Überblick über die Vorlesung *Algorithmen und Datenstrukturen*

1. Unlösbarkeit des Halte-Problems

   Gegeben: Programm P
   Eingabe x **Frage**: terminiert Aufruf $P(x)$

2. Komplexität von Algorithmen

   Hilfsmittel:

   (a) *Rekurrenz-Gleichungen*
   diskrete Variante von Differential-Gleichungen

   (b) O*-Schreibweise*
   beschreibt Wachstumsverhalten von Funktionen

3. Abstrakte Daten-Typen und Daten-Strukturen

   Stack, später *Map*

4. Sortier-Algorithmen

   (a) *Insertion-Sort*

   (b) *Min-Sort*

   (c) *Merge-Sort*

   (d) *Quick-Sort*

5. Abbildungen (ADT *Map*)

   binäre Bäume, AVL-Bäume

6. Graphen

   Berechnung kürzester Wege

Ziel der Vorlesung

Einführung in die Denkweisen und Methoden zur

1. Konstruktion
2. Analyse
3. Verifikation

von Algorithmen.

Algorithmen und Programme

1. Algorithmus:
   (a) Konzept für systematisches Lösen eines Problems
   (b) abstrakt
   (c) oft: Darstellung als Pseudo-Code
   (d) auch: natürlich-sprachlicher Text
   (e) Logik und Mengen-Lehre
2. Programm:
   (a) ausführbar
   (b) konkret

Literatur
Algorithmen und Datenstrukturen

1. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

2. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *Data Structures and Algorithms*, Addison-Wesley, 1987.

3. *Frank M. Carrano*: *Data Abstraction and Problem Solving with* C++, Benjamin/Cummings Publishing Company, 1995.

4. *Frank M. Carrano* and *Janet J. Prichard*: *Data Abstraction and Problem Solving with Java*, Addison-Wesley, 2003.

   Eine Neuauflage des obigen Werkes, bei der die Algorithmen jetzt in *Java* statt C++ beschrieben werden.

5. *Thomas H. Cormen*, *Charles E. Leiserson*, *Ronald L. Rivest*, *Clifford Stein*: *Introduction to Algorithms*, MIT Press, 2001.

6. *Robert Sedgewick*: *Algorithms in Java*, Pearson, 2002.

7. *Heinz-Peter Gumm und Manfred Sommer*, *Einführung in die Informatik*, Oldenbourg Verlag, 2006.

Test-Funktionen: Beispiele

Def.: (Test-Funktion)

t Test-Funktion mit Namen n g.d.w.

1. t ist ein String,
2. $t =$ "`int` n(`char* x`) $\{ \cdots \}$"
3. t läßt sich vom C-Compiler fehlerfrei parsen

Beispiele

1. s_1 = "`int simple(char* x) { return 0; }`"
2. s_2 = "`int loop(char* x) { while (1) ++x; }`"
3. s_3 = "`int loop(char* x);`"

   Deklaration, keine Definition, also keine Test-Funktion
4. s_4 = "`int hugo(char* x) begin i := 1; end;`"

   s_4 ist keine Test-Funktion, denn es läßt sich mit einem `C`-Compiler nicht fehlerfrei parsen.
5. s_5 = "`int hugo(int x) { return i*i; }`"

   s_5 ist auch keine Test-Funktion, denn der Typ des Arguments ist `int` und nicht `char*`.

Der String Turing

Halte-Problem: Gibt es C-Funktion

$\texttt{int stops(char*}\ t,\ \texttt{char*}\ a)$

mit fogenden Eigenschaften:

1. $t \notin TF \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 2.$
2. $t \in TF \wedge \texttt{name}(t) = n \wedge n(a) \downarrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 1.$
3. $t \in TF \wedge \texttt{name}(t) = n \wedge n(a) \uparrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 0.$

```
Turing := "int turing(char* x) {
               int result;
               result = stops(x, x);
               if (result == 1) {
                   while (1) {
                       ++result;
                   }
               }
               return result;
           }"
```

Das Äquivalenz-Problem

Äquivalenz-Problem: Gibt es `C`-Funktion

```
int equal(char* p1, char* p2, char* a)
```

mit folgender Spezifikation:

1. $p_1 \notin TF \;\vee\; p_2 \notin TF \quad \Leftrightarrow \quad \texttt{equal}(p_1, p_2, a) \rightsquigarrow 2$.
2. Falls
   (a) $p_1 \in TF \;\wedge\; \texttt{name}(p_1) = n_1$,
   (b) $p_2 \in TF \;\wedge\; \texttt{name}(p_2) = n_2$ und
   (c) $n_1(a) \simeq n_2(a)$

   gilt, dann: $\texttt{equal}(p_1, p_2, a) \rightsquigarrow 1$.
3. Sonst: $\texttt{equal}(p_1, p_2, a) \rightsquigarrow 0$.

```
int stops(char* p, char* a) {
    char* s =
        "int loop(char* x) { while (1) {++x;} }";
    int e = equal(s, p, a);
    if (e == 2) {
        return 2;
    } else {
        return 1 - e;
    }
}
```

Business-Plan: *Bunnies Incorporated*

Biologie: Kaninchen-Axiome
Fibonacci alias Leonardo Bigollo, ca. 1170 − 1250

1. Jedes Kaninchen-Paar bringt jeden Monat ein neues Kaninchen-Paar zur Welt.

2. Kaninchen haben nach zwei Monaten Junge.

3. (Genetisch manipulierte) Kaninchen leben ewig.

Geschäftsplan: ein frisches Kaninchen-Paar kaufen

1. $k(0) = 1$

2. $k(1) = 1$

3. $k(2) = 1 + 1$

4. Nach $n + 2$ Monaten:
$$k(n+2) = k(n+1) + k(n)$$
   (a) $k(n)$: Kaninchen, die vor zwei Monaten schon da waren, bekommen Junge

   (b) $k(n+1)$: Kaninchen sind unsterblich

Berechnung der Fibonacci-Zahlen

```java
public class Fibonacci
{
  public static int fibonacci(int n)
  {
      if (n == 0)
          return 1;
      if (n == 1)
          return 1;
      return fibonacci(n - 1) + fibonacci(n - 2);
  }

  public static void main(String[] args)
  {
      for (int i = 0; i < 100; ++i) {
          int n = fibonacci(i);
          System.out.printf("fib(%d) = %d\n", i, n);
      }
  }
}
```

Übersetzen: `javac Fibonacci.java`

Ausführen: `java Fibonacci`

Lineare Rekurrenz-Gleichungen (homogen, nicht entartet)

$$a_{n+k} = \sum_{i=0}^{k-1} c_i \cdot a_{n+i} \quad \text{für alle } n \in \mathbb{N}.$$

Anfangs-Bedingungen:

$$a_0 = d_0, \cdots, a_{k-1} = d_{k-1}$$

charakteristisches Polynom: $\chi(x) = x^k - \sum_{i=0}^{k-1} c_i \cdot x^i$ charakteristisches Polynom nicht entartet, falls

$$\chi(x) = \prod_{i=1}^{k} (x - \lambda_i) \quad \text{mit } i \neq j \rightarrow \lambda_i \neq \lambda_j$$

allgemeine Lösung:

$$a_n = \sum_{i=1}^{k} \alpha_i \cdot \lambda_i^n$$

Anfangs-Bedingungen liefern lineares Gleichungssystem zur Bestimmung der α_i

$$\begin{array}{lcl} d_0 & = & \lambda_1^0 \cdot \alpha_1 + \cdots + \lambda_k^0 \cdot \alpha_k \\ d_1 & = & \lambda_1^1 \cdot \alpha_1 + \cdots + \lambda_k^1 \cdot \alpha_k \\ \vdots & & \vdots \\ d_{k-1} & = & \lambda_1^{k-1} \cdot \alpha_1 + \cdots + \lambda_k^{k-1} \cdot \alpha_k \end{array}$$

Lineare Rekurrenz-Gleichungen bei entartetem charakteristischen Polynom

$$a_{n+k} = \sum_{i=0}^{k-1} c_i \cdot a_{n+i} \quad \text{für alle } n \in \mathbb{N}.$$

Anfangs-Bedingungen:

$$a_0 = d_0, \cdots, a_{k-1} = d_{k-1}$$

charakteristisches Polynom: $\chi(x) = x^k - \sum_{i=0}^{k-1} c_i \cdot x^i$

charakteristisches Polynom entartet, falls $\chi(x)$ mehrfache Nullstelle hat:

$$\chi(x) = (x - \lambda)^r \cdot \phi(x) \quad \text{mit } r \geq 2$$

allgemeine Lösung:

$$a_n = \sum_{i=0}^{r-1} \alpha_i \cdot n^i \cdot \lambda^n + \text{Lsg. von } \phi(x)$$

bestimme Koeffizienten α_i aus Anfangs-Bedingungen

Inhomogene lineare Rekurrenz-Gleichungen

$$a_{n+k} = \sum_{i=0}^{k-1} c_i \cdot a_{n+i} + c_{-1} \quad \text{für alle } n \in \mathbb{N}.$$

Anfangs-Bedingungen:

$$a_0 = d_0, \cdots, a_{k-1} = d_{k-1}$$

charakteristisches Polynom: $\chi(x) = x^k - \sum_{i=0}^{k-1} c_i \cdot a_{n+i}$.

Spur: $\mathtt{sp}(\chi) := \chi(1) = 1 - \sum_{i=0}^{k-1} c_i$.

