Computational Geometry

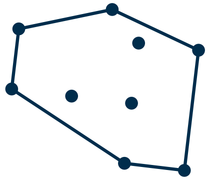
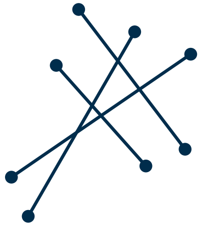
Orthogonal Range Queries: Range-Trees

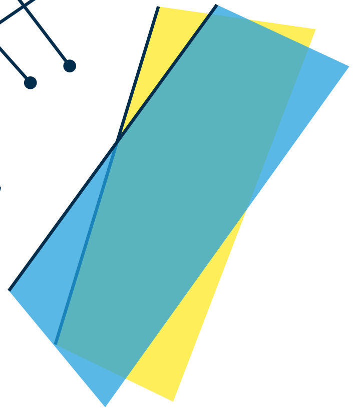
Thomas Bläsius

Overview

Basic Toolbox

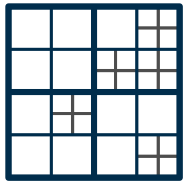
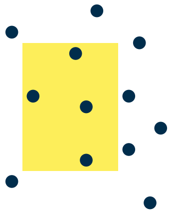
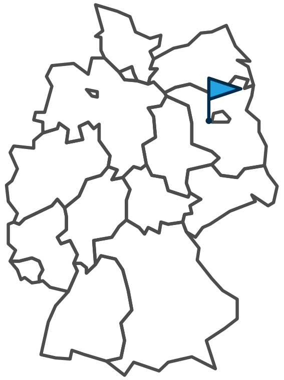
- convex hull
- line intersection
- triangulation
- plane intersection



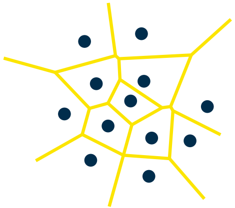
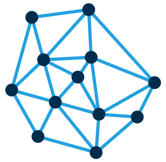
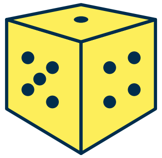
Geometric Data Structures

- orthogonal range searching
- space partitioning
- point location

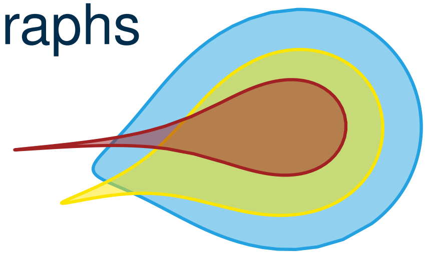
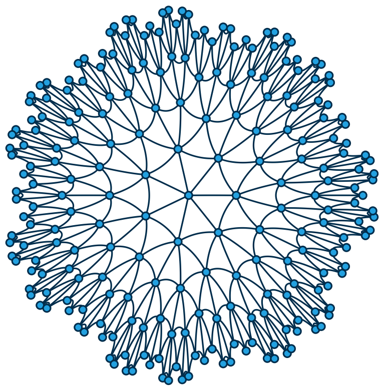
Advanced Toolbox

- Voronoi diagrams
- Delaunay triangulations
- randomized algorithms
- complexity

Related Topics

- What is geometry?
- hyperbolic geometry
- geometric graphs

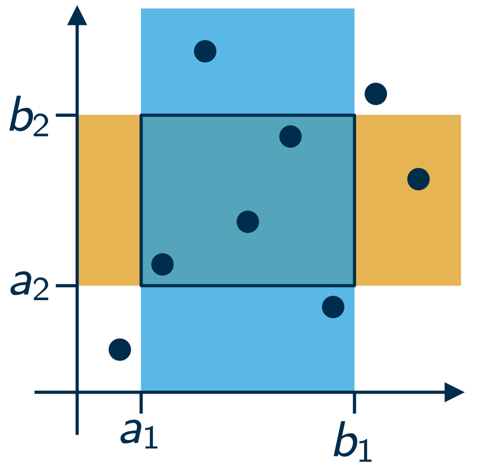



Range Queries

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
  - data structure can be build efficiently
  - data structure requires little space



What are possible applications?

