

Kapitel 7: Dynamische Datenstrukturen

- 1. Informatik: Eine Übersicht
- 2. Logik (Einführung / Grundlagen)
- 3. Objektorientierte Modellierung
- 4. Algorithmen und ihre Eigenschaften
- 5. Entwurfsmethoden für Algorithmen
- 6. Sortieralgorithmen
- **7. Dynamische Datenstrukturen**
 - 7.1 Datenstrukturen → Dynamische Datenstrukturen
 - 7.2 Abstrakte Datentypen (ADT) zur Beschreibung
 - 7.3 Dynamische Datenstrukturen
 - Listen
 - Einfach und doppelt verkettet
 - Keller (stack)
 - Schlangen (queue)
 - Bäume
 - Binäre Bäume
 - Binäre Suchbäume

Vorab: Ein Hinweis zur Klausur

Hauptklausur vs. Nachklausur

- Klausur SS 2004 und Klausur WS 04/05 sind gleichwertig.
- Bitte zu der Klausur anmelden, die Sie auch schreiben wollen.
 - Anmeldung jetzt (SS 2004) hat nichts mit der Klausur im WS 04/05 zu tun.
- Weitere Infos zur Prüfungsanmeldung finden sich unter:

<http://www.aifb.uni-karlsruhe.de/StudiumUndPruefung/>

7.1 Datenstrukturen

Eine **Datenstruktur** ist eine bestimmte Art Daten im Speicher eines Computers anzuordnen und zu verwalten.

Eine Datenstruktur bietet mindestens Operationen an, um **neue Daten aufzunehmen** sowie **vorhandene Daten** zu **löschen** und zu **lesen**.

■ Beispiele für Datenstrukturen:

- Arrays (auch Felder, Elemente gleichen Datentyps)
 - eindimensional (Vektoren)
 - zweidimensional (Tabellen)
 - mehrdimensional
- Verbunde (Records, Units, Elemente unterschiedlichen Datentyps)
- **Listen**, meist verkettet
- Kellerspeicher (auch **Keller**, Stapel, Stack)
- Warteschlange (auch **Schlange**, Queue)
- **Bäume**
 - **Binärbaum**

Es gibt **statische** (=fixe Größe) und **dynamische** (=variable Größe) Datenstrukturen.

Motivation

Im Folgenden werden behandelt:

- Grundlegende **Konzepte** wichtiger **dynamischer Datenstrukturen**.
Diese gehören oftmals nicht zum Kernbestandteil einer Programmiersprachen.
- In der Regel werden diese dann selbst implementiert oder aus einer Bibliothek übernommen
 - Z.B. das *Java Collection Framework*
- Große Unterschiede in Laufzeit- und Speicherverhalten je nach Implementierung.
- → Wichtig ist dann ein grundlegendes Verständnis, so dass man weiß, wonach man in den Bibliotheken suchen muss und wie man die Datenstrukturen effizient anwendet.

Warum dynamische Datenstrukturen?

Eigenschaften von Feldern

(=statischen Datenstrukturen in Programmiersprachen):

- (1) Zusammenhängender Speicherbereich
- (2) **Größe des Speicherbereichs zur Laufzeit nicht veränderbar**
- (3) **Direktzugriff** auf Feldelemente über Speicheradresse
⇒ Laufzeit = $O(1)$

Simple **Problem**: Anzahl von zu speichernden Datenobjekten ist selten bekannt

Beispiel: Wieviele Anschriften sollen gespeichert werden?

(genauer: gleichzeitig im Speicher gehalten werden)

- Lösung I: max. 100 Objekte → reicht dem Anwender nicht
- Lösung II: max. 1000000 Objekte → Verschwendung von Speicher, Speicher reicht nicht

→ **Wunsch: Speicherverbrauch ungefähr proportional zur Menge der gespeicherten Daten**

Warum dynamische Datenstrukturen?

Neuer Ansatz:

- (1) Speicherbereich muss nicht mehr zusammenhängend sein
- (2) Definition von **zusätzlichen Verwaltungsstrukturen** (Knoten; *nodes*)
- (3) Knoten werden aneinandergehängt (mit **Referenzen**)

- → Definition von **verketteten Listen, Bäumen, ...**
- → **Zugriff nicht mehr direkt möglich → Laufzeit?**

Dynamische Datenstrukturen sind eine **Abstraktion über dem direkten Speichermodell einer Programmiersprache**. Sie werden mit Hilfe von statischen Datenstrukturen implementiert.

Was muss eine dyn. Datenstruktur leisten?

→ Hilfe bei der Datenverwaltung:

- Neue Daten aufnehmen → „**schreiben**“
- Alte Daten **löschen**
- Gespeicherte Daten nutzen, abrufen → „**lesen**“
- In den verwalteten Daten **suchen**
- Alle Objekte **ablaufen** (traversieren der Datenstruktur)
 - Verschiedene Reihenfolgen

7.2 Abstrakter Datentyp (ADT)

- Beschreibung einer komplexen Schnittstelle
- unabhängig von einer Programmiersprache, die Beschreibung kann in natürlicher Sprache abgefasst werden.
- beschreibt was die Operationen tun, aber nicht wie sie das tun
 - Prinzip der versteckten Information

Definition

Ein ADT besteht aus:

- Einer Menge M von Objekten
- Einer Menge O von Operationen auf dieser Menge
- Beschreibung der Semantik der Operationen

→ Beschreibt etwas komplexeres als ein einzelnes Objekt

→ **ADTs eignen sich gut um (dynamische) Datenstrukturen zu beschreiben.**

Abstrakter Datentyp (ADT)

Implementierung eines ADT z.B. in Java

- Als Menge M von Java-Klassen und Schnittstellen
- Diese besitzen gut dokumentierte Funktionen, welche nur primitive Datentypen und Typen aus M verwenden
- Wichtig: Saubere Trennung von bereitgestellten Operationen einer Datenstruktur und ihrer Implementierung.
- Große Unterschiede in Laufzeit- und Speicherverhalten je nach Implementierung.

Bespiel:

Im *Java Collection Framework* (seit Java 1.4 überarbeitet)

- Interfaces: z.B. Collection, List, Set, SortedSet, Iterator
- Implementierungen: z.B. LinkedList, Stack, TreeSet

Operationen dynamischer Datenstrukturen

Minimal:

- Objekt **schreiben** – teilweise unter Angabe eines Index
- Objekt **löschen** – um Speicher wieder frei zu geben
- Objekt **lesen** – versch. Zugriffsarten/Adressierungen

Häufige zusätzliche Operationen,

deren Zeitverhalten je nach Implementierung sehr unterschiedlich sein kann:

■ Suchen

eines Objektes mit einem bestimmten (oder ähnlichem) Wert

- Naiv: Alle Objekte durchgehen und jeweils vergleichen $\rightarrow O(n)$
- Besser: z.B. $O(\log n)$ im binären Suchbaum

■ Zwei Datenstrukturen a und b vereinigen

- Naiv:
Für jedes Element x in a: **b.insert(x)** $\rightarrow O(\text{Elemente in a})$
- Besser: Listen über Zeigeroperationen direkt aneinander hängen $\rightarrow O(1)$

7.3 Dynamische Datenstrukturen als ADT

Die Grundoperationen einer Dyn. Datenstruktur **D**:

- **add(Object o)**
 - o in D einfügen
- **remove(Object o)**
 - Entfernt o aus D, wenn es in **D** vorhanden ist
- **boolean contains(Object o)**
 - wahr \Leftrightarrow **D** enthält o (mind. einmal)

- **Ablaufen aller Elemente** (evtl. in bestimmter Reihenfolge)

Ziel: alle Knoten eines Baumes besuchen \Rightarrow Markieren, Kopieren, Ausdrucken, ... aller Knoten

Implementierung in Java:

- **Iterator iterator()** $\rightarrow O(n)$ für **Ablaufen von n El.**
 - liefert einen Iterator für **D**

Iterator:

- Hilfsobjekt zum sequentiellen Ablaufen von dyn. Datenstrukt.
- **Object next()** - liefert ein aktuelles, seit Erstellung des Iterators noch nicht abgelaufenes Element

Warum gibt es nicht *die* dyn. Datenstruktur?

→ unterschiedliche Anforderungen:

- Unterschiedliches **Laufzeitverhalten von Operationen**
 - Reihenfolge von Operationen spielt eine Rolle für die Ausführungsgeschwindigkeit

- Große Unterschiede in der **Adressierung der Daten**
 - **Über die Ordnung der Datenstruktur**
 - Erstes oder letztes Element in der Datenstruktur
 - Über genaue Angabe einer Position (Index)
 - **Über den Inhalt**
 - Suchen nach exaktem/ähnlichem Wert

- Sind **Doubletten** erlaubt?
 - Add(x), Add(x) gleich/ungleich Add(x)

- Ist die Datenstruktur **sortiert**?
 - Add(x), Add(y) gleich/ungleich Add(y), Add(x)
 - Das zuerst/zuletzt geschriebene → Schlange, Keller

Implementierung einer Dyn. Datenstruktur:

- Aufbau aus Verwaltungs- und Datenelementen
- Verbinden der Elemente z.B. über Zeiger
- Verbergen der internen Datenstrukturen durch eine wohldefinierte Schnittstelle

Auswahl dynamischer Datenstrukturen

- Abwägen zwischen
 - + Bessere Laufzeit im O-Kalkül für bestimmte Operationen
 - - zusätzlicher Mehraufwand für Verwaltungsstrukturen
 - → mehr Speicherbedarf,
 - → fehleranfälligere Implementierung,
 - → langsamere Abarbeitung wg. Verwaltung
- → Bessere Laufzeit im O-Kalkül ist in der Praxis nicht unbedingt beste Laufzeit
- **Verschiedene Operationen haben unterschiedliche Laufzeiten (praktisch und im O-Kalkül)**
- Reihenfolge der Operationen spielt eine Rolle
 - → u. U. zeitaufwändige interne Umorganisationen notwendig
- Auswahl einer dyn. Datenstruktur abhängig von
 - Erwartete **Datenmenge**?
 - Mit welcher **Häufigkeit/Wahrscheinlichkeit/Reihenfolge von Operationen** wird gerechnet?
 - Semantik der Daten
 - Sortiert nach Wert oder Reihenfolge?
 - Doubletten erlaubt?
 - **Adressierung der Daten** beim lesen und schreiben

Listen

Im Folgenden:

- Der abstrakte Datentyp [Liste](#)
- [Einfach verkettete Liste](#)
- [Doppelt verkettete Liste](#)

■ Literatur

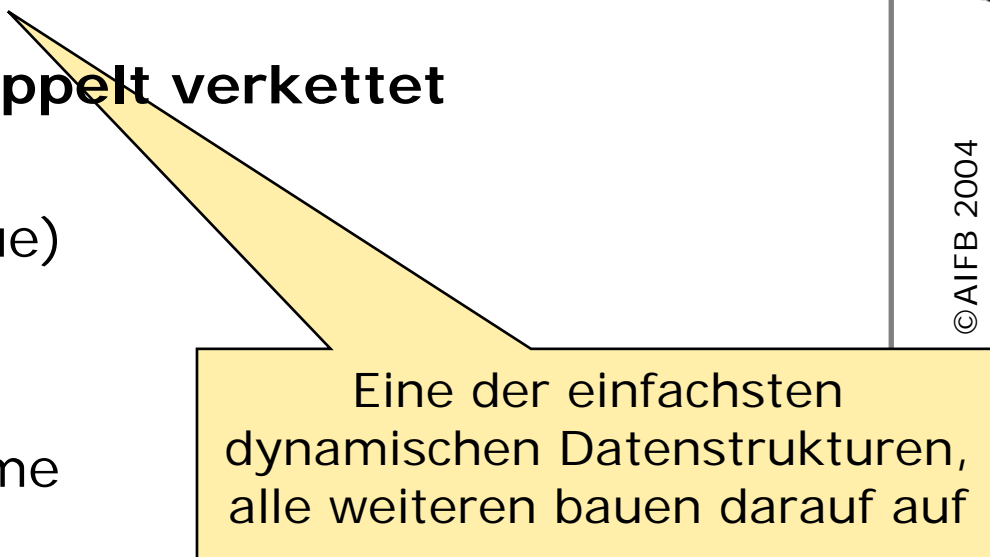
- Balzert, Helmut: Lehrbuch Grundlagen der Informatik; Spektrum (1999), Kap.3.7
- Sedgewick, Robert: Algorithmen; Addison-Wesley (1999), Kap. 3

■ Siehe auch:

Dipl.-Inform. G. Langemeier (Uni-Hildesheim), Programmierung spezieller Probleme in Java, <http://www.mathematik.uni-hildesheim.de/SS2002/Java/scripts.asp>, Folie 10

Gliederung

- Datenstrukturen → Dynamische Datenstrukturen
- Abstrakte Datentypen (ADT) zur Beschreibung
- **Dynamische Datenstrukturen**
 - **Listen**
 - **Einfach und doppelt verkettet**
 - Keller (stack)
 - Schlangen (queue)
 - **Bäume**
 - Binäre Bäume
 - Binäre Suchbäume



Eine der einfachsten dynamischen Datenstrukturen, alle weiteren bauen darauf auf

Der Abstrakte Datentyp Liste

Operationen:

■ Einfügen

- ... am Anfang:

```
insertAtHead( Object o )
```

- ... an Stelle *i*:

```
insertAt( int i, Object o )
```

■ Lesen

- Prüfen auf Enthaltensein

```
boolean contains(Object o)
```

- Erstes Element lesen

```
Object getFirst()
```

- An Stelle *i* lesen

```
Object get(int i)
```

■ Löschen:

- Erstes Element löschen

```
removeAtHead(),
```

- An Stelle *i* löschen

```
removeAt( int i )
```

Verkettete Listen: Definition

Eine **nichtleere verkettete Liste (VL)** ist eine **durch Referenz verkettete Folge von Elementen desselben Typs**, wobei gilt:

