

# Algorithmische Geometrie

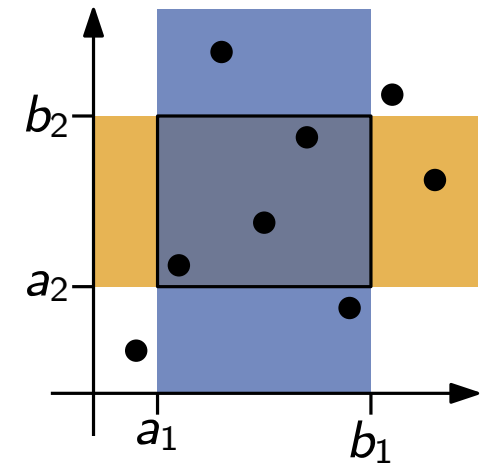
## Orthogonale Bereichsanfragen – Range-Trees



# Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .



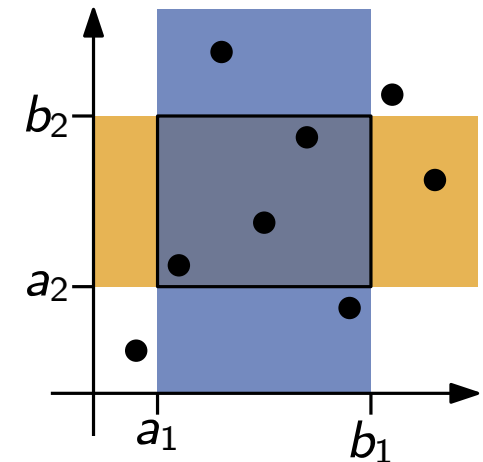
# Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Statisch Variante

- Punktmenge  $P$  ist fest
- viele verschiedene Bereichsanfragen



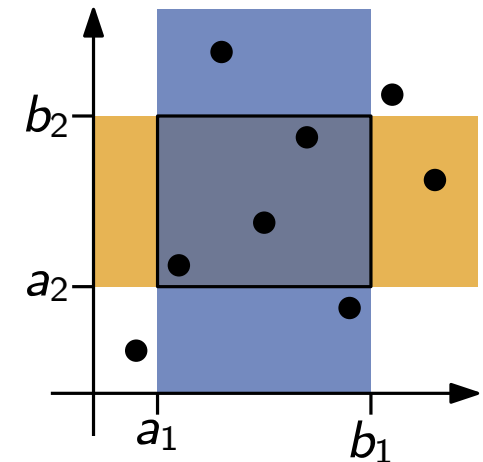
# Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktemenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Statisch Variante

- Punktemenge  $P$  ist fest
- viele verschiedene Bereichsanfragen
- entwickle Datenstruktur auf  $P$ , sodass
  - jede Anfrage ist schnell



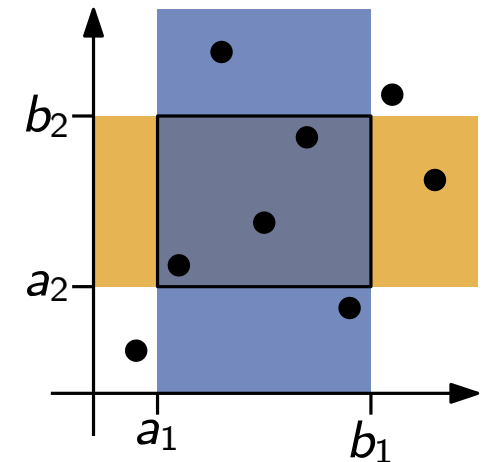
# Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktemenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Statisch Variante

- Punktemenge  $P$  ist fest
- viele verschiedene Bereichsanfragen
- entwickle Datenstruktur auf  $P$ , sodass
  - jede Anfrage ist schnell
  - Datenstruktur kann schnell aufgebaut werden



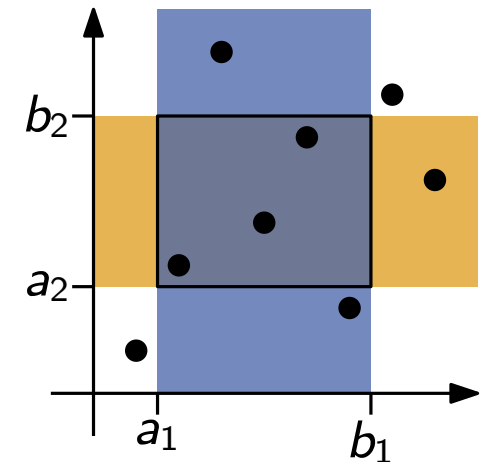
# Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Statisch Variante

- Punktmenge  $P$  ist fest
- viele verschiedene Bereichsanfragen
- entwickle Datenstruktur auf  $P$ , sodass
  - jede Anfrage ist schnell
  - Datenstruktur kann schnell aufgebaut werden
  - Datenstruktur benötigt wenig Platz



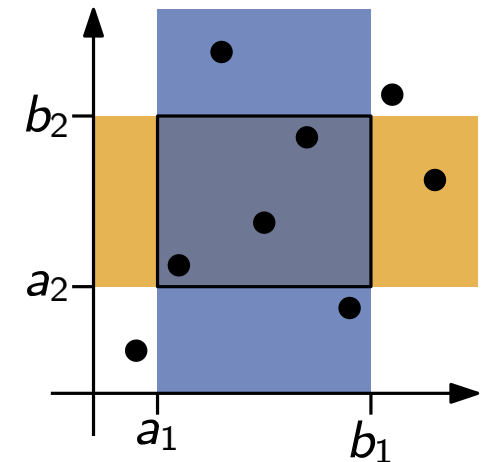
# Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktemenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Statisch Variante

- Punktemenge  $P$  ist fest
- viele verschiedene Bereichsanfragen
- entwickle Datenstruktur auf  $P$ , sodass
  - jede Anfrage ist schnell
  - Datenstruktur kann schnell aufgebaut werden
  - Datenstruktur benötigt wenig Platz



Was sind mögliche Anwendungen?

# 1D-Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Einfachster Fall: $d = 1$

- die Punkte sind einfach nur Zahlen
- wir suchen alle Zahlen in einem gegebenen Intervall



# 1D-Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Einfachster Fall: $d = 1$

- die Punkte sind einfach nur Zahlen
- wir suchen alle Zahlen in einem gegebenen Intervall

## Lösung 1

# 1D-Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Einfachster Fall: $d = 1$

- die Punkte sind einfach nur Zahlen
- wir suchen alle Zahlen in einem gegebenen Intervall

## Lösung 1

- Vorbereitung: sortiere die Zahlen  $\rightarrow O(n \log n)$ 

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---
- Anfrage: binärer Suche  $\rightarrow O(\log n + k)$     ( $k = \text{Ausgabegröße}$ )

# 1D-Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Einfachster Fall: $d = 1$

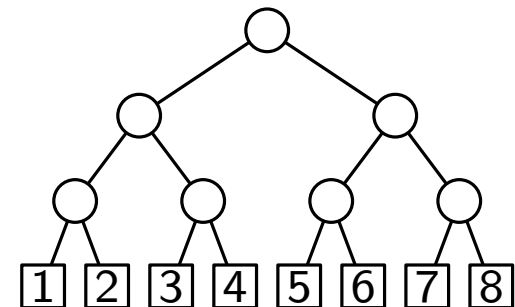
- die Punkte sind einfach nur Zahlen
- wir suchen alle Zahlen in einem gegebenen Intervall

## Lösung 1

- Vorbereitung: sortiere die Zahlen  $\rightarrow O(n \log n)$     1 2 3 4 5 6 7 8
- Anfrage: binärer Suche  $\rightarrow O(\log n + k)$     ( $k = \text{Ausgabegröße}$ )

## Lösung 2

- binärer Suchbaum mit Punkten in den Blättern
- Anfrage: suche im Suchbaum



# 1D-Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Einfachster Fall: $d = 1$

- die Punkte sind einfach nur Zahlen
- wir suchen alle Zahlen in einem gegebenen Intervall

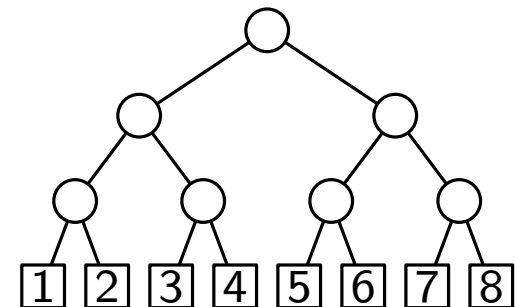
## Lösung 1

- Vorbereitung: sortiere die Zahlen  $\rightarrow O(n \log n)$     1 2 3 4 5 6 7 8
- Anfrage: binärer Suche  $\rightarrow O(\log n + k)$     ( $k = \text{Ausgabegröße}$ )

## Lösung 2

- binärer Suchbaum mit Punkten in den Blättern
- Anfrage: suche im Suchbaum

Wie suchen wir?



# 1D-Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktmenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Einfachster Fall: $d = 1$

- die Punkte sind einfach nur Zahlen
- wir suchen alle Zahlen in einem gegebenen Intervall

## Lösung 1

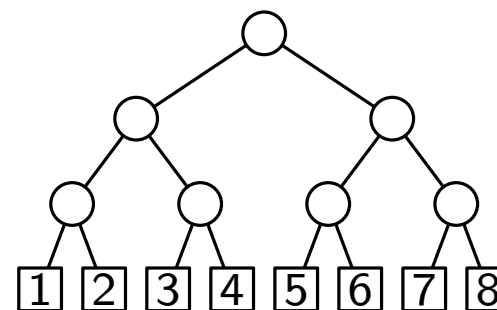
- Vorbereitung: sortiere die Zahlen  $\rightarrow O(n \log n)$     1 2 3 4 5 6 7 8
- Anfrage: binärer Suche  $\rightarrow O(\log n + k)$     ( $k = \text{Ausgabegröße}$ )

## Lösung 2

- binärer Suchbaum mit Punkten in den Blättern
- Anfrage: suche im Suchbaum

Wie suchen wir?

Welche Werte haben die inneren Knoten?



# 1D-Bereichsanfragen

## Problem: Bereichsanfragen

Gegeben eine Punktemenge  $P \in \mathbb{R}^d$  sowie eine Box  $B = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ , finde alle Punkte in  $P \cap B$ .

## Einfachster Fall: $d = 1$

- die Punkte sind einfach nur Zahlen
- wir suchen alle Zahlen in einem gegebenen Intervall

## Lösung 1

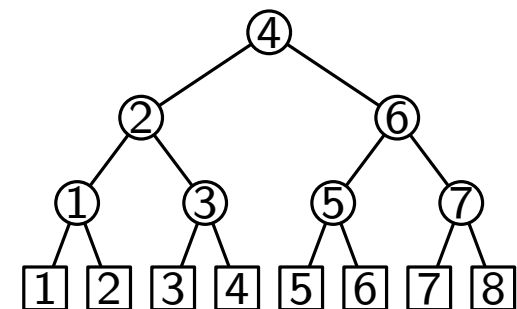
- Vorbereitung: sortiere die Zahlen  $\rightarrow O(n \log n)$      1 2 3 4 5 6 7 8
- Anfrage: binärer Suche  $\rightarrow O(\log n + k)$      ( $k = \text{Ausgabegröße}$ )

## Lösung 2

- binärer Suchbaum mit Punkten in den Blättern
- Anfrage: suche im Suchbaum

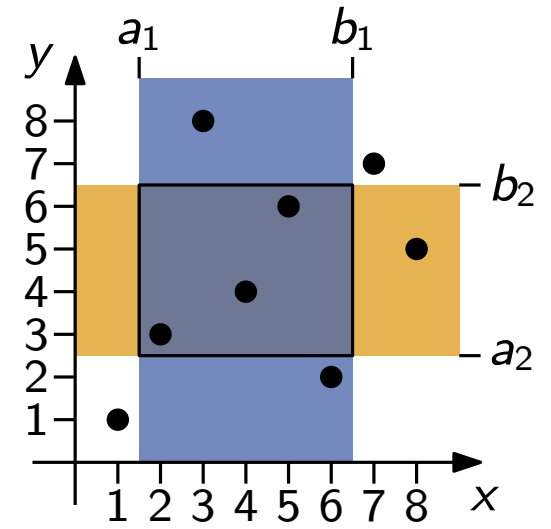
Wie suchen wir?

Welche Werte haben die inneren Knoten?



# 2D-Bereichsanfragen

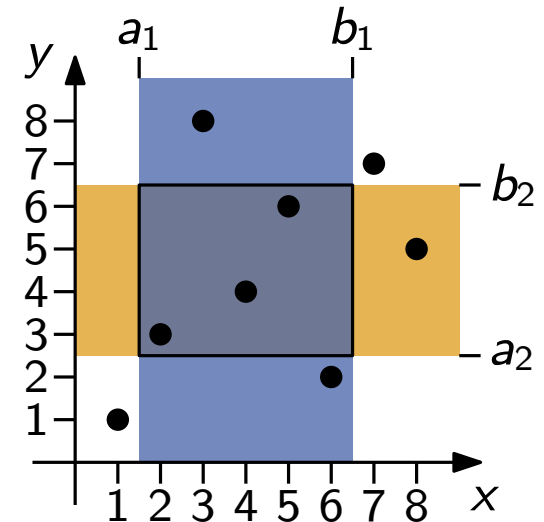
## Idee



# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )



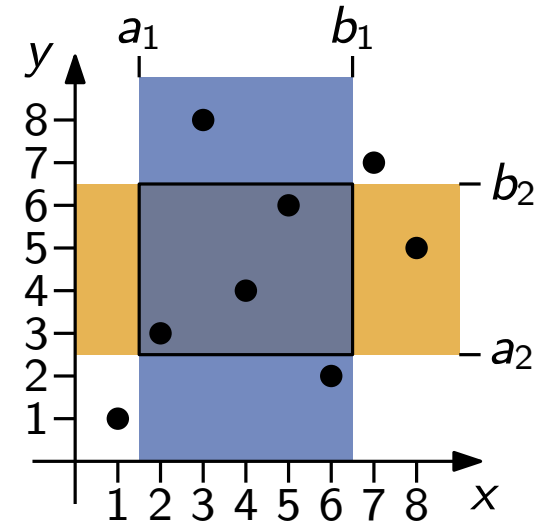


# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

suche nach  $x \in [a_1, b_1]$ :



# 2D-Bereichsanfragen

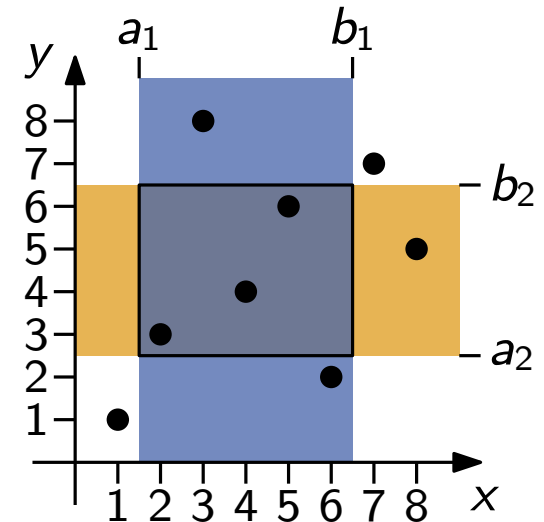
## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

suche nach  $x \in [a_1, b_1]$ :



## Problem



# 2D-Bereichsanfragen

## Idee

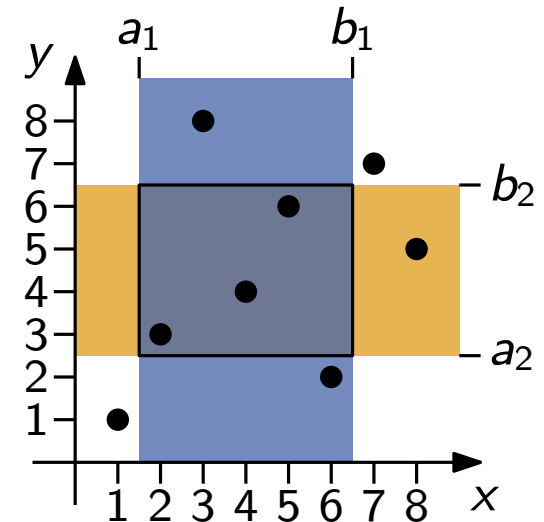
- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

suche nach  $x \in [a_1, b_1]$ : 

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

## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte



# 2D-Bereichsanfragen

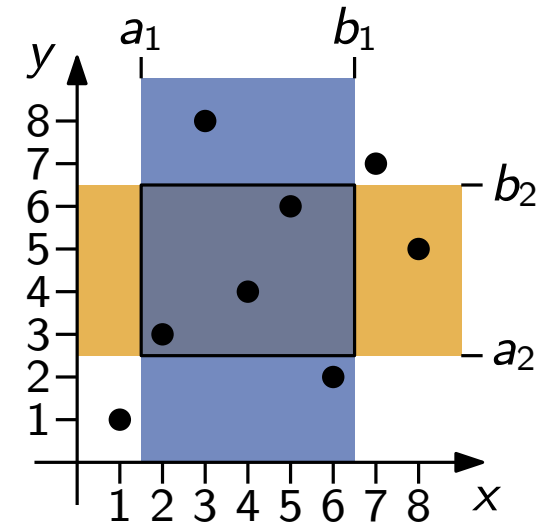
## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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

## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen



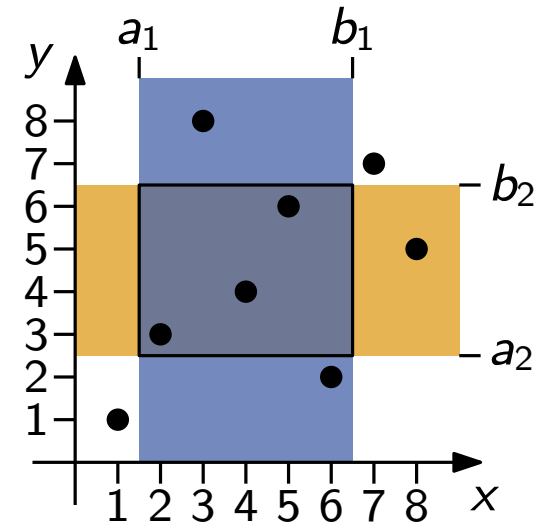
Warum nicht?

# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

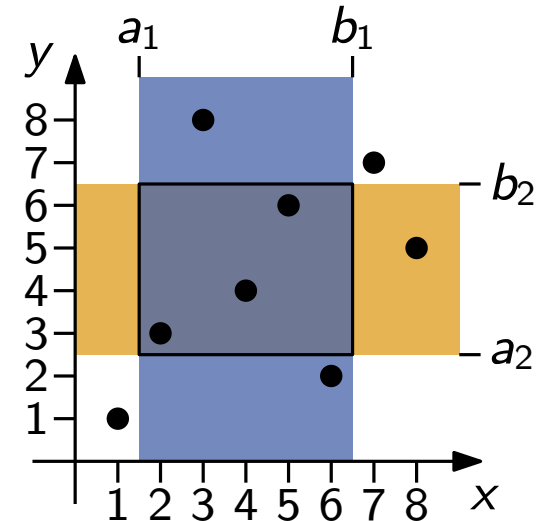
- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen

# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



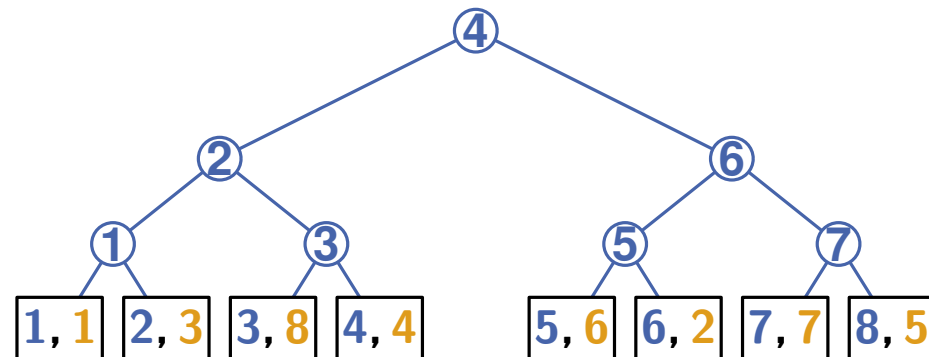
## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen

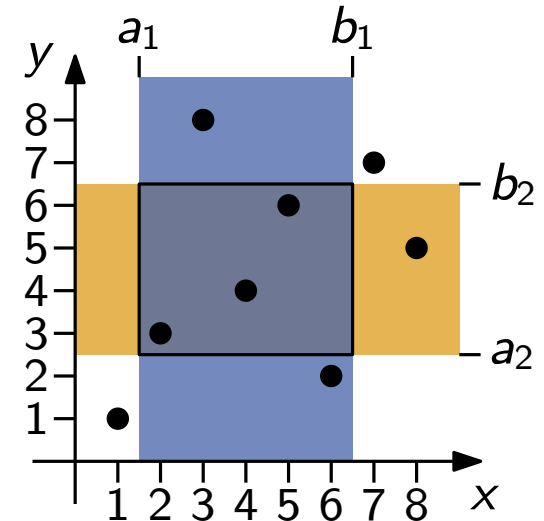


# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



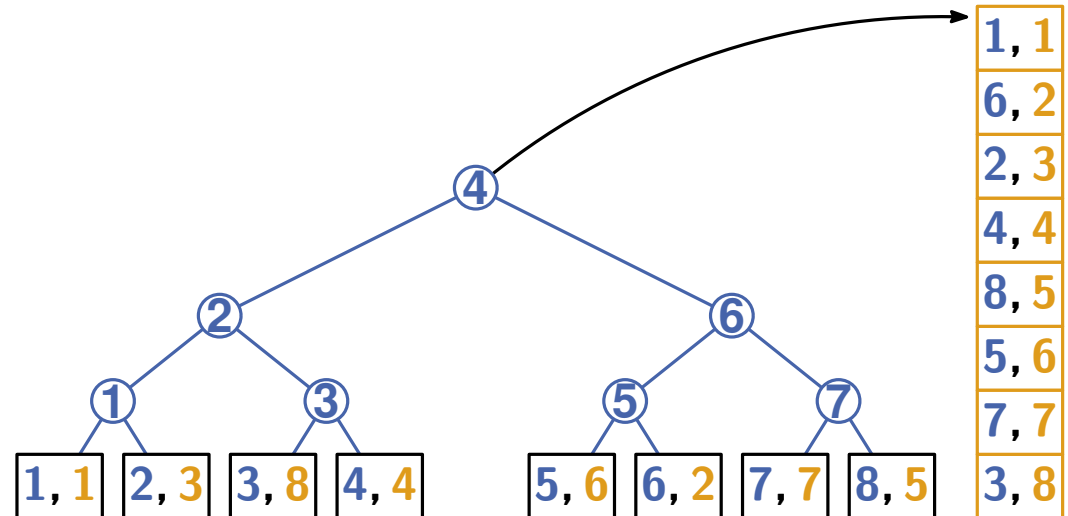
## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen

