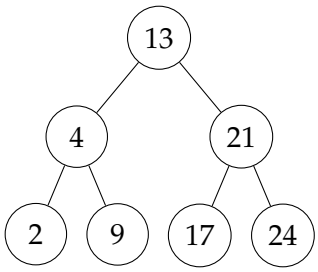
Datenstrukturen und Algorithmen
Übung 7 – Suchbäume und Repetition

Musterlösung

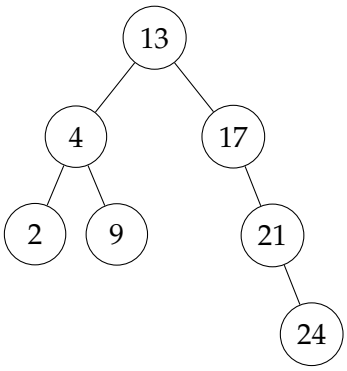
Theoretische Aufgaben: Binäre Suchbäume

1. Gegeben seien die Schlüssel $\{2,4,9,13,17,21,24\}$. Zeichnen Sie binäre Suchbäume der Höhe 2,3,4,5 und 6, die aus genau diesen Schlüsseln bestehen. **(1 Punkt)**

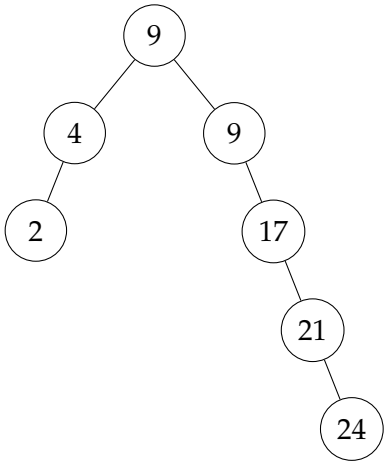
Höhe 2:



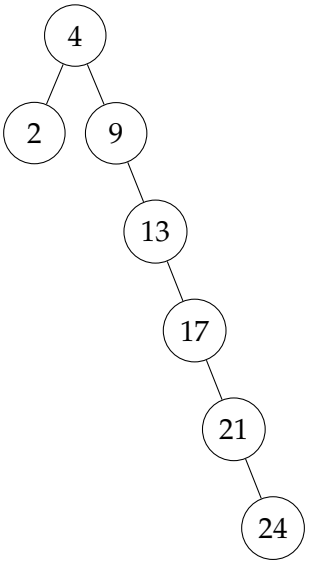
Höhe 3:



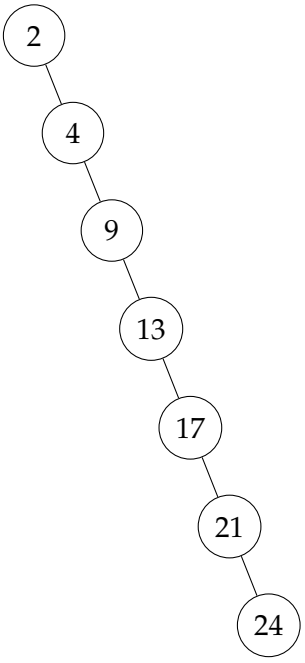
Höhe 4:



Höhe 5:



Höhe 6:



Hinweis: Bis auf $h = 2$ sind viele verschiedene Bäume mit diesen Eigenschaften möglich

2. Was ist der Unterschied zwischen der binären Suchbaum-Eigenschaft und der *Min-Heap*-Eigenschaft?

   Die *Min-Heap*-Eigenschaft ist gleichbedeutend damit, dass jeder Unterbaum eines Knotens y keinen größeren Schlüssel enthält als $y.key$ selbst.

   In einem binären Suchbaum gilt dies für den linken Unterbaum, für den rechten Unterbaum hingegen soll das Gegenteil gelten, alle Knoten sollen dort $\geq y.key$ sein.

   Kann die *Min-Heap*-Eigenschaft benutzt werden, um Schlüssel in einem Baum mit n Knoten in sortierter Reihenfolge in $\mathcal{O}(n)$ Zeit auszugeben?

   Wir können einen Heap in Linearzeit aufbauen. Wenn wir nun eine Möglichkeit hätten, ihn sortiert in Linearzeit auszugeben, so könnte man die Algorithmen zu einem in Linearzeit laufenden vergleichenden Sortieralgorithmus kombinieren. Da ein solcher nicht existieren kann (vgl. Vorlesung), kann es auch keinen Algorithmus geben, der jeden Heap in Linearzeit in sortierter Reihenfolge ausgibt.

