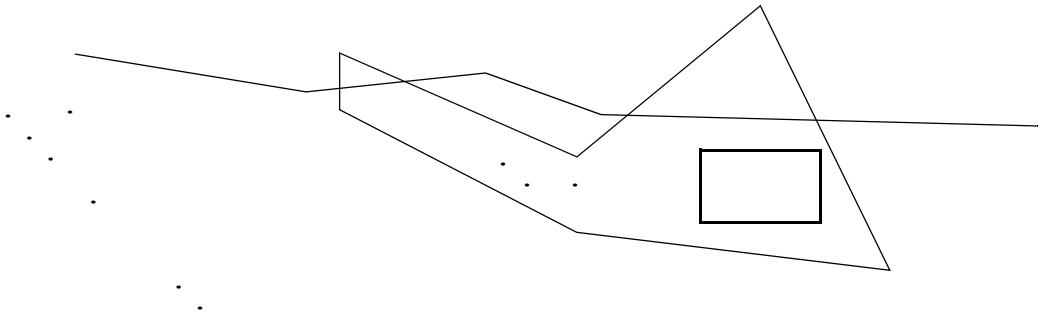
Kapitel 6 Ausgewählte Themen

6.1 Geometrische Algorithmen

Geometrische Objekte sind Objekte mit Lage und Ausdehnung in einem multidimensionalen Raum. Sie treten z.B. in Geo-Informationssystemen, beim Entwurf hochintegrierter Schaltungen (VLSI) oder in der Computer-Graphik auf. Der Begriff des geometrischen Objekts wird im folgenden präzisiert.

Ein **k-dimensionales geometrisches Objekt** ist eine zusammenhängende Teilmenge des $\mathbb{R}^k$, d.h. zwei Punkte des Objekts lassen sich immer durch eine Kurve innerhalb des Objekts miteinander verbinden.

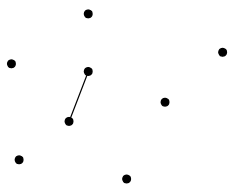
Beispiel



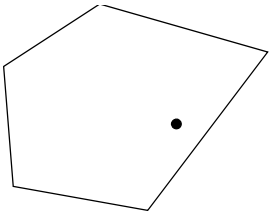
$k = 2$

In dem Gebiet der **algorithmischen Geometrie** werden effiziente Algorithmen für zahlreiche geometrische Probleme entwickelt, z.B. für

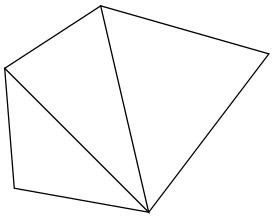
- Distanzprobleme,
- Inklusionstest und -berechnungen,
- Zerlegungen von Objekten in Primitive,
- Berechnung der konvexen Hülle.



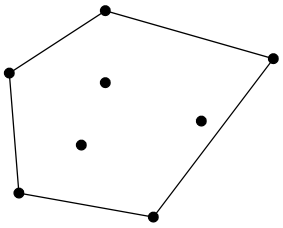
minimaler Abstand



Inklusionstest



Zerlegung in Dreiecke

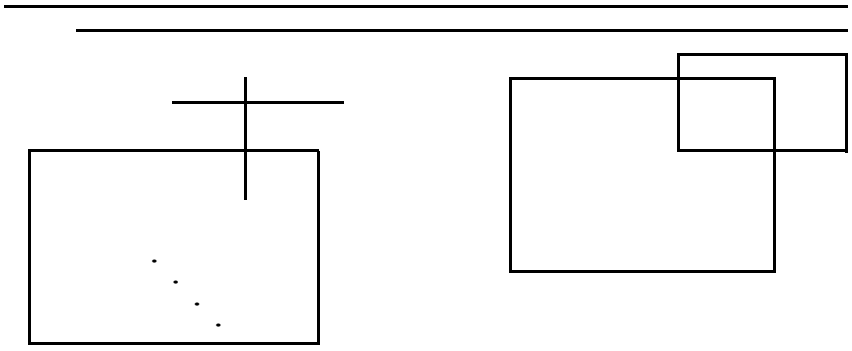


konvexe Hülle

Wir beschränken uns hier auf den wichtigen Spezialfall orthogonaler geometrischer Objekte. Ein k-dimensionales geometrisches Objekt heißt **orthogonal**, wenn es als kartesisches Produkt von k Intervallen beschrieben werden kann. Die Intervalle dürfen dabei zu Punkten entarten.

Beispiele

- Geo-Informationssysteme: im Filterschritt der Anfragebearbeitung werden minimal umgebende Rechtecke zur Approximation der komplexen geometrischen Objekte verwendet.
- Entwurf von VLSI-Schaltungen: vielfach dürfen die Schaltelemente aufgrund von Beschränkungen des Produktionsprozesses nicht beliebig, sondern nur orthogonal orientiert werden.



Wir werden an Hand *zwei-dimensionaler orthogonaler Objekte* die wichtigsten Paradigmen geometrischer Algorithmen behandeln:

- Plane-Sweep
- geometrisches Divide-and-Conquer.

Die vorgestellten Algorithmen lassen sich sowohl für drei Dimensionen als auch für beliebig orientierte Objekte verallgemeinern.

