Informatik II

Bernhard Seeger

Fachbereich Mathematik und Informatik
Universität Marburg

email: seeger@informatik.uni-marburg.de
Tel.: 28-1526
Sprechstunde: freitags, 10-12 Uhr und nach Vereinbarung

Literatur

Literatur

- Einführung in die Informatik
  - Goldschlager, Lister: “Informatik - Eine moderne Einführung”, Hanser
  - Gumm, Sommer: “Einführung in die Informatik”, Addison-Wesley
  - Rechenberg: “Was ist Informatik”, Hanser Verlag
  - Goos: “Vorlesungen über Informatik”, Band 2 (Objektorientiertes Programmieren und Algorithmen), Springer, 1996.
- Objektorientiertes Programmieren
  - Goos: “Vorlesungen über Informatik”, Band 2 (Objektorientiertes Programmieren und Algorithmen), Springer, 1996.
  - Meyer: “Object-Oriented Software Construction”, Prentice Hall, 1988.

  JAVA
  - Arnold, Gosling: “JAVA - Die Programmiersprache”, Addison-Wesley, 1996. Deutsche Übersetzung ist durchaus empfehlenswert.
  - D. Flanagan: “Java in a Nutshell”, O’Reilly, 1997, 2. Auflage (Java 1.1.x)

- Algorithmen und Datenstrukturen
  - Ottmann & Widmayer: “Algorithmen und Datenstrukturen”, Spektrum Akademischer Verlag, 1996 (3. Auflage).
  - Güting: “Datenstrukturen und Algorithmen”, Teubner, 1992.
  - Cormen, Leierson & Rivest: “Introduction to Algorithms”, MIT Press, 1990.
  - Sedgewick: “Algorithmen”, Addison-Wesley, 1992.
  - Mark A. Weiss: “Data Structures and Algorithm Analysis”, Benjamin/Cummings, 1995.
  - Knuth: “The Art of Computer Programming (Vol 1, Vol 3)”, Addison-Wesley
  - Gonnet & Baeza-Yates: “Handbook of Algorithms and Data Structures”, 2nd Edition, Addison-Wesley, 1991
  - Horowitz, Sahni & Anderson-Ford: “Grundlagen von Datenstrukturen in C”, Thompson, 1994.
  - Reingold & Hansen: “Data Structures in Pascal”, Little, Brown Comp. System Series, 1986.
- Semesterapparat in der Bibliothek
  - Kopien von für Informatik II relevanten Berichten werden in einem Ordner bereitgestellt.

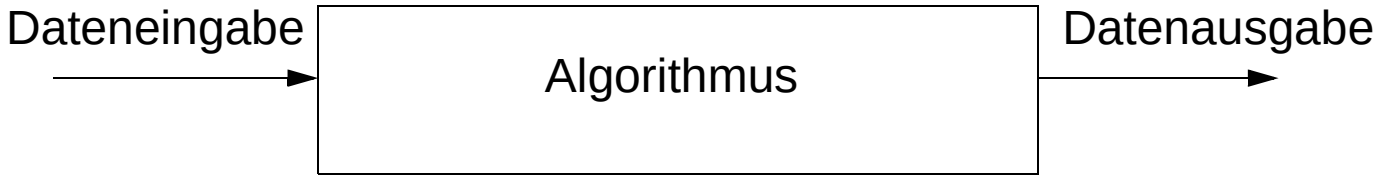
Themenübersicht

- Einführung
  - Algorithmen und ihre Analyse
  - Datentypen, Datenstrukturen und Klassen
  - Konzepte zur Implementierung von Datenstrukturen in Java: Standardtypen, Klassen, Schnittstellenklassen
- Teil 1: Datenstrukturen
  - Listen
  - Hashverfahren
  - Bäume
  - Graphen
- Teil 2: Algorithmen
  - Sortierverfahren
  - Algorithmische Methoden und Techniken

