

Algorithmen, Datenstrukturen und Programmierung

Hannes Federrath



Webseite und Literaturempfehlungen

⌘ Webseite

⊗ <http://www-sec.uni-regensburg.de/inf/>

⌘ Materialien in VUR

⊗ Übungsblätter und Aufgabensammlung

⊗ <http://vur.uni-regensburg.de>

⊗ Bitte die Veranstaltung abonnieren

⌘ Vorlesungsmanuskripte

⊗ Hannes Federrath:
Algorithmen, Datenstrukturen und Programmierung.

⌘ Bücher

⊗ Robert Sedgewick: Algorithmen in Java. Addison-Wesley, Pearson-Studium, 2003. ISBN 3-8273-7072-8

⊗ Robert Sedgewick: Algorithmen. Addison-Wesley, Pearson-Studium, 2002. ISBN 3-8273-7032-9

⊗ Gunter Saake, Kai-Uwe Sattler: Algorithmen und Datenstrukturen. Eine Einführung mit Java. dpunkt.verlag, 2004. ISBN 3-89864-255-0

⊗ Christian Ullenboom: Java ist auch eine Insel, 7.Auflage
<http://www.galileocomputing.de/openbook/javainsel7/index.htm>

⌘ Übung

- ⊗ CIP SG 1
- ⊗ 3 Gruppen: Mi 12-14, Mi 14-16, Do 8-10
- ⊗ Aufgabensammlung
- ⊗ Theoretische und praktische Aufgaben

⌘ Leistungserbringung

- ⊗ Studienbegleitende Leistung in Form eines Kurztests im Rahmen der Vorlesung (25 % der Note)
- ⊗ 60-minütige Open Book Klausur am Ende des Semesters (75 % der Note)

Ziele der Veranstaltung

- ⌘ Wichtige Algorithmen verstehen und anwenden
 - ⊗ z.B. Sortieren, Suchen, Syntaxanalyse
- ⌘ Wichtige Datenstrukturen verstehen und anwenden
 - ⊗ z.B. Arrays, Bäume, Stapel
- ⌘ Entwurfsprinzipien für Algorithmen verstehen und anwenden
 - ⊗ z.B. Teile und Herrsche, Backtracking, Rekursion
- ⌘ Effizienz von Algorithmen analysieren und beurteilen können
 - ⊗ z.B. Laufzeitanalyse, O-Notation

- ⌘ Problemstellungen abstrahieren und analysieren, geeignete algorithmische Lösungen erarbeiten und implementieren

Klassischer Algorithmusbegriff

- ⌘ Der klassische Algorithmusbegriff abstrahiert von Rechnern und Programmiersprachen und ist imperativ.
- ⌘ (Imperativer) Algorithmus:
 - ⊗ Vorschrift zur Lösung einer Klasse gleichartiger Probleme, bestehend aus Einzelschritten.
- ⌘ Eigenschaften:
 - ⊗ Jeder Einzelschritt ist für die ausführende Instanz unmittelbar verständlich und ausführbar.
 - ⊗ Das Verfahren ist endlich beschreibbar.
- ⌘ Praxisanforderungen:
 - ⊗ Das Verfahren benötigt eine endliche Zeit; der Algorithmus terminiert.
 - ⊗ Das Problem lässt sich mit endlichem Speicherplatzaufwand lösen.

⌘ Komplexität

- ⊗ Der nötige Aufwand an Betriebsmitteln, hier insbesondere Zeit (Laufzeit) und Raum (Speicherplatz), eines Algorithmus in Abhängigkeit von der Problemgröße wird abstrakt als Komplexität des Algorithmus bezeichnet.

⌘ Anforderungen an Algorithmen sind:

- ⊗ Sicherheit,
- ⊗ Portabilität,
- ⊗ Interoperabilität,
- ⊗ Testbarkeit,
- ⊗ leichte Änderbarkeit und Erweiterbarkeit sowie
- ⊗ Wiederverwendbarkeit und
- ⊗ Dokumentation.

⌘ Man unterscheidet:

- ⊗ Sequenzielle (sequential) und nichtsequenzielle (concurrent, nebenläufige) Algorithmen,
- ⊗ Deterministische und nicht-deterministische Algorithmen

Algorithmen in der alltäglichen Erfahrungswelt

- ⌘ Kochrezepte
- ⌘ Stricken
- ⌘ Bauanleitungen
- ⌘ Spielen eines Musikstücks nach Noten
- ⌘ ...

Algorithmus	Prozess	Einzelschritt
Rezept	ein Wiener Schnitzel backen	Schnitzel durch Ei ziehen; Schnitzel in Bröseln wenden; in Fett herausbacken
Notenblatt	Sonate spielen	Ton bestimmter Höhe und Länge spielen
Strickmuster	Pulli stricken	Einzelmasche (bestimmter Farbe) stricken

Wie viele F sehen Sie?

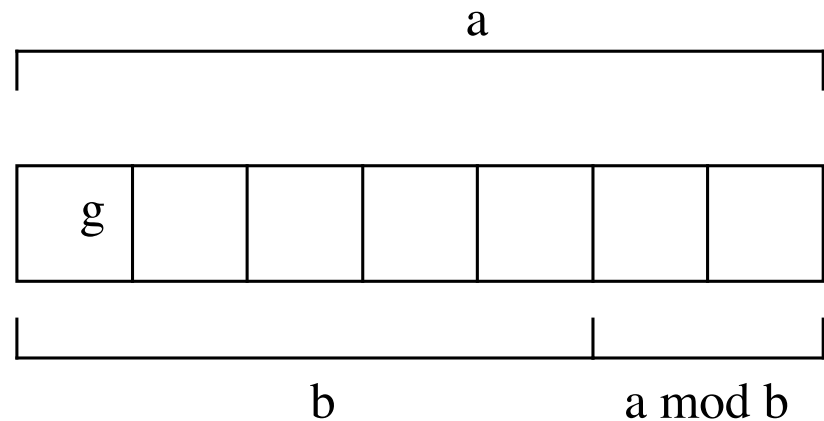
FINISHED FILES ARE THE RESULT OF YEARS OF SCIENTIFIC STUDY
COMBINED WITH THE EXPERIENCE OF YEARS.

Beispiel: Algorithmus $\text{ggt}(a,b)$

Wh.

- ⌘ Algorithmus $\text{ggt}(a,b)$ zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen a und b .

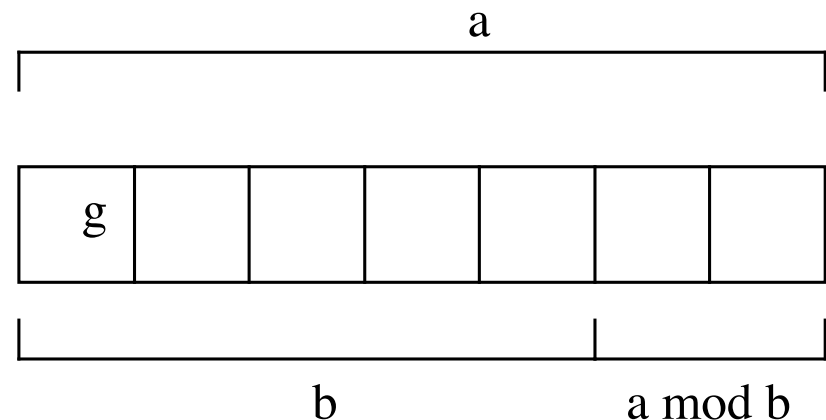
```
int getGGTOf(int a, int b) {  
    // requires ((a > 0) && (b > 0)); ensures return > 0;  
    int h;  
    while (b != 0) {  
        h = b;  
        b = a % b; // % is the modulo operator  
        a = h;  
    }  
    return a;  
}
```



Beispiel: Algorithmus $\text{ggt}(a,b)$

Wh.

- ⌘ Wenn $a < b$ ist, vertauscht der Algorithmus im ersten Durchlauf die beiden Zahlen. Angenommen, die Zahl a ist größer als b und g ist der $\text{ggT}(a,b)$. Dann lässt sich nachvollziehen, dass g auch $\text{ggT}(a \bmod b, b)$ ist.
- ⌘ Damit wird die Bildung des $\text{ggT}(a,b)$ auf die Berechnung von $\text{ggt}(a \bmod b, b)$ zurückgeführt. Durch Iteration erhält man immer kleinere Zahlenpaare und das Verfahren terminiert, wenn der Teilerrest, gleich 0 ist. Die andere Zahl a ist dann der $\text{ggt}(a,b)$.



Beispiel: getFactorialOf(int n)

Wh.

⌘ Algorithmus zur Berechnung der Fakultät $n!$ einer natürlichen Zahl n .

```
int getFactorialOf(int n) {
    int fact = 1;
    for (int i=1; i<=n; ++i) {
        fact = fact * i;
    }
    return fact;
}
```

In einer Schleife von $i=1$ bis n wird i mit dem derzeitigen Wert von `fact` multipliziert.

- ⌘ Um die Ausführung eines Algorithmus steuern zu können, sind Kontrollstrukturen erforderlich, insbesondere:
- ⊗ **Sequenz** (Abfolge von Einzelschritten) – "normale" Abarbeitung aufeinander folgender Anweisungen (z. B. ein einzelner Rechenschritt, eine Zuweisung, allg. die Auswertung eines Ausdrucks) in einem Programm
 - ⊗ **Selektion** (Auswahl unter Alternativen) – Bedingungen und Fallunterscheidungen in einem Programm
 - ⊗ **Iteration** (Wiederholung von Anweisungen, Schleifen, verschiedene Formen: Zählschleifen, bedingte Schleifen)

Darstellung von Algorithmen

- ⌘ umgangssprachliche Darstellung (Beschreibung als Text)
 - ⊗ beim Entwurf
 - ⊗ zur Einführung in die Lösungsstrategie
 - ⊗ i. d. R. nicht formal genug
- ⌘ Pseudocode: formalisierte Darstellung unabhängig von einer konkreten Programmiersprache
- ⌘ graphische Darstellung
 - ⊗ Programmablaufplan
 - ⊗ Struktogramm / Nassi-Shneiderman-Diagramm
- ⌘ interaktive Darstellung durch Algorithmen-Simulationen
- ⌘ Darstellung durch ein Programm

Darstellung in Pseudocode (Knuth 1997)

By 1950, the word algorithm was most frequently associated with Euclid's algorithm, a process for finding the greatest common divisor of two numbers that appears in Euclid's *Elements* (Book 7, Propositions 1 and 2). It will be instructive to exhibit Euclid's algorithm here:

Algorithm E (*Euclid's algorithm*). Given two positive integers m and n , find their *greatest common divisor*, that is, the largest positive integer that evenly divides both m and n .

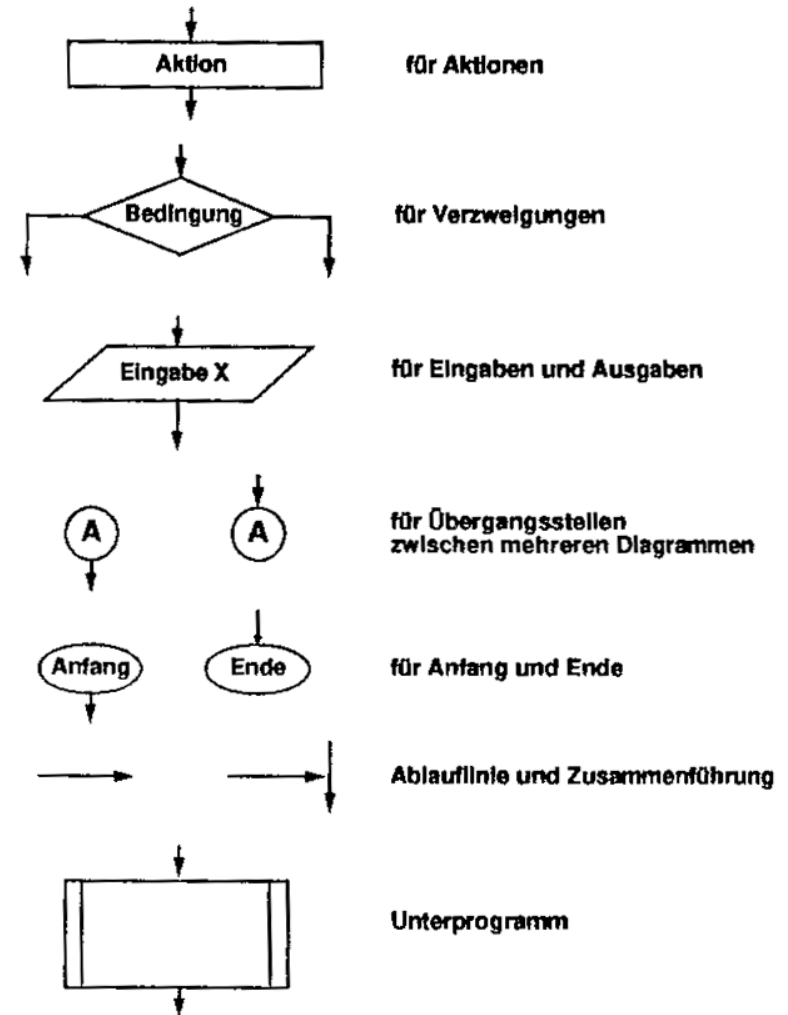
- E1.** [Find remainder.] Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)
- E2.** [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.
- E3.** [Reduce.] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1. ■

Of course, Euclid did not present his algorithm in just this manner. The format above illustrates the style in which all of the algorithms throughout this book will be presented.

Programmablaufplan nach DIN 66001

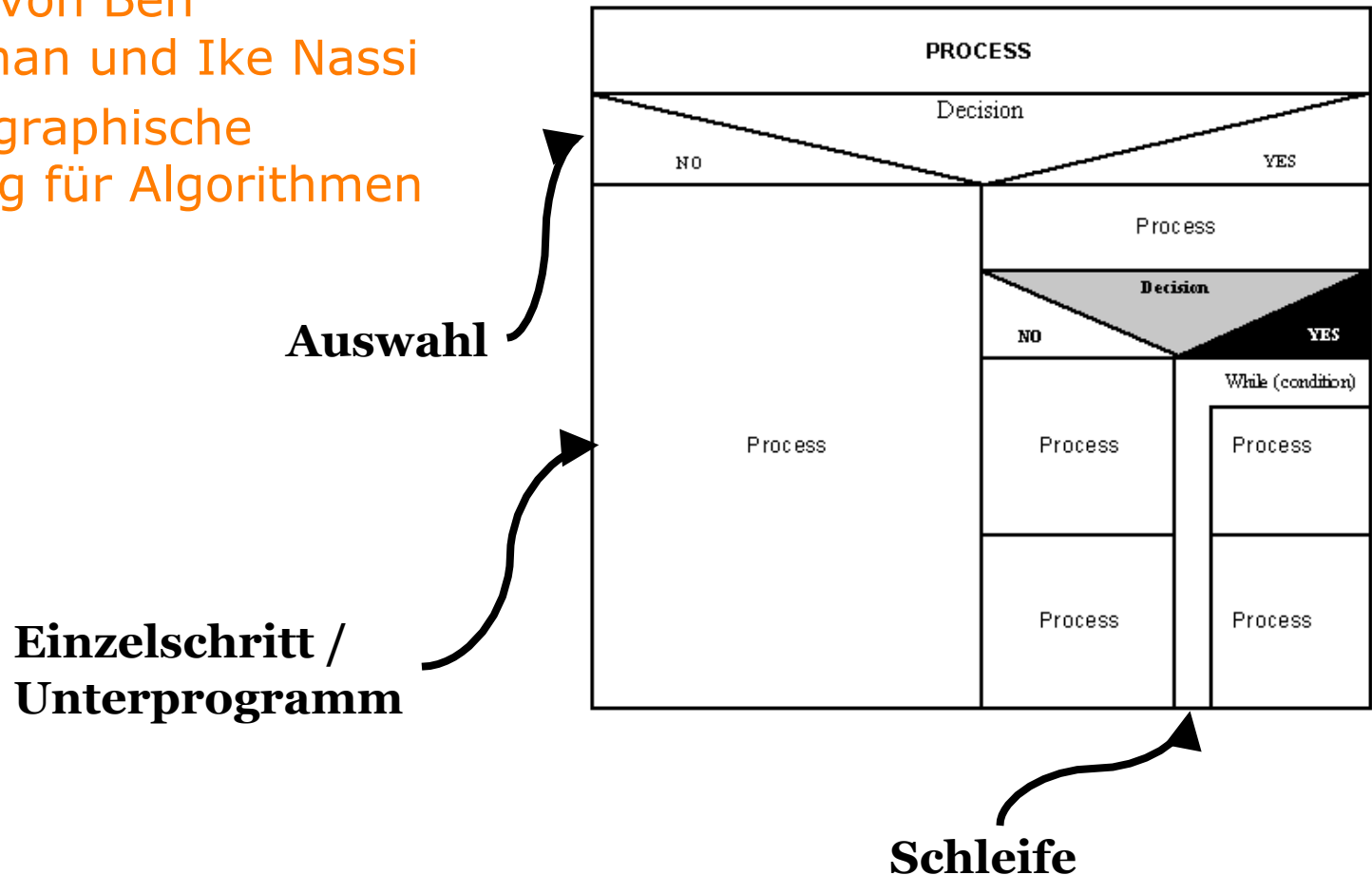
⌘ Deutsche Industrienorm für Sinnbilder für Datenfluss- und Programmablaufpläne, wesentliche Elemente:

- ⊗ Rechteck: Operation
- ⊗ Raute: Verzweigung
- ⊗ Rechteck mit doppelten, vertikalen Linien: Unterprogramm
- ⊗ Parallelogramm: Ein- und Ausgabe
- ⊗ Pfeil, Linie: Verbindung zum nächstfolgenden Element
- ⊗ Oval: Start, Endpunkt, weitere Grenzpunkte



Nassi-Shneiderman-Diagramme (Struktogramme)

- ⌘ Entwickelt von Ben Shneiderman und Ike Nassi
- ⌘ kompakte graphische Darstellung für Algorithmen



Quelle: <http://www.smartdraw.com/resources/centers/software/nassi4.htm>

Grundelemente von Algorithmen bzw. Programmen

- ⌘ **N. Wirth** postuliert für die prozedurale Programmierung die Symbiose von Datenstruktur und Algorithmus:

Programm = Algorithmus + Datenstruktur

- ⌘ Grundelemente von Algorithmen sind:
 - ⊗ Variablen und Konstanten
 - ⊗ Ausdrücke
 - ⊗ Zuweisungen
 - ⊗ Kontrollstrukturen (Sequenzen, Verzweigungen, Schleifen /Iteratoren)
 - ⊗ Prozeduren und Funktionen

 - ⊗ (komplexe) Datentypen und (dynamische) Datenstrukturen

Einfache und zusammengesetzte Datenstrukturen

Datentypen	
Einfache Datentypen <ul style="list-style-type: none">• Wert ist unteilbar• Kann nur als Ganzes manipuliert werden• Beispiele: <code>char</code>, <code>int</code>, <code>boolean</code>	Zusammengesetzte Datentypen <ul style="list-style-type: none">• Wert ist aus mehreren, einzeln manipulierbaren Teilwerten zusammengesetzt• Deren Typen können wieder einfach oder zusammengesetzt sein• Rekursives Bauprinzip• Erlaubt Einführung beliebig komplexer Typen• Beispiele: Mengen, Tupel

Beispiel: Zusammengesetzter Datentyp

- ⌘ Zusammengesetzter Datentyp termin: (Modula 2)

```
TYPE termin =  
    RECORD  
        datum: DATUM;  
        beschreibung: STRING;  
    END;
```

Modula 2

- ⌘ Notation in Java: Zusammengesetzter Datentyp wird durch ein Objekt realisiert.

```
import java.util.Date;  
class Termin {  
    public Date datum;  
    public String beschreibung;  
}
```

Java

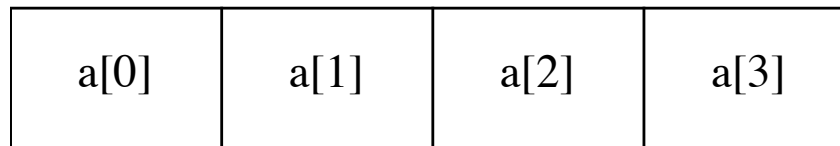
- ⌘ Programmiersprache C: Verwenden einer Struktur

```
struct termin {  
    struct date datum;  
    char beschreibung[80];  
}
```

C/C++

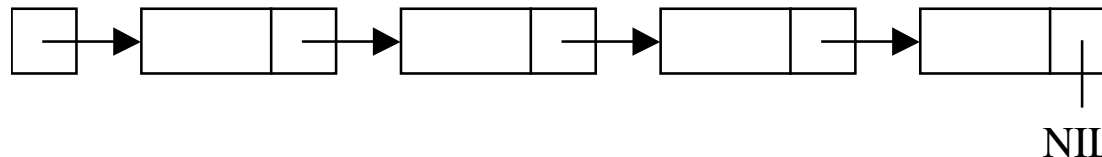
⌘ Felder (Arrays)

- ⊠ sind zusammengesetzte Datenstrukturen, in denen Werte in geordneter Form abgelegt werden können. Der Zugriff auf die Elemente eines Arrays erfolgt durch **Indices**.



⌘ Eine verkettete Liste

- ⊠ ist eine lineare, **dynamisch erweiterbare** Anordnung von Datenelementen, die explizit durch Kanten verbunden sind (realisiert durch **Zeiger** bzw. Referenzen).



Wh.

⌘ Eindimensionales Array:

```
int[] a; // eindimensionales Feld aus int-Werten  
a = new int[DIM]; //Alternativ: int[] a = new int[DIM];
```

a[0]	a[1]	a[2]	a[3]	...	a[DIM-1]
------	------	------	------	-----	----------

⌘ Mehrdimensionales Array:

```
int[][] b; // zweidimensionales Feld  
b = new int[DIMY][DIMX];
```

Höhere Dimensionen?
weitere [...] anfügen

	j					
i	b[0][0]	b[0][1]	b[0][2]	b[0][3]	...	b[0][DIMX-1]
	b[1][0]	b[1][1]	b[1][2]	b[1][3]		b[1][DIMX-1]
	⋮				⋮	
					b[i][j]	

⌘ Hinter dem Namen des Arrays folgt der Index in eckigen Klammern:

```
int[] a = new int[DIM]; // eindimensionales Feld aus int-Werten
```

```
int[][] b = new int[DIMY][DIMX]; // zweidimensionales Feld
```

```
int i,k;
```

```
...
```

```
a[i] = ... // ganz normale Wertzuweisung
```

```
b[i][k] = ...
```

a[0]	a[1]	a[2]	a[3]	...	a[DIM-1]
------	------	------	------	-----	----------

	j					
i	b[0][0]	b[0][1]	b[0][2]	b[0][3]	...	b[0][DIMX-1]
	b[1][0]	b[1][1]	b[1][2]	b[1][3]		b[1][DIMX-1]
	⋮				⋮	

Wh.

⌘ Syntax (vereinfacht):

```
Type: ... | Identifier{[]}  
ArrayAccess: Primary[Expression]  
ArrayCreationExpression: new Type{Length}{[]}  
Length: [Expression]
```

⌘ Beachte

- ⊗ Sei n die Länge eines Feldes a ; Indizes laufen von 0 bis $n-1$.
- ⊗ Wenn Index nicht aus $[0, n-1]$
 - ⊕ Fehler: »array index out of bounds«.
- ⊗ Die Länge des Feldes: $a.length$
- ⊗ Typ des Index muss mit int verträglich sein.
- ⊗ Alle Elemente sind vom gleichen Typ.
- ⊗ Die Teilfelder mehrdimensionaler Arrays können verschiedene Längen haben.

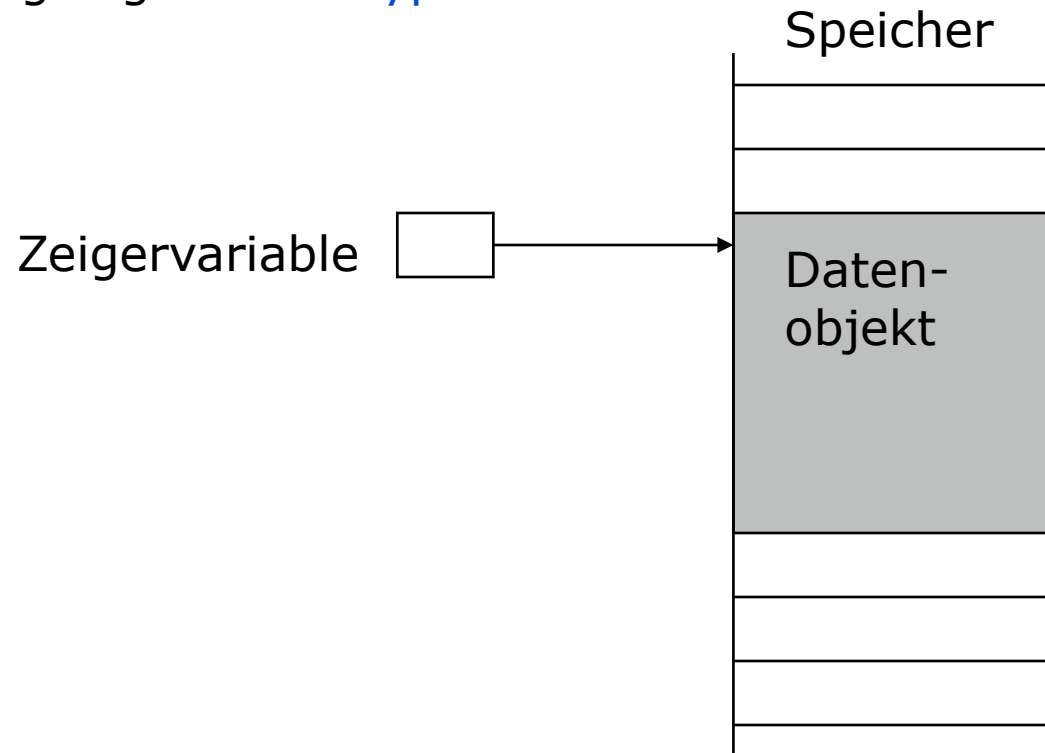
Wh.

