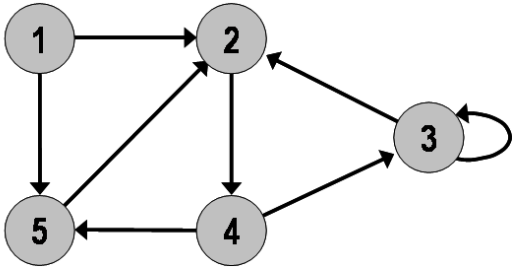
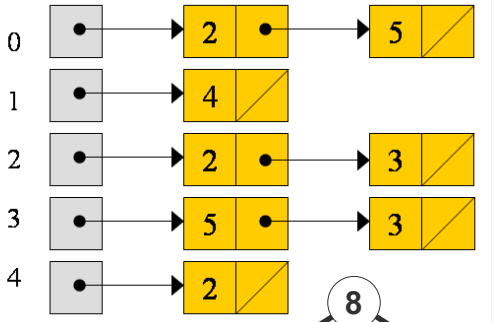
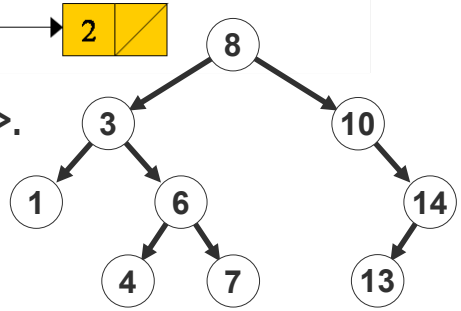
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

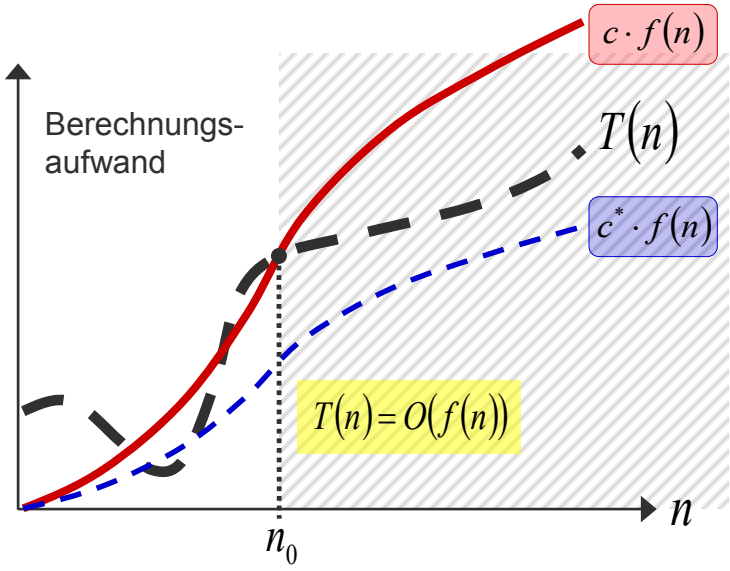
FACULTY OF
SCIENCE
Fachbereich Informatik







<Zahl> ::= <Ziffer> | <Ziffer> <Zahl>.
<Ziffer> ::= “0“ | “1“.
<Op> ::= “*“ | “/“ | “+“ | “-“.



Informatik II – SoSe 2016

Hendrik Lensch, hendrik.lensch@uni-tuebingen.de

X II. DYNAMISCHE DATENSTRUKTUREN

1. Dynamische Arrays und lineare Listen
2. Stapel und Schlangen
3. Bäume
4. Graphen

Lernziele

Bisher wurden Datenmengen in statischen Feldern repräsentiert. Für viele Anwendungen ist es sinnvoll, die entstehenden Daten dynamisch zu repräsentieren und zu verwalten. Dabei hat die Struktur des zu lösenden Problems Einfluss auf die Art der Repräsentation.

Sie sollten in der Lage sein …

- dynamische Datenmengen mit unbekannter Größe mit *Arrays* zu verwalten;
- für dynamisch generierte Daten verkettete Listen (in einem objektorientierten Ansatz) zu verwalten und elementare Operationen auf diesen Listenstrukturen anzugeben;
- für spezielle Anforderungen einfach und doppelt verkettete Listen und deren Eigenschaften zu bewerten;
- für die Verwaltung von Datenmengen mit beschränktem Zugriff, d.h. mit *last-in-first-out* (LIFO) Zugriff bei Stapeln (*stacks*) oder mit first-in-first-out (FIFO) Zugriff bei Schlangen (*queues*), zu spezifizieren;
- Baumstrukturen zu definieren und für gegebene Anwendungen zu implementieren sowie hierauf geeignete Operationen zu realisieren;
- für verschiedene Anwendungsfragen Graphen zu definieren und in geeigneten Datenstrukturen zu implementieren sowie auch hierauf geeignete (elementare) Operationen anzugeben.

Voraussetzungen

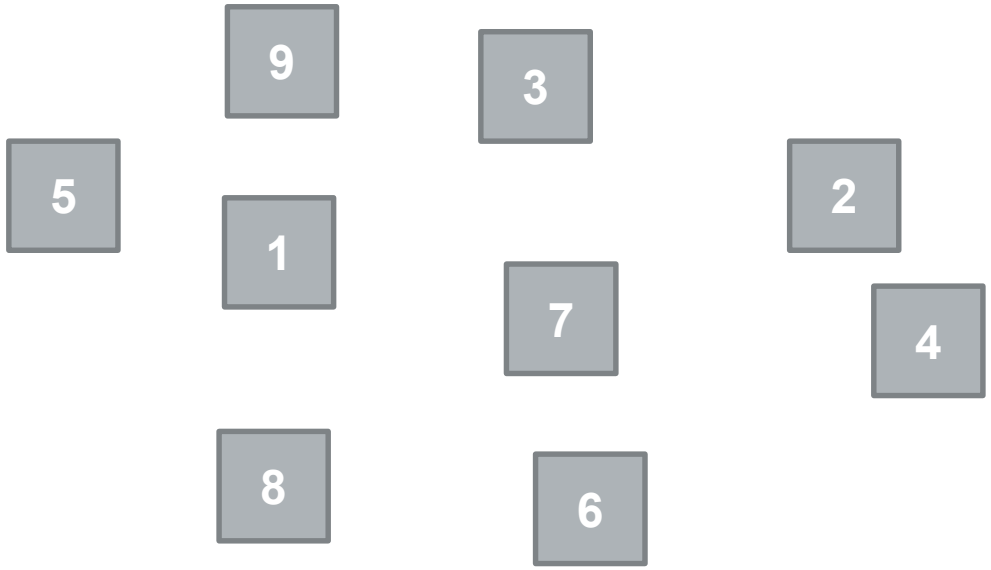
Der Umgang mit den sprachlichen Werkzeugen (in Java) wird vorausgesetzt, insbesondere die Verwendung von Arrays, das dynamische Erzeugen von Objekten als Instanzen von Klassen, die die verschiedenen Datenstrukturen implementieren.

DYNAMISCHE ARRAYS UND LINEARE LISTEN

- Dynamische Arrays
- Lineare Listen
- Einseitig verkettete Listen als Objekte
- Operationen auf einfach verketteten Listen
- Doppelt verkettete Listen (doubly-linked lists)

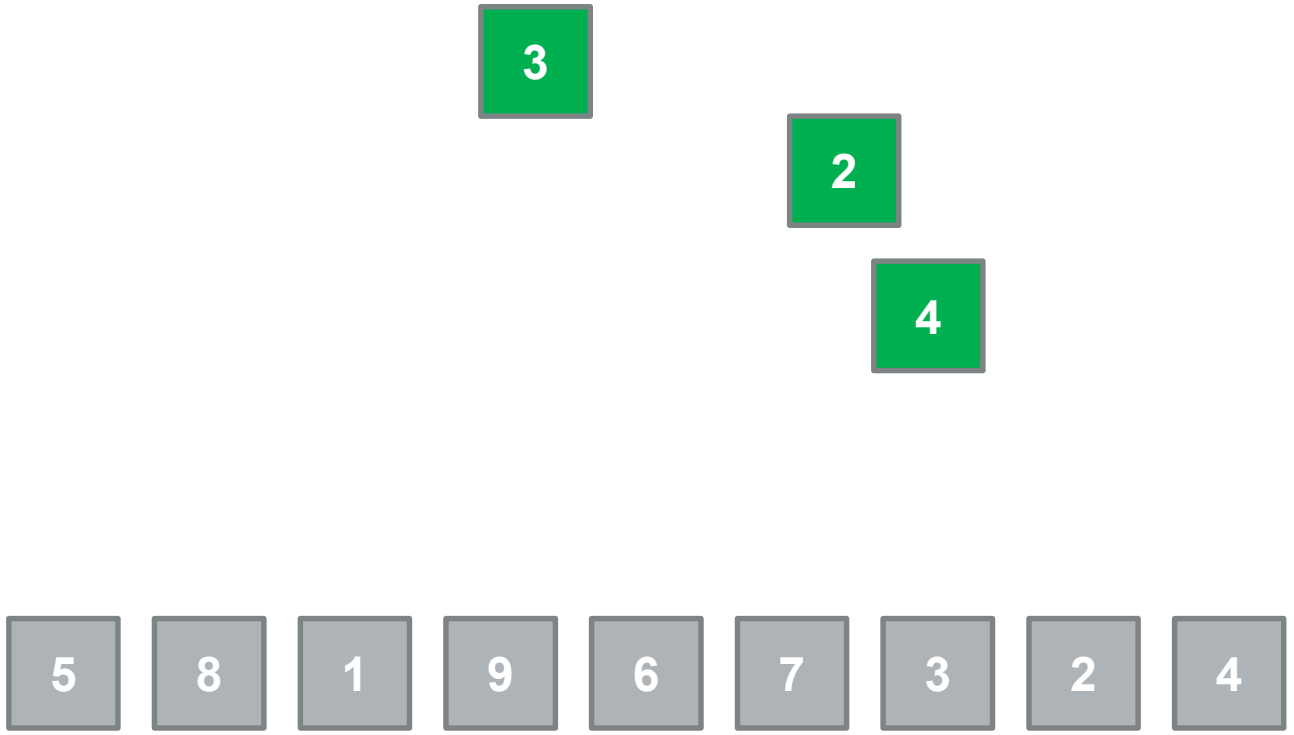
Motivation - Dynamische Operationen auf Daten

- Ungeordnete Menge an Daten



Motivation - Dynamische Operationen auf Daten

- Füllen eines Arrays
  - Einfügen am Ende



Motivation - Dynamische Operationen auf Daten

- Löschen / Entfernen am Ende

Motivation - Dynamische Operationen auf Daten

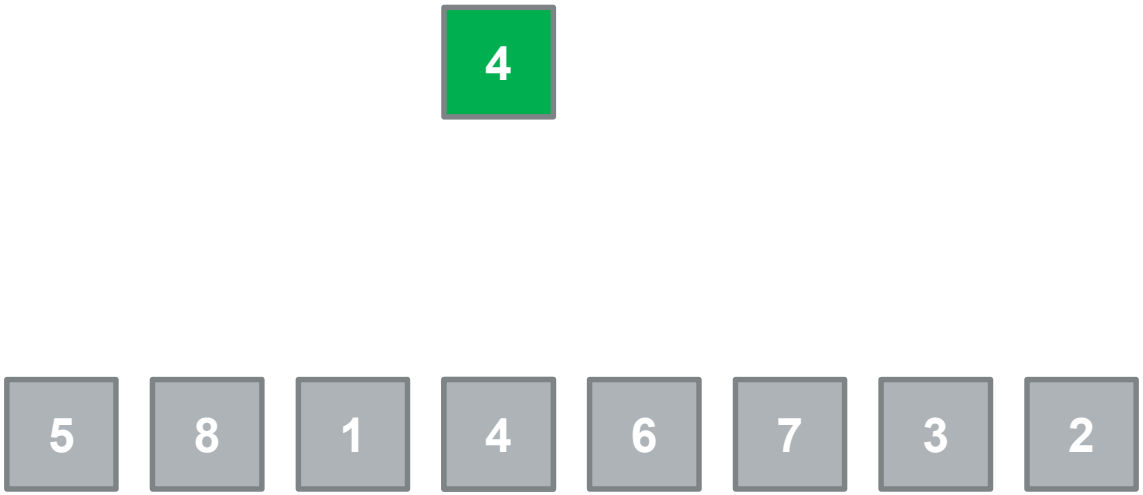
- Einfügen am Anfang
  - im Array: Verschieben aller anderen Elemente

Motivation - Dynamische Operationen auf Daten

- Löschen am Anfang
  - im Array: Verschieben aller anderen Elemente

Motivation - Dynamische Operationen auf Daten

- Einfügen mittendrin
  - im Array: Verschieben aller Elemente dahinter



Motivation - Dynamische Operationen auf Daten

- Löschen mittendrin
  - im Array: Verschieben aller Elemente dahinter

Dynamische Arrays

Partiell gefüllte Datenfelder

- Die **Anzahl der Elemente** eines Arrays ändert sich oft zur Laufzeit.
- Verwaltung von Daten mit linearer Ordnung
  - Es wird eine **maximale Anzahl von Zeichen** vorgegeben (die nicht überschritten werden darf) und für die dann Speicher allokiert wird.
  - Die **aktuelle Anzahl** der Elemente in dem Array („Füllstand“) wird durch eine Variable, z.B. `count`, repräsentiert.

Dynamische Arrays

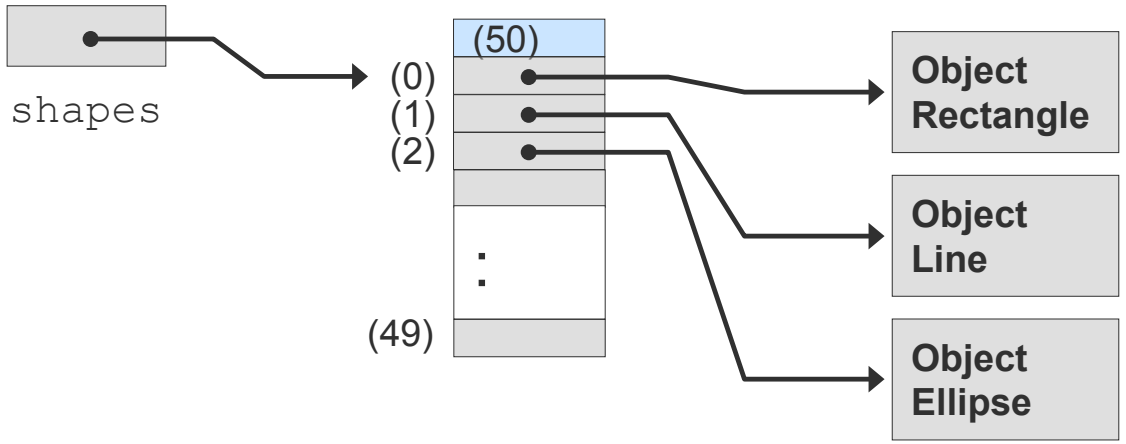
Anwendungsbeispiele:

- Ausgabe einer Zahlenfolge in umgekehrter Reihenfolge aus einem int-Array,
  `int[] numbers = new int[100];` (vgl. frühere VL)

- Verwaltung der Anzeigeliste von geometrischen Formen einer Klasse Shape mit Formen unterschiedlicher Ausprägungen (Rechteck, Linien, Ovale, …)

```
Shape[] shapes = new Shape[50]; // Array mit max. 50 Formen
shapes[0] = new Shape();  // einige Objekte
shapes[1] = new Shape();
shapes[2] = new Shape();
int shapeCnt = 3;  // verwaltet die Anzahl der Objekte
```

Beispiel: Darstellung



Die Objekte sind alle vom Typ (Klasse) `Shape`, die mit Hilfe des *Default*-Konstruktors generiert wurden.

- Annahme: Die Klasse Shape enthält eine Methode **`redraw(...)`** zur Anzeige in einem Graphik-Kontext `g`.
- Die Menge der instanziierten Formen kann dann zyklisch abgearbeitet werden. Zum Beispiel mittels:

```
for (int i = 0; i < shapeCnt; i++)
        shapes[i].redraw(g);
```

- Die maximale Anzahl graphischer Anzeigeobjekte darf 50 nicht überschreiten.

Dynamische Datenfelder

Motivation: Einschränkungen mit statischen Arrays:

- In den bisherigen Beispielen wurde die **maximale Anzahl der Elemente** eines Arrays durch einen beliebigen Wert beschränkt.
- Aufgrund der maximalen Anzahl können nicht beliebig viele Elemente gespeichert werden, z.B. 101 Elemente in der Zahlenfolge.
- Die Array-Größe so zu **dimensionieren**, dass sie für praktisch alle Fälle genügend viel Speicherplatz bereit stellt, ist keine gute Lösung:
  - **Speicherplatz wird unnötig verschwendet.**

Dynamische Datenfelder

Alternativer Ansatz

- Es wird im Falle einer festen Array-Dimensionierung ein Überlauf festgestellt:

  (z.B. `count >= 100` oder `shapeCnt >= 50`),

  In diesem Fall wird dynamisch eine neues Array mit größerem Speicherplatz generiert und die Elemente des alten Arrays in das neue kopiert.

- Realisierung:
  DynamicArrayOfInt Klasse, die den Array dynamisch wachsen lässt.

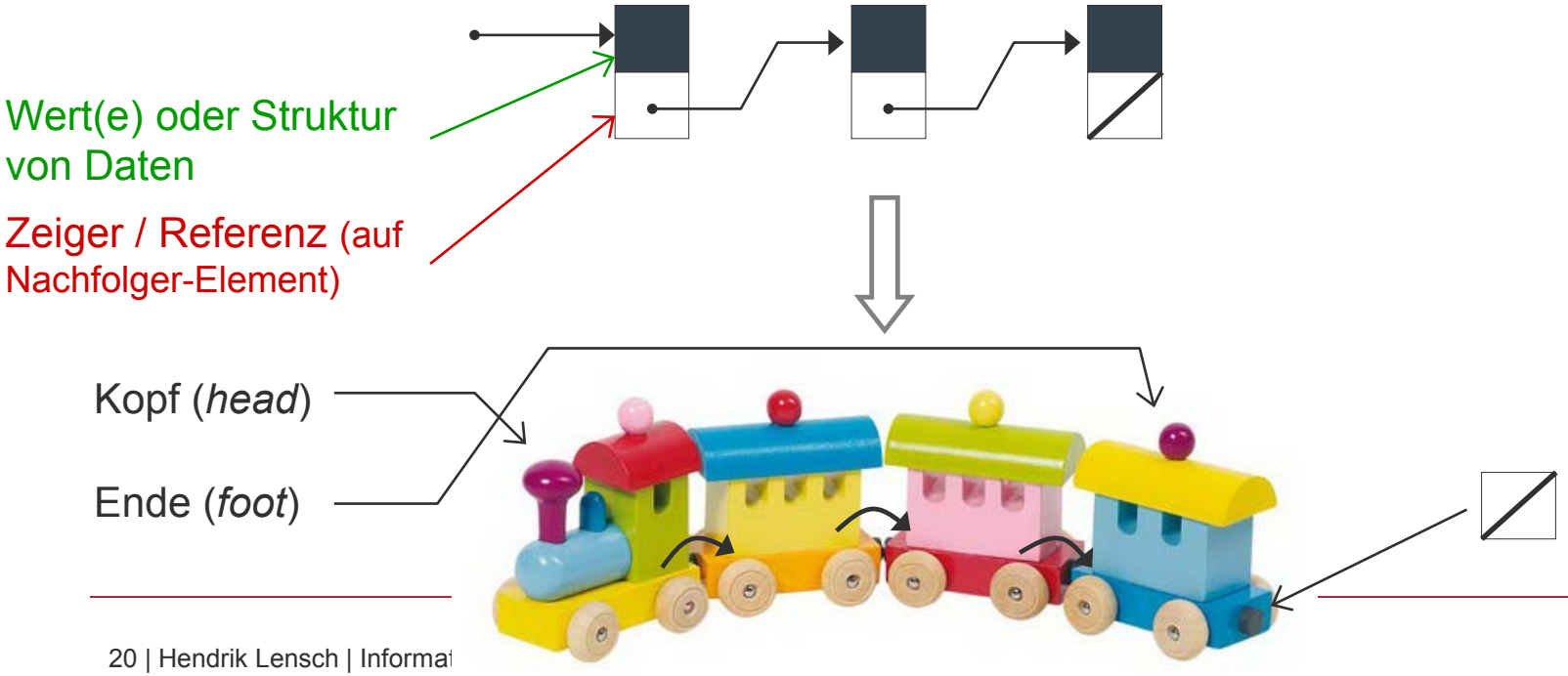
```java
public class DynamicArrayOfInt {
	private int[] data;  // the maintained dynamic array
	private int count = 0;

	public DynamicArrayOfInt() {
		data = new int[1];
	}
	public int getVal(int position) {
		if (position >= count)
			return 0;
		return data[position];
	}
	public void putVal(int position, int value) {
		if (position >= data.length) {  // position not covered by current array size -> expand it!
			int newSize = 2 * data.length;
			if (position >= newSize) // if doubling does not suffice…
				newSize = 2 * position;
			int[] newData = new int[newSize];    // construct new array.
			System.arraycopy(data, 0, newData, 0, data.length);  // copy data
			data = newData;                      // reset referencee
		}
		data[position] = value; // add the value at the requested position.
		if(position >= count)
			count = position+1;
	}
	public String toString() {
		StringBuffer sb = new StringBuffer();
		for(int d:data) {
			sb.append(d); sb.append(", ");
		}
		return sb.toString();
	}
	public static DynamicArrayOfInt getReverseAray(DynamicArrayOfInt daoi) {
		int[] data = new int[daoi.data.length];
		for(int i=0; i<daoi.count; i++) {
			data[i] = daoi.data[daoi.count-1-i];
		}
		DynamicArrayOfInt ret = new DynamicArrayOfInt();
		ret.data = data;
		ret.count = daoi.count;
		return ret;
	}
}
```

Alternative zu Arrays: Listen

- **Bisher**: Arrays als Aggregat von Elementen gleicher Art mit der Möglichkeit des sequenziellen Zugriffs mittels Zeiger.
  - Die Struktur hat momentan eine feste max. Anzahl von Elementen.
  - Der „Füllstand“ wird durch einen Zeiger angezeigt und jeweils aktualisiert.
- **Probleme** (auch mit dynamischen Arrays)
  - Müssen mehr Daten in einem Array gespeichert werden als Platz reserviert wurde, muss zunächst ein **neues Array** mit entsprechend angepasster Größe angelegt und danach die **Elemente des alten Arrays in das neue kopiert** werden.
  - Soll ein **Wert an einer bestimmten Position im Array eingeordnet** werden, müssen alle nachfolgende Elemente verschoben werden, um die Speicherposition für das neue Element frei zu geben.
- **Lösung: Listen**
  - Benötigen beim Elementeinfügen weder Elementkopier- noch Elementverschiebe-Operationen.
  - **Aber**: direkter Zugriff auf Elemente via Index ist NICHT mehr möglich.

Struktur einer Linearen Listen

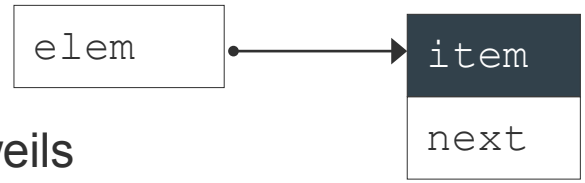
- Es werden **für neue Elemente jeweils dynamisch** einzeln und nacheinander **neue Repräsentationen** erzeugt.
- Es können **beliebig viele Elemente** repräsentiert werden – und zwischendurch auch wieder frei gegeben werden.
- Die **Elemente** werden nicht wie bei Arrays in eine feste Struktur mit Elementen eingetragen, sondern **hintereinander verkettet.**



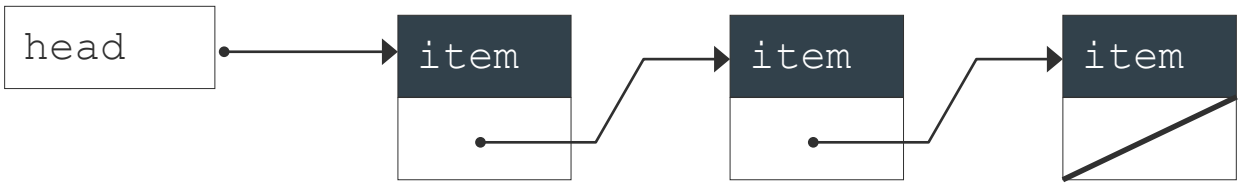
Struktur linearer Listen

Aufbau der einfach verketteten Liste:

- Eine lineare Liste besteht aus einer beliebigen Anzahl von Elementen:
  - Jedes Element enthält **Daten** – hier abstrakt repräsentiert durch einen Wert (`item`).
  - Jedes Element enthält einen **Zeiger** auf das jeweils nächste nachfolgende Element der Liste (`next`).



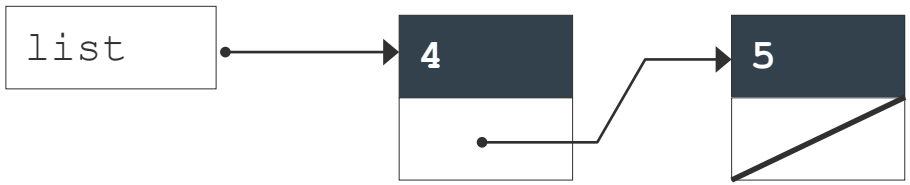
- Beispiel einer Liste mit Artikeln einer Produktionsstraße; die Daten sind als `item` repräsentiert (`item` ist ein Platzhalter und kann ein primitives Datum oder selbst ein Objekt einer eigenen Klasse sein).



Implementierung: ListElem.java

```
public class ListElem
{
    public int      item;  // Platzhalter
    public ListElem next;

    public ListElem(int newItem, ListElem next) {  // Konstruktor
        this.item = newItem;
        this.next = next;
    }

    // ... <beispielsweise in main-Methode>
    ListElem list = null;

    list = new ListElem(4, new ListElem(5, null));
    ...
}  // end class ListElem
```

erzeugt dann:



Verwaltung einer einfach verketteten Liste

- Initialisierung einer **leeren Liste** mit Zeiger auf erstes Element.

```
ListElem head = null;
```

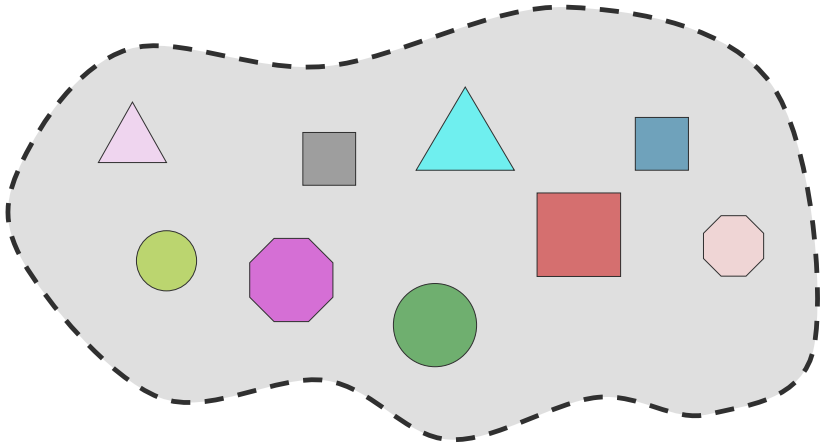
- Einfügen des Elements (`article`) in leere Liste

```
if (head == null) {
   head = new listElem(article, null);
}
```

Klassen für Repräsentationen strukturierter Datenobjekte

Definition als Klassen in Java

- In dem Listenbeispiel diente `item` vom Datentyp `int` als Platzhalter.
- Man braucht eine Klasse, um Elemente zu verwalten, die selbst aus verschiedenen unterschiedlich **strukturierten Komponenten zusammen gesetzt** sind (unterschiedliche Typen mit unterschiedlichen Werten)



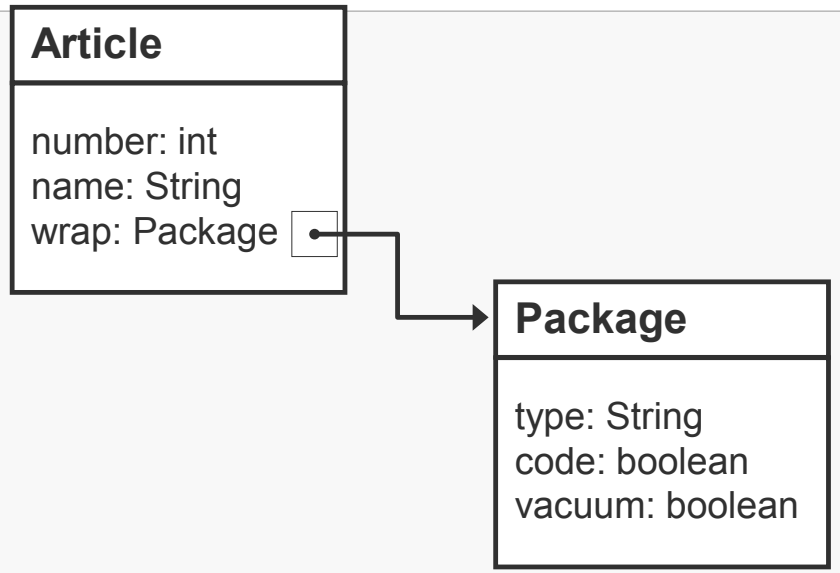
= Klasse

- Hinweis: Die Elemente selbst können auch strukturiert und als Klassen repräsentiert sein.

Strukturierte Daten als Klasse

Bsp.:

```
class Article {
   int     number;
   String  name;
   Package wrap;
}

class Package {
   String  type;
   boolean code,
           vacuum;
}
```



- **Hinweis**: Die Klassen enthalten hier in dem Beispiel keine Methoden für die Änderung des Zustands; die Klassen dienen nur als **Container für Daten**!

Records: Strukturierte Daten in prozeduralen Sprachen

- Prozedurale (imperative) Programmiersprachen erlauben neben Arrays auch Strukturen für die Deklaration von Datenmengen als Zusammenfassung von Elementen (feste Anordnung der Typen).

- Diese werden in solchen Sprachen **`Record`** oder **`struct`** genannt.

- Bsp: `Structs` in C

```
struct address {
    char street[20];
    int  number;
    int  zipcode;
    char city[20];
}

struct student {
    int  regnumber;
    char name[20];
    char prename[20];
    struct address home;
}
```

Einseitig verkettete Listen als Objekte

Motivation

- Die Verwaltung von Listen basierend auf `ListElem` ist umständlich: Das Einfügen von Elementen in leeren Listen muss von nicht-leeren Listen unterschieden werden.

- Auf Listen sollen **elementare Operationen** zum **Hinzufügen**, **Suchen**, **Löschen**, … von Elementen definiert werden;

  - Denn die Fallunterscheidungen bergen Fehler, die die Stabilität von Programmen gefährden.

Einseitig verkettete Listen als Objekte

Objektorientierte Vorgehensweise

- Neben der Klasse `ListElem` wird eine Klasse `List` zur Verwaltung der Liste deklariert.

- In der Klasse `List` ist ein expliziter Verweis auf den Kopf der Liste enthalten.

- Zusätzlich enthält die Klasse List auch die wichtigsten Elementaroperationen auf Listen als Methoden
  - Hinzufügen eines Elements
  - Suchen eines Elements
  - Löschen eines Elements
  - ...

