Klausuraufgaben

Algorithmen und Datenstrukturen

29. Januar 2013

Zur Lösung der nachfolgenden Aufgaben haben Sie **90 Minuten** Zeit inklusive Abgabe, nach Ausgabe der Aufgabenblätter. Es sind keine Hilfsmittel zugelassen! Tipp's finden Sie in der Klausur am Ende der jeweiligen Aufgabe, d.h. nach einem Tipp kann es noch weitere Aufgaben geben.

- Für alle Aufgaben ist der **Lösungsweg** und die Lösung **nachvollziehbar** anzugeben, d.h. Sie müssen mich von Ihrer Lösung mittels des Lösungsweges überzeugen! Antworten ohne Begründung, d.h. ohne Lösungsweg, oder zu schlechte Handschrift, d.h. für mich nicht lesbarer Schrift, werden nicht bewertet!
- Bitte schreiben Sie zuerst **Ihren Namen und Ihre Matrikelnummer** zumindest auf das erste Blatt und auf alle weiteren (losen) Blätter zumindest die Matrikelnummer!

Die Aufgaben sind nicht nach dem Schwierigkeitsgrad geordnet!

Aufgabe	1	2	3	4	5	$\sum$
	Grundlagen	Komplexität	Sortieren	Bäume	Hashing	Summe
mögliche Punkte	16	15	37	22	17	107
erreichte Punkte						

Erreichte Bewertung:

Aufgabe 1 (Grundlagen) *(16P)*

1. **Algorithmus** *(5 Punkte)*

   (a) *Beschreiben Sie den Begriff Algorithmus und erklären Sie die vier allgemeinen Anforderungen an einen solchen Algorithmus.*

   (b) *Erklären Sie folgende vier Eigenschaften eines Algorithmus:* terminiert, determiniert, deterministisch und nicht deterministisch.

2. **Rekursion** *(11 Punkte)*
   Zum deterministischen Testen von Anwendungen lassen sich sogenannte Pseudo-Zufallszahlen verwenden. Folgende Berechnungsformel erzeugt für $0 \leq n \in \mathbb{N}^0$ *solche Zahlen:*

$$f(n) = \begin{cases} n+1 & \textit{falls } 3 > n \\ 1 + (((f(n-1) - f(n-3)) * f(n-2)) \ \textit{mod}\ 100) & \textit{falls } 3 \leq n \end{cases}$$

   (a) *Implementieren Sie diese Funktion rekursiv ohne Optimierungsüberlegungen, also direkt, in Java in folgendem Rahmen:*
   `public static int f(int n) { }`

   (b) *Implementieren Sie die Funktion f linear-rekursiv (rekursiv mit iterativem Ablauf) mit Durchreichen von Zwischenergebnissen mittels der Funktion linrek in Java in folgendem Rahmen:*

```
public static int f(int n) {
    return linrek(1, 2, 3, n-3);
}
public static int linrek(int a, int b, int c, int steps) {
}
```

   (c) *Implementieren Sie die Funktion f iterativ (ohne Rekursion) in Java in folgendem Rahmen:*
   `public static int f(int n) { }`

   (d) *Welche der drei Lösungen ist bereits linear in der Laufzeitkomplexität und wie kann man die Funktion f in den noch nicht linear in der Laufzeitkomplexität implementierten Varianten realisieren, so dass sie im Zeitaufwand linear werden, also die Komplexität* $O(n)$ *besitzten ? Transformieren Sie dazu ggf. Laufzeitaufwand der (jeweiligen) naiven Lösungen in Speicherplatzaufwand, jedoch mit maximal* $O(n)$ *Speicherplatzaufwand! Skizieren Sie dazu (jeweils) ein Java-Programm und begründen kurz, warum der Aufwand (jeweils) (Laufzeit-/Speicherplatz) linear sein sollte.*

Aufgabe 2 (Komplexitätstheorie) *(15P)*

1. **Aufwandabschätzung** *(9 Punkte)*
   Gegeben seien folgende Codefragmente; n sei eine positive natürliche Zahl.

```
algorithm alg1(n)
var prod;
begin prod := 1;
      for i := 1 to n do
          prod := prod*i;
      endfor
      return prod;
end alg1;
```

```
algorithm alg2(n)
var p;
begin p := 1;
      while n > 1 do
         p := p*n;
         n := n/3;
      endwhile
      return p;
end alg2;
```

```
algorithm alg3(n)
var p,r;
begin p := 1; r:= 0;
      for i := 1 to n do
          for j := 1 to p do
              r := r + 1;
          endfor
          p := p*2;
      endfor
      return r;
end alg3;
```

```
algorithm alg4(n)
var p,t;
begin p := 1;
      for i := 0 to n do
          for j := i to n do
              t := n;
              while t > 0 do
                 p := p + 1;
                 t := t/2;
              endwhile
          endfor
      endfor
      return p;
end alg4;
```

   Geben Sie für diese Algorithmen in Abhängigkeit von n eine Laufzeitkomplexität an. Begründen Sie kurz Ihre Entscheidung. Geben Sie anschließend bzgl. der Komplexität eine Rangordnung der Algorithmen an.

2. **Problembehandlung** *(6 Punkte)*
   In der Vorlesung wurden sechs Möglichkeiten genannt, wie man auf ein komplexes Problem reagieren kann bzw. das Problem handhabbar machen kann. Nennen und erklären Sie diese evtl. auch mit Hilfe eines Beispieles.