1D 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 $P \cap B$.

Simplest Case: $d = 1$

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

Solution 1

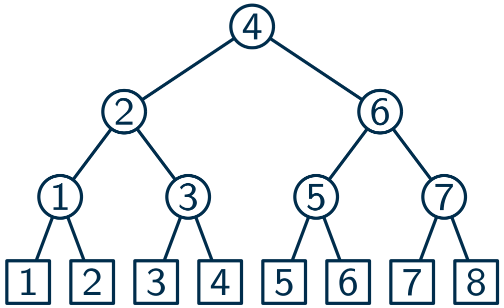
1	2	3	4	5	6	7	8

- data structure: sorted array $\rightarrow O(n \log n)$
- query: binary search $\rightarrow O(\log n + k)$

(k = output size)

Solution 2

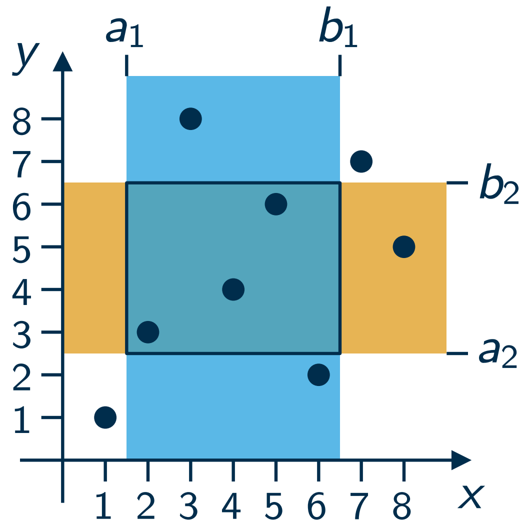
- binary search tree with one leaf for each point
- query: search in the search tree



How does the search work?

Which values do we store at the inner nodes?