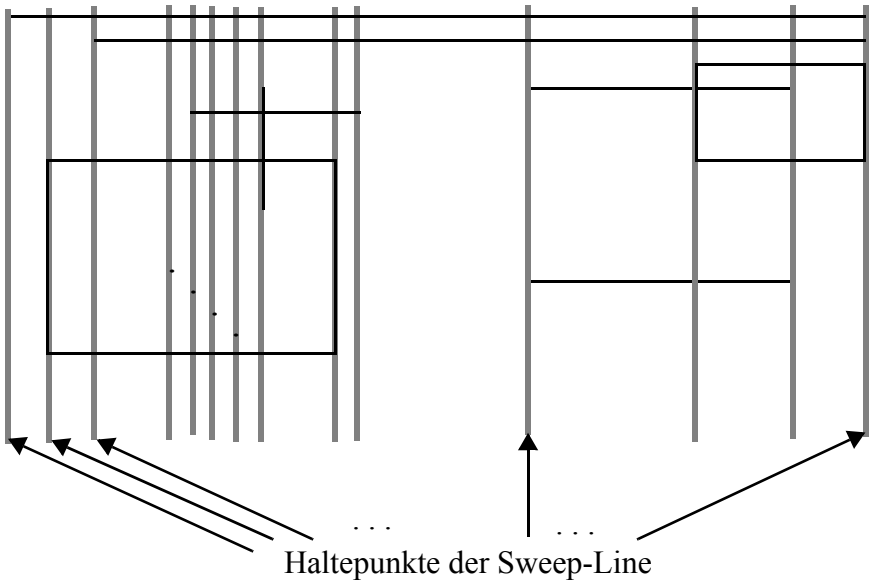
6.1.1 Plane-Sweep-Algorithmen

Das Plane-Sweep-Paradigma ist ein natürlicher Ansatz, um Probleme auf Mengen geometrischer Objekte zu lösen.

Idee

- Eine vertikale (oder horizontale) Gerade (**Sweep-Line**) wird von links nach rechts (bzw. von oben nach unten) über die Ebene geschoben.
- Dabei wird der Schnitt der Sweep-Line mit der Menge der Objekte beobachtet. Für die Objekte, die die Sweep-Line gerade schneiden (**aktive Objekte**), werden geeignete Aktionen durchgeführt.

Die kontinuierliche Bewegung der Geraden über die Ebene lässt sich folgendermaßen diskretisieren. Man berechnet die sogenannten **Haltepunkte** der Sweep-Line, d.h. die Positionen, bei denen sich die Menge der aktiven Objekte oder ihre Sortierreihenfolge ändert. Die Haltepunkte werden aufsteigend sortiert, und die Sweep-Line "springt" von Haltepunkt zu Haltepunkt über die Ebene.



Datenstrukturen

- **Event-Point-Schedule**
  Haltepunkte in aufsteigender Reihenfolge
- **Sweep-Line-Status**
  eine Teilmenge der momentan aktiven Objekte

Während der Event-Point-Schedule bei der Beschränkung auf orthogonale Objekte meist statisch ist (weil sich die Sortierreihenfolge der aktiven Objekte nicht ändert), ist der Sweep-Line-Status immer eine dynamische Datenstruktur.

Algorithmus

```
ALGORITHM Simple-Plane-Sweep;
    Initialisiere Event-Point-Schedule;
    Sweep-Line-Status:= {};
    FOR EACH Haltepunkt IN Event-Point-Schedule DO
        aktualisiere Sweep-Line-Status;
        verarbeite die neuen aktiven Objekte und gebe Teilantworten aus;
```

Rechteckschnitt-Problem

Gegeben ist eine Menge R von Rechtecken, gesucht ist die Menge aller sich schneidenden Paare (r_1,r_2), $r_1,r_2 \in R$.

Dieses Problem tritt z.B. bei der Bearbeitung eines Spatial Join (Geo-Informationssysteme) oder beim Finden überlappender Schaltelemente (Entwurf von VLSI-Schaltungen) auf.

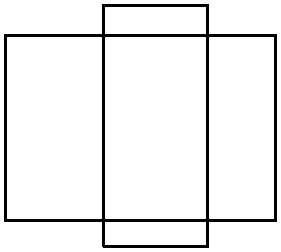
Schnitte können in zwei Fällen auftreten:

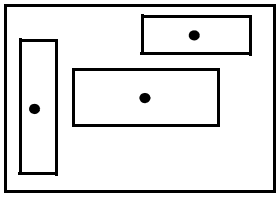
- Fall1: Ein Kantenpaar schneidet sich.

  ➙ Schnitt einer Menge von orthogonalen Strecken (Segmentschnitt-Problem)

- Fall 2: Ein Rechteck liegt vollständig innerhalb des anderen.

  ➙ Schnitt einer Menge von Punkten und einer Menge von Rechtecken (Punkteinschluss-Problem)

(1) 

(2) 

Segmentschnitt-Problem Punkteinschluss-Problem

Segmentschnitt-Problem

Gegeben

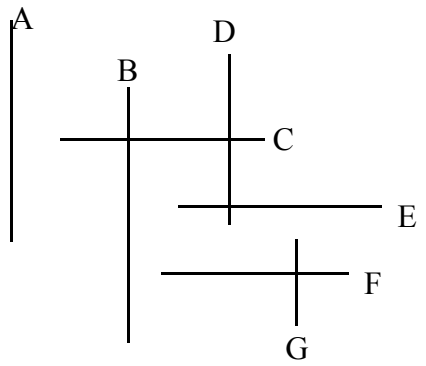
Menge von horizontalen bzw. vertikalen *(orthogonalen)* Strecken.

Gesucht

Alle Paare sich schneidender Strecken.

Annahme

Die Endpunkte verschiedener Strecken besitzen jeweils unterschiedliche x- und y-Koordinaten, d.h. es sind nur Schnitte zwischen je einer horizontalen und einer vertikalen Strecke möglich.



Antworten:
(B,C)
(C,D)
(D,E)
(F,G)

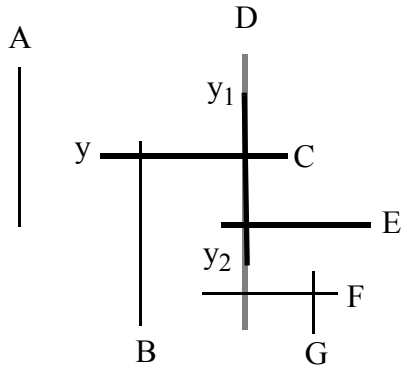
Der naive Algorithmus testet alle Paare von Strecken auf Schnitt. Seine Laufzeit ist $O(n^2)$ für n orthogonale Strecken.

Der Plane-Sweep-Algorithmus basiert auf folgenden Beobachtungen:

- Horizontale Strecken schneiden die Sweep Line in einer festen y-Koordinate.
- Vertikale Strecken schneiden die Sweep Line in einem y-Intervall $[y_1, y_2]$. Horizontale Strecken, die diese vertikale Strecke schneiden, erfüllen folgende Bedingungen:

  (1) Sie schneiden auch die Sweep-Line.

