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
Holger Flier, Beat Gfeller

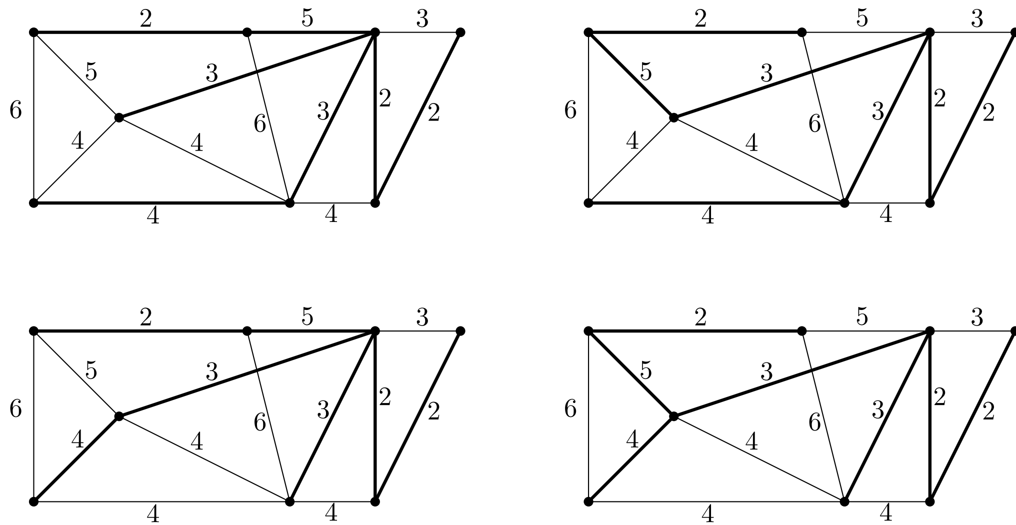
# Prüfung
# Datenstrukturen und Algorithmen
# D-INFK

10. Februar 2010

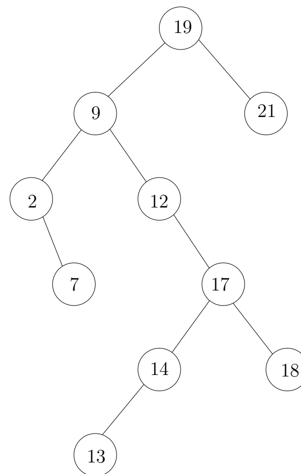
**Musterlösung**

**Aufgabe 1:**

**1 P** a) Es gibt vier korrekte Lösungen (Gesamtgewicht 21):



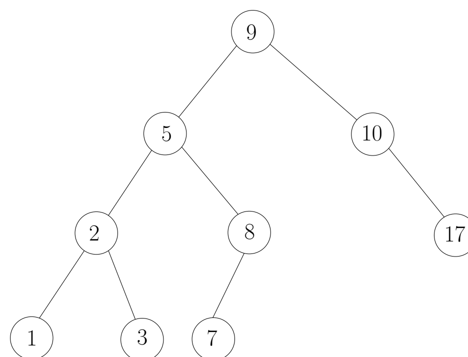
**1 P** b)




**1 P** c) nach dem ersten Durchgang:

| 2 | 5 | 6 | 9 | 12 | 14 | 1 | 4 | 8 | 11 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

**1 P** d)



**1 P** e) Mit Sondieren nach links:

| 14 | 8 | 16 | 9 | 4 | | 1 | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | (Index) |

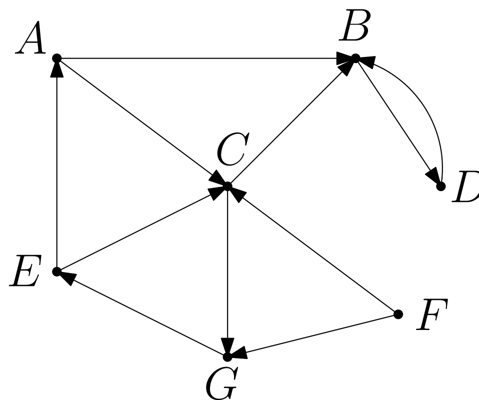
Mit Sondieren nach rechts:

| 14 | 8 | 16 | 1 | 4 | 9 | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | (Index) |