Elementarer Aufbau

- Listenelemente (Knoten) – Wiederholung
  - Datenteil / Attribute (**`item`**; hier weiterhin **`int`** als Platzhalter)
  - Zeiger / Referenz auf Folgeelement der Liste (**`next`**)

```
public class List {
    ListElem head;

    public List() {           // Kontruktor
        this.head = null;
    }

    public List(int item) {  // Konstruktor
        this.head = new ListElem(item, null);
    }

    void     insertElem(int item) {...}
    ListElem searchElem(int item) {...}
    int      getElem(int index) {...}
    ...
}
```

Operationen auf einfach verketteten Listen

- Einfügen von Elementen am Anfang einer Liste (als Methode in **`List`**).
  - (Bei einem Array müssten zunächst alle Elemente um eine Position nach hinten verschoben werden…)
- Mit Listen geht dieses Einfügen sehr einfach:

Kopie der Referenz auf das 1. Element der bisherigen Liste

```
void insertElemFirst(int item){
   head = new ListElem(item, head);
}
```

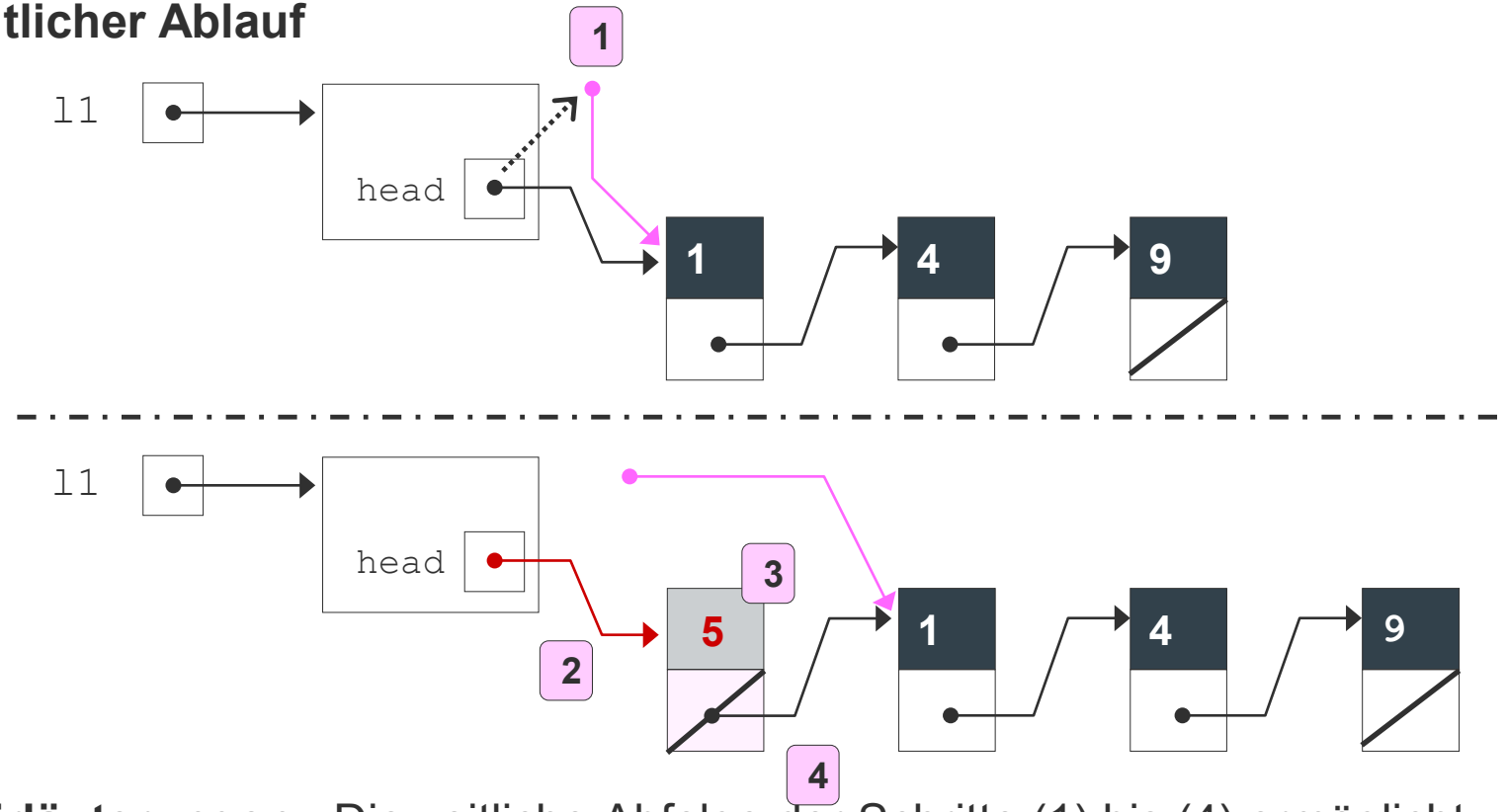
Aufruf im Hauptprogramm:

```
List l1 = ...;  // Konstruktion
...
l1.insertElemFirst(5);
```

Einfügen am Anfang

```
void insertElemFirst(int item) {
    head = new ListElem(item, head);
}
```

Zeitlicher Ablauf



- **Erläuterungen**: Die zeitliche Abfolge der Schritte (1) bis (4) ermöglicht die korrekte Generierung und das Einfügen des neuen Elementes an den Kopf der Liste.

Einfügen am Ende der Liste

- Da man (hier) **nicht direkt** auf das **letzte Element** der Liste zugreifen kann, muss es berechnet werden.
- Rekursive Implementierung in Java (als Methode in List)

```
void insertElemLast(int item) {  // komfortabler Aufruf
    if (head == null)
        head = new ListElem(item, null);
    else
        insertElemLast(item, head);  // Methode ueberladen ...
}

void insertElemLast(int item, ListElem elem) {
    if (elem.next == null)
        elem.next = new ListElem(item, null);
    else
        insertElemLast(item, elem.next);  // rekursiver Aufruf
}
```

➢ **Hinweis**: Die Bestimmung des Listen-Endes ist end-rekursiv, daher kann ein Algorithmus auch leicht in eine iterative Form überführt werden.

Suchen eines Elements in einer Liste (iterativ)

- **Ziel**: Liefere die Referenz auf das erste Element in der Liste mit dem gegebenen Inhalt **`item`**.
  Wenn das Element nicht vorhanden ist, soll ein **`null`**-Zeiger zurück geliefert werden.

```
ListElem searchElemIter(int value) {
   ListElem current = this.head;

   while (current != null) {        // weitere Listen-Elemente?
      if (current.item == value)    // gefunden?
         return current;
      current = current.next;
   }
   return null;                     // Wert nicht gefunden ...
}
```

Zusatzvariable zur Speicherung des aktuell betrachteten Elements.

Die Wiederholungsschleife such nach dem Item – Abbruch bei Listenende oder beim finden des Items.

Suchen eines Elements in einer Liste - Alternative

- Einfachere Alternative, die sich das „short-circuited“ Prinzip während der Auswertung eines Ausdrucks zu nutze macht:
  - Zweiter Vergleich wird nicht ausgeführt, wenn point==null
  - ➢ Verhinderung einer Null-Pointer-Exception.

```
ListElem searchElem(int value) {
    ListElem point = this.head;
    while(point != null && point.item != value)
        point = point.next;
    return point;
}
```

Ausgabe der Elemente einer Liste

- **Ziel**: Es soll jedes Element in der Reihenfolge in der Liste besucht und der Inhalt ausgegeben werden

```
void printList() {
   ListElem elem = head;
   System.out.print("Liste ( ");
   while(elem != null) {
      System.out.print(elem.item + " ");
      elem = elem.next;
   }
   System.out.println(" )");
}
```

Liste rückwärts ausgeben

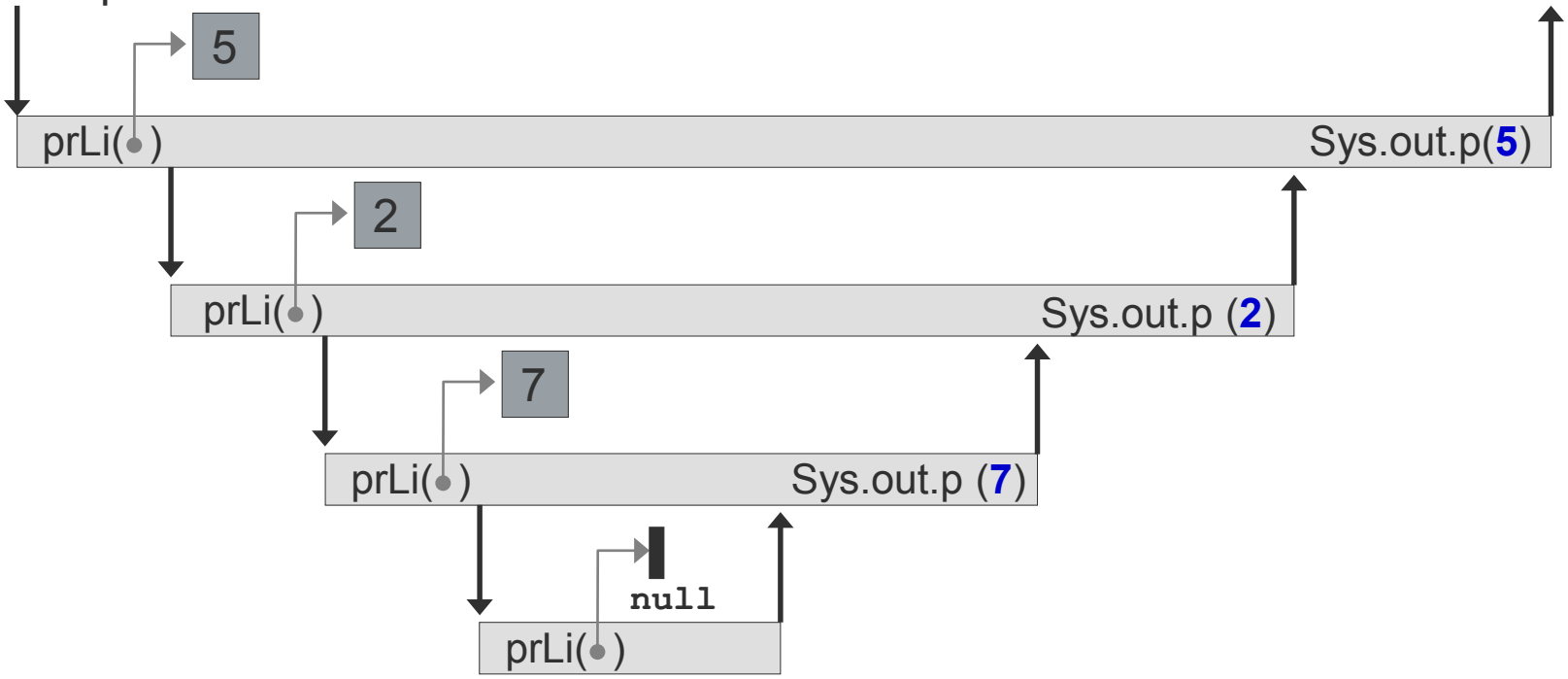
- **Ziel**: Jedes Element der Liste soll besucht und deren Inhalt in umgekehrter Reihenfolge ausgegeben werden (von hinten nach vorne)

```
void printListReverse() {
   System.out.print("Liste rueckwaerts ( ");
   printListReverse(this.head);
   System.out.println(" )");
}

void printListReverse(ListElem elem) {
   if (elem != null) {
      printListReverse(elem.next);
      System.out.print(elem.item + " ");
   }
}
```

Die Methode ist **nicht end-rekursiv**, da die Rekursion vor dem letzten Befehl erfolgt; die Inhalte werden erst nach dem rekursiven Methoden-Aufruf gedruckt!

Liste rückwärts ausgeben - iterativ

- Nicht ohne weiteres iterativ umzusetzen, da Methode nicht end-rekursiv.
- Es wird eine weitere Liste zur Speicherung der bisher bereits besuchten Elemente benötigt.
- Beispielablauf für eine Liste 5 – 2 – 7



Verkettete Listen mit Kopf und Fuß

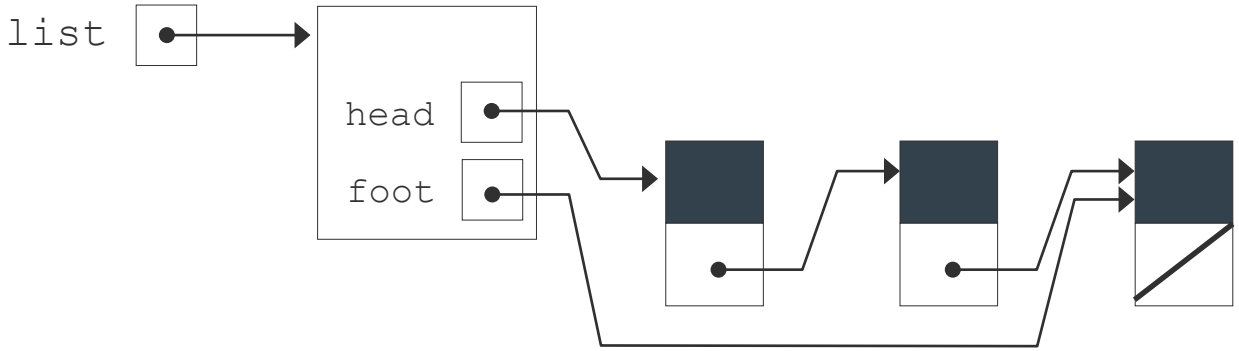
Struktur

- Die Listenelemente bleiben gleichermaßen definiert mit Datenteil (**`item`**) und Zeiger auf das nachfolgende Element (**`next`**).
- Da das vollständige Ablaufen der Liste zum Einfügen eines Elements am Ende umständlich und zeitraubend ist, wird häufig die Referenz auf das letzte Element der Liste (Fuß, **`foot`**) zusätzlich gespeichert:

```
public class List {
   ListElem head,
            foot;

   public List(int item) {   //Konstruktor
      this.head = new ListElem(item, null);
      this.foot = head;
   }
   ...
}
```

Verkettete Listen mit Kopf und Fuß

- Beispielstruktur:



Einfügen am Ende einer Liste – 2. Variante

- Mithilfe des Listen-Fußes kann ein Element **einfach am Ende** der Liste angehängt werden:

```
void insertElemLast(int item) {
    if (head == null) {
        this.head = new ListElem(item, null);
        this.foot = this.head;
    }else {
        this.foot.next = new ListElem(item, null);
        this.foot      = this.foot.next;
    }
}
```

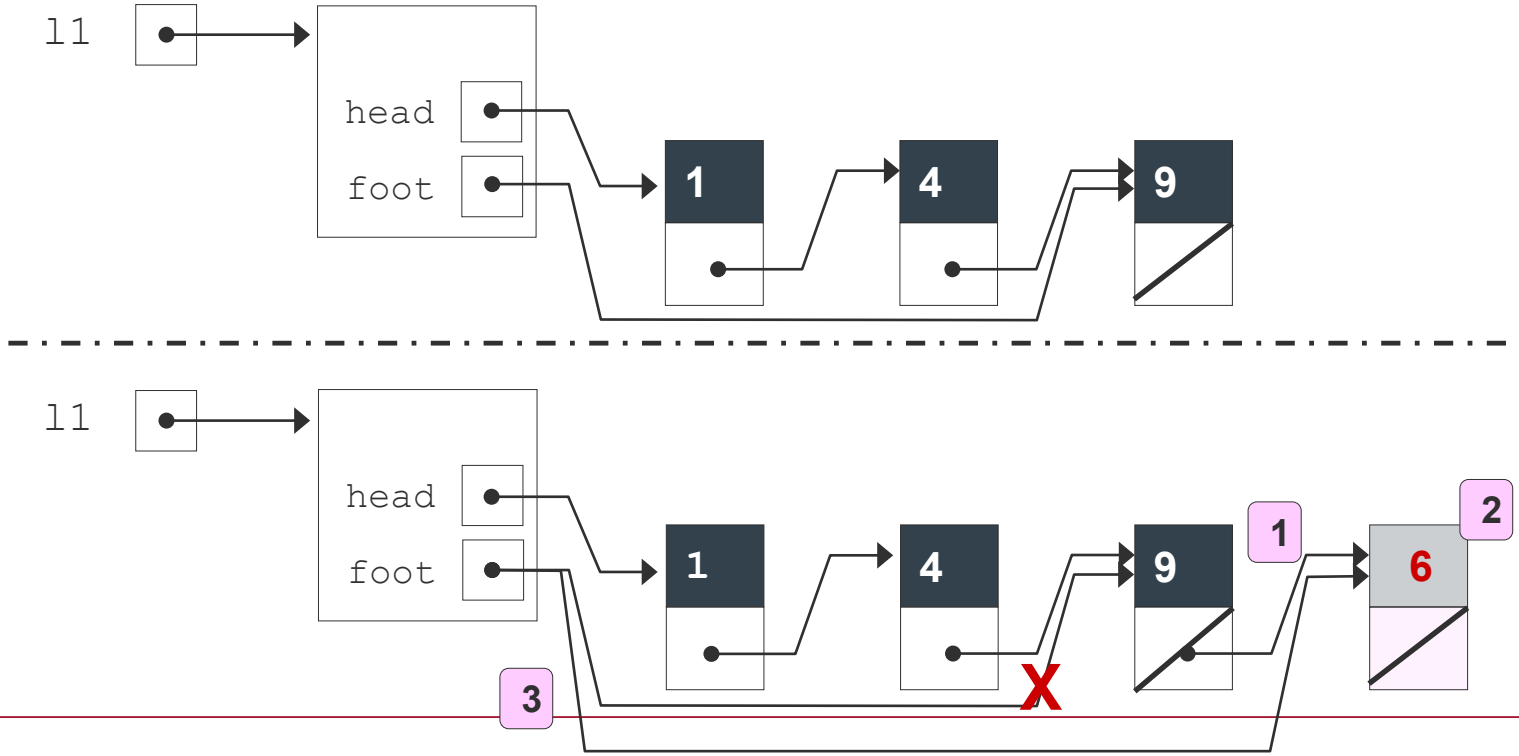
- Einfügen am Anfang muss entsprechend angepasst werden.

Verkettete Listen mit Kopf und Fuß – Einfügen (Ende)

```
/* Aufruf im Hauptprogramm */
List l1 = ...;  // Konstruktion
    ...
l1.insertElemLast(6);
```

```
void insertElemLast(int item) {
     if (head == null) {
          this.head = new ListElem(item, null);
          this.foot = this.head;
     }else {
          this.foot.next = new ListElem(item, null);
          this.foot      = this.foot.next;
     }
}
```

Zeitlicher Ablauf:



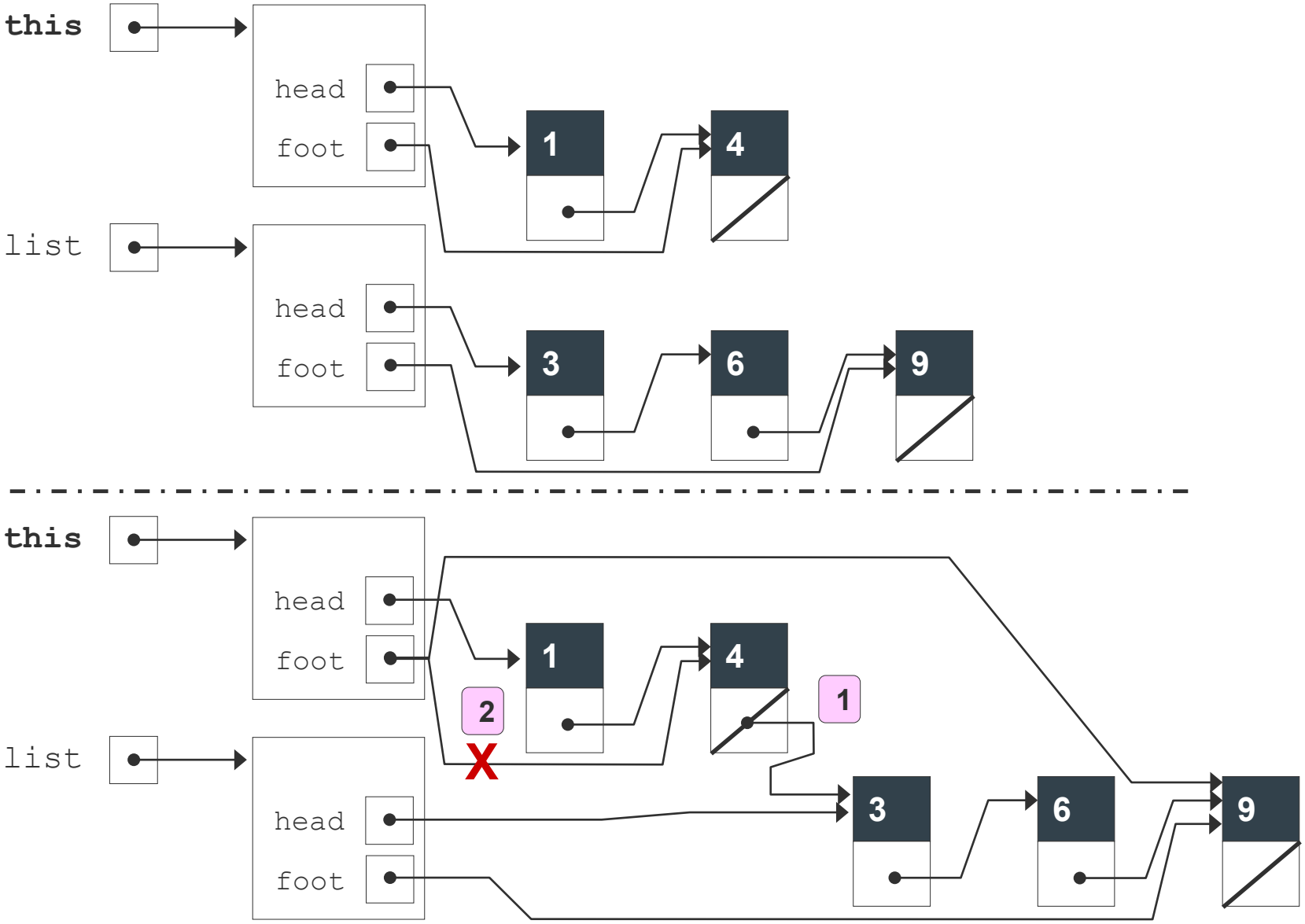
Anhängen einer Liste an eine andere

Einfache Realisierung

- **Ziel**: An eine Liste soll eine andere angehängt werden; dabei können neben zwei nicht-leeren Listen **verschiedene Spezialfälle mit leeren Listen** auftreten:

```
public void appendList(List list) {
   if (this.head == null) {
      this.head = list.head;
      this.foot = list.foot;
   }else if (list.head == null) {
      System.out.println("2. Liste ist leer");
   }else {
      this.foot.next = list.head;
      this.foot      = list.foot;
   }
}  // end appendList
```

Anhängen einer Liste



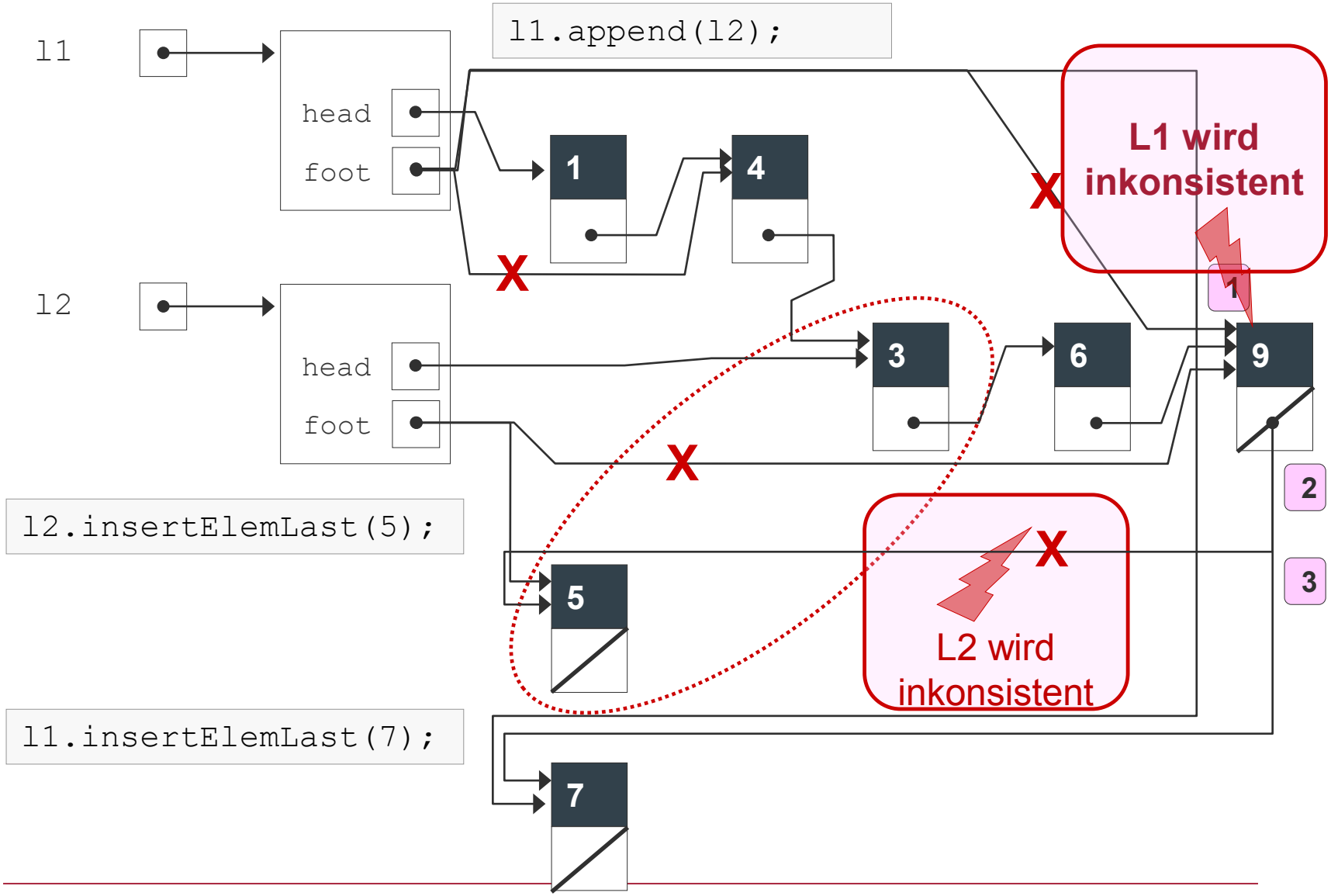
Seiteneffekte

- **Problem**: Es können Seiteneffekte auftreten!

```
public void appendList(List list) {
    if (this.head == null) {
        this.head = list.head;
        this.foot = list.foot;
    }else if (list.head == null) {
        System.out.println("2. Liste ist leer");
    }else {
        this.foot.next = list.head;
        this.foot      = list.foot;
    }
}  // end appendList
```

```
List l1 = ...,
     l2 = ...;

l1.append(l2);
l2.insertElemLast(5);
l1.insertElemLast(7);
```

Seiteneffekte - Auswirkung



Anhängen - konsistent

- **Lösung**: Bei **`append`** müssen alle Elemente der 2. Liste (die angehängt werden soll) kopiert werden; dies kann durch elementweises Anhängen realisiert werden

1. Beginne mit dem Kopf der Liste

2. Überprüfe, ob Liste leer ist

```
void appendList(List list) {
    ...
    else {
        for (ListElem elem = list.head; elem != null; elem = elem.next)
            insertElemLast(elem.item);
    }
}
```

3. Anhängen der Elemente am Ende der Liste als neues Element

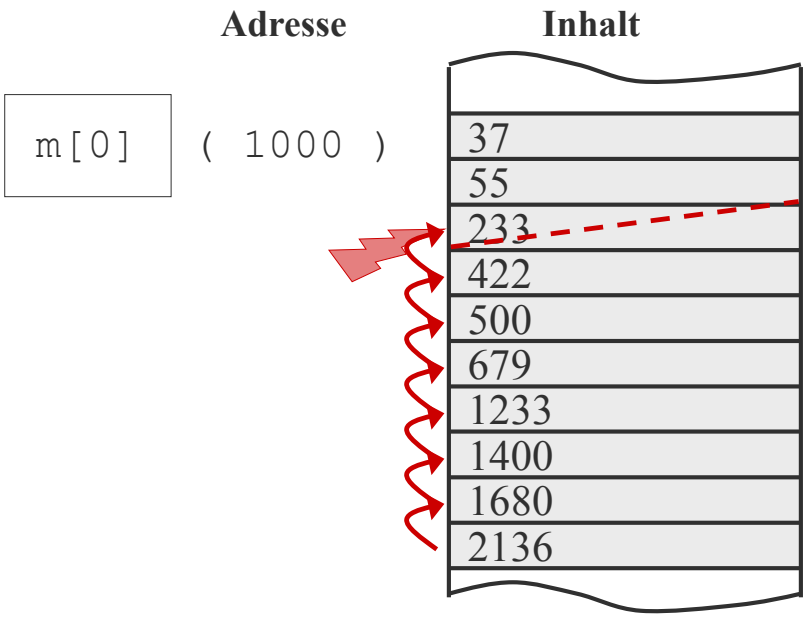
4. Weiterschalten auf das nächste Listenelement

Löschen / Einfügen von Listen-Elementen an beliebiger Position

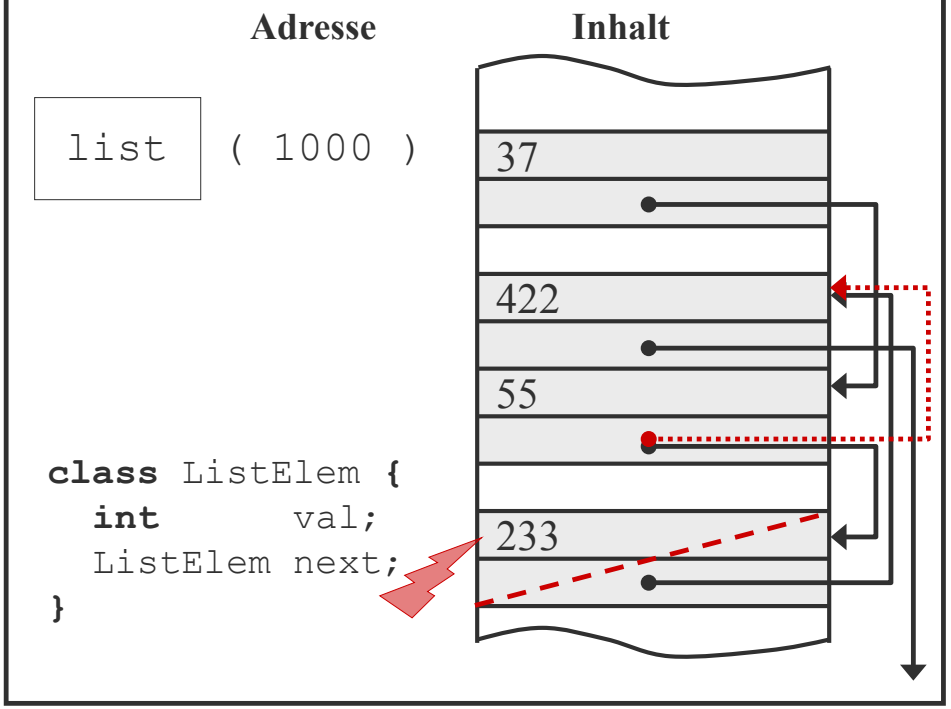
- **Ziel**: Es soll das Element der Liste an der Position `pos` gelöscht werden – und weiterhin eine kohärente Liste erhalten bleiben.

- **Repräsentation** von Feldern (Arrays) und linearen Listen (im Speicher):

```
int[] m = new int[10];
```




```
ListElem list = ...;
```

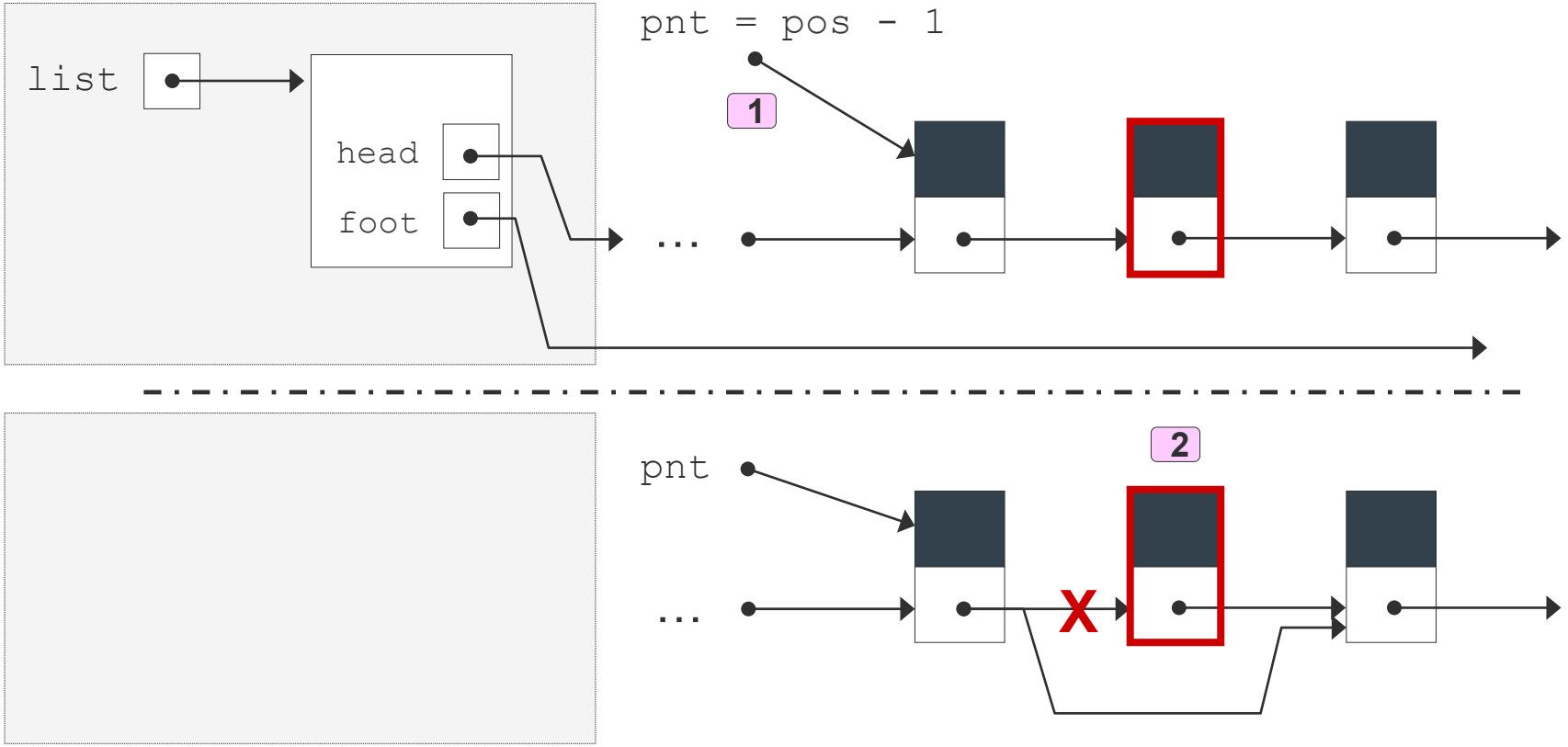


Element Löschen

```
void deleteListElem(int pos) {
    if (this.head != null) {
        if (pos == 0) {                        // Element am Listenanfang
            if (this.head == this.foot)        // Nur ein Element in der Liste
                this.foot = null;
            this.head = this.head.next;
        } else {
            // vorheriges Element mit getListElem(...)
            ListElem pnt = getListElem(pos-1);   1
            if ((pnt != null) && (pnt.next != null))
                if (pnt.next == this.foot) {   // Element am Listenende
                    pnt.next  = null;
                    this.foot = pnt;
                }else                                  // Element mittendrin
                    pnt.next  = pnt.next.next;
        }
    }                                                  2
}
```

Löschen eines Elements

- Bsp.: Löschen des Elements an der Listenposition **`pos`** (erster Index = 0)



- **Anmerkung**: Das vormals an pos stehende Element der Liste kann im Anschluss nicht mehr referenziert werden (die Garbage collection eliminiert u. U. dieses Datenobjekt)

Methode `getListElem(...)`

```
ListElem getListElem(int pos) {
    ListElem pnt = null;

    if (this.head != null) {
        pnt = this.head;
        for (int i = 0; i < pos; i++) {
            if (pnt.next == null) {
                System.out.println("pos ausserhalb Listenlaenge!");
                pnt = null;
                break;
            }else
                pnt = pnt.next;
        }
    }
    return pnt;
}  // end getListElem
```

Erläuterungen:

- Wenn die Liste leer ist, dann wird ein **`null-Zeiger`** zurückgeliefert.
- Wenn der Positionsindex **`pos`** > Länge der Liste, dann **`null-Zeiger`** und Fehlermeldung.
- Iteration durch alle nacheinander folgenden Elemente.