- ⌘ Teilfelder mehrdimensionaler Arrays können verschiedene Längen haben
- ⌘ Beispiel (Partl):

```
double[][] tagesUmsatz = new double[12][];  
int[] monatsLaenge =  
    { 31,29,31,30,31,30,31,31,30,31,30,31 }; // Feld-Initialisierung  
                                           // ueber Wertbezeichner  
for (int monat=0; monat<tagesUmsatz.length; monat++) {  
    tagesUmsatz[monat] = new double[ monatsLaenge[monat] ];  
    for (int tag=0; tag<tagesUmsatz[monat].length; tag++) {  
        tagesUmsatz[monat][tag] = 0.0;  
    }  
}
```

⌘ Zeigervariable (pointer variable)

- ⊗ enthält die Adresse eines Datenobjektes. Die Bezeichnung eines Datenobjektes durch eine Zeigervariable heißt Referenz.
- ⊗ Zeiger sind insbesondere im Zusammenhang mit dynamischen Datenstrukturen interessant.
- ⊗ Speicherplatz für dynamische Datenstruktur wird erst bei Bedarf zur Laufzeit angelegt: `new <type>`



- ⌘ Eine verkettete Liste ist eine dynamische Datenstruktur die in ihrer Länge veränderlich ist
- ⌘ Besteht aus einer Menge linear miteinander verbundener Elemente die Knoten genannt werden
- ⌘ Jedes Element (Knoten) besitzt

- ⊗ einen Datenteil
- ⊗ einen Referenzteil



⌘ Typen

- ⊗ Einfach verkettete Listen: Referenzteil enthält eine Referenz auf den Nachfolgeknoten des Elements
- ⊗ Doppelt verkettete Listen: Referenzteil enthält eine Referenz auf den Nachfolge- und eine Referenz auf den Vorgängerknoten

⌘ Eigenschaften

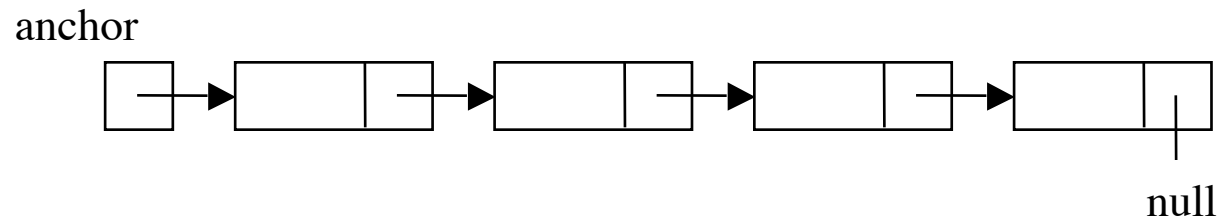
- ⊗ Einfügen und Löschen von Elementen einfach möglich
- ⊗ Durchlaufen der Liste nur über die Referenzen von Element zu Element möglich

Einfach verkettete Liste

- ⌘ Jedes Element (Knoten) referenziert auf seinen Nachfolger und ermöglicht so die Navigation durch die Liste (von Knoten zu Knoten)
- ⌘ Die Referenz des letzten Elements hat den Wert null oder zeigt auf einen Pseudoknoten
- ⌘ Durchlaufen (traversieren) der Liste nur sequentiell von vorne nach hinten möglich

⌘ Beispiel für einen Knoten:

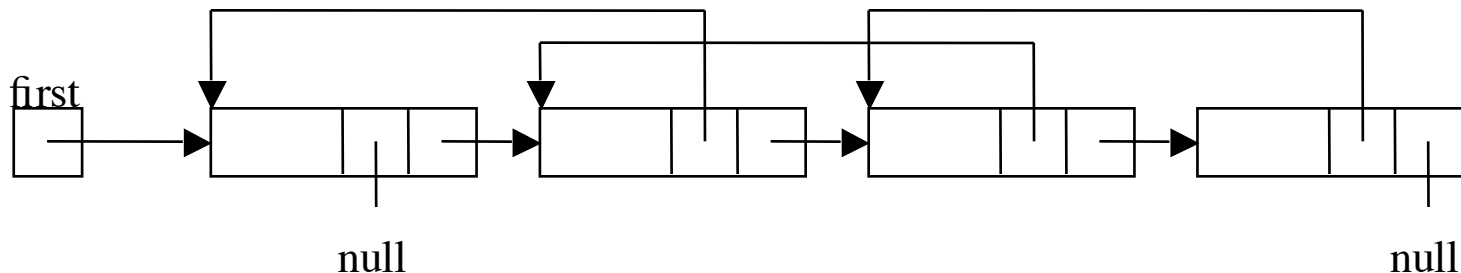
```
public class Node{  
    Object o;  
    Node nachfolger;  
}
```



Doppelt verkettete Listen

- ⌘ Jedes Element besitzt eine Referenz auf Vorgänger und Nachfolger
- ⌘ Erlaubt das Traversieren vorwärts und rückwärts
- ⌘ Beachte die auftretenden Situationen beim Aus- und Einketten von Elementen:
 - ⊗ Leere Liste (erstes Element einketten, letztes Element ausketten),
 - ⊗ Vor erstem Element ein/ausketten,
 - ⊗ Nach letztem Element ein/ausketten.
- ⌘ Beispiel für einen Knoten:

```
public class Node{  
    Object o;  
    Node vorgaenger;  
    Node nachfolger;  
}
```

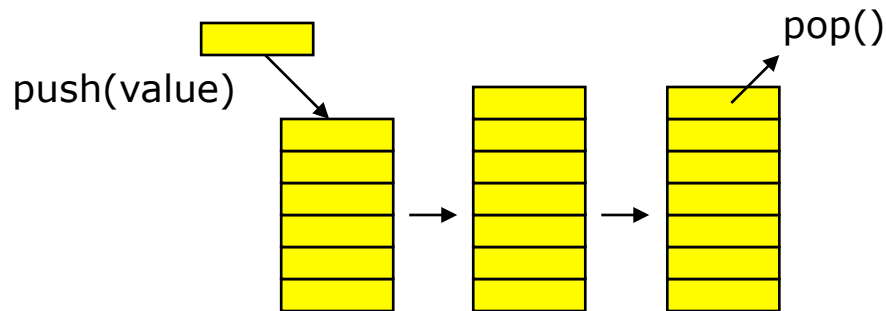


⌘ Abstrakter Datentyp (ADT, abstract data type)

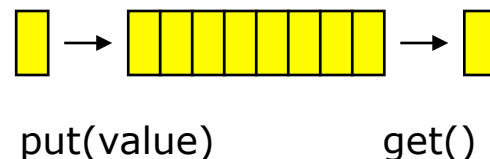
- ⊠ definiert eine Softwarekomponente (aufgefasst als Datentyp) durch eine **Menge von Werten** und durch die in abstrakter Form, d.h. ohne Bezug auf eine Implementierung) definierten **Operationen** auf diesen Werten.

⌘ Beispiele

- ⊠ Stapel (Keller, Stack) oder LIFO-(last in first out)-Behälter.



- ⊠ FIFO-Behälter (Warteschlange, Queue)



⌘ Beispiele

- ⊗ Stapel (Keller, Stack) oder LIFO-(last in first out)-Behälter.

```
push(int v); // Element v auf Stapel legen
int pop();   // oberstes Element vom Stapel holen
boolean isEmpty();
```

- ⊗ FIFO-Behälter (Warteschlange, Queue)

```
put(int v); // Element hinten einreihen
int get();  // erstes Element vorne holen
boolean isEmpty();
```

- ⊕ bei bekannter Größe der Warteschlange:
 - Ringspeicher (z.B. auch angewendet beim Cache)

Stapel als Array-Implementierung

```
public class Stack {
    private Object[] s;
    private int N;

    Stack(int maxN) {
        s = new Object[maxN];
        N = 0;
    }
    public void push(Object item) {
        if(N<maxN){
            s[N] = item;
            N++;
        }
    }
    public Object pop() {
        if(isEmpty())
            return null;
        N--;
        Object t = s[N];
        s[N] = null;
        return t;
    }
    public boolean isEmpty() {
        return N == 0;
    }
}
```

Stapel als verkettete Liste (1/2)

```
public class Stack {
    private Node anchor;

    class Node {
        Object content;
        Node link;
        Node(Object value, Node prevLink) {
            content = value;
            link = prevLink;
        }
    }

    public Stack() { anchor = null; }

    public void push(Object element) {
        // neues Element vorne einketten
        anchor = new Node(element, anchor);
    }
}
```

Stapel als verkettete Liste (2/2)

```
public Object pop() {
    if (isEmpty())
        // return new String("[Error: Stack is empty]");
        return null;
    Object value = anchor.content; // zwischenspeichern
    anchor = anchor.link; // letztes eingetragenes Element
                              ausketten
    return value;
}

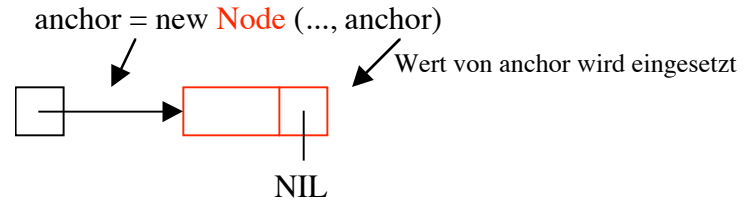
public boolean isEmpty() {
    return anchor == null;
}
}
```

Wie funktioniert's?

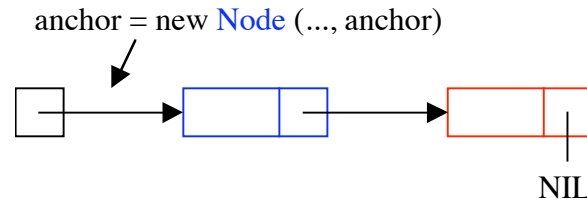
⌘ Nach Initialisierung: anchor:



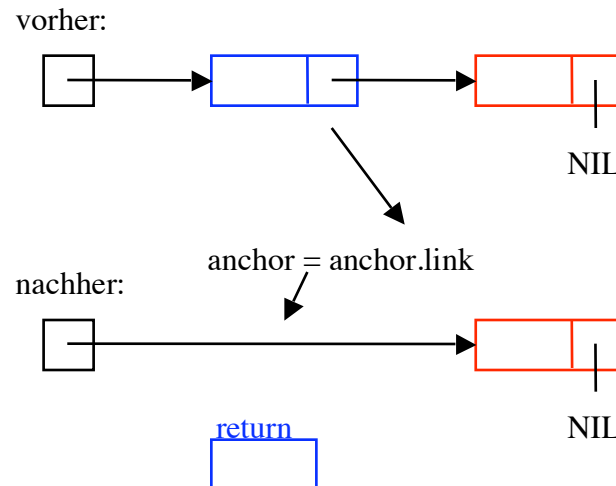
⌘ Erstes Element hinzufügen:



⌘ Nächstes Element hinzufügen:



⌘ Element holen: pop()



Wozu wird der Stapel gebraucht?

⌘ Speichern von Zwischenwerten bei Unterprogrammaufrufen

⌘ Auswertung arithmetischer Ausdrücke:

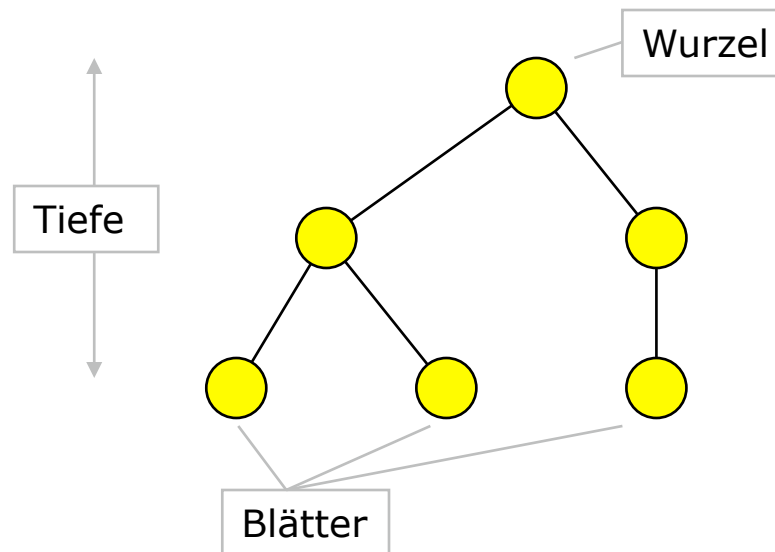
⊠ Beispiel: $5 * (((9+8) * (4*6)) + 7)$

```
push(5)
push(9)
push(8)
push(pop() + pop())
push(4)
push(6)
push(pop() * pop())
push(pop() * pop())
push(7)
push(pop() + pop())
push(pop() * pop())
```

→ Auf dem Stack steht das Ergebnis. Holen mit pop()

⌘ Bäume

- ⊠ sind zweidimensionale verkettete Strukturen
- ⊠ spezielle Graphen, bestehen aus **Knoten** und **Kanten**
- ⊠ besonderer Knoten: **Wurzel**
- ⊠ zwischen Wurzel und jedem beliebigen Knoten gibt es genau einen **Pfad**

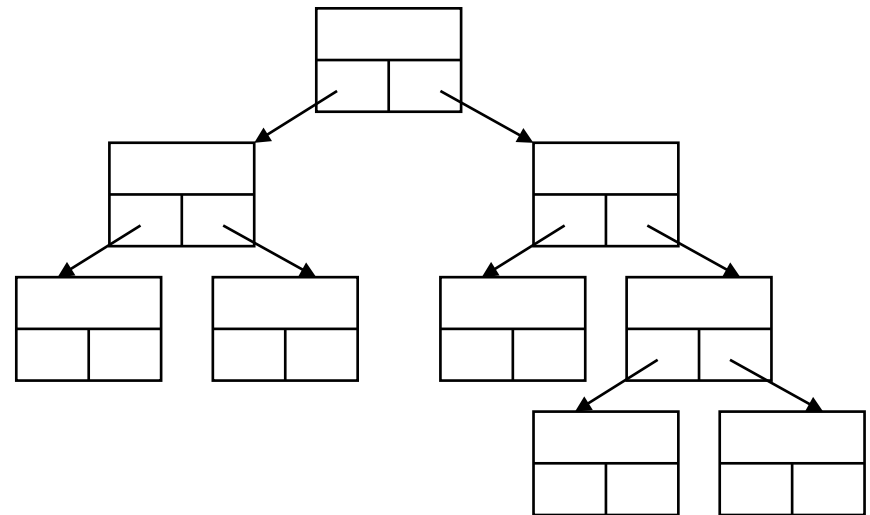
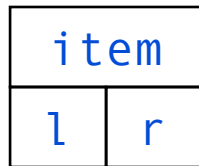


⌘ Spezielle Art:

- ⊠ Binärbaum: Innere Knoten besitzen genau zwei Nachfolger

- ⌘ Innere Knoten besitzen genau zwei Nachfolger

```
class Node { Object item; Node l, r; };
```



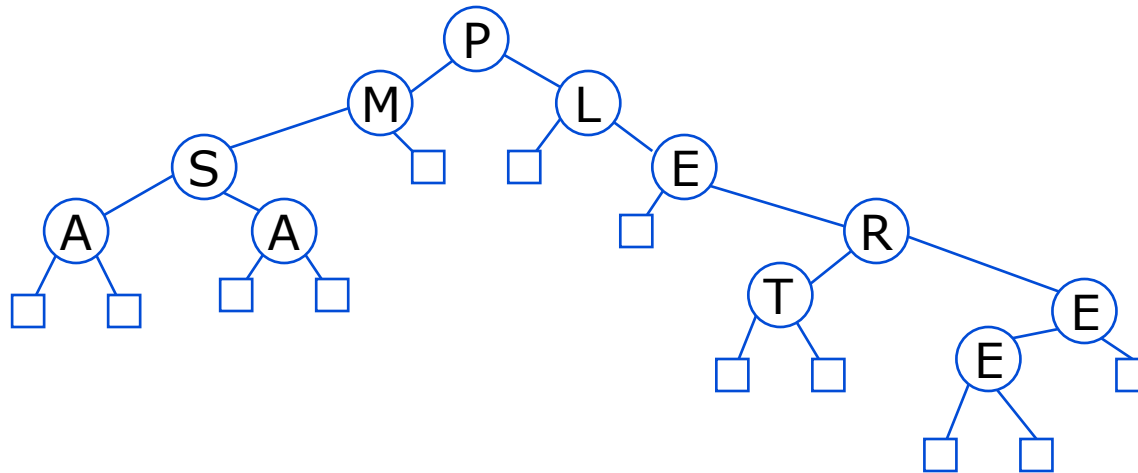
- ⌘ Traversierung:

- ⊠ Preorder : Wurzel → linker Teilbaum → rechter Teilbaum
- ⊠ Inorder : linker Teilbaum → Wurzel → rechter Teilbaum
- ⊠ Postorder : linker Teilbaum → rechter Teilbaum → Wurzel

⌘ Traversierung:

- ⊗ Preorder : Wurzel → linker Teilbaum → rechter Teilbaum
- ⊗ Inorder : linker Teilbaum → Wurzel → rechter Teilbaum
- ⊗ Postorder : linker Teilbaum → rechter Teilbaum → Wurzel

⌘ Beispiel:



- ⊗ Preorder : PMSAALERTEE
- ⊗ Inorder : ASAMPLETREE
- ⊗ Postorder : AASMTEERELP

Preorder-Traversieren iterativ (mit Stack)

```
Stack s = new Stack(max);
```

```
void traverse(Node t) {  
    s.push(t);  
    while(!s.isEmpty()) {  
        t=s.pop();  
        t.item.visit();  
        if (t.r != null) s.push(t.r);  
        if (t.l != null) s.push(t.l);  
    }  
}
```

Preorder-Traversieren rekursiv

```
void traverse(Node t) {
    if (t == null)
        return;
    t.item.visit();
    traverse(t.l);
    traverse(t.r);
}
```

```
/*Annahme: Die im Knoten gespeicherten Objekte verfügen
über eine Methode visit(); */
```

```
//Beispiel für Item das in Knoten referenziert werden kann
public class KnotenItem{
    char c;
    public void visit(){
        System.out.print(c);
    }
}
```

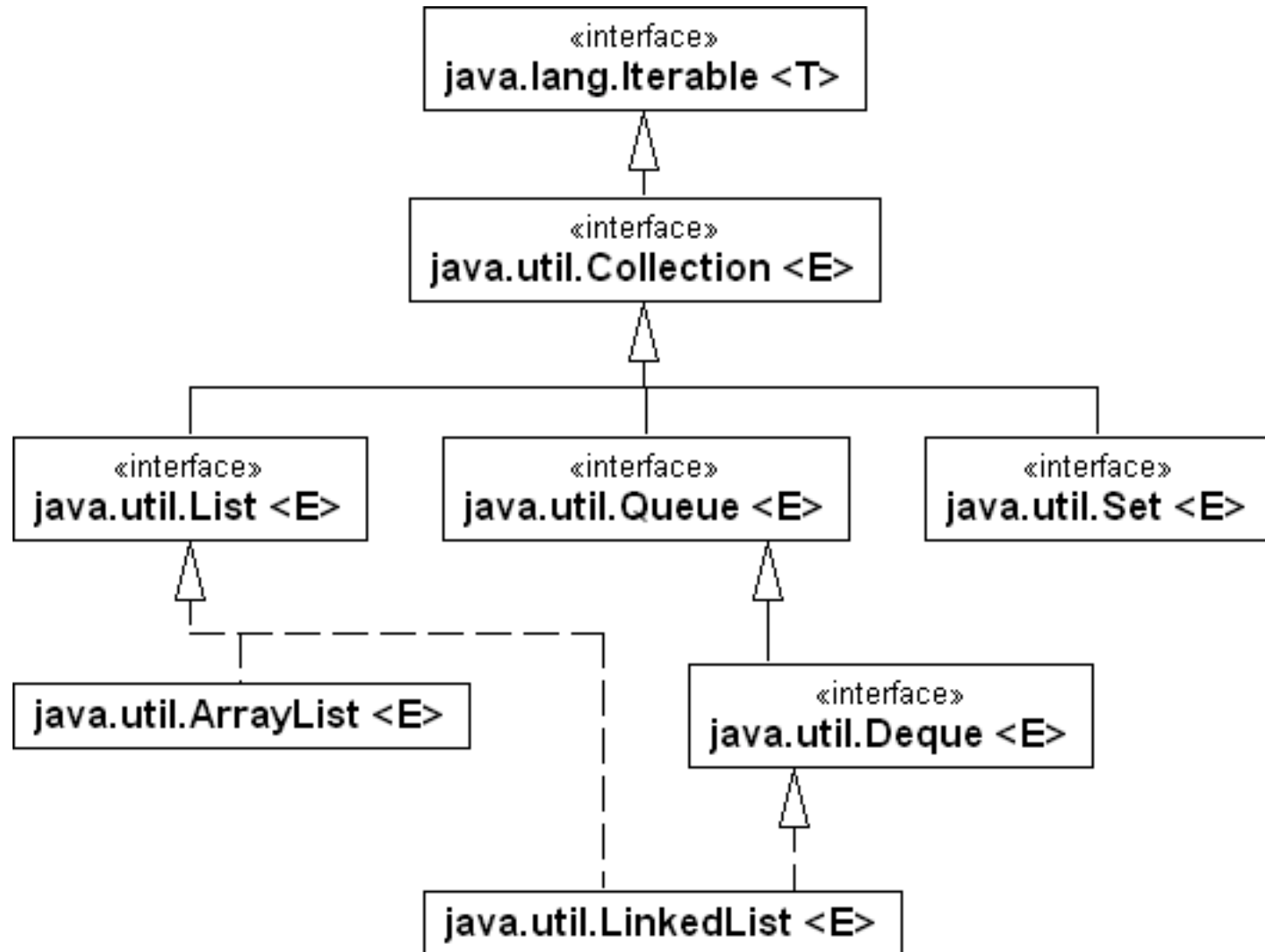
Datenstrukturen in der Java Collection API

- ⌘ Für komplexe Datenstrukturen stellt Java eine einheitliche API zur Verfügung
 - ⊗ Implementierungen der am häufigsten benötigten Datenstrukturen
 - ⊗ Einheitliches Verwendungskonzept

- ⌘ Eine Collection (Container, Sammlung) ist ein Objekt, das wiederum mehrere andere Objekte aufnehmen kann und diese verwaltet

- ⌘ Designprinzipien der Collection-Klassen (vgl. Javainsel 7, 12.1.1)
 - ⊗ Schnittstellen: legen Operationen für Behältertypen fest
 - ⊗ Abstrakte Basisklassen: definieren Grundoperationen
 - ⊗ Konkrete Klassen: Erben von den Basisklassen und stellen Implementierung der Grundoperationen bereit
 - ⊗ Algorithmen: Bestimmte „Basisalgorithmen“ sind zum Teil in den Schnittstellen festgelegt

Elementare Schnittstellen



Quelle: Christian Ullenboom, *Java ist auch eine Insel*

Konkrete Container-Klassen für Listen

⌘ Klassen

- ⊗ ArrayList: Auf Basis eines Feldes
- ⊗ LinkedList: Auf Basis einer doppelt verketteten Liste

⌘ Implementieren das Interface List

⌘ Gemeinsame Methoden

- ⊗ Element einfügen
- ⊗ Prüfen ob bestimmtes Element in der Liste enthalten ist
- ⊗ Element an gegebener Position zurückliefern
- ⊗ Element löschen
- ⊗ Letztes Vorkommen eines bestimmten Elements in der Liste finden

LIFO und FIFO Speicher in der Java API

⌘ Die Klasse Stack

- ⊗ realisiert einen Kellerspeicher (Stack, LIFO)
- ⊗ Erbt von Vector
- ⊗ Passt nicht so recht in die Systematik der übrigen hier vorgestellten Klassen

⌘ Das Interface Queue

- ⊗ Verschiedene implementierende Klassen
- ⊗ Unterschiedliche Schlangentypen damit einfach realisierbar
- ⊗ Beispiele: ArrayBlockingQueue, LinkedBlockingQueue, DelayQueue.

