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
Beat Gfeller

Prüfung

Datenstrukturen und Algorithmen

D-INFK

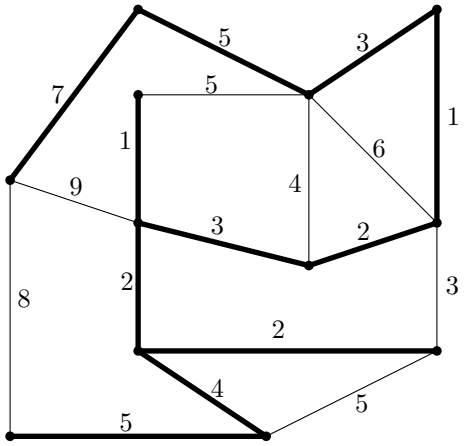
Musterlösung[1]

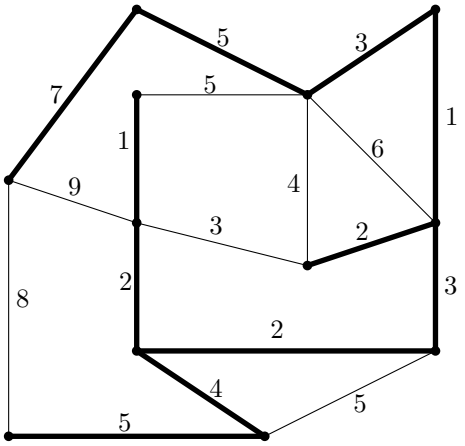
der Prüfung vom 4. Februar 2009

[1] Verfasst von Beat Gfeller. Fragen, Fehler und Bemerkungen bitte an `gfeller@inf.ethz.ch` melden. Danke!

Aufgabe 1:

a) Es gibt genau zwei richtige Lösungen:





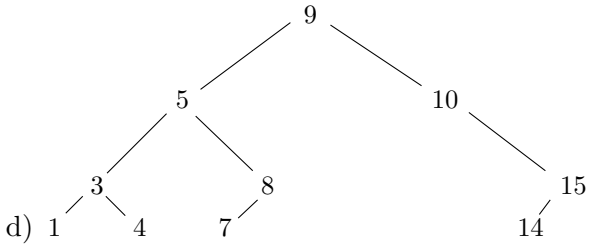
b)

10	12	94	14	07	27	19	99

c)

1	3	5	8	7	11	10	9

d)



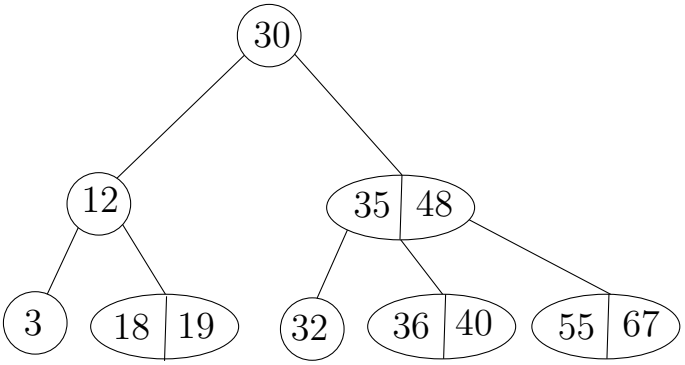
e) Es gibt zwei Lösungen. Wenn man nach links sondiert (wie im Buch stets gemacht), erhält man:

99	23	27	14	30	16		12	19		10
0	1	2	3	4	5	6	7	8	9	10 (Index)