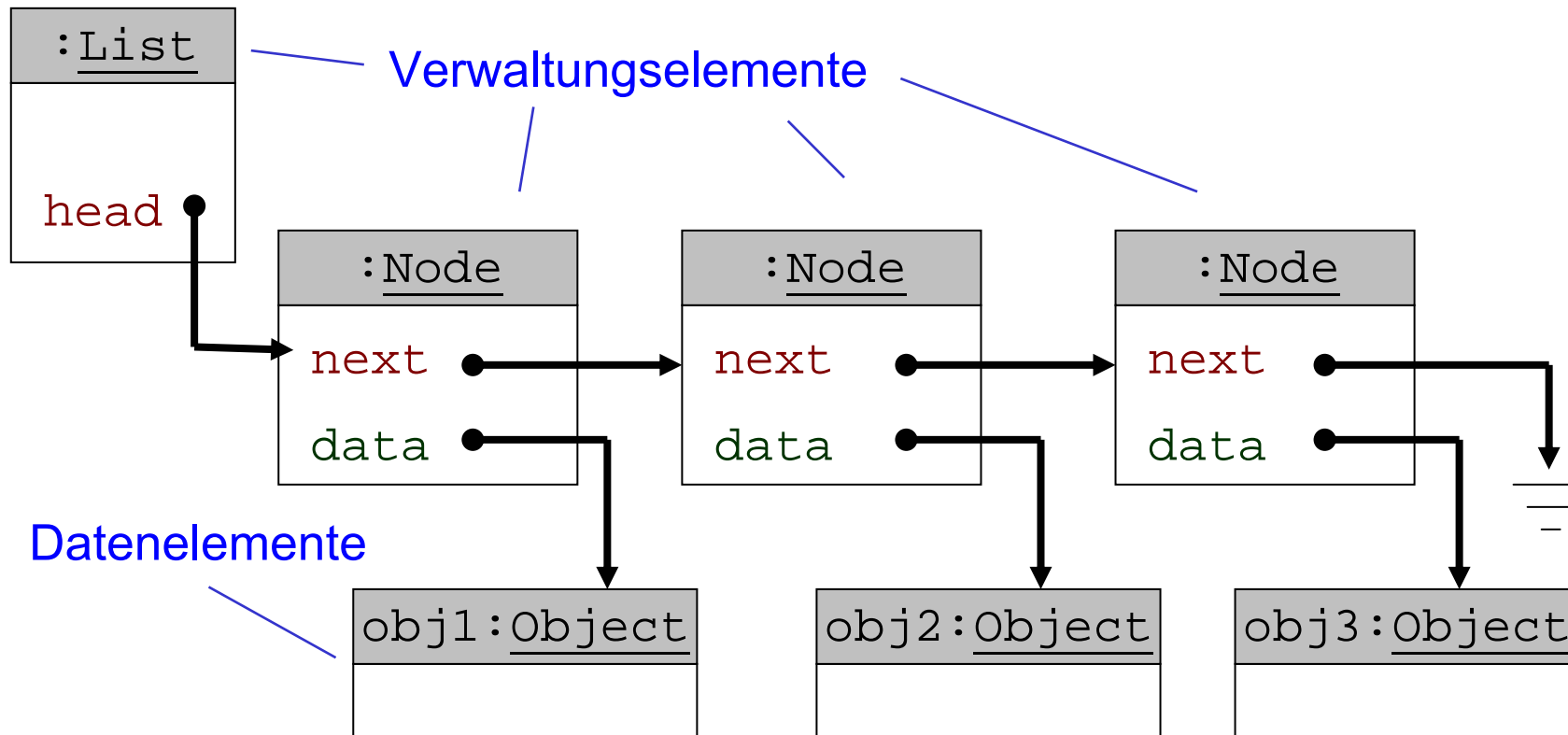
- Es gibt **genau ein Listenelement ohne Vorgänger** (erstes Listenelement).
- Es gibt **genau ein Listenelement ohne Nachfolger** (letztes Listenelement).
- Alle übrigen Listenelemente haben genau einen Vorgänger und genau einen Nachfolger.
- Eine leere VL enthält kein Listenelement.
 - Sie wird durch Hinzunahme eines Elements in eine nichtleere VL überführt.

Konsequenzen:

- | | |
|---|---------------------|
| ■ Kein Direktzugriff auf Elemente per Index | ⇒ Laufzeit = $O(n)$ |
| ■ Einfügen von Elementen | ⇒ Laufzeit = $O(n)$ |
| • Spezialfall: Einfügen am Anfang | ⇒ Laufzeit = $O(1)$ |
| ■ Löschen von Elementen | ⇒ Laufzeit = $O(n)$ |
| • Spezialfall: Löschen am Anfang | ⇒ Laufzeit = $O(1)$ |

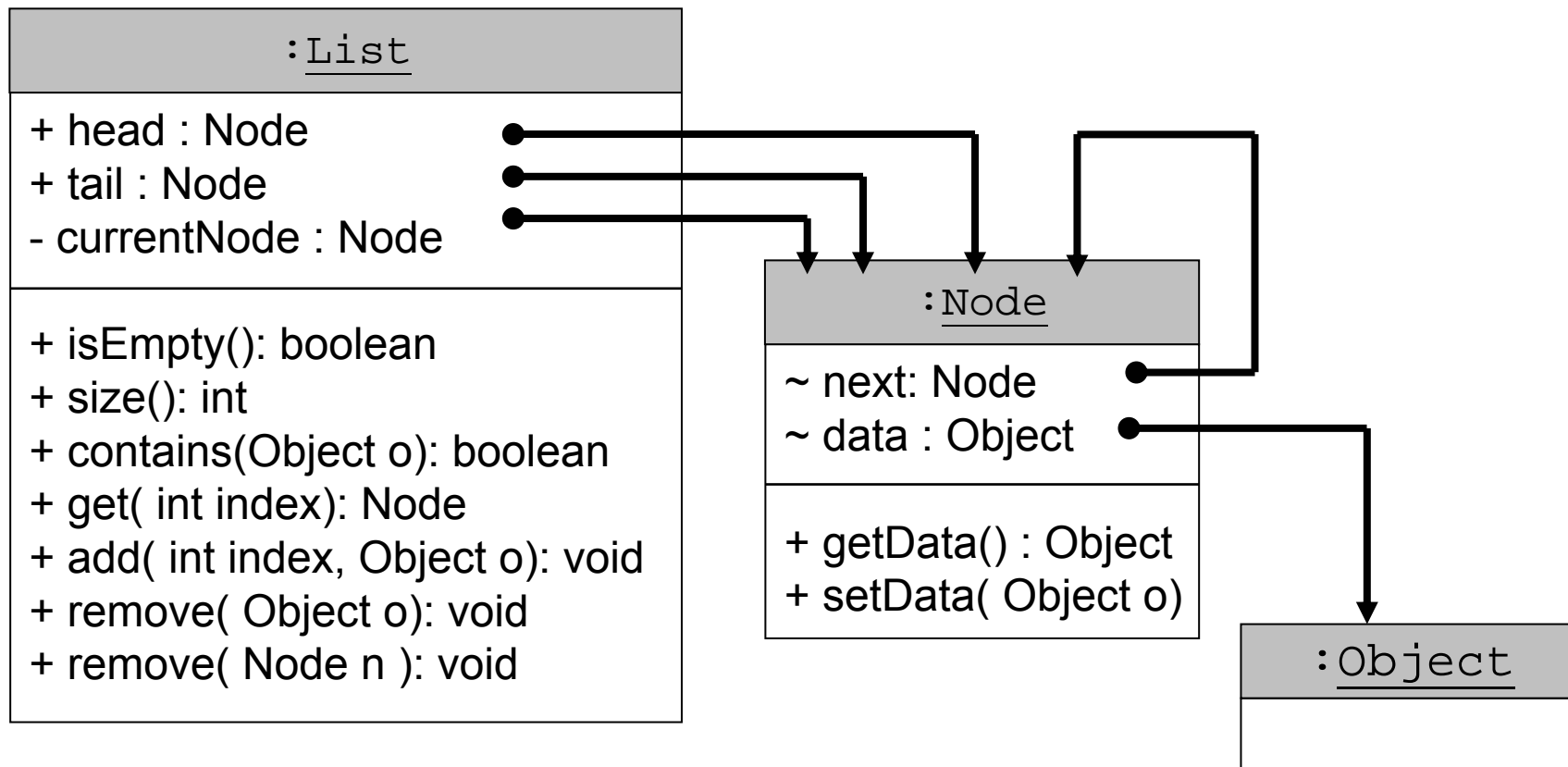
Liste: Aufbau

- Eine der einfachsten dynamischen Datenstrukturen, alle weiteren bauen darauf auf
- Aufbau der Liste aus zwei Sorten Verwaltungselementen:
 - **Knoten** (Nodes) als Verwaltungselemente kennen jeweils ihren Nachfolger und ein **Datenelement**
 - Ein **Listen-Element** kennt den ersten Knoten



Verkettete Listen: Implementierung

- Hier: Einfach verkettet, mit Zeiger auf das Ende
- Klassen:
 - List – Listenklasse, stellt Methoden bereit
 - Node – Verwaltungsknoten
 - Object – Datenobjekte



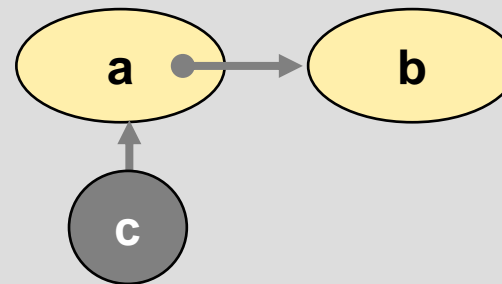
Generelles zur Implementierung

- Oft besondere Behandlung von **Sonderfällen**
 - Einfügen in **leere Datenstruktur**
 - Löschen aus **Datenstruktur mit keinem oder einem Element**
- Traversieren
 - Sonderfälle bei leeren Listen, Listen mit einem Element, am Listenende
- Techniken
 - Rekursive Algorithmen
 - Schleppzeigertechnik (Zeiger auf aktuelles Element und dessen Vorgänger)

Grafische Notation für die Folien:

a und **b** sind **Nodes**
c ein Zeiger auf einen **Node**
(in Java dasselbe)

a.next == **b**
b.next == null
c == **a**



Einfach verkettete Listen: Einfügen

Z.B. `add(int index, Object o)`

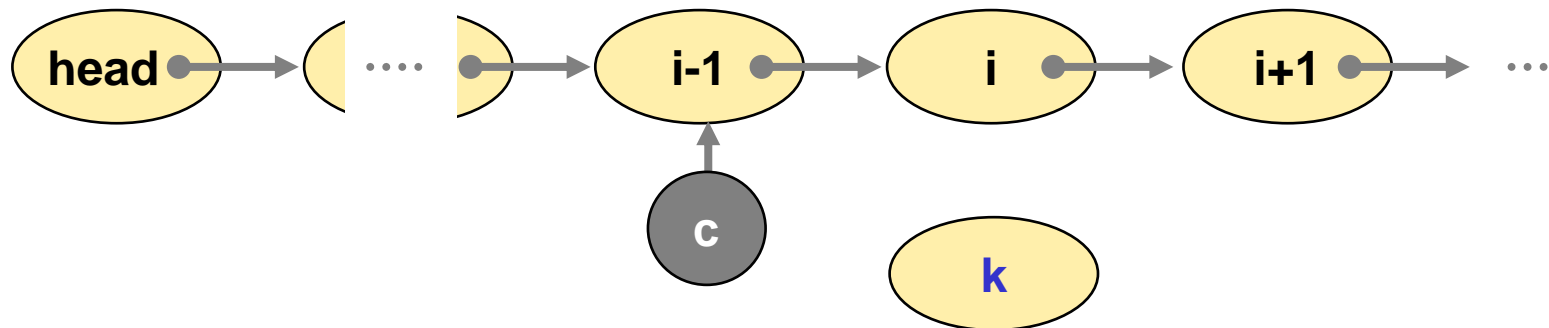
`o` an Stelle `index` einfügen

- Traversiere die Liste bis zum Knoten `index-1` mit Hilfe des Zeigers `c` (current)

- `c = head;`
 for (`int j=0; j < index-1; j++`)
 `c = c.next`

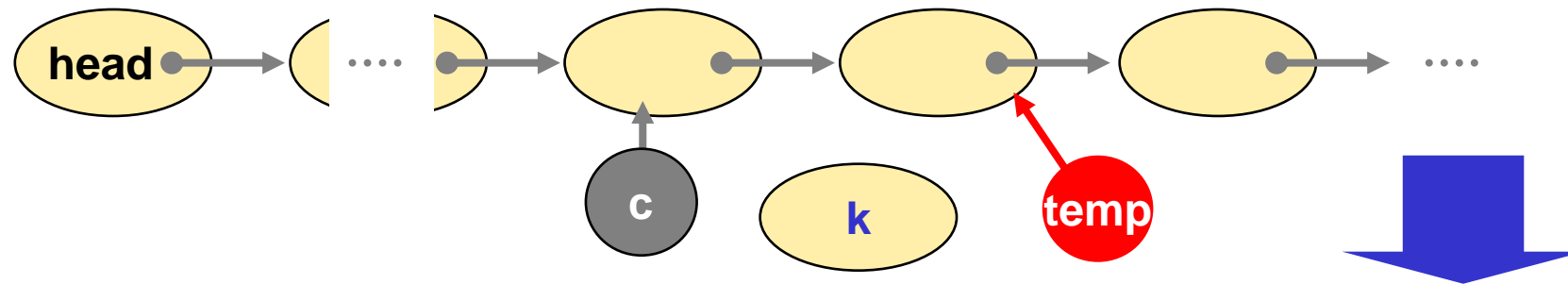
- Erzeuge einen neuen Knoten `k` mit `o` als Inhalt

- `Node k = new Node(o);`

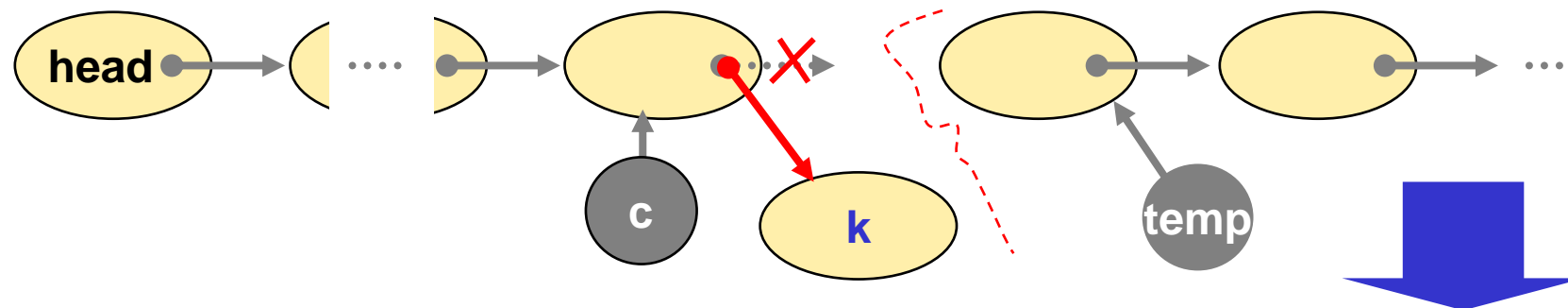


Einfach verkettete Listen: Einfügen

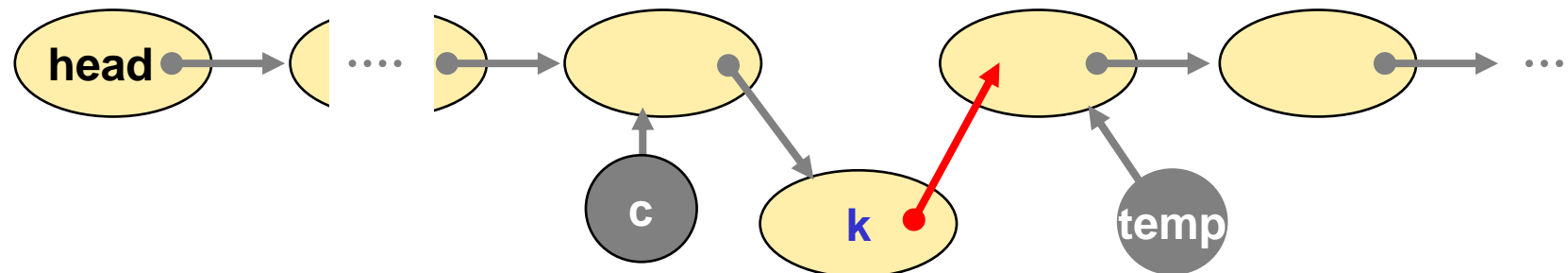
- (1) `Node temp = c.next;` // hinteren Listenteil merken