**1 P** f)

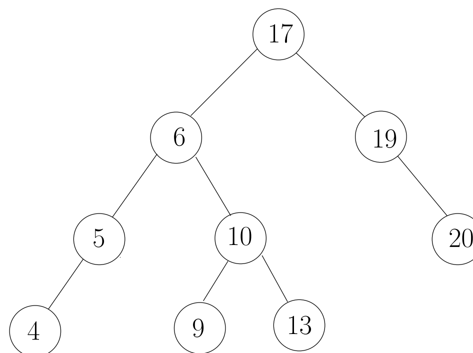
| 2 | 5 | 7 | 6 | 9 | 8 | 12 | 10 |
|---|---|---|---|---|---|---|---|

**1 P** g) Der folgende gerichtete Graph wird mit Tiefensuche traversiert. Die Suche startet beim Knoten $A$. Geben Sie eine Reihenfolge an, in der die Knoten erreicht werden können.

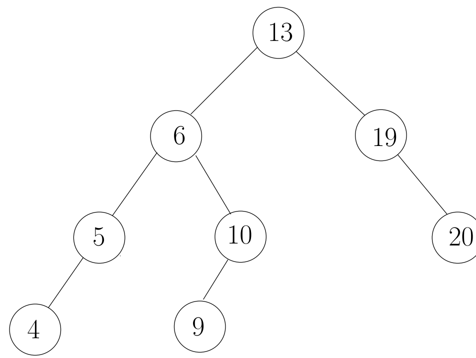



Es gibt drei korrekte Lösungen: A,B,D,C,G,E, oder A,C,B,D,G,E, oder A,C,G,E,B,D

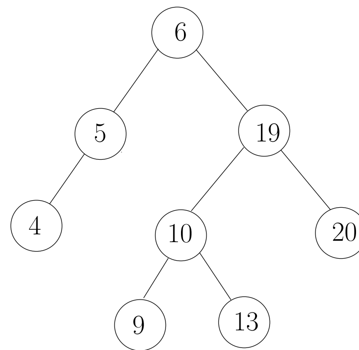
**1 P** h) Nach Einfügen von 9:



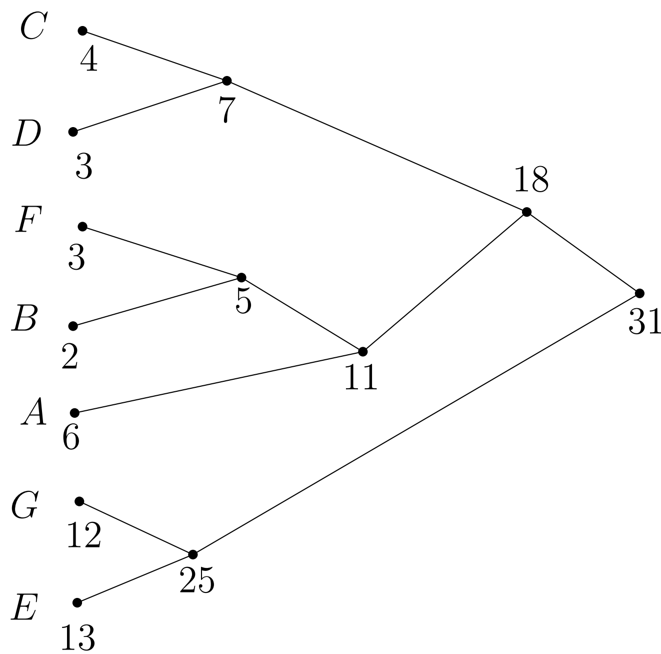

Nach Löschen von 17:




**Alternative Lösung nach Löschen von 17:** Benutze den symmetrischen Nachfolger, d.h. 19. Dann ist aber eine Rotation nötig:




**1 P** i)



**Aufgabe 2:**

a)

$$n \log n, \quad n\sqrt{n}, \quad \frac{n^2}{\log n}, \quad n^3, \quad 3^{\sqrt{n}}, \quad n!$$

b)

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{4}\right) + 2n + 1 \\ &= 2\left(2T\left(\frac{n}{4^2}\right) + 2n/4 + 1\right) + 2n + 1 \\ &= 2\left(2T\left[2T\left(\frac{n}{4^3}\right) + 2n/4^2 + 1\right] + 2n/4 + 1\right) + 2n + 1 \\ &= 2^i \cdot T\left(\frac{n}{4^i}\right) + 2n \cdot (1 + 1/2 + \ldots + 1/2^{i-1}) + (1 + 2 + \ldots + 2^{i-1}) \\ &= 2^i \cdot T\left(\frac{n}{4^i}\right) + 2n \cdot \left(1 - \frac{1}{2^{i-1}}\right) + (2^i - 1) \end{aligned}$$