Tipp*:*

$$\sum_{i=1}^{n} i = \frac{n * (n+1)}{2}$$

Aufgabe 3 (Sortieren) *(37P)*

1. **CornSort** *(14 Punkte)*
   Gegeben sei folgender Algorithmus:

```
algorithm CornSort(A : array of sortable items)
  swapped := true;
  while swapped == true do
    swapped := false;
    for i := 0 to length(A) - 2 do
      if A[i] > A[i+1] then swap(A[i],A[i+1]);
                            swapped := true;
      endif endfor
    if swapped == false then exit; endif
    swapped := false;
    for i := length(A) - 2 downto 0 do
      if A[i] > A[i+1] then swap(A[i],A[i+1]);
                            swapped := true;
      endif endfor
  endwhile
end CornSort;
```

   Die Funktion `length(A)` *liefert die Größe des Arrays* `A`*. Arrays werden von 0 bis length(A)-1 indiziert. Die Methode* `swap(X,Y)` *vertauscht die Inhalte der Variablen* `X` *und* `Y`*. Das Schlüsselwort* `exit` *bewirkt, dass die Bearbeitung des Algorithmus unmittelbar beendet wird.*

   (a) *Wie funktioniert* `CornSort` *? Beschreiben Sie die Funktionsweise!*
   Hinweis: Damit ist nicht eine rein wörtliche Übersetzung in natürliche Sprache gemeint!

   (b) *Geben Sie für* `CornSort` *in Abhängigkeit von* `n = length(A)` *eine Laufzeitkomplexität an. Es reicht, die Anzahl der Vergleiche zu zählen. Begründen Sie kurz Ihre Entscheidung. Ist* `CornSort` *ein optimales Sortierverfahren (basierend auf Schlüsselvergleichen) ?*

   (c) *Betrachten und beschreiben Sie die Behandlung des kleinsten und größten Elements der Eingabe unter* `CornSort`*.*

   (d) *Geben Sie Vorschläge, wie Sie die Laufzeit von* `CornSort` *verbessern können.*

   (e) *Hat sich durch Ihre Optimierung die Laufzeitkomplexität von* `CornSort` *verbessern können ? Begründen Sie kurz Ihre Entscheidung.*

2. **Teile und Herrsche** *(11 Punkte)*
Manchmal ist es notwendig, den minimalen oder den maximalen Wert einer Folge zu finden. Eine Verallgemeinerung hiervon ist es, das k-kleinste Element einer Folge zu finden. Eine Möglichkeit, dies zu erreichen ist es, die Folge zunächst zu sortieren und das k-te Element auszuwählen. Der dazu notwendige Aufwand beträgt $O(nlog(n))$ wobei n die Folgenlänge darstellt. Es ist jedoch nicht notwendig, die gesamte Folge zu sortieren, um das gesuchte Element zu finden!

   (a) *Entwickeln Sie einen „Teile und Herrsche"-Algorithmus, der das k-kleinste Element einer gegebenen Folge (mit $n > k$) berechnet. Ihr Algorithmus sollte im durchschnittlichen Fall mit $O(n)$ Zeit auskommen, also im durchschnittlichen Fall linear sein. Sie können davon ausgehen, dass alle Werte in der Eingabefolge paarweise verschieden sind.*

   (b) *Begründen Sie, warum Ihr Algorithmus korrekt arbeitet.*

   (c) *Nehmen Sie einen Ansatz vor, den Aufwand (Charakter) Ihres Algorithmus zu bestimmen.*

3. **Heap Increase** *(6 Punkte)*
Schreiben Sie eine Funktion mit der Signatur `heap-increase-key(A,i,k)`*. Die Funktion soll in einen bestehenden Heap den Schlüssel an der Stelle i des Arrays A durch den neuen größeren Schlüssel k ersetzen. Die Laufzeitkomplexität soll $O(log(n))$ betragen. Begründen Sie, warum Ihr Algorithmus die geforderte Laufzeitkomplexität einhält.*

4. **Mehr-Wege-Mergesort** *(6 Punkte)*
Sortieren Sie die unten angegebene Folge mittels Mehr-Wege-Mergesort für $k = 3$ und einer initialen Run-Länge von 1:

$$T_0 : 43, 71, 12, 35, 11, 80, 99, 47, 62, 15, 77, 26, 69, 93, 19, 45, 23, 38, 56, 91, 10$$

Tipp:

- **Algorithmus Merge Sort**

```
mergeSort(int left, int right) {
 if (left < right) {
    int mid = (right + left)/2;
    mergeSort(left, mid);
    mergeSort((mid+1), right);
    merge(left, mid, right); } }

merge(int left, int mid, int right) {
   Datensatz tmp[(right-left)+1]; int tmp_i = 0;
   int i = left; int j = mid+1;

   while ((i <= mid) && (j <= right)) {
      if (a[i].key < a[j].key) { tmp[tmp_i] = a[i]; i++;
                        } else { tmp[tmp_i] = a[j]; j++; };
      tmp_i++; }

   while (i <= mid) { tmp[tmp_i] = a[i]; tmp_i++; i++; };
   while (j <= right) { tmp[tmp_i] = a[j]; tmp_i++; j++; };

   tmp_i = 0;
   for (i=left; i <= right; i++) {
       a[i] = tmp[tmp_i]; tmp_i++; } }
```

- **Algorithmus Heap Sort**
  Löschen eines Knoten's (Top-Down):

  1. *Schreibe das Element mit dem höchste Index an die Position der Wurzel*
  2. *Vertausche dieses Element so lange mit dem größeren seiner Söhne, bis seine beiden Söhne kleiner sind, oder bis es keine Söhne mehr hat.*

  Erstellung eines Heap's (Bottom-Up): Gegeben sei die Folge von Schlüsseln: $k_1, \ldots, k_N$.

  1. *In der Reihenfolge* $k_{\lfloor \frac{N}{2} \rfloor}, \ldots, k_1$ *führe für diese Schlüssel durch:*
     (a) *Vertausche dieses Element so lange mit dem größeren seiner Söhne, bis seine beiden Söhne kleiner sind, oder bis es keine Söhne mehr hat.*

Aufgabe 4 (Bäume) *(22P)*

1. **Allgemeiner Baum** *(12 Punkte)*
   Ein allgemeiner Baum kann pro Knoten beliebieg viele Söhne haben und kann auch als gerichteter Graph ohne Zyklen aufgefasst werden. Gesucht ist nun ein solcher allgemeiner Baum mit folgenden Eigenschaften:

   - *Ein preorder-Durchlauf beginnend am Knoten mit Markierung C ergibt die Knotenfolge $C, X, R, T, S, U, W, J, K, L, V, M$.*
   - *Ein postorder-Durchlauf durch den Baum erzeugt die Konotenfolge $R, T, U, W, S, X, J, V, L, M, K, C$.*

   Zeichnen Sie den durch die beiden Folgen gegebenen allgemeinen Baum und begründen Sie, wie Sie dies aus der Vorgabe abgeleitet haben. Eine Zeichnung ohne diese Herleitung wird nicht gewertet!

