3. Punktsuche

Raumbezogene Datenbanken (spatial databases) enthalten geometrische Objekte wie Punkte, Liniensegmente, Kurvensegmente, Polygone, Flächenstücke oder Polyeder. An solche Datenbanken werden verschiedene Suchanfragen gerichtet. Man unterscheidet [Gaede/Günther, S. 11/12]

- **Exact Match Query (EMQ):** Zu einem Objekt *o*' mit räumlicher Ausdehnung $o'.G \in E^d$ finde alle Objekte *o* mit der gleichen räumlichen Ausdehnung wie *o*'!

$$EMQ(o') = \{o / o.G = o'.G\}$$

- **Point Query (PQ):** Zu einem Punkt $p \in E^d$ finde alle Objekte *o*, die *p* enthalten!

$$PQ(p) = \{o / p \cap o.G = p\}$$

- **Range Query (RQ):** zu einem *d*-dimensionalen Interval $I^d = [l_1, u_1]x \ldots x\ [l_d, u_d]$ finde alle Objekte *o* mit mindestens einem Punkt in I^d!

$$RQ(I^d) = \{o / I^d \cap o.G \neq \emptyset\}$$

3. Punktsuche

- **Intersection Query (IQ):** Zu einem Objekt *o*' mit räumlicher Ausdehnung $o'.G \subset E^d$ finde alle Objekte *o*, die *o*' schneiden!

$$IQ(o) = \{o/o.G \cap o'.G \neq \emptyset\}$$

3. Punktsuche

- **Enclosure Query (EQ):** Zu einem Objekt *o*' mit räumlicher Ausdehnung *o*'. G E^d finde alle Objekte *o*, die *o*' umschließen!

$$EQ(o') = \{o/o'.G \cap o.G = o'.G\}$$

- **Containment Query (CQ):** Zu einem Objekt *o*' mit räumlicher Ausdehnung $o'.G \subset E^d$ finde alle Objekte *o*, die *o*' enthalten sind!

$$CQ(o') = \{o/o'.G \cap o.G = o.G\}$$

- **Adjacency Query (AQ):** Zu einem Objekt *o*' mit räumlicher Ausdehnung $o'.G \subset E^d$ finde alle zu *o*' inzidenten Objekte *o*!

$$AQ(o') = \{o/o.G \cap o'.G \neq \emptyset \wedge o.G^o \cap o'G^o = \emptyset\}$$

3. Punktsuche

.**Nearest Neighbor Query (NNQ):** Zu einem Objekt *o*' mit räumlicher Ausdehnung $o'.G \subset E^d$ finde alle Objekte *o* mit minimalem Abstand von *o*'!

$$NNQ(o') = \{o / \forall\, o'' : dist(o'.G, o.G) \leq dist(o'.G, o''.G)\}$$

•**Spatial Join:** Für zwei Mengen räumlicher Objekte *R*, *S* und ein raumbezogenes Prädikat Θ finde alle Paare $(o, o') \in R \times Smit \quad \Theta(o.G, o'.G) = TRUE!$

$$R\Theta S = \{(o/o')/o \in R \wedge o' \in S \wedge \Theta(o.G, o'.G)\}$$

Beispiele für Q sind „intersects“, „contains“, „is-enclosed-by“, „adjacent“, „northwest distance is larger than ...“.

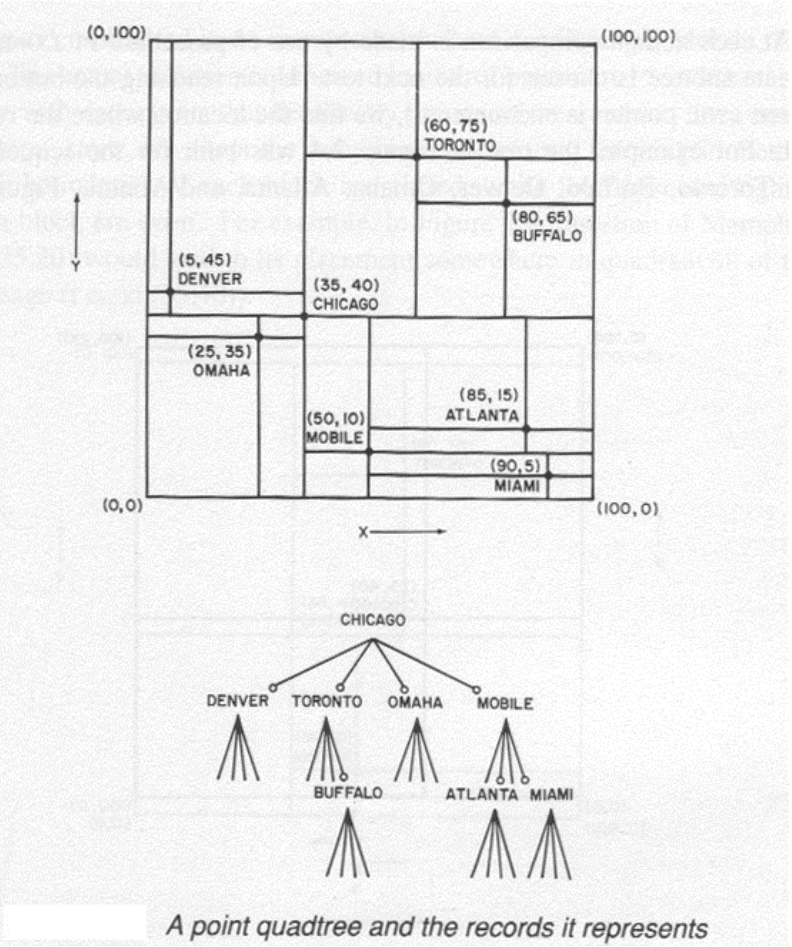
3. Punktsuche

Wir werden uns in diesem Kapitel nur mit der Point Query befassen, sowie der Range Query für Punktdatenbanken. Weitere Datenstrukturen werden in Kapitel 5 besprochen.

3.1. kd-Bäume

Eine ältere, bekannte Datenstruktur für Punktdaten ist der Point Quadtree [R.A. Finkel, J.L. Bentley, Quad trees: a data structure for retrieval on composite keys, Acta Informatica 4(1):1-9, 1974]. Hier werden die Punkte in der Datenbank zur Zerlegung der Ebene genutzt, indem die Punkte in einem Quadtree einsortiert werden und jeder Punkt ein Rechteck durch seine Koordinaten in vier kleinere Rechtecke zerlegt, die den Kindern zugeordnet werden.

3.1. kd-Bäume



A point quadtree and the records it represents

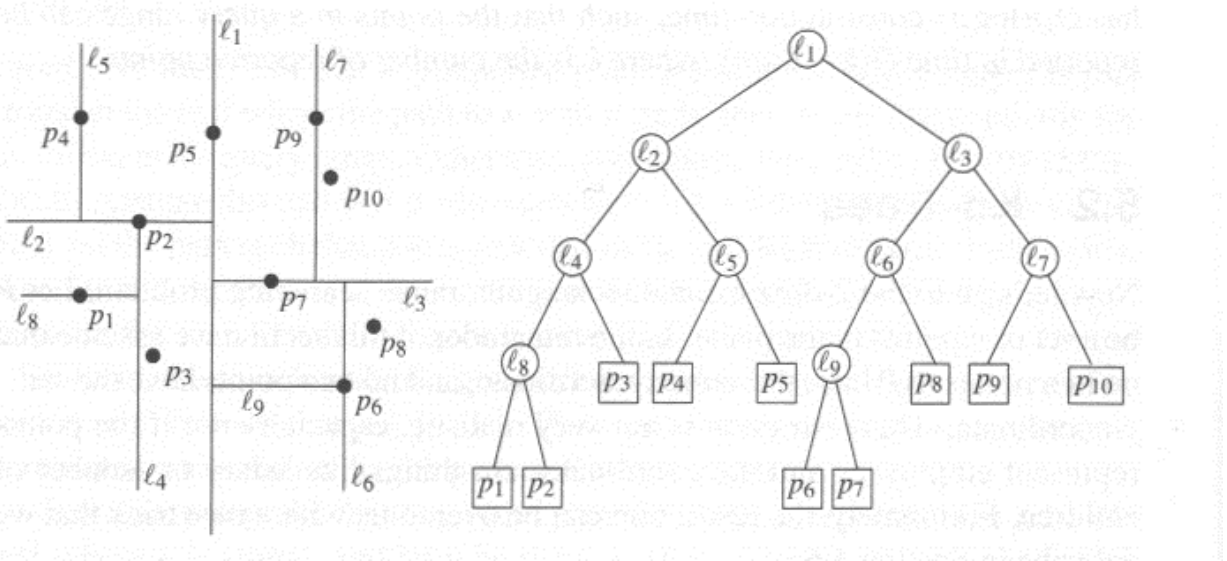
Natürlich gibt es auch Octrees für Punkte in E^3 und *d*-dim. Point Quadtrees für Punkte im E^d.

3.1 kd-Bäume

Da im Point Quadtree an jedem Knoten *d* Tests durchgeführt werden und auch *d* Werte abzulegen sind, hat man den kd-Baum eingeführt, der als binärer Baum organisiert ist und in der Praxis deutlich effizienter arbeitet, sofern die Punkte irregulär sind.

3.1 kd-Bäume

Wir betrachten eine Menge *P* von Punkten in der Ebene. Um diese in einen binären Suchbaum einsortieren zu können, unterteilen wir abwechselnd nach *x*- und nach *y*-Koordinate. Also wechselt der Sortierschlüssel von Ebene zu Ebene und jeder Knoten zerlegt ein Rechteck in zwei Rechtecke.



A kd-tree: on the left the way the plane is subdivided and on the right the corresponding binary tree

3.1 kd-Bäume

Algorithm BUILDKDTREE(P, *depth*)
Input. A set of points P and the current depth *depth*.
Output. The root of a kd-tree storing P.

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** *depth* is even
4. **then** Split P into two subsets with a vertical line ℓ through the points to the left of ℓ or on ℓ, and let P_2 be the set of points to the right of ℓ.
5. **else** Split P into two subsets with a horizontal line ℓ through the median y-coordinate of the points in P. Let P_1 be the set of points below ℓ or on ℓ, and let P_2 be the set of points above ℓ.
6. $v_{left} \leftarrow$ BUILDKDTREE(P_1, *depth* + 1)
7. $v_{right} \leftarrow$ BUILDKDTREE(P_2, *depth* + 1)
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

Damit der Algorithmus korrekt arbeitet, sollte bei geraden *n* der Median die kleinere der beiden mittleren Positionen liefern.

3.1 kd-Bäume

Der Median einer Menge von *n* Elementen kann in linearer Zeit ermittelt werden, etwa indem man beim Quicksort-Algorithmus nur das längere der beiden entstandenen Teile weiter durchsucht. (Dies ist kein worst-case linearer Algorithmus, aber zufällige Pivotwahl führt auf erwartete lineare Zeit. Ein echter worst case linearer Algorithmus ist kompliziert.)