1. Fall: $\mathtt{sp}(\chi) \neq 0$.

   Ansatz für spezielle Lösung: $a_n = \delta$

   Lösung:
   $$a_n = \delta = \frac{c_{-1}}{\mathtt{sp}(\chi)}$$

2. Fall: $\mathtt{sp}(\chi) = 0$, $\chi'(1) \neq 0$.

   Ansatz für spezielle Lösung: $a_n = \varepsilon \cdot n$

   Lösung:
   $$a_n = \frac{c_{-1}}{\chi'(1)} \cdot n$$

3. Fall: $\mathtt{sp}(\chi) = 0$, $\chi'(1) = 0$, $\chi''(1) \neq 0$.

   Ansatz für spezielle Lösung: $a_n = \varepsilon \cdot n^2$

   Lösung:
   $$a_n = \frac{c_{-1}}{\chi''(1)} \cdot n^2$$

Inhomogene lineare Rekurrenz-Gleichungen

allg. Lösung inhomogenene lineare Rek. Gl.:

allg. Lösung homogene Rek. Gl. $+$ spezielle Lösung

Bestimmung der Parameter der allg. Lösung:

Einsetzen der Anfangs-Bedingungen

Inhomogene lineare Rekurrenz-Gleichungen nicht-konstante Inhomogenität

$$a_{n+1} = 2 \cdot a_n + n \quad \text{mit } a_0 = 0 \tag{1}$$

1. Substitutions-Schritt: $n \mapsto n+1$

   $$a_{n+2} = 2 \cdot a_{n+1} + n + 1 \tag{2}$$

2. Subtraktions-Schritt: (2) - (1)

   $$a_{n+2} = 3 \cdot a_{n+1} - 2 \cdot a_n + 1 \tag{3}$$

   1.+2.: *diskretes Differenzieren*

3. Berechne zusätzliche Anfangs-Bedingungen:

   $a_1 = 2 \cdot a_0 + 0 = 0.$

4. Lösen der inhomogenen Rekurrenz-Gleichung mit konstanter Inhomogenität

quadratische Inhomogenität: $a_{n+k} = \sum_{i=0}^{k-1} c_i \cdot a_{n+i} + c_{-1} \cdot n^2$

2 mal diskretes Differenzieren

$\mathcal{O}$-Notation: Motivation

Berechnung der Rechenzeit eines Programms

1. Kodiere Algorithmus in Programmiersprache
2. Rekurrenz-Gleichungen für Anzahl

   Zuweisungen, Additionen, Subtraktionen, $\cdots$
3. Dauer der Operationen: Prozessor-Handbuch:

   Gesamtlaufzeit

Probleme:

1. Verfahren sehr kompliziert
2. Cache und Pipelining:

   genaue Dauer von Operationen nicht vorhersagbar
3. Ergebnis hängt ab von

   Programmiersprache und Prozessor

   Keine Aussage über Algorithmus

Benötigt: abstrakterer Begriff zur Angabe von Rechenzeit

Abstrakter Begriff zur Erfassung der Rechenzeit

1. Abstraktion von konstanten Faktoren
   kein wesentlicher Unterschied zwischen n und $2 \cdot n$

2. Abstraktion von unwesentlichen Termen

   $T(n) = 3 \cdot n^3 + 2 \cdot n^2 + 7$

n	$\dfrac{2 \cdot n^2 + 7}{3 \cdot n^3 + 2 \cdot n^2 + 7}$
1	0.75000000000000
10	0.06454630495800
100	0.00662481908150
1000	0.00066622484855
10 000	6.6662224852 e -05

3. Wachstum Rechenzeit vs. Wachstum Eingabe
   Verhalten bei großen Eingaben entscheidend

Naive Berechnung der Potenz

```
static BigInteger power(BigInteger base, int n)
{
    if (n == 0)
        return BigInteger.valueOf(1);
    BigInteger result = base;
    for (int i = 2; i <= n; ++i) {
        result = result.multiply(base);
    }
    return result;
}
```

Berechnung der Potenz durch *Iterative Squaring*

```
// power(m, n) = m^n
static BigInteger power(BigInteger m, int n)
{
    if (n == 0)
        return BigInteger.valueOf(1);
    BigInteger p = power(m, n / 2);
    if (n % 2 == 0) \{
        return p.multiply(p);
    } else {
        return p.multiply(p).multiply(m);
    }
}
```

Wertverlaufs-Induktion

1. **Induktions-Anfang**

   Methode korrekt, wenn kein rekursiver Aufruf

2. **Induktions-Schritt**

   Methode korrekt bei rekursivem Aufruf

   Induktions-Voraussetzung:

   rekursiver Aufruf liefert korrektes Ergebnis

Rekursive Divide-and-Conquer Multiplikation

Spezifikation des Algorithmus

1. $n \cdot 0 = 0$,

2. $n \,\%\, 2 = 0 \rightarrow m \cdot n = m \cdot (n/2) \cdot 2$,

3. $n \,\%\, 2 = 1 \rightarrow m \cdot n = m \cdot (n/2) \cdot 2 + m$.

   $n/2$: Ganzzahl-Divison durch 2.

Implementierung:

```
static int multiply(int m, int n)
{
    if (n == 0)
        return 0;
    int p = multiply(m, n >> 1);
    if (n % 2 == 0) {
        return p << 1;
    } else {
        return (p << 1) + m;
    }
}
```

Hauptsatz der Laufzeit-Funktionen
Master Theorem

Vor.:

1. $\alpha, \beta \in \mathbb{N}$ mit $\alpha \geq 1$ und $\beta > 1$,

2. $f : \mathbb{N} \to \mathbb{R}_+$,

3. $g : \mathbb{N} \to \mathbb{R}_+$ genügt der Rekurrenz-Gleichung
   $$g(n) = \alpha \cdot g\,(n/\beta) + f(n),$$
   n/β: ganzzahlige Division

Beh.:

1. Falls $\varepsilon > 0$ mit
   $$f(n) \in \mathcal{O}(n^{\log_\beta(\alpha)-\varepsilon})$$
   existiert, dann gilt:
   $$g(n) \in \mathcal{O}\left(n^{\log_\beta(\alpha)}\right).$$

2. Falls sowohl $f(n) \in \mathcal{O}(n^{\log_\beta(\alpha)})$ als auch $n^{\log_\beta(\alpha)} \in \mathcal{O}(f(n))$ gilt, dann folgt
   $$g(n) \in \mathcal{O}(\log_\beta(n) \cdot n^{\log_\beta(\alpha)}).$$

3. Falls $\gamma < 1$, $k \in N$ und $n \geq k$
   $$\alpha \cdot f\,(n/\beta) \leq \gamma \cdot f(n)$$
   gilt, dann folgt
   $$g(n) \in \mathcal{O}(f(n)). \qquad \square$$

Abstrakte Daten-Typen (ADTs)

Abstrakter Daten-Typ: 5-Tupel $\langle T, P, Fz, Ts, Ax\rangle$

1. T: *Name*

2. P: Menge der *Typ-Parameter*
   Typ-Parameter: String

3. Fz: Menge der Funktions-Zeichen

4. Ts: Menge von Typ-Spezifikationen
   $f : T_1 \times \cdots \times T_n \to S$.

   T_1, $\cdots$, T_n, S: Namen von Daten-Typen
   (a) konkrete Daten-Typen, z. B. "`int`" oder "`String`"
   (b) abstrakte Daten-Typen
   (c) Typ-Parameter

   Forderung: $T_1 = T$ oder $S = T$
   $S = T$: f ist *Konstruktor*
   sonst: f *Methode*

5. Ax: Menge von *Axiomen*

ADT Stack

ADT Stack $= \langle T, P, Fz, Ts, Ax \rangle$

1. $T =$ *Stack*
2. $P = \{$*Element*$\}$
3. $Fz = \{$*Stack*, *push*, *pop*, *top*, *isEmpty*$\}$
4. Typ-Spezifikationen:
   (a) *Stack* : *Stack*
   Default-Konstruktor, weil keine Argumente
   (b) *push* : *Stack* $\times$ *Element* $\rightarrow$ *Stack*
   (c) *pop* : *Stack* $\rightarrow$ *Stack*
   (d) *top* : *Stack* $\rightarrow$ *Element*
   (e) *isEmpty* : *Stack* $\rightarrow \mathbb{B}$
5. Axiome
   (a) *Stack*().*top*() $= \Omega$
   (b) S.*push*(x).*top*() $= x$
   (c) *Stack*().*pop*() $= \Omega$
   (d) S.*push*(x).*pop*() $= S$
   (e) *Stack*().*isEmpty*() $=$ `true`
   (f) S.*push*(x).*isEmpty*() $=$ `false`

Stack: Anwendungen

1. Java-Byte-Code

   JVM (Java Virtual Machine): Stack-Maschine

2. *PostScript*

3. Parameter-Übergabe bei Funktionen

4. Parser

   (a) *Operator Precedence Parser*

   (b) *Bottom-Up Parser*

5. Such-Algorithmen: *Depth First Search*

6. $\cdots$

Berechnung des Produktes

```
public static int multiply(int m, int n) {
    if (n == 0) {
        return 0;
    }
    int mTimesN2 = multiply(m, n / 2);
    int twice    = mTimesN2 + mTimesN2;
    if (n % 2 == 1) {      // n is odd
        return twice + m;
    }
    return twice;
}
```

Berechnung der Wurzel

```
public static int intSqrt(int n) {
    if (n == 0) {
        return 0;
    } else {
        int root2  = 2 * intSqrt(n / 4);
        int root21 = root2 + 1;
        if (root21 * root21 <= n) {
            return root21;
        } else {
            return root2;
        }
    }
}
```

Sortieren durch Einfügen

1. $\mathtt{sort}([]) = []$

2. $\mathtt{sort}([x] + R) = \mathtt{insert}(x, \mathtt{sort}(R))$

3. $\mathtt{insert}(x, []) = [x]$.

4. $x \preceq y \rightarrow \mathtt{insert}(x, [y] + R) = [x, y] + R$.

5. $\neg x \preceq y \rightarrow \mathtt{insert}(x, [y] + R) = [y] + \mathtt{insert}(x, R)$.

6. $\mathtt{le}(x, []) = \mathtt{true}$

7. $x \preceq y \rightarrow \mathtt{le}(x, [y] + R) = \mathtt{le}(x, R)$

8. $\neg x \preceq y \rightarrow \mathtt{le}(x, [y] + R) = \mathtt{false}$

9. $\mathtt{isSorted}([]) = \mathtt{true}$.

10. $\mathtt{isSorted}([x] + R) = (\mathtt{le}(x, R) \wedge \mathtt{isSorted}(R))$.

11. $$\mathrm{eq}(x, y) = \begin{cases} 1 & \text{falls} \quad x = y \\ 0 & \text{falls} \quad x \neq y. \end{cases}$$

12. $\mathtt{count}(x, []) = 0$.

13. $\mathtt{count}(x, [y] + R) = \mathrm{eq}(x, y) + \mathtt{count}(x, R)$.

Eigenschaften des Algorithmus *Sortieren durch Einfügen*

1. Distributivität von `le` über `insert`

   $\texttt{le}(x, \texttt{insert}(y, L)) \leftrightarrow x \preceq y \wedge \texttt{le}(x, L)$.