Datentypen in Collections

⌘ Grundsätzlich sind die Datenstrukturen offen für alle Typen

- ⊗ Object wird als Datentyp entgegengekommen und zurückgegeben
- ⊗ `void add(Object o)`
- ⊗ `Object get(int index)`
- ⊗ So lassen sich prinzipiell Daten unterschiedlicher Typen in eine Collection einfügen → häufig nicht erwünscht

⌘ Durch Nutzung des Generics Konzepts Einschränkung auf bestimmte Typen möglich (ab Java 5)

- ⊗ Bei der Konstruktion einer Datenstruktur kann angegeben werden, welcher Datentyp erlaubt ist
- ⊗ Beispiel:
`List<Secigotchi> secis = new LinkedList<Secigotchi>();`
→ Liste nimmt nur Objekte der Secigotchi-Klasse auf

⌘ Zu allen primitiven Datentypen existieren in Java sogenannte Wrapper-Klassen (Mantelklassen)

⌘ Gründe:

- ⊗ Komplexe Datenstrukturen können nur Objekte, aber keine primitiven Datentypen aufnehmen
- ⊗ Bereitstellung von Umwandlungsfunktionen (z.B. mit `Integer.parseInt()` übergebenen String in int wandeln)

⌘ Beispiel

```
⊗ Integer i = new Integer(29);  
List nummern = new ArrayList();  
nummern.add(i);
```

⌘ Autoboxing (seit Java 5)

- ⊗ Bei Bedarf werden primitive Typen und Wrapper ineinander umgewandelt

- ⌘ Problemstellung: Häufig müssen Datenstrukturen (z.B. verkettete Listen oder Arrays) Element für Element durchlaufen werden
- ⌘ Je nach Datenstruktur und Implementierung unterschiedliche
 - ⊗ verkettete Liste: über Referenzen auf nächstes Element
 - ⊗ Arrays: über Index-/Laufvariable
- ⌘ Ziel von Iteratoren: Einheitliche Behandlung des Durchlaufens unabhängig von der internen Realisierung
- ⌘ In Java
 - ⊗ Interface `java.util.Iterator`
 - ⊗ Stellt einheitliche Schnittstelle zum Durchlaufen von Kollektionen zur Verfügung
- ⌘ Methoden
 - ⊗ `boolean hasNext()`
 - ⊗ `Object next()`
 - ⊗ `void remove()`
- ⌘ Die Klassen der Java Collection API liefern durch Aufruf der Methode `iterator()` einen Iterator für die jeweilige Datenstruktur

⌘ Problemstellung

- ⊗ Eine Menge von Elementen (Datensätzen) soll in eine bestimmte Ordnung gebracht werden

⌘ Sortierschlüssel

- ⊗ Der Teil des Datensatzes nach dem sortiert wird (z.B. ein bestimmtest Attribut eines Objekts)

⌘ Ordnung

- ⊗ z.B. lexikographisch oder numerisch
- ⊗ z.B. aufsteigend oder absteigend

⌘ Realwelt-Beispiel

- ⊗ Eine Menge von Personen (Elemente)
- ⊗ der Größe (Sortierschlüssel) nach
- ⊗ aufsteigend (Ordnung) aufstellen.

Sortieralgorithmen (Teil 1)

⌘ Algorithmen

- ⊗ Selection Sort
- ⊗ Bubble Sort
- ⊗ Quick Sort
- ⊗ Merge Sort
- ⊗ Heap Sort

⌘ Für die Einführung der Algorithmen wird zunächst vom einfachsten Fall ausgegangen:

- ⊗ Sortieren eines Arrays von Integer-Zahlen `int[] a;`

Selection-Sort auf einem Integer-Array

Wh.

⌘ Prinzip:

- ⊗ Kleinstes Element heraussuchen und auf seine endgültige Position bringen, mit dem Rest jeweils analog verfahren

```
void selectionSort() {
    for(int i=0;i<a.length-1;i++) {
        // small auf den Index des ersten Vorkommens
        // des kleinsten verbleibenden Elements setzen
        int small = i;
        for(int j=i+1;j<a.length;j++)
            if (a[j] < a[small]) small = j;
        // wenn man hier ankommt, ist small der Index des
        // ersten kleinsten Elements in a[i..n]. Nun wird
        // a[small] mit a[i] vertauscht
        int temp = a[small];
        a[small] = a[i];
        a[i]      = temp;
    }
}
```

Bubble-Sort auf einem Integer-Array

Wh.

⌘ Prinzip:

- ⊗ Sortieren durch wiederholtes Vertauschen benachbarter Elemente bis die endgültige Reihenfolge feststeht

```
void bubbleSort() {
    for(int i=0;i<a.length;i++) {
        for(int j=0;j<a.length-1;j++)
            if (a[j+1] < a[j]) {
                // zwei benachbarte Elemente
                // werden vertauscht,
                // wenn das groessere vorne liegt
                int temp = a[j+1];
                a[j+1]    = a[j];
                a[j]      = temp;
            }
        }
    }
```

⌘ Fragestellung:

- ⊗ Wie verhalten sich Bubble- und Selection-Sort bzgl. Ihrer Laufzeit mit wachsender Array-Länge (Problemgröße)?

- ⌘ Ausführungseffizienz eines Algorithmus ist von Bedeutung, wenn
 - ⊗ Algorithmus auf einer großen Menge von Daten operiert
 - ⊗ wiederholt (sehr häufig) ausgeführt wird
- ⌘ Wichtige Effizienzmaße
 - ⊗ Menge an Speicherplatz,
 - ⊗ **Ausführungszeit**
 - ⊗ ferner
 - ⊕ Datenbewegungen in einem Rechnernetz,
 - ⊕ Menge der Ein-/Ausgabeoperationen mit (externen) Geräten und Datenträgern.
- ⌘ Ansätze zur Messung der Laufzeit:
 - ⊗ Bewertungsprogramme (Benchmarks)
 - ⊗ Analyse

- ⌘ Vergleich von zwei oder mehreren Programmen, die zur Bearbeitung derselben Aufgaben entworfen wurden
 - ⊗ Abarbeitung einer Sammlung typischer Aufgaben bzw. Eingaben und Messung der Laufzeit

- ⌘ Ergebnisse sind abhängig von der „Umgebung“, in der die Zeitmessungen durchgeführt werden:
 - ⊗ Programmiersprache (compiliert oder interpretiert),
 - ⊗ Betriebssystem und Ausführungsumgebung, welche(s) das Programm interpretiert oder den Binärcode (oder Java: Bytecode) ausführt,
 - ⊗ Hardware, auf der das Betriebssystem oder die Ausführungsumgebung ausgeführt wird.

- ⌘ Benchmarks liefern sehr konkrete Angaben zur Schnelligkeit eines Programms, die aber nicht allgemeingültig sind.

Analyse eines Programms

- ⌘ Laufzeit ist abhängig von der sog. Problemgröße, die ein Algorithmus lösen soll, aber unabhängig von konkreten Umgebungen

- ⌘ Beispiele für „Problemgröße“ n
 - ⊗ Sortieralgorithmus: Anzahl der zu sortierenden Elemente
 - ⊗ Programm zum Lösen linearer Gleichungssysteme: Anzahl der Unbekannten
 - ⊗ Länge einer Liste, Größe eines Arrays, ...

- ⌘ Laufzeit eines Programms
 - ⊗ Sei $T(n)$ eine Funktion, die die Anzahl der Zeiteinheiten repräsentiert, die ein Algorithmus für eine Eingabe der Größe n verbraucht. Wir bezeichnen $T(n)$ als Laufzeit des Programms.

⌘ Intuitive Vorstellung

- ⊗ $T(n)$ ist die Anzahl der abgearbeiteten Java-Anweisungen eines Programms
- ⊗ keine nähere Spezifikation der Einheit von $T(n)$

⌘ Man unterscheidet

- ⊗ **Laufzeit im schlimmsten Fall** (worst case running time): Maximale Laufzeit, die eine Eingabe aus allen Eingaben der Größe n hervorruft.
- ⊗ **Durchschnittliche Laufzeit** $T_{\text{avg}}(n)$ über allen Eingaben der Größe n ; setzt voraus, dass alle Eingaben der Größe n gleich wahrscheinlich sind.

Beispiel: Worst-Case-Analyse von Selection Sort

Annahme: Eine Zeiteinheit für jede Anweisung.

```
void selectionSort() {
(1)     for(int i=0;i<n-1;i++) {
(2)         int small = i;
(3)         for(int j=i+1;j<n;j++)
(4)             if (a[j] < a[small])
(5)                 small = j;
(6)         int temp = a[small];
(7)         a[small] = a[i];
(8)         a[i] = temp;
    }
}
```

(1)	Initialisierung	: i=0		1	
	letzter Vergleich	: i<n-1		1	

	Inkrementierung	: i++		1	(n-1) mal
	Vergleich	: i<n-1		1	
(2)	Zuweisung	: small=i		1	
(3)	Initialisierung	: j=i+1		1	
	letzter Vergleich	: j<n		1	

	Inkrementierung	: j++	1		(n-i-1) mal
	Vergleich	: j<n	1		
(4)	Vergleich	: a[j]<a[small]	1		
(5)	Zuweisung (worst case):	small=j	1		

				4	* (n-i-1)

(6)	Zuweisung	: temp=a[small]		1	
(7)	Zuweisung	: a[small]=a[i]		1	
(8)	Zuweisung	: a[i]=temp		1	

				4	* (n-i-1) + 8

Beispiel: Worst-Case-Analyse von Selection Sort

⌘ Beobachtung:

- ⊠ Mit aufsteigendem i (äußere Schleife) sinkt die Anzahl der Durchläufe der inneren Schleife jeweils um 1.
- ⊠ Wegen der Iterationen der äußeren Schleife mit $i = 0, \dots, n-2$ ($n > 1$) folgt für die Anzahl der Iterationen der inneren Schleife: $(n-i-1) := n-1, n-2, \dots, 2, 1$

⌘ Folglich ist

- ⊠ $T(n) = 2 + (4 \cdot (n-1) + 8) + (4 \cdot (n-2) + 8) + \dots + (4 \cdot 1 + 8)$

Beispiel: Worst-Case-Analyse von Selection Sort

$$\boxtimes T(n) = 2 + (4 \cdot (n-1) + 8) + (4 \cdot (n-2) + 8) + \dots + (4 \cdot 1 + 8)$$

⌘ Als Summenformel hingeschrieben:

$$T(n) = 2 + \sum_{i=1}^{n-1} (4i + 8)$$

$$= 2 + 4 \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 8$$

$$= 2 + 8(n-1) + 4 \sum_{i=1}^{n-1} i$$

$$= 2 + 8(n-1) + 2(n^2 - n)$$

$$= 2 + 8n - 8 + 2n^2 - 2n$$

$$T(n) = 2n^2 + 6n - 6$$

$$\text{mit } \sum_{i=1}^{n-1} i = \frac{1}{2}(n^2 - n)$$

⌘ Die konkrete Laufzeit der Implementierung von Selection Sort in Abhängigkeit der Anzahl der zu sortierenden Elemente ist durch das angegebene Polynom beschrieben.

- ⌘ Mit steigendem n gewinnt Koeffizient der höchsten Potenz von n in $T(n)$ zunehmend an Bedeutung für die Laufzeit.
- ⌘ Wenn $T(n)$ ein Polynom in n ist mit n^k ($k > 0$) als höchster Potenz von n , dann hat der zugehörige Algorithmus eine Zeitkomplexität der Größenordnung n^k . Dies wird mit $O(n^k)$ ausgedrückt.
- ⌘ Die O -Notation dient dazu, das asymptotische Wachstum einer Funktion abzuschätzen. Sei $T(n)$ die Laufzeit eines Programms, die über der Eingabegröße n berechnet wird. Man sagt „ $T(n)$ ist $O(f(n))$ “ für Funktionen $T, f: \mathbb{N} \rightarrow \mathbb{N}$, wenn es eine **ganze Zahl n_0** und eine **Konstante $c > 0$** gibt, dass für alle $n \geq n_0$ gilt: $T(n) \leq c \cdot f(n)$.
- ⌘ **Eigenschaften:**
 - ⊗ gilt nur für genügend große n
 - ⊕ aber allgemeingültig für alle Ausführungsumgebungen
 - ⊕ deshalb auch eine Eigenschaft des abstrakten Algorithmus.
 - ⊗ $O(f(n))$ gibt eine obere Schranke für $T(n)$ an.

O-Notation: Typische Aussagen

⌘ $T(n)$ ist **konstant**:

⊗ $O(1)$ Speicher- oder Zeitverbrauch sind von der Problemgröße unabhängig, also konstant.

⌘ $T(n)$ ist **polynomial**: $O(n^k)$ mit $k > 0$, beispielsweise

⊗ **linear**: $O(n)$ Speicher- oder Zeitverbrauch wachsen direkt proportional mit der Problemgröße.

⊗ **quadratisch**: $O(n^2)$ Speicher- oder Zeitverbrauch wachsen quadratisch mit der Problemgröße.

⌘ $T(n)$ ist **logarithmisch**:

⊗ $O(\log n)$, $O(n \log n)$, ... Speicher- oder Zeitverbrauch wachsen lediglich logarithmisch mit der Problemgröße. Die Basis des Logarithmus wird häufig 2 sein, d.h. vierfache Datenmenge verursacht doppelten Ressourcenverbrauch, 8-fache Datenmenge verursacht 3-fachen Verbrauch und 1024-fache Datenmenge 10-fachen Verbrauch. $O(n \log n)$ liegt zwischen $O(n)$ und $O(n^2)$.

⌘ $T(n)$ ist **exponentiell**:

⊗ $O(2^n)$ Bei doppelter, dreifacher und 10-facher Datenmenge steigt der Ressourcenverbrauch auf das 4-, 8- bzw. 1024-fache.

Addieren von O-Ausdrücken: Summenregel

⌘ Problemstellung

- ⊗ Die Laufzeiten $T_1(n)$ und $T_2(n)$ zweier Programmabschnitte sind $O(f_1(n))$ und $O(f_2(n))$. Gesucht ist $O(f_1(n) + f_2(n))$.

⌘ Summenregel

- ⊗ Angenommen $T_1(n)$ ist $O(f_1(n))$ und $T_2(n)$ ist $O(f_2(n))$. Es gelte, dass f_2 nicht schneller wächst als f_1 .
- ⊗ Das bedeutet: $f_2(n)$ ist $O(f_1(n))$.
- ⊗ Daraus folgt: $T_1(n) + T_2(n)$ ist $O(f_1(n))$.

⌘ Beweis

- ⊗ in: Alfred V. Aho, Jeffrey D. Ullman: Informatik — Datenstrukturen und Konzepte der Abstraktion. International Thomson Publishing, 1996, S. 150.

⌘ „Rechenregeln“

- ⊗ $f = O(f)$
- ⊗ $c O(f) = O(f)$ $c = \text{const}$
- ⊗ $O(O(f)) = O(f)$
- ⊗ $O(f) + O(g) = O(f+g)$

Beispiel: Summenregel

⌘ Programm zum Herstellen einer Einheitsmatrix

```
(1)   int n = keyboard.readInt();
(2)   for(int i=0; i<n; i++)
(3)       for(int j=0; j<n; j++)
(4)           a[i][j]=0;
(5)   for(int i=0; i<n; i++)
(6)       a[i][i]=1;
```

⌘ Es sei

- ⊗ $T_1(n)$ ist $O(1)$ für Zeile 1,
- ⊗ $T_2(n)$ ist $O(n^2)$ für Zeilen 2-4
- ⊗ $T_3(n)$ ist $O(n)$ für Zeilen 5-6

⌘ Gesucht

- ⊗ Obere Schranke für die Laufzeit von $T_1(n) + T_2(n) + T_3(n)$

⌘ Lösung

- ⊗ Da die Konstante 1 mit Sicherheit $O(n^2)$ ist, folgt: $T_1(n)+T_2(n)$ ist $O(n^2)$
- ⊗ Da n ebenfalls mit Sicherheit $O(n^2)$ ist, folgt: $T_1(n)+T_2(n)+ T_3(n)$ ist $O(n^2)$
- ⊗ Das bedeutet: Das Programm verbraucht den wesentlichen Teil seiner Abarbeitungszeit in Programmzeilen (2)-(4).

Regeln zur Laufzeitanalyse von Anweisungen

⌘ Einfache Anweisung

⊗ Die Schranke für eine **einfache Anweisung** ist $O(1)$.

⌘ Iteration

⊗ Sei $O(f(n))$ eine obere Schranke für den Rumpf einer **Schleife** (while, for etc.), die durch die rekursive Anwendung dieser Regeln gebildet wurde. Sei $O(g(n))$ eine obere Schranke für die Anzahl der möglichen Schleifendurchläufe, jedoch mindestens 1 für alle n . Dann ist $O(f(n)g(n))$ eine obere Schranke für die Laufzeit der Schleife.

⌘ Verzweigung

⊗ Sei $O(f_1(n))$ eine obere Schranke für den if-Zweig und $O(f_2(n))$ für den else-Zweig einer **bedingten Anweisung**. $O(f_2(n))$ ist 0 falls der else-Zweig fehlt. Dann ist $O(\max(f_1(n), f_2(n)))$ eine obere Schranke für die Laufzeit der bedingten Anweisung.

⌘ Sequenz

⊗ Seien $O(f_1(n)), O(f_2(n)), \dots, O(f_k(n))$ obere Schranken für die Anweisungen innerhalb eines **Blockes**. Dann ist $O(f_1(n)+f_2(n)+\dots+f_k(n))$ eine Schranke für die Laufzeit des Blockes. (→ Summenregel)

Laufzeitanalyse rekursiver Funktionen

⌘ Vorgehen:

- ⊗ Bilden einer induktiven Definition (Rekurrenzbeziehung) für die Laufzeit $T(n)$ einer rekursiven Funktion
- ⊗ Argument n muss während des Fortschreitens der Rekursion kleiner werden.

⌘ Beispiel (Skizze):

1. Gegeben sei folgende rekursive Funktion zur Berechnung der Fakultät:

```
int fact(int n) {  
  (1)   if (n<=1)  
  (2)       return 1;  
       else  
  (3)       return n*fact(n-1);  
}
```

2. Aufstellen der Rekurrenzbeziehung

Laufzeitanalyse rekursiver Funktionen

2. Aufstellen der Rekurrenzbeziehung:

⊕ *Basis:* ($n=1$)

- Da `fact()` keine rekursiven Aufrufe macht, gilt
 $T(1) = O(1)$

⊕ *Induktion:*

- Das Prüfen in Zeile (1) ist $O(1)$.
- Da die Bedingung in Zeile (1) nicht erfüllt ist, wird Zeile (3) abgearbeitet.
- Die Multiplikation ist $O(1)$, der Aufruf von `fact($n-1$)` benötigt $T(n-1)$.
- Vereinfachen (Summenregel) ergibt
 $T(n) = O(1) + T(n-1)$

Laufzeitanalyse rekursiver Funktionen

⊗ Lösen der Rekurrenzbeziehung durch wiederholte Substitution:

⊕ Es gilt:

$$T(n) = O(1) + T(n-1)$$

$$T(n-1) = O(1) + T(n-2)$$

$$T(n-2) = O(1) + T(n-3)$$

...

$$T(2) = O(1) + T(1)$$

⊕ Substitution:

$$T(n) = O(1) + (O(1) + T(n-2)) = 2 O(1) + T(n-2)$$

$$T(n) = 2 O(1) + (O(1) + T(n-3)) = 3 O(1) + T(n-3)$$

...

$$T(n) = (n-1) O(1) + T(1)$$

⊕ Einsetzen der Basis ergibt:

$$T(n) = (n-1) O(1) + O(1)$$

$$T(n) = O(n)$$

⊗ Anmerkung:

⊕ Das Gleichheitszeichen wird zwar missbräuchlich verwendet, jedoch hat sich diese Schreibweise eingebürgert.

Sortieralgorithmen (Teil 2)

⌘ Algorithmen

- ⊠ Bubble Sort
- ⊠ Selection Sort
- ⊠ Quick Sort
- ⊠ Merge Sort
- ⊠ Heap Sort

⌘ Aufgabe:

- ⊠ Sortieren eines Arrays von Integer-Zahlen `int[] a;`

Schnelles Sortieren mit Quicksort

⌘ Quicksort:

- ⊗ auch: Sortieren durch Partitionieren
- ⊗ rekursiver Sortieralgorithmus
- ⊗ 1962 von C. A. R. Hoare veröffentlicht
- ⊗ $O(n \log n)$



⌘ Grundalgorithmus:

```
quicksort(int[] a, int l, int r) {  
    if (r > l) {  
        int p = partition(l,r);  
        quicksort(a, l, p-1);  
        quicksort(a, p+1, r);  
    }  
}
```

⌘ Funktion `partition(...)`

⊗ erfüllt drei Bedingungen:

- ⊕ Element `a[p]` mit `p = partition(l,r)` befindet sich an seinem endgültigen Platz im Array.
- ⊕ Alle Elemente `a[l]...a[p-1]` sind $\leq a[p]$.
- ⊕ Alle Elemente `a[p+1]...a[r]` sind $\geq a[p]$.

⊗ Element `a[p]` wird als Pivot-Element (engl. für Drehpunkt, Drehzapfen) bezeichnet.

```
quicksort(int[] a, int l, int r) {
    if (r > l) {
        int p = partition(l,r);
        quicksort(a, l, p-1);
        quicksort(a, p+1, r);
    }
}
```

⌘ Rekursion nach **Teile-und-Herrsche-Strategie**:

⊗ Anschließend

- ⊕ beide Hälften auf die gleiche Weise behandeln,
- ⊕ wieder teilen usw.,
- ⊕ bis die zu sortierende Teilfolge die Länge 0 oder 1 hat.

⌘ Was ist eine sinnvolle Partitionierung?

- ⊗ Pivotelement so wählen, dass möglichst jeweils gleich große Teilfelder entstehen

```
quicksort(int[] a, int l, int r) {  
    if (r > l) {  
        int p = partition(l,r);  
        quicksort(a, l, p-1);  
        quicksort(a, p+1, r);  
    }  
}
```



⌘ Optimale Partitionierung

- ⊗ Wähle den **Median** aller zu sortierenden Elemente:
 - ⊕ Der Median ist das *Mittелеlement einer Verteilung*.
 - ⊕ weniger empfindlich gegenüber extremen „Ausreißern“ der Verteilung.
- ⊗ Berechnung des **Median**
 - ⊕ Ungerade Anzahl der Werte:
 - mittlere Wert der (sortierten) Werte.
 - Beispiel: $\text{Median}([2,3,5,7,100])=5$.
 - ⊕ Gerade Anzahl der Werte:
 - Mittelwert der beiden mittleren Zahlen.
 - Beispiel $\text{Median}([2,3,5,100])=4$.
- ⊗ Finden des Medians ist $O(n)$
- ⊗ Quicksort ist so immer $O(n \log n)$

⌘ In der Praxis:

- ⊗ Wähle irgendein Element, damit nur im *average case* $O(n \log n)$

Quicksort: quicksort(a, 1, N);

```
public void quicksort(int[] a, int l, int r) {
    if (r > l) {
        swap(a, l, (l+r)/2); // Wähle das Pivotelem. (hier: Mitte)
        int p = l;           // und tausche es mit dem linken.
        for (int i=l+1; i<=r; ++i) // Gehe alle Elem. durch und
            if (a[i] < a[l]) // hole die Elemente < Pivotelem.
                swap(a, ++p, i); // nach vorne.
        // p zeigt jetzt auf die Stelle, wo das Pivotelem. hingehört,
        swap(a, l, p); // deshalb bringe Pivotelement an endgült. Pos.

        // rekursiv die beiden Teilfelder sortieren
        quicksort(a, l, p-1);
        quicksort(a, p+1, r);
    }
}
```

Beispiel

```
quicksort(9 10 4 3 15 5 6 1 8 12 7)
```

PARTITIONIERUNG:

```
9 10 4 3 15 5 6 1 8 12 7
9 10 4 3 15 5 6 1 8 12 7
5 10 4 3 15 9 6 1 8 12 7
```

```
p      i
5 10 4 3 15 9 6 1 8 12 7
5 4 10 3 15 9 6 1 8 12 7
```

```
      p      i
5 4 10 3 15 9 6 1 8 12 7
5 4 3 10 15 9 6 1 8 12 7
```

```
      p      i
5 4 3 10 15 9 6 1 8 12 7
5 4 3 1 15 9 6 10 8 12 7
```

```
l      p      i
5 4 3 1 15 9 6 10 8 12 7
1 4 3 5 15 9 6 10 8 12 7
```

```
1 4 3
      15 9 6 10 8 12 7
```

<--- AUSGANGSSITUATION

Pivotelement ganz nach links: `swap(a, l, (l+r)/2);`

`p=l;`

`for (int i=l+1; i<=r; ++i)`

`if (a[i] < a[l])`

`swap(a, ++p, i);`

i läuft bis zum ersten Elem < Pivotel.

4 < 5, deshalb Austausch mit Element an Pos. p+1

ist das Ergebnis. p ist jetzt p+1

i läuft weiter bis zum nächsten Element < Pivotel.

3 < 5, deshalb Austausch mit Element an Pos. p+1

ist das Ergebnis. p ist jetzt p+1

i läuft weiter bis zum nächsten Element < Pivotel.

