Technische Universität Braunschweig

Institut für
Betriebssysteme und Rechnerverbund

Prof. Dr. H. Langendörfer · Prof. Dr. M. Zitterbart

Vordiplomsklausur Praktische Informatik

28. September 1998

Name, Vorname: ______________________

Matrikelnummer: ______________________

Fachrichtung, Fachsemester: ______________________

Zeitdauer: 4 Stunden

Zugelassene Hilfsmittel: Vorlesungs-Unterlagen, Übungen und eigene Aufzeichnungen.

Die Klausur umfaßt 10 Seiten. Bitte prüfen Sie Ihr Exemplar auf Vollständigkeit.

Maximale Punktzahl: 73. Die Klausur ist mit 37 Punkten bestanden.

Aufgabe	max. Punkte	err. Punkte
1	10	
2	12	
3	6	
4	10	
5	5	
6	10	
7	6	
8	8	
9	6	
$\sum$	73	

Note: ________

Aufgabe 1: Endliche Automaten

(5 + 5)

Def.: Sei $\delta : Z \times \Sigma \to Z$ eine beliebige Überführungsfunktion eines endlichen Automaten. Die Abbildung $\delta^* : Z \times \Sigma^* \to Z$ wird als Σ^*-*Erweiterung* von δ bezeichnet und ist durch

$$\delta^*(z, \varepsilon) = z \text{ und } \delta^*(z, w\sigma) = \delta(\delta^*(z, w), \sigma)$$

für alle $z \in Z, w \in \Sigma^*, \sigma \in \Sigma$ gegeben
(Σ^* ist die Menge aller Worte über dem Alphabet Σ).

Weiterhin gilt: $\delta^*(z, \sigma) = \delta(z, \sigma)$ für alle $z \in Z, \sigma \in \Sigma$.

(a) Zeigen Sie, daß gilt:

$$\delta^*(z, w'w) = \delta^*(\delta^*(z, w'), w)$$

für alle $z \in Z,\ w, w' \in \Sigma^*$.

Führen Sie dazu eine vollständige Induktion über die Wortlänge von w durch.

(b) **Def.:** Ein Automat heißt *vereinfacht*, wenn es zu jedem Zustand $z \in Z$ ein Wort $w \in \Sigma^*$ mit $\delta^*(z_0, w) = z$ gibt. Das Wort w wird *Zeuge* der Erreichbarkeit von z genannt.

(1) Es sei der nichtdeterministische endliche Automat $N = (Z, \Sigma, \delta, z_0, E)$ mit $Z = \{z_0, z_1, z_2, z_3, z_4, z_5\},\ \Sigma = \{a, b\},\ E = \{z_0\}$ gegeben.

δ wird durch folgende Tabelle spezifiziert:

z :	z_0	z_1	z_2	z_3	z_4	z_5
$\delta(z, a)$:	z_3, z_4	z_4	z_1	z_0	–	z_2
$\delta(z, b)$:	–	z_4	z_5	–	z_3, z_4	z_1, z_3

Geben Sie zu N den vereinfachten Automat N' sowie Zeugen für jeden erreichbaren Zustand an. Zeichnen Sie den Zustandsgraphen zu N'.

(2) Welche Sprache erkennt der Automat N ?

Aufgabe 2: Programmverifikation

(6 + 6)

(a) Gegeben sei das folgende JAVA-Programmstück. Ermitteln Sie die Nachbedingung N als funktionale Beziehung zwischen den Variablen `x` und `z`. Beweisen Sie Ihre Behauptung mit Hilfe des wp-Kalküls.

```
int x, y, z;
{ P ≡ (z > 0) }
x = 1; y = 1;
while (z != (y+1)/2)
{
   y += 2;
   x += y;
}
{ N }
```

(b) Gegeben sei der folgende Ausschnitt aus einem JAVA-Programm.

```
int p, x, y;
{ P ≡ (x = y ∧ x > 0) }
p = 1;
while (y > 1)
{
   y /= 2;
   p *= 2;
}
{ N ≡ (p ≤ x < 2 * p) }
```

Weisen Sie unter Verwendung des wp-Kalküls die totale Korrektheit des Programmausschnitts nach. Geben Sie zuerst die Schleifeninvariante Q an.

Aufgabe 3: Programmentwurf

(4 + 2)

Eine Permutationsmatrix ist eine quadratische Matrix, die in jeder Zeile exakt einmal den Wert 1 enthält, alle anderen Werte sind 0.

