Klausuraufgaben

Algorithmen und Datenstrukturen

01. Februar 2006

Zur Lösung der nachfolgenden Aufgaben haben Sie 90 Minuten Zeit. Es sind **keine Hilfsmittel** zugelassen!

- Für alle Aufgaben ist der **Lösungsweg** und die Lösung **nachvollziehbar** anzugeben, d.h. Sie müssen mich von Ihrer Lösung und dem Lösungsweg überzeugen! (Dies setzt u.a. eine **saubere Schrift** voraus!). In diesem Sinne werden Antworten ohne Begründung oder zu schlechter Schrift **nicht bewertet**!
- Bitte schreiben Sie zuerst Ihren Namen und Ihre Matrikelnummer zumindest auf das erste Blatt und auf alle losen Blätter zumindest die Matrikelnummer!

Lesen Sie erst alle Aufgaben durch! Sie sind **nicht** nach dem Schwierigkeitsgrad geordnet. Am Ende der Klasur finden Sie Tip's zu den Aufgaben.

Aufgabe	1	2	3	4	5	$\sum$
	Grundlagen	Komplexität	Sortieren	Graphen	Hashing	Summe
mögliche Punkte	24	13	25	24	21	107
erreichte Punkte						

Erreichte Bewertung:

Aufgabe 1 (Grundlagen) *(24P)*

1. **Algorithmus** *(8 Punkte)*
   Erklären Sie die vier allgemeinen Anforderungen an einen Algorithmus.

   Erklären Sie folgende vier Eigenschaften eines Algorithmus: terminiert, determiniert, deterministsich und nicht deterministisch.

2. **Datenabstraktion** *(6 Punkte)*
   Beschreiben Sie das Prinzip bzw. die Grundidee der Datenabstraktion.

   Welche vier Typen von Funktionen benötigt man im Allgemeinen? (unabhängig von der konkreten Datenstruktur als „Schnittstelle“) Geben Sie für einen Stapel jeweils ein erklärendes kleines Beispiel an. Es genügt der Name einer Funktion/Methode und eine kurze Erklärung, was sie macht.

3. **Rekursion** *(10 Punkte)*
   Die Addition kann durch Verwendung der Nachfolger- (`succ`*) und Vorgänger-(*`pred`*) Funktionen realisiert werden. Im Falle der Addition sind das die Funktionen* `+1` *und* `-1`*.*

   Schreiben Sie in einer (Pseudo-)Programmiersprache Ihrer Wahl **zwei Versionen** *der Addition: eine Version, die einen* **rekursiven Ablauf** *beschreibt, und eine Version, die einen* **iterativen Ablauf** *beschreibt. Beide Versionen sollen jeweils nur genau* **zwei ganze Zahlen** *addieren können und dies ausschließlich mittels der beiden genannten Funktionen!*

Aufgabe 2 (Komplexitätstheorie) *(13P)*

1. **Begriffsbestimmung** *(4 Punkte)*
   Beschreiben Sie die drei interessanten Fälle **Bester Fall**, **Schlechtester Fall** *und* **Mittlerer Fall**.

2. **Problembehandlung** *(9 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 drei davon und erläutern Sie diese kurz an einem Beispiel.

Aufgabe 3 (Sortieren) *(25P)*

1. **Teile-und-Herrsche** *(5 Punkte)*
   Erklären Sie kurz das Teile-und-Herrsche-Prinzip. Beschreiben Sie dazu grob den Ablauf.

   Welches Problem könnte bei der Durchführung bestehen ? Denken Sie dabei z.B. an die Praktikumsaufgabe, in der die Teilsumme einer Folge von Zahlen zu bestimmen war (siehe die Tip's am Ende der Klausur dazu).

2. **Rekursion** *(6 Punkte)*
   Quicksort und Mergesort unterscheidet nicht nur die Anwendung: Quicksort wird zu den „internen Sortierverfahren" gezählt und Mergesort zu den „externen Sortierverfahren".

