universitätbonn

Objektorientierte Realisierung komplexer Datenstrukturen

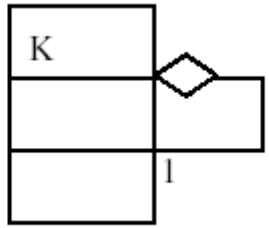
Referenzvariablen als Zeigervariablen
Listen
Stapel und Warteschlangen
Generisches Datenstrukturen
Bäume

Standardbibliotheken

- In Java Standardbibliotheken viele komplexe Datenstrukturen definiert
  - Kapselung durch Verwendung Objekt-orientierter Konzepte
- Standard-Bibliotheken können leicht um eigene komplexe Datenstrukturen erweitert werden
- Wollen im folgenden Grundprinzipien der Realisierung komplexer Datenstrukturen darstellen
  - Und nicht die Funktionalität der Standard-Bibliotheken

Komplexe Datenstrukturen

- In imperativen Sprachen
  - Realisierung komplexer Datenstrukturen durch Verwendung von Zeigervariablen
    - Siehe auch Vorlesung „Algorithmisches Denken und imperative Programmierung“
      - Insbesondere Beispiel „Listen“
  - In Java: Referenzvariable auf Klasse vom selben Typ kann wie Zeigervariable verwendet werden
    - Eine Referenzvariable vom gleichen Typ
      - Einfach verkettete Liste
    - Bei zwei Referenzvariablen vom gleichen Typ
      - Je nach Interpretation
        - Doppelt verkettete Listen
        - Binärbäume
        - Allgemeine Bäume



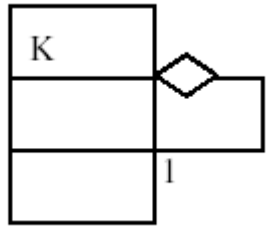
universitätbonn

Listen

Listen (linked lists)

- Falls Objekte vom Typ einer Objektklasse K eine Referenzvariable vom selben Typ K als Feld enthalten, so kann man K-Objekte über diese Referenzen zu einer **primitiven Liste** verknüpfen

Klassendiagramm

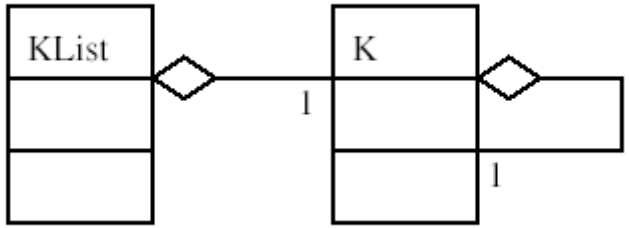


Listen (linked lists)

- Verzeigerte Listen besonders geeignet zur Repräsentation von dynamisch wachsenden und schrumpfenden Mengen von Objekten
  - Über die Referenzen kann man neue Mitglieder an beliebiger Stelle **einflechten** oder **ausketten**
  - Man kann mehrere Mengen zu einer einzigen verschmelzen
  - Weitere typische Listenoperationen sind das **Anfügen** eines neuen Knotens an den Anfang oder das Ende der Liste
- Listen stellen auch Aggregationen von Objekten des selben Typs dar
  - Ähnlich wie Arrays

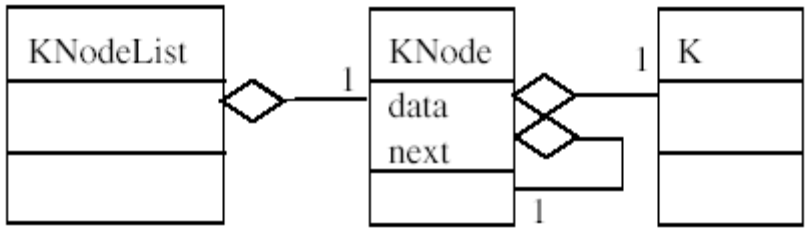
Container-Klassen für Listen

- Containerklassen für Listen sinnvoll
  - Enthalten Referenz auf Kopf der Liste
  - Sowie die Methoden



Listenknoten

- Listenzelle (list cell) minimaler Knoten
  - Enthält nur zwei Felder
    - Ein **Datenfeld** mit einer Referenz auf den **Inhalt** der Zelle (hier ein Objekt vom Typ K)
    - Eine Referenz auf den nächsten Knoten