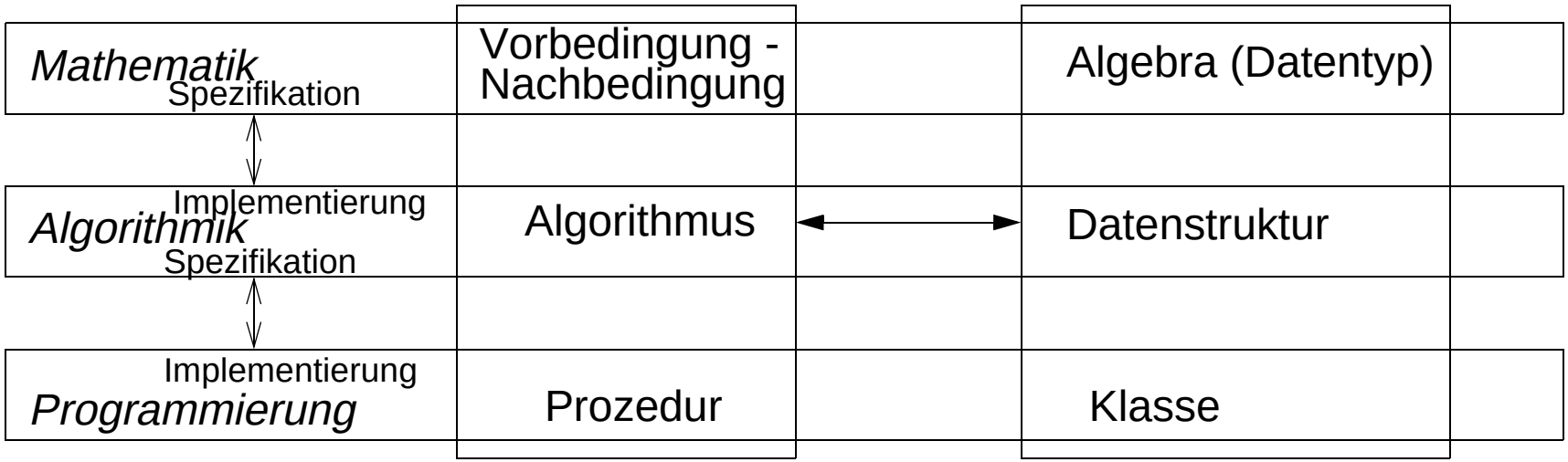
1. Einführung

Algorithmen und Datenstrukturen

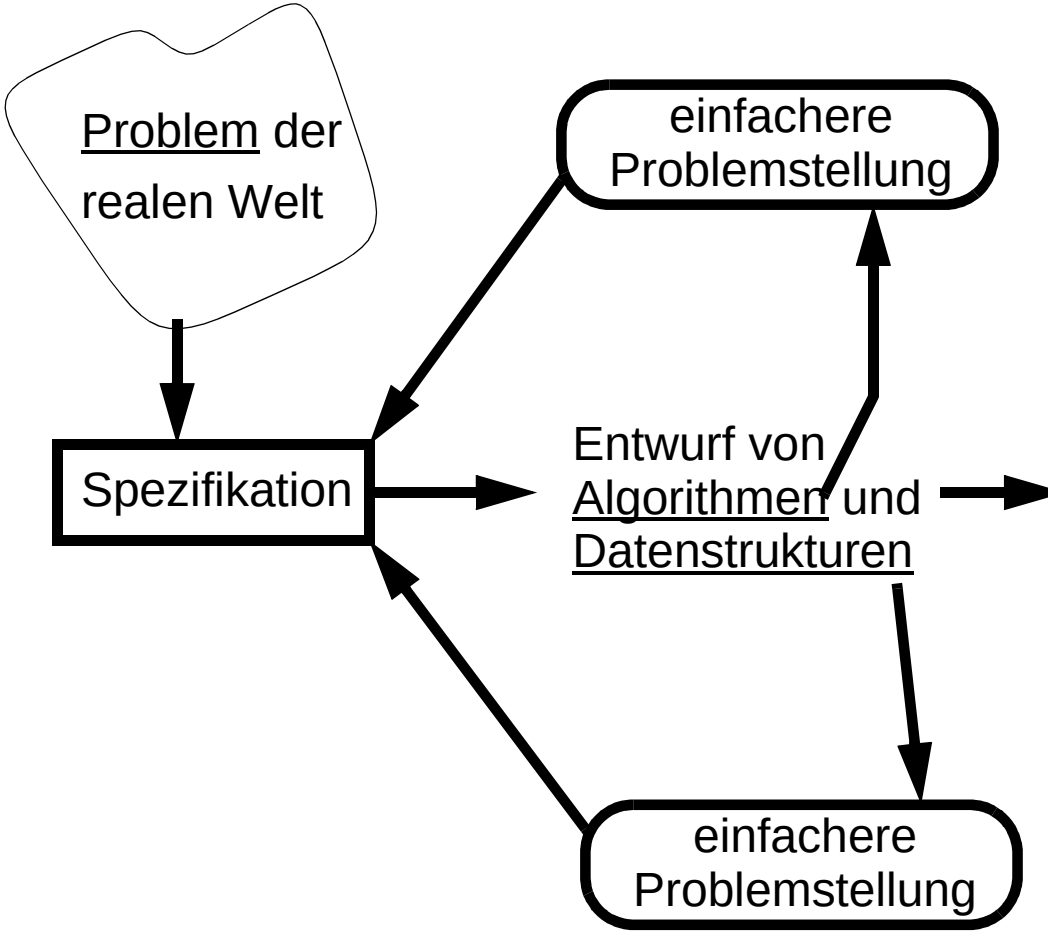
- Algorithmus ist ein Verfahren zur Lösung eines Problems
  - Spezifikation des Problems durch eine Vor- und Nachbedingung.
  - Algorithmen arbeiten auf Daten, die eingelesen und ausgegeben werden. Ein- und Ausgabe werden mittels einer Datenstruktur repräsentiert.



- Datenstruktur
  - Beschreibung der **Repräsentation** der Daten
  - Operationen (als Algorithmen gegeben) zum
    a) Erzeugung von neuen Datenobjekten
    b) Lesen des Zustands bzw. eines Teilzustands von Datenobjekten
    c) Ändern des Zustandes von Datenobjekten.



Prinzip der schrittweisen Verfeinerung



Operationen und Datenstrukturen sind elementar genug, d. h. ihre Realisierung ist “einfach”!
(Ob eine Operation einfach ist, ist vom Prozessor abhängig.)

Formulierung von Algorithmen

- Anforderung
  - Algorithmen sollen **keine** speziellen Eigenschaften von Rechnern oder von Programmiersprachen ausnutzen.
  - Ziel ist es bei der Formulierung eines Algorithmus, einem anderen **Menschen** mitzuteilen, **wie** eine Lösung zu einem vorgegebenen Problem berechnet werden kann.
  - Im Algorithmus **kann** auf Details verzichtet werden, wenn deren Realisierung dem Zielpersonenkreis offensichtlich klar ist.
- Unterschied zwischen Algorithmus und Programm
  - Ein Programm teilt einem **Rechner** mit, wie die Lösung berechnet werden soll.
  - Formulierung eines Programms erfolgt in einer konkreten **Programmiersprache**.
  - Ein Mensch kann ein Programm i. a. sehr schlecht verstehen (auch wenn z. B. im Programm die Bezeichner geeignet gewählt wurden).
  - Zu einem guten Programm gehört immer eine **Dokumentation**, wo insbesondere die Kernalgorithmen des Programms beschrieben sind.

Beispiel für einen Algorithmus in JAVA

- Problem:

  Gegeben sei eine Menge von ganzen Zahlen. Stelle fest, ob eine bestimmte Zahl in der Menge enthalten ist.

- Algorithmus:

```
boolean contains(int[] S, int c) {
// Vor- und Nachbedingung:
// Eingabe: S, ein Array mit ganzen Zahlen, und c, eine ganze Zahl
// Ausgabe: true, falls c in S, sonst false
    boolean res = false;
    for (int i = 0; i < S.length; i++) {
    // Schleifeninvariante: ??
      if (S[i] == c)
          {res = true; break;}
    }
    return res;
}
```

Analyse von Algorithmen

Korrektheit

- wichtigste formale Eigenschaft (siehe Informatik I)

Effizienz

- Kriterien
  - **Rechenzeit**
  - Speicherplatz
- erster Vorschlag zur Ermittelung der Effizienz (Rechenzeit)
  - Implementierung des Algorithmus in einer bestimmten Programmiersprache auf einem konkreten Rechner.
  - Messen der Laufzeit des Programms in Abhängigkeit der Eingabe

  Nachteil:
  - Ergebnisse aus Experimenten sind von folgenden Parametern abhängig:

    Rechnerarchitektur, Betriebssystem, Rechnerlast zur Laufzeit des Programms, Übersetzer, …

Elementaroperationen

Zweiter Vorschlag

- Zählen der benötigten **Elementaroperationen** des Algorithmus.
- Was versteht man unter einer Elementaroperation?
  - Operationen, die üblicherweise in jeder Programmiersprache vorhanden sind.
  - Beispiele für Elementaroperationen sind Zuweisungen, Vergleiche, arithmetische Operationen, Verfolgung einer Objektreferenz oder Arrayzugriffe.
  - keine Elementaroperationen sind z. B. Schleife und Prozeduraufruf. Die Realisierung dieser Kontrollstrukturen hängt wesentlich von der Programmiersprache und dem Compiler ab.
- Beispiel (Algorithmus contains):

  1. S = (1, 4, 2, 7), c = 6 $\Rightarrow$ 1 Z, 4 V, 4 A
  2. S = (2, 7, 6, 1), c = 2 $\Rightarrow$ 2 Z, 1 V, 1 A
  3. S = (1, 2, 3, 4, 5, 19, 49, 50), c = 8 $\Rightarrow$ 1 Z, 8 V, 8 A

  wobei Zuweisungen (Z), Arrayzugriffe (A) und Vergleiche (V) als Elementaroperationen betrachtet werden.

Vereinfachung der Analyse

Zählen von allen Elementaroperationen ist zu aufwendig!

1. Vereinfachungsschritt

- Es werden nur noch die zeitlich dominanten Elementaroperationen gezählt!
- Jede zeitlich dominante Operation benötigt die gleiche Ausführungszeit (= 1 E)

Sei M die Menge aller möglichen Eingaben eines Algorithmus. Durch diese Vereinfachung wird eine Funktion T definiert, die jeder Eingabe $x \in M$ eine ganze Zahl n, $n \geq 0$, zuordnet.

$$T: M \rightarrow \mathrm{IN}$$

Bemerkung:

Die Funktion T wird auch als **Kostenfunktion** bezeichnet. Analog sagen wir, daß für eine Eingabe x ein Algorithmus **Kosten** T(x) verursacht. Beim Gebrauch des Wortes “Kosten” sollte aber immer klar sein, was die zugrundeliegenden Elementaroperationen sind. Statt Kosten wird auch häufig **Laufzeit** und **Aufwand** benutzt.

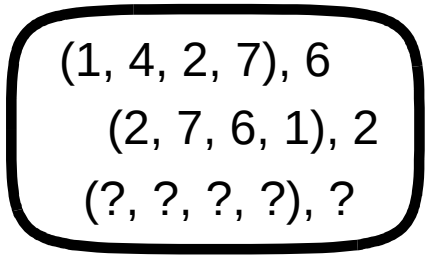
Beispiel (Algorithmus contains):

- Zuweisungen, Arrayzugriffe und Vergleiche sind zeitlich dominant.
  1. S = (1, 4, 2, 7), c = 6 ⇒ 9 E
  2. S = (2, 7, 6, 1), c = 2 ⇒ 4 E
  3. S = (1, 2, 3, 4, 5, 19, 49, 50), c = 8 ⇒ 17 E
- Vergleiche und Zuweisungen sind zeitlich dominant; Arrayzugriffe sind nicht dominant und werden deshalb beim Zählen nicht mehr berücksichtigt.
  1. S = (1, 4, 2, 7), c = 6 ⇒ 5 E
  2. S = (2, 7, 6, 1), c = 2 ⇒ 3 E
  3. S = (1, 2, 3, 4, 5, 19, 49, 50), c = 8 ⇒ 9 E

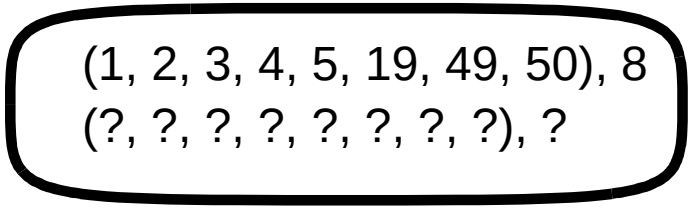
⇒ Die Kostenfunktion T hängt davon ab, welche Operationen wir als dominant erachten!

2. Vereinfachungsschritt

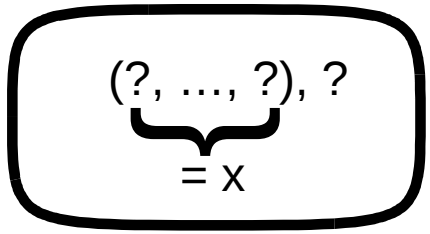
- Zusammenfassen von verschiedenen Eingaben zu Komplexitätsklassen. Eine Komplexitätsklasse K_n ist dabei durch die Größe der Eingabe (n) charakterisiert.



Klasse K_4 der
4-elementigen Arrays



Klasse K_8 der
8-elementigen Arrays



Klasse K_x der
x-elementigen Arrays

- Statt jeder möglichen Eingabe wird jetzt nur noch jeder Komplexitätsklasse Kosten zugeordnet.

- Für jede Komplexitätsklasse K_n betrachten wir folgende Spezialfälle
  - der **beste Fall** (best case): $T_{best} : \mathbf{IN} \rightarrow \mathbf{IN}$ mit

$$T_{\text{best}}(n) = \min_{x \in K_n} T(x)$$

  - der **schlimmste Fall** (worst case): $T_{worst} : \mathbf{IN} \rightarrow \mathbf{IN}$ mit

$$T_{\text{worst}}(n) = \max_{x \in K_n} T(x)$$

Beispiel (contains)

Im folgenden betrachten wir nur die Vergleiche und Zuweisungen als Elementaroperationen.

