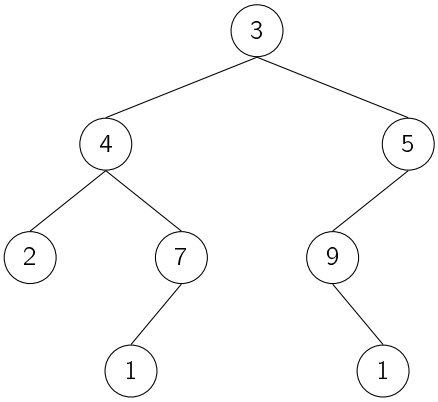
Prof. Dr. Ir. Joost-Pieter Katoen

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.

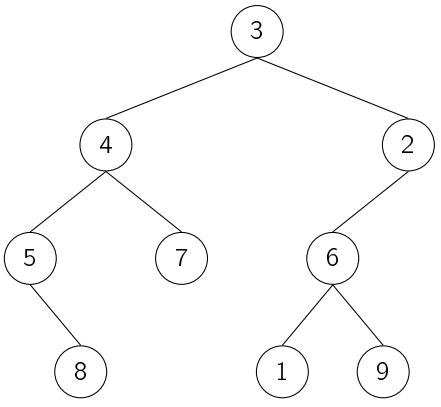
Lösung:

a) Die Linearisierungen sind wie folgt:

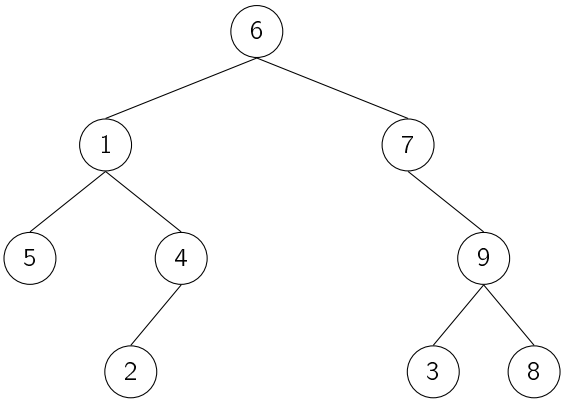
preorder: 3 4 2 7 1 5 9 1
inorder: 2 4 1 7 3 9 1 5
postorder: 2 1 7 4 1 9 5 3

b) Es ergeben sich die folgenden Bäume:

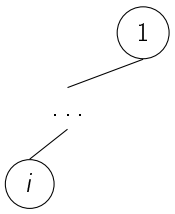
(i)



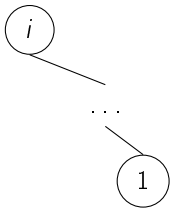
(ii)



