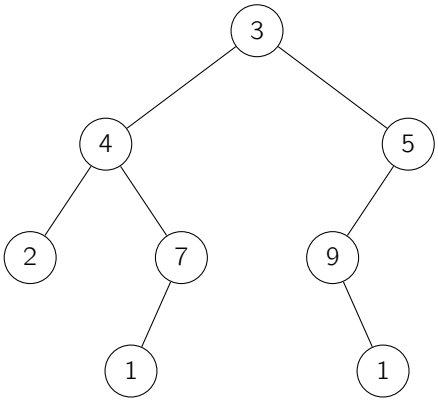
Prof. Dr. Ir. Joost-Pieter Katoen

Christian Dehnert, Friedrich Gretz, Benjamin Kaminski, Thomas Ströder

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.

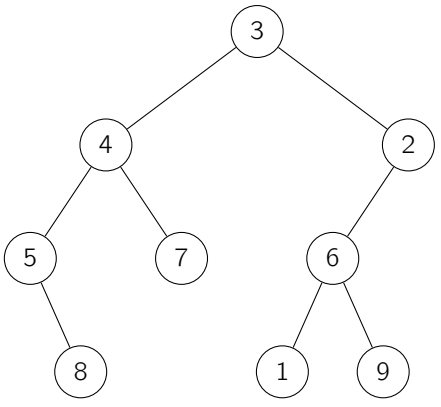
Lösung:

a) Die Linearisierungen sind wie folgt:

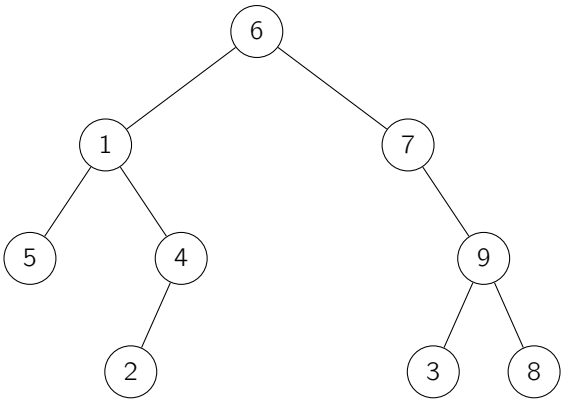
preorder: 3 4 2 7 1 5 9 1
inorder: 2 4 1 7 3 9 1 5
postorder: 2 1 7 4 1 9 5 3

b) Es ergeben sich die folgenden Bäume:

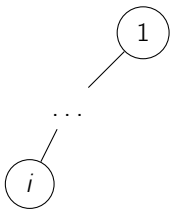
(i)



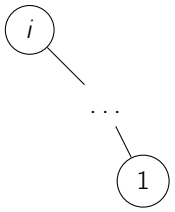
(ii)



c) Wir definieren $f(i)$ als



und $g(i)$ als



Dann ist die Preorder-Linearisierung von $f(i)$ identisch mit der Postorder-Linearisierung von $g(i)$.