- Der beste Fall liegt vor, wenn das gesuchte Element sich in der ersten Zelle des Arrays befindet. Dies gilt für alle Komplexitätsklassen.

$$T_{best}(n) = 3$$

- Der schlechteste Fall liegt vor, wenn das gesuchte Element sich in der letzten Zelle des Arrays befindet. Dies gilt ebenfalls für alle Komplexitätsklassen.

$$T_{worst}(n) = n + 2$$

Durchschnittsverhalten

- Der schlimmste Fall bzw. der beste Fall liefern nur obere bzw. untere Schranke. In der Praxis ist man aber stattdessen am Durchschnittsverhalten (average case) innerhalb einer Komplexitätsklasse interessiert: T_{avg}: $\mathbf{IN} \rightarrow \mathbf{IR}^{+}$

Vorgehensweise

- Aufteilung der Komplexitätsklasse K_n in m disjunkte Teilklassen $K_{n,j}$, $1 \leq j \leq m$, so daß für alle $x, y \in K_{n,j}$ T(x) = T(y) gilt. $T_{n,j}$ bezeichne nun die Kosten des Algorithmus für eine Eingabe aus der Teilklasse $K_{n,j}$.
- Jeder Teilklasse $K_{n,j}$ wird ein Wert *p($K_{n,j}$) mit* $0 \leq p(K_{n,j}) \leq 1$ zugeordnet. *p($K_{n,j}$)* bezeichnet die Wahrscheinlichkeit, wie oft eine Eingabe aus K_n sich in $K_{n,j}$ befindet (relativ zu den anderen möglichen Eingaben aus K_n).
  - Für die Funktion *p* gilt stets $\sum_{1 \leq j \leq m} p(K_{n,j}) = 1$.
- Die durchschnittlichen Kosten sind dann folgendermaßen definiert

$$T_{\text{avg}}(n) = \sum_{1 \leq j \leq m} T_{n,j} \cdot p(K_{n,j})$$

Beispiel (contains)

- Annahmen
  a) Alle Werte im Array sind verschieden.
  b) Das gesuchte Element x liegt mit Wahrscheinlichkeit 1/n in der j-ten Zelle eines n-elementigen Arrays, $1 \le j \le n$.
- Aufteilung der Komplexitätsklasse K_n in n disjunkte Teilklassen $K_{n,j}$, $1 \le j \le n$.
  – $K_{n,j}$ beinhaltet die Eingaben, bei welchen x in der j-ten Zelle des Arrays liegt. Es gilt dann $T_{n,j} = 2 + j$.
- Aufsummieren

$$
\begin{aligned}
T_{\text{avg}}(n) &= \sum_{1 \le j \le m} T_{n,j} \cdot p(K_{n,j}) \\
&= \sum_{1 \le j \le n} (2 + j) \cdot \frac{1}{n} \\
&= 2 + \frac{1}{n} \sum_{1 \le j \le n} j \qquad = \qquad 2 + \frac{n+1}{2}
\end{aligned}
$$

Dritter Vereinfachungsschritt

- Weglassen von multiplikativen Konstanten und unwichtigen additiven Termen in den Formeln von T_{worst}, T_{best} und T_{avg}.

Motivation:

- Trotz der bereits gemachten Vereinfachung ist eine genaue Analyse von Algorithmen oft sehr schwierig.
- Berechnung der genauen Kosten ist für interessante Fragestellungen nicht immer notwendig:
  - Wie hoch sind die Kosten eines Algorithmus der Komplexitätsklasse K_{2n} in Relation zu den Kosten der Komplexitätsklasse K_n?

    Beispiel: $T_1(n) = 10 \cdot n + \lceil \log_2 n \rceil$

    $\Rightarrow$ Konstanten sind bei dieser Fragestellung nicht relevant.

    $\Rightarrow$ unwichtige Terme wie $\log_2$ n (<< n) haben kaum Relevanz.
  - Sind die Kosten eines Algorithmus A niedriger als die eines Algorithmus B?
  - Ist für “große n” $\log_2$ n eine obere Schranke von T_{worst} (n)?

Die O-Notation

Definition: *O-Notation*

Seien $f: \mathrm{IN} \to \mathrm{IR}^+$ und $g: \mathrm{IN} \to \mathrm{IR}^+$.

$f = O(g) \quad \Leftrightarrow \quad \exists\, n_0 \in \mathrm{IN},\, c \in \mathrm{IR}^+$ mit $\forall\, n \geq n_0$ ist $f(n) \leq c \cdot g(n)$.

Man sagt auch: *f* wächst asymptotisch höchstens so schnell wie *g*.

Anmerkung:

Da *O(g)* genau genommen eine (unendliche) Menge von Funktionen beschreibt, ist es genauer, zu sagen: $f \in O(g)$. In der Literatur hat sich jedoch das '=' eingebürgert. Wichtig ist darauf zu achten, daß insbesondere die Kommutativität des '='-Operators hier **nicht** gilt, d. h.

$$f = O(g) \not\Rightarrow g = O(f)$$

Beispiele:

- $T_1(n) = n + 3 = O(n)$
- $T_2(n) = 1000n + 7 = O(n)$
- $T_3(n) = 4711\ n^2 + 20000\ n + 1 = O(n^2)$
- $T_3(n) = O(n^3)$

Da die Kostenfunktionen i. a. überall von 0 verschieden und monoton wachsend sind, geht man bei der Überprüfung von f = O(g) oft folgendermaßen vor:

– Betrachte den Grenzwert: $\lim_{n \to \infty} \frac{f(n)}{g(n)}$. Falls dieser existiert, so ist $f = O(g)$.

Beispiel:

$$\lim_{n \to \infty} \frac{T_3(n)}{n^2} = \lim_{n \to \infty} \frac{4711n^2 + 20000n + 1}{n^2} = 4711$$

Achtung: Man beachte die Richtung der Implikation! Aus $f = O(g)$ folgt also **nicht**, daß der Grenzwert existiert!

Allgemeine Regeln für die Analyse

Schleifen

- Sei O(g(n)) eine obere Schranke für die Kosten eines Schleifendurchlaufes und sei O(f(n)) eine obere Schranke für die Anzahl der Schleifendurchläufe. Dann ist

$$T(n) = O(f(n) \cdot g(n))$$

- Beispiel:

  Der folgende Teil eines Programms besitzt eine Laufzeit $O(n^2)$

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)        // Laufzeit der Schleife beträgt O(n)
        k = k + 1;
```

- Schwieriger wird es, wenn die Laufvariablen voneinander abhängen.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)        // Laufzeit der Schleife beträgt O(i)
        k = k + 1;
```

Sequenz

- Seien S_1 und S_2 zwei Teile (Sequenzen) eines Algorithmus mit Kosten $T_1(n) = O(f(n))$ und $T_2(n) = O(g(n))$. Dann gilt für die Laufzeit:

  $$T(n) = O(\max(f(n), g(n)))$$

- Beispiel:

```
for (i = 0; i < n; i++)                 // Laufzeit: O(n)
    a[i] = 0;

for (i = 0; i < n; i++)                 // Laufzeit O(n^2)
    for (j = 0; j < n; j++)
            a[i] = a[i] + a[j] + i + j;
```

Bedingte Anweisung

- Gegeben ist eine bedingte Anweisung “if (B) S1 else S2”. Dann beträgt die Laufzeit

  $$T(n) = O(h(n) + \max(f(n), g(n)))$$

  unter der Annahme, daß h(n) der Aufwand für die Auswertung von B ist.

