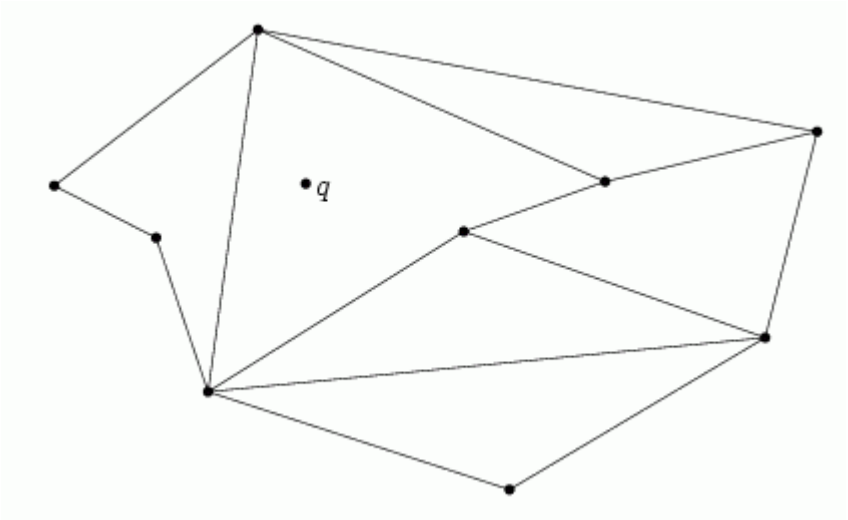
7. Punktlokalisierung: Wo bin ich?

- Punktlokalisierungsproblem:
  Gegeben eine Karte und ein Anfragepunkt q, der durch seine Koordinaten spezifiziert wird, bestimme die Region der Karte, in der sich q befindet.

- Gesucht:
  Datenstruktur, die schnelle Punktlokalisierungsanfragen unterstützt.



thematische Karten (Beisp.: Bodennutzung, Geologie...) als Beispiel
– Unterteilung der Ebene in Regionen, denen Attribute zugeordnet sind

Präzisierungen des Begriffs "Karte":

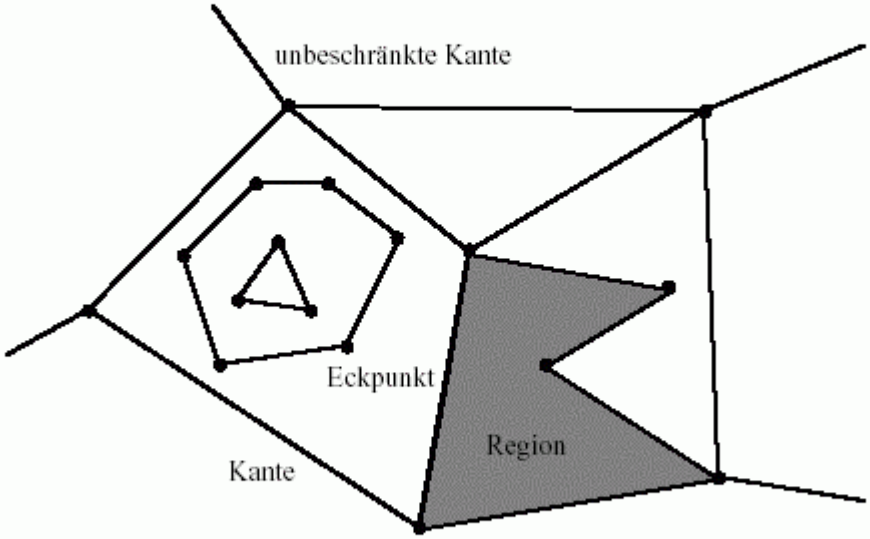
- *planare Unterteilung*: Zerlegung der Ebene in (beschränkte oder unbeschränkte) Regionen
  – Regionen sind durch Ecken und Strecken bzw. Halbgeraden begrenzt
  – Regionen sind zusammenhängend
  – das Innere zweier verschiedener Regionen ist disjunkt

- *planare Einbettung eines endlichen Graphen*:
  – nur (beschränkte) Strecken als Kanten zugelassen
  – nur 1 unbeschränkte Region (Außengebiet)

wir betrachten im Folgenden planare Einbettungen
(keine wesentl. Einschränkung!)

Präzisierung "Region" (Hinrichs 2001):

Eine *Region* einer planaren Unterteilung ist eine maximale zusammenhängende Teilmenge der Ebene, die keinen Eckpunkt und keinen Punkt einer Kante enthält ⇒ Region ist offenes Polygon, begrenzt durch eine alternierende Folge von Eckpunkten und Kanten der planaren Unterteilung:



- Die *Komplexität* einer planaren Unterteilung ist definiert als die Summe der Anzahl Knoten, der Anzahl Kanten und der Anzahl Regionen, aus denen sie besteht.

– Wieviel Platz braucht man zum Abspeichern einer planaren Einbettung eines Graphen mit v Knoten?
– Wie hängen der Anzahlen der Knoten (*vertices*), Kanten (*edges*) und Regionen (*faces*) zusammen?

Für Graphen allgemein: Kantenzahl = $O(v^2)$
jedoch für planare Graphen sind Ecken- und Regionenzahl linear in v. Das folgt aus dem folgenden klassischen Satz.

Eulerscher Polyedersatz (Euler 1758):
Sei E eine planare Einbettung eines Graphen in die Ebene mit v Knoten, e Kanten und f Regionen (einschließlich der einzigen unbeschränkten Region), und sei c die Anzahl der Zusammenhangskomponenten von E. Dann gilt:

$$v - e + f = c + 1.$$

Beweis:
Induktion über die Kantenzahl e.
Induktionsanfang: $e = 0$, d.h. E hat keine Kanten $\Rightarrow$ E ist Ansammlung von v Punkten in der Ebene $\Rightarrow$ $f = 1$ und $c = v$
$\Rightarrow$ es gilt $v - e + f = c + 1$.

Sei $e > 0$; der Satz gelte für alle Einbettungen E' von Graphen mit denselben Knoten wie E, aber mit weniger als e Kanten.
Seien p, q zwei Knoten und (p, q) eine Kante in E.
Entferne Kante (p, q) aus $E \Rightarrow$ es entsteht E'.

Fall 1: E und E' haben dieselbe Zahl von Zusammenhangskomponenten
$\Rightarrow$ Entfernen der Kante erzeugt keine neue Komponente
$\Rightarrow$ 2 Regionen von E werden zu einer Region von E' verschmolzen
$\Rightarrow$ E' hat v Knoten, $e-1$ Kanten, $f-1$ Regionen und c Zusammenhangskomponenten
$\Rightarrow$ nach Ind.vorauss.: $v - (e-1) + (f-1) = c + 1$, es folgt die Beh.

Fall 2: E und E' haben nicht dieselbe Zahl von Zus.h.komp.
$\Rightarrow$ Entfernen der Kante (p, q) teilt eine Komponente von E auf in 2 Komponenten von E'
$\Rightarrow$ E' hat v Knoten, $e-1$ Kanten, f Regionen und $c+1$ Zusammenhangskomponenten
$\Rightarrow$ nach Ind.vorauss.: $v - (e-1) + f = (c+1) + 1$, es folgt die Beh.

Folgerung:
Sei E eine planare Einbettung eines zusammenhängenden Graphen mit $n \geq 3$ Knoten. Dann gilt: E hat
(a) höchstens $3n - 6$ Kanten, (b) höchstens $2n - 4$ Regionen.

Beweis:
Für $n = 3$ klar (denn dann ist $e \leq 3$ und $f \leq 2$).
Sei $n \geq 4$. Sei m_i die Kantenzahl der i-ten Region von E und
$M = \sum m_i$ die Summe all dieser Zahlen über alle Regionen.
Jede Kante gehört zu höchstens 2 Regionen $\Rightarrow$ $M \leq 2e$
Jede Region hat mindestens 3 Kanten $\Rightarrow$ $M \geq 3f$.

Es folgt: $f \leq 2e/3$.
Wegen $c = 1$ folgt aus der Euler-Formel:
$e = n + f - 2 \leq n + 2e/3 - 2$
$\Rightarrow e \leq 3n - 6$
$\Rightarrow f = e - n + 2 \leq (3n - 6) - n + 2 = 2n - 4$.

Übungsaufgabe:
(a) Man zeige, dass der vollständige Graph K_5 nicht planar ist.
(b) *(etwas schwieriger)* Man zeige, dass der vollständige bipartite Graph $K_{3,3}$ nicht planar ist. (Hinweis: Bipartite Graphen enthalten keine Dreiecke.)

Damit klar:
für planare Einbettungen sind e und f in $O(v)$.

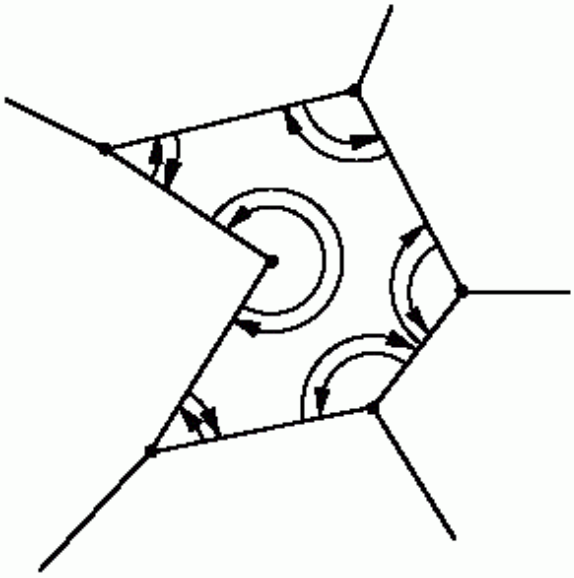
Wie lassen sich planare Einbettungen von Graphen überhaupt abspeichern?