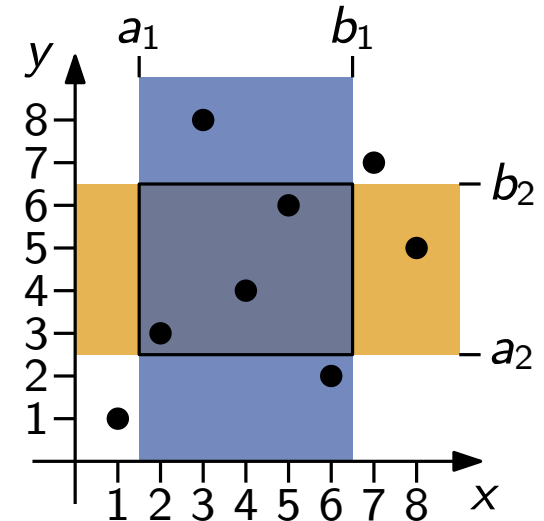


# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



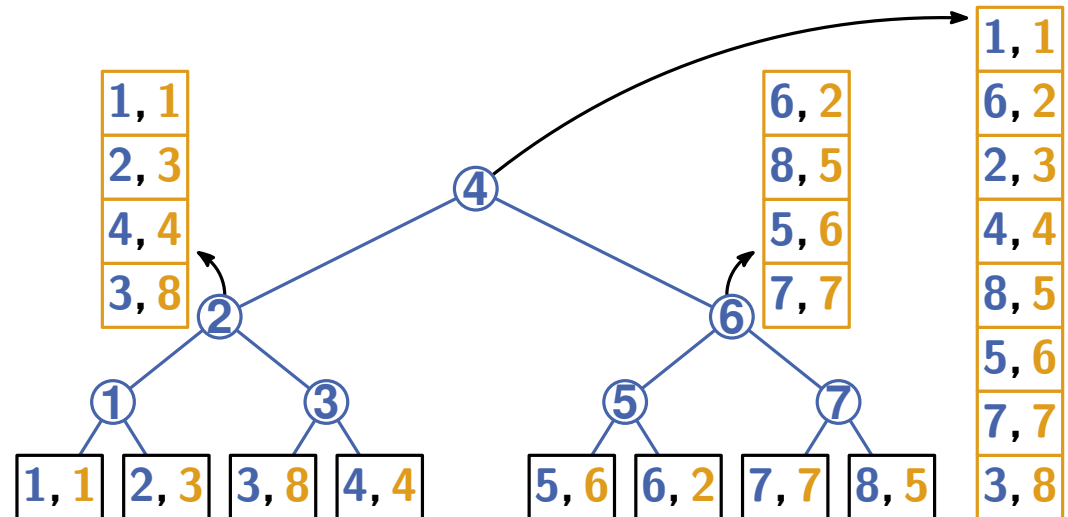
## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen



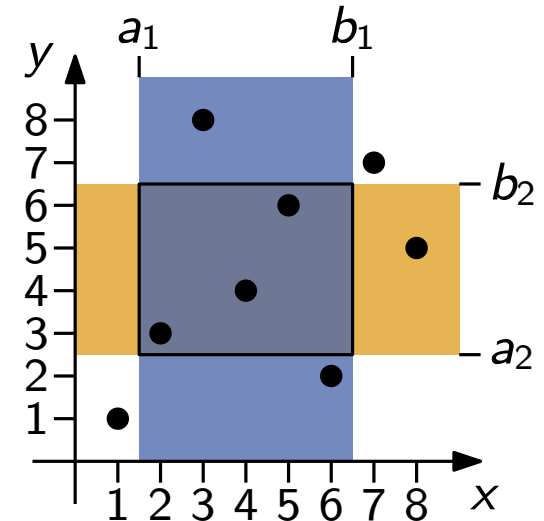


# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



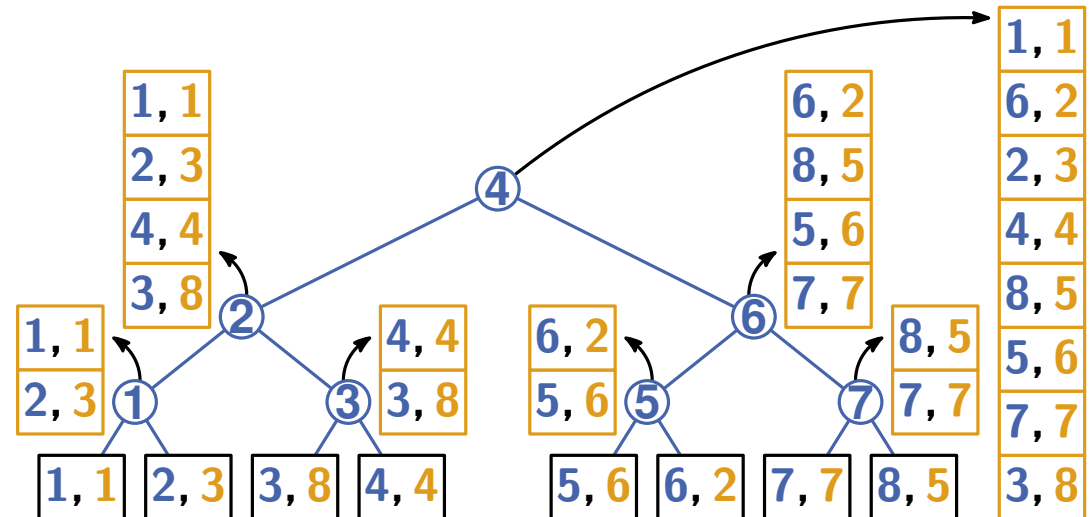
## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen

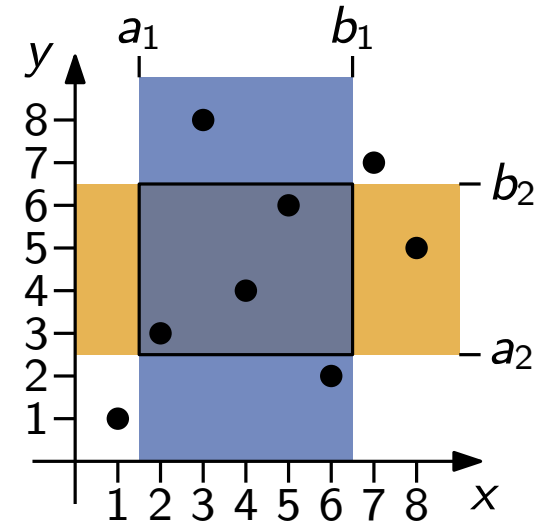


# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



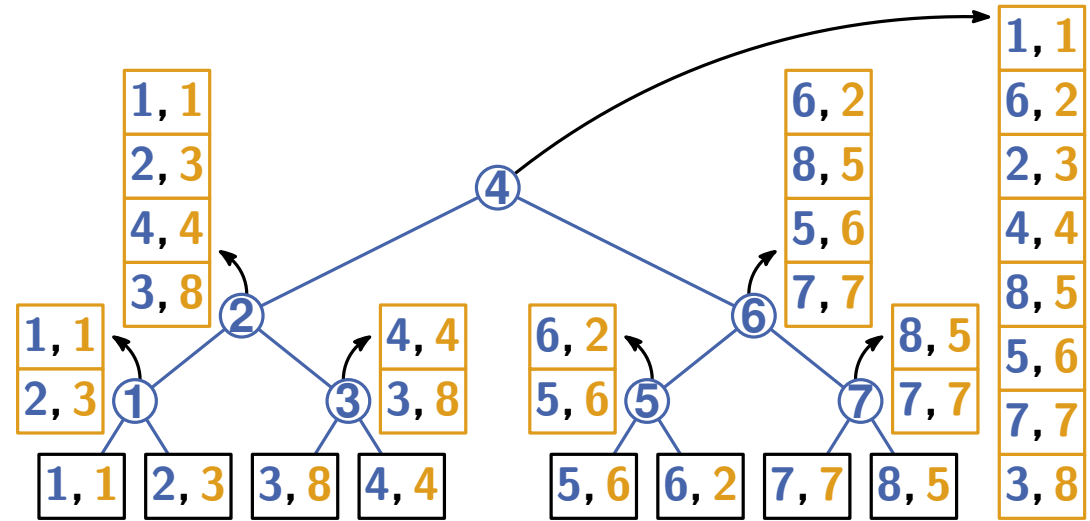
## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree

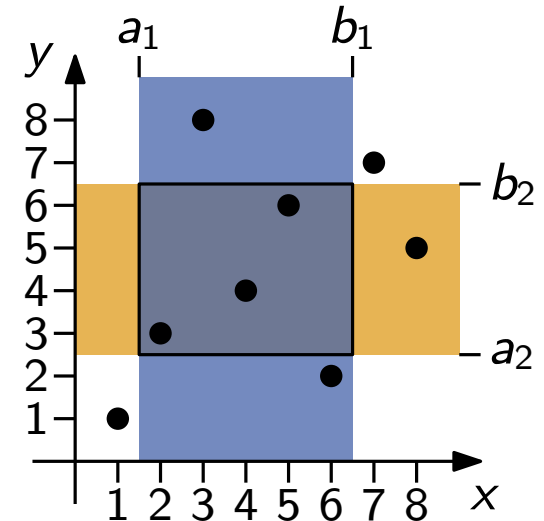


# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



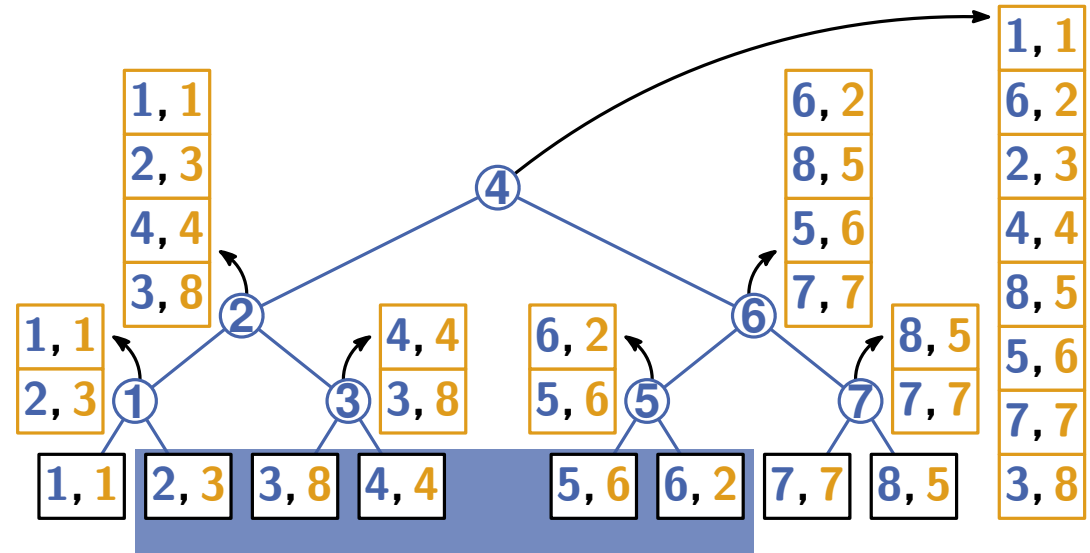
## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree

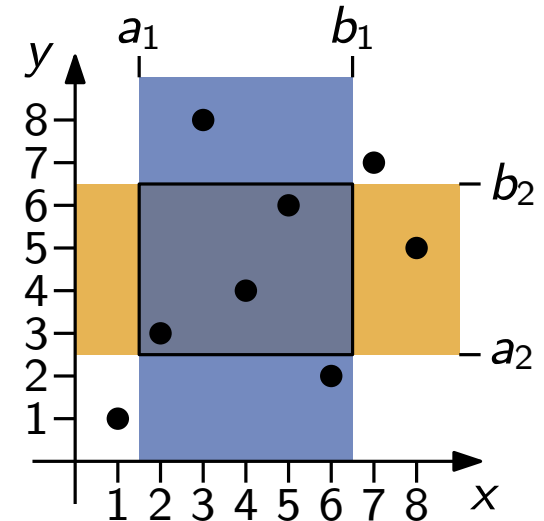


# 2D-Bereichsanfragen

## Idee

- suche zunächst in der ersten Dimension ( $x$ )
- suche auf dem Ergebnis in zweiter Dimension ( $y$ )

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



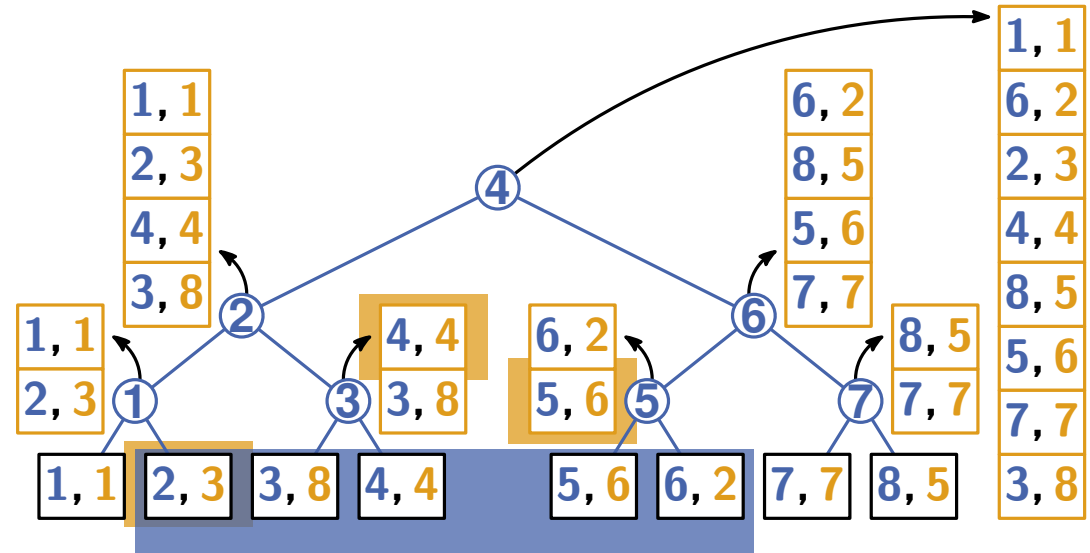
## Problem

- $y$ -Suche läuft auf einer Teilmenge der Punkte
- wir können nicht für jede mögliche Teilmenge ein  $y$ -sortiertes Array bauen

Warum nicht?

## Idee

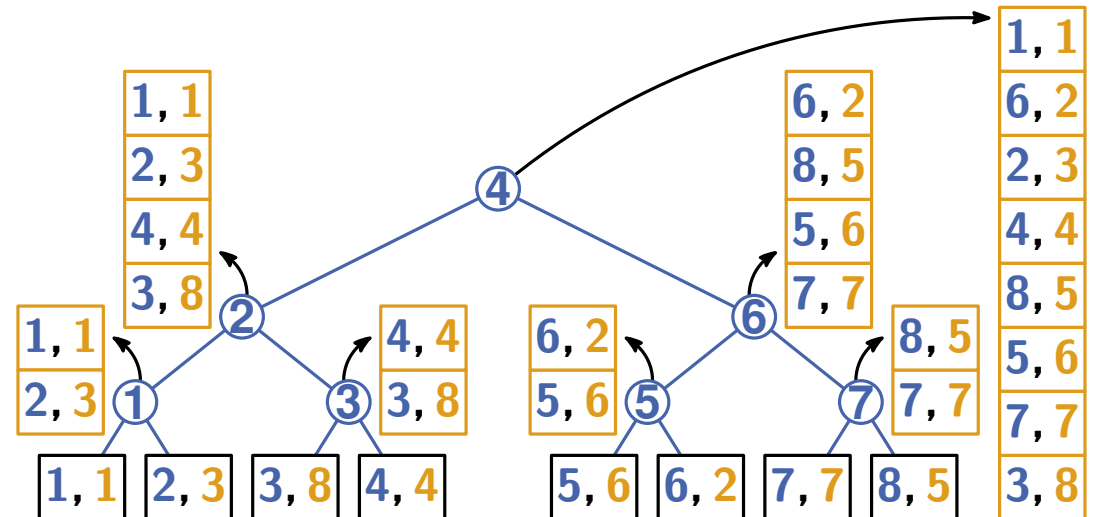
- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



# Bereichsanfrage in einem 2D-Range-Tree

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree

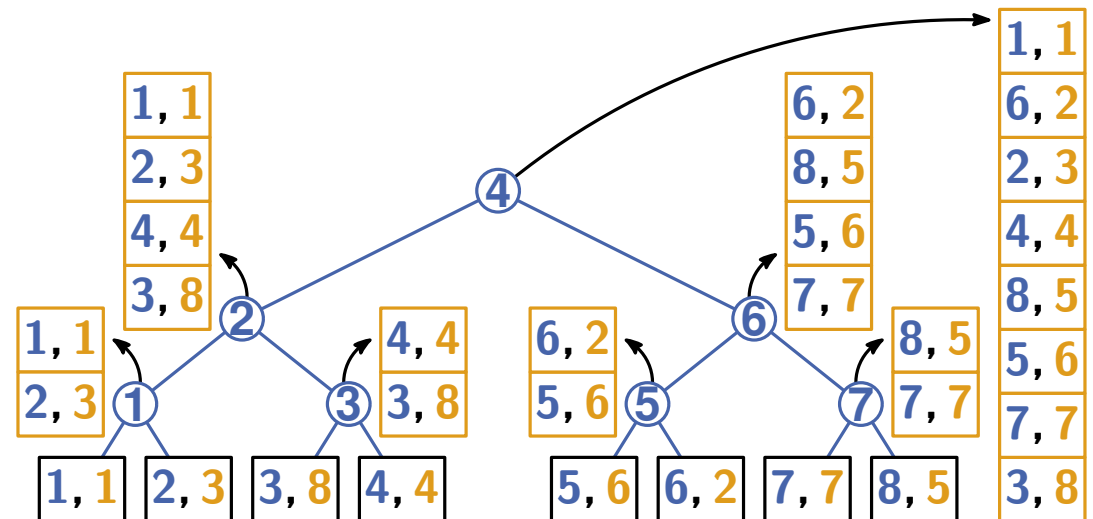


# Bereichsanfrage in einem 2D-Range-Tree

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree

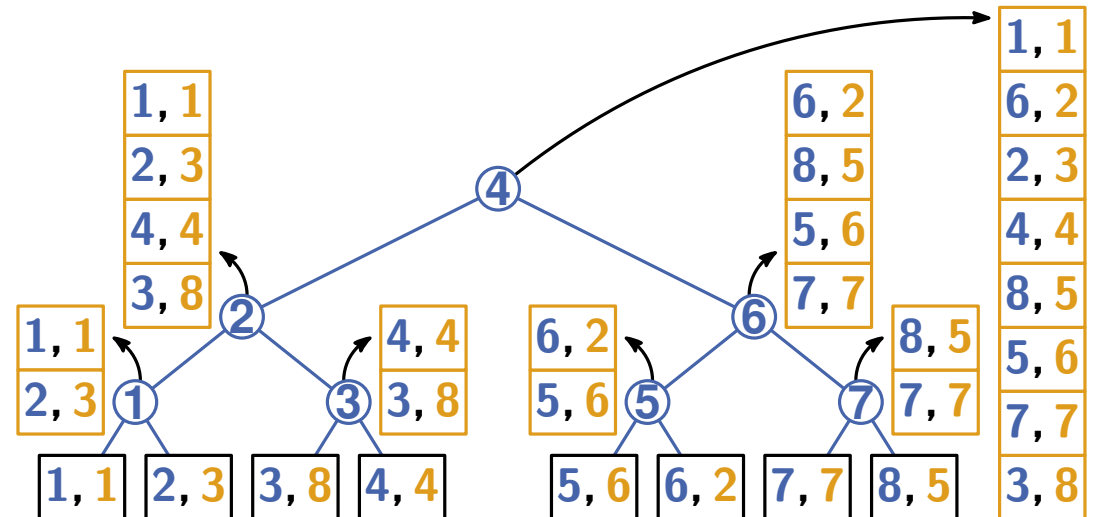
Anfrage  $[a_1, b_1] \times [a_2, b_2]$



# Bereichsanfrage in einem 2D-Range-Tree

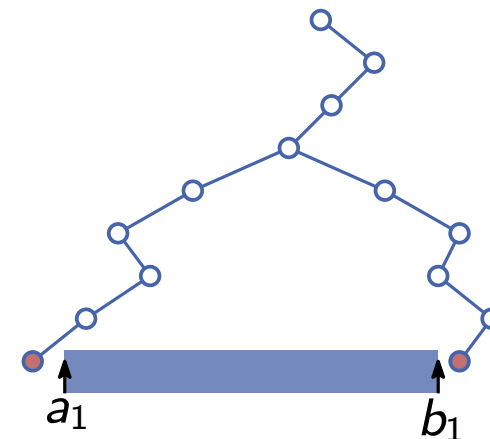
## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



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

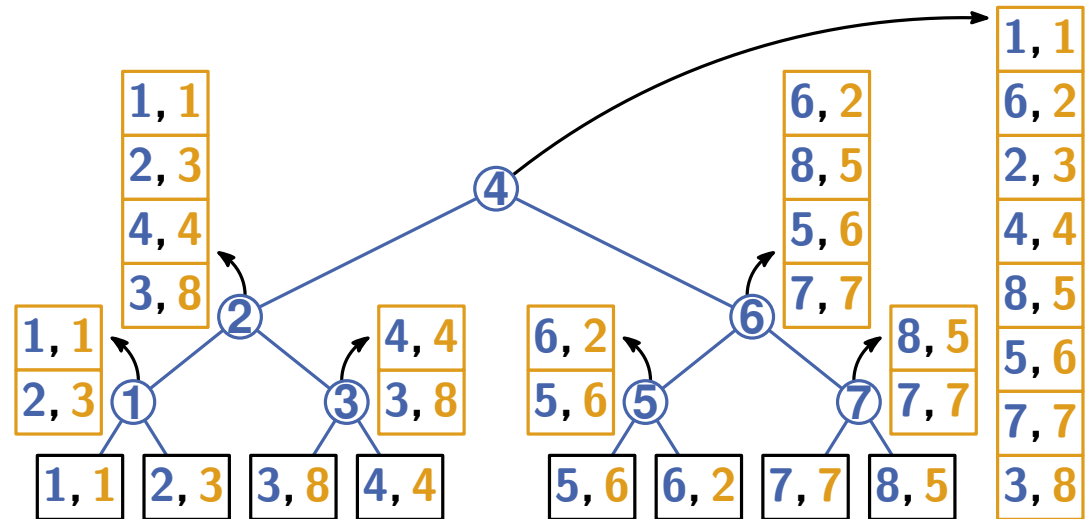
- suche nach Vorgänger von  $a_1$  und Nachfolger von  $b_1$  im  $x$ -Baum