Für unseren Algorithmus ist es jedoch klüger, gleich alle Punkte nach *x*- und *y*-Koordinate zu sortieren, da man den Median dann in konstanter Zeit finden kann und in linearer Zeit die sortierten Teilfelder für die Rekursion zu finden sind. Als Zeitkomplexität ergibt sich

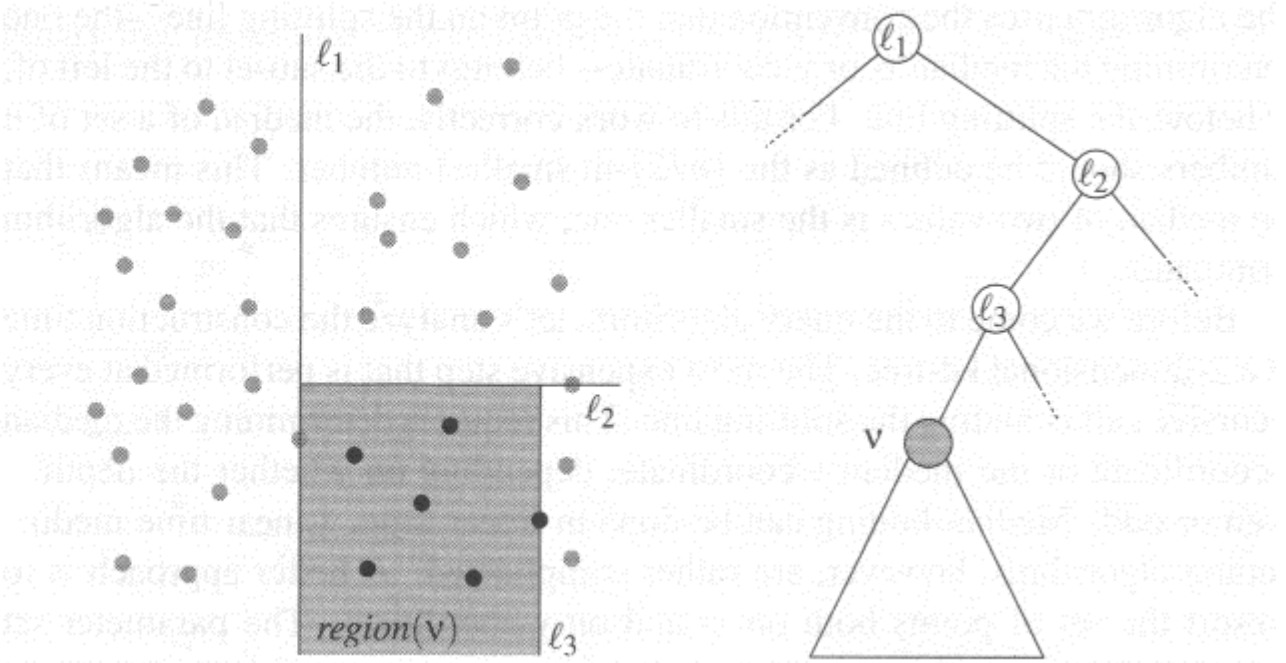
$$T(n) = \begin{cases} O(1) & n = 0,1 \\ O(n) + 2\mathrm{T}(\lceil n/2 \rceil), & n > 1 \end{cases}$$

also O(n log n), was auch für die Sortierung gilt.

Lemma 3.1: Ein kd-Baum für *n* Punkte nutzt *O*(*n*) Speicher und kann in *O*(*n* log *n*) Zeit gebaut werden.

3.1 kd-Bäume

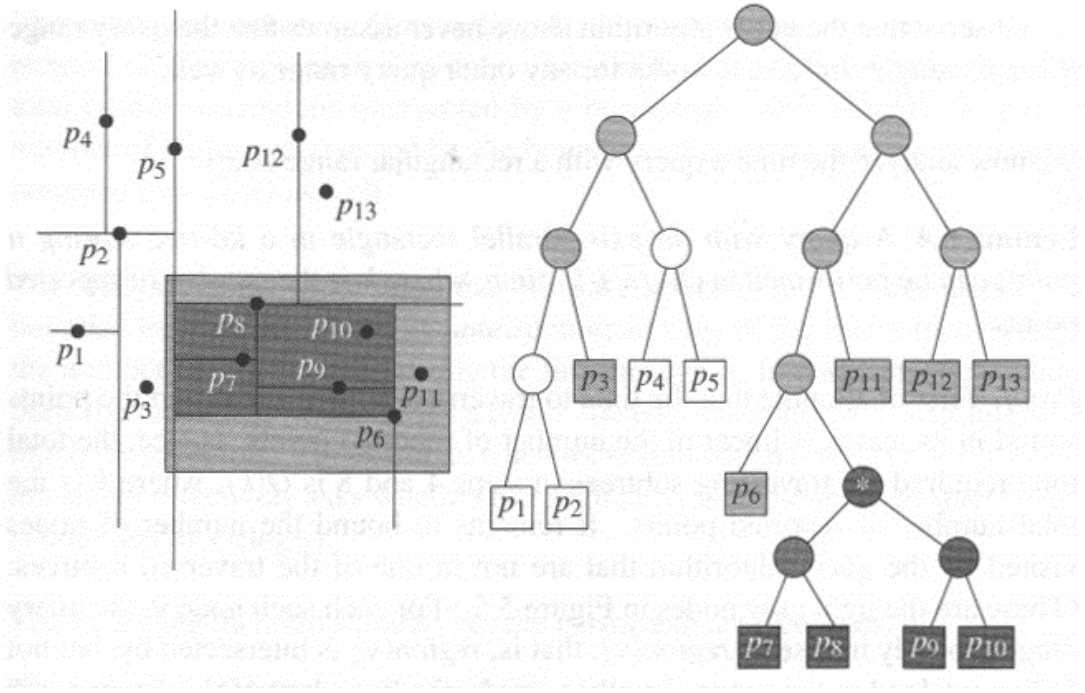
Zu jedem Knoten eines kd-Baumes gehört ein (möglicherweise unbeschränktes) achsenparalleles Rechteck der Ebene.



Correspondence between nodes in a kd-tree and regions in the plane

3.1 kd-Bäume

Wir wenden uns nun einer Range Query zu. Wir wollen also alle Punkte im kd-Baum angeben, die in einem Bereich $R = [l_1, n_1] \times [l_2, n_2]$ liegen. Dazu müssen wir bei jedem Knoten des Baumes prüfen, ob das zugehörige Rechteck R schneidet oder nicht. Wenn R geschnitten wird und das Rechteck ganz in R liegt, können wir alle Punkte im Teilbaum einfach ausgeben.



A query on a kd-tree

3.1 kd-Bäume

Algorithm SEARCHKDTREE(ν, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below ν that lie in the range.

```
1.  if ν is a leaf
2.     then Report the point stored at ν if it lies in R.
3.     else if region(lc(ν)) is fully contained in R
4.             then REPORTSUBTREE(lc(ν))
5.             else if region(lc(ν)) intersects R
6.                     then SEARCHKDTREE(lc(ν), R)
7.          if region(rc(ν)) is fully contained in R
8.             then REPORTSUBTREE(rc(ν))
9.             else if region(rc(ν)) intersects R
10.                    then SEARCHKDTREE(rc(ν), R)
```

Für die Schnitttests sollte man die achsenparallele Box des zugehörigen Knotens am besten stets übergeben und mittels der Koordinaten anpassen.

3.1 kd-Bäume

Lemma 3.2: Die Bereichssuche (Range Query) mit einem achsenparallelen Rechteck *R* in einem Kd-Baum mit *n* Punkten kann in $O(\sqrt{n} + k)$ Zeit ausgeführt werden, wobei *k* die Anzahl der Punkte in *R* ist.

Beweis: Zunächst halten wir fest, dass die Zeit zur Ausgabe eines Teilbaumes linear ist, also in den Zeilen 4 und 8 $O(k)$ Schritte anfallen. Nun müssen wir die Anzahl der Rekursionssschritte zählen. Diese fallen an, wenn die Region eines Knotens einen echten Schnitt mit *R* liefert. Dazu berechnen wir eine Schranke für die Anzahl der Unterteilungslinien, die die linke und rechte Kante von *R* schneiden.

3.1 kd-Bäume

Sei *l* eine vertikale Linie und *T* ein kd-Baum. Sei *l* (root (*T*)) die Linie zur Wurzel des kd-Baumes. *l* schneidet nun entweder die Region links von *l* (root(*T*)) oder rechts davon, aber nicht beide. Wegen der Konstruktion der kd-Bäume müssen wir noch eine Stufe tiefer gehen, bevor wir die gleiche Situation wie zu Beginn haben. Da in der 2. Ebene die Linien senkrecht zu *l* verlaufen, werden jetzt in einem Teil beide Bereiche von l getroffen. Als Rekursion für die Zeit *Q*(*n*) ergibt sich

$$Q(n) = \begin{cases} 0(1) & n = 1 \\ 2 + 2\mathrm{Q}(\lceil n/4 \rceil), & n > 1 \end{cases}$$

und letztlich $Q(n) = O(\sqrt{n})$. Die gilt natürlich analog für eine horizontale Linie. Insgesamt ergeben sich $Q(\sqrt{n})$ Schnitte mit den Rändern von *R*. QED.

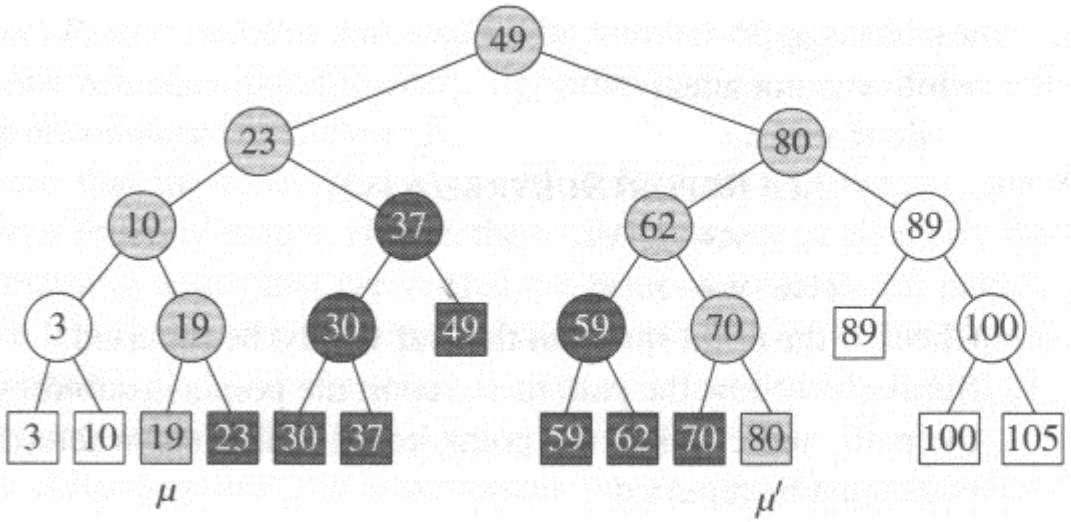
3.1 kd-Bäume

Insgesamt gilt für kd-Bäume:

Theorem 3.3: Ein kd-Baum für eine Menge *P* von n Punkten in der Ebene nutzt *O*(*n*) Speicher und kann in *O*(*n* log *n*) Zeit gebaut werden. Eine Bereichsabfrage (Range Query) benötigt $O(\sqrt{n} + k)$ Zeit, wobei *k* die Anzahl der Punkte im Bereich ist. kd-Bäume sind auch in höheren Dimensionen einsetzbar. Kontruktion erfolgt wieder in *O*(*n* log *n*) und für die Range Query ergibt sich $O(n^{1-1/d} + k)$.

3.2 Range Trees

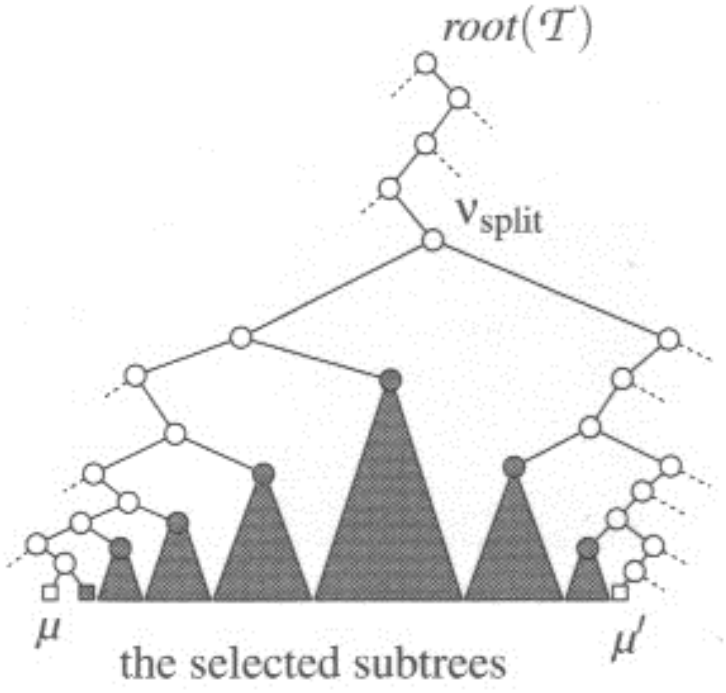
Bevor wir zwei Dimensionen und dann die höheren Dimensionen betrachten, nehmen wir uns den eindimensionalen Fall vor. Wir nutzen einen binären Suchbaum. (Arrays lassen sich nicht gut verallgemeinern.) Für ein Intervall [x, x'] suchen wir dann sowohl x als auch x' und finden so den gesuchten Teil des Baumes, etwa für [x, x'] =[18, 77] im folgenden Bild.