2. Transitivität von `le`

   $x \preceq y \wedge \texttt{le}(y, L) \rightarrow \texttt{le}(x, L)$

3. Es sei $L = [y] + R$. Dann gilt

   $x \preceq y \wedge \texttt{isSorted}(L) \rightarrow \texttt{le}(x, L)$.

4. Distributivität von `isSorted` über `insert`

   $\texttt{isSorted}(\texttt{insert}(x, S)) \leftrightarrow \texttt{isSorted}(S)$

5. Korrektheit von `sort`, Teil 1 $\texttt{isSorted}(\texttt{sort}(L))$.

6. Distributivität von `count` über `insert`

   $\texttt{count}(x, \texttt{insert}(y, S)) = \texttt{eq}(x, y) + \texttt{count}(x, S)$

7. Distributivität von `count` über `sort`

   $\texttt{count}(x, L) = \texttt{count}(x, \texttt{sort}(L))$

1. $\texttt{sort}([]) = []$
2. $L \neq [] \rightarrow \texttt{sort}(L) = [\texttt{min}(L)] + \texttt{sort}(\texttt{delete}(\texttt{min}(L), L))$
3. $\texttt{delete}(x, []) = []$.
4. $\texttt{delete}(x, [x] + R) = R$.
5. $x \neq y \rightarrow \texttt{delete}(x, [y] + R) = [y] + \texttt{delete}(x, R)$.
6. $\texttt{min}([]) = \infty$.
7. $\texttt{min}([x] + R) = \texttt{min}(x, \texttt{min}(R))$.
8. $\texttt{min}(x, y) = \begin{cases} x & \text{falls } x \preceq y\,; \\ y & \text{sonst.} \end{cases}$

Eigenschaften des Algorithmus *Sortieren durch Auswahl*

1. $\texttt{le}(x, L) \leftrightarrow x \preceq \texttt{min}(L)$
2. $\texttt{le}(\texttt{min}(L), L)$
3. $x \preceq y \;\rightarrow\; (\texttt{le}(x, \texttt{delete}(y, L)) \leftrightarrow \texttt{le}(x, L))$
4. $\texttt{le}(x, \texttt{sort}(L)) \leftrightarrow \texttt{le}(x, L)$
5. $\texttt{isSorted}(\texttt{sort}(L))$
6. $\texttt{member}(x, []) = 0$
7. $\texttt{member}(x, [x] + R) = 1$
8. $x \neq y \;\rightarrow\; \texttt{member}(x, [y] + R) = \texttt{member}(x, R)$
9. $\texttt{count}(x, \texttt{delete}(y, L)) =$
   $\texttt{count}(x, L) - \texttt{eq}(x, y) * \texttt{member}(y, L)$
10. $\texttt{count}(x, \texttt{sort}(L)) = \texttt{count}(x, L)$

Sortieren durch Mischen

1. $\#L < 2 \rightarrow \texttt{sort}(L) = L$.
2. $\#L \geq 2 \ \rightarrow$
   $\texttt{sort}(L) = \texttt{merge}(\texttt{sort}(\texttt{split}_1(L)), \texttt{sort}(\texttt{split}_2(L)))$
3. $\texttt{split}(\texttt{[]}) = [\texttt{[]}, \texttt{[]}]$.
4. $\texttt{split}([x]) = [[x], \texttt{[]}]$.
5. $\texttt{split}(R) = [R_1, R_2] \ \rightarrow$
   $\texttt{split}([x, y] + R) = [[x] + R_1, [y] + R_2]$
6. $\texttt{merge}([], L_2) = L_2$.
7. $\texttt{merge}(L_1, []) = L_1$.
8. $x \preceq y \ \rightarrow$
   $\texttt{merge}([x] + R_1, [y] + R_2) = [x] + \texttt{merge}(R_1, [y] + R_2)$.
9. $\neg x \preceq y \ \rightarrow$
   $\texttt{merge}([x] + R_1, [y] + R_2) = [y] + \texttt{merge}([x] + R_1, R_2)$.

Eigenschaften des Algorithmus *Sortieren durch Mischen*

1. $\texttt{le}(x, \texttt{merge}(L_1, L_2)) \leftrightarrow \texttt{le}(x, L_1) \wedge \texttt{le}(x, L_2)$.
2. $\texttt{isSorted}(\texttt{merge}(L_1, L_2)) \leftrightarrow \texttt{isSorted}(L_1) \wedge \texttt{isSorted}(L_2)$.
3. $\texttt{isSorted}(\texttt{sort}(L))$.
4. $\texttt{count}(x, \texttt{sort}(L)) = \texttt{count}(x, L)$.

```
private void merge(int start, int middle, int end) {
    for (int i = start; i < end; ++i) {
        mAux[i] = mArray[i];
    }
    int idx1 = start;
    int idx2 = middle;
    int i    = start;
    while (idx1 < middle && idx2 < end) {
        if (mAux[idx1] <= mAux[idx2]) {
            mArray[i++]  = mAux[idx1++];
        } else {
            mArray[i++]  = mAux[idx2++];
        }
    }
    while (idx1 < middle) {
        mArray[i++]  = mAux[idx1++];
    }
    while (idx2 < end) {
        mArray[i++]  = mAux[idx2++];
    }
}
```

Spezifikation: Wenn vor dem Aufruf

1. $\texttt{mArray}[\texttt{start}, \cdots, \texttt{middle} - 1]$ und
2. $\texttt{mArray}[\texttt{middle}, \cdots, \texttt{end} - 1]$ sortiert ist,

dann ist nach dem Aufruf

$$\texttt{mArray}[\texttt{start}, \cdots, \texttt{end} - 1]$$

sortiert.

Merge-Sort, nicht-rekursiv

```
private void mergeSort() {
    for (int l = 1; l < mArray.length; l *= 2) {
        int k;
        for (k = 0; l * (k + 1) <= mArray.length;
             k += 2)
        {
            merge(l * k, l * (k + 1),
                  min(l * (k + 2), mArray.length));
        }
    }
}
```

Invariante:

Für alle passenden k und l gilt: Das Teilfeld

$$\Big[\texttt{mArray}[k \cdot l,\ \cdots,\ k \cdot l + (l-1)]\Big]$$

ist sortiert.

Quick-Sort (Spezifikation)

1. $\texttt{sort}([]) = []$.

2. $\textit{partition}(x, R) = \langle S, B \rangle \rightarrow$
   $\textit{sort}([x] + R) = \textit{sort}(S) + [x] + \textit{sort}(B)$.

3. $\textit{partition}(x, []) = \langle [], [] \rangle$,

4. $x \preceq y \ \wedge \textit{partition}(x, R) = \langle S, B \rangle \rightarrow$
   $\textit{partition}(x, [y] + R) = \langle [x] + S,\ B \rangle$

5. $\neg\, (x \preceq y) \wedge \textit{partition}(x, R) = \langle S, B \rangle \rightarrow$
   $\textit{partition}(x, [y] + R) = \langle S,\ [x] + B \rangle$,

Quick-Sort `partition`

```java
private int partition(int start, int end) {
    Double x  = mArray[start];
    int left  = start + 1;
    int right = end;
    while (true) {
        while (left <= end && mArray[left] <= x) {
            ++left;
        }
        while (mArray[right] > x) { --right; }
        if (left >= right) { break; }
        swap(left, right);
    }
    swap(start, right);
    return right;
}
```

Invarianten

1. $\forall i \in \{\texttt{start} + 1, \cdots, \texttt{left} - 1\} \colon \texttt{mArray}[i] \preceq x$
2. $\forall j \in \{\texttt{right} + 1, \cdots, \texttt{end}\} \colon x \prec \texttt{mArray}[j]$
3. $\texttt{start} + 1 \leq \texttt{left}$
4. $\texttt{right} \leq \texttt{end}$
5. $\texttt{left} \leq \texttt{right} + 1$

Hoare-Tripel: $\{F\}$ `P` $\{G\}$

P erfüllt die Spez. "wenn vorher F, dann nachher G"

Zuweisungs-Regeln:

1. $\{F\}$ `x = h(x);` $\{F\sigma\}$ mit $\sigma = [x \mapsto h^{-1}(x)]$
2. $\{F\}$ `x = c;` $\{F \wedge x = c\}$

Abschwächungs-Regel:

$$\frac{\{F\} \quad \texttt{P} \quad \{G\}, \qquad G \to H}{\{F\} \quad \texttt{P} \quad \{H\}}$$

Zusammengesetze Anweisungen:

$$\frac{\{F_1\} \quad \texttt{P} \quad \{G_1\}, \qquad \{G_1\} \quad \texttt{Q} \quad \{G_2\}}{\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}}$$

Alternativ-Anweisung

$$\frac{\{F \wedge B\} \quad \texttt{P} \quad \{G\}, \qquad \{F \wedge \neg B\} \quad \texttt{Q} \quad \{G\}}{\{F\} \quad \texttt{if } (B) \ \{\ \texttt{P}\ \} \ \texttt{else} \ \{\ \texttt{Q}\ \} \quad \{G\}}$$

Schleifen

$$\frac{\{I \wedge B\} \quad \texttt{P} \quad \{I\}}{\{I\} \quad \texttt{while } (B) \ \{\ \texttt{P}\ \} \quad \{I \wedge \neg B\}}$$

Zahlentheorie

1. $\texttt{teiler}(a) = \{q \in \mathbb{N} \mid a \,\%\, q = 0\}$

2. $$\begin{aligned}\texttt{gt}(a,b) &= \texttt{teiler}(a) \cap \texttt{teiler}(b) \\ &= \{q \in \mathbb{N} \mid a \,\%\, q = 0 \wedge b \,\%\, q = 0\}\end{aligned}$$

3. $$\begin{aligned}\texttt{ggt}(a,b) &= \max(\texttt{gt}(a,b)) \\ &= \max\{q \in \mathbb{N} \mid a \,\%\, q = 0 \wedge b \,\%\, q = 0\}\end{aligned}$$

4. $x \,\%\, q = 0 \wedge y \,\%\, q = 0 \;\leftrightarrow\; (x+y) \,\%\, q = 0 \wedge y \,\%\, q = 0$

5. $\texttt{ggt}(x+y, y) = \texttt{ggt}(x,y)$

```
unsigned ggt(unsigned x, unsigned y) {
    while (x != y) {
        if (x < y) {
            y = y - x;
        } else {
            x = x - y;
        }
    }
    return x;
}
```

Invariante

$$I := (x > 0 \wedge y > 0 \wedge \texttt{ggt}(x,y) = \texttt{ggt}(a,b)).$$

Iterative Berechnung der Potenz

```
static BigInteger power(BigInteger x, int y)
{
    BigInteger r = BigInteger.valueOf(1);
    while (y > 0) {
        if (y % 2 == 1) {
            r = r.multiply(x);
        }
        x = x.multiply(x);
        y = y / 2;
    }
    return r;
}
```

Invariante: $I := (r * x^y = a^b)$

Indiziertes Programm

```
int power(int x_0, int y_0)
{
    int r_0 = 1;
    while (y_n > 0) {
        if (y_n % 2 == 1) {
            r_{n+1} = r_n * x_n;
        }
        x_{n+1} = x_n * x_n;
        y_{n+1} = y_n / 2;
    }
    return r_N;
}
```

1. $\textit{nil.insert}(k, v) = \textit{node}(k, v, \textit{nil}, \textit{nil})$.