(a) Schreiben Sie in Java-Pseudo-Code einen Algorithmus, der mit einem Zeitaufwand von höchstens $O(n^2)$ zwei Permutationsmatrizen multipliziert.

(b) Begründen Sie den Zeitaufwand.

Hinweis: Matrixmultiplikation (Laufzeit: $O(n^3)$):

$$C = A \times B \Rightarrow c_{i,k} = \sum_{j=1}^{N} a_{i,j} b_{j,k}$$

Aufgabe 4: Laufzeitabschätzung

(3 + 5 + 2)

Gegeben ist die folgende Java-Funktion:

```
int calculate ( int n )
{
  int x, y, i, a = 1;

  for ( i = 1; i < n; i++ )
  {
    x = a;
    y = i;
    while ( y > 0 )
    {
      if ( (y % 2) != 0 )
        a += x;
      y /= 2;
      x += x;
    }
  }
  return a;
}
```

(a) Was berechnet die Funktion calculate(n) ? Begründen Sie Ihre Antwort.

(b) Wie oft wird die innere while-Schleife der Funktion calculate(n) genau durchlaufen, in Abhängigkeit von Parameter n ? Gesucht ist eine konkrete Formel für $T(n)$. Geben Sie wenn nötig Abschätzungen nach oben und unten für $T(n)$ an.

(c) Leiten Sie die Komplexitätsklasse $O()$ von calculate(n) aus $T(n)$ ab.

Hinweis: $\ln(a) + \ln(b) = \ln(ab)$
$\lfloor a+b \rfloor - 1 \leq \lfloor a \rfloor + \lfloor b \rfloor \leq \lfloor a+b \rfloor$
bzw. $\lfloor \sum_{i=1}^{n} a_i \rfloor - n + 1 \leq \sum_{i=1}^{n} \lfloor a_i \rfloor \leq \lfloor \sum_{i=1}^{n} a_i \rfloor$

Aufgabe 5: Java-Programmierung

(4 + 1)

Betrachten Sie die folgenden zwei Java-Klassen.

```
class Da
{
  int a;

  int get ()
  {
    return a;
  }
}

class DaDa extends Da
{
  int a;

  int set ()
  {
    a = super.a;
    return a;
  }

  DaDa ( int n )
  {
    int a;

    a = 1;
    this.a = a;
    while ( --n > 0 )
    {
      super.a = a + this.a;
      a = this.a;
      this.a = 2 * set() - get();
    }

    System.out.println ( a );
  }
}
```

(a) Geben Sie die Zustandsänderungen aller Variablen für den Aufruf `new DaDa(6)` an.

(b) Was berechnet `DaDa(n)` ?

Aufgabe 6: Abstrakte Datentypen

(2 + 3 + 2 + 3)

Gegeben sei die folgende Spezifikation des abstrakten Datentypes **BAUM**, der einen binären Suchbaum mit ganzzahligen Schlüsseln darstellt. Der Datentyp **BAUM** soll eine geordnete Einfachmenge repräsentieren.

Signatur:

`create:`		$\rightarrow$ **BAUM**
`node:`	**BAUM** $\times$ $\mathbb{N}\times$ **BAUM**	$\rightarrow$ **BAUM**
`add:`	**BAUM** $\times$ $\mathbb{N}$	$\rightarrow$ **BAUM**
`value:`	**BAUM**	$\rightarrow$ $\mathbb{N}$
`empty:`	**BAUM**	$\rightarrow$ `Boolean`

Axiome: $L, R \in$ **BAUM** , $x, y \in \mathbb{N}$

```
empty ( create ) := TRUE
empty ( node(L, x, R) ) := FALSE
add ( create, x ) := node ( create, x, create )
add ( node(L, x, R), y) := IF ( x > y ) THEN node ( add(L, y), x, R )
                                        ELSE node ( L, x, add(R, y) )
value ( node(L, x, R) ) := x
```

(a) Wird durch das Axiomensystem der Aufbau eines binären Suchbaums für eine geordnete Einfachmenge korrekt spezifiziert? Begründen Sie Ihre Antwort und korrigieren Sie gegebenenfalls das Axiomensystem.

(b) Leiten Sie den folgenden Ausdruck entsprechend den Axiomen ab und zeichnen Sie den dazugehörigen Baum.