Beispiel (contains):

- $T_{best}(n) =$ O(1)
- $T_{worst}(n) =$ O(n)
- $T_{avg}(n) =$ O(n)

Definition:

(i) $f = \Omega(g)$, falls $g = O(f)$; f wächst mindestens so schnell wie g.
(ii) $f = \Theta(g)$, falls $f = O(g)$ und $g = O(f)$; f und g wachsen mit gleicher Ordnung.

- Die Ω-Notation wird häufig dazu benutzt, untere Schranken für die Kosten aller Algorithmen zur Lösung eines Problems anzugeben. Wir bezeichnen die größte dieser unteren Schranken als die **Komplexität des Problems**.
- Ein Algorithmus heißt **asymptotisch optimal**, wenn seine worst-case Kosten mit der Komplexität des Problems zusammenfällt.

Beispiel

- Betrachten wir das Problem, das größte Element in einem Feld mit n ganzen Zahlen zu finden. Wir nehmen weiterhin an, daß die Elemente **nicht** werteabhängig geordnet sind (z. B. aufsteigend sortiert).

  Die untere Schranke für dieses Problem beträgt dann $f = \Omega(n)$.

  Ein Algorithmus, der mit weniger Vergleiche auskommt, würde gewisse Elemente überhaupt nicht in Betracht ziehen. Wenn eins dieser Elemente aber gerade den maximalen Wert im Feld besitzt, arbeitet dieser Algorithmus nicht korekt!

- I. a. ist die Berechnung der Komplexität einer Problemstellung nicht so einfach wie die Analyse eines Algorithmus. Oft kennt man nur obere und untere Schranken eines Problems.

- wichtige Funktionen zur Messung der Kosten eines Algorithmus in Abhängigkeit der Problemgröße N:
  - logarithmisches Wachstum: $\log N$
  - lineares Wachstum: N
  - N-logN-Wachstum: $N \log N$
  - quadratisches Wachstum: N^2
  - …
  - exponentielles Wachstum: 2^N
- **wichtige Anmerkung** für die praktische Beurteilung von Algorithmen
  - Algorithmen mit einer größeren Laufzeit können für “kleine” Eingaben schneller sein als Algorithmen mit niedriger Laufzeit.
  - für große Eingabemengen eignen sich praktisch nur Algorithmen mit Laufzeit $O(n)$ oder $O(n \log n)$.

1.2 Analyse rekursiver Algorithmen

- Modifikation der Problemlösung für die Suche in einem Array durch Wahl einer anderen Datenstruktur:
  - Elemente des Arrays werden aufsteigend sortiert abgespeichert, d. h.

    für alle i, j mit i < j gilt: S[i] < S[j]
- Damit können wir folgenden Algorithmus für die Suche verwenden:

```
Algorithmus contains2(int[] S, int low, int high, int c)
// Eingabe:    S und c wie bei Algorithmus contains
//             low und high sind die untere und obere Grenze des zu durch
//             suchenden Bereichs im Array.
// Ausgabe:    true, falls c in S[low], S[low+1], …, S[high] vorkommt. Andernfalls, false.
    if (low < high)return false;
    m = (low + high)/2;
    if (S[m] == c) return true;
    if (S[m] < c)
        return contains2(S, m+1, high, c);
    else
        return contains2(S, low, m+1, c);
```

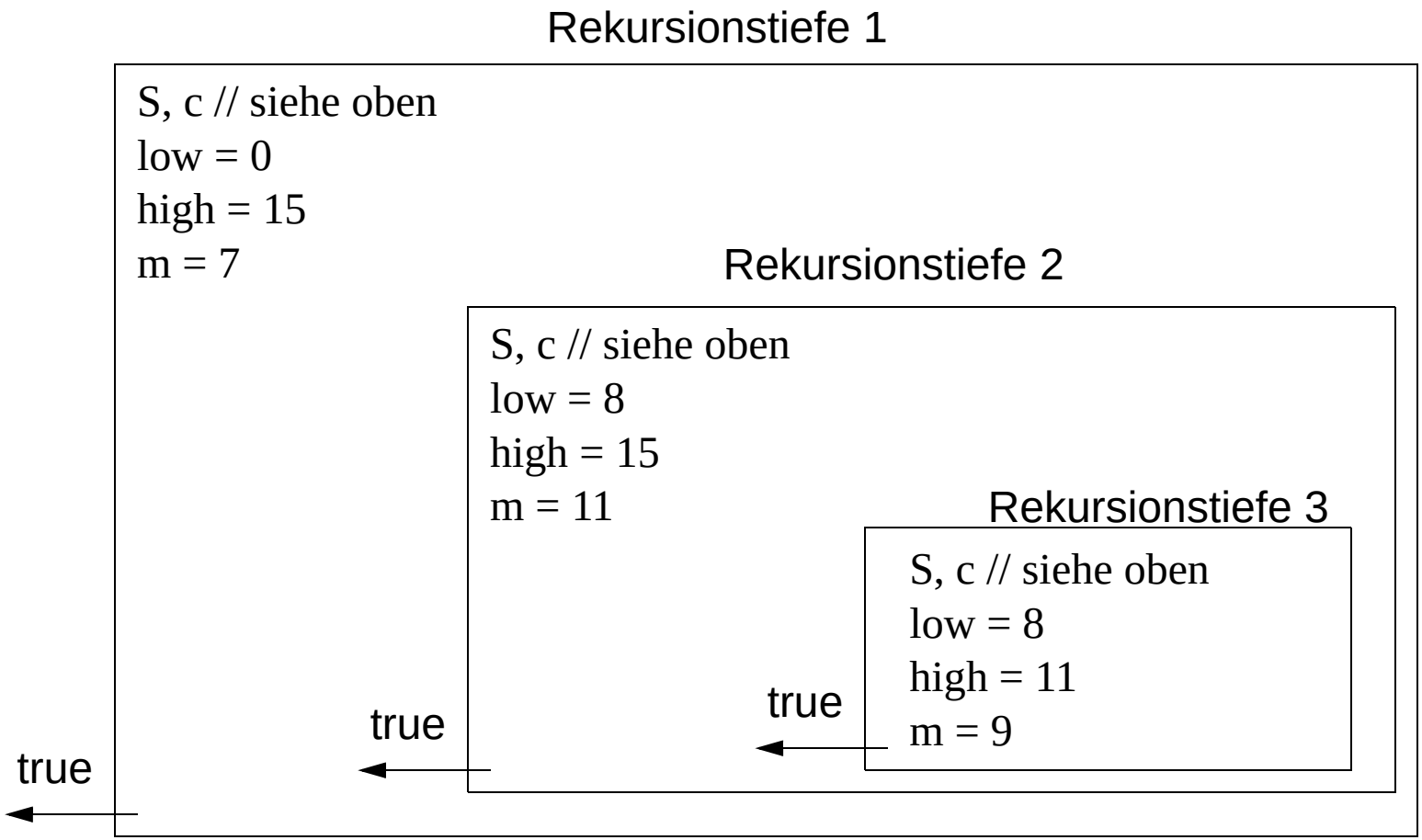
- erster Aufruf des Algorithmus: contains2(S, 0, S.length-1, c)

