Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

8. Juni 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 14 FS 16

Lösung 14.1 *Rollende Kugeln.*

Haltepunkte: Wir verwenden eine Scanline, die von links nach rechts läuft. Die Haltepunkte sind die Anfangs- und Endpunkte der Liniensegmente, sowie die Startposition der Kugel. (Gäbe es mehr als einen Punkt, der auf der Scanline liegt, dann würden wir die Punkte in absteigender Reihenfolge ihrer y-Koordinaten betrachten.)

Scanline-Datenstruktur: Wir benutzen einen AVL-Baum und speichern die Geradengleichung der Liniensegmente. Um ein neues Liniensegment einzufügen, müssen die Geradengleichungen an der Scanline-Position ausgewertet werden und das neue Segment gemäss der Ordnung an der Scanline-Position in den AVL-Baum einfügen. Wir speichern zudem für jedes Liniensegment a, welches Segment b direkt unter demjenigen Endpunkt dieses Segments a liegt, dessen y-Koordinate kleiner ist.

Aktualisierung: Entspricht der Haltepunkt dem linken Endpunkt eines Segments, so wird dieses in den AVL-Baum eingefügt. Beim rechten Endpunkt wird das Segment aus dem AVL-Baum gelöscht. In beiden Fällen wird zusätzlich geprüft, ob der Endpunkt eine kleinere y-Koordinate als der andere Endpunkt des Segments hat. Dazu betrachten wir die Steigung der Geradengleichung. Ist dies der Fall, so wird zusätzlich das Segment ermittelt, das an der Scanline-Position direkt unter diesem Endpunkt liegt, und gespeichert. Beim Haltepunkt, der dem Startpunkt der Kugel entspricht, wird lediglich das direkt darunterliegende Segment berechnet und gespeichert.

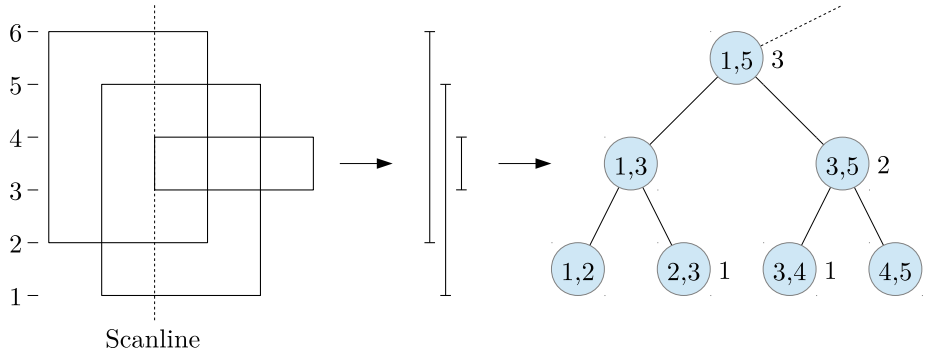
Auslesen der Lösung: Wir kennen das erste Segment, auf das die Kugel trifft und für jedes Segment wissen wir, auf welches Segment die Kugel fallen wird, nachdem es von diesem Segment herunterfällt. Daher kann die Lösung leicht berechnet werden.

Laufzeit: Es gibt $\mathcal{O}(n)$ Haltepunkte, die sortiert werden müssen. Eine Aktualisierungs-Operation benötigt $\mathcal{O}(\log n)$ Zeit. Die Lösung kann dann in $\mathcal{O}(n)$ Laufzeit ausgelesen werden. Daher hat unser Algorithmus eine Gesamtlaufzeit von $\mathcal{O}(n \log n)$.

Lösung 14.2 *Aufspiessen von orthogonalen Rechtecken.*

Haltepunkte: Wir lassen die Scanline von links nach rechts laufen und halten genau dann an, wenn wir entweder auf ein neues Rechteck treffen oder ein Rechteck geschlossen wird. Hierzu sortieren wir die x-Koordinaten der linken und der rechten Seite jedes Rechtecks aufsteigend und verwenden sie als Haltepunkte.

