X. Dynamische Datenstrukturen

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

1. 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)

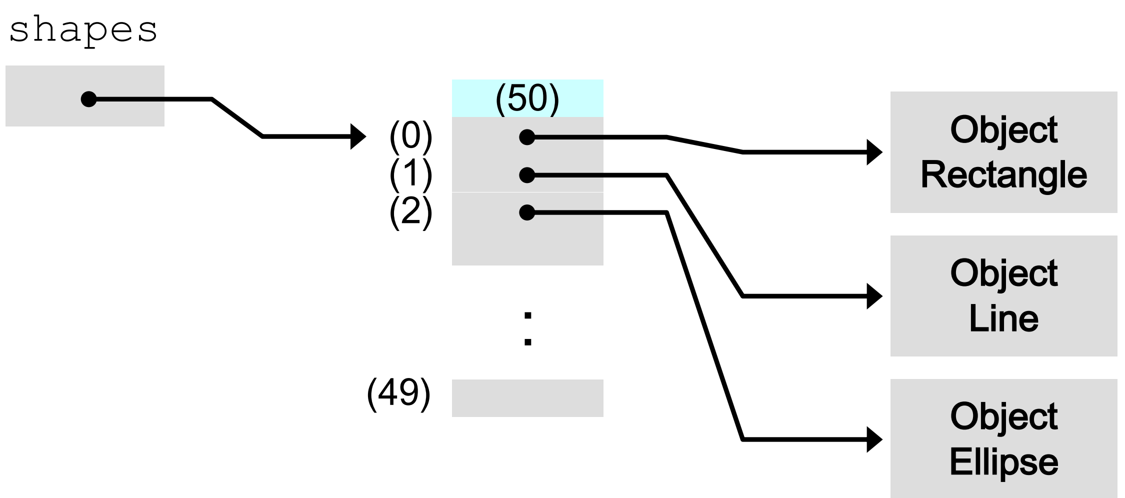
Dynamische Arrays

Partiell gefüllte Datenfelder

- Die Anzahl der Elemente eines *Arrays* ändert sich oft zur Laufzeit
- Verwaltung von Daten mit linearer Ordnung (vgl. Teil V)
  - 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
- Anwendungsbeispiele:
  - Ausgabe einer Zahlenfolge in umgekehrter Reihenfolge aus einem `int`-*Array*,
    `int[] numbers = new int[100];` (vgl. Teil V)
  - 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
```

- Darstellung:



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

<u>**Annahme**</u>: Die Klasse `Shape` enthält eine Methode `redraw(...)` zur Anzeige in einem Graphik-Kontext `g`;

Die Menge der instanziierten Formen kann dann zyklisch abgearbeitet werden:

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

Die maximale Anzahl graphischer Anzeigeobjekte darf 50 nicht überschreiten!

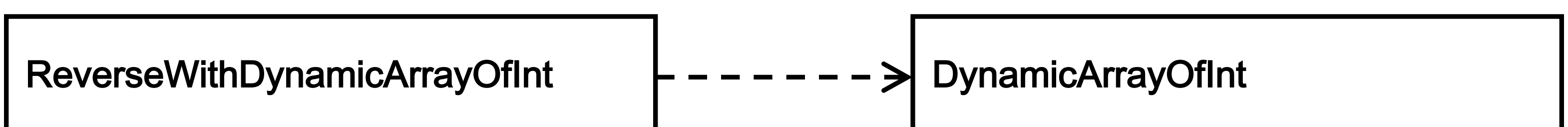
Dynamische Datenfelder

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

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 ein neues *Array* mit größerem Speicherplatz generiert und die Elemente des alten *Arrays* in das neue kopiert
- **Realisierung**: Eine beliebige Anzahl von Zahlen soll in umgekehrter Reihenfolge ausgegeben werden; hierfür werden zwei Klassen `DynamicArrayOfInt` und `ReverseWithDynamicArrayOfInt` definiert (UML-Notation)



- Implementierung in Java (Demo: ReverseWithDynamicArrayOfInt.java und DynamicArrayOfInt.java)
  - Basis-Klasse für die dynamischen *Arrays*

```
public class DynamicArrayOfInt {
    private int[] data;  // Attribute: ein dynamisches Array zum Speichern der Daten (verdeckt)

    public DynamicArrayOfInt() {  // Konstruktor
        data = new int[1];
    } // end constructor DynamicArrayOfInt

    /* -- getter-/setter-Methoden ... */
    public int getVal(int position) {  // lese int Wert (ohne exception)
        if (position >= data.length)
            return 0;
        else
            return data[position];
    } // end getVal

    public void putVal(int position, int value) {  // schreibe int Wert
        if (position >= data.length) {  // aktuelle Position ist ausserhalb
                                        // der aktuellen Groesse des Arrays
            int newSize = 2 * data.length;

            if (position >= newSize)
                newSize = 2 * position;          // neue Feldlaenge

            int[] newData = new int[newSize];    // generiere neues Feld
            System.arraycopy(data, 0, newData, 0, data.length);  // Daten kopieren
            data = newData;                      // Referenz zeigt auf neues Objekt

            /* -- nur zu Demonstrationszwecken ... */
            System.out.println("dynamisches Array: neue Groesse " + newSize);
        }
        data[position] = value;
    } //  end putVal
} //  end class DynamicArrayOfInt
```

- Hilfs-Klasse zur Verwendung der dynamischen Arrays

```
class ReverseWithDynamicArray {  // Hilfsklasse
    public static void main(String[] args) {
      DynamicArrayOfInt numbers;  // Speichern der Eingabezahlen
      int numCnt,  // Anzahl der im Array gespeicherten Zahlen
          num;     // eine Eingabezahl (durch Benutzer)

      numbers = new DynamicArrayOfInt();
      numCnt  = 0;

      TextIO.putln("Eingabe einer positiven Zahl (0: Ende) ");
      while (true) {  // lies Zahlen ein, ablegen im dynamischen Array
         TextIO.put("? ");
         num = TextIO.getlnInt();
         if (num <= 0)
            break;
         numbers.putVal(numCnt, num);  // Zahl in dynamisches Array
         numCnt++;
      }
      TextIO.putln("\nDie Zahlen in umgekehrter Reihenfolge:\n");
      for (int i = numCnt - 1; i >= 0; i--)
         TextIO.putln(numbers.getVal(i));  // drucke die i-te Zahl

   } //  end main
} //  end class ReverseWithDynamicArray
```

Lineare Listen

Einordnung und Motivation

Verwaltung von 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

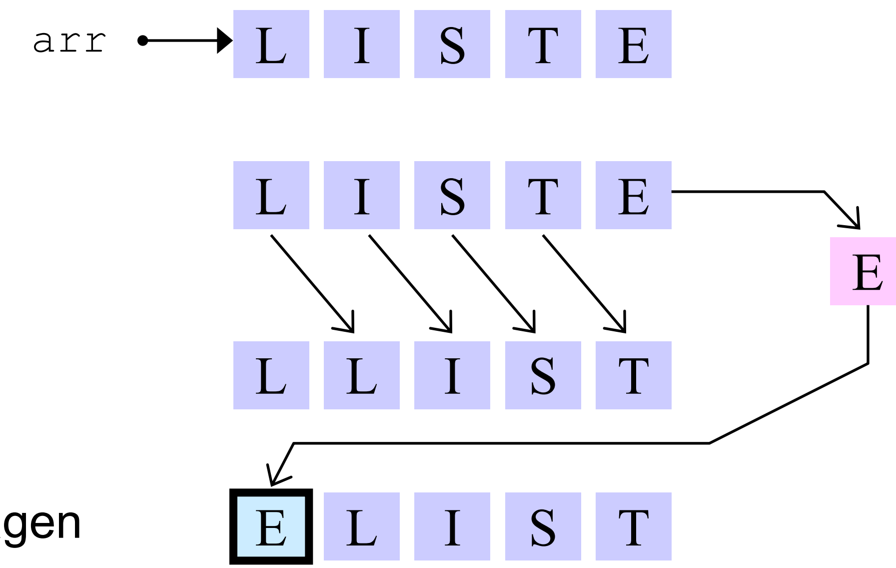
Bsp.: Das letzte Element eines Feldes (= E) soll an den Anfang verschoben werden

1. Start

2. Element kopieren und zwischenspeichern

3. Listenelemente verschieben
   (`arr.length-1`) × kopieren

4. Gespeichertes Element eintragen

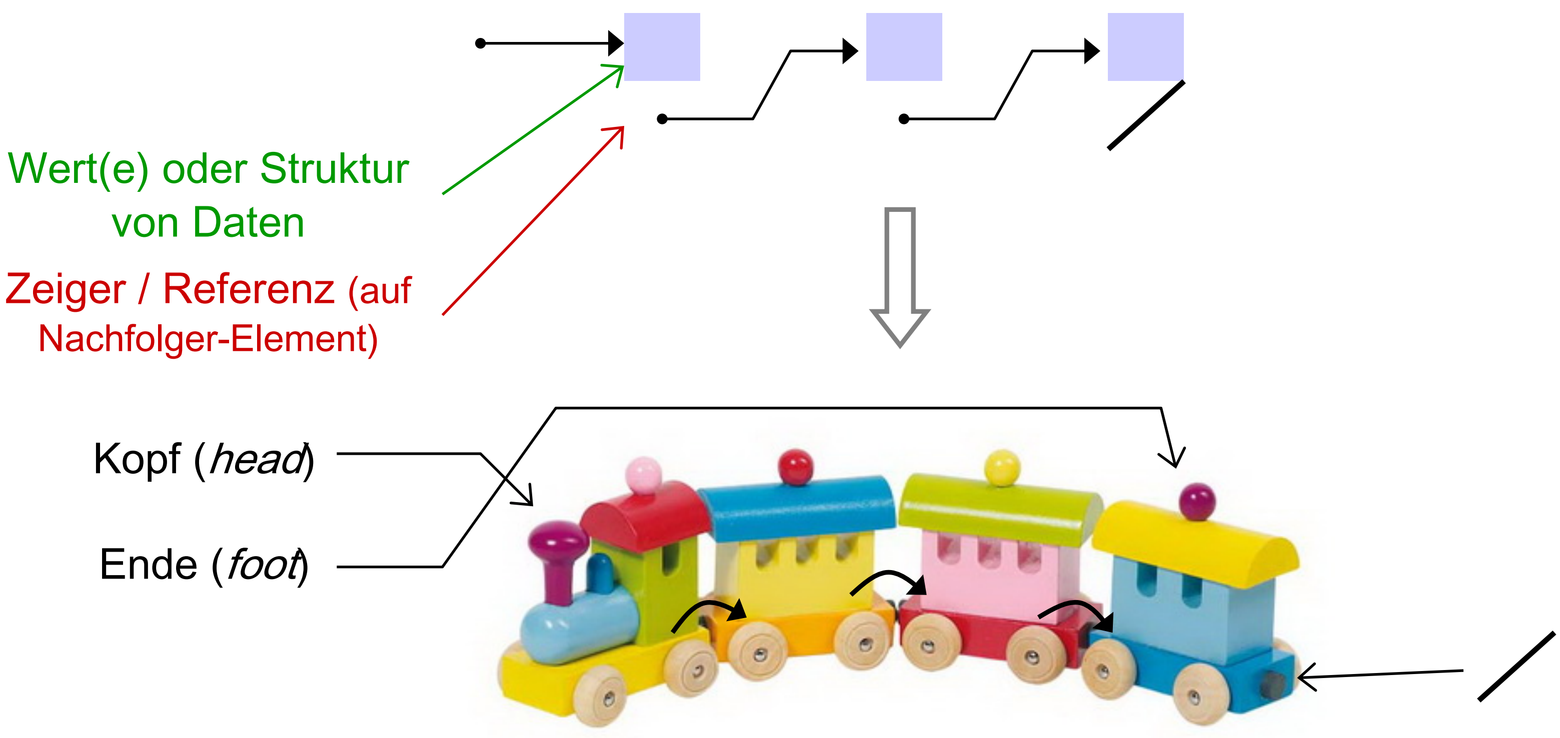


Aufwand: Es werden `arr.length+1` Zuweisungen benötigt

Dynamische Speicherverwaltung

- In vielen Aufgaben fallen Daten nach Bedarf an, die verwaltet werden müssen, beispielsweise
  - die Teilnehmer an einer Veranstaltung
  - die Teller auf einem Stapel
  - die Prozesse in einer Warteschlage

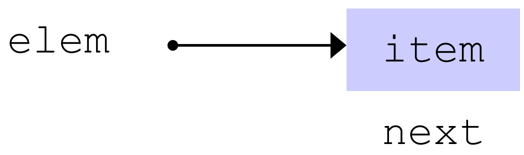
- Es werden für neue Elemente jeweils dynamisch einzeln und nacheinander neue Repräsentationen erzeugt
- Es können beliebig viele Elemente repräsentiert werden – und zwischendurch auch wieder frei gegeben werden
- Die Elemente werden <u>nicht</u> wie bei *Arrays* in eine feste Struktur mit Elementen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



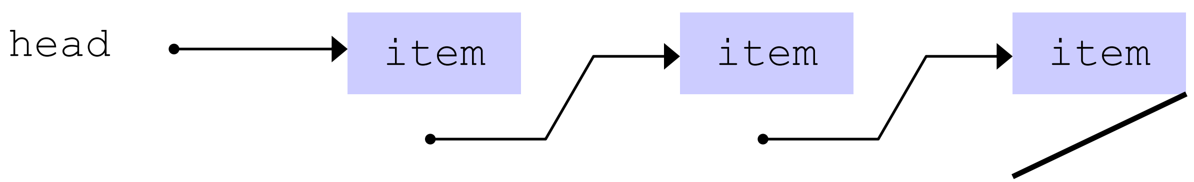
Einfache Struktur linearer Listen

Aufbau

- Eine lineare Liste besteht aus einer beliebigen Anzahl von Elementen
- Wichtige Strukturmerkmale der Repräsentation
  - Jedes Element enthält Daten, hier abstrakt repräsentiert durch einen Wert (`item`)
  - Jedes Element enthält einen Zeiger auf das jeweils nächste Element der Liste (`next`); man spricht von **einfach verketteten Listen**



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

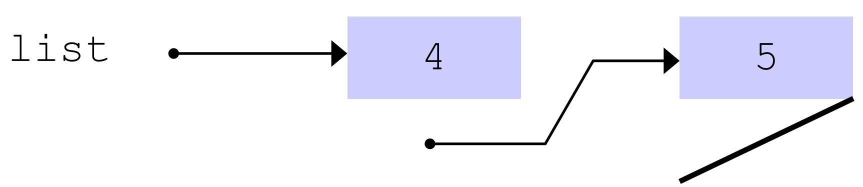


- Realisierung in Java

```
public class ListElem {

    public int      item;  // Platzhalter
    public ListElem next;

    public ListElem() {                            // Konstruktor
    }

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

```
... <beispielsweise in main-Methode>
ListElem list = new ListElem(4, new ListElem(5, null));
...
```

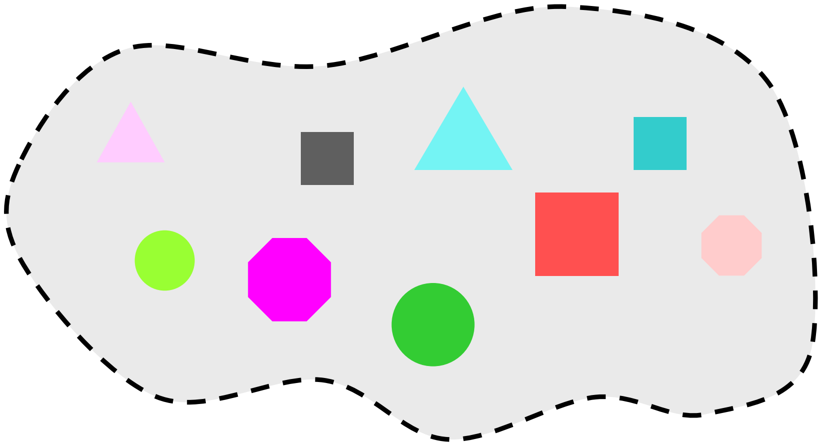
erzeugt dann:



Klassen für Repräsentationen strukturierter Datenobjekte

Definition als Klassen in Java

- In dem Listenbeispiel diente `item` vom Datentyp **`int`** als Platzhalter; will man Elemente verwalten, die selbst aus verschiedenen unterschiedlich strukturierten Komponenten zusammen gesetzt sind, dann kann hierfür eine eigene Klasse definiert werden
- Zusammenfassung von Elementen unterschiedlicher Elementtypen (unterschiedliche Formen) mit unterschiedlichen Werten (Farben) – vgl. die Visualisierung von *Arrays* in **Teil V**)

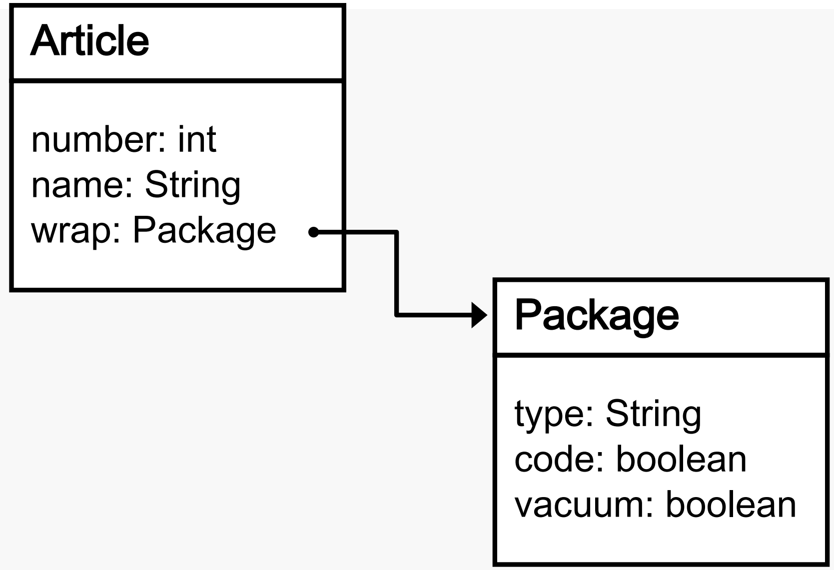


= Klasse

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

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!