  (2) Ihre y-Koordinate liegt im Intervall $[y_1, y_2]$.



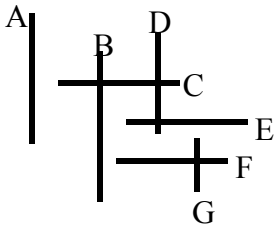
Datenstrukturen

- *Event-Point-Schedule*: enthält die x-Koordinate jeder vertikalen Strecke und die min. und max. x-Koordinaten jeder horizontalen Strecke (sortiert).
- *Sweep-Line-Status*: enthält alle horizontalen Strecken, die die Sweep Line momentan schneiden.

Algorithmus

```
Antworten:= {};
FOR EACH s.x IN Event-Point-Schedule DO
    IF s ist horizontales Segment THEN
        IF x ist linker Punkt von Segment s THEN
            füge s in Sweep-Line-Status ein;
        ELSIF x ist rechter Punkt von Segment s THEN
            entferne s aus Sweep-Line-Status;
    ELSIF s ist vertikales Segment THEN
        Strecken := Sweep-Line-Status.range-query (s.y1 ≤ t.y ≤ s.y2);
        FOR EACH t IN Strecken DO
            Antworten:= Antworten ∪{(s,t)};
RETURN Antworten;
```

Beispiel

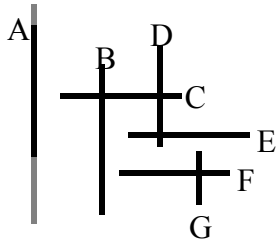


Initialisierung

Event Point Schedule =

A.x, C.x1, B.x, F.x1, E.x1, D.x, C.x2, G.x, F.x2, E.x2

Sweep Line Status = {}

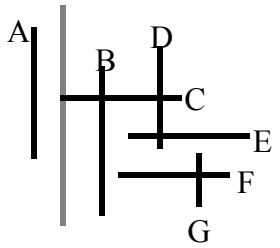


Haltepunkt: A.x

Aktion: Finde alle I ∈ Status
mit A.y1 ≤ I.y ≤ A.y2

Sweep Line Status = {}

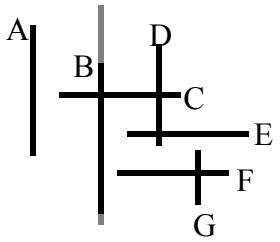
keine Antwort



Haltepunkt: C.x1

Aktion: Füge C in Status ein

Sweep Line Status = {C}



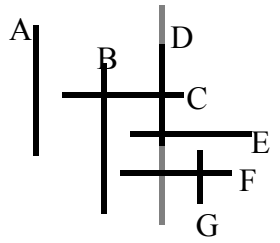
Haltepunkt: B.x

Aktion: Finde alle I ∈ Status
mit B.y1 ≤ I.y ≤ B.y2

Sweep Line Status = {C} ⇒ I = C

Antwort: (B,C)

weitere Haltepunkte . . .

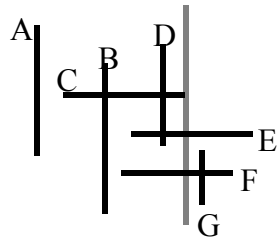


Haltepunkt: D.x

Aktion: Finde alle I ∈ Status
mit D.y1 ≤ I.y ≤ D.y2

Sweep Line Status = {C,E,F} ⇒ I ∈ {C,E}

Antworten: (D,C), (D,E)



Haltepunkt: C.x2

Aktion: Lösche C aus Status

Sweep Line Status = {E,F}

weitere Haltepunkte . . .

Laufzeit

Bezeichne n die Anzahl der orthogonalen Strecken, k die Anzahl der auftretenden Schnitte.

Zum Abspeichern des Sweep-Line-Status wird ein höhenbalancierter Baum (z.B. ein AVL-Baum oder ein B-Baum) mit folgenden Modifikationen verwendet:

- Alle Schlüssel befinden sich in den Blättern.
- Die Blätter sind entsprechend der Sortierreihenfolge der Schlüssel miteinander verkettet.

Einfügen und Löschen im Sweep-Line-Status benötigt also O(log n) Zeit, eine Bereichsanfrage (mit r Antworten) benötigt O(log n + r) Zeit.

Berechnung der Laufzeitkomplexität

- Aufbau des Event-Point-Schedule: O(n log n)
- Durchlauf der Sweep-Line:

  alle Einfügungen in Sweep-Line-Status: O(n log n)

  alle Löschungen aus Sweep-Line-Status: O(n log n)

  alle Bereichsanfragen: O(n log n + k)

  ➙ Gesamtkosten: O(n log n + k)

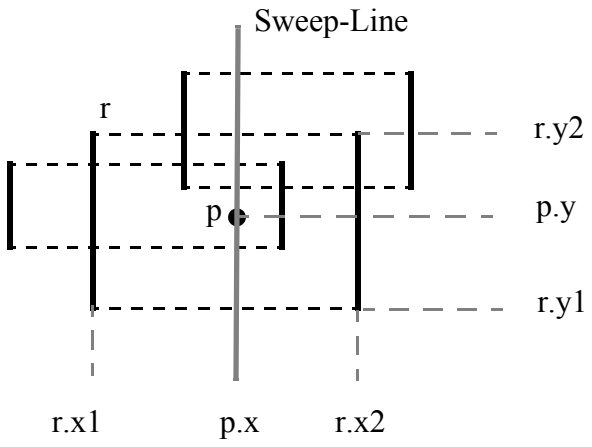
Punkteinschluss-Problem

Gegeben

Eine Menge P von Punkten und eine Menge R von Rechtecken.

Gesucht

Alle Paare (p,r) mit $p \in P$, $r \in R$, wobei p in r liegt.