- (2) `c.next = k;` // Listenrest durch k ersetzen



- (3) `k.next = temp;` // hintere Liste wieder anhängen



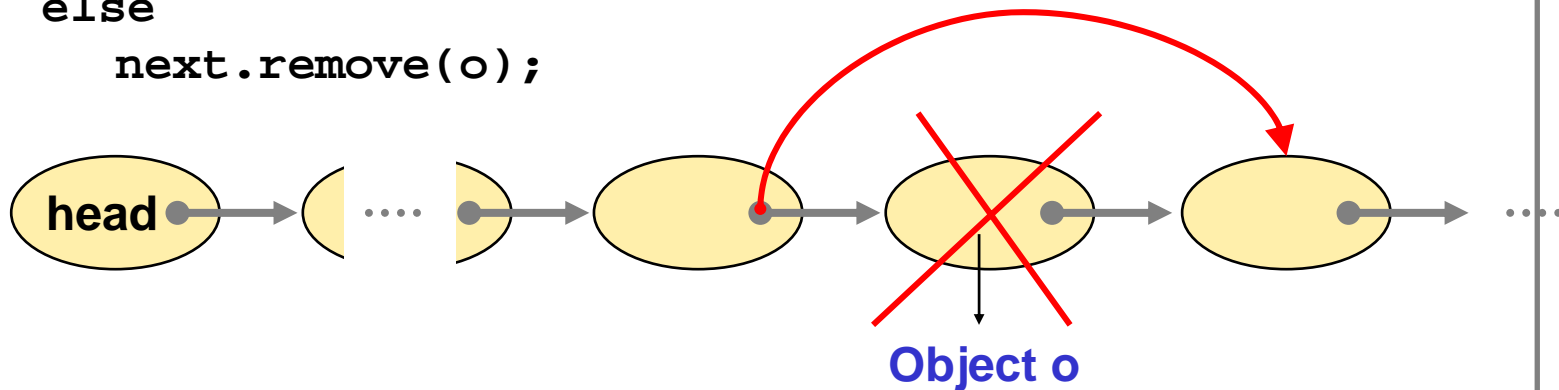
Einfach verkettete Listen: Löschen

Z.B. `remove(Object o)` - `o` aus der Liste löschen

- Rekursiver Algorithmus ohne Zeiger
 - Es gibt auch Varianten mit Zeiger → in der Praxis schneller

```
List.remove( Object o ) {
    if( head != null )
        head.remove( o );
}
```

```
Node.remove( Object o ) {
    if (next != null) {
        if( next.data.equals(o))           // o gefunden → löschen
            next = next.next;
        else
            next.remove(o);
    }
}
```



Mehr Aufwand \Rightarrow bessere Laufzeiten

Einfach verkettete Liste

- Einfügen in $O(n)$
- Nachfolger $O(1)$
- **Vorgänger $O(n)$**
 - **Jeweils von vorne suchen**

Doppelt verkettete Liste

- Einfügen in $O(n)$
- Nachfolger in $O(1)$
- **Vorgänger in $O(1)$**
 - \rightarrow **Zeiger auf Vorgänger „prev“**

(-) höherer Speicherbedarf

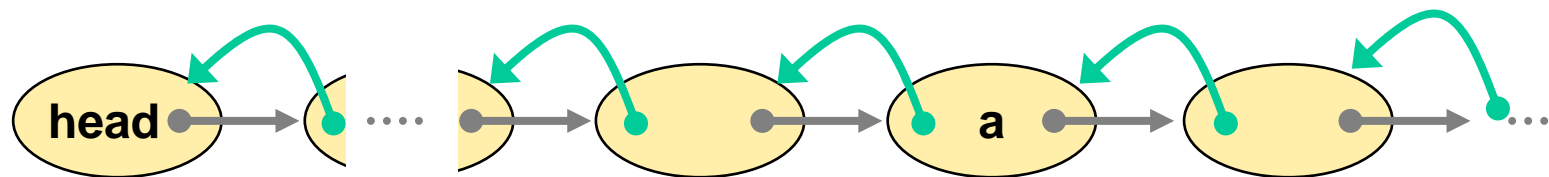
(+) Löschen von Knoten wird effizienter:

Löschen von a:

```
a.prev.next = a.next;
```

```
a.next.prev = a.prev;
```

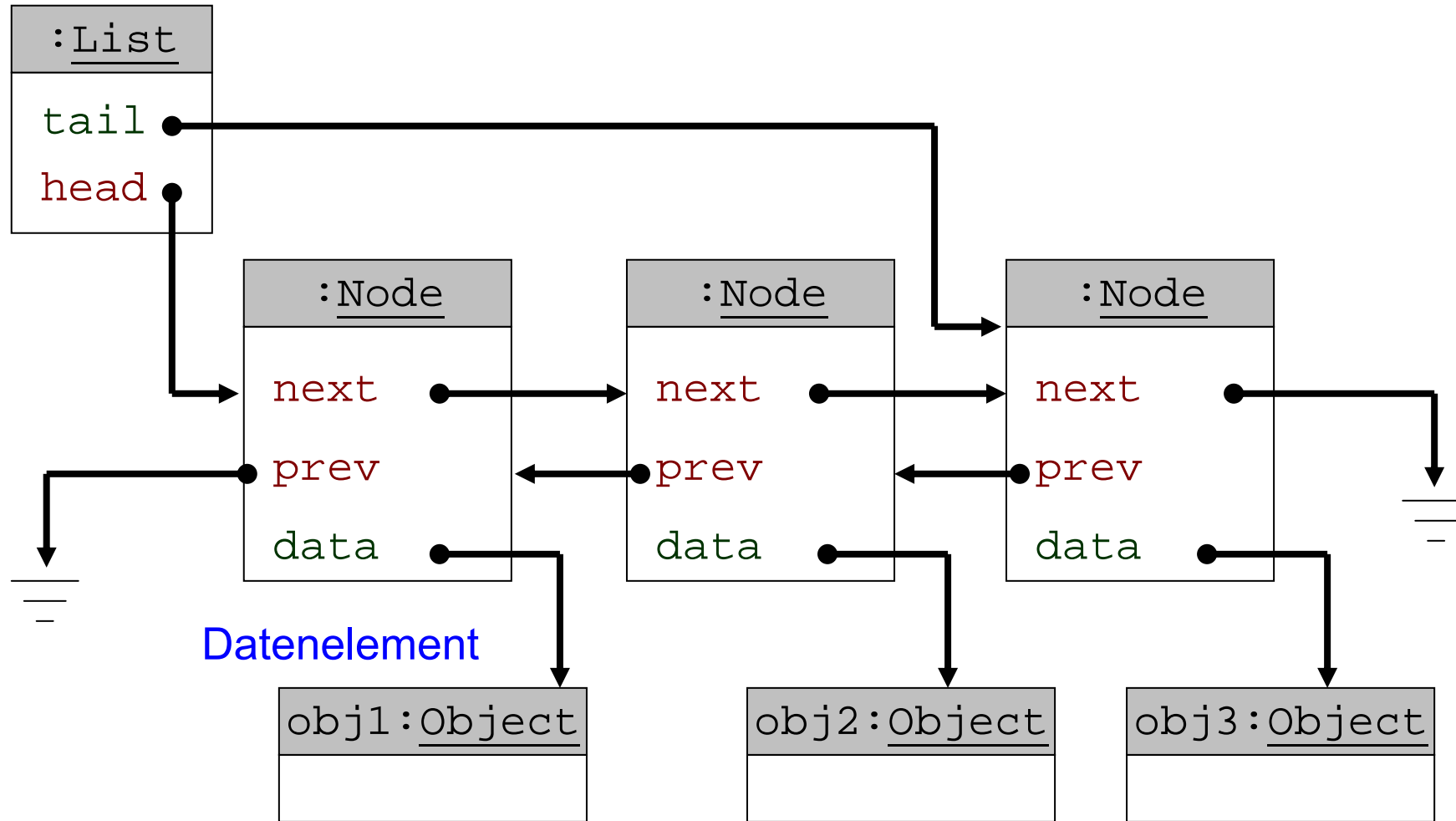
\rightarrow keine Traversierung der Liste notwendig



Doppelt verkettete Listen: Aufbau

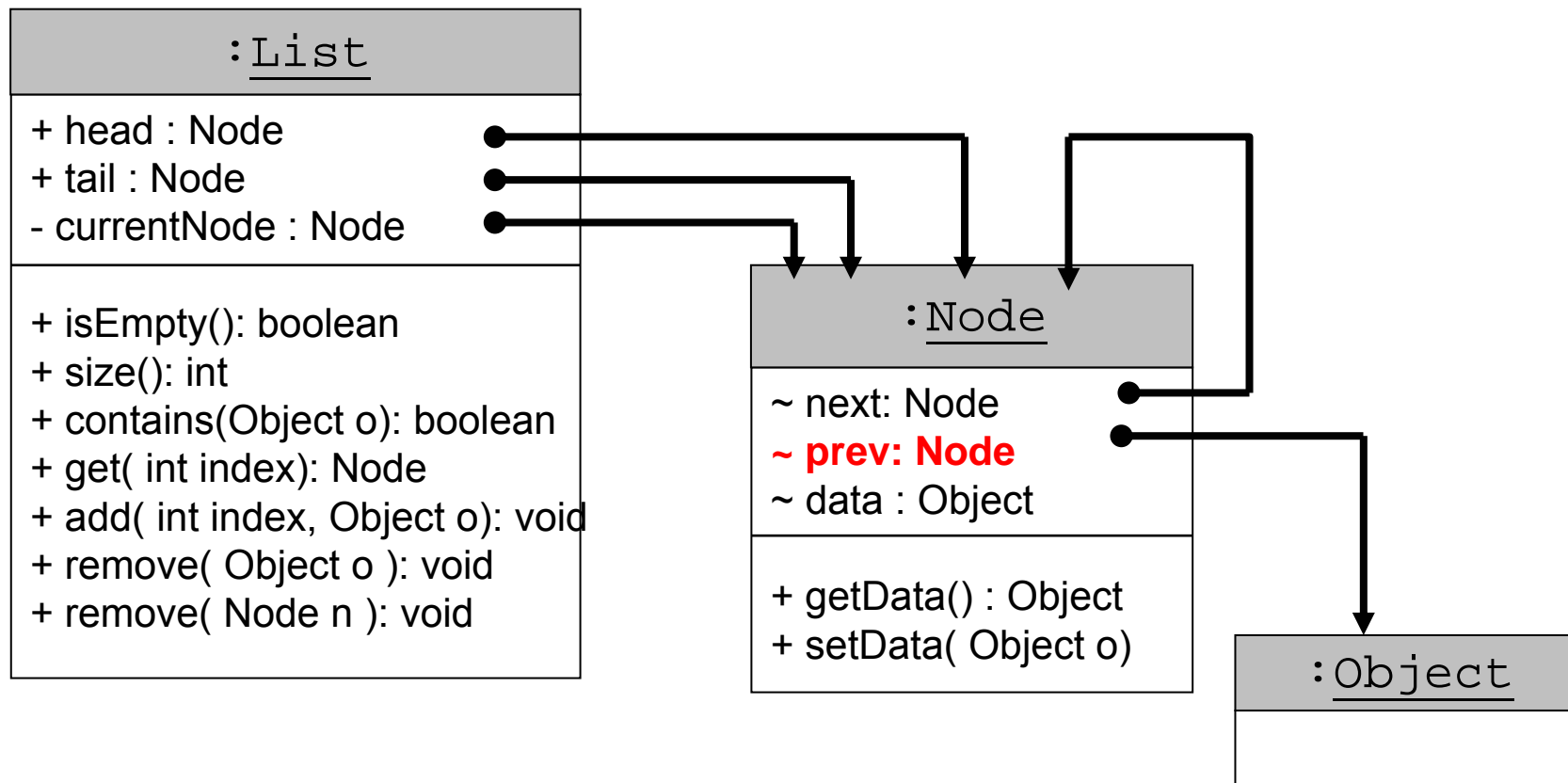
Hier: mit Zeiger auf das letzte Element

Verwaltungselement



Doppel verkettete Liste: Implementierung

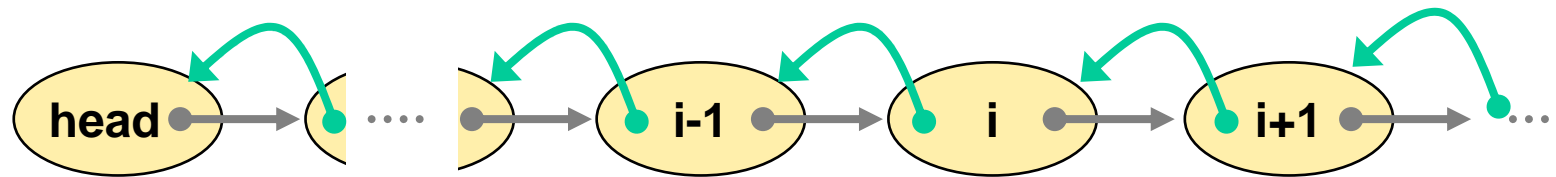
- Hier: **Doppelt** verkettet, mit Zeiger auf das Ende
- Klassen:
 - List – Listenklasse, stellt Methoden bereit
 - Node - Verwaltungsknoten
 - Object - Datenobjekte



Doppelt verkettete Listen: Einfügen

Z.B. `add(int index, Object o)` `o` an Stelle `index` einfügen

- Traversiere die Liste bis zum Knoten `index-1` mit Hilfe des Zeigers `c` (current)
- Erzeuge einen neuen Knoten `k` mit `o` als Inhalt



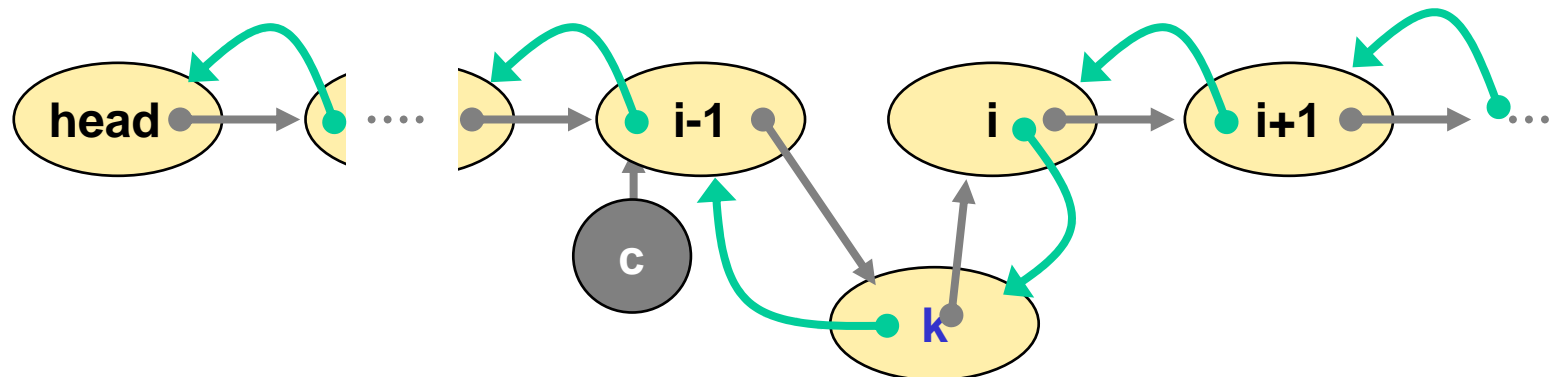
- Zeigeroperationen:

