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 20. August 2008

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

Aufgabe 1:

Hinweise:

1. In dieser Aufgabe sollen Sie **nur die Ergebnisse** angeben. Diese können Sie direkt bei den Aufgaben notieren.
2. Sofern Sie die Notationen, Algorithmen und Datenstrukturen aus der Vorlesung "Datenstrukturen & Algorithmen" verwenden, sind Erklärungen oder Begründungen nicht notwendig. Falls Sie jedoch andere Methoden benutzen, müssen Sie diese **kurz** soweit erklären, dass Ihre Ergebnisse verständlich und nachvollziehbar sind.
3. Als Ordnung verwenden wir für Buchstaben die alphabetische Reihenfolge, für Zahlen die aufsteigende Anordnung gemäss ihrer Grösse.

1 P a) Zeichnen Sie die Liste, welche entsteht, wenn die Move-to-Front-Regel benutzt wird, und auf folgende Elemente zugegriffen wird (in dieser Reihenfolge): Q, F, X, Q, Z, V, F.

Liste vor den Zugriffen: $\mathrm{X} \rightarrow \mathrm{V} \rightarrow \mathrm{Z} \rightarrow \mathrm{F} \rightarrow \mathrm{B} \rightarrow \mathrm{Q}$

Liste nach den Zugriffen: $\mathrm{F} \rightarrow \mathrm{V} \rightarrow \mathrm{Z} \rightarrow \mathrm{Q} \rightarrow \mathrm{X} \rightarrow \mathrm{B}$

1 P b) Führen Sie auf dem folgenden Array die ersten zwei Schritte des Sortierverfahrens *Natürliches 2-Wege-Mergesort* durch.

7	6	8	17	2	20	5	14	3	1

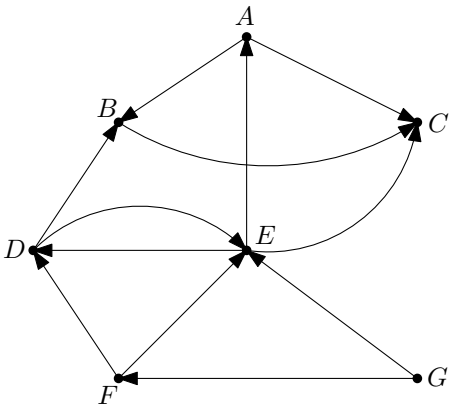
nach Schritt 1:

6	7	8	17	2	5	14	20	1	3

nach Schritt 2:

2	5	6	7	8	14	17	20	1	3

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



Lösung: G, E, A, B, C, D, F und viele andere..

1 P d) Fügen Sie die Schlüssel $19, 14, 30, 10, 12, 99$ in dieser Reihenfolge mittels Offenem Hashing in die folgende Hashtabelle ein (die bereits ein paar Schlüssel enthält), und benutzen Sie dabei Double Hashing. Die zu verwendende Hash-Funktion ist $h(k) = k \mod 11$, und für das Sondieren die Hashfunktion $h'(k) = 1 + (k \mod 9)$.

11			3		16				20	0
0	1	2	3	4	5	6	7	8	9	10

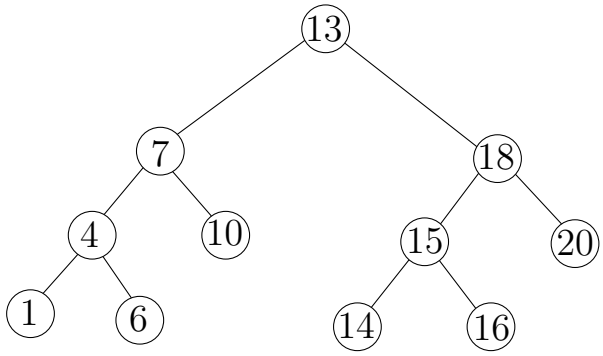
Mit Sondieren nach links:

