

Algorithmische Geometrie

Orthogonale Bereichsanfragen – Fractional Cascading



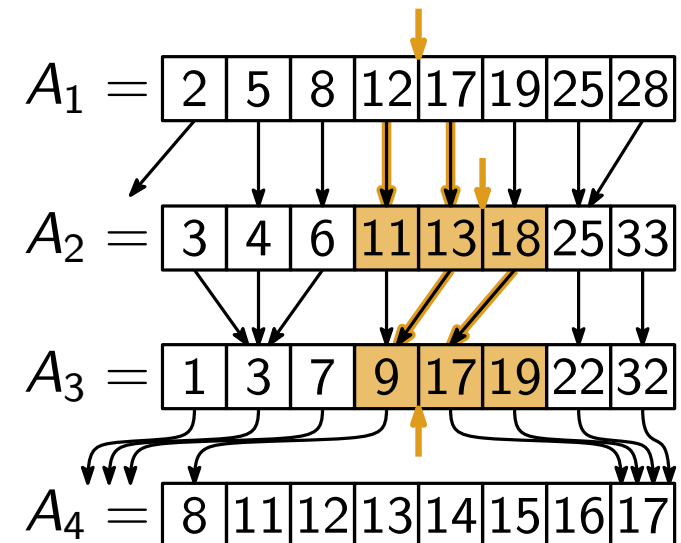
Suche in vielen Arrays

Situation

- betrachte ℓ sortierte Arrays A_1, \dots, A_k mit jeweils $\leq n$ Elementen
- finde die Position von einem x in allen Arrays
- Offensichtliche Lösung: $O(\ell \log n)$
- letzte Woche: $O(\ell + \log n)$ falls $A_1 \supseteq A_2 \supseteq \dots \supseteq A_\ell$ **Beispielanfrage: $x = 14$**

Geht $\ell + \log n$ auch im Allgemeinen?

- Idee: wie letzte Woche
 - suche x in A_1
 - finde x in A_2, \dots, A_ℓ mittels Zeiger
- Problem: Position von x in A_i lässt ggf. keine Rückschlüsse auf Position in A_{i+1} zu



Beobachtung

- $A_i \supseteq A_{i+1} \Rightarrow$ Position in A_{i+1} kann aus Position in A_i abgelesen werden
- A_i enthält viele Elemente aus $A_{i+1} \Rightarrow$ grobe Position ablesbar
- Idee: füge ein paar Elemente aus A_{i+1} in A_i ein

Fractional Cascading

Geteilte Elemente

- neues Array A'_3 : füge jedes zweite Element aus A_4 in A_3 ein
- speichere Zeiger zu Kopien
- Zeiger in A'_3 : von Elementen aus $A'_3 \setminus A_4$ zu Vorgänger/Nachfolger aus A_4
 \Rightarrow Position in A'_3 liefert Position in A_4 (± 1)
- Zeiger in A'_3 : von Elementen aus A_4 zu Vorgänger/Nachfolger aus $A'_3 \setminus A_4$
 \Rightarrow Position in A'_3 liefert Position in A_3
- kaskadiere den Prozess für alle vorherigen A_i

 $A_1 = [2 \mid 5 \mid 8 \mid 12 \mid 17 \mid 19 \mid 25 \mid 28]$
 $A_2 = [3 \mid 4 \mid 6 \mid 11 \mid 13 \mid 18 \mid 25 \mid 33]$
 $A_3 = [1 \mid 3 \mid 7 \mid 9 \mid 17 \mid 19 \mid 22 \mid 32]$
 $A_4 = [8 \mid 11 \mid 12 \mid 13 \mid 14 \mid 15 \mid 16 \mid 17]$
 $A'_1 = [1 \mid 2 \mid 4 \mid 5 \mid 7 \mid 8 \mid 11 \mid 12 \mid 14 \mid 17 \mid 18 \mid 19 \mid 25 \mid 25 \mid 28]$
 $A'_2 = [1 \mid 3 \mid 4 \mid 6 \mid 7 \mid 9 \mid 11 \mid 13 \mid 14 \mid 17 \mid 18 \mid 22 \mid 25 \mid 33]$
 $A'_3 = [1 \mid 3 \mid 7 \mid 8 \mid 9 \mid 12 \mid 14 \mid 16 \mid 17 \mid 19 \mid 22 \mid 32]$
 $A_4 = [8 \mid 11 \mid 12 \mid 13 \mid 14 \mid 15 \mid 16 \mid 17]$

Fractional Cascading – Laufzeit

Kosten für die Suche

- eine Suche in $A'_1 \rightarrow O(\log(|A'_1|))$
- $O(1)$ für jedes nachfolgende Array $\rightarrow O(\ell)$
- Gesamtlaufzeit: $O(\ell + \log(|A'_1|))$

Wie groß wird A'_1 ?