`c.next = k;`

`k.next = i;`

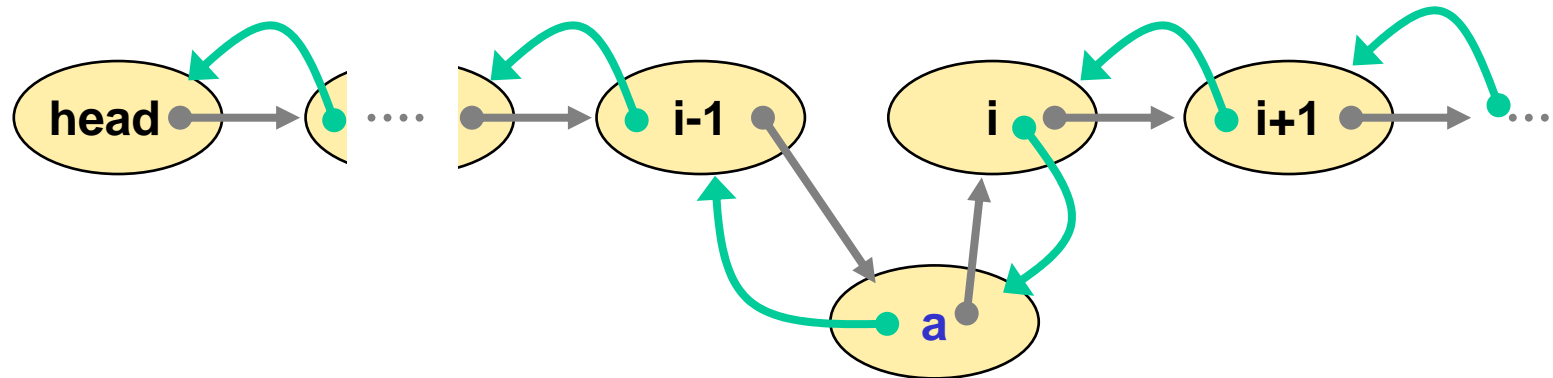
`k.prev = i-1;`

`i.prev = k;`



Doppelt verkettete Listen: Löschen

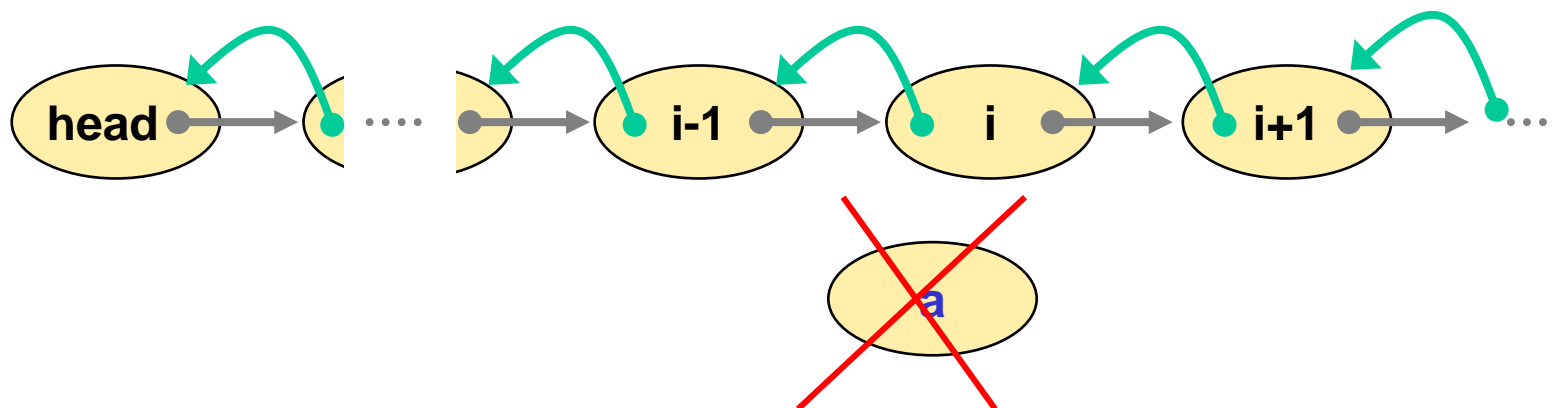
Z.B. `remove(Node a)` Knoten a löschen



■ Zeigeroperationen:

`a.prev.next = a.next;`

`a.next.prev = a.prev;`



Verkettete Liste in Java: `java.util.LinkedList`

Implementierung von `List`, doppelt verkettete Liste,
 Zeiger auf erstes und letztes Element.

Operationen:

Erzeugen

- `new LinkedList()` Erzeugen einer leeren Liste

Schreiben

- `add(Object o)` `o` in `D` einfügen
- `addFirst(Object o), addLast(Object o)`
- `add(int index, Object o)` `o` in `D` an Stelle *index* einfügen
- `set(int index, Object o)` setzt Datenelement der Stelle *index* auf `o`

Löschen

- `clear()`
- `remove(Object o)` Entfernt **ein** `o` aus Liste, sofern vorhanden
- `removeFirst(), removeLast()`
- `remove(int index)` Entfernt Objekt an Stelle *index*

Lesen und Suchen

- `boolean contains(Object o)` wahr ⇔ Liste enthält `o` (mind. einmal)
- `Object getFirst(), Object getLast()`
- `Object get(int index)`
- `size()` Anzahl der Elemente

Ablaufen der Elemente über `iterator()`

[Quelle: <http://java.sun.com/j2se/1.4.2/docs/api/java/util/LinkedList.html>]

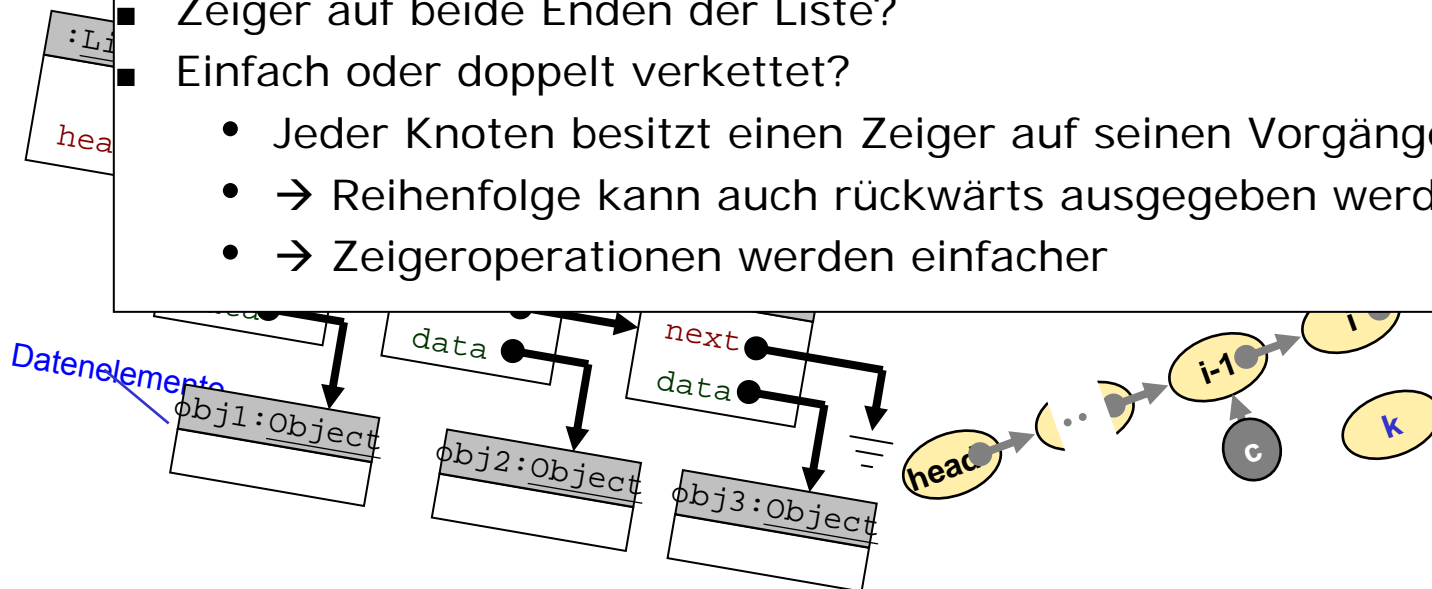
Verkettete Listen: Zusammenfassung

Effizienz:

- Einfügen, Löschen und Suchen
i. A. mit Aufwand $O(n)$
 - → Verkettete Listen eignen sich nicht gut, um Daten **nach Wert sortiert zu verwalten**
- Einfügen und Löschen
am Listenende in $O(1)$
 - → Gute Verwendung für Verwaltung nach Reihenfolge,
z.B. als FIFO (Keller) und LIFO (Schlange)

Verschiedene Typen von Listen

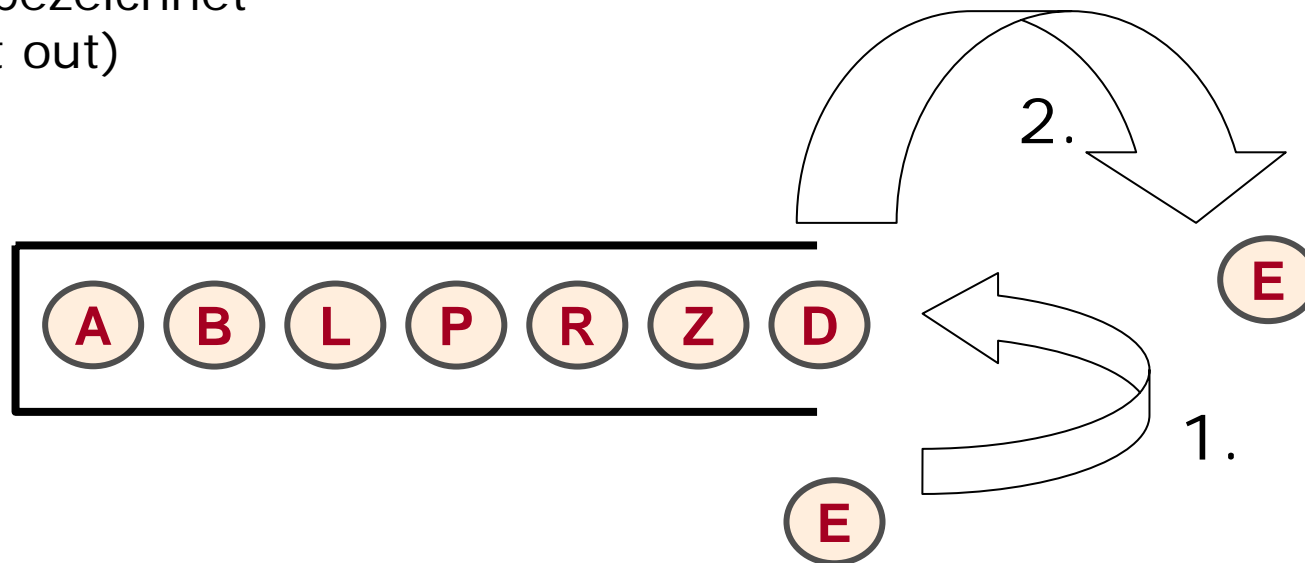
- Zeiger auf beide Enden der Liste?
- Einfach oder doppelt verkettet?
 - Jeder Knoten besitzt einen Zeiger auf seinen Vorgänger
 - → Reihenfolge kann auch rückwärts ausgegeben werden
 - → Zeigeroperationen werden einfacher



Keller: Einführung

- Beispiele für Schlangen im Alltag:
 - **Tellerstapel**, Überfüllte Busse, Rekursive Probleme
- Operationen:
 - Anfügen nur an einem Ende (hinten)
 - → Vorne sind die ältesten Elemente
 - Entfernen am gleichen Ende
- Verwendung in der Informatik:
 - Sichern von lokalen Variablen bei Funktionsaufrufen, Labyrinth oder Schachprobleme, Syntaxanalysen

Wird als **LIFO** bezeichnet
(last in, first out)



Keller: Operationen

Minimale Operationen:

Schreiben

- `push(Object o)`

Einfügen (Anhängen) von Element am Ende

Löschen

- `Object pop()`

Entfernen eines Element **vom Ende**

Lesen und Suchen

- `Object peek()`

Liefert Endelement zurück, ohne zu entfernen

Optionale Operationen:

- `size()`
- `boolean empty()`

Anzahl der Elemente
Keller leer?

Keller: Beispiel

(Elemente des Kellers seien natürliche Zahlen)

Operation	Keller s	Ausgabe
new Stack()	()	-
push (1)	(1)	-
push (3)	(1, 3)	-
peek ()	(1, 3)	3
push (7)	(1, 3, 7)	-
pop ()	(1, 3)	-
peek ()	(1, 3)	3
pop ()	(1)	-
peek ()	(1)	1

Keller: Implementierungen

- Keller mit fester Länge (kein dyn. Datentyp):
- Keller mit variabler Länge (dyn. Datentyp)
 - → Umsetzung mit Hilfe einer einfach verketteten Liste
 - **Java** verfügt bereits über eine Implementierung der Klasse **Stack** im Package `java.util`.

```
void push(Object o) {
    list.addFirst( Object o );
}

Object peek() {
    return list.getFirst();
}

Object pop() {
    Node temp = peek(); // zugreifen
    list.removeFirst(); // löschen
    return temp;        // zurückgeben
}
```

Keller: Implementierungen

Implementierung mit Hilfe eines Arrays

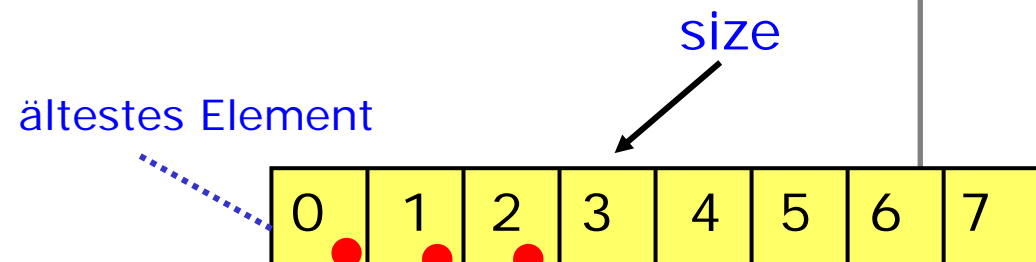
```
private Object[] s; private int size;
```

```
public StackImpl(int capacity) {
    s = new Object[capacity]; size = 0;
}
```

```
public void push(Object o) throws StackException {
    if (size == s.length) throw new StackException ("voll");
    s[size] = o; size++;
}
```

```
public Object pop() throws StackException {
    if (size == 0) throw new StackException("leer");
    Object temp = s[size-1];
    s[size-1] = null;
    size--;
    return temp;
}
```

```
public Object peek() throws StackException {
    if (size == 0) throw new StackException("leer");
    return s[size-1];
}
```



Keller-Beispiel

Erkennen wohlgeformter Klammersausdrücke (WKA)

Definition eines WKA:

1. $()$, $\{ \}$ und $[]$ sind WKAs;
2. sind w_1 und w_2 WKAs, dann ist w_1w_2 ein WKA;
3. ist w ein WKA, dann sind auch (w) , $\{w\}$, $[w]$ WKAs.

Vorgehensweise:

- Zeichenreihe wird zeichenweise gelesen.
- Wird eine öffnende Klammer gelesen, so wird sie auf einen Keller gelegt.
- Wird eine schließende Klammer gelesen, so wird die zuletzt abgelegte Klammer vom Keller gelesen und verglichen, ob sie zur öffnenden Klammer passt.
- Zeichenreihe ist ein WKA, wenn sie auf diese Weise vollständig gelesen werden kann und anschließend der Keller leer ist.