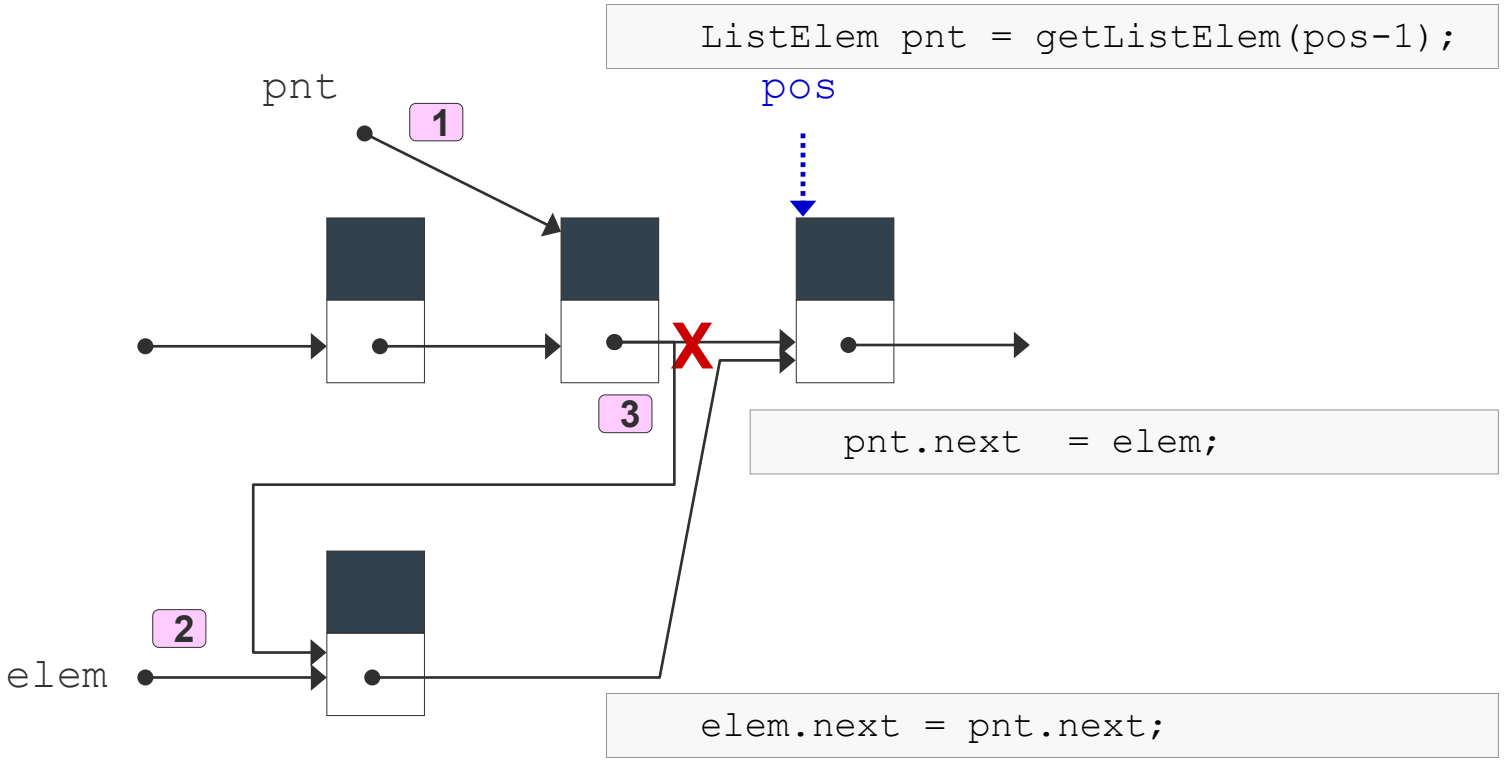
Einfügen an gegebener Position

- **Ziel**: Es soll ein neues Element an der Position `pos` in eine vorhandene Liste eingefügt werden:

```
void insertListElem(int pos, ListElem elem) {
    if (pos == 0) {  // 1. Element
        if (this.head == null) {  // leere Liste
            this.head = elem;
            this.foot = this.head;
        }else{                    // nicht-leere Liste
            elem.next = this.head;
            this.head = elem;
        }
    } else {
        ListElem pnt = getListElem(pos-1);
        if (pnt != null) {  // pos im Bereich der Listen-Laenge
            elem.next = pnt.next;
            pnt.next  = elem;
        }
    }
}  // end insertListElem
```

Einfügen an gegebener Position

- Bsp.: Einfügen an `pos = 3` (so dass das eingefügte Element anschließend an der angegebenen Position in der Liste steht)



Erläuterungen:

- Das neue Element steht nach der Einfügeoperation an der als Parameter angegebenen Position in der Liste.

Sortiertes Einfügen eines Elements in eine Liste

- **Ziel**: Neue Elemente sollen sortiert in eine Liste eingefügt werden, d.h. die aktuelle Einfügeposition **`pos`** muss aus dem Wert des Schlüsselelements bestimmt werden; die Ordnung ist aufsteigend:

```
void insertListElemSorted(ListElem elem) {
    if (this.head == null) { // Element in leerer Liste
        this.head      = elem;
        this.foot      = elem;
        this.foot.next = null;
    } else {
        if (elem.item < this.head.item) {  // als 1. Element einfuegen
            elem.next = this.head;
            this.head = elem;
        } else
            insertListElemSorted(elem, this.head); // Methode überladen
    }
}  // end insertListElemSorted
```

- **Erläuterungen**: Die Realisierung von **`insertListElemSorted(...)`** wird hier **mittels Überladens** der Methode realisiert
  - **`insertListElemSorted(ListElem elem)`**
  - **`insertListElemSorted(ListElem elem, ListElem)`**

Sortiertes Einfügen eines Elements in eine Liste (2)

```
void insertListElemSorted(ListElem elem, ListElem list) {
    if (list.next == null) {                    // elem am Ende einfuegen
        list.next = elem;
        this.foot = elem;
        elem.next = null;
    }else if (elem.item < list.next.item) {   // Position gefunden
        elem.next = list.next;
        list.next = elem;
    }else
        insertListElemSorted(elem, list.next);  // rekursiv weiter
}  // end insertListElemSorted
```

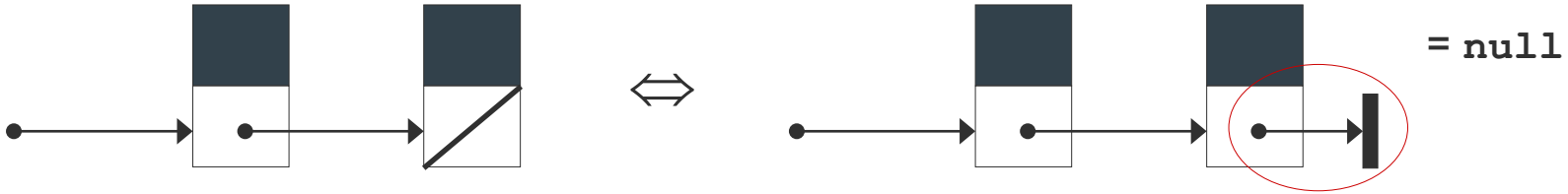
Bemerkungen:

- Wenn ein einzufügendes Listen-Element den gleichen Schlüsselwert wie ein schon vorhandenes Listen-Element besitzt, dann wird das neue Element vor dem vorhandenen eingeordnet.
- Durch die (rekursive) Methode wird sichergestellt, dass ein neues einzufügendes Element mit dem momentan höchsten Schlüssel-Element am Ende der Liste eingeordnet wird.

Alternative Techniken zur Verwaltung des Listenendes

Bisherige Vorgehensweisen

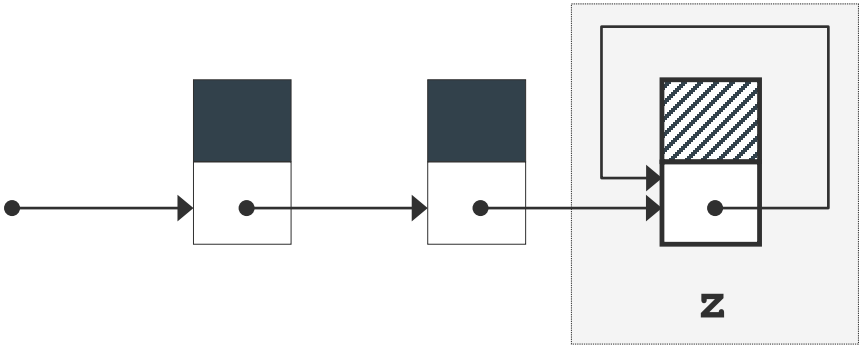
- Referenz **`next`** des letzten Listen-Elements ist **`null`**



- **Nachteil**: Der **`next`**–Zeiger muss beim Löschen und Einfügen von Elementen stets auf null geprüft werden, um bei Zugriffen **`pnt.next.item`** einen Zugriffsfehler (Null-Pointer Exception) auszuschließen.

Wächterknoten als Abschluss

- Verwendung eines „**Dummy“-Knotens `z` als Wächter (sentinel)** zum Abschluss der Liste
  - Die **`next-Referenz`** des letzten Listen-Knotens zeigt auf `z` und referenziert sich damit selbst.
  - Der **`item-Eintrag`** von `z` ist undefiniert.



Wächter (*sentinel*)

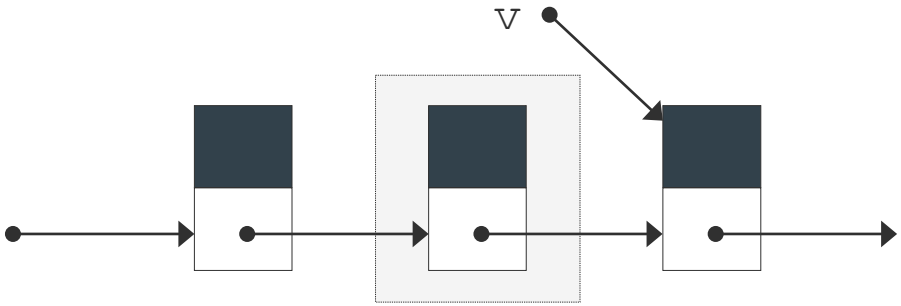
- **Vorteil**: Die Zugriffsfunktionen können vereinfacht werden, da die Dereferenzierung für alle Listenelemente gültig ist.

Problematik bei einfach verketteten Listen

- Bei **einfach verketteten Listen** speichert ein Element stets die Referenz auf seinen **direkten Nachfolger.**

- **Aufgabe**: Bestimme den **Vorgänger-Knoten** von `v`

- **Lösung**: Suche mit dem Kopf der Liste bis der Vorgänger `elem.next == v` erreicht ist…

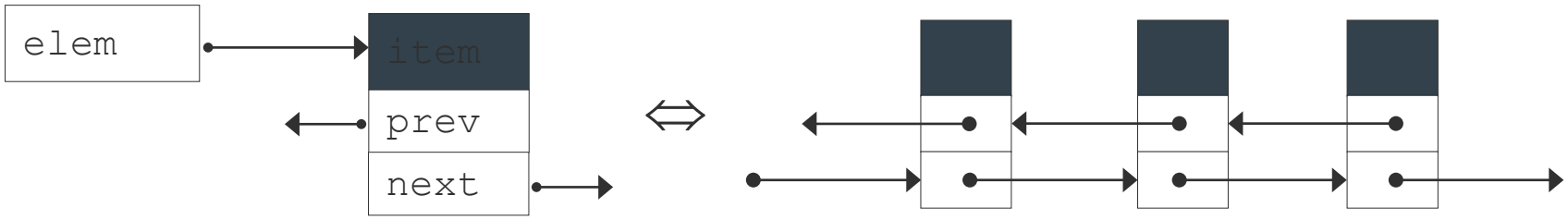


➢ Der Aufwand für die Operation ist abhängig von der Position von `v` und der Anzahl der Elemente in der Liste!

➢ Das sollte doch schneller machbar sein, oder?

Doppelt verkettete Listen (doubly-linked lists)

- **Alternative**: Erweiterung der Repräsentation von Listenelementen um eine **Referenz zum jeweiligen Vorgänger-Knoten**



Doppelt verkettete Listen – Struktur

Repräsentation

- Erweiterung der Klasse für Listen-Elemente

```
class ListElem {
   char item;
   ListElem next,
            prev;

   ListElem() {
   }

   ListElem(char item, ListElem next, ListElem prev) {
      ...
   }
   ...
}
```

- Die **Operationen** auf Listen mit Doppel-Verkettung der Elemente müssen entsprechend angepasst werden.

Doppelt verkettete Listen – Vor- und Nachteile

Vorteile

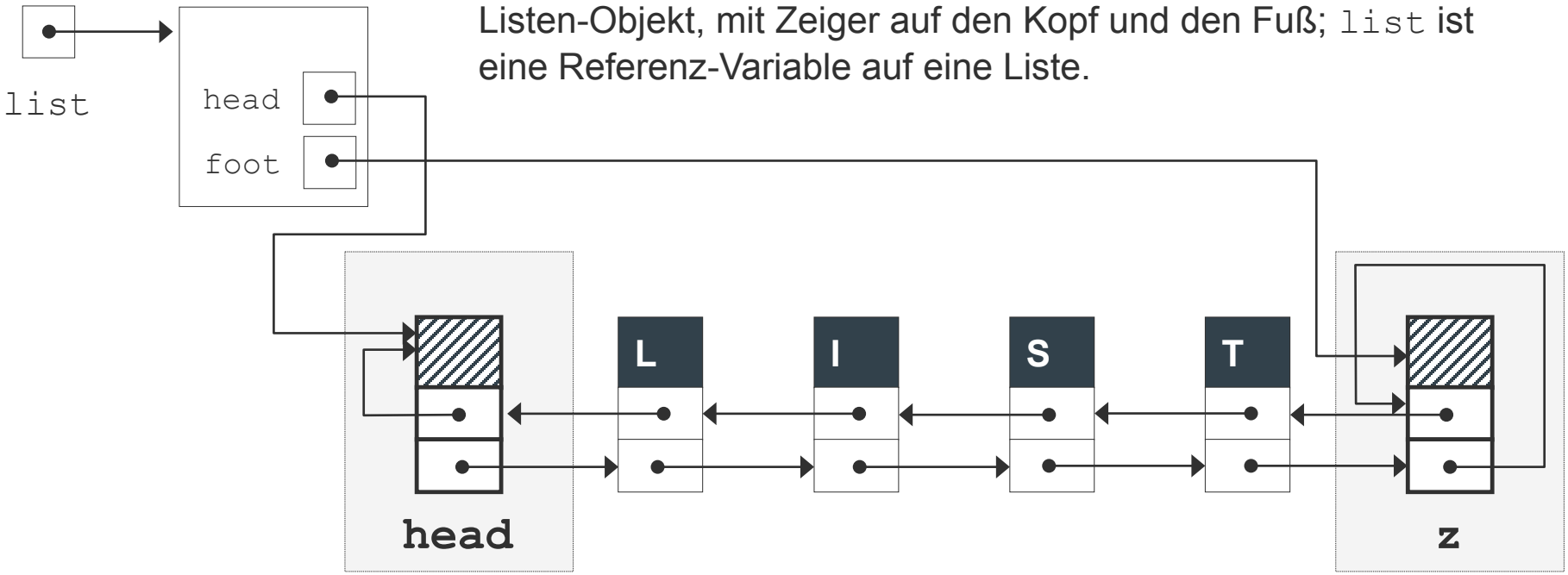
- Einfügeoperationen können jetzt auch einfach ein Element vor einem selektierten Listenelement in eine Liste einfügen.
- Der Aufwand für die Bestimmung des Elements an der Position pos kann reduziert werden, wenn die Anzahl der Elemente in der Liste bekannt ist (Start vom Anfang oder Ende).

Nachteile

- Für jede Grundoperation – zur Manipulation der Grundstruktur – wird die Anzahl der Manipulationen der Zeiger verdoppelt.
- Die Verwaltung des Listenanfangs (head) und –endes (foot) wird aufwändiger.
- Der Speicherbedarf für Listenelemente steigt.

Wächter und Doppelt verkettete Listen

- Erweiterung der bisherigen Struktur einfach verketteter Listen zu doppelt verketteten Listen – mit **Wächter-Elemente am Ende und am Anfang**

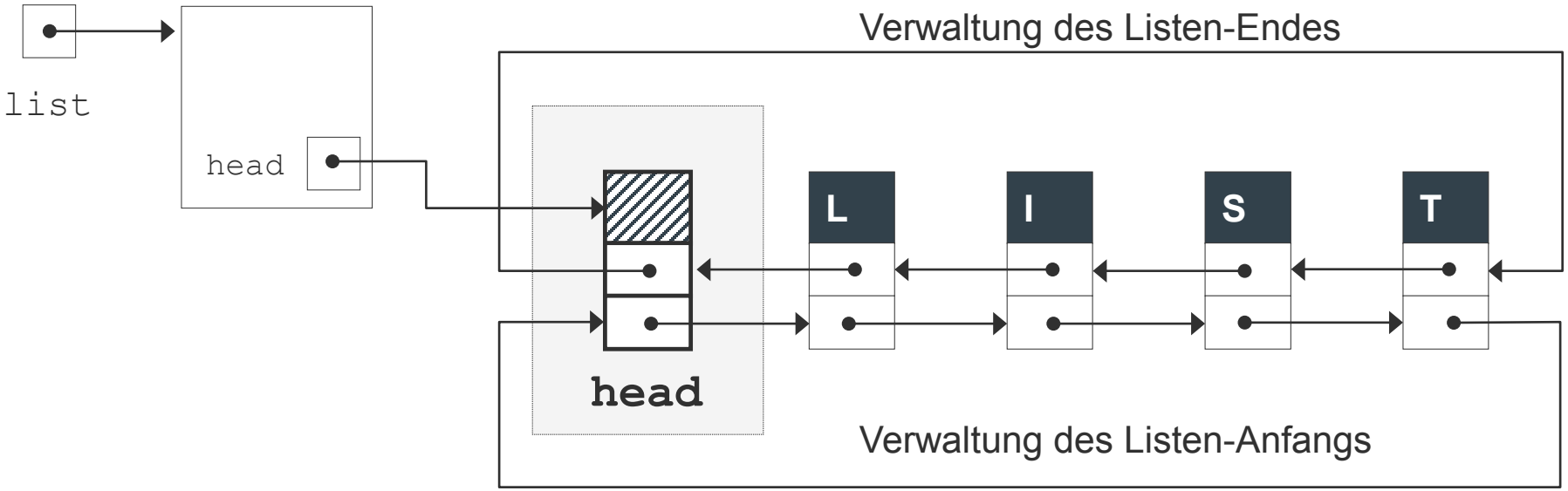


Listen-Objekt, mit Zeiger auf den Kopf und den Fuß; `list` ist eine Referenz-Variable auf eine Liste.

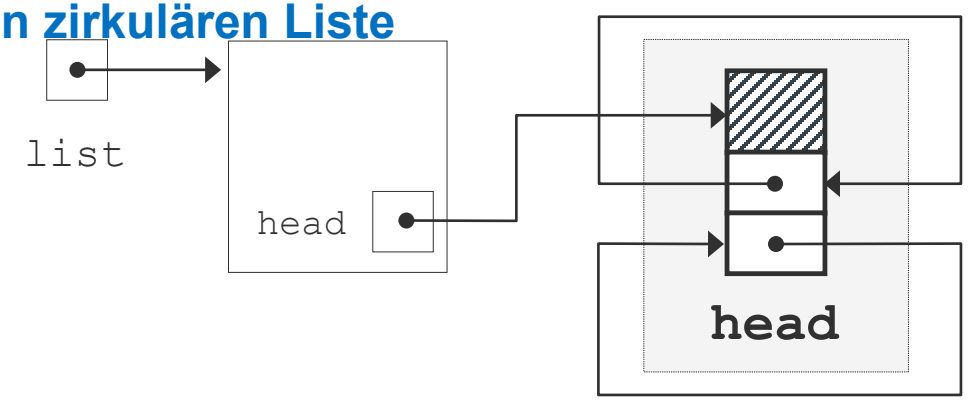
Elemente der Liste mit zwei **ausgezeichneten (Wächter-) Elementen** (`head`, `z`), die **keine Inhalte speichern.**

Zirkuläre Liste

- Verwendung nur eines ausgezeichneten Wächters - das letzte Element (`foot`) muss für die Liste nicht explizit verwaltet werden



- Spezialfall einer **leeren zirkulären Liste**



Listen vs. Arrays – eine Gegenüberstellung

Arrays	Listen
+ Einfaches Einfügen am Ende − Beliebiges Einfügen und Löschen schwierig / aufwändig	+ Einfügen und Löschen an beliebigen Stellen
+ Wahlfreier Zugriff: Index = Adresse + Schnell	− Nur sequenzielles Durchlaufen − Etwas langsamer
+ Implizite Topologie: keine weiteren Daten notwendig, um die Organisation zu speichern.	− Explizite Topologie: next und previous müssen gespeichert werden.
− Begrenzte Größe	+ Beliebig erweiterbar

STAPEL UND SCHLANGEN

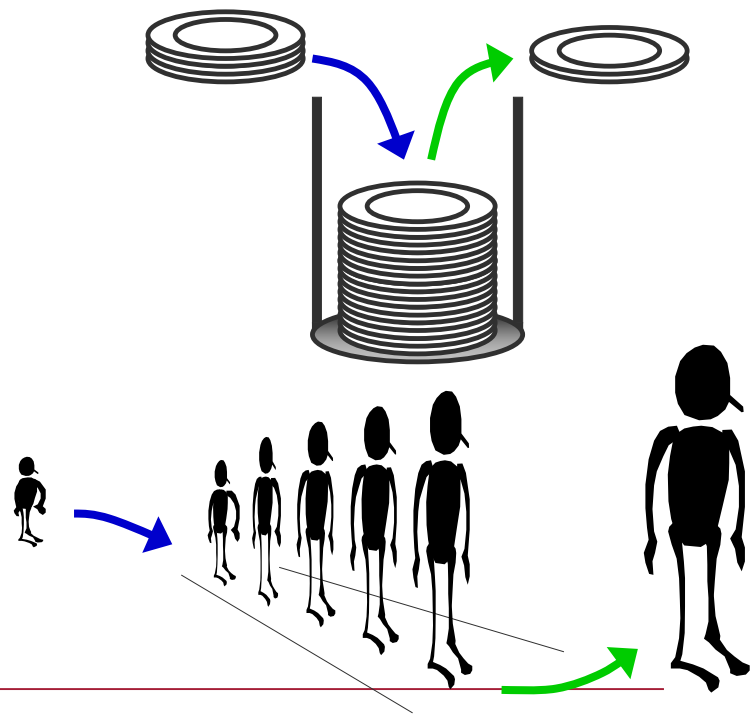
- Lineare Strukturen mit Zugriffsbeschränkungen
- Stapel (*Stacks*)
- Schlangen (*Queues*)

Lineare Strukturen mit Zugriffsbeschränkungen

- **Bisher**: Geordnete Strukturen**, Zugriff auf Elemente grundsätzlich unabhängig von ihrer Position** (z.B. Arrays, lineare Listen)
  - ➢ D.h.: auf alle Elemente konnte zugegriffen werden.

- **Jetzt**: Strukturen, bei denen zu einem bestimmten Zeitpunkt **jeweils nur auf ausgezeichnete Elemente zugegriffen** werden kann:

1. **Stapel** (Papier, Teller, Tabletts, etc.)
   - Teller, Seiten, Dokumente, etc. werden jeweils oben aufgelegt.
   - Nur der oberste Teller, etc. kann weggenommen werden.
2. **Warteschlange** (Ski-Lift, Kasse, Druckaufträge, etc.)
   - hinten anstellen – und geduldig sein …
   - jeweils erste Position in der Schlange wird abgefertigt



Zugriffsmuster

Stapel ((pushdown) stacks)

- Der Zugriff auf die Elemente erfolgt durch **Anfügen und Entfernen am selben Ende der Datenstruktur**; ***LIFO (last-in first-out) Prinzip:***
  - Lesezugriff: oberstes Element des Stapels (= letztes Listen-Element)
  - Schreibzugriff: Einfügen hinter das zuletzt eingefügte Element (hinter das letzte Listen-Element)

Schlangen (queues)

- Der Zugriff auf die Elemente erfolgt durch **Anfügen und Entfernen an verschiedenen Enden** der Datenstruktur; ***FIFO (first-in first-out) Prinzip***
  - Lesezugriff: erstes Element der Schlange (= erstes Listen-Element)
  - Schreibzugriff: Einfügen hinter das zuletzt eingefügte Element (hinter das letzte Listen-Element)

Verwendung von Stacks und Queues

Stapel (stacks)

- **Auswertung geklammerter Ausdrücke**, z.B. mathematische / logische Ausdrücke und ihre syntaktische Prüfung (Anwendungen im Compiler-Bau)
- **Rekursive Methoden**: Parameterversorgung und Auslesen aktueller Parameter (vgl. auch Formularmaschine)
- **Iterative Methodenaufrufe**: Wird eine innere Methode fertig, so liegt die Methode, die diese aufgerufen hat, dann oben auf dem Stack.

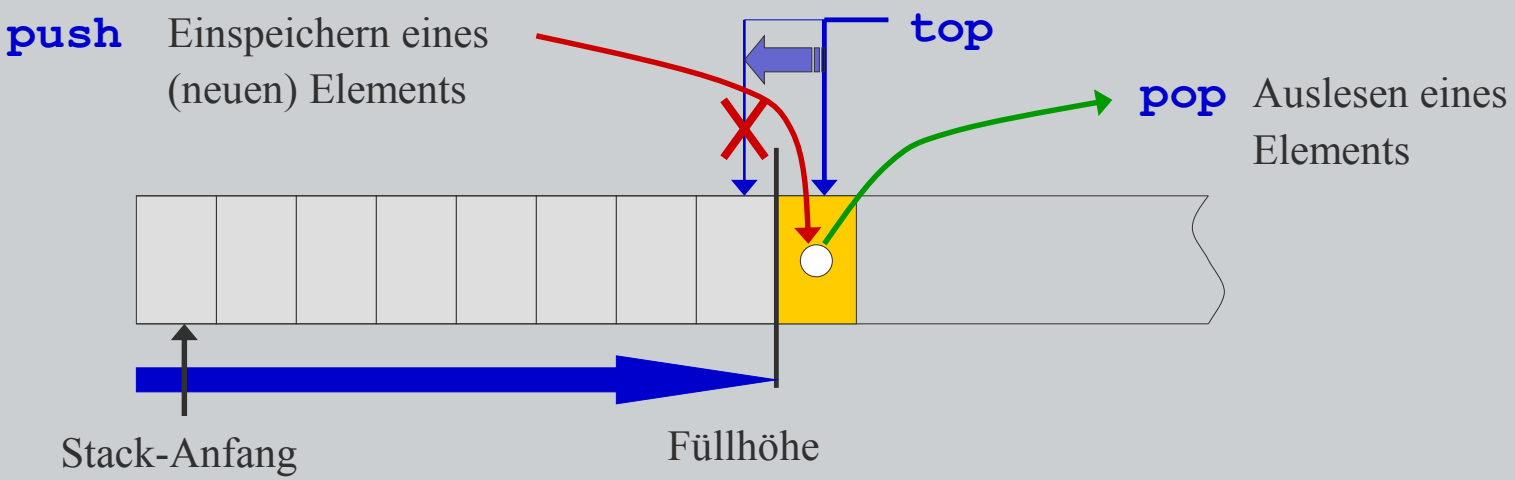
Schlangen (queues)

- **Verwaltung sequentiell abzuarbeitender Verbraucher**, z.B.
  - Verwaltung und Abarbeitung von Prozessen (scheduling),
  - Plattenzugriff,
  - Abarbeitung von Druckaufträgen,
  - etc.
- **Synchronisation asynchroner Prozesse**: Erzeuger-Verbraucher Paradigma, z.B. Tastatur → Editor (Zeichenpuffer)

Stapel (stacks)

Basisoperationen

- Anfügen eines Elements: `push` (Einspeichern eines Datenelements)
- Löschen eines Elements: `pop` (Auslesen des obersten Datenelements)



Realisierung mittels …

1. Felder (Arrays) und „Kellerpegel“ (hier: Variable top)
2. verketteter Listen (Anfügen und Löschen am Kopf (head) der Liste)

Realisierung von Stapeln mittels Feldern (Arrays)

- Klasse mit Attributen, die den Stack repräsentieren; Implementierung mit einem statischen Array mit fest definierter oder dynamischer maximaler Füllhöhe.
- Erzeugung von Stack-Objekten mit Default-Konstruktor:

```
public class StackInt {
    private final int MAX_STACK_HEIGHT = 1000;
    private int[] stack = new int[MAX_STACK_HEIGHT];
    private int   top   = 0;  // Zeiger auf oberstes Element (init = 0)

    public int pop() {                    // Element von Stack lesen
        return stack[top--];
    }

    public boolean isEmpty() {            // Stack leer?
        return (top == 0);
    }

    public void push(int value) {  // Element auf Stack speichern
        stack[++top] = value;
    }
}  // end class StackInt
```

Beachte: es fehlt die Behandlung des leeren Stacks.