2. $\textit{node}(k, v_2, l, r).\textit{insert}(k, v_1) = \textit{node}(k, v_1, l, r)$.

3. $k_1 < k_2 \;\rightarrow\; \textit{node}(k_2, v_2, l, r).\textit{insert}(k_1, v_1) =$
   $\textit{node}(k_2, v_2, l.\textit{insert}(k_1, v_1), r)$.

4. $k_1 > k_2 \;\rightarrow\; \textit{node}(k_2, v_2, l, r).\textit{insert}(k_1, v_1) =$
   $\textit{node}(k_2, v_2, l, r.\textit{insert}(k_1, v_1))$.

5. $\textit{node}(k, v, \textit{nil}, r).\textit{delMin}() = [r, k, v]$.

6. $l \neq \textit{nil} \wedge l.\textit{delMin}() = [l', k_{min}, v_{min}] \;\rightarrow$
   $\textit{node}(k, v, l, r).\textit{delMin}() = [\textit{node}(k, v, l', r), k_{min}, v_{min}]$.

7. $\textit{nil.delete}(k) = \textit{nil}$.

8. $\textit{node}(k, v, \textit{nil}, r).\textit{delete}(k) = r$.

9. $\textit{node}(k, v, l, \textit{nil}).\textit{delete}(k) = l$.

10. $l \neq \textit{nil} \wedge r \neq \textit{nil} \wedge r.\textit{delMin}() = [r', k_{min}, v_{min}] \;\rightarrow$
    $\textit{node}(k, v, l, r).\textit{delete}(k) = \textit{node}(k_{min}, v_{min}, l, r')$.

11. $k_1 < k_2 \;\rightarrow\; \textit{node}(k_2, v_2, l, r).\textit{delete}(k_1) =$
    $\textit{node}(k_2, v_2, l.\textit{delete}(k_1), r)$.

12. $k_1 > k_2 \;\rightarrow\; \textit{node}(k_2, v_2, l, r).\textit{delete}(k_1) =$
    $\textit{node}(k_2, v_2, l, r.\textit{delete}(k_1))$.

1. $\mathit{nil.restore}() = \mathit{nil}.$

2. $|\mathit{height}(l) - \mathit{height}(r)| \leq 1 \rightarrow$
$$\mathit{node}(k, v, l, r).\mathit{restore}() = \mathit{node}(k, v, l, r).$$

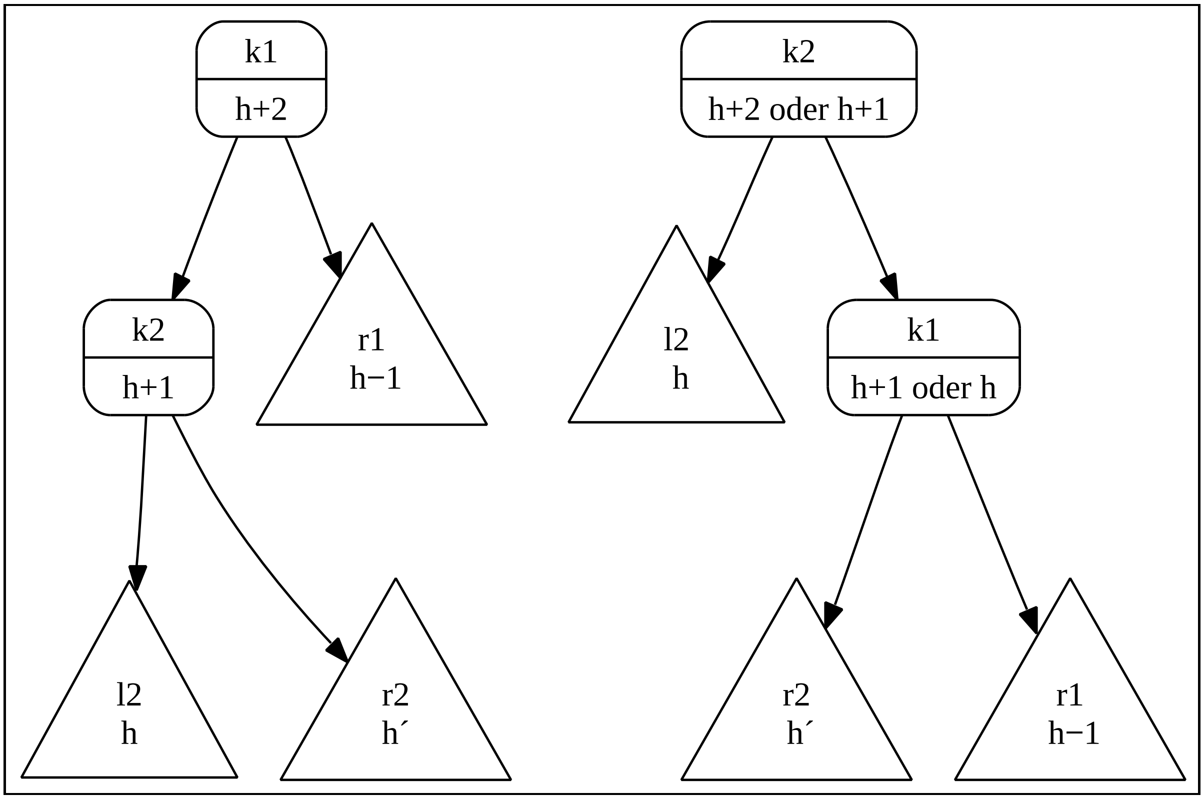
3.
$$\begin{array}{ll} & \mathit{height}(l_1) = \mathit{height}(r_1) + 2 \\ \wedge & l_1 = \mathit{node}(k_2, v_2, l_2, r_2) \\ \wedge & \mathit{height}(l_2) \geq \mathit{height}(r_2) \\ \rightarrow & \quad \mathit{node}(k_1, v_1, l_1, r_1).\mathit{restore}() \\ & = \mathit{node}(k_2, v_2, l_2, \mathit{node}(k_1, v_1, r_2, r_1)) \end{array}$$

4.
$$\begin{array}{ll} & \mathit{height}(l_1) = \mathit{height}(r_1) + 2 \\ \wedge & l_1 = \mathit{node}(k_2, v_2, l_2, r_2) \\ \wedge & \mathit{height}(l_2) < \mathit{height}(r_2) \\ \wedge & r_2 = \mathit{node}(k_3, v_3, l_3, r_3) \\ \rightarrow & \quad \mathit{node}(k_1, v_1, l_1, r_1).\mathit{restore}() \\ & = \mathit{node}(k_3, v_3, \mathit{node}(k_2, v_2, l_2, l_3), \mathit{node}(k_1, v_1, r_3, r_1)) \end{array}$$

5. weiterer Fall analog zu 3.

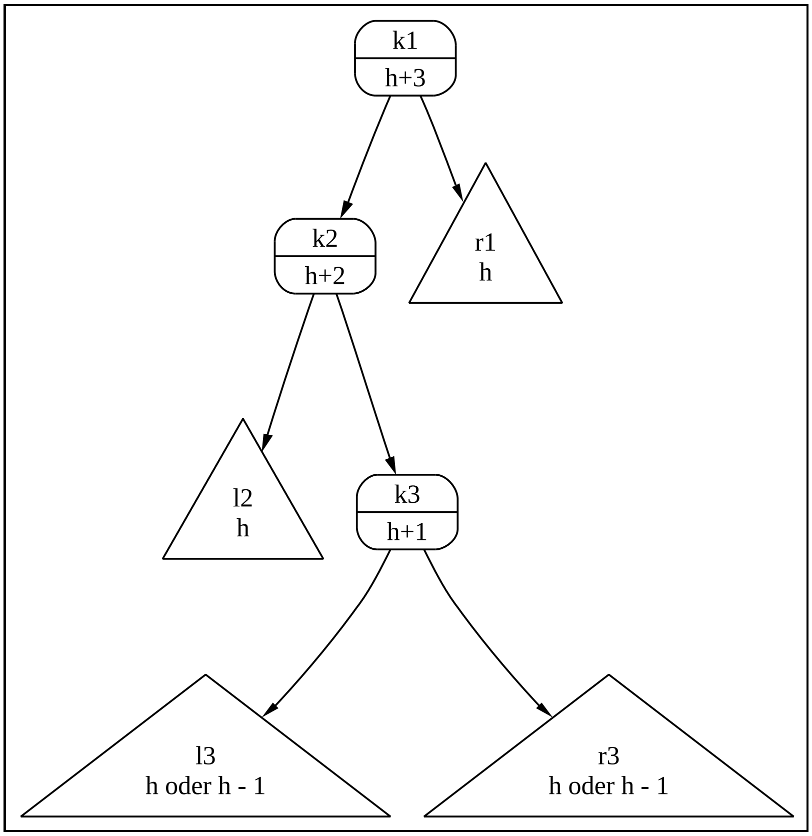
6. weiterer Fall analog zu 4.