2D Range Queries



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 |

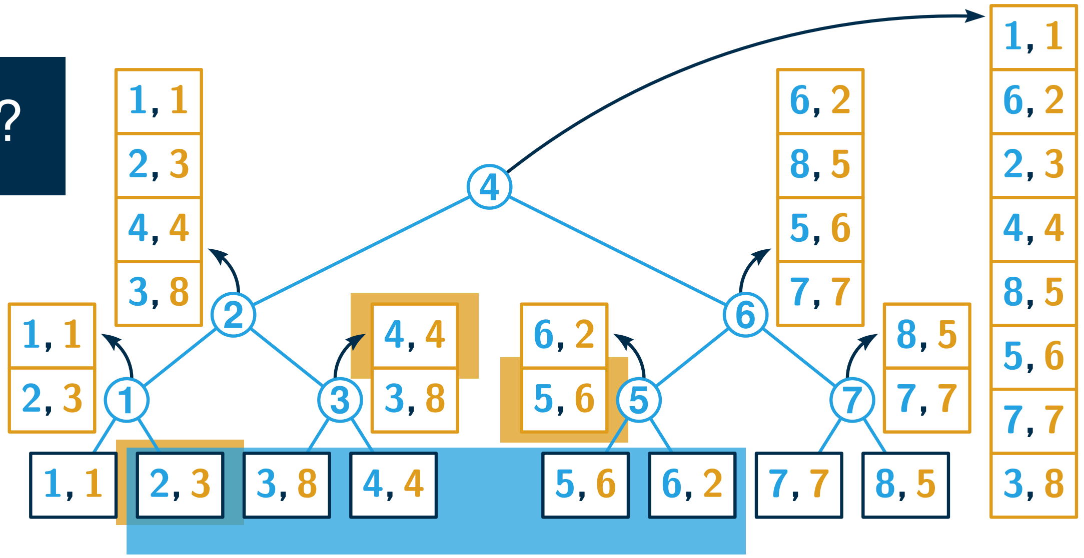
Problem

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

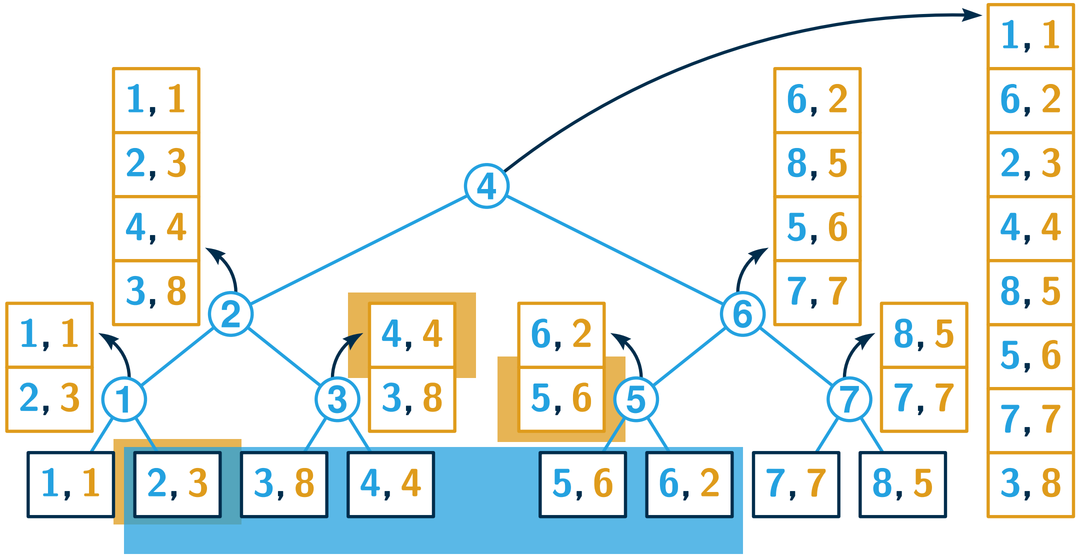
Why not?

Idea

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



Queries In A 2D Range Tree

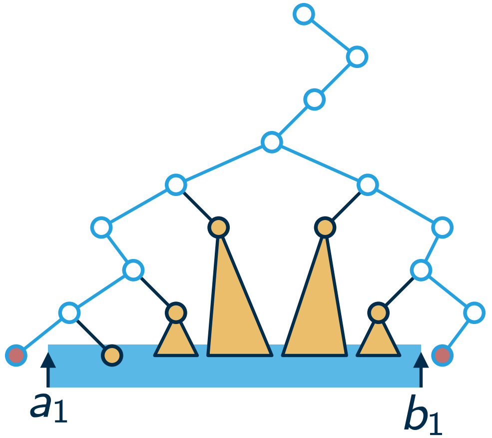


Idea

- 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 $[a_2, b_2] \to$ output found points