11	12	14	3	30	16	10	99	19	20	0
0	1	2	3	4	5	6	7	8	9	10

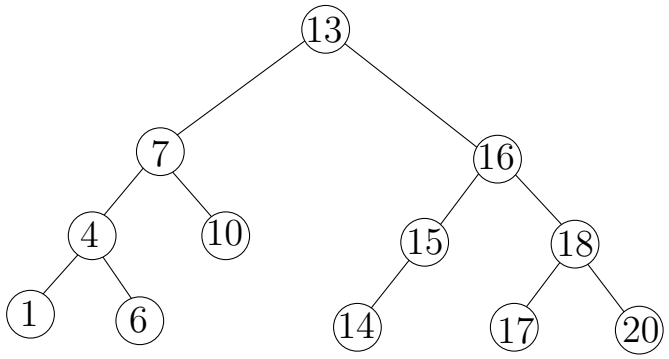
Alternativ mit Sondieren nach rechts:

11	30	12	3	14	16	99	10	19	20	0
0	1	2	3	4	5	6	7	8	9	10

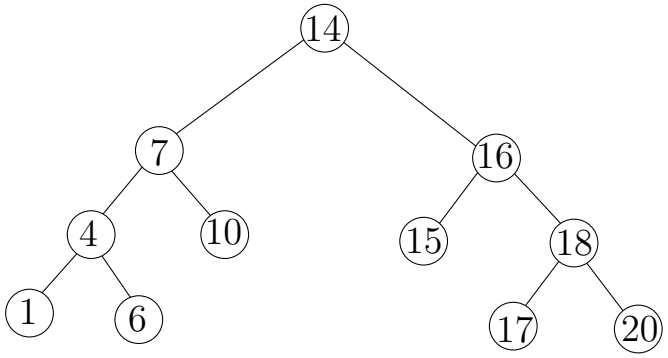
1 P e) Fügen Sie in den untenstehenden AVL-Baum den Schlüssel 17 ein, und löschen Sie *danach* den Schlüssel 13 daraus.



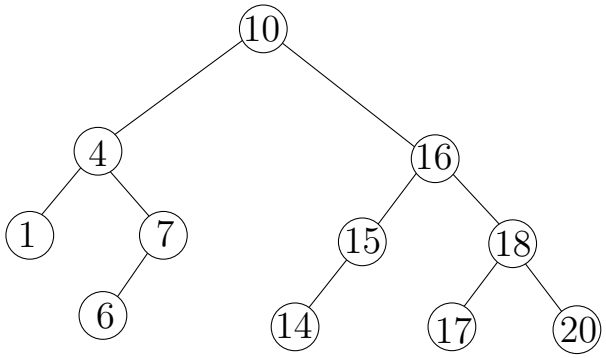
Nach dem Einfügen von 17 und Rebalancieren sieht der Baum wie folgt aus:



Für das Löschen von 13 danach gibt es zwei Möglichkeiten. Wenn der symmetrische Vorgänger als Ersatz für den gelöschten Knoten benutzt wird, entsteht der folgende AVL-Baum:

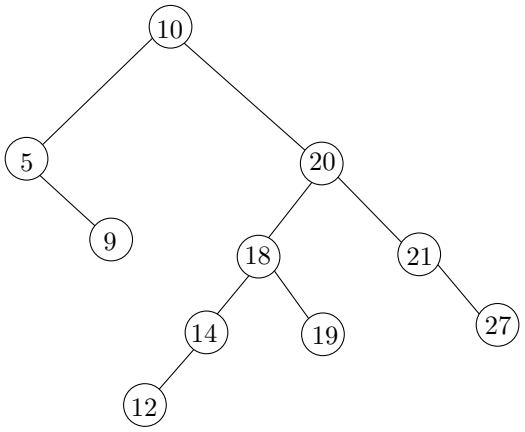


Wenn hingegen der symmetrische Nachfolger benutzt wird, entsteht (nach Rebalancieren) folgender AVL-Baum:



Beide dieser Lösungen werden akzeptiert.

1 P f) Zeichnen Sie den binären Suchbaum, dessen Postorder-Traversierung die Folge 9, 5, 12, 14, 19, 18, 27, 21, 20, 10 ergibt.