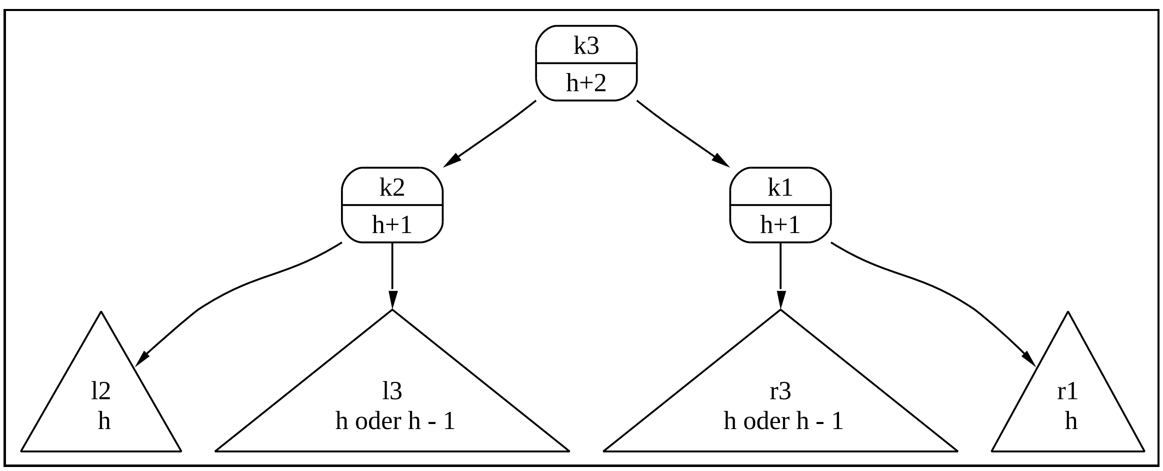
Implementierung von *restore*



$$
\begin{array}{ll}
 & \mathit{height}(l_1) = \mathit{height}(r_1) + 2 \\
\wedge & l_1 = \mathit{node}(k_2, v_2, l_2, r_2) \\
\wedge & \mathit{height}(l_2) \geq \mathit{height}(r_2) \\
\rightarrow & \quad \mathit{node}(k_1, v_1, l_1, r_1).\mathit{restore}() \\
 & = \mathit{node}(k_2, v_2, l_2, \mathit{node}(k_1, v_1, r_2, r_1))
\end{array}
$$

Implementierung von *restore*, 2. Fall





AVL-Bäume

1. $\mathit{nil}.\mathit{insert}(k, v) = \mathit{node}(k, v, \mathit{nil}, \mathit{nil})$.

2. $\mathit{node}(k, v_2, l, r).\mathit{insert}(k, v_1) = \mathit{node}(k, v_1, l, r)$.

3. $k_1 < k_2 \;\rightarrow\; \mathit{node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) =$
   $\mathit{node}(k_2, v_2, l.\mathit{insert}(k_1, v_1), r).\mathit{restore}()$.

4. $k_1 > k_2 \;\rightarrow\; \mathit{node}(k_2, v_2, l, r).\mathit{insert}(k_1, v_1) =$
   $\mathit{node}(k_2, v_2, l, r.\mathit{insert}(k_1, v_1)).\mathit{restore}()$.

5. $\mathit{node}(k, v, \mathit{nil}, r).\mathit{delMin}() = [r, k, v]$.

6. $l \neq \mathit{nil} \wedge l.\mathit{delMin}() = [l', k_{min}, v_{min}] \;\rightarrow$
   $\mathit{node}(k, v, l, r).\mathit{delMin}() \;=$
   $[\mathit{node}(k, v, l', r).\mathit{restore}(), k_{min}, v_{min}]$.

7. $\mathit{nil}.\mathit{delete}(k) = \mathit{nil}$.

8. $\mathit{node}(k, v, \mathit{nil}, r).\mathit{delete}(k) = r$.

9. $\mathit{node}(k, v, l, \mathit{nil}).\mathit{delete}(k) = l$.

10. $l \neq \mathit{nil} \;\wedge\; r \neq \mathit{nil} \;\wedge\; r.\mathit{delMin}() = [r', k_{min}, v_{min}] \;\rightarrow$
    $\mathit{node}(k, v, l, r).\mathit{delete}(k) = \mathit{node}(k_{min}, v_{min}, l, r').\mathit{restore}()$.

11. $k_1 < k_2 \;\rightarrow\; \mathit{node}(k_2, v_2, l, r).\mathit{delete}(k_1) =$
    $\mathit{node}(k_2, v_2, l.\mathit{delete}(k_1), r).\mathit{restore}()$.

12. $k_1 > k_2 \;\rightarrow\; \mathit{node}(k_2, v_2, l, r).\mathit{delete}(k_1) =$
    $\mathit{node}(k_2, v_2, l, r.\mathit{delete}(k_1)).\mathit{restore}()$.

```
public class ArrayMap<Value>
    implements MyMap<Integer, Value>
{
    Value[] mArray;

    public ArrayMap(int n) {
        mArray = (Value[]) new Object[n+1];
    }
    public Value find(Integer key) {
        return mArray[key];
    }
    public void insert(Integer key, Value value) {
        mArray[key] = value;
    }
    public void delete(Integer key) {
        mArray[key] = null;
    }
}
```

Abbildung von Namen auf Zahlen

1. Annahme: Namen haben Länge 8
   - kürzere Namen mit Blanks auffüllen
   - längere Namen abschneiden

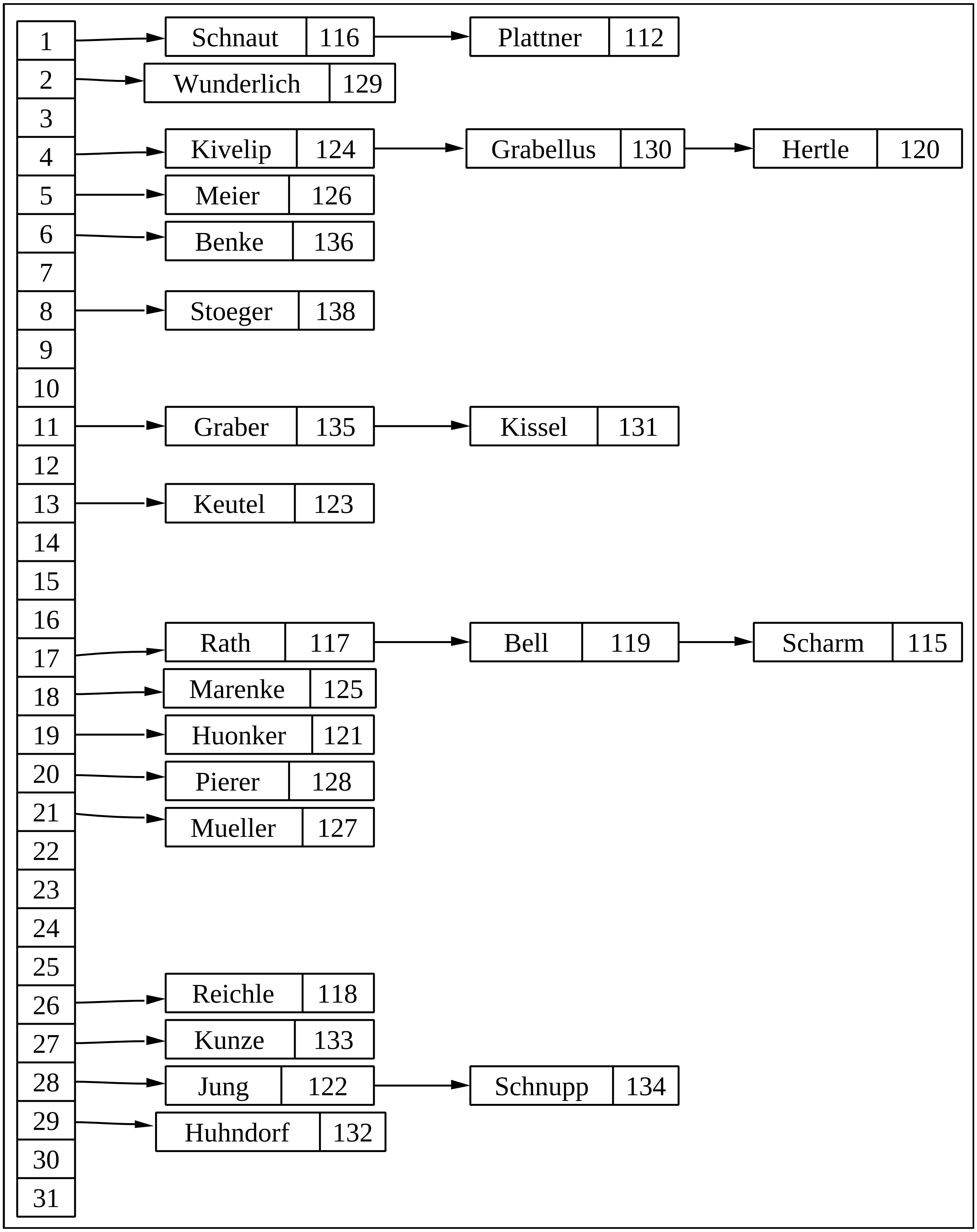
2. Buchstaben als Ziffern $\{0, \cdots 26\}$ interpretieren:
   $\Sigma = \{\texttt{' '}, \texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \cdots, \texttt{'x'}, \texttt{'y'}, \texttt{'z'}\}$
   Funktion *ord* $: \Sigma \to \{0, \cdots s6\}$ definieren durch:
   ord$(\texttt{' '}) = 0$, *ord*$(\texttt{'a'}) = 1$, $\cdots$, *ord*$(\texttt{'z'}) = 26$.

3. Definiere *code* $: \Sigma^* \to \mathbb{N}$
   $$\mathit{code}(c_0 c_1 \cdots c_7) = \sum_{i=0}^{7} \mathit{ord}(c_i) \cdot 27^i.$$

Probleme:

1. Größe des Feldes:
   $$(27^8 - 1)/26 + 1 = 10\,862\,674\,481$$

2. Funktion *code* $: \Sigma^* \to \mathbb{N}$ nicht bijektiv:
   nur die ersten 8 Buchstaben werden berücksichtigt.


1 Schnaut 116 Plattner 112
2 Wunderlich 129
3
4 Kivelip 124 Grabellus 130 Hertle 120
5 Meier 126
6 Benke 136
7
8 Stoeger 138
9
10
11 Graber 135 Kissel 131
12
13 Keutel 123
14
15
16
17 Rath 117 Bell 119 Scharm 115
18 Marenke 125
19 Huonker 121
20 Pierer 128
21 Mueller 127
22
23
24
25
26 Reichle 118
27 Kunze 133
28 Jung 122 Schnupp 134
29 Huhndorf 132
30
31

Das Interface `Collection<E>`: *Zusammenfassungen*

1. `boolean add(E` o`)`
   (a) $c.\texttt{add}(e) \to c' = c \cup \{e\}$
   c': Wert der Zusammenfassungen nachher
   (b) $c.\texttt{add}(e) = (c' \neq c)$

2. `boolean addAll(Collection<E>` d`)`
   (a) $c.\texttt{addAll}(d) \to c' = c \cup d$,
   (b) $c.\texttt{addAll}(d) = (c' \neq c)$.