   Beide arbeiten nach dem Teile-und-Herrsche-Prinzip. Was unterscheidet dennoch beide Algorithmen im Punkt Rekursion? Welche Auswirkung hat dies für eine iterative Variante von Quicksort?

3. **Quicksort** *(14 Punkte)*
   Sotieren Sie die folgende Zahlenreihe mit dem im Tip aufgeführten Quicksort-Algorithmus. Notieren Sie die Zwischenergebnisse nach jeder Tauschoperation.

43	*19*	*81*	*2*	*39*	*41*	*3*	*99*	*12*

   Geben Sie die Anzahl der Rekursionsaufrufe und die Rekursionstiefe an

Aufgabe 4 (Bäume und Graphen) *(24P)*

1. **Greedy-Algorithmen** *(11 Punkte)*
   Bestimmen Sie für nachfolgendes Problem folgende Elemente des im Tip aufgeführten allgemeinen Greedy-Algorithmus: Eine Menge von Kandidaten **C**, *aus der die Lösung konstruiert wird; Die Lösungsmenge* **S**$\subseteq C$*; Eine boolesche Funktion* **solution**, *die angibt, ob eine Menge von Kandidaten eine legale Lösung des Problems darstellt; Eine Testfunktion* **feasible**, *die angibt, ob eine Teillösung noch zu einer kompletten legalen Lösung erweitert werden kann; Eine Auswahlfunktion* **select**, *die einen noch unbenutzten Kandidaten liefert, der im Sinne der Greedy-Strategie der erfolgversprechendste ist; Eine Zielfunktion* **val**, *die den Wert einer gewissen Lösung angibt.*

   Wenden Sie den daraus resultierenden Algorithmus auf folgendes „Bepackungsproblem" an: Gegeben sei ein leerer Rucksack mit maximalem Fassungsvermögen von 30kg. Versuchen Sie gemäß Ihrem Algorithmus den Rucksack mit folgenden Teilen zu bepacken: 7kg,12kg,14kg,16kg,20kg.

   Welche Teile würden Sie „von Hand" (also ohne erkennbares Prinzip) auswählen ?

2. **AVL-Bäume** *(7 Punkte)*
   In dieser Aufgabe ist ein AVL-Baum zu erstellen. Fügen Sie in der folgenden, vorgegebenen Reihenfolge die Elemente in einen leeren AVL-Baum ein. Als Ordnungsrelation gilt die lexikographische Ordnung. Erläutern Sie, wann und welche Rotationen stattfinden (Der Ablauf des Einfügens muss klar sein)! Zeichnen Sie den Baum nach jeder Rotation neu auf.

   Söndre_Strömfjord, Berlin, Casablanca, Dakar, El_Salvador, Oslo, Windhuk

3. **optimale Wege** *(6 Punkte)*
Führen Sie den Algorithmus von Dijkstra mit dem Graphen in Abbildung 1 (Seite 6) durch (bis zum Abbruch!). Wählen Sie als Ausgangspunkt den Knoten x_1. Fertigen Sie, ähnlich wie in der Vorlesung, eine Dokumentation an, aus der der Ablauf deutlich wird. Bei Wahlmöglichkeiten ist immer die lexikographisch niedrigste Ecke zu wählen.

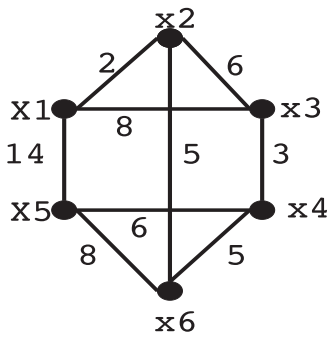

Abbildung 1: Optimaler Weg gesucht!

Aufgabe 5 (Hashing) *(19P)*

1. **Konstruktionsprinzip** *(9 Punkte)*
   Suchalgorithmen, die mit Hashing arbeiten, bestehen aus zwei Teilen. (2P) Welche sind dies? (4P)Welche Ziele verfolgen die einzelnen Teile?