(Annahme: $|A_i| = n$ für alle i)

- $|A'_{\ell-1}| = (\frac{1}{2} + 1)n$
- $|A'_{\ell-2}| = (\frac{1}{4} + \frac{1}{2} + 1)n$
- $|A'_{\ell-3}| = (\frac{1}{8} + \frac{1}{4} + \frac{1}{2} + 1)n$
- $|A'_1| \leq 2n \quad \Rightarrow$ Suche geht in $O(\ell + \log n)$

Speicherverbrauch

- nur ein konstanter Faktor Overhead
- stimmt auch wenn nicht alle Arrays gleich groß sind

Vorberechnung

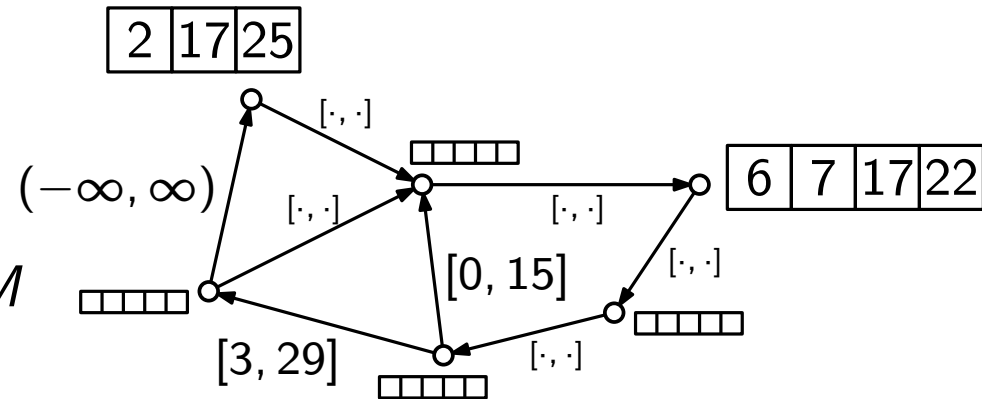
- linear in der Eingabegröße

Wie?

Fractional Cascading – etwas allgemeiner

Ausgangssituation

- eine geordnete Menge M
- gerichteter Graph $G = (V, E)$
- pro Knoten v : sortiertes Array $A_v \subseteq M$
- pro Kante e : ein Intervall $I_e \subseteq M$
- für jedes $x \in M$ und $v \in V$ ist der Grad von v eingeschränkt auf Kanten mit x in ihrem Intervall konstant



Ein Spiel zwischen Alice und Bob



- darf Datenstrukturen vorberechnen
- beantwortet die Frage
- beantwortet die Frage



- sucht sich ein $x \in M$ aus
- sucht sich ein $u \in V$ aus
- fragt wo x in A_u liegt
- wählt Kante uv mit $x \in I_{uv}$
- fragt wo x in A_v liegt

Vorbereitung: $O(s)$ Zeit und $O(s)$ Platz

Anfrage: $O(\log s)$ für die erste, danach $O(1)$ ($s =$ Gesamtgröße der Arrays)

Zurück zu den Bereichsanfragen

2-D: $O(n \log n)$ Vorberechnung/Speicher und $O(\log n + k)$ pro Anfrage
jede weitere Dimension: kostet einen $\log n$ Faktor

Genauere Laufzeitbetrachtung des 3D-Falls

- Anfrage: $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$
 Richtung: x y z
- binärer Suchbaum für x -Richtung
- jeder Knoten des x -Baums enthält 2D-DS (auf der zugehörigen Punktmenge)
- suche nach $[a_2, b_2] \times [a_3, b_3]$ in den 2D-Datenstrukturen für $O(\log n)$ Knoten
 - binärer Suchbaum für y -Richtung
 - nach z sortiertes Array für jeden Knoten im y -Baum
 - suche nach $[a_3, b_3]$ im z -Array der y -Wurzel $\rightarrow O(\log n)$
 - laufe y -Baum runter (suche $[a_2, b_2]$, verfolge $[a_3, b_3]$) $\rightarrow O(\log n)$
- Gesamtlaufzeit: $O(\log n \cdot \log n + \log n \cdot \log n)$
- $\log n \cdot \log n$ werden wir „leicht“ los:
 - suche nach $[a_3, b_3]$ im z -Array der x -Wurzel
 - verfolge $[a_3, b_3]$ beim runterlaufen im x -Baum

