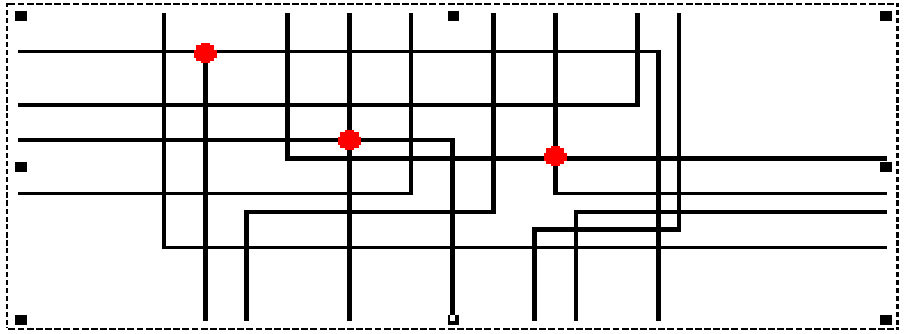
Kapitel 7

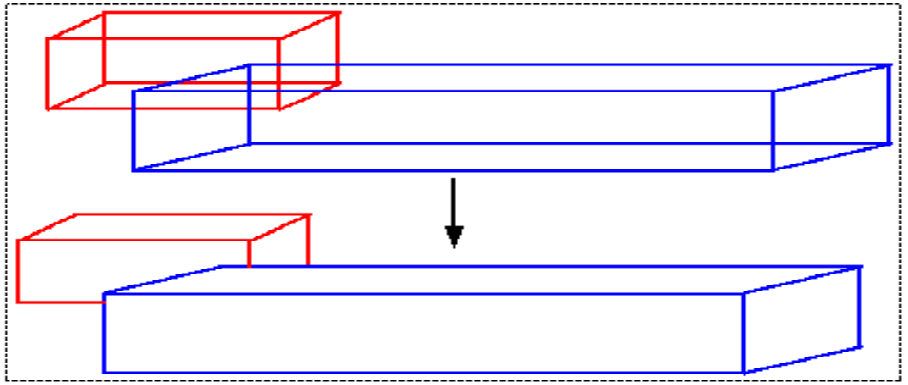
(Einfache)
Geometrische Algorithmen

Einige Anwendungsbereiche und -probleme

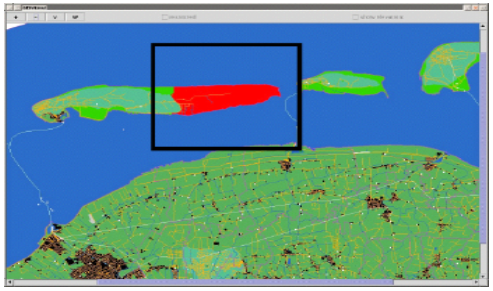
Design integrierter Schaltungen
(z.B. Schnittpunkte)



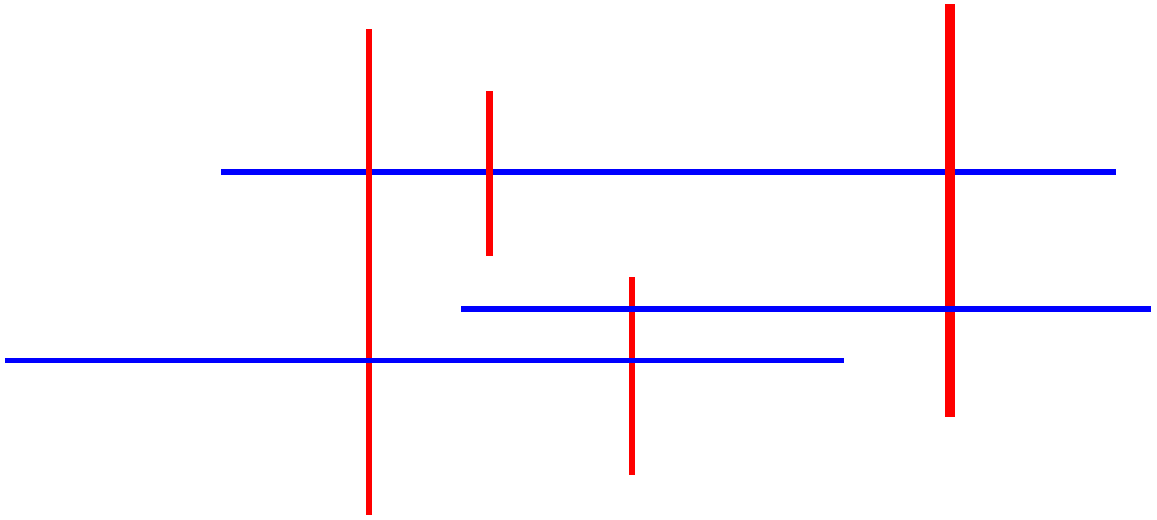
Computer-Graphik
(z.B. Entfernung verdeckter Linien)



Geoinformationssysteme
(z.B. Suchanfragen)



Schnitte zwischen achsenparallelen Segmenten



- Gegeben:
  H = Menge der horizontalen Segmente
  V = Menge der vertikalen Segmente
  $S = H \cup V$, $n = |S|$ sei die Gesamtzahl der Segmente
  In Abschätzungen gehen wir von $|H|, |V| \approx \frac{n}{2}$ aus.
- Gesucht: Alle Paare sich schneidender Segmente.
- Annahme: Es gibt keine überlappenden Segmente.

Schnitte zwischen achsenparallelen Segmenten (Forts.)

Der naive Algorithmus:

for each $h \in H$ **do**
  for each $v \in V$ **do**
    if h schneidet v **then** report(h, v)

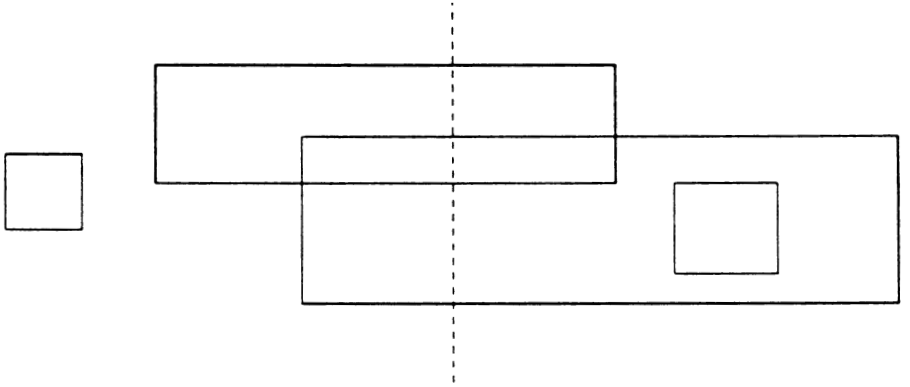
- Dieser Algorithmus hat eine Laufzeit von $O(|H| \times |V|)$, d. h. $O(n^2)$.
- Aber die Anzahl der Schnitte k könnte $<< n^2$ sein.
- Idealerweise hätten wir gerne einen *ausgabe-sensitiven* Algorithmus, d.h. einen Algorithmus, dessen Laufzeit hauptsächlich von der Größe der Ausgabe abhängt.
- Zunächst versuchen wir, das Divide-and-Conquer-Paradigma anzuwenden, was bei Beschränkung auf achsenparallele Objekte erfolgreich ist.
- Danach führen ein neues Paradigma, das Plane-Sweep-Paradigma ein, das sich auch für andere geometrische/räumliche Probleme eignet.