Strukturierte Daten in prozeduralen Sprachen – *Records*

- Prozedurale (imperative) Programmiersprachen erlauben neben *Arrays* auch Strukturen für die Deklaration von Datenmengen als Zusammenfassung von Elementen
- Diese werden in solchen Sprachen `Record` oder `struct` genannt

- **`Structs`** in <u>C</u>

<u>Bsp</u>.:

```
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;
}
```

- **`Records`** in <u>Modula-2</u> (als Deklarationen neuer Datentypen)

<u>Bsp</u>.:

```
Date = RECORD
   Day   : [1 .. 31];
   Month : [1 .. 12];
   Year  : [1900 .. 2020];
END;
Courses = (PI, TI, PvS, Theo, Math);
Student = RECORD
   Name, Prename : ARRAY [0 .. 19] OF CHAR;
   Birthday      : Date;
   Grades        : ARRAY Courses OF REAL;
END;
```

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 Operatonen zum Hinzufügen, Suchen, Löschen, … von Elementen definiert werden; die Fallunterscheidungen bergen Fehlerrisiken, die die Stabilität von Programmen gefährden
- 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`)

- Realisierung in Java

```
public class ListElem {
    ...  // Definition und Konstruktoren wie auf S.13
}

public class List {
    private ListElem head;

    public List() {          // Konstruktor
    }

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

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

Operationen auf einfach verketteten Listen

Einfügen von Elementen

Einfügen am Anfang einer Liste (als Methode in `List`)

- Bei einem *Array* müssten zunächst alle Elemente um eine Position nach hinten verschoben werden
- Bei der anfänglichen herkömmlichen Listendefinition müssten Fallunterscheidungen auf leere und nicht-leere Listen berücksichtigt werden (s.oben)
- Implementierung in Java

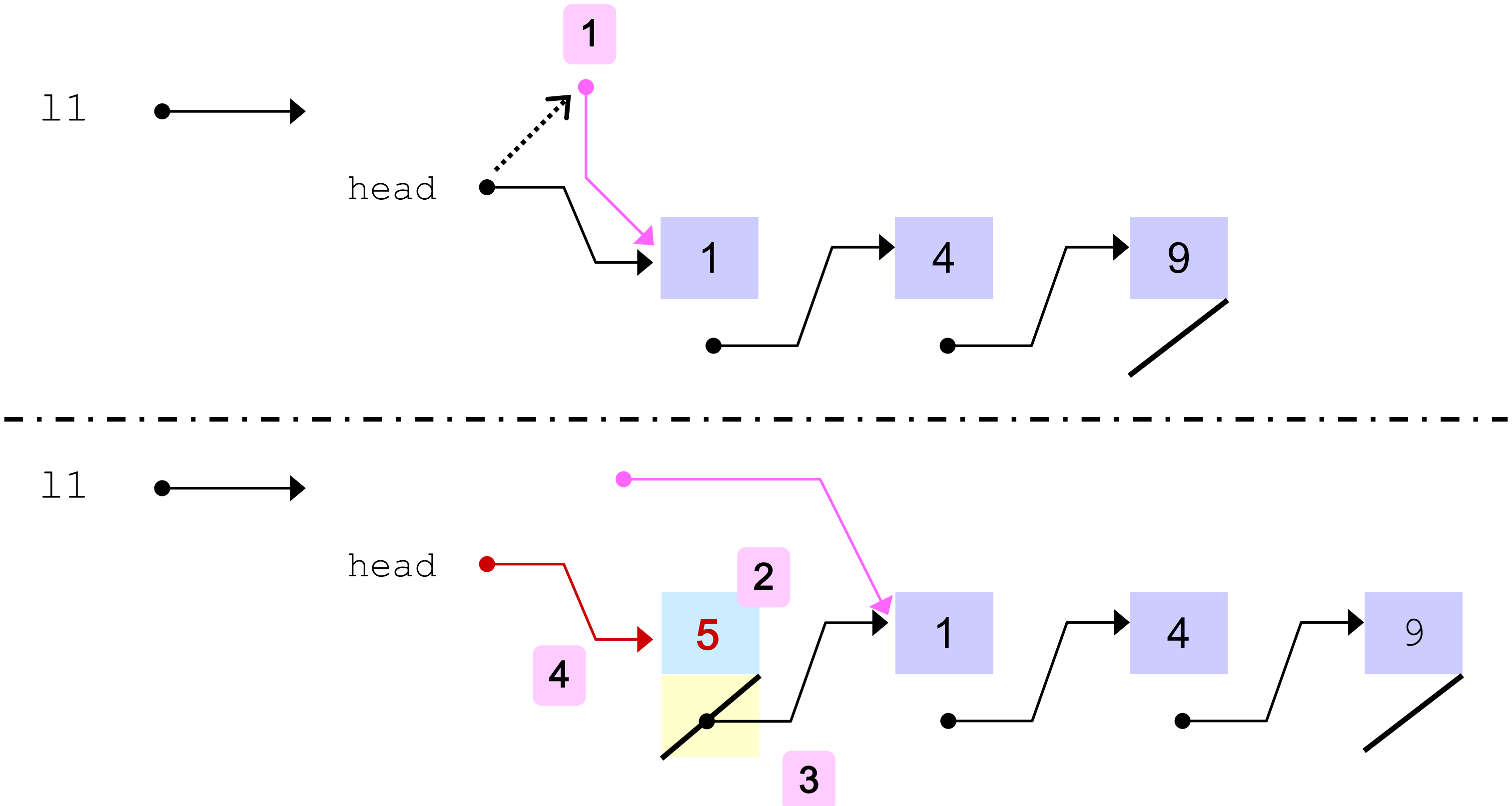
Kopie der Referenz auf das 1. Element der bisherigen Liste

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

Aufruf im Hauptprogramm:

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

Zeitlicher Ablauf: Die einzelnen Schritte bei der Datengenerierung und Verkettung in der Liste sind das Ergebnis des Aufrufs von `insertElemFirst(5)` und damit auch dem Aufruf der Klassenspezifischen Konstruktoren



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 einer Liste

- Da man (hier) nicht direkt auf das letzte Element der Liste zugreifen kann, muss es berechnet werden
- Die Bestimmung des Listen-Endes ist end-rekursiv, daher kann ein Algorithmus auch leicht in eine iterative Form überführt werden
- Rekursive Implementierung in Java (als Methode in `List`)

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

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

Einordnung: Das Überladen der Methode zum Eintragen des Listenelements dient dazu, die Benutzung für den Benutzer zu vereinfachen und die Fallunterscheidung ob die Liste bereits Elemente enthält zu verdecken

Suchen eines Elements in einer Liste

- **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
- Die Suchoperation ist aufgrund der rekursiven Struktur der Liste auch selbst rekursiv realisierbar
- Realisierung in Java

```
public ListElem searchElem(int value) {
    return searchElem(value, head);
}

private ListElem searchElem(int value, ListElem elem) {
    if (elem == null)                // Liste leer, Elem. nicht vorhanden
        return null;
    else if (elem.item == value)   // gefunden
        return elem;
    else
        return searchElem(value, elem.next);  // rekursiver Aufruf
}
```

Rest der Liste

- Die Implementierung ist end-rekursiv und daher einfach iterativ zu implementieren

```
public ListElem searchElemIter(int value) {
   ListElem current = 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 übersetzt die end-rekursive Prozedur mit ihrer Terminations-bedingung und der Weiterschaltung der verketteten Elemente

Ausgabe der Elemente einer Liste

Einfache Ausgabe der linearen Folge von Elementen

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

```
public void printList() {
   System.out.print("Liste ( ");
   printList(head);
   System.out.println(" )");
}

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

- Die Methode ist end-rekursiv und damit wiederum auch iterativ implementierbar

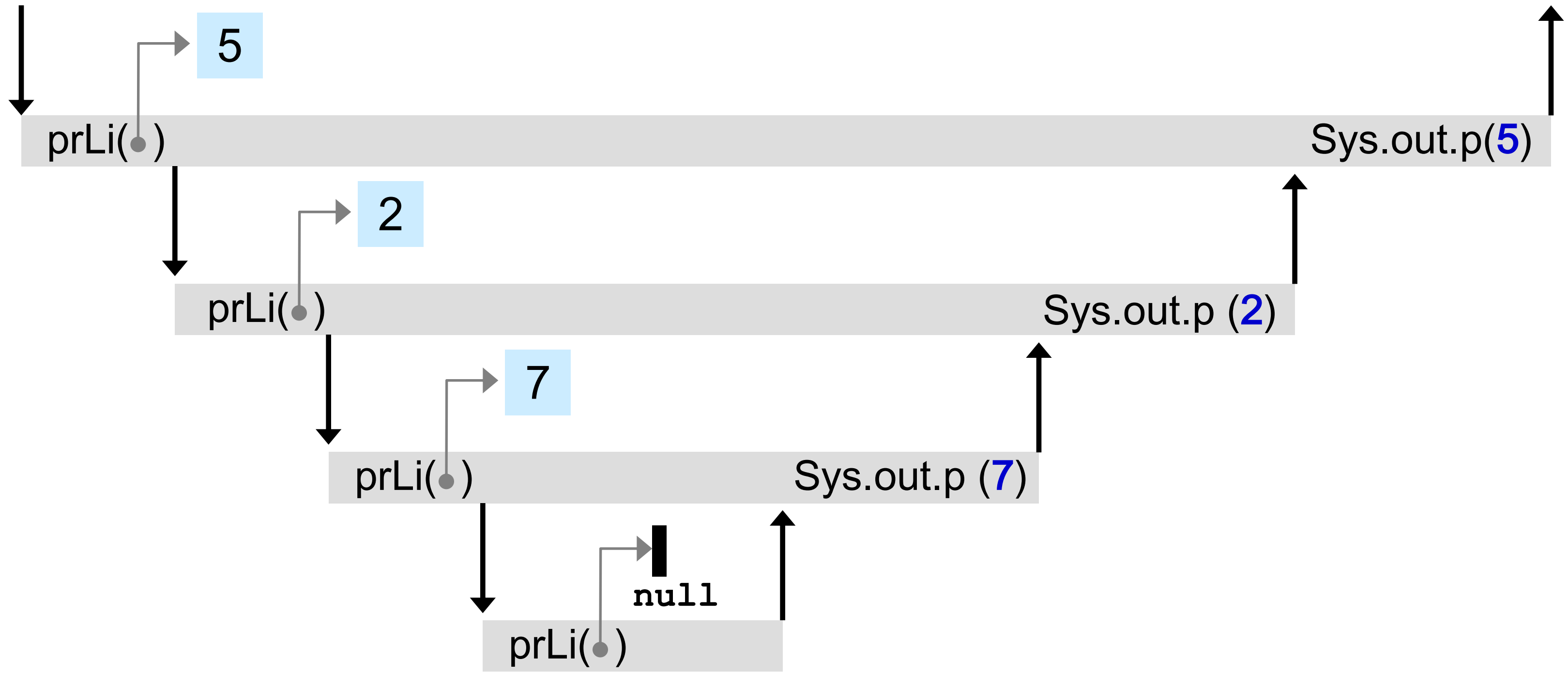
Liste rückwärts ausgeben

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

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

private 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!

- Da die Methode nicht end-rekursiv ist, lässt sie sich nicht ohne weiteres iterativ umsetzen
- 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

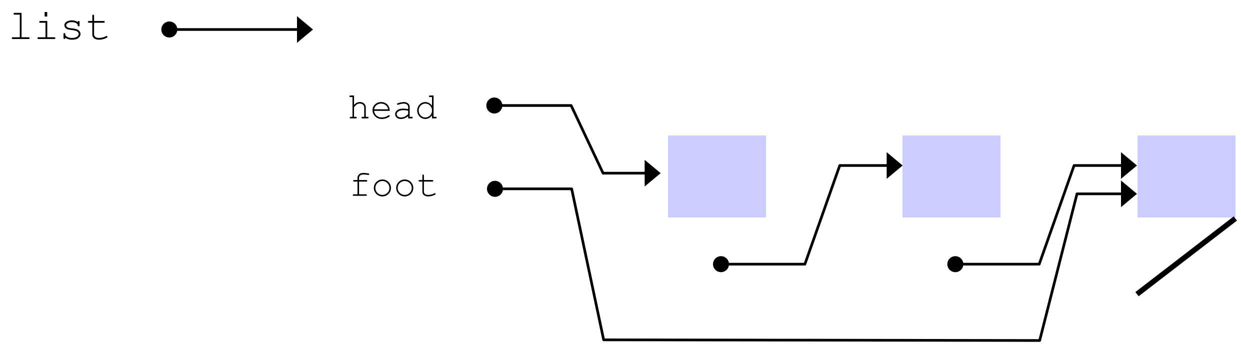
Struktur

- Die Listenelemente bleiben gleichermaßen definiert mit Datenteil (`item`) und Zeiger auf das nachfolgende Element (`next`)
- Das vollständige Ablaufen der Liste zum Einfügen eines Elements ist umständlich und zeitraubend, daher wird die Referenz auf das letzte Element der Liste (Fuß, `foot`) gespeichert

- Realisierung in Java

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

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

- Beispiel der Repräsentation einer Liste



Einfügen am Ende einer Liste – 2. Variante

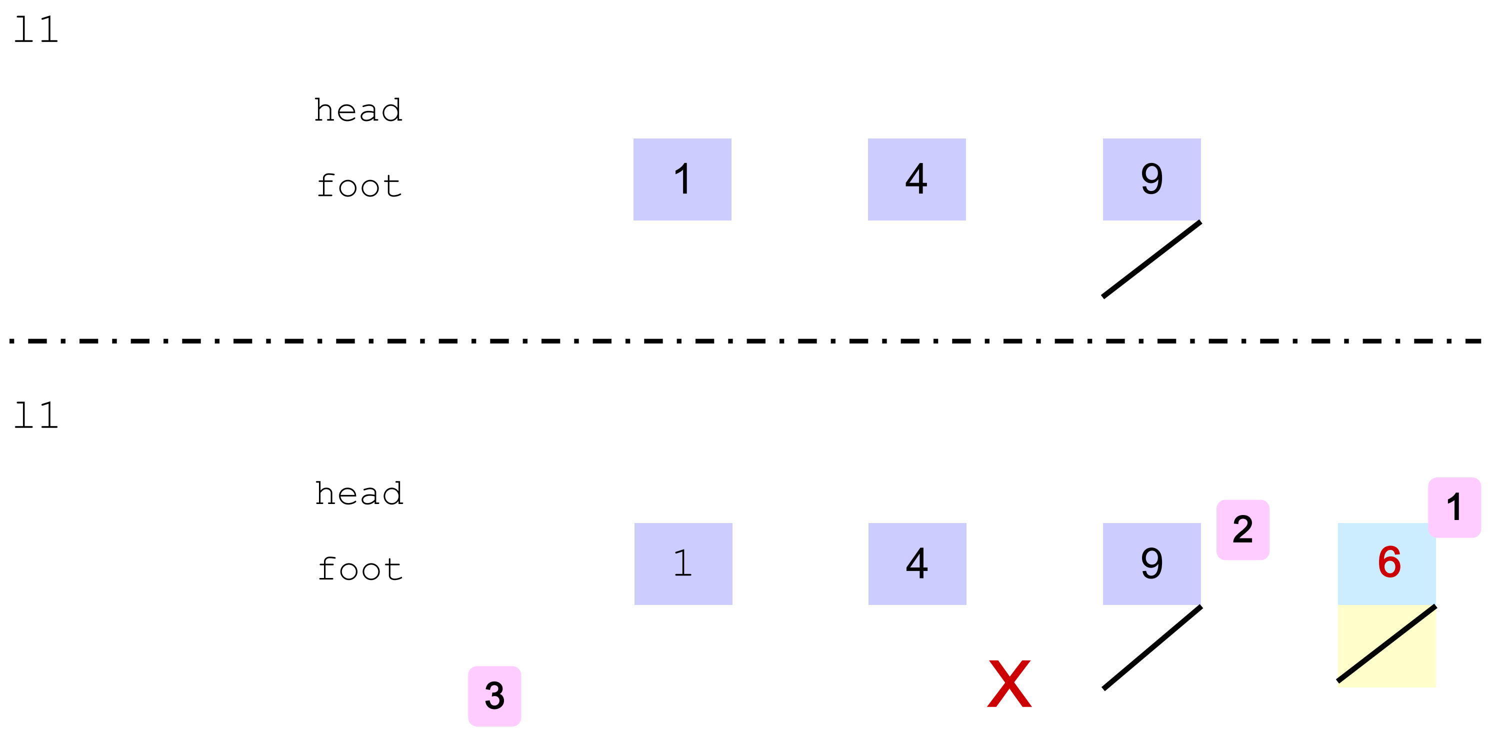
- Mithilfe des Listen-Fußes kann ein Element einfach am Ende der Liste angehängt werden
- Realisierung in Java (als Methode in `List`; 1. Variante auf S.23) ?

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

- Aufruf im Hauptprogramm:

```
List l1 = ...;  // Konstruktion
...
l1.insertElemLast(6);
```

- **Zeitlicher Ablauf** des Einfügens eines Elements am Listenende:



Anhängen einer Liste an eine andere

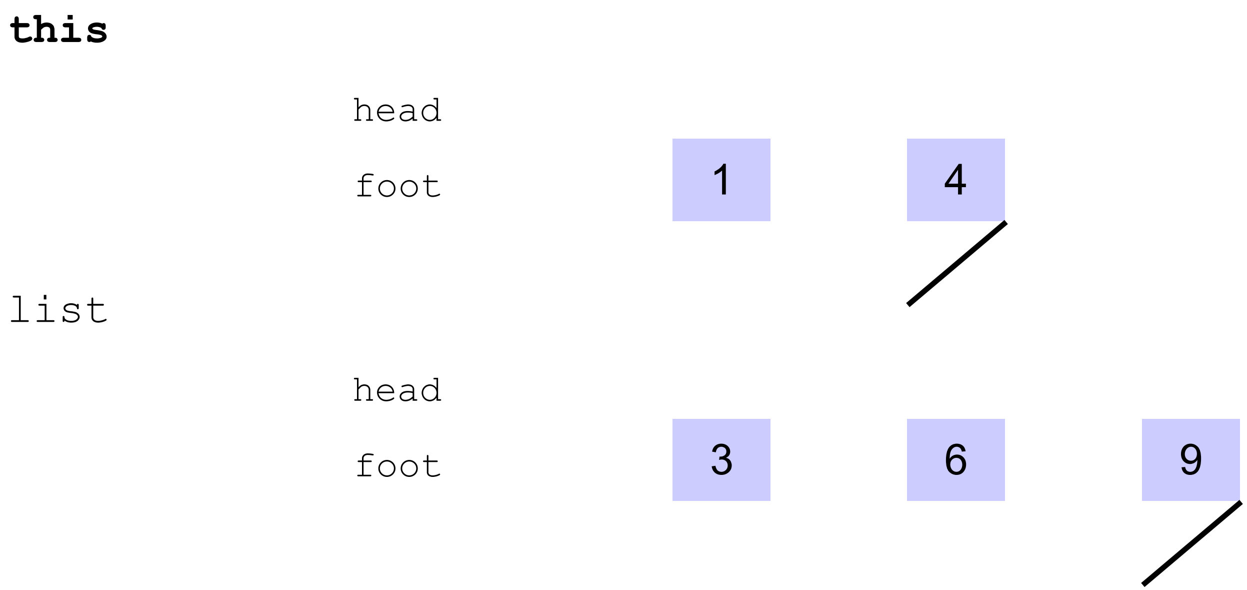
Einfache Realisierung

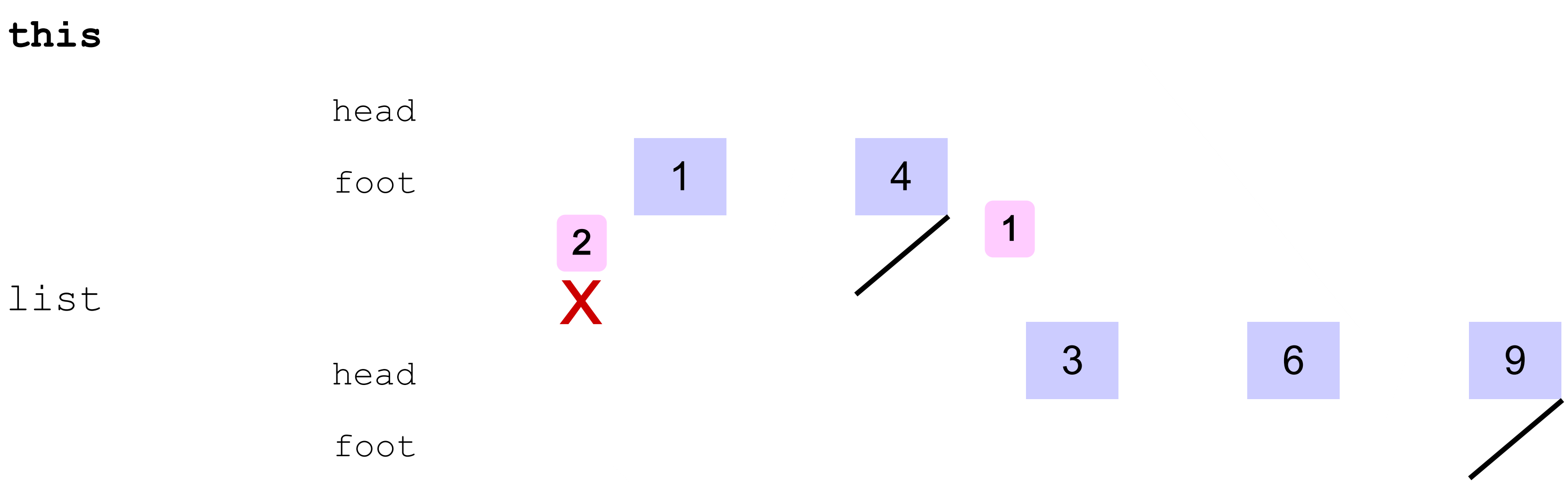
- **Ziel**: An eine Liste soll eine andere angehängt werden; dabei können neben zwei nicht-leeren Listen verschiedene Spezialfälle mit leeren Listen auftreten

- Realisierung in Java

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

- **Zeitlicher Ablauf** am Beispiel des `else`-Teils:





Seiteneffekte

- **Problem**: Beim Arbeiten mit Zeigern muss auf eventuelle Seiteneffekte geachtet werden, **wenn Objekte von mehren Stellen aus referenziert werden**

Bsp.:

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

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

l1.appendList(l2);     1
l2.insertElemLast(5);  2
l1.insertElemLast(7);  3
```