# Bereichsanfrage in einem 2D-Range-Tree

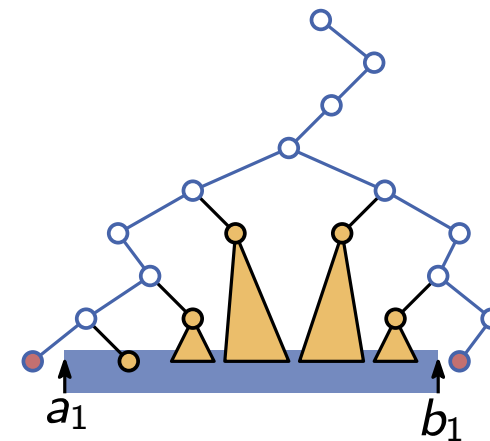
## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



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

- suche nach Vorgänger von  $a_1$  und Nachfolger von  $b_1$  im  $x$ -Baum
- für Knoten direkt unter dem Pfad: suche in entsprechenden  $y$ -Array nach  $[a_2, b_2]$  und gib gefundene Punkte aus

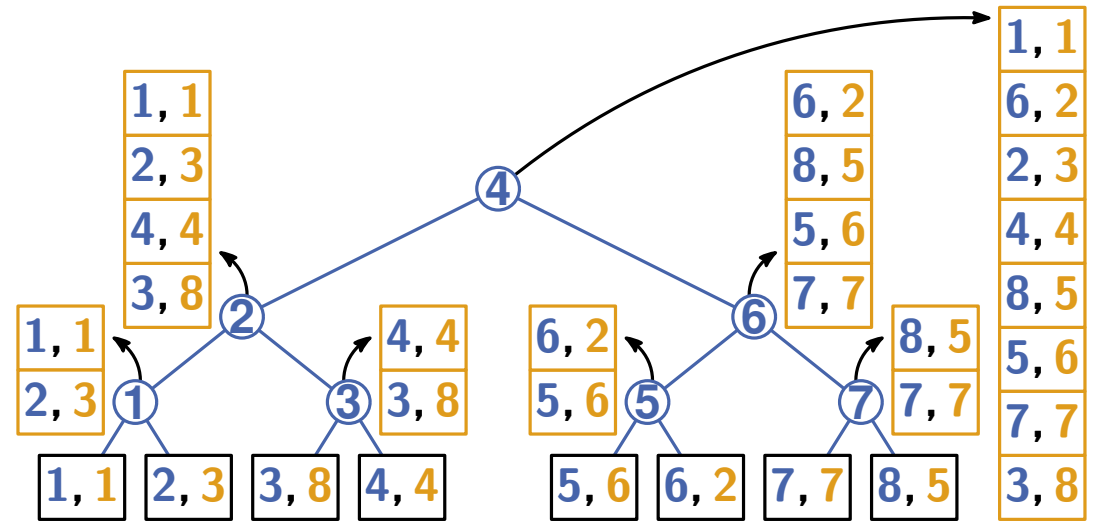




# Bereichsanfrage in einem 2D-Range-Tree

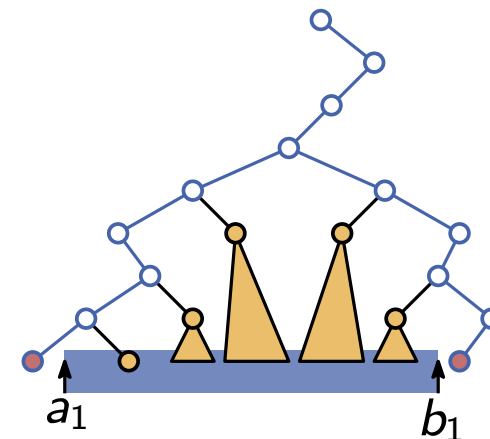
## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



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

- suche nach Vorgänger von  $a_1$  und Nachfolger von  $b_1$  im  $x$ -Baum
- für Knoten direkt unter dem Pfad: suche in entsprechenden  $y$ -Array nach  $[a_2, b_2]$  und gib gefundene Punkte aus

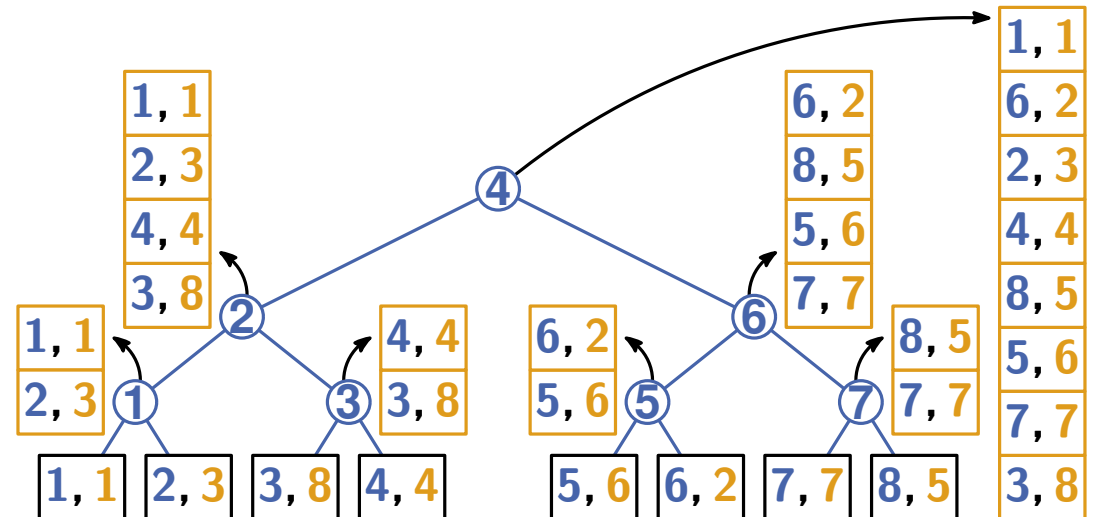


## Laufzeit einer Anfrage

# Bereichsanfrage in einem 2D-Range-Tree

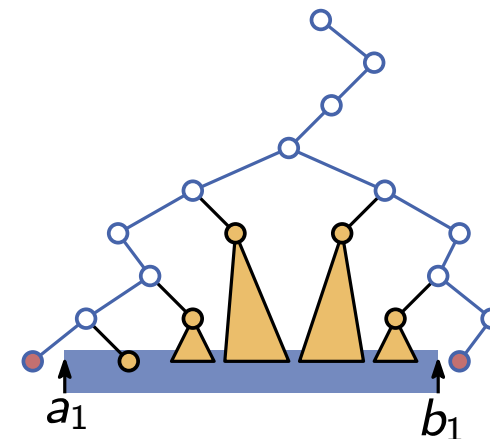
## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



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

- suche nach Vorgänger von  $a_1$  und Nachfolger von  $b_1$  im  $x$ -Baum
- für Knoten direkt unter dem Pfad: suche in entsprechenden  $y$ -Array nach  $[a_2, b_2]$  und gib gefundene Punkte aus



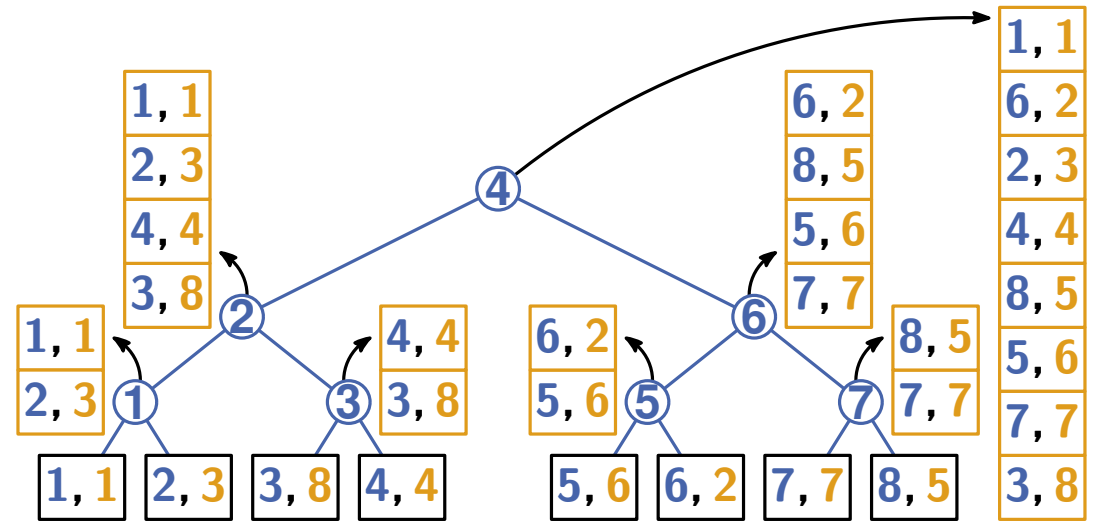
## Laufzeit einer Anfrage

- suche im  $x$ -Baum  $\rightarrow O(\log n)$

# Bereichsanfrage in einem 2D-Range-Tree

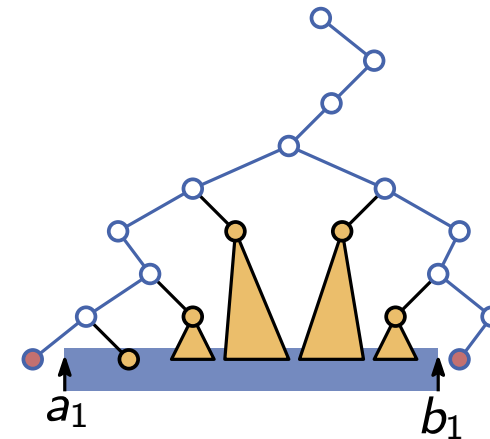
## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



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

- suche nach Vorgänger von  $a_1$  und Nachfolger von  $b_1$  im  $x$ -Baum
- für Knoten direkt unter dem Pfad: suche in entsprechenden  $y$ -Array nach  $[a_2, b_2]$  und gib gefundene Punkte aus



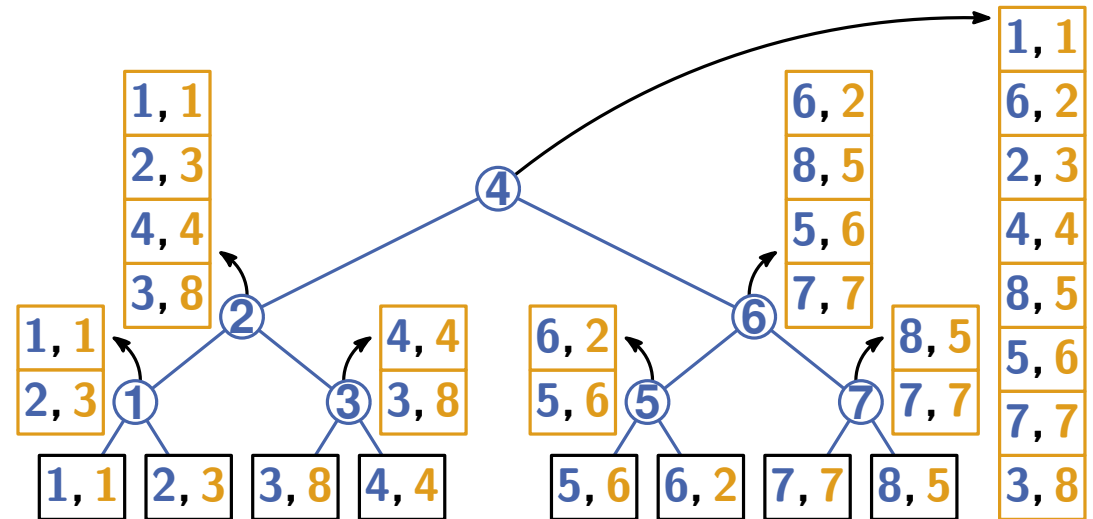
## Laufzeit einer Anfrage

- suche im  $x$ -Baum  $\rightarrow O(\log n)$
- suche in  $O(\log n)$   $y$ -Arrays  $\rightarrow O(\log^2 n)$

# Bereichsanfrage in einem 2D-Range-Tree

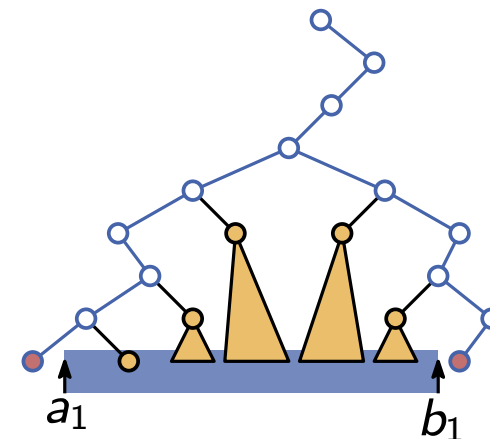
## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



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

- suche nach Vorgänger von  $a_1$  und Nachfolger von  $b_1$  im  $x$ -Baum
- für Knoten direkt unter dem Pfad: suche in entsprechenden  $y$ -Array nach  $[a_2, b_2]$  und gib gefundene Punkte aus



## Laufzeit einer Anfrage

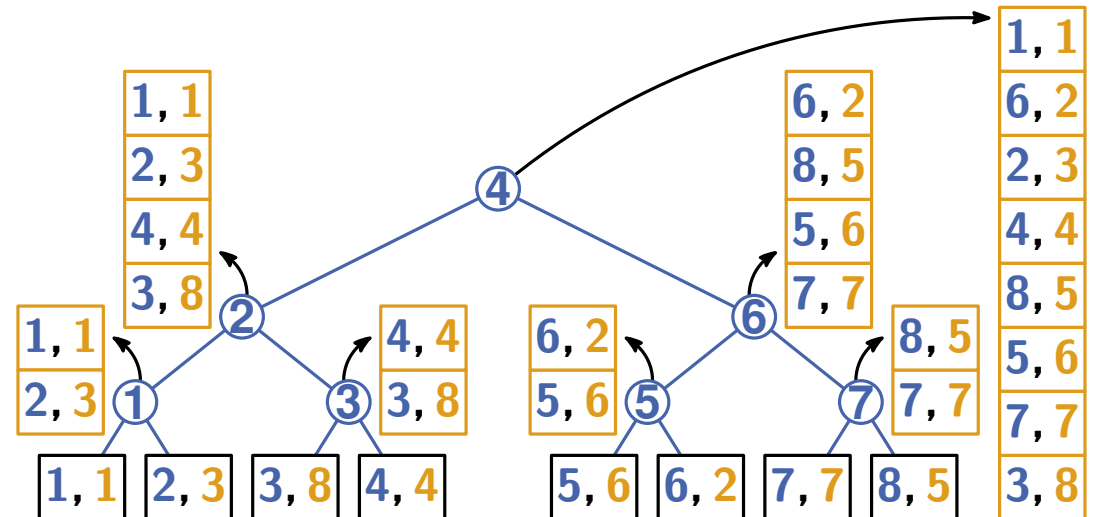
- suche im  $x$ -Baum  $\rightarrow O(\log n)$
- suche in  $O(\log n)$   $y$ -Arrays  $\rightarrow O(\log^2 n)$

Die Teilbäume sind disjunkt!  
Wird es dadurch besser als  $\log^2 n$ ?

# Bereichsanfrage in einem 2D-Range-Tree

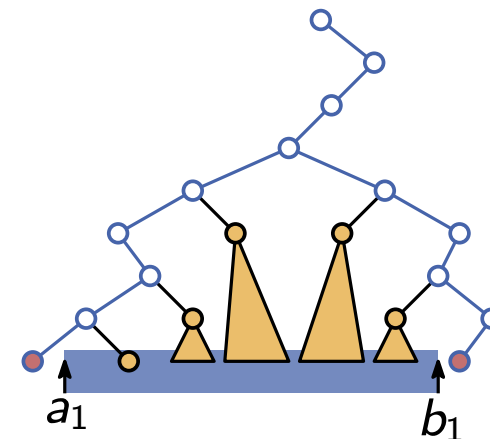
## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



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

- suche nach Vorgänger von  $a_1$  und Nachfolger von  $b_1$  im  $x$ -Baum
- für Knoten direkt unter dem Pfad: suche in entsprechenden  $y$ -Array nach  $[a_2, b_2]$  und gib gefundene Punkte aus



## Laufzeit einer Anfrage

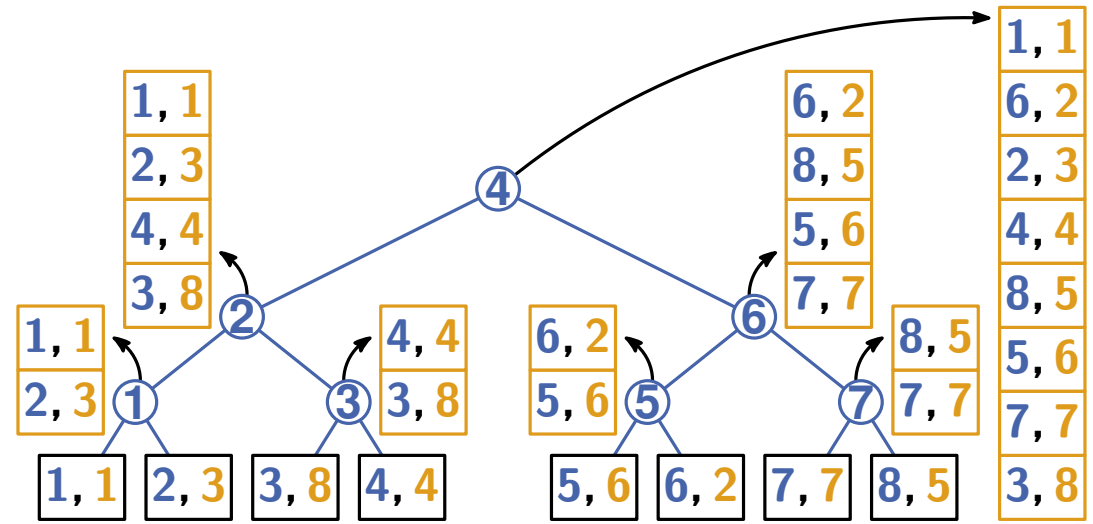
- suche im  $x$ -Baum  $\rightarrow O(\log n)$
- suche in  $O(\log n)$   $y$ -Arrays  $\rightarrow O(\log^2 n)$  (nicht vergessen:  $O(k)$  für die Ausgabe)

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree

## Aufbau



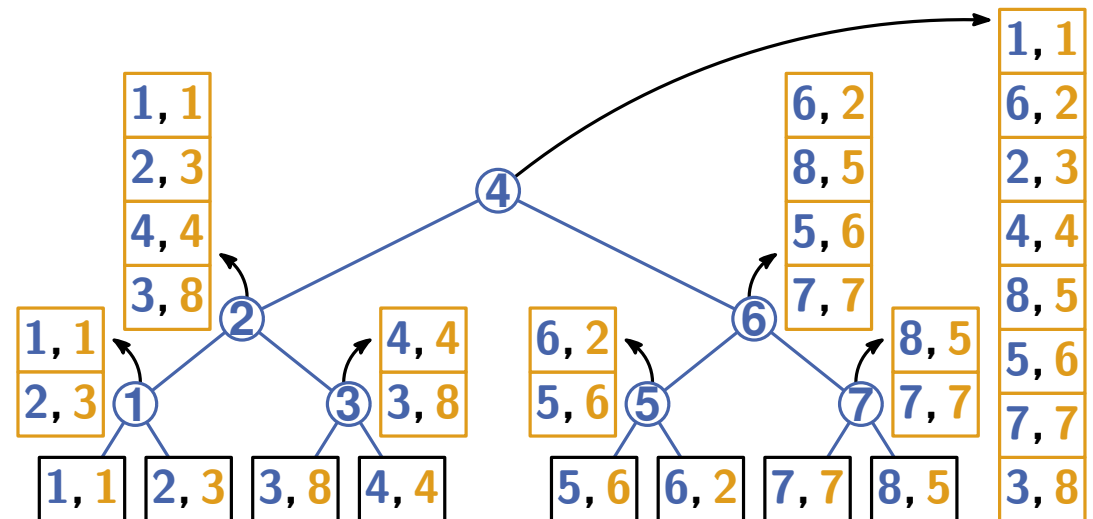
# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree

## Aufbau

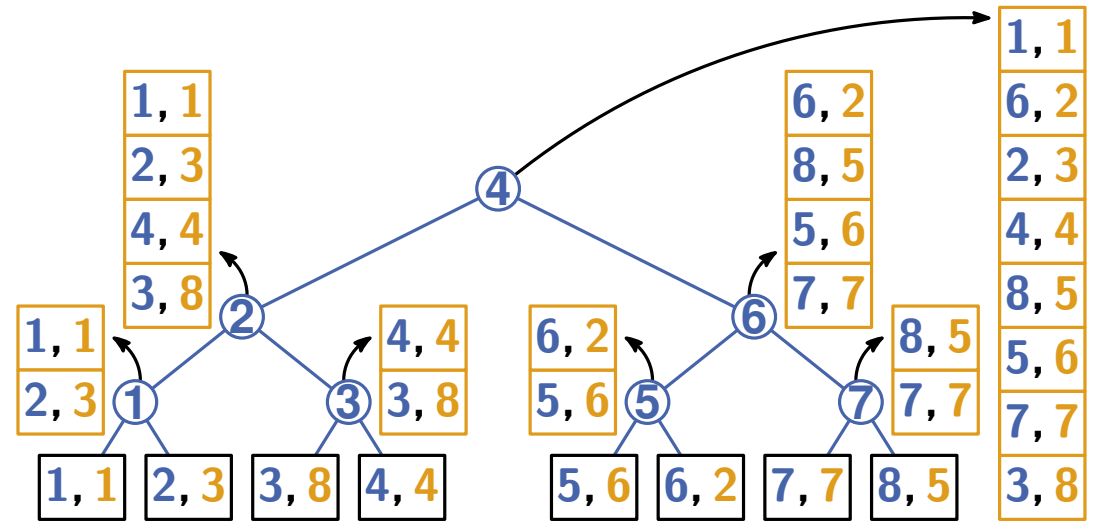
- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$



# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



## Aufbau

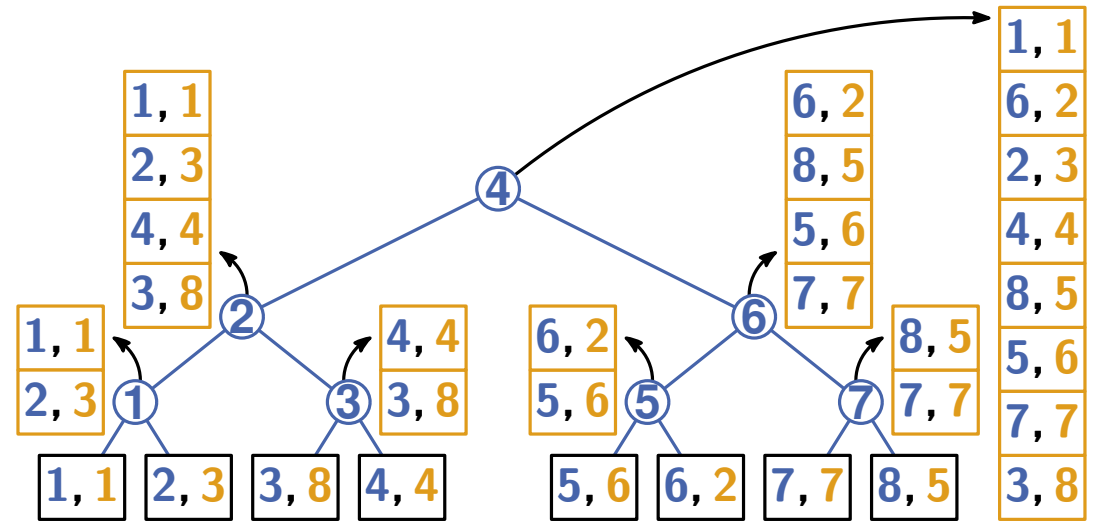
- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:



# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



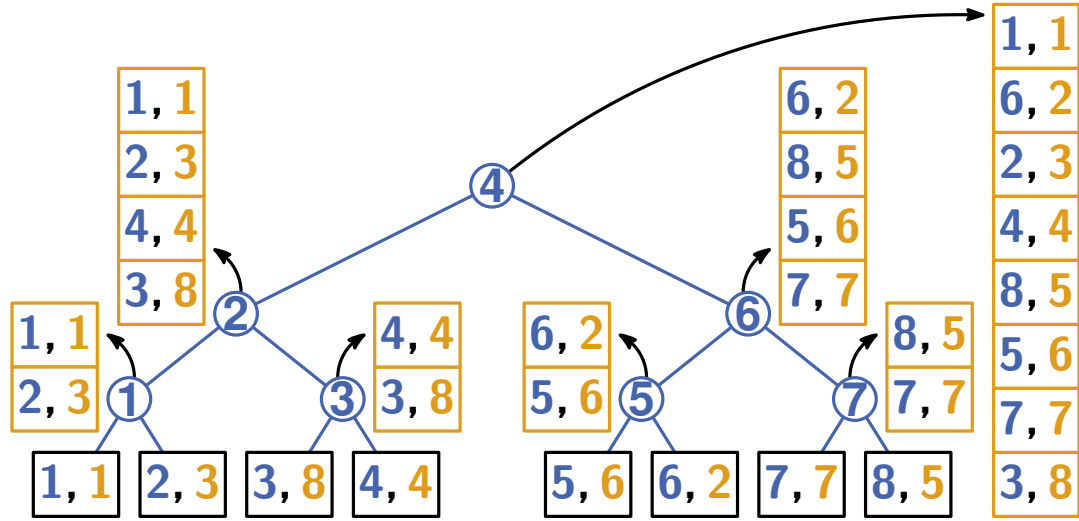
## Aufbau

- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



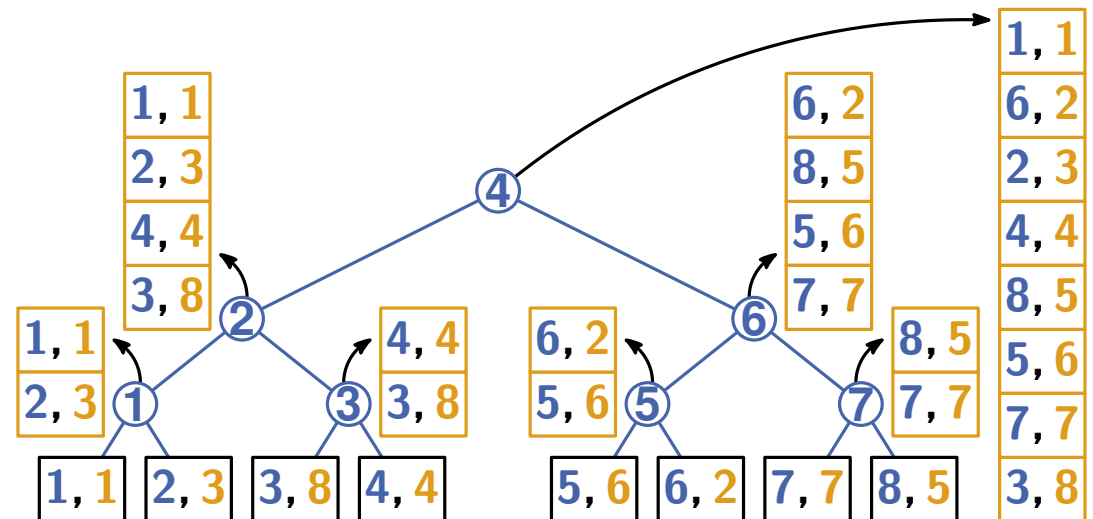
## Aufbau

- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$
- Verbesserung des zweiten Schritts

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



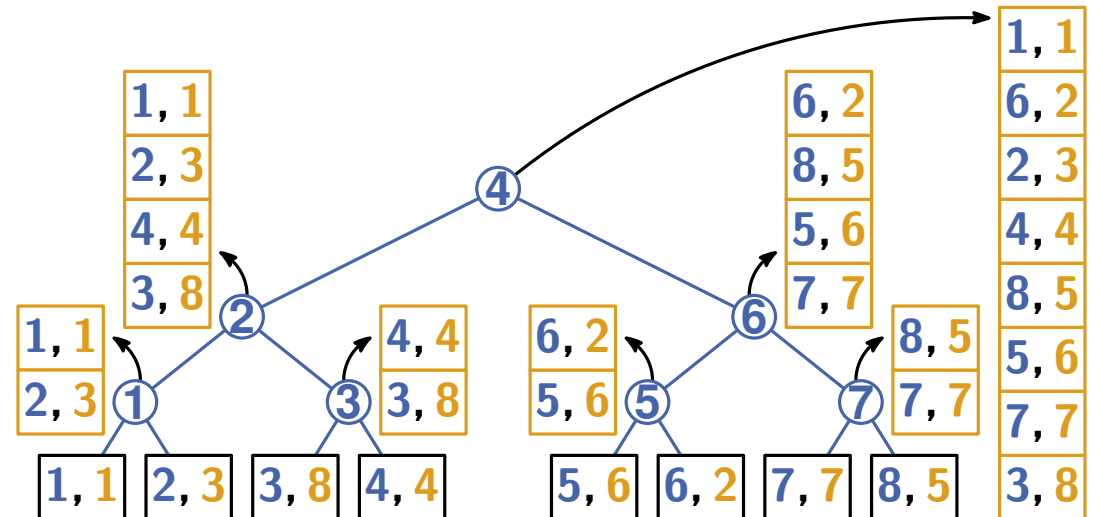
## Aufbau

- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$
- Verbesserung des zweiten Schritts
  - sortiere einmal alle Punkte nach  $y \rightarrow O(n \log n)$

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



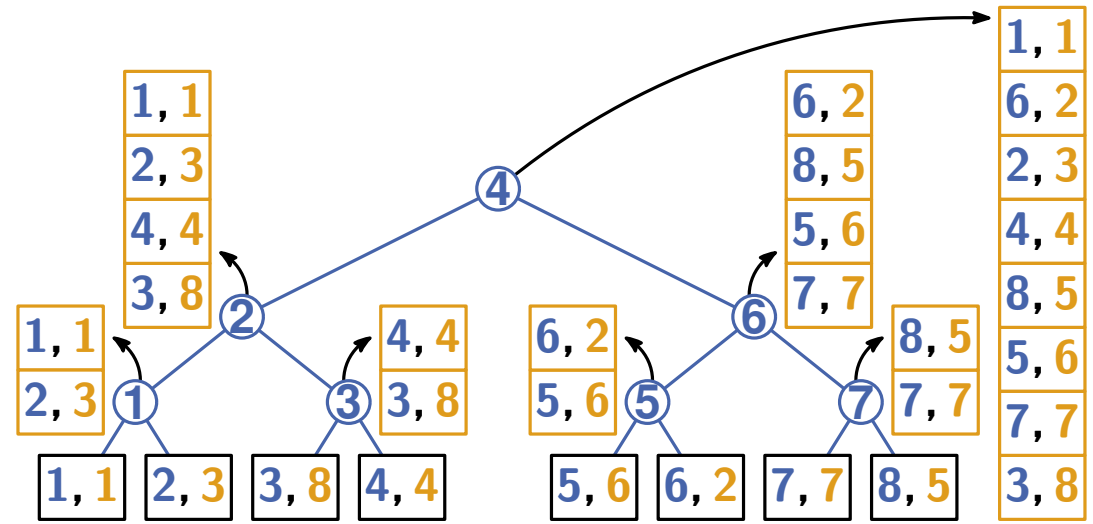
## Aufbau

- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$
- Verbesserung des zweiten Schritts
  - sortiere einmal alle Punkte nach  $y \rightarrow O(n \log n)$
  - spalte sortiertes Array auf um sortierte Arrays für die Kinder zu erhalten

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



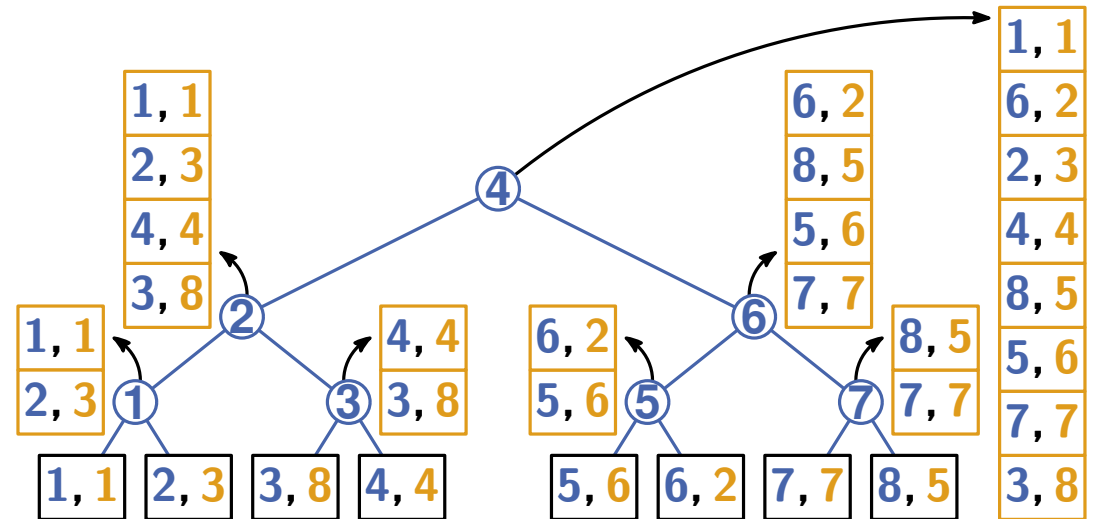
## Aufbau

- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$
- Verbesserung des zweiten Schritts
  - sortiere einmal alle Punkte nach  $y \rightarrow O(n \log n)$
  - spalte sortiertes Array auf um sortierte Arrays für die Kinder zu erhalten
  - $O(n)$  pro Layer  $\rightarrow O(n \log n)$

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



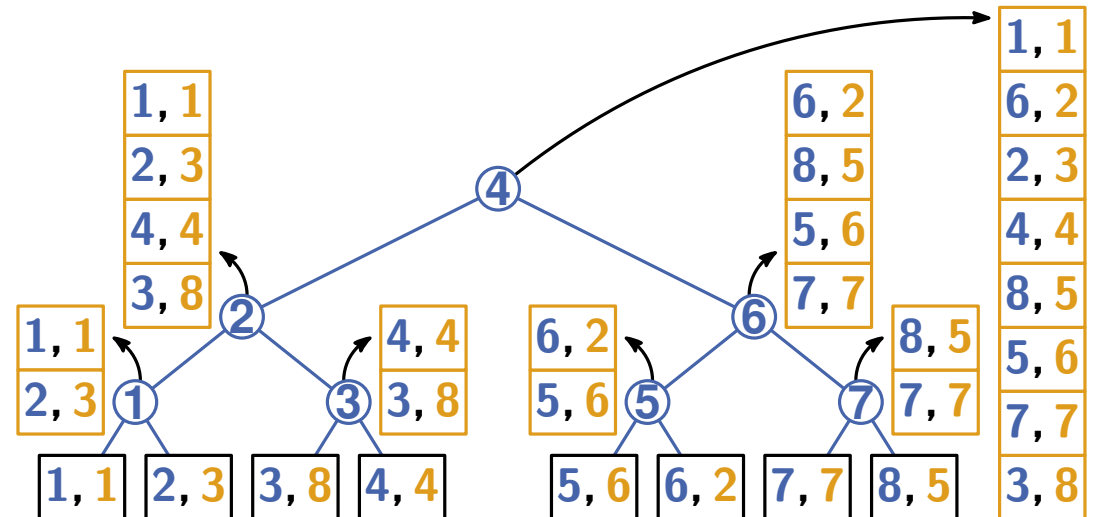
## Aufbau

- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$
- Verbesserung des zweiten Schritts  $\rightarrow O(n \log n)$ 
  - sortiere einmal alle Punkte nach  $y \rightarrow O(n \log n)$
  - spalte sortiertes Array auf um sortierte Arrays für die Kinder zu erhalten
  - $O(n)$  pro Layer  $\rightarrow O(n \log n)$

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



## Aufbau

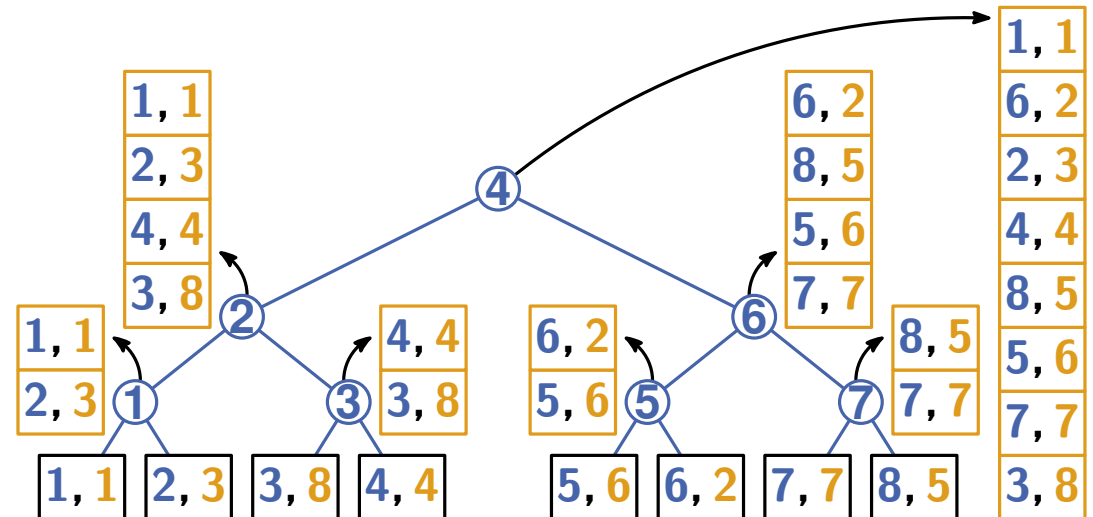
- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$
- Verbesserung des zweiten Schritts  $\rightarrow O(n \log n)$ 
  - sortiere einmal alle Punkte nach  $y \rightarrow O(n \log n)$
  - spalte sortiertes Array auf um sortierte Arrays für die Kinder zu erhalten
  - $O(n)$  pro Layer  $\rightarrow O(n \log n)$

## Speicherverbrauch

# Berechnung eines 2D-Range-Trees

## Idee

- berechne  $y$ -sortiertes Array für wenige wichtige Teilmengen
- Knoten in  $x$ -Suchbaum liefern wichtige Teilmengen
- diese Datenstruktur heißt 2D-Range-Tree



## Aufbau

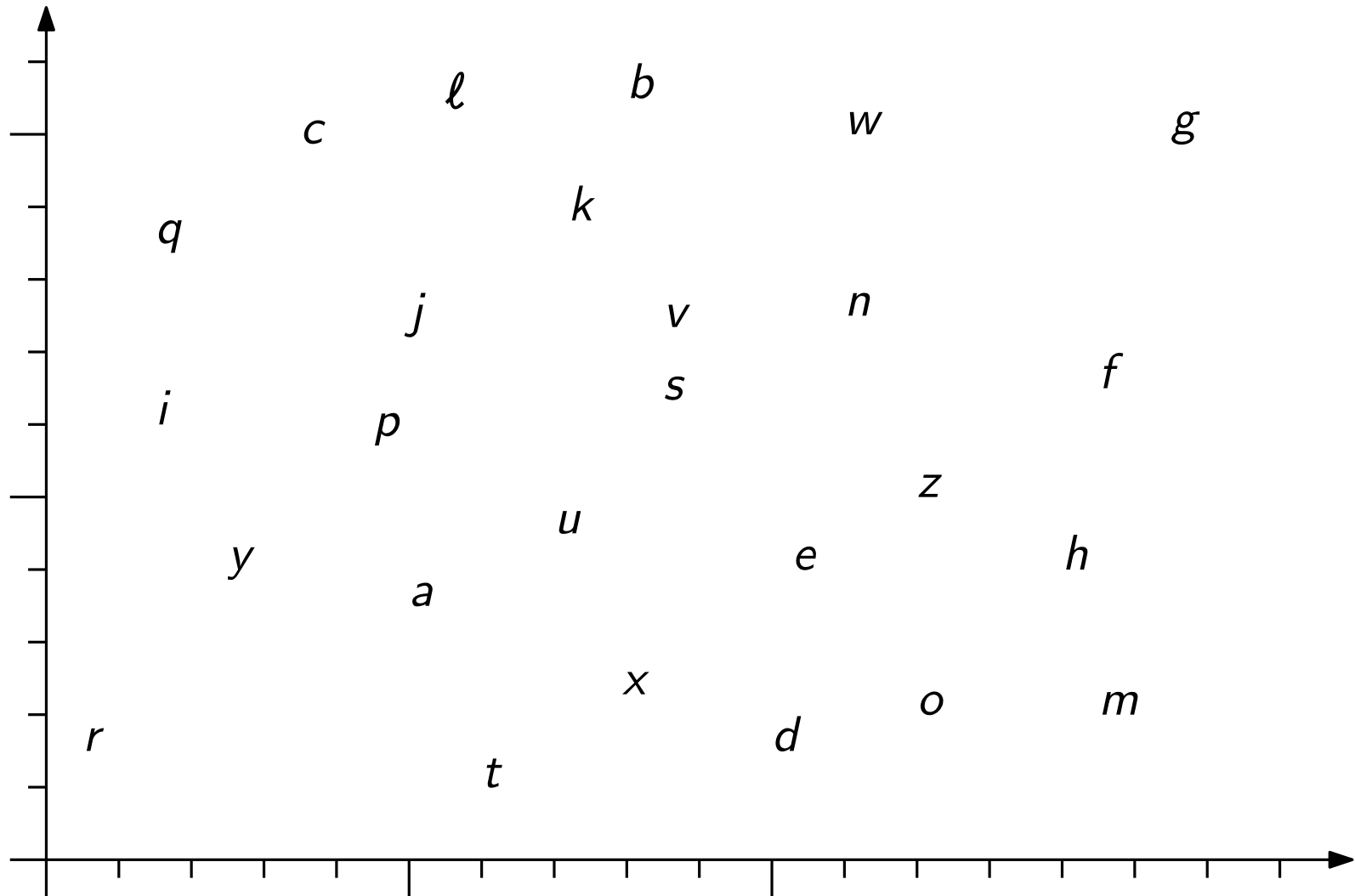
- berechne den  $x$ -Baum  $\rightarrow O(n \log n)$
- jedes  $y$ -Array für sich sortieren:  $O(n \log n)$  pro Layer  $\rightarrow O(n \log^2 n)$
- Verbesserung des zweiten Schritts  $\rightarrow O(n \log n)$ 
  - sortiere einmal alle Punkte nach  $y \rightarrow O(n \log n)$
  - spalte sortiertes Array auf um sortierte Arrays für die Kinder zu erhalten
  - $O(n)$  pro Layer  $\rightarrow O(n \log n)$

## Speicherverbrauch

- $O(n)$  pro Layer  $\rightarrow O(n \log n)$

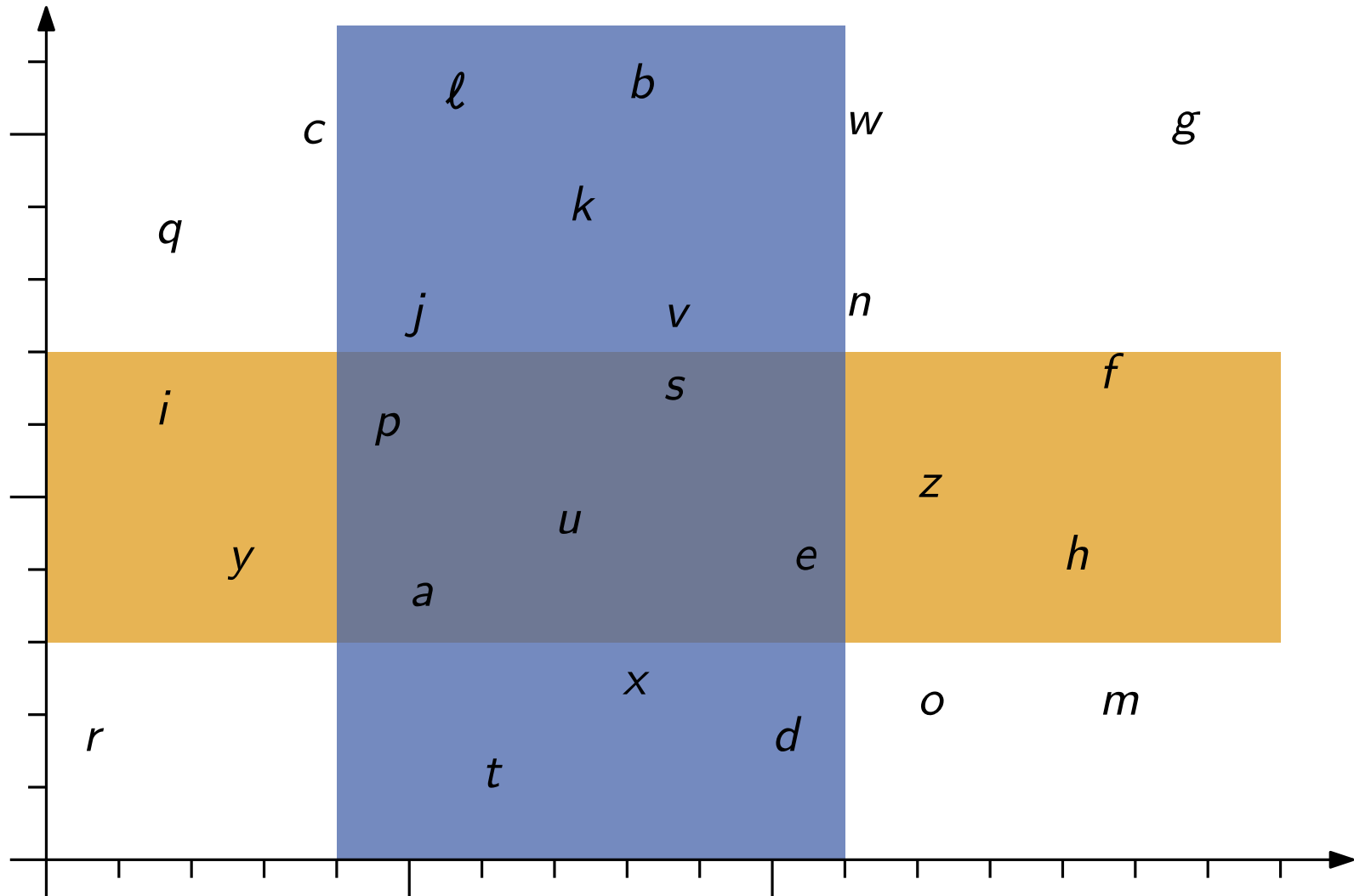


# Was ist das Ergebnis?



Bereichsanfrage:  $[4, 11] \times [3, 7]$

# Was ist das Ergebnis?



Bereichsanfrage:  $[4, 11] \times [3, 7]$

# Allgemeine Range-Trees

## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

# Allgemeine Range-Trees

## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## Dimension $d$

# Allgemeine Range-Trees

## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## Dimension $d$

- binärer Suchbaum für Dimension 1

# Allgemeine Range-Trees

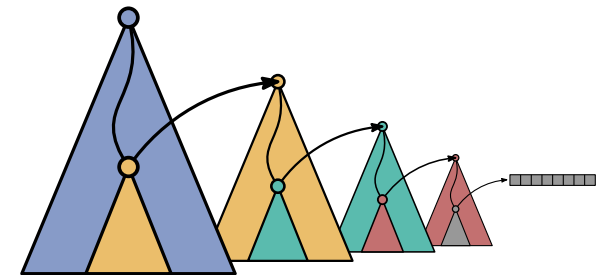
## Theorem

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## (2D-Range-Trees)

## Dimension $d$

- binärer Suchbaum für Dimension 1
- jeder Knoten kennt  $(d-1)$ -dim Range-Tree für Dimensionen  $2 \dots d$  der entsprechenden Elemente



# Allgemeine Range-Trees

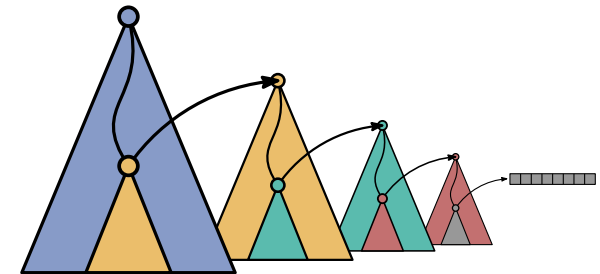
## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## Dimension $d$

- binärer Suchbaum für Dimension 1
- jeder Knoten kennt  $(d-1)$ -dim Range-Tree für Dimensionen  $2 \dots d$  der entsprechenden Elemente



## Theorem

## (Range-Trees für $d \geq 2$ )

Der Range-Tree für  $n$  Punkte in  $\mathbb{R}^d$  kann in  $O(n \log^{d-1} n)$  berechnet werden, benötigt  $O(n \log^{d-1} n)$  Speicher und erlaubt Anfragen in  $O(\log^d n + k)$ .