   (3P) Warum ist Hashing ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf?

2. **Separate Chaining** *(5 Punkte)*
   Fügen Sie die Zahlen $13, 5, 27, 60, 2, 65, 8, 37, 18$ *in dieser Reihenfolge unter Verwendung der Hashfunktion* $h(k) = k mod 7$ *in eine Hashtabelle der Größe* $N = 7$ *(nummeriert von 0 bis 6) ein. Die Kollisionsbehandlung erfolge durch verkettete Listen.*

   Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion an.

3. **Lineares Sondieren** *(7 Punkte)*
   Fügen Sie die Zahlen $12, 7, 22, 14, 8, 4, 11, 3$ *in dieser Reihenfolge unter Verwendung der Hashfunktion* $h(k) = k mod 8$ *in eine Hashtabelle der Größe* $N = 8$ *(nummeriert von 0 bis 7) ein. Die Kollisionsbehandlung erfolge durch lineares Sondieren (*$h_i(k) = (h(k) + i) \ mod \ 8$*).*

   Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h *bzw. bei Kollisionen der Hashfunktionen* h_i *an.*

Im folgenden sind Hinweise/**Tip**'s zu den Klausuraufgaben gegeben.

- **Teilsumme** Sei $folge := (5, -8, 3, 3, -5, 7, -2, -7, 3, 5)$ eine Folge von ganzen Zahlen. Dann ist eine Teilfolge dieser Folge ein zusammenhängender Teil der Folge. Jede Folge hat zwei extreme Teilfolgen: die leere Teilfolge () und die gesamte Teilfolge (auch als unechte Teilfolge bezeichnet). Teilfolgen wären z.B.: $teilfolge_1 := (3, 3); teilfolge_2 := (-5, 7, -2)$; Keine Teilfolgen wären $keineteilfolge_1 := (5, 3, 3, 7, 3, 5); keineteilfolge_2 := (-8, -5, -2)$; insbesondere darf die ursprüngliche Folge nicht umsortiert werden!

  Gesucht war die Teilfoge mit der maximalen Summe bei Addierung ihrer Elemente.

- **Algorithmus Quicksort**

```
static void quicksort(ITEM[] a, int l, int r)
  {if ( r <= l) return;
   int i = partiotion(a,l,r);
   quicksort(a,l,i-1);
   quicksort(a,i+1,r);}

static int partition(ITEM[] a, int l, int r)
  {int i = l, j = r;   ITEM p = a[r];
   for (;;) // Endlosschleife
     {while (less(a[i],p)) i = i+1;
      while (less(p, a[j]))
       {j = j-1;
        if (j == i) break;}
      if (i >= j) break;
      exch(a,i,j);}
   return i;}

static void exch(ITEM[] a, int l, int r)
  {ITEM tmp = a[r];
   a[r] = a[l]; a[l] = tmp;}
```

- **Algorithmus Mergesort**
  Falls F einen oder keinen Datensatz enthält, tue nichts
  sonst:
  Divide Teile F in zwei gleich große Teilfolgen F_1 und F_2.
  Conquer Mergesort(F_1); Mergesort(F_2)
  Merge Bilde Resultatfolge durch Zusammenführen von F_1 und F_2.

- **Greedy-Algorithmus**

```
function greedy(c) : set
 R := emptyset
 while not solution(R) and C <> emptyset do
 { x := select(C,R)
   C := C - {x}
   if feasible(R + {x}) then
      R := R + {x}
  } // end-while
 if solution(R) then
    return R
   else
    {Es gibt keine Loesung!}
```

- **AVL-Bedingungen** für Rotationen (Siehe Abbildung 2 (Seite 10)):

  1. **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.
  2. **Linksrotation**: analog der Rechtsrotation
  3. **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.
  4. **Problemsituation rechts**: (Doopelte Rechtsrotation) analog der Problemsituation links.