Auswirkung:

```
l1.append(l2);
```

l1

head

foot

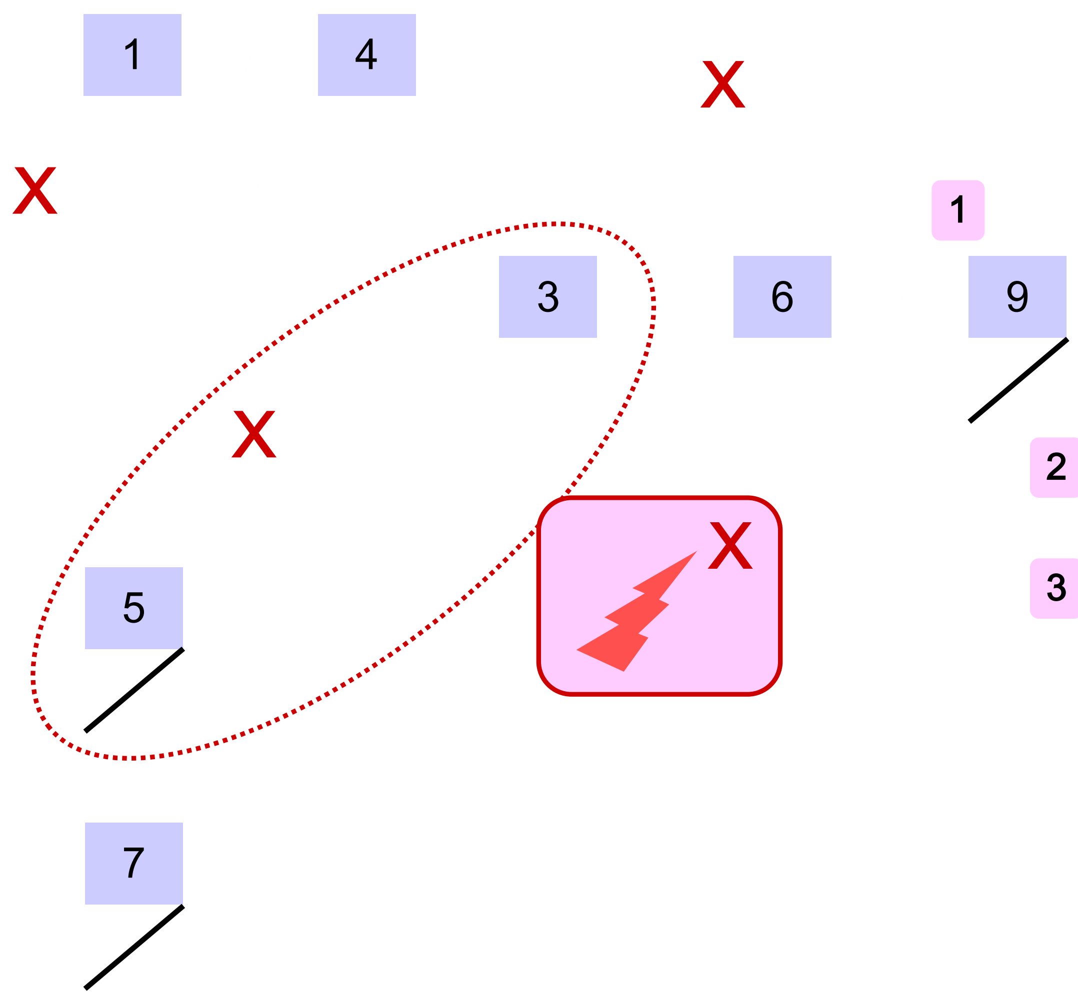
l2

head

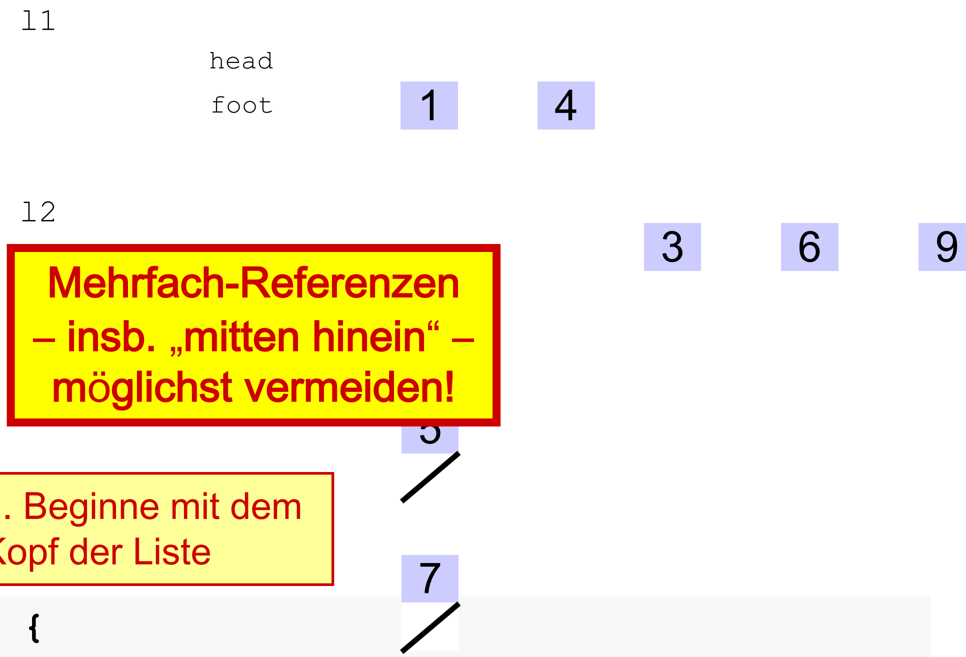
foot

```
l2.insertElemLast(5);
```

```
l1.insertElemLast(7);
```



Problem: Wegen der Referenz an ⚡ wird die Listenstruktur für `l2` inkonsistent (`head` und `foot` Element sind nicht verbunden; außerdem ist das Element mit `item == 5` nicht Element der Liste `l1`)



- **Lösung**: Bei `append` müssen alle Elemente der 2. Liste (die angehängt werden soll) kopiert werden; dies kann durch element-weises Anhängen realisiert werden, wozu die Methode `insertElemLast(...)` verwendet werden kann

Mehrfach-Referenzen – insb. „mitten hinein“ – möglichst vermeiden!

1. Beginne mit dem Kopf der Liste

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

2. Überprüfe, ob Liste leer ist

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

4. Weiterschalten auf das nächste Listenelement

5. Überprüfe, ob die Liste zu Ende ist

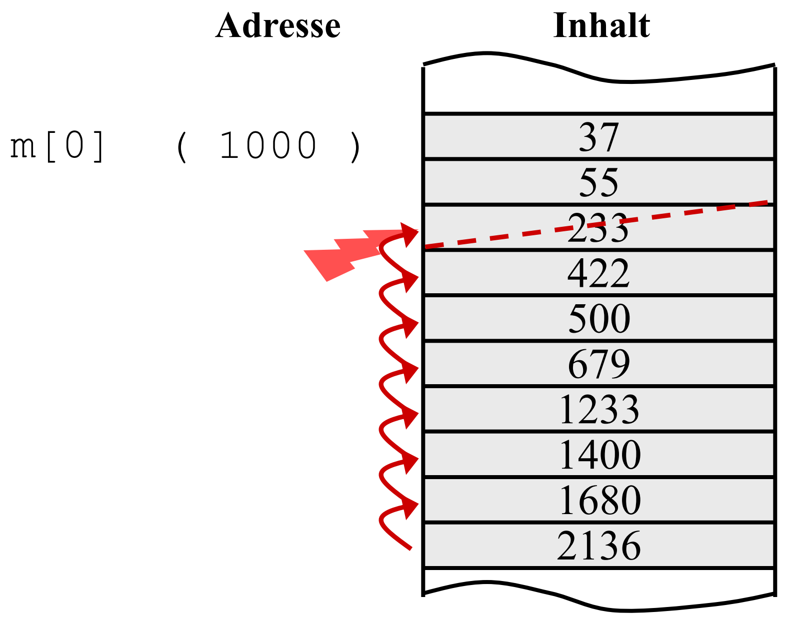
6. Einfügen des Wertes des Elements am Ende der Liste

Löschen / Einfügen von Listen-Elementen an einer beliebigen Position

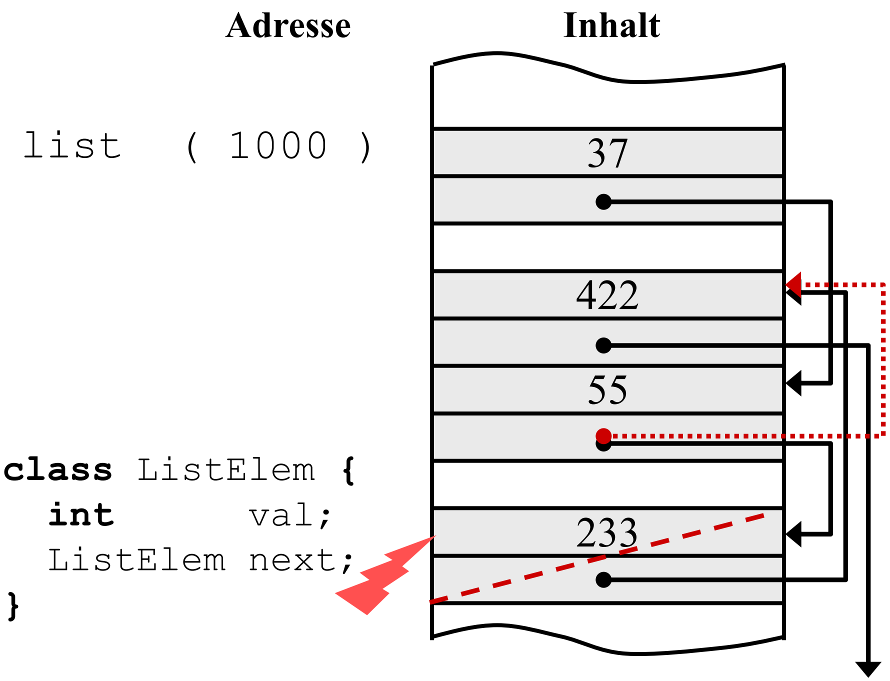
Löschen eines Elements an gegebener Position

- Ziel: Es soll das Element der Liste an der Position `pos` gelöscht werden – und weiterhin eine konsistente Liste erhalten bleiben
- Repräsentation von Feldern (*Arrays*) und linearen Listen (im Speicher)

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



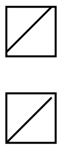

```
ListElem list = ...;
```




```
class ListElem {
  int      val;
  ListElem next;
}
```

Löschen des Elements `pos` in einer linearen Liste mit Kopf- + Fußzeiger

- Liste ist leer / `pos` existiert nicht:

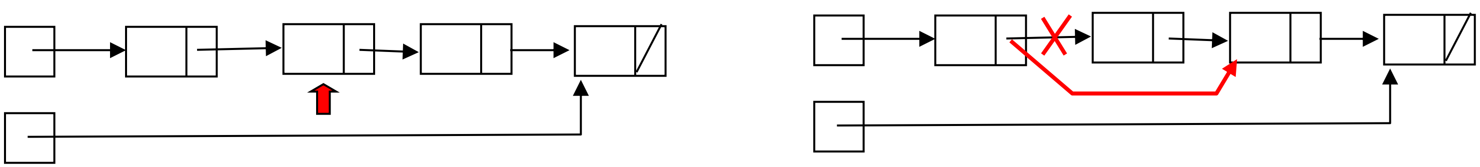


Entweder 'OK' oder 'Fehler' melden

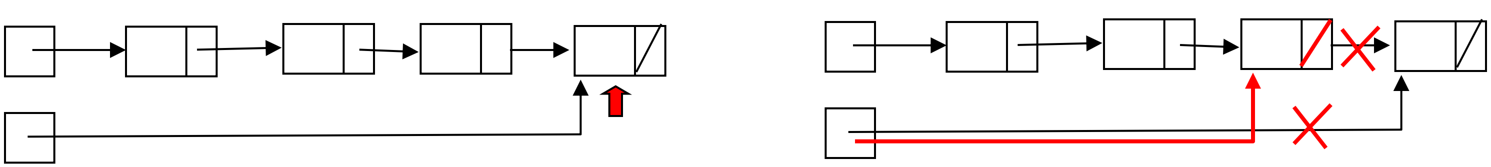
- `pos` ist einziges Element

- `pos` ist erstes Element

- `pos` ist „mittendrin“



- `pos` ist letztes Element

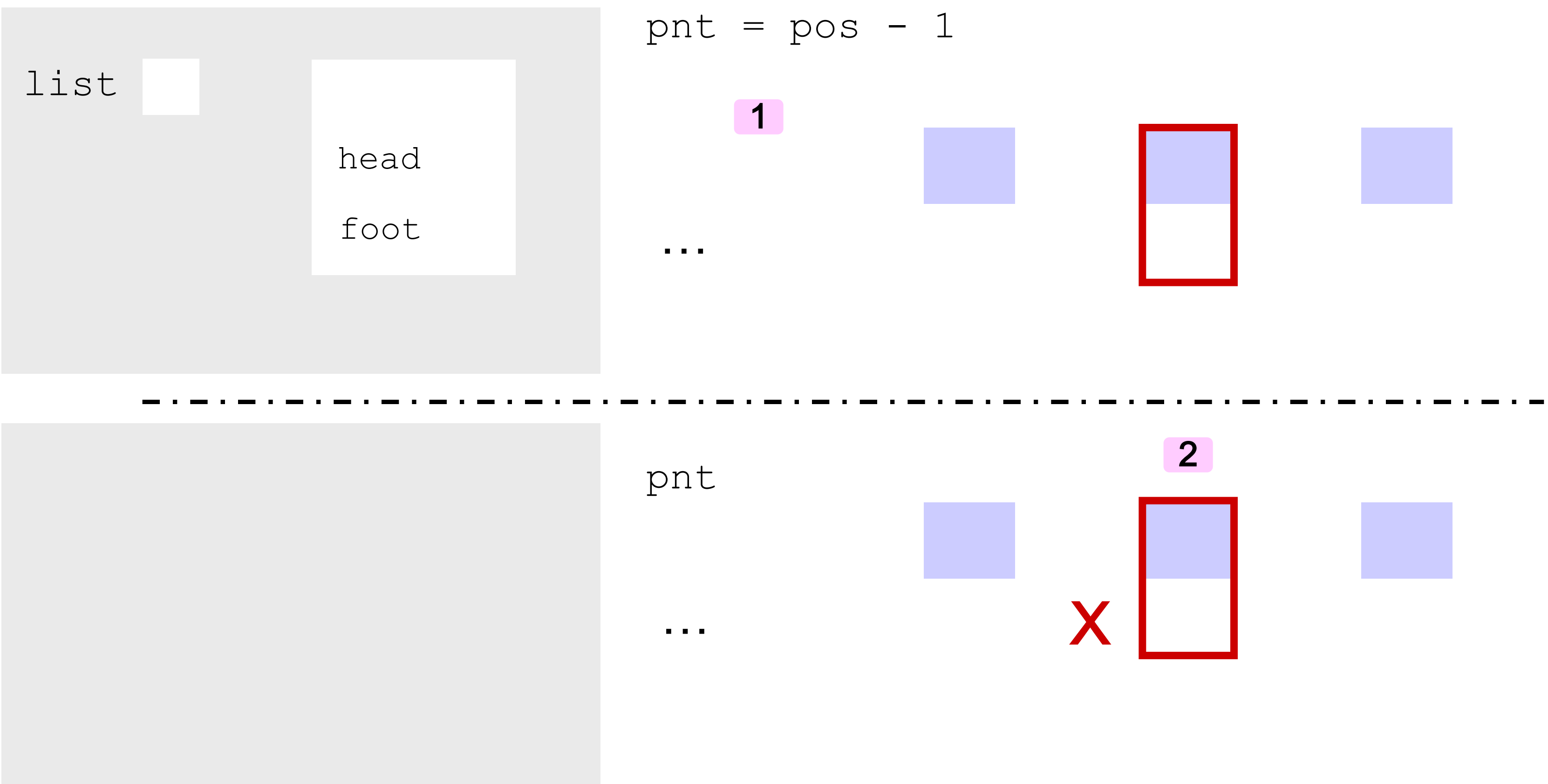


- Realisierung in Java

```
public void deleteListElem(int pos) {
    if (head != null) {
        if (pos == 1) {                      // Element am Listenanfang
            if (head == foot)
                foot = null;
            head = head.next;
        }
        else {  // vorheriges Element mit getListElem(...)
            ListElem pnt = getListElem(pos-1);  1

            if ((pnt != null) && (pnt.next != null))
                if (pnt.next == foot) {  // Element am Listenende
                    pnt.next  = null;
                    foot = pnt;
                }
                else                         // Element mittendrin
                    pnt.next  = pnt.next.next;   2
        }
    }
}  // end deleteListElem
```