1 < 5, deshalb Austausch mit Element an Pos. p+1

ist das Ergebnis. p ist jetzt p+1

i läuft weiter bis zum Ende, kein Aust. mehr nötig

Pivotel. an endgültige Pos. bringen: `swap(a, l, p)`

<--- ERGEBNIS

ist die linke Teilfolge mit `a[l]..a[p-1] <= a[p]`

ist die rechte Teilfolge mit `a[p+1]..a[r] >= a[p]`

REKURSION:

```
quicksort(1 4 3)
```

```
quicksort(15 9 6 10 8 12 7)
```

- ⌘ Laufzeit ist abhängig von `partition(...)`
- ⌘ Fall 1: Wahl des Median als Pivotelement
 - ⊗ Gleich große Teilfelder bei jeder Partitionierung
 - ⊕ stets $O(n \log n)$
- ⌘ Fall 2: Zufällige Wahl des Pivotelements
 - ⊗ Fall 2.1: Ungünstigster Fall:
 - ⊕ bei jeder Partitionierung erwischt man das kleinste oder größte Element der Teilfolge
 - Partitionierung bringt nichts
 - ⊗ worst case: $O(n^2)$
 - ⊗ Verbesserung: Wahl des Pivotelements
 - ⊕ Wähle (zufällig) 3, 5 oder mehr Elemente und bestimme den Median

⊗ Fall 2.2: Durchschnittlicher Fall:

- ⊕ $T(n)$: Zeit für das Aufteilen in zwei Teilarrays + erwartete Zeit für die beiden Rekursionen
- ⊕ p_{pivot} : Wahrscheinlichkeit, dass Pivotelement das i -größte Feldelement ist

$$\begin{aligned}\oplus T(n) &= O(n) + \sum_{i=1}^n (p_{\text{Pivot}} \cdot (T(i-1) + T(n-i))) \\ &= O(n) + \sum_{i=1}^n \left(\frac{1}{n} (T(i-1) + T(n-i))\right) \\ &= O(n) + \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i)) \\ &= O(n) + \frac{2}{n} \sum_{i=1}^n T(i) \\ &= O(n \log n)\end{aligned}$$

Beweis durch vollständige Induktion

Sortieren durch Mischen: Mergesort

- ⌘ 1945 von John Neumann vorgeschlagen

- ⌘ externes Sortierverfahren
 - ⊗ Daten müssen nicht im Hauptspeicher vorliegen
 - ⊗ geeignet zum Sortieren sehr großer Datenmengen

- ⌘ sequentielle Methode
 - ⊗ Operiert auf einem Band (tape)
 - ⊗ Operationen:
 - ⊕ Lesen des nächsten Datums,
 - ⊕ Schreiben des nächsten Datums,
 - ⊕ Band zurückspulen
 - ⊗ Sortierte Teilsequenz auf Band heißt Lauf (run)

Grundalgorithmus von Mergesort

⌘ Solange die Daten aus mehr als einen Lauf bestehen:

⊗ **Verteile** die Läufe des Eingabebandes gleichmäßig auf zwei neue Bänder: schreibe den 1.,3.,5., ... Lauf auf das erste Band, den 2.,4.,6.,... auf das zweite Band.

⊗ **Mische:** (x = letztes geschriebenes Element)

⊕ Wenn beide Elemente $> x$

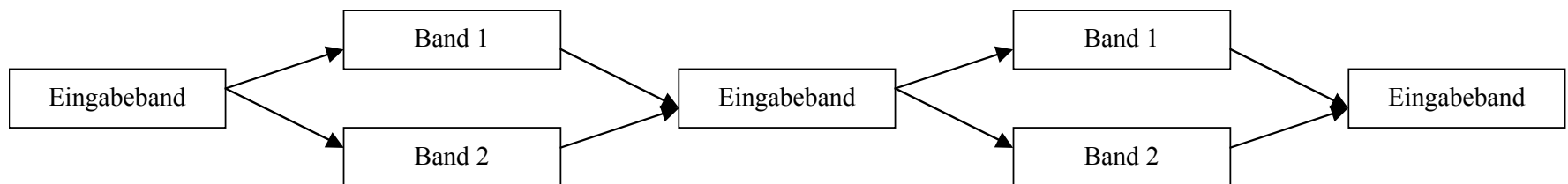
– wähle das kleinere von beiden

⊕ Wenn beide Elemente $< x$

– es entsteht ein Sprung, d.h. ein neuer Lauf, fange den neuen Lauf mit dem kleineren Element an

⊕ Wenn ein Element $\geq x$, das andere $\leq x$

– nimm das Element $\geq x$, damit der aktuelle Lauf länger wird



Beispiel

Eingabe:

75 214 73 157 169 103 213 81 5 80 142 107 129 227 201 67 57 165 41 184

Verteilen der Läufe:

Band 1: 75 214 103 213 5 80 142 201 57 165

Band 2: 73 157 169 81 107 129 227 67 41 184

Mischen der Laufpaare:

73 75 157 169 214 81 103 107 129 213 227 5 67 80 142 201 41 57 165 184

Verteilen der Läufe:

Band 1: 73 75 157 169 214 5 67 80 142 201

Band 2: 81 103 107 129 213 227 41 57 165 184

Mischen der Laufpaare:

73 75 81 103 107 129 157 169 213 214 227 5 41 57 67 80 142 165 184 201

Verteilen der Läufe:

Band 1: 73 75 81 103 107 129 157 169 213 214 227

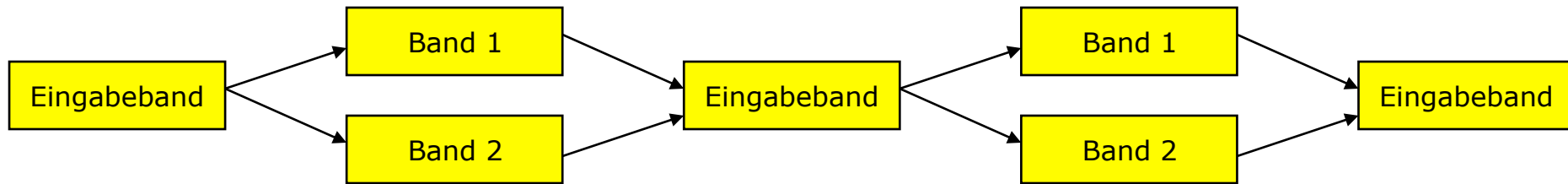
Band 2: 5 41 57 67 80 142 165 184 201

Mischen der Laufpaare:

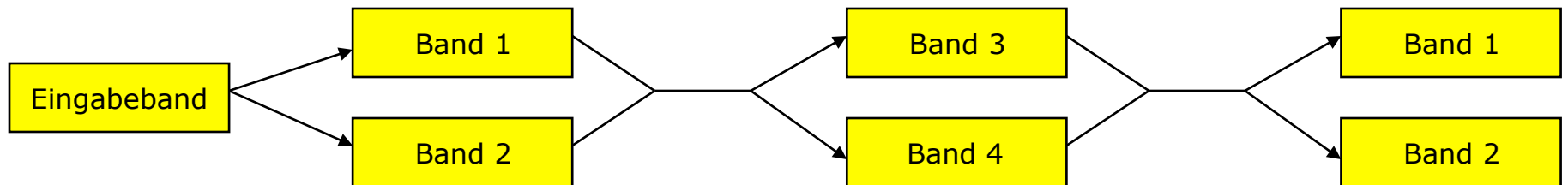
Ausgabe:

5 41 57 67 73 75 80 81 103 107 129 142 157 165 169 184 201 213 214 227

⌘ Direct Merge



⌘ Balanced Merge



- ⊠ Verteilen und Mischen werden einer Phase auf einmal realisiert
- ⊠ gemischte Läufe werden abwechselnd auf das erste und das zweite Band geschrieben.

Beispiel: Balanced Merge

Eingabe:

Band 0: 75 214 73 157 169 103 213 81 5 80 142 107 129 227 201 67 57 165 41 184

Verteilen der Läufe:

Band 1: 75 214 103 213 5 80 142 201 57 165

Band 2: 73 157 169 81 107 129 227 67 41 184

Mischen der Laufpaare und Verteilen der Läufe:

Band 3: 73 75 157 169 214 5 67 80 142 201

Band 4: 81 103 107 129 213 227 41 57 165 184

Verteilen der Läufe:

Band 1: 73 75 81 103 107 129 157 169 213 214 227

Band 2: 5 41 57 67 80 142 165 184 201

Mischen der Laufpaare:

Ausgabe:

Band 0: 5 41 57 67 73 75 80 81 103 107 129 142 157 165 169 184 201 213 214 227

⌘ Beobachtung:

- ⊗ In jeder Verteile/Mische-Phase wird die Zahl der Läufe mindestens halbiert.
- ⊗ Wenn die Daten nur noch aus einem Lauf bestehen, sind sie komplett sortiert.

⌘ In jeder Phase (Verteilen/Mischen) wird konstant viel Laufzeit pro Datenelement benötigt, also

- ⊗ $O(n)$ pro Phase.

⌘ Vor der ersten Phase haben die Läufe mindestens die Länge eins, d.h. insgesamt

- ⊗ höchstens n Läufe.

⌘ In jeder Phase halbiert sich die Zahl der Läufe, d.h.

- ⊗ nach $\log_2 n$ Phasen nur noch einen Lauf.

⌘ Gesamtlaufzeit von

- ⊗ $O(n \log n)$ im durchschnittlichen und schlechtesten Fall.

Heapsort

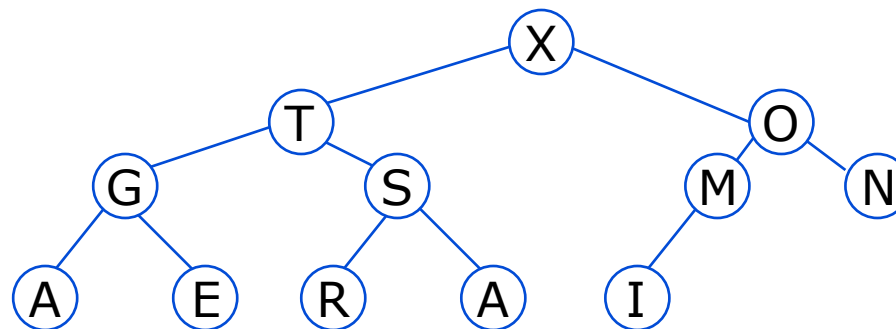
⌘ Definition: Eine Folge von Schlüsseln $a[1], a[2], \dots, a[N]$ auf der eine Ordnungsrelation (z.B. \geq) definiert ist, heißt **Heap**, wenn für alle i gilt

⊗ $a[i] \geq a[2i]$, falls $2i \leq N$ und

⊗ $a[i] \geq a[2i+1]$, falls $2i+1 \leq N$.

⌘ Beispiel:

i	1	2	3	4	5	6	7	8	9	10	11	12
$a[i]$	X	T	O	G	S	M	N	A	E	R	A	I



Heap ist besonders geeignet zur Realisierung einer Prioritätswarteschlange;
Anwendung: Job-Scheduling in Betriebssystemen)

⌘ Heap-Operationen sind effizient implementierbar: $O(\log n)$

⊗ Einfügen eines neuen Elements

⊕ `insert(...);`

⊗ Entfernen des größten Elements

⊕ `remove();`

⊗ Ersetzen des größten Elements durch ein neues Element

⊕ `replace(...);`

⊗ Löschen eines beliebigen Elements

⊕ `delete(...)`

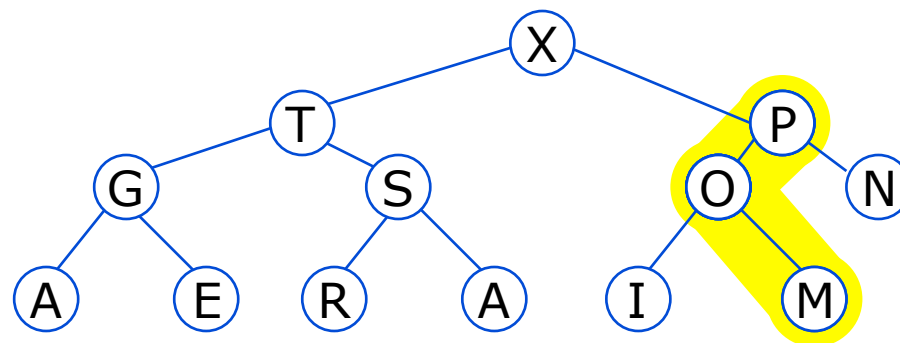
Heap: Einfügen eines neuen Elements

⌘ Algorithmus:

- ⊠ Erhöhe Größe des Heap um 1: $N++$
- ⊠ Lege einzufügendes Element in $a[N]$ ab.
- ⊠ Falls Verletzung der Heap-Bedingung ($a[N]$ ist größer als sein Vorgänger)
 - ⊕ korrigiere Verletzung durch Austausch mit Vorgänger
 - ⊕ Falls durch Austausch erneute Verletzung der Heap-Bedingung
 - Tausche mit Vorgänger
 - u.s.w., bis Heap-Bedingung erfüllt

⌘ Beispiel:

- ⊠ `insert(P)`



Heap: Einfügen eines neuen Elements

```
void heap_insert(int v) {  
    N++;          //Erhöhe Anzahl der Elemente auf  
    Heap  
    a[N]=v;      //Element hinten anfügen  
    upheap();    //Bringe Element an richtige Position  
}
```

```
void upheap() {  
    int k=N;  
    int v=a[k];          //Wert zwischenspeichern  
    a[0]=Integer.MAX_VALUE; //Markenschlüssel, Abbruch  
                          //auf jeden Fall bei a[0]  
    while(a[k/2]<v) {    //solange Heap-Bed. verletzt  
        a[k]=a[k/2];    //Elemente vertauschen  
        k=k/2;          //im Heap „hochgehen“  
    }  
    a[k]=v;             //Elem. an richtiger Position  
}
```

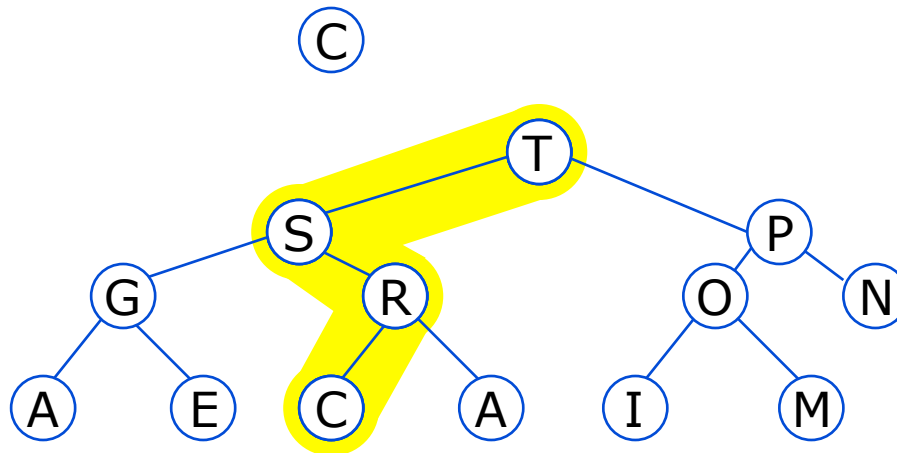
Heap: Ersetzen des größten Elements (Wurzelement)

⌘ Algorithmus:

- ⊠ Sichere den Inhalt von $a[1]$.
- ⊠ Lege einzufügendes Element in $a[1]$ ab.
- ⊠ Beseitige Verletzung der Heap-Bedingung durch Tausch des Elements mit dem *größeren* der beiden Nachfolger

⌘ Beispiel:

- ⊠ `replace(C)`



Heap: Ersetzen des größten Elements (Wurzelelement)

```
int heap_replace(int v) {
    int z=a[1];           //Wert der Wurzel sichern
    if (v >= z) return v; //Übergebener Wert > Wurzel: nichts tun
    a[1]=v;              //Element vorn einfügen
    downheap();         //an richtige Pos. bringen
    return z;
}

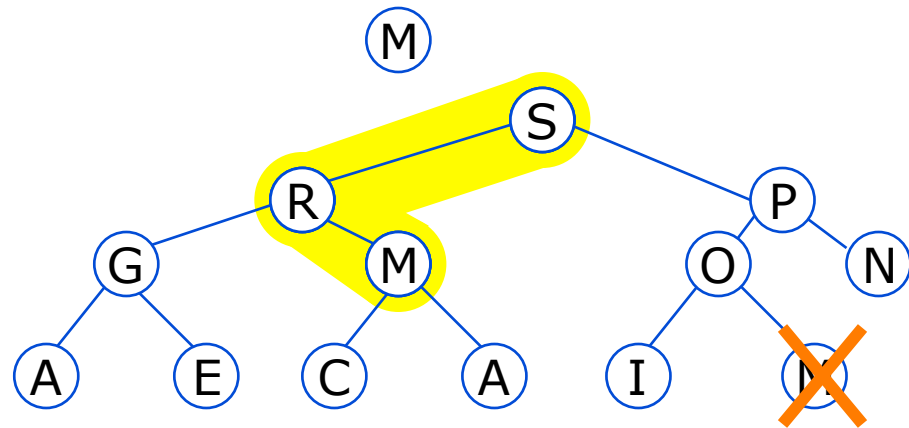
void downheap() {
    int k=1;
    int v=a[k];          //Wert der Wurzel zwischenspeichern
    while(k<=N/2) {     //den Baum hinunterwandern
        int j=2*k;
        if(j<N && a[j]<a[j+1]) j++; //Rechter Nachf. größer
        if(v>=a[j]) break; //Heap-Bed. nicht mehr verletzt, Abbruch
        a[k]=a[j];      //Elemente vertauschen
        k=j;
    }
    a[k]=v;             //Element einfügen
}
```

Heap: Entfernen des größten Elements (Wurzelement)

⌘ Algorithmus:

- ⊗ Sichere den Inhalt von $a[1]$.
- ⊗ Lege letztes Element $a[N]$ in $a[1]$ ab.
- ⊗ Verringere Größe des Heap um 1: $N--$
- ⊗ Beseitige Verletzung der Heap-Bedingung durch Tausch des Elements mit dem *größeren* der beiden Nachfolger

```
int heap_remove() {  
    int v=a[1];  
    a[1]=a[N];  
    N--;  
    downheap();  
    return v;  
}
```



⌘ Algorithmus:

- ⊗ Lege die zu sortierenden Elemente nacheinander auf dem Heap ab.
- ⊗ Entferne nacheinander das Element in der Wurzel und setze es an den Platz, der von dem sich verkleinernden Heap gerade freigegeben wurde.

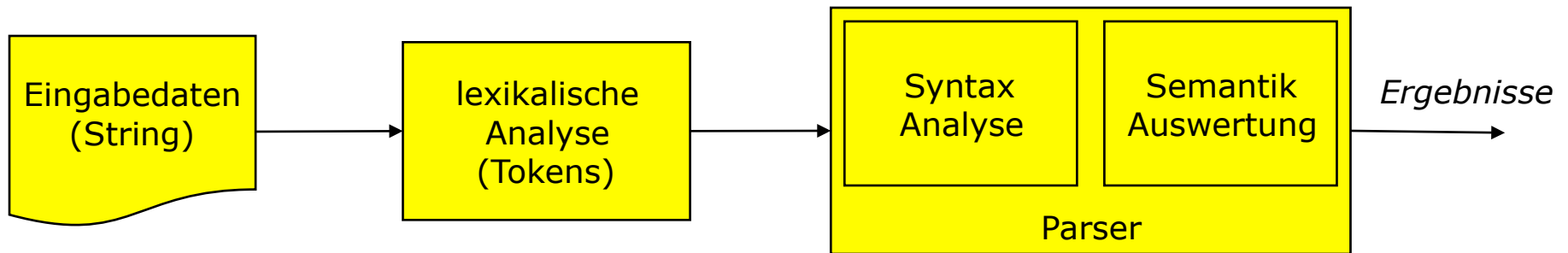
⌘ Skizze:

```
for(int i=1;i<=N;i++) heap_insert( element );  
for(int i=N;i>=1;i--) a[i]=heap_remove();
```

⌘ Leistungsfähigkeit:

- ⊗ worst und best case: $O(n \log n)$

Datenfluss in datenorientierten Programmsystemen



Quelle: nach Pomberger/Dobler 2008

Syntaxanalyse von arithmetischen Ausdrücken

⌘ Problemstellung:

- ⊗ Interpretieren von arithmetischen Ausdrücken, die einer vorgegebenen Syntax genügen sollen.

⌘ Anwendung:

- ⊗ Programmiersprachen, Compilerbau
- ⊗ Analyse von Eingaben

⌘ Beispiel:

- ⊗ Einfacher Taschenrechner
 - ⊕ erwartet zunächst Eingabe einer Zahl
 - ⊕ dann die Eingabe eines Operators (z. B. + oder -)
 - ⊕ danach wieder Eingabe einer Zahl
 - ⊕ anschließend Eingabe eines weiteren Operators
 - ⊕ danach wieder Zahl u.s.w.
 - ⊕ Ausdrücke können geklammert sein

⌘ **Syntax** einer Programmiersprache

- ⊗ beschreibt die Menge der erlaubten Zeichenketten für Programme.
- ⊗ wird heute unter Verwendung kontextfreier Grammatiken angegeben.
- ⊗ Formales Beschreibungsmittel
 - ⊕ Backus-Naur-Form (BNF)
 - ⊕ erweiterte Backus-Naur-Form (EBNF)
 - ⊕ Syntaxdiagramme

⌘ **Semantik** einer Programmiersprache

- ⊗ definiert die Bedeutung der einzelnen Sprachkonstrukte
- ⊗ wird für die meisten Programmiersprachen heute textuell beschrieben

⌘ Erweiterte Backus-Naur-Form (EBNF)

⊠ formale Beschreibung der Syntax einer Programmiersprache

⌘ Metazeichen:

Metazeichen in EBNF	Bedeutung
<code><Sprachkonstrukt></code>	Sprachkonstrukt der Programmiersprache
<code><l> ::= <r></code>	Ableitungsregel: Der linke Teil <code><l></code> wird durch den rechten Teil <code><r></code> definiert.
	alternative Definition (oder)
{ }	Das in Mengenklammern eingeschlossene Konstrukt kann 0-mal oder mehrfach vorkommen.
[]	Das in Intervallklammern eingeschlossene Konstrukt kann 0-mal oder höchstens 1-mal vorkommen.

⌘ Idee bei der Syntaxanalyse:

- ⊗ Es wird versucht, die Erfüllbarkeit eines Prädikats zu zeigen (oder zu widerlegen)
- ⊗ Für jedes Syntaxelement existiert ein Prädikat, d.h. eine boolesche Funktion, die auf dem „Eingabeband“ operiert.