   Gibt es einen vergleichenden Algorithmus, der für beliebige Schlüsselfolgen einen binären Suchbaum in $\mathcal{O}(n)$ aufbaut? Erklären Sie Ihre Antwort.

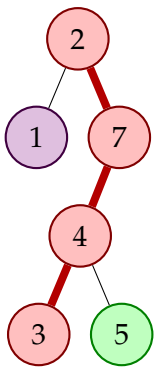
   Leider nein, hier können wir ähnlich argumentieren:

   Wir können einen binären Suchbaum in Linearzeit ausgeben. Angenommen, es gäbe einen Algorithmus, der ihn auch in Linearzeit aufbaut, so müsste die Kombination der beiden einen vergleichenden Sortieralgorithmus ergeben, der in $\mathcal{O}(n)$ arbeitet. Da ein solcher nicht existieren kann, kann auch der gefragte Algorithmus nicht existieren.

   (1 Punkt)

3. Angenommen, die Suche nach einem Schlüssel k in einem binären Suchbaum endet in einem Blatt. Wir unterscheiden drei Mengen: A, die Schlüssel links vom Suchpfad, B die Schlüssel auf dem Suchpfad, und C, die Schlüssel rechts vom Suchpfad. Die Vermutung ist, dass für jeweils drei Schlüssel $a \in A, b \in B$ und $c \in C$ gilt, dass $a \leq b \leq c$. Widerlegen Sie diese Vermutung mit einem Gegenbeispiel, das einen möglichst kleinen Baum verwendet. **(1 Punkt)**

   Folgender Baum erfüllt die Binäre-Suchbaum-Eigenschaft.
   Betrachte den Pfad von der Wurzel 2 zum Blatt 3:

   $A = \{1\}$
   $B = \{2,7,4,3\}$
   $C = \{5\}$

   Aber $B \ni 7 \not\leq 5 \in C$!
   Also kann die Behauptung nicht stimmen.

4. Schreiben Sie Pseudocode für eine rekursive Version des Einfügens eines Knotens in einen nicht leeren binären Suchbaum. Beschreiben Sie ihren Algorithmus in 1-2 Sätzen. **(1 Punkt)**

```
TREE-INSERT-RECURSIVE(n, x)
 1  // n: Teilbaum
 2  // x: Einzufügender Knoten
 3  if n.key < x.key // rechter Teilbaum
 4      if n.right-child == NIL
 5          n.right-child = x
 6          x.parent = n
 7      else
 8          TREE-INSERT-RECURSIVE(n.right-child, x)
 9  else // linker Teilbaum – verfahre analog
10      if n.left-child == NIL
11          n.left-child = x
12          x.parent = n
13      else
14          TREE-INSERT-RECURSIVE(n.left-child, x)
```

Jeder Knoten n eines Baumes T definiert einen Teilbaum, der auch ein binärer Suchbaum ist. Gegeben einen solchen Teilbaum bestimmen wir, ob das neue Element im linken oder rechten Kind-Teilbaum eingefügt werden soll — das können wir bei einem NIL-Kind direkt machen, ansonsten per rekursivem Aufruf für den bestehenden Kind-Teilbaum.

5. Ein Knoten x in einem binären Suchbaum habe zwei Kinder. Zeigen Sie, dass der Nachfolger von x kein linkes Kind und der Vorgänger von x kein rechtes Kind hat. **(1 Punkt)**

Nachfolger Nach Definition ist der Nachfolger y von x das Minimum des rechten Unterbaumes von x. Angenommen z ist ein linkes Kind des Nachfolgers y, dann ist z kleiner als y. Somit wäre y nicht der Nachfolger von x. Daraus folgt die Behauptung.

Vorgänger analog.

Theoretische Aufgaben: Repetition

In diesem Teil werden prüfungsrelevante Aufgabenstellungen aus den vorderen Kapiteln wiederholt.

1. Gegeben seien zwei *zyklische, doppelt verkettete* Listen a und b. Nehmen Sie an, die Listen hätten je ein Wächterelement NIL_a und NIL_b. Geben Sie Pseudocode für eine Funktion CONCATENATE(NIL_a, NIL_b), welche der Liste a alle Elemente der Liste b anhängt. Nach Aufruf von CONCATENATE besteht also die Liste a aus den Elementen von a gefolgt von den Elementen von b. Das Wächterelement NIL_b soll natürlich nicht in a eingefügt werden. Die Liste b soll am Ende leer sein. Behandeln Sie die Fälle korrekt, wo a und/oder b keine Elemente ausser dem Wächter enthalten. **Wichtig:** Ihr Algorithmus soll eine Zeitkomplexität von $\mathcal{O}(1)$ haben. **(1 Punkt)**

```
CONCATENATE(NIL_a, NIL_b)
 1  if NIL_b.next ≠ NIL_b // b ist nicht leer
 2      // Das erste Element von b an das letzte von a anhängen:
 3      NIL_b.next.prev = NIL_a.prev
 4      NIL_a.prev.next = NIL_b.next
 5
 6      // Das letzte Element von b vor NIL_a einhängen:
 7      NIL_b.prev.next = NIL_a
 8      NIL_a.prev = NIL_b.prev
 9
10      // b zur leeren Liste machen:
11      NIL_b.prev = NIL_b
12      NIL_b.next = NIL_b
```