Mit $i = \log_4(n)$ erhalten wir

$$T(n) = 3\sqrt{n} + 2n \cdot 2(1 - 1/\sqrt{n}) + (\sqrt{n} - 1) = 4n - 1.$$

Beweis per Induktion:
Verankerung: $T(1) = 4 - 1 = 3$.
Induktionsschritt: von $n/4$ nach $n$:

$$T(n) = 2T\left(\frac{n}{4}\right) + 2n + 1 = 2(4n/4 - 1) + 2n + 1 = 4n - 1.$$

**1 P** c) Die Laufzeit ist ungefähr $n + n/2 + n/4 + n/8 + \ldots$, was insgesamt $\Theta(n)$ ergibt.

**1 P** d)

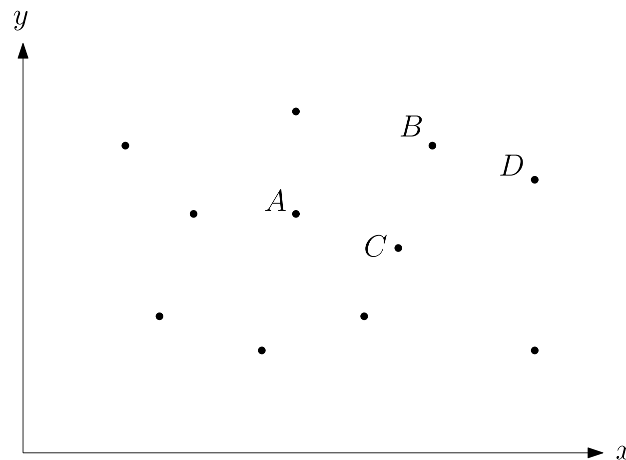
$$\sum_{k=1}^{n/8} (n - 2k) = \Theta(n^2)$$

**1 P** e)

$$\Theta(\sqrt{n} \cdot \log n)$$

**Aufgabe 3:**

In dieser Aufgabe betrachten wir Mengen von Punkten in der Ebene. Wir sagen, dass ein Punkt $(x, y)$ einen anderen Punkt $(x', y')$ *dominiert*, wenn $x > x'$ und gleichzeitig $y > y'$ gilt. Ein Punkt heisst *dominant*, wenn er von keinem anderen Punkt dominiert wird.
Im untenstehenden Bild wird beispielsweise der Punkt $A$ vom Punkt $B$ dominiert, aber nicht vom Punkt $C$. Der Punkt $D$ ist dominant.




**3 P** a) Wir benutzen einen (sehr einfachen) Scanline-Ansatz mit einer vertikalen Scanline, welche von rechts nach links läuft. Dazu sortieren wir die Punkte zuerst absteigend nach x-Koordinate – bei gleicher x-Koordinate sortieren wir aufsteigend nach y-Koordinate (wichtig!). Beim Scan gehen wir alle Punkte in dieser Reihenfolge durch. In der Scanline speichern wir nur einen Punkt ab, nämlich den mit der bisher grössten y-Koordinate. Bei jedem neuen Punkt beim Scan prüft man, ob dessen y-Koordinate ein neues Maximum ist, oder gleich gross wie das bisherige Maximum (wichtig!). Falls ja, ist dieser Punkt dominant und man gibt ihn aus (und merkt sich diesen Punkt als neues Maximum).

Dieser Ansatz hat eine Laufzeit von $O(n \log n)$ (für das Sortieren, der Rest geht in Linearzeit).

**6 P** b) Wir benutzen einen AVL-Baum als Datenstruktur. Diese enthält per Invariante jeweils genau alle dominaten Punkte der aktuellen Punktmenge, und zwar so sortiert wie in a) beschrieben: absteigend nach x-Koordinate, und bei gleicher x-Koordinate aufsteigend nach y-Koordinate.

Die Operation `Init`() erstellt in $O(1)$ einen leeren AVL-Baum.

Die Operation `ListEfficient`() traversiert einfach den AVL-Baum und gibt alle Punkte (als Liste) aus. Laufzeit: $O(k)$, wobei $k$ die Anzahl dominater Punkte in der aktuellen Punktmenge ist.