Keller-Beispiel

Algorithmus zum Erkennen von WKA (grob):

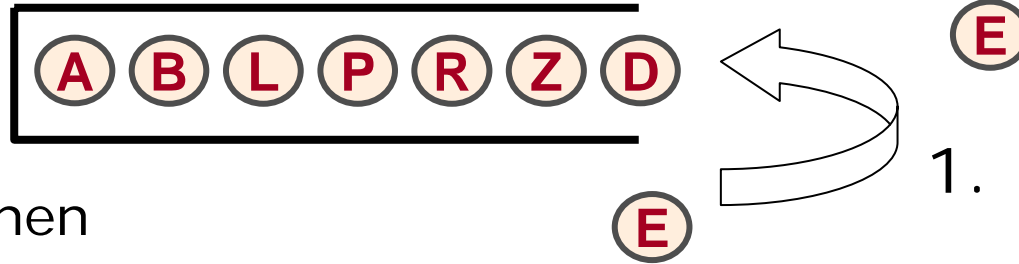
```
public boolean isWKA(String ka) {
    Stack ok = new Stack(); // Stack für öffnende Klammern
    int pos = 0;
    while (pos < ka.length()) {
        char c = ka.charAt(pos);
        if (c == '(' | c == '{' | c == '[')
            ok.push(c);
        else { // schliessende Klammer
            if (ok.empty())
                return false;
            char d = ok.pop(); // passende öffnende Klammer vorher?
            if ( c == '(' && d != ')' |
                | c == '[' && d != ']' |
                | c == '{' && d != '}' )
                return false;
        }
    }
    return ok.empty(); // empty => WKA!
}
```

Keller: Zusammenfassung

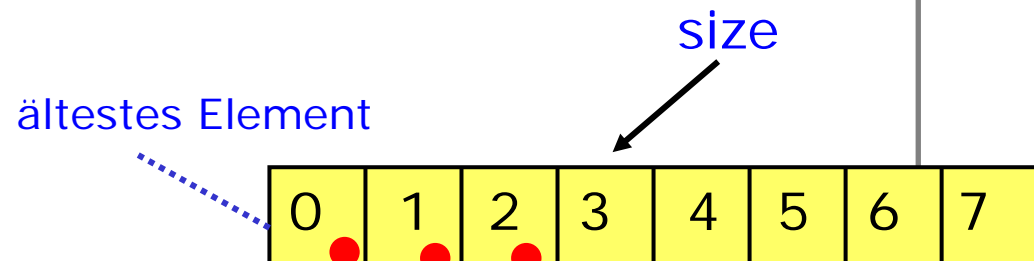
- Beispiele für Schlangen im Alltag:

Tellerstapel,

Überfüllte Busse,
Rekursive Probleme



- Kann z.B. WKAs erkennen
- Wird als **LIFO** bezeichnet (last in, first out)
- Operationen:
 - **push(Object o)** - Anfügen nur an einem Ende (hinten)
 - → Vorne sind die ältesten Elemente
 - **pop()** - Entfernen am gleichen Ende
 - **peek()** – Letztes Element ansehen ohne zu entfernen
- Implementierungen:
 - Mit verketteter Liste
 - Mit einem Array



Schlangen: Einführung

- Beispiele für Schlangen im Alltag:
 - Bankschalter, Supermarkt, Wartezimmer beim Arzt
 - Operationen:
 - Das **erste Element wird „bedient“**
 - Ein **neues Element wird hinten angefügt**
 - Verwendung in der Informatik:
 - Z.B. Verwaltung von Druckaufträgen oder Prozessen
- Wird als **FIFO** bezeichnet (first in, first out)



Schlangen: Operationen

Minimale Operationen:

Schreiben

- `enqueue(Object o)` Einfügen (Anhängen) von Element am Ende

Löschen

- `void dequeue()` Entfernen eines Element vom Anfang

Lesen und Suchen

- `Object front()` Liefert Anfangselement zurück, ohne zu entfernen

Optionale Operationen:

- `size()` Anzahl der Elemente
- `boolean empty()` Schlange leer?

Schlangen: Beispiel

(Elemente der Schlange seien natürliche Zahlen)

Operation	Schlange q	Ausgabe
new Queue()	()	-
enqueue (1)	(1)	-
enqueue (3)	(1, 3)	-
front ()	(1, 3)	1
enqueue (7)	(1, 3, 7)	-
dequeue ()	(3, 7)	-
front ()	(3, 7)	3
dequeue ()	(7)	-
front ()	(7)	7

Schlangen: Implementierungen

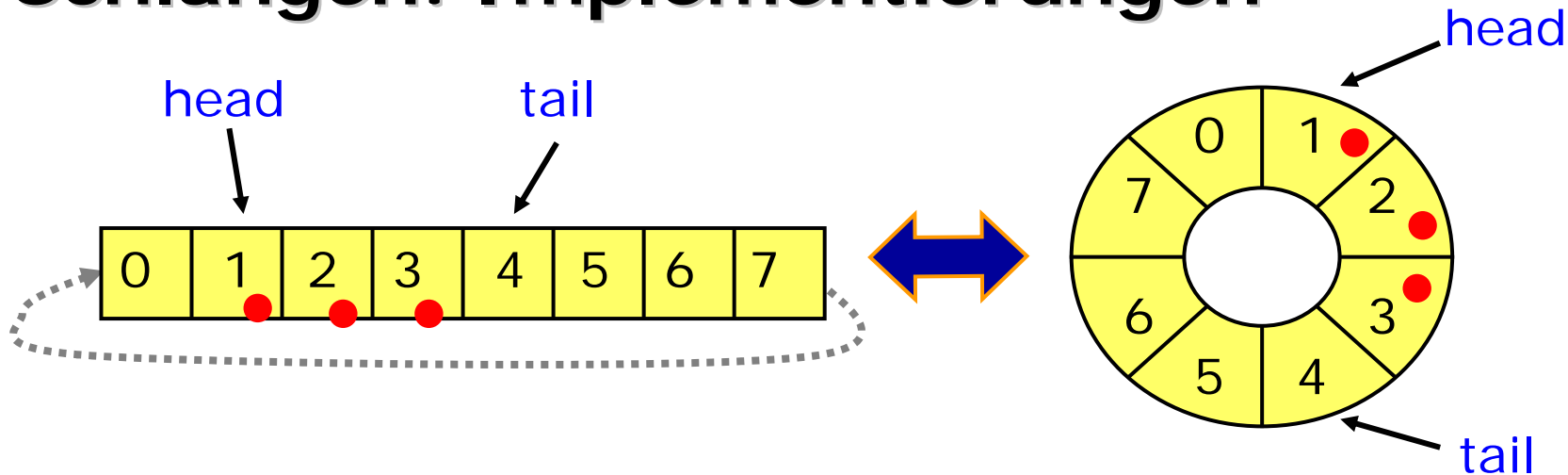
- Schlangen mit fester Länge (kein dyn. Datentyp):
 - Zirkuläre Schlange (Array wird als Kreis aufgefasst)
- Schlangen mit variabler Länge (dyn. Datentyp)
 - Umsetzung mit verketteter Liste, die Zeiger auf erstes und letztes Element verwaltet:

```
void enqueue(Object o) {
    list.addLast( Object o );
}

void dequeue() {
    list.removeFirst();
}

Object front() {
    return list.getFirst();
}
```

Schlangen: Implementierungen



Implementierung mit Hilfe eines Arrays

- Elemente der Schlange befinden sich zwischen den Array-Komponenten "head" und "tail-1".
- Erstes/ältestes Element befindet sich an Position "head",
 - → hier werden Elemente abgerufen/entfernt.
- "tail" entspricht Position **nach** dem letzten Element
 - → hier werden neue Elemente eingefügt.
 - tail kann als $head + size$ dargestellt werden
- → kein Umkopieren von Elementen notwendig

Schlangen: Implementierungen

Implementierung mit Hilfe eines Arrays

```
private Object[] s; private int head, size;
```

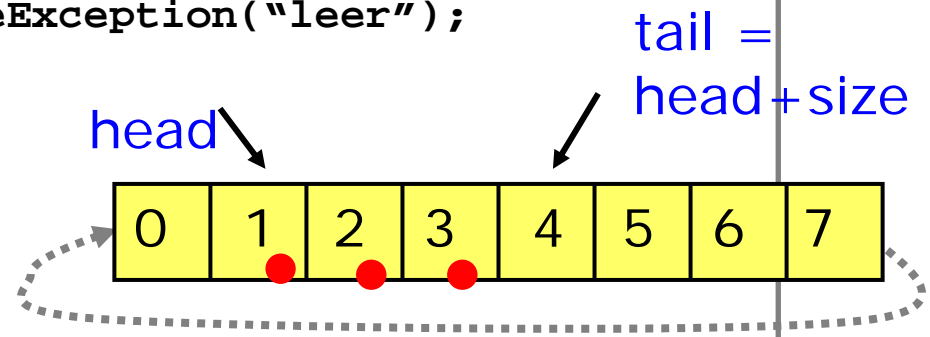
```
public SchlangeImpl(int capacity) {
    s = new Object[capacity]; head = 0; size = 0;
}
```

```
public void enqueue(Object o) throws QueueException {
    if (size == s.length) throw new QueueException("voll");
    s[(head + size) % s.length] = o;
    size++;
}
```

```
public void dequeue() throws QueueException {
    if (size == 0) throw new QueueException("leer");
    s[head] = null;
    head = (head + 1) % s.length;
    size--;
}
```

```
public Object front() throws QueueException {
    if (size == 0) throw new QueueException("leer");
    return s[head];
}
```

„%“ = *modulo* in Java
 Beispiel:
 $(5 + 4) \% 8 = 1$
 $(7 + 1) \% 8 = 0$



Schlangen: Zusammenfassung

- Beispiele für Schlangen im Alltag: Bankschalter, Supermarkt, Wartezimmer beim Arzt

- Wird als **FIFO** bezeichnet (first in, first out)

- Operationen:

- `dequeue()`

- Das **erste Element** wird „bedient“

- `enqueue(Object o)`

- Ein **neues Element** wird hinten angefügt

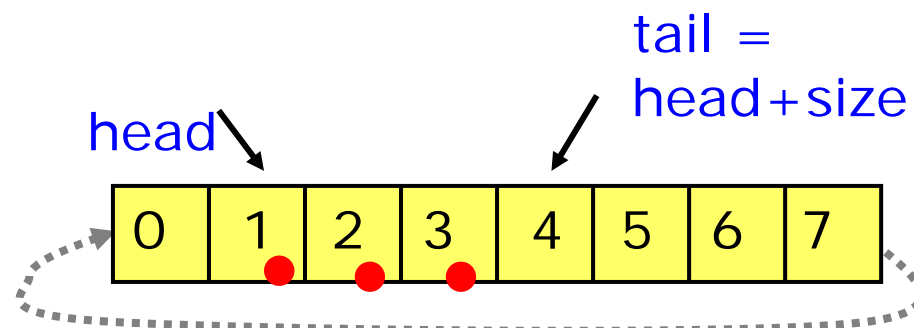
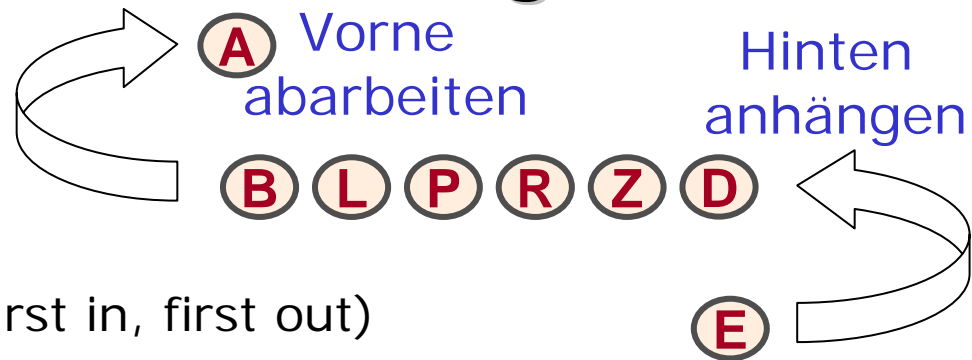
- `front()`

- Liefert **Anfangselement** zurück, ohne zu entfernen

- Implementierungen:

- Mit verketteter Liste (mit Zeiger auf das letzte Element)

- Mit zirkulärem Array



Vergleich: Keller und Schlangen

Operation

■ Einfügen eines Objektes x am Ende

■ Entfernen des
 • ersten
 • letzten
 Objektes

■ Liefern des
 • ersten
 • letzten
 Objektes

Schlange

■ Enqueue(x)

■ Dequeue()

■ ---

■ ---

■ Front()

Keller

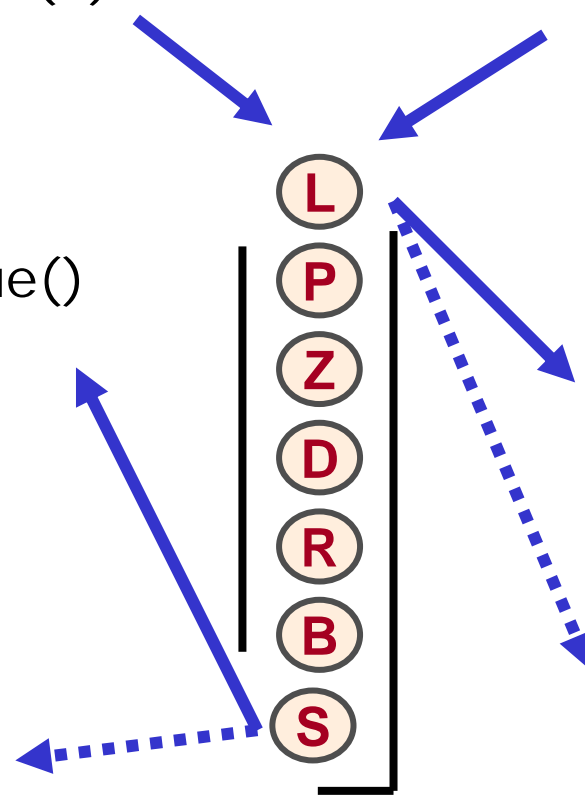
■ Push(x)

■ ---

■ Pop()

■ Peek()

■ ---



Bäume: Einführung

■ Listen

- Eindimensional
- Linear
- 1 Nachfolger
- 1 Vorgänger
- Suchen in $O(n)$

■ Verwendung z.B.

- Keller
- Schlange

■ Bäume

- Mehrdimensional
- ---
- Mehrere Nachfolger
- 1 Vorgänger
- **Schneller suchen?**
- → neue Datenstruktur

■ Verwendung z.B.

- Binärer Baum
- Binärer Suchbaum

■ Literatur

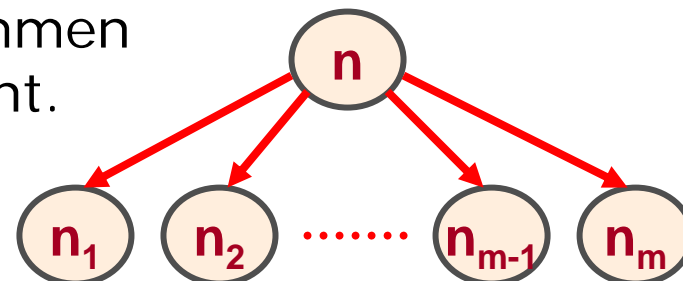
- [Balzert,99] Kap. 3.8; [Sedgewick,99] Kap. 4;
- [Ottmann99] Kap. 5
- Siehe auch: Dipl.-Inform. G. Langemeier (Uni-Hildesheim), Programmierung spezieller Probleme in Java, <http://www.mathematik.uni-hildesheim.de/SS2002/Java/scripts.asp>, Folie 11a

Bäume: Definition

- Sei N eine nicht-leere Menge von Knoten n_i (*nodes*).
- Ein Knoten n_0 enthält eine (möglicherweise leere) Menge von *Nachfolger*-Knoten $n_1 \dots n_m$.
Diese werden auch *Kinder*, *children*, *sons* von n genannt.