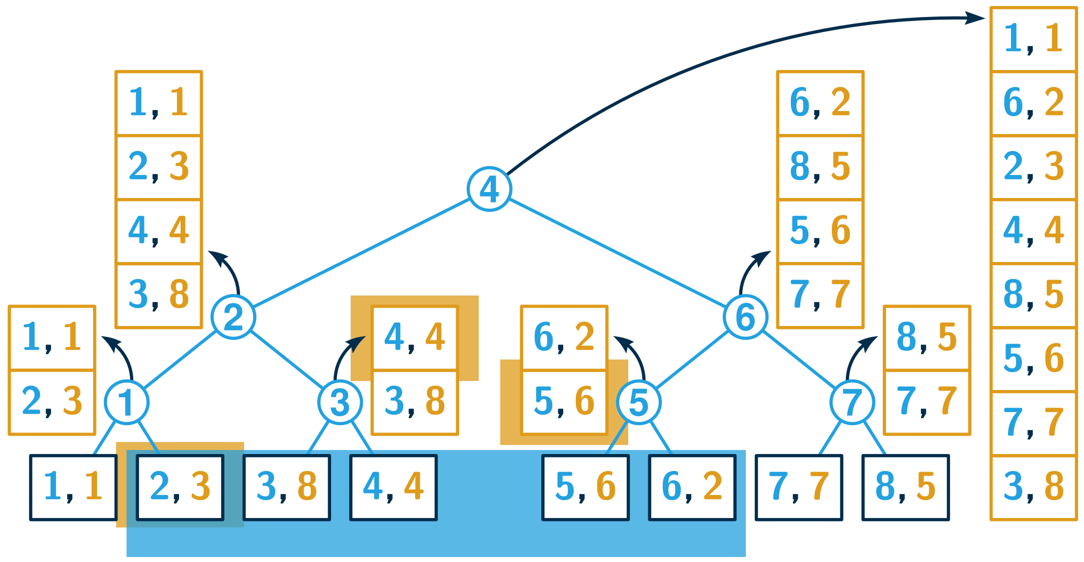
Running Time Of The Query

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

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

The subtrees yield disjoint sets of points!
Does this make it better than $\log^2 n$?