- **Algorithmus von Dijkstra**

  Vorbereitung l_{ij}: Länge der Kante $x_i x_j$. Falls es keine Kante $x_i x_j$ gibt, sei $l_{ij} := \infty$ Für jede Ecke $x_i \in X$ des zu untersuchenden Graphen werden drei Variable angelegt:

  1. $Entf_i$ gibt die bisher festgestellte kürzeste Entfernung von x_1 nach x_i an. Der Startwert ist 0 für $i = 1$ und ∞ sonst.
  2. $Vorg_i$ gibt den Vorgänger von x_i auf dem bisher kürzesten Weg von x_1 nach x_i an. Der Startwert ist x_1 für $i = 1$ und undefiniert sonst.
  3. OK_i gibt an, ob die kürzeste Entfernung von x_1 nach x_i schon bekannt ist. Der Startwert für alle Werte von i ist *false*.

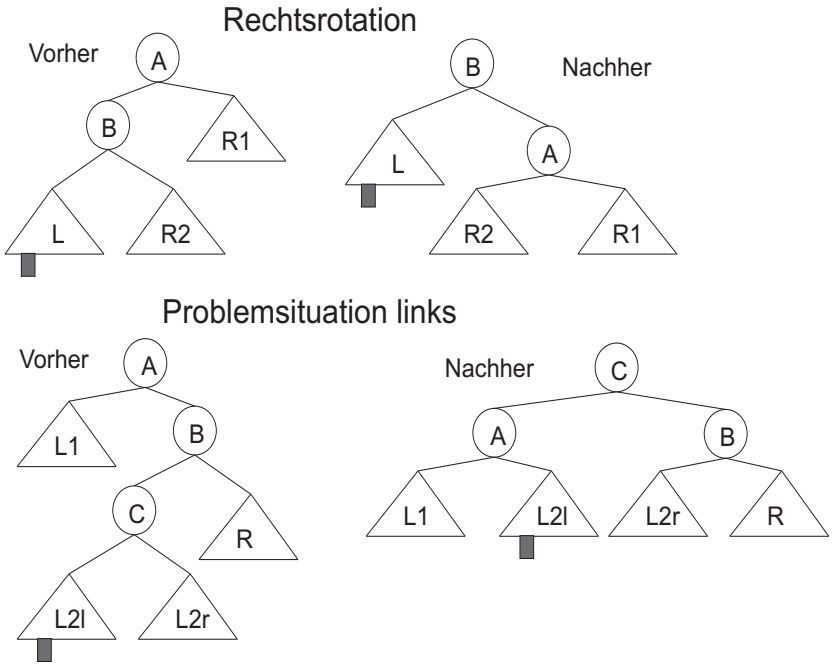

Abbildung 2: Die Rotationsarten eines AVL-Baumes

Iteration Wiederhole

- Suche unter den Ecken x_i mit $OK_i = \mathit{false}$ eine Ecke x_h mit dem kleinsten Wert von Entf_i.
- Setze $OK_h := \mathit{true}$.
- Für alle Ecken x_j mit $OK_j = \mathit{false}$, für die die Kante $x_h x_j$ existiert:
  - Falls gilt $\mathit{Entf}_j > \mathit{Entf}_h + l_{hj}$ dann
    - Setze $\mathit{Entf}_j := \mathit{Entf}_h + l_{hj}$
    - Setze $\mathit{Vorg}_j := h$

solange es noch Ecken x_i mit $OK_i = \mathit{false}$ gibt.

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

Lösung 1 (Grundlagen) *(24P)*

1. **Algorithmus** *(8 Punkte)*
   (4P) Zu nennen waren die 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,*

   (4P) 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. **Datenabstraktion** *(6 Punkte)*
   (2P) Mit der Datenabstraktion stehen Methoden zur Verfügung, mit denen die Art der Verwendung eines Datenobjektes von den Details seiner Konstruktion aus elementareren Datenobjekten getrennt werden kann, d.h. dass nicht bekannt ist, wie die verwendete Datenstruktur „intern“ realisiert ist. Die Grundidee bei der Datenabstraktion besteht darin, die Programme zur Verwendung von zusammengesetzten Datenobjekten so zu strukturieren, dass sie mit „abstrakten Daten“ arbeiten. Gleichzeitig wird eine „konkrete“ Darstellung der Daten unabhängig von den die Daten verwendenden Programmen definiert.