3. `void clear()`

   $$c.\texttt{clear}() \to c' = \{\}.$$

4. `boolean contains(Element` e`)`

   $$c.\texttt{contains}(e) = (e \in c).$$

5. `boolean containsAll(Collection<E>` d`)`

   $$c.\texttt{containsAll}(d) = (d \subseteq c).$$

6. `boolean isEmpty()`

   $$c.\texttt{isEmpty}() = (c = \{\}).$$

8. `boolean remove(Object` e`)`
   (a) $c.\mathtt{remove}(e) \rightarrow c' = c \setminus \{e\}$,
   (b) $c.\mathtt{remove}(e) = (e \in c)$.

9. `boolean removeAll(Collection<?>` d`)`
   (a) $c.\mathtt{removeAll}(d) \rightarrow c' = c \setminus d$,
   (b) $c.\mathtt{removeAll}(d) = (d \cap c \neq \{\})$.

10. `boolean retainAll(Collection<?>` d`)`
    (a) $c.\mathtt{retainAll}(d) \rightarrow c' = c \cap d$,
    (b) $c.\mathtt{retainAll}(d) = (d \subseteq c)$.

11. `int size()`
    $c.\mathrm{size}() = \#c$ Anzahl der Elemente

12. `Object[] toArray()`
    Umwandlung: `Collection<T>` $\mapsto$ Feld

13. `T[] toArray(T[]` a`)`
    Umwandlung: `Collection<T>` $\mapsto$ Feld vom Typ `T[]`

Generische Felder

1. Keine generische Erzeugung von Feldern in Java:

   ```
   T[] a = new T[10]; // compile time error
   ```

2. Kein Casten von Feldern:

```
public class TestCast
{
    public static void main(String[] args) {
        List<Integer> l = new LinkedList<Integer>();
        for (Integer i = 0; i < 10; ++i) {
            l.add(i);
        }
        Object [] a = l.toArray();
        Integer[] b = (Integer[]) a;
    }
}
```

Exception: `ClassCastException`

Erzeugung generischer Felder:

```
public class TestCast2
{
    public static void main(String[] args) {
        List<Integer> l = new LinkedList<Integer>();
        for (Integer i = 0; i < 10; ++i) {
            l.add(i);
        }
        Integer[] a = new Integer[0];
        Integer[] b = l.toArray(a);
    }
}
```

`Collection<E>`, Fortsetzung

14. `Iterator<E> iterator()`

    c.`iterator`() liefert Iterator für Zusammenfassungen c

    ermöglicht erweiterte `for`-Schleife

    ```
    for (E e: c) {
       System.out.println(e);
    }
    ```

Schnittstelle `Iterator<E>`

1. `boolean hasNext()`

   noch nicht alle Elemente aufgezählt

2. `E next()`

   liefert nächstes Element

3. `void remove()`

   entfernt zuletzt geliefertes Element

`Set<E> implements Collection<E>`

Implementierung: `TreeSet<E>`

Bedingung: Elemente vergleichbar

1. **`E first()`**

   s.first(): kleinstes Element von s

2. **`E last()`**

   s.last(): größtes Element von s.

Konstruktoren

1. **`TreeSet()`**: leere Menge

2. **`TreeSet(Collection<E>`** c**`)`**

   alle Elemente aus c

Implementierung: Rot-Schwarz-Bäume

Rot-Schwarz-Bäume

1. Näherungsweise balancierte binäre Bäume.
2. Knoten markiert: rot oder schwarz
3. Wurzel: schwarz.
4. Kinder eines roten Knotens: schwarz.
5. Kinder eines schwarzen Knotens: rot oder schwarz.
6. Höhe: rote Knoten werden nicht gezählt.
7. Linker und rechter Teil-Baum: selbe Höhe.

Komplexität

1. *find*(): $\mathcal{O}(\log(n))$
2. *insert*(): $\mathcal{O}(\log(n))$
3. *delete*(): $\mathcal{O}(\log(n))$

`HashSet<E>`

1. `HashSet()`: leere Hash-Tabelle.
   Load-Faktor: 0.75
   Initiale Größe des Feldes: 16

2. `HashSet(Collection<E>` c`)`
   Load-Faktor: 0.75.

3. `HashSet(int` n`)`
   Initiale Größe: n.
   Load-Faktor: 0.75.

4. `HashSet(int` n`, float` α`)`
   Initiale Größe: n.
   Load-Faktor: α.

`List<E> implements Set<E>`

zusätzlich: Index

1. `E get(int` i`)`

   l.get(i): i-tes Element von l

2. `void set(int` i`, E` e`)`

   l.set(i, e): ersetze i-te Element.

3. `add`, `addAll`:

   Einfügen am Ende der Liste

4. `void add(int` i`, E` e`)`

   Einfügen an Position i

5. `void addAll(int` i`, Collection<E>` c`)`

   Einfügen an Position i

Implementierungen

1. `LinkedList`: verkettete Listen

   Komplexität l.get(i), l.set(i, e): linear

   Komplexität l.add(e): konstant

2. `ArrayList`: Feld

   Komplexität l.get(i), l.set(i, e): konstant

   Komplexität l.add($0, e$): linear

`Queue<E> implements Collection<E>`

1. `E element()`

   q.`element()`: erstes Element

   q wird nicht verändert

   q leer, dann `NoSuchElementException`

2. `boolean offer(E` e`)`

   q.`offer`(e): einfügen von e e am Ende

   q voll: `false`, sonst `true`.

3. `E peek()`

   q.`element()`: erstes Element

   q wird nicht verändert

   q leer: Ergebnis `null`

4. `E poll()`

   q.`poll()`: erstes Element

   Element wird entfernt

   q leer: Ergebnis `null` Warteschlange leer ist, wird `null` zurück gegeben.

5. `E remove()`

   q.`remove()`: erstes Element

   Element wird entfernt

   q leer: `NoSuchElementException`

Abbildungen: `Map<K,V>`

1. `V get(K` k`)`

   entspricht `find()`

2. `boolean containsKey(K` k`)`

   m.containsKey$(k) \approx (\exists k : m.\text{get}(k) \neq \texttt{null})$

3. `V put(K` k`, V` v`)`

   entspricht `insert()`

4. `V remove(K` k`)`

   entspricht `delete()`

5. `void clear()`

   löscht alle Zuordnungen

6. `boolean containsValue(V` v`)`

   m.containsValue$(v) \leftrightarrow (\exists k : m.\text{get}(k) = v)$

   Komplexität: linear

7. `boolean isEmpty()`

8. `Set<K> keySet()`

   $m.\texttt{keySet}() = \{k \mid m.\texttt{get}(k) \neq \texttt{null}\}$

   Ansicht der Schlüssel (engl. *view*)

   (a) Löschen in $m.\texttt{keySet}()$
       Zuordnung verschwindet aus m

   (b) kein Einfügen:
       $m.\texttt{keySet}().\texttt{add}(x) \mapsto \texttt{UnsupportedOperationException}$

9. `Collection<V> values()`

   Ansicht der Werte

   $m.\texttt{values}() = \{v \mid \exists k : m.\texttt{get}(k) = v\}$

   Löschen erlaubt, Einfügen nicht

10. `void putAll(Map<K,V>` t`)`

    $m.\texttt{putAll}(t) \approx m \cup t$

    Zuordnungen aus t überschreiben Zuordnungen aus m

11. `int size()`

    $m.\texttt{size}() = \#\{k \mid m.\texttt{get}(k) \neq \texttt{null}\}$

`TreeMap<K,V> implements Map<K,V>`

Bedingung: Schlüssel geordnet

1. natürlicher Vergleich: `K implements Comparable<K>`

   $k_1.\texttt{compareTo}(k_2)$

   (a) $k_1 < k_2 \leftrightarrow k_1.\texttt{compareTo}(k_2) < 0$

   (b) $k_1 = k_2 \leftrightarrow k_1.\texttt{compareTo}(k_2) = 0$

   (c) $k_1 > k_2 \leftrightarrow k_1.\texttt{compareTo}(k_2) > 0$

2. *Komparator-Objekt*

   $c.\texttt{compare}(o_1, o_2)$

Konstruktoren

1. `TreeMap()`

2. `TreeMap(Comparator<K>` c`)`

3. `TreeMap(Map<K,V>` m`)`

`HashMap<K,V> implements Map<K,V>`

1. `HashMap()`

   leere Hash-Tabelle.

   Load-Faktor: 0.75

   initiale Größe: 16

2. `HashMap(int` n`)`

   leere Hash-Tabelle.

   Load-Faktor: 0.75

   initiale Größe: n

3. `HashMap(int` n`, float` α`)`

   leere Hash-Tabelle.

   Load-Faktor: α

   initiale Größe: n

4. `HashMap(Map<K,V>` m`)`

   Load-Faktor: 0.75

Prioritäts-Warteschlangen

1. Namen: *PrioQueue*.

2. Typ-Parameter: $\{Key, Value\}$.
   $\langle Key, \leq \rangle$: totale Quasi-Ordnung

3. Funktions-Zeichen:
   $\{PrioQueue, insert, remove, top, change\}$.

4. Typ-Spezifikationen:
   (a) $PrioQueue : PrioQueue$
   (b) $top : PrioQueue \rightarrow (Key \times Value) \cup \{\Omega\}$
   (c) $insert : PrioQueue \times Key \times Value \rightarrow PrioQueue$
   (d) $remove : PrioQueue \rightarrow PrioQueue$
   (e) $change : PrioQueue \times Key \times Value \rightarrow PrioQueue$

Prioritäts-Warteschlangen — Axiome

Referenz-Implementierung

5. $\textit{new PrioQueue}() = \{\}$

6. $Q.\textit{insert}(k, v) = Q \cup \big\{\langle k, v\rangle\big\}$

7. $Q = \{\} \;\rightarrow\; Q.\textit{top}() = \Omega$

8. $\langle k_1, v_1\rangle \in Q \;\wedge\; Q.\textit{top}() = \langle k_2, v_2\rangle \;\rightarrow$
   $k_2 \leq k_1 \;\wedge\; \langle k_2, v_2\rangle \in Q$

9. $Q = \{\} \rightarrow Q.\textit{remove}() = Q$

10. $Q \neq \{\} \rightarrow Q.\textit{remove}() = Q \backslash \big\{Q.\textit{top}()\big\}$

11. $Q.\textit{change}(k_1, v_1) = \big\{\langle k_2, v_2\rangle \in Q \mid v_2 \neq v_1\big\} \cup \big\{\langle k_1, v_1\rangle\big\}$

Einfache Implementierungen: geordnete Liste

Komplexität von $Q.\textit{insert}(k, v)$: $\mathcal{O}(\#Q)$

$\leq$: $\mathit{Key} \times \mathcal{B} \to \mathbb{B}$

1. $k_1 \leq \mathit{nil}$

2. $k_1 \leq \mathit{node}(k_2, v, l, r) \overset{\text{def}}{\longleftrightarrow} k_1 \leq k_2 \ \wedge \ k_1 \leq l \ \wedge \ k_2 \leq r$

$\mathit{count} : \mathcal{B} \to \mathbb{N}$

1. $\mathit{nil.count}() = 0$.