- Operation auf den Listenelementen

Bsp.: Löschen des Elements an der Listenposition `pos` (erster Index = 1)



Methode `getListElem(...)`

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

    if (head != null) {
        pnt = head;
        for (int i = 1; i < pos; i++) {
            if (pnt.next == null)
                throw new IndexOutOfBoundsException();
            else
                pnt = pnt.next;
        }
    }
    return pnt;
}  // end getListElem
```

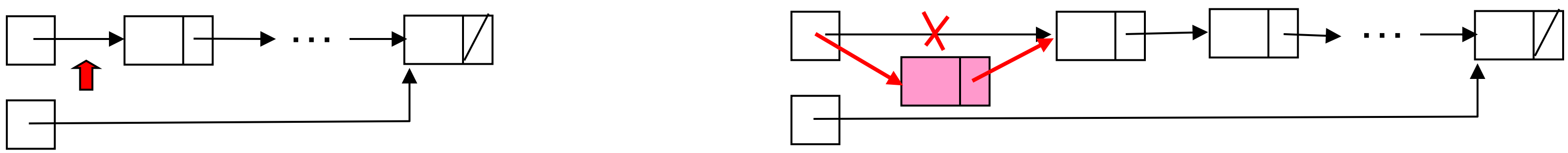
Erläuterungen:

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

Einfügen Element an Stelle `pos` in einer linearen Liste mit Kopf- + Fußzeiger

- Liste ist leer

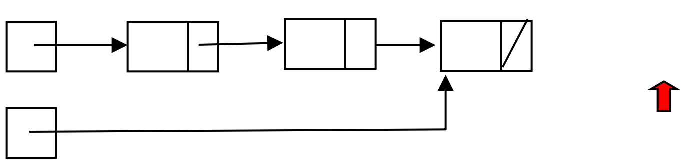
- Einfügen als erstes Element



- Einfügen „mittendrin“

- Einfügen als letztes Element

- Einfügen (weit) hinter Listenende



Entweder 'Fehler' oder Einfügen "leerer" Listenelemente zum Auffüllen des Zwischenraums

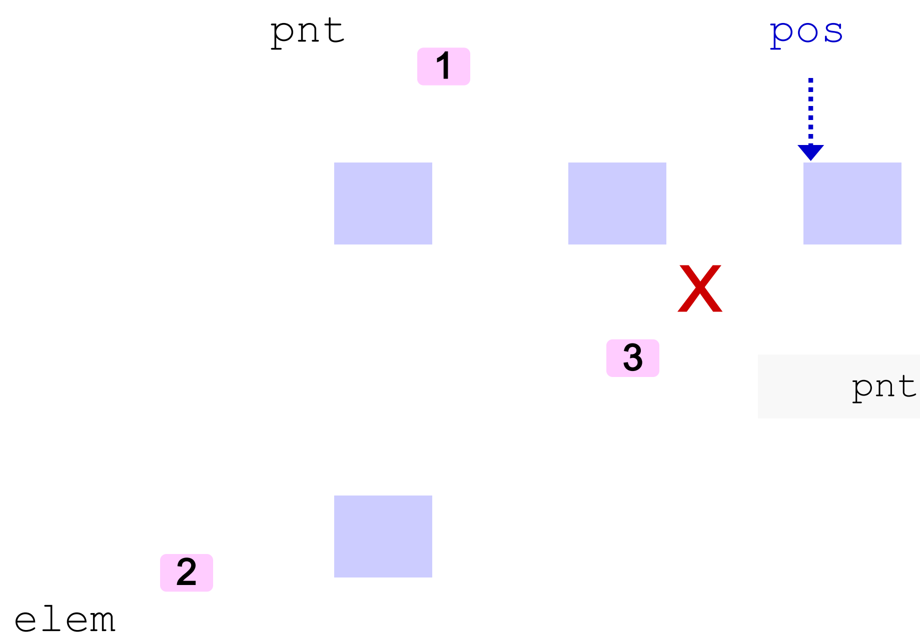
Einfügen **eines Elements an gegebener Position**

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

```
public void insertListElem(int pos, ListElem elem) {
    if (pos == 1) {  // 1. Element
        if (head == null) {  // leere Liste
            head = elem;
            foot = head;
        }
        else {                          // nicht-leere Liste
            elem.next = head;
            head = elem;
        }
    }
    else {
        ListElem pnt = getListElem(pos-1);  1
        if (pnt != null) {  // pos im Bereich der Listen-Laenge
            elem.next = pnt.next;  2
            pnt.next  = elem;  3
            if (elem.next == null) foot = elem;
        }
    }
}  // end insertListElem
```

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

```
ListElem pnt = getListElem(pos-1);
```




```
pnt.next  = elem;
```

```
elem.next = pnt.next;
```

<u>**Erläuterungen**</u>:

- Das neue Element steht nach der Einfügeoperation an der als Parameter angegebenen Position in der Liste
- Wollte man die Einfügeoperation dahingehend ändern, dass das Element <u>nach</u> dem Element der gegebenen Position eingefügt wird, muss `getListElem(pos)` aufgerufen werden

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

- Realisierung in Java

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

Erläuterungen: Die Realisierung von `insertListElemSorted(...)` wird hier mittels Überladens der Methode realisiert, indem zwei Methoden desselben Namens mit unterschiedlicher Parameterliste definiert werden:

- `insertListElemSorted(ListElem elem)`
- `insertListElemSorted(ListElem elem, ListElem list)` (nächste Seite)

Weiterführung der Methoden-Definition

```
private void insertListElemSorted(ListElem elem, ListElem list) {
    if (list.next == null) {                      // elem am Ende einfuegen
        list.next = elem;
        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
```

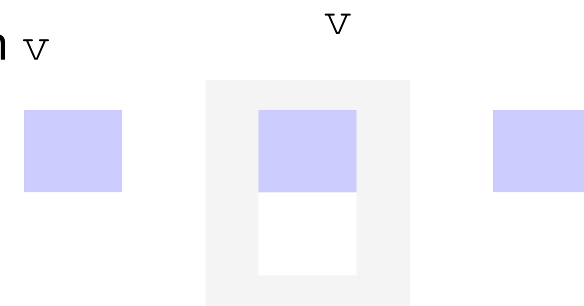
Erläuterungen:

- Wenn ein einzufügendes Listen-Element den gleichen Schlüsselwert wie ein schon vorhandenes Listen-Element besitzt: wie wird dann das neue eingeordnet?
- Mit der (rekursiven) Methode wird sicher gestellt, dass ein einzufügendes Element mit dem momentan höchsten Schlüssel-Wert am Ende der Liste eingeordnet wird

Doppelt verkettete Listen (*doubly-linked lists*)

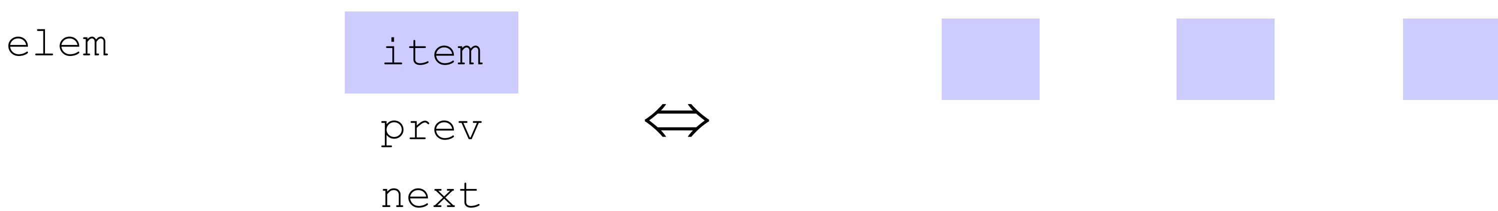
Motivation

- Bei einfach verketteten Listen speichert ein Element stets die Referenz auf seinen direkten Nachfolger
- Aufgabe: Bestimme den Vorgänger-Knoten von `v`



<u>Lösung</u>: Suche vom Kopf der Liste bis der Vorgänger `elem.next == v` erreicht ist; der Aufwand für die Operation ist abhängig von der Position von `v`

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



Symmetrische Struktur

Repräsentation

- Erweiterung der Klasse für Listen-Elemente (in Java)

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

   public ListElem() {
   }

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

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

- **Vorteile** doppelt verketteter Listen
  - 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

    Die Bestimmung eines Elements – an einer gegebenen Position – kann vom Anfang oder vom Ende her starten, je nachdem, von welcher Seite der Abstand kürzer ist; der Aufwand wird im Mittel halbiert

- **Nachteile** doppelt verketteter Listen
  - 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

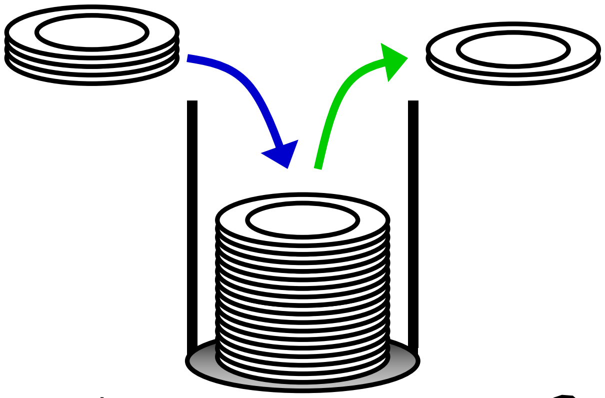
2. Stapel und Schlangen

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

Lineare Strukturen mit Zugriffsbeschränkungen

Einordnung

- Bisher: Geordnete Strukturen, auf deren Elemente unabhängig von ihrer Position zugegriffen werden konnte (z.B. Arrays, lineare Listen)
- Jetzt: Strukturen, bei denen zu einem bestimmten Zeitpunkt jeweils nur auf ausgezeichnete Elemente zugegriffen werden kann

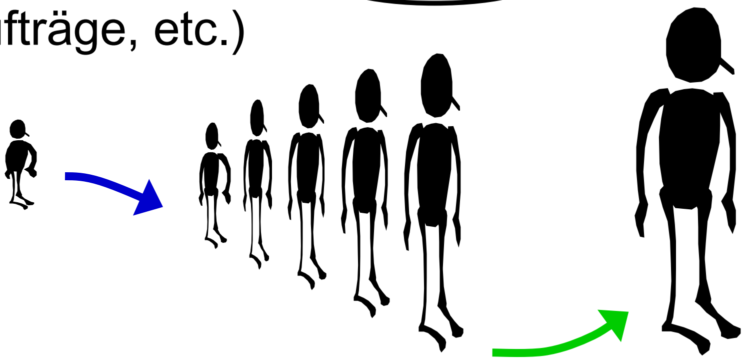


1. Stapel (Papier, Teller, Tabletts, etc.)

   Prinzip:
   - Objekte werden jeweils oben aufgelegt
   - jeweils das oberste Objekt wird wieder herunter genommen

2. Warteschlange (Ski-Lift, Kasse, Druckaufträge, etc.)

   Prinzip:
   - hinten anstellen – und geduldig sein …
   - jeweils erste Position in der Schlange wird abgefertigt



Entsprechende Datenstrukturen

Stapel (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 in Anwendungen

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)

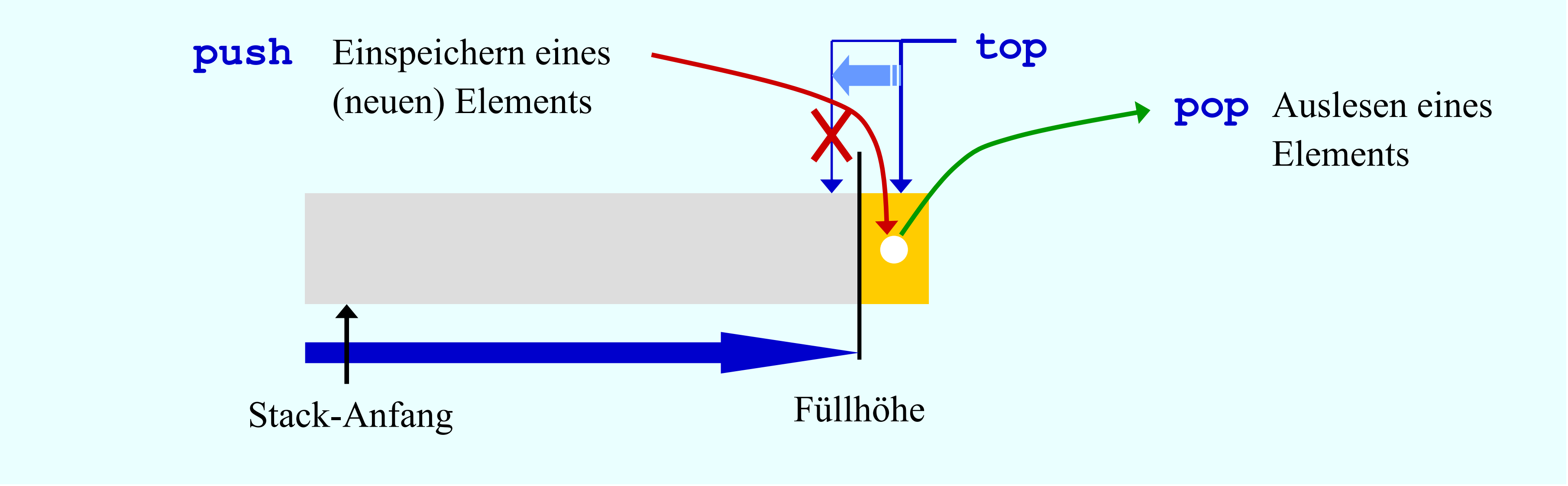
Schlangen (queues)

- 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 $\rightarrow$ Editor (Zeichenpuffer)

Stapel (*stacks*)

Basisoperationen

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



- Realisierung mittels …
  - Felder (*Arrays*) und „Kellerpegel" (hier: Variable `top`)
  - verketteter Listen (bei einer einfach verketteten Liste kann der Stapel einfach am Kopf (*head*) gefüllt werden und dort auch das oberste Element direkt ausgelesen werden)

Realisierung von Stapeln mittels Feldern (*Arrays*)

- Klasse mit Attributen, die den *Stack* repräsentieren; Implementierung mit einem *Array* mit fest definierter maximaler Füllhöhe (Details verdeckt) – Abhilfe durch die Verwendung dynamischer Arrays (s. Abschnitt 1)
- 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   = -1;  // Zeiger auf oberstes Element

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

    public boolean isEmpty() {        // Stack leer?
        return (top == -1);
    }

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

Beachte: `pop` ist nur für einen nicht-leeren *Stack* definiert. Deshalb muss **vor jedem** Aufruf von `pop` sichergestellt sein, dass der *Stack* nicht-leer ist.

Hier wäre der Einsatz von *Exceptions* (**Teil X**) angebracht: Wenn jemand versucht, ein Element von einem leeren *Stack* zu nehmen, sollte ein Fehler gemeldet werden.

- Hilfsklasse zum Test eines (dynamisch erzeugten) Stacks

```
public class TestStackInt {

   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 Stack liegen ... */
      while (!s.isEmpty())
         System.out.println(s.pop());  // Ausgabe der Elemente ...
}  // end class TestStackInt
```

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 für *Stack*-Elemente `stack` enthält beides:
  - Listenelemente in `stack[0 .. top]`
  - Freier Speicher in `stack[top+1 .. 999]`

Realisierung von Stapeln mittels (einfach) verketteter Listen

- *Stack* wird mit einfach verketteten linearen Listen implementiert
- Erzeugung von *Stack*-Objekten und Elementen jeweils mit *Default*-Konstruktor

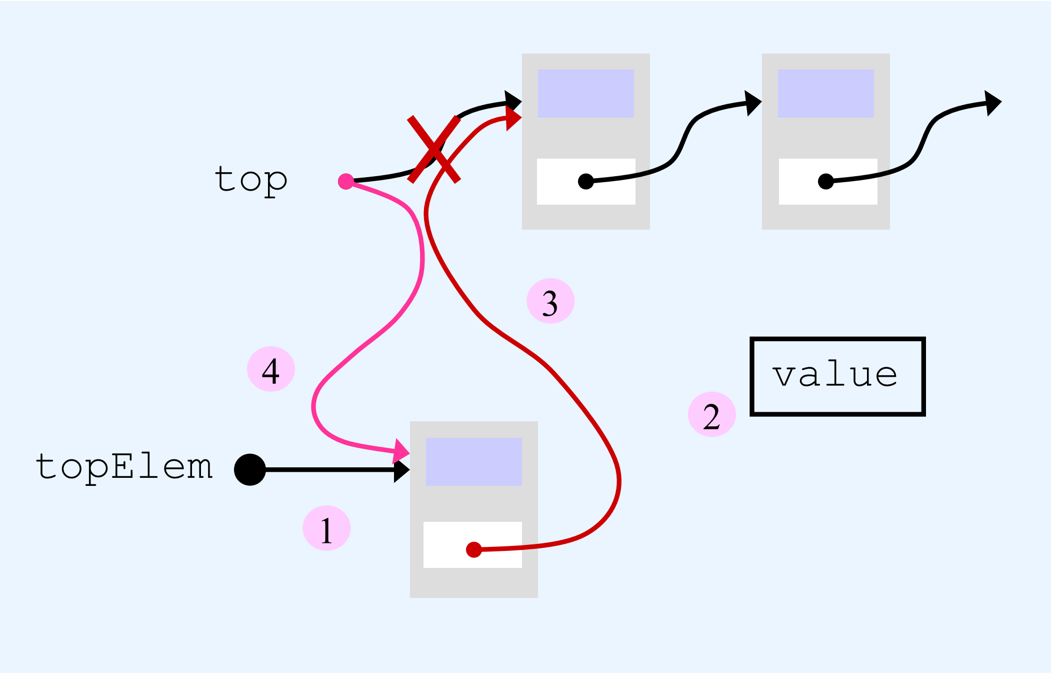
```
public class StackInt {
    private ListElem top = null;

    public int pop() {                  // Element von Stack lesen
        int value = top.item;

        top = top.next;
        return value;
    }

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

    public void push(int value) {   // Element auf Stack speichern
        ListElem topElem = new ListElem();

        topElem.item = value;
        topElem.next = top;
        top      = topElem;
    }
}  // end class StackInt
```

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

- Funktionsweise der `push`-Operation

```
public void push(int value) {  // Element auf Stack speichern
1  ListElem topElem = new ListElem();  // erzeuge neues Listen-Element