2. **AVL Baum** *(4 Punkte)*
   Student Karl behauptet, dass in einem AVL-Baum der Höhe h sich jedes Blatt in der Ebene h oder h-1 befindet. Widerlegen Sie Karl durch ein Gegenbeispiel oder zeigen Sie diese Behauptung durch einen Beweis mittels vollständiger Induktion. Die Wurzel sei dabei Ebene eins und deren Söhne auf Ebene zwei usw.

3. **AVL-Baum** *(6 Punkte)*
   Fügen Sie folgende Zahlen **in der vorgegebenen Reihenfolge** *in einen leeren AVL-Baum ein:* $(7, 6, 5, 1, 3, 9, 8)$ *und löschen Sie dann die Zahlen* $(1, 5, 3)$ **in der vorgegebenen Reihenfolge**. *Stellen Sie die AVL-Bäume dar, die sich als Zwischenschritte ergeben. Geben Sie die durchgeführten Rotationsoperationen und die zuvor vorhandenen Höhenunterschiede für die relevanten Knoten an.*

Tipp:

1. *Gegeben sei ein allgemeiner Baum T mit Wurzel V und k Sohnknoten $S_1, S_2, \ldots, S_k$. Dann ist*

   preorder(T) $: < V, preorder(S_1), preorder(S_2), \ldots, preorder(S_k) >$,

   postorder(T) $: < postorder(S_1), postorder(S_2), \ldots, postorder(S_k), V >$,

2. **AVL-Bedingungen** *für Rotationen (Siehe Abbildung 1 (Seite 8)):*

   (a) **Rechtsrotation***: Die Höhe des Teilbaums* `R1` *ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel* `B`*. Der Unterschied sei durch den Teilbaum* `L` *begründet. Auf dem Teilbaum mit Wurzel* `A` *wird eine Rechtsrotation durchgeführt.*

   (b) **Linksrotation***: analog der Rechtsrotation*

   (c) **Problemsituation links***: (Doppelte Linksrotation) Die Höhe des Teilbaums* `L1` *ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel* `B`*. Der Unterschied sei durch den Teilbaum* `L2` *begründet. Auf dem Teilbaum mit Wurzel* `B` *wird eine Rechtsrotation durchgeführt und dann wird auf dem Baum mit Wurzel* `A` *eine Linksrotation durchgeführt.*

   (d) **Problemsituation rechts***: (Doppelte Rechtsrotation) analog der Problemsituation links.*

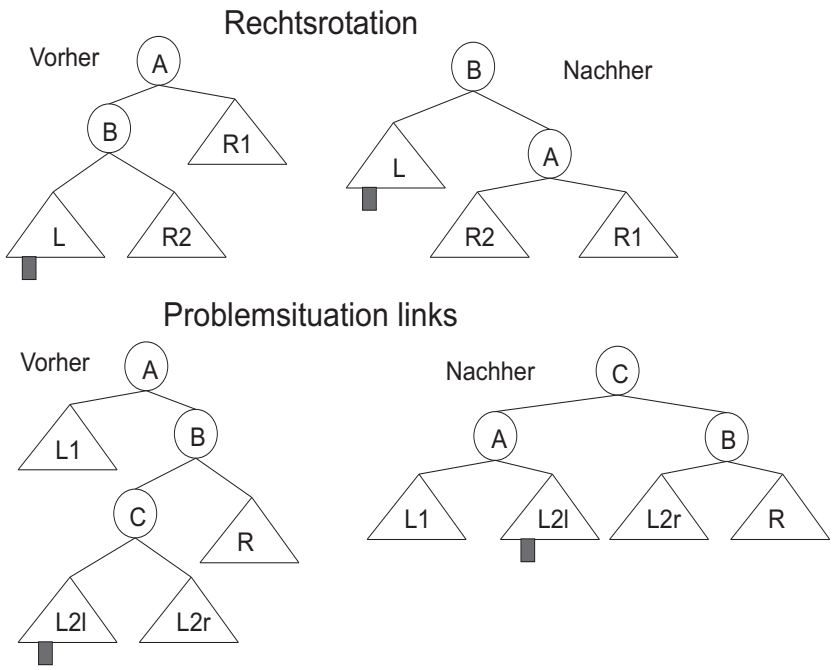

Abbildung 1: Rotationsarten eines AVL-Baumes

Aufgabe 5 (Hashing) *(17P)*

1. **Kollisionsbehandlung** *(12 Punkte)*
   Fügen Sie die Zahlen

   $$107, 20, 37, 46, 1, 2, 4, 50, 0$$

   in dieser Reihenfolge in eine Hashtabelle ein. m gebe dabei die Größe der Hashtabelle an und b die Anzahl der möglichen Elemente pro Index. Die Kollisionsbehandlung ist eine Kombination von Verkettung und offener Adressierung. Als Hashfunktion soll die Divisions-(Rest-)Methode verwendet werden.

   (a) *Es sei* $b = 2, m = 5$ *und verwenden Sie lineares Sondieren* ($s(j,k) = j$).

   (b) *Es sei* $b = 1, m = 11$ *und verwenden Sie Double Hashing* ($s(j,k) = j * h'(k)$) *mit* $h'(x) = 2 * h(x)$ *und* $h'(x) = 1$, *falls* $2 * h(x) = 0$.