Die Plane-Sweep-Lösung basiert auf folgenden Überlegungen:

- (p,r) kann nur eine Antwort sein, wenn r.x1 ≤ p.x ≤ r.x2 , d.h. p und r müssen gleichzeitig die Sweep-Line schneiden.
- Zusätzlich muss für (p,r) gelten: r.y1 ≤ p.y ≤ r.y2 .

Datenstrukturen

Der Event-Point-Schedule setzt sich zusammen aus den x-Koordinaten der Punkte und den x-Koordinaten der vertikalen Kanten (aufsteigend sortiert).

Im Sweep-Line-Status werden die aktiven Rechtecke verwaltet, d.h. Schlüssel der Form $[y_1, y_2]$. Für die Implementierung des Sweep-Line-Status sind also bei diesem Problem die (eindimensionalen) Suchbäume wie AVL-Bäume oder B-Bäume nicht geeignet. Geeignete mehrdimensionale Datenstrukturen werden weiter unten eingeführt.

Skizze des Algorithmus

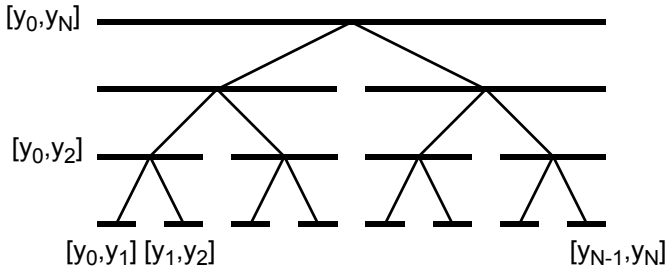
- "Linke Kanten" werden in den Sweep-Line-Status eingefügt.
- "Rechte Kanten" werden aus dem Sweep-Line-Status entfernt.
- Für einen Punkt p des Event-Point-Schedule müssen alle Kanten im Sweep-Line-Status bestimmt werden, deren y-Intervall die y-Koordinate von p enthält.

Segmentbaum

Der Segmentbaum ist eine Datenstruktur zur effizienten Implementierung des Sweep-Line-Status beim Punkteinschluss-Problem.

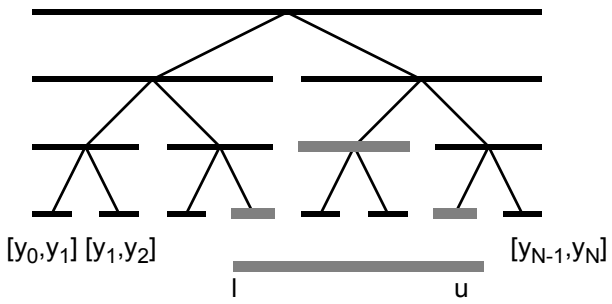
Sei $y_0,\ldots y_N$ eine Menge von Punkten im 1-dimensionalen Datenraum mit $y_{i-1} < y_i$ für $1 \le i \le N$. Der **Segmentbaum** für diese Menge ist ein binärer Baum mit minimaler Höhe, der folgende Eigenschaften besitzt:

- Jedem Blattknoten ist ein Segment $[y_{i-1}, y_i]$ zugeordnet.
- Jedem inneren Knoten ist die Vereinigung der Segmente seines Teilbaums zugeordnet.
- Jeder Knoten besitzt zusätzlich eine Liste für abzuspeichernde Intervalle.



Einfügen eines Intervalls

Ein Intervall $i = [l,u]$ wird in der Liste eines Knotens abgelegt, falls i das zugehörige Segment des Knotens überdeckt, jedoch das Segment des Vaterknotens **nicht** überdeckt.



Punktanfrage

- Finde zu einem gegebenen Punkt $p \in [y_0, y_N]$ alle Intervalle i, die p überdecken.
- Bestimme den Pfad von der Wurzel zu einem Blatt, so dass das zugeordnete Segment des Knotens stets p enthält. Die Antwortmenge der Anfrage ist die Vereinigung der Intervalllisten aller Knoten dieses Pfades.

Komplexitätsanalyse des Segmentbaums

- Aufbau des Segmentbaums mit leeren Intervalllisten:

  O(N) Laufzeit und O(N) Speicherplatz

- Einfügen eines Intervalls in den Baum:
  - Maximal zwei Knoten auf jeder Stufe des Baums sind durch das Einfügen betroffen.
  - Einfügen in eine (unsortierte) Intervallliste benötigt O(1) Laufzeit.

    ➞ O(log N) Laufzeit und O(log N) Speicherplatz

- Löschen aus dem Baum:

  analog zu Einfügen, d.h. O(log N) Laufzeit

- Punktanfrage:
  - O(log $N + r$) Laufzeit, wenn r die Größe der Antwortmenge ist

- Gesamtspeicherplatzbedarf (für den leeren Segmentbaum und die gefüllten Intervalllisten):

  O($N + N$ log N) = O(N log N)

Komplexitätsanalyse des Plane-Sweep-Algorithmus für das Punkteinschluss-Problem

Im folgenden analysieren wir die Komplexität des obigen Plane-Sweep-Algorithmus für den Fall, dass der Sweep-Line-Status durch einen Segmentbaum implementiert wird.

Bezeichne n_r die Anzahl der Rechtecke, n_p die Anzahl der Punkte und sei $n = n_r + n_p$. Bezeichne ferner k die Anzahl der Antworten, d.h. die Zahl der gefundenen Paare (Punkt,Rechteck).