Realisierung von Stapeln mittels Feldern (Arrays)

```
public static void main(String[] args) {
    StackInt s = new StackInt();

    /* -- Zahlen auf den Stack schreiben … */
    s.push(1);
    s.push(2);
    s.push(3);

    /* -- Auslesen, solange noch Zahlen auf dem 1. Stack liegen ... */
    while (!s.isEmpty())
        System.out.println(s.pop());  // Ausgabe der Elemente ...
}
```

Erläuterungen:

- Der Index auf das oberste Element im Stack ist in **`top`** gespeichert.
- Die separate Verwaltung der Listenelemente bzw. des freien Speicher entfällt, denn der Speicher der Stack-Elemente **`stack`** enthält beides:
  - Listenelemente in **`stack[0 .. top]`**
  - Freier Speicher in **`stack[top+1 .. 1000]`**

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Realisierung von Stapeln mittels verketteter Listen

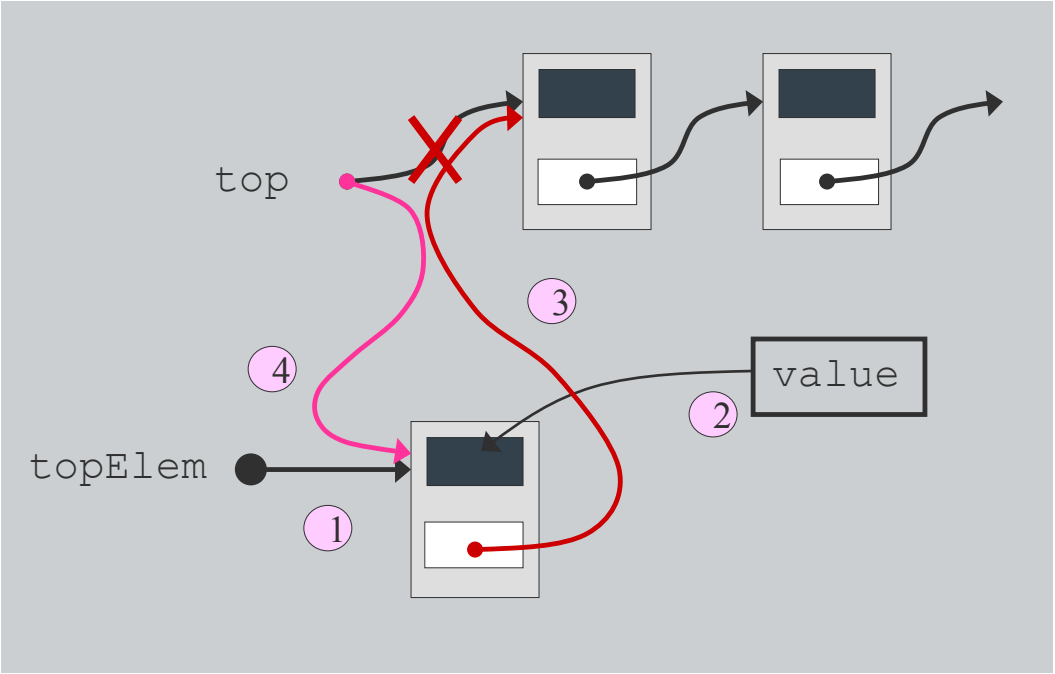
```
public class StackIntAsList {
   private ListElemInt top = null;

   /**
    * Pop the top-value of the stack and return its value.
    *
    * @return the value of the top element.
    */
   public int pop() {
      int value = this.top.item;
      this.top = this.top.next;
      return value;
   }

   /**
    * @return true if empty.
    */
   public boolean isEmpty() {
      return (this.top == null);
   }

   /**
    * Push an additional value onto the stack.
    * @param value
    */
   public void push(int value) {
      ListElemInt topElem = new ListElemInt(value, this.top);
      this.top     = topElem;
   }
}
```

```
class ListElem {
    int      item;
    ListElem next;
}  // end class ListElem
```

Funktionsweise der `push`-Operation

```
public void push(int value) {  // Element auf Stack speichern
    ListElemInt topElem = new ListElemInt(value, this.top);
    this.top     = topElem;
}
```

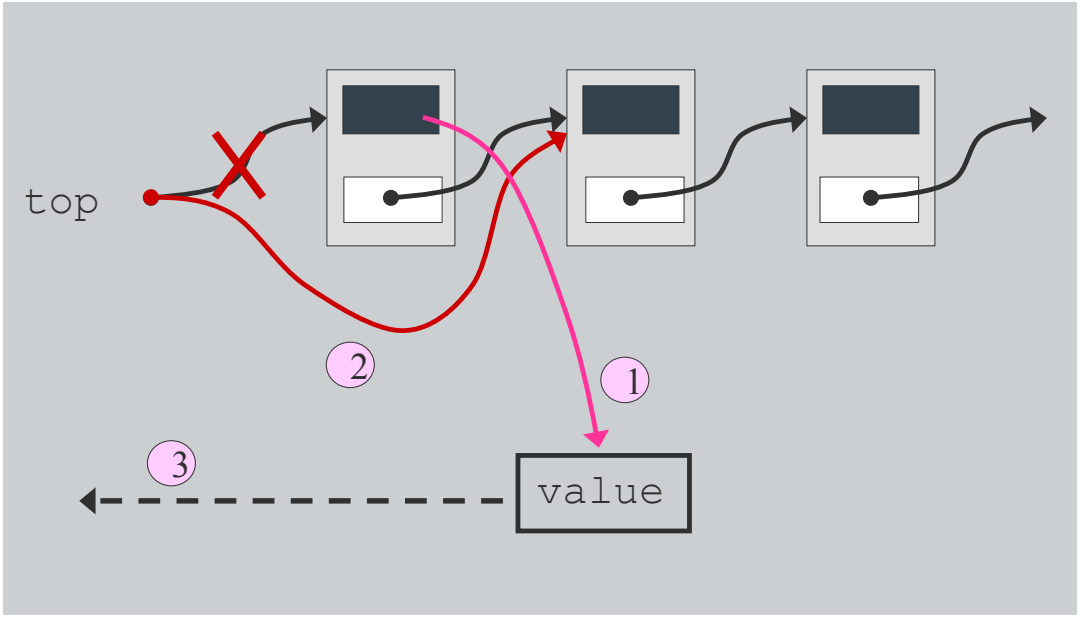
1. neues Listen-Element anlegen
2. `value` ablegen
3. neues Element vor die restlichen Elemente des *Stacks*
4. neues Element als neuen Listen-Anfang speichern



Funktionsweise der pop-Operation

```
public int pop() {                    // Element von Stack lesen
  int value = this.top.item;          // Wert des obersten Elements speichern
  this.top = this.top.next;           // oberstes Element loeschen
  return value;                       // Wert zurueck liefern
}
```

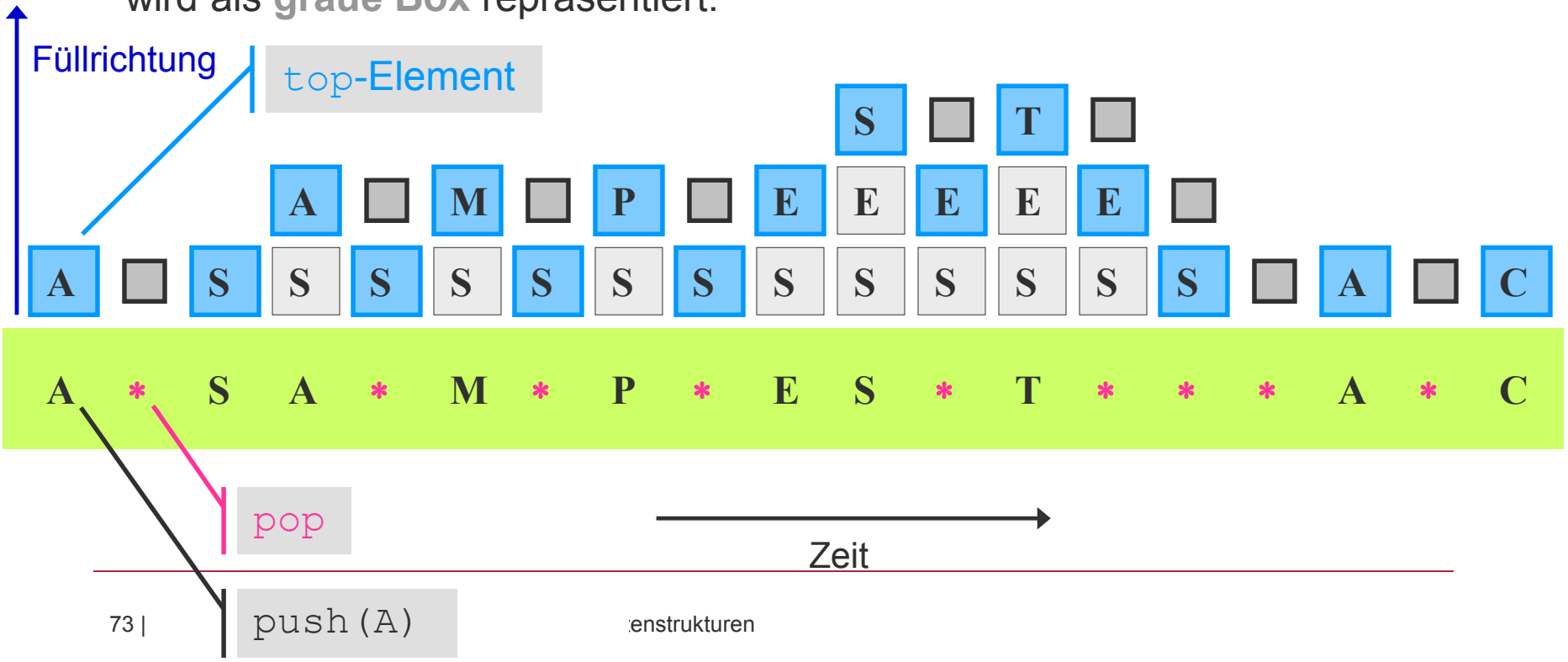
1. Wert des obersten Elements lesen und speichern.
2. Oberstes Listen-Element wird aus dem *Stack* entfernt.
3. Wert des obersten Elements wird als Ergebniswert zurück geliefert.



Bemerkung: Für diese Realisierung der pop-Operation gilt dieselbe Beachtung wie bei der Realisierung von *Stacks* mit Feldern – es muss vor der Ausführung von pop geprüft werden, ob der *Stack* nicht leer ist (dies mittels *Exceptions* absichern)

Dynamische Eigenschaft eines Stacks

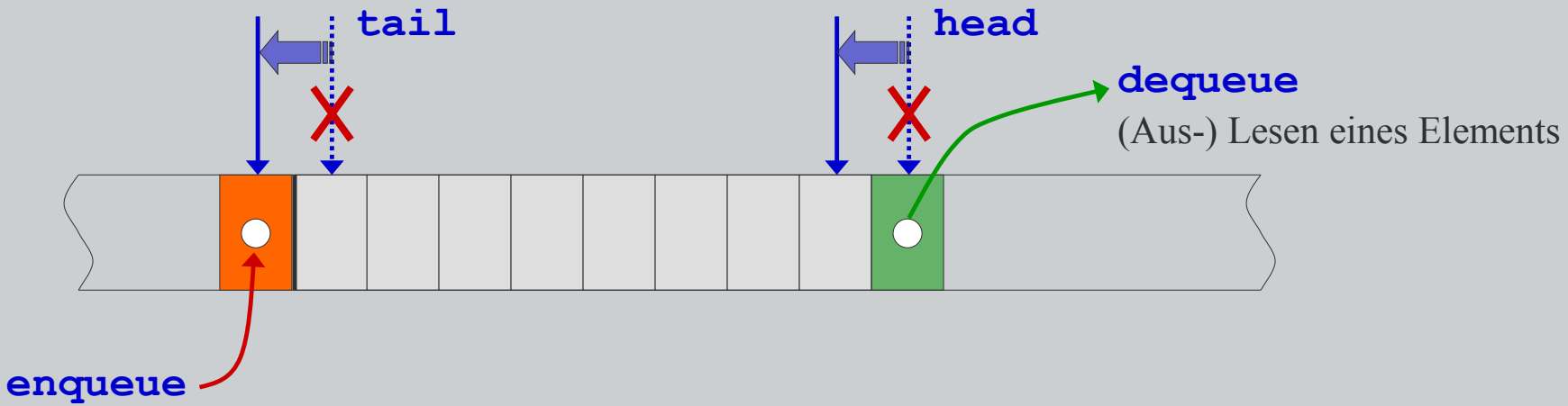
- Zur intuitiveren Darstellung werden hier **`char`**-Elemente für die Ablage verwendet.
- Die momentan zugreifbaren (obersten) Elemente werden in **BLAU** dargestellt; das nach einer **`pop`**-Operation frei gewordene Stack-Element wird als **graue Box** repräsentiert.



Schlangen

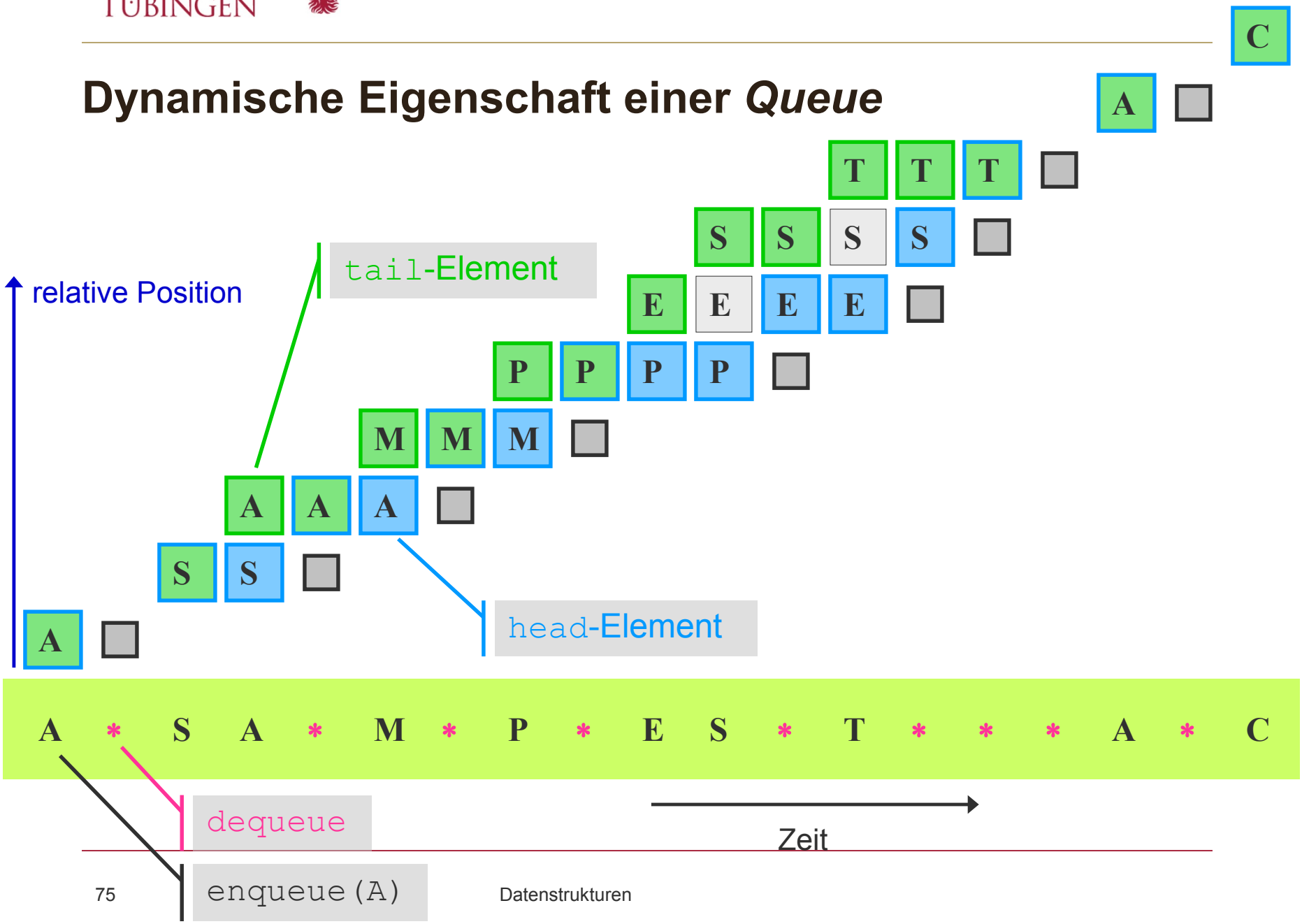
Basisoperationen

- Anfügen eines Elements: **enqueue** (Einspeichern hinten) (push)
- Löschen eines Elements: **dequeue** (Auslesen vorne) (pop)



Realisierung (wie bei Stack) mittels Felder oder verketteter Listen.

Dynamische Eigenschaft einer *Queue*

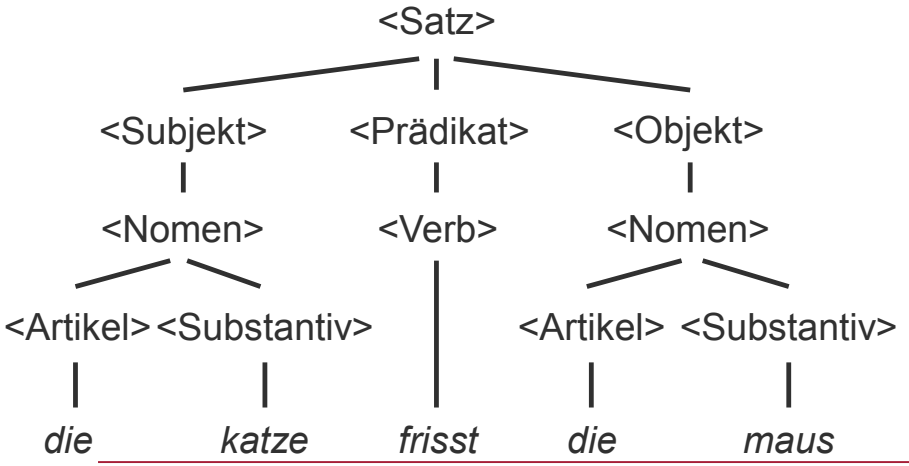


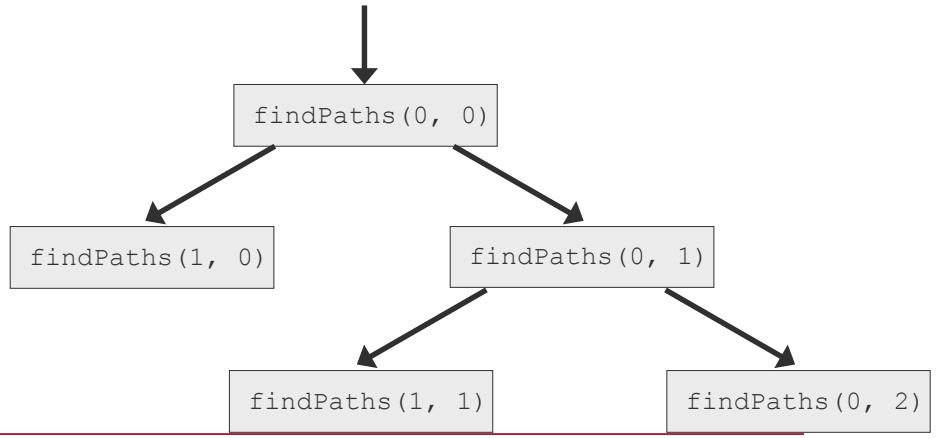
BÄUME

- Motivation und Einordnung
- Definitionen und Eigenschaften
- Binärbäume
- Operationen auf Binärbäumen
- Geordnete Binärbäume – Suchbäume und Operationen
- Eigenschaften von Binärbäumen
- Repräsentation allgemeiner Bäume
- Termbäume – Auswertung arithmetischer Ausdrücke

Motivation und Einordnung

- Bei vielen Aufgabenstellungen und Lösungsverfahren (Algorithmen, Datenstrukturen) hat ein Element mehrere Nachfolger.
- Beispiele sind …
  - Struktur von Ausdrücken, Termen, usw. bei Sprachen und Grammatiken (vgl. Ableitungsbäume, Syntaxbäume, …)
  - Dateisystem in Betriebssystemen, Dateistruktur (Windows, UNIX, …)
  - Organisationsstrukturen in Unternehmen, Verwaltungen, …
  - Rekursive Methoden mit baum-/kaskadenartiger Aufrufstruktur

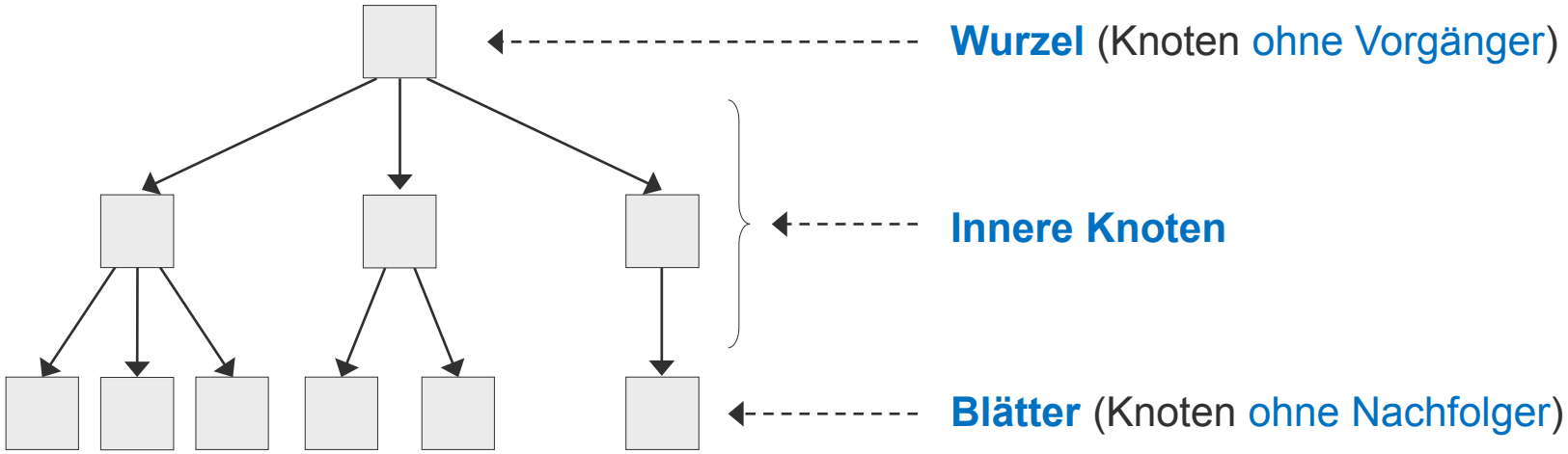




Bäume – Struktur

- **Darstellung**:
  - In der Informatik werden Bäume meist „auf dem Kopf stehend“ dargestellt.
  - Die **Wurzel** (= Ausgangselement) **steht oben.**

- Die **Knoten** in Bäumen haben bestimmte Rollen und Bezeichnungen.



- **Hinweis**:
  Die Knoten entsprechen den Elementen in Listen und enthalten die `items`.

Listen und Bäumen

Bäume können als verallgemeinerte Listenstruktur aufgefasst werden:

- In einer **Liste** hat ein Element höchstens **einen Nachfolger.**
- In einem **Baum** hat ein Element im allgemeinen **mehrere Nachfolger** (es kann aber auch nur einen oder keinen Nachfolger für ein Element geben).

Definition eines Baums

Ein **Baum** T ist ein Tupel

$$G \equiv T(V, E),$$

mit einer Knoten-Menge $\mathbf{V} = \{v_i \mid i \in \mathbf{IN}\}$ (*vertices*; auch **V**(**T**)), mit $0 \leq \text{card}(\mathbf{V}) < \infty$ und einer Kanten-Menge $\mathbf{E} = \mathbf{V} \times \mathbf{V}$ (*edges*; **E**(**T**)).

Die Kanten sind **gerichtet**; d.h. sie sind über eine irreflexive (d.h. nicht auf sich selbst verweisende) nicht-symmetrische Relation auf den Knoten festgelegt, mit

$$E = \{(v_1, v_2), (v_2, v_3), (v_3, v_2), ...\}$$

so dass $\mathbf{E} = \{e_1 = v_1v_2, e_2 = v_2v_3, ...\}$ mit $e \in \mathbf{E}$ durch jeweils

- einen Anfangs-Knoten v_i und
- einen End-Knoten v_j

festgelegt wird.

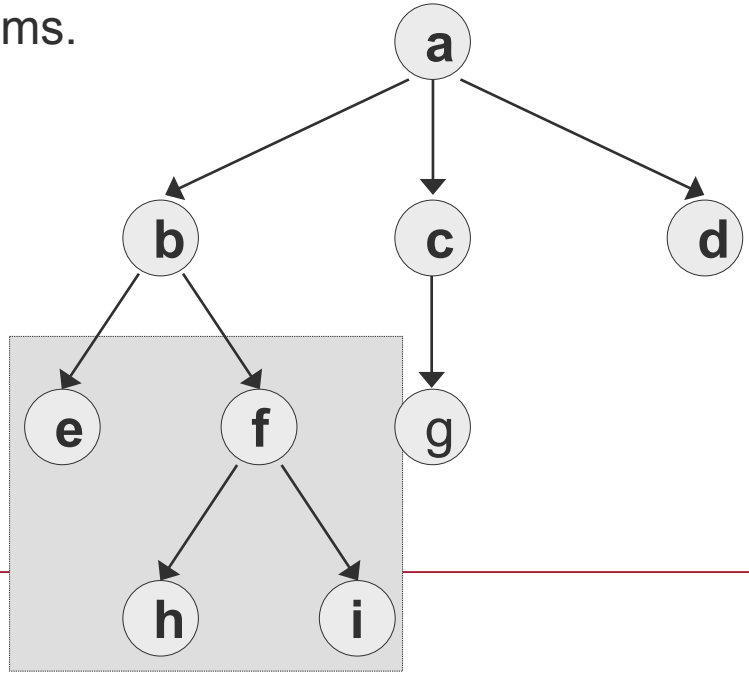
Zusätzlich gelten die Eigenschaften, dass

- jeder Knoten höchstens eine eingehende Kante hat und
- keine Zyklen vorkommen dürfen.

Struktur

- Die Nachfolger eines Knotens werden als **Kinder** bezeichnet.
- Vorgängerknoten eines Kindes bezeichnet man als **Elternknoten.**
- Ein Knoten ohne Eltern heißt **Wurzel** (Wurzel-Knoten; root) des Baums.
- **Knoten ohne Kinder heißen Blätter** (oder Blatt-Knoten; leaf).
- Knoten, die keine Blätter sind, heißen **innere Knoten.**
- Jeder innere Knoten ist Wurzel des von ihm ausgehenden Teil- oder Unterbaums.

- **Hinweis**: Ein Baum ist somit eine **rekursive Datenstruktur.**
- Bsp: Knoten **b** ist Wurzel des Unterbaums {**e**, **f** {**h**, **i**} }



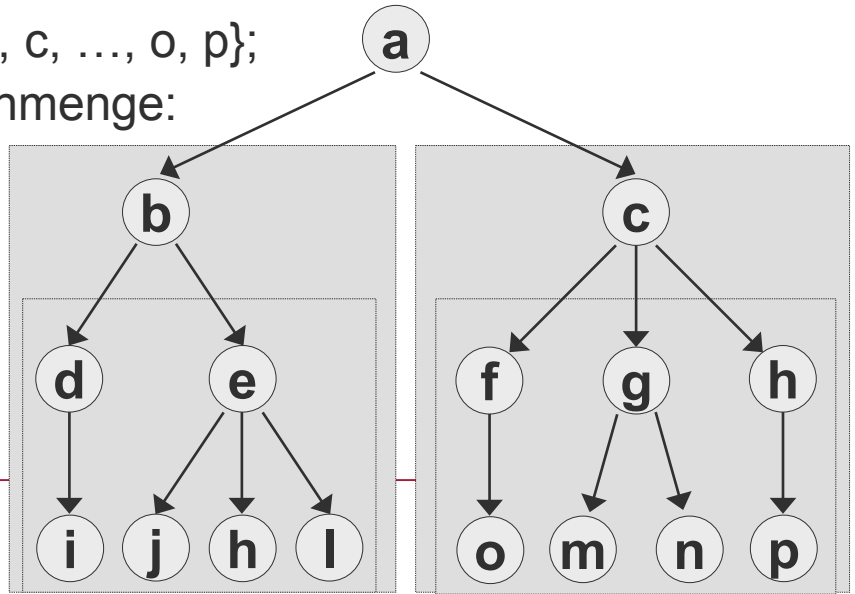
Rekursive Definition eines Baums

Ein Baum T ist eine endliche Menge V bestehend aus Elementen eines Typs mit folgenden Eigenschaften:

- die Menge V ist entweder leer („leerer Baum“) oder
- es existiert ein ausgezeichnetes Element (Knoten), der die Wurzel (root) des Baums T bildet,
- die übrigen Elemente zerfallen in disjunkte Mengen, die ebenfalls Bäume bilden.

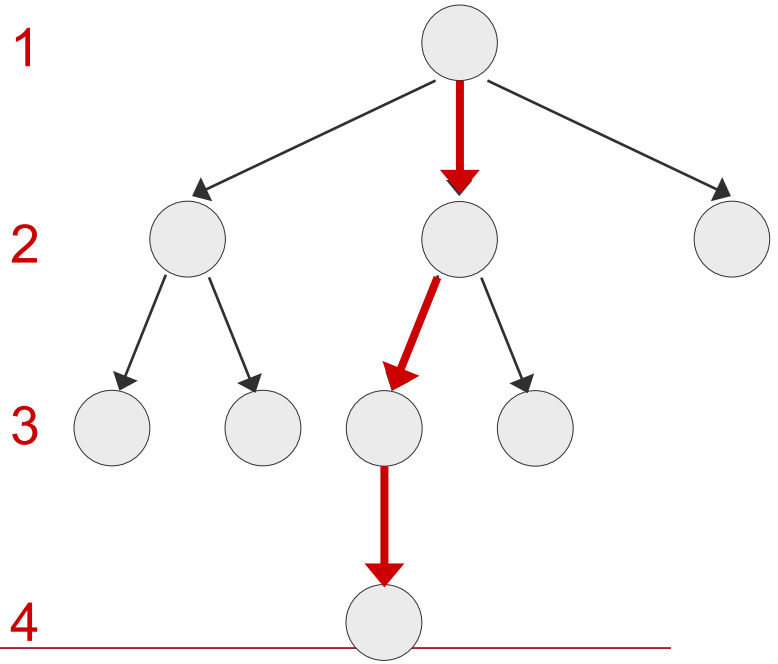
Bsp.: Geg. sei die Knotenmenge V = {a, b, c, …, o, p};
Möglicher Baum T mit dieser Knotenmenge:

```
T = {a {b {d {i},
           e {j, h, l}},
       c {f {o},
          g {m, n},
          h {p} } } }
```



Eigenschaften von Bäumen

- Eine **Kante** (edge) ist die Verbindung zwischen einem Knoten und einem seiner Kinder.
- Ein **Pfad ist eine Knotenfolge** entlang von Kanten.
- Jedem Knoten ist eine **Ebene (level)** zugeordnet; diese Ebene entspricht **der Länge des Pfades von der Wurzel** bis zu dem betreffenden Knoten.
- Die **Höhe** (auch **Tiefe**) eines Baums ist die maximale Anzahl an Ebenen, die ein Pfad von der Wurzel bis zu einem Blatt durchlaufen kann.
- Bsp.: Höhe / Tiefe = 4



Verzweigungsgrad

- Der (**Verzweigungs-) Grad (degree)** eines Knotens **$\deg_T(v)$** ist die Anzahl seiner Kinder.

- Ein **n-ärer Baum (n-ary tree)** ist ein Baum, dessen Knoten höchsten den Grad n besitzen.

- Ein **Binärbaum (binary tree)** ist ein Baum, dessen Knoten höchstens **Grad 2** besitzen.

- Der Grad beeinflusst
  - die Höhe des Baums
  - den Speicherbedarf für jeden inneren Knoten (naiv).

Binärbäume

Definition

- Ein Binärbaum hat einen Verzweigungs-Grad $\deg_T(v) \leq 2, \quad \forall\, v \in V(T)$

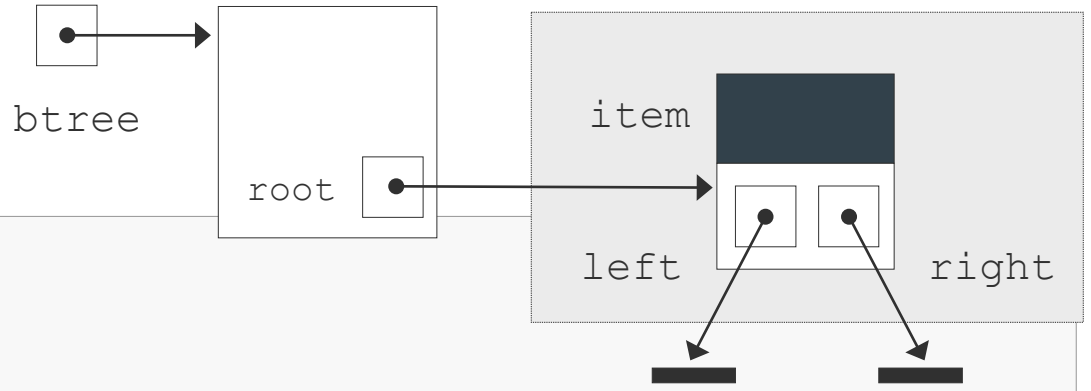
Definition (Binärbaum, abstrakte Definition):

Ein Binärbaum $\mathbf{T_B}$ ist entweder
- **leer** (*empty*), oder
- er besteht aus
  - **einem Knoten**, der einen Wert des Elementtyps enthält sowie
  - **maximal zwei Teilbäumen**, die wiederum Binärbäume sind.

Objektorientierte Realisierung in Java

Einfache Verkettung der Knoten

- Die Realisierung eines Binärbaums ist kaum aufwändiger als der Aufwand für eine einfach verkettete Liste.
- Deklaration eines Knotens




```
public class NodeBinary {
    char item;
    NodeBinary left;
    NodeBinary right;

    public NodeBinary(char item) {
        this.item = item;
    }

    public NodeBinary(char item, NodeBinary left, NodeBinary right) {
        this.item  = item;
        this.left  = left;
        this.right = right;
    }
}
```

Klasse für Binärbäume

- Wie bei den linearen Listen wird neben der Klasse für die Elemente (hier: **`Node`**) eine Klasse `BTree` für die Verwaltung von Binärbäumen deklariert.

```
public class TreeBinary {

    private NodeBinary root;  // Wurzelknoten des Baums

    public TreeBinary() {
        this.root = null;
    }

    public TreeBinary(char value) {
        this.root = new NodeBinary(value);
    }

    // … weitere Methode folgen hier.
```

Operationen auf Binärbäumen

Durchlaufen (Traversieren) von Binärbäumen

Für das Durchlaufen von (Binär)-bäumen gibt es verschiedene Vorgehensweisen.
Man unterscheidet je nach Durchlaufrichtung:

- **Tiefendurchläufe:**
  - Besuche alle Kinderknoten, **bevor** der nächste innere Knoten auf dem selben Level besucht wird.

- **Breitendurchläufe**
  - Besuche alle innere Knoten auf dem selben Level, bevor ein Kinderknoten besucht wird.

Strategien beim Tiefendurchlauf

- Ausgehend von einem Knoten `k` wird ein Unterbaum von `k` vollständig durchlaufen, bevor der zweite Unterbaum durchlaufen wird.

- Je nachdem, ob ein Knoten `k` vor, zwischen oder nach seinen Unterbäumen bearbeitet wird, unterscheidet man beim Tiefendurchlauf zwischen der:
  - **Pre-Order** Strategie,
  - **In-order** Strategie, bzw.
  - **Post-order** Strategie.