⌘ Beispielsyntax:

```
<Expression> ::= <Term> |  
                <Term> "+" <Expression> |  
                <Term> "-" <Expression>  
  
<Term> ::= <Atom> |  
           "(" <Expression> "  
  
<Atom> ::= <a_positive_number>
```

- ⌘ **Semantik:** + ist Addition, - ist Subtraktion, kein automatischer Vorrang von Operatoren

⌘ Zu implementierende Funktionen:

⊗ entsprechend Syntax:

```
boolean isExpression()  
boolean isTerm()  
boolean isAtom()
```

⊗ Start der Syntaxanalyse:

```
boolean solve()
```

⊗ Hilfsfunktionen:

```
boolean eof()  
void scan()  
boolean nextIs(char c)
```

```
<Expression> ::= <Term> |  
               <Term> "+" <Expression> |  
               <Term> "-" <Expression>  
<Term>        ::= <Atom> |  
               "(" <Expression> ")"  
<Atom>        ::= <a_positive_number>
```

```
String  expressionS; // enthaelt die Expression
int     pos  = 0;    // aktuelle Position in expressionS
char    next = ' '; // naechstes Zeichen
boolean eof   = false; // signalisiert Lese-Ende

/**
 * starte Syntaxanalyse
 */
boolean solve(String s) {
    expressionS = s;
    scan(); // erstes Zeichen nach next einlesen
    return isExpression() && eof();
}

/**
 * zeigt an, ob das Ende des Ausdrucks erreicht wurde
 */
boolean eof() {
    return eof;
}
```

```
/**
 * kopiert das naechste Zeichen nach next
 */
void scan() {
    while(!eof()) {
        if ( pos < expressionS.length() ) {
            next = expressionS.charAt(pos);
            pos++;
        } else {
            eof = true;
            next = ' ';
        }
        if (next == ' ') continue; // Leerzeichen ueberlesen
        else break;
    }
}
```

```
/**
 * prüft, ob naechstes Zeichen dem uebergebenen Zeichen
 * entspricht
 */
boolean nextIs(char c) {
    if (next == c) {
        scan();
        return true;
    } else
        return false;
}
```

```
/**
 * prueft auf Atom (Folge von Ziffern)
 */
boolean isAtom() {
    String atom = "";
    while(!eof() && next >= '0' && next <= '9') {
        atom += next;
        scan();
    }
    if (atom.length() == 0)
        return false;
    else
        return true; // Atom erfolgreich erkannt
}
```

<code><Atom></code>	<code>::= <a_positive_number></code>
---------------------------	--


```
/**
 * prüft auf Expression
 */
boolean isExpression() {
    if (isTerm()) {
        switch (next) {
            case '+':
            case '-':
                scan();
                return isExpression();
        }
        return true; // Term erkannt
    } else
        return false;
}
```

```
<Expression> ::= <Term> |
                <Term> "+" <Expression> |
                <Term> "-" <Expression>
```

⌘ Beispiele für Testfälle:

⊗ Gültige Syntax:

1	Expression → Term → Atom
(1)	Expression → Term → Expression → Term → Atom
((1))	
1+2	Expression → Term + Expression
(1)+2	
((1+2)+3)	

⊗ Ungültige Syntax:

(1+2	missing)
()	missing Expression
1(2)	Syntax Error
1+2(Syntax Error

Vorausschauende Analyse und Rücksetzen

⌘ Erweiterung:

- ⊗ Manche Ausdrücke erfordern vorausschauende Analyse und ggf. Rücksetzen

⌘ Beispiel: Bedingte Ausdrücke analysieren

- ⊗ Condition "/" Expression "/" Expression
- ⊗ analog in Java: Condition "?" Expression ":" Expression

⌘ Beispielsyntax:

```
<Term> ::= <Atom> |  
         "(" <Expression> ")" |  
         "(" <Condition> "/" <Expression> "/" <Expression> )"  
<Condition> ::= <Expression> "=" <Expression> |  
              "(" <Condition> )"
```

Vorausschauende Analyse und Rücksetzen

⌘ Vorgehen:

⊠ Einführen neuer Funktionen

`int getPos()` gibt aktuelle Position im expression-String an

`void setPos(int)` setzt (zurück) auf angegebene Position

⊠ Bei Nicht-Erfolg Rücksetzen auf letzte erfolgreiche Position

⌘ Die Strategie des Rücksetzens wird auch **Backtracking** genannt.

```
int getPos() {  
    return pos;  
}
```

```
void setPos(int p) {  
    pos = p-1;  
    scan();  
}
```

Implementierung (Skizze)

```
boolean isCondition() {
    int pos = getPos();
    if (isExpression() && nextIs('=') && isExpression())
        return true;
    else {
        setPos(pos); // Ruecksetzen auf pos
        return nextIs('(') && isCondition() && nextIs(')');
    }
}
```

```
<Condition> ::= <Expression> "=" <Expression> |
               "(" <Condition> ")"
```

Implementierung (Skizze)

```
public boolean isTerm() {
    if ( nextIs('(') ) {
        int pos = getPos();
        if ( isExpression() && nextIs(')') )
            return true;
        else {
            setPos(pos); // Ruecksetzen auf pos
            return isCondition() && nextIs('/') && isExpression()
                && nextIs('/') && isExpression()
                && nextIs(')');
        }
    } else
        return isAtom();
}
```

```
<Term> ::= <Atom> |
          "(" <Expression> ")" |
          "(" <Condition> "/" <Expression> "/" <Expression> ")"
```


Wegsuche aus einem Labyrinth

⌘ Ergebnisse können sein:

⊠ Erfolg (Ausgabe der Lösungen) oder

⊠ Misserfolg (keine Lösung)

⌘ Vorgehen (Ausweg aus Labyrinth suchen):

sucheAusweg:

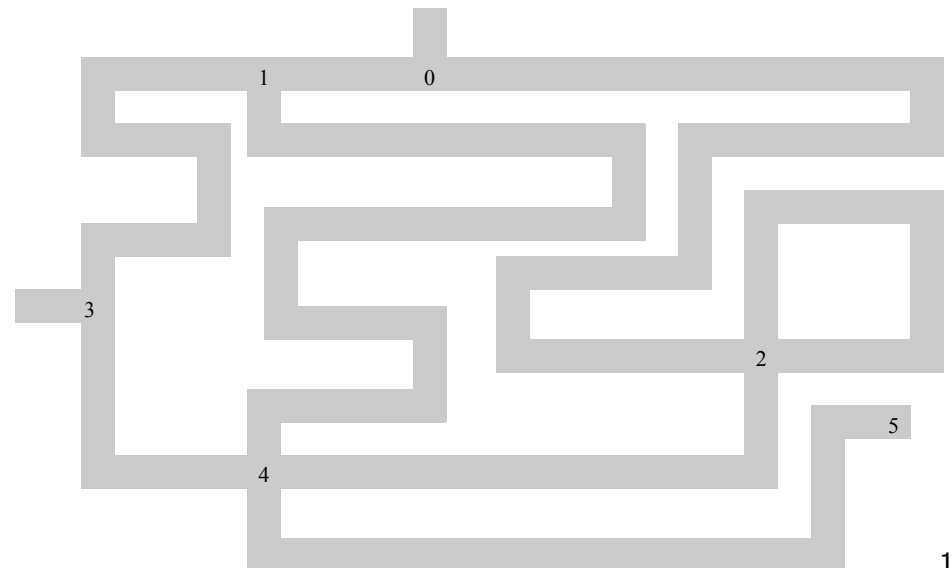
 gehe zum Ende des Gangs;

 if (draußen) Halt;

 for (alle abzweigenden Wege)

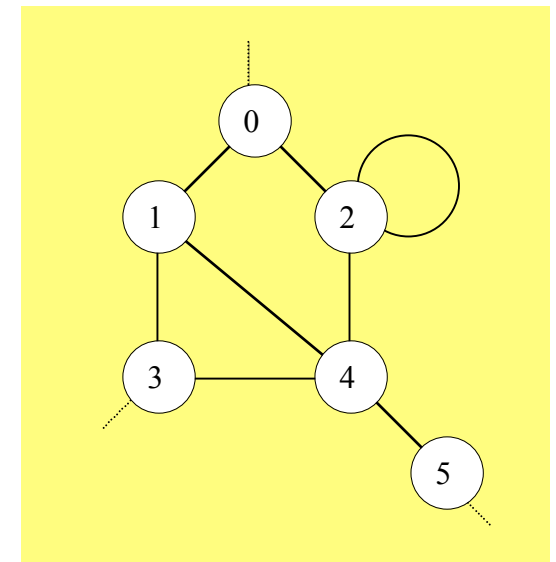
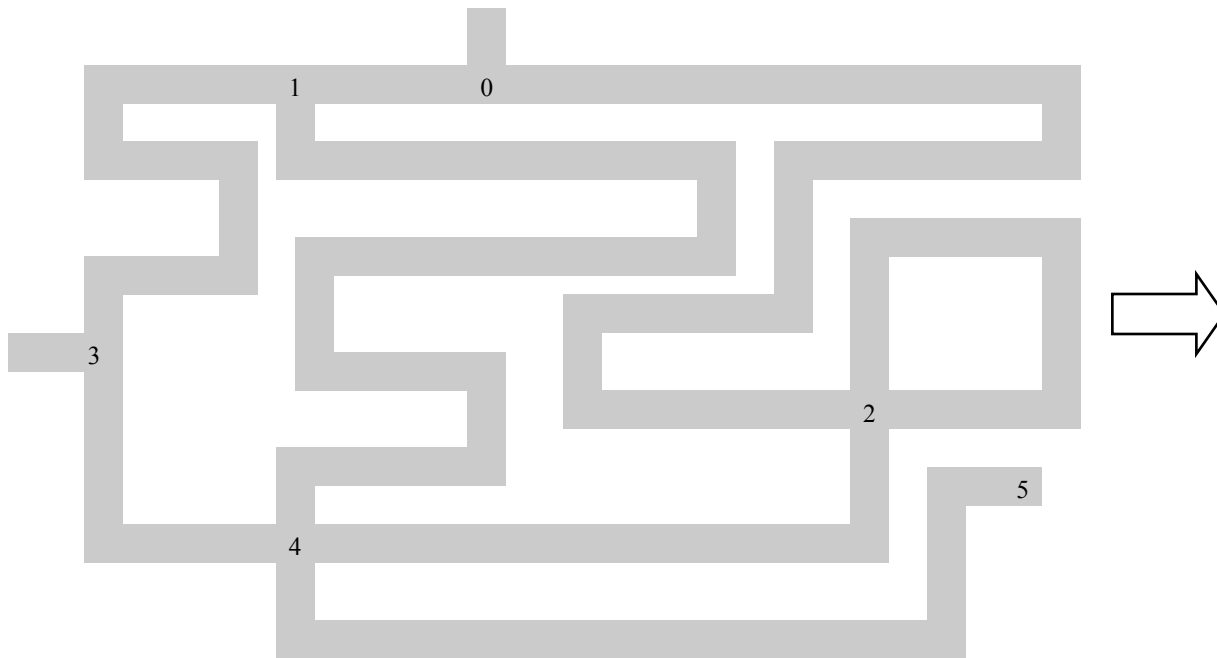
 sucheAusweg; // hochgradig nichtlinear rekursiv

 gehe den Gang zurück



Wegsuche aus einem Labyrinth

- ⌘ Wie verhindern, dass man im Kreis läuft?
 - ⊠ Ariadnefaden
 - ⊠ Markierung an besuchter Kante anbringen
- ⌘ Modellierung: ist Graph:
 - ⊠ Knoten beschreiben Abzweigungen,
 - ⊠ Kanten beschreiben Wege

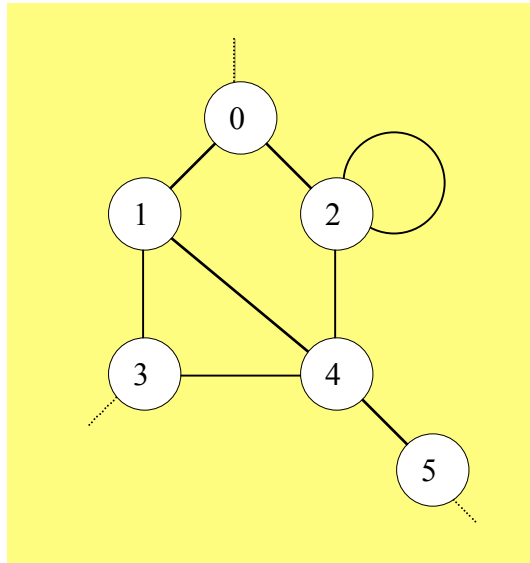


Wegsuche aus einem Labyrinth

⌘ Speichern der Knoten-Kanten-Beziehungen in einer Adjazenzmatrix:

// Adjazenzmatrix: `true` bedeutet: Kante vorhanden

```
static final boolean[][] gang = { { false, true, true, false, false, false },  
  { true, false, false, true, true, false },  
  { true, false, true, false, true, false },  
  { false, true, false, false, true, false },  
  { false, true, true, true, false, true },  
  { false, false, false, false, true, false } };
```



⌘ Ausgänge:

```
static final boolean[] Ausgang = { true, false, false, true, false, true };
```

⌘ Markierungen:

```
static boolean[] marke = { false, false, false, false, false, false };
```

Wegsuche aus einem Labyrinth

⌘ Implementierung: (Skizze)

```
/** sucht Ausweg, von Index 'von' nach Index 'nach' gehend
 */
static void sucheAusweg(int von, int nach) {
    if ( gang[von][nach] && !marke[nach] ) {
        if(DEBUG)System.out.print(""+von+"-->"+"nach+" ");
        if ( ausgang[nach] ) {
            System.out.println("Endlich draussen!");
            System.exit(1);
        }
        marke[nach] = true; // Marke setzen, um Zyklus zu verhindern
        for (int i=0; i<gang.length; i++)
            sucheAusweg(nach,i);
    }
}
```

⌘ Beispiel:

sucheAusweg(5,4) liefert 5-->4 4-->1 1-->0 Endlich draussen!

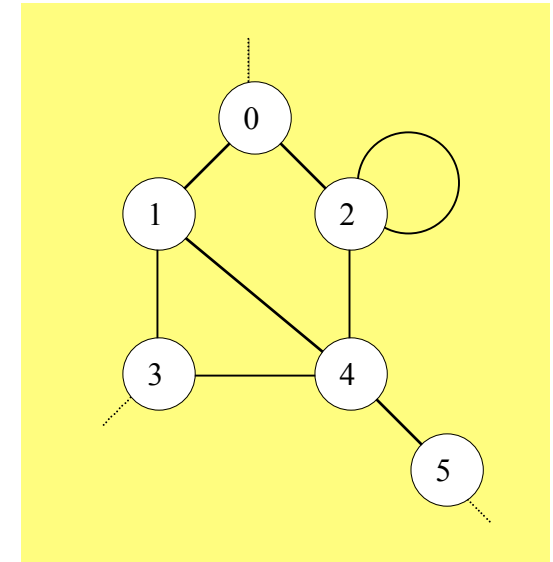
sucheAusweg(5,4) liefert

5-->4 4-->1 1-->0 Endlich draussen!

⌘ Warnung:

- ⊠ trial-and-error führt theoretisch stets zum Ziel
- ⊠ Anzahl der zu untersuchenden Alternativen kann jedoch derart groß werden, dass die Lösungen nicht mehr in zumutbarer Zeit ermittelt werden können (**kombinatorische Explosion**)

⌘ Außerdem wird nicht unbedingt die effizienteste Lösung gefunden.



Schema:

```
boolean solve(a) {
    if (legal(a)) {
        forward(a);
        if (complete(a))
            return true;
        for (all branches b)
            if (solve(b))
                return true;
        backward(a);
    }
    return false;
}
```

Falls mehrere Lösungen gewünscht:

```
void solve(a) {
    if (legal(a)) {
        forward(a);
        if (complete(a))
            print(a);
        else
            for (all branches b)
                solve(b);
        backward(a);
    }
    return;
}
```

Benutzung:

```
if (solve(initialValue))
    print(a);
else {
    ...;
}
```

Benutzung:

```
solve(initValue);
```

Backtracking (2): Der Weg des Springers

⌘ Problemstellung:

- ⊗ Kann ein Springer aus einer gegebenen Startposition nacheinander alle $n \cdot n$ Felder eines Schachbretts besuchen?

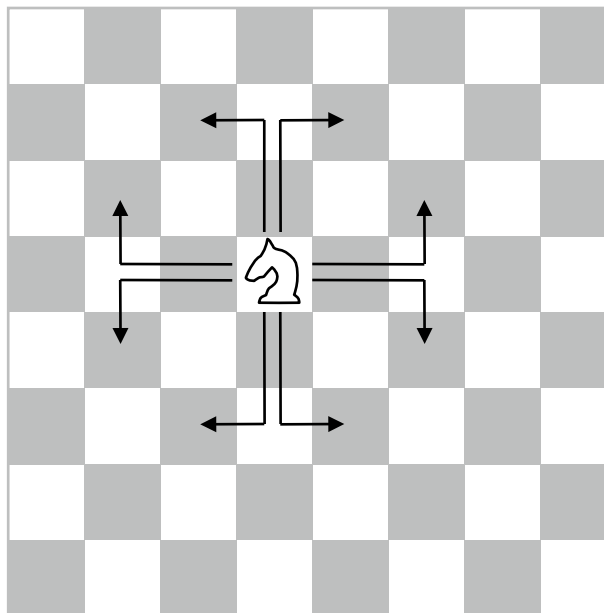
⌘ Modellierung:

- ⊗ Mögliche Züge des Springers:

```
static final int[] springerX = { 2, 1, -1, -2, -2, -1, 1, 2 };  
static final int[] springerY = { 1, 2, 2, 1, -1, -2, -2, -1 };
```

- ⊗ Schachbrett:

```
static int[][] schachbrett = new int[n][n];
```



Im zweidimensionalen Feld `schachbrett[][]` wird die Zugnummer (beginnend mit 1 gespeichert)

Backtracking (2): Der Weg des Springers

⌘ Algorithmus

- ⊠ versuche nacheinander alle möglichen Felder
- ⊠ falls ein Feld noch nicht besucht wurde, besuche es
- ⊠ versuche es von dort aus rekursiv weiter
- ⊠ falls Du in eine Sackgasse geraten bist, gehe zurück und versuche alle Alternativen
- ⊠ gehe ggf. weiter zurück und versuche von dort aus alle Alternativen
- ⊠ u.S.W.

⌘ Beispiel für eine Lösung:

01	38	59	36	43	48	57	52
60	35	02	49	58	51	44	47
39	32	37	42	03	46	53	56
34	61	40	27	50	55	04	45
31	10	33	62	41	26	23	54
18	63	28	11	24	21	14	05
09	30	19	16	07	12	25	22
64	17	08	29	20	15	06	13

⌘ Applet:

- ⊠ <http://dan.deam.org/fh/oop/springer/Dokumentation/Beschreibung.html>

```

static final int[] springerX = { 2, 1,-1,-2,-2,-1, 1, 2};
static final int[] springerY = { 1, 2, 2, 1,-1,-2,-2,-1};

static int[][] schachbrett = new int[n][n];

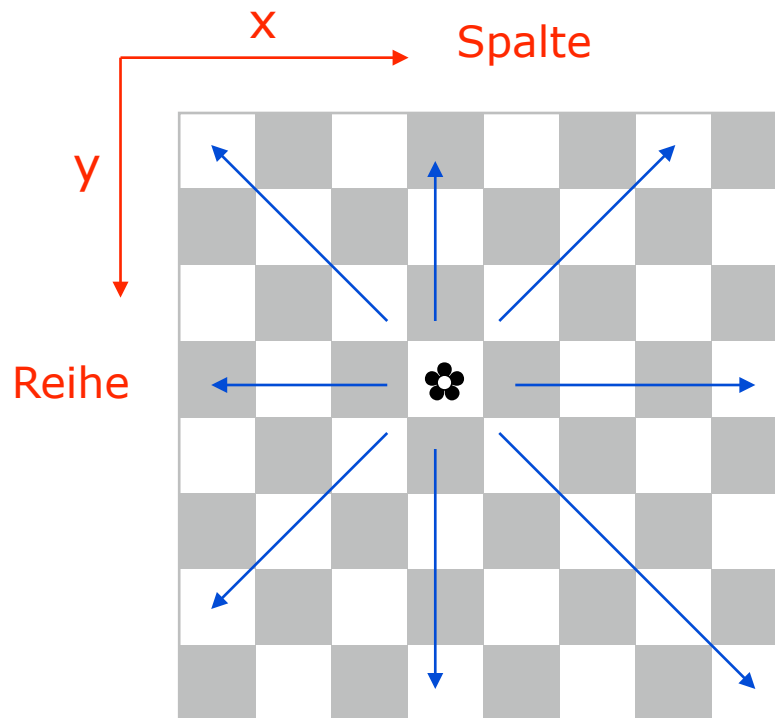
boolean versuchen ( int x, int y, int nr ) {
    schachbrett[x][y] = nr; // Feld besetzen
    if (nr == n*n) return true; // alle Felder wurden besetzt
    else
        for (int versuch=0; versuch<springerX.length; versuch++) {
            // alle moeglichen Zuege durchprobieren
            int xNeu = x + springerX[versuch];
            int yNeu = y + springerY[versuch];
            if ( // xNeu und yNeu sind auf dem Brett und
                0 <= xNeu && xNeu < n && 0 <= yNeu && yNeu < n &&
                // Feld ist noch nicht besucht worden,
                schachbrett[xNeu][yNeu] == 0 &&
                // dann besuchen
                versuchen (xNeu, yNeu, nr+1)
            )
                return true;
        }
    schachbrett[x][y]=0; // Feld zuruecksetzen
    return false;      // alle Versuche blieben erfolglos
}

```

Backtracking (3): Die acht Damen

⌘ Problemstellung:

- ⊠ Platziere n Damen so auf $n \cdot n$ -Schachbrett, dass sie sich nicht gegenseitig bedrohen



Backtracking (3): Die acht Damen

⌘ Problemstellung:

- ⊗ Platziere n Damen so auf $n \cdot n$ -Schachbrett, dass sie sich nicht gegenseitig bedrohen

⌘ Modellierung:

- ⊗ Feld `dameInDerSpalte` zeigt an, in welcher Spalte die Dame der jeweiligen Reihe steht

```
static final int n = 8;  
int[] dameInDerSpalte = new int[n]; // Werte 0 bis 7
```

- ⊗ Vermerken, in welche Reihen und Diagonalen noch frei sind:

```
// die folgenden drei Felder muessen mit true vorbelegt werden
```

```
// false gibt an, dass Reihe belegt wurde
```

```
boolean[] reihe = new boolean[n];
```

```
// false gibt an, dass die jeweilige Diagonale belegt ist
```

```
boolean[] diagonaleLinks = new boolean[2*n-1];
```

```
boolean[] diagonaleRechts = new boolean[2*n-1];
```

Backtracking (3): Die acht Damen

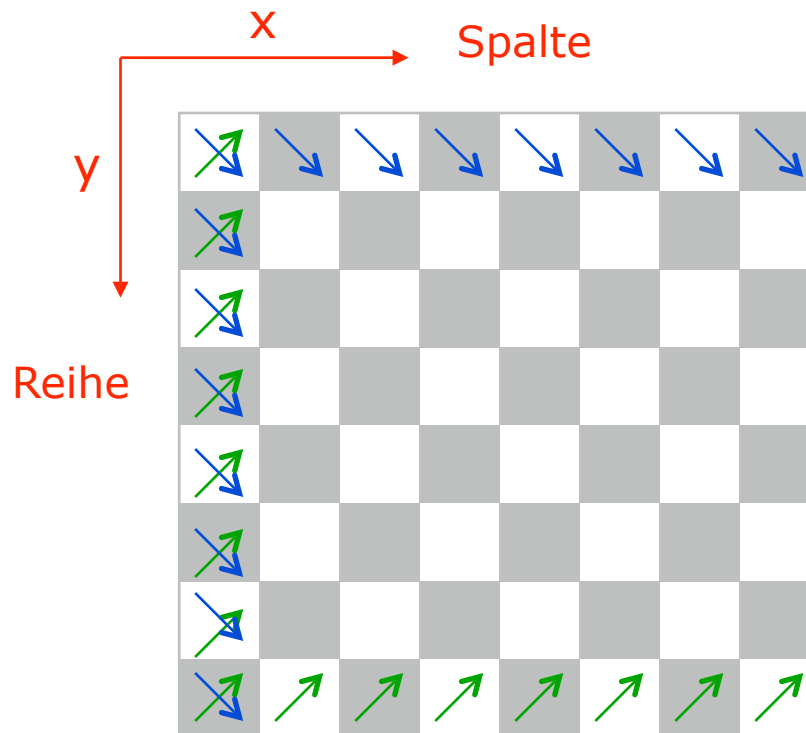
⌘ Modellierung der Diagonalen

- ⊠ Es existieren $2*n-1$ linke und rechte Diagonalen

```
boolean[] diagonaleLinks = new boolean[2*n-1];  
boolean[] diagonaleRechts = new boolean[2*n-1];
```

- ⊠ Index der jeweiligen Diagonale hängt von x und y ab

```
int links = (x+y) % (2*n-1); // Idx d. Diagonale links  
int rechts = (x-y-1+2*n) % (2*n-1); // Idx d. Diagonale rechts
```



Backtracking (3): Die acht Damen

```
boolean versuchen( int x ) {
    if (x == n) // letzte Dame wurde erfolgreich platziert
        return true;
    else // probiere alle Reihen durch
        for (int y=0; y<n; y++) {
            int links  = (x+y)          % (2*n-1); // Idx d. Diagonale links
            int rechts = (x-y-1+2*n) % (2*n-1); // Idx d. Diagonale rechts
            if (reihe[y] && diagonaleLinks[links] && diagonaleRechts[rechts]) {
                dameInDerSpalte[x]=y; // Dame in der Spalte x steht in Reihe y
                reihe[y] = diagonaleLinks[links] = diagonaleRechts[rechts] = false;
                // Reihe und zwei Diagonalen sind jetzt besetzt
                boolean erfolg = versuchen(x+1);
                if (erfolg)
                    return true;
                else // Reihe und Diagonalen wieder freigeben (Backtracking)
                    reihe[y] = diagonaleLinks[links] = diagonaleRechts[rechts] = true;
            }
        }
    return false;
}
```

Eigenschaften von Backtracking

⌘ Prinzip

- ⊗ Versuch und Irrtum
- ⊗ Tiefensuche

⌘ Lösung wird gefunden

- ⊗ Alle möglichen Lösungswege werden ausprobiert bis einer zum Ziel führt (Exhaustionsalgorithmen)
- ⊗ Wenn es eine Lösung gibt, so wird diese auch gefunden

⌘ Sehr aufwändig

- ⊗ Laufzeit wächst im worst-case exponentiell
- ⊗ Viele Probleme nicht mehr effizient lösbar

⌘ Zur praktischen Lösung mancher Probleme sind alternative Ansätze nötig

- ⊗ Heuristiken (Abwägung zwischen Optimalität und Rechenzeit)
- ⊗ dynamische Programmierung

Klassische Probleme mit hoher Komplexität

⌘ n-Damen Problem

- ⊗ in der Vorlesung ausführlich behandelt

⌘ Rundreiseproblem (traveling salesman problem)

- ⊗ kürzeste Rundreise von und zu festem Ausgangspunkt durch n Städte finden, sodass jede Stadt genau einmal besucht wird
- ⊗ z.B. Routenplanung in einer Spedition

⌘ Rucksackproblem (knapsack problem)

- ⊗ gesucht ist eine Auswahl von m aus n Gegenständen mit spezifischen Gewichten und Werten, sodass der Wert maximal wird und ein bestimmtes Maximalgewicht nicht überschritten wird
- ⊗ alle Arten von Verpackungsproblemen (LKW-Beladung, Zuschnittoptimierung, Optimierung von Anlageentscheidungen)

generische heuristische Algorithmen

- ⌘ Lösungsschemata für verschiedenartige Probleme
- ⌘ Komplexe Probleme in vertretbarer Zeit lösen
- ⌘ Keine Garantie auf Erfolg, Näherungslösungen

- ⌘ Evolutionäre Algorithmen (evolutionary algorithms - EA)
 - ⊗ Biologische Evolution als Vorbild
 - ⊗ Grundprinzipien: Mutation, Rekombination, Selektion

- ⌘ Schwarmalgorithmen (particle swarm optimization - PSO)
 - ⊗ „A single ant or bee isn't smart, but their colonies are.“
<http://ngm.nationalgeographic.com/2007/07/swarms/miller-text>

- ⌘ Simulierte Abkühlung (simulated annealing - SA)
 - ⊗ Erhitzen und Abkühlen ermöglicht neue Lösungen
 - ⊗ kann lokale Optima wieder verlassen

- ⌘ Weitere: Tabu-Suche (tabu search - TS), Ameisenalgorithmen

⌘ Rekursion (Wh.)