A 1-dimensional range query in a binary search tree

3.2 Range Trees

Wir erhalten letztlich eine Reihe von Teilbäumen, die komplett innerhalb [*x*, *x*'] liegen und die Elemente entlang der Suchpfade, die drinnen oder draußen liegen können.



3.2 Range Trees

Um zu einem Algorithmus zu kommen, müssen wir den Knoten finden, an dem obere und untere Schranke getrennt werden.

FINDSPLITNODE($\mathcal{T}, x, x'$)
Input. A tree $\mathcal{T}$ and two values x and x' with $x \leqslant x'$.
Output. The node v where the paths to x and x' split, or the leaf where both paths end.

```
1.  v ← root(T)
2.  while v is not a leaf and (x' ⩽ x_v or x > x_v)
3.      do if x' ⩽ x_v
4.             then v ← lc(v)
5.             else v ← rc(v)
6.  return v
```

3.2 Range Trees

Dann können wir den Algorithmus angeben.

Algorithm 1DRANGEQUERY($\mathcal{T}$, $[x : x']$)
Input. A binary search tree $\mathcal{T}$ and a range $[x : x']$.
Output. All points stored in $\mathcal{T}$ that lie in the range.

```
1.  v_split ← FINDSPLITNODE(T, x, x')
2.  if v_split is a leaf
3.     then Check if the point stored at v_split must be reported.
4.     else (* Follow the path to x and report the points in subtrees right of
            the path. *)
5.          v ← lc(v_split)
6.          while v is not a leaf
7.             do if x ≤ x_v
8.                   then REPORTSUBTREE(rc(v))
9.                        v ← lc(v)
10.                  else  v ← rc(v)
11.         Check if the point stored at the leaf v must be reported.
12.         Similarly, follow the path to x', report the points in subtrees left of
            the path, and check if the point stored at the leaf where the path
            ends must be reported.
```

3.2 Range Trees

Es gilt offensichtlich

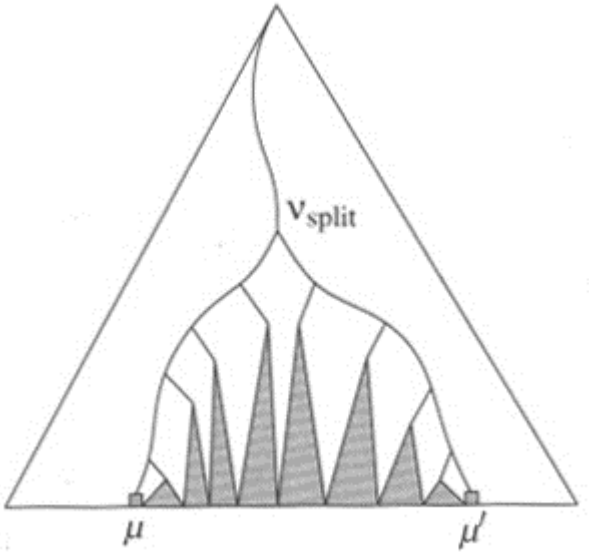
Lemma 3.4: Algorithmus 1DRANGEQUERY liefert genau die Punkte der RANGEQUERY.

Da ReportTree in linearer Zeit ablaufen kann (Grundstudium!), gilt

Theorem3.5: Sei P eine Menge von n Punkten in IR. Die Menge P kann in $O(n \log n)$ Zeit in einem balancierten binären Suchbaum gespeichert werden, der $O(n)$ Speicher benötigt. Die Range Query benötigt $O(k + \log n)$ Zeit.

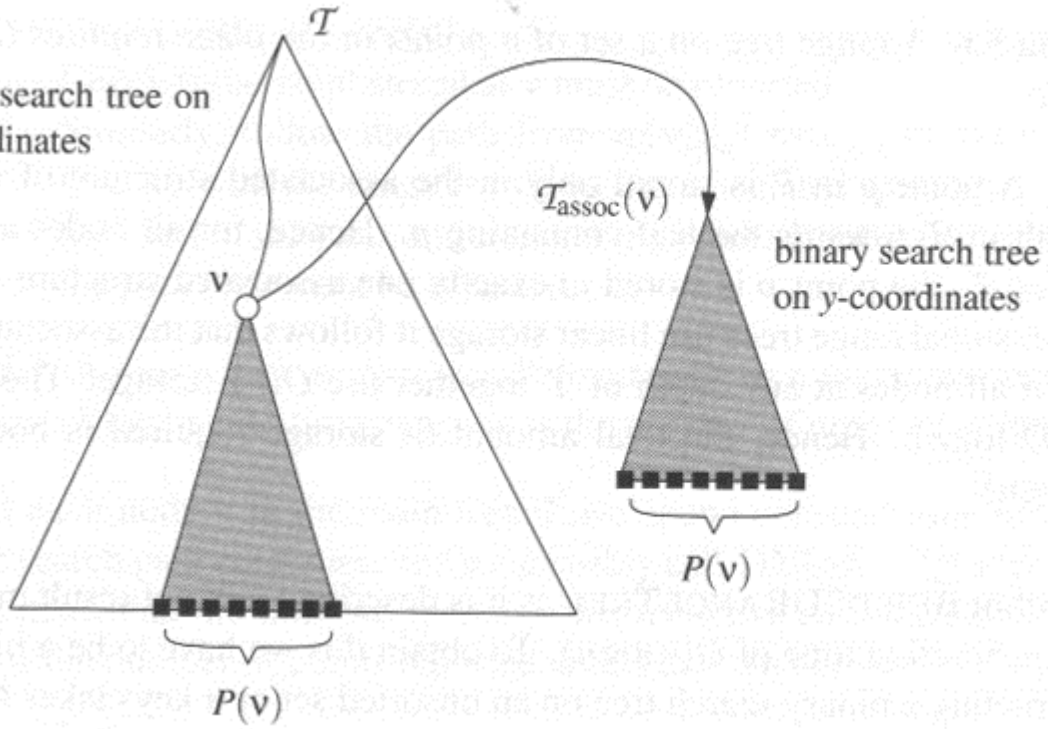
3.2 Range Trees

Um von den $O(\sqrt{n}+k)$ des planaren kd-Baumes zu $O(\log n + k)$, genauer $O(\log^2 n + k)$, zu kommen, kann man mehr Speicher einsetzen. Der Range Tree benötigt $O(n \log n)$. Wieder trennen wir x-Koordinatensuche und y-Koordinatensuche. Für eine Range Query $R = [x, x'] \times [y, y']$ lösen wir den x-Teil durch einen binären Suchbaum, der uns $\log n$ komplette Teilbäume liefert, deren Elemente in $[x, x']$ sind:



3.2 Range Trees

Um nun letztlich die Punkte in *R* zu finden, nutzen wir an jedem Knoten im binären Baum der *x*-Koordinaten einen Baum der *y*-Koordinaten, der nach dem gleichen Prinzip organisiert ist und alle Punkte im entsprechenden *y*-Teilbaum anhand der *y*-Koordinaten organisiert. Dies ist eine Datenstruktur mit mehreren Ebenen (multi-level data structure).



A 2-dimensional range tree

3.2 Range Trees

Algorithm BUILD2DRANGETREE(P)
Input. A set P of points in the plane.
Output. The root of a 2-dimensional range tree.

1. Construct the associated structure: Build a binary search tree $\mathcal{T}_{assoc}$ on the set P_y of y-coordinates of the points in P. Store at the leaves of $\mathcal{T}_{assoc}$ not just the y-coordinate of the points in P_y, but the points themselves.
2. **if** P contains only one point
3. **then** Create a leaf v storing this point, and make $\mathcal{T}_{assoc}$ the associated structure of v.
4. **else** Split P into two subsets; one subset P_{left} contains the points with x-coordinate less than or equal to x_{mid}, the median x-coordinate, and the other subset P_{right} contains the points with x-coordinate larger than x_{mid}.
5. $v_{left} \leftarrow$ BUILD2DRANGETREE(P_{left})
6. $v_{right} \leftarrow$ BUILD2DRANGETREE(P_{right})
7. Create a node v storing x_{mid}, make v_{left} the left child of v, make v_{right} the right child of v, and make $\mathcal{T}_{assoc}$ the associated structure of v.
8. **return** v

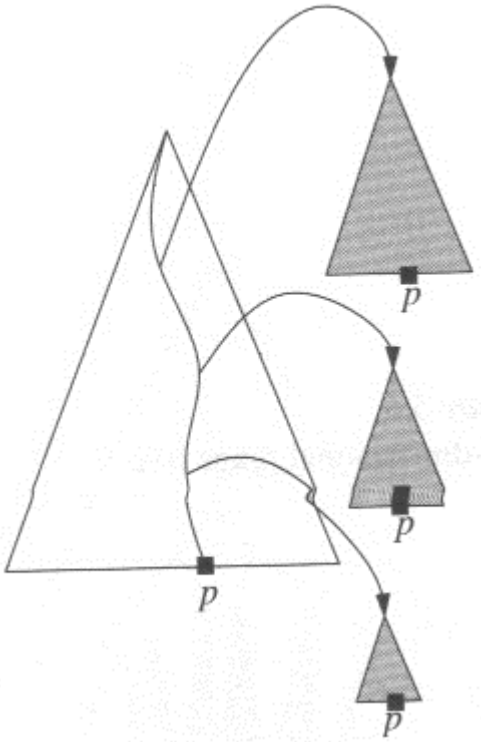
Man beachte, dass in den Blättern der *y*-Bäume die Punkte und nicht nur die *y*-Koordinaten stehen.

3.2 Range Trees

Lemma 3.6: Ein Range Tree für eine Menge von n Punkten benötigt $O(n \log n)$ Speicher.

Beweis: Für eine feste Tiefe des *x*-Baumes steht ein Punkt nur in einem der dort verankerten (assoziierten) *y*-Bäume. Da die *y*-Bäume als binäre Bäume $O(m)$ Speicher für m Elemente benötigen, benötigen alle y-Bäume $O(n)$ Speicher. Der *x*-Baum hat Tiefe $O(\log n)$, also folgt $O(n \log n)$ für den Speicher des Range Trees. QED.

3.2 Range Trees



3.2 Range Trees

Der Algorithmus BUILD2DRANGETREE benötigt mehr als $O(n \log n)$ Zeit zum Aufbau, wenn die Punkte nicht bzgl. der y-Koordinate sortiert sind, da ein binärer Suchbaum im Allgemeinen nur in $O(m \log m)$ zu bauen ist. Nach einer Sortierung der Punkte genügt $O(m)$ und wir erhalten $O(n \log n)$ auch für die Zeit.

3.2 Range Trees

Mit Hilfe von 1DRANGEQUERY lässt sich nun auch 2DRANGEQUERY formulieren.

Algorithm 2DRANGEQUERY($\mathcal{T}, [x:x'] \times [y:y']$)
Input. A 2-dimensional range tree $\mathcal{T}$ and a range $[x:x'] \times [y:y']$.
Output. All points in $\mathcal{T}$ that lie in the range.