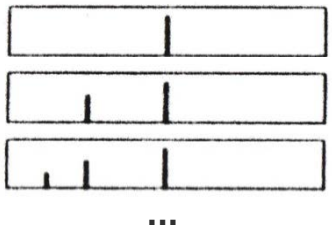
  ➢ **Hinweis**: Die Bearbeitung eines Knotens ist im einfachsten Fall der Besuch und die Anzeige des darin gespeicherten Inhalts.

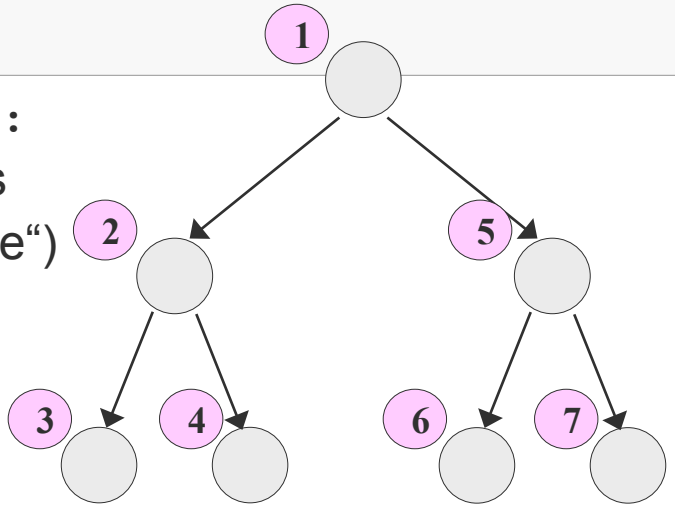
Pre-order Traversierung

- Der **Knoten** `k` wird **bearbeitet** (hier: `visit` mit Ausgabe des Inhalts) ***bevor*** seine **beiden Unterbäume bearbeitet** werden:

```
void traverse(Node n) {
    if (n != null) {  // wenn Blatt erreicht, keine Aktion
        visit(n.item);
        traverse(n.left);    // Bearbeitung linker Unterbaum
        traverse(n.right);   // Bearbeitung rechter Unterbaum
    }
}  // end traverse

void visit(char content) {
    System.out.print(content + " ");
}  // end visit
```

- Reihenfolge der Aufrufe von `visit(...)`:
- **Hinweis**: Vergleiche die Markierung eines Lineals (als Beispiel für „Teile-und-herrsche“)



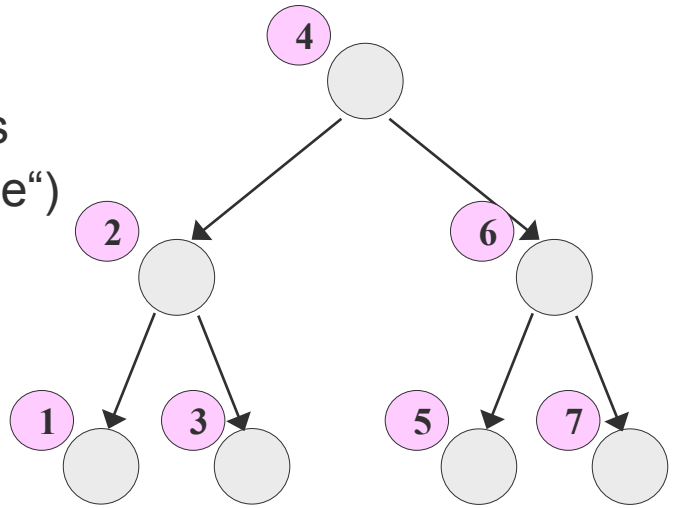


In-order Traversierung

- Der **Knoten** `k` wird **zwischen** der Bearbeitung seiner **beiden Unterbäume bearbeitet:**

```
void traverse(Node n) {
    if (n != null) {  // wenn Blatt erreicht, keine Aktion
        traverse(n.left);   // Bearbeitung linker Unterbaum
        visit(n.item);
        traverse(n.right);  // Bearbeitung rechter Unterbaum
    }
}  // end traverse
```

- Reihenfolge der Aufrufe von `visit(...)`
- **Hinweis**: Vergleiche die Markierung eines Lineals (als Beispiel für „Teile-und-herrsche“)

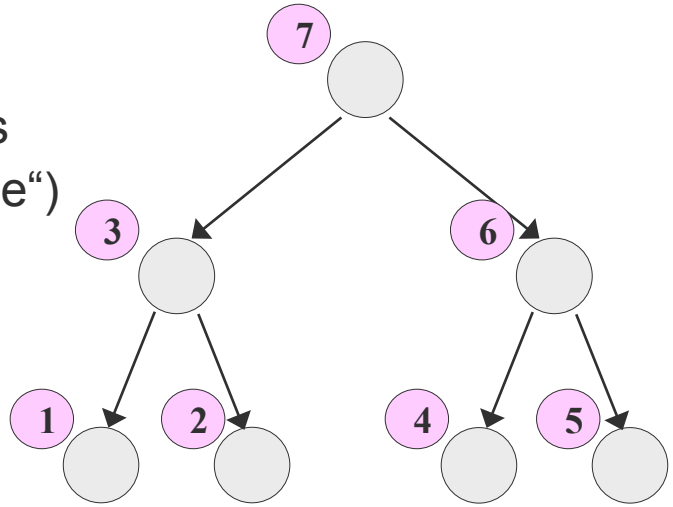


Post-order Traversierung

- Der **Knoten** `k` wird **nach** der Bearbeitung seiner **beiden Unterbäume bearbeitet**

```
void traverse(Node n) {
    if (n != null) {  // wenn Blatt erreicht, keine Aktion
        traverse(n.left);   // Bearbeitung linker Unterbaum
        traverse(n.right);  // Bearbeitung rechter Unterbaum
        visit(n.item);
    }
}  // end traverse
```

- Reihenfolge der Aufrufe von `visit(...)`
- **Hinweis**: Vergleiche die Markierung eines Lineals (als Beispiel für „Teile-und-herrsche“)

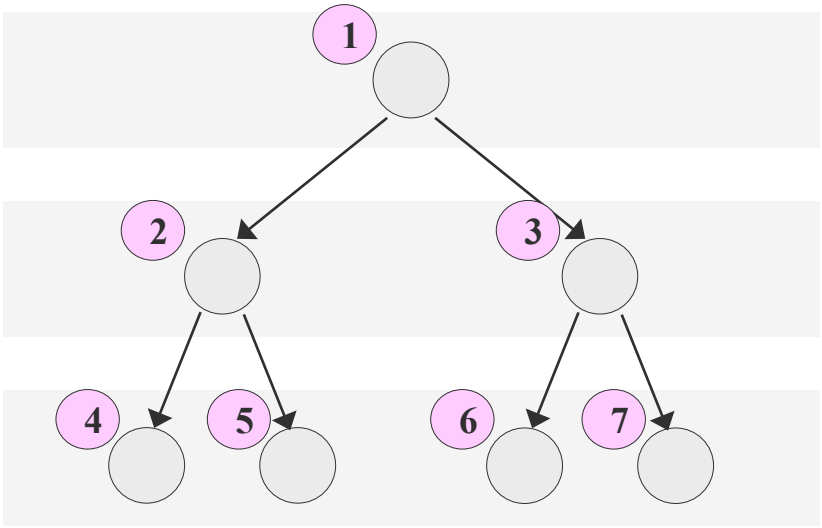


Breitendurchlauf in Binärbäumen

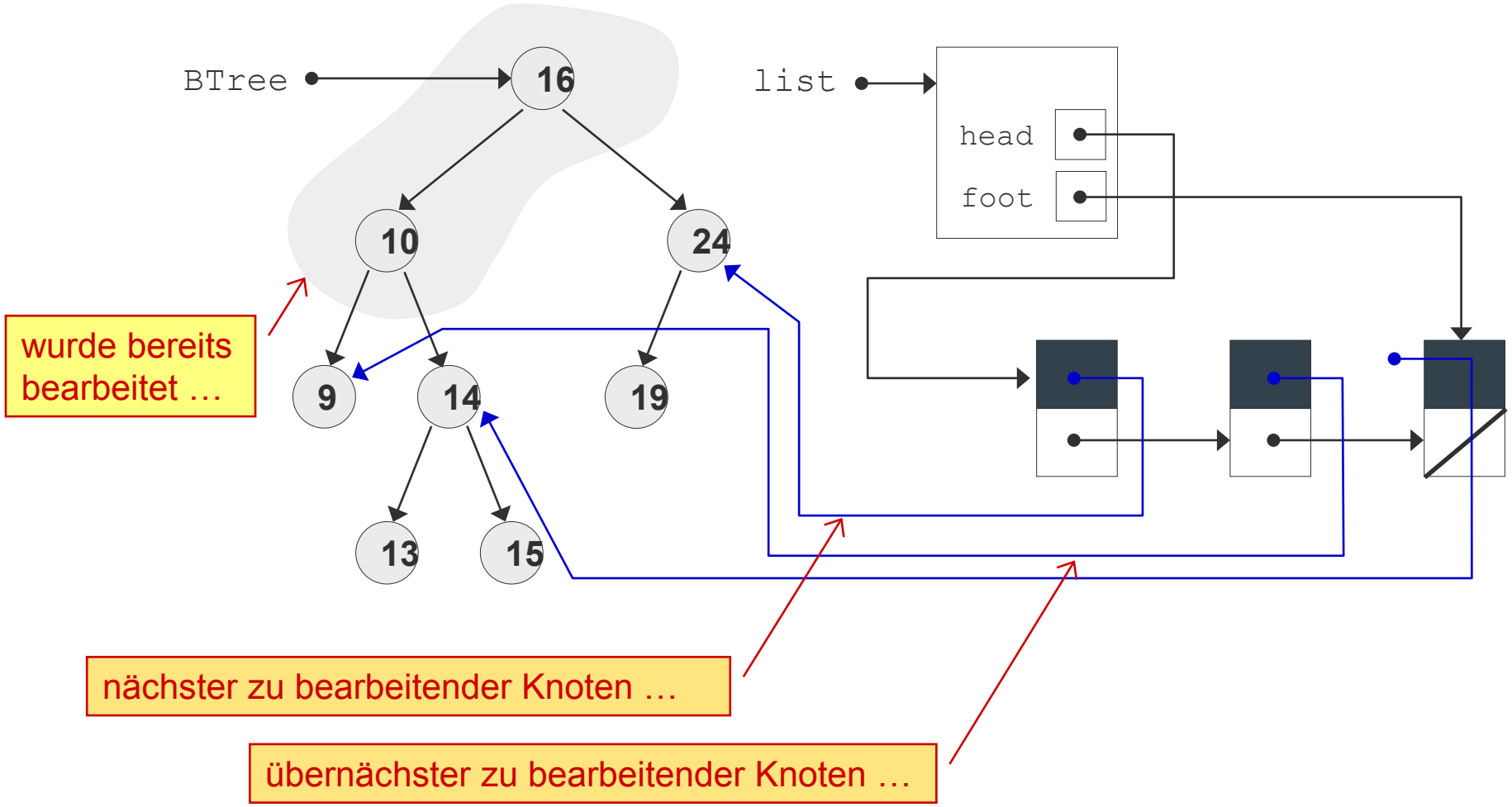
- Beim Breitendurchlauf wird der Baum ebenen-weise durchlaufen:

  Hinweis: Zum Durchlauf wird eine Schlange benötigt, um alle Kinderknoten der aktuellen Ebene aufzunehmen.

- Zeitliche Abfolge



Breitendurchlauf – konkretes Beispiel



Suche eines Elements

- Bei der **Suche nach einem Element** (Knoten) in einem (Binär-) Baum **ohne spezielle Ordnung** der Elemente, muss der gesamte Baum betrachtet werden

```
public NodeBinary searchBTree(int content) {
    return searchBTree(content, this.root);
}

private NodeBinary searchBTree(int content, NodeBinary n) {
    if (n == null)
        return null;  // Element nicht vorhanden
    if (n.item == content)
        return n;     // Element gefunden
    else {
        NodeBinary result = searchBTree(content, n.left);

        if (result == null)
            result = searchBTree(content, n.right);
        return result;
    }
}  // end searchBTree
```

Absuchen des **linken** Teilbaums

Falls noch nichts gefunden, dann Absuchen des **rechten** Teilbaums

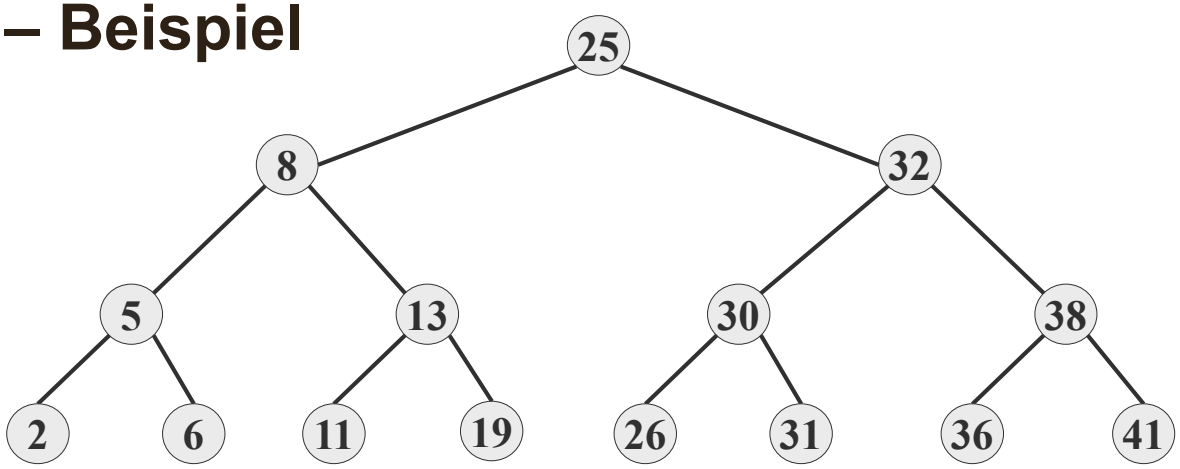
Geordnete Binärbäume – Suchbäume und Operationen

Suchen von Elementen und geordnetes Einfügen

- Sind die Elemente geordnet eingetragen (z.B. aufsteigend), so kann bei der **Suche die Ordnung ausgenutzt werden** und nur in dem Teil des (Binär-) Baums gesucht werden, in dem das gesuchte Element vorkommen kann.
- Wird eine Menge ungeordneter `int-Zahlen` (oder Elemente, für die eine Ordnung gilt) eingelesen, so sollten diese in einem Binärbaum abgespeichert werden, so dass jeder Knoten ein Element enthält und für jeden Teilbaum gilt:

Elemente im linken Teilbaum	$\leq$	**Wurzel-Element**	$<$	**Elemente im rechten Teilbaum**

Suche – Beispiel



- **Problem**: *Was macht man mit gleichen Elementen?*
- **Lösungen**: In den Baum einzufügende Elemente, die mit bereits im Baum eingefügten Elementen überein stimmen,
  - werden entweder ignoriert oder
  - sie werden eingefügt, wobei eine Asymmetrie des Baumes in Kauf genommen werden muss: Elemente links $\leq$ Wurzel $<$ Elemente rechts.

➢ **Anmerkung**: In der Folge gehen wir davon aus, dass alle Elemente paarweise verschieden sind!

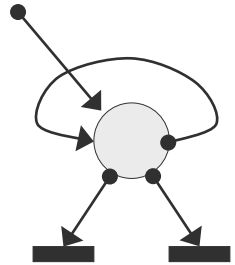
Sortiertes Einfügen eines Elements (1)

Generierung eines Knotens

- Für ein gegebenes Element (elem) wird ein Knoten generiert, der anschließend eingefügt werden muss.
- Knoten erzeugen:

```
/* Generierung eines Knotens */
Node nodeNew      = new Node(elem);
```

nodeNew



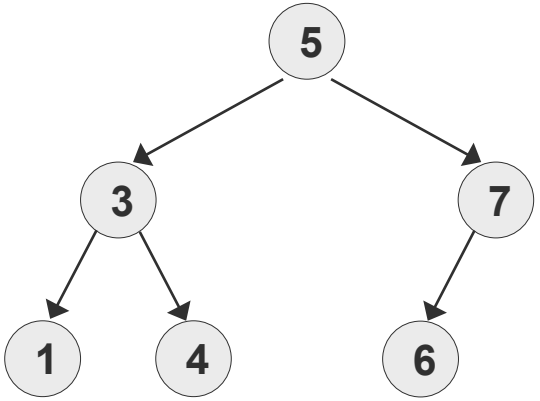
Sortiertes Einfügen eines Elements (2)

Sortiertes Einfügen

```
/* Generierung eines Knotens */
Node nodeNew      = new Node(elem);
```

- Vorbetrachtung: Ein **neues Element** kann an **unterschiedlichen Positionen** des Baums eingefügt werden:
  - Einfügen **als Blatt** oder
  - Einfügen **als innerer Knoten**

- Bsp.: Einfügen von Werten 5, 7, 3, 6, 4, 1
  (Ann.: Die Knoten wurden jeweils generiert)



Rekursiver Algorithmus

- Geg.:
  - Zeiger `x` referenziert neuen einzufügenden Knoten.
  - Zeiger `tree` referenziert die Baumwurzel
- Wenn Baum noch leer (`tree = null`),
  dann wird der neue Knoten zur Wurzel (`tree := x`) – fertig!
  sonst Vergleich der Elemente:
  wenn `x`-Element < `tree`-Element
  dann Einbau des Knotens in den linken Teilbaum,
  `insertNodeSorted(x, tree.left)`
  sonst Einbau des Knotens in den rechten Teilbaum,
  `insertNodeSorted(x, tree.right)`
- Ist die richtige Position des Knotens gefunden, wird das Element an diese Position gesetzt in dem der rechte bzw. linke Zeiger des Elternknotens auf den neuen Knoten zeigt.

- **Kommentar**: Die Rekursion bildet in natürlicher Form die Bearbeitung der separaten Teilbäume ab, die von einem besuchten Knoten als Wurzel aufgespannt werden.

Sortiertes Einfügen

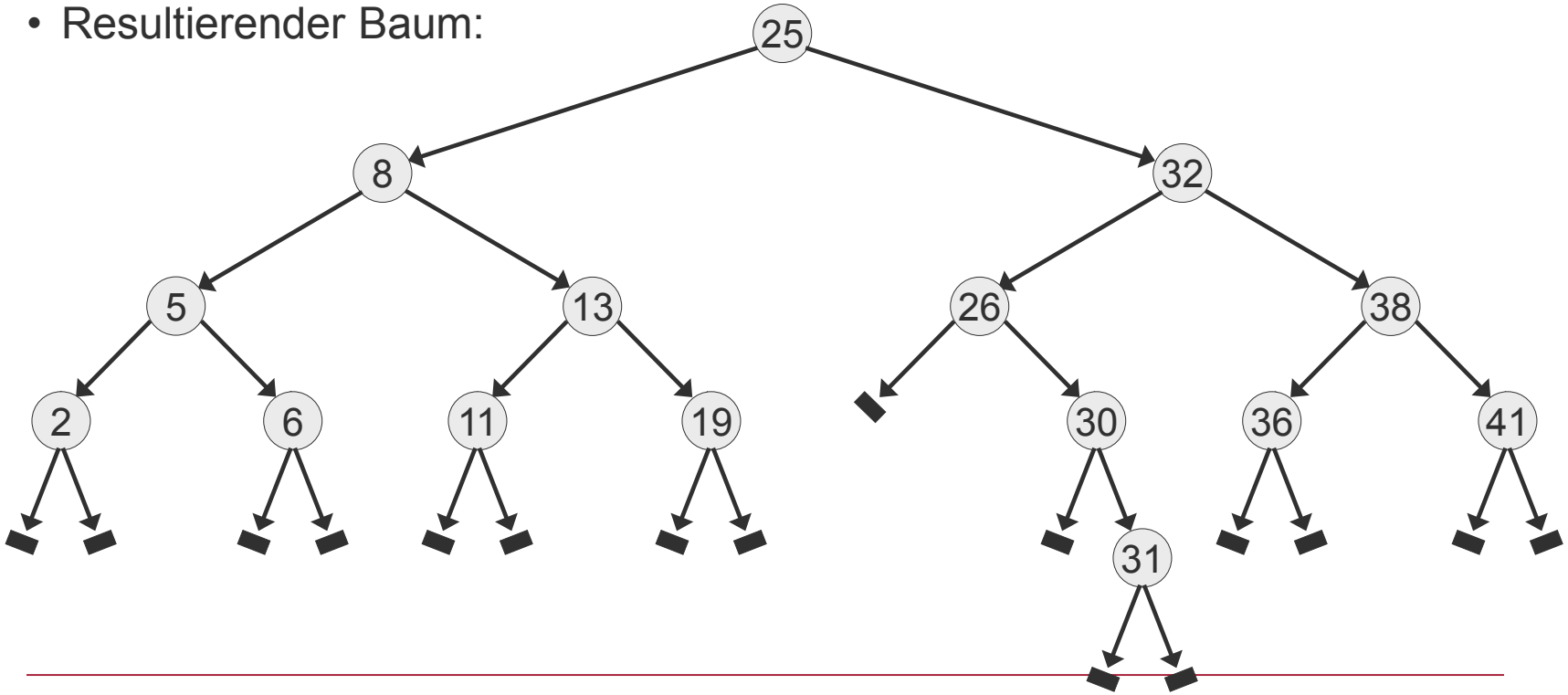
```
public void insertBTreeNodeSorted(NodeBinary x) {
    if (this.root == null)  // tree is still empty
        this.root = x;
    else                    // tree contains nodes
        insertBTreeNodeSorted(x, this.root);
}

private void insertBTreeNodeSorted(NodeBinary x, NodeBinary tree) {
    if (x.item <= tree.item) {// insert to the left
        if (tree.left == null) {
            tree.left = x; // found location - position it here.
        }else{
            insertBTreeNodeSorted(x, tree.left);
        }
    } else { // insert to the right
        if (tree.right == null) {
            tree.right = x; // found location - position it here.
        }else{
            insertBTreeNodeSorted(x, tree.right);
        }
    }
}
```

Beispiele zum Einfügen von Knoten

- Beispiel 1 – Ungeordnete Eingabesequenz
- Folge:

25	8	32	5	2	6	13	11	19	26	30	31	38	36	41

- Resultierender Baum:

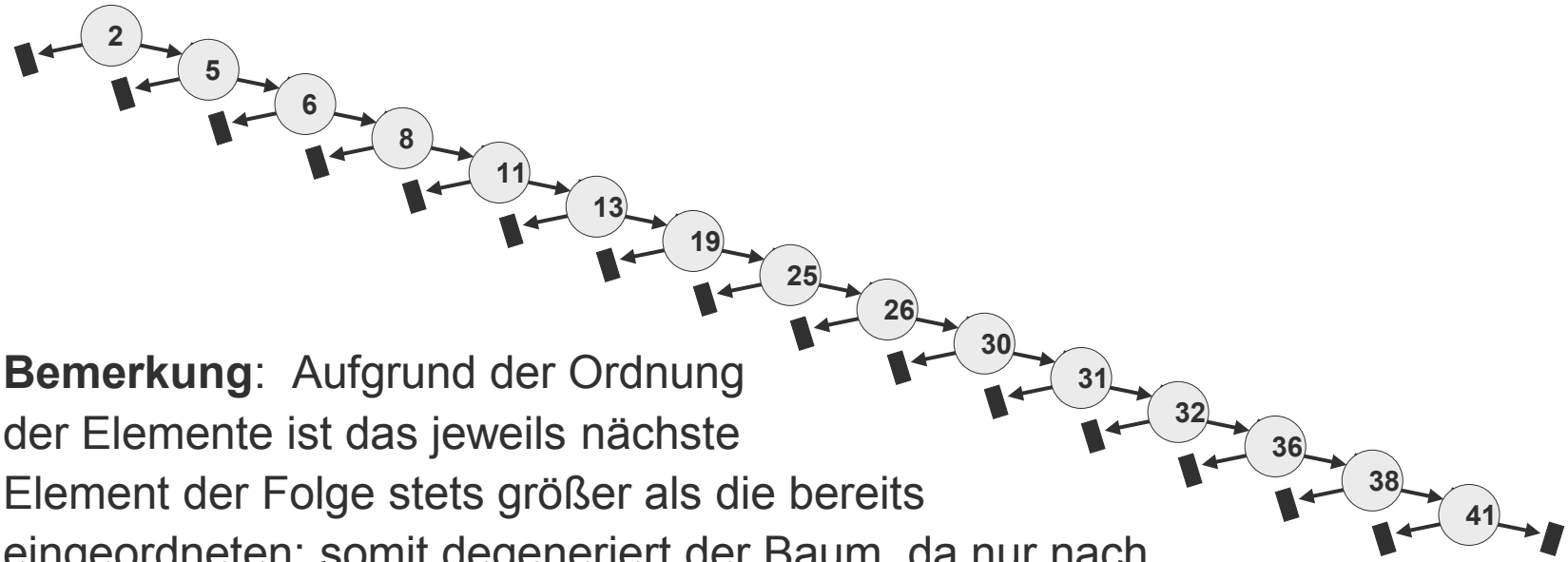


Beispiele zum Einfügen von Knoten (2)

- Beispiel 2 – Geordnete Eingabesequenz, mit denselben Elementen der Größe nach geordnet
- Folge:

2	5	6	8	11	13	19	25	26	30	31	32	36	38	41

- Resultierender Baum:



- **Bemerkung**: Aufgrund der Ordnung der Elemente ist das jeweils nächste Element der Folge stets größer als die bereits eingeordneten; somit degeneriert der Baum, da nur nach rechts eingefügt wird.

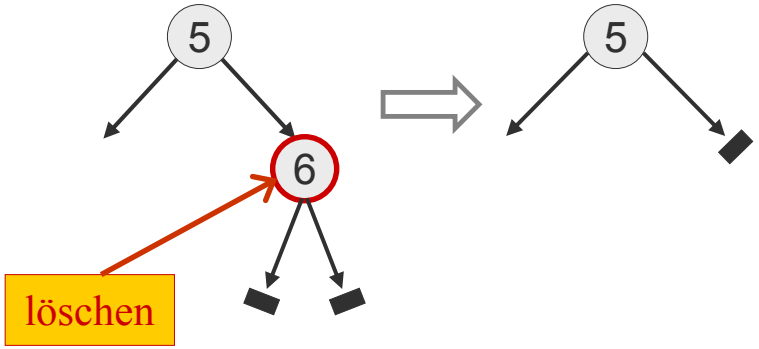
Löschen eines Knotens aus einem sortierten Baum

Vorbetrachtungen:

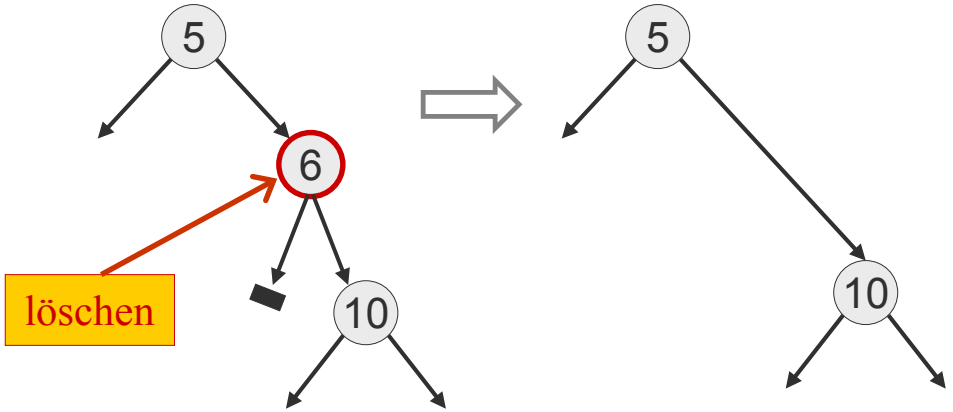
Eingabe: Zeiger `x` zeigt auf den zu löschenden Knoten, `tree` zeigt auf die Baumwurzel. Es treten 3 Fälle auf:

- **Fall 0**: Der (zu löschende) Knoten besitzt keine Nachfolger (Blattknoten)
  - **Lösung**: Verweis auf diesen Knoten im Vorgänger (parent) wird auf `null` gesetzt.
- **Fall 1**: Innerer Knoten mit **genau einem Nachfolger**, entweder `x.left == null` oder `x.right == null`
  - **Lösung**: Der Knoten wird herausgeschnitten und der anhängende Teilbaum muss „hochgezogen“ werden (wie bei einer linearen Liste).
- **Fall 2**: Innerer **Knoten mit zwei Nachfolgern**
  - **Lösung**: Im rechten Teilbaum des Knotens (enthält die Elemente größer als das Knoten-Element) wird das kleinste Element bestimmt (immer links absteigen bis zu demjenigen Knoten, der nur rechts einen Nachfolger hat oder ein Blattknoten ist). Das gefundene Element löschen (führt zu Fall 0 oder 1) und als neuen inneren Knoten verwenden.