7.1 Geometrische Divide-and-Conquer-Algorithmen [1]

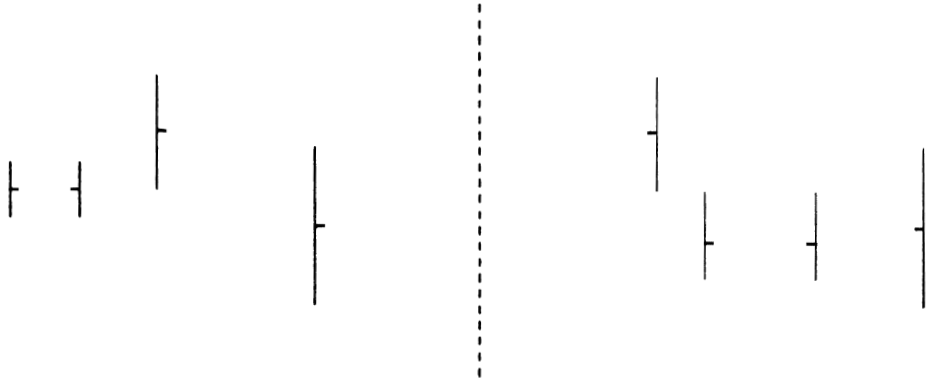
Wie könnte man z.B. ein Rechteckschnitt-Problem zerlegen?

- halbiere Objektmengen: problematisch, da keine geometrische Eigenschaft genutzt wird

- verwende teilende (z.B. vertikale) Gerade:
  problematisch, weil die Mengen der linken und rechten Objekte nicht disjunkt sind.



- besser: nutze eine *getrennte Repräsentation* der Objekte durch ihre linken und rechten Enden



[1] basiert auf Güting,R.H./Dieker,S.: Datenstrukturen und Algorithmen, 2.Auflage. B.G.Teubner, Stuttgart 2003 (Kapitel 7)

zurück zum **Segmentschnitt-Problem:**

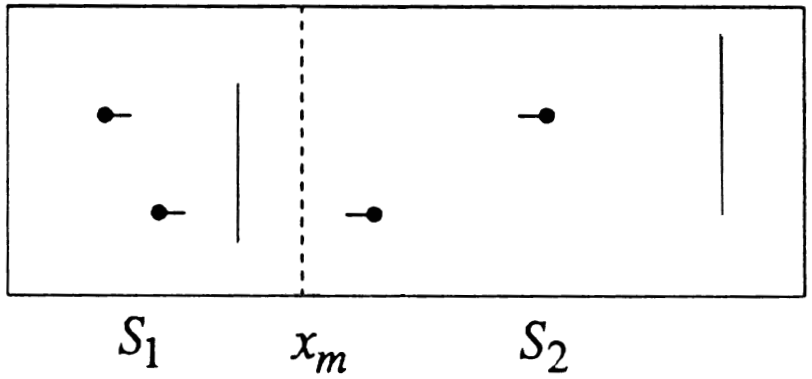
Getrennte Repräsentation bei achsenparallelen Segmenten bedeutet:

Jedes horizontale Segment wird durch seinen linken und seinen rechten Endpunkt repräsentiert.

Verwendet wird deshalb eine Menge S von vertikalen Segmenten und von Punkten, die als linke oder rechte Endpunkte identifizierbar sind.

Algorithmusskizze ReportCuts(S):

Divide: Wähle eine x-Koordinate x_m (nicht unbedingt mittig), die S in zwei gleich große Teilmengen S_1 und S_2 zerlegt.



Conquer: ReportCuts(S_1); ReportCuts(S_2);
Merge: ?

Segmentschnitt-Problem (Forts.)

Rekursionsinvariante: ReportCuts(X) liefert alle Schnitte derjenigen Originalobjekte, die in X durch mindestens ein Ende repräsentiert sind.

Der Merge-Schritt muss dafür sorgen, dass diese Eigenschaft von S_1 und S_2 auf S übertragen wird. Er braucht also nur noch Schnitte zu suchen, die es zwischen einem in S_1 repräsentierten Objekt und einem in S_2 repräsentierten Objekt gibt.

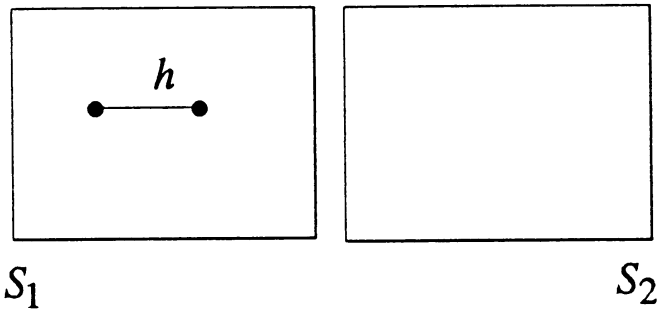
Fallanalyse $\longrightarrow\longrightarrow\longrightarrow$

Segmentschnitt-Problem: Merge-Fälle

Wir betrachten ein einzelnes horizontales Segment h, das in S_1 repräsentiert ist (für S_2 symmetrisch). Es gibt folgende Fälle:

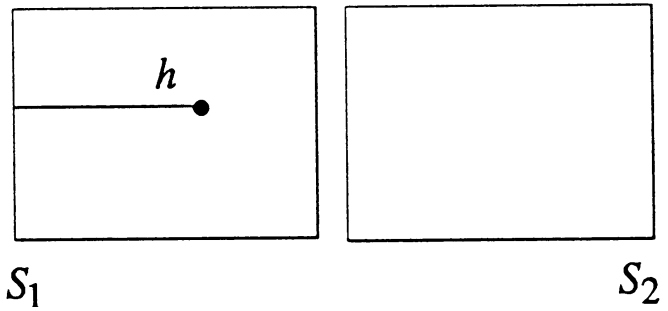
1. Beide Endpunkte von h liegen in S_1.

Offensichtlich schneidet h kein Segment in S_2.



2. Nur der rechte Endpunkt liegt in S_1.

Auch in diesem Fall schneidet h kein Segment in S_2.

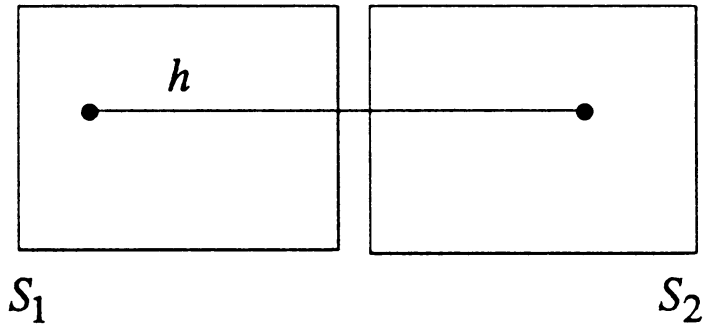


Segmentschnitt-Problem: Merge-Fälle (Forts.)

3. Nur der linke Endpunkt liegt in S_1.

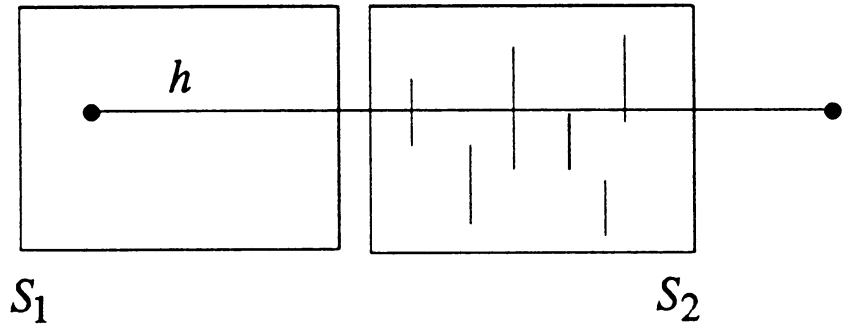
   (a) Der rechte Endpunkt liegt in S_2.