   (c) *Welche Eigenschaften sollte eine Hashfunktion haben und was muß zusätzlich für* $h(x), h'(x)$ *beim Double Hashing gelten ? Beurteilen Sie die grundsätzliche Eignung von* $h'(x)$ *aus Aufgabenteil (b) als Hashfunktion sowie im Zusammenhang mit Double Hashing.*

   Das Einfügen ist jeweils so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie die dazu die Zwischenergebnisse der Sondierungsfolge $h_i(x)$ *an und abschließend die Hashtabelle.*

2. **Personalnummer** *(5 Punkte)*
   Die etwa 8000 Mitarbeiter eines Unternehmens mit mehreren Standorten in Deutschland sollen mit einer Hashtabelle verwaltet werde. Als Schlüssel wird die Personalnummer gewählt, die folgenden Aufbau besitzt: `ttmmjjjjaaappppp`. `ttmmjjjj` *stellt das Geburtsdatum dar, bestehend aus Tag, Monat und Jahr.* `aaa` *gibt die Abteilung an, der der Mitarbeiter zugeordnet ist, und* `ppppp` *die Postleitzahl des Standorts. Es soll ein offenes Hashverfahren mit quadratischem Sondieren gewählt werden.*

   (a) *Welches Problem entsteht, wenn als Tabellengröße* $m = 10000$ *und die übliche Hashfunktion* $h(k) = k \bmod m$ *gewählt wird.*

   (b) *Geben Sie eine Hashfunktion an, die das Problem aus (a) vermeidet. Die Tabellengröße* $m = 10000$ *soll dabei unverändert bleiben. Begründen Sie Ihre Entscheidung.*

   (c) *Wie lässt sich das Problem aus (a) vermeiden, falls die Tabellengröße leicht verändert werden darf ?*

Tipp:

1. *Divisions-(Rest-)Methode:* $h(k) = k \ mod \ m$
2. *Sondierungsfolgen:* $h_i(k) = (h(k) - s(i,k)) \ mod \ m$
3. *lineares Sondieren:* $s(j,k) = j$
4. *Double Hashing:* $s(j,k) = j * h'(k)$
5. *quadratisches Sondieren:* $s(j,k) = \left\lceil \frac{j}{2} \right\rceil^2 * (-1)^j$

Die nachfolgenden Lösungen verstehen sich als Musterlösungen, d.h. auch andere Lösungen bzw. Lösungswege sind möglich.

Lösung 1 (Grundlagen) *1.* **Algorithmus** *(5 Punkte)*

(a) Für die Informatik hat der Algorithmus die Bedeutung einer eindeutigen Vorschrift zur Lösung eines Problems mit Hilfe eines Computers.

Zu nennen waren zudem die vier Anforderungen an ein Verfahren, das den Namen „Algorithmus“ verdient:

Eindeutigkeit *: Jeder elementare Schritt ist unmißverständlich beschrieben und läßt keine Wahlmöglichkeit offen.*

Endlichkeit *: Die Beschreibung des Algorithmus hat eine endliche Länge (durch Schleifen o.Ä. kann das ablaufende Programm allerdings durchaus unendlich lange brauchen, wenn der Algorithmus fehlerhaft ist).*

Terminiertheit *: Ein aktuelles Programm, das diesen Algorithmus implementiert, soll in endlicher Zeit beendet sein.*

Reproduzierbarkeit *: Der Algorithmus soll unabhängig von einer speziellen Implementation sein. Verschiedene (korrekte) Implementationen sollen dasselbe Ergebnis liefern,*

(b) Folgende Eigenschaften kann man u.U. beobachten:

- *Liefert der Algoritmus ein Resultat nach endlich vielen Schritten, so sagt man, der Algorithmus* **terminiert**.
- *Ein Algorithmus heißt* **determiniert**, *falls er bei gleichen Eingaben und Startbedingungen stets dasselbe Ergebnis liefert.*
- *Ein Algorithmus heißt* **deterministisch**, *wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht.*
- *Ein Algorithmus heißt* **nicht deterministisch**, *wenn mindestens zu einem Zeitpunkt seiner Ausführung eine Wahlmöglichkeit bei der Fortsetzung besteht.*

2. **Rekursion** *(11 Punkte)*
Beachten Sie: wegen der dreifachen Rekursion sollte diese Funktion nicht naiv direkt so implementiert werden.

(a) *f als rekursive Funktion:*

```
public static int fr(int n) {
    int erg = 0;
    if (n < 3) erg = n+1;
       else erg = 1 + (((fr(n-1) - fr(n-3)) * fr(n-2)) % 100);
return erg;
    }
```

(b) *f als linear-rekursive Funktion:*

```
public static int fl(int n) {
    return linrek(1, 2, 3, n-3);
}
public static int linrek(int a, int b, int c, int steps) {
   int erg = 0;
   if (steps >= 0) {
         int tmp = 1 + (((c - a) * b) % 100);
         steps = steps -1;
         erg = linrek(b,c,tmp,steps); }
      else if (steps == -1) erg = c;
               else if (steps == -2) erg = b;
                       else if (steps == -3) erg = a;
   return erg;
}
```

(c) *f als iterative Funktion:*

```
public static int fi(int n) {
   int f1 = 3; int f2 = 2; int f3 = 1; int erg = 0;
   if (n > 2) {
      while (n > 2) {
         int tmp = 1 + (((f1 - f3) * f2) % 100);
         f3 = f2; f2 = f1; f1 = tmp;
         n--; };
      erg = f1; }
      else if (n == 2) erg = f1;
               else if (n == 1) erg = f2;
                       else if (n == 0) erg = f3;
   return erg;
```

```
}
```

(d) Die Idee für die rekursive Lösung ist, in einem Speicherplatz die bisher berechneten Ergebnisse festzuhalten (z.B. `private static int[] speicher;`*). Statt der rekursiven Aufrufe wird auf den Speicherplatz zugegriffen.*

Linear ist die Lösung, da nur für jedes n eine Berechnung vorgenommen wird. Linear ist der Speicheraufwand, da nur für jedes n ein Eintrag exisitiert.

In den beiden anderen Lösungen ist die Implementierung schon linear, da die drei zuletzt berechneten Ergebnisse und damit die relevanten Ergebnisse in drei Variablen gespeichert werden.

Lösung 2 (Komplexitätstheorie) *(15P)*

1. **Aufwandabschätzung** *(9 Punkte)*
   In `alg1` *wird lediglich eine* `for`*-Schleife in n-Schritten durchlaufen, daher liegt* `alg1` *in* $O(n)$.