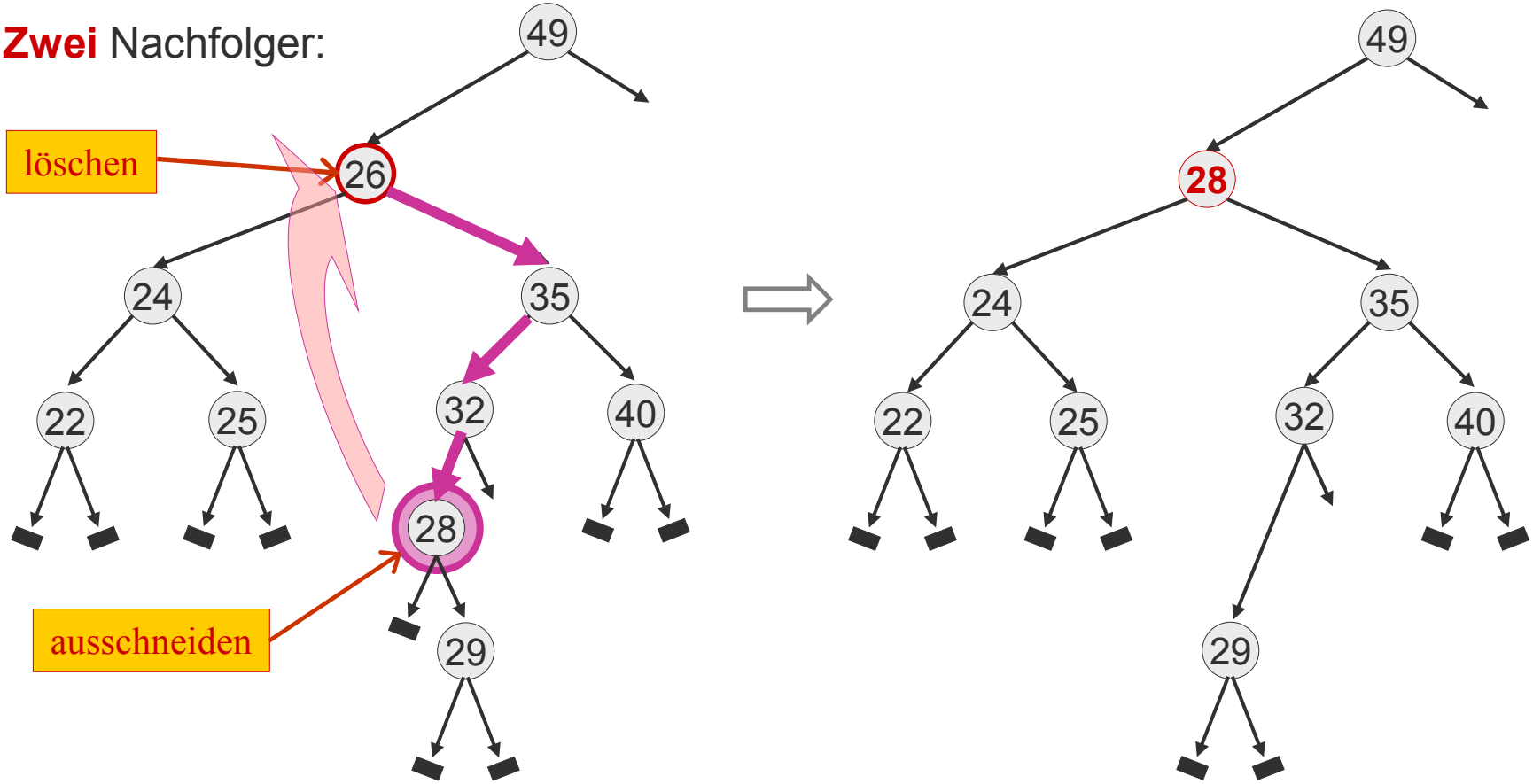
Keine Nachfolger:



Ein Nachfolger:



Zwei Nachfolger:



Methode zum Löschen von Knoten

- **Struktur**: Löschen von Knoten in einem Baum mit Basis-Methode, es werden weitere Hilfsmethoden zum Ersetzen und der Suche von Ersatzknoten definiert:

```
public void deleteTreeNodeSorted(NodeBinary x) {
    if (root == null)
    return;
    else
    root = deleteTreeNodeSorted(x, root);
}

private NodeBinary deleteTreeNodeSorted(NodeBinary x, NodeBinary tree)
{
    if (x.item < tree.item)
        tree.left  = deleteTreeNodeSorted(x, tree.left);
    else if (x.item > tree.item)
        tree.right = deleteTreeNodeSorted(x, tree.right);
    else
        tree = replaceTreeNodeSorted(tree);
    return tree;
}
```

Suchen des zu löschenden Knotens

Ersetzen des Knotens

Zusätzliche Methoden

```
public NodeBinary replaceTreeNodeSorted(NodeBinary tree) {
    if (tree.right == null)
       tree = tree.left;
    else if (tree.left == null)
       tree = tree.right;
    else
       tree.right = findReplacement(tree.right, tree);

    return tree;
}

private NodeBinary findReplacement(NodeBinary tree, NodeBinary replace)
{
    if (tree.left != null)
       tree.left = findReplacement(tree.left, replace);
    else {
       replace.item = tree.item;
       tree         = tree.right;
    }
    return tree;
}
```

Zu löschender Knoten hat einen Nachfolger; durch Nachfolger ersetzen

Zu löschender Knoten hat mehr als einen Nachfolger; Ersatz suchen

Knoten durch passenden Ersatzknoten ersetzen

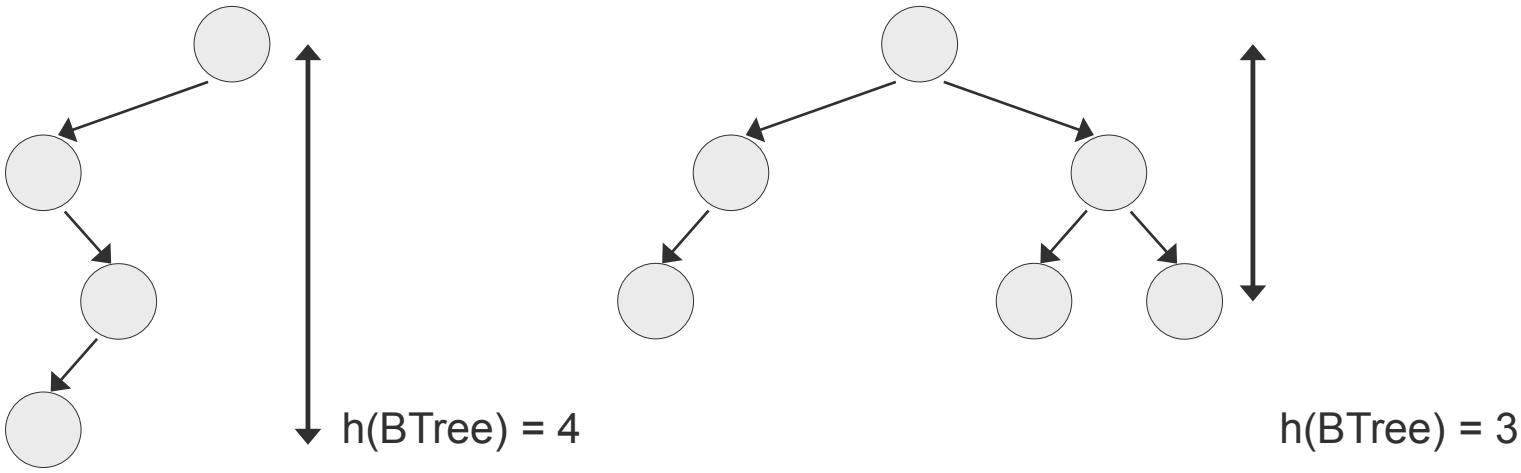
Eigenschaften von Binärbäumen

Tiefe eines Binärbaums

- Rekursive Definition der Höhe h(•) für Binärbäume

$$h(\texttt{BTree}) = \begin{cases} 1 + \max(h(\texttt{BTree.left}), h(\texttt{BTree.right})) & \text{falls } \texttt{BTree != null} \\ 0 & \text{sonst} \end{cases}$$

Bsp.:



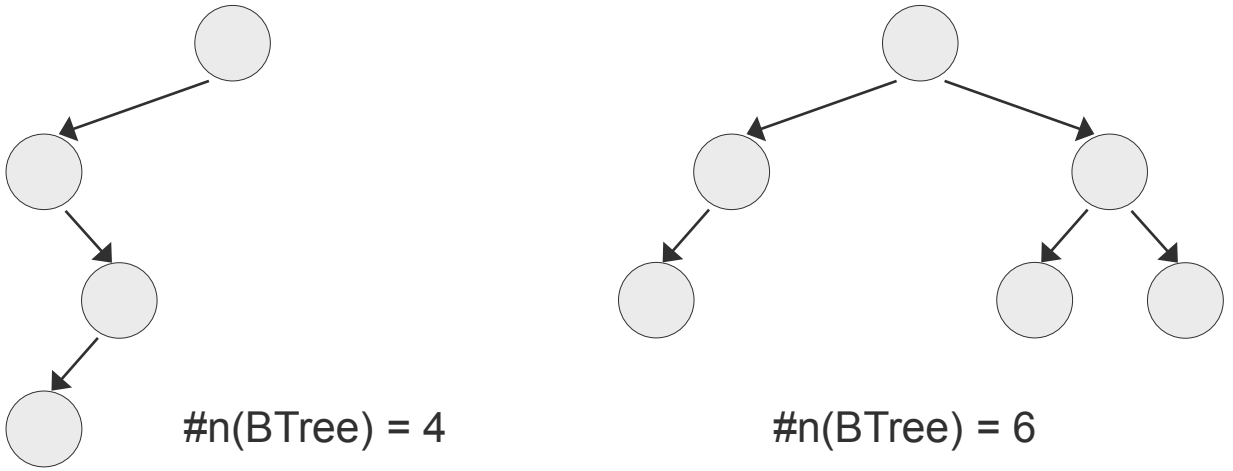
Anzahl der Knoten eines Binärbaums

Berechnung der Anzahl der Knoten eines Binärbaums

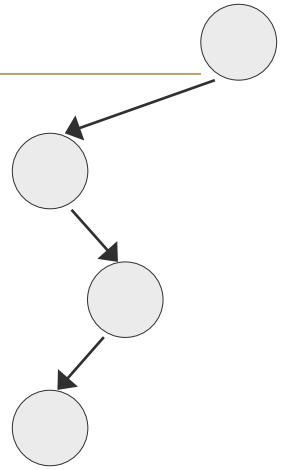
- Schema zur Berechnung von #n(•)

$$\#n(\texttt{BTree}) = \begin{cases} 1 + \#n(\texttt{BTree.left}) + \#n(\texttt{BTree.right}) & \text{falls } \texttt{BTree != null} \\ 0 & \text{sonst} \end{cases}$$

Bsp.: Anzahl der Knoten in einem gegebenen Baum



Extreme Binärbäume



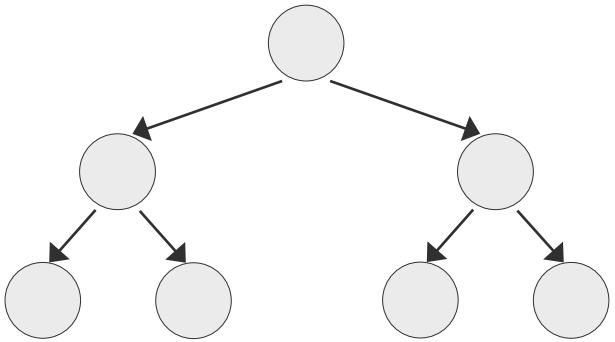
- Ist ein Binärbaum **minimal besetzt**, ist die Anzahl der Knoten gleich der Höhe des Baums

  `#n(BTree) = h(BTree)`

- **Anmerkung**: Ein minimal besetzter Baum entspricht einer linearen Liste.

Ist ein Binärbaum **voll besetzt**, so gilt

$$\#n(BTree) = 2^{h(BTree)} - 1$$



- **Herleitung**:

Ebene 1:	$1 =$		2^0
Ebene 2:	$2 \cdot 1 =$		2^1
Ebene 3:	$2 \cdot 2 \cdot 1$	$=$	2^2
Ebene 4:	$2 \cdot 2 \cdot 2 \cdot 1$	$=$	2^3
…			
Ebene n:	$2 \cdot \ldots \cdot 2 \cdot 1$	$=$	2^{n-1}

$\Rightarrow$ $\#n(BTree) = 2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$

Repräsentation allgemeiner Bäume

Allgemeine Bäume und ihre Implementierung

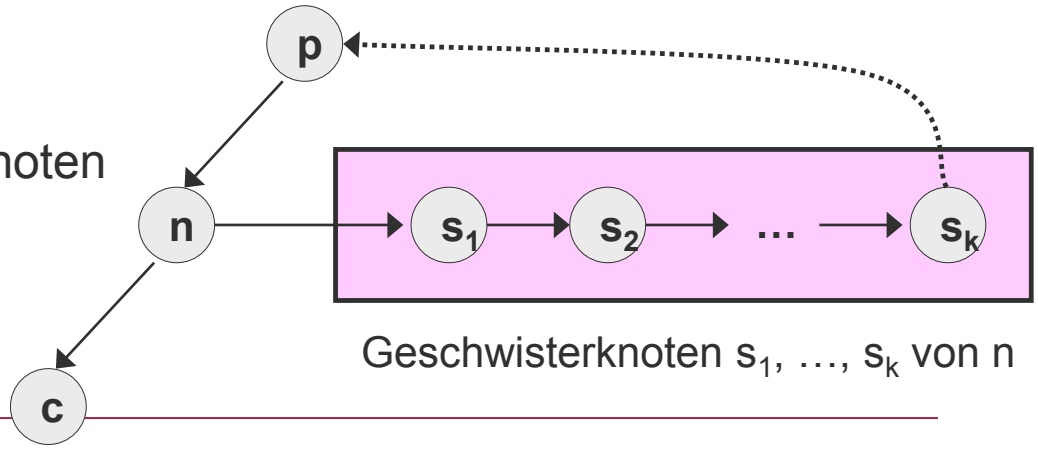
- Allgemeine Bäume besitzen Knoten mit einer variablen Anzahl von Kinderknoten.
- Die Repräsentation bestimmt die Art der Operationen und umgekehrt.
- In jedem Fall benötigt man den Zugriff auf die Wurzel sowie den Zugriff auf die Teilbäume.
- **Idee**: Verwende verkettete Listen für die Repräsentation der Kinderknoten, mit variabler Anzahl von Nachfolgern jeweils eines Knotens.

Repräsentation allgemeiner Bäume

- **Konzept**: Knoten des Baums haben genau 2 Referenzen:
  - Verweis L (links) zum 1. (am weitesten links stehenden) Kindsknoten
  - Verweis R (rechts) zu den (eigenen) Geschwister-Knoten (auf derselben Ebene)
  - Der letzte Geschwister-Knoten zeigt zurück auf den jeweiligen Eltern-Knoten (parent) – dies wird „Fädelung“ genannt.

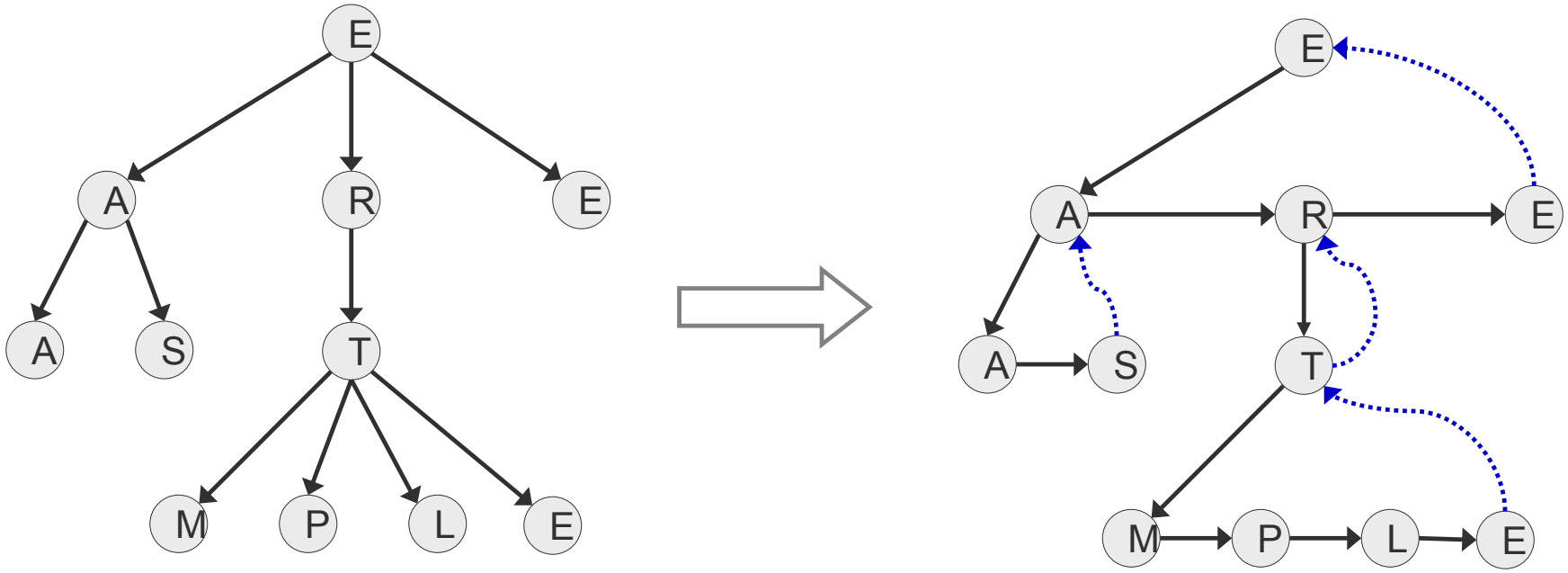
- **Anmerkung**: Die „Fädelung“ führt eine Schleife in die Repräsentation ein, die zwar nicht erforderlich aber empfehlenswert ist, z.B. für das effiziente Durchlaufen.

- Bsp.: Baum mit k Nachfolgeknoten



Beispiel für die Repräsentation

- Ein Beispielbaum mit Repräsentation der Geschwister-Knoten



- **Anmerkung**: Bäume mit einer beliebigen (variablen) Anzahl von Nachfolgern eines Knotens lassen sich auch in Binärbäume umformen – das ist aber nicht immer sinnvoll.

Termbäume – Auswertung arithmetischer Ausdrücke

Arithmetische Ausdrücke und ihre Notation

- **Infix-Notation** (3 + 4)
- **Präfix-Notation** (+ 3 4)
- **Postfix-Notation** (3 4 +)

Infix-Notation

Standard-Notation mathematischer Ausdrücke

- 1-stellige Operatoren:

```
<aus> ::= <op> <operand>.
<op>  ::= + | -.
```

- 2-stellige Operatoren:

```
<aus> ::= <operand> <op> <operand>.
<op>  ::= + | - | * | /.
```

- Strukturierung durch Klammern: `( | )`

Präfix-Notation

Notation ohne Klammern durch „Anhängen“ der Operanden an die Operatoren; Notation in **funktionalen Sprachen**

- 2-stellige Operatoren:

```
<aus> ::= <op> <operand> <operand>.
<op>  ::= + | - | * | /.
```

- n-stellige Operatoren:

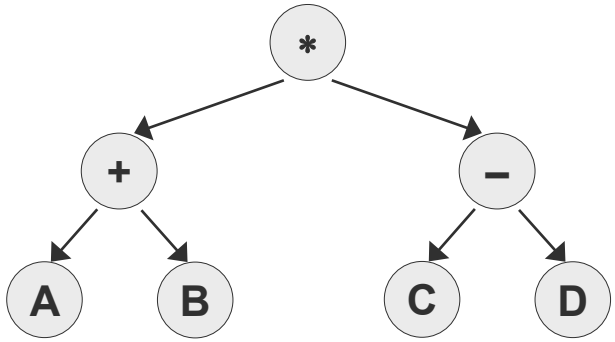
```
<aus> ::= <op> <operand> <operand> {<operand>}.
<op>  ::= + | - | * | /.
```

Postfix-Notation

- Maschinelle Auswertung von arithmetischen Ausdrücken
  - 2-stellige Operatoren: `<aus> ::= <operand> <operand> <op>.`
  - Auch hier ist keine Strukturierung durch Klammern notwendig

Beispiel für einen arithmetischen Ausdruck

- Baum-Notation (Binärbaum)



- **Infix**-Notation: (A + B) ∗ (C – D)

  Entspricht der Aufzählung der Baumknoten nach dem rekursiven ***In-order* Schema**

  `linker Teilbaum – Knoten-Inhalt – rechter Teilbaum`

- **Präfix**-Notation: ∗ + A B – C D

  Entspricht der Aufzählung der Baumknoten nach dem rekursiven ***Pre-order* Schema**

  `Knoten-Inhalt – linker Teilbaum – rechter Teilbaum`

- **Postfix**-Notation: A B + C D – ∗

  Entspricht der Aufzählung der Baumknoten nach dem rekursiven ***Post-order* Schema**

  `linker Teilbaum – rechter Teilbaum – Knoten-Inhalt`

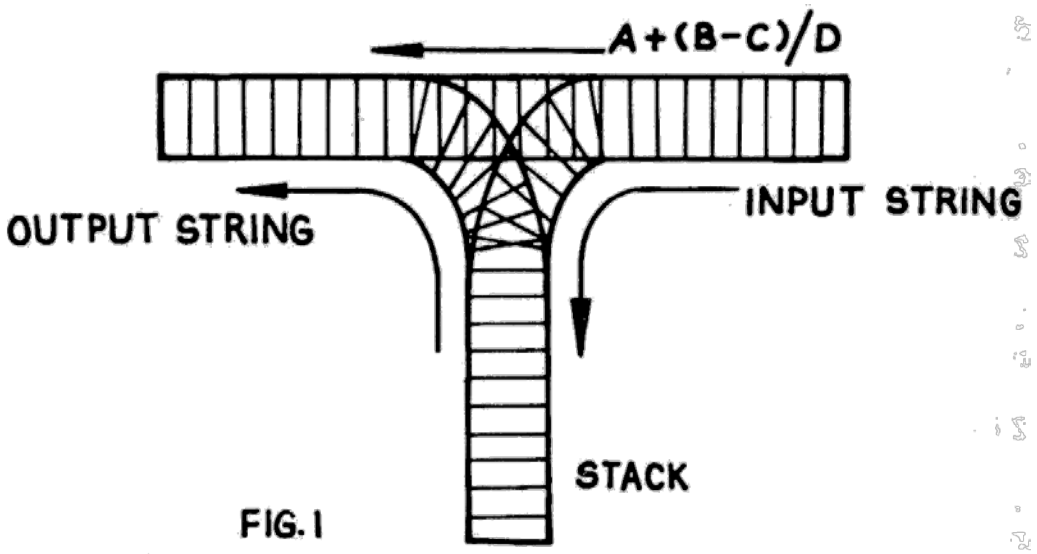
Transformation eines arithmetischen Ausdrucks in Postfix-Notation

Algorithmus (mit Verwendung eines Stacks)

- Der arithmetische Ausdruck sei in **Infix-Notation** (Standard-Notation) gegeben

  A + (B – C) / D

- **Shunting-yard algorithm** (Verschiebebahnhof-Algorithmus) realisiert die Transformation:

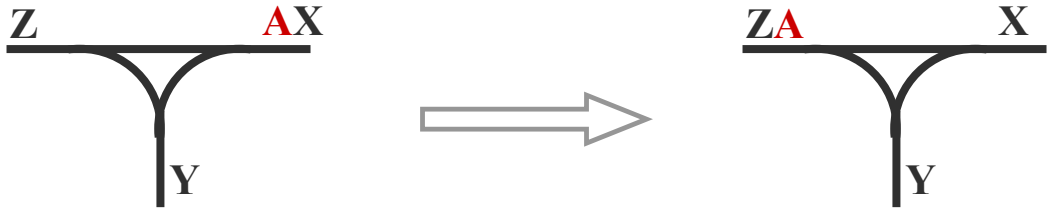


(aus M. Fogiel. The Computer Science Problem Solver. Research and Education Assoc., Piscataway, NJ, USA, 1991)

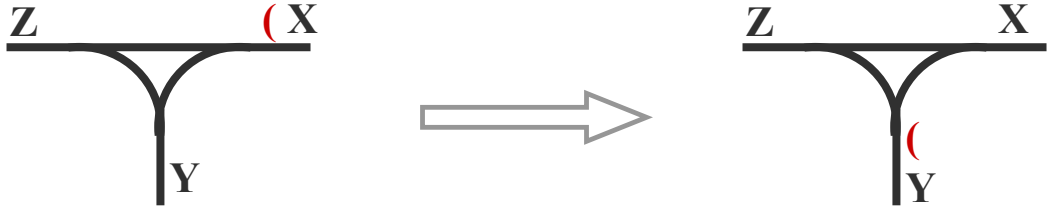
Shunting-yard Algorithmus

- Regeln des **shunting-yard** Algorithmus: Die Zeichenfolge des Ausdrucks wird von rechts nach links elementweise bewegt; dabei wird jedes Symbol individuell nach **5 Regeln** evaluiert

1. Behandlung von **Operanden**
   - Operanden (am Anfang der Eingabefolge) direkt in Ausgabefolge (transfer)



2. Behandlung von **öffnenden Klammern**
   - Linke (öffnende) Klammer wird auf den Stack gelegt (push)

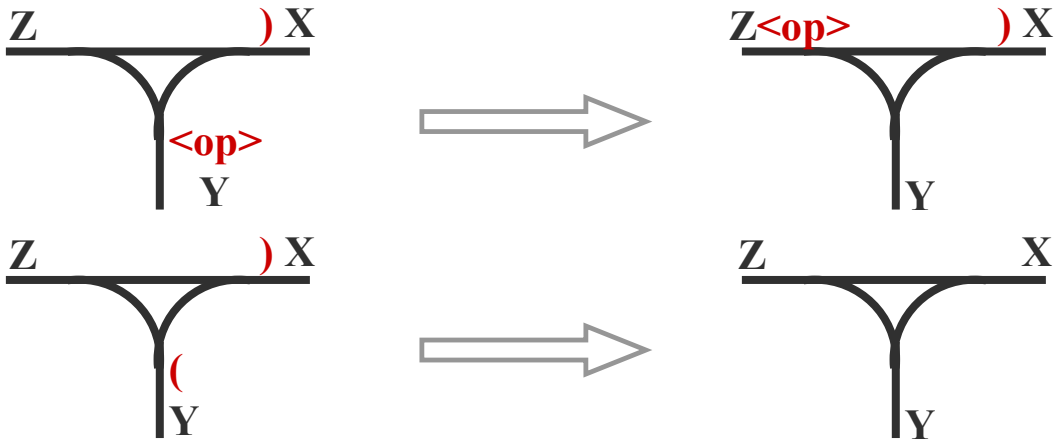


(aus M. Fogiel. The Computer Science Problem Solver. Research and Education Assoc., Piscataway, NJ, USA, 1991)

3. Behandlung von **schließenden Klammern**
   - Wenn in der Eingabefolge eine schließende Klammer erscheint, dann müssen alle Operatoren auf dem Stack s bis (ausschließlich) zur öffnenden Klammer in die Ausgabefolge in die Ausgabefolge gelesen und gelöscht werden:

```
while (char != '(') {
        out = s.top();
        s.pop();
}
```

   - Zuletzt werden die beiden (zueinander gehörenden) Klammern gelöscht



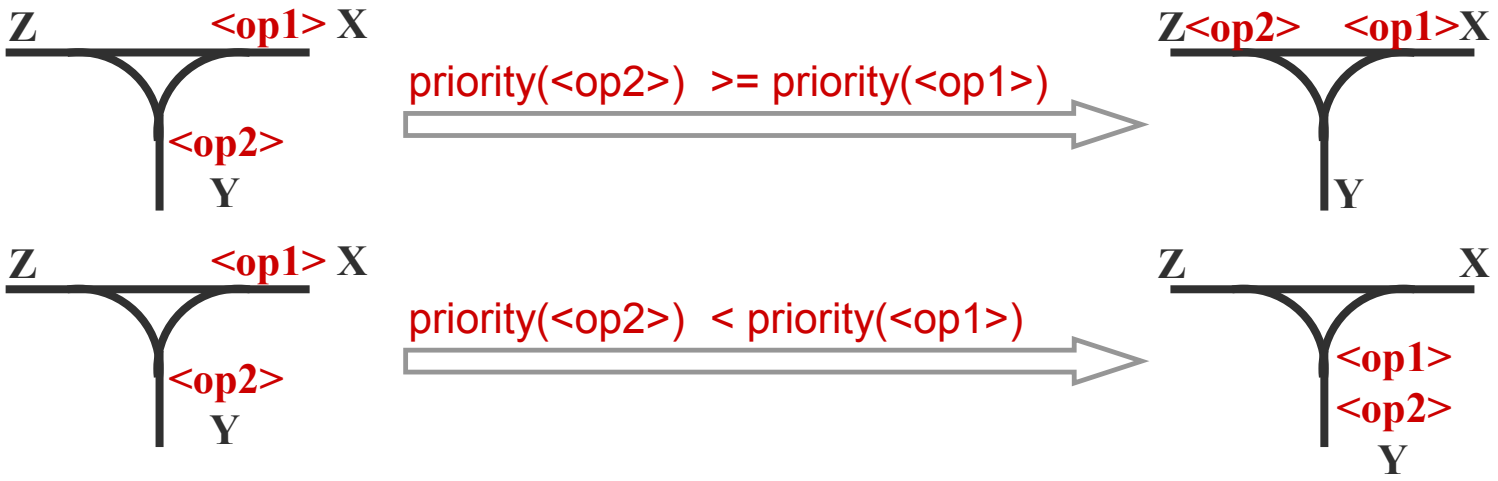
(aus M. Fogiel. The Computer Science Problem Solver. Research and Education Assoc., Piscataway, NJ, USA, 1991)

4. Behandlung von **Operatoren (+ – ∗ /)**

Wenn in der Eingabefolge ein Operator erscheint, dann werden die Elemente auf dem *Stack* solange in die Ausgabefolge gelesen, bis nächster Operator auf *Stack* von geringerer Priorität als der Operator in der Eingabe ist (Priorität: + – → ∗ /);

Wenn (a) *Stack* leer ist oder (b) eine Klammer enthält oder (c) das oberste *Stack*-Element von geringerer Priorität ist, dann wird der Operator in der Eingabe auf dem *Stack* abgelegt:

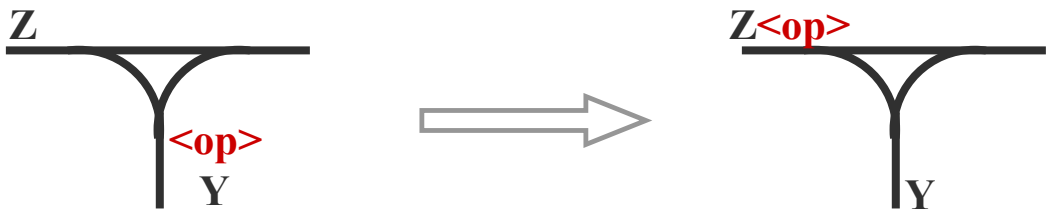
```
while (priority(<op>top) >= priority(<op>input)) {
        out = s.top();
        s.pop();
}
```



Shunting-yard Algorithmus

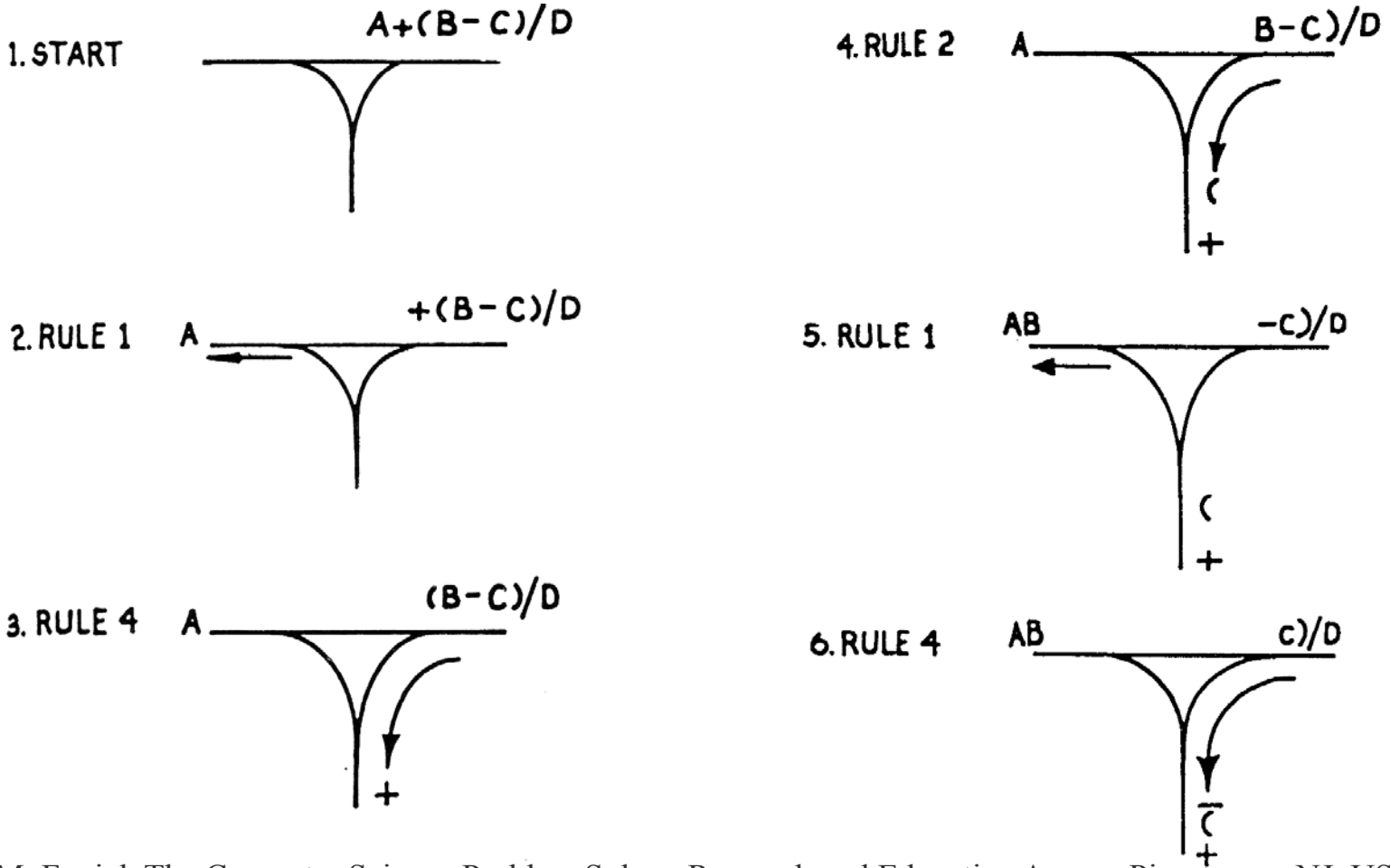
5. Behandlung von leerer Eingabe
   - Falls die Eingabefolge leer ist, dann werden alle Elemente auf dem Stack in die Ausgabefolge geschrieben

```
while (!s.isEmpty()) {
        out = s.top();
        s.pop();
}
```



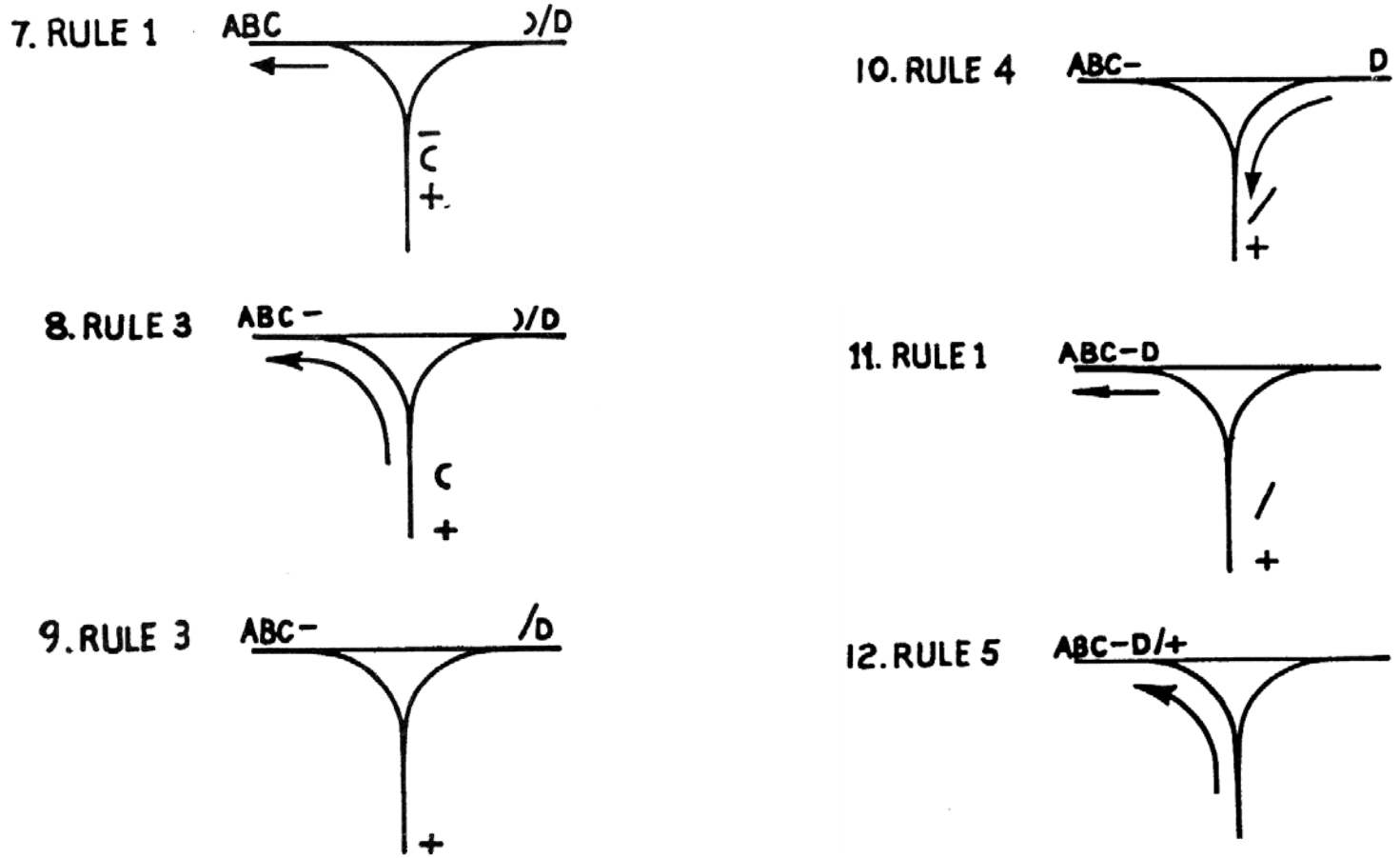
(aus M. Fogiel. The Computer Science Problem Solver. Research and Education Assoc., Piscataway, NJ, USA, 1991)