Datenstruktur "Doubly connected edge list" (*DCEL*)

- jede Ecke, jede Kante und jede Region wird repräsentiert
- für jedes dieser Elemente:
  – geometrische Information
  – topologische Information (Inzidenzen, Adjazenzen)
  – Attributinformation

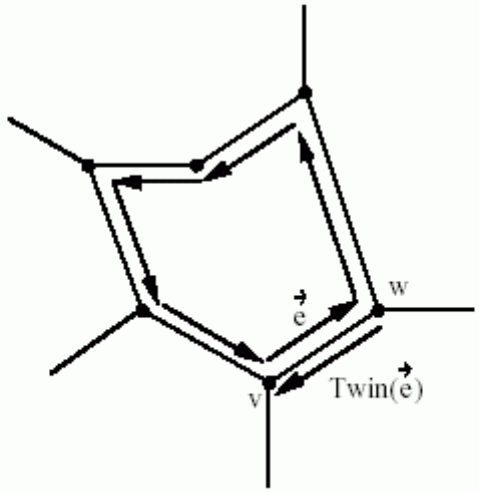
Ziel: bestimmte Basisoperationen sollen durchführbar sein
- Umlaufen des Randes einer Region
- Besuch aller Kanten, die mit einem Knoten inzidieren
- Zugriff auf eine Nachbarregion
- ...

Um eine Region entgegen dem Uhrzeigersinn zu umlaufen, speichern wir einen Zeiger von einer Kante zur nachfolgenden Kante. Um eine Region im Uhrzeigersinn zu umlaufen, benötigen wir einen Zeiger auf die vorhergehende Kante:

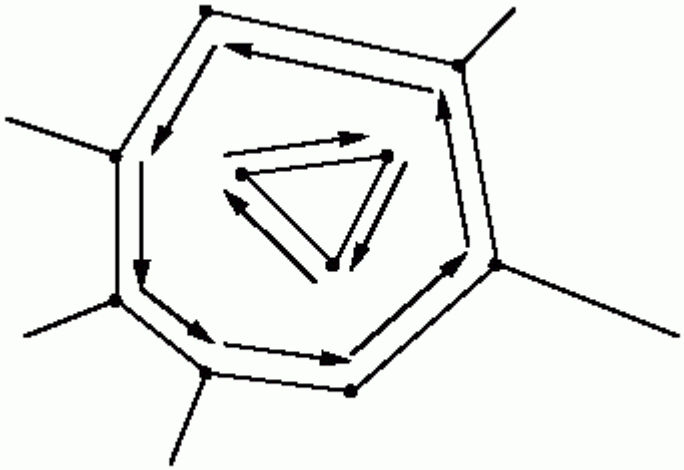


- Kante begrenzt gewöhnlich zwei Regionen $\Rightarrow$ zwei Zeigerpaare für jede Kante
- Betrachte die zwei Seiten einer Kante als zwei verschiedene *Halbkanten* $\Rightarrow$ für jede Halbkante gibt es eine eindeutige vorhergehende und eine eindeutige nachfolgende Halbkante $\Rightarrow$ eine Halbkante begrenzt nur eine Region.
- Die einer Kante zugeordneten zwei Halbkanten bezeichnet man auch als *Zwillinge*.
- Legt man den Nachfolger einer gegebenen Halbkante bzgl. einer Traversierung des Randes einer Region im Gegenuhrzeigersinn fest, so induziert dies eine *Orientierung* für jede Halbkante:
  Eine Halbkante ist so orientiert, daß die Region, die von ihr begrenzt wird, zur Linken liegt, wenn ein Beobachter entlang der Halbkante geht. Da Halbkanten orientiert sind, sprechen wir auch von *Anfangs-* und *Endpunkt* einer Halbkante.
- Besitzt eine Halbkante $\vec{e}$ den Knoten v als Anfangspunkt und den Knoten w als Endpunkt, so besitzt der Zwilling *Twin*($\vec{e}$) w als Anfangspunkt und v als Endpunkt.

Um von einer Region aus ihren Rand zu erreichen, brauchen wir in dem zugehörigen Record der Region nur einen Zeiger auf eine beliebige Halbkante zu speichern. Ausgehend von dieser Halbkante können wir von jeder Halbkante zur nachfolgenden Halbkante gehen und so die Region umwandern:



- Problem: Rand eines Loches in einer Region:



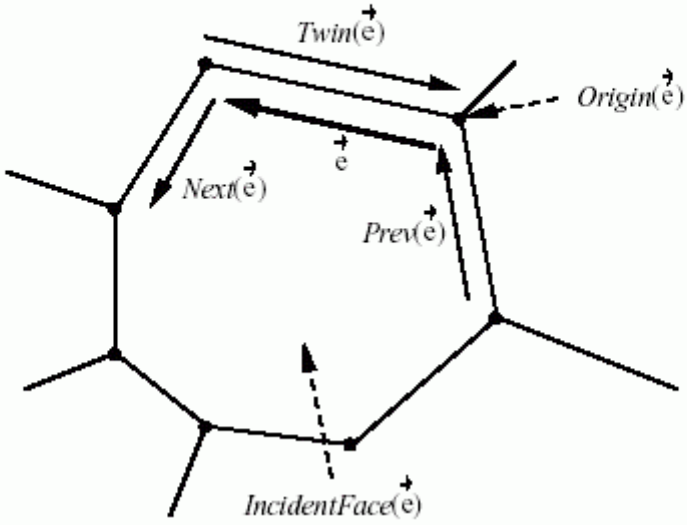
- Bei Traversierung des Loches entgegen dem Uhrzeigersinn liegt die Region zur Rechten.

- Orientiere die Halbkanten so, daß die Region immer auf der gleichen Seite, z.B. zur Linken, liegt
  ⇒ Rand eines Loches wird immer im Uhrzeigersinn traversiert
  ⇒ Region liegt immer zur Linken einer Halbkante auf ihrem Rand
  ⇒ Zwillingshalbkanten immer entgegengesetzt orientiert
- Region hat ein oder mehrere Löcher ⇒ ein Zeiger von der Region auf eine beliebige Halbkante ihres Randes reicht nicht mehr aus, um den gesamten Rand zu durchlaufen:
  Man benötigt einen Zeiger auf eine Halbkante in jeder Rand-komponente.
  Besitzt eine Region isolierte Eckpunkte, die keine inzidenten Kanten besitzen, kann man auch Zeiger auf diese speichern.

Zusammenfassung: DCEL besteht aus drei Mengen von Records: einer für die Knoten, einer für die Kanten und einer für die Regionen. Diese Records speichern die folgenden geometrischen und topologischen Informationen:

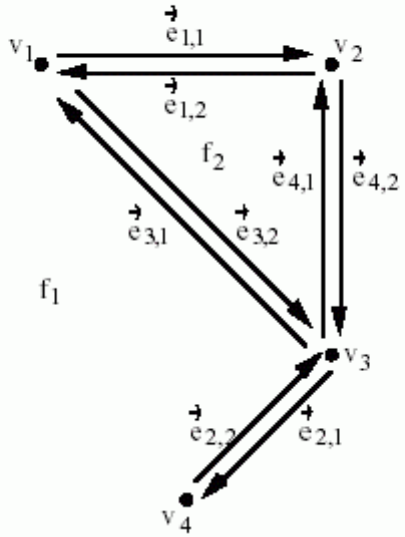
- *Vertex*-Record eines Knotens v:
  Koordinaten von v in einem Feld, genannt *Coordinates*(v)
  Zeiger *IncidentEdge*(v) zu einer beliebigen Halbkante, die v als Anfangspunkt besitzt
- *Face*-Record einer Region f:
  Zeiger *OuterComponent*(f) zu einer Halbkante auf ihrem äußeren Rand. Für die unbeschränkte (äußere) Region ist dieser Zeiger *nil*.
  Liste *InnerComponents*(f) enthält für jedes Loch in der Region einen Zeiger auf eine Halbkante auf dem Rand dieses Loches

– *Half-edge*-Record einer Halbkante $\vec{e}$:
  Zeiger *Origin*($\vec{e}$) auf seinen Anfangspunkt
  Zeiger *Twin*($\vec{e}$) auf seine Zwillingshalbkante
  $\Rightarrow$ Endpunkt von $\vec{e}$: *Origin*(*Twin*($\vec{e}$))
  Zeiger *IncidentFace*($\vec{e}$) auf die Region, in deren Rand sie enthalten ist
  Anfangspunkt *Origin*($\vec{e}$) wird so gewählt, daß *IncidentFace*($\vec{e}$) zur Linken von $\vec{e}$ liegt, wenn $\vec{e}$ vom Anfangs- zum Endpunkt durchlaufen wird.
  Zeiger *Next*($\vec{e}$) und *Prev*($\vec{e}$) auf die nachfolgende und vorhergehende Halbkante auf dem Rand von *IncidentFace*($\vec{e}$).
  $\Rightarrow$ *Next*($\vec{e}$) ist die eindeutige Halbkante auf dem Rand von *IncidentFace*($\vec{e}$), die den Endpunkt von $\vec{e}$ als Anfangspunkt besitzt, und *Prev*($\vec{e}$) ist die eindeutige Halbkante auf dem Rand von *IncidentFace*($\vec{e}$), die *Origin*($\vec{e}$) als Endpunkt besitzt.



Für jeden Knoten und jede Kante wird konstanter Speicherplatz benötigt.
Eine Region kann mehr Speicherplatz erfordern, da die Anzahl der Zeiger in der Liste *InnerComponents*(f) gleich der Anzahl Löcher der Region f ist.
Auf jede Halbkante wird höchstens einmal von allen *InnerComponents*(f) Listen zusammen verwiesen $\Rightarrow$ der benötigte Speicherplatz ist linear in der Komplexität der planaren Unterteilung.