Scanline-Datenstruktur: Als derzeit aktive Objekte werden genau die Intervalle gespeichert, die durch den Schnitt der Scanline mit den Rechtecken entstehen. Hierzu wird ein modifizierter Segmentbaum benutzt. Allerdings speichern wir in jedem Knoten nicht die Intervalle selbst ab, sondern lediglich einen Zähler, der angibt, wie viele der im Segmentbaum unterhalb von diesem Knoten repräsentierten Intervalle sich maximal überlappen. Diese Zahl kann leicht rekursiv berechnet werden: Wenn sich für die zwei Kinder eines Knotens v jeweils maximal m_1 bzw. m_2



Beispiel: Schneidet die Scanline die links dargestellten Rechtecke, dann erhalten wir die mittig dargestellen Intervalle $[2,6]$, $[1,5]$, $[3,4]$ (v.l.n.r). Rechts ist ein Ausschnitt der Scanline-Datenstruktur dargestellt. Die jeweilige Anzahl der maximal überlappenden Intervalle ist rechts neben jedem Knoten notiert. Man beachte, dass das Intervall $[2,6]$ in den Knoten $[2,3]$, $[3,5]$ und $[5,6]$ gespeichert wird. Aus diesem Grund ist in $[2,3]$ der Zähler 1 und in $[3,5]$ der Zähler 2 gespeichert. Ein unterhalb von $[1,5]$ maximal überdecktes Elementarintervall ist $[3,4]$.

Intervalle überlappen, dann ist die Anzahl der sich maximal überlappenden Intervalle unterhalb von v genau die Summe aus der Anzahl der durch v repräsentierten Intervalle und $\max\{m_1, m_2\}$ (alle durch v repräsentierten Intervalle überdecken auch die an den Kindern repräsentierten Intervalle, also muss die Anzahl der sich maximal überlappenden Intervalle entsprechend erhöht werden). Zusätzlich wird in jedem Knoten v ein Elementarintervall gespeichert, das durch eine maximale Anzahl von Intervallen, die im Segmentbaum unter v gespeichert sind, überdeckt wird.

Man beachte, dass die y-Koordinaten der Rechtecke rational sein dürfen, der Segmentbaum aber standardmässig nur Intervalle mit ganzzahligen Grenzen aufnehmen kann. Also sortieren wir die möglichen y-Koordinaten der Rechtecke im Voraus, entfernen doppelte Vorkommen und erhalten die möglichen Koordinaten $y_1, \ldots, y_k$ mit $k \leq 2n$. Nun ändern wir die y-Koordinaten jedes Rechtecks: Hat ein Rechteck R die Ecken (x, y_i), (x, y_j), (x', y_i) und (x', y_j), dann wird es durch ein Rechteck R' mit den Ecken (x, i), (x, j), (x', i) und (x', j) ersetzt. Auf diese Weise kann ein Segmentbaum mit Intervallgrenzen aus $\{1, \ldots, k\}$ benutzt werden.

Aktualisierung: Hier unterscheiden wir zwei Fälle.

1. *Fall: Ein neues Rechteck startet.* Seien I das Intervall, das durch den Schnitt der Scanline mit dem neuen Rechteck entsteht und $v_1, \ldots, v_k$ die Knoten, in denen I in einem üblichen Segmentbaum gespeichert würde. In all diesen Knoten muss der Zähler der sich maximal überlappenden Intervalle um 1 erhöht werden. Zusätzlich müssen die Zähler aller entsprechenden Vorgängerknoten auf dem Weg zur Wurzel aktualisiert werden. Die Werte aller anderen Knoten bleiben unverändert.

2. *Fall: Ein Rechteck endet.* In diesem Fall gehen wir analog zum 1. Fall vor, verringern den Zähler in den entsprechenden Knoten des Intervalls um 1 und aktualisieren die Zähler der entsprechenden Vorgängerknoten.

