Prof. Dr. Ir. Joost-Pieter Katoen

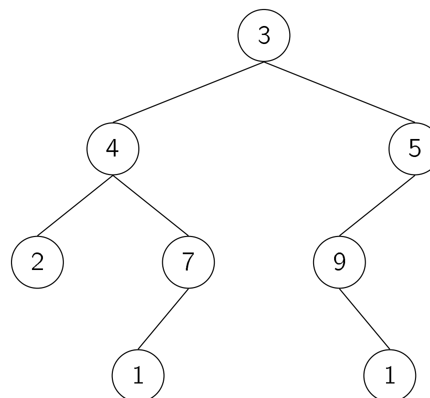
Christian Dehnert, Friedrich Gretz, Benjamin Kaminski, Thomas Ströder

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2-3 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Mittwoch, den 06.05.2015 um 12:15 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor abgeben.
- In Aufgaben, bei denen Sie Algorithmen implementieren sollen, dürfen Sie Ihre Lösung als Pseudo-Code abgeben. Abgaben in verbreiteten imperativen Sprachen wie Java oder C++ sind ebenfalls erlaubt.

Tutoraufgabe 1 (Linearisierungen von binären Bäumen):

a) Geben Sie jeweils das Ergebnis der In-, Pre- und Postorder-Traversierung des folgenden Baumes an:



b) Bestimmen Sie zu den folgenden Paaren von Linearisierungen den jeweils zugehörigen Baum:

(i) in-order: 5 8 4 7 3 1 6 9 2
pre-order: 3 4 5 8 7 2 6 1 9

(ii) in-order: 5 1 2 4 6 7 3 9 8
post-order: 5 2 4 1 3 8 9 7 6

c) Geben Sie zwei Funktionen f und g an, die zu jeder natürlichen Zahl i einen Baum mit i Knoten liefern und für die gilt, dass

- die Schlüssel in $f(i)$ paarweise unterschiedlich sind und
- dass für alle $i \in \mathbb{N}$ die Preorder-Linearisierung von $f(i)$ genau der Postorder-Linearisierung von $g(i)$ entspricht.

Tutoraufgabe 2 (Programmanalyse):

Bestimmen Sie die Komplexitätsklasse der Laufzeit von `calculate` in Abhängigkeit der Eingabelänge n des Parameters value. Hierbei ist die Eingabelänge einer Zahl definiert als die Zahl selbst. Gehen Sie davon aus, dass sowohl die Grundrechenarten +, -, *, / als auch Vergleiche (">") und Zuweisungen ("=") in konstanter Zeit $\Theta(1)$ ausgeführt werden.

```
int calculate(int value) {
    int result = value;
    for (int i = value; i > 0; i = i/2) {
        result = result * result;
        for (int j = i; j > 0; j--) {
            result = 2 * result;
        }
    }
    return result;
}
```

Tutoraufgabe 3 (Beweise):

Zeigen oder widerlegen Sie die folgenden Aussagen:

a) $o(f(n)) \cap \omega(f(n)) = \emptyset$

b) Aus $f(n) \in \Omega(g(n))$ und $g(n) \in \Omega(h(n))$ folgt $f \in \Omega(h(n))$.

Tutoraufgabe 4 (Lineare Suche):

Geben Sie einen Algorithmus an, der für eine Folge von n ganzen Zahlen (gegeben als Array) eine maximale Teilfolge findet und dessen Worst-Case-Laufzeitkomplexität in $\mathcal{O}(n)$ liegt. Eine Teilfolge wird hierbei von beliebig vielen (maximal n) *direkt aufeinanderfolgenden* Array-Einträgen gebildet. Sie ist maximal, wenn die Summe ihrer Elemente maximal ist, d. h., wir suchen aus allen möglichen Teilfolgen eine mit maximaler Summe.
Die Teilfolge soll dabei als *Startindex*, *Endindex* und *Summe* der Folge ausgegeben werden. Die Eingangsfolge

$$12,\ -34,\ 56,\ -5,\ -6,\ 78,\ -32,\ 8$$

liefert beispielsweise die Indizes 3 und 6 sowie die Summe $56 - 5 - 6 + 78 = 123$.

Aufgabe 5 (Programmanalyse): (3 + 3 = 6 Punkte)

Gegeben sei der folgende Algorithmus.

```
int[] f(int[] E) {
    for (int i = 0; i < E.length; ++i) {
        int cnt = i;
        if (E[i] != 0) {
            while (cnt > 0) {
                for (int j = cnt; j > 0; --j) {
                    E[i] = E[i] + 1;
                }
                --cnt;
            }
        }
    }
    return E;
}
```

Die Laufzeit des Algorithmus soll anhand der **Schreibzugriffe** auf das Array `E` gemessen werden (die elementaren Operationen sind hier also lediglich die Schreibzugriffe auf `E`; alles andere ist zu vernachlässigen). Dabei soll die **konkrete Laufzeit** des Algorithmus bestimmt werden und nicht die asymptotische Komplexität.

a) Welche Eingaben der Länge n führen zur Best- bzw. Worst-Case-Laufzeit des Algorithmus und welche Laufzeiten $B(n)$ und $W(n)$ ergeben sich? Begründen Sie Ihre Antwort.

b) Was ist die Average-Case-Laufzeit des Algorithmus? Dabei soll angenommen werden, dass die Wahrscheinlichkeit, dass das Element an Position i des Arrays `E` den Wert null hat, gegeben ist durch

$$\Pr(\text{“}E[i] == 0\text{''}) = \begin{cases} \frac{1}{i} & \text{wenn } i \neq 0 \\ 0 & \text{sonst} \end{cases}$$

Aufgabe 6 (Beweise): (2 + 2 = 4 Punkte)