Beispiel

S =

3	5	7	8	9	11	13	17	18	19	21	23	24	27	30	31

c = 19



Analyse

- Laufzeit eines rekursiven Algorithmus ist durch eine rekursive Gleichung gegeben. Für den **schlimmsten Fall** erhalten wir bei contains2 folgende Gleichung:

  $T_{worst}(n) = 1 + T_{worst}(n/2 - 1)$ falls n (= high - low + 1) > 0

  $T_{worst}(0) = 0$ sonst (n = 0)

  wobei wir vereinfachend angenommen haben:

  – n (Anzahl der für die Suche relevanten Elemente im Array) = $2^k - 1$
  – Elementaroperationen sind nur Schlüsselvergleiche “<“.

- Verallgemeinerung der Rekursionsgleichung:

  $T(0) = a$ und $T(n) = b + T((n-1)/2)$

- Lösung der Rekursionsgleichung durch **iterative Substitution**

  $T(n) = b + T((n-1)/2) = b + T(2^{k-1} - 1)$

  $= b + b + T((2^{k-1} - 1 - 1)/2)) = 2b + T(2^{k-2} - 1)$

  $= \ldots$

  $= b + \ldots + b + T(0) = k*b + T(0) = kb + a$

  $= \log_2 (n + 1)*b + a$

Die Laufzeit von contains2 ist

- im schlimmsten Fall:

$$T_{worst}(n) = \log_2 (n + 1) = O(\log_2 n)$$

- im besten Fall:

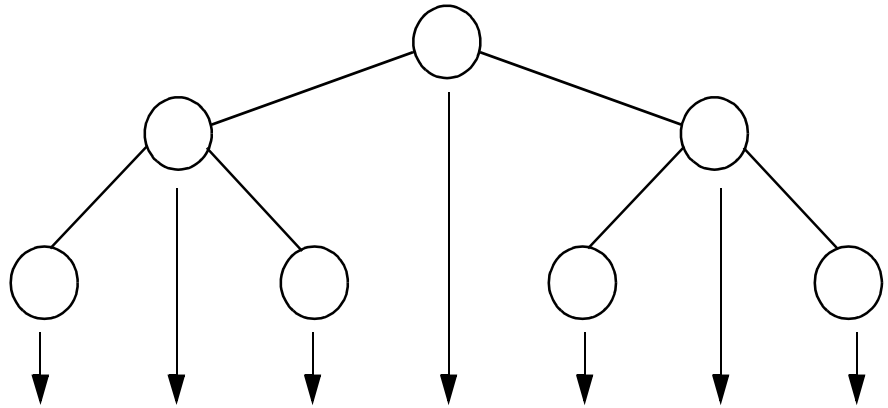
$$T_{best}(n) = 1 = O(1)$$

Durchschnittliche Laufzeit von contains2

- Folgende Annahme werden getroffen:
  - Aufwand pro Rekursionsschritt: 1 Vergleich.
  - Jede Suche endet erfolgreich.
  - Das gesuchte Element x liegt mit Wahrscheinlichkeit 1/n in der j-ten Zelle des n-elementigen Arrays, $0 \leq j < n$.
- Aufteilung der Komplexitätsklasse K_n in n disjunkte Teilklassen $K_{n,j}$, $0 \leq j < n$.
  - $K_{n,j}$ beinhaltet die Eingaben, bei welchen x in der j-ten Zelle des Arrays liegt.
- vereinfachende Annahme: $n = 2^k-1$

Veranschaulichung von $T_{n,j}$, j = 0,…,6

Position im Array	0	1	2	3	4	5	6
$T_{n,j}$	3	2	3	1	3	2	3



$$T_{\text{avg}}(n) = \sum_{0 \le j < n} T_{n,j} \cdot \frac{1}{n} = \frac{1}{n} \sum_{i=0}^{k-1} (i+1) \cdot 2^i$$

$$= \frac{1}{n}((k-1) \cdot 2^k + 1) = \frac{1}{n}((n+1) \cdot \log_2(n+1) - n)$$

$$\approx \log_2(n+1) - 1$$

$$= \Theta(\log_2 n)$$

1.2 Datentyp, Datenstruktur und Klasse

- Entscheidend für die Laufzeit eines Algorithmus ist eine geeignete Organisation der Daten mittels einer effizienten Datenstruktur.
- Entsprechend wie bei Algorithmen unterscheiden wir zwischen
  - **Spezifikationsebene**:
    Ein Datentyp besteht aus einer speziellen Sorte, der Signatur, der Zuordnung von Wertebereichen zu den Sorten und der Spezifikation der Operationen.
  - **Algorithmische Ebene**:
    Die zu dem Datentyp zugehörige Datenstruktur legt die Repräsentation der Objekte fest. Weiterhin wird für jede Operation des Datentyps ein Algorithmus angegeben.
  - **Programmierungsebene**:
    Die Datenstruktur wird nun in einer konkreten Programmiersprache (in unserem Fall Java) implementiert. Die Datenstruktur wird in eine Klasse umgesetzt.
- Beispiel für folgendes Problem:
  Implementiere eine Klasse Intset in Java. Objekte der Klasse sollen Mengen ganzer Zahlen repräsentieren und folgende Operationen anbieten: Einfügen und Löschen von Zahlen und Test auf Enthaltensein.