Beispiel: die zu einer Kante e_i gehörigen Halbkanten sind mit $\vec{e}_{i,1}$ und $\vec{e}_{i,2}$ bezeichnet:



Vertex	*Coordinates*	*IncidentEdge*
v_1	(0, 4)	$\vec{e}_{1,1}$
v_2	(2, 4)	$\vec{e}_{4,2}$
v_3	(2, 2)	$\vec{e}_{2,1}$
v_4	(1, 1)	$\vec{e}_{2,2}$

Face	*OuterComponent*	*InnerComponents*
f_1	nil	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	nil

Half-edge	*Origin*	*Twin*	*IncidentFace*	*Next*	*Prev*
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_4	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

- In der DCEL gespeicherte Information reicht aus, um die Basisoperationen durchzuführen:
  - Durchlaufen der äußeren Berandung einer Region f, indem man von der Halbkante *OuterComponent*(f) ausgehend den *Next*($\vec{e}$) Zeigern folgt
  - Besuch aller zu einem Knoten inzidenten Kanten

(aus Hinrichs 2001)

Anwendung für Punktlokalisierungs-Anfrage:

- Durchlaufen aller Regionen
- prüfe für jede Region, ob der Punkt *q* enthalten ist (Strahltest)

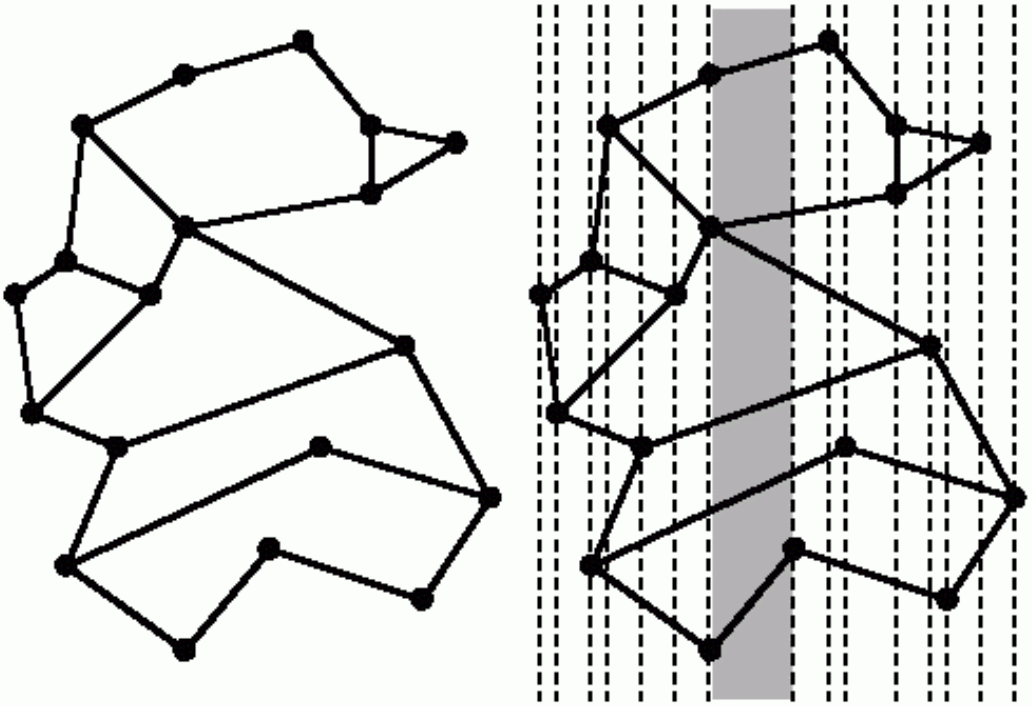
$\Rightarrow$ Anzahl Schnittpunktberechnungen: für alle Kanten aller Regionen, also $O(n)$ (n = Knotenzahl)

das ist als Anfragezeit bei großen Strukturen problematisch

Idee: die Regionen in kleinere Regionen einteilen, wo Zugehörigkeit schneller geprüft werden kann

$\Rightarrow$ *slab method* (Streifenmethode)

- Ziehe durch alle Eckpunkte der planaren Unterteilung vertikale Linien, die die Ebene in vertikale Streifen unterteilen:



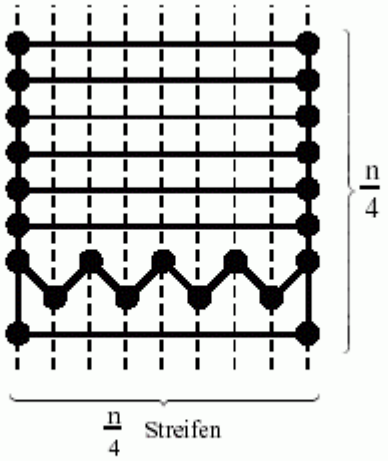
- Speichere die x-Koordinaten der Eckpunkte in sortierter Reihenfolge in einem Array
  ⇒ Bestimmung des Streifens, der den Anfragepunkt enthält, durch binäres Suchen in O(log n) Zeit.
- Im Innern eines Streifens gibt es keine Eckpunkte von S
  ⇒ alle Kanten, die einen Streifen schneiden, kreuzen ihn vollständig, d.h. sie haben keinen Endpunkt im Innern des Streifens, und sie schneiden einander nicht
  ⇒ diese Kanten können von oben nach unten total geordnet werden
- Jedes Flächenstück innerhalb des Streifens, das durch zwei Kanten eingeschlossen wird, gehört zu genau einer Region von S.

- Oberstes und unterstes Flächenstück eines Streifens sind unbeschränkt, sie gehören zur unbeschränkten äußeren Region von S.
- Kanten innerhalb eines Streifens können in sortierter Reihenfolge in einem Array gespeichert werden. Markiere jede Kante mit der Region von S, die unmittelbar oberhalb der Kante liegt.

- Anfragealgorithmus:
  - Binäre Suche mit der x-Koordinate des Anfragepunktes q in dem Array, das die x-Koordinaten der Eckpunkte von S in sortierter Reihenfolge speichert ⇒ Streifen, der q enthält
  - Weitere binäre Suche mit q in dem Array, das diesem Streifen zugeordnet ist ⇒ direkt unterhalb von q liegendes Segment, dessen Markierung die Region von S liefert, in der q enthalten ist. Falls es kein Segment unterhalb von q gibt, so liegt q in der unbeschränkten, äußeren Region von S.

- O(log n) Zeit wird benötigt, denn:
  binäre Suche in einem Array mit höchstens 2·n Elementen
  sowie in einem Array mit höchstens n Elementen
- $O(n^2)$ Speicherplatzbedarf, denn:
  Array, das x-Koordinaten der 2·n Endpunkte speichert: O(n)
  Array für kreuzende Kanten eines Streifens: O(n)
  O(n) Streifen ⇒ gesamter Speicherplatzbedarf im schlimmsten Fall $O(n^2)$

- Schlimmster Fall kann eintreten:



- $O(n^2)$ Speicherplatzbedarf
  ⇒ Algorithmus uninteressant, selbst für moderate Werte von n

Verbesserung: Trapezoidzerlegungsverfahren

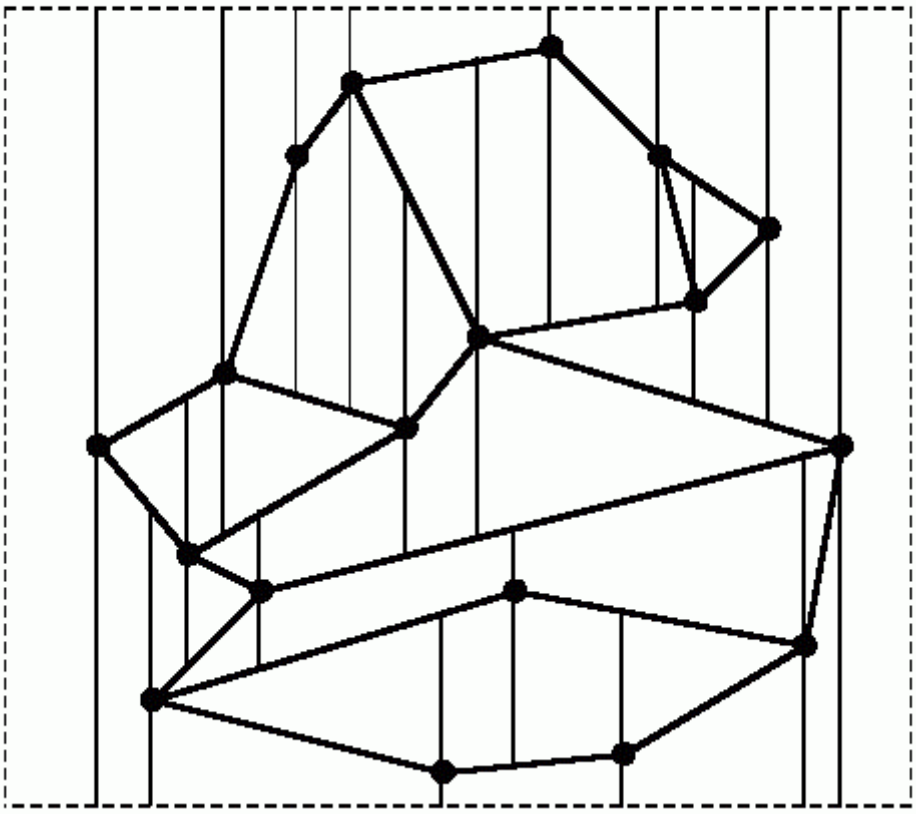
- Wodurch wird das quadratische Verhalten bedingt?
- Die Segmente von S und die vertikalen Linien durch die Eckpunkte von S definieren eine neue planare Unterteilung S', deren Regionen Trapeze, Dreiecke und unbeschränkte trapezförmige Gebilde sind.
- S' ist eine Verfeinerung der ursprünglichen Unterteilung S, jede Region von S' liegt vollständig in einer Region von S
  ⇒ Algorithmus führt eine Punktlokalisierung in dieser verfeinerten Unterteilung durch
- Mit der Region f' in S', die q enthält, kennen wir auch die q enthaltende Region f in S.
- Verfeinerte Unterteilung hat quadratische Komplexität
  ⇒ Datenstruktur hat quadratischen Speicherplatzbedarf

- Idee:
  Finde eine andere Verfeinerung von S, die wie S' effiziente Punktlokalisierung ermöglicht, aber im Gegensatz zu S' eine Speicherplatzkomplexität besitzt, die nicht wesentlich über der der gegebenen Unterteilung S liegt.
- *Trapezoidzerlegung* erfüllt diese gewünschten Eigenschaften
- Zwei Liniensegmente in der Ebene heißen*nicht-kreuzend*, falls ihr Schnitt entweder leer oder nur aus einem gemeinsamen Endpunkt besteht
  ⇒ Kanten einer planaren Unterteilung sind nicht-kreuzend.