1. $v_{split} \leftarrow$ FINDSPLITNODE($\mathcal{T}, x, x'$)
2. **if** v_{split} is a leaf
3. **then** Check if the point stored at v_{split} must be reported.
4. **else** (∗ Follow the path to x and call 1DRANGEQUERY on the subtrees right of the path. ∗)
5. $v \leftarrow lc(v_{split})$
6. **while** v is not a leaf
7. **do if** $x \leqslant x_v$
8. **then** 1DRANGEQUERY($\mathcal{T}_{assoc}(rc(v)), [y:y']$)
9. $v \leftarrow lc(v)$
10. **else** $v \leftarrow rc(v)$
11. Check if the point stored at v must be reported.
12. Similarly, follow the path from $rc(v_{split})$ to x', call 1DRANGEQUERY with the range $[y:y']$ on the associated structures of subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.

3.2 Range Trees

Lemma 3.7: Eine Bereichsuche mit einem achsenparallelen Rechteck in einem Range Tree mit n Punkten benötigt $O(\log^2 n + k)$ Zeit, wobei k die Anzahl der Elemente im gesuchten Bereich ist.

Beweis: An jedem der bis zu $O(\log n)$ Punkte entlang des Suchpfades rufen wir möglicherweise 1DRANGEQUERY auf. Dies erfordert $O(\log n + k_v)$ Schritte, wobei k_v die im Teilbaum gesuchten Punkte sind. Insgesamt ergeben sich $O(\log^2 n + k)$ Schritte, da $\Sigma\, k_v = k$. QED.

Theorem 3.8: Sei P eine Menge von n Punkten in der Ebene. Ein Range Tree für P nutzt $O(n \log n)$ Speicher und kann in $O(n \log n)$ gebaut werden. Die Bereichsabfrage (Range Query) mit einem achsenparallenen Rechteck erfordert $O(\log^2 n + k)$ Schritte, wobei k die Anzahl der Punkte im Rechteck ist.

3.3 Trapezoidal Maps

Problemstellung

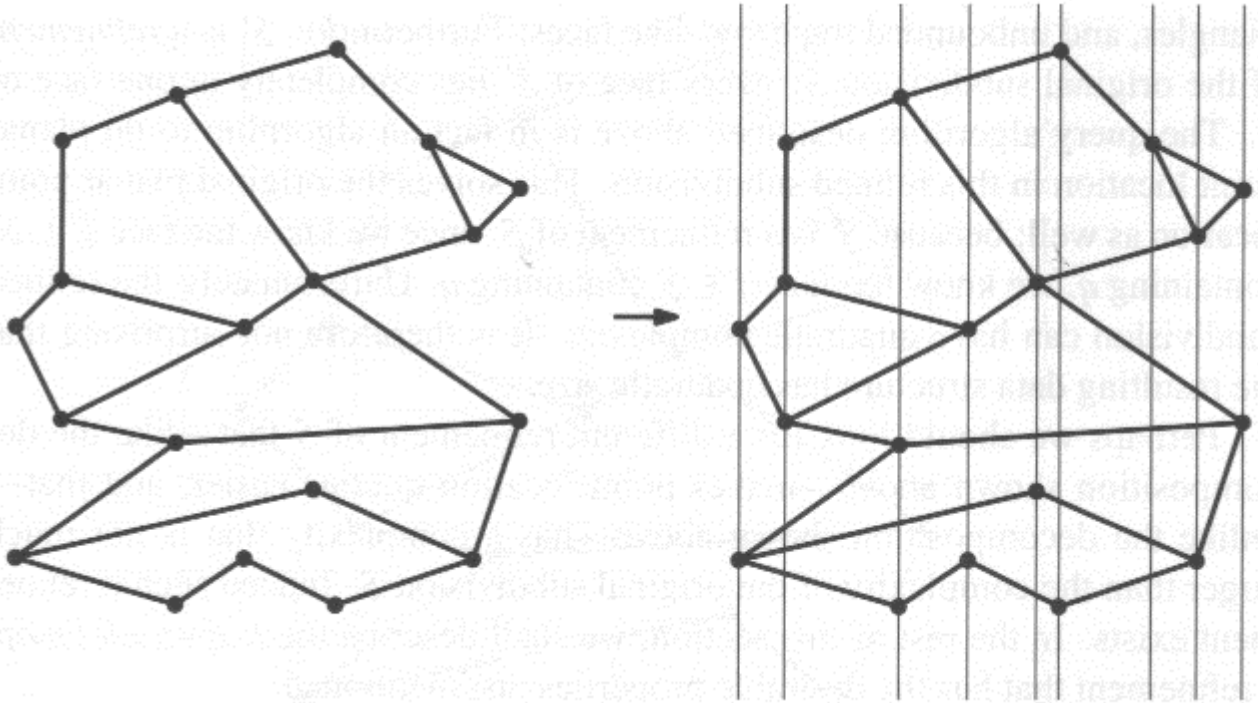
Wir haben bisher nur Punkte in unserer Datenstruktur abgelegt und dann Exact Match Query und Range Query durchgeführt. Eine typische Aufgabe ist jedoch insbesondere, eine Point Query auf einer planaren Unterteilung durchzuführen. Dazu dienen unter anderem trapezförmige Karten (Trapezoidal Maps).

3.3 Trapezoidal Maps

Erste Idee

Um auf den richtigen Weg zu gelangen, führen wir senkrechte Linien durch jeden Eckpunkt der planaren Unterteilung ein. Es entstehen Scheiben aus Stücken, die alle trapezförmig sind, da keine Schnittpunkte oder Eckpunkte zwischen den senkrechten Linien liegen. Man kann nun die Trapeze zwischen zwei Linien von oben nach unten ordnen und in $O(\log n)$ das richtige Stück zu einem Punkt ermitteln. Die gesamte Suche erfordert $O(\log 2n)$ Schritte für die Suche der richtigen beiden Linien und $O(\log n)$ für das richtige Stück, falls n die Anzahl der Karten in der Unterteilung ist.

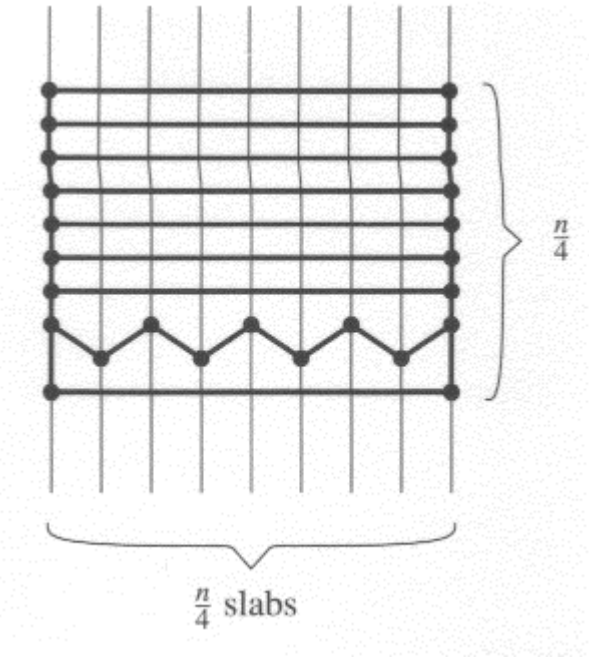
3.3 Trapezoidal Maps



Partition into slabs

3.3 Trapezoidal Maps

Leider benötigt man ein nach den *x*-Koordinaten sortiertes Array der Punkte mit $O(n)$ Speicher und n Arrays für die Scheiben (Bereich zwischen zwei Linien), die jeweils auch $O(n)$ Speicher benötigen können. Ein Beispiel mit $n/4$ Scheiben mit jeweils $n/4$ Stücken ist unten skizziert. Also ist $O(n^2)$ Speicher nötig und unsere Idee benötigt deutliche Verbesserungen.



3.3 Trapezoidal Maps

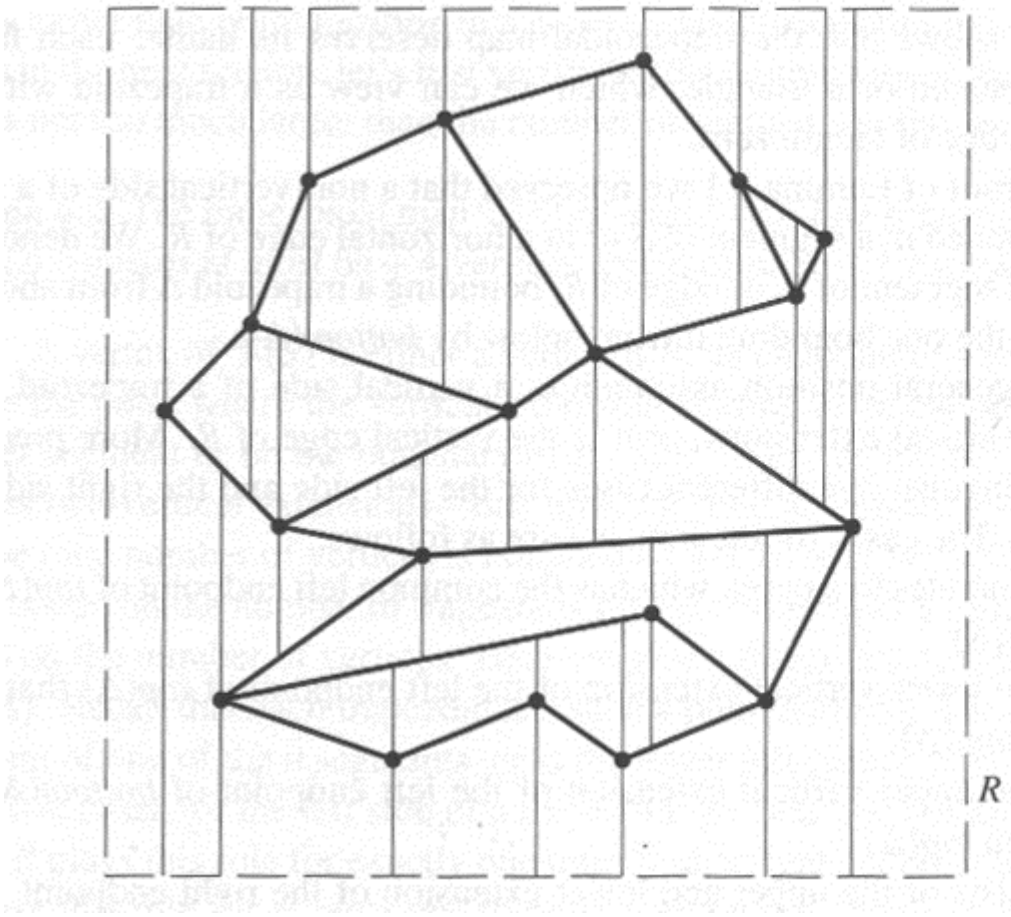
Datenstruktur und Algorithmus

Wir formulieren nun trapezförmige Karten als Lösung. Dazu definieren wir ein alle Eckpunkte enthaltendes Rechteck *R*, so dass wir nur diesen Bereich untersuchen müssen und keine unbeschränkten Stücke behandeln. Ferner nehmen wir zunächst an, dass zwei Eckpunkte verschiedene *x*-Koordinaten haben. Dies korrigieren wir später.

Eine solche planare Unterteilung heißt **planare Unterteilung in allgemeiner Lage**.

Definition 3.9: Sei *S* eine planare Unterteilung in allgemeiner Lage. Die trapezförmige Karte *T*(*S*) von *S* entsteht, indem man zwei vertikale Verlängerungen jedes Endpunktes *P* durch die benachbarten Facetten zeichnet. Die Verlängerung in positiver y-Richtung heißt **obere vertikale Verlängerung**, die andere (in negativer *y*-Richtung) **untere vertikale Verlängerung**.

3.3 Trapezoidal Maps



A trapezoidal map

3.3 Trapezoidal Maps

Lemma 3.10: Eine Facette von $T(S)$ hat eine oder zwei vertikale Seiten und genau zwei nicht-vertikale Seiten.