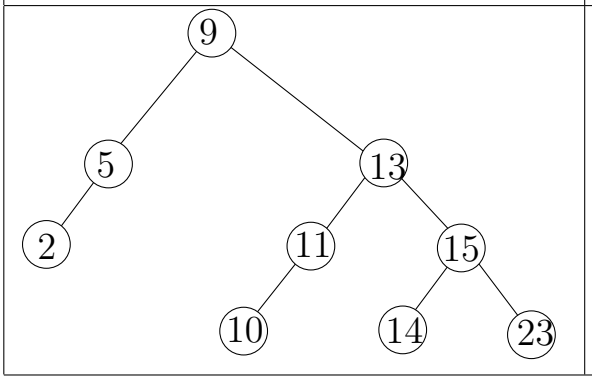
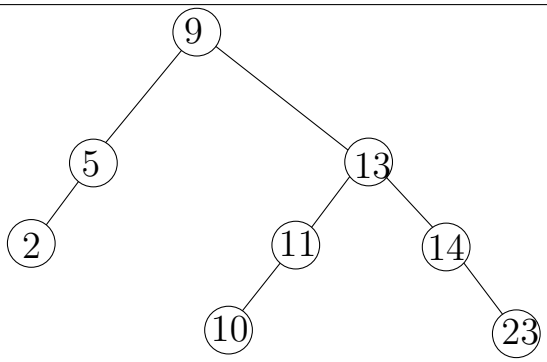
Wenn man jeweils nach rechts sondiert (ist OK, muss aber dann konsistent sein), erhält man:

99	23	12	14		16	27		19	30	10
0	1	2	3	4	5	6	7	8	9	10 (Index)

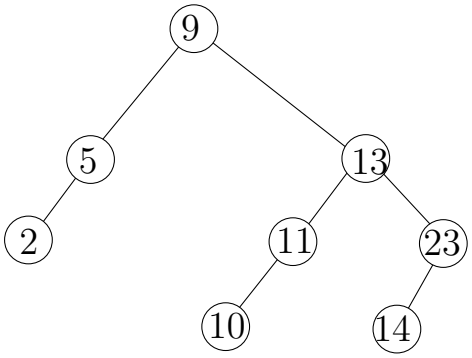
f)



g)

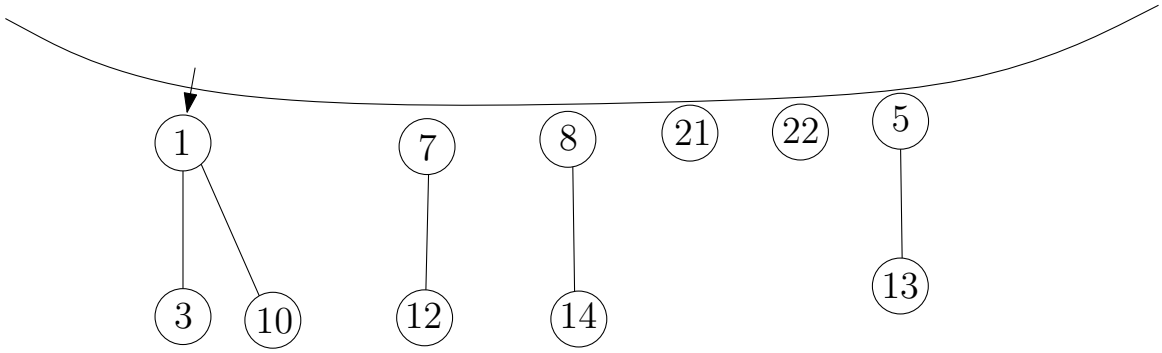
Nach Einfügen von 10:	Nach Löschen von 15:
	

Zweite Lösung für Löschen:



h) Hier gibt es viele Möglichkeiten. Eine korrekte Lösung ist: A, B, C, E, F, G, H, D.

i) Sowohl 42 als auch die beiden markierten Knoten werden in die Wurzelliste umgehängt: (Je nach Markierungsregel werden die abgehängten Knoten in der Wurzelliste unmarkiert oder nicht. Wir akzeptieren beide Varianten als korrekt.)