# Allgemeine Range-Trees

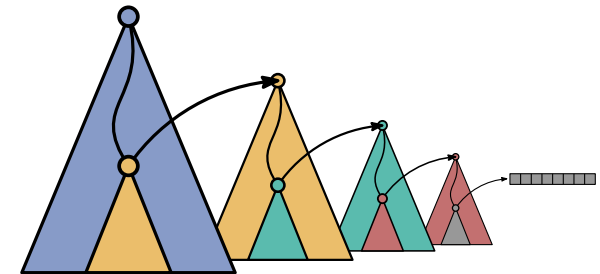
## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## Dimension $d$

- binärer Suchbaum für Dimension 1
- jeder Knoten kennt  $(d-1)$ -dim Range-Tree für Dimensionen  $2 \dots d$  der entsprechenden Elemente



## Theorem

## (Range-Trees für $d \geq 2$ )

Der Range-Tree für  $n$  Punkte in  $\mathbb{R}^d$  kann in  $O(n \log^{d-1} n)$  berechnet werden, benötigt  $O(n \log^{d-1} n)$  Speicher und erlaubt Anfragen in  $O(\log^d n + k)$ .

**Beweis:** Induktion über  $d$  (Induktionsanfang  $d = 2$  bereits abgehandelt)



# Allgemeine Range-Trees

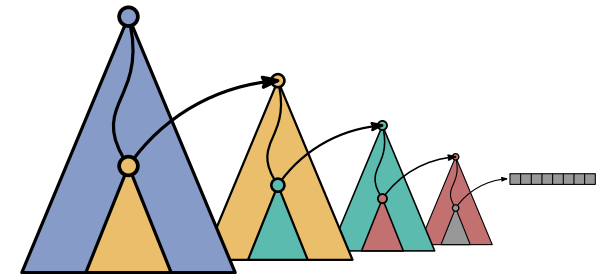
## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## Dimension $d$

- binärer Suchbaum für Dimension 1
- jeder Knoten kennt  $(d-1)$ -dim Range-Tree für Dimensionen  $2 \dots d$  der entsprechenden Elemente



## Theorem

## (Range-Trees für $d \geq 2$ )

Der Range-Tree für  $n$  Punkte in  $\mathbb{R}^d$  kann in  $O(n \log^{d-1} n)$  berechnet werden, benötigt  $O(n \log^{d-1} n)$  Speicher und erlaubt Anfragen in  $O(\log^d n + k)$ .

**Beweis:** Induktion über  $d$  (Induktionsanfang  $d = 2$  bereits abgehandelt)

- binärer Suchbaum für Dim. 1 aufbauen:  $O(n \log n)$  Zeit und  $O(n)$  Platz

# Allgemeine Range-Trees

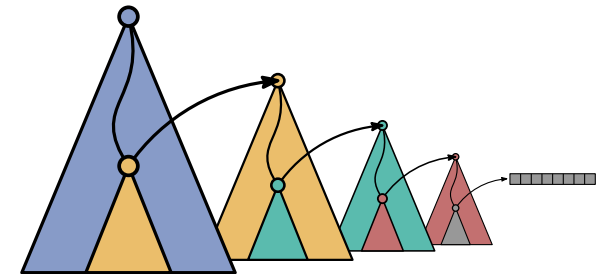
## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## Dimension $d$

- binärer Suchbaum für Dimension 1
- jeder Knoten kennt  $(d-1)$ -dim Range-Tree für Dimensionen  $2 \dots d$  der entsprechenden Elemente



## Theorem

## (Range-Trees für $d \geq 2$ )

Der Range-Tree für  $n$  Punkte in  $\mathbb{R}^d$  kann in  $O(n \log^{d-1} n)$  berechnet werden, benötigt  $O(n \log^{d-1} n)$  Speicher und erlaubt Anfragen in  $O(\log^d n + k)$ .

**Beweis:** Induktion über  $d$  (Induktionsanfang  $d = 2$  bereits abgehandelt)

- binärer Suchbaum für Dim. 1 aufbauen:  $O(n \log n)$  Zeit und  $O(n)$  Platz
- pro Knoten: Range-Tree mit Dim.  $d - 1$  der entsprechenden Punkte:  $O(n \log^{d-2} n)$  Zeit,  $O(n \log^{d-2} n)$  Platz pro Layer  $\rightarrow$  gesamt:  $O(n \log^{d-1} n)$

# Allgemeine Range-Trees

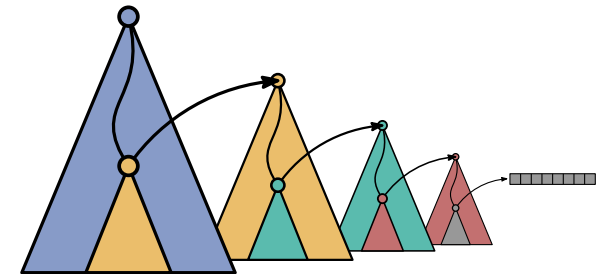
## Theorem

## (2D-Range-Trees)

Der 2D-Range-Tree für  $n$  Punkte kann in  $O(n \log n)$  berechnet werden, benötigt  $O(n \log n)$  Speicher und erlaubt Bereichsanfragen in  $O(\log^2 n + k)$ .

## Dimension $d$

- binärer Suchbaum für Dimension 1
- jeder Knoten kennt  $(d-1)$ -dim Range-Tree für Dimensionen  $2 \dots d$  der entsprechenden Elemente



## Theorem

## (Range-Trees für $d \geq 2$ )

Der Range-Tree für  $n$  Punkte in  $\mathbb{R}^d$  kann in  $O(n \log^{d-1} n)$  berechnet werden, benötigt  $O(n \log^{d-1} n)$  Speicher und erlaubt Anfragen in  $O(\log^d n + k)$ .

**Beweis:** Induktion über  $d$  (Induktionsanfang  $d = 2$  bereits abgehandelt)

- binärer Suchbaum für Dim. 1 aufbauen:  $O(n \log n)$  Zeit und  $O(n)$  Platz
- pro Knoten: Range-Tree mit Dim.  $d - 1$  der entsprechenden Punkte:  $O(n \log^{d-2} n)$  Zeit,  $O(n \log^{d-2} n)$  Platz pro Layer  $\rightarrow$  gesamt:  $O(n \log^{d-1} n)$
- Anfrage:  $O(\log n)$  für Dim. 1 und  $O(\log n)$  Anfragen in  $(d - 1)$ D-Range-Trees (mit disjunkten Ausgaben!)

# Geht es besser?

	$d = 1$	$d = 2$	$d > 2$
<b>Bereichsanfrage</b>	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
<b>Vorberechnung</b>	$n \log n$	$n \log n$	$n \log^{d-1} n$
<b>Speicherplatz</b>	$n$	$n \log n$	$n \log^{d-1} n$

# Geht es besser?

	$d = 1$	$d = 2$	$d > 2$
<b>Bereichsanfrage</b>	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
<b>Vorberechnung</b>	$n \log n$	$n \log n$	$n \log^{d-1} n$
<b>Speicherplatz</b>	$n$	$n \log n$	$n \log^{d-1} n$

- pro Dimension verlieren wir einen  $\log n$  Faktor

# Geht es besser?

	$d = 1$	$d = 2$	$d > 2$
<b>Bereichsanfrage</b>	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
<b>Vorberechnung</b>	$n \log n$	$n \log n$	$n \log^{d-1} n$
<b>Speicherplatz</b>	$n$	$n \log n$	$n \log^{d-1} n$

- pro Dimension verlieren wir einen  $\log n$  Faktor
- wenn wir den Fall  $d = 2$  verbessern wird  $d > 2$  auch besser

# Geht es besser?

	$d = 1$	$d = 2$	$d > 2$
<b>Bereichsanfrage</b>	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
<b>Vorberechnung</b>	$n \log n$	$n \log n$	$n \log^{d-1} n$
<b>Speicherplatz</b>	$n$	$n \log n$	$n \log^{d-1} n$

- pro Dimension verlieren wir einen  $\log n$  Faktor
- wenn wir den Fall  $d = 2$  verbessern wird  $d > 2$  auch besser
- von 1 zu 2 haben wir schon getrickst um  $\log n$  Vorberechnung zu sparen

# Geht es besser?

	$d = 1$	$d = 2$	$d > 2$
<b>Bereichsanfrage</b>	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
<b>Vorberechnung</b>	$n \log n$	$n \log n$	$n \log^{d-1} n$
<b>Speicherplatz</b>	$n$	$n \log n$	$n \log^{d-1} n$

- pro Dimension verlieren wir einen  $\log n$  Faktor
- wenn wir den Fall  $d = 2$  verbessern wird  $d > 2$  auch besser
- von 1 zu 2 haben wir schon getrickst um  $\log n$  Vorbereitung zu sparen

## Heute

- spare  $\log n$  für die Anfragen für  $d = 2$
- spart auch einen  $\log n$  Faktor für alle höheren Dimensionen



# Geht es besser?

	$d = 1$	$d = 2$	$d > 2$
<b>Bereichsanfrage</b>	$\log n + k$	$\log^2 n + k$	$\log^d n + k$
<b>Vorberechnung</b>	$n \log n$	$n \log n$	$n \log^{d-1} n$
<b>Speicherplatz</b>	$n$	$n \log n$	$n \log^{d-1} n$

- pro Dimension verlieren wir einen  $\log n$  Faktor
- wenn wir den Fall  $d = 2$  verbessern wird  $d > 2$  auch besser
- von 1 zu 2 haben wir schon getrickst um  $\log n$  Vorbereitung zu sparen

## Heute

- spare  $\log n$  für die Anfragen für  $d = 2$
- spart auch einen  $\log n$  Faktor für alle höheren Dimensionen

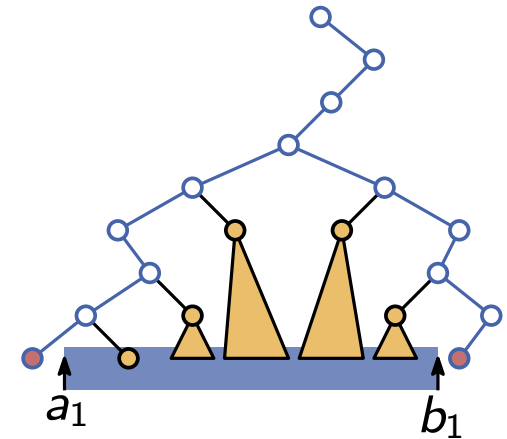
## Nächste Woche

- spare einen weiteren  $\log n$  Faktor für  $d = 3$  bei den Anfragen
- dafür bezahlt man einen  $\log n$  Faktor bei Vorbereitung und Speicher

# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

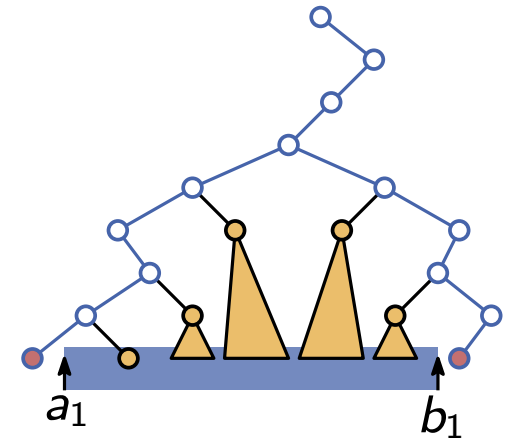
- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume



# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

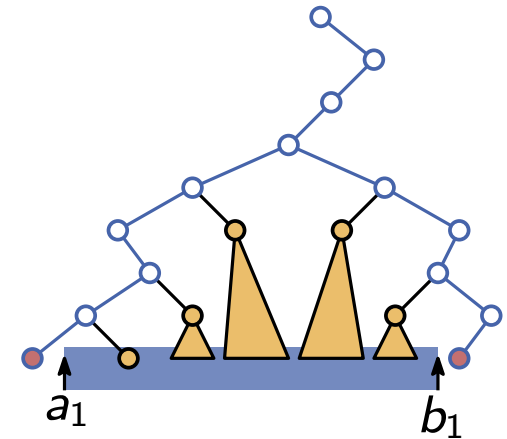
- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



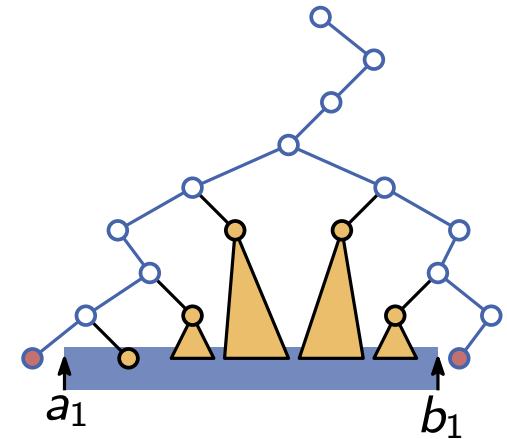
## Eigentlich...

- suchen wir ja insgesamt nur auf  $\leq n$  Zahlen

# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



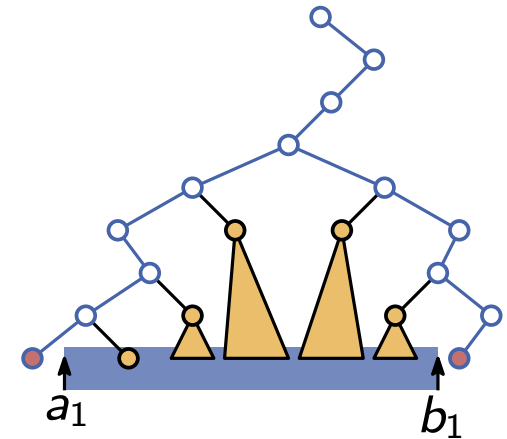
## Eigentlich...

- suchen wir ja insgesamt nur auf  $\leq n$  Zahlen
- und wir suchen immer nach den gleichen Zahlen ( $a_2$  und  $b_2$ )

# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



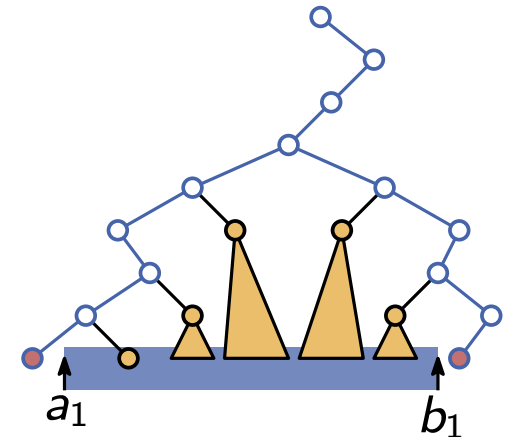
## Eigentlich...

- suchen wir ja insgesamt nur auf  $\leq n$  Zahlen
- und wir suchen immer nach den gleichen Zahlen ( $a_2$  und  $b_2$ )
- brauchen wir nur so lange, weil die Zahlen in Teilmengen zerstückelt sind

# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



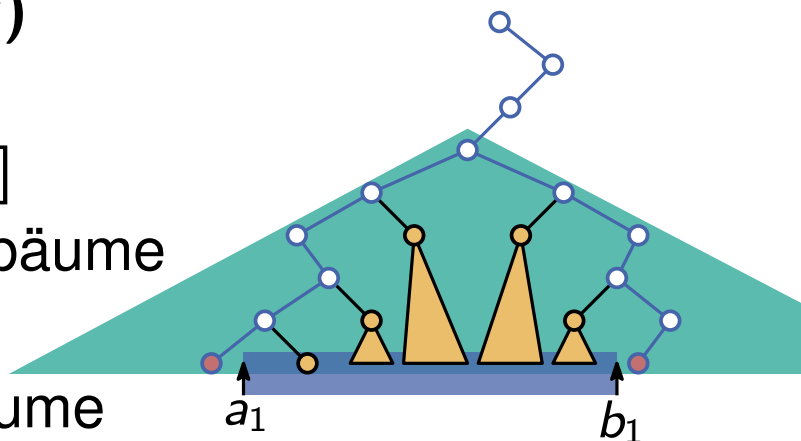
## Eigentlich...

- suchen wir ja insgesamt nur auf  $\leq n$  Zahlen
- und wir suchen immer nach den gleichen Zahlen ( $a_2$  und  $b_2$ )
- brauchen wir nur so lange, weil die Zahlen in Teilmengen zerstückelt sind
- könnten wir sogar auf allen  $n$  Zahlen suchen und wären schneller

# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



## Eigentlich...

- suchen wir ja insgesamt nur auf  $\leq n$  Zahlen
- und wir suchen immer nach den gleichen Zahlen ( $a_2$  und  $b_2$ )
- brauchen wir nur so lange, weil die Zahlen in Teilmengen zerstückelt sind
- könnten wir sogar auf allen  $n$  Zahlen suchen und wären schneller

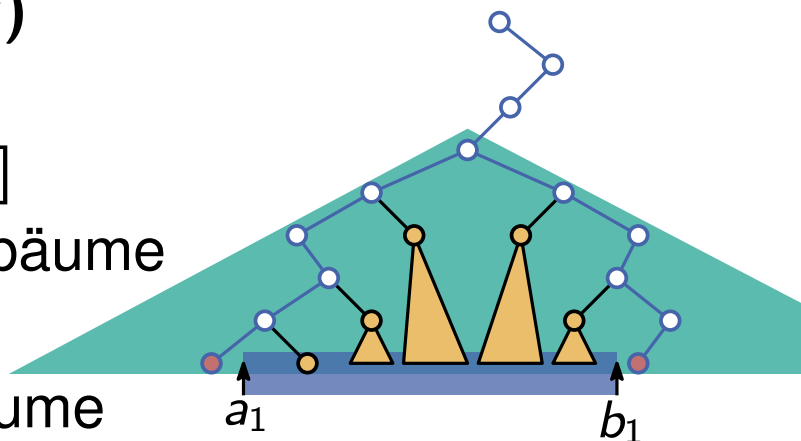
**Idee:** suche nur einmal auf einer **Obermenge** der Punkte



# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



## Eigentlich...

- suchen wir ja insgesamt nur auf  $\leq n$  Zahlen
- und wir suchen immer nach den gleichen Zahlen ( $a_2$  und  $b_2$ )
- brauchen wir nur so lange, weil die Zahlen in Teilmengen zerstückelt sind
- könnten wir sogar auf allen  $n$  Zahlen suchen und wären schneller

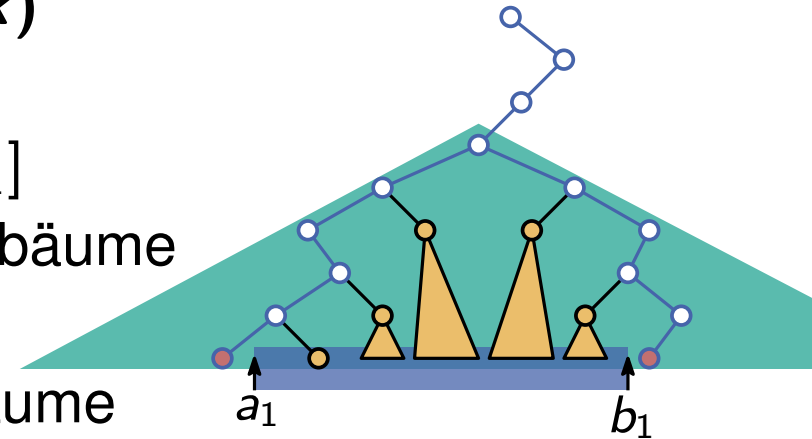
**Idee:** suche nur einmal auf einer **Obermenge** der Punkte