- ⊗ Ein Unterprogramm heißt rekursiv, wenn es sich direkt oder indirekt selbst aufruft. Auch Datenstrukturen können rekursiv definiert werden, indem in derselben Definition auf sie Bezug genommen wird in Form einer Referenz.

⌘ Beispiele für rekursive Datenstrukturen

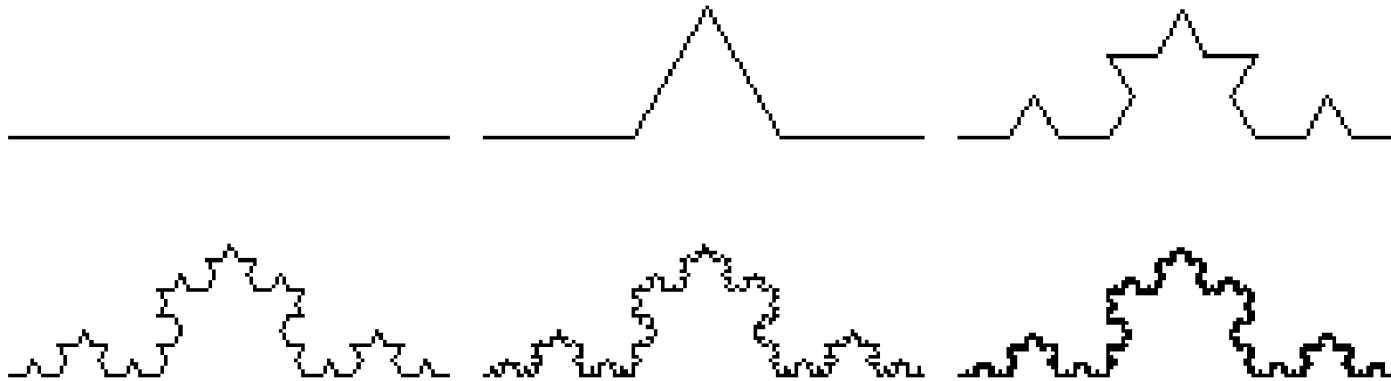
- ⊗ verkettete Listen
- ⊗ Bäume

⌘ Beispiele für rekursive Algorithmen

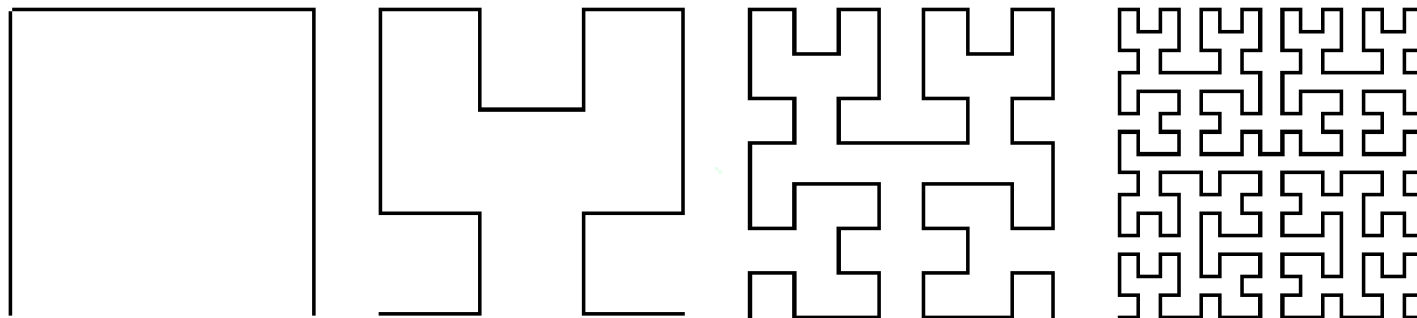
- ⊗ Berechnung der Fakultät, des ggT
- ⊗ Ackermann-Funktion, Fibonacci-Zahlen
- ⊗ Türme von Hanoi
- ⊗ Rekursive Kurven: Monster-, Schneeflocken-, Drachen-, Hilbertkurve

Beispiele für rekursive Kurven

⌘ Schneeflockenkurve: (die ersten sechs Rekursionsschritte)



⌘ Hilbert-Kurve: (die ersten vier Rekursionsschritte)



- ⌘ Rekursion und Iteration können jeweils transformiert werden.
- ⌘ Mögliche Gründe für Transformierung in rekursive Form:
 - ⊠ rekursive Formulierung und Implementierung eines Problems ist meist »elegant«
 - ⊠ In einigen Programmiersprachen existieren keine Sprachkonstrukte für Iteration (z.B. Prolog).
- ⌘ Mögliche Gründe für Transformierung in iterative Form:
 - ⊠ iterative Abarbeitung eines Problems meist **effizienter** als die rekursive
 - ⊠ Auf Maschinenebene existieren nur selten Tools zur Implementierung von Rekursion.
- **Verzicht auf Rekursion**
 - oder Rekursion muss ggf. nachgebildet werden

⌘ Allgemeine Äquivalenz für parameterlose Prozeduren

```
void whileBdoS() {  
    // rekursiv  
    if (B) {  
        S;  
        whileBdoS();  
    }  
}
```

ist äquivalent zu
<==>

```
void whileBdoS() {  
    // iterativ  
    while (B) {  
        S;  
    }  
}
```

⌘ Man unterscheidet:

- ⊕ Endrekursive Funktionen
- ⊕ Linear rekursive Funktionen
- ⊕ Nichtlinear rekursive Funktionen

- ⌘ Eine rekursive Funktion heißt **linear rekursiv**, wenn die Ausführung der Funktion zu höchstens einem rekursiven Aufruf der Funktion führt.
- ⌘ Eine rekursive Funktion heißt **endrekursiv**, wenn sie linear ist und jede Ausführung der Funktion entweder nicht zu einem rekursiven Aufruf führt oder das Ergebnis des rekursiven Aufrufs gleich dem Ergebnis der Funktion ist.

⌘ Grundschemata für Entrekursivierung *endrekursiver* Funktionen

```
aType pRekursiv( ... x ... ) {  
    ...  
    if ( condition(x) ) {  
        s1;  
        return pRekursiv( f(x) );  
    } else {  
        s2;  
        return g(x);  
    }  
}
```

<==>

```
aType pIterativ( ... x ... ) {  
    ...  
    while ( condition(x) ) {  
        s1;  
        x = f(x);  
    }  
    s2;  
    return g(x);  
}
```

Beispiel

```
aType pRekursiv( ... x ... ) {
    ...
    if ( condition(x) ) {
        s1;
        return pRekursiv( f(x) );
    } else {
        s2;
        return g(x)
    }
}
```

<==>

```
aType pIterativ( ... x ... ) {
    ...
    while ( condition(x) ) {
        s1;
        x = f(x);
    }
    s2;
    return g(x);
}
```

⌘ ggt(a,b) berechnet den größten gemeinsamen Teiler von a und b ($a, b > 0$)

```
// rekursiv
int ggt(int a, int b) {
    if (a > b)
        return ggt(a-b,b);
    else if (a < b)
        return ggt(a,b-a);
    else
        return a;
}
```

<==>

```
//iterativ
int ggtIt(int a, int b) {
    while(a != b)
        if (a > b)
            a = a - b;
        else // if (a < b)
            b = b - a;
    return a;
}
```

Beispiel

```
aType pRekursiv( ... x ... ) {  
    ...  
    if ( condition(x) ) {  
        s1;  
        return pRekursiv( f(x) );  
    } else {  
        s2;  
        return g(x)  
    }  
}
```

<==>

```
aType pIterativ( ... x ... ) {  
    ...  
    while ( condition(x) ) {  
        s1;  
        x = f(x);  
    }  
    s2;  
    return g(x);  
}
```

Beachte: S1 und S2 sind hier leer.

while(a != b) bei ggtIt() entspricht while((a > b) || (a < b)).

```
// rekursiv  
int ggt(int a, int b) {  
    if (a > b)  
        return ggt(a-b,b);  
    else if (a < b)  
        return ggt(a,b-a);  
    else  
        return a;  
}
```

<==>

```
//iterativ  
int ggtIt(int a, int b) {  
    while(a != b)  
        if (a > b)  
            a = a - b;  
        else // if (a < b)  
            b = b - a;  
    return a;  
}
```

Entrekursivierung nicht endrekursiver Funktionen

⌘ Linear rekursive Funktionen: Einbettungsmethode

- ⊠ Nicht endrekursive, aber noch linear rekursive Funktionen werden in iterative Algorithmen überführt, indem sie eingebettet werden in endrekursive Funktionen.

⌘ Vorgehen

- ⊠ am Beispiel der rekursiven Berechnung der Fakultät: $n!$ ($n \geq 0$)

⌘ Schritt 0: **Ausgangszustand:** linear, nicht endrekursiv

```
int fak(int n) {  
    if (n < 2) return 1;  
    else      return n * fak(n-1);  
}
```

- ▶ 0. Rekursiv
- ▶ 1. Einbetten
- ▶ 2. Entrekursivieren
- ▶ 3. vereinfachen

Entrekursivierung nicht endrekursiver Funktionen

⌘ **Schritt 0: Ausgangszustand:** linear, nicht endrekursiv

```
int fak(int n) {
    if (n < 2) return 1;
    else      return n * fak(n-1);
}
```

⌘ **Schritt 1: Einbettung:**

⊗ Es wird ein Akkumulator als lokale Variable definiert.

```
int fak(int n) {
    return f(n,1);
}
int f(int n, int accumulator) {
    if (n < 2) return accumulator;
    else      return f(n-1, n * accumulator);
}
```

- ▶ 0. Rekursiv
- ▶ 1. Einbetten
- ▶ 2. Entrekursivieren
- ▶ 3. vereinfachen

Entrekursivierung nicht endrekursiver Funktionen

⌘ Schritt 1: Einbettung:

⊗ Es wird ein Akkumulator als lokale Variable definiert.

```
int fak(int n) {
    return f(n,1);
}
int f(int n, int accumulator) {
    if (n < 2) return accumulator;
    else      return f(n-1, n * accumulator);
}
```

⌘ Schritt 2: Entrekursivierung von f

```
int fak(int n) {
    return f(n,1);
}
int f(int n, int accumulator) {
    while (n >= 2) {
        accumulator = n * accumulator;
        n = n - 1;
    }
    return accumulator;
}
```

- ▶ 0. Rekursiv
- ▶ 1. Einbetten
- ▶ 2. Entrekursivieren
- ▶ 3. Vereinfachen

Beachte: Zuerst kommt

```
accumulator = n * accumulator,
dann n = n - 1, da im rekursiven
Fall f(n-1, n*accumulator) das
nicht dekrementierte n für die
Multiplikation mit accumulator
benutzt wird.
```

Entrekursivierung nicht endrekursiver Funktionen

⌘ Schritt 2: Entrekursivierung von f

```
int fak(int n) {
    return f(n,1);
}
int f(int n, int accumulator) {
    while (n >= 2) {
        accumulator = n * accumulator;
        n = n - 1;
    }
    return accumulator;
}
```

⌘ Schritt 3: Vereinfachen

```
int fak(int n) {
    int accumulator = 1;
    while (n > 1) {
        accumulator = n * accumulator;
        n = n - 1;
    }
    return accumulator;
}
```

- ▶ 0. Rekursiv
- ▶ 1. Einbetten
- ▶ 2. Entrekursivieren
- ▶ 3. Vereinfachen

Entrekursivierung nicht endrekursiver Funktionen

⌘ Nichtlineare Funktionen

- ⊗ Nichtlineare, nicht endrekursive Funktionen lassen sich nach diesem Schema nicht einfach entrekursivieren.

⌘ Beispiel:

// Türme von Hanoi

```
void move (int size, Place from, Place to, Place via) {
    if (size > 0) {
        move (size-1, from, via, to);
        carry (from, to);
        move (size-1, via, to, from);
    }
}
```

⌘ Vorgehen

- ⊗ Stack verwenden, um die Zwischenergebnisse beim rekursiven Abstieg aufzunehmen

Anwendung des Stack beim Preorder-Traversieren

Wh.

```
Stack s = new Stack(max);
```

⌘ Iterativ: (mit Stack)

```
void traverse(Node t) {  
    s.push(t);  
    while(!s.isEmpty()) {  
        t=s.pop();  
        t.item.visit();  
        if (t.r != null) s.push(t.r);  
        if (t.l != null) s.push(t.l);  
    }  
}
```

⌘ Rekursiv:

```
void traverse(Node t) {  
    if (t == null)  
        return;  
    t.item.visit();  
    traverse(h.l);  
    traverse(h.r);  
}
```

⌘ Problemstellung:

- ⊗ Auffinden von Objekten mit bestimmten Eigenschaften unter vielen ähnlichen Objekten.

⌘ Beispiele:

- ⊗ Suchen eines Musters in einem String (Textsuche)

 - ⊕ `int suchen(char[] muster, char[] text)`

 - liefert Index des ersten Auftretens des Musters im Text
 - bei Nichterfolg: -1

- ⊗ Suchen eines Elementes in einer (geordneten oder ungeordneten) Datensammlung.

 - ⊕ Typische Datenstruktur: `Element` besteht aus einem sog. *Suchschlüssel* und weiteren *Komponenten*.

 - ⊕ `int search(Element[] db, <type> searchKey)`

Suchen in einer Datensammlung

⌘ Zu unterscheiden:

- ⊗ geordnete oder ungeordnete Datensammlung

⌘ Es wird nach einem Suchschlüssel gesucht. Beispiele:

- ⊗ Matrikelnummer (siehe folgende Datenstruktur)

- ⊗ Name einer Person

- ⊗ Fahrzeug-Kennzeichen

⌘ Implementierung:

```
class Element {
    int key;    // search key (z.B. Mat-nr)
    // additional components ...
    // (z.B. Name, Vorname, Fachrichtung)
}
```

Datensammlung mit MAXELEM Elementen

```
Element[] database = new Element[MAXELEM];
```

Gesucht wird ein Objekt Element, dessen Schlüssel `element.key` gleich einem Suchschlüssel `searchKey` ist.

Suchen in einer unsortierten Datensammlung

⌘ Idee (trivial):

- ⊗ Durchgehen aller Elemente solange, bis ein Element die Bedingung `database.key == searchKey` erfüllt.
- ⊗ Zum Finden aller passenden Elemente muss die gesamte Datensammlung durchgegangen werden.

⌘ Implementierung:

- ⊗ Funktion `search` gibt im Erfolgsfall (Element gefunden) den Index zurück
- ⊗ sonst: `-1`

Lineare Suche (unsortiert)

```
/**
 * Linear search in an unsorted array
 *
 * @param db array (database) to search
 * @param freeIdx first free index of db is freeIdx
 * @param searchKey key to find in db
 *
 * @return index of element or -1 (not found)
 */
int search(Element[] db, int freeIdx, int searchKey) {
    int index = 0;
    while ( (index < freeIdx) && (db[index].key != searchKey) )
        index++;
    if (index == freeIdx)
        return -1; // not found
    return index;
}
```

Lineare Suche (unsortiert)

⌘ Laufzeit

- ⊗ n sei die Problemgröße; hier: Anzahl der Elemente in der Datensammlung; alle Elemente werden gleich häufig gesucht
- ⊗ worst-case: n Vergleichsoperationen (nicht gefunden)
- ⊗ durchschnittlich: $n/2$ Vergleichsoperationen

⌘ Leichte Verbesserung: Suche durch Markieren:

- ⊗ Suchschlüssel wird in die erste freie Komponente, d.h. hinter das letzte Element der Datensammlung, kopiert
- ⊗ dadurch: Vereinfachung des Abbruchs

Suchen in einer sortierten Datensammlung

⌘ Lineare Suche auf einem sortierten Array

⊠ Algorithmus wie Suche auf unsortiertem Array

⊠ zusätzlich:

⊕ Da die Datensammlung sortiert ist, kann die Suche abgebrochen werden, wenn `database.key > searchKey` ist (aufsteigende Sortierung vorausgesetzt).

⊠ **Laufzeit:** n sei die Problemgröße; hier: Anzahl der Elemente in der Datensammlung; alle Elemente werden gleich häufig gesucht

⊕ worst-case: n Vergleichsoperationen, aber:

⊕ durchschnittlich: $n/2$ Vergleichsoperationen für »nicht gefunden« und »gefunden«

Binäre Suche auf einem sortierten Array

⌘ Strategie Teile-und-Herrsche (divide-and-conquer)

⊠ Idee:

- ⊕ Problem in einzelne Teilprobleme zerlegt, die jeweils für sich gelöst werden, wobei sich die Problemgröße Schritt für Schritt stark verkleinert.

⊠ Vorgehen:

- ⊕ Suche lässt sich beschleunigen, wenn zunächst mit mittlerem Feldelement `database[mitte]` verglichen wird.

Fall 1: `searchKey == database[...].key` → Element gefunden → Abbruch

Fall 2: `searchKey < database[...].key` → links von mitte weitersuchen

Fall 3: `searchKey > database[...].key` → rechts von mitte weitersuchen

- ⊕ Iteration des Verfahrens führt zur Halbierung des Suchbereichs nach jeder Abfrage.

```
    /**
 *   binary search in an sorted array
 *
 *   @param bd array (database) to search
 *   @param freeIdx first free index of db is freeIdx
 *   @param searchKey key to find in db
 *
 *   return index of element or -1 (not found)
 */
int searchBin(Element[] db, int freeIdx, int searchKey) {
    int left    = 0;
    int right   = freeIdx - 1;
    int middle  = (left + right) / 2;
    while((left < right) && (db[middle].key != searchKey)){
        if (searchKey < db[middle].key)
            right = middle - 1;
        else
            left  = middle + 1;
        middle = (left + right) / 2;
    }
    if (db[middle].key == searchKey)
        return middle;
    return -1; // not found
}
```

⌘ Array mit 16 Elementen, belegt mit Zahlen 0–15, searchKey=14

⌘ **Lineare Suche:** <...> markiert den Index

<0>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	<1>	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	<2>	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	<3>	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	<4>	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	<5>	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	<6>	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	<7>	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	<8>	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	<9>	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	<10>	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	<11>	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	<12>	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	<13>	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	<14>	15

database[14].key=14

⌘ **Binäre Suche:** [. . .] markiert left, middle und right

<0>	1	2	3	4	5	6	<7>	8	9	10	11	12	13	14	<15>
0	1	2	3	4	5	6	7	<8>	9	10	<11>	12	13	14	<15>
0	1	2	3	4	5	6	7	8	9	10	11	<12>	<13>	14	<15>
0	1	2	3	4	5	6	7	8	9	10	11	12	13	<14>	<15>

database[14].key=14

⌘ Laufzeit

- ⊠ n sei die Problemgröße; hier: Anzahl der Elemente in der Datensammlung; alle Elemente werden gleich häufig gesucht
- ⊠ worst-case sowie durchschnittlich: $\log n$ Vergleichsoperationen (Log. zur Basis 2) für »nicht gefunden« und »gefunden«

⌘ Probleme

- ⊠ Probleme bei mehreren Elementen mit gleichem Schlüssel (vereinf.: gleicher Elemente)
 - ⊕ Finden aller gleichen Elemente ist nicht trivial
 - i. Allg. findet man nicht das erste Element, sondern lediglich irgendeines
 - ⊕ weitere gleiche Elemente können rechts und links vom gefundenen liegen
- ⊠ Was tun, wenn die Datensammlung nicht vollständig in den Hauptspeicher passt?

⌘ Wie binäre Suche,

jedoch wird die Mitte nicht nach

```
middle = (left + right) / 2;  
        = left + 0.5*(right-left);
```

sondern nach

```
middle = left + estimate*(right-left);
```

mit

```
estimate = (key-a[left].key)/(a[right].key-a[left].key);
```

berechnet.

⌘ Mitte wird als die erwartete Position des Suchschlüssels berechnet

⊗ vorteilhaft, wenn Schlüsselwerte etwa gleichverteilt sind

⊗ Best case:

⊕ Id Id $N+1$ Vergleiche sind nötig, wenn auf N unabhängigen und gleichverteilten Zufallszahlen gesucht wird

⌘ Motivation

- ⊗ Binäre Suche auf Array war selbst im worst-case noch sehr effizient: $O(\log n)$
- ⊗ Voraussetzungen: sortierte Datensammlung und Direktzugriff
- ⊗ Geordnetes Hinzufügen im Array: worst-case: $O(n)$
 - ⊕ Finden der Einfügeposition, Platz schaffen, Einfügen
 - ⊕ ist zwar linear, aber **praktisch** relativ aufwendig

⌘ Hash-Tabelle

- ⊗ Kompromiss:
 - ⊕ Eintragen von Elementen wird **praktisch**
 - in den meisten Fällen **sehr effizient**: $O(1)$
 - bleibt aber im worst-case linear: $O(n)$
 - ⊕ Suchen wird **praktisch**
 - in den meisten Fällen ebenfalls **sehr effizient** sein: $O(1)$
 - dafür aber im worst-case aufwendiger: $O(n)$

⌘ Eintragen:

- ⊗ Aus dem Schlüssel `element.key` wird der Tabellenindex berechnet, an dem das Element `element` eingetragen wird
- ⊗ Verwenden einer **Hash-Funktion** zum Berechnen der Speicherposition (Tabellenindex): `int hash(key)`
 - ⊕ **Beispiel:** key ist eine Zeichenkette:
Summe oder Polynomdarstellung der ASCII- oder Unicode-Werte von key modulo Tabellengröße
- ⊗ Tabellengröße sollte möglichst eine Primzahl sein

element

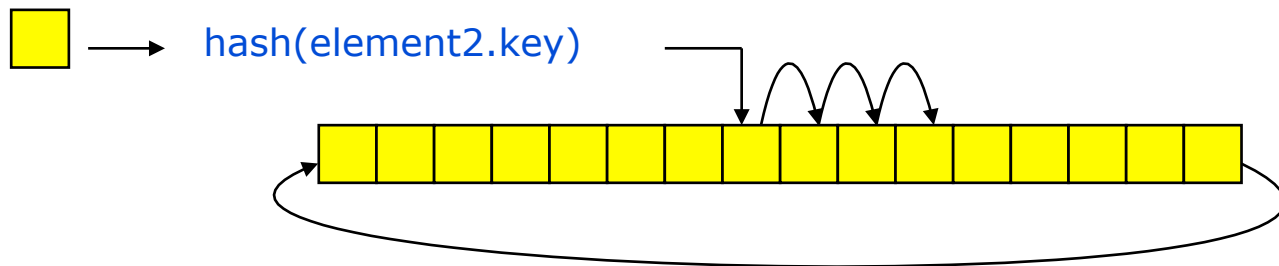


`hash(element.key)`



- ⊗ Was passiert, wenn anderer Schlüssel `element2.key` den gleichen Hash-Wert ergibt? (Annahme: `element.key != element2.key`)
 - ⊕ **Kollision** tritt auf
- ⊗ Suchen eines anderen freien Platzes mittels einer **Sondierungsstrategie**
 - ⊕ **Beispielsweise:** Lineare Sondierung: Suche (zyklisch) den nächsten freien Platz

element2



⌘ Beispiele für Sondierungsstrategien

⊗ **Lineare Sondierung:** Suche (zyklisch) den nächsten freien Platz:

$$i_k = (i_0 + k) \bmod n$$

⊗ **Problem:**

⊕ Klumpenbildung (Elemente mit gleichem Hash-Wert)