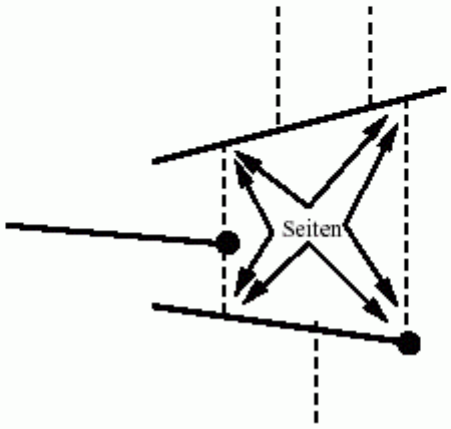
- S Menge von n nicht-kreuzenden Liniensegmenten in der Ebene
- 1. Vereinfachung:
  Achsenparalleles Rechteck R, das alle Segmente enthält
  $\Rightarrow$ keine unbeschränkten Regionen am Rand der Szene
  Anfragepunkt q, der außerhalb R liegt, ist in der unbeschränkten äußeren Region von S enthalten
- 2. Vereinfachung:
  Keine zwei Segmentendpunkte von S besitzen gleiche x-Koordinate $\Rightarrow$ es gibt insbesondere keine vertikalen Segmente
- S Menge von n nicht-kreuzenden Liniensegmenten in der Ebene, die in einem achsenparallelen Rechteck R enthalten sind, mit der Eigenschaft, daß keine zwei verschiedenen Endpunkte auf einer Vertikalen liegen
  $\Rightarrow$ *Menge von Segmenten in allgemeiner Lage*

- *Trapezoidzerlegung* T(S) von S, auch vertikale Zerlegung genannt, erhält man, indem man von jedem Endpunkt p eines Segments zwei *vertikale Erweiterungen* zeichnet, eine nach oben und eine nach unten. Diese Erweiterungen enden, sobald sie ein anderes Segment oder den Rand von R treffen.
  $\Rightarrow$ *obere vertikale Erweiterung* bzw. *untere vertikale Erweiterung* von p

- Trapezoidzerlegung von S ist die planare Unterteilung, die durch S, R und die unteren und oberen vertikalen Erweiterungen in allen Segmentendpunkten induziert wird:



- Region in T(S) wird begrenzt durch eine Anzahl von Kanten von T(S). Einige dieser Kanten können benachbart und kollinear sein. Wir nennen die Vereinigung solcher Kanten eine *Seite* einer Region. Die Seiten einer Region sind also die Segmente maximaler Länge, die im Rand einer Region enthalten sind:



Hilfssatz 1:

Jede Region in einer Trapezoidzerlegung einer Menge S von Liniensegmenten in allgemeiner Lage hat ein oder zwei vertikale Seiten und exakt zwei nicht-vertikale Seiten.

Beweis:

f Region in T(S) $\Rightarrow$ f konvex, denn:

Segmente in S nicht-kreuzend $\Rightarrow$ jeder Eckpunkt von f ist

- entweder ein Endpunkt eines Segmentes in S oder
- ein Punkt, in dem eine vertikale Erweiterung auf ein Segment von S oder auf eine Kante von R trifft,
- oder ein Eckpunkt von R.

vertikale Erweiterungen $\Rightarrow$ kein Eckpunkt, der zugleich Endpunkt eines Segmentes ist, kann einen inneren Winkel > π besitzen.

Jeder Winkel an einem Punkt, an dem eine Vertikale ein Segment trifft, $\leq \pi$

Winkel in den Ecken von R = $\pi/2$

$\Rightarrow$ f konvex, vertikale Erweiterungen haben alle Nicht-Konvexitäten beseitigt.

f konvex $\Rightarrow$ f hat höchstens zwei vertikale Seiten

Annahme: f besitzt mehr als zwei nicht-vertikale Seiten

$\Rightarrow$ es gibt zwei solche Seiten, die benachbart sind und f entweder von oben oder von unten begrenzen

Jede nicht-vertikale Seite ist in einem Segment von S oder in einer Kante von R enthalten und die Segmente sind nicht-kreuzend $\Rightarrow$ die zwei benachbarten Seiten haben einen Endpunkt gemeinsam

die vertikalen Erweiterungen für diesen Endpunkt verhindern, daß diese beiden Seiten benachbart sind, ein Widerspruch!

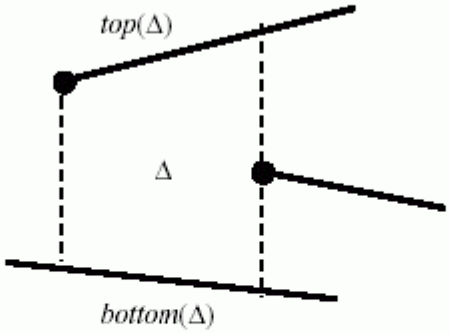
⇒ f hat höchstens zwei nicht-vertikale Seiten

f beschränkt, da gesamte Szene in einem Rechteck eingeschlossen ⇒ f kann nicht weniger als zwei nicht-vertikale Seiten haben, und es besitzt mindestens eine vertikale Seite. ■

Aus Hilfssatz 1 folgt:

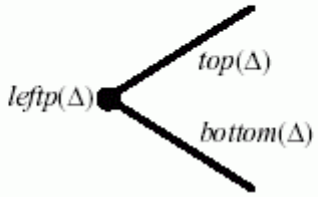
Jede Region einer Trapezoidzerlegung ist entweder ein Trapez oder ein Dreieck, das wir als ein Trapez mit einer degenerierten Kante der Länge 0 ansehen können.

- *top*(Δ) bzw. *bottom*(Δ):
  nicht-vertikales Segment von S, oder die Kante von R, die ein Trapez Δ von oben bzw. unten begrenzt

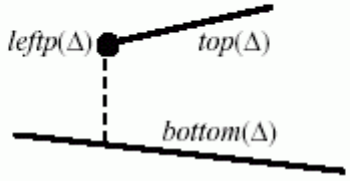


- Annahme allgemeiner Lage ⇒ vertikale Seite eines Trapezes besteht entweder aus vertikalen Erweiterungen, oder es ist die vertikale Kante von R.
- 5 Fälle für die linke Seite
- 5 Fälle für die rechte Seite sind symmetrisch

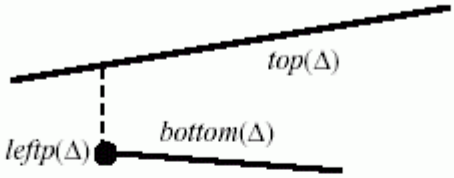
(1) Die linke Seite degeneriert zu einem Punkt, der der gemeinsame linke Endpunkt von *top*(Δ) und *bottom*(Δ) ist:



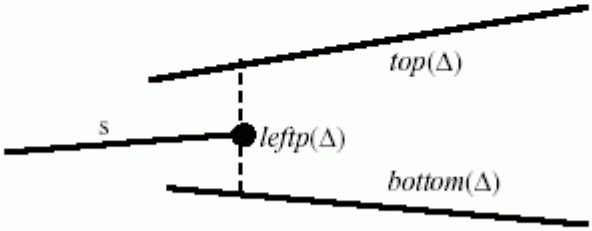
(2) Die linke Seite ist die untere vertikale Erweiterung des linken Endpunkts von *top*(Δ), die auf *bottom*(Δ) endet:



(3) Die linke Seite ist die obere vertikale Erweiterung des linken Endpunkts von *bottom*(Δ), die auf *top*(Δ) endet:



(4) Die linke Seite besteht aus der oberen und unteren Erweiterung des rechten Endpunkts p eines dritten Segments s. Diese Erweiterungen enden auf *top*(Δ) bzw. *bottom*(Δ):



(5) Die linke Seite ist die linke Kante von R. Dieser Fall trifft für ein einziges Trapez von T(S) zu, nämlich das am weitesten links liegende Trapez von T(S).

Außer im Fall (5) gilt:

- Für jedes Trapez Δ von T(S) wird die linke vertikale Seite von Δ durch einen Endpunkt p eines Segments festgelegt: sie ist entweder in den vertikalen Erweiterungen von p enthalten, oder sie ist, falls sie degeneriert ist, der Punkt p selbst.
- *leftp*(Δ): Endpunkt, der die linke Seite von Δ definiert *leftp*(Δ) ist der linke Endpunkt von *top*(Δ) oder *bottom*(Δ), oder es ist der rechte Endpunkt eines dritten Segments
- *rightp*(Δ): Endpunkt, der die rechte vertikale Seite von Δ festgelegt
- Trapez Δ ist eindeutig bestimmt durch *top*(Δ), *bottom*(Δ), *leftp*(Δ) und *rightp*(Δ)

im Fall (5): *leftp*(Δ) = linke untere Ecke von R.

Hilfssatz 2:

Die Trapezoidzerlegung T(S) einer Menge S von n Liniensegmenten in allgemeiner Lage enthält höchstens $6 \cdot n + 4$ Eckpunkte und höchstens $3 \cdot n + 1$ Trapeze.

Beweis:

Eckpunkt von T(S) ist

- entweder ein Eckpunkt von R,
- ein Endpunkt eines Segmentes in S oder
- ein Punkt, in dem die vertikale Erweiterung, die von einem Endpunkt eines Segments ausgeht, auf einem anderen Segment oder auf dem Rand von R endet.

Jeder Endpunkt eines Segments hat zwei vertikale Erweiterungen, nämlich eine obere und eine untere

$\Rightarrow$ Gesamtzahl Eckpunkte ist beschränkt durch

$4 + 2 \cdot n + 2 \cdot (2 \cdot n) = 6 \cdot n + 4$

Betrachte *leftp*(Δ) ⇒ Jedes Trapez hat solch einen Punkt.

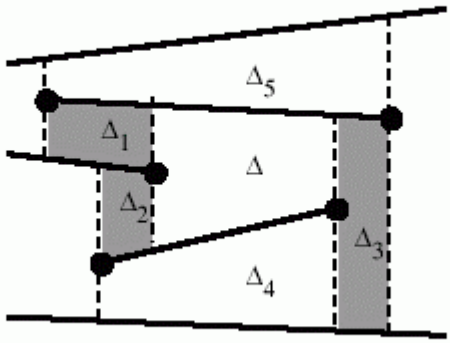
leftp(Δ) ist Endpunkt eines der n Segmente, oder er ist der linke, untere Eckpunkt von R.