2. Ordnen Sie die folgenden Funktionen nach ihrer asymptotischen Wachstumsrate:

   (a) $\sqrt{n^7}$
   (b) 2^n
   (c) $\log(n!)$
   (d) $\frac{n!}{n^n}$
   (e) $\log(\log(4n))$
   (f) $3n^2 + 2n + 1$
   (g) $5n - 2$
   (h) $2^{log_3(n)}$
   (i) $n^3 \log(n)$
   (j) 2^{12^8}

   (1 Punkt)

   1. $\frac{n!}{n^n} \in o\left(\frac{1}{n}\right)$
   2. $2^{12^8} \in \Theta(1)$
   3. $\log(\log(4n))$
   4. $2^{log_3(n)} = 2^{\frac{\log_2 n}{\log_2 3}} = \left(2^{\log_2 n}\right)^{\frac{1}{\log_2 3}} = n^{\frac{1}{\log_2 3}} = n^{\log_3 2} \approx n^{0.631}$
   5. $5n - 2 \in \Theta(n)$
   6. $\log(n!) \in \Theta(n \log n)$
   7. $3n^2 + 2n + 1 \in \Theta(n^2)$
   8. $n^3 \log(n)$
   9. $\sqrt{n^7} = n^{3.5}$
   10. 2^n

3. Geben Sie die asymptotische Laufzeit in Abhängigkeit von n für folgenden Algorithmus in Theta-Notation an:

MYSTERYFUNCTION(int n)

```
1  int x = n
2  int y = n
3  int i = 1
4  while x > 1
5        x = x/3
6        y = 2 · y
7        while i ≤ y
8              i = i + i
9  return i
```

(1 Punkt)

Die äuSSere Schleife wird $\Theta(\log_3(n))$ mal durchlaufen (so oft muss man $x = n$ durch 3 teilen, bis es ≤ 1 ist). Die innere Schleife hat im ersten Durchgang $\log_2(2n)$ viele Durchläufe (so oft muss man $i = 1$ verdoppeln, bis ein Wert $> y = 2n$ erreicht wird. In jedem folgenden äusseren Schleifendurchlauf wird y nur einmal verdoppelt, also gibt es nur $\Theta(1)$ innere Schleifendurchläufe (beachte, dass i nicht zurückgesetzt wird).

$$\begin{aligned} T(n) &= \log_2(2 \cdot n) + \Theta(1) \cdot log_3(n) \\ &= \Theta(\log(n)) \end{aligned}$$

4. Gegeben ist die folgende Rekursionsgleichung:

$$T(n) = \begin{cases} 9T(\frac{n}{3}) - 1 & \text{wenn } n > 1 \\ \frac{1}{4} & \text{wenn } n \leq 1 \end{cases}$$

Beweisen Sie mit vollständiger Induktion, dass die Rekursionsgleichung eine Lösung in $\mathcal{O}\left(\frac{n^2+1}{8}\right)$ hat. **(1 Punkt)**

Seien $c, d \in \mathbb{R}^+$. Wir wollen zeigen: $T(n) \leq cn^2 + d$ für alle $n \in \mathbb{R}, n \geq 1$.

Induktionsannahme Gelte $T(n') \leq c(n')^2 + d$ für alle n', $\frac{1}{3} \leq n' < n$

Induktionsschritt Betrachte $n \in \mathbb{R} > 1$.

$$\begin{aligned} T(n) &= 9T\left(\frac{n}{3}\right) - 1 \\ &\leq 9\left(c\left(\frac{n}{3}\right)^2 + d\right) - 1 && \text{Induktionsannahme} \\ &= cn^2 + 9d - 1 \\ &= cn^2 + d + \underbrace{8d - 1}_{\leq 0?} \\ &\leq cn^2 + d && \text{Wenn } 8d - 1 \leq 0 \iff d \leq \frac{1}{8} \end{aligned}$$

Mit einem geeignet gewähltem d können wir also den Induktionsschritt ausführen.