   (4P) Die Schnittstelle zwischen diesen beiden Teilen ist eine Menge von Funktionen:

- *Für die Datenabstraktion werden* **Konstruktoren** *benötigt, die die gewünschte Datenstruktur erzeugen. Beispiel:* `Empty` *erzeugt einen leeren Stapel.*
- *Mittels* **Selektoren** *kann auf die erzeugte Datenstruktur zugegriffen werden. Beispiel:* `top` *liefert das oberste Element eines Stapels.*
- **Bedingungen** *können bestimmte Eigenschaften abprüfen. Beispiel:* `isempty` *prüft, ob ein Stapel leer ist.*
- *Mittels* **Mutatoren** *kann die erzeugte Datenstruktur destruktiv verändert werden. Beispiel:* `pop` *entfernt das oberste Element eines Stapels.*

3. **Rekursion** *(10 Punkte)*
Die Funktion `rplus` *stellt einen rekursiven Ablauf dar; die Funktion* `iplus` *stellt einen iterativen Ablauf dar.*

```
; Addiert zwei Zahlen a und b
; input: Zahlen a, b
; output: a + b
static int iplus(int a, int b)
  { if (a > 0)
      { while (a > 0)
          { a = -1(a); b = +1(b); } }
      else
      { while (a < 0)
          { a = +1(a); b = -1(b); } }
    return b; }

static int rplus(int a, int b)
  { if (a > 0) { a = -1(a); b = +1(b); }
    if (a < 0) { a = +1(a); b = -1(b); }
    if (a <> 0) { b = rplus(a,b) }
    return b; }
```

Lösung 2 (Komplexitätstheorie) *(13P)*

1. **Begriffsbestimmung** *(4 Punkte)*

Der beste Fall *(best case): (1P) Das ist diejenige Struktur der Eingabedaten, für die der Algorithmus am effektivsten ist.*

Der schlechteste Fall *(worst case): (1P) Diejenige Eingabestruktur, für die der Algorithmus am wenigsten effektiv ist.*

Der mittlere Fall *(average case): (2P) Die zu erwartende Effizienz bei beliebiger Strukturierung der Eingabedaten.*

2. **Problembehandlung** *(9 Punkte)*
   Nur drei der folgenden sechs sollten genannt werden!

   - **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) *(25P)*

1. **Teile-und-Herrsche** *(5 Punkte)*
   (3P) Teile: *Der Algorithmus teilt ein Problem in Teilprobleme auf.* Herrsche:

Der Algorithmus löst die Teilprobleme u.U. nach dem gleichen Prinzip, also rekursiv. Zusammenführen: *Die gelösten Teilprobleme werden zu einer Lösung des Problems zusammengeführt.*

(2P) Bei der Teilung des Problems ist darauf zu achten, dass eine mögliche Lösung nicht geteilt und damit nicht erkannt wird. Im Praktikum war dies eine Teilsumme, die über die Teilungsgrenzen hinweg geht. In einem solchen Fall entsteht zusätzlicher Aufwand, um dieses Problem zu handhaben.

2. **Rekursion** *(6 Punkte)*
(2P) Quicksort führt eine Art Endrekursion durch, d.h. sind alle Rekursionsaufrufe erfolgt (ohne deren Beendung!), steht das Ergebnis fest. Die eigentiche Arbeit wird also vor den Rekursionsaufrufen durchgeführt. (2P) Im Praktikum war dies daran zu merken, dass eine echte iterative Variante nicht implementierbar war: die Grenzen der zu sortierenden Intervalle mussten auf einem Stapel gespeichert werden, um einen korrekten Ablauf zu erzeugen.