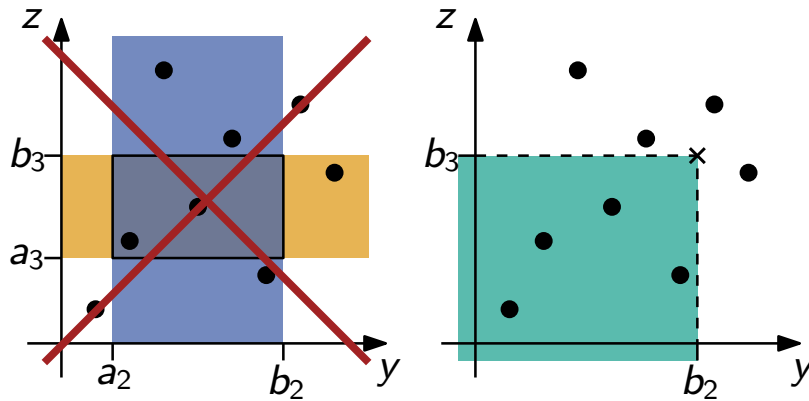
Ziel im Folgenden:
 reduziere im 2D-Fall
 $\log n + \log n$ auf $\log n$
 (einmal nach z suchen)

Halbe 2D-Bereichsanfragen

Halbe Bereichsanfragen sind halb so schwer

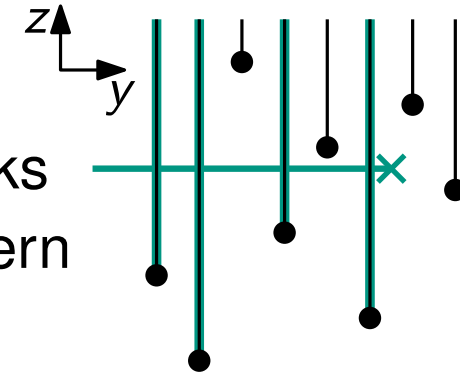
- Ziel: baue Datenstruktur, die alle Punkte in $(-\infty, b_2] \times (-\infty, b_3]$ findet (statt $[a_2, b_2] \times [a_3, b_3]$)

Ziel im Folgenden:
 reduziere im 2D-Fall
 $\log n + \log n$ auf $\log n$
 (einmal nach z suchen)



Alternative Sichtweise

- schieße Strahlen von den Punkten nach oben
- Strahl von $\langle b_2, b_3 \rangle$ nach links
- kreuzende Strahlen liefern gewünschte Punkte

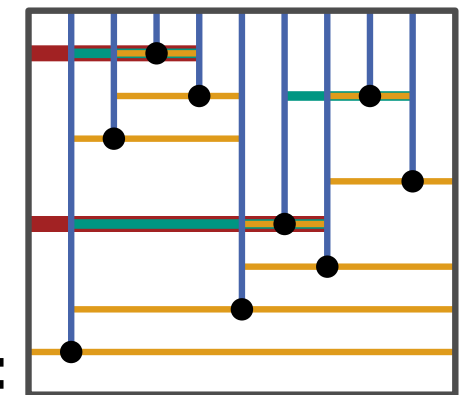


Finde die kreuzenden Strahlen

- sammle die kreuzenden Strahlen von links auf
- pro Schritt: eine Suche in z -Richtung
- wir laufen quasi von Zelle zu Zelle
- jede Zelle kennt die Zellen rechts von sich, sortiert nach z
 $\Rightarrow O(k \log n)$, wenn wir jedes mal suchen