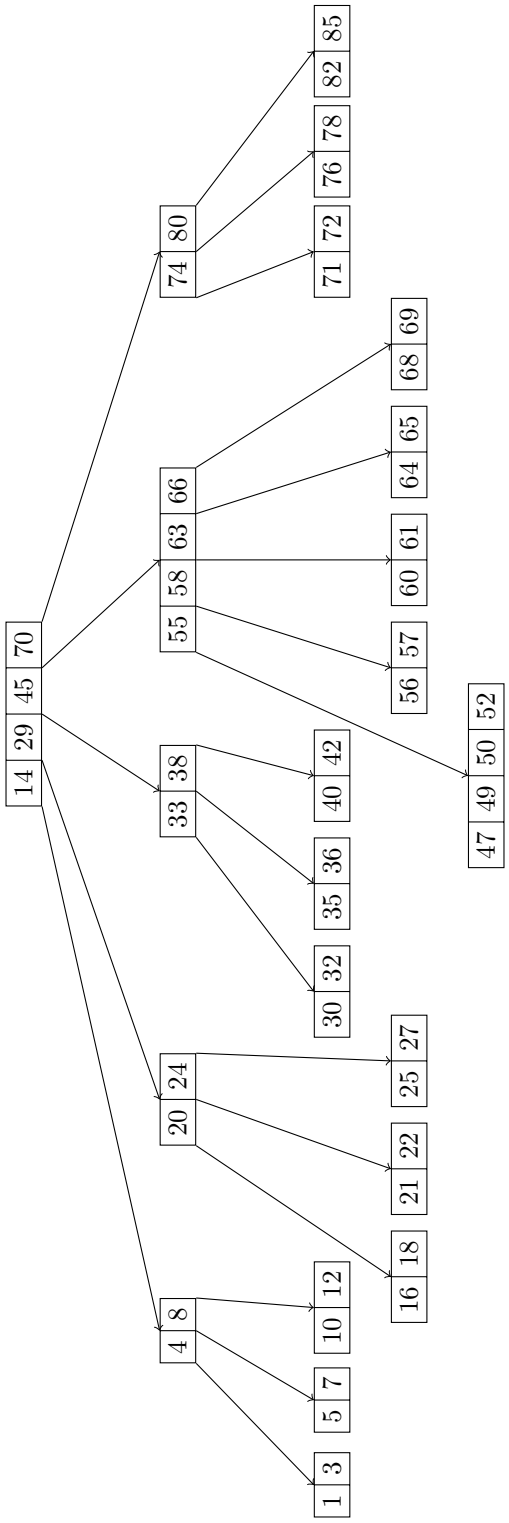
Auslesen der Lösung: Wir prüfen jedes Mal, wenn die Scanline-Datenstruktur aktualisiert wird, ob sich die Anzahl der sich insgesamt maximal überlappenden Rechtecke erhöht. Dieser Wert ist in der Wurzel des Segmentbaums gespeichert. Wir speichern dann die maximale Anzahl sich überlappender Rechtecke, ein maximal überlapptes Elementarintervall sowie die zugehörige Position der Scanline. Am Ende kann mit diesen Informationen dann ein Punkt bestimmt werden, der auf der gespeicherten Scanline-Position innerhalb des Elementarintervalls liegt.

Laufzeit und Platzverbrauch: Die initiale Sortierung der Haltepunkte kann in Zeit $\Theta(n \log n)$ durchgeführt werden. Gleiches gilt natürlich auch für die Sortierung der y-Koordinaten. Einfügen und Löschen können im modifizierten Segmentbaum weiterhin in Zeit $\Theta(\log n)$ durchgeführt werden, da nur die Knoten entlang der betrachteten Pfade aktualisiert werden müssen. Da es $2n$ Haltepunkte gibt, ergibt sich eine Gesamtlaufzeit von $\Theta(n \log n)$. Der Platzverbrauch ist durch $\Theta(n)$ beschränkt, da anders als in einem normalen Segmentbaum nicht alle Intervalle abgespeichert werden, sondern lediglich ein Zähler (der nur konstanten Platz benötigt).