   In `alg2` *wird die Zahl n in jedem Durchgang gedrittelt* ($\frac{1}{3^0}, \frac{1}{3^1}, \frac{1}{3^2}, \ldots$). *Daher liegt* `alg2` *in* $O(log(n))$, *da es hierbei nicht auf die Basis ankommt, die in diesem Fall* 3 *wäre.*

   In `alg3` *wird die äußere Schleife n-mal, und die innere* 2^n*- mal durchlaufen (wegen* `p:=p*2`*). Daher liegt* `alg3` *in* $O(2^n)$, *da* $n * 2^n \in O(2^n)$*!*

   In `alg4` *bilden die äußere Schleife zusammen mit der mittleren Schleife die Summe der Zahlen von* 0 *bis n, womit diese beiden in* $O(n^2)$ *liegen. In der inneren Schleife wird* $t(= n)$ *in jedem Durchgang halbiert* ($\frac{1}{2^0}, \frac{1}{2^1}, \frac{1}{2^2}, \ldots$), *womit diese in* $O(log(n))$ *liegt. Daher liegt* `alg4` *in* $O(n^2 log(n))$.

   Es ergibt sich die Rangordnung `alg2 < alg1 < alg4 < alg3`.

2. **Problembehandlung** *(6 Punkte)*

   - **Natürliche Einschränkung** *des Problems: Evtl. muss „nur" ein Spezialfall gelöst werden. Hier hilft ein Nachfragen beim Auftraggeber bzw. eine genauere Analyse des gestellten Problems*

     Im Beispiel: Möchte der Kunde wirklich alle Stellungen des Schachspiels oder evtl. nur des Tic-Tac-Toe Spiels ?

   - **Erzwungene Einschränkung** *des Problems: „nur" gutmütige Probleme oder Spezialfälle lassen sich aufschreiben bzw. werden gelöst. Der Auftraggeber muss darauf hingewiesen werden.*

     Im Beispiel: Es werden ausgehend von nur einer Spielsituation alle Stellungen berechnet, die durch Züge der Bauern möglich sind.

   - **Redefinition des Problems***: Betrachtet man das Problem aus einer anderen Sichtweise ist die Komplexitätsanalyse einfacher (quasi Vermeidung des „mit Kanonen auf Spatzen schießen").*

     Im Beispiel: Könnte man das Schachspiel auch als Tic-Tac-Toe Spiel auffassen ? Oder als Problem des minimalen Gerüstes ?

   - **Approximationen***: Reduktion der Ansprüche an die Qualität der Antworten; Aufgabe des Optimalitätsanspruches.*

     Im Beispiel: Es werden nur wenige Stellungen berechnet, um daraus abzuschätzen, was wohl der beste Zug sein könnte. Das Ergebnis kann ein sehr guter bis hin zu einem sehr schlechten Zug sein.

- **Zeitbeschränkte Exploration** *des Suchraums: Größe der Instanzen, die in zumutbarer Zeit lösbar sind.*

  Im Beispiel: Es werden ausgehend von einer Spielsituation nur alle Stellungen berechnet, die in den nächsten vier Zügen möglich sind.

- **Veränderung** *des Algorithmus: Mögliche Redundanzen vermeiden bzw. verkleinern und ggf. andere Operationen für die Problemlösung verwenden.*

  Im Beispiel: Es werden keine Stellungen mehrfach berechnet. Durch z.B. eine Stellungs-DB wird zunächst geprüft, ob die Stellung schon einmal bewertet wurde. Die Stellungen selbst werden in anderer Form dargestellt, z.B. binäres Muster und die Züge durch `shift`*-Operationen realisiert.*

Lösung 3 (Sortieren) *(37P)*

1. **CornSort** *(14 Punkte)*

   (a) `CornSort` *sortiert das übergebene Array* `A` *aufsteigend. Die while-Schleife wird ausgeführt, solange Elemente vertauscht wurden, also solange* `A` *nicht aufsteigend sortiert ist. Die while-Schleife besteht aus zwei Phasen. In der ersten Phase erfolgt ein aufsteigender Durchlauf durch* `A`*, wobei zwei jeweils benachbarte Elemente miteinander verglichen und ggf. durch Vertauschen lokal sortiert werden. Wurde in der ersten Phase kein Paar vertauscht, so ist* `A` *bereits sortiert und der Algorithmus terminiert. Andernfalls erfolgt in der zweiten Phase ein absteigender Durchlauf durch* `A`*. Dabei werden wiederum benachbarte Elemente verglichen und ggf. durch Vertauschen lokal sortiert. Wurde in der zweiten Phase kein Paar vertauscht, so ist* `A` *bereits sortiert und der Algorithmus terminiert.*

   `CornSort` *ist ein allgemeines internes Sortierverfahren, das durch Vertauschen mit einer quadratischen Laufzeit in situ sortiert.*

   (b) *Sei* `n = length(A)`*. In jedem Durchlauf der while-Schleife wird das gesamte Array* `A` *zweimal durchlaufen. Es werden also jeweils* $2n - 2$ *Vergleiche durchgeführt. Im schlechtesten Fall (Zahlen sind absteigend sortiert) muss das Array* n*-mal durchlaufen werden, womit die while-Schleife offensichtlich maximal* $\frac{n}{2}$*-mal durchlaufen wird. Wir erhalten somit* $O(\frac{n}{2} * (2n - 2)) = O(n * (n - 1)) = O(n^2)$ *als Laufzeit im schlechtesten Fall.*

   `CornSort` *ist kein optimales Sortierverfahren (basierend auf Schlüsselvergleichen), da diese eine Laufzeitkomplexität von* $O(nlog(n))$ *haben.*

   (c) *In der ersten Phase der while-Schleife wird das größte Element* a_{max} *auf jeden Fall gefunden. Es gilt (wenn* $max \neq n - 1$*):* $a_{max} > a_{max+1}$*, womit durch die Vertauschung* a_{max} *in Richtung* A_{n-1} *geschoben wird. Da der Index* i *sich nur um* 1 *erhöht, wird so nun mit jedem weiteren Vergleich* a_{max} *bis zur Position* a_{n-1} *„vertauscht". In der zweiten Phase geschieht nun das gleiche mit dem kleinsten Element* a_{min}*, da der Index dieses mal um* 1 *erniedrigt wird.*