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

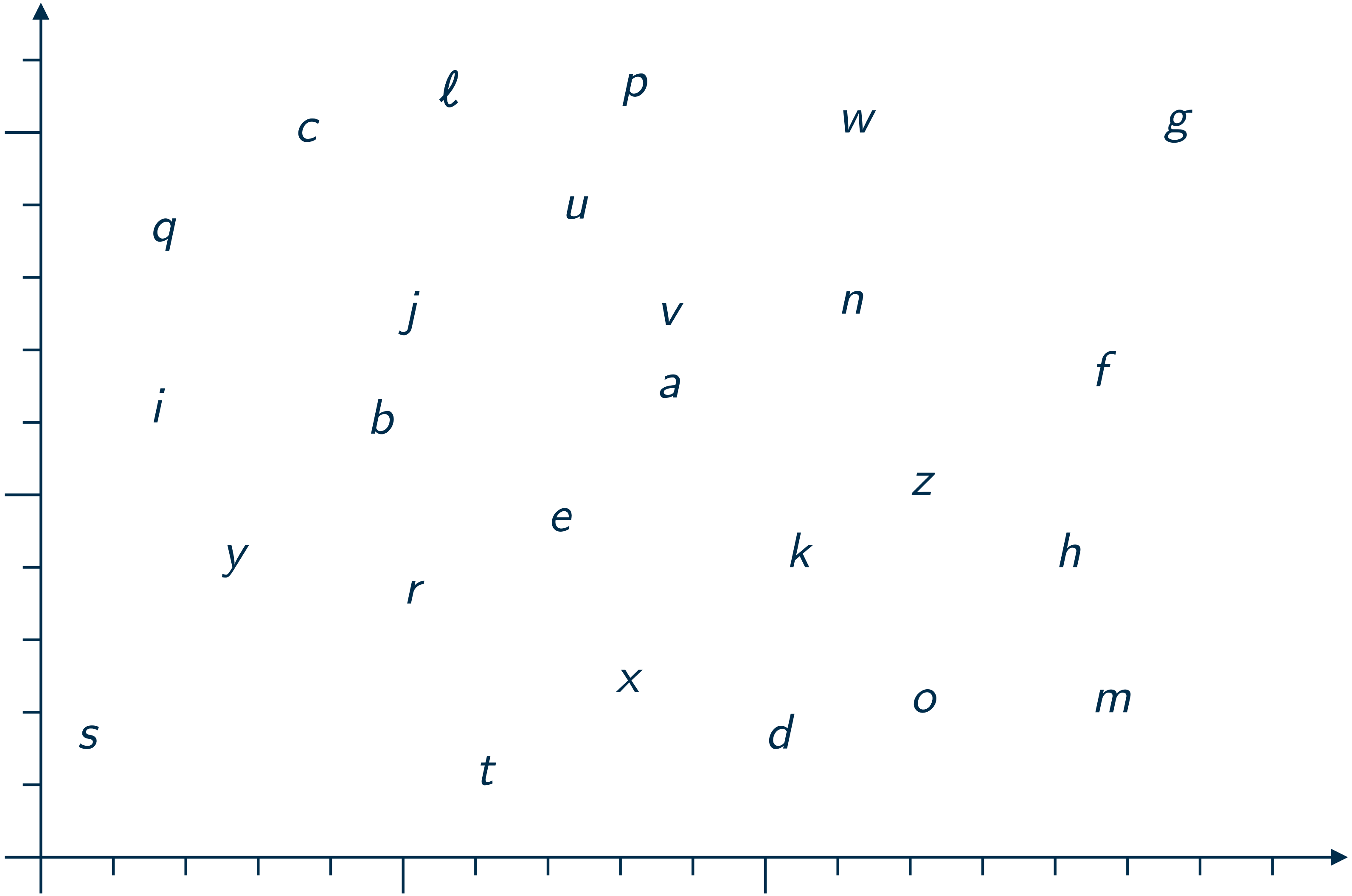
Computing The Data Structure

- compute the x-tree $\rightarrow O(n \log n)$
- sort each of the y-arrays: $O(n \log n)$ per layer $\rightarrow O(n \log^2 n)$
- improving the second step $\rightarrow O(n \log n)$
  - sort all points once by y $\rightarrow O(n \log n)$
  - split sorted array to obtain sorted array for the children: $O(n)$ per layer $\rightarrow O(n \log n)$