Aufgabe 2:

a) Die richtige Reihenfolge ist:
$\log(n^5)(=\Theta(\log n)), \quad (\log(n))^3, \quad \sqrt{n}, \quad \frac{n}{\log(n)}, \quad 2^{3\log_2(n)}, \quad 2^{\sqrt{n}}$

b) $T(n) = 2T(n/2) + 4n - 1$

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

$$= 2^3 T\left(\frac{n}{2^3}\right) + 4n \cdot 3 - (1 + 2^2 + 2^3) = 2^i T\left(\frac{n}{2^i}\right) + 4n \cdot i - \sum_{j=0}^{i-1} 2^j$$

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

Wir setzen $i = \log_2(n)$ ein:

$$= nT(1) + 4n \cdot \log_2(n) - n + 1 = n + 4n\log_2(n) + 1.$$

Beweis mit vollständiger Induktion:
Verankerung:
$T(1) = 1 + 4 \cdot 0 + 1 = 2$. OK.
Induktionsschritt (von $n/2$ nach n):
$T(n) = 2T(n/2) + 4n - 1 = 2(n/2 + 4n/2\log_2(n/2) + 1) + 4n - 1 = n + 4n\log_2(n) - 4n + 2 + 4n - 1 = 4n\log_2(n) + 1$. OK.

c) $\Theta(\log n)$

Grund: i durchläuft alle 2er-Potenzen, und j ist stehts die Summe aller bisherigen Werte von i. Wenn es k Iterationen gibt, ist daher $j = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$. Daher ist nach $k = \Theta(\log n)$ Iterationen $j \geq n$.

d) $\Theta(\log n)$

Grund: i wird mindestens jede zweite Iteration halbiert. Daher braucht es höchstens $O(\log n)$ Schleifeniterationen, bis n auf 0 gesunken ist. Zudem braucht es auch mindestens $\Omega(\log n)$ Halbierungen, um n auf 0 zu senken.

e) $\Theta(n \log n)$

Grund: i durchläuft alle 3er-Potenzen, und die innere Schleife macht jeweils $n - 4 \cdot i - 1$ Durchläufe. Dies ergibt eine Gesamtlaufzeit (ungefähr, da es auf +/- eine Schleifeniteration nicht ankommt):

$$\sum_{i=1}^{\log_3 n} (n - 4 \cdot 3^{i-1}) = n\log_3 n - 4 \cdot \sum_{i=1}^{\log_3 n} 3^{i-1} = n\log_3 n - 4 \cdot (n-1) = \Theta(n \log n).$$

Aufgabe 3:

Lösung:

a) $A[i]$ ist genau dann der Median von $A \bigodot B$, wenn $B[n-i] < A[i]$ und $B[n-i+1] > A[i]$. Man kann die Frage also in $O(1)$ beantworten.

Begründung: Damit $A[i]$ Median ist, müssen genau $n-1$ Elemente in $A \bigodot B$ kleiner als $A[i]$ sein, und genau n Elemente grösser als $A[i]$. In A sind genau $i-1$ Elemente kleiner als $A[i]$, daher müssen in B genau $n-i$ Elemente kleiner als $A[i]$ sein. Dies gilt genau wenn $B[n-i] < A[i]$ und $B[n-i+1] > A[i]$.

b) Die Idee ist, die Position, an welcher der Median in A stehen muss, mit binärer Suche zu bestimmen. Entweder wird so der Median gefunden, oder man stellt fest dass A den Median von $A \bigodot B$ nicht enthält. Man beginnt mit einem beliebigen Index, z.B. $m = n/2$, und prüft ob $A[m]$ der Median von $A \bigodot B$ ist. Dazu verwendet man die Lösung aus a). Falls $A[m]$ der Median ist, können wir abbrechen. Falls $A[m]$ nicht der Median ist, weiss man zudem, ob der wahre Median grösser oder kleiner als m sein muss. Daher muss man nur noch auf der entsprechenden Seite in A weitersuchen, usw. Die Laufzeit dieses Verfahrens ist $O(\log n)$, weil man jeden Schritt mit a) in konstanter Zeit ausführen kann, und jeder Schritt das verbleibende Intervall halbiert.

c) Man kann einfach die Lösung aus b) zweimal benutzen: zuerst prüft man, ob der Median von $A \bigodot B$ in A liegt (wie in b) beschrieben). Falls ja, sind wir fertig, falls nein, muss der Median in B liegen. Jetzt wenden wir dasselbe Verfahren auf B an, und finden so heraus, welches Element in B der Median ist. Die Laufzeit bleibt so $O(\log n)$.