Implementierung von Listen in Java

Listenzelle

```
public class TNode {
    T data;     // Datenelement
    TNode next; // nächste Listenzelle

    // Selektoren
    public T getData()
    { return data; }

    public TNode getNext ()
    { return next; }
    void setData(T nd)
    { data = nd; }
    void setNext(TNode nn)
    { next = nn; }

    //Konstruktoren
    public TNode(T a)
    { data = a; next = null; }
    public TNode(T a, TNode n)
    { data = a; next = n; }
}
```

Container-Klasse

```
public class TList {
    // Kopf der Liste
    private TNode head;
    //Konstruktoren
    public TList() {
        head = null;
        // null repräsentiert leere Liste
    }
    // weitere Methoden siehe unten
}
```

Durchlauf durch Listen

- Beispiel für Durchlauf durch Listen
  - `toString()`-Methode

Typischer Listendurchlauf

Aus Effizienzgründen direkter Zugriff auf Felder von TNode

```
public class TList { // [..]
    /**
     * Liste wird von runden Klammern begrenzt
     * und die Listenelemente durch Leerzeichen
     * getrennt ausgegeben.
     */
    public String toString() {
        // Initialize
        StringBuffer sb = new StringBuffer();
        TNode x;
        sb.append("(");
        x=head;
        // Print
        while (x != null) {
            sb.append(x.data.toString());
            sb.append(" ");
            x = x.next;
        }
        // Finalize
        sb.append(")");
        return sb.toString();
    }
}
```

Typische Operationen auf Listen

- Einfügen eines Elements am Anfang

```
public class TList { // [..]
    /**
     * Fügt elem am Anfang der Liste ein.
     * Ist auch richtig für den
     * Spezialfall der leeren Liste.
     */
    public void insertFirst(T elem) {
        // Neue Listenzelle erzeugen,
        // deren next Feld auf head zeigt.
        TNode tmp = new TNode(elem, head);
        // head auf neuen Anfang setzen
        head = tmp;
    }
}
```