Beweis: Sei f eine Facette von $T(S)$. Wir zeigen zunächst, dass f konvex ist. Da sich die Kanten von S nicht schneiden, sind die Ecken von f entweder Eckpunkte von S oder R oder Schnitte vertikaler Verlängerungen mit Kanten von S oder R. Wegen der vertikalen Segmente können die Schnittpunkte keine inneren Winkel größer 180^o haben. Für Ecken von R gilt das auch und wegen der vertikalen Verlängerungen auch für die Ecken von S. Also ist f konvex.

3.3 Trapezoidal Maps

Nun zeigen wir die Aussagen des Lemmas.

Hätte *f* mehr als zwei nicht-vertikale Seiten, müssten zwei benachbart sein. Diese begrenzen *f* nach oben oder unten wegen der Konvexität. Da jede solche Seite ein Teil einer Kante von *R* oder *S* ist und sich diese alle nicht schneiden, gibt es einen Eckpunkt, an dem die beiden Elemente ineinander übergehen. Als unterer oder oberer Rand würden diese Seiten aber durch vertikale Segmente getrennt. Also gibt es maximal zwei nicht-vertikale Seiten.

Schließlich, da *f* beschränkt ist, muss *f* mind. eine vertikale Seite haben (mind. drei Seiten). QED

3.3 Trapezoidal Maps

Wir bezeichnen die obere Seite des Trapez Δ mit top(Δ) und die untere mit bottom(Δ). Die vertikalen Seiten stammen entweder von einer Verlängerung oder von R. Es gibt 5 Fälle für die linke Seite:

(a) Die Seite besteht aus einem Punkt, dem Eckpunkt zwischen top(Δ) und bottom(Δ).

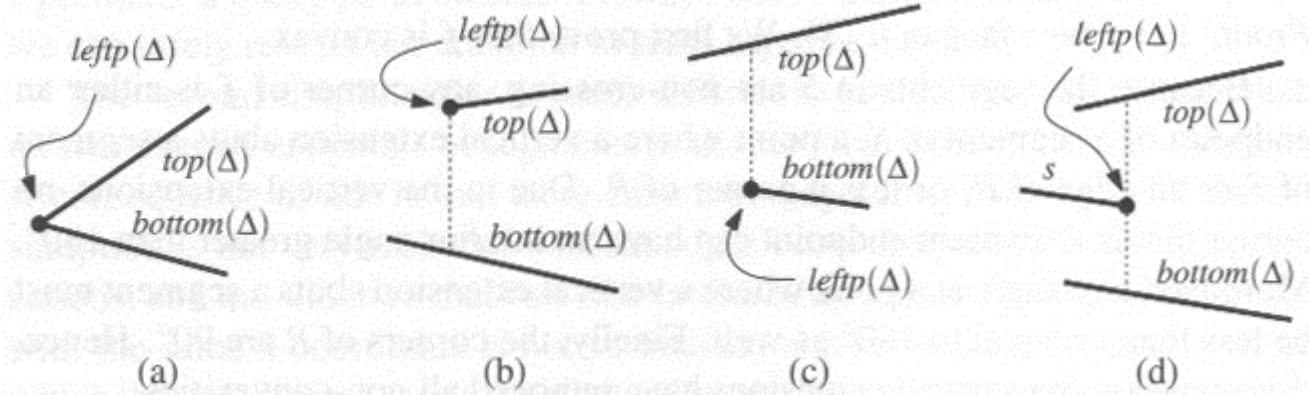
(b) Die Seite ist die untere Verlängerung des linken Eckpunktes von top(Δ) , die bottom(Δ) schneidet und im Schnittpunkt endet.

(c) Die Seite ist die obere Verlängerung des linken Eckpunktes von bottom (Δ), die top(Δ) schneidet und im Schnittpunkt endet.

(d) Die Seite besteht aus der oberen und unteren Verlängerung des rechten Eckpunktes Δ eines weiteren Segmentes *s*.

(e) Die Seite ist der linke Rand von *R*. Dieser Fall tritt genau einmal auf.

3.3 Trapezoidal Maps



Four of the five cases for the left edge of trapezoid Δ

3.3 Trapezoidal Maps

Da die linke Seite außer in (*e*) stets durch einen Endpunkt definiert wird, nennen wir diesen leftp(Δ). Analog gibt es ein rightp(Δ). Δ ist dann durch top(Δ), bottom(Δ), leftp(Δ) und rightp(Δ) definiert.

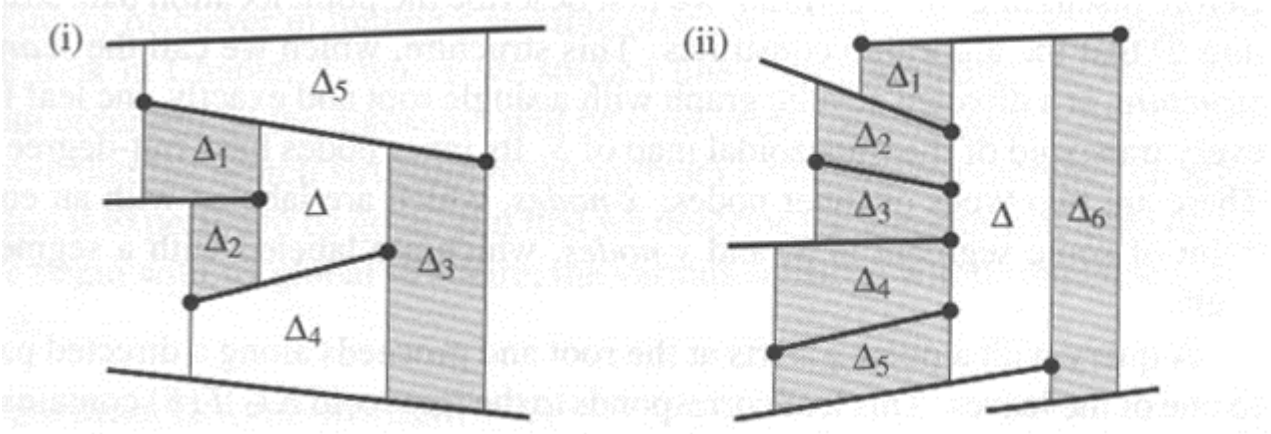
Lemma 3.11: Die trapezförmige Karte *T*(*S*) einer planaren Unterteilung *S* in allgemeiner Lage enthält maximal 6*n* + 4 Eckpunkte und 3*n* +1 Trapeze.

Beweis: Ein Eckpunkt von *T*(*S*) ist entweder ein Eckpunkt von *S* bzw. *R* oder ein Schnitt einer vertikalen Verlängerung mit einer Kante von *S* bzw. *R*. Zu jedem Eckpunkt von *S* gibt es zwei Verlängerungen, also insgesamt maximal 4 + 2*n* + 2(2*n*) = 6*n* + 4 Eckpunkte in *T*(*S*).

Neben der Eulerformel lassen sich die Trapeze auch direkt zählen: Jedes Trapez hat einen Punkt leftp(Δ). Die untere linke Ecke von *R* ergibt ein leftp(Δ). Ein rechter Endpunkt eines Segmentes tritt als maximal ein leftp(Δ) auf. Ein linker Endpunkt kann leftp(Δ) für maximal zwei Trapeze sein. (Dabei zählen wir die Segmente bottom(Δ), top(Δ) und nicht die Eckpunkte in S, da diese leftp(Δ) für viele Δ sein können, wenn mehrere Segmente dort zusammentreffen.) Es gibt n Segmente, also max. 3n + 1 Trapeze. QED

3.3 Trapezoidal Maps

Wir nennen Trapeze **benachbart (inzident)**, falls sie sich entlang einer vertikalen Kante treffen. Wenn *S* in allgemeiner Lage ist (links), gibt es maximal 4 benachbarte Trapeze für ein ausgewähltes Trapez.



Trapezoids adjacent to Δ are shaded.

Wenn benachbarte Trapeze das gleiche Segment von *S* als top(Δ) haben, sprechen wir von **upper left neighbor**, falls das andere Trapez links liegt. Analog gibt es **lower left neighbor**, **upper right neighbor** und **lower right neighbor**.

3.3 Trapezoidal Maps

Als Datenstruktur verwenden wir nicht die doppelt verknüpfte Kantenliste aus 2.3. Stattdessen legen wir Felder der Eckpunkte und der Segmente von S an und ein Feld der Trapeze, wobei ein Trapez leftp(Δ), rightp(Δ), bottom(Δ), top(Δ) und Zeiger auf die vier Nachbarn enthält.

3.3 Trapezoidal Maps

Der Aufbau der Datenstruktur erfolgt auch nicht durch einen Plane Sweep, damit wir eine gute Suchstruktur *D* für Point Queries erhalten! Diese Suchstruktur ist ein gerichteter azyklischer Graph mit genau einem Blatt für jedes Trapez. Die inneren Knoten haben stets zwei wegführende Kanten. Es gibt zwei Typen innerer Knoten:

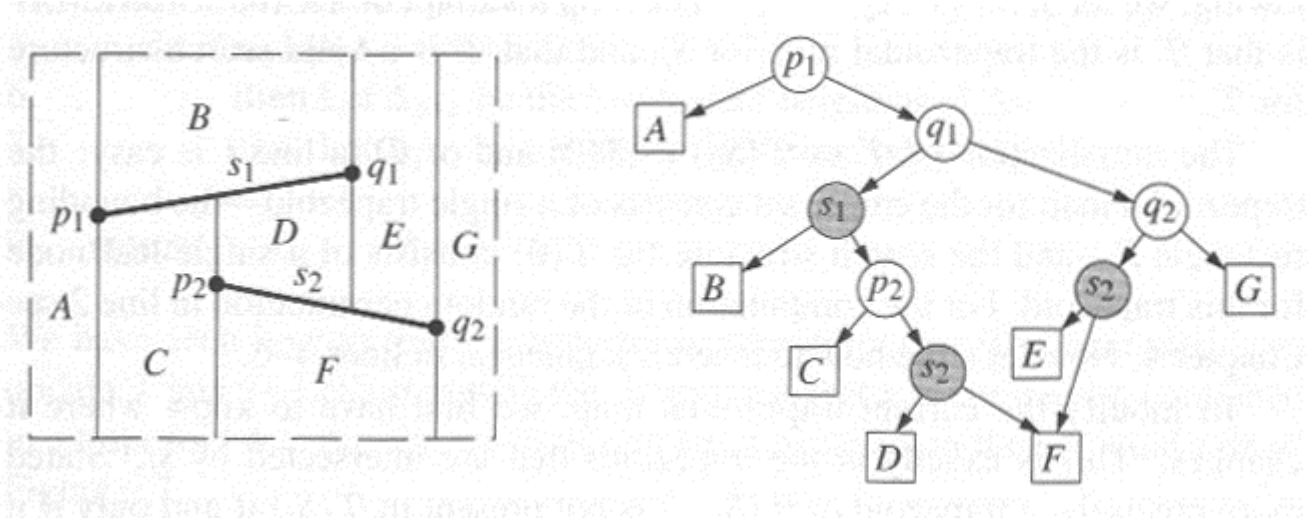
- *x*-Knoten, die auf einen Eckpunkt in *S* verweisen,

- *y*-Knoten, die auf ein Segment in S verweisen.

3.3 Trapezoidal Maps

Die Point Query nach einem Punkt q beginnt an der Wurzel und läuft zu einem Blatt. Dieses Blatt gehört zu dem Trapez, das q enthält. Bei x-Knoten wird gefragt, ob q links oder rechts der vertikalen Linie durch diesen Punkt liegt. Bei y-Knoten wird gefragt, ob q oberhalb oder unterhalb des Segmentes liegt. Dabei stellen wir sicher, dass bei einem y-Knoten die vertikale Linie durch q das Segment schneidet.

Falls q direkt auf der vertikalen Linie oder dem Segment liegt, haben wir ein kleines Problem, das wir später lösen werden.



The trapezoidal map of two segments and a search structure

3.3 Trapezoidal Maps

Der Aufbau von *D* und (über die Blätter) *T*(*S*) erfolgt **inkrementell,** indem wir ein Segment nach dem anderen einfügen. Ferner sortieren wir die Segmente **zufällig**, um zu einer effektiven Suchstruktur zu gelangen. (Die Effektivität zeigen wir anschließend.)