2  topElem.item = value;      // trage Wert value ein
3  topElem.next = top;        // Stack-Elemente an neues Element haengen
4  top          = topElem;    // neues Element bildet neuen Anfang
}
```

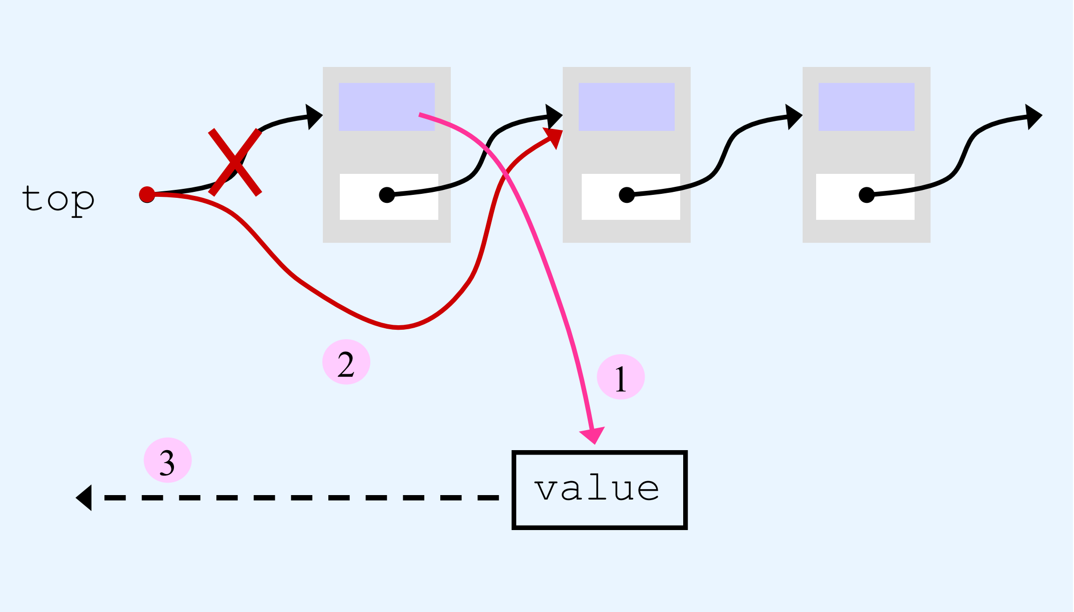
1. neues Listen-Element anlegen
2. `value` ablegen
3. neues Element vor die restlichen Elemente des *Stacks*
4. neues Element als neuen Listen-Anfang speichern



- Funktionsweise der `pop`-Operation

```
public int pop() {              // Element von Stack lesen
1  int value = top.item;        // Wert des obersten Elements speichern

2  top = top.next;              // oberstes Element loeschen
3  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

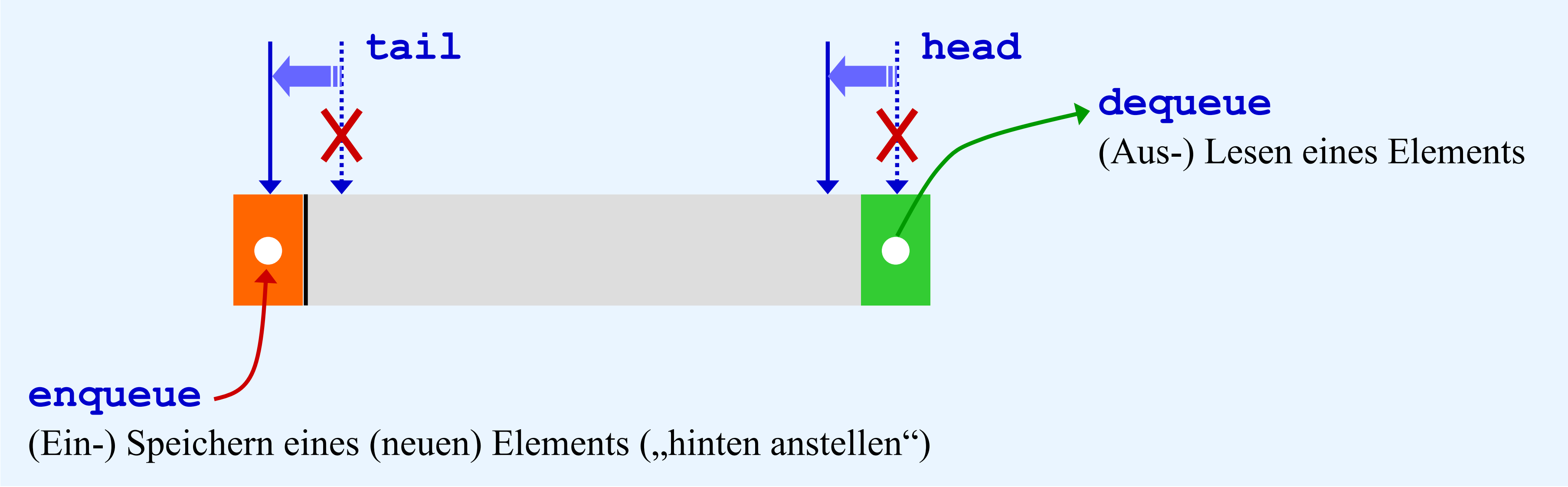


<u>**Bemerkung**</u>: Für diese Realisierung der `pop`-Operation gilt dieselbe Beobachtung wie bei der Realisierung von *Stacks* mit Feldern – es muss vor der Ausführung von `pop` geprüft werden, ob der *Stack* nicht leer ist (dies mittels *Exceptions* absichern)

Schlangen (*queues*)

Basisoperationen

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



- Realisierung (wie bei Stack) mittels Felder oder verketteter Listen
- **Spezialfall einer Schlange**: Es werden auch Schlangen mit Doppelende (*double-ended queue*) definiert, bei denen die Operationen `enqueue` und `dequeue` auf beiden Listenenden operieren

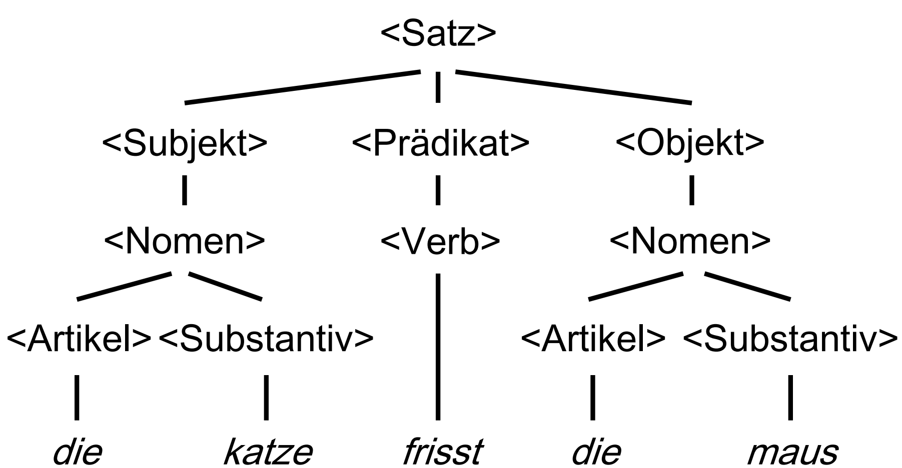
3. Bäume

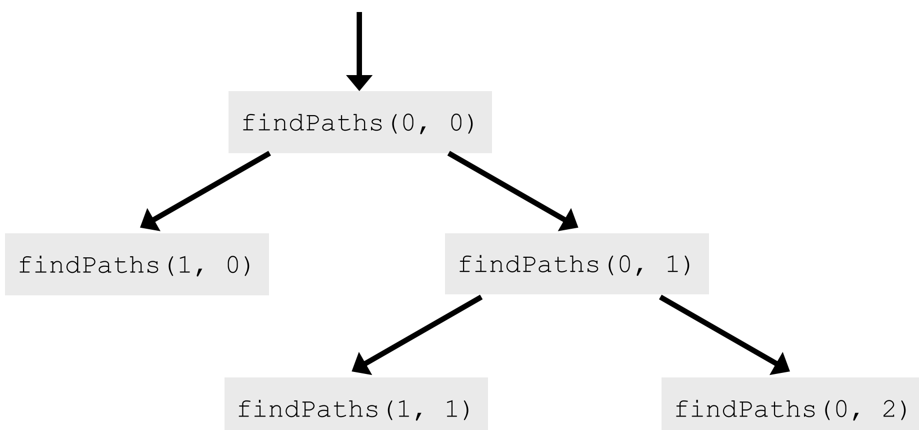
- Motivation und Einordnung
- Definitionen und Eigenschaften
- Binärbäume
- Operationen auf Binärbäumen
- ~~Geordnete Binärbäume – Suchbäume und Operationen~~
- Eigenschaften von Binärbäumen
- ~~Repräsentation allgemeiner Bäume~~
- ~~Termbäume – Auswertung arithmetischer Ausdrücke~~

Motivation und Einordnung

Allgemeine 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 (Teil IX)





Betrachtungen zu 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)
- **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

Wurzel (Knoten ohne Vorgänger)

Innere Knoten

Blätter (Knoten ohne Nachfolger)

Hinweis: Die Knoten entsprechen den Elementen in Listen und enthalten die `items`; die Knoten werden häufig in Form von Rechtecken oder Ellipsen/Kreisen dargestellt

Definitionen und Eigenschaften

Struktur und Generierung

- Definition (Bäume):

Ein Baum T ist ein Tupel

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

mit einer Knoten-Menge $V = \{v_i \mid i \in \mathbb{N}\}$ (vertices; auch V(T)), mit $0 \leq card(V) < \infty$ und einer Kanten-Menge $E = V \times V$ (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

$$R = \{(v_1, v_2), (v_2, v_3), (v_3, v_2), \ldots\}$$

so dass $E = \{e_1 = (v_1,v_2), e_2 = (v_2,v_3), \ldots\}$ durch jeweils

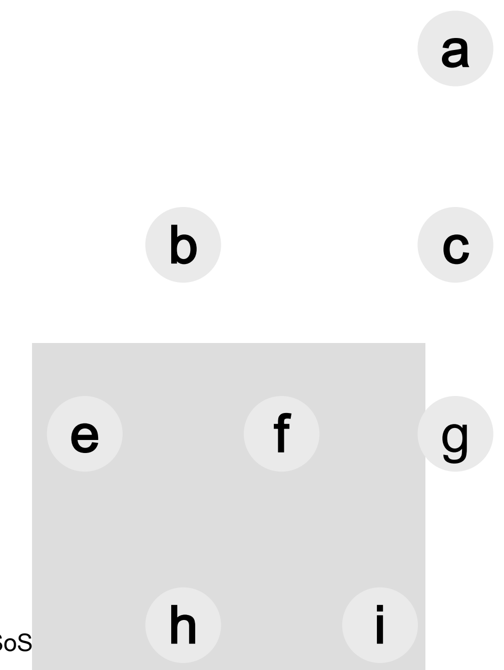
- einen Anfangs-Knoten v_i und
- einen End-Knoten v_j

festgelegt wird. Zusätzlich gelten die Eigenschaften, dass

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

- Bezeichnungen:
  - Die Nachfolger eines Knotens werden als Kinder bezeichnet; Vorgängerknoten eines Kindes bezeichnet man als Eltern- oder Vaterknoten
  - 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



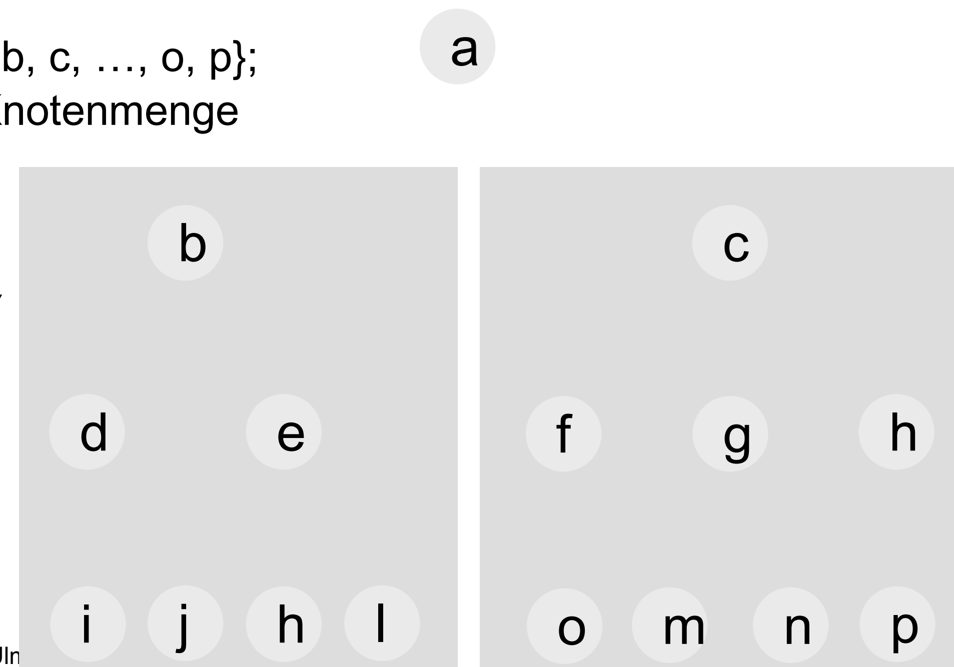
- Definition (Rekursive Definition eines Baums):

  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}; ein möglicher Baum T mit dieser Knotenmenge ist

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

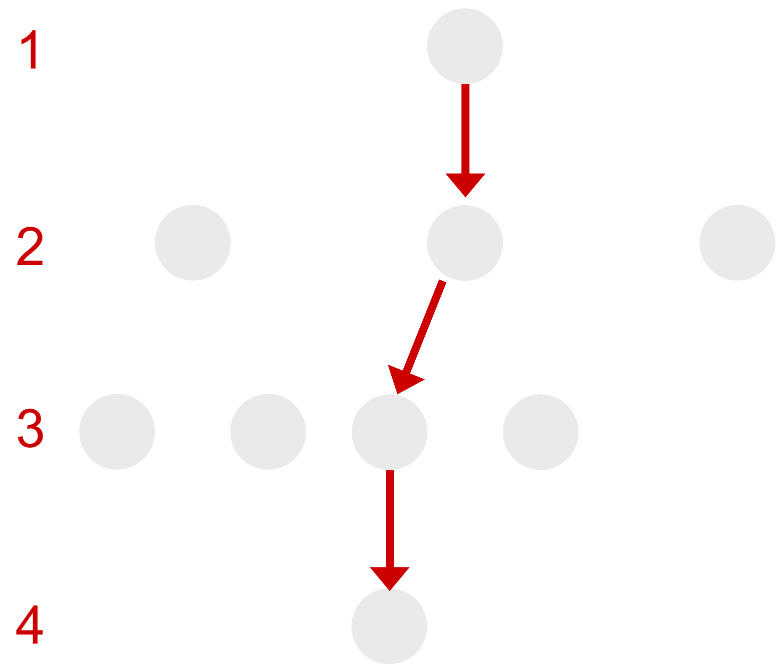


Eigenschaften von Bäumen

- Eine Kante 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 ; sie entspricht der Länge des Pfades von der Wurzel bis zu dem betreffenden Knoten
- Die Höhe (auch Tiefe) eines Baums ist die maximale Länge eines Pfades von der Wurzel bis zu einem Blatt

Bsp.: Höhe / Tiefe eines gegebenen Baums

Ergebnis: Höhe / Tiefe = 4



- 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

Binärbäume

Definition

- Ein Binärbaum hat einen Verzweigungs-Grad $deg_T(v) \leq 2, \ \forall v \in E(T)$
- **Definition (Binärbaum, abstrakte Definition)**:

  Ein Binärbaum T_B ist entweder
  - leer (*empty*), oder
  - er besteht aus
    - einem Knoten, der einen Wert des Elementtyps enthält sowie
    - zwei Teilbäumen.

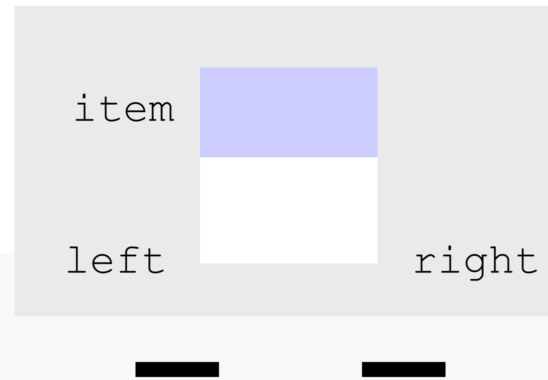
Objektorientierte Realisierung in Java

Einfache Verkettung der Knoten

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

bTree

root




```
public class Node {
    char item;
    Node left,    // Zeiger auf das linke Kind
         right;   // Zeiger auf das rechte Kind

    public Node() {
    }

    public Node(char item, Node left, Node right) {
        this.item  = item;
        this.left  = left;
        this.right = right;
    }
}  // end class Node
```

- 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 BTree {
    private Node root;  // Wurzelknoten des Baums

    public BTree() {
        root = null;
    }

    public BTree(char value) {
        root = new Node(value, null, null);
    }

    ...  // Methoden, die auf (Binaer-) Baeumen operieren
}  // end class BTree
```

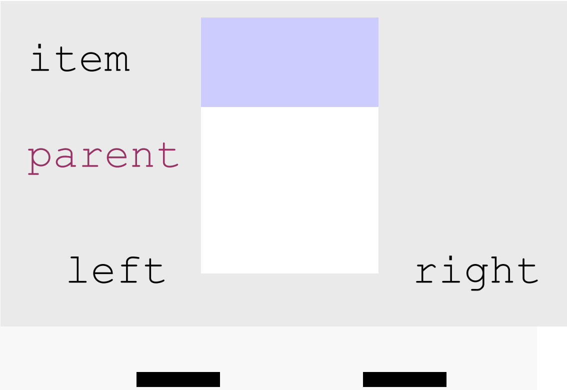
Doppelte Verkettung der Knoten

- Wie bei den linearen Listen kann es Vorteile bringen durch die Baumstruktur
    - von oben nach unten von der Wurzel zu den Blättern (*top-down*) als auch
    - von unten nach oben von den Blättern zur Wurzel (*bottom-up*)

  laufen zu können

- Mit der einfachen Verkettung werden *top-down* Durchläufe unterstützt; für *bottom-up* Durchläufe muss in die Knotenrepräsentation ein Zeiger auf den Vorgängerknoten (*parent*) aufgenommen werden

- Deklaration eines Knotens (erweitert)