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;
}
```

Lösung:

Das vorgegeben Programm besitzt eine verschachtelte Schleife. Die Zählervariable i der äußeren startet bei n und wird in jedem Durchlauf halbiert. Sie besitzt somit im i-ten Durchlauf den Wert $\frac{n}{2^i}$. D. h., die äußere Schleife wird exakt $\lfloor log_2(n) \rfloor + 1$-mal durchlaufen, bevor sie den Wert $\frac{n}{2^{log_2(n+1)}} = \frac{n}{n+1} < 1$ annimmt und damit die Schleifenbedingung verletzt wird.
Die innere Schleife wird jeweils von der Zählervariable der äußeren Schleife bis zu 0 runtergezählt, wird also in der k-ten Iteration $\frac{n}{2^k}$-mal durchlaufen. Alle weiteren Anweisungen haben konstante Laufzeit. Somit ergibt sich eine Laufzeit von:

$$
\begin{aligned}
T(n) &= \underbrace{\frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} \cdots + \frac{n}{2^{\lfloor log_2 n \rfloor}}}_{\text{innere Schleife}} + \underbrace{\lfloor \log_2 n \rfloor + 1}_{\text{äußere Schleife}} \\
&= \lfloor \log_2 n \rfloor + 1 + \sum_{i=0}^{\lfloor log_2 n \rfloor} \frac{n}{2^i} \\
&= \lfloor \log_2 n \rfloor + 1 + n \cdot \underbrace{\sum_{i=0}^{\lfloor log_2 n \rfloor} \frac{1}{2^i}}_{1 \leq c \leq 2} \\
&= \lfloor \log_2 n \rfloor + 1 + c \cdot n \in \Theta(n)
\end{aligned}
$$

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 $f \in \Omega(h(n))$.

Lösung:

a) **Behauptung:** Die Aussage gilt.

Beweis:
Wir führen den Beweis durch Widerspruch. Angenommen der Schnitt sei nicht leer. Dann gibt es eine Funktion $g(n)$, für die gilt:

$$g(n) \in o(f(n)) \ \wedge \ g(n) \in \omega(f(n))$$

Aus $g(n) \in o(f(n))$ folgt:

$$\forall c \in \mathbb{R}_{>0}, \exists n_1 \in \mathbb{N} \text{ mit } \forall n \geq n_1 : 0 \leq g(n) < c \cdot f(n) \tag{1}$$

Aus $g(n) \in \omega(f(n))$ folgt:

$$\forall c \in \mathbb{R}_{>0}, \exists n_2 \in \mathbb{N} \text{ mit } \forall n \geq n_2 : c \cdot f(n) < g(n) \tag{2}$$

Aus (1) und (2) folgt, dass für $n_0 = max(n_1, n_2)$ gilt:

$$\forall c \in \mathbb{R}_{>0}, \forall n \geq n_0 : 0 \leq g(n) < c \cdot f(n) < g(n)$$

Dies ist ein Widerspruch. Daher muss die Annahme $o(f(n)) \cap \omega(f(n)) \neq \emptyset$ falsch sein. □

b) **Behauptung:** Die Aussage gilt.

Beweis:
Aus $f(n) \in \Omega(g(n))$ folgt:

$$
\begin{aligned}
\exists c_1 \in \mathbb{R}_{>0}, n_1 \in \mathbb{N} \ mit \ \forall n \geq n_1 : \quad & 0 \quad \leq c_1 \cdot g(n) \leq f(n) \\
\exists c_1 \in \mathbb{R}_{>0}, n_2 \in \mathbb{N} \ mit \ \forall n \geq n_2 : \quad & 0 \quad \leq g(n) \leq \frac{1}{c_1} \cdot f(n)
\end{aligned}
$$

Aus $g(n) \in \Omega(h(n))$ folgt: $\exists c_2 \in \mathbb{R}_{>0}, n_2 \in \mathbb{N} : \ 0 \leq c_2 \cdot h(n) \leq g(n) \ \forall n \geq n_2$
Dann gilt:

$$\begin{aligned}
& 0 \leq c_2 \cdot h(n) \leq g(n) && \text{für alle } n \geq n_2 \\
\Rightarrow \quad & 0 \leq c_2 \cdot h(n) \leq g(n) \leq \frac{1}{c_1} \cdot f(n) && \text{für alle } n \geq \underbrace{max(n_1, n_2)}_{n_3} \\
\Rightarrow \quad & 0 \leq \underbrace{c_1 \cdot c_2}_{c_3} \cdot h(n) \leq f(n) && \text{für alle } n \geq n_3 \\
\Rightarrow \quad & \exists c_3 \in \mathbb{R}_{>0}, n_3 \in \mathbb{N} \ mit \ f(n) \geq c_3 \cdot h(n) && \text{für alle } n \geq n_3 \\
\Rightarrow \quad & f(n) \in \Omega(h(n))
\end{aligned}$$

□

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 $56 - 5 - 6 + 78 = 123$.

Lösung:

Der folgende Algorithmus bestimmt zu einer Folge die Teilfolge mit maximaler Summe in Laufzeit $\Theta(n)$.

```
public static Triple maxSum(int[] array){
    // die folgenden drei Variablen benoetigen wir um uns die
    // Eckdaten der maximalen Folge zu merken.
    int max = 0;
    int maxStart = 0;
    int maxEnd = -1;
    // start ist die Startposition der aktuell betrachteten Folge
    int start = 0;
    // sum ist Summe der aktuell betrachteten Folge
    int sum = 0;
    for (int i = 0; i < array.length; i++) {
        // erweitere die aktuell betrachtete Folge um die Position i
        sum += array[i];
        // ist die Summe der aktuellen Folge negativ, so hat jede
        // maximale Folge,  die dahinter startet eine hoehere
        // Summe als die aktuelle auf diese Position erweitert
        if (sum <= 0) {
            sum = 0;
            start = i+1;
        }
        // stelle fest, ob neue maximale Folge gefunden wurde
        if (sum > max){
```

```
                max = sum;
                maxStart = start;
                maxEnd = i;
            }
        }
        return (maxStart, maxEnd, max);
    }
```