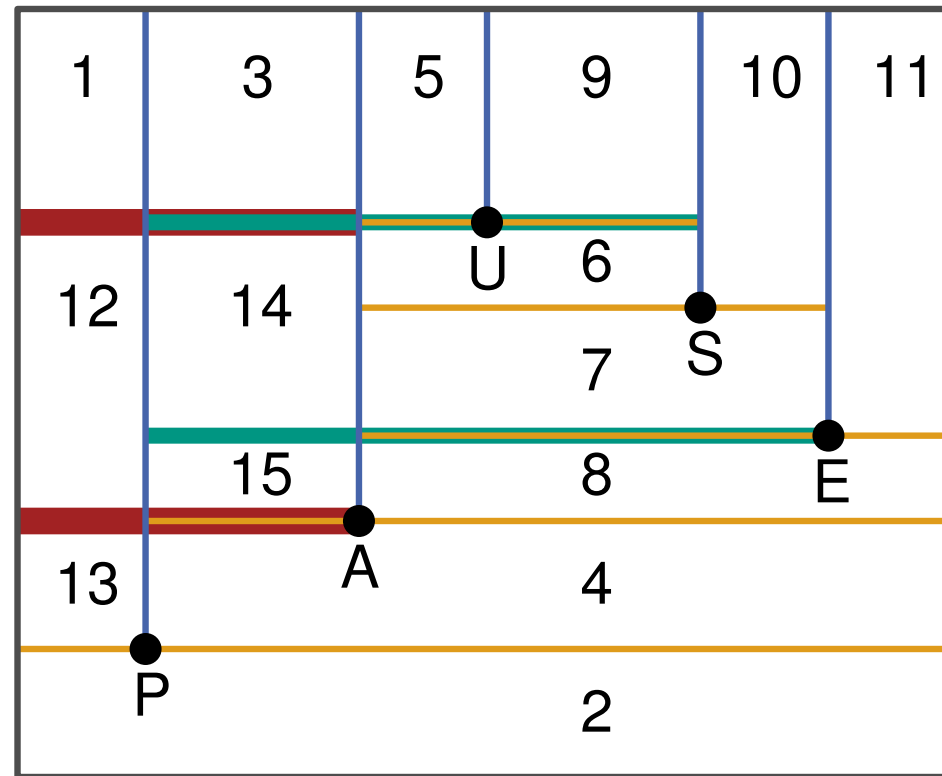
Schaffen wir $\log n + k$?

Fractional Cascading! Beispiel:



Zähle die Zellen

Wie viele Zellen erhalten wir (mit und ohne fractional cascading)?



Halbe 3D-Bereichsanfragen

Gerade gesehen

- Anfragen der Form $(-\infty, b_2] \times (-\infty, b_3]$ in $O(\log n + k)$ Ausgabegröße (DS1)

eine Suche in z-Richtung

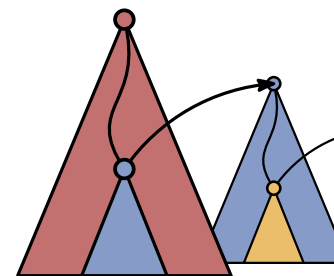
Jetzt

- Anfragen der Form $[a_1, b_1] \times (-\infty, b_2] \times (-\infty, b_3]$

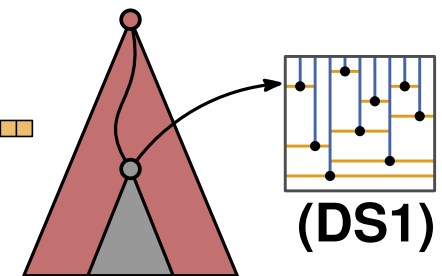
Wir wissen schon, wie das geht...

- binärer Suchbaum für x-Richtung
- jeder Knoten speichert eine (DS1) mit den entsprechenden Punkten

Letzte Vorlesung



Jetzt



Muss ich jetzt nicht in $O(\log n)$ vielen (DS1) suchen?

- ja, aber... **Fractional Cascading!**
- suche einmal in z-Richtung in der x-Wurzel
- verfolge die z-Position beim runterlaufen im x-Baum
- in (DS1) spart man sich die erste Suche \Rightarrow Gesamtlaufzeit $O(\log n + k)$

Zwei Halbe ergeben ein Ganzes

Lemma

(DS2)

Für n Punkte in \mathbb{R}^3 können $[a_1, b_1] \times (-\infty, b_2] \times (-\infty, b_3]$ -Anfragen nach $O(n \log n)$ Vorber. mit $O(n \log n)$ Speicher in $O(\log n + k)$ Zeit beantworten.