Induktionsverankerung Wir zeigen, dass unsere Aussage für alle n mit $\frac{1}{3} \leq n \leq 1$ gilt. Dies ist eine geeignete Verankerung, da von jeder Zahl $k > 1$ mit einer endlichen Anzahl

Rekursionsschritten (Division durch 3) ein Wert ≤ 1 erreicht wird, aber niemals einer kleiner als $\frac{1}{3}$.

Für $n \leq 1$ gilt $T(n) = \frac{1}{4}$, also z.Z. $cn^2 + d \geq T(n) = \frac{1}{4}$. d muss weiterhin die Bedingung $d \leq \frac{1}{8}$ erfüllen (s.o.). Da $cn^2 + d$ für $n > 0$ streng monoton wächst, reicht die Erfüllung der Gleichung für $n = \frac{1}{3}$, also $\frac{c}{9} + d \geq \frac{1}{4}$.

Dies ist beispielsweise für $d = \frac{1}{8}$ und $c = \frac{9}{8}$ erfüllt. (Anmerkung: wir könnten auch ein riesiges c wählen, wenn wir nur die Aussage in $\mathcal{O}$-Notation zeigen wollen; mit einem kleinen c erhalten wir aber eine bessere Schranke). Somit ist unsere Aussage per vollständiger Induktion bewiesen, es folgt $T \in \mathcal{O}\left(n^2\right)$.

Anmerkung: Wir haben hier eine Induktion über die reellen (oder rationalen) Zahlen geführt, da sich durch die Division ohne Runden nicht unbedingt ein ganzzahliger Wert ergibt. Wenn wir von $n \in \mathbb{N}$ und implizitem Runden ausgehen und aufrunden (abrunden macht es höchstens kleiner $\Longleftrightarrow$ einfacher zu beweisen), können wir trotzdem mit einem geeignetem d diese Rundung „ausgleichen“ und erfolgreich einen Induktionsbeweis führen.

5. In einem Naturgarten möchte man einen geraden Weg durch eine grosse Wiese bauen. Der Weg muss vom einen Ende zum anderen gehen, muss also eine exakt festgelegte Gesamtlänge L (ganzzahlig, in cm) haben. Er soll aus Granitplatten gelegt werden, von denen eine grosse Menge M zum Kauf bereitsteht. Alle verfügbaren Granitplatten haben die gleiche Breite, gerade so wie für den Weg gewünscht, haben aber ganz unterschiedliche Einzellängen l_i (ganzzahlig, in cm). Jede Granitplatte hat ihren Preis p_i, unabhängig von ihrer Einzellänge, aufgrund des unterschiedlichen Herstellers. Wir möchten nun einige der Granitplatten kaufen, so dass exakt die gewünschte Gesamtlänge des Wegs getroffen wird, dass wir aber insgesamt für die gekauften Platten möglichst wenig bezahlen müssen.

   Beispiel: Unser Weg ist 100 cm lang, und es stehen Platten mit (Länge, Preis) von $(60,5)$, $(55,6)$, $(35,7.5)$, $(25,12)$, $(15,14)$, und $(40,32)$ zur Verfügung. Wir kaufen die erste, vierte und fünfte Platte in dieser Aufzählung.

   Entwerfen Sie ein rekursives Programm in Pseudocode, welches den minimalen Preis für eine genau passende Auswahl von Platten berechnet (und ∞ liefert, falls es keine solche Auswahl gibt). Geben Sie die Laufzeit Ihres Algorithmus an. **(1 Punkt)**

   Vorschlag 1

   BEST-PRICE(M, L)

```
1  // M: Menge von Tupeln (l, p)
2  // L ≥ 0: Gewünschte Länge
3  if L == 0
4        return 0
5  M' = {(l, p) ∈ M | l ≤ L}
6  best = ∞
7  for (l, p) ∈ M'
8        x = p + BEST-PRICE(M' \ (l, p), L − l)
9        best = min{best, x}
10 return best
```

   Die Laufzeit dieses Algorithmus ist in $\mathcal{O}(n!)$.

   Vorschlag 2

BEST-PRICE2(M, L)

```
1  // M: Menge von Tupeln (l, p)
2  // L ≥ 0: Gewünschte Länge
3  if M == ∅
4       if L == 0
5            return 0
6       else
7            return ∞
8  Pick (l, p) ∈ M
9  p_mit = p + BEST-PRICE(M', L − l)
10 p_ohne = BEST-PRICE(M', L)
11 return min{p_mit, p_ohne}
```

Die Laufzeit dieses Algorithmus ist in $\mathcal{O}(2^n)$.