(2P) Bei Mergesort wird die eigentliche Arbeit nach den Rekursionsaufrufen durchgeführt, d.h. wenn alle Rekursiuonsaufrufe erfolgt sind, beginnt durch die Zusammenführung, also das mischen *die eigentliche Arbeit des Sortierens. Das Ergebnis steht erst fest, wenn aus der erste Aufruf abgeschlossen ist.*

3. **Quicksort** *(14 Punkte)*
*(12P) Das Pivotelement wird durch // markiert. Ist bezüglich einem Pivotelement sortiert worden, wird dieses weggelassen. Wird quicksort mit einem Feld der Länge kleiner oder gleich 1 aufgerufen, wird dies mit * markiert. Ganz rechts werden die Rekursionsaufrufe gezählt.*

43	*19*	*81*	*2*	*39*	*41*	*3*	*99*	*/12/*	*0*
12	*19*	*81*	*2*	*39*	*41*	*3*	*99*	*43*	*0*
3	*19*	*81*	*2*	*39*	*41*	*12*	*99*	*43*	*0*
3	*12*	*81*	*2*	*39*	*41*	*19*	*99*	*43*	*0*
3	*2*	*81*	*12*	*39*	*41*	*19*	*99*	*43*	*0*
3	*2*	*12*	*81*	*39*	*41*	*19*	*99*	*43*	*0*
3	*/2/*		*81*	*39*	*41*	*19*	*99*	*/43/*	*2*
2	*3*		*43*	*39*	*41*	*19*	*99*	*81*	*2*
	3		*19*	*39*	*41*	*43*	*99*	*81*	*4*
			19	*39*	*/41/*		*99*	*/81/*	*6*
			19	*39*	*41*		*81*	*99*	*6*
			19	*/39/*				**99**	*10*
			19						*12*

(2P) Die Anzahl der Rekursionsaufrufe ist 12 (siehe Zählung) und die Rekursionstiefe 5 (Anzahl der Rekursionsaufrufe im längsten Zweig).

Lösung 4 (Bäume und Graphen) *(24P)*

1. **Greedy-Algorithmen** *(11 Punkte)*
   Hier die Definitionen für das zu lösende Problem:

   (a) *(0,5P)Menge von Kandidaten* $C := \{7kg, 12kg, 14kg, 16kg, 20kg\}$.

   (b) *(0,5P)Lösungsmenge* $R \subseteq C$*, am Anfang leer, der Rucksack.*

   (c) *(1,5P)Boolesche Funktion solution:* $\sum_{x \in R} x \leq 30kg$

   (d) *(1,5P)Testfunktion* $feasible$*:* $\sum_{x \in R} x \leq 30kg$

   (e) *(2P)select:* $x \in C : (x \notin R) \wedge (\nexists y \in C \setminus R : y < x)$*, d.h. x ist minimal*

   (f) *(1P) val:* $\sum_{x \in R} x$*, das Gewicht des Rucksack's*

   (3P) Hier der Ablauf des Algorithmus:

R	C
$\emptyset$	$\{7kg, 12kg, 14kg, 16kg, 20kg\}$
$\{7kg\}$	$\{12kg, 14kg, 16kg, 20kg\}$
$\{7kg, 12kg\}$	$\{14kg, 16kg, 20kg\}$
$\{7kg, 12kg\}$	$\{16kg, 20kg\}$
$\{7kg, 12kg\}$	$\{20kg\}$
$\{7kg, 12kg\}$	$\emptyset$

   (1P) Von Hand würde ich den Rucksack wie folgt bepacken: $R := \{14kg, 16kg\}$.