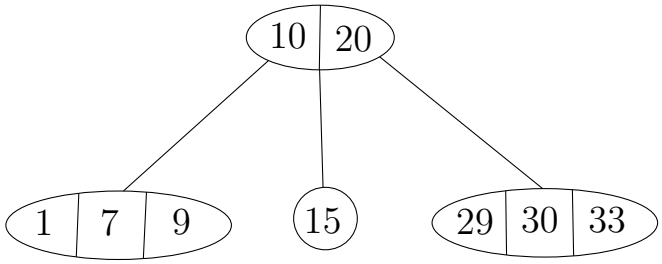


1 P g) Das untenstehende Array enthält die Elemente eines Min-Heaps in der üblichen Form gespeichert. Wie sieht das Array aus, nachdem das Minimum entfernt wurde und die Heap-Bedingung wieder hergestellt wurde?

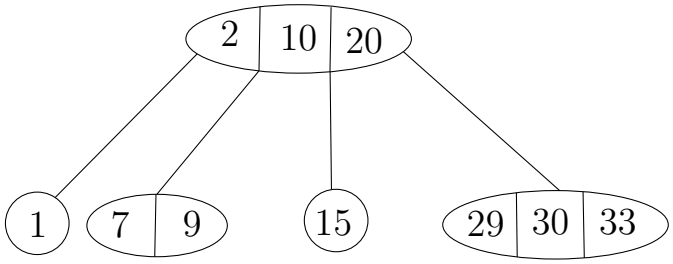
3	8	4	10	16	6	7	27	14	20
1	2	3	4	5	6	7	8	9	10

4	8	6	10	16	20	7	27	14	

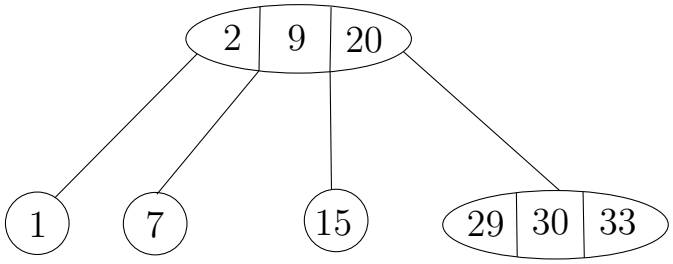
1 P h) Gegeben sei der folgende B-Baum der Ordnung 4. Fügen Sie in diesen zuerst den Schlüssel 2 ein, und löschen Sie danach den Schlüssel 10 aus dem B-Baum.



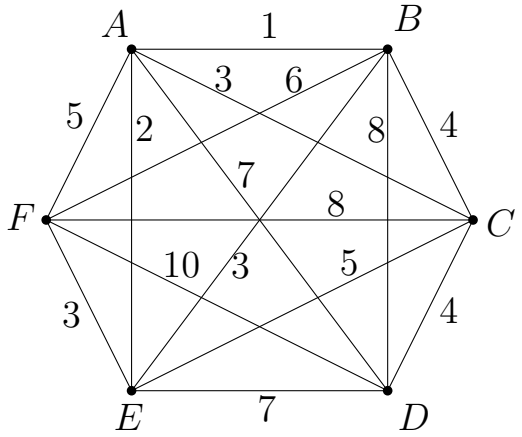
Nach Einfügen von 2:



Nach Löschen von 10:



1 P i) Geben Sie im untenstehenden gewichteten Graphen mittels des Approximationsalgorithmus, der auf dem minimalen Spannbaum basiert, eine Rundreise an, deren Länge höchstens das 2-fache der Länge einer optimalen Rundreise ist. Die Kantengewichte erfüllen die Dreiecksungleichung. Sie müssen die Approximations-Schranke nicht beweisen. Beginnen Sie die Rundreise beim Knoten A.



z.B. A, B, C, D, E, F, A und viele weitere (der Spannbaum ist aber eindeutig).