Betrachte 5 Fälle für die linke Seite eines Trapezes ⇒

- linker, unterer Eckpunkt von R spielt diese Rolle für genau ein Trapez
- rechter Endpunkt eines Segments kann diese Rolle für höchstens ein Trapez spielen
- linker Endpunkt eines Segments kann der *leftp*(Δ) für höchstens zwei verschiedene Trapeze sein

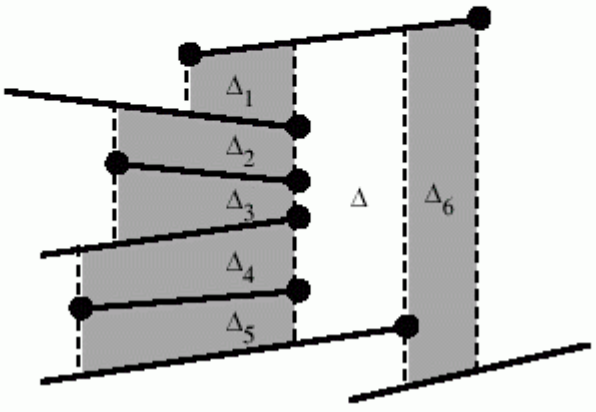
⇒ es gibt höchstens 3·n + 1 Trapeze ■

- Zwei Trapeze Δ und Δ' heißen *benachbart*, wenn sie an einer vertikalen Kante aneinanderstoßen:



Δ ist benachbart zu Δ_1, Δ_2 und Δ_3, aber nicht zu Δ_4 und Δ_5

- Menge der Liniensegmente in allgemeiner Lage
  ⇒ ein Trapez besitzt höchstens 4 benachbarte Trapeze
- Menge der Liniensegmente nicht in allgemeiner Lage
  ⇒ ein Trapez kann beliebige Anzahl Nachbarn haben



- Δ' sei ein Trapez, das zu Δ entlang der linken vertikalen Seite von Δ benachbart ist
  $\Rightarrow$ *top*(Δ) = *top*(Δ') oder *bottom*(Δ) = *bottom*(Δ')
- *top*(Δ) = *top*(Δ') $\Rightarrow$ Δ' *oberer linker Nachbar* von Δ
- *bottom*(Δ) = *bottom*(Δ') $\Rightarrow$ Δ' *unterer linker Nachbar* von Δ
- analog: *oberer rechter Nachbar* bzw. *unterer rechter Nachbar*

Die Trapezoidzerlegung könnte in einer DCEL gespeichert werden

Die spezielle Form der Regionen legt aber eine spezialisiertere Struktur nahe, die die Nachbarschaft der Trapeze benutzt, um die Vernetzung der Struktur herzustellen

- Records für alle Segmente bzw. Endpunkte von S, da sie als *top*(Δ), *bottom*(Δ), *leftp*(Δ) und *rightp*(Δ) fungieren
- Records für die Trapeze von T(S), aber nicht für die Kanten oder Eckpunkte von T(S)
- Record für ein Trapez Δ speichert
  - Zeiger auf *top*(Δ) und *bottom*(Δ)
  - Zeiger auf *leftp*(Δ) und *rightp*(Δ)
  - Zeiger auf die höchstens 4 Nachbarn
- Geometrie eines Trapezes Δ, d.h. die Koordinaten seiner Eckpunkte, nicht explizit verfügbar
- Trapez Δ ist jedoch eindeutig definiert durch *top*(Δ), *bottom*(Δ), *leftp*(Δ) und *rightp*(Δ)
  $\Rightarrow$ Geometrie eines Trapezes Δ kann in konstanter Zeit aus der gespeicherten Information abgeleitet werden

(Hinrichs 2001)

Preprocessing zur Punktlokalisierungs-Anfrage:

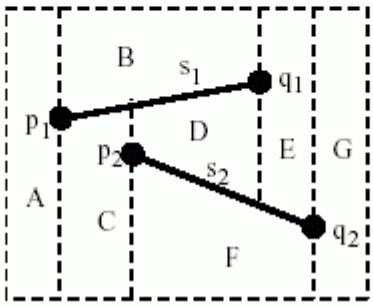
Ein randomisierter, inkrementeller Algorithmus zur Berechnung von Trapezoidzerlegung und Suchstruktur

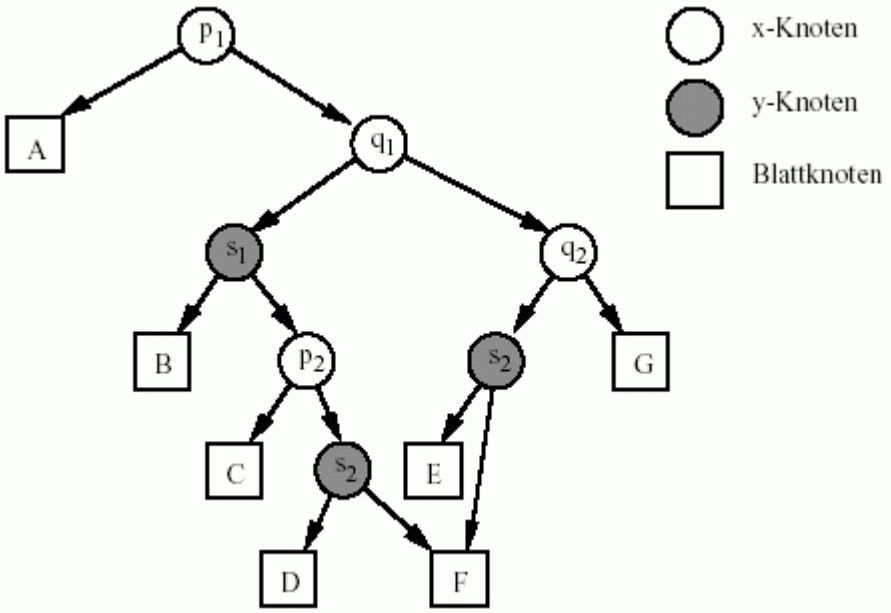
- randomisierter, inkrementeller Algorithmus, der die Trapezoid-zerlegung T(S) einer Menge von n Liniensegmenten in allgemei-ner Lage sowie eine Datenstruktur D konstruiert
  D kann benutzt werden, um Punktlokalisierungsanfragen auf T(S) zu beantworten
- D wird auch *Suchstruktur* genannt
- D ist gerichteter azyklischer Graph mit einer einzigen, ausgezeichneten Wurzel und genau einem Blatt für jedes Trapez von T(S)
- Innere Knoten von D haben jeweils genau zwei verlassende Kanten (out-degree: 2)
- 2 Typen von inneren Knoten:
  - *x-Knoten*, markiert mit einem Endpunkt eines Segments
  - *y-Knoten*, markiert mit einem Segment

- Anfrage mit einem Punkt q startet an der Wurzel und folgt einem Pfad zu einem der Blätter, das dem Trapez Δ von T(S) entspricht, das den Punkt q enthält
- In jedem Knoten auf diesem Pfad wird q getestet, um festzustellen, zu welchem der beiden Nachfolgeknoten man weitergeht:
  - In einem x-Knoten wird getestet, ob q zur Linken oder zur Rechten der vertikalen Linie durch den Endpunkt liegt, der in diesem Knoten gespeichert ist.
  - Ein y-Knoten wird nur angelaufen, wenn die Vertikale durch q das in dem y-Knoten gespeicherte Segment s schneidet. In solch einem y-Knoten wird getestet, ob q oberhalb oder unterhalb des in diesem Knoten gespeicherten Segments s liegt.

- Tests in den inneren Knoten haben nur zwei mögliche Ergebnisse: links oder rechts eines Endpunktes für einen x-Knoten, oberhalb oder unterhalb eines Segmentes für einen y-Knoten.
- Problem:
  Anfragepunkt exakt auf vertikaler Linie oder auf Segment
- vorläufige Annahme: das passiert nicht!

- Suchstruktur D und Trapezoidzerlegung T(S), die von dem Algorithmus konstruiert werden, sind miteinander verknüpft: ein Trapez Δ von T(S) hat einen Zeiger auf das Δ entsprechende Blatt von D, und ein Blatt von D hat einen Zeiger auf das entsprechende Trapez von T(S):





- Algorithmus zur Konstruktion der Suchstruktur arbeitet inkrementell: er fügt ein Segment nach dem anderen hinzu, wobei nach jedem Segment jeweils die Suchstruktur und die Trapezoidzerlegung aktualisiert werden
- Reihenfolge, in der die Segmente hinzugefügt werden, beeinflußt die Qualität der Suchstruktur
- Wähle eine zufallsmäßig bestimmte Einfügereihenfolge
  $\Rightarrow$ *randomisierter, inkrementeller* Algorithmus

- Globale Struktur des Algorithmus:

TrapezoidalMap(S)
Eingabe: Eine Menge S von n nicht-kreuzenden Segmenten.
Ausgabe: Die Trapezoidzerlegung T(S) und eine Suchstruktur D für T(S) in einem Container.

1. Bestimme ein achsenparalleles Rechteck R, das alle Segmente enthält, und initialisiere für R die Trapezoidzerlegung T und die Suchstruktur D.
2. Bestimme eine Zufallspermutation $s_1, s_2, \ldots, s_n$ der Elemente von S.
3. for i := 1 to n do
4. Finde die Menge $\Delta_0, \Delta_1, \ldots, \Delta_k$ der Trapeze in T, die durch s_i echt geschnitten werden.
5. Entferne $\Delta_0, \Delta_1, \ldots, \Delta_k$ aus T und ersetze sie durch die neuen Trapeze, die durch das Einfügen von s_i erzeugt werden.
6. Entferne die Blätter für $\Delta_0, \Delta_1, \ldots, \Delta_k$ aus D, und erzeuge Blätter für die neuen Trapeze. Verbinde die neuen Blätter zu den existierenden inneren Knoten, indem neue innere Knoten wie unten erklärt eingefügt werden.