2. **AVL-Bäume** *(7 Punkte)*
   Die resultierenden Bäume sind in Abbildung 3 (Seite 16) dargestellt. Nach dem ersten Baum wird eine doppelte Rechtsrotation (Problemsituation rechts) bei der Ecke `Söndre_Strömfjord` *durchgeführt. Nach dem zweiten Baum (in der Abbildung oben rechts) wird bei der Ecke* `Söndre Strömfjord` *eine doppelte Rechtsrotation (Problemsituation rechts) durchgeführt. Nach dem dritten Baum wird bei der Ecke* `Casablanca` *eine Linksrotation durchgeführt.*

3. **optimale Wege** *(6 Punkte)*
   Algorithmus von Dijkstra:

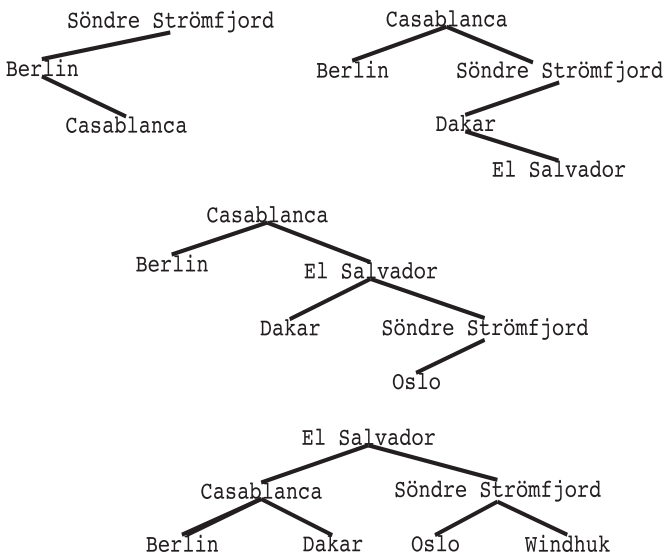

Abbildung 3: Einfügen im AVL-Baum

x_6	x_5	x_4	x_3	x_2	x_1	T					
-	-	-	-	-	*0*	$\{x_6$	x_5	x_4	x_3	x_2	$x_1\}$
-	*14*,x_1	-	*8*,x_1	*2*,x_1		$\{x_6$	x_5	x_4	x_3	x_2	$\}$
7,x_2	*14*	-	*8*			$\{x_6$	x_5	x_4	x_3		$\}$
	14	*12*,x_6	*8*			$\{$	x_5	x_4	x_3		$\}$
	14	*11*,x_3				$\{$	x_5	x_4			$\}$
	14					$\{$	x_5				$\}$
						$\{$					$\}$

Der optimale Weg z.B. von x_1 nach x_4 ist: $x_1x_3x_4$.

Lösung 5 (Hashing) *(21P)*

1. **Konstruktionsprinzip** *(9 Punkte)*
   (6P) Suchalgorithmen, die mit Hashing arbeiten, bestehen aus folgenden zwei Teilen:

   (a) *(1P) Im ersten Schritt transformiert der Algorithmus den Suchschlüssel mithilfe einer Hashfunktion in eine Tabellenadresse.*

   (1P) Diese Funktion bildet im Idealfall unterschiedliche Schlüssel auf unterschiedliche Adressen ab. Oftmals können aber auch zwei oder mehrere unterschiedliche Schlüssel zur gleichen Tabellenadresse führen.

   (1P) Die Hashfunktion versucht eine eventuell vorhandene Häufung der Schlüssel aufzuheben, also das Ziel ist eine normale Verteilung der

Schlüssel zu erreichen, ohne von der Verteilung der Schlüssel selbst beeinflusst zu werden.

(b) *(1P) Somit führt eine Hashing-Suche im zweiten Schritt eine Kollisionsbeseitigung durch, die sich mit derartigen Schlüsseln befasst.*

(2P) Hier gibt es zwei Möglichkeiten: Zum Einen kann auf ein anderes Verfahren zurückgegriffen werden, z.B. lineare Listen oder ausgeglichene Bäume. Zum anderen kann rekursiv gearbeitet werden: Hierbei ist das Ziel, durch die Kollisionsbeseitigung möglichst wenige Kollisionen für die nachfolgenden Einfügevorgänge zu erzeugen.