`add ( add ( add ( create, 5 ), 10 ), 20 )`

(c) Es seien für den abstrakten Datentyp **BAUM** die zusätzlichen Operationen `OP1` und `OP2` definiert:

`OP1`: **BAUM** $\rightarrow$ $\mathbb{N}$
`OP2`: **BAUM** $\times$ $\mathbb{N}\rightarrow$ `Boolean`

```
OP1(node(L, x, R)) := IF (empty(R)) THEN x
                                    ELSE OP1(R)

OP2(node(L, x, R), y) :=
  IF (empty(L)) THEN IF (y > x) THEN TRUE
                                ELSE FALSE
                ELSE IF (y > value(L)) THEN TRUE
                                       ELSE OP2(L)
```

Beschreiben Sie in einem Satz, welche Funktionen von den Operationen `OP1` und `OP2` ausgeführt werden.

(d) Konstruieren Sie einen Operator, der in einem Baum ein Element löscht.
`OP3`: **BAUM** $\times$ $\mathbb{N}\rightarrow$ **BAUM**
Der Operator `OP3` soll nur die Kompenente mit dem angegebenen Schlüssel löschen, jedoch nicht die zugehörigen Unterbäume entfernen.

Aufgabe 7: Textsuche

(2 + 1 + 3)

Das Verfahren von Knuth-Morris-Pratt ist ein Suchalgorithmus, mit dem in einer Zeichenkette nach dem Vorkommen eines Musters gesucht werden kann. Suchen Sie nach dem Muster P=**anano** im Text T=**anbanananobano**.

(a) Beschreiben Sie kurz die Grundidee und das Verfahren des Suchalgorithmus.

(b) Bestimmen Sie die Verschiebetafel des Musters P.

(c) Führen Sie den Algorithmus von Knuth-Morris-Pratt aus und bestimmen Sie das/die Vorkommen des Musters P im Text T. Verdeutlichen Sie das Verfahren indem Sie die wesentlichen Zwischenschritte der Textsuche darstellen.

Aufgabe 8: Datenstrukturen

(3 + 1 + 1 + 3)

(a) Zeichnen Sie den AVL-Baum, der entsteht, wenn die Schlüssel { 20, 10, 5, 30, 40, 15, 12, 13, 11, 1 } der Reihe nach in den (anfangs leeren) Baum eingefügt werden. Geben Sie dabei alle entstehenden (Teil-)Bäume an.

(b) Welchen Einfluß hätte das Einfügen der Schlüssel in (aufsteigend) sortierter Reihenfolge auf die mittlere Anzahl von Vergleichen zum Finden eines bestimmten Schlüssels (im Vergleich zu dem unter (a) entstandenen Baum)? Begründen Sie Ihre Antwort.

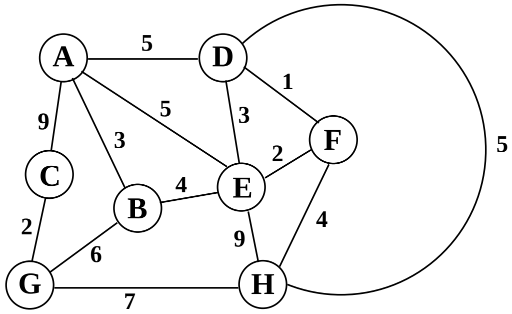
(c) Welchen wesentlichen Vorteil bieten Rot-Schwarz-Bäume in der Praxis gegenüber AVL-Bäumen?

(d) Gegeben sei eine Hashtabelle aus $n = 20$ Plätzen mit der Hashfunktion $g(i) = k * i \; mod \; n; \; i, k \in \mathbb{N}$. i stellt den Schlüssel der Hashfunktion dar und kann Werte von 1 bis 20 annehmen. Als Kollisionsstrategie wird quadratisches Sondieren verwendet.
Für welche $k \quad (1 \leq k \leq n)$ werden alle Plätze in der Hashtabelle erreicht?

Aufgabe 9: Spannbäume

(2 + 3 + 1)

Gegegen ist der folgende Graph G mit gewichteten Kanten.



(a) Geben Sie mit dem in der Vorlesung verwendeten Verfahren den minimalen Spannbaum zu G an. Geben Sie dabei an, in welcher Reihenfolge die Kanten und Knoten in den minimalen Spannbaum aufgenommen werden.

