x*-range



### Why Is It So Expensive?

### **Recall:** Query In $O(\log^2 n + k)$

- search in the *x*-tree  $\rightarrow O(\log n)$ 
  - finds all points with x-coordinate in  $[a_1, b_1]$
  - implicit representation via  $O(\log n)$  subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$  per search  $\rightarrow O(\log^2 n)$

## 

#### Actually...

- we only search on  $\leq n$  numbers in total
- we always search for the same numbers
- we only need so long as the numbers are split into subsets
- searching on all *n* numbers would be faster

Idea: search only once in a superset of the relevant pointsProblem: result potentially contains points not in the *x*-rangeIdea: search the position in the superset but list the result in the correct subsets



Situation (Simplified)



Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$   $A = \begin{bmatrix} 2 & 5 & 8 & 12 \\ 2 & 5 & 8 & 12 \end{bmatrix}$
- search for x in A
- find x in B without searching again



Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$   $A = \begin{bmatrix} A \\ A \end{bmatrix}$
- search for x in A
- find x in B without searching again

Case 1:  $x \in B$ 

$$A = 2 5 8 12 16 17 25 28 32 33 34$$
$$B = 2 8 12 16 32 34$$



#### Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$
- search for x in A
- find x in B without searching again
- Case 1:  $x \in B$
- pointers form elements in A to copies in  $B \rightarrow \text{find } x$  in B in O(1)





Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$
- search for x in A
- find x in B without searching again

Case 1:  $x \in B$ 

• pointers form elements in A to copies in  $B \rightarrow \text{find } x$  in B in O(1)

Case 2:  $x \in A$  but  $x \notin B$ 





#### Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$
- search for x in A
- find x in B without searching again
- Case 1:  $x \in B$
- pointers form elements in A to copies in  $B \rightarrow \text{find } x$  in B in O(1)
- Case 2:  $x \in A$  but  $x \notin B$
- goal: find predecessor of x in B





Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$
- search for x in A
- find x in B without searching again

Case 1:  $x \in B$ 

• pointers form elements in A to copies in  $B \rightarrow \text{find } x$  in B in O(1)

Case 2:  $x \in A$  but  $x \notin B$ 

- goal: find predecessor of *x* in *B*
- pointer from every  $a \in A \setminus B$  to its predecessor in  $A \cap B \to \text{find } x$  in B in O(1)



Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$
- search for x in A
- find x in B without searching again

Case 1:  $x \in B$ 

• pointers form elements in A to copies in  $B \rightarrow \text{find } x$  in B in O(1)

Case 2:  $x \in A$  but  $x \notin B$ 

- goal: find predecessor of *x* in *B*
- pointer from every  $a \in A \setminus B$  to its predecessor in  $A \cap B \to \text{find } x$  in B in O(1)

Case 3:  $x \notin A$ 





Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$
- search for x in A
- find x in B without searching again

Case 1:  $x \in B$ 

• pointers form elements in A to copies in  $B \rightarrow \text{find } x$  in B in O(1)

Case 2:  $x \in A$  but  $x \notin B$ 

- goal: find predecessor of *x* in *B*
- pointer from every  $a \in A \setminus B$  to its predecessor in  $A \cap B \to \text{find } x$  in B in O(1)

Case 3:  $x \notin A$ 

• goal: find predecessor of x in B, when knowing the predecessor of x in A





Situation (Simplified)

- consider sorted arrays of numbers A and B with  $B \subseteq A$
- search for x in A
- find x in B without searching again

Case 1:  $x \in B$ 

• pointers form elements in A to copies in  $B \rightarrow \text{find } x$  in B in O(1)

#### Case 2: $x \in A$ but $x \notin B$

- goal: find predecessor of *x* in *B*
- pointer from every  $a \in A \setminus B$  to its predecessor in  $A \cap B \to \text{find } x$  in B in O(1)

### Case 3: $x \notin A$

- goal: find predecessor of x in B, when knowing the predecessor of x in A
- use case 1 or 2  $\rightarrow$  find x in B in O(1)



Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching





#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

#### So Many Subsets

 $\blacksquare$  many subsets  $\rightarrow$  too many pointers



#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

- $\blacksquare$  many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children





#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

- many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children









#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

- many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children







#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

- $\blacksquare$  many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children







#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

#### So Many Subsets

- $\blacksquare$  many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children





#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

### So Many Subsets

- $\blacksquare$  many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children





#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

### So Many Subsets

- $\blacksquare$  many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children





#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

### So Many Subsets

- $\blacksquare$  many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children



 $a_1$ 



#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

### So Many Subsets

- $\blacksquare$  many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children



 $a_1$ 



#### Plan

- search for  $a_2$  and  $b_2$  in the superset
- find  $a_2$  and  $b_2$  in the subsets without searching

### **So Many Subsets**

- many subsets  $\rightarrow$  too many pointers
- solution: store pointers only for children



 $a_1$ 



**Range Query** 



#### **Range Query**

- search in y-Array at the root
- walk down the x-tree (log n steps)
  - decision for left or right
  - finding the range  $[a_2, b_2]$  in the y-array (without searching)
- output result

 $O(\log n)$  $O(\log n)$ O(1)O(1)O(k)

#### **Range Query**

- search in y-Array at the root
- walk down the x-tree (log n steps)
  - decision for left or right
  - finding the range  $[a_2, b_2]$  in the y-array (without searching)
- output result
- **Memory:** only constant factor overhead for each *y*-array