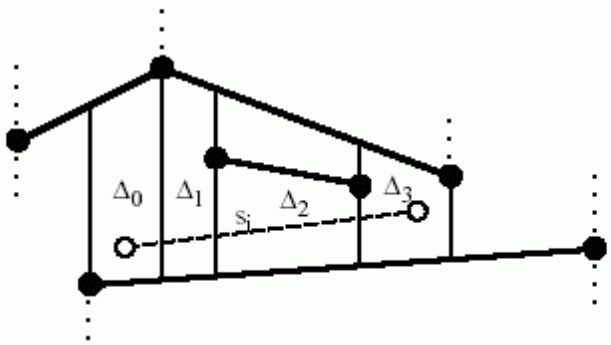
- $S_i := \{s_1, s_2, \ldots, s_i\}$
- Schleifeninvariante von *TrapezoidalMap*: T ist die Trapezoidzerlegung für S_i und D eine gültige Suchstruktur für T.
- Initialisierung von T als $T(S_0) = T(\emptyset)$ und von D in Schritt 1:
  - Trapezoidzerlegung für die leere Menge besteht aus einem einzigen Trapez, dem einschliessenden Rechteck R
  - Suchstruktur D für $T(\emptyset)$ besteht aus einem einzigen Blattknoten für dieses Trapez
- Berechnung der Zufallspermutation in Schritt 2:
  Zufallszahlengenerator *Random*(i) erzeugt in O(1) Zeit zu einer Integer-Eingabe i eine Integer-Zufallszahl zwischen 1 und i
- Erzeugung einer Zufallspermutation in linearer Zeit:

RandomPermutation(A)
Eingabe: Ein Array A[1..n].
Ausgabe: Das Array A[1..n] mit den gleichen Elementen, die aber gemäß einer Zufallspermutation umgeordnet sind.

```
1.  for  i := n downto 2  do
2.       rn := Random(i);
3.       A[i] :=: A[rn];
```

(↑ vertauschen)

- Einfügen eines Segments s_i in den Schritten 4-6:
  Um die gegenwärtige Trapezoidzerlegung zu modifizieren, muß zunächst festgestellt werden, in welchen Bereichen sich diese ändert. Dies ist genau in den Trapezen der Fall, die von s_i geschnitten werden.
- Ein Trapez von $T(S_{i-1})$ ist genau dann nicht in $T(S_i)$ enthalten, wenn es durch s_i geschnitten wird. Seien $\Delta_0, \Delta_1, \ldots, \Delta_k$ diese Trapeze, geordnet von links nach rechts entlang s_i:



- Δ_{j+1} muß einer der rechten Nachbarn von Δ_j sein:
  Falls *rightp*(Δ_j) oberhalb s_i liegt, so ist Δ_{j+1} der untere rechte Nachbar von Δ_j, ansonsten der obere rechte Nachbar.
- Trapez Δ_0 bekannt $\Rightarrow$ $\Delta_1, \ldots, \Delta_k$ können durch Traversierung der Darstellung der Trapezoidzerlegung bestimmt werden
- Linker Endpunkt p von s_i nicht Endpunkt eines Segmentes in S_{i-1} $\Rightarrow$ p muß wegen allgemeiner Lage im Innern von Δ_0 liegen
  $\Rightarrow$ Bestimmung von Δ_0 durch Lokalisierung von p in $T(S_{i-1})$.
- Benutze D als Suchstruktur für $T = T(S_{i-1})$.
- Achtung, falls p bereits Endpunkt eines Segmentes in S_{i-1} !

- Bestimmung von Δ_0: starte Suche in D
- Falls p noch nicht als Endpunkt eines Segmentes in S_{i-1} vorkommt, so läuft der Suchalgorithmus ohne Probleme und endet in dem Δ_0 entsprechenden Blatt.
- Falls p jedoch bereits Endpunkt eines Segmentes in S_{i-1} ist, so können während der Traversierung der Suchstruktur die folgenden Situationen eintreten:
  - p liegt auf Vertikale durch einen Punkt in einem x-Knoten:
    Simuliere leichte Verschiebung des Punktes p nach rechts, indem Traversierung der Suchstruktur mit dem rechten Kindknoten des x-Knotens fortgesetzt wird.
  - p liegt auf dem Segment s eines y-Knotens, d.h. s_i hat gemeinsamen linken Endpunkt mit s:
    Steigung von s_i > Steigung von s $\Rightarrow$ p oberhalb s
    Steigung von $s_i \leq$ Steigung von s $\Rightarrow$ p unterhalb s.

$\Rightarrow$ Suche endet in dem ersten Trapez Δ_0, das von s_i echt geschnitten wird

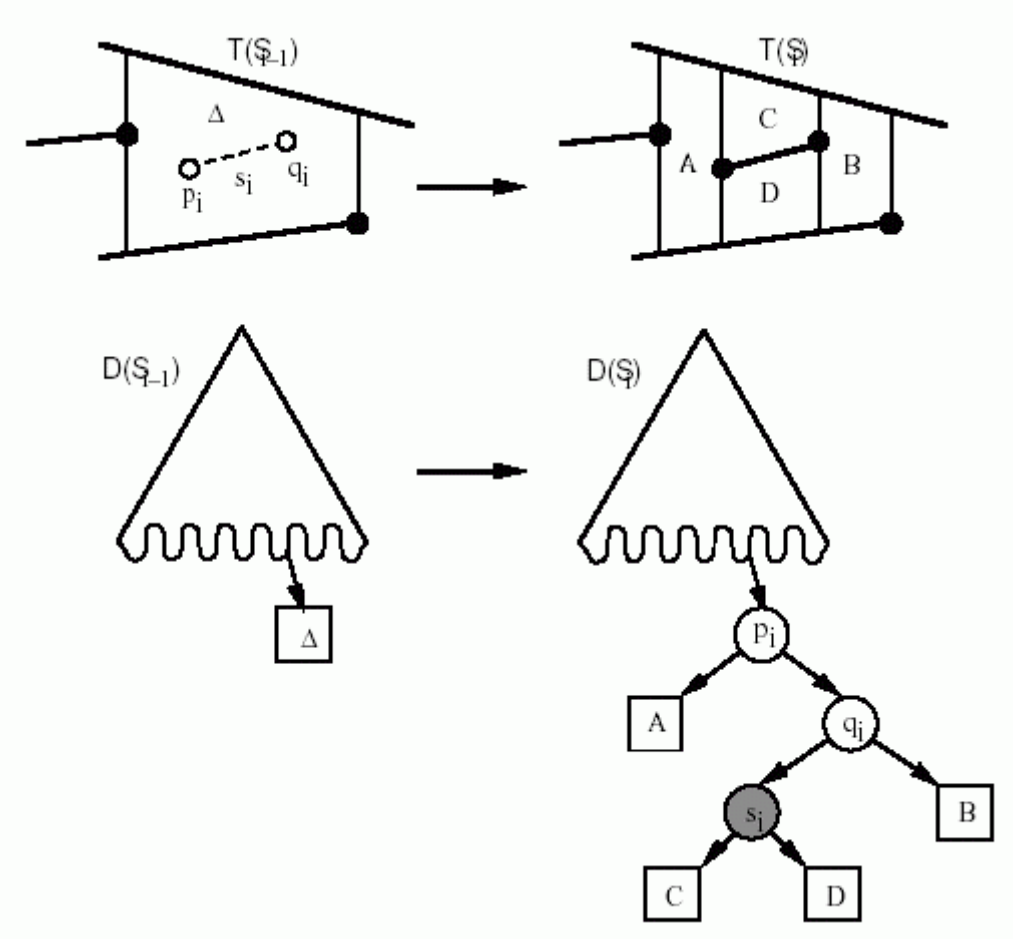
- Algorithmus zum Auffinden von $\Delta_0, \ldots, \Delta_k$:

FollowSegment(T, s_i)
Eingabe: Eine Trapezoidzerlegung T und ein neues Segment s_i.
Ausgabe: Die Folge $\Delta_0, \ldots, \Delta_k$ der Trapeze, die von s_i echt geschnitten werden.

1. Seien p und q der linke und rechte Endpunkt von s_i.
2. Suche mit p in der Suchstruktur D, um Δ_0 zu finden.
3. j := 0;
4. while q liegt rechts von *rightp*(Δ_j) do begin
5. if *rightp*(Δ_j) liegt oberhalb s_i then
6. Δ_{j+1} := unterer rechter Nachbar von Δ_j;
   else
7. Δ_{j+1} := oberer rechter Nachbar von Δ_j;
8. j := j + 1;
   end;
9. return $\Delta_0, \Delta_1, \ldots, \Delta_j$

Nächster Schritt: Aktualisierung von *T* und *D*

s_i vollständig in einem Trapez $\Delta = \Delta_0$ enthalten $\Rightarrow$
Aktualisiere T, indem Δ durch 4 neue Trapeze A, B, C und D ersetzt wird: Die zur Initialisierung der Records für die neuen Trapeze benötigten Informationen, d.h. die Nachbarn, die oberen und unteren begrenzenden Segmente und die die linken und rechten vertikalen Kanten festlegenden Punkte, können in O(1) Zeit aus dem Segment s_i und der für Δ gespeicherten Information hergeleitet werden.
Aktualisiere die Suchstruktur D durch Ersetzen des Blattes für Δ durch einen kleinen Teilbaum mit 4 Blättern: Dieser Teilbaum enthält zwei x-Knoten, in denen gegen die beiden Endpunkte p_i und q_i von s_i getestet wird, und einen y-Knoten, in dem gegen das Segment s_i selbst getestet wird. Dies reicht aus, um festzustellen, in welchem der Trapeze A, B, C oder D ein Anfragepunkt liegt, wenn wir bereits wissen, daß dieser Anfragepunkt in Δ liegt:



s_i schneidet zwei oder mehr Trapeze $\Delta_0, \Delta_1, \ldots, \Delta_k \Rightarrow$

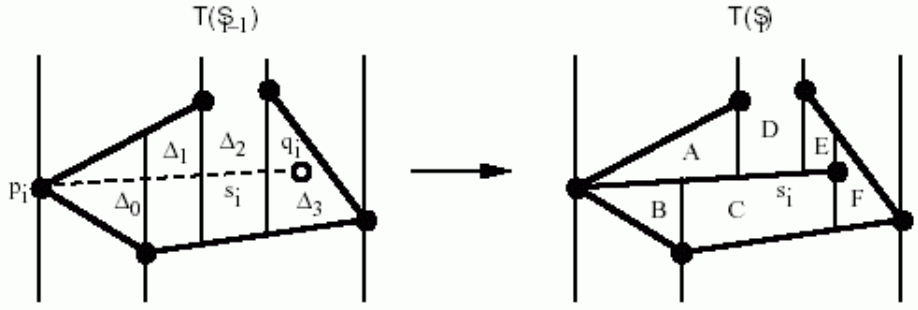
Aktualisierung von T:

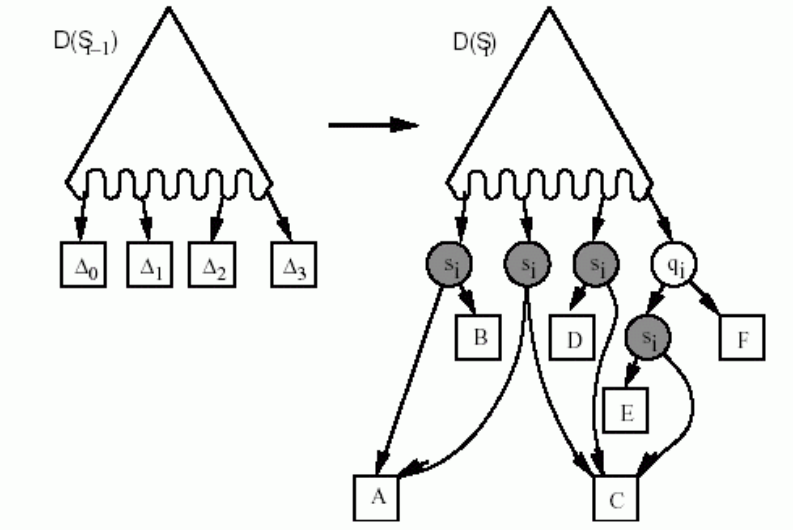
Errichte zuerst vertikale Erweiterungen in den Endpunkten p_i und q_i von s_i, durch die Δ_0 und Δ_k in jeweils drei neue Trapeze aufgeteilt werden. Dies ist allerdings nur notwendig für solche Endpunkte, die noch nicht vorhanden sind. Dann werden die vertikalen Erweiterungen gekürzt, die jetzt auf s_i enden. Unter Ausnutzung der in den Trapezen $\Delta_0, \Delta_1, \ldots, \Delta_k$ gespeicherten Informationen benötigt dieser Schritt Zeit linear in der Anzahl der von s_i geschnittenen Trapeze.

Aktualisierung der Suchstruktur D:

Entferne die Blätter für $\Delta_0, \Delta_1, \ldots, \Delta_k$ und erzeuge die Blätter für die neuen Trapeze sowie neue innere Knoten.
Die Blätter für $\Delta_1, \ldots, \Delta_{k-1}$ werden durch einzelne y-Knoten für das Segment s_i ersetzt.
Enthält Δ_0 den linken Endpunkt p_i von s_i in seinem Inneren, d.h. Δ_0 wird in 3 neue Trapeze aufgeteilt, so wird das Blatt für Δ_0 durch einen x-Knoten für p_i und einen y-Knoten für s_i ersetzt. Enthält Δ_k den rechten Endpunkt q_i von s_i in seinem Inneren, so wird das Blatt für Δ_k durch einen x-Knoten für q_i und einen y-Knoten für s_i ersetzt. Enthalten Δ_0 den Punkt p_i bzw. Δ_k den Punkt q_i nicht, so werden sie wie $\Delta_1, \ldots, \Delta_{k-1}$ behandelt, d.h. ihr Blatt wird jeweils durch einen einzelnen y-Knoten für s_i ersetzt. Die neuen inneren Knoten verweisen auf die korrekten Blätter der neuen Trapeze.

Da in $T(S_i)$ evtl. mehrere Teiltrapeze, die zu verschiedenen Trapezen von $T(S_{i-1})$ gehören, zusammengefaßt werden, kann es in D mehrere innere Knoten geben, die auf dasselbe Blatt und somit auf dasselbe Trapez verweisen.





- Schleifeninvariante von *TrapezoidalMap*: T ist die Trapezoid-zerlegung für S_i und D eine gültige Suchstruktur für T
  $\Rightarrow$ Korrektheit des Algorithmus
- Reihenfolge, in der die Segmente bearbeitet werden, hat signifikante Auswirkungen auf die sich ergebende Suchstruktur D und auf die Laufzeit des Algorithmus
- Für einige Fälle hat die Suchstruktur quadratische Größe und lineare Suchzeit, aber für andere Permutationen der gleichen Menge von Liniensegmenten ist die sich ergebende Suchstruktur sehr viel besser.

- Versuche nicht, eine möglichst optimale Eingabereihenfolge zu bestimmen, sondern nehme eine Zufallsreihenfolge $\Rightarrow$ probabilistische Analyse: *erwartetes* Leistungsverhalten des Algorithmus.
- *Erwartete Laufzeit* des Algorithmus ist das Mittel der Laufzeiten über alle n! Permutationen.
- Für jede Permutation resultiert eine andere Suchstruktur. Die *erwartete Größe* von D ist der Mittelwert der Größen aller n! Suchstrukturen.
- Die *erwartete Anfragezeit* für einen Anfragepunkt q ist das Mittel der Anfragezeiten für Punkt q, genommen über alle n! Suchstrukturen.

Abschätzung des Aufwandes (hier ohne Beweis, siehe Hinrichs 2001):

Algorithmus *TrapezoidalMap* berechnet die Trapezoidzerlegung T(S) einer Menge S von n Liniensegmenten in allgemeiner Lage und eine Suchstruktur D für T(S) in O(n log n) erwarteter Zeit. Die Suchstruktur hat erwartete Größe O(n), und für einen Anfrage-punkt q beträgt die erwartete Anfragezeit O(log n).

wenn man die Einschränkung an die Lage der Strecken noch fallenlassen kann, folgt:

Sei S eine planare Unterteilung mit n Kanten. In O(n log n) erwarteter Zeit kann man eine Datenstruktur konstruieren, die O(n) erwarteten Speicherplatz benötigt, so daß für jeden Anfragepunkt q eine Punktlokalisierung in O(log n) erwarteter Zeit durchgeführt werden kann.

(Anmerkung: Die Konstruktionszeit O(n log n) ist für das Problem noch nicht optimal.)

Behandlung der degenerierten Fälle (spezielle Lage der Strecken):

- Bisher zwei vereinfachende Annahmen:
  - Die Menge der Liniensegmente soll sich in allgemeiner Lage befinden, d.h. keine zwei verschiedenen Segmentendpunkte von S dürfen die gleiche x-Koordinate besitzen, es kann also insbesondere keine vertikalen Segmente geben.
  - Ein Anfragepunkt q darf nicht auf der vertikalen Linie eines x-Knotens oder auf dem Segment eines y-Knotens auf seinem Suchpfad liegen.

- Vermeidung der Annahme, daß keine zwei verschiedenen Segmentendpunkte auf einer gemeinsamen vertikalen Linie liegen
- Vertikale Richtung zur Definition der Trapezoidzerlegung war unwesentlich $\Rightarrow$ rotiere Koordinatenachsen ein wenig
- Rotationswinkel klein genug $\Rightarrow$ keine zwei verschiedenen Endpunkte liegen auf einer Vertikalen
- Aber: Rotationen um sehr kleine Winkel führen zu numerischen Problemen

- Vorteilhafter: *Scherung* statt Rotation
- Scherung entlang der x-Achse um einen Wert $\varepsilon > 0$

  $$\varphi: \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x + \varepsilon \cdot y \\ y \end{pmatrix}$$

  bildet eine vertikale Gerade auf eine Gerade mit der Steigung $1/\varepsilon$ ab $\Rightarrow$ zwei verschiedene Punkte auf einer Vertikalen werden auf Punkte mit verschiedenen x-Koordinaten abgebildet.
- $\varepsilon > 0$ klein genug $\Rightarrow$ Transformation verändert die Anordnung der Eingabepunkte bzgl. der x-Koordinaten nicht

- Annahme: $\varepsilon > 0$ so gewählt, daß horizontale Anordnung der Punkte nicht verändert wird
- Später: ε muß gar nicht berechnet werden
- S Menge von n nicht-kreuzenden Liniensegmenten in der Ebene
- Algorithmus *TrapezoidalMap* erhält als Eingabe die Menge

  $\varphi S := \{\varphi s: s \in S\}$
- Trick: Punkt $\varphi p = (x + \varepsilon \cdot y, y)$ wird gespeichert als (x, y)
- Algorithmus berechnet keine geometrischen Objekte $\Rightarrow$ keine numerischen Probleme

mit diesem Ansatz lässt sich der Algorithmus *TrapezoidalMap* auf Strecken in beliebiger Lage anwenden
(Details bei Hinrichs 2001)

Ergebnis:

Algorithmus *TrapezoidalMap* berechnet die Trapezoidzerlegung T(S) einer Menge S von n nicht-kreuzenden Liniensegmenten und eine Suchstruktur D für T(S) in O(n log n) erwarteter Zeit. Die Suchstruktur hat erwartete Größe O(n), und für einen Anfrage-punkt q beträgt die erwartete Anfragezeit O(log n).

Bisher nur Aussage über die erwartete Anfragezeit (Durchschnitt)

- Für jede Permutation der Segmentmenge könnte die sich mit dem Algorithmus ergebende Suchstruktur eine schlechte Anfragezeit für 1 oder mehrere spezielle Anfragepunkte besitzen
- jedoch ist dafür die Wahrscheinlichkeit sehr gering:

Satz:

Sei S eine Menge von n nicht-kreuzenden Liniensegmenten, sei q ein Anfragepunkt, und sei λ ein Parameter mit $\lambda > 0$. Dann beträgt die Wahrscheinlichkeit höchstens $1/(n+1)^{\lambda \cdot \ln 1.25 - 1}$, daß der Suchpfad für q in der durch Algorithmus *TrapezoidalMap* berechneten Suchstruktur D mehr als $3 \cdot \lambda \cdot \ln(n+1)$ Knoten besitzt.

Satz:

Sei S eine Menge von n nicht-kreuzenden Liniensegmenten, und sei λ ein Parameter mit $\lambda > 0$. Dann beträgt die Wahrscheinlichkeit höchstens $2/(n+1)^{\lambda \cdot \ln 1.25 - 3}$, daß die Tiefe der durch Algorithmus *TrapezoidalMap* berechneten Suchstruktur D größer als $3 \cdot \lambda \cdot \ln(n+1)$ ist.