Bemerkung: Die Aufgabenstellung erlaubt explizit, dass sich ein Aufspiesspunkt auch auf dem Rand von Rechtecken befinden kann. Die obige Lösung findet jedoch ausschliesslich Aufspiesspunkte, die sich im Inneren von Rechtecken befinden. Wir können dieses Problem mit der im Buch (Seite 509) beschriebenen alternativen Definition von Segmentbäumen lösen, bei der elementare Segmente nicht beidseitig abgeschlossen sind, sondern beidseitig offen. In den Blättern des Segmentbaums sind dann abwechselnd solche beidseitig offenen Intervalle und Punkte gespeichert, sodass eine disjunkte Partition entsteht und auch Segmente gespeichert werden können, die nur aus einem einzigen Punkt bestehen.

Lösung 14.3 *B-Bäume.*

Im folgenden ist ein B-Baum der Ordnung 5 mit Höhe 3 dargestellt. Wird beispielsweise der Schlüssel 48 eingefügt, so wächst die Höhe des Baumes.



Lösung 14.4 *Konvexe Hülle (**Programmieraufgabe**).*

Für die Beschreibung von Graham's Scan verweisen wir auf Kapitel 8.2.2 des Buchs zur Vorlesung. Eine mögliche Implementierung:

```
1  import java.util.Stack;
2  import java.util.Arrays;
3  import java.util.Comparator;
4  import java.util.Scanner;
5
6  class Point {
7
8      public final Comparator<Point> POLAR_ORDER = new PolarOrder();
9
10     public int x, y;
11
12     public Point( int x, int y ){
13         this.x = x;
14         this.y = y;
15     }
16
17     public String toString(){
18         return x + " " + y;
19     }
20
21     // compare other points relative to polar angle they make with this Point
22     private class PolarOrder implements Comparator<Point> {
23         public int compare(Point q1, Point q2) {
24             double dx1 = q1.x - x;
25             double dy1 = q1.y - y;
26             double dx2 = q2.x - x;
27             double dy2 = q2.y - y;
28
29             if      (dy1 >= 0 && dy2 < 0) return -1;    // q1 above; q2 below
30             else if (dy2 >= 0 && dy1 < 0) return +1;    // q1 below; q2 above
31             else if (dy1 == 0 && dy2 == 0) {            // 3-collinear and
                   horizontal
32                 if      (dx1 >= 0 && dx2 < 0) return -1;
33                 else if (dx2 >= 0 && dx1 < 0) return +1;
34                 else                          return  0;
35             }
36             else return check_turn(Point.this, q1, q2);     // both above or below
37         }
38     }
39
40     //If p, q and r make a clockwise turn return < 0, otherwise return > 0
41     public static int check_turn(Point p, Point q, Point r){
42         return (q.x - p.x)*(r.y - p.y) - (q.y - p.y)*(r.x - p.x);
43     }
44 }
45
46 class Main {
47
48     public static void main(String[] args) {
49
50         Scanner in = new Scanner(System.in);
51         int tests = in.nextInt();
52
53         for( int t = 0; t < tests; t++ ) {
54             Stack<Point> hull = new Stack<Point>();
55
```

```
56              int n = in.nextInt();
57              Point points[] = new Point[n];
58
59              //Read points from std input and keep lexicographically
60              //smallest point index
61              int minx = 100000;
62              int minindex = 0;
63              for (int i = 0; i < n; i++) {
64                  int x = in.nextInt();
65                  int y = in.nextInt();
66                  points[i] = new Point(x, y);
67                  if (x < minx) { minindex = i; minx = x; }
68                  else if (x == minx && y < points[minindex].y) { minindex = i;}
69              }
70
71              //swap points[0] with points[minindex]
72              Point temp = points[0];
73              points[0] = points[minindex];
74              points[minindex] = temp;
75
76              //the leftmost point will be on the convex hull
77              hull.push(points[0]);
78
79              //Sort points by polar order
80              Arrays.sort(points, 1, n, points[0].POLAR_ORDER);
81
82              //We know that the input points are in general position and pairwise
                    different
83              //So, points[1] will also be on the convex hull.
84              hull.push(points[1]);
85
86              // Graham scan; note that points[N-1] is also an extreme point
87              for (int i = 2; i < n; i++) {
88                  Point top = hull.pop();
89                  while (Point.check_turn(hull.peek(), top, points[i]) >= 0) {
90                      top = hull.pop();
91                  }
92                  hull.push(top);
93                  hull.push(points[i]);
94              }
95
96              //output
97              for (Point p : hull) System.out.print(p + " ");
98              System.out.println();
99          }
100     }
101 }
```