Vorgehen im Folgenden

- nutze **(DS2)** als black box
- finde Transformation, die aus $(-\infty, b_2]$ ein $[a_2, b_2]$ macht
- nutze dabei y -invertierte Variante von **(DS2)** für Anfragen der Form $[a_2, \infty)$

Können wir nicht einfach den Schnitt der Anfragen berechnen?

$$[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3] = [a_1, b_1] \times (-\infty, b_2] \times (-\infty, b_3] \cap [a_1, b_1] \times [a_2, \infty) \times [a_3, \infty)$$

Lemma

(DS3)

Für n Punkte in \mathbb{R}^3 können wir $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$ -Anfragen nach $O(n \log^2 n)$ Vorber. mit $O(n \log^2 n)$ Speicher in $O(\log n + k)$ Zeit beantworten.

Theorem

(DS4)

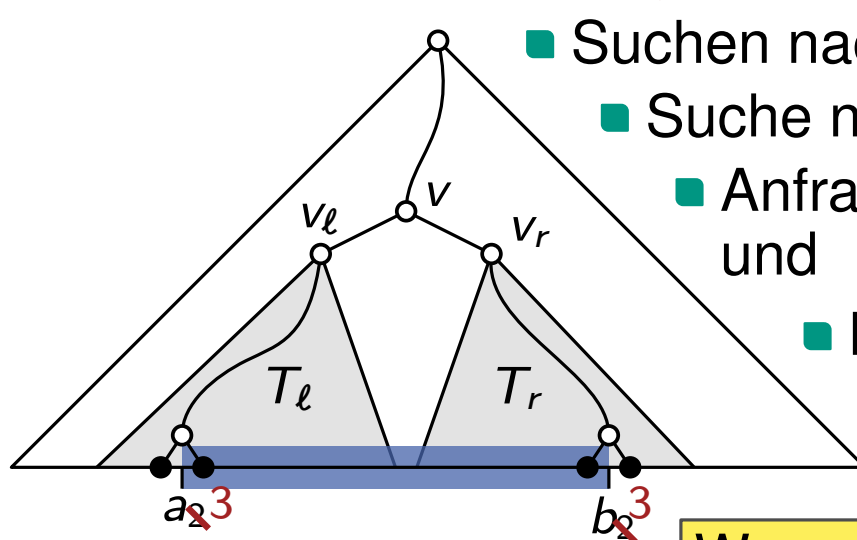
Für n Punkte in \mathbb{R}^3 können wir $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$ -Anfragen nach $O(n \log^3 n)$ Vorber. mit $O(n \log^3 n)$ Speicher in $O(\log n + k)$ Zeit beantworten.

Intervall in ~~y~~^z-Richtung

Vereinfachte Sichtweise

- ignoriere x - und ~~y~~ -Richtung
- (**DS2**)³ erlaubt uns Anfragen der Form $[a_2^3, \infty)$ und $(-\infty, b_2^3]$
- Ziel: baue neue Datenstruktur, die Anfragen der Form $[a_2^3, b_2^3]$ erlaubt