Begriffe:

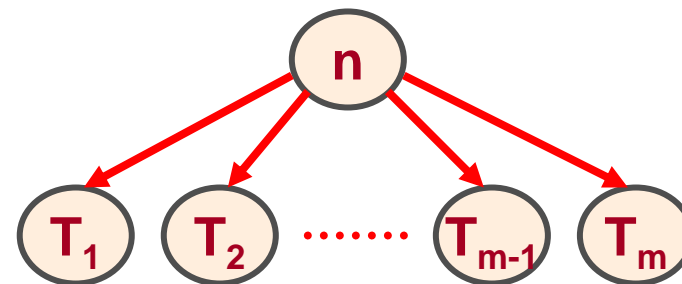
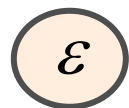
- b ist *Vorgänger* (auch *Elternteil*, *Vater*, *parent*) von a
 $\leftrightarrow a$ ist Kind von b
- Ein Knoten heißt *Wurzel* des Baumes, wenn er keinen Vater hat.
- n_i und n_j sind *Geschwister* (*siblings*)
 $\leftrightarrow \text{Vater}(n_i) = \text{Vater}(n_j)$
- Eine *Kante* geht von n_0 nach n_i , $1 \leq i \leq m$
- Ein Knoten, der keine Nachkommen besitzt, wird *Blatt* (*leaf*) genannt.



Rekursive Definition eines Baumes:

Sei N eine nicht-leere Menge von Knoten.

- ε =: leerer Baum (leeres Wort über N)
- (n_0) =: Baum der nur aus einem (Wurzel-) Knoten besteht.
- (n_0, T_1, \dots, T_m) =: Baum mit Wurzel n_0 , wobei folgende **Konsistenzbedingung für Bäume** gilt:
 - $n_0 \in N$, T_1, \dots, T_m Bäume mit Wurzeln n_1, \dots, n_m
 - T_1, \dots, T_m enthalten **keine gemeinsamen Knoten**
 - T_1, \dots, T_m enthalten nicht den Knoten n_0



Weitere Definitionen:

- T_1, \dots, T_m sind **Unterbäume** von n_0 .
- Auch Unterbäume sind zueinander Kind, Vater und Geschwister.

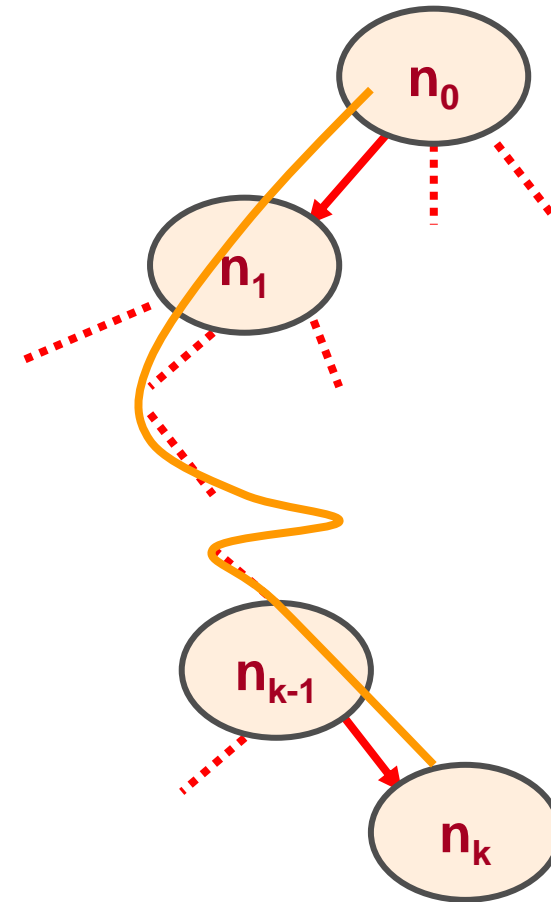
Bäume: Pfad

n_0, \dots, n_k ist Pfad der Länge $k \iff$

- T ist Baum mit Wurzel n_0
- n_0, n_1, \dots, n_k sind Knoten aus T
- $\forall i$ mit $1 \leq i < k$: $\text{parent}(n_i, n_{i+1})$

Begriffe

- n_0 ist Pfad der Länge 0
- n_0 ist **Vorfahre** von n_k
- n_k ist **Nachkomme** von n_0
- n_k ist **direkter Nachkomme** (Kind) von n_{k-1}

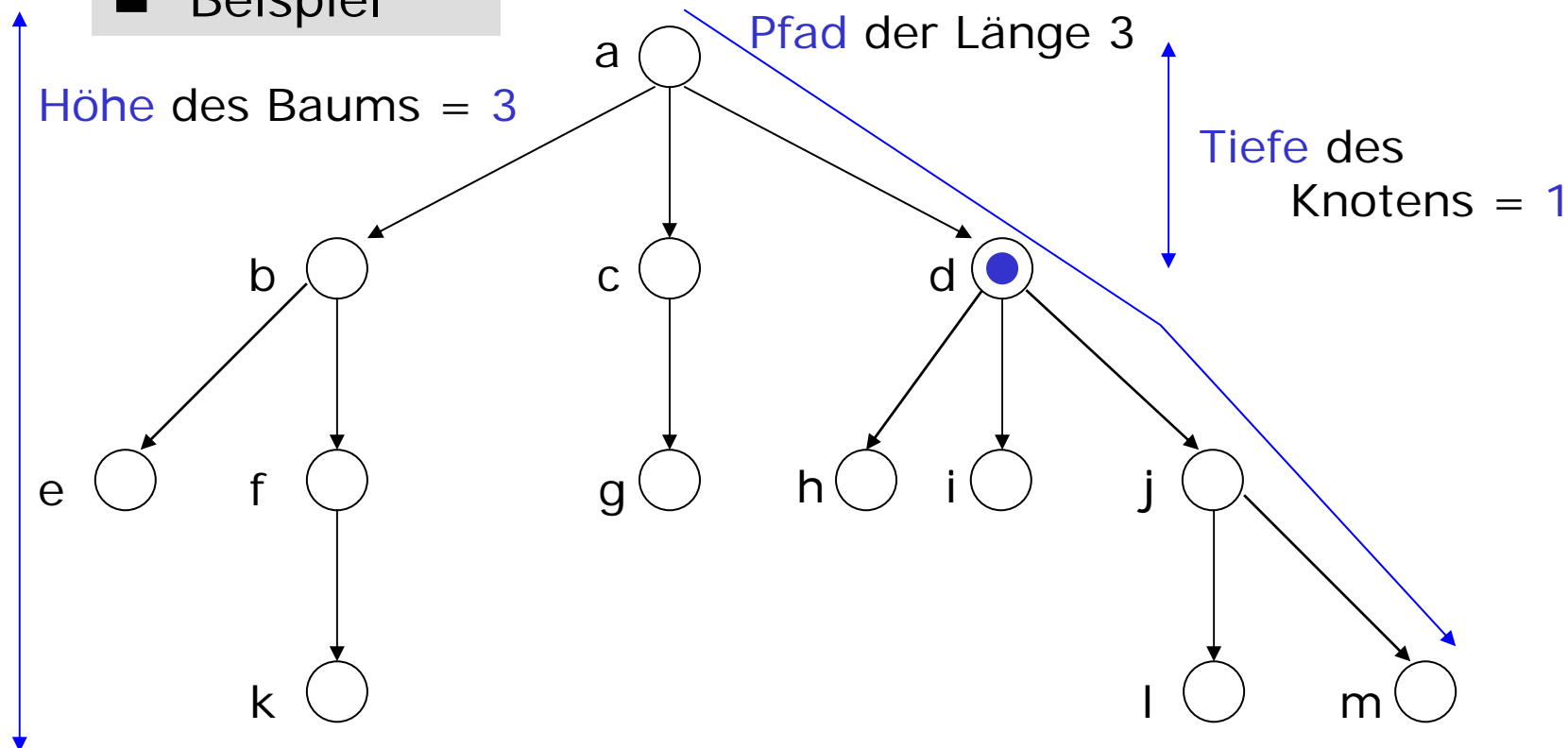


Bäume: Höhe, Tiefe, Ordnung

Sei T ein Baum mit Wurzel n_0 und n_1, \dots, n_k Knoten aus T

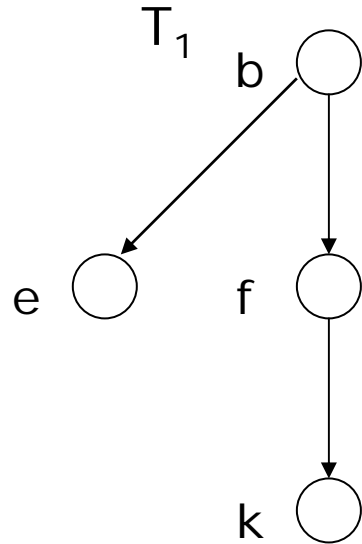
- **Tiefe** von n_i = Länge des Pfades n_0, \dots, n_i
= Abstand von der Wurzel
- **Höhe** von T = Höhe von n_0
= maximale Tiefe
- T hat die **Ordnung** $j \Leftrightarrow$ jeder Knoten $n_i \in T$ hat max. j Kinder

■ Beispiel



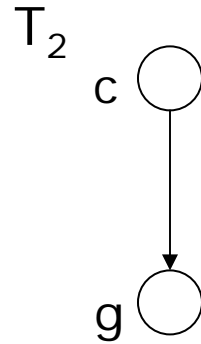
Bäume: Beispiel

- Knoten a hat die drei Teilbäume: T_1 , T_2 und T_3



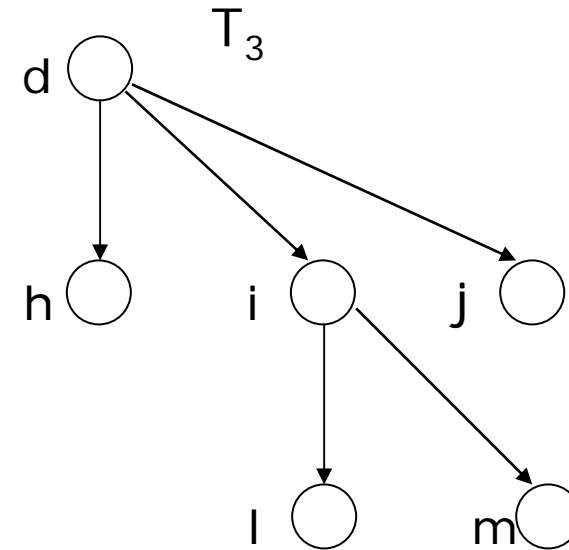
T_1 hat **Ordnung** 2

T_1 hat **Höhe** 2



T_2 hat **Ordnung** 1

T_2 hat **Höhe** 1



T_3 hat **Ordnung** 3

T_3 hat **Höhe** 2

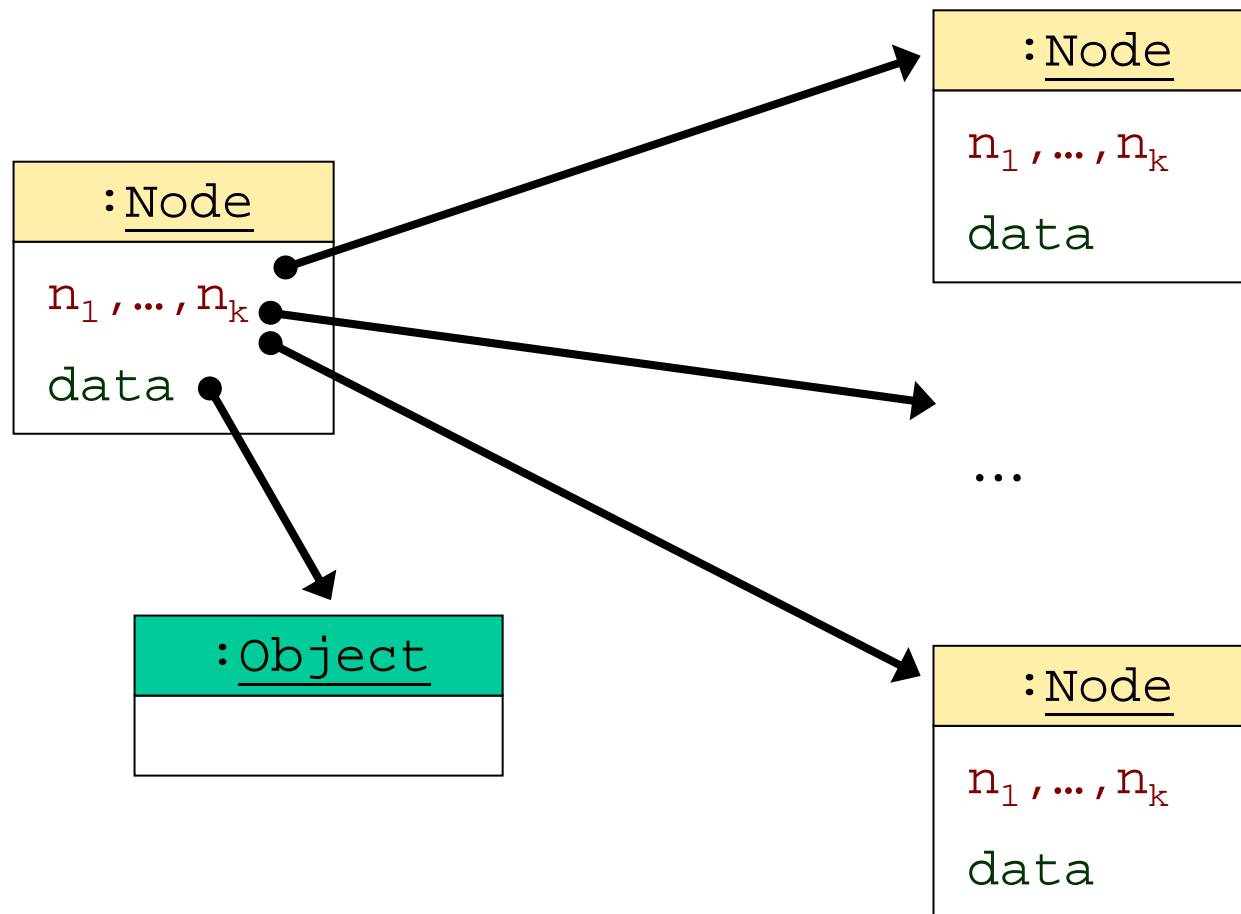
Pfad b zu k hat die Länge 2, k hat die Tiefe 2

Pfad c zu g hat Länge 1, g hat Tiefe 1

Bäume als dynamische Datenstruktur

Ein Knoten ist ein Verwaltungselement mit einer geordneten Menge T von Kinder-Knoten und einem Datenelement.

Daraus lassen sich Bäume (unter Bewahrung der Konsistenzbedingung) zusammensetzen.