Die Operation `Insert`$(x, y)$ implementiert man wie folgt: Man sucht zuerst die Stelle in der sortierten Reihenfolge, und fügt den Punkt entsprechend in den AVL-Baum ein. Nun muss man noch alle Punkte löschen, die vom neu eingefügten Punkt dominiert werden und die somit nicht mehr dominant sind. Dazu verketten wir im AVL-Baum alle Knoten in Inorder-Reihenfolge miteinander: dann geht dieser Schritt in $O(w \cdot \log m)$, wobei $m$ die Anzahl aktueller Punkte vor dem Insert ist, und $w$ die Anzahl zu entfernender Punkte.

**Aufgabe 4:**

In dieser Aufgabe betrachten wir ein zweidimensionales Array $A$ mit $n^2$ verschiedenen Zahlen, bestehend aus $n$ Zeilen und $n$ Spalten. Zwei Felder im Array sind *benachbart*, wenn sie sich an einer Seite berühren, d.h. das Element $B[i,j]$ ist mit den vier Elementen $B[i-1,j]$, $B[i,j-1]$, $B[i+1,j]$ und $B[i,j+1]$ benachbart, falls diese Felder existieren (Elemente am Rand des Arrays sind mit entsprechend weniger Elementen benachbart).
Eine Folge $x_1, x_2, \ldots, x_k$ von Elementen im Array heisst *Sequenz*, wenn folgende Bedingungen erfüllt sind:

- die Folge von Elementen ist aufsteigend sortiert, und
- für jedes $i \in 1, \ldots, k-1$ die Elemente $x_i$ und $x_{i+}$ im Array benachbart sind.

**3 P** a) Entwerfen Sie einen möglichst effizienten Algorithmus, der im gegebenen Array eine längste Sequenz berechnet, die sich ganz in einer Zeile oder ganz in einer Spalte befindet. Im untenstehenden Beispiel-Array wäre eine mögliche Sequenz $4, 6, 10$. Diese ist jedoch keine längstmögliche Sequenz: Die Sequenz $6, 28, 29, 47$ ist länger. Beschreiben Sie Ihren Algorithmus in Worten, und geben Sie dessen Laufzeit an.

**6 P** b) Wir suchen nun eine längstmögliche beliebige Sequenz im gegebenen zweidimensionalen Array.

Im untenstehenden Beispiel-Array wäre eine mögliche Sequenz $4, 6, 28, 29, 47, 49$.

Entwerfen Sie einen möglichst effizienten Algorithmus, der eine solche längste Folge findet. Beschreiben Sie Ihren Algorithmus in Pseudocode, und geben Sie dessen Laufzeit an. Ihr Pseudocode muss in Java- Eiffel- oder C++-ähnlicher Notation geschrieben sein.

**Beispiel-Array:**

| | | | | |
|---|---|---|---|---|
| 9 | 27 | 42 | 41 | 48 |
| 35 | 39 | 8 | 3 | 5 |
| 12 | 49 | 2 | 38 | 4 |
| 15 | 47 | 29 | 28 | 6 |
| 19 | 1 | 25 | 33 | 10 |

**Lösung:**
a) Man muss offensichtlich jede Spalte und jede Zeile separat durchgehen, und jeweils die längste aufsteigende Folge darin ermitteln. Dies geht ganz einfach mit einem linearen Scan: Zuerst beginnt man die aktuelle Sequenz beim ersten Element. Solange das nächstfolgende Element nicht kleiner ist als das letzte (und somit angehängt werden kann), verlängert man die Folge. Wenn das nicht der Fall ist, beginnt man ab dem aktuellen Element eine neue Folge. Natürlich merkt man sich die längste bisherige Folge.
Es gibt $n$ Spalten und $n$ Zeilen, die Gesamtlaufzeit beträgt somit $O(2n^2) = O(n^2)$.
b) Hier wird ein Algorithmus nach dem Muster der dynamischen Programmierung benutzt. Die zentrale Idee ist, dass ein längster aufsteigender Pfad startend bei Feld $x$ immer aus einem längsten Pfad besteht, der von einem Nachbarfeld von $x$ aus startet (ausser, dies ist nicht möglich, weil alle Nachbarn von Feld $x$ eine kleinere Zahl enthalten).
Eine zweite wichtige Beobachtung: Da die Pfade aufsteigend sind und alle Zahlen im Array verschieden sind, kann man einen Pfad “Greedy” fortsetzen, weil nie ein Zyklus auftreten kann. Wir gehen daher einfach jedes Feld als Startfeld durch und verfolgen alle dort startenden Pfade. Damit