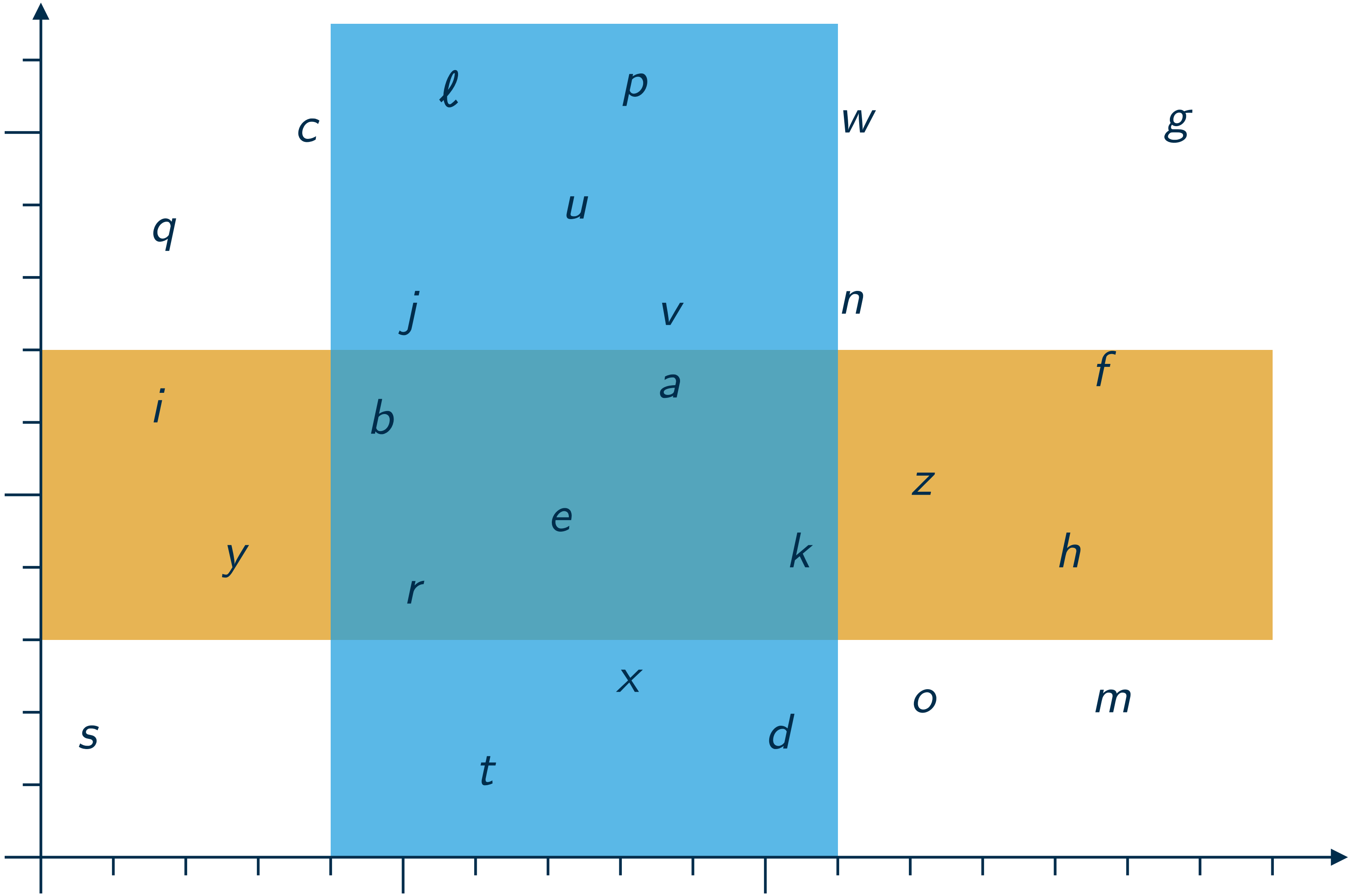
Memory Consumption

- $O(n)$ per layer $\rightarrow O(n \log n)$

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



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

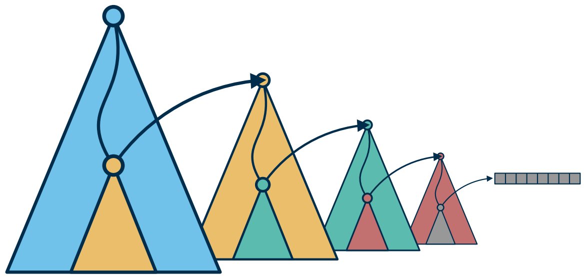


General Range Trees

Theorem
The range tree for n points in $\mathbb{R}^{\cancel{2}\,d}$ can be computed in $O(n\, \cancel{\log n}\, \log^{d-1} n)$ time, requires $O(n\, \cancel{\log n}\, \log^{d-1} n)$ memory and enables range queries in $O(\log^{\cancel{2}\,d} n + k)$ 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!)

Can We Improve?

Current State

	$d = 1$	$d = 2$	$d > 2$
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	n	$n \log n$	$n \log^{d-1} n$

- 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)

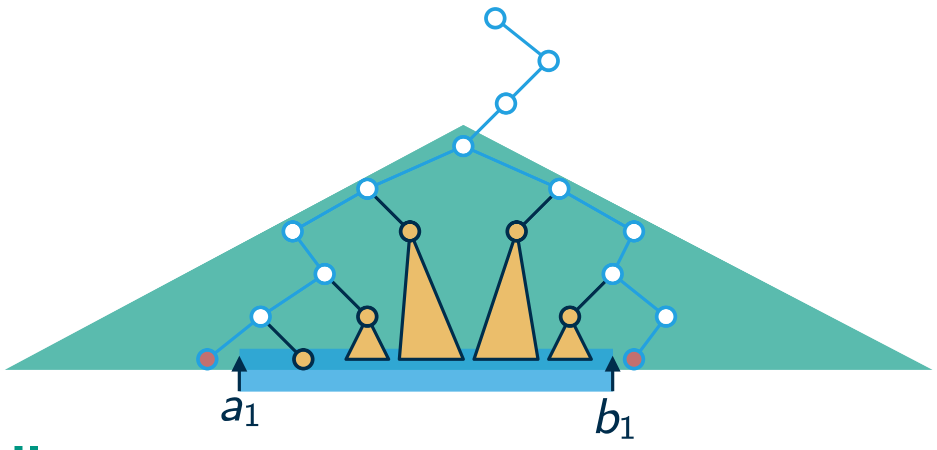
Today

- 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

Why Is It So Expensive?