```
public class Node {
    char item;
    Node parent,  // Zeiger auf Eltern-Knoten
         left,    // Zeiger auf das linke Kind
         right;   // Zeiger auf das rechte Kind

    public Node() {
    }

    public Node(char item, Node parent, Node left, Node right) {
        this.item   = item;
        this.parent = parent;  // this.parent = this
        this.left   = left;
        this.right  = right;
    }
}  // end class Node
```

- **Baumstruktur**: Die Klasse für Binärbäume für die Verwaltung der Knoten kann so belassen werden; der Wurzelknoten eines Baums (*root*) verweist in der erweiterten Deklaration mit dem Vorgänger auf sich selbst

Operationen auf Binärbäumen

Durchlaufen (Traversieren) von Binärbäumen

Einordnung

Für das Durchlaufen von Binärbäumen gibt es verschiedene Vorgehensweisen (hängt von der Aufgabenstellung und den in den Knoten repräsentierten Daten ab); man unterscheidet je nach Durchlaufrichtung

- **Tiefen**durchläufe und
- **Breiten**durchläufe

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
  - *Post-order* Strategie

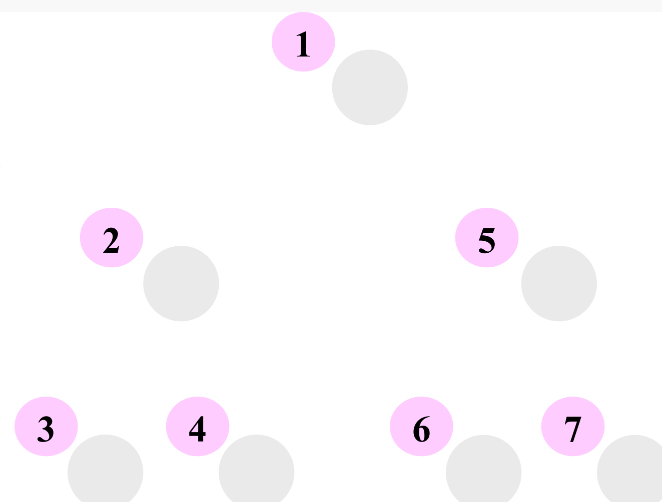
Hinweis: Die Bearbeitung eines Knotens ist im einfachsten Fall der Besuch und die Anzeige des darin gespeicherten Inhalts

Pre-order Traversierung

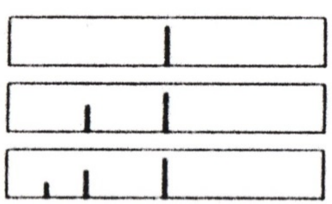
- Der Knoten k wird bearbeitet (hier: `visit` mit Ausgabe des Inhalts) bevor seine beiden Unterbäume bearbeitet werden
- Implementierung in Java

```
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“; Teil VIII)



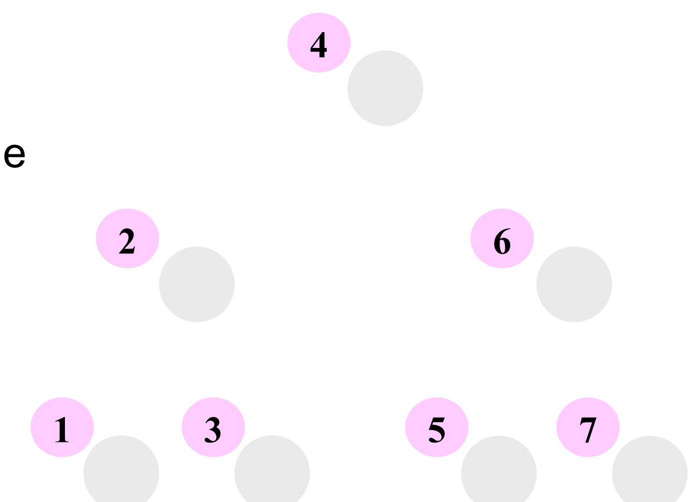
...

In-order Traversierung

- Der Knoten k wird <u>zwischen</u> der Bearbeitung seiner beiden Unterbäume bearbeitet
- Implementierung in <u>Java</u>

```
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(...)`
- Die Markierung eines Lineals mit *In-order* Traversierung würde in einer anderen Reihenfolge erfolgen (entsprechen von links nach rechts …):

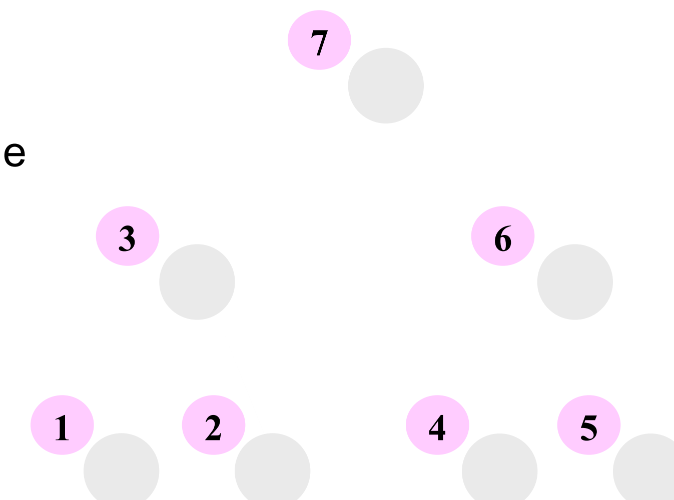


Post-order Traversierung

- Der Knoten k wird nach der Bearbeitung seiner beiden Unterbäume bearbeitet
- Implementierung in Java

```
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(...)`



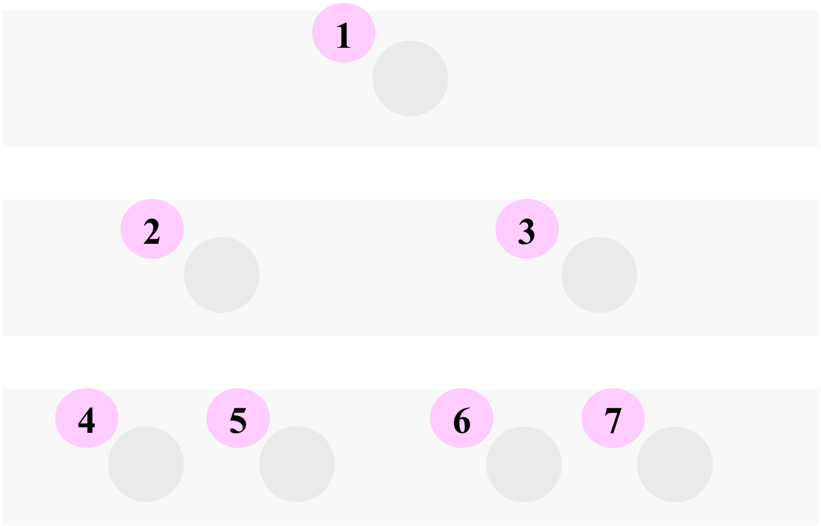
- Die Markierung eines Lineals mit *Post-order* Traversierung würde in einer anderen Reihenfolge erfolgen (entsprechen von links nach rechts …):

Breitendurchlauf in Binärbäumen

- Beim Breitendurchlauf wird der Baum ebenenweise durchlaufen

  Hinweis: Bei der vorgestellten Repräsentation von Bäumen ist ein Breitendurchlauf nicht einfach dadurch realisierbar, dass man rekursiv dem Schema der Datenstruktur folgt; es wird eine zusätzliche Queue benötigt, um die als nächstes zu bearbeitenden Knoten zu speichern (Queueelemente speichern einen Zeiger auf Knoten des Baums)

- Gewünschte Reihenfolge der Durchläufe durch die einzelnen Ebenen des binären Baums