   Alle Schnitte mit Segmenten in S_2 sind bereits ausgegeben, weil h dort repräsentiert war.



   (b) Der rechte Endpunkt liegt rechts von S_2.

   einzig relevanter Fall !



Segmentschnitt-Algorithmus mit Divide-and-Conquer

algorithm SegmentIntersectionDaC(H, V) :

input:

Menge horizontaler Segmente H, Menge vertikaler Segmente V

output:

alle sich schneidenden Paare (h, v) mit $h \in H$ und $v \in V$

$$\begin{aligned}\text{Bilde } S \leftarrow \quad & \{\, (x_1, \mathsf{horizleft}, \quad (x_1, x_2, y)) \mid (x_1, x_2, y) \in H \,\} \\ \cup \; & \{\, (x_2, \mathsf{horizright}, (x_1, x_2, y)) \mid (x_1, x_2, y) \in H \,\} \\ \cup \quad & \{\, (x, \mathsf{vertical}, (x, y_1, y_2)) \mid (x, y_1, y_2) \in V \,\}.\end{aligned}$$

Jedes Objekt wird hier notiert durch seine vollständige Beschreibung, z.B. ein horizontales Segment (x_1, x_2, y) durch sein x-Intervall und seine y-Koordinate; eigentlich wären Verweise auf Originalobjekte angebracht. Vorangestellt ist die x-Koordinate für die nötigen Aufteilungen. Deshalb ist jedes horizontale Objekt zweimal, mit linkem bzw. rechtem (x-) Randwert als linker bzw. rechter Endpunkt, aufgenommen.

Sortiere S nach der 1. Komponente, also nach der x-Koordinate.

Rufe ReportCuts(S, L, R, V) [2] auf.

[3] Die hinteren drei Argumente sind nur für Hilfsergebnisse bei der Rückkkehr aus der Rekursion nötig; für den Aufruf sind sie irrelevant.

algorithm ReportCuts(S, L, R, V) :

input: S — Menge von linken und rechten "Endpunkten" sowie von vertikalen Segmenten, nach x-Koordinaten sortiert

output:

- L — enthält die "linken Endpunkte" in S, deren Partner nicht in S liegt ... in der Form (y-Koordinate, horizontales Segment)
- R — analog für die "rechten Endpunkte"
- V — enthält die vertikalen Segmente in S
  ... in der Form (y-Intervall, vertikales Segment)
- Direkt ausgegeben werden alle sich schneidenden Paare (h, v), wobei $h, v \in S$, h ein horizontales, v ein vertikales Segment ist.

Fall I (Einfacher Fall): S enthält nur ein Element s.

- $s = (x_1, \text{horizleft}, (x_1, x_2, y))$:
  $L \leftarrow \{(y, (x_1, x_2, y))\}$; $R \leftarrow \emptyset$; $V \leftarrow \emptyset$;
- $s = (x_2, \text{horizright}, (x_1, x_2, y))$:
  $L \leftarrow \emptyset$; $R \leftarrow \{(y, (x_1, x_2, y))\}$; $V \leftarrow \emptyset$;
- $s = (x, \text{vertical}, (x, y_1, y_2))$:
  $L \leftarrow \emptyset$; $R \leftarrow \emptyset$; $V \leftarrow \{([y_1, y_2], (x, y_1, y_2))\}$;

Segmentschnitt-Algorithmus, ReportCuts (Forts.)

Fall II (Rekursion): S enthält mehr als ein Element.

Divide: Wähle eine x-Koordinate x_m, die S in zwei etwa gleich große Teilmengen S_1 und S_2 der Objekte links bzw. rechts von x_m teilt.

Conquer: ReportCuts(S_1, L_1, R_1, V_1); ReportCuts(S_2, L_2, R_2, V_2);

Merge: $LR \leftarrow L_1 \cap R_2$;
diejenigen horizontalen Segmente, deren Endpunkte auf S_1 und S_2 aufgeteilt waren

$L \leftarrow (L_1 - LR) \cup L_2$;
$R \leftarrow R_1 \cup (R_2 - LR)$;
$V \leftarrow V_1 \cup V_2$;

report$((L_1 - LR) \otimes V_2)$; der obige Fall 3.(b): Schnitte zwischen horizontalen Segmenten, die nur mit ihrem linken Endpunkt nur in S_1 repräsentiert waren, und vertikalen Segmenten in S_2

report$((R_2 - LR) \otimes V_1)$; sein symmetrisches Gegenstück

Segmentschnitt-Algorithmus (Forts.)

Die Mengenoperationen $\cup, \cap, -$ können auf (nach y-Koordinaten) sortierten Mengen analog zur merge-Operation in linearer Zeit ausgeführt werden und geben die Sortierung weiter.

Benötigt wird noch folgende **Verknüpfung** zwischen einer Menge Y von horizontalen Segmenten h mit ihren y-Koordinaten und einer Menge I von vertikalen Segmenten v mit ihren y-Intervallen $[y_1, y_2]$:

$$Y \otimes I := \{ (h, v) \mid (y, h) \in Y,\ ([y_1, y_2], v) \in I,\ y \in [y_1, y_2] \}$$

Welche y-Koordinate liegt in welchem y-Intervall?

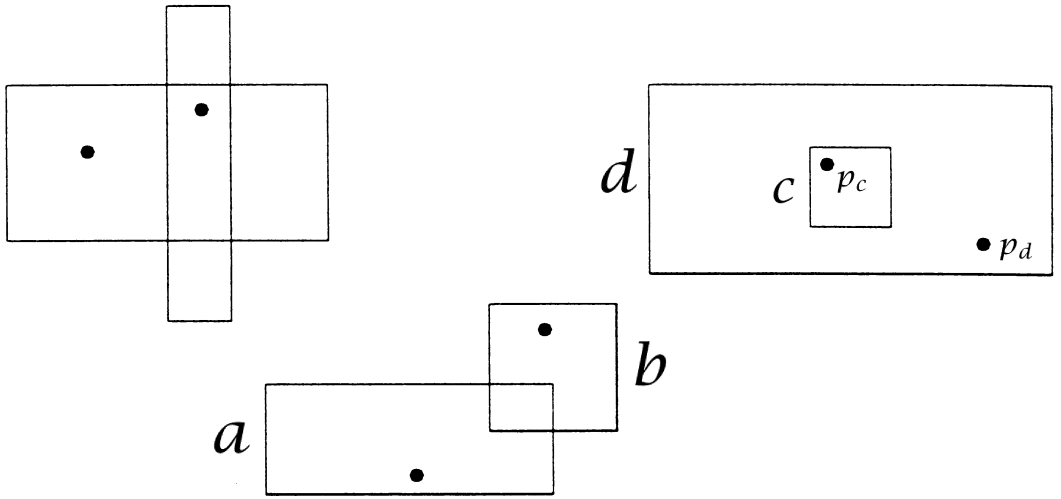
Man kann zeigen, diese Operation in $O(|Y|+|I|+|Y \otimes I|)$ ausgeführt werden kann, wenn Y nach y-Koordinaten und I nach unteren y-Intervallgrenzen sortiert ist. Die Laufzeit von $\otimes$ ist also additiv-linear abhängig von den Eingabegrößen und der Ergebnisgröße.