Aufgabe 2:

a) $\frac{\log^5 n}{\sqrt{n}}$, $\log(n^8)(= O(\log n))$, $2^{\log_2 n}(= n)$, $\sqrt{n^3}(= n^{1.5})$, n^2, $\prod_{i=1}^{n} i^2(= (n!)^2)$.

b) Teleskopieren:

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + \frac{n}{4} + 2 \\ &= 4\Big(4T\left(\frac{n}{4}\right) + \frac{n}{8} + 2\Big) + \frac{n}{4} + 2 \\ &= 4\Big(4\Big[4T\left(\frac{n}{8}\right) + \frac{n}{16} + 2\Big] + \frac{n}{8} + 2\Big) + \frac{n}{4} + 2 \\ &= 4^i T\left(\frac{n}{2^i}\right) + \frac{n}{4} \cdot \sum_{j=0}^{i-1} 2^j + 2 \cdot \sum_{k=0}^{i-1} 4^k \\ &= 4^i T\left(\frac{n}{2^i}\right) + \frac{n}{4} \cdot \frac{2^i - 1}{2 - 1} + 2 \cdot \frac{4^i - 1}{3} \qquad \text{(Summenformel angewandt)} \end{aligned}$$

Setze $i = \log_2(n)$ ein, und vereinfache:

$$= 4n^2 + \frac{n}{4} \cdot (n-1) + 2 \cdot \frac{n^2 - 1}{3} = (4 + \frac{1}{4} + \frac{2}{3})n^2 - \frac{n}{4} - \frac{2}{3} = \frac{59}{12}n^2 - \frac{n}{4} - \frac{2}{3}$$

Beweis durch vollständige Induktion:

Verankerung: $\frac{59}{12}1^2 - \frac{1}{4} - \frac{2}{3} = 4$.

Induktionsschritt: $n/2 \to n$.

$T(n) = 4T\left(\frac{n}{2}\right) + \frac{n}{4} + 2 = 4(\frac{59}{12}n^2/4 - \frac{n}{8} - \frac{2}{3}) + \frac{n}{4} + 2 = \frac{59}{12}n^2 - \frac{n}{2} + \frac{n}{4} - \frac{8}{3} + 2 = \frac{59}{12}n^2 - \frac{n}{4} - \frac{2}{3}$.

c) Anzahl Schritte ist $\sum_{i=1}^{n} i = \Theta(n^2)$.

d) Anzahl Schritte ist $\Theta(n)$: j läuft linear von 1 bis n, welches die dominate Komponente darstellt. i springt von 1 auf n in $\log_2(n)$ vielen Schritten, was keinen Einfluss auf die Gesamtlaufzeit hat.

e) Anzahl Schritte ist $\Theta(n)$: i läuft von 1 nach n, was $\Theta(n)$ Schritte benötigt. Man beachte, dass s jeweils verdoppelt wird, d.h. s erhält die Werte $1, 2, 4, 8, \ldots, n$, insgesamt $O(\log n)$ viele. Die Anzahl Schritte die j durchläuft ist daher $\sum_{i=1}^{\log_2(n)} 2^i = \frac{2^{\log_2(n)}-1}{2-1} = \Theta(n)$.

Aufgabe 3:

a) Wir speichern die Punkte in einem balancierten Suchbaum, in dem die Punkte nach x-Koordinate geordnet sind. Da sich die x-Werte nicht ändern, ist dieser Baum fix. Wir können ihn z.B. wie einen 1-dimensionalen Range-Tree aufbauen, oder auch einen AVL-Baum benutzen. Nachdem dieser Baum berechnet wurde, speichern wir in jedem Knoten i eine zusätzliche Ganzzahl ab, die wir mit Y_i bezeichnen.

   Die Invariante besagt, dass "Y_i = die grösste y-Koordinate eines Punktes, welcher im Teilbaum mit Wurzel i vorkommt."