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

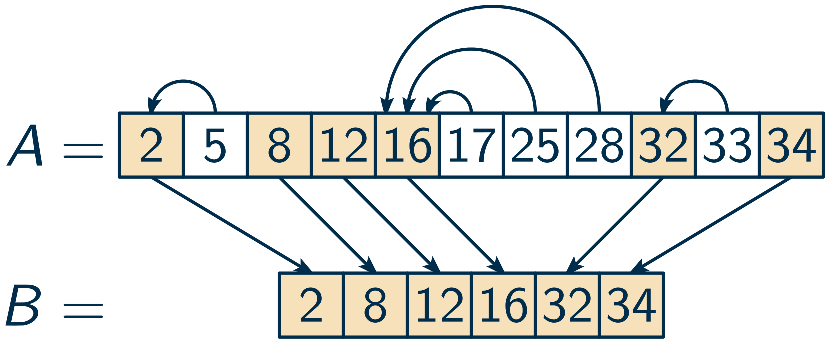
- search in the x-tree $\rightarrow O(\log n)$
  - finds all points with x-coordinate in $[a_1, b_1]$
  - implicit representation via $O(\log n)$ subtrees
- binary search with respect to y
  - one search (or two) for each subtree
  - $O(\log n)$ per search $\rightarrow O(\log^2 n)$

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 points
Problem: result potentially contains points not in the x-range
Idea: search the position in the superset but list the result in the correct subsets

Searching In A Superset



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 $\rightarrow$ find x in B in $O(1)$