Segmentschnitt-Algorithmus: Analyse

- Datenstrukturen:
  - für S ein x-sortiertes Array;
    erfordert einmaligen Sortieraufwand $O(n \log n)$;
    erlaubt Divide-Schritt in $O(1)$
  - für L,R y-sortierte Listen; erlauben lineare Mengenoperationen
  - für V eine nach unteren y-Grenzen sortierte Liste; $\otimes$ linear
- Die Laufzeit für ReportCuts kann wie bei Divide+Conquer+Merge–Algorithmen üblich rekursiv als $T(n) = O(1) + 2 \cdot T(\frac{n}{2}) + O(n)$ angegeben werden, da der Merge-Schritt nach obigen Argumenten mit linearem Auswand auskommt.
- Also ergibt sich bekanntlich der Gesamtaufwand $T(n) = O(n \log n)$; hinzu kommt die gesamte Ausgabe der $\otimes$-Operationen: $O(k)$
- **Gesamtlaufzeit:** $O(n \log n + k)$
- Dieser Ansatz lässt sich auf weitere Probleme übertragen.

Weitere DaC-lösbare Probleme

Schnitte zwischen achsenparallelen Rechtecken:



Problemreduktion: Betrachte sich schneidende Rechtecke (a, b), (c, d).

- Fall (1): Kanten von a schneiden Kanten von b

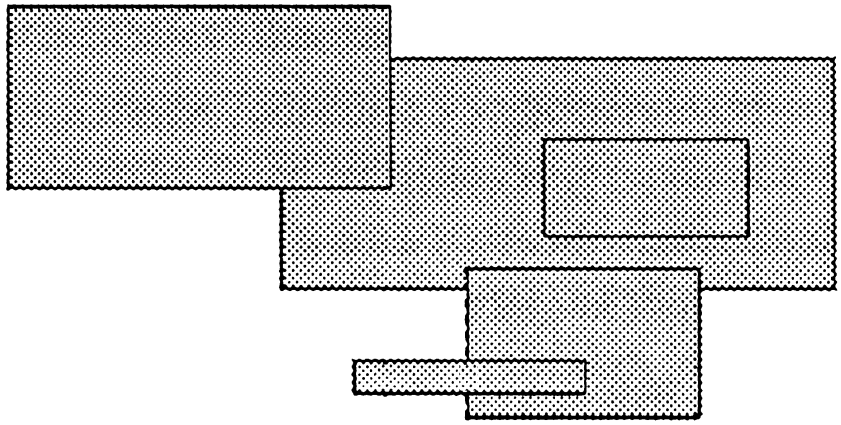
  → Segmentschnitt-Problem

- Fall (2): Ein Rechteck ist im anderen enthalten, z.B. $c \subseteq d$.

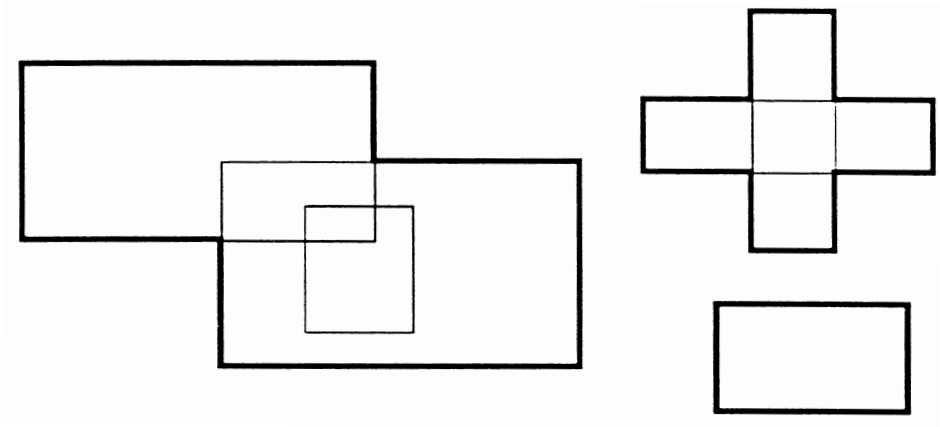
  → Punkteinschluss-Problem: Eine fest gewählte Menge von Repräsentanten-Punkten, für jedes Rechteck c ein Punkt $p_c \in c$, wird darauf geprüft, ob p_c auch in einem anderen Rechteck d liegt. Das deckt alle Fälle (2) und (nochmal) einige Fälle (1) ab. Auch dieses Problem ist mit Divide-and-Conquer und getrennter Repräsentation lösbar.

Weitere DaC-lösbare Probleme: (Forts.)

Maß-Problem: Geg. Menge R von Rechtecken. Bestimme das Gesamtmaß der Rechteckmenge, d.h. ihren Flächeninhalt $\mathsf{area}(R) = \mathsf{area}(\bigcup_{r \in R} r)$.



analog: Kontur-Problem



7.2 Plane-Sweep-Algorithmen

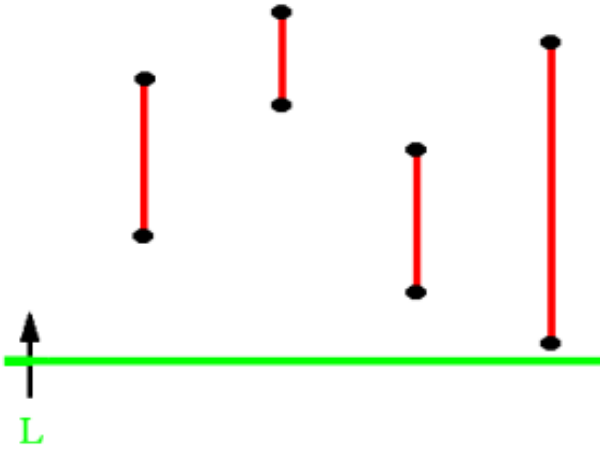
Idee: Die räumliche Situation (in der Ebene, engl. *plane*) wird durch Überstreichen mit einer **Sweepline** (auch: Scanline) L in x- oder y-Richtung beobachtet. An vorbestimmten Haltepunkten ("Ereignissen") werden die dort liegenden, für das Problem relevanten ("aktiven") Objekte festgehalten und ggf. die nächsten Ausgaben daraus ermittelt.

Algorithmusschema Plane-Sweep**:**

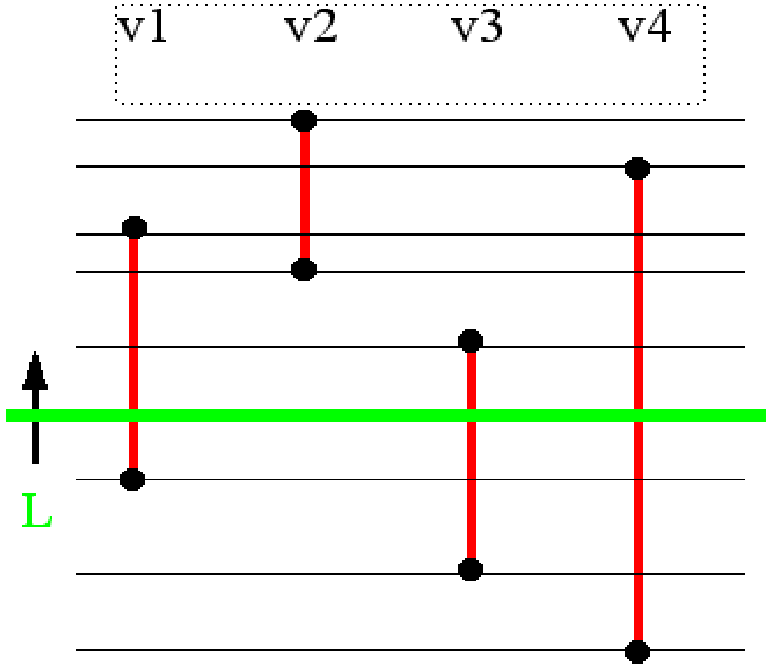
```
// liefert zu einer Menge geometrischer Objekte (zunächst nur
// achsenparallele Objekte) problemabhängige Ausgaben
Q ← objektmengen- und problemabhängiger "Schedule" von Halte-
      punkten ("Ereignissen"), i.w. sortierte x- oder y-Koordinaten;
Status_L ← ∅;
// Status_L enthält immer den aktuellen Status der Sweepline,
// eine (passend strukturierte) Menge der jeweils "aktiven" Objekte
while Q nicht leer do
  wähle nächstes Ereignis event aus Q und entferne es aus Q;
  Aktion: aktualisiere Status_L und/oder
  berechne nächste Teilausgabe aufgrund Status_L und event.
```