(b) Geben Sie den Aufwand O zu dem Algorithmus an, wobei ausschließlich zusammenhängende Graphen zu betrachten sind. Begründen Sie Ihre Aussage.

(c) Ist der minimale Spannbaum zu einem beliebigen Graphen eindeutig? Begründen Sie Ihre Antwort.

Technische Universität Braunschweig

Institut für
Betriebssysteme und Rechnerverbund

Prof. Dr. H. Langendörfer · Prof. Dr. M. Zitterbart

Musterlösung

Vordiplomsklausur Praktische Informatik

28. September 1998

Lösung 1. Endliche Automaten

(a)
- Induktionsanfang: $n = |w| = 0$ bzw. $w = \varepsilon$

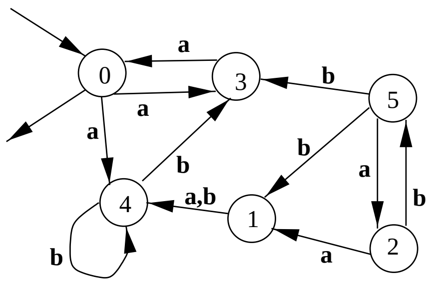
  z.z.: $\delta^*(z, w') = \delta^*(\delta^*(z, w'), \varepsilon)$
  folgt unmittelbar aus der Def. der Σ^*-Umgebung

- Induktionsschritt:
  I.-Voraussetzung: $\delta^*(z, w'\sigma_1 \ldots \sigma_n) = \delta^*(\delta^*(z, w'), \sigma_1 \ldots \sigma_n)$

  z.z.: $\delta^*(z, w'\sigma_1 \ldots \sigma_{n+1}) = \delta^*(\delta^*(z, w'), \sigma_1 \ldots \sigma_{n+1})$

  $\delta^*(z, w'\sigma_1 \ldots \sigma_n\sigma_{n+1}) = \delta(\delta^*(z, w'\sigma_1 \ldots \sigma_n), \sigma_{n+1})$
  $= \delta(\delta^*(\delta^*(z, w'), \sigma_1 \ldots \sigma_n), \sigma_{n+1})$
  $= \delta^*(\delta^*(z, w'), \sigma_1 \ldots \sigma_{n+1})_\square$

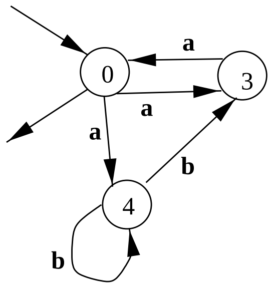
(b) Graph zu N:



$N' : Z = \{z_0, z_3, z_4\}$ mit:

z :	z_0	z_3	z_4
$\delta(z,a)$:	z_3, z_4	z_0	–
$\delta(z,b)$:	–	–	z_3, z_4

Graph zu N':



Zeuge für $0 : \varepsilon$; 3 : a ; 4 : a
Sprache $L = \{a\ b^*\ a\}^*$

Lösung 2. Programmverifikation

(a) $N \equiv x = z^2$
$Q \equiv (x = (\frac{y+1}{2})^2)$

(1) $P \Rightarrow wP(„x = 1, y = 1“, Q)$
$wP(„x = 1, y = 1“, x = (\frac{y+1}{2})^2)$
$\equiv wP(„x = 1“, x = (\frac{2}{2})^2)$
$\equiv 1 = 1^2 \equiv TRUE_\square$

(2) $Q \wedge (z\ != (\frac{y+1}{2})) \Rightarrow wP(„y = y + 2, x = x + y“, Q)$
$wP(„y = y + 2, x = x + y“, x = (\frac{y+1}{2})^2)$
$\equiv wP(„y = y + 2“, x + y = (\frac{y+1}{2})^2)$
$\equiv x + y + 2 = (\frac{y+3}{2})^2$
$\equiv (\frac{y+1}{2})^2 + y + 2 = (\frac{y+3}{2})^2$
$\equiv \frac{(y+1)^2}{4} + y + 2 = \frac{(y+3)^2}{4}$
$\equiv y^2 + 2y + 1 + 4y + 8 = y^2 + 6y + 9$
$\equiv TRUE\ _\square$

(3) $Q \wedge (z = \frac{y+1}{2}) \Rightarrow x = z^2$
$\equiv (x = (\frac{y+1}{2})^2) \wedge (z = \frac{y+1}{2}) \Rightarrow x = z^2$
$\equiv TRUE\ _\square$