- Sortierung der Haltepunkte (pro Rechteck 2, pro Punkt 1):

  $O(n \log n)$ Laufzeit

- Aufbau des leeren Segmentbaums:

  Berechne die Segmentstruktur aus den y-Koordinaten der Rechtecke, sortiere diese Koordinaten und erzeuge den leeren Segmentbaum:

  $O(n_r \log n_r) = O(n \log n)$ Laufzeit

- Alle Einfügungen und Löschungen im Segmentbaum: $O(n_r \log n_r)$ Laufzeit
- Alle Punktanfragen an den Segmentbaum: $O(n_p \log n_r + k)$ Laufzeit

Gesamtkosten:

→ $O(n \log n + k)$ Laufzeit

→ $O(n \log n)$ Speicherplatz

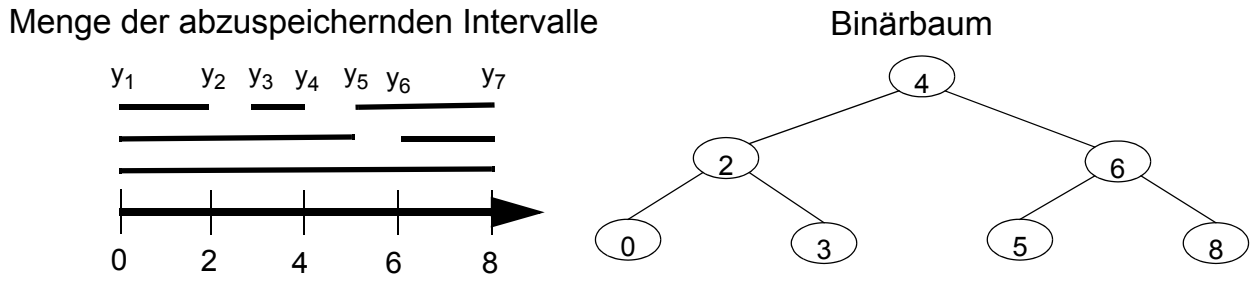
Ein Nachteil dieser Lösung ist der relativ hohe Speicherplatzaufwand von $O(n \log n)$. Deshalb behandeln wir im folgenden eine alternative Datenstruktur für den Sweep-Line-Status beim Punkteinschluss-Problem, die den Speicherplatzaufwand auf $O(n)$ reduziert.

Intervallbaum

Im Unterschied zum Segmentbaum repräsentieren Knoten des Baums nicht ein Segment, sondern einen Punkt.

Aufbau eines Binärbaums

- Sei $I_1, I_2, \ldots, I_n$ eine Menge von Intervallen mit $I_i = [u_i,o_i]$, $1 \le i \le n$.
- Sei $P = \{ y_1, \ldots, y_s \}$ die Menge der Punkte, die Eckpunkt von zumindest einem Intervall sind, $s \le 2n$. Sortiere P, so dass $y_i < y_{i+1}$ gilt.
- Erzeuge einen binären Suchbaum mit minimaler Höhe für die Punkte $y_1,\ldots,y_s$.

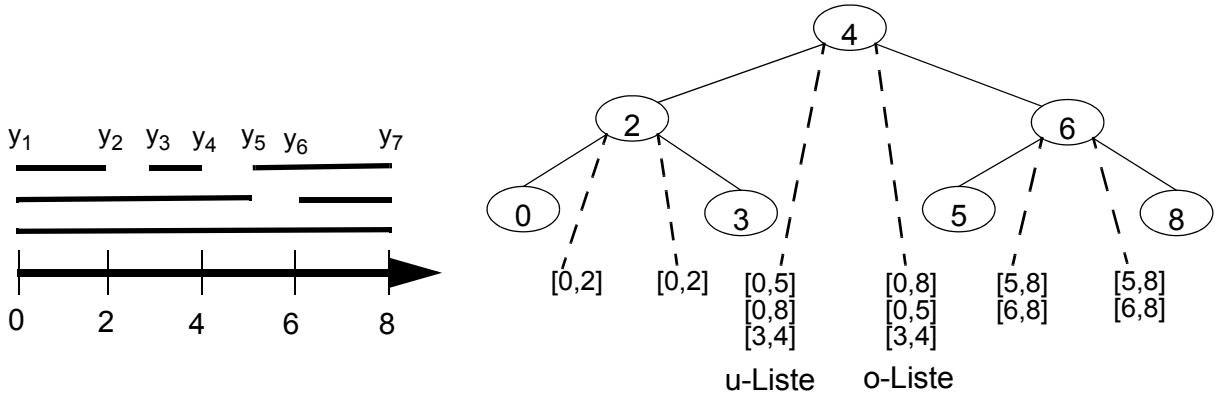


Einfügen eines Intervalls

- Jeder Knoten *Node* repräsentiert einen *Punkt Node.y*; jedem Knoten wird die *Menge von Intervallen* $I_1,\ldots,I_m$ zugeordnet, die den Punkt *Node.y* enthalten.
- Die Menge $I_1,\ldots,I_m$ wird redundant in zwei sortierten Listen verwaltet:
  - *u*-Liste: Alle Intervalle sind nach den *u*-Werten aufsteigend sortiert.
  - *o*-Liste: Alle Intervalle sind nach den *o*-Werten absteigend sortiert.
- Ein Intervall *I* wird in die beiden Listen des *ersten* Knotens *K* eingetragen, dessen Punkt *K.y* in *I* enthalten ist (Durchlauf von der Wurzel beginnend):

```
ALGORITHM Insert (Node, I)                  //erster Aufruf mit Node = Wurzel
    IF I.u ≤ Node.y ≤ I.o THEN
        InsertIntoList (Node.u_list, I.u);
        InsertIntoList (Node.o_list, I.o);
    ELSIF I.o < Node.y THEN
        Insert (Node.left, I);
    ELSE
        Insert (Node.right, I);
```