Felder nicht mehrfach betrachtet werden, speichern wir für jedes bereit besuchte Feld den längsten gefundenen aufsteigenden Pfad ab (es reicht, das nächste Feld in diesem Pfad abzuspeichern). Am einfachsten geht dies mit einer rekursiven Implementation. Hier ein Beispiel in Java:

```
public class LongestPath {
    int[][] B; // Input 2d-Array
    int n;

    static final int[][] neighborIndex = { { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 } };

    class Path {
        int length=0;
        Path next=null;
        int element;
    }

    Path findLongest() {
        Path[][] p = new Path[n][n];
        Path best = new Path();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                Path pa = findLongest(p, i, j, Integer.MIN_VALUE);
                if (pa.length > best.length)
                    best = pa;
            }
        }
        return best;
    }

    Path findLongest(Path[][] p, int x, int y, int last) {
        if (x < 0 || y < 0 || x >= n || y >= n || last > B[x][y]) {
            return new Path(); // empty path, length 0
        }
        if (p[x][y] == null) { // (x,y) not yet visited
            p[x][y] = new Path();
            p[x][y].element = B[x][y];
            for (int[] neighbor : neighborIndex) {
                Path pa = findLongest(p, x + neighbor[0], y + neighbor[1],
                        B[x][y]);
                if (pa.length + 1 >= p[x][y].length){
                    p[x][y].length = pa.length + 1;
                    p[x][y].next = pa;
                }
            }
        }
        return p[x][y];
    }
}
```

So kann auch dieses Problem in Zeit $O(n^2)$, also linear in der Anzahl der Felder, gelöst werden.

**Aufgabe 5:**

Gegeben sind $m$ Rechtecke und $n$ Punkte.

a) Wir benutzen einen Scanline-Ansatz mit (z.B.) einer vertikalen Scanline, welche von rechts nach links läuft. Haltepunkte sind jeweils die x-Koordinaten von Anfang und Ende der Rechtecke sowie der gegebenen Punkte. Man braucht in der Scanline eine Datenstruktur, welche Aufspiess-Abfragen beantworten kann, z.B. einen Intervall-Baum. Bei den Haltepunkten, die Rechteck-Anfang (bzw.) Ende sind, fügt man das entsprechende y-Intervall in den Intervall-Baum ein (bzw. löscht es). Bei den Haltepunkten, die einem Punkt entsprechen, führt man eine Aufspiess-Abfrage im Intervall-Baum durch: Anstatt alle aufgespiessten Intervalle auszugeben, prüft man aber jeweils nur, ob es ein aufgespiesstes Intervall gibt. Falls man irgendwann eines findet, gibt man es aus und bricht ab.

   Falls man alle Punkte durchgegangen ist, ohne dass je ein Intervall aufgespiesst wurde, weiss man, dass kein Punkt im überlappten Bereich liegt.

b) Der Aufbau des leeren Skeletts des Intervall-Baums benötigt $O(m)$ Zeit. Die Kosten für Einfügen und Löschen der Intervalle im Intervall–Baum betragen je $O(\log m)$, und es gibt $O(m)$ davon. Die Aufspiess-Abfragen kosten ebenfalls jeweils nur $O(\log m)$, wenn man sofort abbricht (also nicht auch noch alle $k$ gefundenen Rechtecke ausgibt), und es gibt $O(n)$ davon. Insgesamt hat man also Kosten von $O((m+n)\log m)$.

   Das Sortieren der $2m+n$ Haltepunkte kann man in höchstens $O((m+n)\log(m+n))$ Zeit erledigen. Der Gesamtaufwand beträgt somit $O((m+n)\log(m+n))$ für diese Lösung.

c) Eine mögliche Lösung ist, alle Punkte in einem 2d-Range-Tree zu speichern, der so abgewandelt ist, dass man effizient die Anzahl Punkte in einem Rechteck bestimmen kann (man kann dazu z.B. in jedem Knoten speichern, wie viele Schlüssel der linke Teilbaum enthält). Dann kann man einfach für jedes Rechteck eine Abfrage machen, und schliesslich das Rechteck mit den meisten Punkten darin ausgeben.

   Den 2d-Range-Tree für die $n$ Punkte baut man in $O(n \log n)$ Zeit auf.

   Ein Query kostet jeweils $O(\log^2 n)$ Zeit, wobei insgesamt $m$ Queries ausgeführt werden. Insgesamt kommt man also auf eine Laufzeit von $O(n\log n + m\log^2 n)$.

   Mittels Fractional Cascading kann man die Laufzeit einer Query auf $O(\log n)$ verbessern, d.h. insgesamt würde die Laufzeit nur noch $O(n\log n + m\log n)$ betragen.