(b) Invariante $Q \equiv (y * p \le x < (y + 1) * p)$

(1) $P \Rightarrow wP(„p = 1“, Q)$
$wP(„p = 1“, y * p \le x < (y + 1) * p)$
$\equiv y \le x < y + 1$
folgt aus $P_\square$

(2) $Q \wedge (y > 1) \Rightarrow wP(„y = y/2, p = p * 2“, Q)$
$wP(„y = y/2, p = p * 2“, y * p \leq x < (y+1) * p)$
$\equiv wP(„y = y/2“, y * 2p \leq x < (y+1) * 2p)$
$\equiv \frac{y}{2} * 2p \leq x < (\frac{y}{2} + 1) * 2p$

Fall 1: y gerade
$\equiv y * p \leq x < (y+2) * p$
folgt aus Q (da Q stärker)
Fall 2: y ungerade
$\equiv (y-1) * p \leq x < (y-1) * p + 2 * p$
$\equiv (y-1) * p \leq x < (y+1) * p$
folgt aus Q (da Q stärker) □

(3) $Q \wedge (y = 1) \Rightarrow N$
$\equiv y * p \leq x < (y+1) * p \wedge y = 1$
$\equiv p \leq x < 2 * p \equiv N$ □

(4) Terminiertheit: Bei $y \leq 1$ wird Schleife nicht durchlaufen;
sonst: y wird pro Durchlauf stets verkleinert (halbiert)
und ist somit irgendwann sicher ≤ 1

Lösung 3. Programmentwurf

(a) Matrixmultiplikation: Jedes Element der i-ten Zeile von Matrix A wird mit dem korrespondierenden Element der k-ten Spalte von Matrix B multipliziert. Die Summe wird in Matrix C an Position Zeile i, Spalte k eingetragen.

Da bei Permutationsmatrizen in einer Zeile nur jeweils eine '1' ist, identifiziert die Position (a_s) der '1' in einer Zeile von Matrix A ebenfalls die korrespondierende Zeile in der Matrix B. Es kann in der Lösungsmatrix C nur an der Position einer Zeile eine '1' vorhaden sein, wo auch in Matrix B in der Zeile a_s eine '1' vorhanden ist.

Vorgehen:
1. initialisiere Lösungsmatrix C mit '0'
2. wiederhole für alle Zeilen i = [0...n-1] von Matrix A:
2a. a_s = ermittle Position von '1' in Zeile i in Matrix A,
2b. b_s = finde korrespondierende '1' in Matrix B in Zeile a_s
2c. setze Eintrag an Stelle (Zeile i, Spalte b_s) in Matrix C auf '1'.

```
int i, j, a_s, b_s;

for ( i = 0; i < n; i++ )
  for ( j = 0; j < n; j++ )
    C[i][j] = 0;

for ( i = 0; i < n; i++ )
{
  for ( a_s = 0; A[i][a_s] != 0; a_s++ );
  for ( b_s = 0; B[a_s][b_s] != 0; b_s++ );
  C[i][b_s] = 1;
}
```

(b) Zeitbedarf: Initialisierung der Matrix hat Laufzeit $O(n^2)$.
Die äußere Schleife wird n-mal durchlaufen, also $O(n)$.

Die beiden inneren Schleifen können im schlechtesten Fall auch jeweils n-mal durchlaufen werden (quadratische Matrix), jede innere Schleife hat die Laufzeit $O(n)$.

Daraus folgt die Gesamtlaufzeit:

$$O(n^2) + O(n) * (O(n) + O(n)) = O(n^2) + O(n) * O(n) = O(n^2) + O(n^2) = O(n^2) \ \square$$

Lösung 4. Laufzeitabschätzung

(a) `calculate(n)` berechnet die Fakultät $n!$.

Grundidee: $(n+1)! = n! * (n+1) = n! * n + n!$

Die innere while-Schleife ist die ägyptische Multiplikation und berechnet $x * y$. Zu beachten ist, daß das Multiplikationsergebnis zur Variablen a hinzuaddiert wird.

In der äußeren Schleife wird y auf die Werte von [1...n-1] gesetzt. x wird auf das Ergebnis der vorherigen Multiplikation bzw. '1' am Anfang gesetzt, x enthält also das Ergebnis aller vorhergehenden Multiplikationen (das entspricht $i!$). x wird mit $y = i$ multipliziert und am Ende der Schleife ist $a = (i+1)!$.