(3P) Hashing ist ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf. Wenn es keine Speicherbeschränkung gäbe, könnten jede Suche mit nur einem einzigen Speicherzugriff realisiert werden, indem einfach der Schlüssel als Speicheradresse analog zu einer schlüsselindizierten Suche verwendt wird. Allerdings lässt sich dieser Idealzustand meistens nicht verwirklichen, weil der Speicherbedarf besonders bei langen Schlüsseln viel zu groß ist. Gäbe es andererseits keine Zeitbeschränkung, könnten wir mit einem Minimum an Speicher auskommen, indem ein sequenzielles Suchverfahren eingesetzt wird. Hashing erlaubt es, zwischen diesen beiden Extremen ein vertretbares Maß sowohl für die Zeit als auch den Speicherplatz zu finden. Insbesondere können wir jedes beliebige Verhältnis einstellen, indem wir lediglich die Größe der Hashtabelle anpassen; dazu brauchen wir weder Code neu zu schreiben noch auf andere Algorithmen auszuweichen.

2. **Separate Chaining** *(5 Punkte)*
Die Listen werden in nachfolgender Tabelle durch Klammern dargestellt.

-	-	-	-	-	-	*13*	$13 mod 7 = 6$
-	-	-	-	-	*5*	*13*	$5 mod 7 = 5$
-	-	-	-	-	*5*	*(13,27)*	$27 mod 7 = 6$
-	-	-	-	*60*	*5*	*(13,27)*	$60 mod 7 = 4$
-	*2*	-	-	*60*	*5*	*(13,27)*	$2 mod 7 = 2$
-	*(2,65)*	-	-	*60*	*5*	*(13,27)*	$65 mod 7 = 2$
-	*(2,65,8)*	-	-	*60*	*5*	*(13,27)*	$8 mod 7 = 1$
-	*(2,65,8)*	*37*	-	*60*	*5*	*(13,27)*	$37 mod 7 = 2$
-	*(2,65,8)*	*37*	-	*(60,18)*	*5*	*(13,27)*	$18 mod 7 = 4$

3. **Lineares Sondieren** *(7 Punkte)*
Die Kollisionsbehandlung wird in eigenen Zeilen beschrieben.

-	-	-	-	*12*	-	-	-	$12 mod 8 = 4$
-	-	-	-	*12*	-	-	*7*	$7 mod 8 = 7$
-	-	-	-	*12*	-	*22*	*7*	$22 mod 8 = 6$
-	-	-	-	*12*	-	*22*	*7*	$14 mod 8 = 6$
-	-	-	-	*12*	-	*22*	*7*	$(6 + 1) mod 8 = 7$
14	-	-	-	*12*	-	*22*	*7*	$(6 + 2) mod 8 = 0$
14	-	-	-	*12*	-	*22*	*7*	$8 mod 8 = 0$
14	*8*	-	-	*12*	-	*22*	*7*	$(0 + 1) mod 8 = 1$
14	*8*	-	-	*12*	-	*22*	*7*	$4 mod 8 = 4$
14	*8*	-	-	*12*	*4*	*22*	*7*	$(4 + 1) mod 8 = 5$
14	*8*	-	*11*	*12*	*4*	*22*	*7*	$11 mod 8 = 3$
14	*8*	-	*11*	*12*	*4*	*22*	*7*	$3 mod 8 = 3$
14	*8*	-	*11*	*12*	*4*	*22*	*7*	$(3 + 1) mod 8 = 4$
14	*8*	-	*11*	*12*	*4*	*22*	*7*	...
14	*8*	-	*11*	*12*	*4*	*22*	*7*	$(3 + 6) mod 8 = 1$
14	*8*	*3*	*11*	*12*	*4*	*22*	*7*	$(3 + 7) mod 8 = 2$