Beispiel

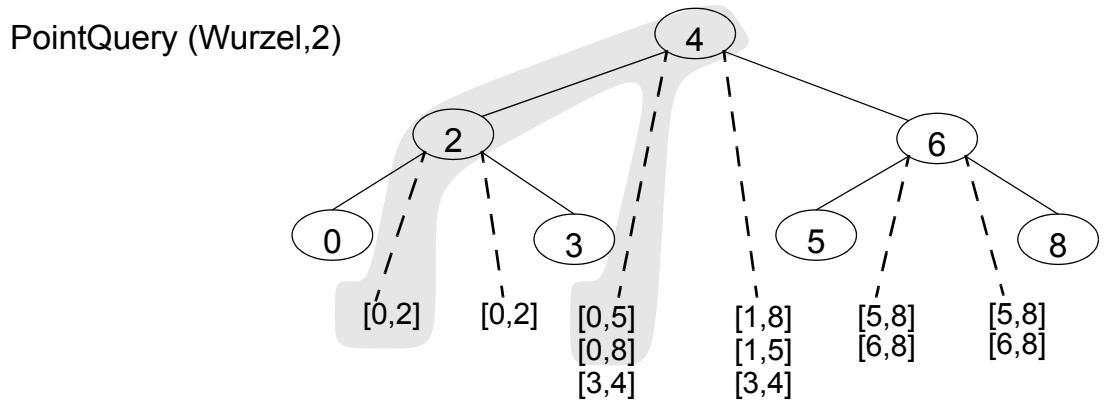


Punktanfrage

- Gegeben: Anfragepunkt *p*.
- Gesucht: alle Intervalle *I*, die *p* enthalten.

```
ALGORITHM PointQuery(Node, p)          //erster Aufruf mit Node = Wurzel
    IF p = Node.y THEN
        Gebe alle Intervalle der u-Liste aus;
    ELSIF p < Node.y THEN
        Gebe sequentiell den Anfang der u-Liste aus, bis ein u-Wert größer als p ist;
        PointQuery (Node.left, p);
    ELSE
        Gebe sequentiell den Anfang der o-Liste aus, bis ein o-Wert kleiner als p ist;
        PointQuery (Node.right, p);
```

Beispiel



Komplexitätsanalyse

Zur Organisation der *o*-Listen und der *u*-Listen verwenden wir balancierte Bäume, z.B. AVL-Bäume, mit folgenden Modifikationen:

- Alle Schlüssel befinden sich in den Blättern.
- Die Blätter sind entsprechend der Sortierreihenfolge der Schlüssel miteinander verkettet.

Dann erhalten wir folgende Komplexitäten:

Einfügen

- Erreichen des Einfügeknotens im Intervallbaum: O(log n) Laufzeit.
- Einfügen in die o- und die u-Liste: O(log n) Laufzeit.
- Gesamtaufwand für die Einfügung *eines* Intervalls: O(log n) Laufzeit, O(1) Speicherplatz.

  ➞ Speicherplatzaufwand für n Intervalle: O(n)

Punktanfrage

- Die Anfragebearbeitung ist auf einen Pfad des Baums beschränkt.
- Für jeden Knoten des Pfades wird maximal auf ein Intervall zugegriffen, das nicht Antwort ist (wegen der Sortierung der Intervalllisten).

➞ Laufzeit für eine Punktanfrage: O(log $n + r$) für r Antworten

Schlussfolgerung

Der Intervallbaum beantwortet eine Punktanfrage also genauso effizient wie der Segmentbaum, d.h. in O(log $n + r$), und reduziert den Speicherplatzbedarf für n Intervalle auf O(n). Der Sweep-Line-Status für das Punkteinschluss-Problem sollte also durch einen Intervallbaum implementiert werden.

6.1.2 Geometrisches Divide-and-Conquer

Beim Entwurf eines Divide-and-Conquer-Algorithmus für ein geometrisches Problem stellt sich insbesondere die Frage nach einem geeigneten Divide-Schritt.

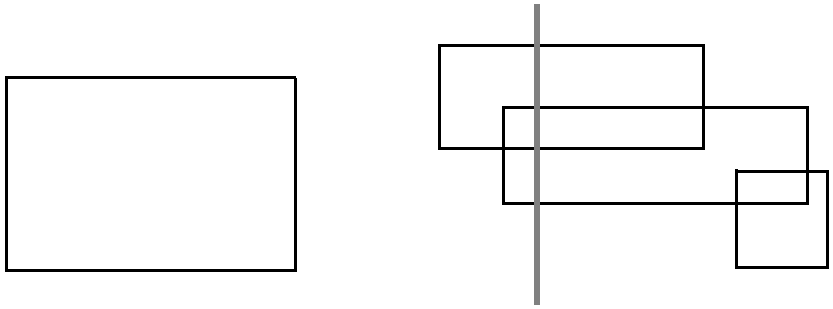
Ansätze für einen Divide-Schritt

- Geometrie nicht berücksichtigen

$$\{o_1, \ldots, o_n\} \rightarrow \{o_1, \ldots, o_{\lfloor n/2 \rfloor}\}\{o_{\lfloor n/2 \rfloor + 1}, \ldots, o_n\}$$

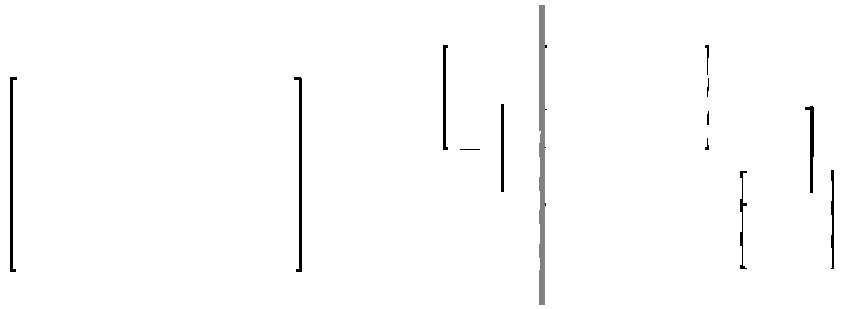
  ➞ keine Lösung für den Merge-Schritt

- teilende Gerade durch die Objektmenge legen



➞ drei Teilmengen: “Links”, “Geschnitten”, “Rechts”

- Objekte getrennt repräsentieren und dann teilende Gerade verwenden
  Die linken und rechten Kanten werden vom Algorithmus als unabhängige Einheiten behandelt.



  ➞ zwei Teilmengen: “Links” und “Rechts”,
  Teillösungen können im Merge-Schritt zu Gesamtlösung zusammengesetzt werden.

Segmentschnitt-Problem

Im folgenden entwickeln wir einen Divide-and-Conquer-Algorithmus für das Segmentschnitt-Problem. Bezeichne *S* die Menge der x-Koordinaten aller vertikalen Strecken und aller Endpunkte horizontaler Strecken.