⊗ **Quadratische Sondierung:**

$$i_k = (i_0 + k^2) \bmod n$$

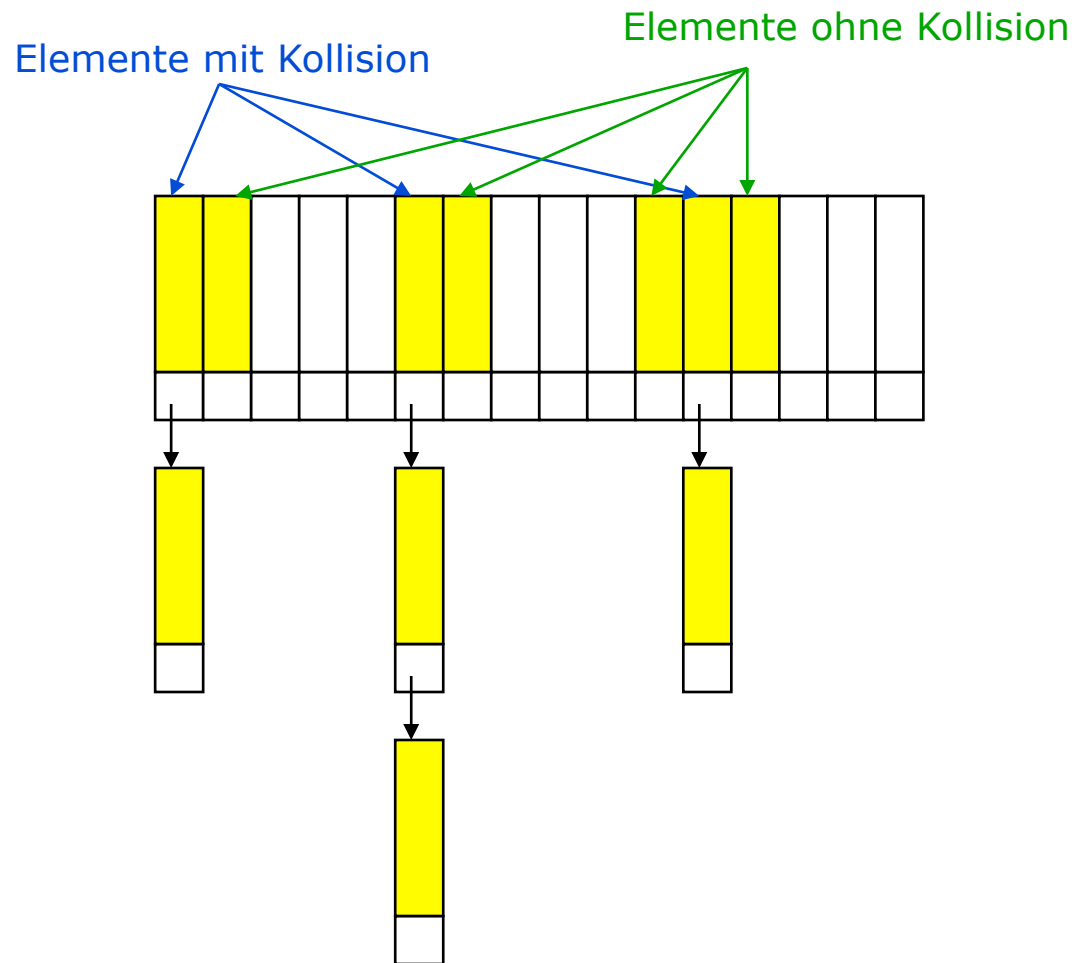
⊗ **Potentiell Problem nichtlinearer Sondierungsstrategien:**

⊕ finden möglicherweise nicht alle freien Plätze

i_0	Hash-Wert
i_k	Index beim k-ten Versuch (k=1,2, ...)
n	Tabellengröße

⌘ Beispiele für Sondierungsstrategien

⊠ Kollisionsbehandlung mit verketteten Listen



⌘ Suchen:

- ⊗ Berechnen des Hash-Wertes `hash(searchKey)`
- ⊗ wenn Element an Tabellenindex gefunden, Abbruch
- ⊗ sonst Klumpen-Elemente gemäß Sondierungsstrategie durchgehen

⌘ Beachte:

- ⊗ Es wird eine Marke für einen freien Tabellenplatz benötigt

- ⌘ Werden auch als assoziative Speicher bezeichnet
- ⌘ Java Collection API: Klasse HashMap
- ⌘ Schlüsselobjekte müssen hashbar sein, d.h.
 - ⊗ equals() implementieren
 - ⊗ hashCode() implementieren
- ⌘ Methoden
 - ⊗ Hinzufügen: put(key, value)
 - ⊗ Auslesen: get(key)
 - ⊗ Suchen: containsKey(key), containsValue(value)
 - ⊗ Löschen: remove(key)
- ⌘ Kollisionsbehandlung
 - ⊗ mit verketteten Listen

Suchen in einem Binärbaum

⌘ Jeder *Datensatz* hat neben dem

- ⊗ Schlüsselwert auch
- ⊗ linke und rechte Verkettung

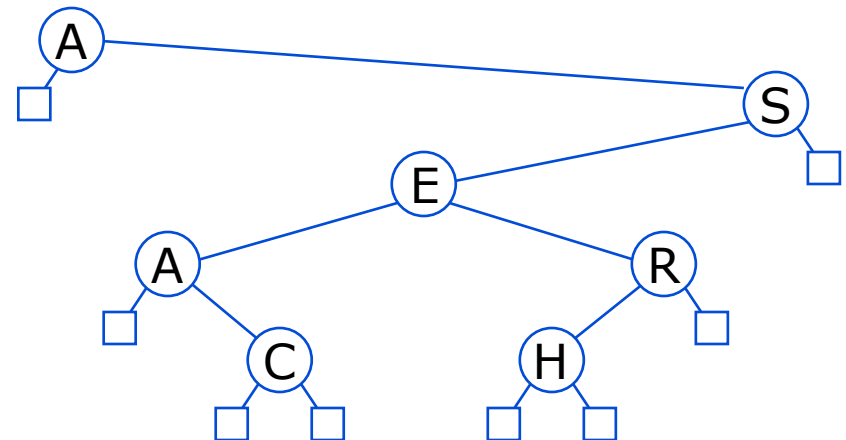
⌘ Forderung

- ⊗ Alle Datensätze mit **kleineren Schlüsselwerten** befinden sich im **linken Unterbaum**.
- ⊗ Alle Datensätze mit **größeren Schlüsselwerten** befinden sich im **rechten Unterbaum**.

⌘ Elementaroperationen

- ⊗ Suchen
- ⊗ Einfügen
- ⊗ Entfernen
- ⊗ Ausgeben

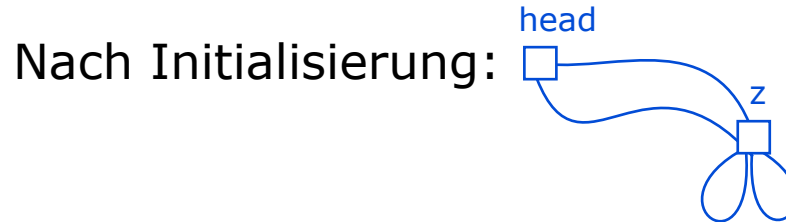
Beispiel: Suchbaum ASEARCH



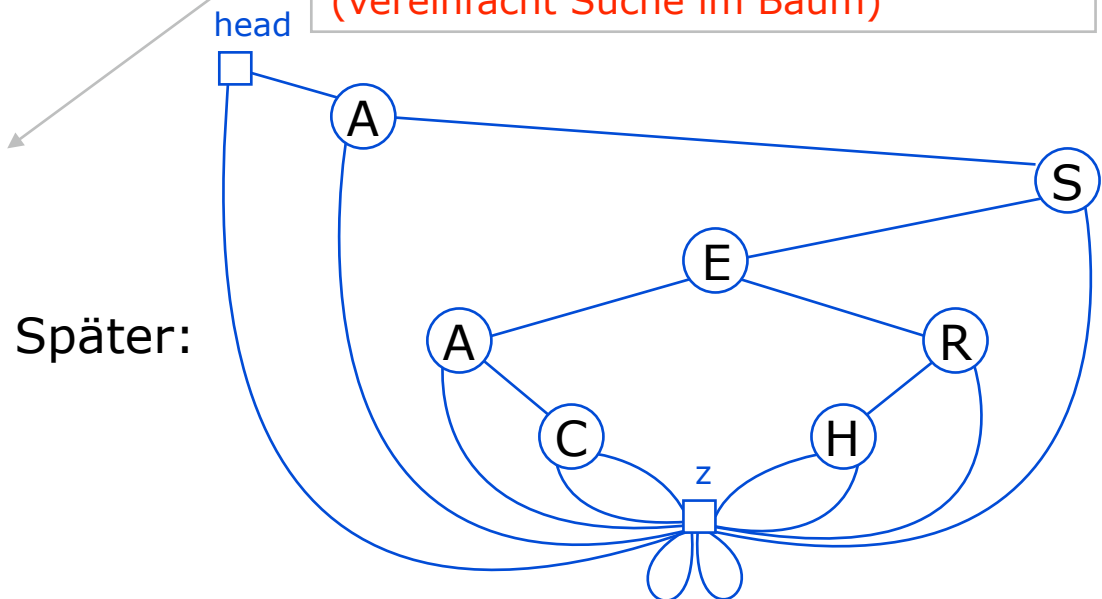
Suchen in einem Binärbaum

⌘ Lösung mit Pseudoknoten head und z

```
class Binaerbaum {  
  class Node {  
    char key;  
    Node l,r;  
  }  
  Node head,z;  
  Binaerbaum() {  
    head=new Node();  
    z=new Node();  
    head.key='\u0000';  
    head.l=z;  
    head.r=z;  
    z.l=z;  
    z.r=z;  
  }  
}
```



head mit Schlüsselwert < als alle anderen Schlüsselwerte initialisieren!
(vereinfacht Suche im Baum)



...

Suchen in einem Binärbaum

⌘ Suchen: `treesearch(char searchKey)`

- ⊗ Vergleiche `searchKey` mit Schlüsselwert `t.key` der Wurzel
- ⊗ falls `searchKey < t.key`, gehe zum **linken Unterbaum l**
- ⊗ falls `searchKey > t.key`, gehe zum **rechten Unterbaum r**
- ⊗ Wende dieses Verfahren rekursiv an.

```
Node treeSearch(char searchKey) {
```

```
    Node t=head;
```

```
    z.key=searchKey;
```

```
    do
```

```
        if (searchKey<t.key)
```

```
            t=t.l;
```

```
        else
```

```
            t=t.r;
```

```
    while(searchKey!=t.key);
```

```
    if(t!=z)
```

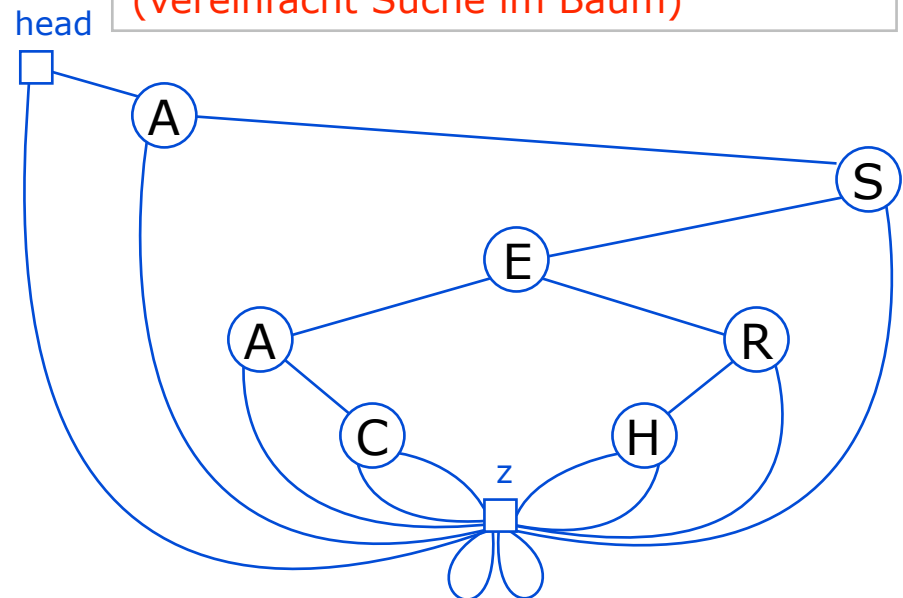
```
        return t;
```

```
    else
```

```
        return null;
```

```
}
```

head mit Schlüsselwert < als alle anderen Schlüsselwerte initialisieren!
(vereinfacht Suche im Baum)



Suchen in einem Binärbaum

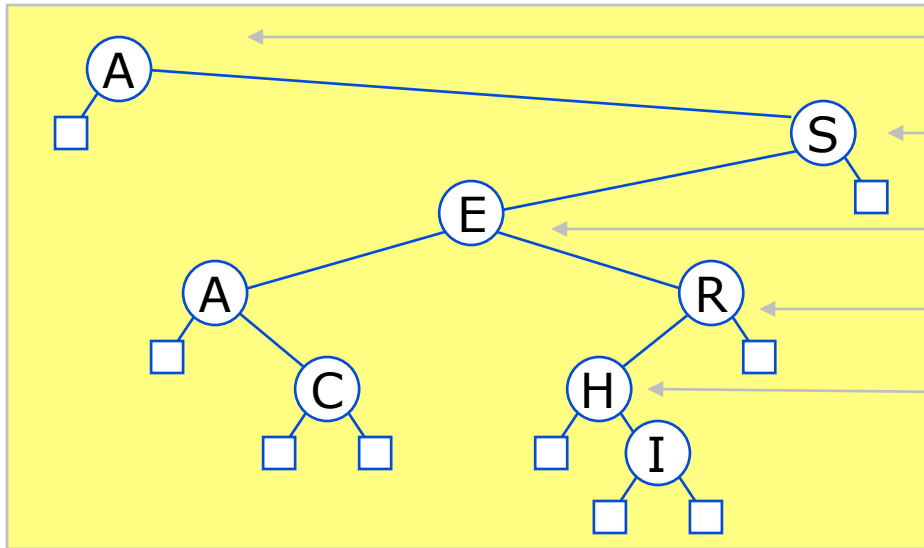
⌘ Einfügen: treesInsert(char value)

- ⊗ Führe erfolglose Suche aus und füge neuen Knoten (auf der untersten Ebene des Baumes) anstelle von z dort ein, wo die Suche beendet wurde

```
Node treeInsert(char value) {  
    Node t=head;  
    Node p; // Vorgaengerknoten (von z) fuer  
    do {    // spaetere Verkettung merken  
        p=t;  
        if (value<t.key) t=t.l; else t=t.r;  
    } while(t!=z); // unterste Ebene des Baums erreicht  
    Node x=new Node(); // neuen Knoten erzeugen  
    x.key=value; x.l=z; x.r=z;  
    if (value<p.key) p.l=x; else p.r=x; // einketten  
    return x;  
}
```

Suchen in einem Binärbaum

⌘ Beispiel: treeinsert('I')



A<I: rechten Unterbaum durchsuchen

S>I: linken Unterbaum durchsuchen

E<I: rechten Unterbaum durchsuchen

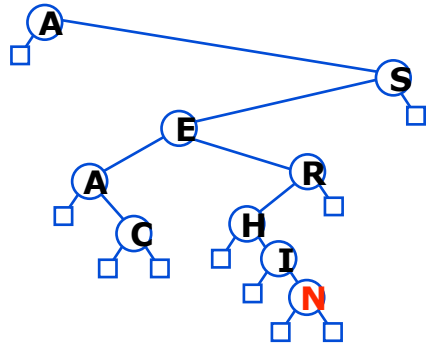
R>I: linken Unterbaum durchsuchen

H<I: (rechten Unterbaum durchs.)
t == z: neuen Knoten erzeugen
I>H: rechts verketteten

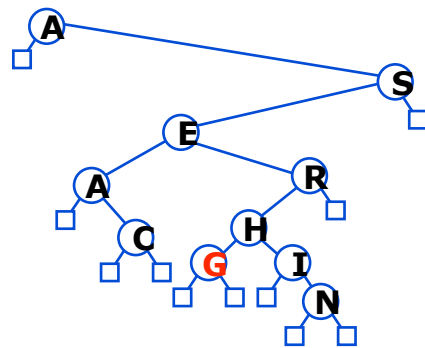
```
Node treeInsert(char value) {  
    Node t=head;  
    Node p; // Vorgaengerknoten (von z) fuer  
    do { // spaetere Verkettung merken  
        p=t;  
        if (value<t.key) t=t.l; else t=t.r;  
    } while(t!=z); // unterste Ebene des Baums erreicht  
    Node x=new Node(); // neuen Knoten erzeugen  
    x.key=value; x.l=z; x.r=z;  
    if (value<p.key) p.l=x; else p.r=x; // einketten  
    return x;  
}
```

ASEARCHI

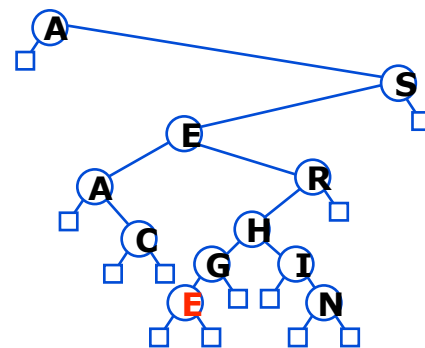
insert(N)



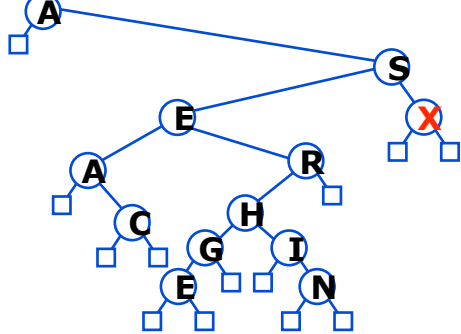
insert(G)



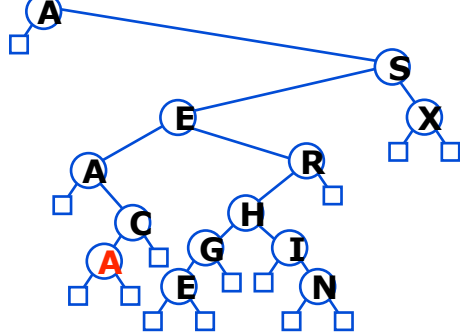
insert(E)



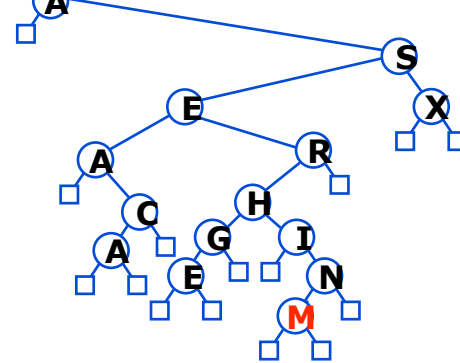
insert(X)



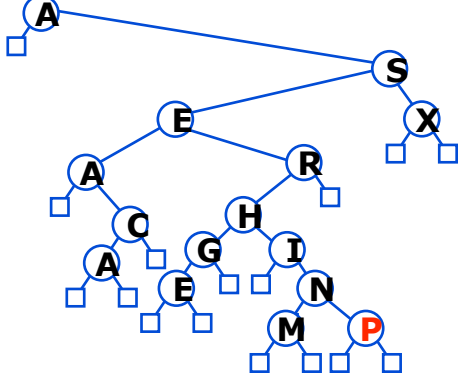
insert(A)



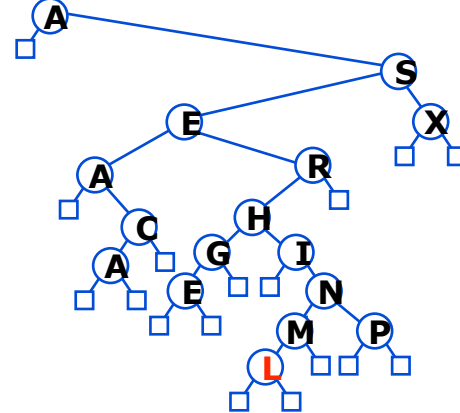
insert(M)



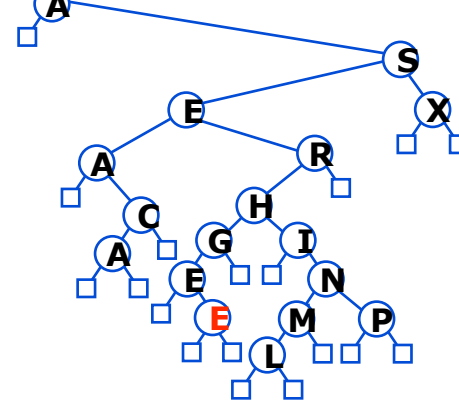
insert(P)



insert(L)



insert(E)



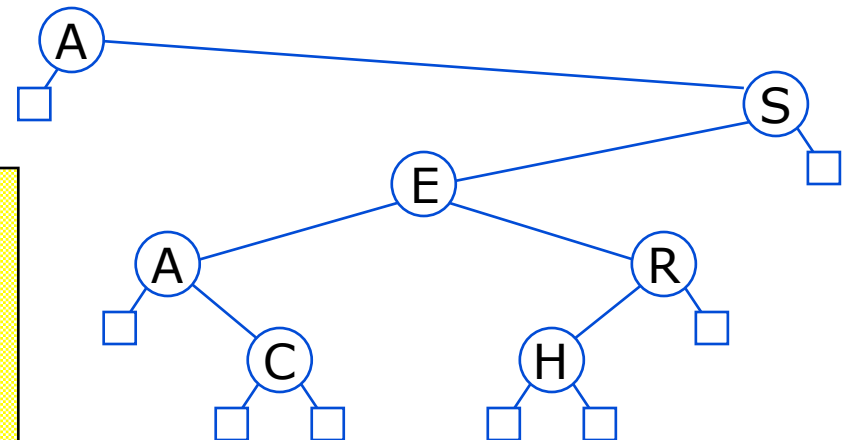
ASEARCHINGEXAMPLE

Suchen in einem Binärbaum

```
⌘ Ausgeben: treePrintSorted() {  
    traverseInorder(head);  
}
```

⌘ Sortierte Ausgabe der Elemente des Baumes erhält man gratis

```
void traverseInorder(Node t) {  
    if(t!=z) {  
        traverseInorder(t.l);  
        System.out.print(t.key);  
        traverseInorder(t.r);  
    }  
}
```

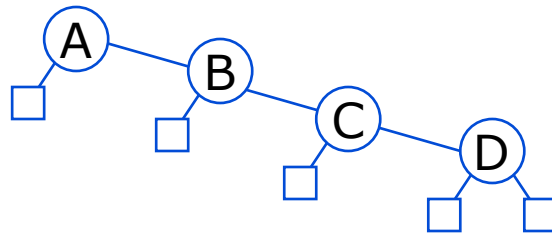


Fazit: Suchen oder Einfügen in einem binären Suchbaum erfordert durchschnittlich $2 \lg N$ Vergleiche, wenn der Baum aus N zufälligen Schlüsseln aufgebaut ist.

⌘ Binäre Bäume:

- ⊗ Im ungünstigsten Fall schlechtes Laufzeitverhalten
 - ⊕ Einfügen bereits geordneter Dateien, umgekehrt geordnete Dateien, Daten mit abwechselnd großen und kleinen Schlüsseln, vorstrukturierte Daten

⌘ Beispiel: insert(A), insert(B), insert(C), insert(D), ...



⌘ Ausgeglichene Bäume:

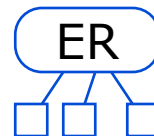
- ⊗ garantieren, dass der ungünstigste Fall nicht eintreten wird
- ⊗ Technik: Ausgleichen (balancing)

⌘ Ausgeglichene Bäume:

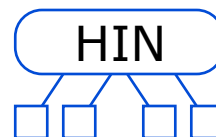
- ⊗ garantieren, dass der ungünstigste Fall nicht eintreten wird
- ⊗ Technik: Ausgleichen (balancing)

⌘ 2-3-4 Bäume

- ⊗ 3-Knoten enthält 3 von ihm ausgehende Verkettungen
 - ⊕ 1 Verkettung für alle Datensätze, die **kleiner sind** als seine beiden Schlüssel
 - ⊕ 1 Verkettung für alle Datensätze, die **zwischen beiden Schlüsseln liegen**
 - ⊕ 1 Verkettung für alle Datensätze, die **größer sind** als seine beiden Schlüssel

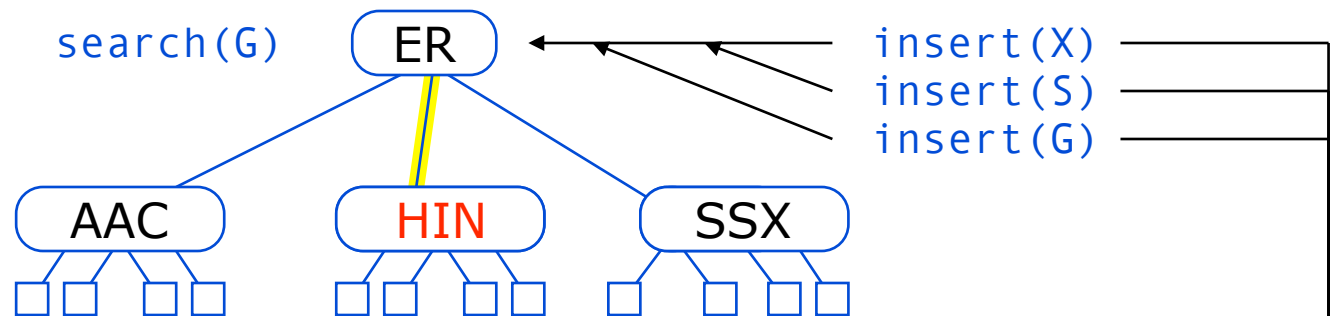


- ⊗ 4-Knoten: entsprechend: 4 Verkettungen, 3 Schlüssel



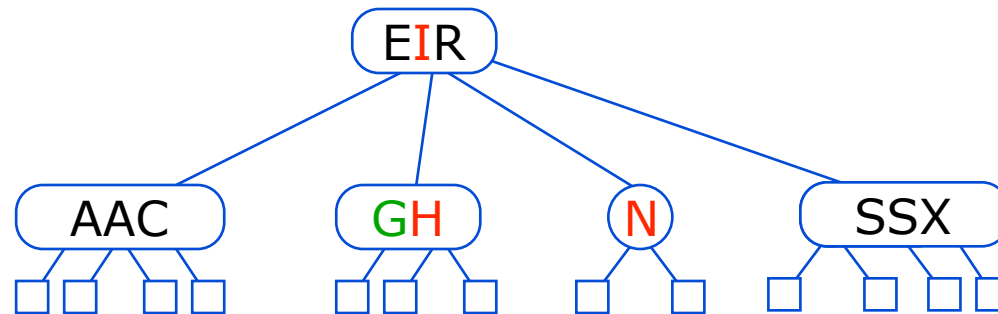
⌘ Suchen

- ⊠ Folge der Verkettung entsprechend der Konstruktionsregel eines Knotens



⌘ Einfügen:

- ⊠ Erfolgreiche Suche durchführen, dann
 - ⊕ bei 2-Knoten: 2-Knoten in 3-Knoten verwandeln
 - ⊕ bei 3-Knoten: 3-Knoten in 4-Knoten verwandeln
 - ⊕ bei 4-Knoten: **Noch verbesserungsfähige Möglichkeit:** Spalte 4-Knoten in zwei 2-Knoten auf und übergebe einen seiner Schlüssel nach oben an seinen Vorgänger
 - Was tun, wenn Vorgänger ebenfalls 4-Knoten?