Eine weitere Möglichkeit um die konvexe Hülle zu berechnen ist ein sogenannter linearer Scan (Kapitel 8.2.3). Dieser Algorithmus verwendet zwei symmetrischen Scans. Der erste Scan berechnet alle Eckpunkte auf der *oberen* konvexen Hülle, beginnend mit dem am weitesten links liegenden Punkt, endend im am weitesten rechts liegenden Punkt, sortiert im Uhrzeigersinn. Der zweite Scan berechnet alle verbleibenden Eckpunkte der konvexen Hülle, die sog. *untere* konvexe Hülle. Da beide Scans symmetrisch verlaufen, beschreiben wir lediglich die Berechnung der oberen konvexen Hülle.

Die Scanline hält an jedem Punkt der Eingabe. Die Punkte werden von links nach rechts gescannt, und haben zwei Punkte die gleiche x-Koordinate, dann scannen wir denjenigen mit der kleineren y-Koordinate zuerst.

Wir speichern die Eckpunkte der konvexen Hülle in einer Liste vom Typ `ArrayList<Point>`. Jedes mal wenn die Scanline auf einen Haltepunkt trifft, wird dieser in die Liste eingefügt. Enthält die Liste mindestens drei Punkte, dann prüfen wir, ob die letzten drei Punkte p, q und r eine Linkskurve bilden. Falls ja, entfernen wir q aus der Liste (da dies offensichtlich kein Eckpunkt der konvexen Hülle ist) und fahren so fort, bis die letzten drei Punkte eine Rechtskurve bilden.

Wurde jeder Haltepunkt verarbeitet, dann enthält die obige Liste genau die Eckpunkte der oberen konvexen Hülle im Uhrzeigersinn. Ein zweiter Scan, der analog zum ersten abläuft, berechnet die Menge der Eckpunkte der unteren konvexen Hülle im Uhrzeigersinn, startend mit dem am weitesten rechts liegenden Punkt. Zur Berechnung der Gesamtlösung müssen nur der rechteste Punkt aus der oberen Hülle sowie der linkeste Punkt aus der unteren Hülle entfernt und die Listen der Eckpunkte konkateniert werden.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

class Point implements Comparable<Point> { ... };

class Main {

    // Returns a value < 0 if p, q, r form a clockwise turn, and a value > 0
        otherwise
    public static int check_turn(Point p, Point q, Point r) { ... }
    public static ArrayList<Point> upper_hull(Point points[]) { ... }
    public static ArrayList<Point> lower_hull(Point points[]) { ... }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int tests = in.nextInt();

        for(int t = 0; t < tests; t++){
            int n = in.nextInt();
            Point points[] = new Point[n];

            // Read points from the input
            for(int i = 0; i < n; i++) {
                points[i] = new Point(in.nextInt(), in.nextInt());
            }

            // Sort points from left to right
            Arrays.sort(points);

            ArrayList<Point> hull = new ArrayList<Point>();
            hull.addAll(upper_hull(points)); // Remove the rightmost point from the
                upper hull
            hull.remove(hull.size()-1);
            hull.addAll(lower_hull(points)); // Remove the leftmost point from the
                lower hull
            hull.remove(hull.size()-1);

            // Print the convex hull
            for(int i = 0; i < hull.size()-1; i++) {
                System.out.print(hull.get(i) + " ");
            }
            System.out.println(hull.get(hull.size()-1));
        }

        in.close();
    }

}
```