Informatik B – Sommersemester 2007

Musterlösung zur Klausur vom 20.07.2007

Aufgabe 1: Graphen und Graphalgorithmen $2+3+2+(3)$ **Punkte**

Für eine beliebige positive, ganze Zahl n definieren wir einen Graphen $G_n = (V_n, E_n)$ auf der Knotenmenge $V_n = \{1, 2, \ldots, n\}$, wobei zwei Knoten genau dann durch eine Kante verbunden sind, wenn der Betrag ihrer Differenz beim Teilen durch 3 den Rest 2 lässt:

$$E_n = \{\, \{i, j\} \mid |i - j| \bmod 3 = 2\}$$

a) Ergänzen Sie das linke Bild zu einer vollständigen Zeichnung des Graphen G_8 und ordnen Sie den Kanten die folgenden Gewichte zu:

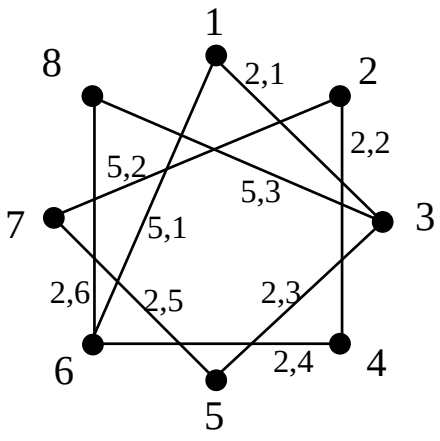
$$w(\{i, j\}) = |i - j| + \frac{\min(i, j)}{10}$$

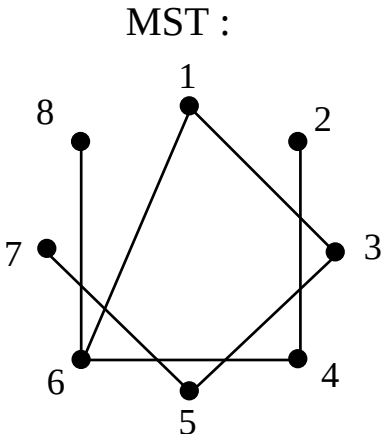
b) Konstruieren Sie mit dem Algorithmus von **Prim** einen MST von G_8 mit der gegebenen Gewichtsfunktion und Startknoten 1. Zeichnen sie in das rechte Bild den MST ein und geben Sie die Reihenfolge an, in der die Knoten in die Startkomponente aufgenommen werden.

c) Welche der Graphen G_n sind zusammenhängend und welche nicht? Geben Sie dafür eine kurze Begründung.

d) Zusatzaufgabe: Zeigen Sie, dass $|E_n| = \Omega(n^2)$.

Lösung 1.a und 1.b:





Reihenfolge der Knoten bei MST nach Prim: $1, 3, 5, 7, 6, 4, 2, 8$

Lösung 1.c: Aus der Definition von E_n folgt, dass alle geraden Knoten (in geordneter Reihenfolge) durch einen Weg verbunden sind und Gleiches gilt für die ungeraden Knoten. Da die "kleinste" Kante, die einen ungeraden Knoten mit einem geraden verbindet, die Kante $\{1, 6\}$ ist, sind die Graphen G_2, G_3, G_4 und G_5 nicht zusammenhängend. Dagegen sind die Graphen G_n für alle $n \geq 6$ zusammenhängend.

Lösung 1.d: Zur Erleichterung des Zählens geben wir allen Kanten eine Richtung, nämlich vom kleineren zum größern Knoten. Sei $i \leq n-2$ ein Knoten in G_n, dann gehen von i die folgenden Kanten aus: $(i, i+2), (i, i+5), \ldots (i, i+3k_i+2)$ für das größte k_i, das $i + 3k_i + 2 \leq n$ erfüllt. Daraus ergibt sich $k_i = \left\lfloor \frac{n-i-2}{3} \right\rfloor$, d.h. von i gehen $k_i + 1$ Kanten aus. Damit ist

$$|E_n| = \sum_{i=1}^{n-2} \left(\left\lfloor \frac{n-i-2}{3} \right\rfloor + 1 \right) = \sum_{i=1}^{n-2} \left\lfloor \frac{n-i+1}{3} \right\rfloor$$