- $\lambda = 20 \Rightarrow$ Wahrscheinlichkeit, daß die Tiefe von D größer als $3 \cdot \lambda \cdot \ln(n+1)$ ist, beträgt höchstens $2/(n+1)^{1.4}$, was für $n > 4$ kleiner 1/4 ist
  $\Rightarrow$ Wahrscheinlichkeit, daß D eine gute Anfragezeit besitzt, ist mindestens 3/4.
  Ähnlich:
  Wahrscheinlichkeit, daß die Größe der Suchstruktur O(n) ist, beträgt mindestens 3/4.
  maximale Länge eines Suchpfades in der Suchstruktur O(log n), Größe der Suchstruktur O(n) $\Rightarrow$ Laufzeit des Konstruktionsalgorithmus O(n log n)
  $\Rightarrow$ Wahrscheinlichkeit, daß Anfragezeit, Größe und Konstruktionszeit gut sind, beträgt mindestens 1/2.

- Algorithmus, der eine Suchstruktur konstruiert, die im schlimmsten Fall O(log n) Anfragezeit und O(n) Speicherplatz benötigt:
  Rufe Algorithmus *TrapezoidalMap* für die Menge S auf und führe Buch über die Größe und Tiefe der erzeugten Suchstruktur. Sobald für geeignet gewählte Konstanten c_1 bzw. c_2 die Größe der Suchstruktur $c_1 \cdot n$ oder die Tiefe $c_2 \cdot \log n$ überschreitet, stoppe den Algorithmus und starte erneut mit einer neuen Zufallspermutation.
  Wahrscheinlichkeit, daß eine Permutation zu einer Suchstruktur mit der gewünschten Größe und Anfragezeit führt, beträgt mindestens $1/2 \Rightarrow$ nach spätestens 2 Versuchen sollte man erfolgreich sein.
  Für große n ist die Wahrscheinlichkeit fast 1, so daß die erwartete Anzahl Versuche nur geringfügig größer als 1 ist.

somit:

Satz:

Sei S eine planare Unterteilung mit n Kanten. Es existiert eine Punktlokalisierungsstruktur für S, die im schlimmsten Fall O(n) Speicherplatz benötigt und Punktlokalisierungsanfragen im schlimmsten Fall in O(log n) Zeit beantwortet.

Der Satz sagt nichts aus über die Zeit, die zum Aufbau der Trapezoidzerlegung und der Suchstruktur benötigt wird. Um die maximale Weglänge der Suchstruktur zu beschränken, benötigt man $2 \cdot (n+1)^2$ Testpunkte. Es ist möglich, diese Anzahl Testpunkte auf O(n log n) zu reduzieren. Damit erhält man eine Konstruktionszeit von $O(n \log^2 n)$.

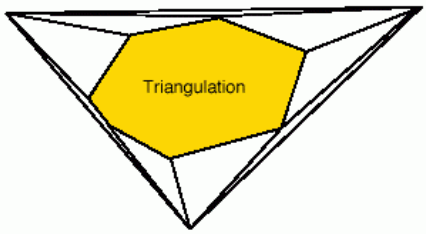
(Hinrichs 2001)

ein alternativer Ansatz:
Triangulation statt Trapezoidzerlegung

Der Algorithmus von Kirkpatrick

(hier nach Vahrenhold 2002 skizziert)

- Asymptotisch optimaler Algorithmus [Kirkpatrick, 1983].
  Preprocessing: $\mathcal{O}(n)$ Platz: $\mathcal{O}(n)$ Anfragezeit: $\mathcal{O}(\log n)$.
- Annahme: Planare Partition in Dreieck einbeschrieben und trianguliert.
- Auffinden eines geeigneten Dreiecks und Triangulation in $\mathcal{O}(n)$ Zeit möglich.



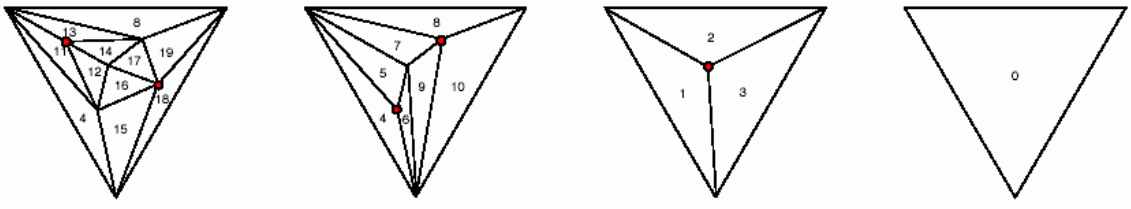
- Bezeichne resultierende Triangulation mit $\mathcal{P}$.
- Euler-Formel $\Rightarrow |\mathcal{P}| \in \mathcal{O}(n)$.

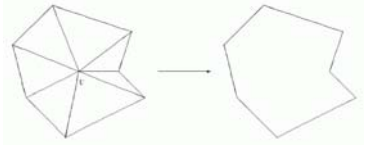
Aufbau der Datenstruktur:

Definition: Zwei Knoten eines Graphen heissen unabhängig, wenn sie nicht durch eine Kante verbunden sind.

Konstruktion einer Folge von Triangulationen $S_1, \ldots, S_{h(n)}$:

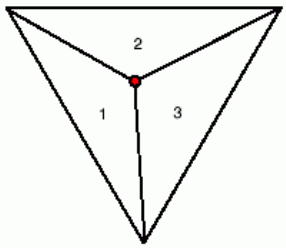
- Setze $S_1 := \mathcal{P}$; $i := 1$.
- Solange $|V(S_i)| > 3$, erzeuge S_{i+1} aus S_i wie folgt:
  1. Setze $S_{i+1} := S_i$.
  2. Entferne eine Menge von unabhängigen Knoten und die hiervon ausgehenden Kanten aus S_{i+1}.
  3. Vervollständige S_{i+1} zu einer Triangulation.

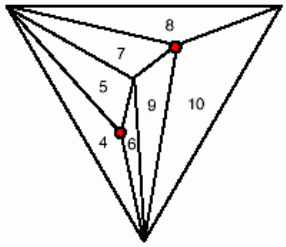


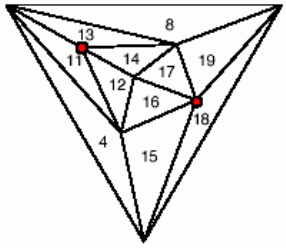
(Entfernen eines Knotens aus einer Triangulierung: )

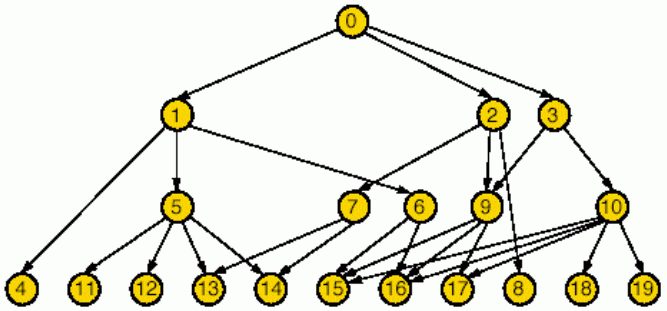
Suchstruktur für Triangulationen:

- Gerichteter azyklischer Graph $\mathcal{T}$.
- Knoten R_k entspricht Dreieck T_k.
- $(R_k, R_j) \in E(\mathcal{T}) \iff$
  - T_k wird aus S_i gelöscht.
  - T_j wird in S_{i+1} erzeugt.
  - $T_k \cap T_j \neq \emptyset$.
- Anfrage: Verfolgen eines Pfades in $\mathcal{T}$.









- Tiefe des Suchgraphen entscheidend für Laufzeit der Anfrage.
- Anzahl der in jedem Schritt entfernten Knoten entscheidend für Tiefe des Suchgraphen. Sei $n_i := |V(S_i)|$.

Invarianten für Konstruktion des Suchgraphen:

1. $n_i = \alpha_i n_{i-1}$, mit $\alpha_i \leq \alpha < 1, \quad 2 \leq i \leq h(n)$.
2. Ein Dreieck in S_i schneidet $\leq H$ Dreiecke in S_{i-1} (und umgekehrt).

Konsequenzen:

- Anfragezeit: $H \cdot h(n) \leq H \cdot \lceil \log_{1/\alpha} n \rceil \in \mathcal{O}(\log n)$.
- Speicherplatz: $\mathcal{O}(n)$, denn $|E(\mathcal{T})| \leq H \cdot |V(\mathcal{T})|$ und:
  - $|V(\mathcal{T})| = \sum_{i=1}^{h(n)} |F(S_i)|$ und $|F(S_i)| \leq 2 \cdot n_i$ (Euler-Formel).
  - $|V(\mathcal{T})| \leq 2 \cdot \sum_{i=1}^{h(n)} n_i \leq 2 \cdot n_1 \cdot \sum_{i=0}^{h(n)-1} \alpha^i \leq 2 \cdot \frac{n}{1-\alpha}$.

Auswahl unabhängiger Knoten:

- Wähle unabhängige Knoten vom Grad $< K$.
- Markiere Nachbarn, entferne Knoten. Reihenfolge unrelevant.

Verifikation der Invarianten:

1. Benutze Euler-Formel für Triangulationen:
   - $e = 3n - 6$; jede Kante trägt 2 zum Gesamt-Knotengrad bei.
   - Gesamt-Knotengrad: $6n - 12 < 6n \Rightarrow \geq n/2$ Knoten vom Grad < 12.
   - Drei Randknoten und pro gewähltem Knoten ≤ 11 Knoten nicht wählbar.
   - Resultat für Anzahl v wählbarer Knoten und Konstante α:

$$v \geq \left\lfloor \frac{1}{12}\left(\frac{n}{2} - 3\right) \right\rfloor \quad \Rightarrow \quad \alpha \approx 1 - \frac{1}{24} < 0.959 < 1.$$

2. Entfernter Knoten erzeugt Polygon mit $< K$ Kanten, dieses ergibt höchstens $H := K - 2$ Dreiecke.

- Optimaler Algorithmus. Suchstruktur basierend auf Verfeinerung von Triangulationen eines planaren Graphen.
- Anschauliches Konzept, einfach zu realisieren.
- Praktische Laufzeit relativ hoch (Konstante α etwas besser abschätzbar).

Weitere Anwendung (in modifizierter Form):

- Punktlokalisierung in konvexem dreidimensionalen Polyeder $\mathcal{P}$.