Operationen auf Bäumen

- `add(Object o)` = insert
- `remove(Object o)` = delete
- `boolean contains(Object o)`
- Durchlaufen aller Knoten in bestimmter Reihenfolge:
`preOrder`, `postOrder`, `inOrder`
- `Tree[] split` = Aufspalten eines Baumes in Teilbäume:
- `Merge(Tree t)` = Zusammenfügen mehrerer Bäume zu einem neuen Baum

- Konstruieren eines Baums mit bestimmten Eigenschaften:
 - Baum mit minimaler Höhe
 - AVL-Baum (ausgeglichen bzgl. Höhendifferenz)
 - Suchbäume

- **Rekursive Struktur → rekursive Algorithmen**

Binäre Bäume

Definitionen

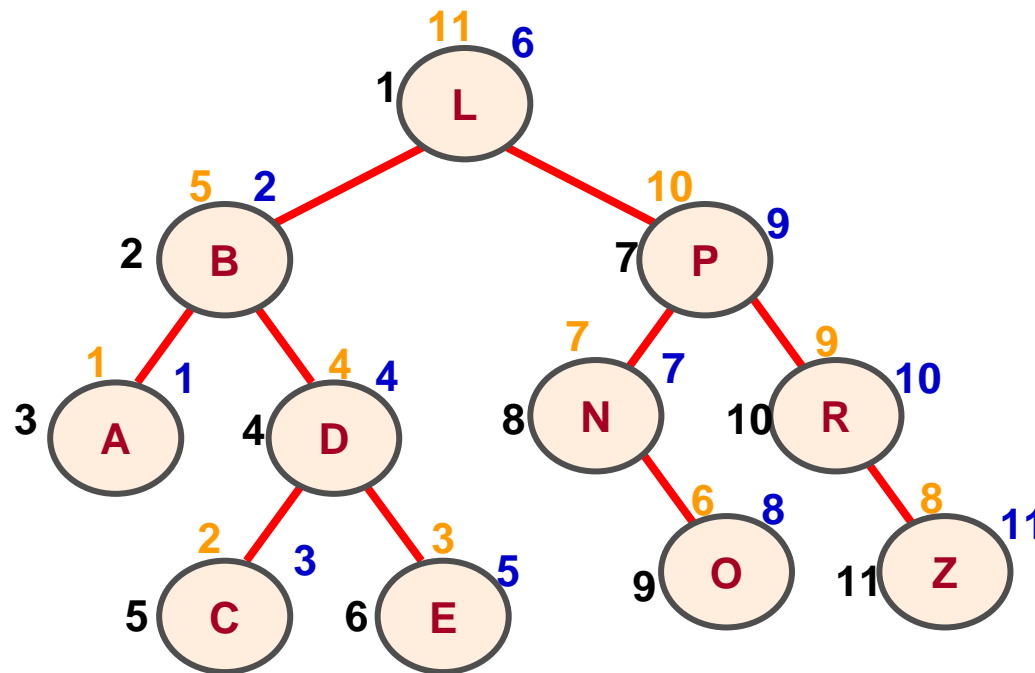
- Ein Baum der Ordnung n heißt **n -ärer Baum**
- **Ein Baum der Ordnung 2 heißt binärer Baum**
d.h. Jeder Knoten eines Binärbaums hat **maximal 2 Kinder**

Gängige Traversierungsstrategien

(zum Ablaufen aller Elemente, immer „links vor rechts“):

- **Hauptreihenfolge** (*preorder*):
Wurzel, linker Teilbaum, rechter Teilbaum → „**WLR**“
- **Symmetrische Reihenfolge** (*inorder*):
Linker Teilbaum, Wurzel, rechter Teilbaum → „**LWR**“
- **Nebenreihenfolge** (*postorder*):
Linker Teilbaum, rechter Teilbaum, Wurzel → „**LRW**“

Beispiel: Traversierungsstrategien



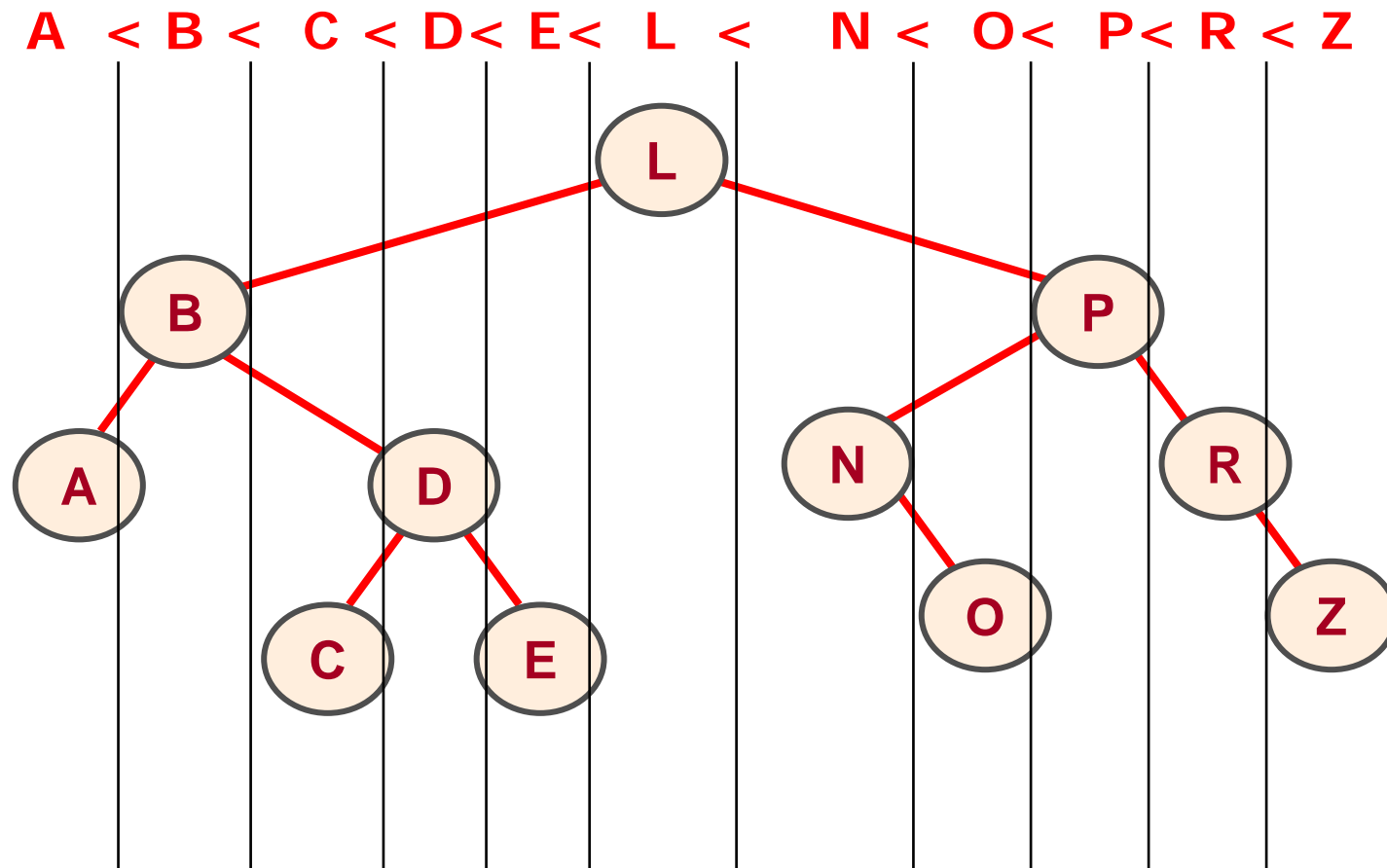
preorder liefert L , B , A , D , C , E , P , N , O , R , Z

postorder liefert A , C , E , D , B , O , N , Z , R , P , L

inorder liefert A , B , C , D , E , L , N , O , P , R , Z

→ inorder liefert für Binärbäume die aufsteigend sortierte Reihenfolge

Beispiel: Traversierungsstrategien

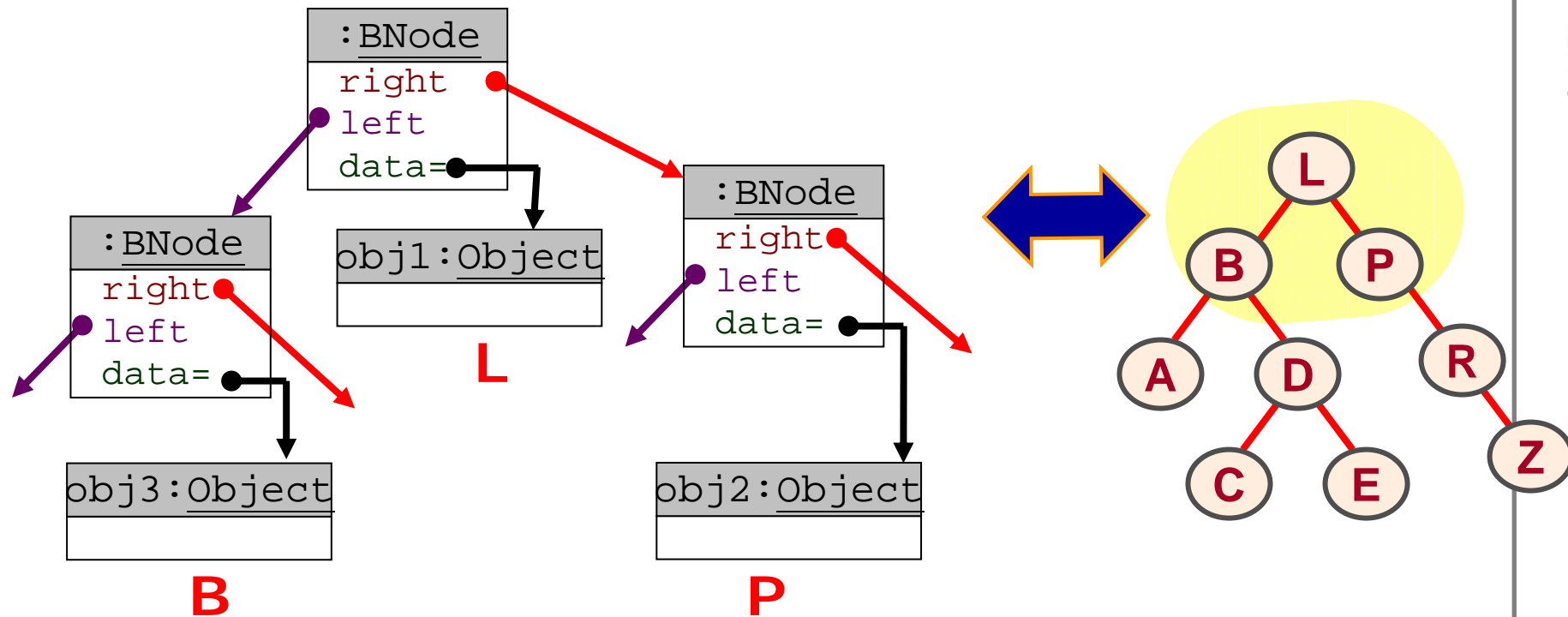


inorder liefert A , B , C , D , E , L , N , O , P , R , Z

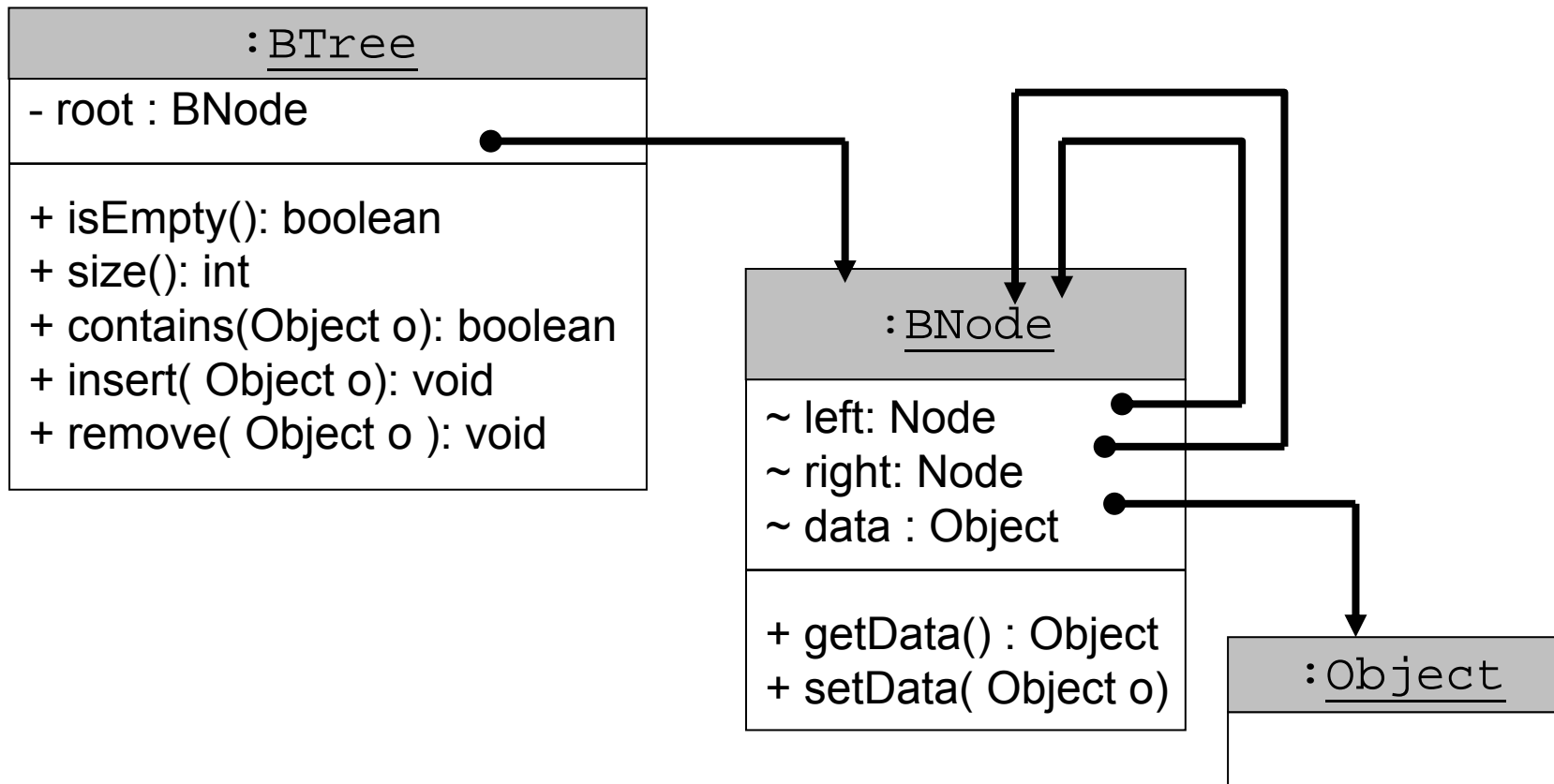
→ *inorder* liefert für Binärbäume die aufsteigend sortierte Reihenfolge

Binäre Bäume: Aufbau

- Verwaltungselemente „**BNode**“
 - Referenz **data** auf Datenelemente vom Typ **Object**
 - Verkettung der Verwaltungselemente:
 - Referenzen **left**, **right** auf linken und rechten Nachfolger vom Typ **BNode**



Binäre Bäume: Klassendiagramm



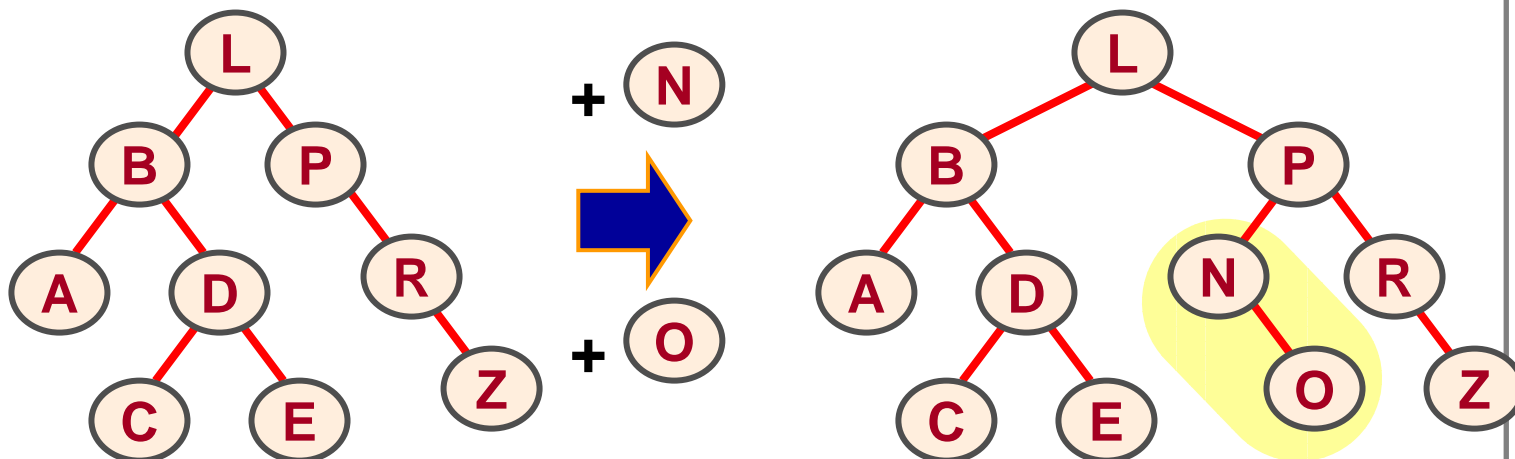
Binäre Suchbäume

Definition

- Ein *binärer Suchbaum* ist ein *binärer Baum* in dem alle Knoten geordnet sind, d.h.
 - Alle **Knoten eines linken Teilbaums sind kleiner** als seine Wurzel
 - Alle **Knoten eines rechten Teilbaums sind größer** als seine Wurzel

Eigenschaften

- → **Suchen relativ schnell** möglich, wenn der Baum balanciert ist: Laufzeit = $O(\log n)$
- → **Einfügen und Löschen müssen die Ordnung erhalten**



Binäre Suchbäume: Suchen

Knoten mit Objekt **o** soll im Baum gesucht werden. Dazu wird ein Zeiger auf einen jeweils aktuellen Knoten **a** verwendet.