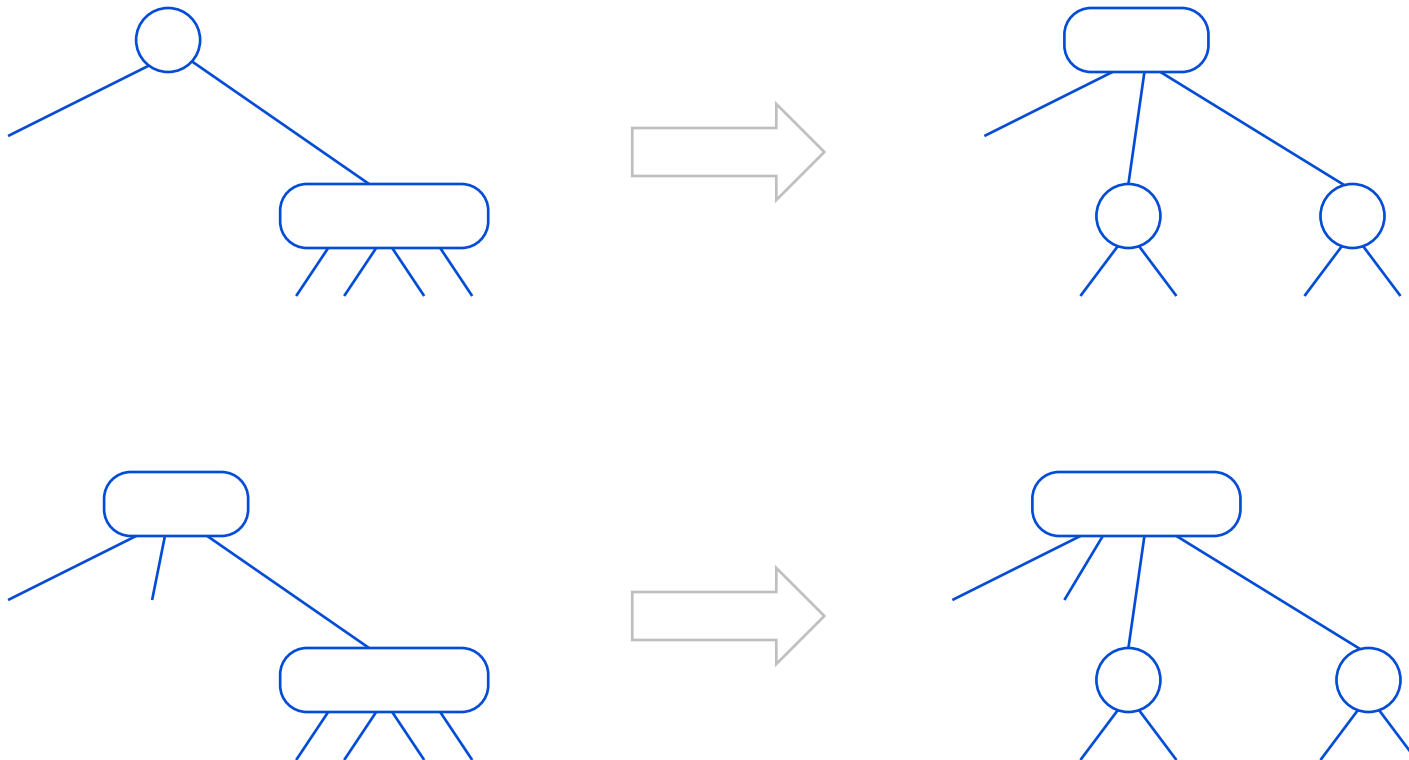
⌘ Einfügen:

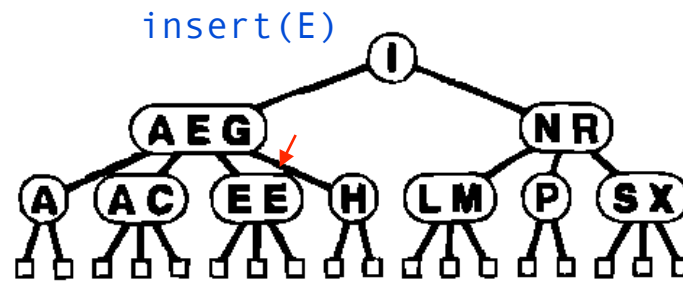
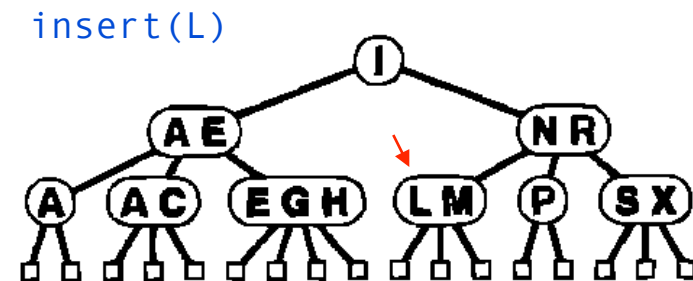
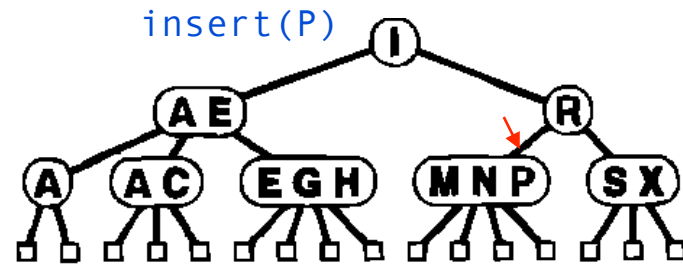
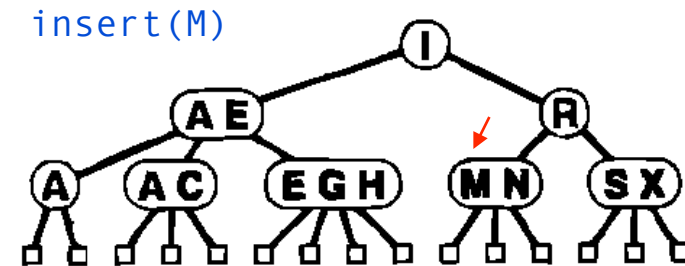
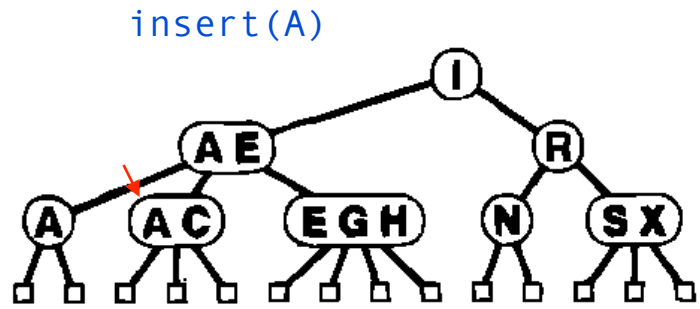
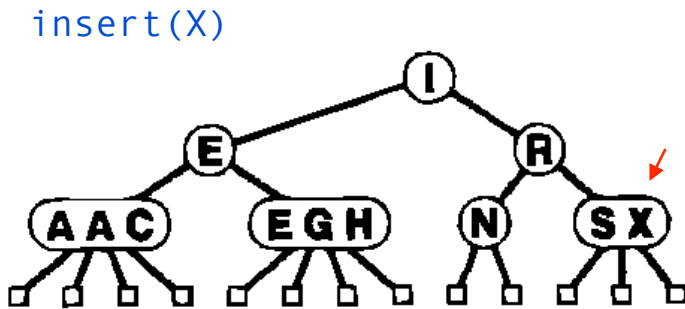
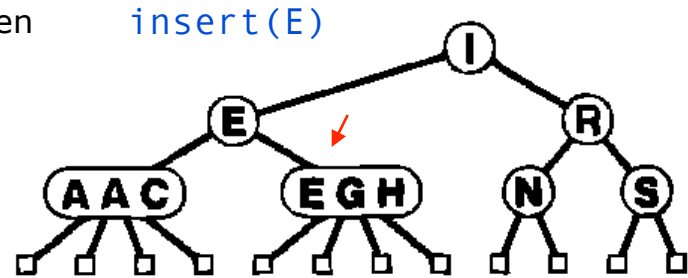
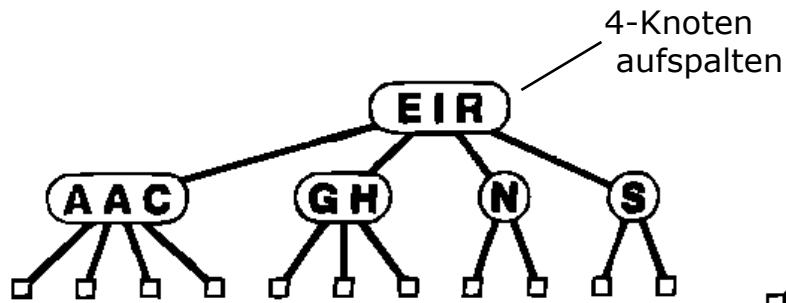
⊠ Erfolgreiche Suche durchführen, dann

- ⊕ bei 2-Knoten: 2-Knoten in 3-Knoten verwandeln
- ⊕ bei 3-Knoten: 3-Knoten in 4-Knoten verwandeln
- ⊕ bei 4-Knoten: **Noch verbesserungsfähige Möglichkeit:** Spalte 4-Knoten in zwei 2-Knoten auf und übergebe einen seiner Schlüssel nach oben an seinen Vorgänger
 - Was tun, wenn Vorgänger ebenfalls 4-Knoten?

⌘ Cleverere Methode zum Einfügen:

- ⊠ Bei erfolgloser Suche abwärts wird jeder 4 Knoten in zwei 2-Knoten aufgespalten
 - ⊕ nur rein lokale Transformationen nötig





⌘ Eigenschaften:

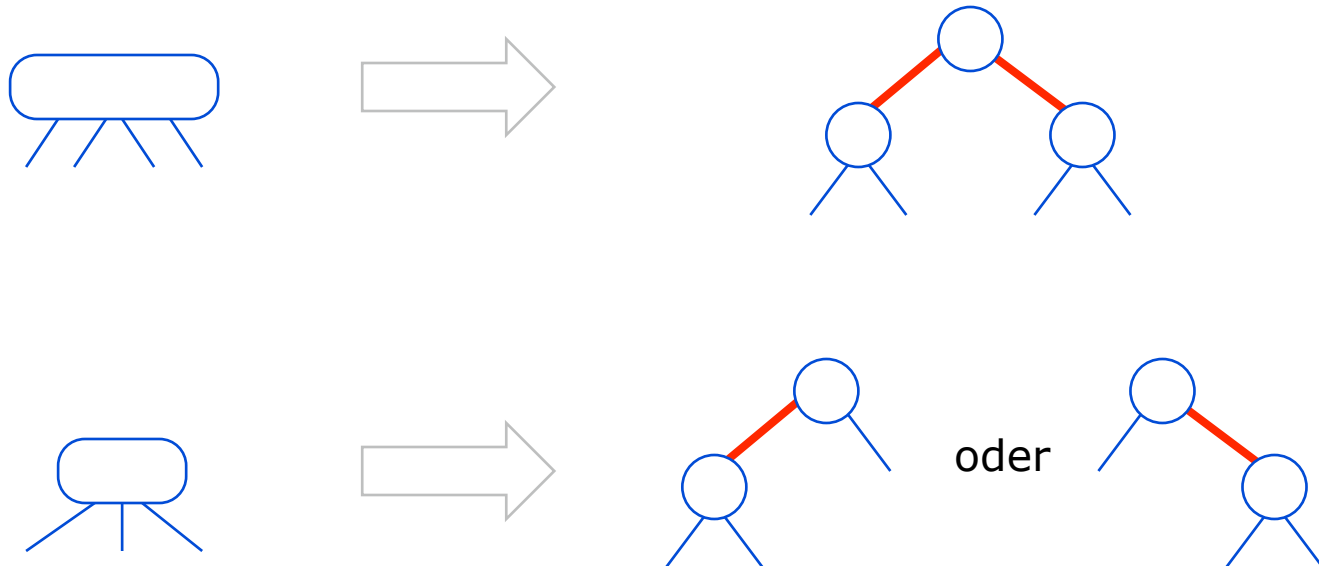
- ⊗ Beim Suchen in 2-3-4-Bäumen mit n Knoten werden niemals mehr als $\log n+1$ Knoten besucht.
- ⊗ Einfügungen in 2-3-4-Bäume mit n Knoten erfordern im *ungünstigsten Fall weniger als $\log n+1$ Aufspaltungen* von Knoten und scheinen im *Durchschnitt weniger als eine Aufspaltung* eines Knotens zu erfordern.
 - ⊕ durchschnittlicher Fall bisher nur empirisch belegt

⌘ Problem

- ⊗ Direkte Implementierungen mit 2-3-4-Bäumen sind meist weniger effizient als Implementierungen mit Binärbäumen,
- ⊗ Aber: Es ist möglich, 2-3-4-Bäume als binäre Bäume darzustellen.
- ⊗ Abwandlung der 2-3-4-Bäume:
 - ⊕ Rot-Schwarz-Bäume

⌘ Abwandlung der 2-3-4-Bäume

- ⊠ 3- und 4-Knoten werden als kleine Binärbäume dargestellt, die durch rote Verkettungen miteinander verbunden sind.



⌘ Signatur der Funktion:

⊗ int suchen(char[] muster, char[] text)

⌘ Triviale Lösung:

- ⊗ Zeichen von **text** werden der Reihe nach mit den Zeichen von **muster** verglichen
- ⊗ Bei Misserfolg Rücksprung zum Anfang und Neubeginn mit nächstem Zeichen von **text**

```
int naiveSearch(char[] pattern, char[] text) {
    for (int i=0; i < text.length - pattern.length + 1; i++) {
        boolean success = true;
        for (int j=0; j < pattern.length; j++)
            if ( text[i+j] != pattern[j])
                success = false;
        if (success)
            return i;
    }
    return -1; // no success
}
```

⌘ Beispiel:

Text:	a	b	a	b	a	a	b	b	b
	0	1	2	3	4	5	6	7	8
Muster:	a	b	a	<u>a</u>	b	b			
		a	b	a	a	b	b		
			a	b	a	a	b	b	Erfolg

```
int naiveSearch(char[] pattern, char[] text) {
    for (int i=0; i < text.length - pattern.length + 1; i++) {
        boolean success = true;
        for (int j=0; j < pattern.length; j++)
            if ( text[i+j] != pattern[j])
                success = false;
        if (success)
            return i;
    }
    return -1; // no success
}
```

⌘ Komplexität

⊗ Naiver Ansatz: $O(n \cdot m)$

n: Länge des Textes

m: Länge des Musters

⊗ Geht es auch besser?

⊗ Knuth-Morris-Pratt-Algorithmus: $O(n+m)$

⊕ Ansatz: Ein bereits gefundenes Submuster muss nicht erneut gesucht werden

⌘ Beobachtung:

- ⊗ Es muss bei Misserfolg nicht zwangsweise zum Anfang des Musters p zurückgesprungen werden
- ⊗ Wie weit zurückgesprungen werden muss, ist eine Eigenschaft des Musters, nicht des Textes

⊗ Beispiel:

```
j: 0 1 2 3 4 5 6 7
p: b a b a a b b b
next: _ 0 0 1 2 0 1 1
```

- ⊗ Bei Misserfolg an Pos. j zurückgehen auf Position $\text{next}[j]$ im Muster

Next-Tabelle für Muster b a b a a b b b

```
j: 0 1 2 3 4 5 6 7
p: b a b a a b b b
next: _ 0 0 1 2 0 1 1
```

- next[1]=0:** Wenn Misserfolg an Pos. j=1 in p festgestellt wird, muss auf Pos. j=0 im Muster zurückgesetzt werden.
- next[2]=0:** Wenn Misserfolg an Pos. j=2 in p festgestellt wird, muss auf Pos. j=0 im Muster zurückgesetzt werden.
- next[3]=1:** Wenn Misserfolg an Pos. j=3 in p festgestellt wird, kann auf Pos. j=1 im Muster zurückgesetzt werden, weil das vorhergehende Zeichen (j=2) dem ersten Zeichen in p entspricht und somit nicht noch einmal in a gefunden werden muss.
- next[4]=2:** Wenn Misserfolg an Pos. j=4 in p festgestellt wird, kann auf Pos. j=2 im Muster zurückgesetzt werden, weil die beiden vorhergehenden Zeichen (Pos. 2 und 3) den ersten beiden Zeichen (Pos. 0 und 1) in p entsprechen und somit nicht noch einmal in a gefunden werden müssen.
- next[5]=0:** Wenn Misserfolg an Pos. j=5 in p festgestellt wird, muss auf Pos. j=0 im Muster zurückgesetzt werden.
- next[6]=1:** Wenn Misserfolg an Pos. j=6 in p festgestellt wird, kann auf Pos. j=1 im Muster zurückgesetzt werden, weil das vorhergehende Zeichen (j=5) dem ersten Zeichen in p entspricht und somit nicht noch einmal in a gefunden werden muss.
- next[7]=1:** Wenn Misserfolg an Pos. j=7 in p festgestellt wird, kann auf Pos. j=1 im Muster zurückgesetzt werden, weil das vorhergehende Zeichen (j=6) dem ersten Zeichen in p entspricht und somit nicht noch einmal in a gefunden werden muss.

⌘ Next-Tabelle:

- ⊠ Bedeutung: Wenn Misserfolg an Pos. j in Muster p festgestellt wird, muss auf Pos. $\text{next}[j]$ im Muster zurückgesetzt werden.

⌘ Weitere Beispiele:

$j =$	0	1	2	3	4	5
abab	—	0	0	1		
ababab	—	0	0	1	2	3
aaaaa	—	0	1	2	3	
abbbb	—	0	0	0	0	

⌘ Berechnung der next-Tabelle

- ⊠ verwendet im [Knuth-Morris-Pratt-Algorithmus](#)

Berechnung der Next-Tabelle

```
int[] nextTable(char[] pattern) {
    final int start = -1; //constant
    int[] next = new int[pattern.length];
    next[0] = start; // Mark
    int i = 0; // index for table to build
    int j = start;
    while (i < pattern.length-1) {
        if (j==start || pattern[i]==pattern[j]) {
            // Begin of pattern or pattern matches
            i++; j++; next[i]=j;
        } else {
            j = next[j];
        }
    }
    return next;
}
```

Berechnung der Next-Tabelle: Beispiel

Beispiel:

0 1 2 3 4 5 6 7
p= b a b a a b b b

```
int[] nextTable(char[] p) {  
    final int start = -1; //constant  
    int[] next = new int[p.length];  
    next[0] = start; // Mark  
    int i = 0; // index for table to build  
    int j = start;  
    while (i < p.length-1) {  
1:         if (j==start || p[i]==p[j]) {  
2:             i++; j++; next[i]=j;  
           } else {  
3:             j = next[j];  
           }  
    }  
    return next;  
}
```

Initialisierung: i=0, j=-1, next[0]=-1

```
1: j== -1  
2: i=1, j=0, next[1]=0  
1: p[1] != p[0]  
3: j=next[0]=-1  
1: j== -1  
2: i=2, j=0, next[2]=0  
1: p[2] == p[0]  
2: i=3, j=1, next[3]=1  
1: p[3] == p[1]  
2: i=4, j=2, next[4]=2  
1: p[4] != p[2]  
3: j=next[2]=0  
1: p[4] != p[0]  
3: j=next[0]=-1  
1: j== -1  
2: i=5, j=0, next[5]=0  
1: p[5] == p[0]  
2: i=6, j=1, next[6]=1  
1: p[6] != p[1]  
3: j=next[1]=0  
1: p[6] == p[0]  
2: i=7, j=1, next[7]=1
```

ENDE, da (i<p.length)==false

⌘ Next-Tabelle:

index :	0	1	2	3	4	5
muster:	a	b	a	a	b	b
next[] = Rückschritte:	_	0	0	1	1	2

⌘ Textsuche:

Text:	a	b	a	b	a	a	b	b	b
musterIndex:	0	1	2	3	4	5	6	7	8
Muster:	a	b	a	a	b	b			

textIndex=3, musterIndex=3

Misserfolg, deshalb
musterIndex=next[3]=1

	a	b	a	a	b	b	Erfolg
musterIndex =	0	1	2	3	4	5	

textIndex++;
musterIndex++;

Knuth-Morris-Pratt-Algorithmus

```
int kmpSearch(char[] pattern, char[] text) {
    int[] next      = nextTable(pattern); // calculate next table
    int  start      = -1;
    int  textIndex   = start;
    int  patternIndex = start;
    do {
        if ( patternIndex == start ||
            text[textIndex] == pattern[patternIndex] ) {
            textIndex++; patternIndex++;
        } else {
            patternIndex = next[patternIndex]; // go back in pattern
        }
    } while ( patternIndex < pattern.length &&
              textIndex < text.length );
    return (patternIndex >= pattern.length)?
        (textIndex-pattern.length): // success
        (-1);                       // no success
}
```

Beispiel 2: Berechnung der next-Tabelle

idx:	0	1	2	3	4	5	6	7
muster:	b	a	b	a	b	b	b	b
next:	-	0	0	1	2	3	1	1

	b							(*)
		b	a	b	a			(**)
			b	a				(***)
				b	a			(****)
					b			

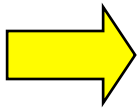
```

1: i = 0  j = -1
2: i = 1  j = 0      next [1] = 0
1: (p [1] != p [0])      (*)
3: j = next [0] = -1
1: (j == -1)
2: i = 2  j = 0      next [2] = 0
1: (p [2] == p [0])      (**)
2: i = 3  j = 1      next [3] = 1
1: (p [3] == p [1])      (**)
2: i = 4  j = 2      next [4] = 2
1: (p [4] == p [2])      (**)
2: i = 5  j = 3      next [5] = 3
1: (p [5] != p [3])      (**)
3: j = next [3] = 1
1: (p [5] != p [1])      (***)
3: j = next [1] = 0
1: (p [5] == p [0])      (****)
2: i = 6  j = 1      next [6] = 1
1: (p [6] != p [1])      (****)
3: j = next [1] = 0
1: (p [6] == p [0])
2: i = 7  j = 1      next [7] = 1
(i < 7) == false → Abbruch

```

Knuth-Morris-Pratt-Algorithmus und Next-Tabelle

- ⌘ Besonderheit der Implementierung: $next[0] = -1$
 - ⊗ ist Marke, die anzeigt, dass bereits beim ersten Zeichen im Muster Misserfolg eingetreten ist
 - ⊗ Folge: $musterIndex = -1$
 - ⊗ Somit können $musterIndex$ *und* $textIndex$ wie beim Erfolgsfall erhöht werden.



$musterIndex=0, \quad textIndex=textIndex+1$

- ⌘ Knuth-Morris-Pratt-Algorithmus:
 $O(n+m)$ im Gegensatz zum naiven Ansatz: $O(n \cdot m)$

Zusammenfassung

Algorithmus	Komplexität	Strategien, Datenstrukturen, etc.
Bubble, Selection, Insert sort	quadratisch	Feld, (Liste)
Merge sort	$O(n \log n)$	Dateien (extern)
Quick sort	$O(n \log n)$	Teile und Herrsche, Rekursion, Feld
Heap sort	$O(n \log n)$	Binärer Baum
Syntaxanalyse	(exponentiell)	Versuch-Irrtum, (Backtracking)
Wegsuche, Springer, Acht Damen	exponentiell	Versuch-Irrtum, Backtracking
Lineare Suche (ungeordnet)	linear für Suchen, konstant für Einfügen und Löschen	Liste, Feld, Dateien
Binäre Suche (geordnet)	logarithmisch für Suchen, linear für Einfügen und Löschen	Feld, Teile und Herrsche
Hashing (teilw. geordnet)	linear für Suchen, Einfügen und Löschen	Feld, Liste
Baumsuche (geordnet)	logarithmisch für Suchen, Einfügen und Löschen	Baum