b) Das Maximum kann gefunden werden, indem man ein 1-dimensionales Range-Query durchführt (wie in einem 1d-Range-Tree): D.h. man sucht im Baum nach $x_{\min}$ sowie nach $x_{\max}$, und besucht somit insgesamt $O(\log n)$ Knoten, deren Teilbäume alle Punkte mit x-Koordinaten im gesuchten Bereich einhalten (die Knoten auf den Suchpfaden können entweder innerhalb oder ausserhalb liegen, diese müssen einzeln geprüft werden, und deren y-Koordinate separat mit dem bisherigen Maximum verrechnet werden). Genauer: die Suche geht zuerst zum "split-Punkt", wo sich die beiden Suchpfade trennen. Wenn ab da die Suche nach $x_{\min}$ in den linken Teilbaum absteigt, weiss man dass alle Punkte im rechten Teilbaum im gesuchten x-Bereich liegen. Analog für die Suche nach $x_{\max}$ (nur spiegelverkehrt). Anstatt nun sämtliche Punkte in diesen $O(\log n)$ Teilbäumen durchzugehen, berechnet man einfach das Maximum der Y_i-Werte in all diesen Knoten. Somit erhält man den gesuchten y-Wert in Zeit $O(\log n)$.

c) Für das Update sucht man zuerst den Knoten mit x-Koordinate x_i im Baum. Dann ändert man dessen y-Koordinate auf y_{new}. Dadurch ändert sich möglicherweise das Maximum in einigen Teilbäumen (aber nur in denen, welche den Punkt $(x_i, \cdot)$ enthalten. Um nun die Invariante wiederherzustellen, muss man den Suchpfad von unten nach oben zurücklaufen, und dabei alle Y_i-Werte der Knoten auf diesem Pfad gegebenenfalls updaten. Dies geht pro Knoten in konstanter Zeit, wie folgt: Am Knoten j berechnet man einfach $Y_j = \max\{Y_l, Y_r, y_j\}$, wobei l der linke Sohn von j ist (falls vorhanden), und r der rechte Sohn von j ist (falls vorhanden, einer von beiden muss vorhanden sein). Für Y_l und Y_r gilt die Invariante schon, weil wir die Knoten von unten nach oben updaten. y_j ist natürlich einfach die y-Koordinate des in Knoten j gespeicherten Punkts. Da jeder Knoten in $O(1)$ updatet werden kann, benötigt das Update somit nur $O(\log n)$ Zeit.

Aufgabe 4:

a) Die Grundidee ist dynamische Programmierung: man berechnet Bottom-Up für jeden Knoten i zwei Werte:

1) Das bestmögliche Gewicht eines Matchings in i's Teilbaum falls die Kante i benutzt wird (enthält insbesondere w_i). Dieses Gewicht erhält man, indem man zum Gewicht w_i das bestmögliche Gewicht jedes Kindes von i dazuzählt, unter der Bedingung dass die Kante vom Kind zu i nicht benutzt wird.
2) Das bestmögliche Gewicht eines Matchings in i's Teilbaum falls die Kante i nicht benutzt wird. Dieses Gewicht erhält man, indem man für jedes Kind j von i die folgende Summe berechnet: w_j plus das bestmögliche Gewicht aller anderen Kinder, unter der Bedingung dass die Kante von jedem Kind $k \neq j$ zu i nicht benutzt wird. Zu beachten ist hier, dass man auch die Möglichkeit berücksichtigen muss, dass man weder die Kante i noch eine Kante zu einem Kind j benutzt (d.h. im bestmöglichen Matching ist vielleicht keine zum Knoten i adjazente Kante enthalten!).

Gegeben diese zwei Werte für alle c_i Kinder von Knoten i, kann man 1) und 2) für Knoten i in Zeit $O(c_i)$ berechnen (siehe unten). Die Gesamtlaufzeit ist somit $O(n + \sum_{i \in V} c_i) = O(n)$.