**Problem:** Ergebnis enthält potentiell zu viele Punkte (bzgl.  $x$ -Koordinate)

# Warum ist das so teuer?

## Erinnerung: Bereichsanfrage in $O(\log^2 n + k)$

- Suche im  $x$ -Baum  $\rightarrow O(\log n)$ 
  - findet alle Punkte mit  $x$ -Koordinate in  $[a_1, b_1]$
  - implizite Repräsentation durch  $O(\log n)$  Teilbäume
- binäre Suchen in  $y$ -Richtung
  - eine Suche (bzw. zwei) für jeden der Teilbäume
  - $O(\log n)$  pro Suche  $\rightarrow O(\log^2 n)$



## Eigentlich...

- suchen wir ja insgesamt nur auf  $\leq n$  Zahlen
- und wir suchen immer nach den gleichen Zahlen ( $a_2$  und  $b_2$ )
- brauchen wir nur so lange, weil die Zahlen in Teilmengen zerstückelt sind
- könnten wir sogar auf allen  $n$  Zahlen suchen und wären schneller

**Idee:** suche nur einmal auf einer **Obermenge** der Punkte

**Problem:** Ergebnis enthält potentiell zu viele Punkte (bzgl.  $x$ -Koordinate)

**Idee:** Suche in Obermenge; Ausgabe in den korrekten Teilmengen

# Suche in einer Obermenge

Situation (etwas vereinfacht)

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen

$A =$ 

2	5	8	12	16	17	25	28	32	33	34
---	---	---	----	----	----	----	----	----	----	----

$B =$ 

2	8	12	16	32	34
---	---	----	----	----	----

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen

$A =$ 

2	5	8	12	16	17	25	28	32	33	34
---	---	---	----	----	----	----	----	----	----	----

$B =$ 

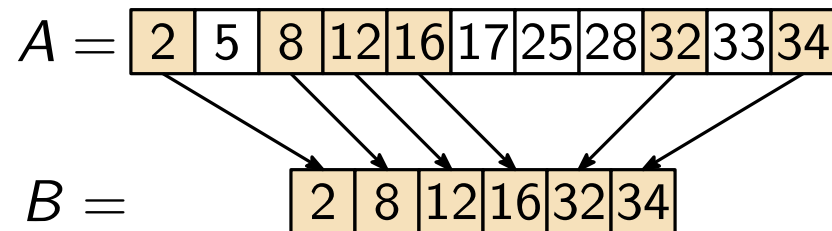
2	8	12	16	32	34
---	---	----	----	----	----

**Fall 1:  $x \in B$**

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen



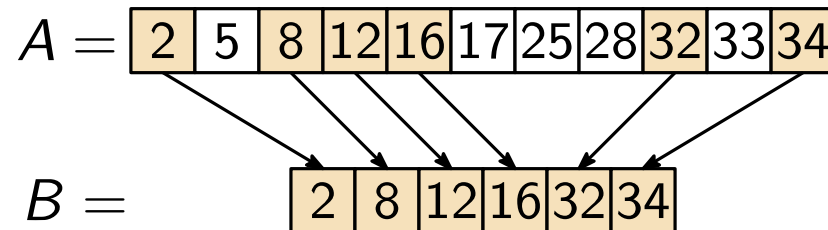
### Fall 1: $x \in B$

- Zeiger von Elementen aus  $A$  zu Kopien in  $B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen



### Fall 1: $x \in B$

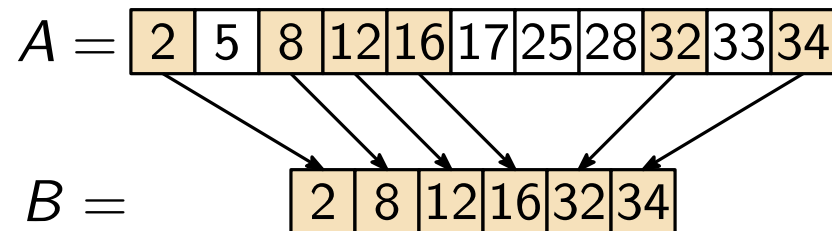
- Zeiger von Elementen aus  $A$  zu Kopien in  $B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

### Fall 2: $x \in A$ aber $x \notin B$

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen



### Fall 1: $x \in B$

- Zeiger von Elementen aus  $A$  zu Kopien in  $B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

### Fall 2: $x \in A$ aber $x \notin B$

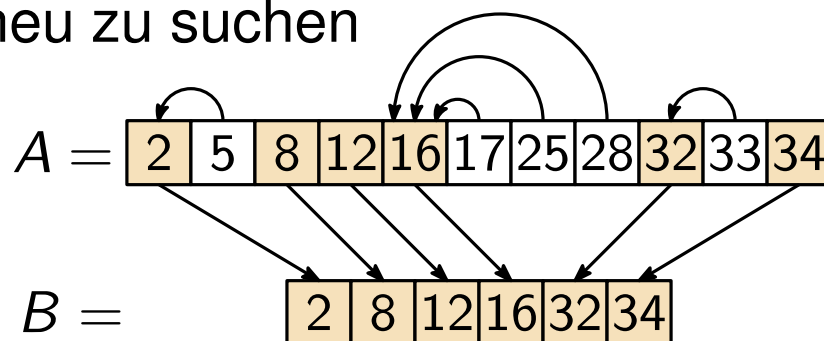
- Ziel: finde Vorgänger von  $x$  in  $B$



# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen



### Fall 1: $x \in B$

- Zeiger von Elementen aus  $A$  zu Kopien in  $B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

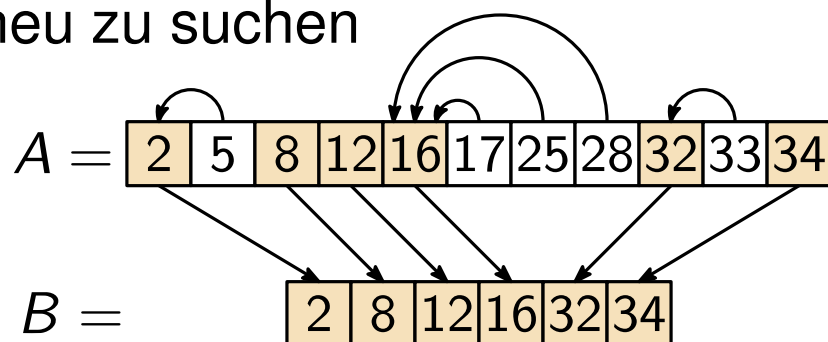
### Fall 2: $x \in A$ aber $x \notin B$

- Ziel: finde Vorgänger von  $x$  in  $B$
- Zeiger von jedem  $a \in A \setminus B$  zu Vorgänger in  $A \cap B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen



### Fall 1: $x \in B$

- Zeiger von Elementen aus  $A$  zu Kopien in  $B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

### Fall 2: $x \in A$ aber $x \notin B$

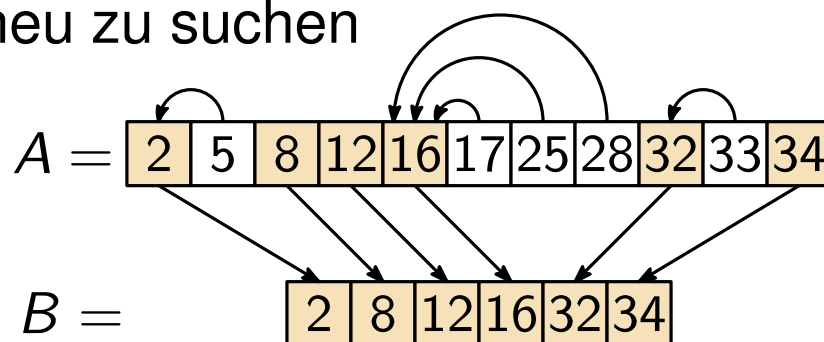
- Ziel: finde Vorgänger von  $x$  in  $B$
- Zeiger von jedem  $a \in A \setminus B$  zu Vorgänger in  $A \cap B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

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

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen



### Fall 1: $x \in B$

- Zeiger von Elementen aus  $A$  zu Kopien in  $B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

### Fall 2: $x \in A$ aber $x \notin B$

- Ziel: finde Vorgänger von  $x$  in  $B$
- Zeiger von jedem  $a \in A \setminus B$  zu Vorgänger in  $A \cap B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

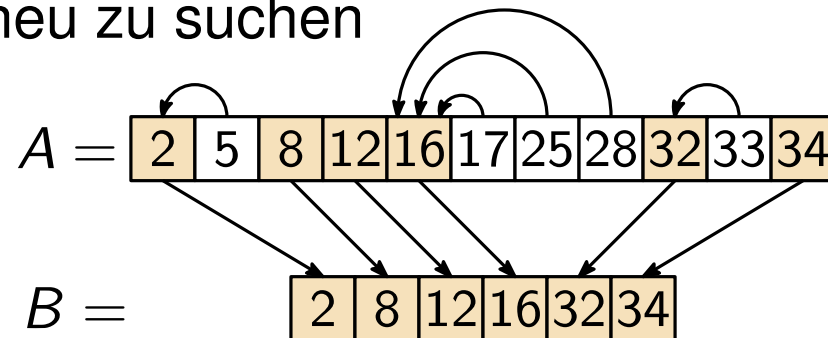
### Fall 3: $x \notin A$

- Ziel: finde Vorgänger von  $x$  in  $B$ , wobei wir Vorgänger von  $x$  in  $A$  kennen

# Suche in einer Obermenge

## Situation (etwas vereinfacht)

- betrachte sortierte Arrays von Zahlen  $A$  und  $B$  mit  $B \subseteq A$
- suche nach  $x$  in  $A$
- finde  $x$  in  $B$  ohne neu zu suchen



### Fall 1: $x \in B$

- Zeiger von Elementen aus  $A$  zu Kopien in  $B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

### Fall 2: $x \in A$ aber $x \notin B$

- Ziel: finde Vorgänger von  $x$  in  $B$
- Zeiger von jedem  $a \in A \setminus B$  zu Vorgänger in  $A \cap B \rightarrow$  finde  $x$  in  $B$  in  $O(1)$

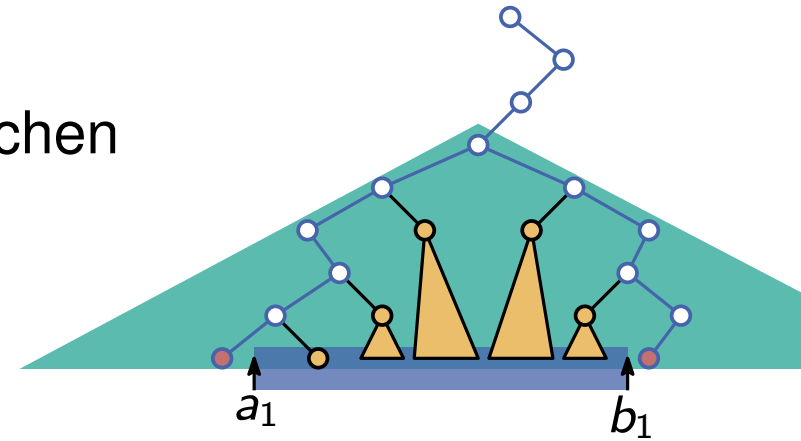
### Fall 3: $x \notin A$

- Ziel: finde Vorgänger von  $x$  in  $B$ , wobei wir Vorgänger von  $x$  in  $A$  kennen
- benutze Fall 1 oder 2  $\rightarrow$  finde  $x$  in  $B$  in  $O(1)$

# Und jetzt für Range-Trees

## Plan

- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen



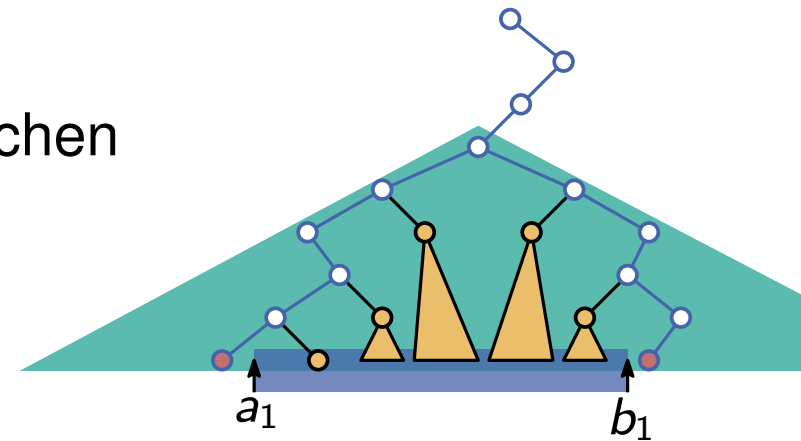
# Und jetzt für Range-Trees

## Plan

- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen  $\rightarrow$  zu viele Zeiger



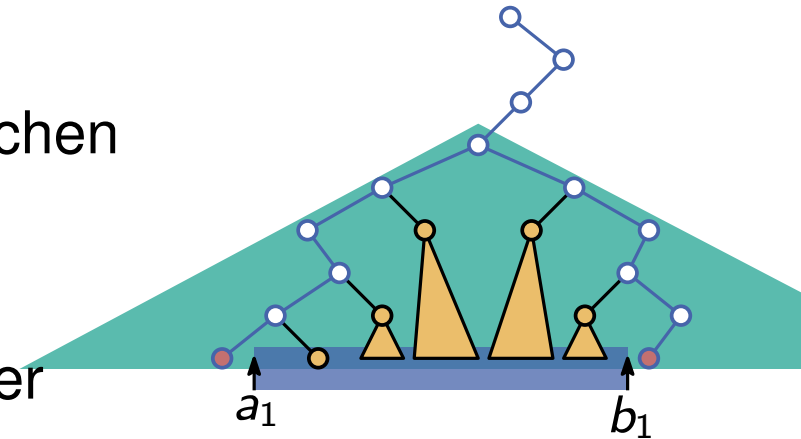
# Und jetzt für Range-Trees

## Plan

- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen  $\rightarrow$  zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



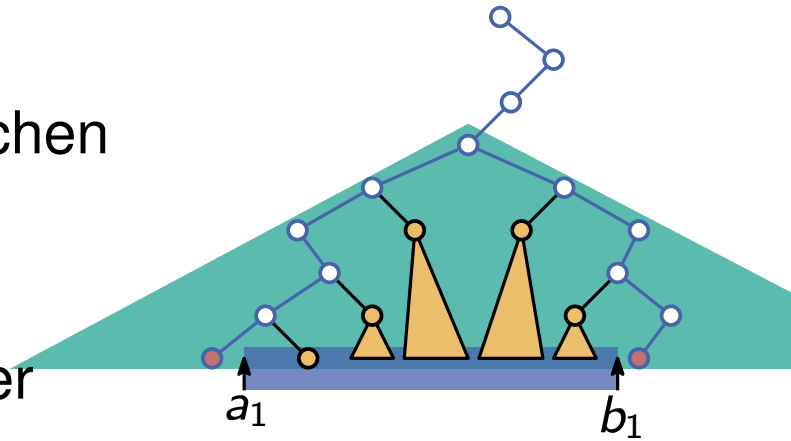
# Und jetzt für Range-Trees

## Plan

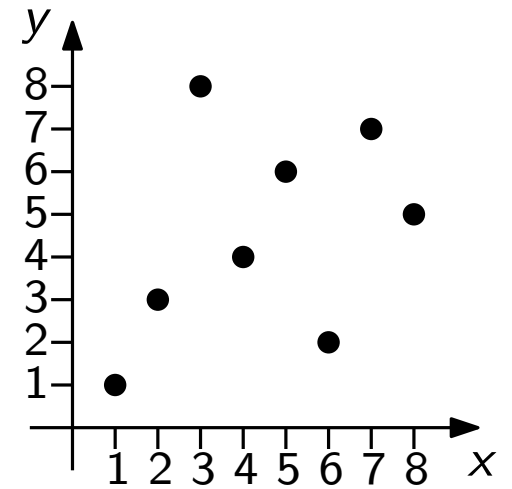
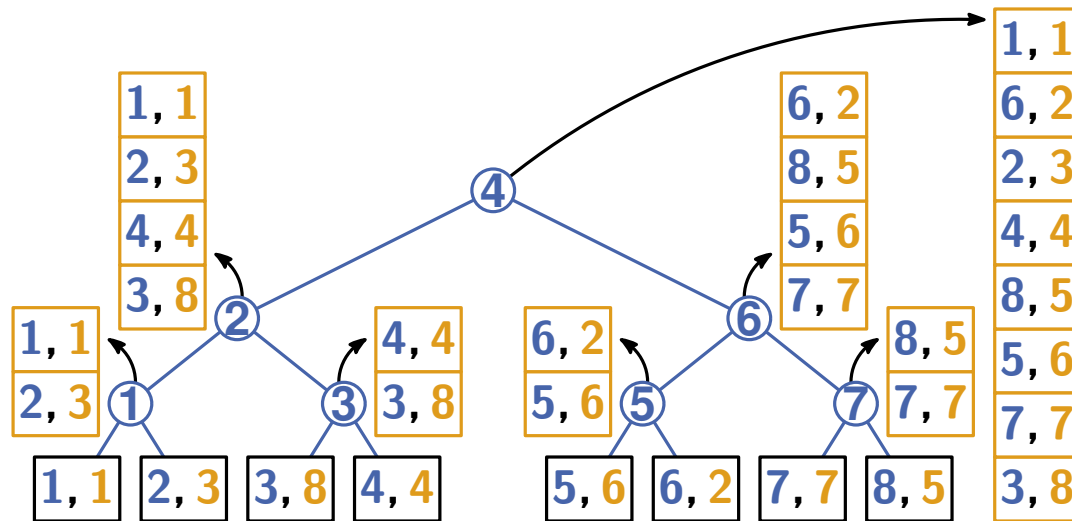
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen → zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel





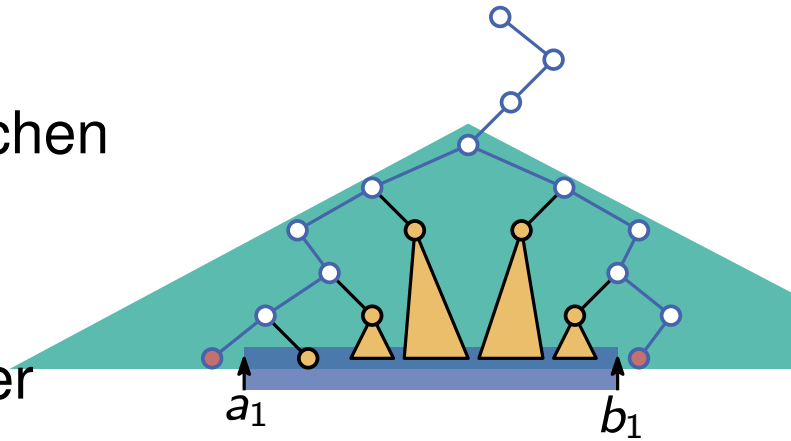
# Und jetzt für Range-Trees

## Plan

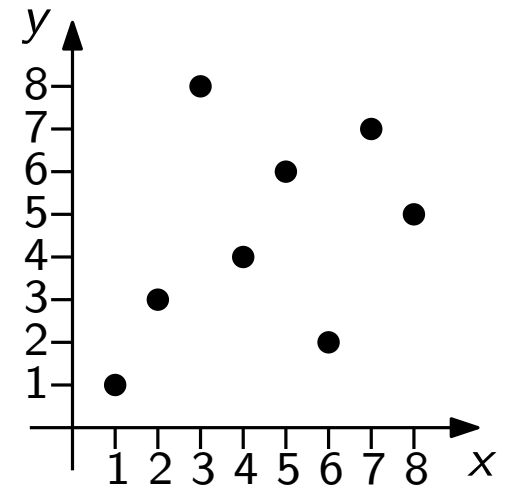
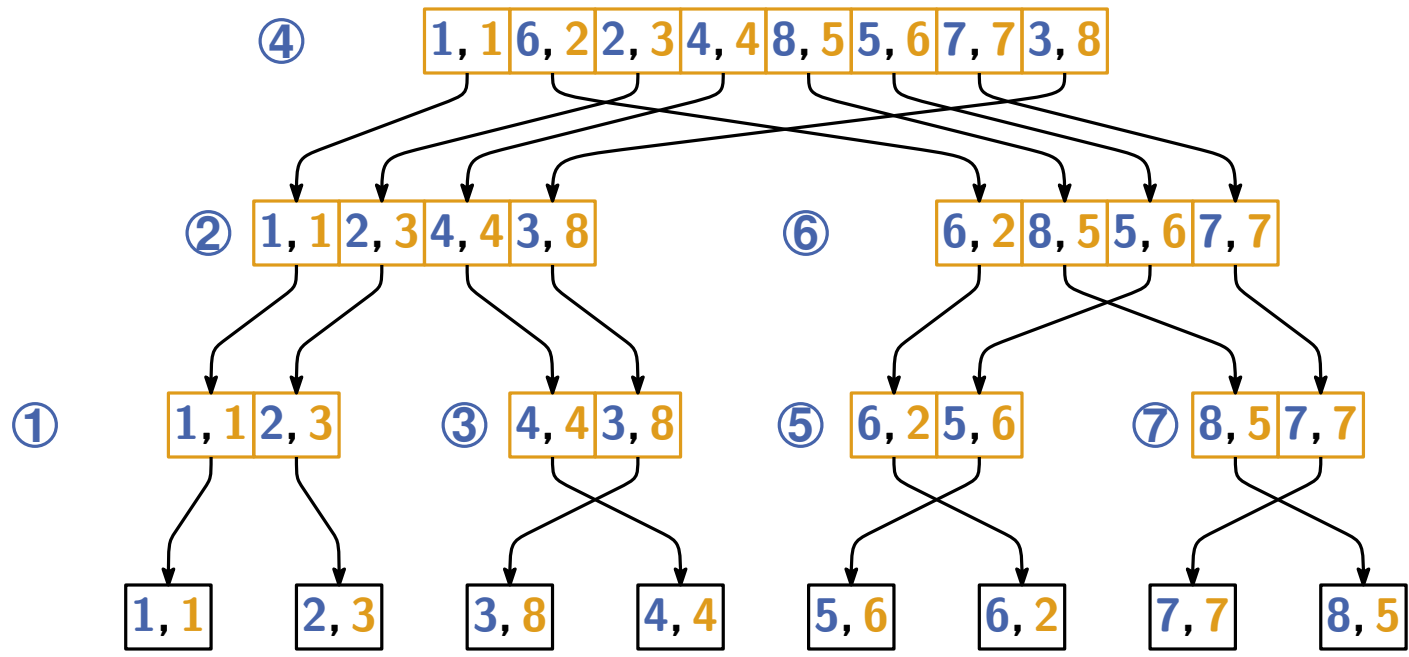
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen → zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel



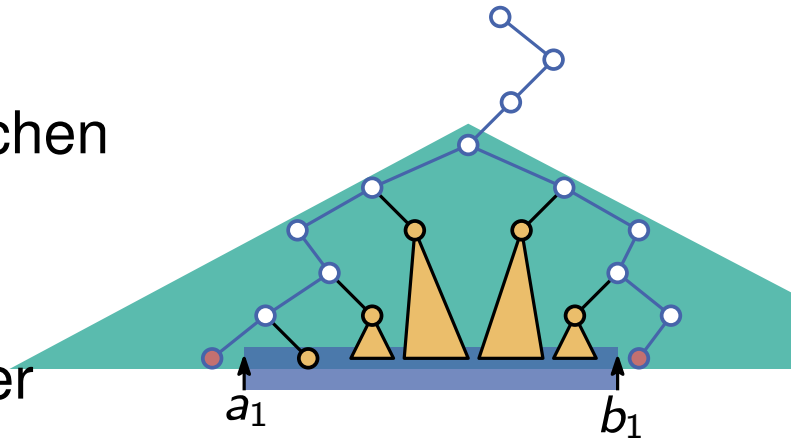
# Und jetzt für Range-Trees

## Plan

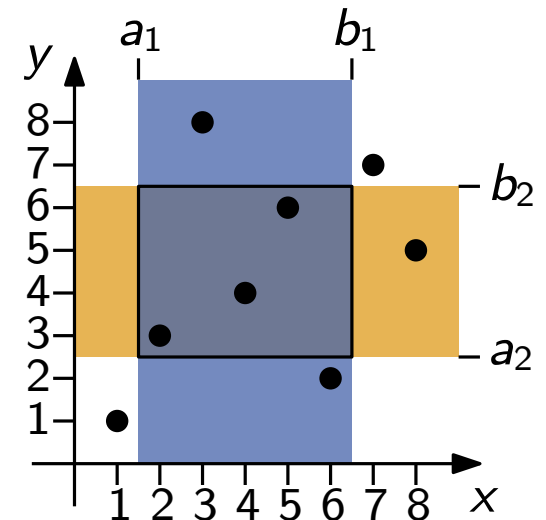
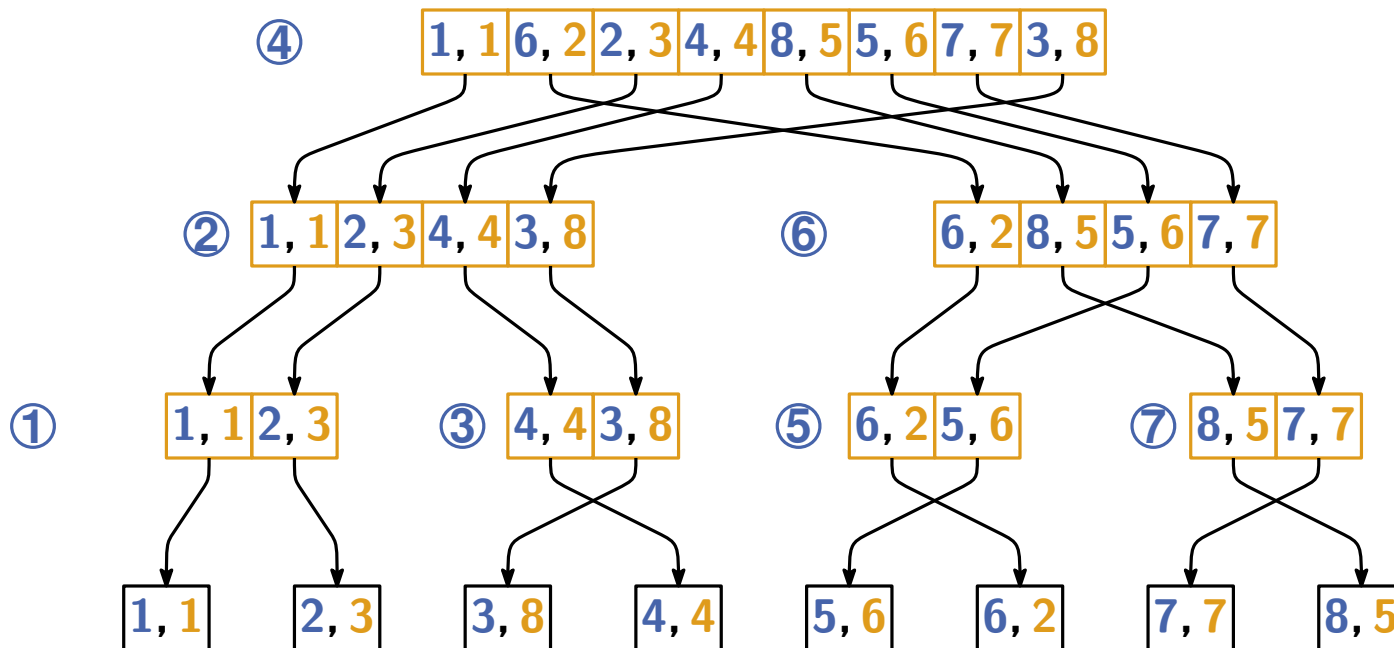
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen  $\rightarrow$  zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel



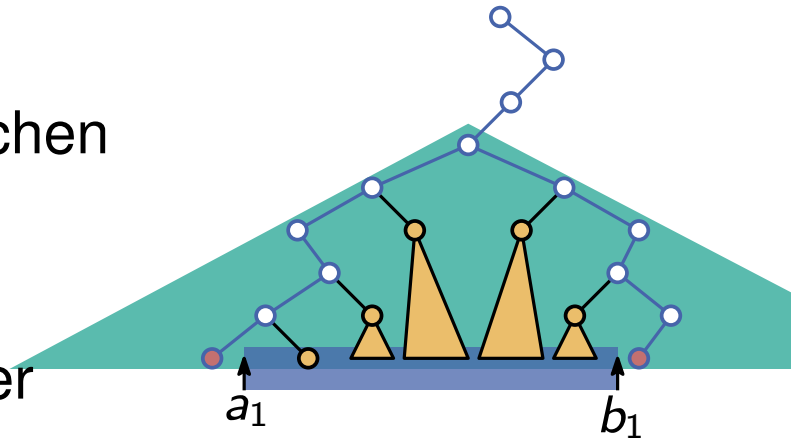
# Und jetzt für Range-Trees

## Plan

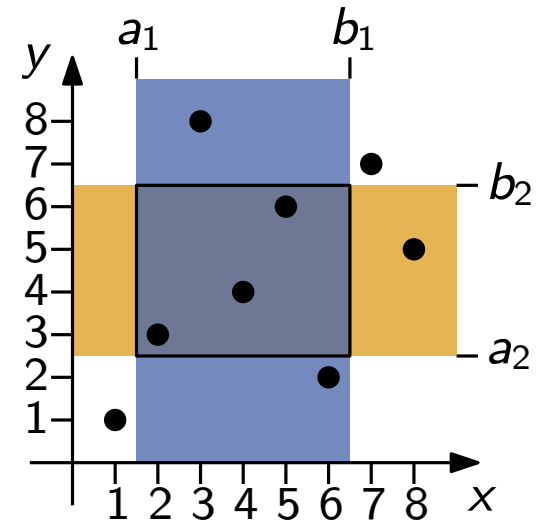
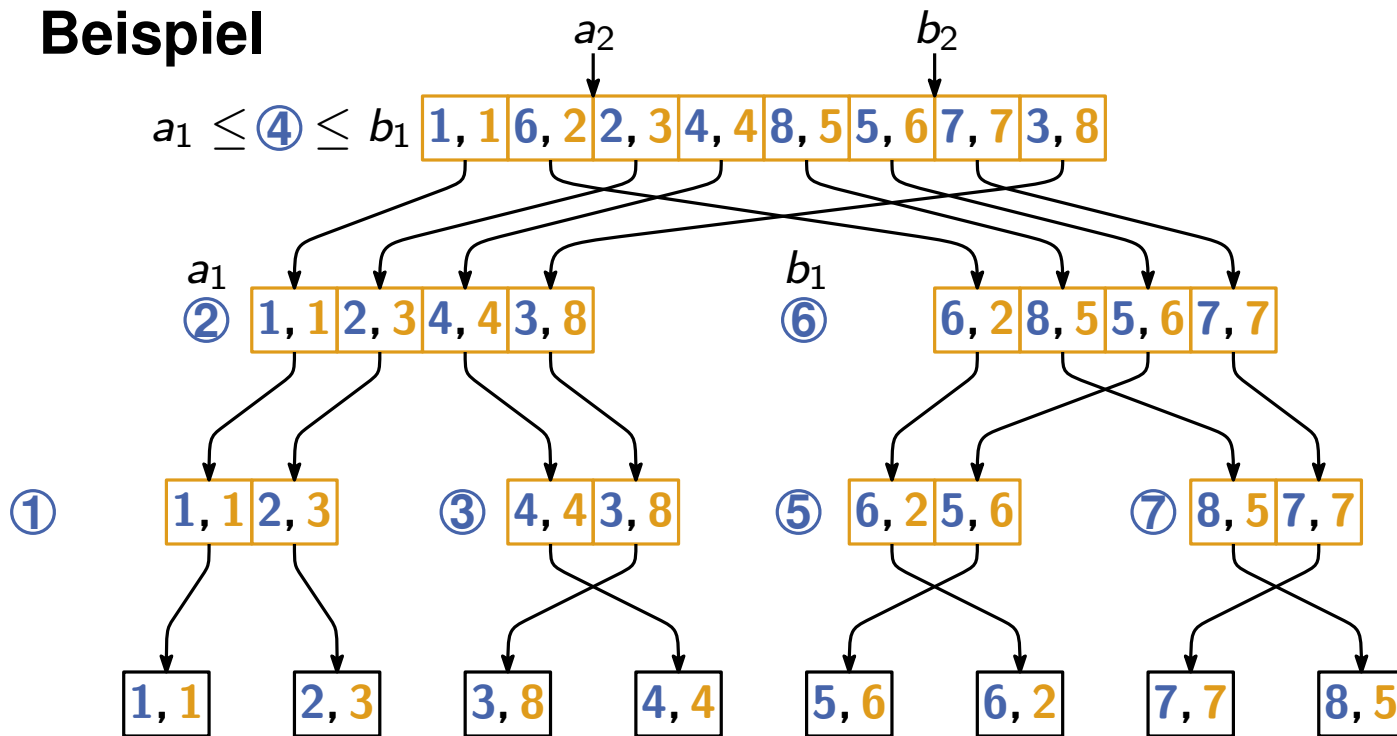
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen  $\rightarrow$  zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel



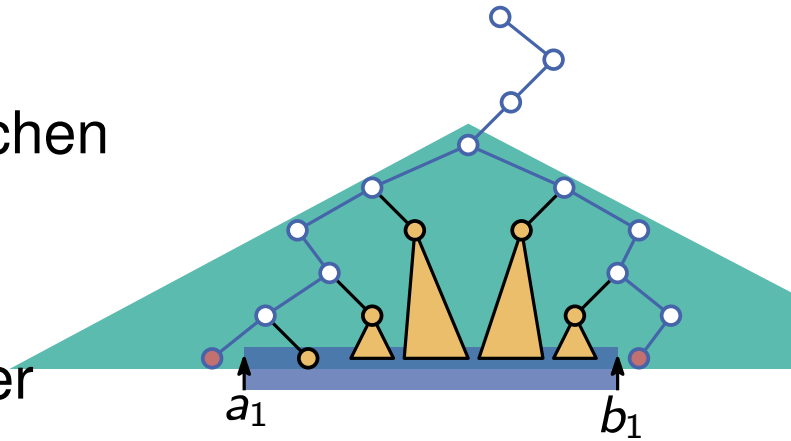
# Und jetzt für Range-Trees

## Plan

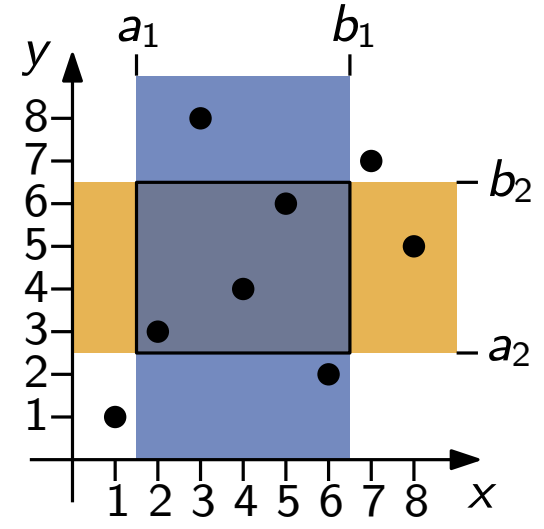
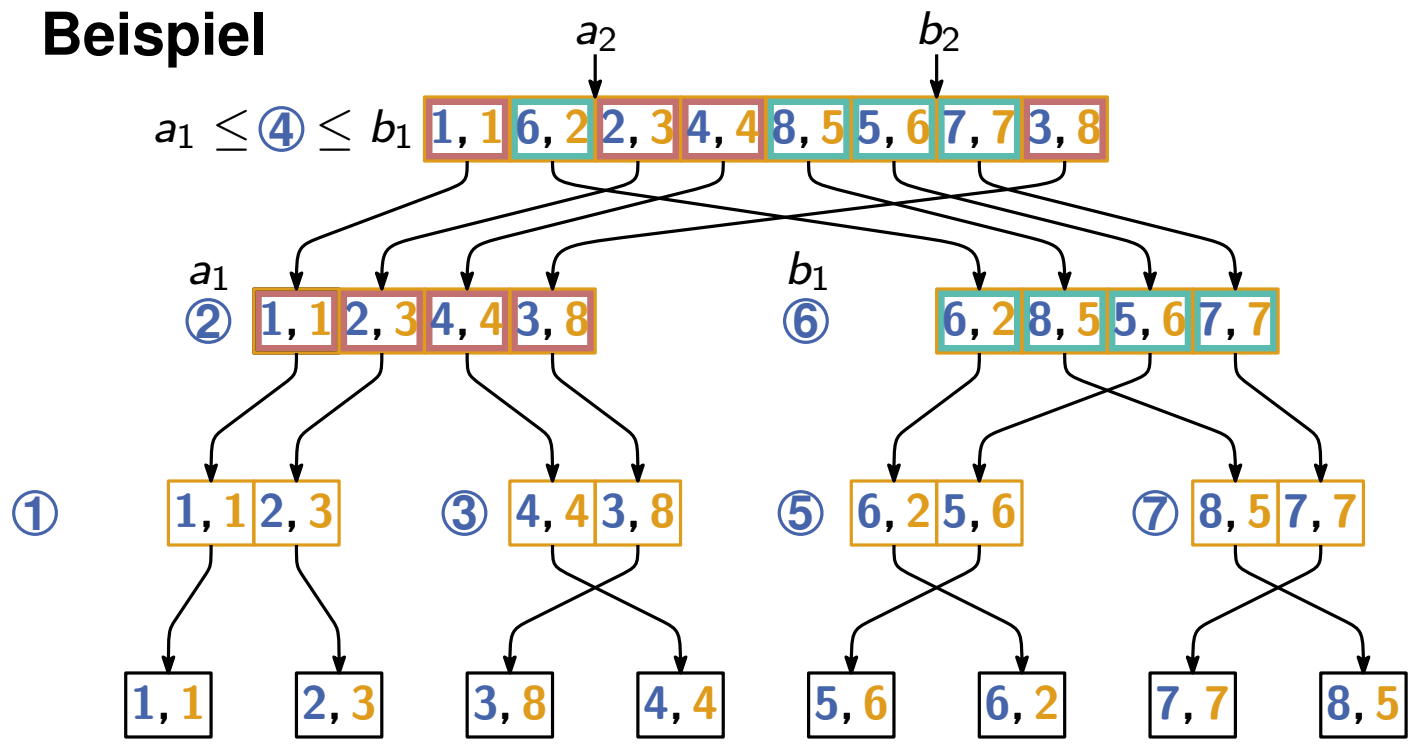
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen  $\rightarrow$  zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel



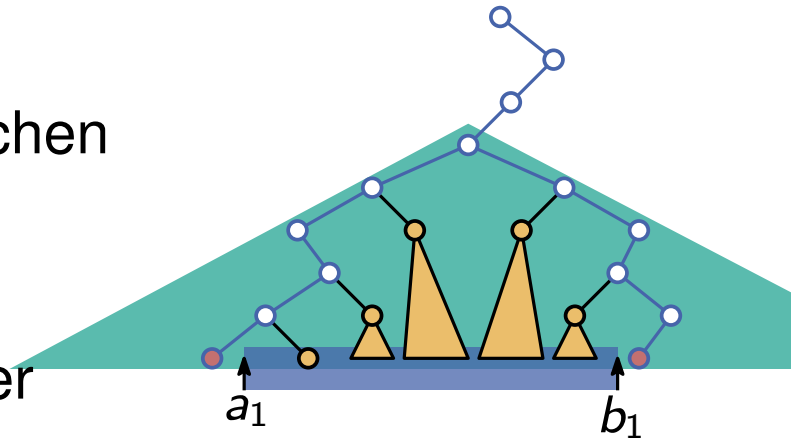
# Und jetzt für Range-Trees

## Plan

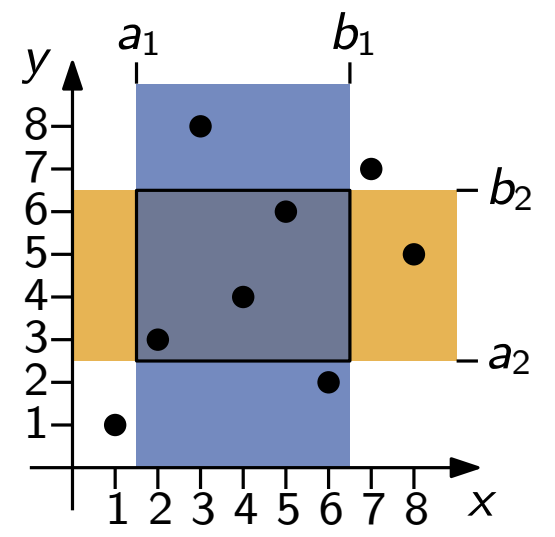
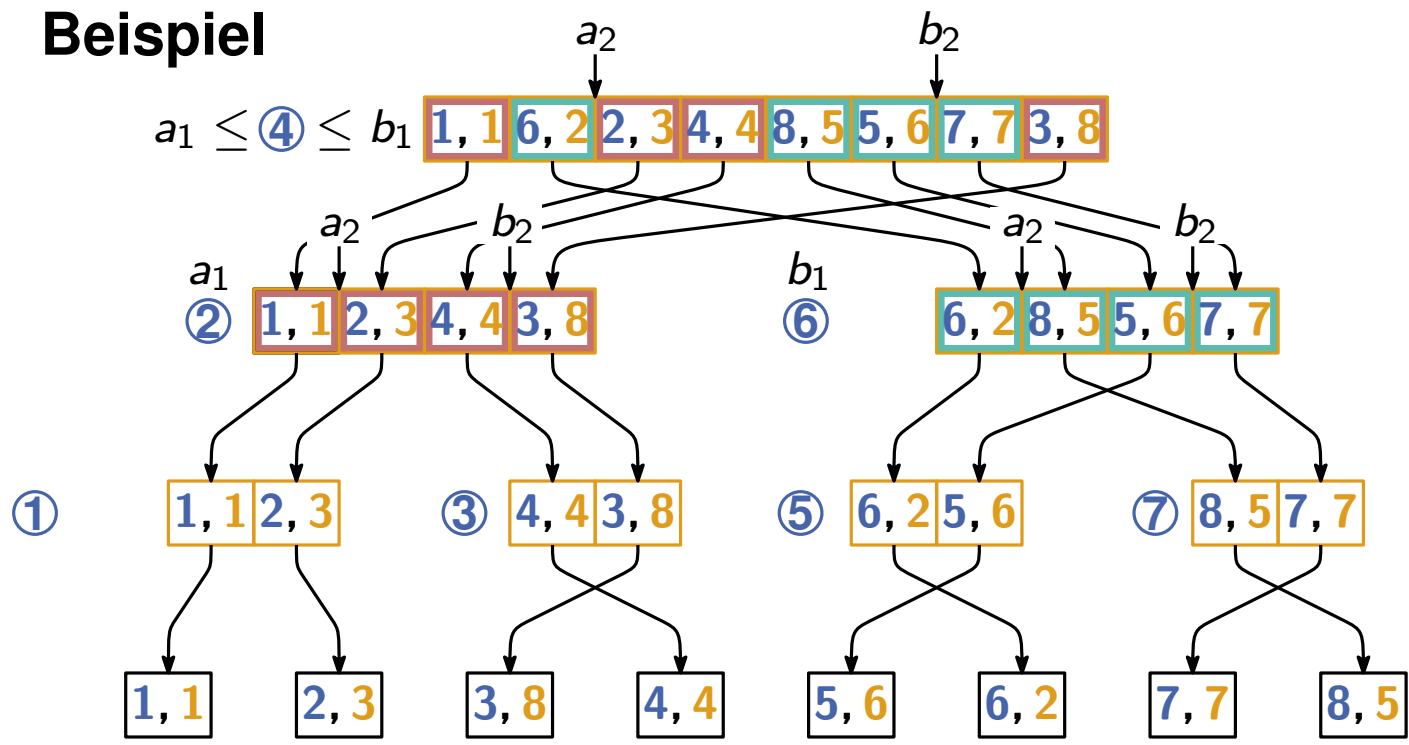
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen  $\rightarrow$  zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel



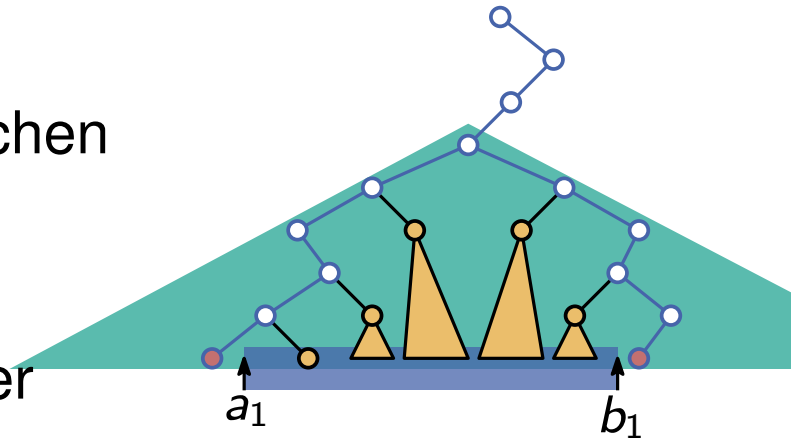
# Und jetzt für Range-Trees

## Plan

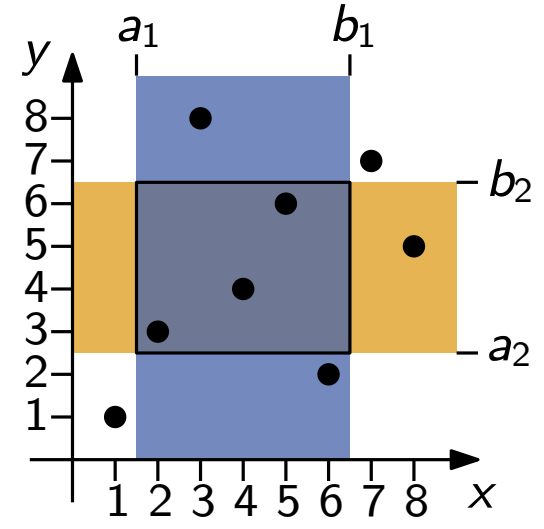
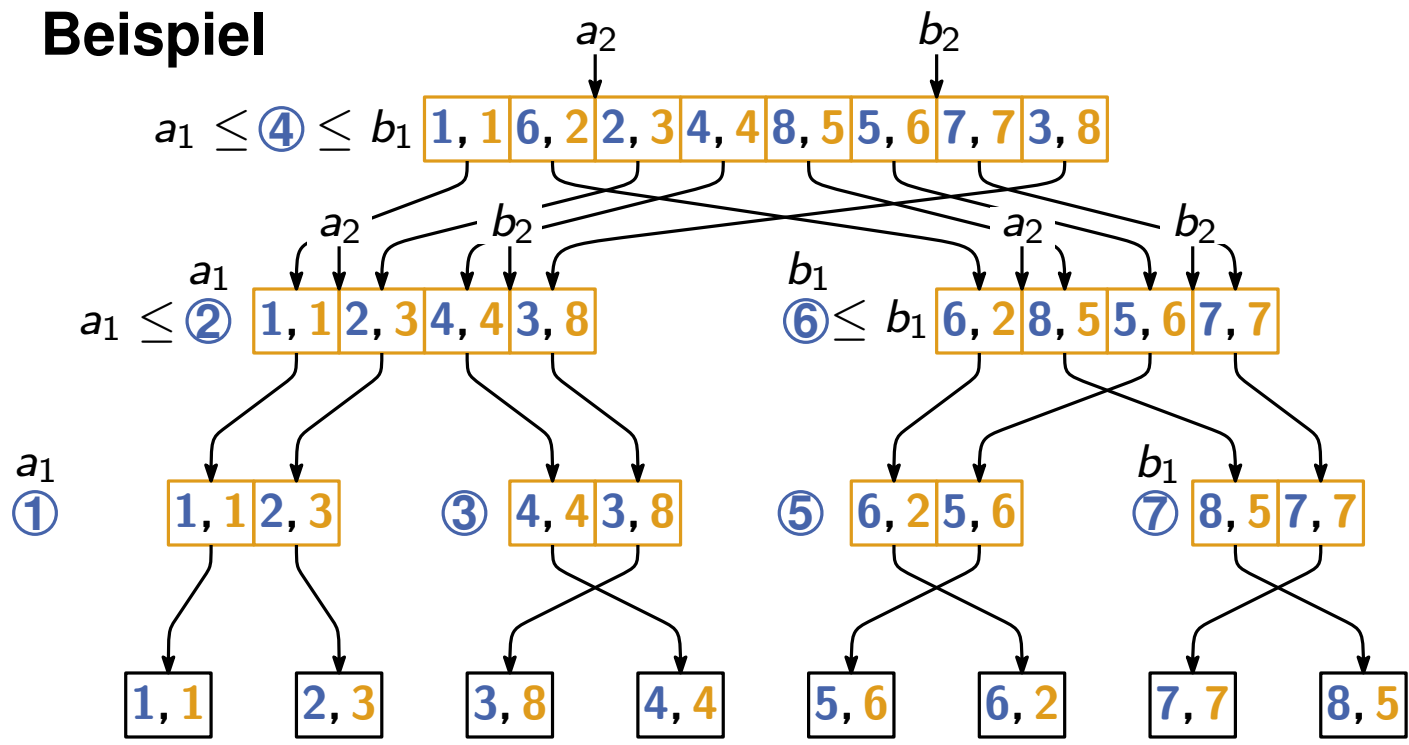
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen → zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel



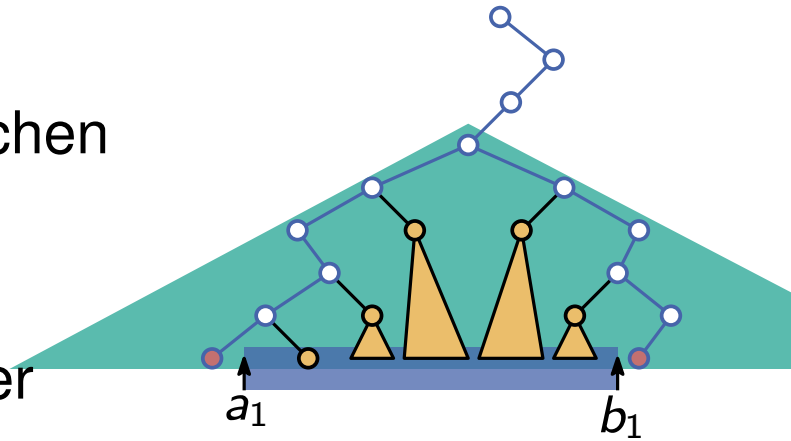
# Und jetzt für Range-Trees

## Plan

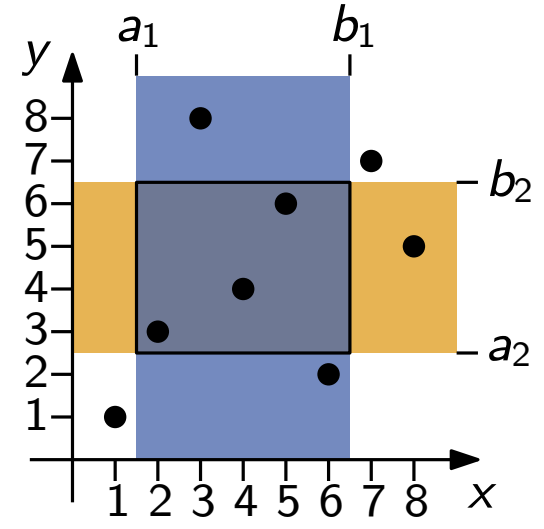
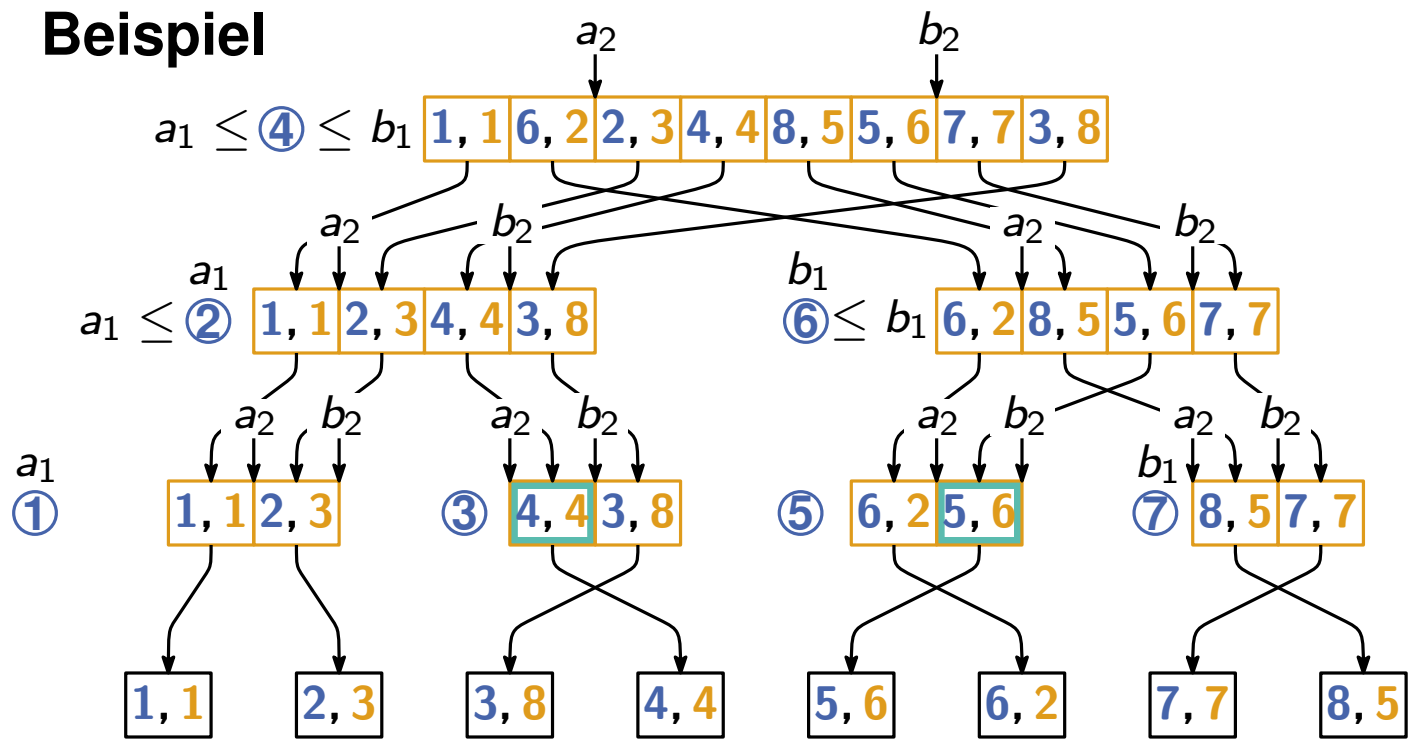
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen → zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel



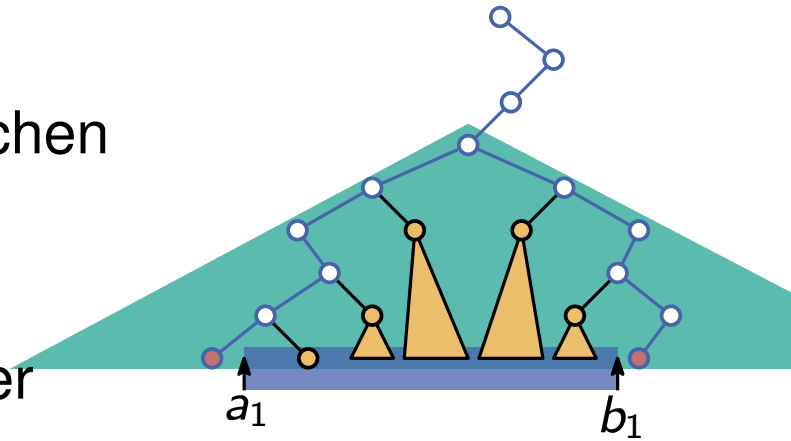
# Und jetzt für Range-Trees

## Plan

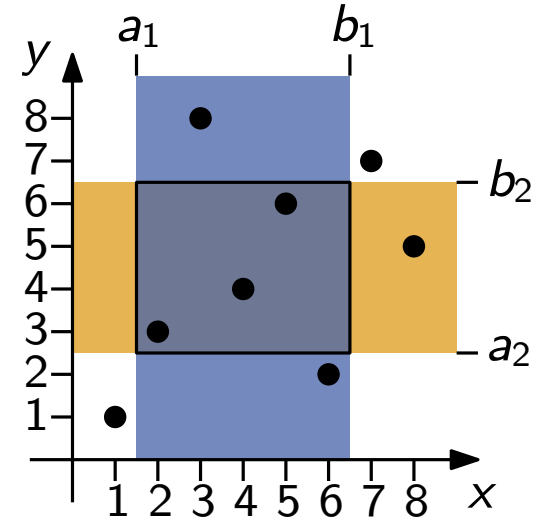
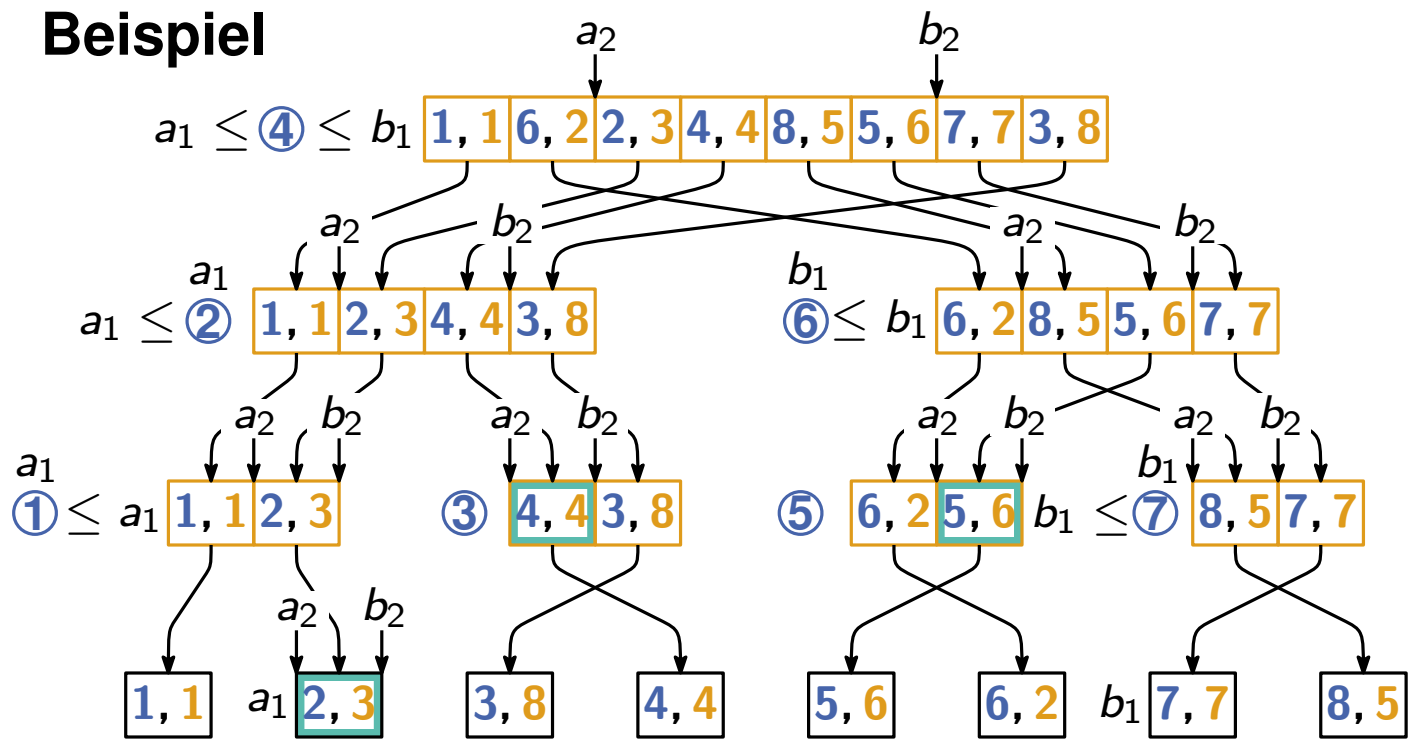
- suche nach  $a_2$  und  $b_2$  in der Obermenge
- finde  $a_2$  und  $b_2$  in den Teilmengen ohne zu suchen

## So viele Teilmengen

- Problem: viele Teilmengen  $\rightarrow$  zu viele Zeiger
- Lösung: speichere Zeiger nur für direkte Kinder



## Beispiel





# Passt das jetzt alles so?

## Bereichsanfrage

# Passt das jetzt alles so?

## Bereichsanfrage

- suche in  $y$ -Array der Wurzel  $\rightarrow O(\log n)$
- herunterlaufen im  $x$ -Baum
  - Entscheidung rechts/links  $\rightarrow O(1)$
  - finden der Begrenzungen im  $y$ -Array (ohne Suche)  $\rightarrow O(1)$
  - nur  $O(\log n)$  Lagen  $\rightarrow O(\log n)$
- Ergebnis ausgeben  $\rightarrow O(k)$

# Passt das jetzt alles so?

## Bereichsanfrage

- suche in  $y$ -Array der Wurzel  $\rightarrow O(\log n)$
- herunterlaufen im  $x$ -Baum
  - Entscheidung rechts/links  $\rightarrow O(1)$
  - finden der Begrenzungen im  $y$ -Array (ohne Suche)  $\rightarrow O(1)$
  - nur  $O(\log n)$  Lagen  $\rightarrow O(\log n)$
- Ergebnis ausgeben  $\rightarrow O(k)$

## Speicherplatz

# Passt das jetzt alles so?

## Bereichsanfrage

- suche in  $y$ -Array der Wurzel  $\rightarrow O(\log n)$
- herunterlaufen im  $x$ -Baum
  - Entscheidung rechts/links  $\rightarrow O(1)$
  - finden der Begrenzungen im  $y$ -Array (ohne Suche)  $\rightarrow O(1)$
  - nur  $O(\log n)$  Lagen  $\rightarrow O(\log n)$
- Ergebnis ausgeben  $\rightarrow O(k)$

## Speicherplatz

- nur konstanter Faktor Overhead für jedes  $y$ -Array

# Passt das jetzt alles so?

## Bereichsanfrage

- suche in  $y$ -Array der Wurzel  $\rightarrow O(\log n)$
- herunterlaufen im  $x$ -Baum
  - Entscheidung rechts/links  $\rightarrow O(1)$
  - finden der Begrenzungen im  $y$ -Array (ohne Suche)  $\rightarrow O(1)$
  - nur  $O(\log n)$  Lagen  $\rightarrow O(\log n)$
- Ergebnis ausgeben  $\rightarrow O(k)$

## Speicherplatz

- nur konstanter Faktor Overhead für jedes  $y$ -Array

## Vorbereitung

# Passt das jetzt alles so?

## Bereichsanfrage

- suche in  $y$ -Array der Wurzel  $\rightarrow O(\log n)$
- herunterlaufen im  $x$ -Baum
  - Entscheidung rechts/links  $\rightarrow O(1)$
  - finden der Begrenzungen im  $y$ -Array (ohne Suche)  $\rightarrow O(1)$
  - nur  $O(\log n)$  Lagen  $\rightarrow O(\log n)$
- Ergebnis ausgeben  $\rightarrow O(k)$

## Speicherplatz

- nur konstanter Faktor Overhead für jedes  $y$ -Array

## Vorbereitung

- Berechnung der Teilarrays (bisher): sortiere in Wurzel und spalte auf
- dabei können die nötigen Zeiger mitberechnet werden

# Passt das jetzt alles so?

## Bereichsanfrage

- suche in  $y$ -Array der Wurzel  $\rightarrow O(\log n)$
- herunterlaufen im  $x$ -Baum
  - Entscheidung rechts/links  $\rightarrow O(1)$
  - finden der Begrenzungen im  $y$ -Array (ohne Suche)  $\rightarrow O(1)$
  - nur  $O(\log n)$  Lagen  $\rightarrow O(\log n)$
- Ergebnis ausgeben  $\rightarrow O(k)$

## Speicherplatz

- nur konstanter Faktor Overhead für jedes  $y$ -Array

## Vorbereitung

- Berechnung der Teilarrays (bisher): sortiere in Wurzel und spalte auf
- dabei können die nötigen Zeiger mitberechnet werden

### Theorem

### (Verbesserte Range-Trees für $d \geq 2$ )

Für  $n$  Punkte in  $\mathbb{R}^d$  können wir Bereichsanfragen nach  $O(n \log^{d-1} n)$  Vorbereitung mit  $O(n \log^{d-1} n)$  Speicher in  $O(\log^{d-1} n + k)$  Zeit beantworten.

# Zusammenfassung

## Heute gesehen

- Verallgemeinerung von binärer Suche auf mehrere Dimensionen



# Zusammenfassung

## Heute gesehen

- Verallgemeinerung von binärer Suche auf mehrere Dimensionen
- Range Trees: verschachtelte binäre Suchbäume

# Zusammenfassung

## Heute gesehen

- Verallgemeinerung von binärer Suche auf mehrere Dimensionen
- Range Trees: verschachtelte binäre Suchbäume
- eine große Suche ist besser als viele kleine  
→ geschickte Verzeigerung spart  $\log n$

# Zusammenfassung

## Heute gesehen

- Verallgemeinerung von binärer Suche auf mehrere Dimensionen
- Range Trees: verschachtelte binäre Suchbäume
- eine große Suche ist besser als viele kleine  
→ geschickte Verzeigerung spart  $\log n$

### Theorem

### (Verbesserte Range-Trees für $d \geq 2$ )

Für  $n$  Punkte in  $\mathbb{R}^d$  können wir Bereichsanfragen nach  $O(n \log^{d-1} n)$  Vorberechnung mit  $O(n \log^{d-1} n)$  Speicher in  $O(\log^{d-1} n + k)$  Zeit beantworten.

# Zusammenfassung

## Heute gesehen

- Verallgemeinerung von binärer Suche auf mehrere Dimensionen
- Range Trees: verschachtelte binäre Suchbäume
- eine große Suche ist besser als viele kleine  
→ geschickte Verzeigerung spart  $\log n$

### Theorem

### (Verbesserte Range-Trees für $d \geq 2$ )

Für  $n$  Punkte in  $\mathbb{R}^d$  können wir Bereichsanfragen nach  $O(n \log^{d-1} n)$  Vorberechnung mit  $O(n \log^{d-1} n)$  Speicher in  $O(\log^{d-1} n + k)$  Zeit beantworten.

## Nächste Woche

- Verallgemeinerung des Konzept der geschickten Verzeigerung  
→ fractional cascading

# Zusammenfassung

## Heute gesehen

- Verallgemeinerung von binärer Suche auf mehrere Dimensionen
- Range Trees: verschachtelte binäre Suchbäume
- eine große Suche ist besser als viele kleine  
→ geschickte Verzeigerung spart  $\log n$

### Theorem

### (Verbesserte Range-Trees für $d \geq 2$ )

Für  $n$  Punkte in  $\mathbb{R}^d$  können wir Bereichsanfragen nach  $O(n \log^{d-1} n)$  Vorberechnung mit  $O(n \log^{d-1} n)$  Speicher in  $O(\log^{d-1} n + k)$  Zeit beantworten.

## Nächste Woche

- Verallgemeinerung des Konzept der geschickten Verzeigerung  
→ fractional cascading
- das lässt uns noch einen  $\log n$  Faktor bei der Anfrage sparen ( $d \geq 3$ )

# Zusammenfassung

## Heute gesehen

- Verallgemeinerung von binärer Suche auf mehrere Dimensionen
- Range Trees: verschachtelte binäre Suchbäume
- eine große Suche ist besser als viele kleine  
→ geschickte Verzeigerung spart  $\log n$

### Theorem

### (Verbesserte Range-Trees für $d \geq 2$ )

Für  $n$  Punkte in  $\mathbb{R}^d$  können wir Bereichsanfragen nach  $O(n \log^{d-1} n)$  Vorberechnung mit  $O(n \log^{d-1} n)$  Speicher in  $O(\log^{d-1} n + k)$  Zeit beantworten.

## Nächste Woche

- Verallgemeinerung des Konzept der geschickten Verzeigerung  
→ fractional cascading
- das lässt uns noch einen  $\log n$  Faktor bei der Anfrage sparen ( $d \geq 3$ )
- kostet einen zusätzlichen  $\log n$  Faktor bei Vorberechnung und Speicher