Shunting-yard Algorithmus – Beispiel



(aus M. Fogiel. The Computer Science Problem Solver. Research and Education Assoc., Piscataway, NJ, USA, 1991)

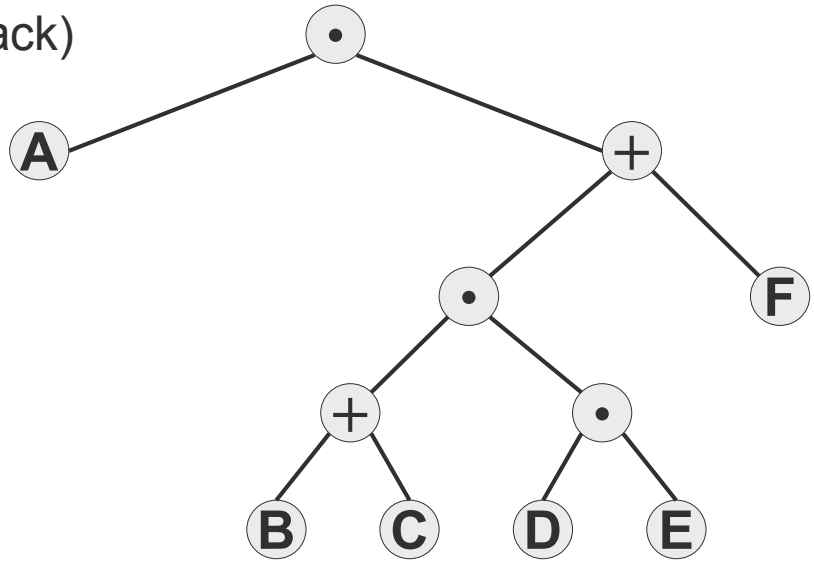
Shunting-yard Algorithmus – Beispiel



(aus M. Fogiel. The Computer Science Problem Solver. Research and Education Assoc., Piscataway, NJ, USA, 1991)

Generierung von Bäumen aus Postfix-Notation

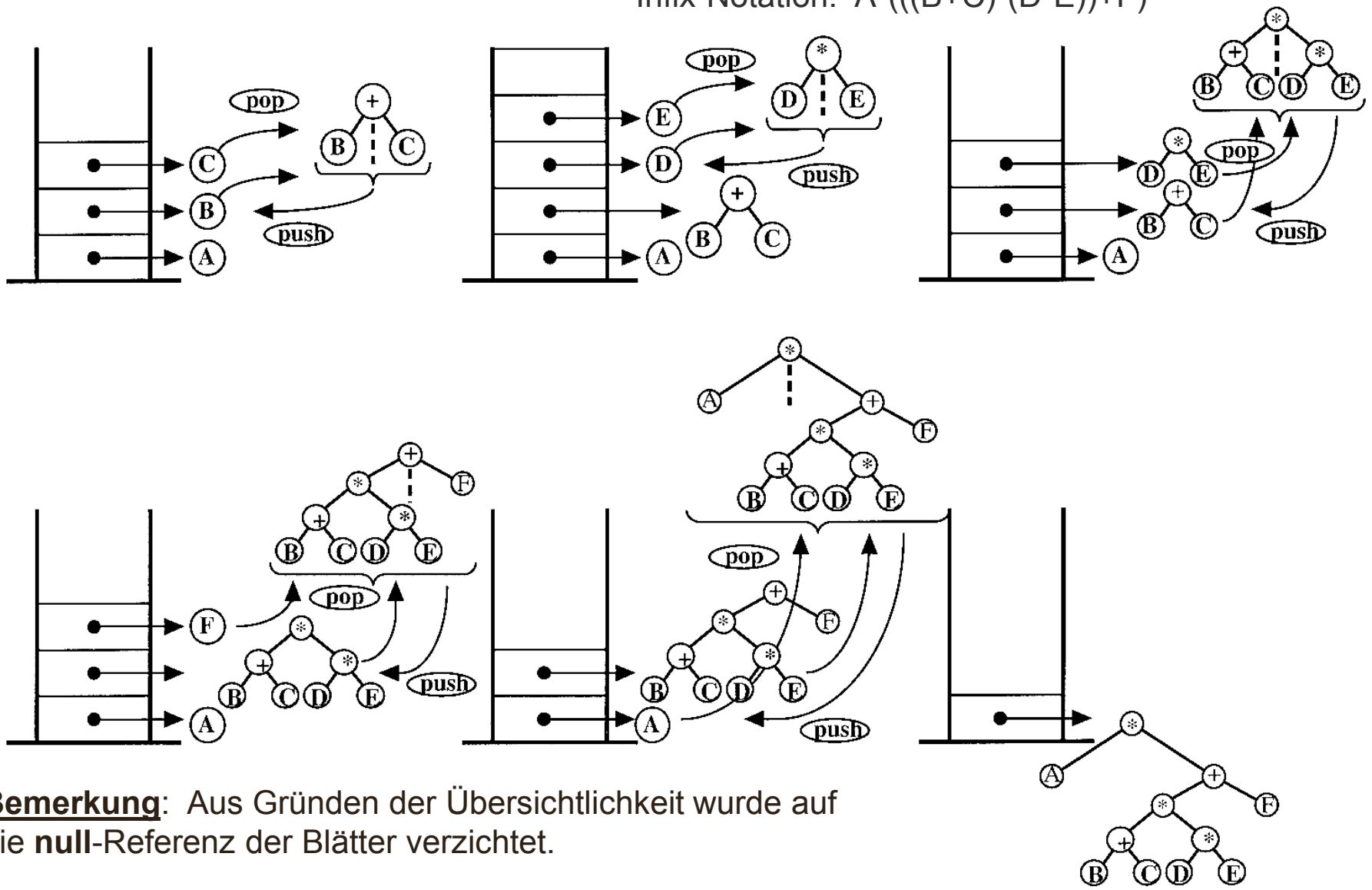
- Inverses Problem zur Transformation arithmetischer Ausdrücke in Postfix-Notation
- Infix- und Postfix-Notation:
  - Infix: A • (((B + C) • (D • E)) + F)
    (Repräsentation für uns)
  - Postfix: A B C + D E • • F + •
    (Rechner-Repräsentation mit Stack)
- Baumrepräsentation



Aufbau des Analysebaums für

A B C + D E * * F + *

Infix-Notation: A•(((B+C)•(D•E))+F)



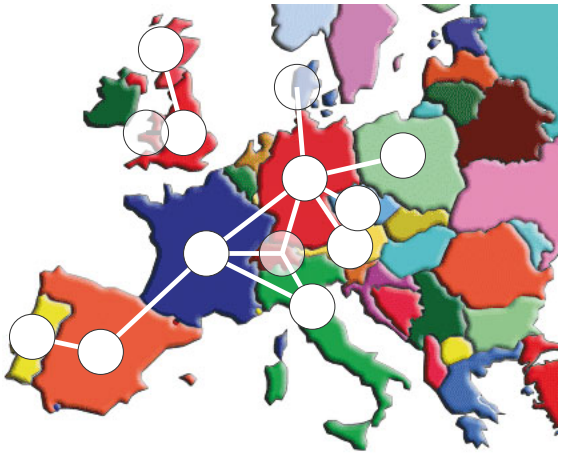
Bemerkung: Aus Gründen der Übersichtlichkeit wurde auf die **null**-Referenz der Blätter verzichtet.

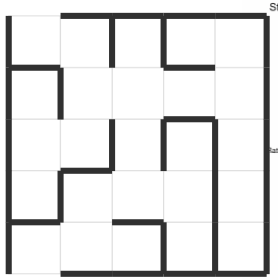
GRAPHEN

- Motivation und Einordnung
- Definitionen und Eigenschaften
- Repräsentation von Graphen
- Elementare Operationen auf Graphen
- Such-Algorithmen auf Graphen

Graphen – Einordnung

- Bei vielen Aufgabenstellungen und Lösungsverfahren (Algorithmen, Datenstrukturen) hat ein Element mehrere gleichwertige Nachbarbeziehungen (z.B. mehrere Vorgänger und mehrere Nachfolger).
- Beispiele sind …
  - Repräsentation von Nachbarschaftsbeziehungen
  - Straßen- oder Netz-Karten
  - Soziale Netzwerke

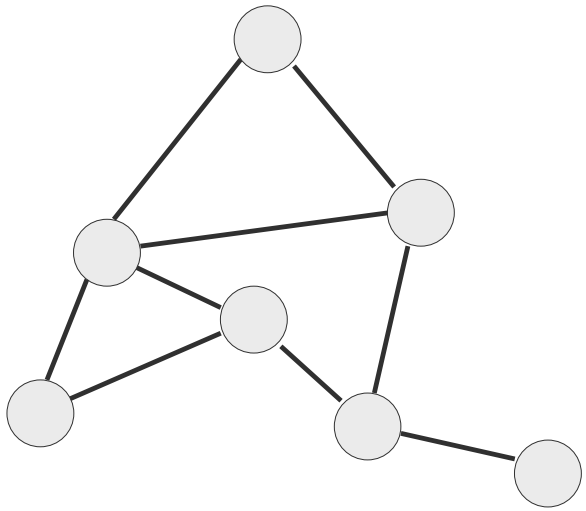




Graphen im Vergleich zu Bäumen

- Ein Graph kann als verallgemeinerte Baumstruktur aufgefasst werden:
  - In **einem Baum** hat ein Knoten **höchstens einen Vorgänger.**
  - In **einem Graph** kann ein Knoten **mehrere Vorgänger** haben.

- Darstellung



- Wie bei den Bäumen enthalten die **Knoten** die zu speichernden Elemente.
- Die **Kanten** repräsentieren Relationen zwischen den Knoten.
  - Kanten können **ungerichtet** oder **gerichtet** sein.

Definition ungerichteter Graphen

Ein ungerichteter Graph G ist ein Tupel

$$G \equiv G(V, E),$$

mit einer **Knoten**-Menge $\mathbf{V} = \{v_i \mid i \in \mathbf{IN}\}$ (*vertices*; auch **V**(**G**)), mit $0 \leq \text{card}(\mathbf{V}) < \infty$ und einer **Kanten**-Menge $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ (*edge set*; **E**(**G**)).

Die Kanten-Menge **E** ist definiert über eine irreflexive, symmetrische Relation R auf der Knoten-Menge **V**, mit

$$R = \{ (v_i, v_j), (v_j, v_i) \mid v_i, v_j \in \mathbf{V}, i \neq j \},$$

so dass

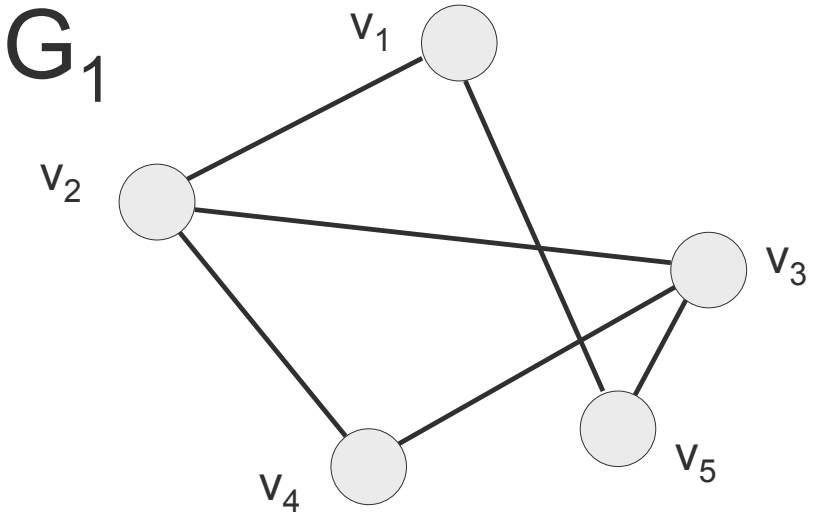
$$E = \{ ((v_i, v_j), (v_j, v_i)) \equiv e_{ij} \mid (v_i, v_j), (v_j, v_i) \in \mathbf{V} \times \mathbf{V} \}$$

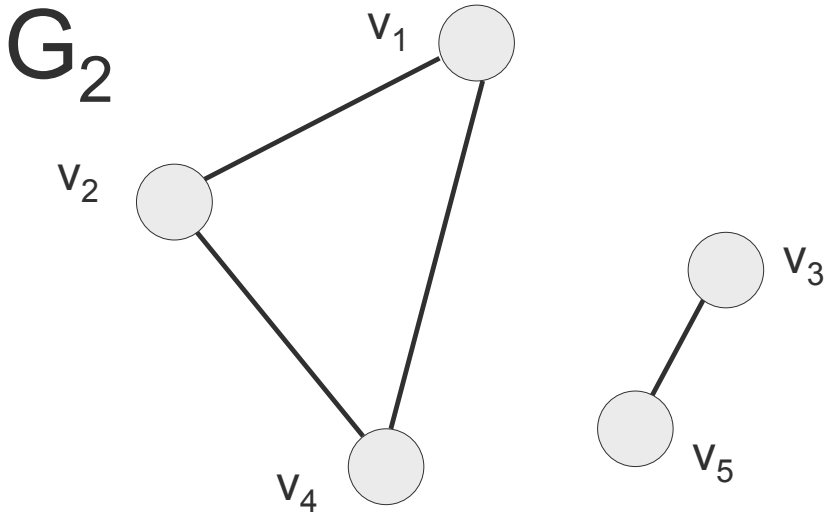
(kompaktere Notation: $((v_i, v_j), (v_j, v_i)) \equiv v_i v_j$).

Ein an eine Kante e_{ij} angrenzender Knoten v heißt „mit e **inzident**“.

(Nicht) Zusammenhängender Graph

- Von jedem Knoten v kann man zu jedem anderen Knoten über eine Folge von Kanten e gelangen, d.h. der Graph G zerfällt nicht in mehrere Teile.





Zusammenhängender Graph:

Jeder Knoten ist von jedem anderen Knoten auf direktem oder indirektem Weg erreichbar.

<u>Nicht</u> zusammenhängender Graph:

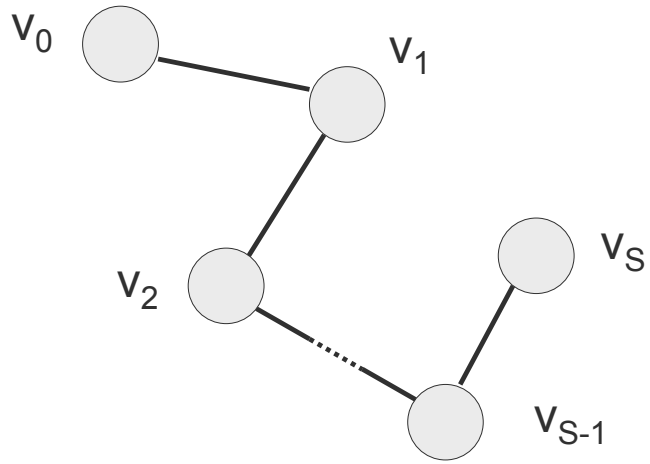
Beispielsweise kann Knoten v_3 nicht von Knoten v_2 erreicht werden, d.h. es existiert keine Kante, die die beiden Knotenmengen $\{v_1, v_2, v_4\}$ und $\{v_3, v_5\}$ verbindet.

Graphen - Eigenschaften

- **Kantenzug** K eines Graphen G: Folge $e^1, e^2, \ldots, e^S$ von Kanten, zu denen es Knoten $v_0, v_1, v_2, \ldots, v_S$ gibt, so dass

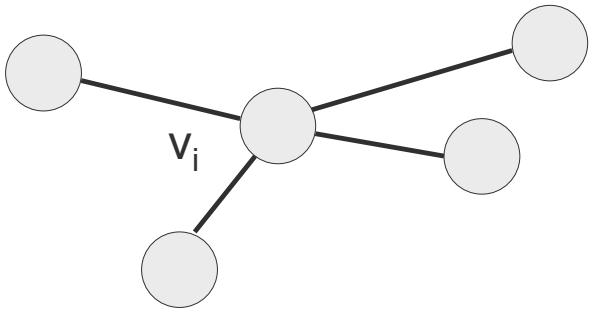
  e_0 die Knoten v_0 und v_1 verbindet,
  e_1 v_1 v_2
  …
  e_S v_{S-1} v_S verbindet

  Der Kantenzug verbindet die Knoten v_0 und v_S



- Grad eines Knotens: Anzahl der mit dem betrachteten Knoten v_i inzidenten Kanten e.

  $\deg_G(v_i) = \text{card}\{ v_iv_j\ (= e_{ij}) \mid v_iv_j \in \mathbf{E}(G)\ ,\ v_i, v_j \in \mathbf{V} \}$



$\deg_G(v_i) = 4$

Definition gerichteter Graphen

Ein gerichteter Graph G ist ein Tupel

$$G \equiv G^{\rightarrow}(V, E),$$

mit einer **Knoten**-Menge $\mathbf{V} = \{v_i \mid i \in \mathbf{IN}\}$ (*vertices*; auch **V**(**G**)), mit $0 \leq \text{card}(\mathbf{V}) < \infty$ und einer **Kanten**-Menge $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ (*edge set*; **E**(**G**)).

Die Kanten-Menge **E** ist definiert über eine nicht-symmetrische Relation R auf der Knoten-Menge **V**, mit

$$R = \{ (v_i, v_j) \mid v_i, v_j \in \mathbf{V} \},$$

so dass

$$E = \{ (v_i, v_j) \equiv e_{ij} \mid (v_i, v_j) \in \mathbf{V} \times \mathbf{V} \}$$

mit v_i: Anfangs-Knoten und v_j: End-Knoten.

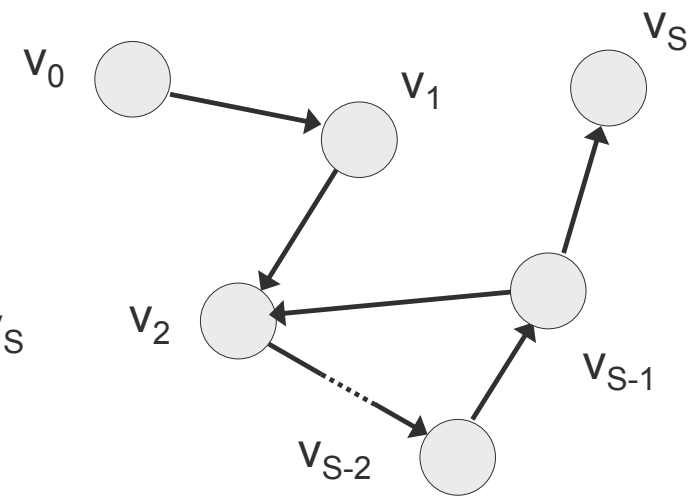
Anmerkung: Im Gegensatz zu den ungerichteten Graphen sind die Verbindungen (Relationen) zwischen den Knoten nicht symmetrisch.

Gerichtete Graphen – Eigenschaften

- **Grad eines Knotens** v_i in einem gerichteten Graphen $G^{\rightarrow}$
  - **Eingangsgrad** von v_i: Anzahl der gerichteten Kanten $e \in E$, mit End-Knoten v_i, $deg_G^+(v_i) = card\{v_j v_i \in E(G^{\rightarrow})\}$
  - **Ausgangsgrad** von v_i: Anzahl der gerichteten Kanten $e \in E$, mit Anfangs-Knoten v_i, $deg_G^-(v_i) = card\{v_i v_j \in E(G^{\rightarrow})\}$
- **Gerichteter Kantenzug** K eines Graphen $G^{\rightarrow}$: Folge $e^1, e^2, \ldots, e^S$ von gerichteten Kanten, zu denen es Knoten $v_0, v_1, v_2, \ldots, v_S$ gibt, so dass

  e_0 die Knoten v_0 und v_1,
  e_1 v_1 v_2
  …
  e_S v_{S-1} v_S verbindet

  Der Kantenzug verbindet die Knoten v_0 und v_S

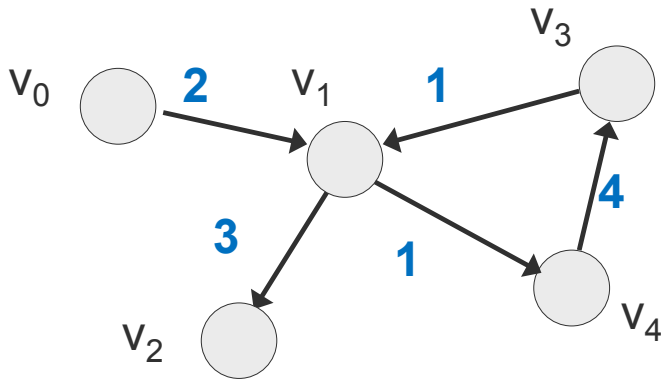


Anmerkung: In einem Kantenzug können Zyklen auftreten (hier: $v_2, \ldots, v_{S-2}, v_{S-1}, v_2$)

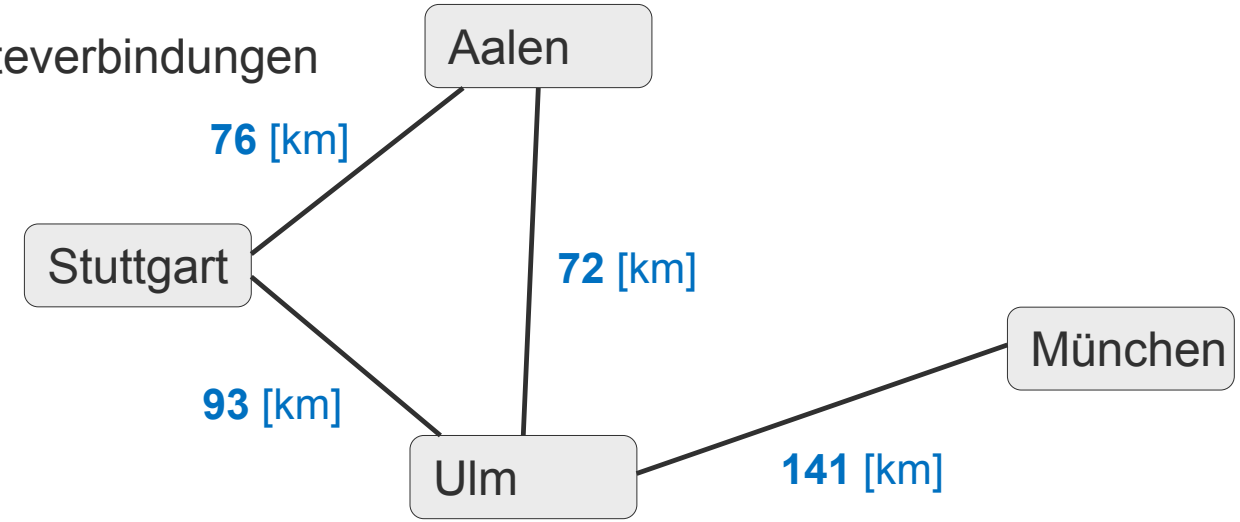
Gerichtete Graphen – Eigenschaften

- Ein **markierter** (**gewichteter**) **Graph** ist ein Tripel G = (V, E, w), wobei
  - das Paar (V, E) ein
    - ungerichteter Graph G oder
    - ein gerichteter Graph $G^{\rightarrow}$

    ist und
  - w eine Abbildung w: E $\rightarrow$ M definiert
    mit M = $\mathbf{IN}_0$ oder M = **IR**



- Bsp.: Städteverbindungen



Repräsentation von Graphen

Adjazenz-Matrizen

- **Adjazenz-Matrix** eines **ungerichteten Graphen** G(**V**, **E**) mit **V** = $\{v_i \mid i = 1, 2, \ldots\}$ und **E** = $\{e_{ij} \equiv v_i v_j \mid v_i, v_j \in \mathbf{V}\}$:
  - (n×n)-Matrix A = $[a_{ij}]$
  - mit symmetrischen Einträgen $a_{ij} = a_{ji}$ mit der Anzahl der Kanten zwischen Knoten v_i und v_j (bei einfachen Graphen **ohne Markierung** ist $a_{ij} \in \{0, 1\}$)
  - Bei irreflexiven (d.h. nicht-diagonalen) Graphen ist $a_{ii} = 0$

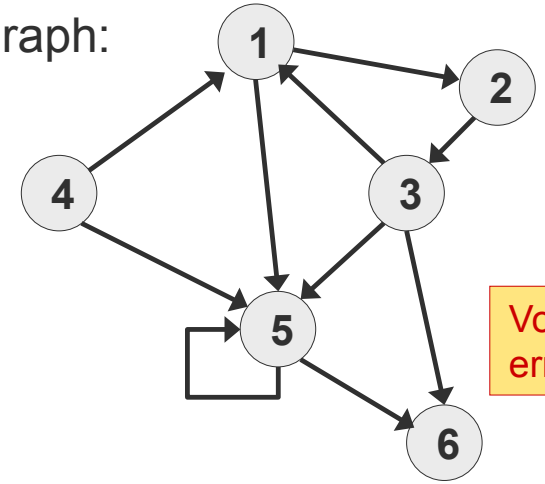
Repräsentation von Graphen (2)

Adjazenz-Matrizen

- **Adjazenz-Matrix** eines **gerichteten Graphen** $G^{\rightarrow}(\mathbf{V}, \mathbf{E})$ mit $\mathbf{V} = \{v_i \mid i = 1, 2, \ldots\}$ und $\mathbf{E} = \{e_{ij} \equiv (v_i, v_j) \mid v_i, v_j \in \mathbf{V}\}$:
  - $(n \times n)$-Matrix $A = [a_{ij}]$
  - mit nicht-diagonalen Einträgen a_{ij} und a_{ji}, wobei $a_{ij} \neq 0$ dann und nur dann, wenn eine gerichtete Kante von v_i nach v_j existiert, entsprechend für a_{ji}; A muss nicht symmetrisch sein, $a_{ij} \neq a_{ji}$
  - Diagonalelemente mit $a_{ii} = 0$ stehen für einen irreflexiven Graphen; $a_{ii} \neq 0$ für einen reflexiven Graph (Knoten mit Selbstreferenz)

Beispiel: Gerichteter Graph G→

Graph:



Adjazenz-Matrix:

Von einem Knoten erreichbare Nachbar-Knoten

i \ j →	1	2	3	4	5	6
1	0	**1**	0	0	**1**	0
2	0	0	**1**	0	0	0
3	**1**	0	0	0	**1**	**1**
4	**1**	0	0	0	**1**	0
5	0	0	0	0	**1**	**1**
6	0	0	0	0	0	0

```
int      noOfNodes = <Anzahl der Knoten>;
int[][] adjM = new int[noOfNodes][noOfNodes];
```

Einordnung und Bewertung:

- Für binäre ungewichtete Kanten kann die Matrix auch bool‘sch deklariert werden:

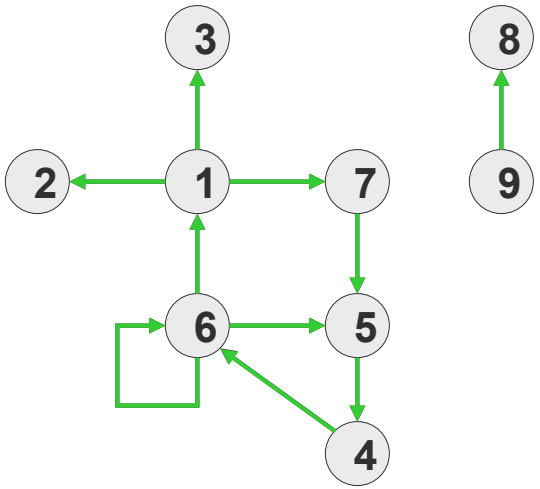
```
boolean[][] adjM = new boolean[noOfNodes][noOfNodes];
```

- Die Repräsentation ist für große Graphen mit wenigen (spärlichen) Kanten ineffizient, da viele nicht existierende Verbindungen aij = 0 repräsentiert werden.

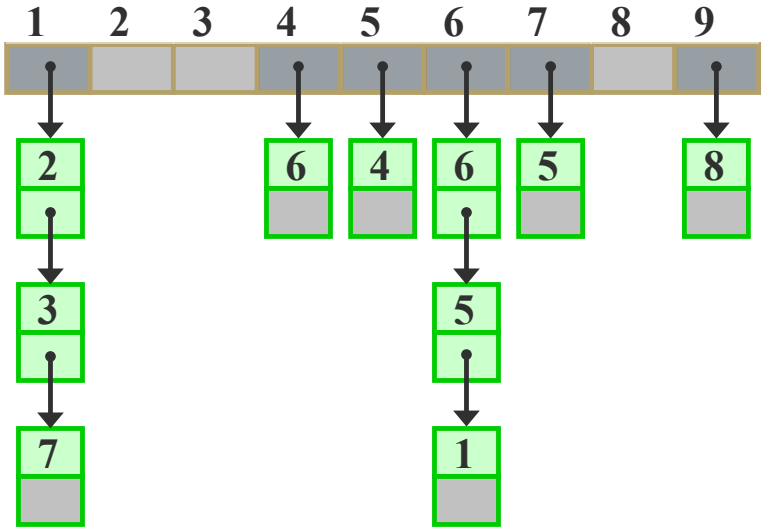
Adjazenz-Listen - Beispiel

- Ein Graph G(**V**, **E**) wird
  - durch ein **1-dimensionales *Array* (Knotenfeld)** der Länge card{**V**} definiert, dessen Elemente Zeiger auf Listen mit den benachbarten Knoten enthalten; die Menge an Knoten kann ebenso in einer Liste repräsentiert werden.
  - Für jeden Knoten v_i (Listeneintrag mit Index i) wird eine **lineare (verkettete) Liste** repräsentiert mit den (a) von diesem Knoten ausgehenden Kanten (gerichtete Graphen) oder (b) mit diesem Knoten verbundenen Knoten (ungerichtete Graphen).