 $O(\log n)$  $O(\log n)$ O(1)O(1)O(k)



#### **Range Query**

- search in y-Array at the root
- walk down the x-tree (log n steps)
  - decision for left or right
  - finding the range  $[a_2, b_2]$  in the *y*-array (without searching)
- output result

**Memory:** only constant factor overhead for each *y*-array

Precomputation: sort only at the root and split for the children (additionally adding pointers)

 $O(\log n)$  $O(\log n)$ O(1)ng)O(1)O(k)



#### **Range Query**

- search in y-Array at the root
- walk down the x-tree (log n steps)
  - decision for left or right
  - finding the range  $[a_2, b_2]$  in the *y*-array (without searching)
- output result
- **Memory:** only constant factor overhead for each *y*-array

Precomputation: sort only at the root and split for the children (additionally adding pointers)

**Theorem** The range tree for *n* points in  $\mathbb{R}^d$  can be computed in  $O(n \log^{d-1} n)$  time, requires  $O(n \log^{d-1} n)$  memory and enables range queries in  $O(\log^{d-1} n + k)$  time.

O(1)O(1)O(k)itionally adding

 $O(\log n)$ 

 $O(\log n)$ 



What Have We Learned Today?

generalization of the binary search to multiple dimensions

#### What Have We Learned Today?

- generalization of the binary search to multiple dimensions
- range trees: nested binary search trees

#### What Have We Learned Today?

- generalization of the binary search to multiple dimensions
- range trees: nested binary search trees
- one big search is better than many small searches  $\rightarrow$  clever pointers save log *n*

#### What Have We Learned Today?

- generalization of the binary search to multiple dimensions
- range trees: nested binary search trees
- one big search is better than many small searches  $\rightarrow$  clever pointers save log *n*

**Theorem** (improved range trees) The range tree for *n* points in  $\mathbb{R}^d$  can be computed in  $O(n \log^{d-1} n)$  time, requires  $O(n \log^{d-1} n)$  memory and enables range queries in  $O(\log^{d-1} n + k)$  time.



#### What Have We Learned Today?

- generalization of the binary search to multiple dimensions
- range trees: nested binary search trees
- one big search is better than many small searches  $\rightarrow$  clever pointers save log *n*

**Theorem** (improved range trees) The range tree for *n* points in  $\mathbb{R}^d$  can be computed in  $O(n \log^{d-1} n)$  time, requires  $O(n \log^{d-1} n)$  memory and enables range queries in  $O(\log^{d-1} n + k)$  time.

#### **Next Lecture**

 $\blacksquare$  generalization of the concept of clever pointers  $\rightarrow$  fractional cascading



#### What Have We Learned Today?

- generalization of the binary search to multiple dimensions
- range trees: nested binary search trees
- one big search is better than many small searches  $\rightarrow$  clever pointers save log *n*

**Theorem** (improved range trees) The range tree for *n* points in  $\mathbb{R}^d$  can be computed in  $O(n \log^{d-1} n)$  time, requires  $O(n \log^{d-1} n)$  memory and enables range queries in  $O(\log^{d-1} n + k)$  time.

#### **Next Lecture**

- $\blacksquare$  generalization of the concept of clever pointers  $\rightarrow$  fractional cascading
- lets us save an additional log *n* factor in the query ( $d \ge 3$ )



#### What Have We Learned Today?

- generalization of the binary search to multiple dimensions
- range trees: nested binary search trees
- one big search is better than many small searches  $\rightarrow$  clever pointers save log *n*

**Theorem** (improved range trees) The range tree for *n* points in  $\mathbb{R}^d$  can be computed in  $O(n \log^{d-1} n)$  time, requires  $O(n \log^{d-1} n)$  memory and enables range queries in  $O(\log^{d-1} n + k)$  time.

#### **Next Lecture**

- $\blacksquare$  generalization of the concept of clever pointers  $\rightarrow$  fractional cascading
- lets us save an additional log *n* factor in the query ( $d \ge 3$ )
- costs an additional log n factor precomputation time and memory


#### **Range Tree**

- stores points
- Which points lie in a given interval?



#### **Range Tree**

- stores points
- Which points lie in a given interval?

## **Segment Tree**

- stores intervals
- Which intervals contain a given point?



### **Range Tree**

- stores points
- Which points lie in a given interval?

## **Segment Tree**

- stores intervals
- Which intervals contain a given point?

### **Interval Tree**

- stores intervals
- Which intervals intersect a given interval?



### **Range Tree**

- stores points
- Which points lie in a given interval?

## **Segment Tree**

- stores intervals
- Which intervals contain a given point?

## **Interval Tree**

- stores intervals
- Which intervals intersect a given interval?

## **Segment Tree**

- stores weighted points
- What is the sum of weights of points in a given interval?

### **Range Tree**

- stores points
- Which points lie in a given interval?

### **Segment Tree**

- stores intervals
- Which intervals contain a given point?

## **Interval Tree**

- stores intervals
- Which intervals intersect a given interval?

## **Segment Tree**

- stores weighted points
- What is the sum of weights of points in a given interval?

### **Similarities**

- can be nested to extend to higher dimensions
- fractional cascading can help to save logarithmic factors