Wählt man aus dieser Summe beginnend mit dem ersten Summanden jeden dritten aus, so ergibt sich

$$|E_n| \geq \left\lfloor \frac{n}{3} \right\rfloor + \left(\left\lfloor \frac{n}{3} \right\rfloor - 1 \right) + \left(\left\lfloor \frac{n}{3} \right\rfloor - 2 \right) + \ldots + 2 + 1 = \frac{\left\lfloor \frac{n}{3} \right\rfloor \cdot \left(\left\lfloor \frac{n}{3} \right\rfloor + 1 \right)}{2} \in \Omega(n^2)$$

Zusatzkommentar: Die hier vorgestellte Abschätzung ist natürlich nur ein möglicher Weg zum Ziel. Eine anderer (naheliegender) Ansatz führt über die bekannte Identität $2|E| = \sum_{v \in V} deg(v)$. Man begründet zuerst, dass in G_n jeder Knoten mindesten den Grad $\left\lfloor \frac{n-2}{3} \right\rfloor$ hat und erhält damit

$$|E_n| \geq \frac{n}{2} \left\lfloor \frac{n-2}{3} \right\rfloor \in \Omega(n^2).$$

Aufgabe 2: **Java** $4+3$ **Punkte**

a) Welche Bildschirmausgabe erhält man bei Auruf der folgenden Methode `test`?

```
public static void test(){
    int[] a = {4,3,2,1};
    int[] b = {0,3,6,9};
    int[] c = a;
    int[] d = (int[]) a.clone();
    int[][] A = {a,b,c,{1,2,3,4}};
    int[][] B = {a,b,c,d};

    A[2][1] *= 2;
    for(int i=0; i<4; i++) B[i][1] += 1;

    System.out.println( "A[1][1] = " + A[1][1]);
    System.out.println( "a[1] = " + a[1]);
    System.out.println( "(a == d) = " + (a==d));
    System.out.println( "a.equals(c) = " + (a.equals(c)));
    System.out.println( "a.equals(d) = " + (a.equals(d)));
    System.out.println( "A[2] == B[2] = " + (A[2]== B[2]));
}
```

Lösung: Die Methode erzeugt die folgende Bildschirmausgabe:

```
A[1][1] = 4
a[1] = 8
(a == d) = false
a.equals(c) = true
a.equals(d) = false
A[2] == B[2] = true
```

b) Von einen `int`–Array `A` der Länge n sei bekannt, dass alle Einträge aufsteigend geordnet sind. Ausserdem setzen wir voraus, dass `A[0]` mit dem Wert 1 und `A[n-1]` mit dem Wert $n-1$ belegt sind. Definieren Sie eine Methode
`public int findDoubleOccur(int[] A)`,
die für ein Feld mit den beschriebenen Eigenschaften mit Laufzeit $O(\log n)$ eine Zahl findet, die in dem Feld (mindestens) doppelt auftritt. Um sich Hilfsfunktionen zu ersparen, können Sie auch die Parameterliste der Funktion erweitern.

Lösung: Man sollte hier die Idee der Binärsuche anwenden. Dazu muss man nur die Anfangsbedingung, die das mehrfache Auftreten einer Zahl erzwingt, etwas allgemeiner fassen, nämlich als $A[n-1]-A[0] < (n-1)-0$. Auch wenn man an Stelle von $n-1$ und 0 zwei Indizes j und i (mit $i<j$) einsetzt, sichert die Bedingung $A[j]-A[i] < j-i$ (zusammen mit der Ordnungseigenschaft) das doppelte Auftreten einer Zahl in dem Array–Bereich zwischen i und j. Wenn man ein solches Intervall in der Mitte teilt, muss sich diese Bedingung auf mindestens eines der beiden Teilintervalle vererben. Man kann also die folgende Hilfsfunktion definieren:

```
public int f(int[] A, int i, int j){
  if (A[i] == A[j]) return A[i];
  k = (i+j)/2;
  if (A[j] - A[k] < j-k) return f(A,k,j);
     else   return f(A,i,k);
}
```