Zeigen oder widerlegen Sie die folgenden Aussagen:

a) $2^{2n} \in \omega(2^n)$

b) $2^{\log_a n} \in \Theta(2^{\log_b n})$ für alle $a, b > 1$

Aufgabe 7 (Lineare Suche): (5 Punkte)

Ein *Meisterintrigant* ist jemand, der zwischen allen Paaren von Personen ein Geheimnis kennt, das die eine Person vor der anderen verbergen will, von dem aber niemand außer ihm selbst ein Geheimnis kennt, das er vor irgendeiner anderen Person verbergen will. Formal ausgedrückt: Sei P eine Menge von Personen und $K \subseteq P \times P \times P$ eine Relation, wobei $K(x, y, z)$ bedeutet, dass die Person x ein Geheimnis kennt, das die Person y vor der Person z verbergen will. Dann ist $m \in P$ ein Meisterintrigant gdw. $\forall x, y \in P : K(m, x, y) \wedge (x \neq m \implies \neg K(x, m, y))$.

Das Problem der Suche nach einem Meisterintriganten ist formal folgendermaßen definiert:

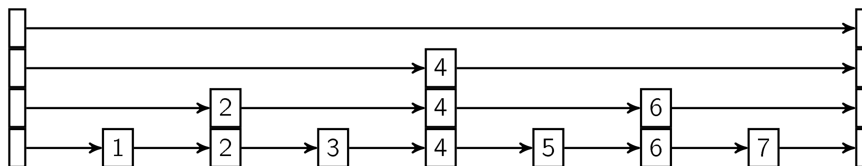
Eingabe:
- Eine Anzahl $n \in \mathbb{N}$ von Personen nummeriert von 0 bis $n - 1$.
- Genau eine Person davon ist ein Meisterintrigant.
- Eine $n \times n \times n$ Matrix `K` mit Wahrheitswerten, wobei `K[x][y][z]` genau dann wahr ist, wenn die Person `x` ein Geheimnis der Person `y` kennt, das die Person `y` vor der Person `z` verbergen will.

Ausgabe: Die Nummer $x \in \{0, \ldots, n-1\}$ des Meisterintriganten.

Geben Sie einen Algorithmus an, der das Problem der Suche nach einem Meisterintriganten löst und eine Worst-Case-Laufzeit $W(n) \in \mathcal{O}(n)$ hat, wobei n die Anzahl der Personen aus der Problemdefinition ist.

Aufgabe 8 (Datenstruktur): (10 Punkte)

In dieser Aufgabe sollen Sie den ADT Liste aus der Vorlesung teilweise implementieren. Wir beschränken uns dabei auf die Operationen `insert` und `search`. Ihre Implementierung soll im Average-Case eine Laufzeit in $\mathcal{O}(\log(n))$ für beide Operationen haben. Dafür bietet sich eine Implementierung als sogenannte Skip List an.
Eine Skip List basiert auf einer einfach verketteten sortierten Liste. Ohne Modifikationen hätten die beiden geforderten Operationen dabei eine Average-Case-Laufzeit $A(n) \in \Theta(n)$. Um diese Laufzeit zu verbessern, hat ein Knoten in einer Skip List außer seinem Element auch eine Höhe. Diese Höhe wird beim Einfügen eines Knotens zufällig nach folgendem Vorgehen bestimmt: Der Knoten wird zunächst mit Höhe 1 eingefügt. Mit 50% Wahrscheinlichkeit behält er seine Höhe und das Einfügen ist beendet. Im anderen Fall wird seine Höhe um 1 erhöht und die gleiche Fallunterscheidung wird wiederholt. Somit hat ein Knoten also mit 50% Wahrscheinlichkeit die Höhe 1, mit 25% Wahrscheinlichkeit die Höhe 2, mit 12,5% Wahrscheinlichkeit die Höhe 3 usw. Eine Skip List hat außerdem jeweils einen speziellen Start- und Endknoten. Diese Knoten haben keine Elemente und als Höhe das Maximum der Höhen aller Knoten in der Skip List plus 1. Der Nachfolger eines Knotens in einer Skip List *auf Höhe h* ist der nächste Knoten in der Skip List, welcher mindestens Höhe *h* hat. Somit haben die Knoten in einer Skip List potentiell so viele verschiedene Nachfolger wie Ihre Höhe. Um nun ein Element schneller zu finden, sucht man zunächst nur auf der höchsten Ebene nach diesem Element. Überläuft man es, geht man einen Schritt zurück und eine Ebene abwärts und sucht weiter, bis man das Element gefunden hat oder an der untersten Ebene 1 angekommen ist und es nicht gefunden hat. Im Average-Case reduziert sich durch dieses Vorgehen die Laufzeit der Operationen `insert` und `search` auf $\Theta(\log(n))$. Die folgende Grafik soll das Konzept einer Skip List veranschaulichen:



Implementieren Sie also die Operationen `insert` und `search` für den ADT Liste als Skip List, sodass beide Operationen eine Average-Case-Laufzeit $A(n) \in \mathcal{O}(\log(n))$ haben. Sie brauchen nicht zu beweisen, dass Ihre Implementierung die geforderte Laufzeitschranke einhält. Um eine Fallunterscheidung durchzuführen, die in 50% der Fälle zu `true` auswertet, können Sie die Operation `boolean flipCoin()` verwenden. Diese liefert mit 50% Wahrscheinlichkeit `true` und sonst `false` zurück. Außerdem können Sie die Operation `new List()` verwenden, um einen neuen Listenknoten zu erzeugen, dessen Felder (Attribute) uninitialisiert sind. Um unendlich kleine oder große Schlüsselwerte auszudrücken, können Sie die Werte $-\infty$ und ∞ verwenden.