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);
    }
```

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

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

Lösung:

a) Die Best-Case-Laufzeit des Algorithmus wird erreicht, wenn alle Elemente von `E` null sind. Dann ergibt sich als Laufzeit

$$\sum_{i=0}^{n-1} 0 = 0 \quad (\in \Theta(0))$$

Achtung: $0 \notin \Theta(1)$!

Dementsprechend wird die Worst-Case-Laufzeit erreicht, wenn alle Elemente von `E` ungleich null sind. Dann ergibt sich als Laufzeit

$$\begin{aligned}
\sum_{i=0}^{n-1}\sum_{j=0}^{i} j &= \sum_{i=0}^{n-1} \frac{i(i+1)}{2} \\
&= \sum_{i=0}^{n-1} \frac{i^2+i}{2} \\
&= \frac{1}{2}(\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i) \\
&= \frac{1}{2}(\frac{1}{6}n(n-1)(2(n-1)+1) + \frac{n(n-1)}{2}) \\
&= \frac{1}{2}(\frac{1}{6}n(n-1)(2n-1) + \frac{1}{2}n(n-1)) \\
&= \frac{1}{4}n(n-1)(\frac{1}{3}(2n-1)+1) \\
&= (\frac{1}{4}n^2 - \frac{1}{4}n)(\frac{2}{3}n - \frac{1}{3} + 1) \\
&= (\frac{1}{4}n^2 - \frac{1}{4}n)(\frac{2}{3}n + \frac{2}{3}) \\
&= \frac{1}{6}n^3 + \frac{1}{6}n^2 - \frac{1}{6}n^2 - \frac{1}{6}n = \frac{1}{6}n^3 - \frac{1}{6}n \quad (\in \Theta(n^3))
\end{aligned}$$

b) Im Average-Case ergibt sich folgende Laufzeit:

$$\begin{aligned}
&\sum_{i=0}^{n-1} \Pr(\text{“}E[i] \neq 0\text{”}) \cdot \sum_{j=0}^{i} j \\
&= \underbrace{1 \cdot \sum_{j=0}^{0} j}_{\text{Kosten äußere Schleife } i=0} + \underbrace{\sum_{i=1}^{n-1} \left(1 - \frac{1}{i}\right) \sum_{j=0}^{i} j}_{\text{Kosten äußere Schleife } i=1,\ldots,n-1} \\
&= \sum_{i=1}^{n-1} \left(1 - \frac{1}{i}\right) \frac{i(i+1)}{2} \\
&= \sum_{i=1}^{n-1} \frac{i(i+1)}{2} - \frac{i+1}{2} \\
&= \sum_{i=1}^{n-1} \frac{(i+1)(i-1)}{2} \\
&= \sum_{i=1}^{n-1} \frac{i^2-1}{2} \\
&= \frac{n(n-1)(2(n-1)+1)}{12} - \frac{n-1}{2} \\
&= \frac{(n^2-n)(2n-1)}{12} - \frac{6n-6}{12} \\
&= \frac{2n^3-3n^2-5n+6}{12} \quad (\in \Theta(n^3))
\end{aligned}$$

Nachtrag: Bei der Definition in der Aufgabenstellung “steigt” die Wahrscheinlichkeit, dass ein Element nicht den Wert null hat über die Elemente im Array an. Definiert man die Wahrscheinlichkeit, dass ein Element

ungleich null ist über die Vorschrift

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

so “sinkt” die Wahrscheinlichkeit, dass ein Element nicht den Wert null hat über die Elemente im Array. Dann ergibt sich folgende Average-Case-Laufzeit:

$$\begin{aligned} \sum_{i=1}^{n-1} \frac{1}{i} \sum_{j=0}^{i} j &= \sum_{i=1}^{n-1} \frac{1}{i} \frac{i(i+1)}{2} \\ &= \sum_{i=1}^{n-1} \frac{i+1}{2} \\ &= \frac{1}{2} \sum_{i=1}^{n-1} i + 1 \\ &= \frac{1}{2} \cdot \left(\frac{n(n+1)}{2} - 1\right) \\ &= \frac{n(n+1)}{4} - \frac{1}{2} \\ &= \frac{1}{4}n^2 + \frac{1}{4}n - \frac{1}{2} \quad (\in \Theta(n^2)) \end{aligned}$$

Achtung: Durch die Definition von $\Pr(\text{“}E[0] \neq 0\text{”}) = 0$ beginnt die Summe erst beim Index $i = 1$.

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 $a, b > 1$

Lösung:

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

Beweis:
Seien $f(n) = 2^n$, $g(n) = 2^{2n}$. Sei $c > 0$ beliebig. Dann gilt:

$$\begin{aligned} & c \cdot f(n) < g(n) \text{ für alle } n \geq n_0 \\ \Leftrightarrow\ & c \cdot 2^n < 2^{2n} \text{ für alle } n \geq n_0 \\ \Leftrightarrow\ & c \cdot 2^n < (2^n)^2 \text{ für alle } n \geq n_0 \\ \Leftrightarrow\ & c < 2^n \text{ für alle } n \geq n_0 \end{aligned}$$

Wähle $n_0 = \lceil c \rceil$ (da $2^n > n$ für alle $n \geq 0$). □

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

Beweis:
Seien $a = 4$ und $b = 2$. Wegen

$$\log_4 n = \frac{\log_2 n}{\log_2 4} = \frac{1}{2} \cdot \log_2 n$$

ist

$$2^{\log_4 n} = 2^{\frac{1}{2} \cdot \log_2 n} = (2^{\log_2 n})^{\frac{1}{2}} = \sqrt{n} \notin \Theta(n) = \Theta(2^{\log_2 n})$$

□

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.

Lösung:

Die beiden folgenden Eigenschaften gelten bzgl. eines Meisterintriganten:

1. Falls `K[x][y][z]` gilt, ist `y` kein Meisterintrigant.
2. Falls `K[x][y][z]` nicht gilt, ist `x` kein Meisterintrigant.

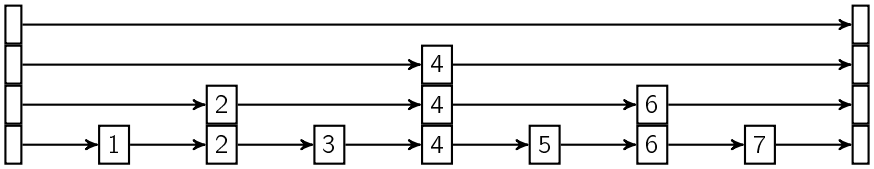
Aus diesen beiden Eigenschaften und der gegebenen Tatsache, dass genau eine der n Personen ein Meisterintrigant ist, lässt sich eine bilineare Suche konstruieren, wobei der Wert der Koordinate `z` beliebig ist:

```
int MasterIntrigantSearch(boolean[][][] K, int n) {
    int x = 0;
    int y = n - 1;
    while (x != y) {
        if (K[x][y][0]) {
            y = y - 1;
        } else {
            x = x + 1;
        }
    }
    return x;
}
```

Dieser Algorithmus hat offensichtlich eine Worst-Case-Laufzeit $W(n) \in \mathcal{O}(n)$, da die Schleife genau $(n - 1)$-mal durchlaufen wird und eine konstante Anzahl von Operationen durchführt.

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.

Lösung:

```
List create(
    int k,
    Element x,
    List up,
    List down,
    List right,
    boolean s,
    boolean e
) {
    List res = new List();
    res.key = k;
    res.element = x;
    res.above = up;
    res.below = down;
    res.next = right;
```

```
    res.start = s;
    res.end = e;
    return res;
}

// l points to start node on level 1
void insert(List l, Element x, int k) {
    List cur = l;
    while (cur.above != null) {
        cur = cur.above;
    }
    List top = skipInsert(cur, x, k);
    if (top != null) {
        top.next = cur.next;
        cur.next = top;
        List higherEnd =
            create(∞, null, null, cur.next, null, false, true);
        List higherStart =
            create(−∞, null, null, cur, higherEnd, true, false);
        List startList = higherStart;
        List endList = higherEnd;
        List middle = top;
        while (flipCoin()) {
            List higherMiddle =
                create(
                    middle.key,
                    middle.element,
                    null,
                    middle,
                    endList,
                    false,
                    false
                );
            startList.next = higherMiddle;
            higherEnd =
                create(∞, null, null, endList, null, false, true);
            higherStart =
                create(
                    −∞,
                    null,
                    null,
                    startList,
                    higherEnd,
                    true,
                    false
                );
            startList = higherStart;
            endList = higherEnd;
            middle = higherMiddle;
        }
    }
}

List skipInsert(List l, Element x, int k) {
```

```
        List next = l.next;
        if (next.end || next.key > k) {
            if (l.below == null) {
                List toAdd = create(k, x, null, null, next, false, false);
                l.next = toAdd;
                if (flipCoin()) {
                    List higher =
                        create(k, x, null, toAdd, null, false, false);
                    toAdd.above = higher;
                    return higher;
                } else {
                    return null;
                }
            } else {
                List added = skipInsert(l.below, x, k);
                if (added != null) {
                    added.next = next;
                    l.next = added;
                    if (flipCoin()) {
                        List higher =
                            create(
                                added.key,
                                added.element,
                                null,
                                added,
                                null,
                                false,
                                false
                            );
                        added.above = higher;
                        return higher;
                    } else {
                        return null;
                    }
                }
            }
        } else {
            return skipInsert(next, x, k);
        }
    }

    // l points to start node on level 1
    Element search(List l, int k) {
        List cur = l;
        while (cur.above != null) {
            cur = cur.above;
        }
        return skipSearch(cur.next, k);
    }

    Element skipSearch(List l, int k) {
        if (l.key == k) {
            return l.element;
        }
```

```
        List next = l.next;
        if (next.end || next.key > k) {
            if (l.height > 1) {
                return skipSearch(l.below, k);
            } else {
                return null;
            }
        } else {
            return skipSearch(next, k);
        }
    }
```