Segmentschnitt-Problem:

- Simuliere eine horizontale Sweep Line L, die sich von unten nach oben bewegt. *Status_L* beinhaltet zu jedem Zeitpunkt die Menge aller vertikalen Segmente, die von L geschnitten werden, sortiert von links nach rechts.
- Sobald L den unteren Endpunkt eines vertikalen Segments erreicht, wird dieses in *Status_L* eingefügt. — Das zugehörige "Ereignis" ist die y-Koordinate des Endpunkts mit Angabe des Segments und der Kennzeichnung 'unterer Endpunkt'.
- Sobald L den oberen Endpunkt eines vertikalen Segments erreicht, wird dieses aus *Status_L* gelöscht.



Plane-Sweep, Segment-Schnitt: Verlauf des Status an einem Beispiel



Status_L

( )
(v2)
(v2 v4)
(v1 v2 v4)
(v1 v4)

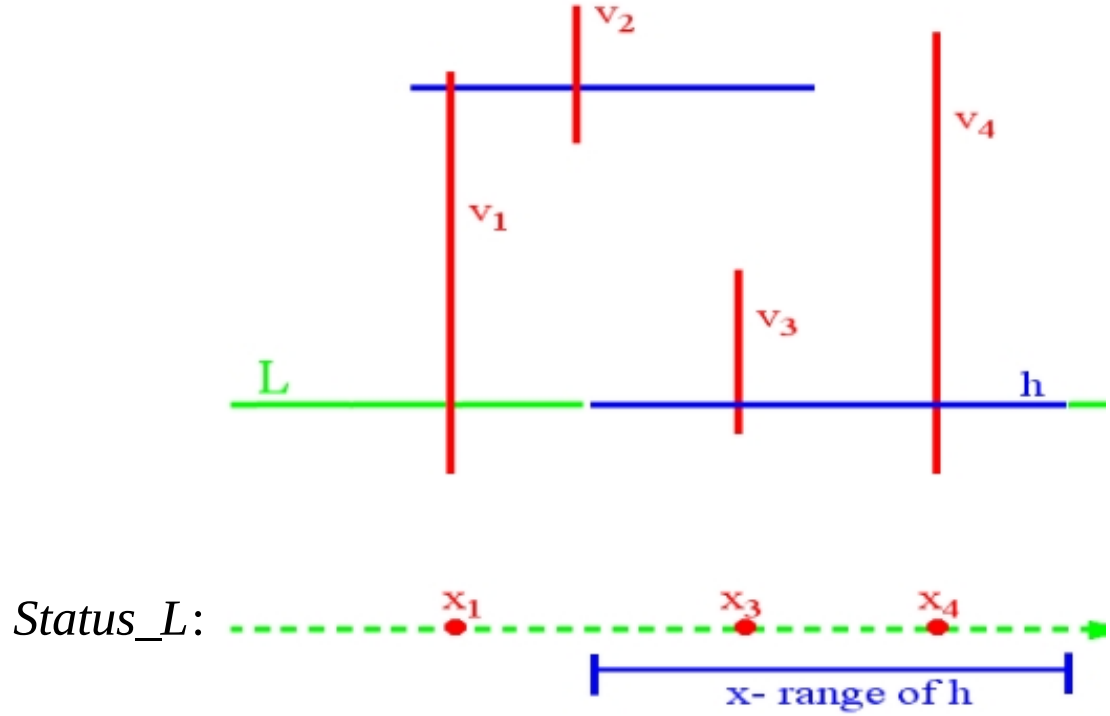
(v1 v3 v4)

(v3 v4)

(v4)
( )

Plane-Sweep, Segment-Schnitt: Kern der Schnittbildung

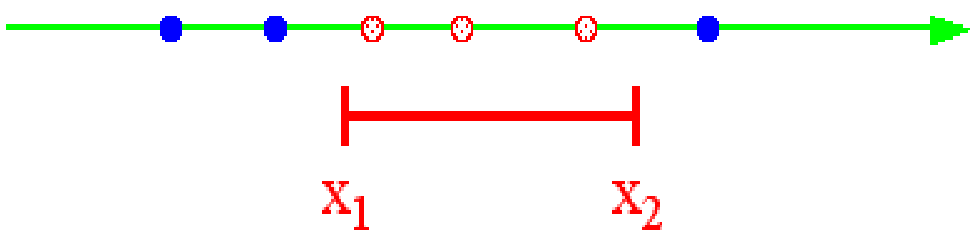
Wenn L ein horizontales Segment h (weiteres Ereignis an einer y-Koordinate!) überstreicht, müssen die schneidenden vertikalen Segmente durch eine *Bereichs-Suche* *(range search)* in *Status_L* gefunden werden.



Plane-Sweep, Segment-Schnitt: Bereichs-Suche

- Gegeben sei eine Menge von x-Koordinaten vertikaler Segmente im *Status_L*. Wir wollen Anfragen der folgenden Art bearbeiten:

  Gib alle x-Werte aus, so dass $x_1 \leq x \leq x_2$ gilt.

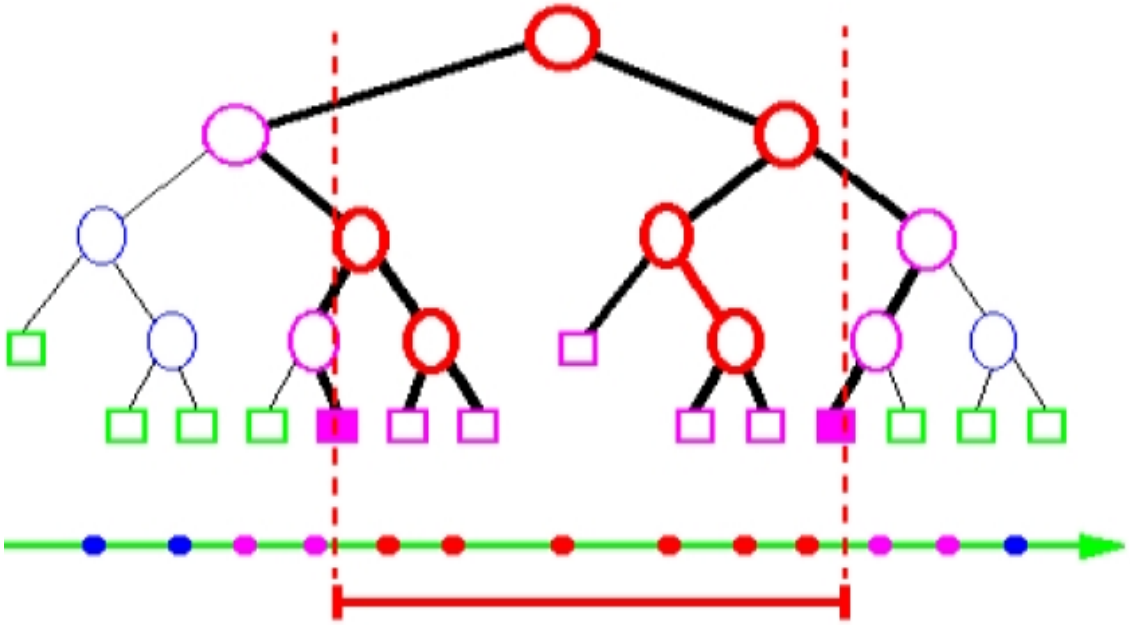


- Wir möchten auch x-Werte einfügen und löschen.
- Darum brauchen wir für *Status_L* eine dynamische Struktur, die folgende Methoden unterstützt:

  insertItem(x,...), removeItem(x), range_search(x_1, x_2)
- x-Werte dienen als Schlüssel; Nutzdaten sind die vertikalen Segmente
- Die Laufzeit sollte optimal sein.
- Deshalb nehmen wir *ausgeglichene Suchbäume*, z. B. AVL-Bäume.

Plane-Sweep, Segment-Schnitt: Bereichs-Suche (Forts.)

- Seien die x-Koordinaten in einem AVL-Baum gespeichert.
- range_search(x_1, x_2): Inorder-Durchlauf, der Teilbäume ignoriert, die *nur* Schlüssel $< x_1$ oder $> x_2$ enthalten.



Die Bereichs-Suche gibt hier die Knoten zurück, die rot markiert sind. Die Suche überprüft auch die violetten Knoten, die aber nicht zurückgegeben werden.

Plane-Sweep, Segment-Schnitt: Bereichs-Suche (Forts.)

- Laufzeit:
  - Sei k die Anzahl der Punkte, die im Suchbereich liegen. Alle k zugehörigen Knoten werden besucht.
  - Zusätzlich werden höchstens ca. $2 \log n$ Knoten (Außenpfade) plus k+1 (externe) Knoten besucht, zu denen nichts zurückgegeben wird.
  - also insgesamt: $O(\log n + k)$

Plane-Sweep, Segment-Schnitt: Steuerung des Plane-Sweeps (Forts.)

Der Status der Sweepline L wird in diskreten Schritten, quasi angestossen durch bestimmte "Ereignisse", aktualisiert oder ausgewertet — das ist der Kern des Plane-Sweep-Algorithmusschemas. Es müssen nur noch die Ereignisse und Aktionen festgelegt werden:

- Ereignis: L trifft auf den unteren Endpunkt eines vertikalen Segments v mit x-Koordinate x:
  - Aktion: $Status\_L$.insertItem(x, v)
- Ereignis: L trifft auf den oberen Endpunkt eines vertikalen Segments v mit x-Koordinate x:
  - Aktion: $Status\_L$.removeItem(x)
- Ereignis: L trifft auf ein horiz. Segment h mit x-Grenzen x_1, x_2:
  - Aktion: $Status\_L$.range_search(x_1, x_2)

Plane-Sweep, Segment-Schnitt: Datenstrukturen (Forts.)

- *Status_L*:
  - *wie oben begründet:*
    AVL-Baum für vertik. Segmente mit x-Koordinaten als Schlüssel
- Ereignis-Schedule Q:
  - soll horiz. Segmente und Endpunkte vertik. Segmente speichern
  - soll von unten nach oben gelesen werden

  $\Rightarrow$ sortierte Sequenz mit solchen Objekten (versehen mit passenden Schlüsseln, s.u.) als Items
  - Bei gleichen y-Koordinaten muss die folgende Sortierreihenfolge der Ereignisse beachtet werden *(warum?)*:
    1. "unterer Endpunkt" ($\rightsquigarrow$ insertItem),
    2. "horizontales Segment" ($\rightsquigarrow$ range_search),
    3. "oberer Endpunkt" ($\rightsquigarrow$ deleteItem).

  $\Rightarrow$ *Schlüssel* = Kombination (y-Koordinate, Kennung bot|hseg|top)
  - Q muss anfangs sortiert werden

Plane-Sweep, Segment-Schnitt: Laufzeit (Forts.)

- anfangs Sortierung von $2|V|+|H| \approx \frac{3}{2}n$ Ereignissen: $O(n \log n)$
- dann Abarbeitung dieser Ereignisse:
  - untere Endpunkte:
    - Anzahl der Vorkommen: $|V| \approx \frac{1}{2}n$
    - Aktion: insertItem
    - Laufzeit für jedes insertItem, da im AVL-Baum: $O(\log n)$
  - obere Endpunkte:
    - analoge Analyse mit removeItem
  - horizontale Segmente h:
    - Anzahl der Vorkommen: $|H| \approx \frac{1}{2}n$
    - Aktion: range_search
    - Laufzeit dafür: $O(\log n + k_h)$
      mit k_h = # vertikale Segmente, die h schneiden.
- Gesamt-Laufzeit: $O(n \log n + \sum_h k_h) = O(n \log n + k)$ *[wie mit Div.and Conq.!]*

Fazit (für Algorithmen auf Mengen achsenparalleler Objekten)

- Plane-Sweep reduziert ein zweidimensionales Mengenproblem auf ein eindimensionales dynamisches Suchproblem.

  Die Reduktion ist relativ einfach, aber u. U. sind die Datenstrukturen komplex; z.B. erfordern Punkteinschluss-/Maß-Probleme spezielle Suchbäume für Intervallmengen.

- Divide-and-Conquer reduziert ein zweidimensionales Mengenproblem auf eindimensionale Mengenprobleme.

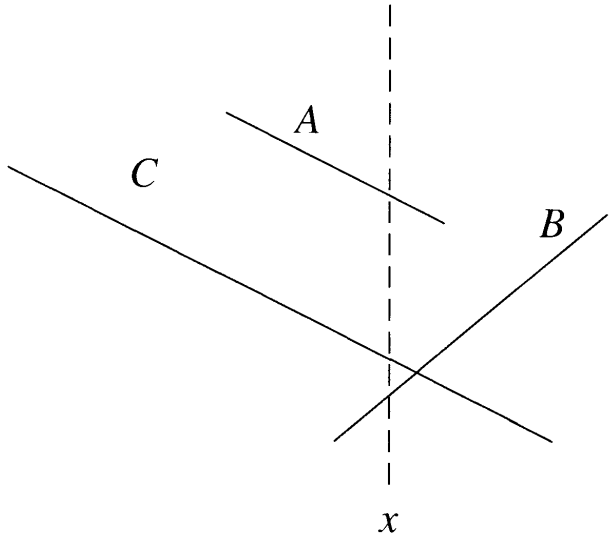
  Die Reduktion ist komplexer, aber die Datenstrukturen sind oft einfache Listen, die auch Umsetzungen auf externe Speicher erlauben.

- Gleiche Laufzeitklassen, aber nur Plane-Sweep nutzbar für nicht-achsenparallele Objekte.

7.3 Ein Plane-Sweep-Algorithmus auf beliebig-orientierten Objekten

Als Beispiel sehen wir uns das Problem an, alle Schnittpunkte in einer Menge von beliebig-orientierten Segmenten[4] zu ermitteln.

Wir verfolgen eine Sweepline L von links nach rechts.