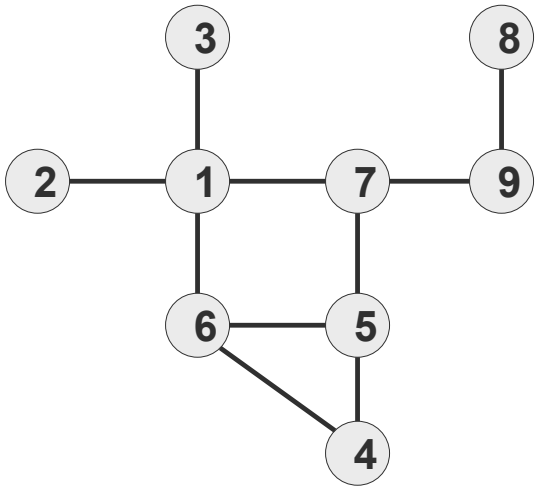
- Beispiel für gerichteten Graph G→

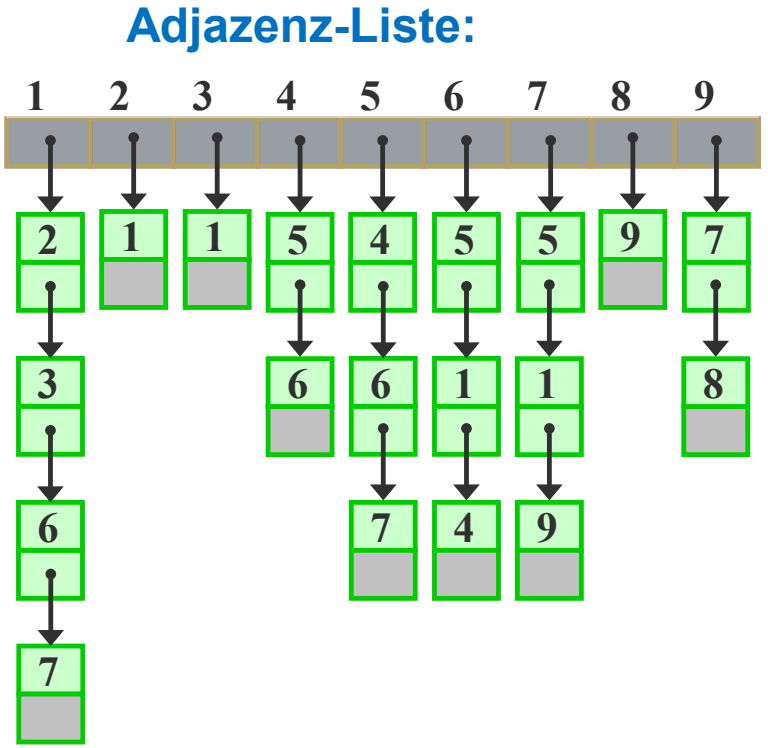


Adjazenz-Liste:



Adjazenz-Listen – Beispiel ungerichteter Graph:



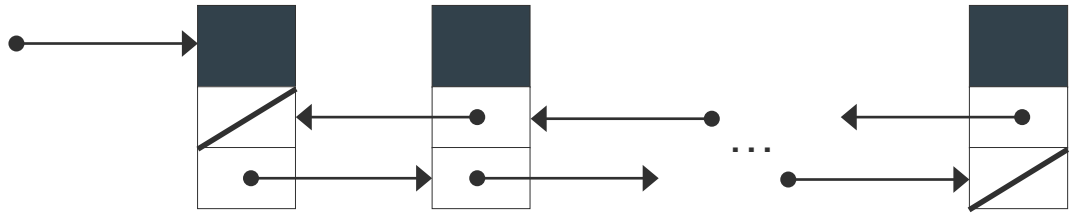


Eigenschaften und Aufwand:

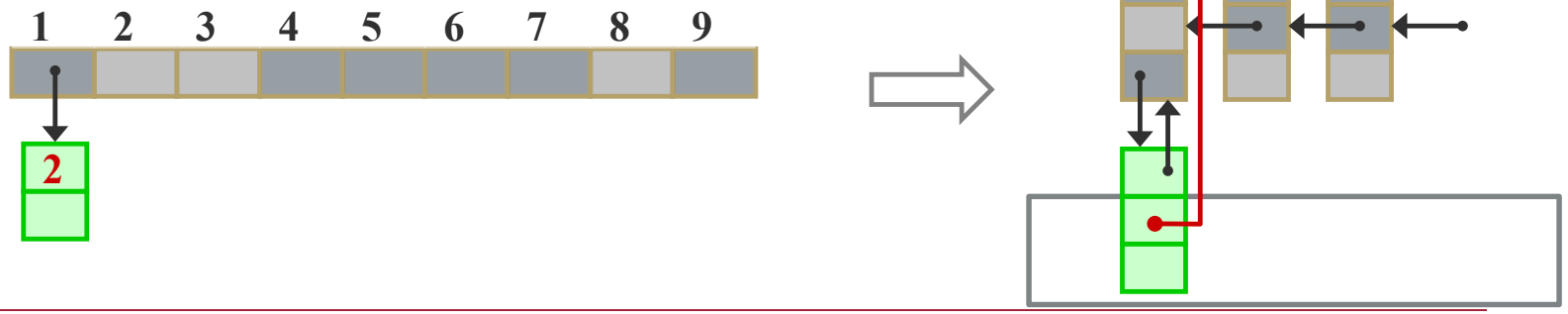
- Bei **markierten** (gewichteten) **Graphen**, in denen z.B. w: **E** $\rightarrow$ $\mathbf{IN}_0$, werden die Gewichte w in den Knoten der Listen gespeichert.
- Speicheraufwand für **gerichtete Graphen**: Summe der Länge aller Adjazenz-Listen zu den Knoten V ist card{**E**}
- Speicheraufwand für **ungerichtete Graphen**: Summe der Länge aller Adjazenz-Listen ist 2 • card{**E**}
- Speicheraufwand insgesamt: O(|**V**| • |**E**|)

Graph mit doppelt-verketteter Kanten-Listen

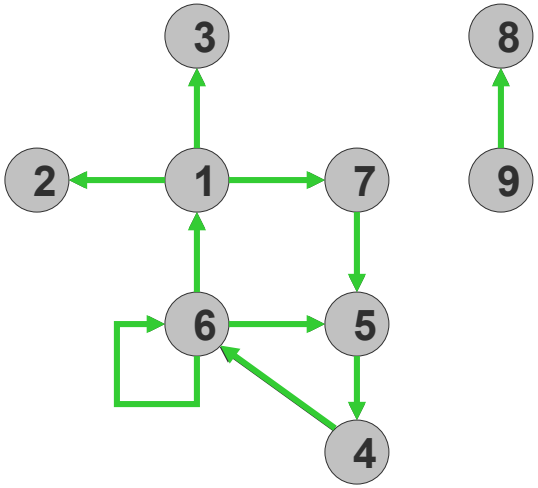
- Die Struktur entspricht im Wesentlichen den Adjazenz-Listen.
- Erweitertes Konzept:
  - Verwendung **doppelt verketteter Knoten-Listen** anstatt eines eindimensionalen Arrays für die Knoten (hier: Darstellung der Repräsentation ohne Wächter-Knoten).



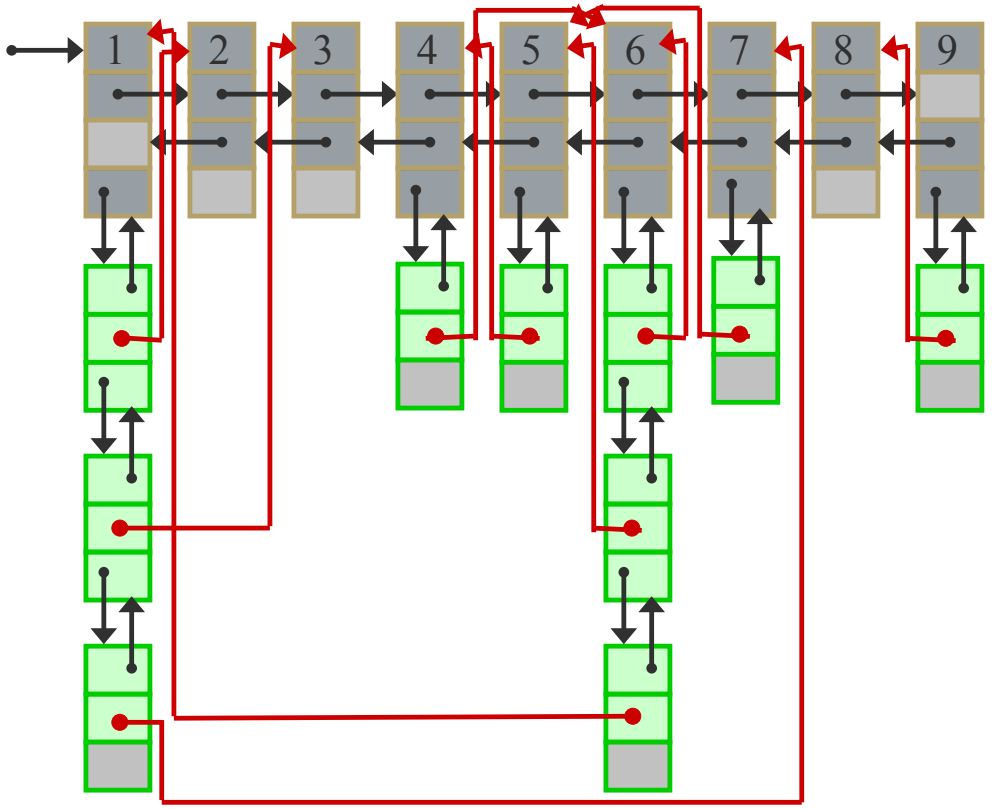
  - Die Adjazenz-Listen sind doppelt verkettet.
  - In den Adjazenz-Listen werden anstatt der Knoten selbst Zeiger auf die Knoten-Listen gespeichert



Beispiel: Gerichteter Graph $G^{\rightarrow}$



Doppelt verkettete Kanten-Liste:



- **Bewertung**: Einfügen und Entfernen von Knoten ist – im Vergleich zu den Adjazenz-Listen – relativ einfach

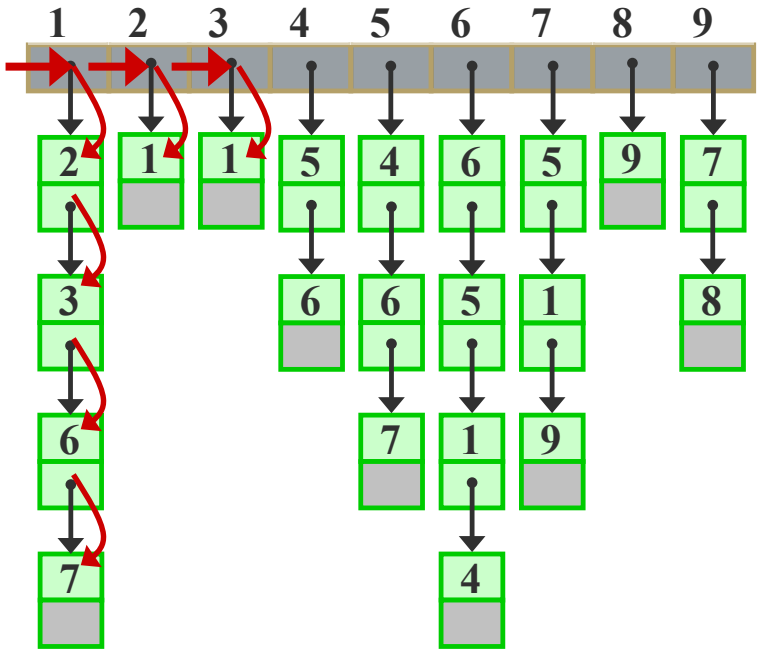
Elementare Operationen auf Graphen

Suchen eines Elements

- Bei der Suche nach Elementen (Knoten) in einem Graph müssen alle Elemente des Graphen G betrachtet werden (eine Ordnungsrelation, wie bei Bäumen, ist hier nicht definiert).
  - Eine **Suche** kann sowohl die **Elemente** (items) oder Eigenschaften in Form der **Verbindungen** (edges) betreffen.
- Der Algorithmus muss die Datenstruktur ausnutzen, in der der Graph definiert wurde:

Adjazenz-Matrix: Elementweises Durchsuchen der einzelnen Spalten eines zweidimensionalen Arrays, int[][] adjM;

Adjazenz-Liste: Das Feld mit den Knoten, int[] nodes; wird elementweise durchlaufen, für jeden Knoten werden jeweils die Elemente der anhängenden Knoten-Liste (sequenziell) durchsucht.



Einfügen eines Elements

Repräsentation mit **Adjazenz-Matrix**

- Eingabe bei **ungerichteten** Graphen: Neuer Knoten (mit Wert, item) besitzt eine Menge von Nachbarschafts-Knoten.
- Eingabe bei **gerichteten** Graphen: Neuer Knoten (mit Wert) besitzt eine Menge von Nachfolge-Knoten und eine Menge von Vorgänger-Knoten.
- Anhängen eines Elements als **neue Zeile** in der Adjazenz-Matrix; für jede Zeile muss auch ein **neues Spaltenelement** angehängt werden

Kommentar:

- Aus der (n×n)-Matrix wird eine (n+1)×(n+1)-Matrix generiert; die bisherigen Einträge der n Zeilen und Spalten müssen übertragen (kopiert) werden.
- Eintragen der (neuen) Verbindungen:
  - Bei **ungerichteten** Graphen für jeden Knoten x ∈ {Nachbarschafts-Knoten}: `adjMnew[n+1][x] = 1; adjMnew[x][n+1] = 1;`
  - Bei **gerichteten** Graphen:
  - Für jeden Knoten x ∈ {Nachfolge-Knoten-Knoten}: `adjMnew[n+1][x] = 1`
  - Für jeden Knoten x ∈ {Vorgänger-Knoten}: `adjMnew[x][n+1] = 1`

Repräsentation mit Adjazenz-Listen

mit Adjazenz-Listen

- Anhängen eines Elements an das 1-dimensionale Array der Knoten
- Aufbau einer Liste mit Elementen für die Nachbarschafts-Knoten
- Anhängen von Listenelementen an die jeweiligen Adjazenz-Listen der Knoten, die in der Liste der Nachbarschafts-Knoten enthalten sind (entsprechende Verfahrensweise für Eingangs-Knoten bei gerichteten Graphen)

- **Bewertung**: Einfügen von Knoten ist einfacher als bei Adjazenz-Matrizen, da weitestgehend die Eigenschaften von Listen zum Einfügen neuer Elemente ausgenutzt werden können.

mit doppelt verketteten Listen

- Gleiche Vorgehensweise, **wie bei den Adjazenz-Listen** – Vorteil ist, dass auch die Referenz-Knoten als Liste repräsentiert werden.
- **Bewertung**: Das Einfügen von Knoten wird weiter vereinfacht.

Demo: `Vertex.java, Graph.java`

- Repräsentation der **Knoten**

```
public class Vertex {
    public enum Color {
        WHITE, GRAY, BLACK
    }

    Color  color;
    int    distance;
    Vertex parent;

    List edges;


    public Vertex() {  // Konstruktor
        color    = Color.WHITE;
        distance = -1;
        edges    = new List();
    }

    public void connectTo(Vertex vertex) {
        if (edges.searchElemIter(vertex) == null)
            edges.insertElemLast(vertex);
    }
}  // end class Vertex
```

Repräsentation eines Graphen (auf der Objektebene)

```
public class Graph {
    private List vertices;

    public Graph() {
        vertices = new List();
    }

    public Graph(int noOfVertices) {
        this();
        for (int i = 0; i < noOfVertices; i++)
            addVertex();
    }

    public Graph(int noOfVertices, boolean fullyConnected) {
        this(noOfVertices);
        for (int i = 0; i < noOfVertices; i++)
            for (int j = i; j < noOfVertices; j++)
                connectBidirectional(i, j);
    }

    public void addVertex() { ... }
    public void deleteVertex(int i) { ... }
    public Vertex getVertex(int i) { ... }
    public int getVertexId(Vertex v) { ... }
    public void connect(int i, int j) { ... }
    public void connectBidirectional(int i, int j) { ... }
    public int getNoOfVertices() { ... }
    public void depthFirstSearch(int rootIdx) { ... }
    private void depthFirstSearch(Vertex v) { ... }
    public void breadthFirstSearch(int rootIdx) { ... }
    public void printGraph() { ... }

}  // end class Graph
```

Repräsentation von Listen-Elementen

```
public class ListElem {
    Vertex   item;  // Inhalt eines Listen-Elements: Knoten eines Graphen
    ListElem next;

    public ListElem() {
    }

    public ListElem(Vertex newItem, ListElem next) {
        this.item = newItem;
        this.next = next;
    }
}  // end class ListElem
```

Repräsentation von Listen-Objekten

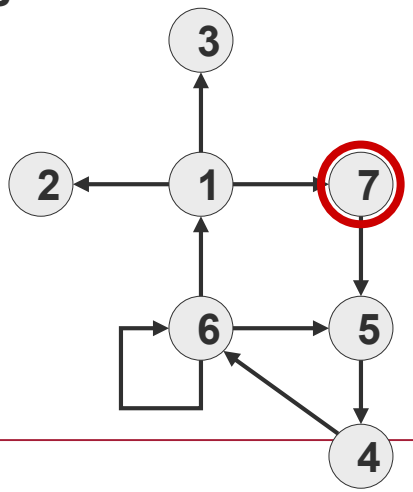
```
public class List {
    ListElem head,
             foot;

    List() {
        this.head = null;
        this.foot = null;
    }

    List(Vertex item) {
        this.head = new ListElem(item, null);
        this.foot = head;
    }

    void insertElemFirst(Vertex item) { ... }
    void insertElemLast(Vertex item) { ... }
    public void appendList(List list) { ... }
    void deleteListItem(int pos) { ... }
    ListElem searchElem(Vertex value) { ... }
    private static ListElem searchElem(Vertex value, ListElem elem) { ... }
    ListElem searchElemIter(Vertex value) { ... }
    void printList() { ... }
    private static void printList(ListElem elem) { ... }
    void printListReverse() { ... }
    private static void printListReverse(ListElem elem) { ... }
    ListElem getListElem(int index) { ... }
    Vertex getListItem(int pos) { ... }
    void insertListItem(int pos, ListElem elem) { ... }
    int sizeList() { ... }
    Vertex getLastElem() { ... }
    Vertex removeLastElem() { ... }
    int indexOfElem(Vertex v) { ... }
}  //  end class List
```

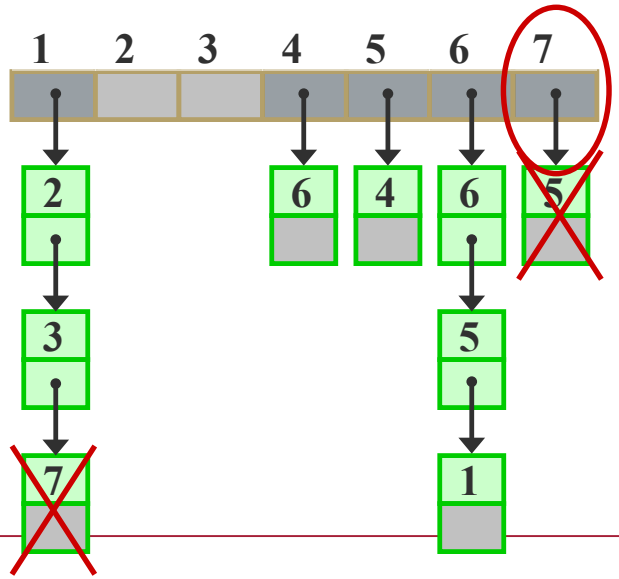
Löschen eines Elements

Allgemeine Vorgehensweise (hier für Adjazenz-Listen)

- Zunächst muss das Element mit der gegebenen Eigenschaft (item) **gesucht** werden; das Element (Knoten) wird markiert (z.B. Zeiger, Index des Knotens in der Knoten-Liste)
- **Löschen der Verbindungen** des markierten Knotens in der Adjazenz-Matrix bzw. in der Adjazenz-Liste / doppelt verketteten Kanten-Liste
- Beispiel – **gerichteter Graph** $G^{\rightarrow}$ und **Adjazenz-Liste** als Repräsentation

- **Aufgabe**: Lösche Knoten 7





Löschen eines Elements - Implementation

- Code-Fragment

```
public void deleteVertex(int idx) {
    Vertex ver2remove = vertices.getListElem(idx);

    vertices.deleteListElem(idx);
    for (ListElem ver = vertices.head; ver != null; ver = ver.next)
        ver.item.edges.deleteListElem(ver2remove);
}  // end deleteVertex
```

Kommentar zur Erläuterung:

- Hier wird zunächst der zu löschende Knoten aus der Liste aller Knoten entfernt.
- Dazu wird der Knoten durchlaufen und in jeder der zu den Knoten gehörenden Kanten-Liste eine eventuell vorhandene Referenz auf den zu löschenden Knoten entfernt.

Such-Algorithmen auf Graphen – Übersicht

- Bei vielen algorithmischen Problemen, die auf Graph-Strukturen als Repräsentation zurückgreifen, wird der Graph durchsucht und es werden dabei bestimmte qualitative oder quantitative Eigenschaften berechnet.
- Dabei muss systematisch und vollständig jeder Knoten und/oder jede Kante in dem Graph „besucht“ werden.

Aufgabenbeispiele:

- Feststellung, ob ein gegebener Graph **vollständig verbunden** ist und/oder ob er Zyklen enthält.
- **Kürzeste-Wege-Problem**: Suchen der kürzesten Wegstrecke ausgehend von einem Knoten S bis zu einem Knoten Z (dabei können die Kanten gewichtet sein).
- Bestimmung **minimal aufspannender Bäume**: Bestimmung der Wege in einem Netzwerk von Knoten, die eine kostengünstige Verbindung zwischen Knoten ergibt.
- Reise-/Optimierungsprobleme (Problem des Handlungsreisenden, **travelling salesman problem**): Berechnung einer Rundreise über alle Knoten, bei der die Knoten jeweils nur einmal besucht werden und dabei die Gesamtkosten der Wegstrecke minimal werden.

Tiefensuche (depth-first search) in einem Graphen

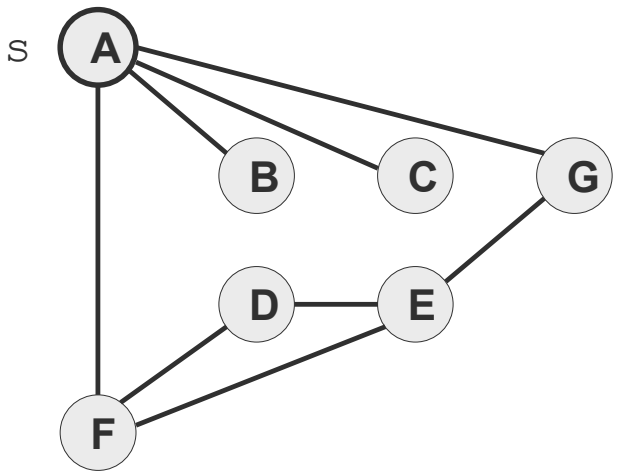
- Hier werden exemplarisch Suchprobleme auf Graphen betrachtet, bei denen die Knoten in einer definierten Art und Weise besucht werden und dabei die schon besuchten Knoten markiert werden.
- Es werden zwei grundsätzliche Strategien unterschieden (vgl. Binärbaum):
  - Tiefensuche
  - Breitensuche

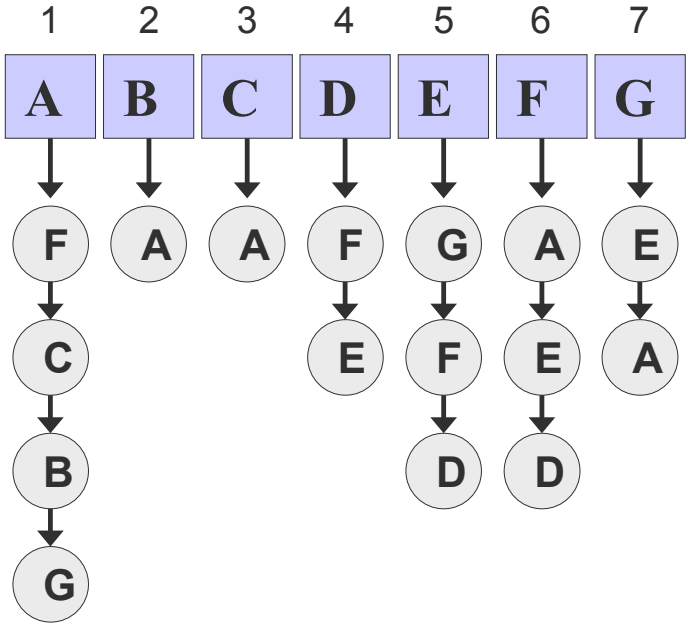
(Rekursiver) Algorithmus der Tiefensuche:

- Von einem (Start-) Knoten `s ∈ G` geht man entlang jeweils einer der bestehenden Kanten und markiert (*coloring*, `visit(•)`) den besuchten Knoten `v`.
- Trifft man auf bereits besuchte Knoten, wird dieser Weg nicht weiter exploriert und statt dessen eine alternative Kante `e ∈ nb(v)` zu einem anderen Nachbar verfolgt.
- Sind alle von einem aktuellen Knoten ausgehenden Kanten abgelaufen, so kehrt der Algorithmus zu dem Knoten zurück, von dem aus der aktuelle Knoten erreicht wurde; verfahre weiter rekursiv.

Beispiel: Tiefensuche - ungerichteter Graph

- Für einen Graph G(V, E) gibt es einen ausgezeichneten Knoten s $\in$ G.
- Graph und seine Repräsentation (Adjazenz-Liste):

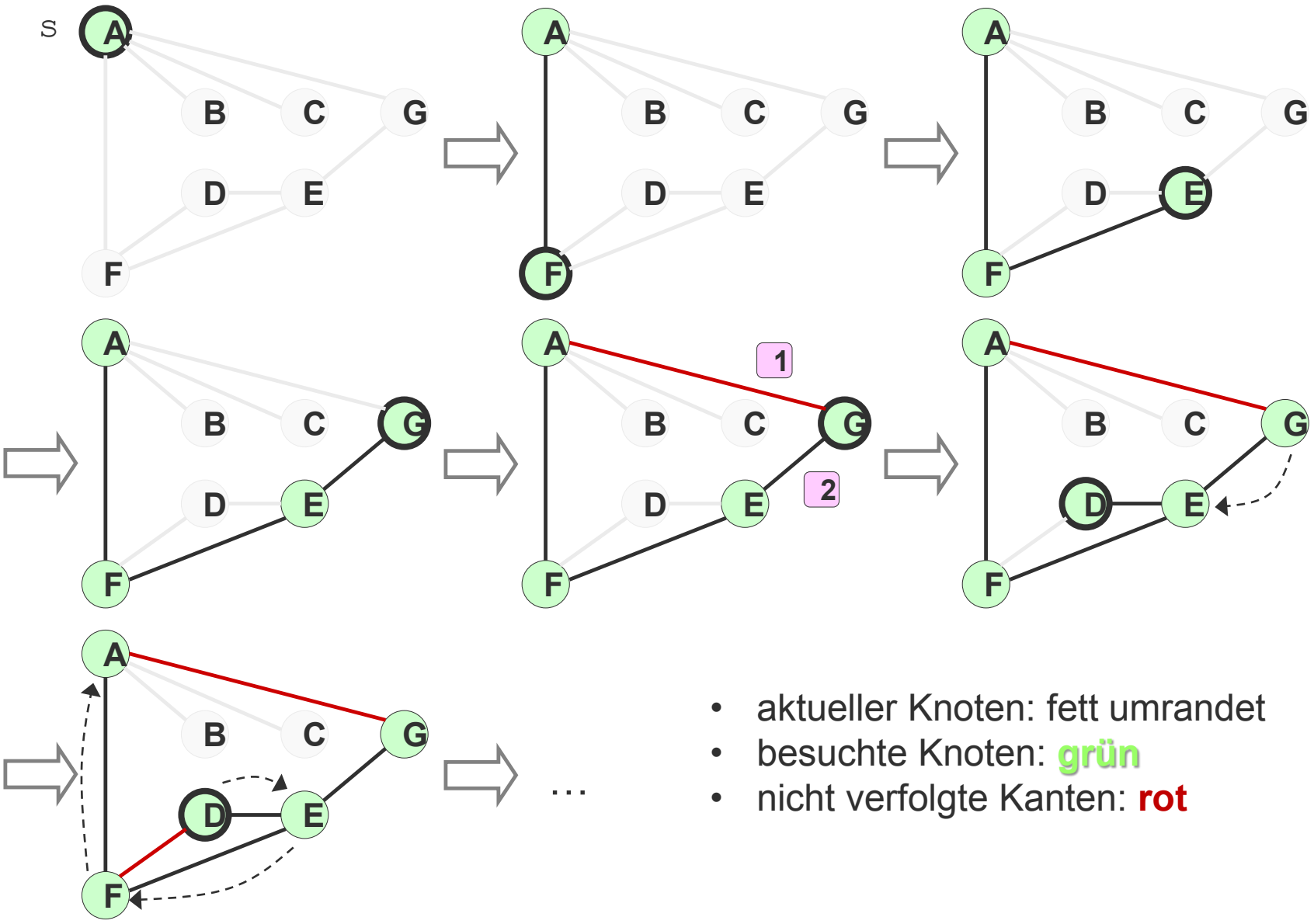




- Die Strategie der **Tiefensuche** läuft von einem Knoten ausgehend (beginnend mit s) entlang einer von ihm ausgehenden Kanten $e_1 = v_{aktuell} v_{NB}$ und prüft, ob der nächste Knoten bereits besucht wurde; wenn nicht, dann laufe eine Verbindung e_2 zum nächsten Knoten, usw.

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

(Tiefen-) Exploration der Knoten

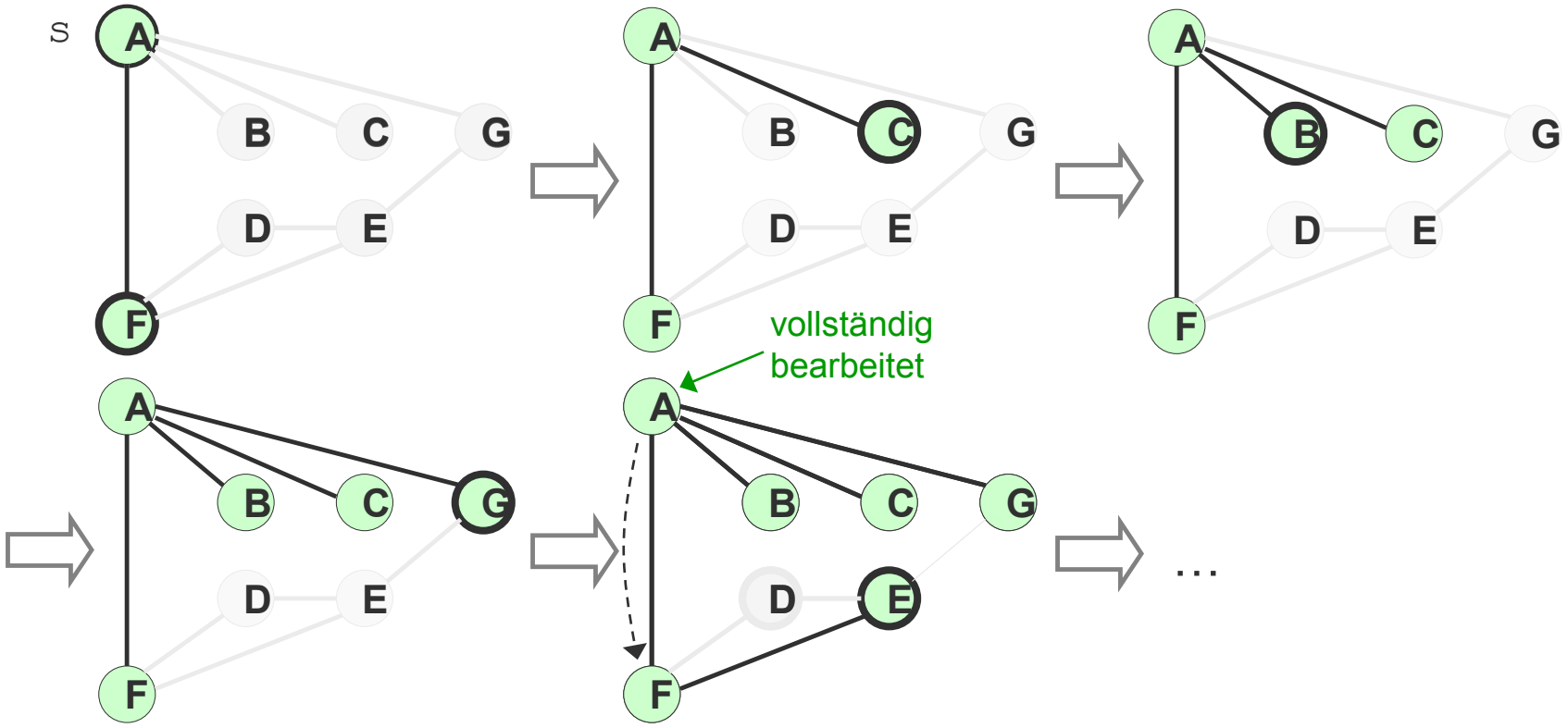


- aktueller Knoten: fett umrandet
- besuchte Knoten: grün
- nicht verfolgte Kanten: **rot**

```
    public void depthFirstSearch(int rootIdx) {
        for (int i = 1; i <= vertices.sizeList(); i++) {
            Vertex v = vertices.getListItem(i);

            v.color    = Vertex.Color.WHITE;
            v.distance = -1;
            v.parent   = null;
        }

        Vertex rootVertex = getVertex(rootIdx);

        rootVertex.distance = 0;
        rootVertex.parent   = rootVertex;

        depthFirstSearch(rootVertex);
    }  // end depthFirstSearch


    private void depthFirstSearch(Vertex v) {
        v.color = Vertex.Color.BLACK;
        for (ListElem edge = v.edges.head; edge != null; edge = edge.next)
{
            Vertex dest = edge.item;

            if (dest.color == Vertex.Color.WHITE) {
                dest.parent   = v;
                dest.distance = v.distance + 1;
                depthFirstSearch(dest);
            }
        }
    }  // end depthFirstSearch
```

Breitensuche (breadth-first search) in einem Graphen

- Beginnend von einem Start-Knoten werden alle direkt verbundenen Knoten besucht, bevor zur nächst tieferen Ebene gegangen wird.
- Für denselben Beispiel-Graphen wird die Exploration in Breitensuche diskutiert (Adjazenzliste zu A: F – C – B – G)
- (Breiten-) Exploration der Knoten (Anfang wie bei Tiefensuche)



```
public void breadthFirstSearch(int rootIdx) {
    for (int i = 1; i <= vertices.sizeList(); i++) {
        Vertex v = vertices.getListItem(i);

        v.color    = Vertex.Color.WHITE;
        v.distance = -1;
        v.parent   = null;
    }

    Vertex rootVertex = getVertex(rootIdx);

    rootVertex.color    = Vertex.Color.GRAY;
    rootVertex.distance = 0;
    rootVertex.parent   = rootVertex;

    List queue = new List(rootVertex);

    while (queue.sizeList() > 0) {
        Vertex v = queue.removeLastElem();

        for (ListElem edge = v.edges.head; edge != null; edge = edge.next) {
            Vertex dest = edge.item;

            if (dest.color == Vertex.Color.WHITE) {
                dest.color    = Vertex.Color.GRAY;
                dest.distance = v.distance + 1;
                dest.parent   = v;
                queue.insertElemFirst(dest);
            }
        }
        v.color = Vertex.Color.BLACK;
    }
}  // end breadthFirstSearch
```

Zusammenfassung

Dynamische Datenstrukturen

- Dynamische Arrays
- Listen
  - einfach verkettet
  - doppelt verkettet
- Stapel und Schlangen
- Bäume
- Graphen