   Somit kann man leicht sehen, das im k*-ten Durchlauf das* k*-größte Element (bzw.* k*-kleinste Element) auf den Platz* a_{n-k} *(bzw.* a_{k-1} *bewegt wird und dort dann auch verbleibt.*

   (d) *Den voran analysierten Sachverhalt kann man ausnutzen, indem man im* k*-ten while-Durchlauf die* $k - 1$ *ersten und letzten Elemente nicht mehr betrachtet. Man spart so im* k*-ten Durchlauf* $2(k - 1)$ *Vergleiche ein.*

(e) *Nun reduzieren sich die Anzahl der Vergleiche pro Durchlauf der while-Schleife von $2n-2$ um 2 Vergleiche, bis keine mehr gemacht werden müssen. Für die Laufzeitkomplexität kann dies mit $\frac{n(n+1)}{2}+\frac{n}{2}$ abgeschätzt werden und wir erhalten: $O(\frac{n(n+1)}{2}+\frac{n}{2}) = O(n^2)$. Daher ändert sich die Laufzeitkomplexität hier nicht.*

2. **Teile und Herrsche** *(11 Punkte)*

(a) *Wir verwenden eine Variante von Quicksort. Die Idee ist, dass das gesuchte Element nur in einer der beiden entstandenen Partitionen gesucht werden muss. Der Algorithmus erhält die zu durchsuchende Folge $S = s_l, \ldots, s_n$ sowie $k < n$ als Eingabe.*
Algorithmus k-kleinstes Element*:*

```
algorithm findk(S, k)
  if |S| == 1
     then return s1.key
     else wähle x = mittlerer key von s1.key,
                    sn.key und s[n/2].key aus S;
          berechne eine Teilfolge S1=s'1 ... s'j
                                  (alle s'i < x) und
                   eine Teilfolge S2=s'j+2 ...s'n
                                  (alle s'i > x) aus
          if k < j+1 then return findk(S1,k)
             else if k > j+1 then return findk(S2,k-(j+1))
                     else return x
  fi      fi      fi
```

(b) *Im Algorithmus ausgelassen ist die Prüfung $n > k$. Dies wird als Voraussetzung dem Anwender publiziert. Durch die Wahl von* `x` *wird zunächst nur die Aufteilung der beiden Folgen* `S1` *und* `S2` *bestimmt und damit auch ihre Größe. Der Median von drei soll für eine ausbalancierte Situation sorgen.*

Der Lösungsraum selbst wird durch die Abfrage `k < j+1` *eingeschränkt. Diese Abfrage stellt jedoch sicher, dass mit der richtigen Hälfte weiter gearbeitet wird. Zu beachten ist, dass wenn mit Folge* `S2` *gearbeitet wird,* `k` *angepasst werden muss, da in Folge* `S1` *sich ja die* `j`*-kleinsten Elemente befinden.*

(c) *Im ersten Durchgang nimmt der Algorithmus* `n` *Schlüsselvergleiche und im schlimmsten Fall* `n` *Datensatzbewegungen vor. In der Rekursion wird dann nur noch mit der Hälfte der Elemente im Durchschnitt gearbeitet. Somit werden* `log(n)` *Rekursionsaufrufe vorgenommen. Die Anzahl der Schlüsselvergleiche bzw. Datensatzbewegungen ist also:*

$$\frac{n}{2^0} + \frac{n}{2^1} + \ldots + \frac{n}{2^{log(n)}} = \sum_{i=0}^{log(n)} \frac{n}{2^i}$$

Nicht erwartet für diese Aufgabe:

$$\sum_{i=0}^{log(n)} \frac{1}{2^i} = 2 * n * (1 - \frac{1}{log(n)}) < n * 2$$

3. **Heap Increase** *(6 Punkte)*
Die Methode `swap(X,Y)` *vertauscht die Inhalte der Variablen* `X` *und* `Y`*. Das Schlüsselwort* `error` *bewirkt, dass die Bearbeitung des Algorithmus unmittelbar mit zugehöriger Fehlermeldung beendet wird. Die Methode* `parent(X)` *ist definiert durch:* $parent(x) := \lfloor \frac{x}{2} \rfloor$

```
algorithm heap-increase-key(A,i,k)
begin if (k < A[i])
         then error("new key is smaller")
      endif
      A[i] := k;
      while (i > 1 and A[parent(i)] < A[i]) do
            swap(A[i],A[parent(i)]);
            i := parent(i);
      endwhile
end heap-increase-key
```

Neben der Korrektheit, dass der neue Schlüssel auch tatsächlich größer als der alte ist, könnte auch die Korrektheit des Index i und des Arrays A geprüft werden. Hier wird vereinfacht davon ausgegangen, dass i gültig im Array A ist und dieses einen Heap implementiert.

Ein Heap mit n Elementen hat maximal die Höhe $log(n)$*, da es sich um einen binären Baum handelt. Der Algorithmus* `heap-increase-key` *läuft im schlechtesten Fall einmal vom Blatt bis zur Wurzel, falls i ein Blatt indiziert. Dadurch ergibt sich die obere Schranke von* $O(log(n))$*.*

4. **Mehr-Wege-Mergesort** *(6 Punkte)*

$T_0 : 43, 71, 12, 35, 11, 80, 99, 47, 62, 15, 77, 26, 69, 93, 19, 45, 23, 38, 56, 91, 10$

Die Eingangsfolge auf Band T_0 hat 21 Elemente, daher werden

$$\lceil log_3(21) \rceil = \lceil 2,771 \rceil = 3$$

Mischphasen benötigt.