(b) Vorgehensweise: 1. $T_a(n)$ der ägyptischen Multiplikation bestimmen, 2. gesamte Laufzeit $T(n)$ ermitteln.

(1) Aufwand der ägyptischen Multiplikation $T_a(n)$: Anzahl der Durchläufe hängt nur von y ab. y wird bei jedem Durchlauf halbiert. Zu beachten ist, daß y ganzzahlig ist.

y =	1	2	3	4	5	6	7	8	9
Anzahl Durchläufe	1	2	2	3	3	3	3	4	4

Hieraus folgt anschaulich:

$$T_a(n) = \lfloor \log_2 n \rfloor + 1 \ , (\lceil \log_2 n \rceil ist falsch)$$

(2) Gesamtaufwand T(n): Die äußere Schleife wird (n-1)-mal durchlaufen. Es wird also die ägyptische Multiplikation (n-1)-mal mit den Werten für y von [1...n-1] berechnet.

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} T_a(i) \\ &= \sum_{i=1}^{n-1} (\lfloor \log_2 i \rfloor + 1) \\ &= n - 1 + \sum_{i=1}^{n-1} \lfloor \log_2 i \rfloor \end{aligned}$$

Dieser Term kann nicht weiter vereinfacht werden, man kann ihn aber abschätzen:

$$n - 1 + \lfloor \sum_{i=1}^{n-1} \log_2 i \rfloor - (n-1) + 1 \leq T(n) \leq n - 1 + \lfloor \sum_{i=1}^{n-1} \log_2 i \rfloor$$

$$\lfloor \log_2 (n-1)! \rfloor + 1 \leq T(n) \leq n - 1 + \lfloor \log_2 (n-1)! \rfloor$$

(c) Komplexitätsklasse: aus (b) erhält man eine Abschätzung nach oben:

$$T(n) \leq n - 1 + \lfloor \log_2 (n-1)! \rfloor$$

$$\Rightarrow O(n) = n + \ln(n-1)!$$

es gilt

$$\ln(n!) = \sum_{i=1}^{n} \ln(i) \leq \sum_{i=1}^{n} \ln(n) = n * \ln(n)$$

$$\Rightarrow O(n) = n + (n-1) * \ln(n-1)$$

$$\Rightarrow O(n) = n * \ln n$$

Lösung 5. Java-Programmierung

(a) Der Aufruf von `new DaDa(5)` ergibt folgende Zustandstabelle (es wird immer der Zustand aller Variablen am Anfang der while-Schleife angegeben):

Variablen	n	a/Konstruktor	DaDa.a/this.a	Da.a/super.a
	5	1	1	0
	4	1	2	2
	3	2	3	3
	2	3	5	5
	1	5	8	8

(b) `DaDa(n)` berechnet die n-te Fibonacci-Zahl F_n.

Lösung 6. Abstrakte Datentypen

(a) Die Vorbedingung ist, daß der Baum eine sortierte Einfachmenge darstellt. Das bedeutet, es darf kein Element zweimal im Baum vorkommen.

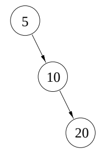
Beim zweiten Axiom für **add** werden die Schlüssel nicht auf Gleichheit überprüft. Das hat zur Folge, daß ein Schlüssel mehrfach in den Suchbaum eingefügt werden kann. Die Gleichheit muß beim Einfügen mit berücksichtigt werden. Dies kann z.B. wie folgt geschehen:

```
add ( node(L, x, R), y) :=
               IF ( x == y ) THEN node ( L, x, R )
                             ELSE IF ( x > y ) THEN node ( add(L, y), x, R
)
                                                ELSE node ( L, x, add(R, y) )
```

(b)
```
add ( add ( add ( create, 5 ), 10 ), 20 ) =
add ( add ( node ( create, 5, create ), 10 ), 20 )
add ( node ( create, 5, add ( create, 10 ) ), 20 )
add ( node ( create, 5, node ( create, 10, create ) ), 20 )
node ( create, 5, add ( node ( create, 10, create ), 20 ) )
node ( create, 5, node ( create, 10, add ( create, 20 ) ) )
node ( create, 5, node ( create, 10, node ( create, 20, create ) ) )
```

(c) `OP1` liefert den größten im Baum vorhandenen Schlüssel.