1. Schritt: Erstellung eines Datentyps

relevante Fragen auf der Spezifikationsebene

- Welche Werte sollen repräsentiert werden?
- Welche Operationen sollen unterstützt werden? **Was** sollen die Operationen leisten?

Antwort in einer formalen Form:

Datentyp Intset

Sorten Intset, int, boolean;

Operationen	empty		→ intset
	insert:	intset x int	→ intset
	delete:	intset x int	→ intset
	contains:	intset x int	→ boolean
	isempty:	intset	→ boolean
Wertebereiche	Intset:	alle endlichen Mengen mit ganzen Zahlen	
	int:	Menge der ganzen Zahlen	
	boolean:	{true, false}	

- Man beachte, daß bei der Spezifikation des Datentyps alle Operationen Funktionen sind. Man spricht dann auch von einer Algebra.
- Vorteil einer Algebra ist, daß die Spezifikation der Operationen sehr einfach ist. Es muß nämlich nur beschrieben werden, **was** das Resultat der Operation ist. Eine Beschreibung, **wie** dieses Resultat berechnet wird, ist **nicht** erforderlich!

Resultat		
	empty	$= \emptyset$
	insert(M,i)	$= M \cup \{ i \}$
	delete(M,i)	$= M - \{ i \}$
	contains(M,i)	$= \begin{cases} \text{true} & \text{falls } i \in M \\ \text{false} & \text{sonst} \end{cases}$
	isempty	$= (M = \emptyset)$

- Die parameterlose Funktion empty nimmt eine Sonderrolle ein. Sie dient später zur Erzeugung neuer Objekte (der Klasse Intset).

2. Schritt: Festlegen der Datenstruktur

Fragen:

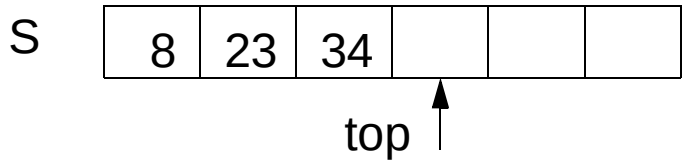
- Wie sollen die Objekte der Sorte Intset repräsentiert werden?
- Wie können die in der Datenstruktur aufgeführten Operation realisiert werden (Angabe eines Algorithmus)?

Antworten

- Repräsentation:
  - Zur Darstellung von Mengen ganzer Zahlen entscheiden wir uns dazu, ein Array zu benutzen, in dem die Zahlen aufsteigend sortiert und nacheinander (beginnend vom Index 0) gespeichert werden.
  - Zusätzlich verwenden wir einen Zähler für die Anzahl der mit Zahlen belegten Zellen des Arrays.

in der Notation von Java:

```
int top;
int[] S;
```



Formulierung der Algorithmen

- Wir betrachten bei den folgenden Algorithmen, S und top als globale Variablen. Dies ist bereits eine erster Schritt in Richtung Objektorientierung.

```
Algorithmus empty
    top = 0;

Algorithmus insert(x)
    Bestimme den Index j des ersten Elements mit S[j] >= x;
    if (S[j] != x) {
        schiebe alle Zahlen ab dem Index j um eine Position nach rechts;
        S[j] = x;
    }

Algorithmus delete(x)
    Bestimme den Index j von Element x.
    Falls x in S gefunden,
        schiebe alle Elemente ab Position j+1 um eine Position nach links;
```

Analyse einer Datenstruktur

- Ähnlich wie bei Algorithmen ist es bei der Analyse einer Datenstruktur **nicht** erforderlich, daß diese bereits z. B. als Klasse implementiert ist.
- Analyse einer Datenstruktur entspricht im wesentlichen der Analyse der Algorithmen, die zur Implementierung herangezogen werden.
- Beispiel (Intset):
  Im schlimmsten Fall entstehen bei den Algorithmen folgende Ausführungskosten:

empty	O(1)	
insert	O(n)	
delete	O(n)	
contains	O(log n)	(nur für den Fall, daß binäre Suche verwendet wird)
isempty	O(1)	

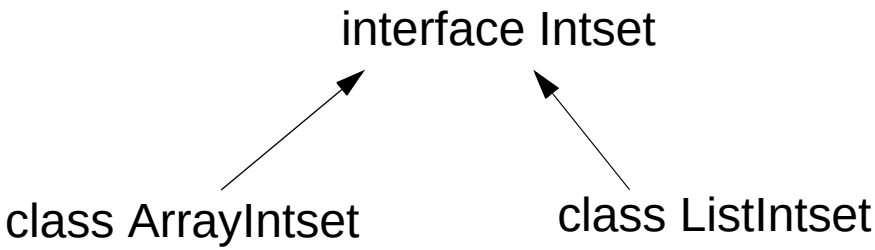
- Speicherplatzbedarf: O(n)

Amortisierte Analyse

- Bei der Analyse von Datenstrukturen ist man auch an der Laufzeit (i. a. im schlimmsten Fall) einer Folge von Operationen interessiert. Diese Laufzeit wird dann wieder einfach zurückgerechnet auf jede Operation. Man spricht dann auch von den **amortisierten Kosten**.
- Da diese Art der Analyse Kenntnisse im Bereich Datenstrukturen voraussetzt, soll das Prinzip an einem Alltagsproblem veranschaulicht werden.
  - Datenstruktur: Lebensmittelgeschäft mit der Operation Kaufe(Artikel x).
  - Die Operation Kaufe gibt uns zu einem Artikel die gewünschte Ware
  - Die Kosten im schlimmsten Fall, um diese Operationen **einmal** auszuführen, beträgt 15 Minuten.
  - Die Kosten im schlimmsten Fall, um diese Operation **zweimal** auszuführen, beträgt 16 Minuten, d h. die amortisierten Kosten betragen somit 8 Minuten.
  - …
  -

3. Schritt: Implementierung in Java

- Eine Datenstruktur wird in Java durch eine Klasse implementiert.
- Zusätzlich besteht in Java die Möglichkeit durch eine sogenannte Schnittstellenklasse (*Interface*) einen Datentyp zu spezifizieren.
  - alle Methoden einer Schnittstellenklasse besitzen **keine** Implementierung. In Java spricht man dann auch von *abstrakten* Methoden.
  - Schnittstellenklassen können als Oberklasse von anderen Klassen benutzt werden. Die von der Schnittstellenklasse abgeleiteten Klassen müssen **jede** Methode der Schnittstellenklasse implementieren.
  - Schnittstellenklassen können mehrere Unterklassen besitzen, die jeweils eine andere Implementierung der gewünschten Funktionalität liefern.