Binärer Suchbaum in ~~y~~^z-Richtung



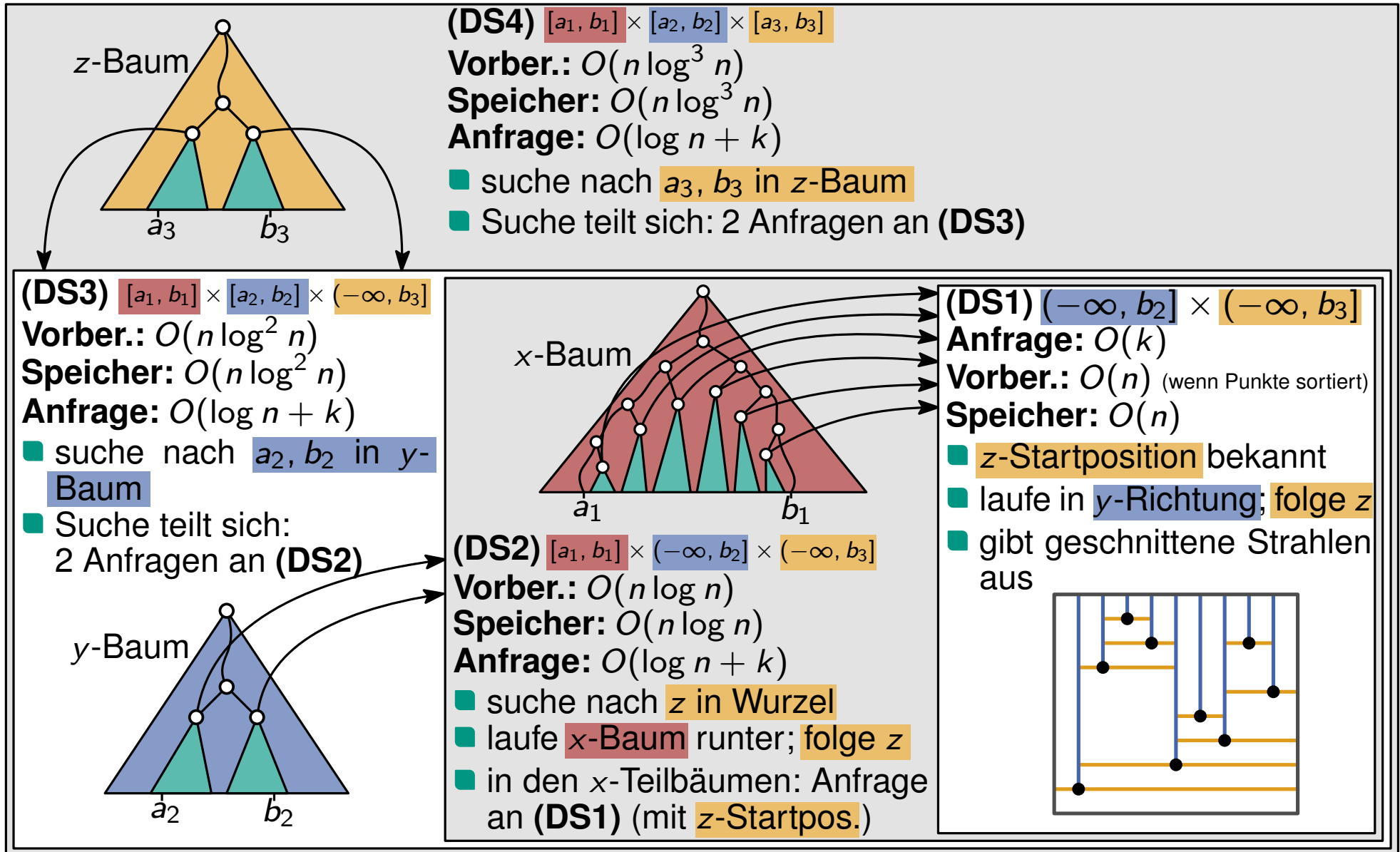
- Suchen nach a_2^3 und b_2^3 trennen sich an Knoten v
 - Suche nach a_2^3 : $\rightarrow v_l$, Suche nach b_2^3 : $\rightarrow v_r$
 - Anfragen an (**DS2**): $[a_2^3, \infty)$ auf Punkten in T_l und $(-\infty, b_2^3]$ auf Punkten in T_r
 - Laufzeit: $O(\log n + \log n + k)$
 - suche in ~~y~~ -Baum
 - zwei Anfragen in (**DS2**)³
 - Speicherplatz: $O(\log n) \cdot (\text{Platz für } \mathbf{DS2})^3$

Warum?

~~Lemma~~ Theorem

Für n Punkte in \mathbb{R}^3 können wir $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$ -Anfragen nach $O(n \log^3 n)$ Vorber. mit $O(n \log^3 n)$ Speicher in $O(\log n + k)$ Zeit beantworten. (**DS3**)⁴

The Big Picture



Zusammenfassung

Heute gesehen

- fractional cascading: nur einmal Suchen und dann Zeigern folgen ist besser als immer neu zu suchen
- Vereinfachung von Bereichsanfragen: $(-\infty, b]$ statt $[a, b]$
- schöne geometrische Lösung für $(-\infty, b_2] \times (-\infty, b_3]$
- Transformation, die aus $(-\infty, b]$ ein $[a, b]$ macht

Theorem

((DS4) $d \geq 3$)

Für n Punkte in \mathbb{R}^d können wir Bereichsanfragen nach $O(n \log^d n)$ Vorberechnung mit $O(n \log^d n)$ Speicher in $O(\log^{d-2} n + k)$ Zeit beantworten.

Was gibt es sonst noch

- diverse Anwendungen für fractional cascading
- dynamische Varianten: Punkte löschen und einfügen
- $O(\log n \cdot (\log n / \log \log n)^{d-3} + k)$ Anfragen
 $O(n \cdot (\log n / \log \log n)^{d-3})$ Speicher
- noch bessere Schranken im Word-RAM-Modell