`OP2` prüft, ob ein Schlüssel im Baum vorhanden ist, der echt kleiner ist als y.



(d)
```
OP3(create, y) := create
OP3(node(L, x, R), y) :=
  if (x == y) THEN IF empty(L) THEN R
                             ELSE IF empty(R) THEN L
                                              ELSE node (OP3(L, OP1(L)), OP1(L), R)
             ELSE IF (x < y) THEN node(L, x, OP3(R,y))
                              ELSE node(OP3(L,y), x, R)
```

Lösung 7. Textsuche

Suche des Musters P="anano" im Text T="anbanananobano".

(a) Gegeben ist ein (kurzes) Muster und ein (langer) Text. Zu suchen ist das Vorkommen des Musters im Text.

Die Idee bei KMP ist, den bisherigen Vergleich zwischen Muster und Text auszunutzen, um möglicherweise das Muster um mehr als eine Stelle im Text zu verschieben.

Die Anzahl Stellen, um die das Muster weiter nach rechts verschoben werden kann, hängt von der Ähnlichkeit des Musters mit sich selbst ab.

Das Muster wird dazu auf Übereinstimmung mit sich selbst geprüft. Hieraus ergibt sich die Verschiebetafel für das Muster.

(b) Verschiebetafel:

i	1	2	3	4	5
P[i]	a	n	a	n	o
next[i]	0	0	1	2	0

(c) Jede Zeile repräsentiert eine Iteration der äußeren Schleife, jedes Zeichen in der Zeile zeigt das Ergebnis des Vergleichs ('X' für Ungleichheit).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
T:	a	n	b	a	n	a	n	a	n	o	b	a	n	o
i=0:	a	n	X											
i=2:			X											
i=3:				a	n	a	n	X						
i=5:						a	n	a	n	o				
i=10:											X			
i=11:												a	n	X

Lösung 8. Datenstrukturen

(a) s. Zeichnung S. 8

(b) Es hätte keinen Einfluß, da die Höhe eines AVL-Baumes (welche für die Anzahl von Vergleichen maßgeblich ist) ausschließlich von der Anzahl der Knoten abhängt, max. $O(ln\ n)$.

(c) Bei R-S-Bäumen ist im Gegensatz zu AVL-Bäumen die Anzahl der Rotationen nach dem Löschen von Knoten unabhängig von der Höhe des Baumes und damit von der Anzahl der Knoten. (Bei R-S-Bäumen max. 3 Rotationen.)