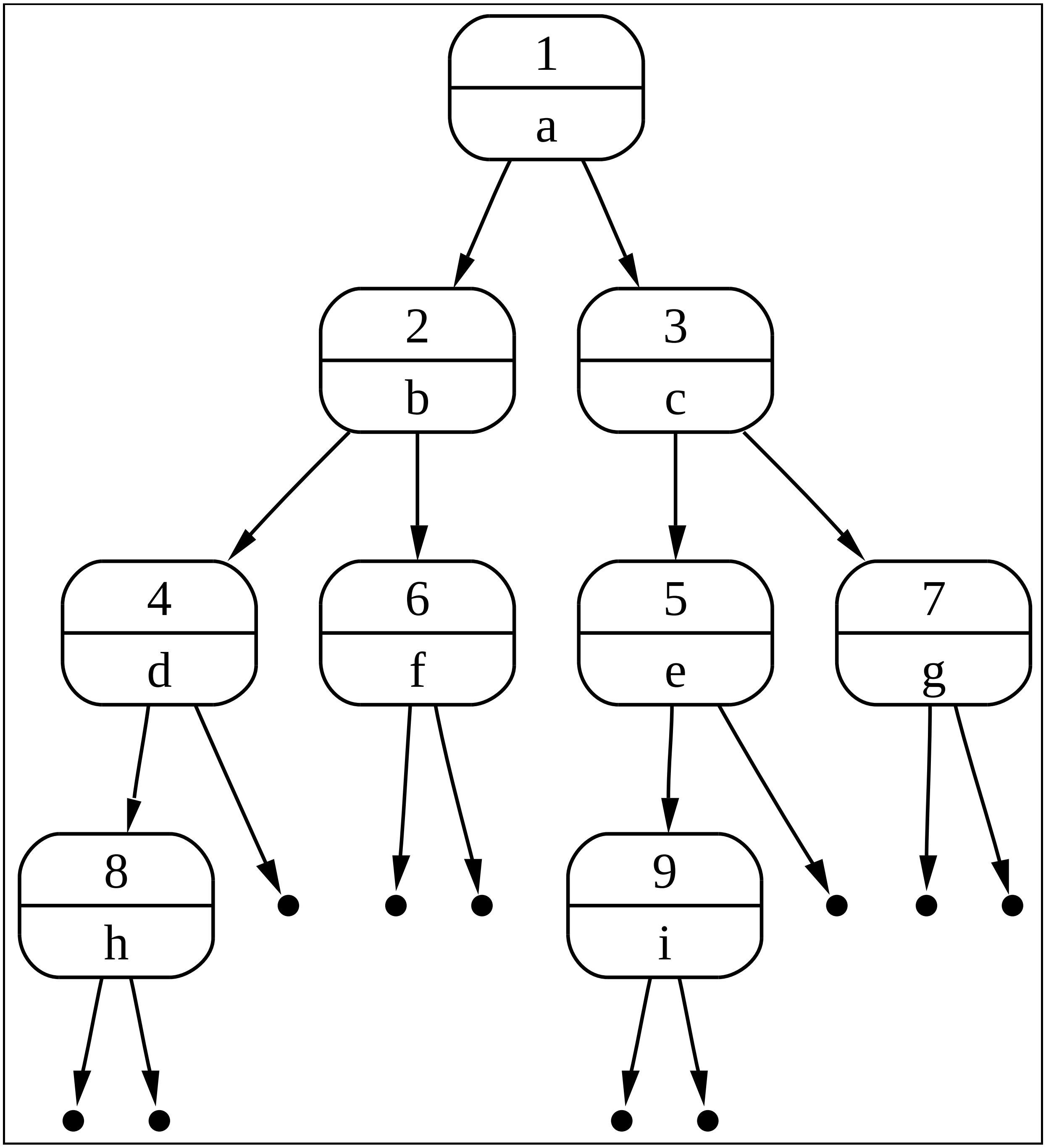
2. $\mathit{node}(k, v, l, r).\mathit{count}() = 1 + l.\mathit{count}() + r.\mathit{count}()$.

Induktive Definition von $\mathit{Heap} \subseteq \mathcal{B}$

1. $\mathit{nil} \in \mathit{Heap}$.

2. $\mathit{node}(k, v, l, r) \in \mathit{Heap}$ g.d.w. folgendes gilt:
   (a) *Heap-Bedingung*
   $k \leq l \ \wedge \ k \leq r$
   (b) *Balancierungs-Bedingung*
   $\mid l.\mathit{count}() - r.\mathit{count}() \mid \leq \ 1$
   (c) $l \in \mathit{Heap} \ \wedge \ r \in \mathit{Heap}$.

Heap — Beispiel



$\mathit{top} : \mathit{Heap} \rightarrow (\mathit{Key} \times \mathit{Value}) \cup \Omega$

1. $\mathit{nil.top}() = \Omega$
2. $\mathit{node}(k, v, l, r).\mathit{top}() = \langle k, v \rangle$

$\mathit{insert} : \mathit{Heap} \times \mathit{Key} \times \mathit{Value} \rightarrow \mathit{Heap}$

1. $\mathit{nil.insert}(k, v) = \mathit{node}(k, v, \mathit{nil}, \mathit{nil})$.
2. $k_{\text{top}} \leq k \ \wedge \ l.\mathit{count}() \leq r.\mathit{count}() \ \rightarrow$
   $\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(k, v) =$
   $\mathit{node}(k_{\text{top}}, v_{\text{top}}, l.\mathit{insert}(k, v), r)$.
3. $k_{\text{top}} \leq k \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
   $\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(k, v) =$
   $\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r.\mathit{insert}(k, v))$.
4. $k_{\text{top}} > k \ \wedge \ l.\mathit{count}() \leq r.\mathit{count}() \ \rightarrow$
   $\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(k, v) =$
   $\mathit{node}(k, v, l.\mathit{insert}(k_{\text{top}}, v_{\text{top}}), r)$.
5. $k_{\text{top}} > k \ \wedge \ l.\mathit{count}() > r.\mathit{count}() \ \rightarrow$
   $\mathit{node}(k_{\text{top}}, v_{\text{top}}, l, r).\mathit{insert}(k, v) =$
   $\mathit{node}(k, v, l, r.\mathit{insert}(k_{\text{top}}, v_{\text{top}}))$.

remove : *Heap* $\rightarrow$ *Heap*

1. $\mathit{nil}.\mathit{remove}() = \mathit{nil}$

2. $\mathit{node}(k, v, \mathit{nil}, r).\mathit{remove}() = r$

3. $\mathit{node}(k, v, l, \mathit{nil}).\mathit{remove}() = l$

4. $k_1 \leq k_2 \wedge l = \mathit{node}(k_1, v_1, l_1, r_1) \wedge r = \mathit{node}(k_2, v_2, l_2, r_2) \rightarrow$
   $\mathit{node}(k, v, l, r).\mathit{remove}() = \mathit{node}(k_1, v_1, l.\mathit{remove}(), r)$

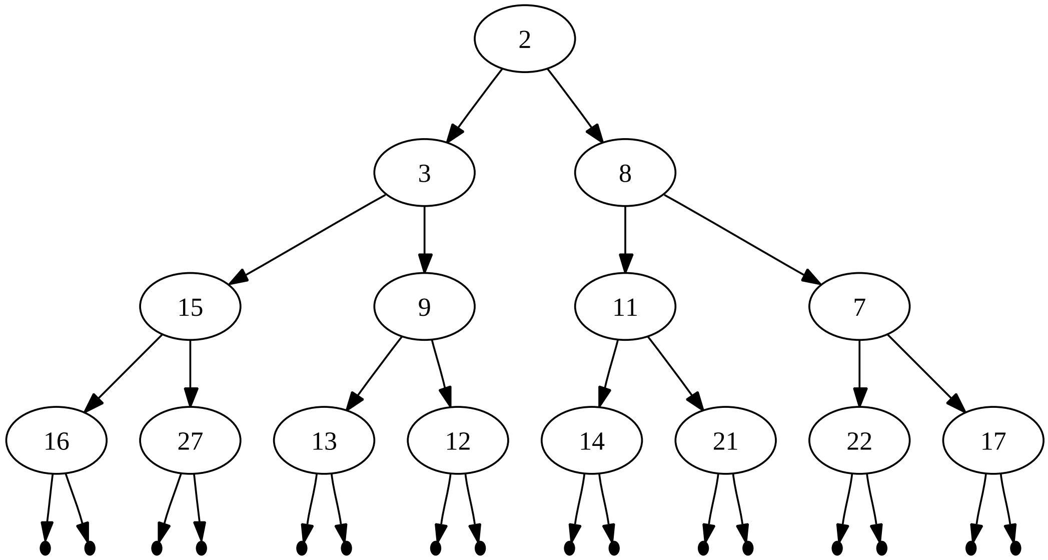
5. $k_1 > k_2 \wedge l = \mathit{node}(k_1, v_1, l_1, r_1) \wedge r = \mathit{node}(k_2, v_2, l_2, r_2) \rightarrow$
   $\mathit{node}(k, v, l, r).\mathit{remove}() = \mathit{node}(k_2, v_2, l, r.\mathit{remove}())$,