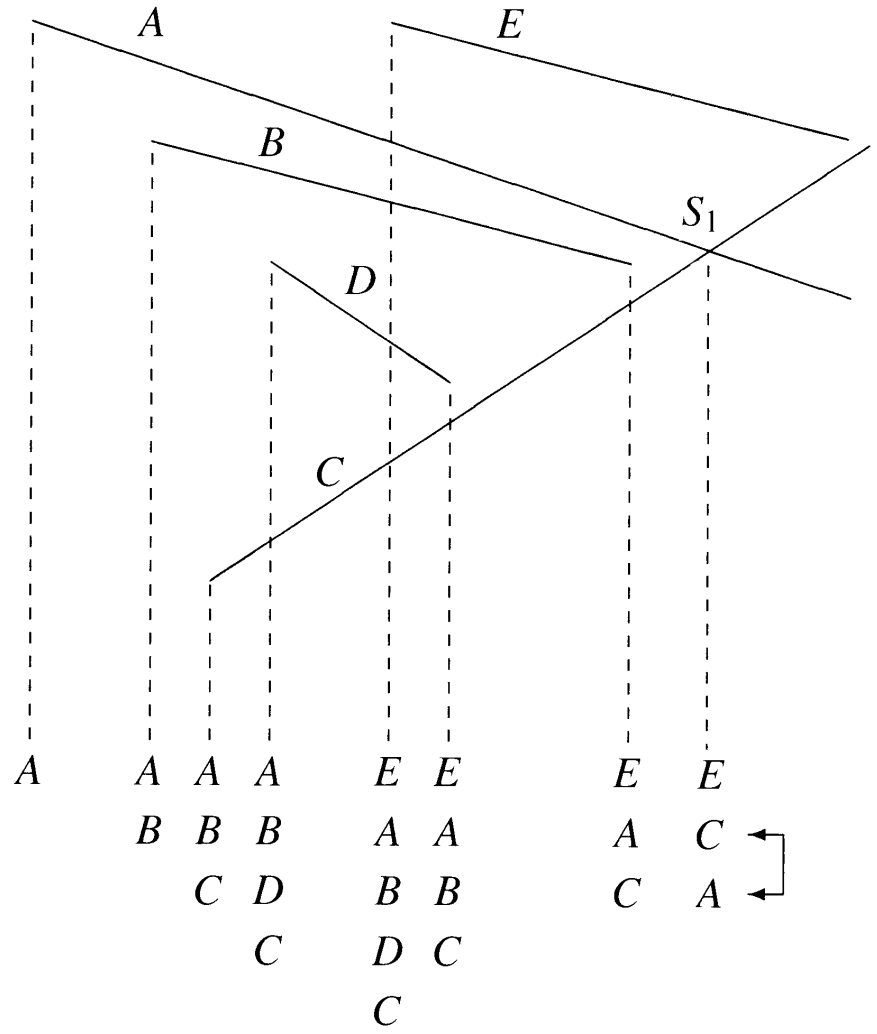
In *Status_L* beobachtet werden die Segmente, die aktuell die Sweepline schneiden, sortiert nach ihrer Lage übereinander; im Bild:

B unter C unter A

[4]Wieder seien keine überlappenden, nur disjunkte oder sich schneidende Segmente zugelassen. Auch vertikale Segmente würden eine Sonderbehandlung erfordern.

Allgemeines Segmentschnitt-Problem (Forts.)

Die Statusmenge kann sich an linken und rechten Endpunkten von Segmenten als Ereignissen ändern, die Anordnung der Segmente ausgerechnet an den zu berechnenden Schnittpunkten !



Allgemeines Segmentschnitt-Problem (Forts.)

Der **Ereignis-Schedule** Q muss also ***dynamisch*** um die nächsten Schnittpunkte rechts von der Sweepline ergänzt werden. Diese können nur von vorher benachbarten Segmenten produziert werden !

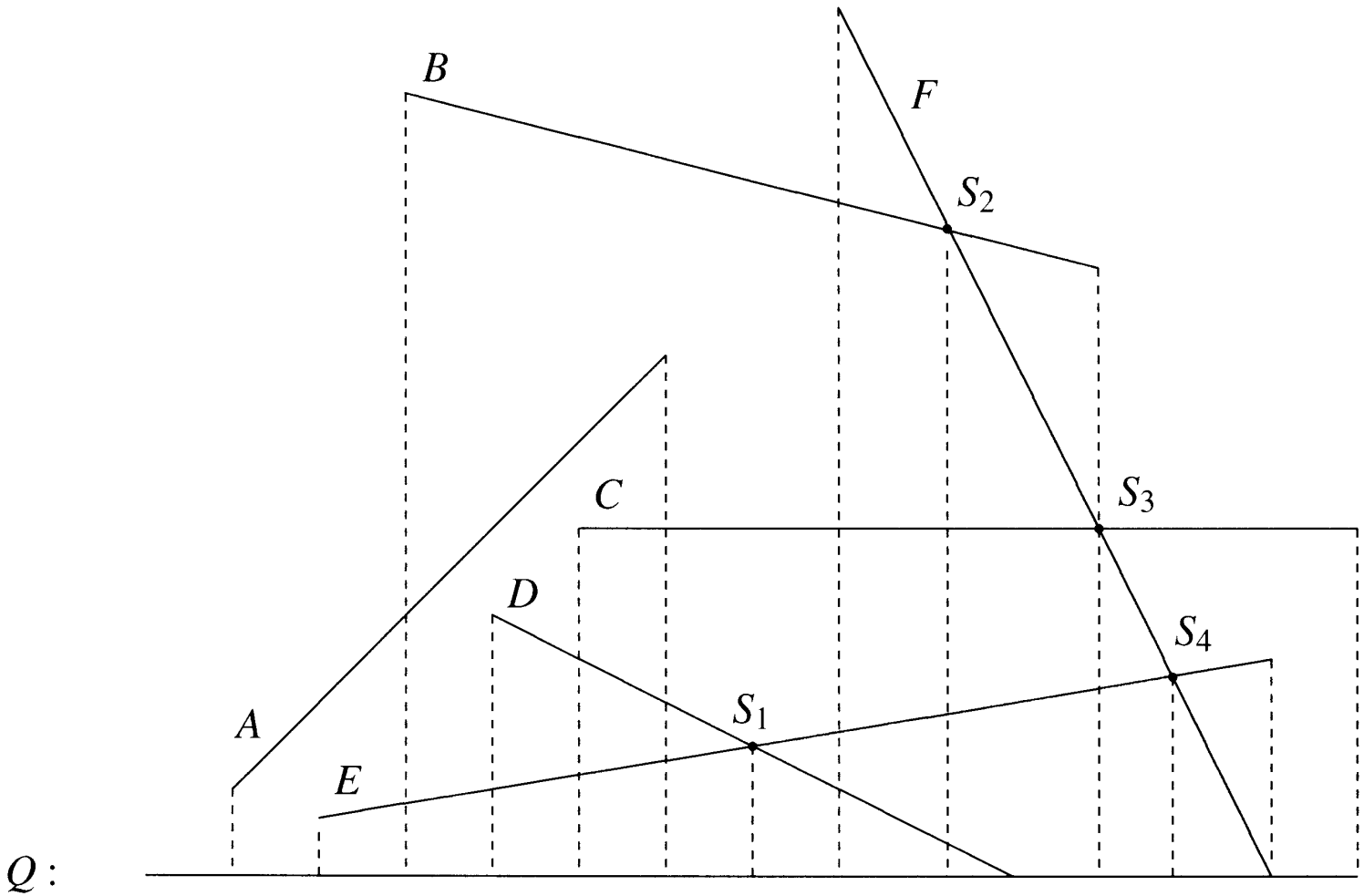
Wann immer sich

- durch das Einfügen eines Segments an seinem linken Endpunkt,
- durch das Löschen eines Segments an seinem rechten Endpunkt
- oder durch das Vertauschen zweier Segmente an deren Schnittpunkt

Nachbarschaften in *Status_L* ändern, sind für die neu benachbarten Segmente deren Schnittpunkte zu berechnen (falls sie sich schneiden) und in Q einzuordnen.