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
- Rekursive Implementierung in Java

```
Node searchBTree(char content) {
    return searchBTree(content, root);
}

Node searchBTree(char content, Node n) {
    if (n == null)
        return null;  // Element nicht vorhanden
    if (n.item == content)
        return n;     // Element gefunden
    else {
        Node 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

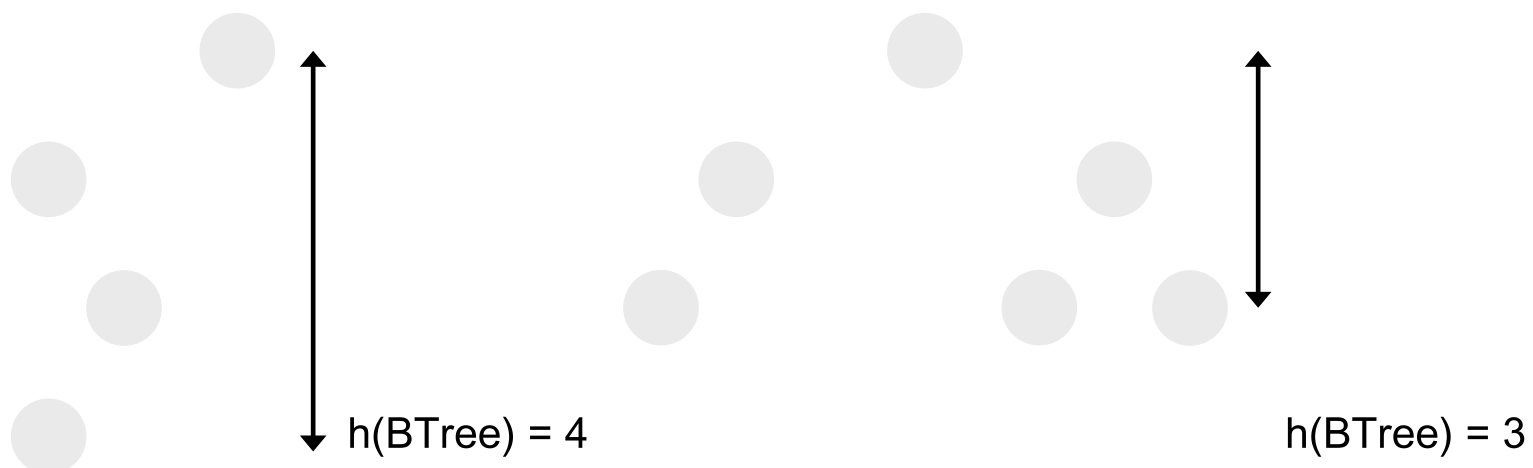
Eigenschaften von Binärbäumen

Tiefe eines Binärbaums

- Zur Erinnerung; Die Tiefe (Höhe) eines Baums ist die max. Länge eines Pfades von der Wurzel zu einem Blatt (vgl. S.65)
- Definition der Höhe h(•) für Binärbäume

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

Bsp.:



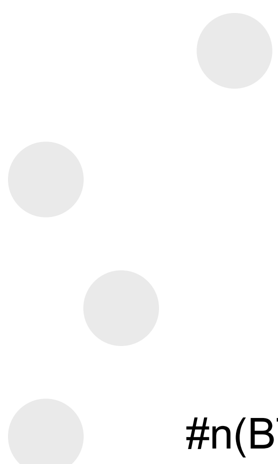
Anzahl der Knoten eines Binärbaums

Berechnung der Anzahl der Knoten eines Binärbaums

- Schema zur Berechnung von #n(•)

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

Bsp.: Anzahl der Knoten in einem gegebenen Baum



#n(BTree) = 4

#n(BTree) = 6

Extremale von Binärbäumen

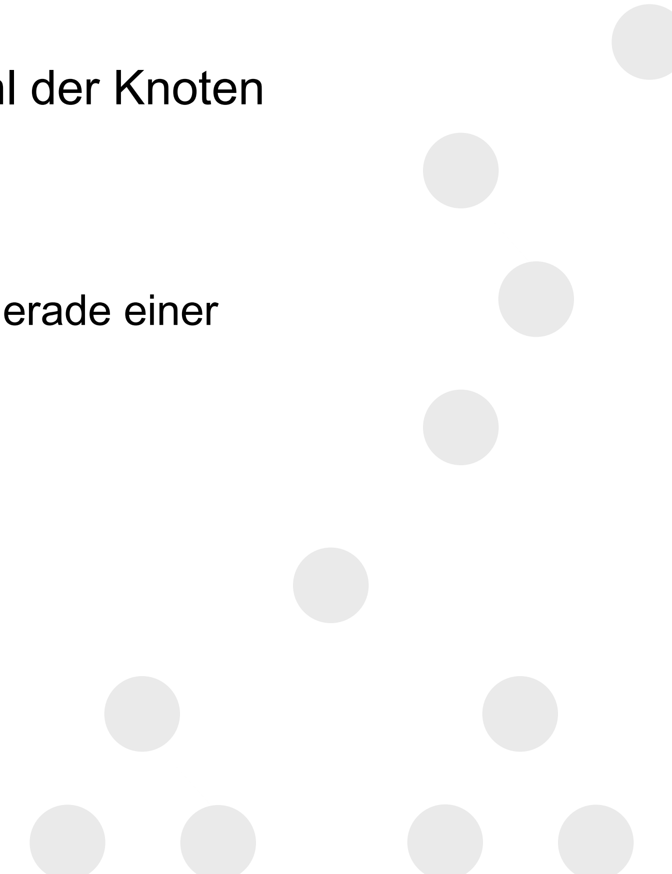
- Ist ein Binärbaum minimal besetzt, ist die Anzahl der Knoten gleich der Höhe des Baums

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

  Anmerkung: Ein minimal besetzter Baum entspricht gerade einer linearen Liste

- Ist ein Binärbaum voll besetzt, so gilt

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



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}

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

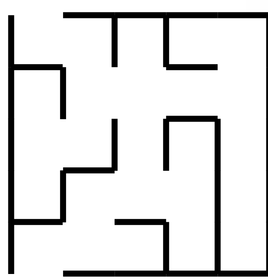
4. Graphen

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

Motivation und Einordnung

Allgemeine Einordnung

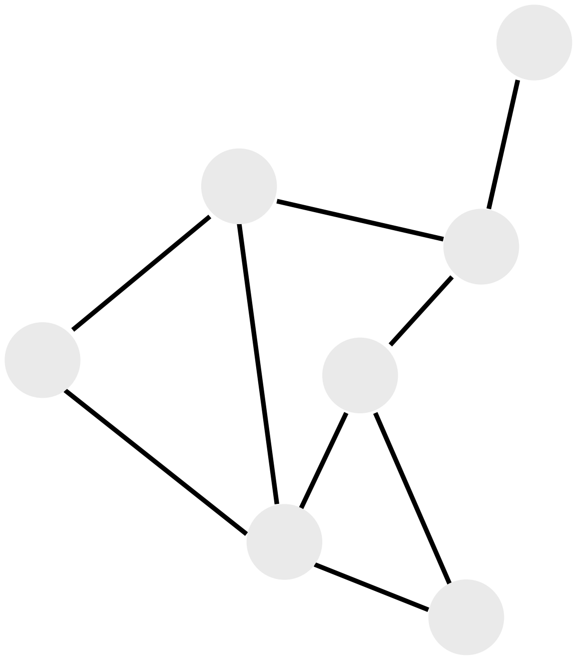
- Bei vielen Aufgabenstellungen und Lösungsverfahren (Algorithmen, Datenstrukturen) hat ein Element mehrere gleichwertige Nachbarbeziehungen (z. B. mehrere Vorgänger und mehrere Nachfolger)
- Beispiele sind …
  - Repräsentation von Nachbarschaftsbeziehungen
  - Straßen- oder Netz-Karten (vgl. auch Wege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



Betrachtungen 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



Hinweis: Wie bei den Bäumen enthalten die Knoten die zu speichernden Elemente (die Knoten werden als Rechtecke oder Ellipsen/Kreise dargestellt); die Kanten repräsentieren Relationen zwischen den Knoten, die Kanten können ungerichtet oder gerichtet sein

Definitionen und Eigenschaften

Struktur ungerichteter Graphen

- **Definition (ungerichtete Graphen):**

  Ein **ungerichteter Graph** G (mit $n \in \mathbb{N}$ Knoten und $k \in \mathbb{N}$ Kanten) ist ein Paar

  $$G = (V, E),$$

  mit einer n-elementigen **Knoten-Menge** *(vertex set)*

  $$V = \{v_i \mid i \in \{1, \ldots, n\}\}$$

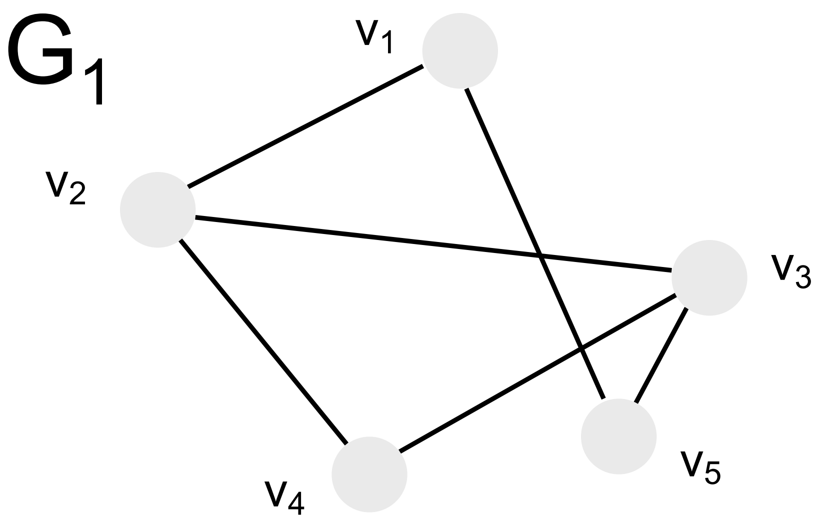
  und einer k-elementigen **Kanten-Menge** *(edge set)*

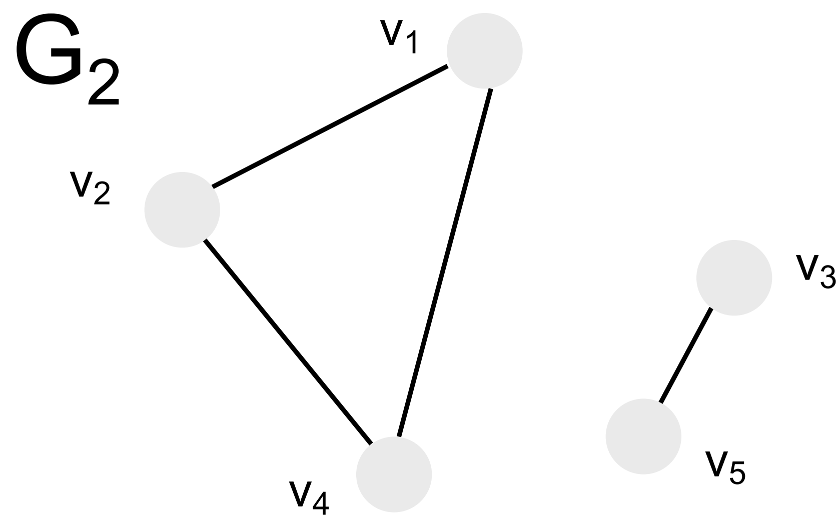
  $$E = \{e_i \mid i \in \{1, \ldots, k\}\},$$

  die zweielementige Teilmengen von V enthält: $e_i = \{u^{(i)}, w^{(i)}\}$ ist eine **Kante** zwischen den Knoten $u^{(i)} \in V$ und $w^{(i)} \in V$. Die Kanten-Menge E entspricht somit einer irreflexiven, symmetrischen Relation auf der Knoten-Menge V.

- Ein an eine Kante e angrenzender Knoten v heißt „mit e inzident“

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





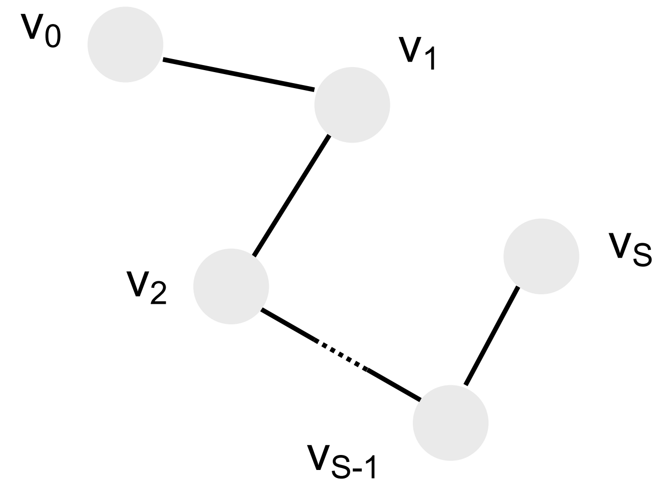
Zusammenhängender Graph:

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

Nicht zusammenhängender Graph:

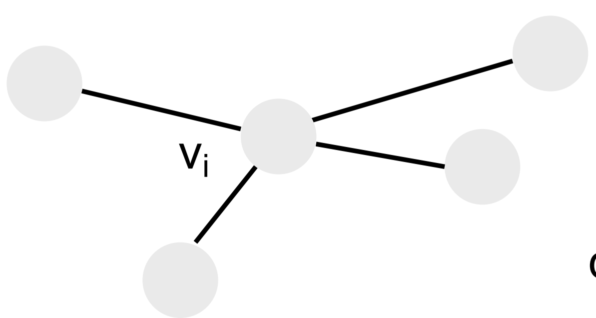
Beispielsweise kann Knoten v_3 nicht von Knoten v_2 erreicht werden, da keine Kante existiert, die die beiden Knotenmengen $\{v_1, v_2, v_4\}$ und $\{v_3, v_5\}$ verbindet

- **Ein** **Weg** im Graphen G = (V, E) ist eine endliche Folge $v_0, v_1, \ldots, v_S$ von Knoten mit $\{v_{i-1}, v_i\} \in E$ für alle i mit $1 \leq i \leq S$.



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

$$\deg_G(v_i) = \mathrm{card}\{\, j \mid \{v_i, v_j\} \in E\}$$



$\deg_G(v_i) = 4$

Struktur gerichteter Graphen

- **Definition (gerichtete Graphen)**:

Ein **gerichteter Graph** G ist ein Paar

$$G = (V, E)$$

mit einer n-elementigen **Knoten-Menge** ($n \in \mathbb{N}$)

$$V = \{v_i \mid i \in \{1, \ldots, n\}\}$$

und einer **Kanten-Menge**

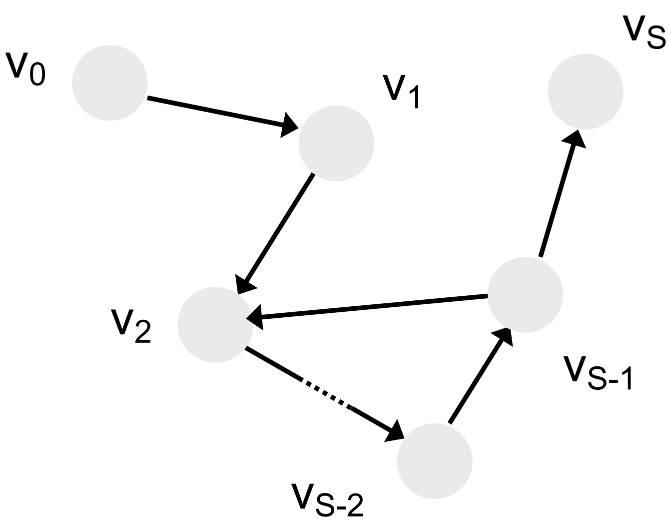
$$E \subseteq V \times V \,.$$

Jedes Element $e_i = (u^{(i)}, w^{(i)})$ von E stellt eine Kante von einem Knoten $u^{(i)} \in V$ zu einem Knoten $w^{(i)} \in V$ dar. Eine Kante $(v_i, v_j) \in E$ notieren wir auch kurz durch $v_i v_j$.

<u>Anmerkung</u>: Die Definition entspricht derjenigen von ungerichteten Graphen mit dem Unterschied, dass die Verbindungen (Relationen) zwischen den Knoten nicht symmetrisch sind

- **Grad eines Knotens** v_i in einem gerichteten Graphen G
  - **Eingangsgrad** von v_i: Anzahl der gerichteten Kanten $e \in E$ mit End-Knoten v_i: $deg_G^+(v_i) = card\{ j \mid v_j v_i \in E\}$
  - **Ausgangsgrad** von v_i: Anzahl der gerichteten Kanten $e \in E$ mit Anfangs-Knoten v_i: $deg_G^-(v_i) = card\{ j \mid v_i v_j \in E\}$

- **Weg oder Pfad** K eines Graphen G: Folge von Knoten $v_0, v_1, v_2, \ldots, v_S$, so dass $(v_{i-1}, v_i) \in E$ für alle i mit $1 \leq i \leq S$

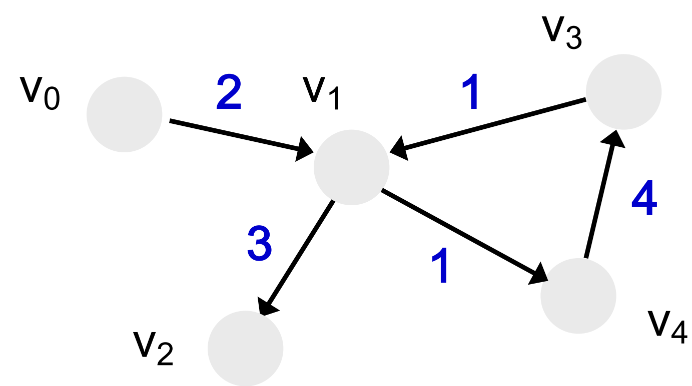


Anmerkung: In einem Weg können Zyklen auftreten (hier: $v_2, \ldots, v_{S-2}, v_{S-1}, v_2$)

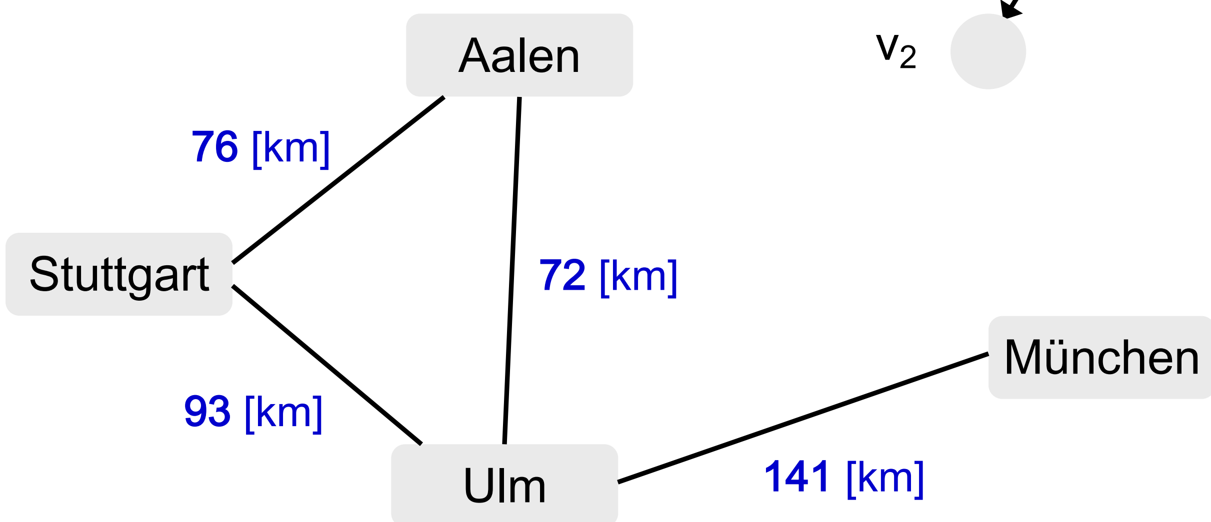
- Ein **gewichteter Graph** ist ein Tripel (V, E, w), wobei
  - das Paar (V, E) ein
    - ungerichteter Graph G oder
    - ein gerichteter Graph G

  ist und

  - w eine Abbildung $w: E \rightarrow M$ definiert, mit (zum Beispiel) $M = \mathbb{N}_0$ oder $M = \mathbb{R}$



Bsp.: Städteverbindungen



Repräsentation von Graphen

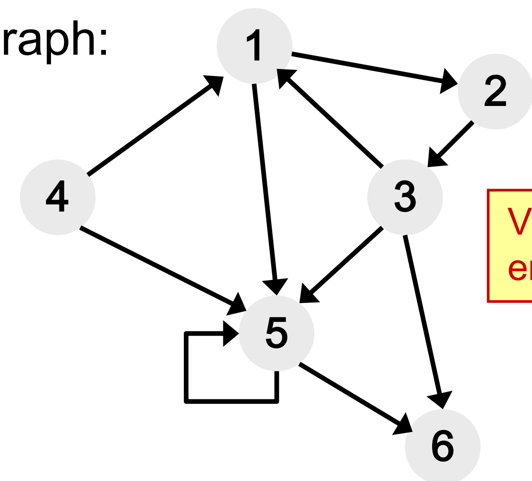
Adjazenz-Matrizen

- Adjazenz-Matrix eines ungerichteten Graphen $G = (V, E)$ mit $V = \{v_i \mid i = 1, 2, \ldots, n\}$ und Kantenmenge E:
  - symmetrische $(n \times n)$-Matrix $A = [a_{ij}]$
  - Eintrag $a_{ij} = a_{ji}$ ist jeweils das Gewicht der Kante zwischen Knoten v_i und v_j (bei ungewichteten Graphen ist $a_{ij} \in \{0, 1\}$)

- Adjazenz-Matrix eines gerichteten Graphen $G = (V, E)$ mit $V = \{v_i \mid i = 1, 2, \ldots, n\}$ und $E \subseteq V \times V$:
  - i. A. asymmetrische $(n \times n)$-Matrix $A = [a_{ij}]$
  - Eintrag a_{ij} ist jeweils das Gewicht der Kante von v_i nach v_j (bei ungewichteten Graphen ist $a_{ij} \in \{0, 1\}$)

- Beispiel: Gerichteter Graph G

Graph:



Adjazenz-Matrix:

j →

i ↓

	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

Von einem Knoten erreichbare Nachbar-Knoten

- Implementierung in <u>Java</u>

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

<u>**Einordnung**</u> und <u>**Bewertung**</u>:

- Für binäre (ungewichtete) Kanten kann die Matrix auch bool'sch deklariert werden:

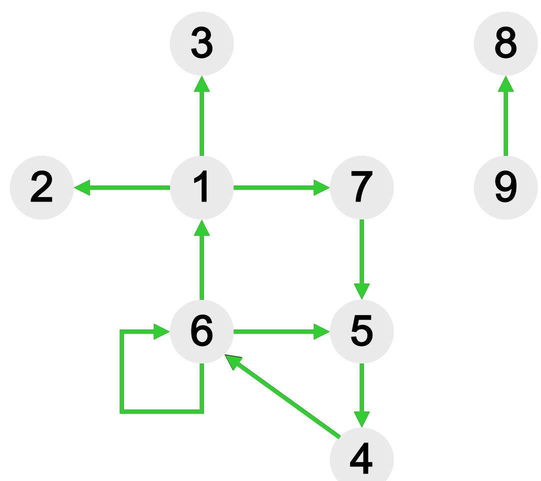
```
boolean[][] adjM = new boolean[noOfNodes][noOfNodes];
```

- Die Repräsentation ist für große Graphen mit wenigen (spärlichen) Kanten ineffizient, da viele nicht existierende Verbindungen $a_{ij} = 0$ repräsentiert werden

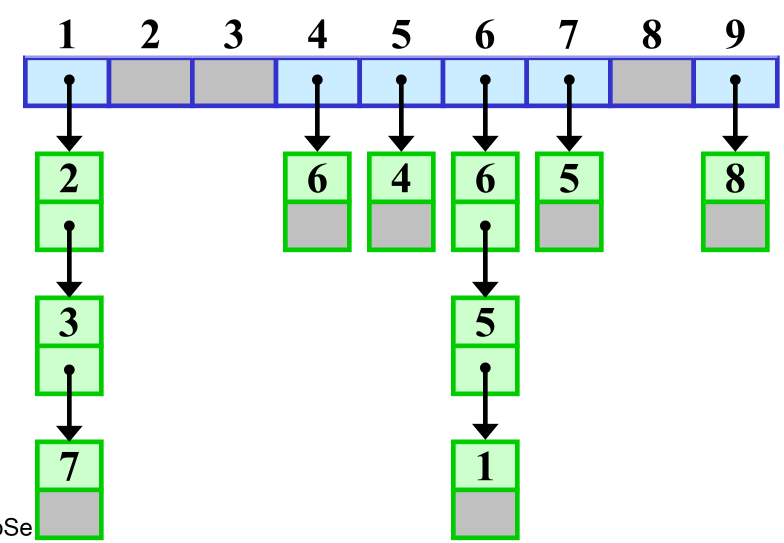
Adjazenz-Listen

- 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** verwendet mit den (a) von diesem Knoten ausgehenden Kanten (gerichtete Graphen) oder (b) mit diesem Knoten verbundenen Knoten (ungerichtete Graphen)
- Beispiel für Graph G

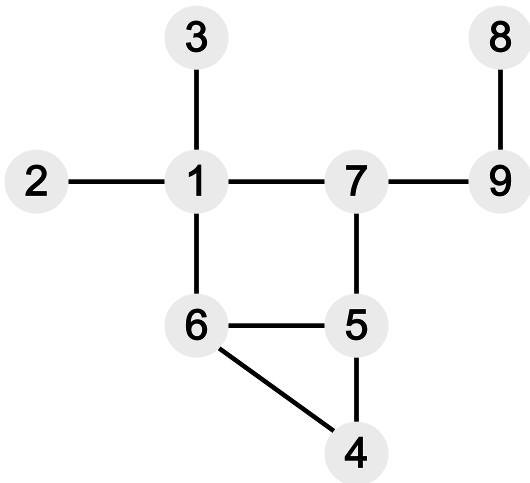
Gerichteter Graph G:



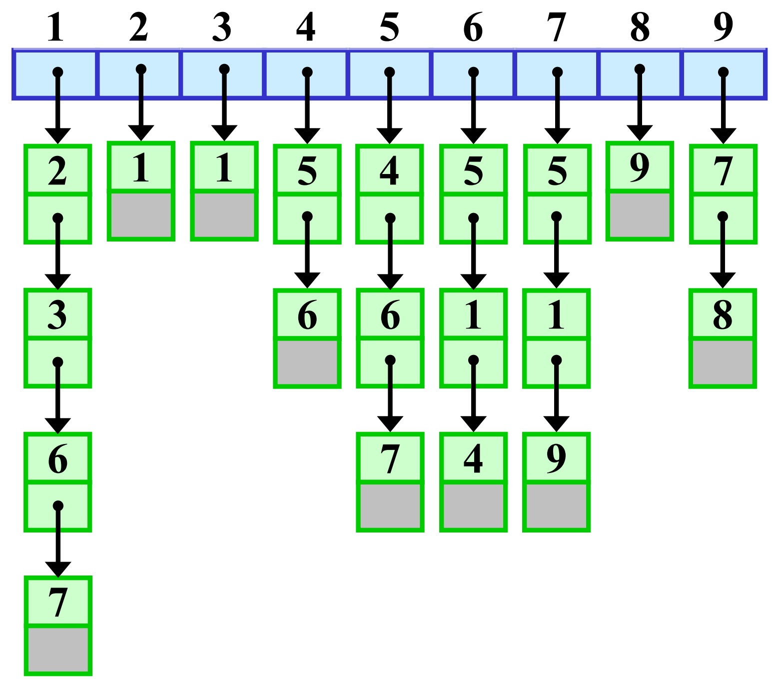
Adjazenz-Liste:



Ungerichteter Graph G:



Adjazenz-Liste:



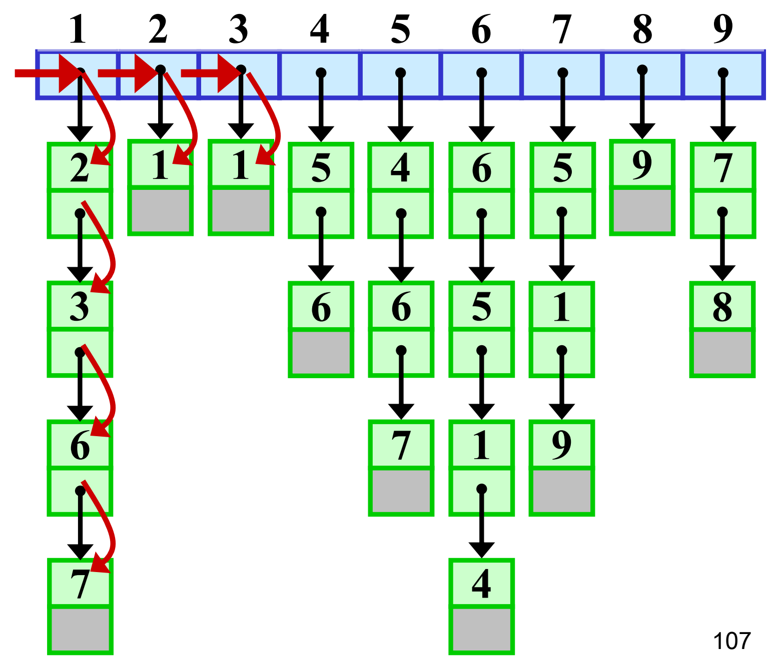
- Eigenschaften und Aufwand:
  - Speicheraufwand für **gerichtete Graphen**: Summe der Länge aller Adjazenz-Listen zu den Knoten V ist card{**E**}
  - Speicheraufwand für **ungerichtete Graphen**: Summe der Länge aller Adjazenz-Listen ist 2 • card{E}
  - Speicheraufwand insgesamt: O(|**V**| • |**E**|)

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 z.B. 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;

  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

- <u>Kommentar</u>:
  - Aus der $(n \times n)$-Matrix wird eine $(n+1) \times (n+1)$-Matrix generiert; die bisherigen Einträge der n Zeilen und Spalten werden übertragen
  - Eintragen der (neuen) Verbindungen:

    Bei ungerichteten Graphen für jeden Knoten $x \in$ {Nachbarschafts-Knoten}:
    `adjMnew[n][x] = 1; adjMnew[x][n] = 1;`

    Bei gerichteten Graphen:
    - Für jeden Knoten $x \in$ {Nachfolge-Knoten-Knoten}: `adjMnew[n][x] = 1`
    - Für jeden Knoten $x \in$ {Vorgänger-Knoten}: `adjMnew[x][n] = 1`

Repräsentation mit Adjazenz-Listen

- Wesentliche Schritte:
  - Anhängen eines Elements an das 1-dimensionale *Array* der Knoten
  - Aufbau einer Liste mit Elementen für die Nachbarschafts-Knoten (entsprechende Vorgehensweise für die Ausgangs-Knoten bei gerichteten Graphen)
  - 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)
- <u>Bewertung</u>: Einfügen von Knoten ist einfacher als bei Adjazenz-Matrizen, da weitestgehend die Eigenschaften von Listen zum Einfügen neuer Elemente ausgenutzt werden können

Implementierung in Java (Fragment für Repräsentation mit Adjazenz-Liste)
(Demo: Vertex.java, Graph.java, basierend auf Listen ListElem.java, List.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 Graph (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 item, ListElem next) {
        this.item = item;
        this.next = next;
    }
}  // end class ListElem
```