Die Schnittstellenklasse des Datentyps Intset

- Eine Schnittstellenklasse entspricht einer “Klasse”, die nur aus abstrakten Methoden (ohne Rumpf) und Konstanten besteht.
  - Es können keine Objekte von dieser Klasse erzeugt werden. Deshalb macht es keinen Sinn (und ist auch nicht erlaubt), Konstruktoren zu definieren!
  - ABER: Der Typ einer Objektvariable kann eine Schnittstellenklasse sein.
- Vorschlag für Intset

```
public interface Intset{
    // empty ist ein Konstruktor (siehe Implementierung von ArrayIntset)
    void insert(int x);
    void delete(int x);
    boolean isempty();
    boolean contains(int x);
}
```

- Vorteil von Schnittstellen
  - Schnittstellen können von mehreren Klassen implementiert werden.
  - Je nach Anwendung kann eine Objektvariable der Schnittstellenklasse ein Objekt jener Klassen zugewiesen bekommen, welche die Schnittstelle implementiert haben.

Anwendung

- Folgendes Java-Programm verwendet nun eine Objektvariable der Schnittstellenklasse Intset, die während der Laufzeit des Programms Objekte der Klasse ArrayIntset bzw. ListIntset zugewiesen bekommt.

```
public static void main(String[] args) throws IOException {
        Intset S;
        Random r = new Random();
        if (args.length == 0)
                S = new ArrayIntset();
        else {
                if args[0].equals("list")
                        S = new ListIntset();
                else
                        S = new ArrayIntset();
        }
        for (int i = 0; i < 20; i++)
                S.insert(r.nextInt() % 1000);
        System.in.read();
}
```

Deklaration einer Objektvariablen der Schnittstellenklasse Intset.

Objektvariable bekommt ein neues Objekt der Klasse ArrayIntset zugewiesen.

Objektvariable bekommt ein neues Objekt der Klasse ListIntset zugewiesen.

Die Klasse des Objekts, auf das die Objektvariable verweist, bestimmt, welche Implementierung benutzt wird.

Implementierung der Klasse ArrayIntset

```
public class ArrayIntset implements Intset {
    private int top;
    // private schützt die Daten vor unerlaubtem Zugriff von außen
    private int[] S;

    private int ssize = 16;// Startgröße des Arrays

    public void insert(int x) {
        int[] newS;
        int pos = globalfind(x);
        if (pos >= 0) { // x nicht gefunden
            if (top == S.length) {
                newS = new int[2*top]; // neues Array
                for (int i = 0; i < top; i++)
                    newS[i] = S[i];
                S = newS;
            }
            shiftright(pos);
            S[pos] = x;
        }
    } // end of insert

    // Implementierung der anderen Methoden der Klasse
}
```

Der Zusatz “implements Intset” zeigt an, daß diese Klasse alle Methoden der Schnittstellenklasse Intset implemntiert.

Beim Algorithmus haben wir uns über die Verwaltung des Arrays bisher nicht gekümmert. Die Lösung hängt ganz wesentlich von der Programmiersprache ab.

Die Methode insert soll außerhalb der Klasse genutzt werden. Deshalb ist sie als public deklariert.

globalfind sucht nach dem Wert x in S. Falls dieser gefunden wurde, wird eine negative Zahl zurückggegeben.

Überlaufbehandlung, d. h. das Array ist bereits voll besetzt und wir wollen noch einen neuen Wert einfügen.

shiftright verschiebt die Elemente ab Position pos um 1 nach rechts.

```
public class ArrayIntset implements Intset {
    …
    // Implementierung der anderen Methoden der Klasse
    private int globalfind(int x) {
      int pos = find(x, 0, top-1);
      if ((pos == top) || (S[pos] != x))
          return pos;
      else
          return -1;
    }

    private int find(int x, int low, int high) {
    // Sucht in S zwischen Position low und high nach dem Wert x
    // Als Resultat wird der kleinste Index j mit S[j] >= x zurueckggeben.
            if (low > high) return high + 1;
            int m = (low + high)/2;
            if (S[m] == x)
                    return m;
            else {
                    if (S[m] > x)
                            return find(x, low, m-1);
                    else
                            return find(x, m+1, high);
            }
    }
}
```

Die Methode globalfind dient nur zur Implementierung anderer Methoden innerhalb der Klasse. Deshalb wird sie als privat deklariert. Sie kann damit nicht außerhalb der Klasse verwendet werden.

Methode find implementiert eine binäre Suche in S, wobei nur der Teil des Array zwischen dem Index low und dem Index high berücksichtigt werden. Falls x gefunden wurde, wird die entsprechende Position zurückgegeben.

Rekursiver Aufruf der Methode find.

- Entsprechend zu Insert müssen nun alle anderen Methoden implementiert werden. Ein Spezialfall stellt die Methode “empty” dar, da sie als ein Konstruktor in Java implementiert wird. Der Bezeichner eines Konstruktors muß aber in Java mit dem Klassennamen identisch sein.

```
public class ArrayIntset implements Intset{
…
ArrayIntset(){ // implementiert empty
        top = 0;
        S = new int[ssize];
    }
…
}
```

Testen der Klassen

- Wichtiger Aspekt bereits während der Implementierung:
  Testen, ob die Prozeduren tatsächlich die Spezifikation der Algorithmen erfüllen.
  - Erstellen von klasseninternen (d. h. mit privat deklarierten) Testprozeduren
  - Es ist vorteilhaft, daß die Implementierung und das Testen einer Datenstruktur von verschiedenen Personen durchgeführt werden.
- Validierung der Analyse von den Algorithmen
  - Dürfen die Konstanten für typische Problemgrößen tatsächlich vernachlässigt werden?

  Gegebenenfalls:
  - Entwurf eines neuen Algorithmus
  - Entwurf einer neuen Datenstruktur