Aufgabe 4:

a) Die Grundidee ist, einen Scanline-Ansatz zu verwenden. Wir benutzen eine vertikale Scanline, die von links nach rechts läuft. Haltepunkte sind Anfangs- und Endpunkte von Segmenten, sowie die x-Position, an der die Kugel startet.

Beim Anfang eines Segments fügt man dieses in die Scanline ein. Beim Ende eines Segments entfernt man es aus der Scanline. Zudem bestimmt man beim unteren Ende jedes Segments (dies kann entweder der Anfang oder das Ende sein), welches Segment direkt unter dem Ende liegt, und speichert sich einen Pointer darauf ab (oder `void`, wenn es keines darunter gibt). Beim Haltepunkt x (wo die Kugel startet), bestimmt man das oberste Segment in der Scanline: darauf fällt die Kugel zuerst. Ein (wichtiges) Detail: die Scanline implementiert man als AVL-Baum, wobei in jedem Knoten die Geradengleichung des Segments gespeichert sein muss: beim Suchen muss man jeweils die Geradengleichung an der Scanlineposition auswerten, damit der korrekte Einfüge-/Lösch-Ort gefunden wird.

Nach dem Scanline-Durchlauf hat man also für jedes Segment s einen Pointer, der angibt auf welches Segment die Kugel als nächstes fällt, wenn sie das Segment s hinunter rollt. Zudem kennt man das Segment, auf das die Kugel zuerst fällt. Man kann also den Weg der Kugel nun Segment für Segment verfolgen.

b) Es gibt $O(n)$ Haltepunkte, die man sortieren muss. Mit einem AVL-Baum für die Scanline kann man jedes Einfügen/Löschen in $O(\log n)$ erledigen. Wenn man zudem die Blätter im Baum verkettet, findet man in $O(1)$ Zeit das Nachbar-Segment (auf das die Kugel fällt).

Das Durchlaufen am Ende kostet nur noch $O(n)$.

Die Laufzeit beträgt insgesamt also $O(n \log n)$.

3 P c) Beschreiben Sie Ihre Lösung für Teilaufgabe a) in Pseudocode. Ihr Pseudocode muss sich an eine der folgenden Sprachen anlehnen: Eiffel, Java, C++. Sie dürfen dabei grundlegende Algorithmen und Datenstrukturen (Sortieren, Balancierter Suchbaum, etc.) verwenden, ohne deren Code aufzuschreiben.

Eiffel-Pseudocode:

```
haltepunkte, scanline: AVLTree[SEGMENT]
next_segment: ARRAY[SEGMENT]
s: SEGMENT
start: SEGMENT -- erstes Segment, auf das die Kugel fällt

-- Haltepunkte einfügen:
create haltepunkte.make (N)
from rinnen.first
until rinnen.after
loop
  haltepunkte.insert ( rinnen.item.xl, rinnen.item )  -- insert: (key, data)
  haltepunkte.insert ( rinnen.item.xr, rinnen.item )
  rinnen.forth
end

haltepunkte.insert ( x, y ) -- Haltepunkt für die Kugel

create scanline.make
```

```
from haltepunkte.first
until haltepunkte.after
loop
  s := haltepunkte.item
  if s.isBegin then
    scanline.insert (s)
    if s.yl <= s.yr then
      s.set_lower ( scanline.get_segment_below(s) )
    end
  else if s.isEnd then
    if s.yl > s.yr then
      s.set_lower ( scanline.get_segment_below(s) )
    end
    scanline.delete (s)
  else -- Haltepunkt ist Kugelposition zu Beginn
    start := scanline.get_uppermost_below (y)
  end

  haltepunkte.forth
end

-- Jetzt die Kugelbahn verfolgen:
from
  s := start
until
  s = void
loop
  io.put_string("Kugel fällt auf Segment " + s.out)
  s := s.get_lower
end
```