- Repräsentation von Listen-Objekten

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

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

    List(Vertex item) {
        head = new ListElem(item, null);
        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
```

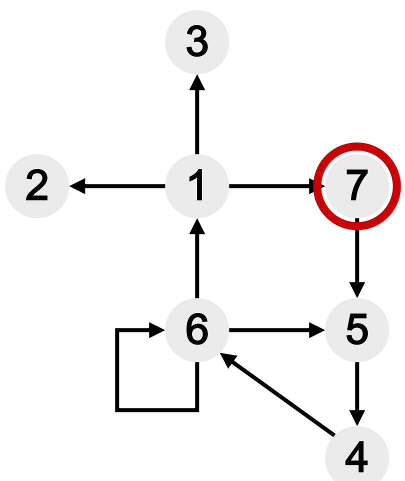
Löschen eines Elements

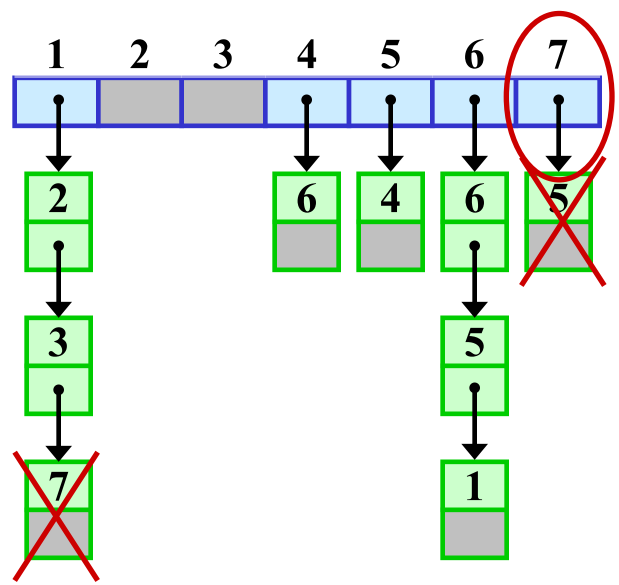
Allgemeine Vorgehensweise (hier für Adjazenz-Listen)

- 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)

- Beispiel – gerichteter Graph G und Adjazenz-Liste als Repräsentation

  Aufgabe: Lösche Knoten 7





Implementierung in Java – Methode `deleteVertex(...)`

- 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
  - Dann wird die Knoten-Liste durchlaufen und in der jeweils zugehörigen Kanten-Liste eine eventuell vorhandene Referenz auf den zu löschenden Knoten entfernt

Such-Algorithmen auf Graphen

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 **zusammenhängend** 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, ***traveling 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

Allgemeine Einordnung

- 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:
  - Tiefensuche
  - Breitensuche

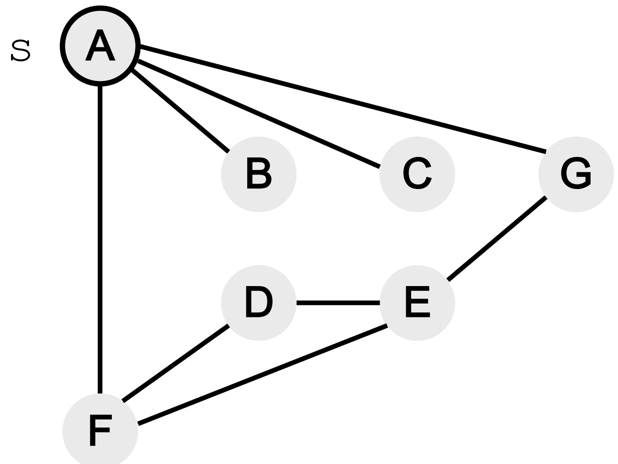
(Rekursiver) Algorithmus der Tiefensuche

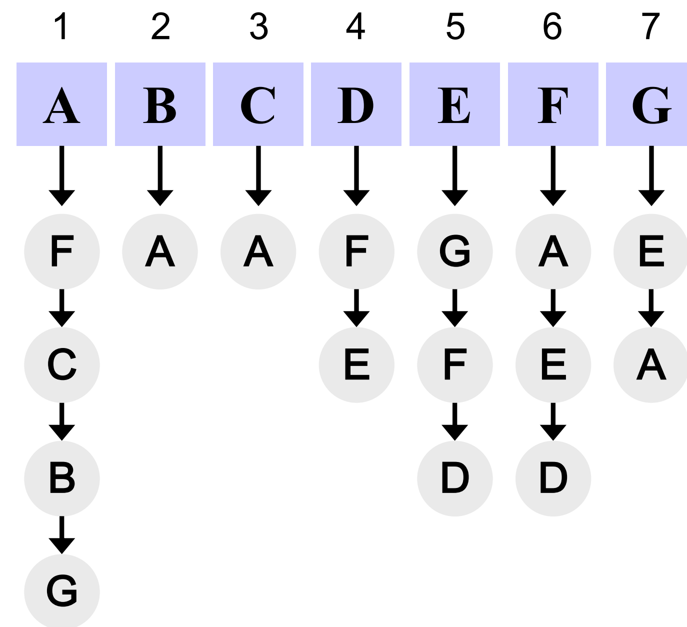
- Von einem (Start-) Knoten `s ∈ G` geht man entlang jeweils einer der bestehenden Kanten und markiert (*coloring*, `visit(•)`) den besuchten Knoten `v`
- Trifft man auf bereits besuchte Knoten, wird dieser Weg nicht weiter exploriert und statt dessen eine alternative Kante `e ∈ nb(v)` zu einem anderen 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

Kommentar: Die Vorgehensweise entspricht einer Tiefensuche in (Binär-) Bäumen

Beispiel für einen gegebenen ungerichteten Graph

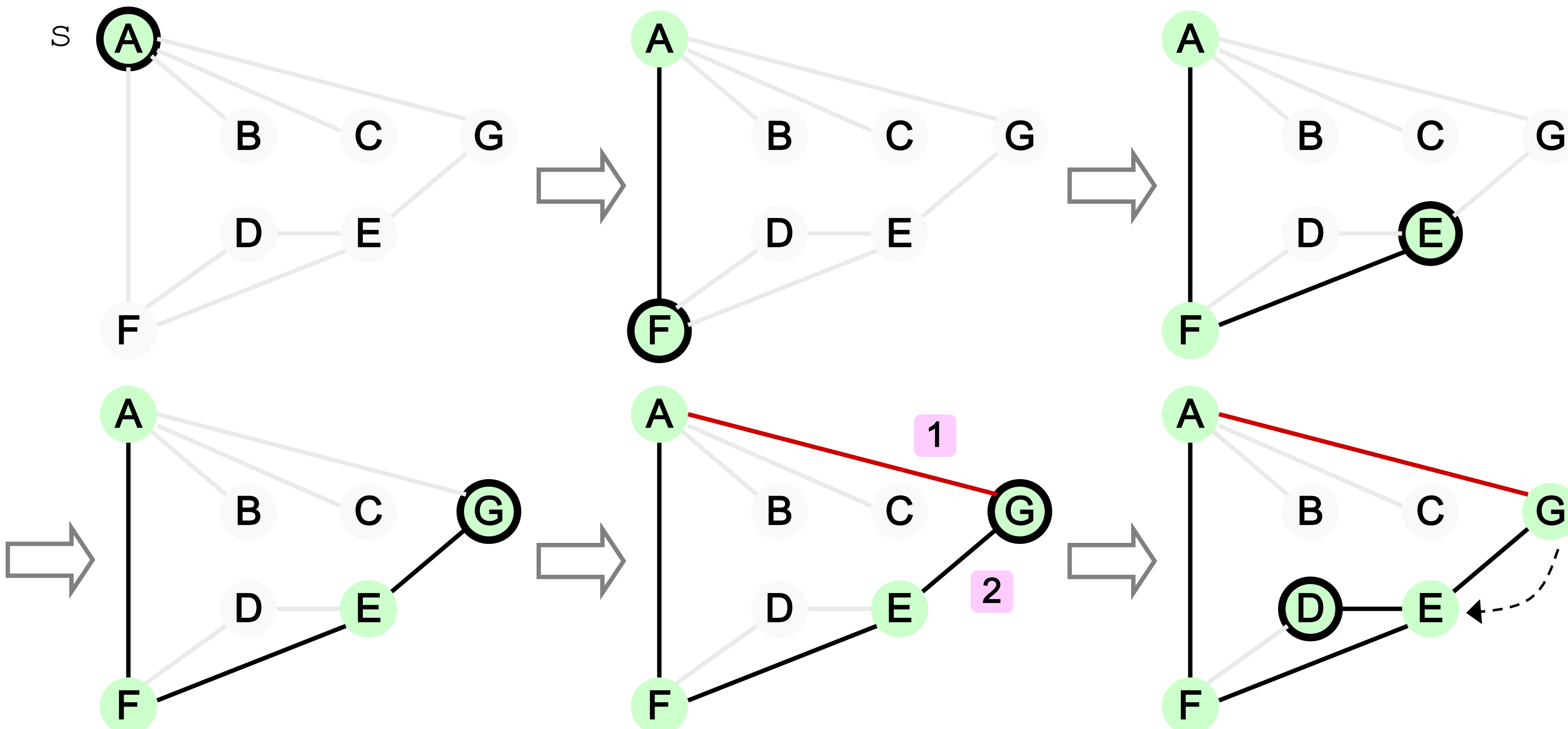
- Für einen Graph G(V, E) gibt es einen ausgezeichneten Knoten `s` $\in$ G
- Graph und seine Repräsentation (Adjazenz-Liste)

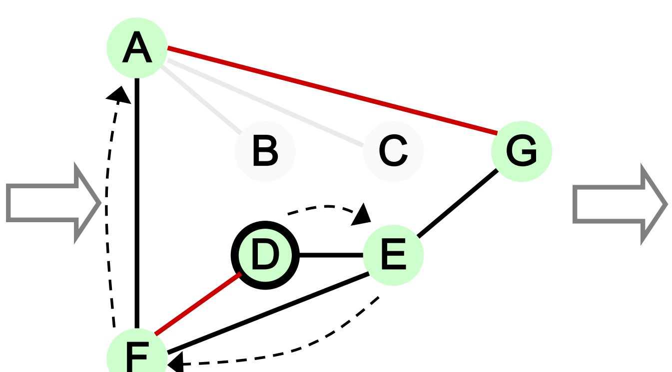




- Die Strategie der Tiefensuche läuft von einem Knoten ausgehend (beginnend mit `s`) entlang einer von ihm ausgehenden Kanten $e_1 = v_{aktuell}v_{NB}$, es wird überprüft, ob der nächste Knoten bereits besucht wurde; wenn nicht, dann laufe eine Verbindung e_2 zum nächsten Knoten, usw.
- __Ziel__: Systematisches Absuchen aller Kanten `v` des Graphen; es wird ein Feld (*Array*) `val` (oder Liste) der Länge |V| ≡ card(V) gefüllt, so dass für den k-ten Knoten **z. B.** der Wert `val[k] = min(dist(s))` gesetzt wird, für k = 1, 2, |V|

- (Tiefen-) Exploration der Knoten





…

- Besuchte Knoten in Grün; aktueller Knoten mit **fettem Rand**
- Erstmals gelaufene Kanten mit unbesuchtem Knoten in <u>schwarz</u>; ist der Zielknoten besucht, dann rot
- Wenn alle Kanten an einem Knoten besucht wurden, dann zurücklaufen (<u>gestrichelte Pfeile</u>)

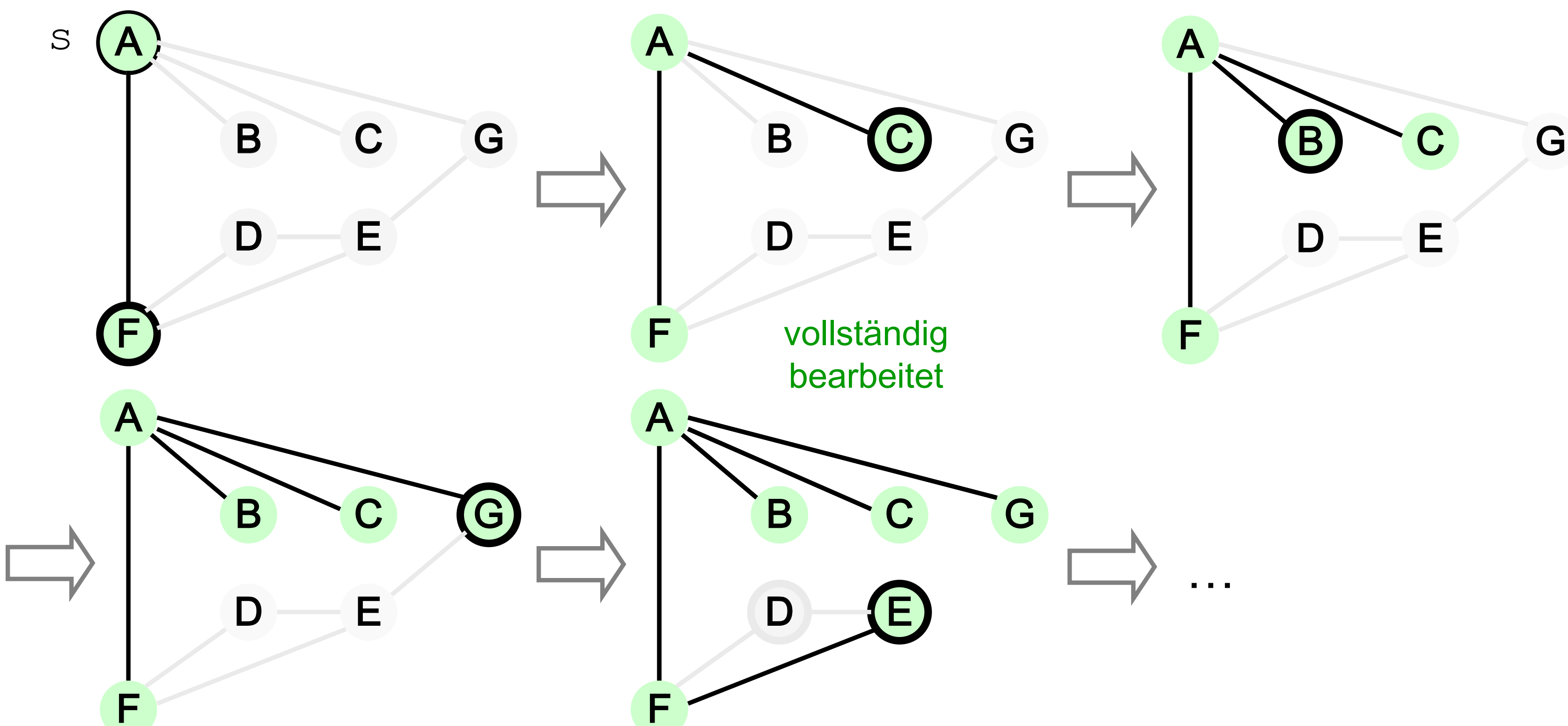
- Implementierung in Java: Methode `depthFirstSearch(...)`

```
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
```

Kommentar: Die Methode nutzt die in der Klasse deklarierte Aufzählung {WHITE, GREY, BLACK} zur Markierung besuchter Knoten

Breitensuche (*breadth-first search*) in einem Graphen

- Beginnend von einem Start-Knoten werden zunächst alle direkt verbundenen Knoten besucht, bevor zur nächsttieferen Ebene gegangen wird
- Für denselben Beispiel-Graphen (S.118) wird die Exploration in Breitensuche diskutiert (Adjazenzliste zu A: F – C – B – G)
- (Breiten-) Exploration der Knoten (Anfang wie bei Tiefensuche)



- Implementierung in <u>Java</u>: Methode `breadthFirstSearch(...)`

```
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
```

Praktische Anwendung von Algorithmen und Datenstrukturen

- Optimale Algorithmen in Praxis oft komplexer als Beispiele hier:
  - Beispiel A* Algorithmus:
    Wegesuche in Navigationssystemen mit Heuristiken
- Verständnis der Algorithmen wichtig,
  aber häufig keine eigene Implementierung notwendig oder sinnvoll
- Gängige Algorithmen oft in (Standard-) Bibliotheken gekapselt:
  - Beispiel: Java Collections Framework in java.util

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* on some collection of objects.
Deque<E>	A linear collection that supports element insertion and removal at both ends.
Enumeration<E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Formattable	The `Formattable` interface must be implemented by any class that needs to perform custom formatting using the `'s'` conversion specifier of **`Formatter`**.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
NavigableMap<K,V>	A **`SortedMap`** extended with navigation methods returning the closest matches for given search targets.
NavigableSet<E>	A **`SortedSet`** extended with navigation methods reporting closest matches for given search targets.

Templates / Java Generics

- Von welchem Datentyp sollen die Knoten der Datenstrukturen in einer Collection sein?
- Lösung: Templates bzw. Java Generics
  - Nutzung eines Typparameters T, der bei der Instantiierung mit konkretem Typ belegt wird
  - Der konkrete Typparameter T muss alle Operationen unterstützen, welche mit Objekten dieses Typs durchgeführt werden

```
public class GenericsExample<T> {

    private T item;

    public static void main(String[] args) {
        String item1 = "This is just a string";
        Integer item2 = new Integer(5);

        GenericsExample<String> ge1 = new GenericsExample<String>(item1);
        GenericsExample<Integer> ge2 = new GenericsExample<Integer>(item2);

        ge1.printItem();
        ge2.printItem();

        // Does not work: GenericsExample<Integer> ge3 = new GenericsExample<Integer>(item1);
    }

    public GenericsExample(T item) { this.item = item; }

    public void printItem() { System.out.println("Item Value: \"" + item.toString() + "\""); }

}
```

- Beispielprogramm für Collections mit ArrayList

```
import …;

public class ArrayListExample {

    public static void main(String args[]) {

        ArrayList<String> obj = new ArrayList<String>();

        obj.add("House");
        obj.add("Car");
        obj.add("Computer");
        obj.add("Speaker");
        obj.add("Smartphone");

        obj.add(0, "Ski");
        obj.add(1, "Telescope");

        obj.remove("Speaker");
        obj.remove("Telescope");

        obj.remove(1); // removes second(!) element from the list

        obj.sort(Comparator.reverseOrder());

        // Display elements
        System.out.println("\nReversely sorted ArrayList:");
        printCollection(obj);
    }

    public static void printCollection(Collection<String> c) {
        for (String str : c) {
            System.out.println(str);
        }
    }

}
```