Die Funktion `findDoubleOccur` muss nun nur noch die Hilfsfunktion mit dem Anfangs- und Endindex des Arrays A aufrufen:

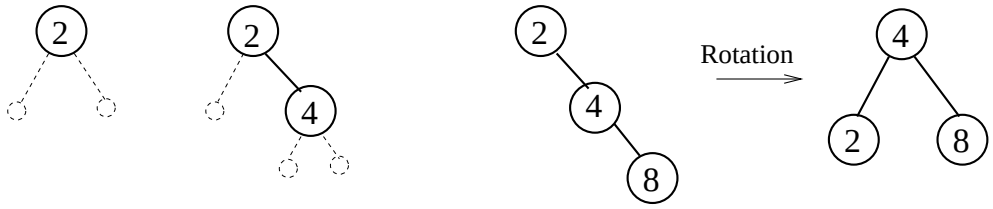
```
public int findDoubleOccur(int[]  A) {
    return f(A, 0, A.length -1);
}
```

Aufgabe 3: **Binäre Bäume** $4 + 4 + 2$ **Punkte**

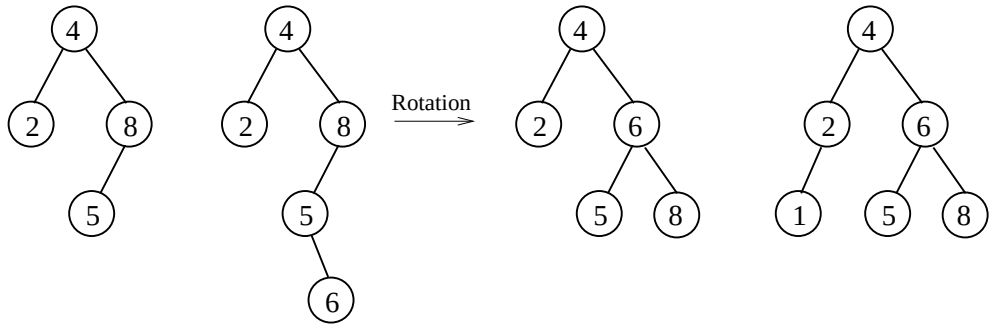
a) Bauen Sie einen AVL–Baum auf, in den nacheinander die Schlüssel $2, 4, 8, 5, 6, 1, 9$ eingefügt werden. Löschen Sie zum Schluss den Schlüssel 4. Nutzen Sie dazu die Vorlage (oberes Drittel der Abbildung) und zeichnen Sie – wie für die ersten drei Operationen gezeigt – die Bäume nach den jeweiligen Suchbaum–Einfüge– bzw. Löschoperationen und gegebenfalls ein zweites Bild nach den notwendigen Rotationen. Zur Vereinfachung der Zeichnungen können Sie alle Blätter ignorieren.

Lösung:

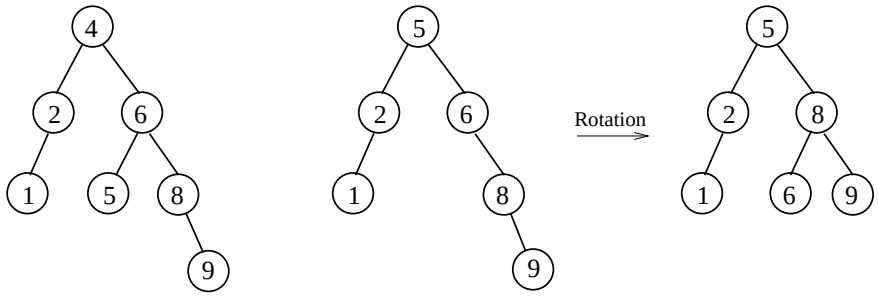
2 einfügen: 4 einfügen: 8 einfügen:



5 einfügen: 6 einfügen: 1 einfügen:

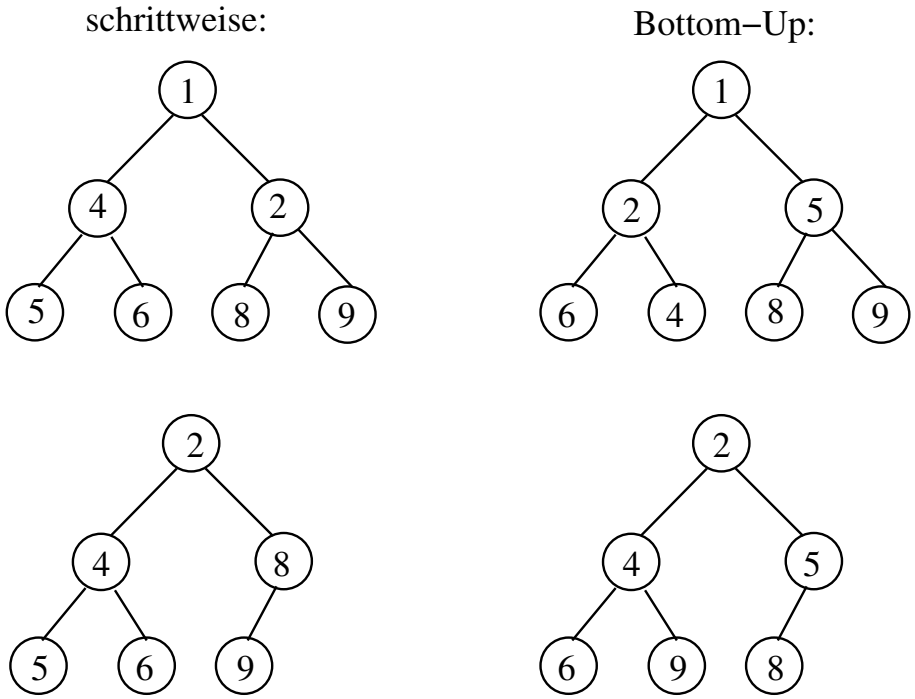


9 einfügen: 4 löschen:



b) Bauen Sie jeweils mit der Schlüsselfolge $2, 4, 8, 5, 6, 1, 9$ zwei Halden auf, die erste durch schrittweises Einfügen und die zweite mit der Bottom–Up–Konstruktion. In beiden Fällen reicht das Endergebnis ohne Zwischenschritte. Entfernen Sie danach in beiden Fällen das minimale Element.

Lösung:



c) Angenommen, Sie haben einen AVL–Baum T_1 der Höhe h, der Schüssel aus dem Bereich $[105, 200]$ hält und einen AVL–Baum T_2 der Höhe $h + 1$, der Schüssel aus dem Bereich $[1, 95]$ hält (wieviele das sind spielt keine Rolle). Beschreiben Sie eine schnelle Methode zur Konstruktion eines AVL–Baums, dessen Schüsselmenge die Vereinigung der Schüsselmengen von T_1 und T_2 ist und analysieren Sie die Laufzeit.

Lösung: Wir definieren einen neuen Knoten r mit den Schlüssel 100 als Wurzel eines neuen Binärbaums T und legen T_2 als linken Teilbaum und T_1 als rechten Teilbaum von r fest. Offensichtlich ist T ein binärer Suchbaum mit Höhen–Balance–Eigenschaft, also ein AVL–Baum. Nun muss man nur noch den Schlüssel 100 löschen, das geht (einschließlich eventueller Rotationen) in $O(h)$ Zeit.