Initiale Runs (Länge 1)

```
T3: 43 | 35 | 99 | 15 | 69 | 45 | 56 |
T4: 71 | 11 | 47 | 77 | 93 | 23 | 91 |
T5: 12 | 80 | 62 | 26 | 19 | 38 | 10 |
```

Mischphase 1:

```
T0: 12 43 71 | 15 26 77 | 10 56 91 |
T1: 11 35 80 | 19 69 93 |
T2: 47 62 99 | 23 38 45 |
```

Mischphase 2:

```
T0: 11 12 35 43 47 62 71 80 99 |
T1: 15 19 23 26 38 45 69 77 93 |
T2: 10 56 91 |
```

Mischphase 3:

```
T3: 10 11 12 15 19 23 26 35 38 43 45 47 56 62 69 71 77 80 91 93 99 |
T4: |
T5: |
```

Lösung 4 (Bäume) *(22P)*

1. **Allgemeiner Baum** *(12 Punkte)*
   Aus einem preorder- und einem postorder-Durchlauf kann man relativ einfach den Baum rekonstruieren. Die Lösung hier ist in Abbildung 2 (Seite 20) zu finden.

   Wir erarbeiten uns den Baum „bottom-up“, da Blätter in beiden Ordnungen die gleiche Darstellung haben: sie liegen in der gleichen Reihenfolge nebeneinander. Der Vater eines Blattes bzw. von Blättern liegt in der preorder direkt vor den Blättern/dem Blatt und in der postorder direkt hinter den Blättern/dem Blatt.

   Eine „top-down“ Herleitung ist ebenfalls möglich: hier müssen die jeweils in eine der Ordnungen möglichen Rollen abgeglichen werden.

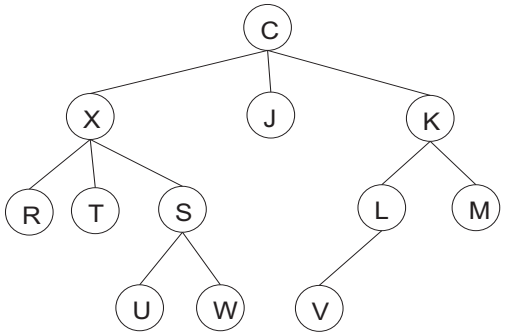

Abbildung 2: Rekonstruierter allgemeiner Baum

In der vorgegebenen Ordnung sind U, W und R, T Blätter. Der Vorgänger von U, W ist S, da er jeweils direkt vor (preorder) bzw. direkt nach (postorder) diesen Knoten aufgeführt ist. Wir ersetzen nun diesen Teilbaum durch S' und erhalten: preorder: $C, X, R, T, S', J, K, L, V, M$; postorder: $R, T, S', X, J, V, L, M, K, C$.

Damit ist X Vater der Knoten R, T, S'. Wir ersetzen diesen Teilbaum durch X' und erhalten: preorder: C, X', J, K, L, V, M; postorder: X', J, V, L, M, K, C.

Auf Grund der Darstellung von V und L in den beiden Ordnungen ist L Vater von V. Wir ersetzen diesen Teilbaum durch L' und erhalten: preorder: C, X', J, K, L', M; postorder: X', J, L', M, K, C.

Damit ist K Vater der Knoten L', M. Wir ersetzen diesen Teilbaum durch K' und erhalten: preorder: C, X', J, K'; postorder: X', J, K', C.

Damit ist C Vater der Knoten X', J, K'. Wir ersetzen diesen Teilbaum durch C' und erhalten: preorder: C'; postorder: C'.

2. **AVL-Baum** *(4 Punkte)*
In Abbildung 3 (Seite 21) ist ein Gegenbeispiel zu finden. In den Knoten

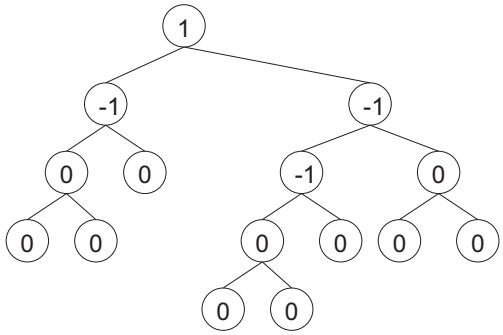

Abbildung 3: Gegenbeispiel

ist der Balance-Faktor (bal(x) = (Höhe des rechten Unterbaumes von x) - (Höhe des linken Unterbaumes von x)) dargestellt. Der Baum hat die Höhe $h = 5$*. Blätter befinden sich auf Ebene drei (=* $h - 2$*), vier (=* $h - 1$*) und fünf (=* $h - 0$*). Beachten Sie: Ein Blatt kann frühestens auf Ebene zwei auftauchen, womit der Baum als Gegenbeispiel mindestens die Höhe vier haben muß!*

3. **AVL-Baum** *(6 Punkte)*
In Abbildung 4 (Seite 21) ist die Lösung bzgl. dem Einfügen zu finden. Angezeigt wurden die Zwischenschritte der Bäume vor und nach einer notwendigen Rotation. In Farbe steht der jeweilige Balance-Faktor an den Knoten. Nach Einfügen von $(7, 6, 5)$ *ist eine Rechtsrotation um Knoten* 7 *notwen-*

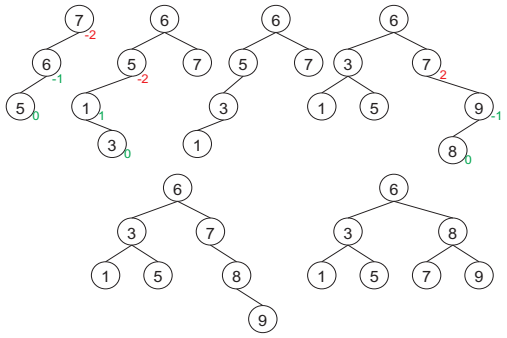

Abbildung 4: Entwicklung eines AVL-Baums

dig. Nach weiterem Einfügen von $(1, 3)$ *ist eine doppelte Rechtsrotation um die Knoten* 1 *und* 5 *notwendig. Nach weiterem Einfügen von* $(9, 8)$ *ist eine doppelte Linksrotation um die Knoten* 9 *und* 7 *notwendig.*