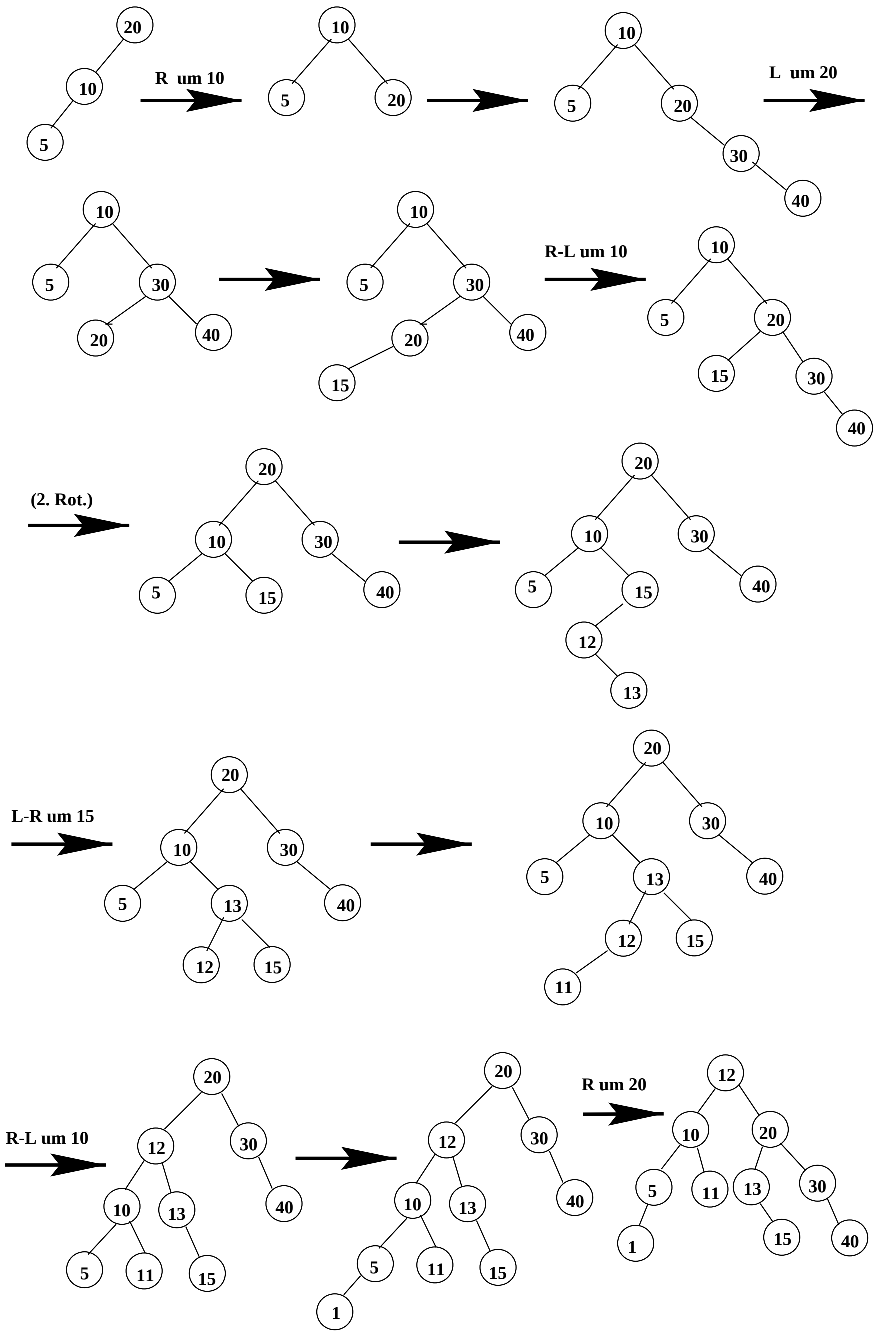
(d) Für alle k, die zu n teilerfremd sind, treten keine Kollisionen auf. Als da wären $\{3, 7, 9, 11, 13, 17, 19\}$

Für alle anderen Werte für k müssen alle Plätze von $k+1$ bis $2k-1$ durch quadratisches Sondieren erreicht werden können, damit die Tabelle vollständig gefüllt wird.

Es gilt für beliebige Werte von a: $a^2\ mod\ 20 = \{0, 1, 4, 5, 9, 16\}$.

Folglich führt quadratisches Sondieren für $k = \{2, 6, 14, 18\}$ zur vollständigen Belegung der Tabelle.

Insgesamt ergibt sich somit für $k:\ \{1, 2, 3, 6, 7, 9, 11, 13, 14, 17, 18, 19\}$

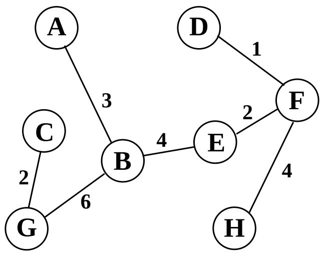


Lösung 9. Spannbäume

(a)

Gewicht	Knoten
1:	$\{D - F\}$
2:	$\{C - G, E - F\}$
3:	$\{A - B\}$
4:	$\{B - E, H - F\}$
5:	—
6:	$\{B - G\}$
7:	—
9:	—

Minimaler Spannbaum zu G:



(b) **Kruskal:**

$E' = 0$
Sortiere Kantenmenge E nach Länge (aufsteigend)
Für alle $v \in V$
    Erzeuge Baum aus v
Sortiere Bäume nach Knoten
Für alle $e \in E$ do
    if find$(e.v_1) \neq$ find$(e.v_2)$ then
        union (v_1, v_2)
        $E' = E' \cup (v_1, v_2)$
    end
end

Kanten sortieren: $O(|E|\ log|E|)$; Baum erzeugen: $O(|V|)$; Bäume sortieren: $O(|V|\ log|V|)$; find: $O(log|V|)$; union: $O(log|V|)$; Vereinigung zu E': $O(1)$.

Da man aus der Anzahl der Kanten die Zahl der Knoten nach oben abschätzen kann (max. doppelt so viele Knoten wie Kanten in einem zshg. Graphen), ergibt sich eine Gesamtkomplexität von

$O(|E|\ log|E|)$.

(c) Nein (Gegenbeispiel: Dreieck mit zwei gleichgewichteten Kanten).