ALGORITHM SegmentIntersect (S)

Divide:	Teile S in zwei etwa gleichgroße Mengen S_1 und S_2 auf, so dass für alle $x_1 \in S_1$ und $x_2 \in S_2$ gilt: $x_1 < x_2$.
Conquer:	SegmentIntersect(S_1);
	SegmentIntersect(S_2);
Merge:	?

Entwurf des Merge-Schrittes

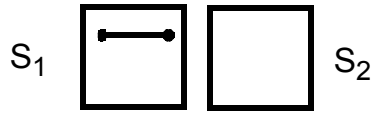
Die Methode SegmentIntersect besitzt folgende *Invariante*:

> SegmentIntersect(S_i) liefert alle Schnitte zwischen den in S_i repräsentierten Segmenten. Dabei sei ein horizontales Segment in S_i repräsentiert, wenn mindestens einer seiner Endpunkte in S_i enthalten ist.

Im Merge-Schritt müssen deshalb nur noch die Schnitte zwischen horizontalen Segmenten in S_1 (bzw. S_2) und vertikalen Segmenten in S_2 (bzw. S_1) gefunden werden.

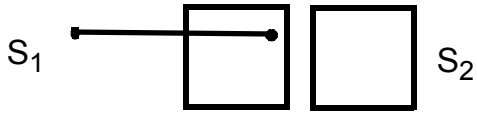
Wir betrachten im folgenden die verschiedenen Fälle für ein horizontales Segment h in S_1 (die anderen Fälle sind analog):

Fall 1: Beide Endpunkte liegen in S_1.



h schneidet kein Segment in S_2.

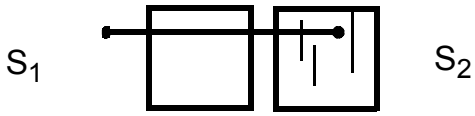
Fall 2: Nur der rechte Endpunkt liegt in S_1.



h schneidet kein Segment in S_2.

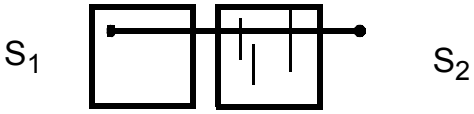
Fall 3: Der rechte Endpunkt liegt rechts von S_1.

Fall 3.1 Der rechte Endpunkt liegt in S_2.



Wegen der Invariante von SegmentIntersect hat SegmentIntersect (S_2) bereits alle Schnitte von h mit vertikalen Segmenten in S_2 geliefert.

Fall 3.2 Der rechte Endpunkt liegt rechts von S_2.



h ist in S_2 nicht repräsentiert, so dass SegmentIntersect (S_2) die Schnitte von h mit vertikalen Segmenten in S_2 *nicht* geliefert hat. Diese Schnitte müssen also im Merge-Schritt noch bestimmt werden.

Implementierung des Merge-Schrittes

Im Merge-Schritt müssen also alle Schnitte von horizontalen Segmenten in S_1 (S_2), deren Endpunkt rechts von S_2 (links von S_1) liegt, mit vertikalen Segmenten in S_2 (S_1) bestimmt werden. Für diese Operation benötigen wir nur die y-Koordinaten der horizontalen Segmente und die y-Intervalle der vertikalen Segmente.

Bezeichne Y eine Menge von y-Koordinaten horizontaler Segmente und V eine Menge von y-Intervallen vertikaler Segmente. Wir definieren:

$$Y \bigstar V := \{ (v.x , y) \mid y \in Y, v \in V, v.y1 \leq y \leq v.y2\}$$

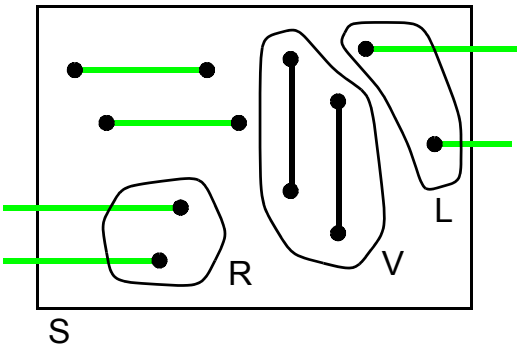
Die $\bigstar$-Operation löst die Aufgabe des Merge-Schrittes. Sei Y als sortierte Liste implementiert und V als Liste, die nach unteren Intervallgrenzen y1 sortiert ist. Dann lässt sich die $\bigstar$-Operation in

$$O(|Y| + |V| + |Y \bigstar V|)$$

Laufzeit ausführen, indem die beiden Listen Y und V synchron durchlaufen werden.

Datenstrukturen

- S ist ein nach den x-Koordinaten sortiertes Array von horizontalen Eckpunkten bzw. vertikalen Strecken.
- L ist eine nach y-Koordinaten sortierte Liste aller linken Endpunkte horizontaler Segmente in S, deren rechter Endpunkt nicht in S liegt.
- R ist eine nach y-Koordinaten sortierte Liste aller rechten Endpunkte horizontaler Segmente in S, deren linker Endpunkt nicht in S liegt.
- V ist eine nach y-Koordinaten des unteren Endpunkts sortierte Liste aller vertikalen Segmente in S.



Algorithmus

SegmentIntersect (**IN** S, **OUT** L, **OUT** R, **OUT** V)

IF S = {s} **THEN**
L := ∅; R := ∅; V := ∅;
IF s ist linker Eckpunkt **THEN** L := {s};
IF s ist rechter Eckpunkt **THEN** R := {s};
IF s ist vertikale Strecke **THEN** V := {s};

ELSE // |S| > 1

// ***Divide***

Teile S in zwei möglichst gleich große Mengen S_1 und S_2 auf, so dass
für alle $x_1 \in S_1$ und $x_2 \in S_2$ gilt: $x_1 < x_2$;