In Abbildung 5 (Seite 22) ist die Lösung bzgl. dem Löschen zu finden. Nach

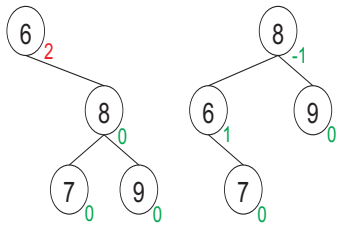

Abbildung 5: Löschen in einem AVL-Baums

dem Löschen von $(1, 5, 3)$ *ist eine Linksrotation um Knoten* 6 *notwendig.*

Lösung 5 (Hashing) *(17P)*

1. **Kollisionsbehandlung** *(12 Punkte)*

   (a) $h_0(107) = 107 \ mod \ 5 = 2$
   $h_0(20) = 20 \ mod \ 5 = 0$
   $h_0(37) = 37 \ mod \ 5 = 2$, *keine Kollision, da pro Index zwei Elemente gespeichert werden können.*
   $h_0(46) = 46 \ mod \ 5 = 1$
   $h_0(1) = 1 \ mod \ 5 = 1$
   $h_0(2) = 2 \ mod \ 5 = 2$, *Kollision:*
   $h_1(2) = (2 - s(1,2)) \ mod \ 5 = (2 - 1) \ mod \ 5 = 1$
   $h_2(2) = (2 - s(2,2)) \ mod \ 5 = (2 - 2) \ mod \ 5 = 0$
   $h_0(4) = 4 \ mod \ 5 = 4$
   $h_0(50) = 50 \ mod \ 5 = 0$, *Kollision:*
   $h_1(50) = (0 - s(1,50)) \ mod \ 5 = (0 - 1) \ mod \ 5 = 4$
   $h_0(0) = 0 \ mod \ 5 = 0$, *Kollision:*
   $h_1(0) = (0 - s(1,0)) \ mod \ 5 = (0 - 1) \ mod \ 5 = 4$
   $h_2(0) = (0 - s(2,0)) \ mod \ 5 = (0 - 2) \ mod \ 5 = 3$
   Hier die abschließende Tabelle:

ht[0]	*ht[1]*	*ht[2]*	*ht[3]*	*ht[4]*
20,2	*46,1*	*107,37*	*0*	*4,50*

   (b) $h_0(107) = 107 \ mod \ 11 = 8$
   $h_0(20) = 20 \ mod \ 11 = 9$
   $h_0(37) = 37 \ mod \ 11 = 4$
   $h_0(46) = 46 \ mod \ 11 = 2$
   $h_0(1) = 1 \ mod \ 11 = 1$
   $h_0(2) = 2 \ mod \ 11 = 2$, *Kollision mit* $h'(2) = 2 * 2 = 4$*:*
   $h_1(2) = (2 - s(1,2)) \ mod \ 11 = (2 - 1 * 4) \ mod \ 11 = 9$
   $h_2(2) = (2 - s(2,2)) \ mod \ 11 = (2 - 2 * 4) \ mod \ 11 = 5$
   $h_0(4) = 4 \ mod \ 11 = 4$, *Kollision mit* $h'(4) = 2 * 4 = 8$*:*
   $h_1(4) = (4 - s(1,4)) \ mod \ 11 = (4 - 1 * 8) \ mod \ 11 = 7$
   $h_0(50) = 50 \ mod \ 11 = 6$
   $h_0(0) = 0 \ mod \ 11 = 0$
   Hier die abschließende Tabelle:

ht[0]	*ht[1]*	*ht[2]*	*ht[3]*	*ht[4]*	*ht[5]*	*ht[6]*	*ht[7]*	*ht[8]*	*ht[9]*	*ht[10]*
0	*1*	*46*		*37*	*2*	*50*	*4*	*107*	*20*	

   (c) *Hashfunktionen sollen im Wesentlichen drei Eigenschaften haben: Sie sollen surjektiv sein, um einen geringen Speicherbedarf zu haben, sie sollen injektiv sein, um die Elemente gleichmäßig auf alle Indizes zu*

verteilen und sie sollen effizient zu berechnen sein. Beim Double Hashing ist zusätzlich darauf zu achten, dass beide Hashfunktionen voneinander unabhängig sind.

Die in Aufgabenteil (b) verwendete Funktion $h'(x)$ ist nicht surjektiv, da sie immer gerade Werte liefert, womit sie auch keine gute Gleichverteilung erreichen kann. Sie ist von $h(x)$ definitiv abhängig, da sie diese Funktion zur Berechnung verwendet. Sie benötigt einen Spezialfall, um nicht null zu werden. Damit ist sie im Allgemeinen und speziell beim Double Hashing als Hashfunktion klar ungeeignet.

2. **Personalnummer** *(5 Punkte)*

(a) Es ist eine sehr schlecht streuende Hashfunktion, da alle Mitarbeiter eines Standorts auf die gleiche Hashadresse abgebildet werden: die letzten vier Zahlen der Postleitzahl:

$$h(ttmmjjjjaaappppp) = ttmmjjjjaaappppp \ mod \ 10000 = pppp$$

(b) Die einzelnen Informationen in der Personalnummer können mehrfach vorkommen (Geburtsdatum, Abteilung oder Postleitzahl des Standorts). Jedoch sollte die Kombination aus allen Elementen einen individuellen Schlüssel erzeugen. Dazu sei die Personalnummer als Array aufgefasst:

$$h(k) := a_{15}(k) \ mit \ a_i(k) = (128 * a_{i-1}(k) + k[i]) \ mod \ m \ und \ a_0(k) = k[0]$$

Die Multiplikation mit 128 *soll eine Verzerrung der einzelnen Ziffern bewirken, damit nicht alle Ziffern auf die gleiche Stelle gleich stark wirken. Mit der Multiplikation ist z.B. es wichtig, an welcher Stelle in der Personalnummer die Ziffer drei steht.*

(c) Wähle als m kleinste Primzahl $p > 10000$ mit der Form $p = 4 * i + 3$: $10007 = 4 * 2501 + 3$ ist Primzahl mit diesen Bedingungen, womit eine Permutation auf den Zahlen $0, \ldots, m-1$ erzeugt wird.