d) Grundsätzlich geht man gleich vor wie in a), mit dem Unterschied, dass man für beide Enden jedes Segments feststellt, welches Segment direkt darunter liegt (2 Pointer pro Segment). Man speichert aber nur solche Pointer ab, die einem Sprung von höchstens H entsprechen, die anderen Pointer setzt man auf `void` (ob die Höhe nicht mehr als H ist, findet man leicht aus den Koordinaten der Segmente heraus). Zudem definiert man künstlich ein "Boden"-Segment, das auf dem Boden liegt und das breit genug ist, dass alle anderen Segmente vollständig darüber liegen. Dies führt zu einem gerichteten Graphen, in dem jedes Segment ein Knoten ist, der höchstens zwei ausgehende Kanten hat. In diesem Graphen sucht man nun einen Pfad vom Startsegment zum Boden-Segment, z.B. mit einer Tiefen- oder Breitensuche, welche $O(n)$ Zeit kostet (Dijkstra's Algorithmus für kürzeste Wege funktioniert auch, kostet aber $O(n \log n)$).

So erreicht man insgesamt eine Laufzeit von $O(n \log n)$, da die Anzahl Kanten und die Anzahl Knoten beide $O(n)$ sind.

Aufgabe 5:

a) Der Aufruf von `risk(place,time)` liefert das Gesamtrisiko einer sichersten Fahrt, welche genau `time` Tage dauert, beim Ort 1 startet und bis zum Ort `place` fährt.

Eiffel-Pseudocode:

```
G: ARRAY2[INTEGER]
n, T: INTEGER

risk(place: INTEGER, time: INTEGER): INTEGER is
do
  if time = 1 then
    if place = 1 then
      Result := G[1,1]
    else
      Result := infinity
    end
  else -- time > 1
    Result := risk( place, time-1 )
    if place > 1 then
      Result := Result.min( risk(place-1, time-1) )
    end
    Result := Result + G[place,time]
  end
end
start is
do
  io.put_integer( risk(n,T) )
end
```

Die Laufzeit dieses Algorithmus ist im worst-case $\Omega(2^T)$, denn wenn z.B. $n = T$, dann wird gibt es bis zu einer Rekursionstiefe von $n = T$ jeweils genau zwei rekursive Aufrufe. Die Laufzeit ist auch durch $O(2^T)$ beschränkt, weil die Rekursionstiefe $O(T)$ ist und jedesmal maximal zwei rekursive Aufrufe erfolgen. Achtung: die Laufzeit ist im allgemeinen nicht durch $O(2^n)$ beschränkt (z.B. für $n = 1$ ist die Laufzeit noch $O(T)$).

Man kann den rekursiven Algorithmus auch etwas effizienter gestalten, indem man Aufrufe für unerreichbare Felder vermeidet. Dann ist die Laufzeit unter Umständen kleiner.

b) Das Tableau R ist $n \times T$ gross (genau wie `G`), wobei in $R[i, j]$ das Gesamtrisiko einer sichersten Fahrt steht, welche am Tag j an Ort i ist.
Initialisierung: $R[1, 1] = G[1, 1]$, und für $1 < i \leq n$: $R[i, 1] := \infty$. Zudem berechnet man für alle $2 \leq j \leq n$: $R[1, j] := G[1, j] + R[1, j - 1]$.

Ausfüllen: jeweils Tag für Tag. Regel für Tag j: $R[i, j] := G[i, j] + \min(R[i, j-1], R[i-1, j-1])$.

Die Antwort steht dann in $R[n, T]$. Die Laufzeit ist $O(nT)$, weil jedes Feld in konstanter Zeit ausgefüllt werden kann.

c) Das Tableau ist dasselbe wie in b), die Initialisierung auch. Das Ausfüllen ist so: jeweils Tag für Tag. Regel für Tag j: $R[i, j] := G[i, j] + \min_{s \in 0 \ldots \min(k, i-1)}\{R[i - s, j - 1]\}$.

Die Laufzeit ist $O(nTk)$, weil für jedes Feld $k + 1$ Möglichkeiten ausprobiert werden müssen.