Die Methode *upheap*()

```
n.upheap() {
    k1 := n.key();
    v1 := n.value();
    p  := n.parent();
    k2 := p.key();
    v2 := p.value();
    if (k1 < k2) {
        n.key()   := k2;
        n.value() := v2;
        p.key()   := k1;
        p.value() := v1;
        nodeMap(v2) := n;
        nodeMap(v1) := p;
        p.upheap();
    }
}
```

Vollständige binäre Bäume

vollständiger binärer Baum



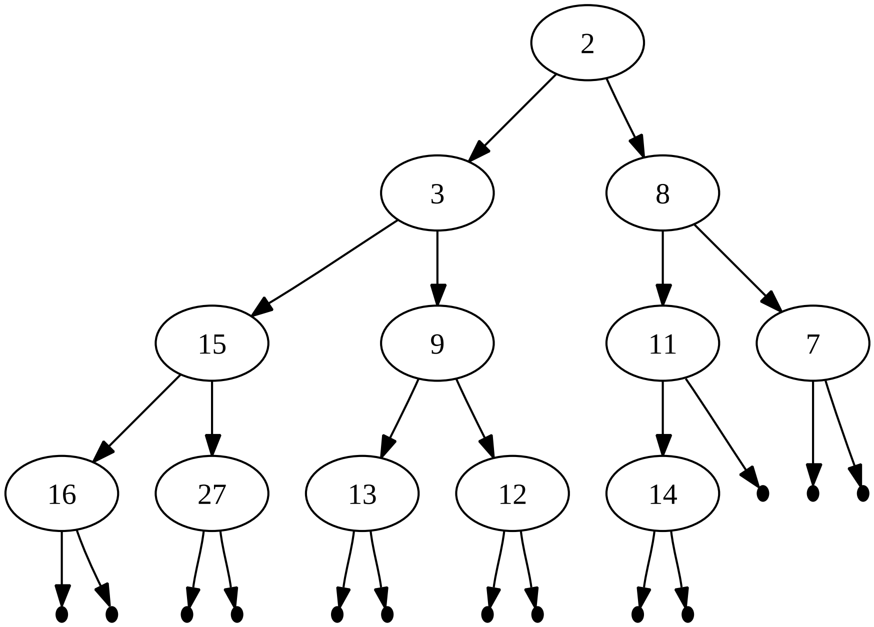
[2, 3, 8, 15, 9, 11, 7, 16, 27, 13, 12, 14, 21, 22, 17]

1. $\mathit{nil} \in \overline{\mathcal{B}}$,

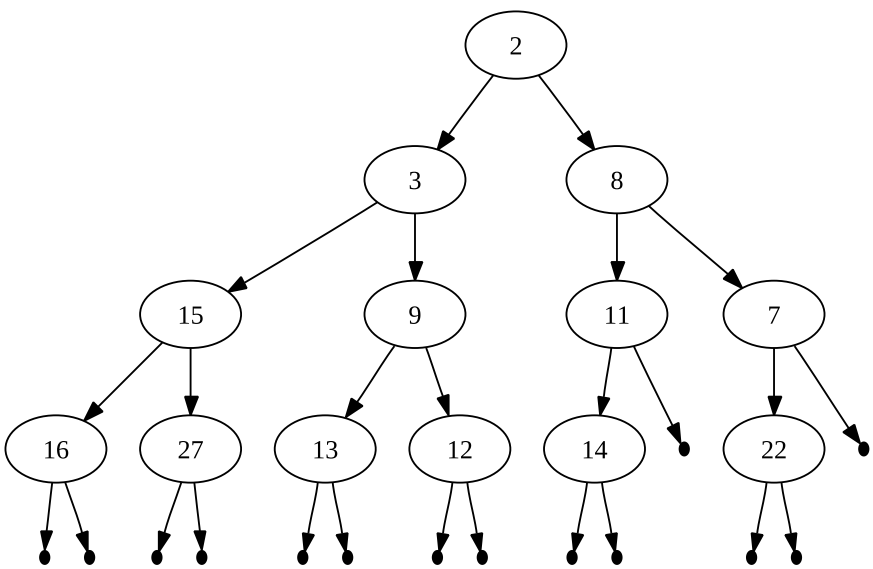
2. $\mathit{bin}(k, v, l, r) \in \overline{\mathcal{B}} \overset{\text{def}}{\longleftrightarrow}$
   $l \in \overline{\mathcal{B}} \wedge r \in \overline{\mathcal{B}} \wedge l.\mathit{height}() = r.\mathit{height}()$

Nahezu vollständige binäre Bäume

nahezu vollständiger binärer Baum



nicht nahezu vollständiger binärer Baum



Die Klasse `PriorityQueue<E>`

wichtig: `E` $=$ *Key* $\times$ *Value*

1. **`boolean offer(E`** e**`)`**

   entspricht q.*insert*(k, v).

2. **`E peek()`**

   entspricht q.*top*$()$

3. **`E poll()`**

   entspricht

   - q.*top*$()$
   - q.*remove*$()$

4. **`boolean remove(E`** e**`)`**

   q.`remove`(e) entfernt e aus q

Huffman-Algorithmus (1952)

Anzahl der Buschstaben, *count*():

1. *leaf*(c, f).*count*() $= f$

2. *node*(l, r).*count*() $=$ l.*count*() $+$ r.*count*()

Anzahl der Bits zur Kodierung eines Strings, *cost*():

1. *leaf*(c, f).*cost*() $= 0$

2. *node*(l, r).*cost*() $=$
   l.*cost*() $+$ r.*cost*() $+$ l.*count*() $+$ r.*count*()

Start: $M = \{$*leaf*$(c_1, f_1), \cdots,$ *leaf*$(c_k, f_k)\}$

Zusammenfassen der einzelnen Knoten:

```
procedure codingTree(M) {
    while (#M > 1) {
        a := minCount(M);
        M := M - { a };
        b := minCount(M);
        M := M - { b };
        M := M + { node(a, b) };
    }
    return arb M;
}
```

Berechnung kürzester Wege

Gewichteter Graph: Tripel $G = \langle \mathbb{V}, \mathbb{E}, \|\cdot\| \rangle$ mit

1. $\mathbb{V}$ ist Menge von *Knoten*.
2. $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ ist Menge von *Kanten*.
3. $\|\cdot\| : \mathbb{E} \rightarrow \mathbb{N}\backslash\{0\}$
   Funktion, die jeder Kante positive *Länge* zuordnet.

Pfad P ist Liste $P = [x_1, x_2, x_3, \cdots, x_n]$ mit

$\forall i \in \{1, \cdots, n-1\}$: $\langle x_i, x_{i+1} \rangle \in \mathbb{E}$.

$\mathbb{P}$: Menge aller Pfade in G

Pfad-Länge: $\|[x_1, x_2, \cdots, x_n]\| \;:=\; \sum\limits_{i=1}^{n-1} \|\langle x_i, x_{i+1} \rangle\|$.

Menge der Pfade, die v mit w verbinden:

$$\mathbb{P}(v, w) \;:=\; \big\{[x_1, x_2, \cdots, x_n] \in \mathbb{P} \mid x_1 = v \wedge x_n = w\big\}.$$

Shortest-Path Funktion

$$\texttt{sp} : \mathbb{V} \rightarrow \mathbb{N}$$

$$\texttt{sp}(v) \;:=\; \texttt{min}\Big\{\|p\| \mid p \in \mathbb{P}(\texttt{source}, v)\Big\}.$$

```
Rule Init
    if  dist(source)  =  Ω
    then
        dist(source) := 0;
    endif

Rule Run
    if choose ⟨u,v⟩ ∈ 𝔼 satisfying
        dist(u)  ≠  Ω and
        (dist(v) = Ω or dist(u) + ‖⟨u,v⟩‖ < dist(v))
    then
        dist(v) := dist(u) + ‖⟨u,v⟩‖;
    endif
```

Moore's Algorithmus

```
Rule Init
    if   dist(source)  =  Ω
    then dist(source) := 0;
         mode  :=  scan;
         fringe := { source };
    endif

Rule Scan
    if   mode = scan and choose u ∈ fringe
    then
        ℰ        := edges(u);
        fringe := fringe \ {u};
        mode   := relabel;
    endif

Rule Relabel
   if    mode  =  relabel
     and choose ⟨u,v⟩ ∈ ℰ satisfying
         dist(v) = Ω or dist(u) + ‖⟨u,v⟩‖ < dist(v);
   then
        dist(v)  := dist(u) + ‖⟨u,v⟩‖;
        fringe  := fringe ∪ {v};
   else
        mode    := scan;
   endif
```

```
Rule Init
    if   dist(source)  =  Ω
    then
         Fringe.insert(0, source);
         dist(source) := 0;
         Visited      := { source };
         mode         := scan;
    endif

Rule Scan
    if     mode  =  scan
       and not Fringe.isEmpty()
    then
        ⟨d, u⟩  := Fringe.top();
        Fringe.remove();
        Visited := Visited ∪  { u };
        ℰ      := edges(u);
        mode   := relabel;
    endif

Rule Relabel
    if mode  =  relabel             and
       choose ⟨u, v⟩ ∈ ℰ satisfying
          dist(v) = Ω or dist(u) + ‖⟨u, v⟩‖ < dist(v);
    then
        dist(v)  := dist(u) + ‖⟨u, v⟩‖;
        if dist(v)  =  Ω then
            Fringe := Fringe.insert(⟨dist(v), v⟩);
        else
            Fringe := Fringe.change(⟨dist(v), v⟩);
        endif
    else
        mode  := scan;
    endif
```

Die Monte-Carlo-Methode

1. Idee: Problem durch zufällige Simulation lösen

2. Anwendungen

   (a) Berechnung komplexer Volumina oder allgemein mehrfacher Integrale

   (b) Verhalten eines Systems hängt von zufälligen Einflüssen ab

   Beispiel: U-Bahn, Anzahl der Passagiere schwankt zufällig

   (c) Berechnung von Wahrscheinlichkeiten beim Glückspiel

   Gewinn-Wahrscheinlichkeit bei Poker

   (d) Historisch: Konstruktion der ersten Atom-Bombe

Beispiele in der Vorlesung

1. Berechnung von π

2. Berechnung zufälliger Permutationen

   Anwendung: Gewinn-Wahrscheinlichkeiten beim Texas Hold'em

Merkmale:

1. gut geeignet für grobe Abschätzungen

2. aufwendig für hohe Genauigkeit