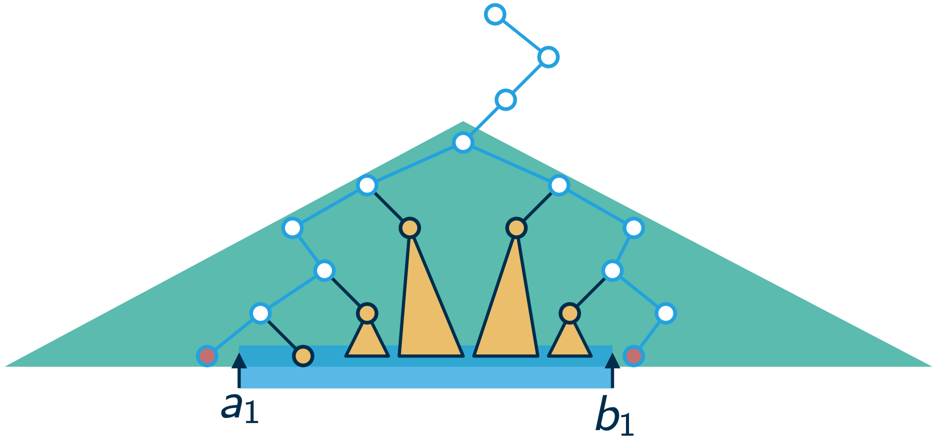
Case 2: $x \in A$ but $x \notin B$

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

And Now For Range-Trees

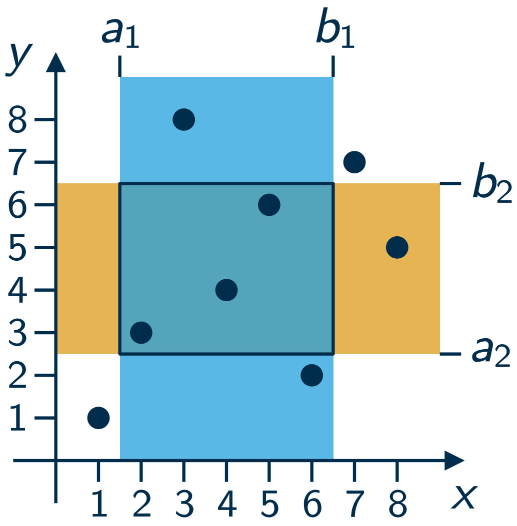


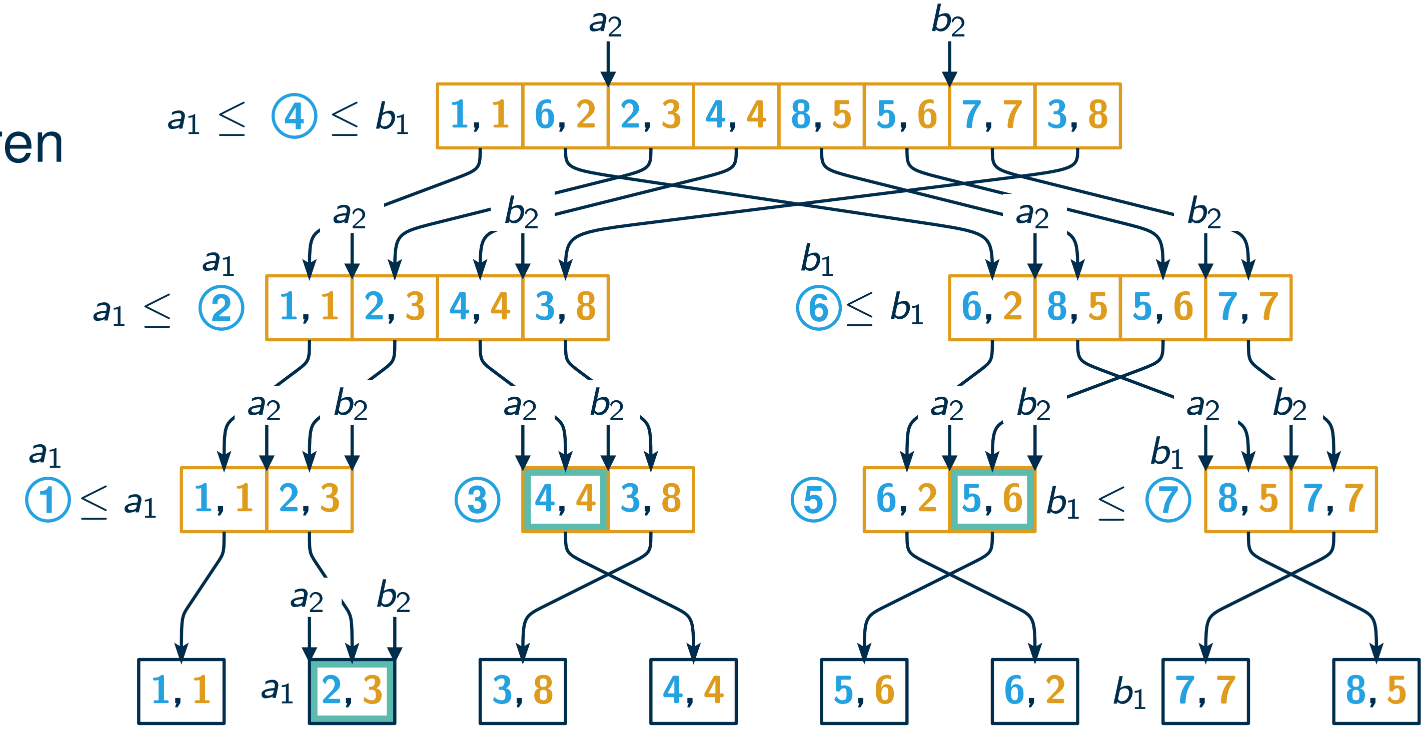
Plan

- 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





Does This All Work Now?

Range Query

- search in y-Array at the root $O(\log n)$
- walk down the x-tree ($\log n$ steps) $O(\log n)$
  - decision for left or right $O(1)$
  - finding the range $[a_2, b_2]$ in the y-array (without searching) $O(1)$
- output result $O(k)$

Memory: only constant factor overhead for each y-array

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

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.

Wrap-Up

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

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

Similar Data Structures

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