Algorithm TRAPEZOIDALMAP(S)
Input. A set S of n non-crossing line segments.
Output. The trapezoidal map $\mathcal{T}(S)$ and a search structure $\mathcal{D}$ for $\mathcal{T}(S)$ in a bounding box.

1. Determine a bounding box R that contains all segments of S, and initialize the trapezoidal map structure $\mathcal{T}$ and search structure $\mathcal{D}$ for it.
2. Compute a random permutation $s_1, s_2, \ldots, s_n$ of the elements of S.
3. **for** $i \leftarrow 1$ **to** n
4. **do** Find the set $\Delta_0, \Delta_1, \ldots, \Delta_k$ of trapezoids in $\mathcal{T}$ properly intersected by s_i.
5. Remove $\Delta_0, \Delta_1, \ldots, \Delta_k$ from $\mathcal{T}$ and replace them by the new trapezoids that appear because of the insertion of s_i.
6. Remove the leaves for $\Delta_0, \Delta_1, \ldots, \Delta_k$ from $\mathcal{D}$, and create leaves for the new trapezoids. Link the new leaves to the existing inner nodes by adding some new inner nodes, as explained below.

3.3 Trapezoidal Maps

Es sei $S_i := \{s_1, \cdots, s_i\}$ die Menge der ersten *i* Segmente. Die Invariante am Ende der Schleife ist, dass *T* die trapezförmige Karte zu S_i und *D* eine zulässige Suchstruktur für *T* ist.

Am Anfang ist S_0 leer und R bildet das einzige Trapez. Für das Update müssen wir wissen, welche Veränderung es durch s_i gibt. Dies sind die Trapeze, die von s_i geschnitten werden. Seien $\Delta_0, \cdots, \Delta_k$ diese Trapeze von links nach rechts.

Dann ist Δ_{j+1} rechter Nachbar von Δ_j und zwar upper right neighbor, falls rightp(Δ_j) unterhalb s_i liegt und lower right neighbor, falls rightp(Δ_j) oberhalb s_i liegt. Also können wir ausgehend von Δ_0 Δ_1, ... , Δ_k finden. Um Δ_0 zu finden, führen wir eine Point Query nach dem linken Startpunkt *p* von S_i in $T(S_{i-1})$ durch. Wenn *p* noch nicht in S_{i-1} liegt, erhalten wir Δ_0 . Andernfalls kann *p* auf der vertikalen Linie in einem *x*-Knoten liegen. Dann laufen wir stets rechts weiter (alle interessierenden Segmente s und s_i haben *p* als linken Endpunkt), vergleichen die Steigung von s und die Steigung von s_i. Wenn die Steigung von s_i größer ist, liegt *p* oberhalb von *s*, sonst darunter.

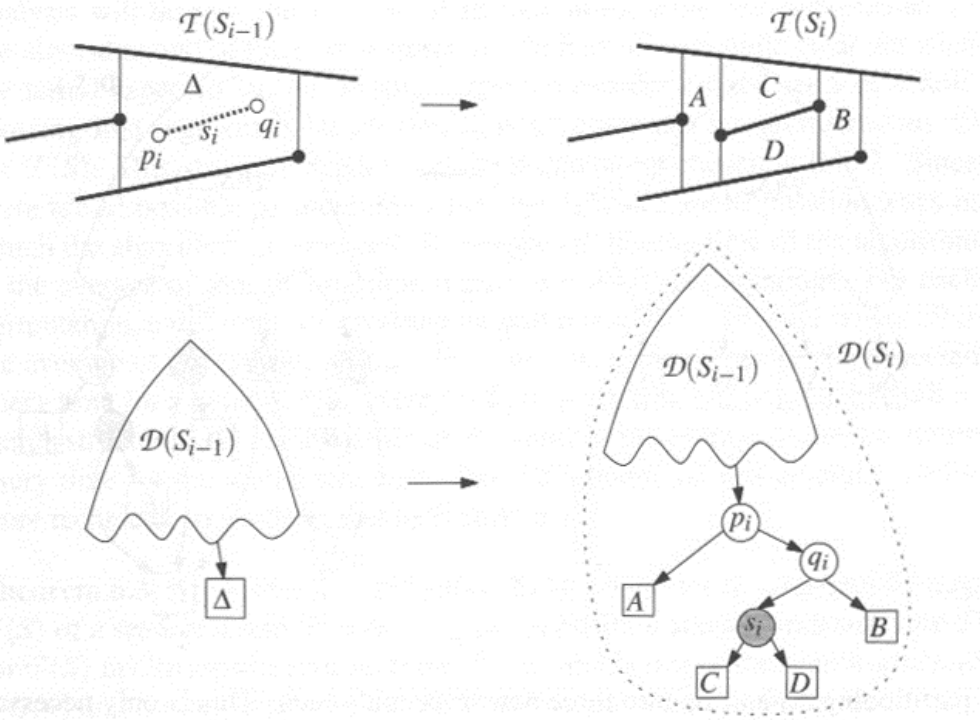
3.3 Trapezoidal Maps

Algorithm FOLLOWSEGMENT($\mathcal{T}, \mathcal{D}, s_i$)
Input. A trapezoidal map $\mathcal{T}$, a search structure $\mathcal{D}$ for $\mathcal{T}$, and a new segment s_i.
Output. The sequence $\Delta_0, \ldots, \Delta_k$ of trapezoids intersected by s_i.

1. Let p and q be the left and right endpoint of s_i.
2. Search with p in the search structure $\mathcal{D}$ to find Δ_0.
3. $j \leftarrow 0$;
4. **while** q lies to the right of $rightp(\Delta_j)$
5. **do if** $rightp(\Delta_j)$ lies above s_i
6. **then** Let Δ_{j+1} be the lower right neighbor of Δ_j.
7. **else** Let Δ_{j+1} be the upper right neighbor of Δ_j.
8. $j \leftarrow j + 1$
9. **return** $\Delta_0, \Delta_1, \ldots, \Delta_j$

3.3 Trapezoidal Maps

Nun ist das Update von T und D zu bestimmen. Im einfachen Fall mit s_i in $\Delta = \Delta_0$ ergibt sich folgendes Bild

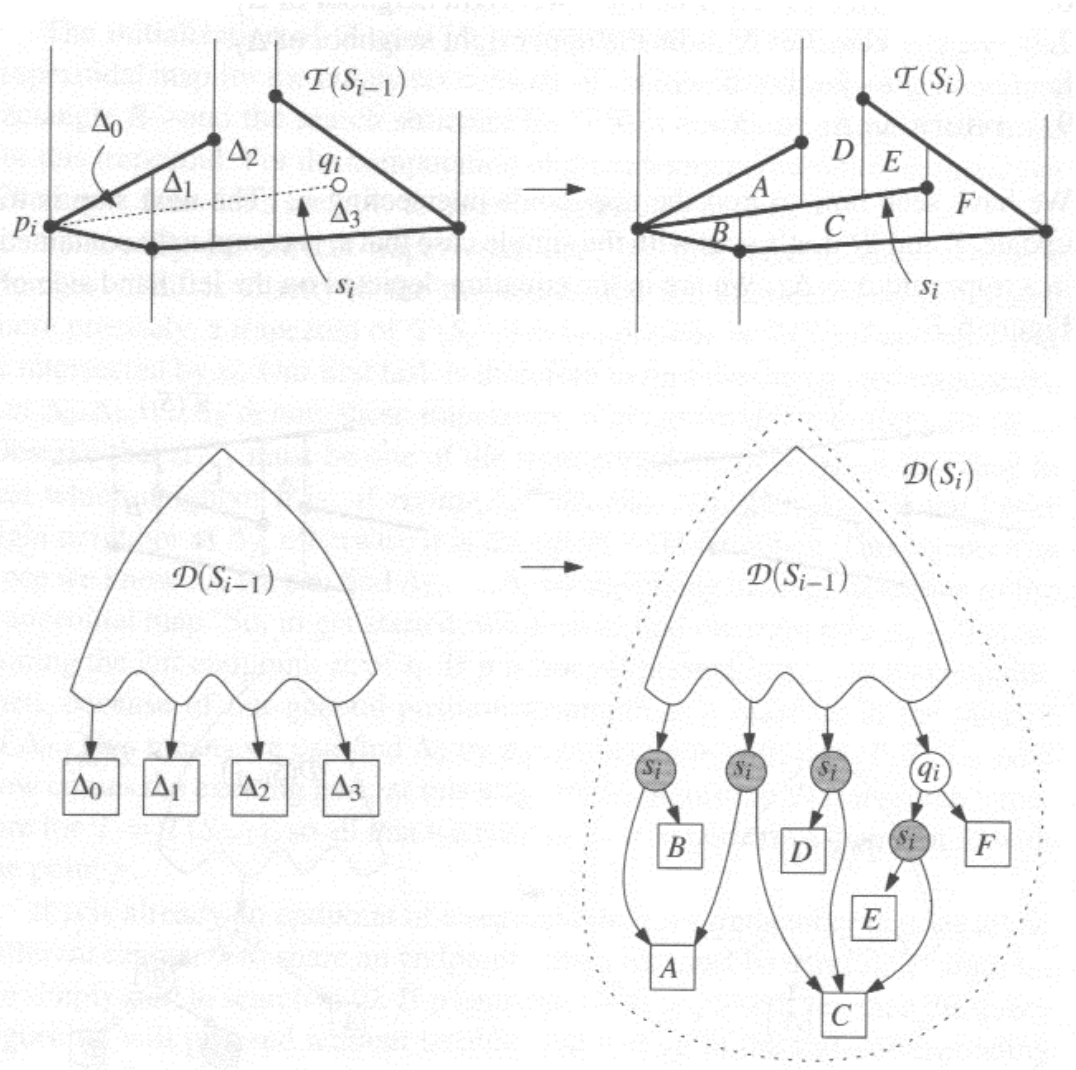


The new segment s_i lies completely in trapezoid Δ.

3.3 Trapezoidal Maps

Wenn s_i mehrere Trapeze unterteilt, ist die Suchstrukturänderung etwas komplizierter, aber noch immer gut durchführbar. Wir erzeugen vertikale Verlängerungen der Endpunkte von s_i in Δ_0 und Δ_k , sofern die Endpunkte nicht bereits in s_{i-1} liegen, wodurch Trapeze erzeugt werden. Dies und das Update von Δ sind wieder bildlich erläutert.

3.3 Trapezoidal Maps



Segment s_i intersects four trapezoids.

3.3 Trapezoidal Maps

Wenn Δ_0 den linken Endpunkt von s_i im Inneren hat, wird das Blatt zu Δ_0 durch einen *x*-Knoten mit dem Endpunkt und einem *y*-Konten für s_i ersetzt. Analog wird das Blatt zu Δ_k, sofern der rechte Endpunkt von s_i im Inneren von Δ_k liegt, durch einen *x*-Knoten und einen y-Knoten ersetzt. Δ_1 bis Δ_{k-1} werden dann durch *y*-Knoten für s_i ersetzt.

3.3 Trapezoidal Maps

Analyse

Die Analyse des randomisierten, inkrementellen Algorithmus erfolgt über eine mit Auftrittswahrscheinlichkeit gewichtete Summe des jeweiligen Zeit- und Platzbedarfs. Wir studieren also hier den erwarteten Aufwand des Algorithmus. Eine Analyse des erwarteten worst case Aufwands der Suche findet sich in [de Berg, van Kreveld, Overmars, Schwarzkopf, Computational Geometry, Springer, Berlin, 2000, S. 140 – 143].

Theorem 3.12: Der Algorithmus TrapezoidalMap berechnet die trapezförmige Abbildung $T(S)$ einer planaren Unterteilung S in allgemeiner Lage und eine Suchstruktur für $T(S)$ in $O(n \log n)$ erwarteter Zeit. Die erwartete Größe der Suchstruktur ist $O(n)$ und für jeden Suchpunkt q liegt die erwartete Suchzeit bei $O(\log n)$.