Aufgabe 4: **Sortieren** $2 + 6$ **Punkte**

Sei K_m der vollständige Graph auf der Knotenmege $V = \{1, 2, \ldots, m\}$. Einer Kante $e = \{u, v\}$ wird der folgende Schlüssel $k(e)$ zugeordnet:

$$k(e) = (\max(u, v) - 1) \cdot m + \min(u, v)$$

a) Gegeben sei eine Folge von n Kanten des K_m, die bezüglich ihrer Schlüssel sortiert werden soll. Skizzieren Sie, wie man die Verfahren Counting–Sort und Radix–Sort in diesem Fall anwenden kann. Es geht nicht darum, die Verfahren im Detail zu erklären, sondern zu erläutern, mit welchen Parametern (welches Feld bzw. wieviele Stufen und was passiert dort) man im konkreten Fall arbeiten sollte, um gute Laufzeiten zu erzielen.

b) Analysieren Sie die Laufzeit beider Verfahren **in Abhängigkeit von der Eingabegröße** n unter den folgenden Voraussetzungen:

$$b1) \quad n = \Theta(\sqrt{m}) \qquad b2) \quad n = \Theta(m \cdot \sqrt{m}) \qquad b3) \quad n = \Theta(m^2)$$

Vergleichen Sie die analysierten Laufzeiten in den drei Fällen mit der Laufzeit von Heap–Sort. Welches oder welche Verfahren sind in dem jeweiligen Fall am besten?

Lösung a: Nach Bildungsvorschrift sind alle Schlüssel positive, ganze Zahlen, die kleiner als m^2 sind. Deshalb reicht für Counting–Sort ein Array der Länge m^2 mit dem man n–elementige Folgen in der Zeit $O(n + m^2)$ sortieren kann.
Um Radix–Sort effizient einzusetzen, sollte man m Warteschlagen verwenden, da man so in zwei Stufen fertig ist. Die Laufzeit zum Sortieren einer n–elementigen Folge ist dann $O(2n + m) = O(n + m)$.

Zusatzkommentar: Für Radix–Sort interpretiert man die Schlüssel als zweistellige Zahlen im Zahlensystem mit m Ziffern. Da jede Zahl dieser Art eine eindeutige Darstellung der Form $a \cdot m + b$ mit $a, b \in \{0, 1, \ldots, m - 1\}$ hat, ist klar, dass der Schlüssel $k(e)$ einer Kante $e = \{u, v\}$ durch die Ziffern $a = \max(u, v) - 1$ und $b = \min(u, v)$ dargestellt wird. Folglich wird e in der ersten Runde in die $\min(u, v)$–te Warteschlange und in der zweiten Runde in die $(\max(u, v) - 1)$–te Warteschlange eingefügt.

Lösung b: Da für Heap–Sort die Laufzeit $O(n \log n)$ bekannt ist, muss man nur noch die oben genannten Laufzeiten von Counting–Sort und Radix–Sort auf die Größe von n umrechnen und dann das jeweils beste Verfahren heraussuchen.

$b1)$ $n = \Theta(\sqrt{m}) \iff m = \Theta(n^2)$ $\quad b2)$ $n = \Theta(m \cdot \sqrt{m}) = \Theta(m^{\frac{3}{2}}) \iff m = \Theta(n^{\frac{2}{3}})$
$b3)$ $n = \Theta(m^2) \iff m = \Theta(\sqrt{n})$

Damit ergeben sich die folgenden Laufzeiten

Fall	Heap–Sort	Counting–Sort	Radix–Sort
b1)	$O(n \log n)$	$O(n + m^2) = O(n + (n^2)^2) = O(n^4)$	$O(n + m) = O(n + n^2) = O(n^2)$
b2)	$O(n \log n)$	$O(n + m^2) = O(n + (n^{\frac{2}{3}})^2) = O(n^{\frac{4}{3}})$	$O(n + m) = O(n + n^{\frac{2}{3}}) = O(n)$
b3)	$O(n \log n)$	$O(n + m^2) = O(n + \sqrt{n}^2) = O(n)$	$O(n + m) = O(n + \sqrt{n}) = O(n)$

Im ersten Fall ist also Heap–Sort am schnellsten, im zweiten Radix–Sort und im dritten Radix–Sort und Counting–Sort.