// ***Conquer***

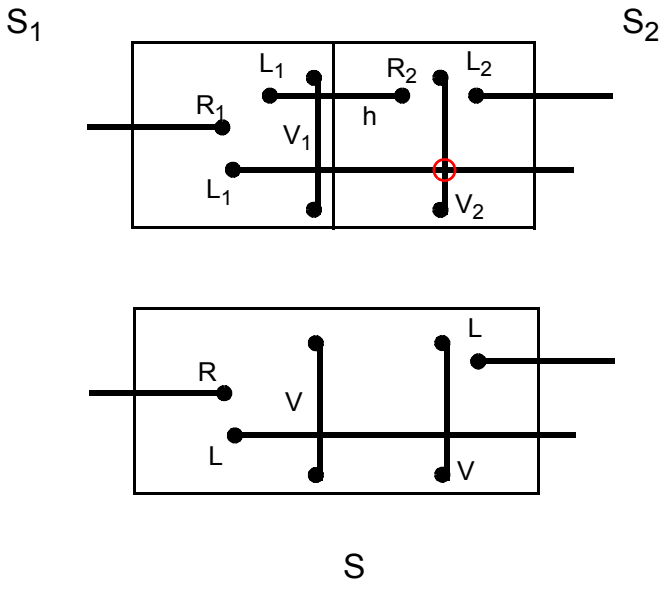
SegmentIntersect (S_1, L_1, R_1, V_1);
SegmentIntersect (S_2, L_2, R_2, V_2);

// ***Merge***

$h := L_1 \cap R_2$;

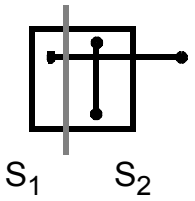
Gebe $(L_1 \setminus h) \star V_2$ aus; (* Fall A *)
Gebe $(R_2 \setminus h) \star V_1$ aus; (* Fall B *)

$L := (L_1 \setminus h) \cup L_2$;
$R := R_1 \cup (R_2 \setminus h)$;
$V := V_1 \cup V_2$;

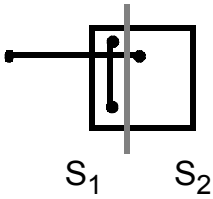


Der obige Algorithmus SegmentIntersect erfüllt die oben angegebene *Invariante*, denn

- Für |S| = 1 gibt es keine Schnitte.
- Für |S| = 2 werden alle Schnitte bestimmt, denn für



wird in Fall A und für



wird im Fall B des Merge-Schrittes der Schnitt gefunden.

- Für |S| > 2 kann man analog zu |S| = 2 argumentieren.

Komplexitätsanalyse

- Sortierung von S: O(n log n)
- Divide (ein Schritt): O(1)
  denn S ist ein nach x-Koordinaten sortiertes Array.
- Merge (ein Schritt):

  Die Operationen $\cap$ und $\cup$ auf Mengen L, R und V von n Elementen können in O(n) bearbeitet werden, da die Mengen als sortierte Listen repräsentiert sind.

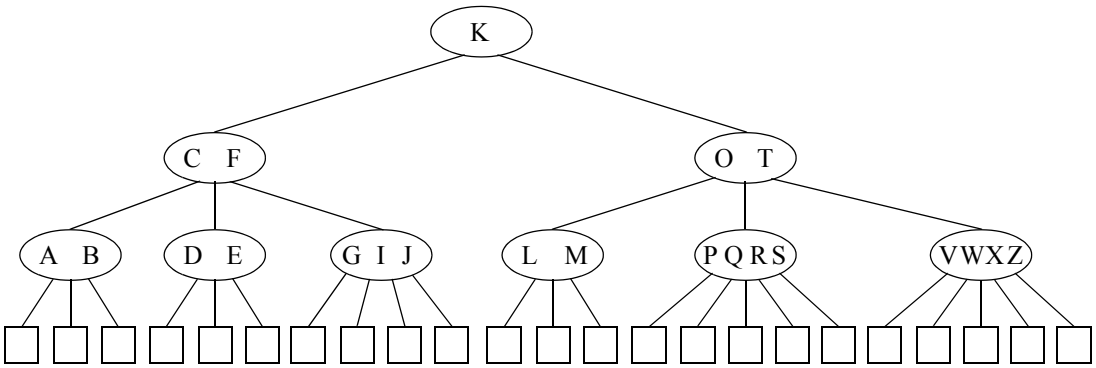
  Die Operation Y ✯ V kann in O(|Y| + |V| + |Y ✯ V|) durchgeführt werden (d.h. lineare Laufzeit + Anzahl der gefundenen Paare), da die Mengen als sortierte Listen repräsentiert sind.
- Tiefe des Rekursionsbaumes: O(log n)
  da eine balancierte Aufteilung des Problems erfolgt.

➞ Gesamtlaufzeit: O(n log n + k)

➞ Gesamtspeicherplatzbedarf: O(n)

Ergänzung zu Bereichsanfragen

Beispiel: für einen B-Baum der Ordnung 2



Warum unterstützt dieser B-Baum die folgende Bereichsanfrage nicht effizient?
"Lies die Informationen aller Datensätze im Bereich [B ... V] aus"

Grundidee:

- Trennung der Indexstruktur in *Directory* und *Datei*.
- *Sequentielle Verkettung* der Daten in der Datei.

B^+-Baum

- ***B^+-Datei:***
  - Die Blätter des B^+-Baumes heißen ***Datenknoten*** oder ***Datenseiten***.
  - Die Datenknoten enthalten alle Datensätze.
  - Alle Datenknoten sind entsprechend der Ordnung auf den Primärschlüsseln *verkettet*.
- ***B^+-Directory***:
  - Die inneren Knoten des B^+-Baumes heißen ***Directoryknoten*** oder ***Directoryseiten***.
  - Directoryknoten enthalten nur noch ***Separatoren*** s.
  - Für jeden Separator s(u) eines Knotens u gelten folgende ***Separatoreneigenschaften***:
    - $s(u) > s(v)$ für alle Directoryknoten v im linken Teilbaum von s(u).
    - $s(u) < s(w)$ für alle Directoryknoten w im rechtenTeilbaum von s(u).
    - $s(u) > k(v')$ für alle Primärschlüssel k(v') und alle Datenknoten v' im linken Teilbaum von s(u).
    - $s(u) \leq k(w')$ für alle Primärschlüssel k(w') und alle Datenknoten w' im rechten Teilbaum von s(u).

Beispiel: B^+-Baum für die Zeichenketten: An, And, Certain, For, From, Which, With.