3.3 Trapezoidal Maps

Beweis: Die Korrektheit folgt aus der Invariante.

Wir beginnen mit der Suchzeit. Da der Baum um bis zu 3 Stufen tiefer wird, ergibt sich 3*n* als worst case Aufwand.

Für die erwartete durchschnittliche Zeit sei $X_i, 1 \leq i \leq n,$ die Anzahl der Knoten aus Schritt *i* des Algorithmus entlang des Suchpfades für einen Punkt *q*. Für den Erwartungswert gilt

$$E\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} E\left(X_i\right)$$

Da maximal drei Knoten auf einem Pfad eingefügt werden, gilt $X_i \leq 3.$

Ist P_i die Wahrscheinlichkeit, dass der Pfad in der i-ten Iteration länger wird, so gilt $E(X_i) \leq 3P_i.$

3.3 Trapezoidal Maps

Nun verändert der i-te Schritt den Pfad für q genau dann, wenn das Trapez

$\Delta_q(S_i)$ dasjenige ist, das q in $T(S_i)$ enthält:

$$P_i = P\left[\Delta_q(S_i) \neq \Delta_q(S_{i-1})\right]$$

Die neuen Trapeze in S_i gegenüber S_{i-1} sind alle zu s_i, dem neuen Segment, inzident, also top(Δ), bottom(Δ) gleich s_i oder leftp(Δ), rightp(Δ) ein Endpunkt von s_i.

3.3 Trapezoidal Maps

Sei nun ein festes $S_i \subset S$ gegeben. Dann ist $\Delta_q(S_i)$ ein bestimmtes Trapez unabhängig von der Reihenfolge $s_1, \ldots, s_i$. Wir nutzen eine Rückwärtsbetrachtung, um zu ermitteln, wie wahrscheinlich Δ_q nicht in S_{i-1} ist. $\Delta_q(S_i)$ verschwindet, falls top($\Delta_q(s_i)$), bottom($\Delta_q(s_i)$), leftp($\Delta_q(s_i)$) oder rightp($\Delta_q(s_i)$) mit s_i verschwindet. Mit welcher Wahrscheinlichkeit verschwindet top($\Delta_q(s_i)$)?

Da alle i Segmente $s_1, \ldots, s_i$ gleich wahrscheinlich das letzte sind, ist dies 1/i. Analog bottom($\Delta_q(s_i)$). Ferner ergibt sich für leftp($\Delta_q(s_i)$) ebenfalls maximal 1/i, da der Punkt nur verschwindet, wenn s_i das einzige Segment mit diesem Eckpunkt ist. Analog, 1/i für rightp($\Delta_q(s_i)$).

$$P_i = P\left[\Delta_q(S_i) \neq \Delta_q(S_{i-1})\right] = P\left[\Delta_q(S_i) \notin T(S_{i-1})\right] \leq \frac{4}{i}$$

3.3 Trapezoidal Maps

Insgesamt folgt

$$E\left(\sum_{i=1}^{n} X_i\right) \leq \sum_{i=1}^{n} 3\,P_i \leq \sum_{i=1}^{n} \frac{12}{i} = 12 \sum_{i=1}^{n} \frac{1}{i} = 12H_n$$

Da ln $n < H_n <$ *ln* $n + 1$ gilt, folgt O(log n).

[Diese Schranke ergibt sich an dem Integral $\int_1^n \frac{1}{x} dx = ln(n)$ durch Vergleich mit dem Integral der Treppenfunktion $t: (1, \infty) \to \mathbb{R}, t(x) \to \frac{1}{\lceil x \rceil}$].

3.3 Trapezoidal Maps

Um die Größe von *D* zu bestimmen, müssen wir die Knoten zählen. Die Blätter sind die Trapeze in *T*, also maximal 3*n* +1 . Es gilt

$$(\text{Anzahl } \textit{der Knoten in} D) = 3n + 1 + \sum_{i=1}^{n} (\textit{Anzahl innere Knoten erzeugt in Iteration})$$

Sei k_i die Anzahl neuer Trapeze im Iterationsschritt *i.* Die Anzahl der neuen inneren Knoten ist k_i -1. Als worst case ergibt sich

$$O(n) + \sum_{i=1}^{n} O\,(i) = O(n^2)$$

Für den erwarteten Aufwand gilt jedoch

$$O(n) + E\,(\sum_{i=1}^{n} (k_i - 1))$$

$$\leq O(n) + \sum_{i=1}^{n} E\,(k_i)$$

3.3 Trapezoidal Maps

Die Analyse ähnelt der Frage nach der Anzahl der Suchschritte. Für $T(S_i)$, $s \in S_i$, sei

$$\delta(\Delta, s) := \begin{cases} 1 & falls \Delta aus T(S_i) verschwindet, wenn s entfernt wird \\ 0 & sonst \end{cases}$$

Es gibt maximal 4 Segmente, die Δ verschwinden lassen, also

$$\sum_{s \in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \leq 4 |S_i| = O(i)$$

Da k_i die Anzahl der Segmente ist, die durch s_i entstehen, ist es auch die Anzahl die verschwindet, wenn s_i entfernt wird. Da s_i zufällig ist, kann es auch jedes andere s sein.

3.3 Trapezoidal Maps

Also

$$E(k_i) = \frac{1}{i} \sum_{s \in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \leq \frac{O(i)}{i} = O(1).$$

Es folgt $O(n)$ als erwarteter Speicherbedarf.

3.3 Trapezoidal Maps

Die Laufzeit des Algorithmus ergibt sich nun schnell. Das Einfügen von s_i dauert $O(k_i)$ Schritte und die Zeit zum Finden des linken Endpunktes in $T(S_{i-1})$, also $O(\log (i-1)) = O(\log i)$. Es ergibt sich

$$O(1) + \sum_{i=1}^{n} \left[O(\log i) + O\left(E(k_i)\right) \right] = O(n \log n)$$

QED.

Es ist wichtig zu sehen, dass die Analyse nur die Zufälligkeit im Algorithmus selbst betrachtet, nicht jedoch bei der Eingabe der Suchpunkte.

3.3 Trapezoidal Maps

Um das Ausgangsproblem, nämlich Point Query in einer planaren Unterteilung zu lösen, nutzen wir D um das Trapez Δ zu finden und geben die unterhalb von top(Δ) liegende Facette aus.

Korollar 3.13: Sei S eine planare Unterteilung mit n Kanten. In $O(n \log n)$ erwarteter Zeit, kann man eine Datenstruktur mit $O(n)$ Speicher konstruieren, so dass für jeden Punkt q die erwartete Suchzeit $O(\log n)$ ist.

3.3 Trapezoidal Maps

Degenerierte Fälle

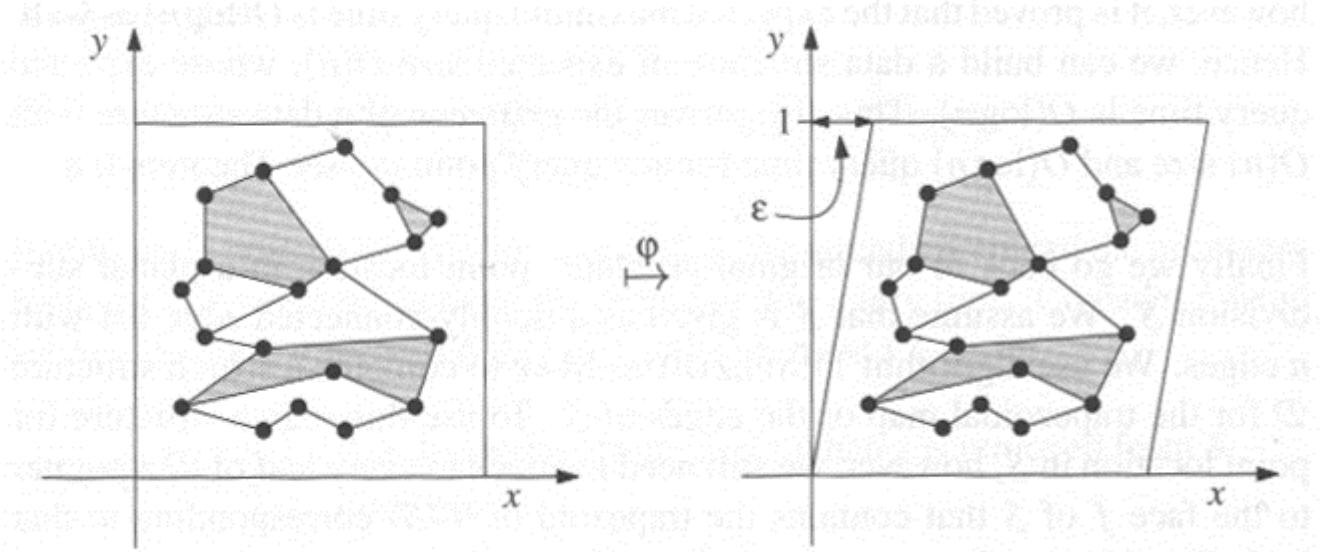
Wir haben bislang vorausgesetzt, dass die Segmente in allgemeiner Lage sind, also zwei verschiedene Endpunkte verschiedene x-Koordinaten haben. Ferner durfte ein gesuchter Punkt kein Endpunkt sein und nicht auf einem Segment liegen.

Die eigentliche Idee zu den Eckpunkten mit verschiedener x-Koordinate ist eine geringfügige Rotation aller Punkte. Da dies numerische Probleme verursacht, werden wir sie nur symbolisch durchführen und geometrisch interpretieren. Statt einer Rotation kann man auch eine Scherung verwenden,

$$\rho: \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} x + \varepsilon y \\ y \end{pmatrix}$$

wobei ein ausreichend kleines ε die Reihenfolge der Punkte nicht ändert.

3.3 Trapezoidal Maps



The shear transformation

3.3 Trapezoidal Maps

Die Berechnung von p(x) wird nie ausgeführt. Bei Berechnungen wird lediglich geprüft, ob sich Veränderungen durch ε ergeben. Da unser Algorithmus keine geometrischen Berechnungen durchführt, sondern nur relative Lagen bestimmt, lässt sich dies einfach bewerkstelligen. Es gibt zwei Operationen:

(a) Liegt q rechts oder links von p?

(b) Liegt q oberhalb oder unterhalb von s (wobei die x-Koordinate von q im Intervall der x-Koordinaten der beiden Endpunkte liegt)?

Zu (a): Die beiden Punkte haben Koordinaten $(x_p + \varepsilon y_p, y_p) und\ (x_q + \varepsilon y_q, y_q).$
Für $x_p \neq x_q$ entscheidet dies den Test wegen des ausreichend klein gewählten Wertes von ε . Ansonsten liegen beide Punkte nicht auf der gleichen Linie.

3.3 Trapezoidal Maps

Zu (b): Wir haben das Segment $\Delta(s)$ mit Endpunkten $(x_1 + \varepsilon y_1, y_1)$, $(x_2 + \varepsilon y_2, y_2)$, und wollen von $\phi(q) = (x + \varepsilon y, y)$ wissen, ob er oberhalb oder unterhalb von s liegt. Dabei gilt stets $x_1 + \varepsilon y_1 \leq x + \varepsilon y \leq x_2 + \varepsilon y_2$ und somit $x_1 \leq x \leq x_2$. Ferner gilt $x = x_1 \Rightarrow y <= y_2$ *und* $x = x_2 \Rightarrow y <= y_2$. Für $x_1 = x_2$ haben wir ein vertikales Segment und der Algorithmus liefert $x_1 = x = x_2$, $y_1 <= y <= y_2$, also $q \in s$ und wegen der Scherung $\varphi(q) \in \varphi(s)$.

Für $x_1 < x_2$ nutzen wir, dass die Scherung die Relation zwischen Punkten und Linien erhält, also für einen Punkt oberhalb *s* (bzw. der Geraden durch *s*) das Bild auch oberhalb von □(*s*) liegt.