Für Q sollte deshalb eine Priority-Queue verwendet werden:

- Objekte: Segmente und Schnittpunkte (mit den zugeh. Segmenten)
- Schlüssel: x- und y-Koordinate des End- oder Schnittpunkts, deshalb mit Kennung left|right|cut,
- sortiert nach x-, dann nach y-Koordinate, dann right $<$ cut $<$ left.



L :	$\emptyset$	A	A	B	B	B	B	B	F	B	B	C	C	C	$\emptyset$
			E	A	A	A	C	C	B	F	F	F	E		
				E	D	C	D	E	C	C	C	E	F		
					E	D	E	D	E	E	E				
						E			D	D					

Allgemeines Segmentschnitt-Problem: Pseudocode-Algorithmus

algorithm GeneralSegmentIntersection_By_PlaneSweep(S) **:**

// liefert zu einer Menge $S=\{s_1, ..., s_n\}$ von Liniensegmenten
// in der Ebene alle verschiedenen Paare (s_i, s_j), die sich schneiden[5]

$Q \leftarrow$ Priority-Queue von Haltepunkten p, initialisiert mit den linken und rechten Endpunkten von Segmenten in S;
Status_L $\leftarrow \emptyset$; // Menge der jeweils aktiven Segmente
while not Q.isEmpty() **do**
  Punkt $p \leftarrow Q$.removeMin();

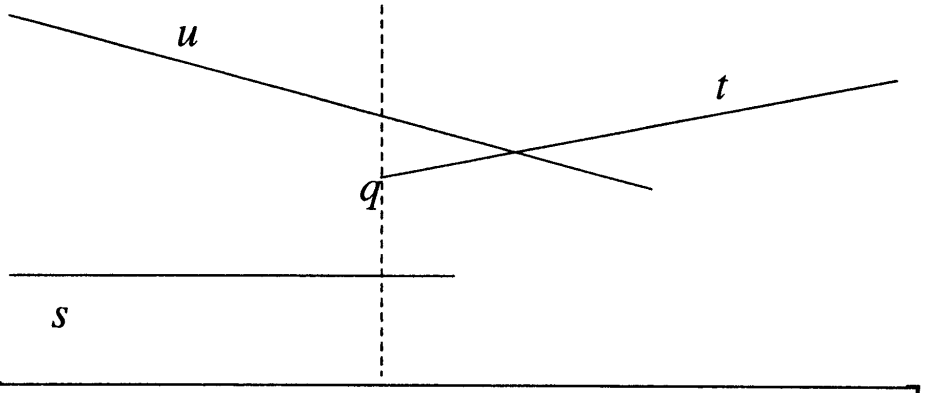
  if p ist linker Endpunkt eines Segments s **then**
    Status_L.insertItem(..., s);
    Segment s' $\leftarrow$ oberer Nachbar von s in *Status_L*;
    Segment s" $\leftarrow$ unterer Nachbar von s in *Status_L*;
    if $s \cap s' \neq \emptyset$ **then** Q.insertItem($s \cap s'$);
    if $s \cap s'' \neq \emptyset$ **then** Q.insertItem($s \cap s''$);

[5] Kurznotation: $s \cap s' \neq \emptyset$ bedeutet: Segment s schneidet Segment s'; $s \cap s'$ bezeichnet den Schnittpunkt von s,s'.

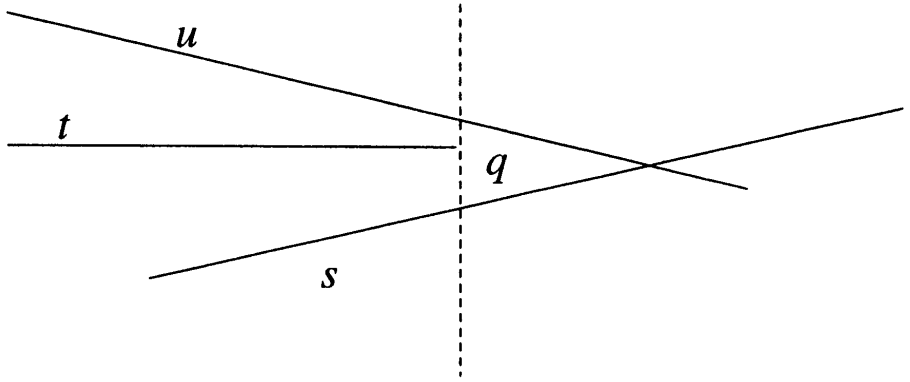
Allgemeines Segmentschnitt-Problem: Pseudocode-Algorithmus (Forts.)

else if p ist rechter Endpunkt eines Segments s **then**
  Segment s' ← oberer Nachbar von s in *Status_L*;
  Segment s'' ← unterer Nachbar von s in *Status_L*;
  if $s \cap s' \neq \emptyset$ **then** Q.insertItem$(s \cap s')$;
  if $s \cap s'' \neq \emptyset$ **then** Q.insertItem$(s \cap s'')$;
  Status_L.remove(s);

else // p ist Schnittpunkt von s' und s'', d.h. $p = s' \cap s''$,
  // und es sei s' oberhalb von s'' in *Status_L*.
  gib das Paar (s', s'') mit Schnittpunkt p aus;
  vertausche s' und s'' in *Status_L*;
  Segment t' ← unterer Nachbar von s' in *Status_L*;
  Segment t'' ← oberer Nachbar von s'' in *Status_L*;
  if $s' \cap t' \neq \emptyset$ **then** Q.insertItem$(s' \cap t')$;
  if $s'' \cap t'' \neq \emptyset$ **then** Q.insertItem$(s'' \cap t'')$.



vorher	*nachher*
$Q = <t_l, s_r, u_r, t_r>$	$Q = <s_r, (t \cap u), u_r, t_r>$
$Status\_L = <s, u>$	$Status\_L = <s, t, u>$



vorher	*nachher*
$Q = <t_r, u_r, s_r>$	$Q = <(s \cap u), u_r, s_r>$
$Status\_L = <s, t, u>$	$Status\_L = <s, u>$

Allgemeines Segmentschnitt-Problem: Laufzeit

- Sei $n = \#\text{Segmente}$, $k = \#\text{Schnittpunkte}$.
- Die Priority-Queue Q enthält max. $2n+k$ Items (End-/Schnittpunkte).
- Jede Operation darauf läuft mit: $O(\log(2n+k)) = O(\log n)$
  da $\log(2n+k) \leq \log(2n+n^2) = O(\log n^2) = O(\log n)$
- Für *Status_L* verwende AVL-Baum mit Bestimmung von Nachbarn und Vertauschung; max. n Items (Segmente in 'unter'-Ordnung).
- Jede Operation darauf ist mit $O(\log n)$ implementierbar.
- $2n+k$ Schleifendurchläufe

$\Rightarrow$ Gesamtlaufzeit: $O((n+k)\log n)$

- besser als naives Verfahren für nicht zu große k,
  sogar noch verbesserbar zu $O(n \log n + k)$