Initialisierung:

Beginne an der Wurzel → **a = root**

Schleife:

Solange **o** nicht gefunden wurde und **a** nicht leer:

- Entscheide für aktuellen Knoten **a**:
 - **a.data == o**: **o** gefunden!
 - **a == leer**: **o** nicht im Baum gefunden!
 - **a.data > o**: Suche im linken Teilbaum → **a = a.links**
 - **a.data < o**: Suche im rechten Teilbaum → **a = a.rechts**

⇒ Methoden **contains(Object obj)** und **contains(Object obj, BNode subtree)** der Klasse **BTree**

Beispielsimplementierung: Suchen

```
boolean contains(Object obj){  
    // Initialisierung: Beginne an der Wurzel → a = root  
    return contains(obj, root);  
}
```

```
boolean contains(Object obj, BNode a) {  
    // Entscheide für aktuellen Knoten a:  
    if ( a == null) {                               // a ist leer  
        return false;                             // Mißerfolg (false)  
    } else {  
        if (a.getData() < obj)                     // a.data < obj  
            // Suche im rechten Teilbaum  
            return contains(obj, a.getRight());  
        else if (a.getData() > obj)                 // a.data > obj  
            // Suche im linken Teilbaum  
            return contains(obj, a.getLeft());  
        } else if (a.getData() == obj)              // a.data = obj  
            return true;                           // Erfolg (true)  
    }  
}
```

Binäre Suchbäume: Einfügen

Knoten mit Objekt **o** soll im Baum **eingefügt** werden. Dazu wird ein Zeiger auf einen jeweils aktuellen Knoten **a** und seinen Vater **v** verwendet.

Initialisierung: Beginne an der Wurzel → **a = root**

Schleife: Solange **a** nicht leer:

- **v = a** **a merken**
- Entscheide für aktuellen Knoten **a**:
 - **a.data > o**: **Einfügen** im linken Teilbaum → **a = a.links**
 - **a.data < o**: **Einfügen** im rechten Teilbaum → **a = a.rechts**
 - **a == leer**: **o** nicht im Baum gefunden!
 - aber die Stelle wo es „hingeht“
 - also als neues Kind von **v** einfügen!
 - je nach Wert: links oder rechts

→ Methoden `insert(Object obj)` und `insert(Object obj, BNode subtree)` der Klasse `BTree`

Binäre Suchbäume: Löschen

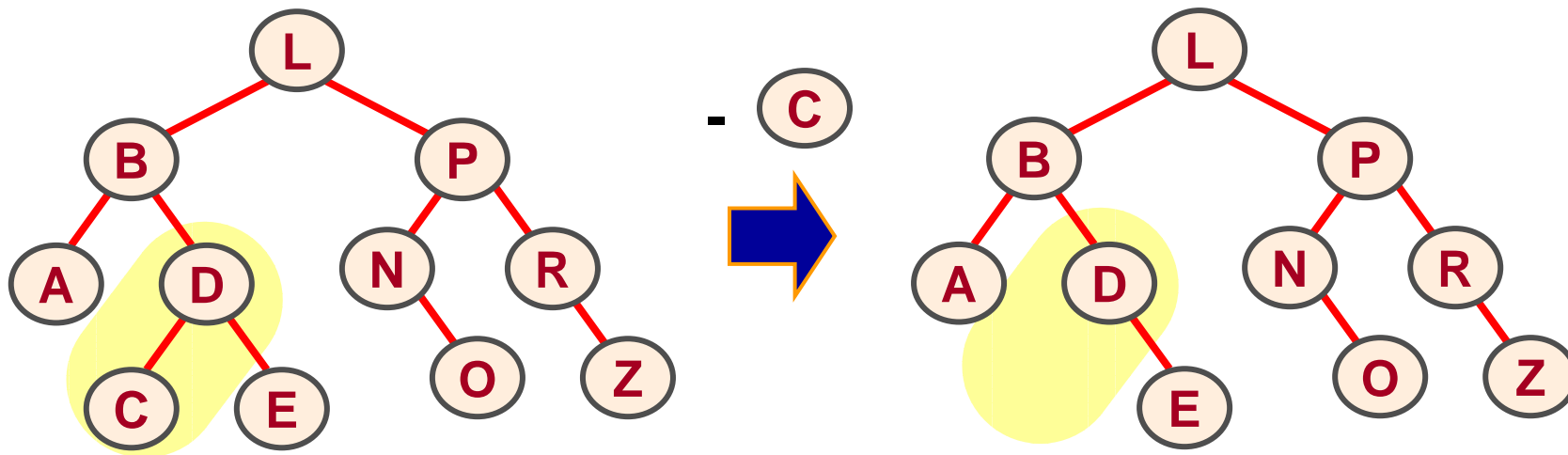
- Knoten mit Objekt **o** soll im Baum **gelöscht** werden.

1. Fall:

Knoten ist Blatt, d.h. hat **keine Nachfolger**

→ Suche Knoten

→ Setze entsprechenden Kindknoten (hier: D.links) des zugehörigen Vaterknotens (hier: D) zu **null**



Binäre Suchbäume: Löschen

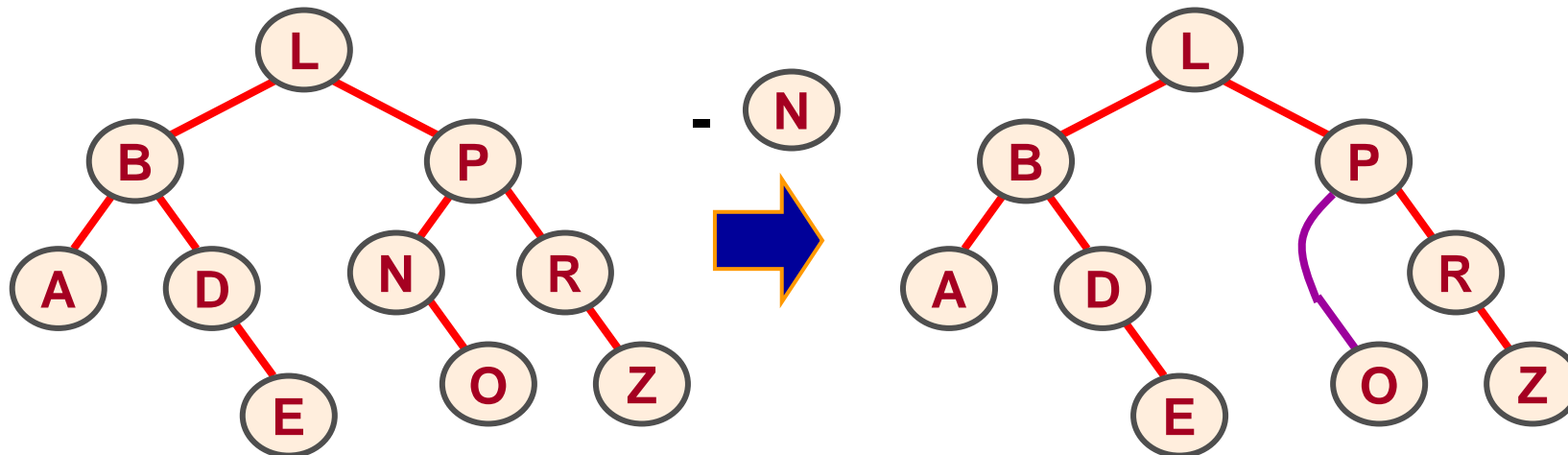
- Knoten mit Objekt **o** soll im Baum **gelöscht** werden.

2. Fall:

Knoten hat **genau einen Nachfolger**

→ Suche Knoten

→ Setze entsprechenden Kindknoten (hier: links) des zug. Vaterknotens (hier: P) auf den nicht-leeren Nachfolger des zu löschenden Knotens (hier N.rechts = O)



Binäre Suchbäume: Löschen

- Knoten mit Objekt **o** soll im Baum **gelöscht** werden.

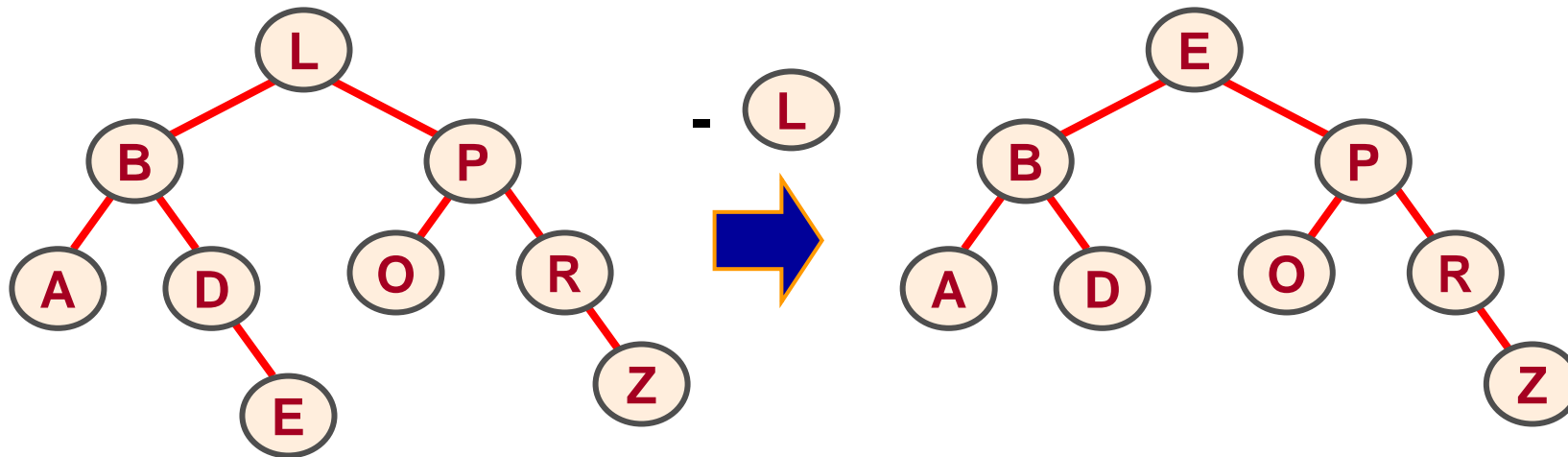
3. Fall:

Knoten hat **zwei Nachfolger**

→ Suche Knoten

→ Ersetze zu löschenden Knoten (hier: L) durch größten Knoten des linken Teilbaums (hier: E, da ganz rechts)

→ korrigiere Verweise (hier: D.rechts = null)



Binäre Suchbäume - Löschen

Knoten mit Objekt **o** soll im Baum **gelöscht** werden.

1. Suche Knoten **k** mit **k.data = o**.
Sei k der linke Sohn, d.h. k.vater.links == k
2. Anzahl der Nachfolger von **k** ist:
 - 0: Blatt löschen:
k.vater.links = null
 - 1: ersetze **k** durch den Sohn-Knoten
if(k.links != null)
k.vater.links = k.links
 - 2: ersetze aktuellen Knoten **k**
durch größten Knoten **g** des linken Teilbaums
 - Lösche **g** (Dieser hat **maximal einen** linken Sohn. Hätte er einen rechten Sohn, wäre dieser grösser.)
 - Ersetze **k** durch **g**
(Also **g.links = k.links** und **g.rechts = k.rechts**).

Bäume: Zusammenfassung

Bäume

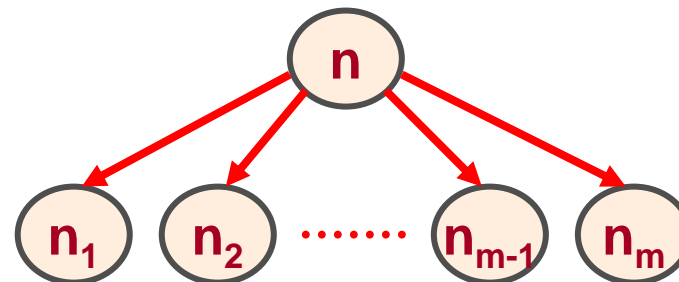
- Begriffe:
 - Nachfolger-Knoten, Kinder (children, sons), Vorgänger, Elternteil, Vater (parent), Wurzel (root), Geschwister (siblings), Blatt (leaf)
 - Kante, Pfad der Länge k , Höhe, Tiefe, Ordnung
- Operationen:
 - `add(Object o)`, `remove(Object o)`, `boolean contains(Object o)`, `Tree[] split`, `Merge(Tree t)`

Binäre Bäume

- Traversierung:
 - `preOrder` (WLR), `postOrder` (LRW), `inOrder` (LWR)
- Implementierung

Binäre Suchbäume

- Operationen: Suchen, Einfügen, Löschen



Vergleich: Semantik

	Position einer Komponente ist abhängig von:
eindimensional <ul style="list-style-type: none">■ Liste<ul style="list-style-type: none">• Sortierte Liste• Keller• Schlange	Wert der Datenelemente Reihenfolge der Operationen Reihenfolge der Operationen
mehrdimensional <ul style="list-style-type: none">■ Baum<ul style="list-style-type: none">• Binärer Baum<ul style="list-style-type: none">▪ Bin. Suchbaum	Reihenfolge der Operationen Wert der Datenelemente

Vergleich: Laufzeiten der Grundoperationen

	<i>Add(x)</i>	<i>Remove(x)</i>	„Lesen“	<i>Contains(x)?</i>
Verkettete Liste	$O(1)$	$O(n)$	First() $\rightarrow O(1)$ Get(i) $\rightarrow O(n)$	$O(n)$
Verkettete sortierte Liste	$O(n)$	$O(n)$	First() $\rightarrow O(1)$ Get(i) $\rightarrow O(n)$	$O(n)$
Binärer Suchbaum	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$