Wir brauchen also die Punkte lediglich lexikographisch ordnen, um unsere symbolische Behandlung des Problems zu erreichen.

3.3 Trapezoidal Maps

Dabei lösen wir auch das Problem mit den Punkten auf einem Segment oder Eckpunkt beim Suchen. Bei *x*-Knoten kann der Punkt symbolisch auf der Linie liegen, falls es der Endpunkt ist, wodurch die Point Query schnell zu beantworten ist. Bei *y*-Knoten ergibt sich wieder, dass der Punkt auf dem Segment liegt. Auch dies kann man als Resultat zurückgeben.

Theorem 3.14: Der Algorithmus TRAPEZOIDALMAP berechnet die trapezförmige Karte $T(S)$ einer planaren Überlagerung S und eine Suchstruktur D für $T(S)$ in $O(n \log n)$ erwarteter Zeit. Die erwartete Größe der Suchstruktur ist $O(n)$ und für jeden Suchpunkt ist die erwartete Suchzeit $O(\log n)$.

3.3 Trapezoidal Maps

Literatur

Zur Punktsuche findet man eine ordentliche Übersicht bei Preparata und Shamos [F. P. Preparata and M. I. Shamos. Computational Geometry: An Introduction. Springer-Verlag. 1985] und für out-of-core Methoden mit weiteren Suchen bei Gaede und Günther [Gaede und Günther. Multidimensional Access, Methods, TU-Berlin]. Es gibt vier verschiedene Lösungsansätze mit $O(\log n)$ Suchzeit und $O(n)$ Speicher. Die chain method von Edelsbrunner et al. [H. Edelbrunner. L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. SIAM J. Comput. 15:317-340,1986]; die triangulation refinement method von Kirkpatrick [D. G. Kirkpatrick, Optimal search in planar subdivisions. SIAM J. Comput. 12:28-35,1983], die persistency method von Sarnack und Tarjan [N. Sarnak, and R. E. Tarjan. Planar point location using persistent search trees. Commun. ACM, 29:669-679, 1986] und die randomisierte, inkrementelle Methode von Mulmuley [K. Mulmuley. A fast planar partition algorithm, I. Journal of Symbolic Computation, 10.253-280, 1990]. Letztere diente mit der Analyse von Seidel [R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Comput. Geom. Theory Appl., 1:51-64, 1991] als Grundlage dieses Abschnittes.

3.3 Trapezoidal Maps

Das dynamische Problem mit Einfügen und Löschen von Segmenten stellt eine zusätzliche Herausforderung dar. Chiang und Tamassia [Y.-J. Chiang and R. Tamassia. Dynamic algoirthms in computational geometry. Proc. IEEE, 80:1412-1434, September 1992] geben einen Überblick.

3.3 Trapezoidal Maps

Leider ist das Suchen in Unterteilungen höherer Dimension bislang nicht vollständig gelöst. Einen Ansatz für konvexe Unterteilungen geben Preparata und Tamassia [H. Samet Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley. Reading. MA, 1990]. Für Hyperebenen gibt es einige Ansätze, etwa [B. Chazelle. Cutting hyperplanes for divide-and conquer. Discrete Comput. Geom., 9:145-158, 1993]. Eine effiziente Punktsuche gestatten konvexe Polytope [K. L. Clarkson. New applications of random sampling in computational geometry. Discete Comput. Geom., 2:195-222, 1987, J. Matousek and O. Schwarzkopf. On ray shooting in convex polytopes. Discrete Comput. Geom., 10:215-232, 1993] und algebraische Varietäten [B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. Theoret. Comput. Sci., 84:77-105, 1991]. Hier kann man auch nach möglichen, effizienten Strukturen für die Visualisierung großer, unstrukturierter Datenmengen suchen.

3.4 Weitere Ansätze

3.4.1. Bit Interleaving

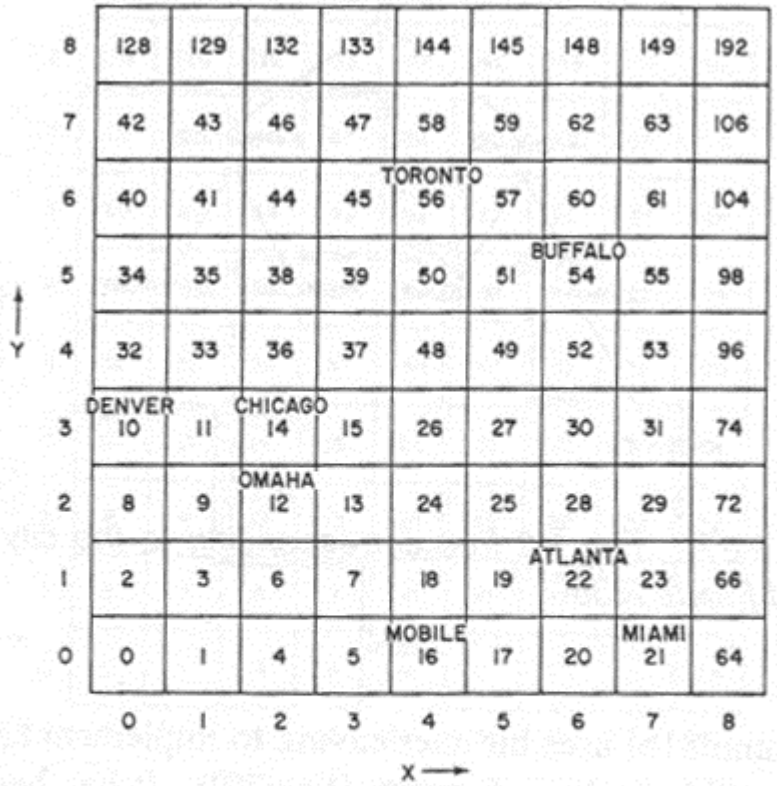
Bei den bisher besprochenen Verfahren ist es leider nicht ohne weiteres möglich, die Suchbäume zu balancieren. Man ist früh auf die Idee gekommen, höher dimensionale Punkte (Schlüssel im Datenbankbereich) auf eindimensionale Punkte abzubilden, um etwa die bekannten AVL-Bäume nutzen zu können. Ein solcher Vorschlag ist bit interleaving. Aus einem Punkt

$$(x, y) = (x_m\, x_{m-1} \ldots x_0, y_m\, y_{m-1} \ldots y_0)$$

wird der Punkt

$$\bar{x} = y_m x_m y_{m-1} x_{m-1} \ldots y_0 x_0$$

3.4 Weitere Ansätze



Example of the bit interleaving mapping as applied to two keys ranging in values from 0 to 8. The city names correspond to the data of Figure 2.1 scaled by a factor of 12.5 to fall in this range.

3.4 Weitere Ansätze

Die Punktsuche entspricht dann der Suche in einem binären Suchbaum. Die Bereichssuche (range query) erfolgt nach dem zu Beginn von 3.2 beschriebenen Konzept.

Bit Interleaving gestattet $O(\log n)$ Einfügen, Löschen und Suchen. Allerdings ist das Interleaving verhältnismäßig aufwändig zu berechnen. Ferner ist Bit Interleaving vor allem für diskrete Daten geeignet. Man sollte lediglich die Idee im Kopf behalten, auch für out-of-core Methoden.

3.4 Weitere Ansätze

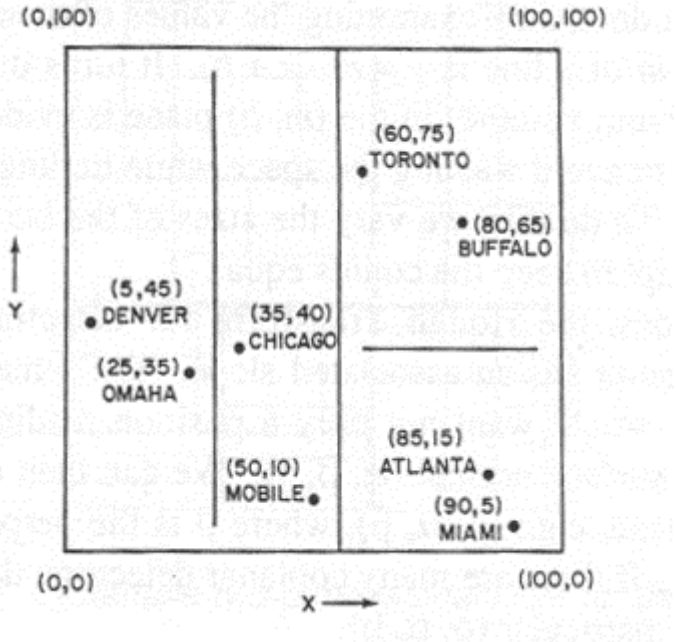
3.4.2. Bucket Methods

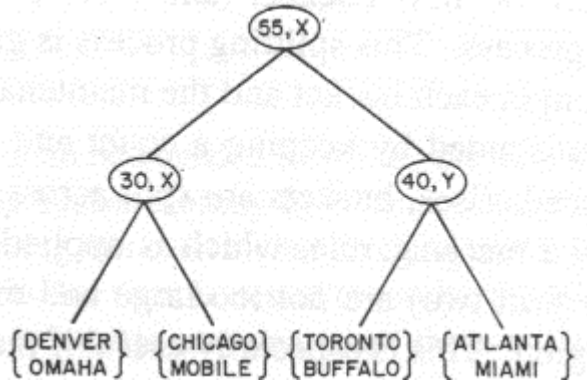
Unsere bisherigen Datenstrukturen in diesem Kapitel basieren auf Bäumen. Dies kann zu ineffizienter Struktur führen, wenn sekundärer Speicher benutzt werden muss. Um in diesem Fall effizient zu sein, sind die Punkte, Segmente oder Zellen in Mengen (buckets) zu organisieren, die einzelnen Seiten auf der Platte (disk pages) entsprechen. Der Zugriff auf die Seiten wird durch ein Verzeichnis (directory) organisiert. Solche Strukturen heißen **bucket methods**. Die Verzeichnisse können hierarchisch (etwa B-Bäume) oder nicht hierarchisch (etwa Gridfile) organisiert sein.

3.4 Weitere Ansätze

Ein Beispiel bildet ein kd-Baum mit Buckets, wobei auch die Richtung des Unterteilens flexibel gehandhabt werden kann.

[Mats84: T. Matsuyama, L. V. Hao, and M. Nagao, A file organization for geographic information systems based on spatial proximity, Computer Vision, Graphics, and Image Processing 26,3(June 1984). 303-318. [points] D.2.8.1 D.3]





Bucket adaptive kd tree corresponding to bucket size =2

3.4 Weitere Ansätze

Wenn das Verzeichnis nicht hierarchisch organisiert ist, kommt ein Feld (array) zur Anwendung. Eine Möglichkeit sind Hashing-Ansätze [Knuth73b: D. E.Knuth, The Art of Computer Programming, vol. 3, Sorting and Searching, Addison-Weslex, Reading, MA., 1973. [general] D.P D.1.2 D.2.1 D.2.2 D.2.3.1 D.2.3.3 D.2.4.1 D.2.4.3 D.2.5 D.2.6.2 D.2.8.2.1 D.2.8.2.2 A.P A.1.3], etwa linear hashing [Litw80: W. LiItwin, Linear hashing: a new tool for file and table addressing, Priceedings of the Sixth International conference on Very Large Data Bases, Montreal, October 1980, 212-223. [point] D.2.7 D.2.8.2.1], [Lars88: P. A. Larson, Dynamic hash tables,Communications of the ACCM 31, 4(April 1988], 446-457. [points]D.2.8.2.1 D.2.8.2.2] oder spiral hashing [Mart79: G. N. N. Martin, Spiral storage: incrementally augmenetable hash addressed storage, Theory of Computation Report No. 27, Department of Computer Science, Univsersity of Warwick, Covernty, Great Britain, March 1979. [points] D.2.8.2.2], [Mull85 J. K. Mullin, Spiral storage: efficient dynamic hashing with constant performance, computer Jouornal 28, 3(August 19985), 330-334. [points] D.2.8.2.2]