b)
```
match(T: ARRAY[INTERVALL]; W: ARRAY[INTEGER]): INTEGER is
   local
      i, j: INTEGER
      -- bestmögliches Gewicht eines Matchings in i's Teilbaum
      -- falls Kante i benutzt / nicht benutzt:
      W1, W2: ARRAY[INTEGER]
      W2sum: INTEGER
      Wnew: INTEGER
   do
      create W1.make (T.lower, T.upper)
      create W2.make (T.lower, T.upper)

      from  -- alle Knoten i in Postorder durchlaufen:
         i := T.upper
      until
         i < T.lower
      loop
         -- alle Kinder von Knoten i durchgehen: berechne W1[i] und W2sum
         from
            W2sum := 0 -- zählt die W2-Werte aller Kinder von Knoten i zusammen
            j := T[i].min
         until
            j > T[i].max
         loop
            W2sum := W2sum + W2[j]
            j := j + 1
         end
         W1[i] := W[i] + W2sum
```

```
            -- alle Kinder von Knoten i nochmals durchgehen: jetzt wird W2[i] berechnet.
            from
               W2[i] := 0
               j := T[i].min
            until
               j > T[i].max
            loop
               -- berechne Gewicht, wenn Kante j genommen wird:
               Wnew := W2sum - W2[j] + W1[j].max( W2[j] )
               if Wnew > W2[i] then
                  W2[i] := Wnew
               end
               j := j + 1
            end
            i := i - 1
         end

      Result := W2[T.lower] -- weil Kante w_1 nicht existiert, kommt W1 nicht in Frage.

   end -- match
```

c) Für jeden Teilbaum i, für den man die beiden Gewichte 1) und 2) berechnet hat (siehe a)), speichert man zusätzlich die Liste der jeweils gewählten Kanten ab. Beim Zusammensetzen eines Matchings aus den Teilmatchings von Teilbäumen kann man die entsprechenden Listen von Kanten einfach zusammensetzen. So erhält man schliesslich zum maximalen Gewicht zusätzlich ein entsprechendes Matching als Liste von Kanten.

Aufgabe 5:

a) Grundidee: fasse die Rechtecke als 4-dimensionale Punkte auf, und speichere sie in einem (4-dimensionalen) Range-Tree. Dann kann man für jedes Rechteck einzeln mit einem Range-Query feststellen, ob es ein anderes Rechteck gibt, das in ihm enthalten ist: dazu führt man einfach ein Range-Query mit den Rechteckgrenzen durch, und prüft ob (mindestens) ein "Punkt" darin enthalten ist.

   Genauer: für jedes Rechteck R_i repräsentiert man als (4-dimensionalen) Punkt mit den Koordinaten (l_i, r_i, u_i, o_i). Nun baut man einen Range-Tree für diese n Punkte auf. Dann führt man für jedes Rechteck R_i die folgende Anfrage aus: "Gibt es einen Punkt (l, r, u, o) mit $l_i < l < \infty, -\infty < r < r_i, u_i < u < \infty, -\infty < o < o_i$?" Dies ist nichts anderes als ein 4-dimensionales Range-Query, wobei man, statt alle Punkte im gegebenen Intervall auszugeben, stoppen kann, sobald man einen Punkt gefunden hat: Denn wenn man einen Punkt im gesuchten Intervall gefunden hat, gibt es ein Rechteck, welches ganz in einem anderen enthalten ist.

b) Laufzeit: $O(n \log^3 n)$ für's Aufbauen des Range-Trees, und dann werden n Queries mit je Laufzeit $O(\log^3 n)$ [mit Fractional Cascading] durchgeführt. Insgesamt also Laufzeit $O(n \log^3 n)$.

   Um zu sehen, dass die Queries in $O(\log^3 n)$ gehen, überlegt man sich folgendes: ein Query, bei dem kein Punkt im gegebenen Intervall liegt, benötigt Zeit $O(\log^3 n)$. Ein Query, bei dem viele Punkte (z.B. k) im gegebenen Intervall liegen, würde eigentlich Zeit $O(\log^3 n + k)$ benötigen. Da wir aber in diesem Fall abbrechen, sobald *ein* Punkt gefunden wurde (da wir dann wissen, dass die Antwort "Ja" sein muss), benötigen wir auch in diesem Fall nur $O(\log^3 n)$ Zeit.

Bemerkung: Man auch versuchen, die Aufgabe mit einem Scanline-Ansatz zu lösen. So kann man in "gutartigen" Instanzen (wenn sich nicht zu viele Rechtecke überlappen) eine Laufzeit in $o(n^2)$ erreichen. Es scheint jedoch schwierig, mit dem Scanline-Ansatz eine Lösung zu konstruieren, welche auch im worst-case Laufzeit $o(n^2)$ hat.