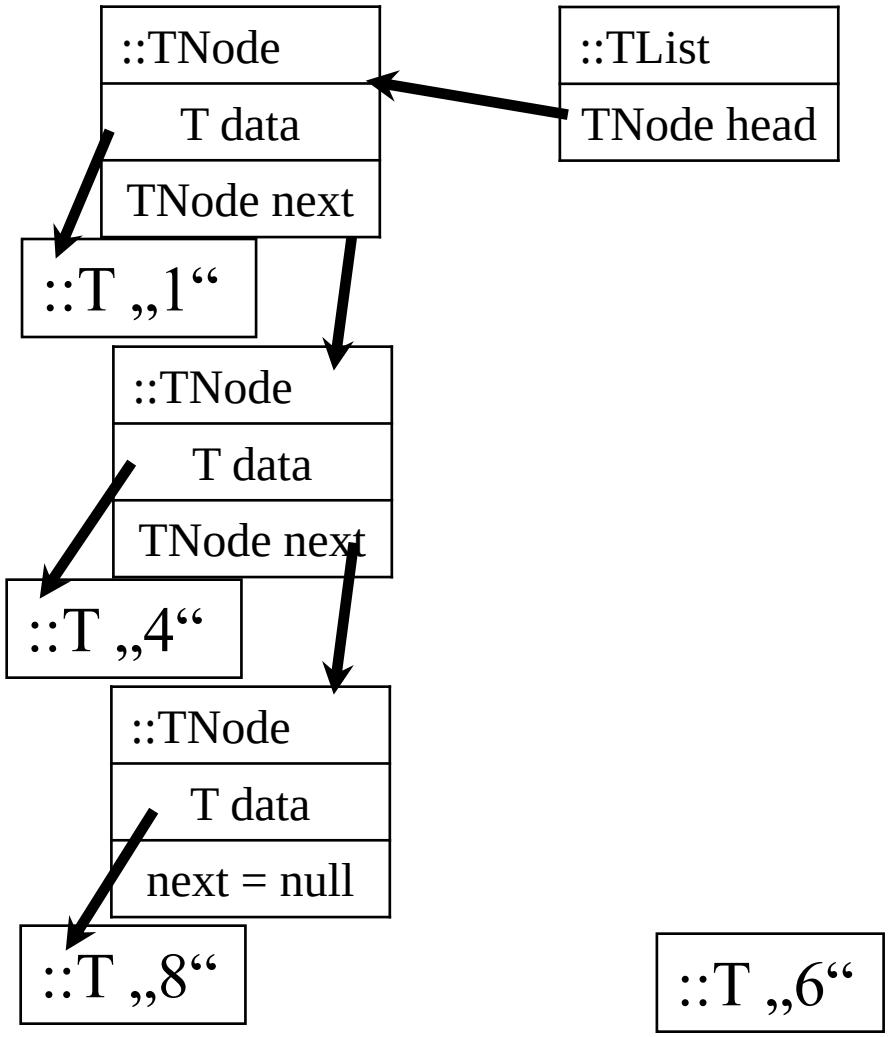
Typische Operationen auf Listen

- Einfügen am Ende

```
public class TList { // [..]
    /**
     * Fügt elem am Ende der Liste an.
     */
    public void insertLast(T elem) {
        TNode tmp=head;
        // Trivialfall: leere Liste
        if (head == null)
        {
            head = new TNode(elem);
            return;
        }
        // Allgemeinfall:
        // zum Ende der Liste gehen
        while (tmp.next != null)
            tmp = tmp.next;
        // neue Listenzelle erzeugen und anfügen
        tmp.next = new TNode(elem);
    }
}
```

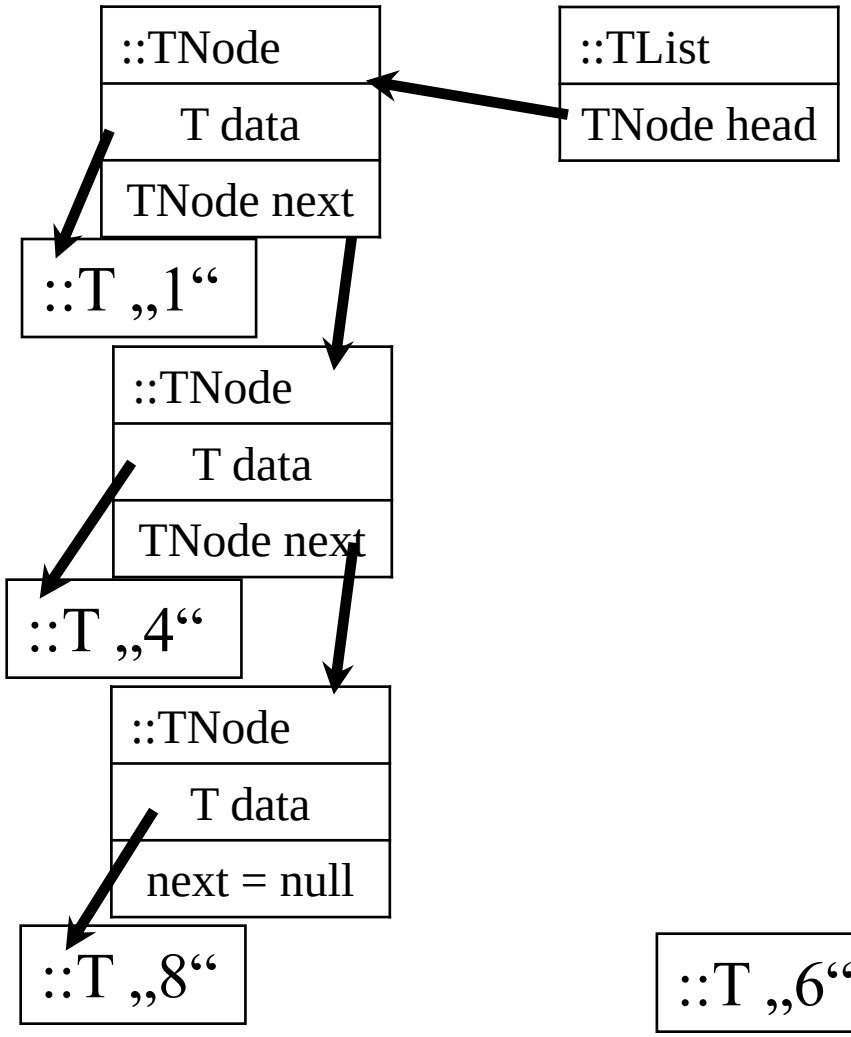
Wichtig:
Test auf
Sonderfälle

Typische Operationen auf Listen



- **Sortiertes Einfügen**
  - Liste sei geordnet
    - Die Ordnung auf den Daten sei etwa durch Methode **`compareTo`** gegeben
    - Wie wird ein weiteres Datenobjekt unter Beibehaltung der Ordnung einsortiert?

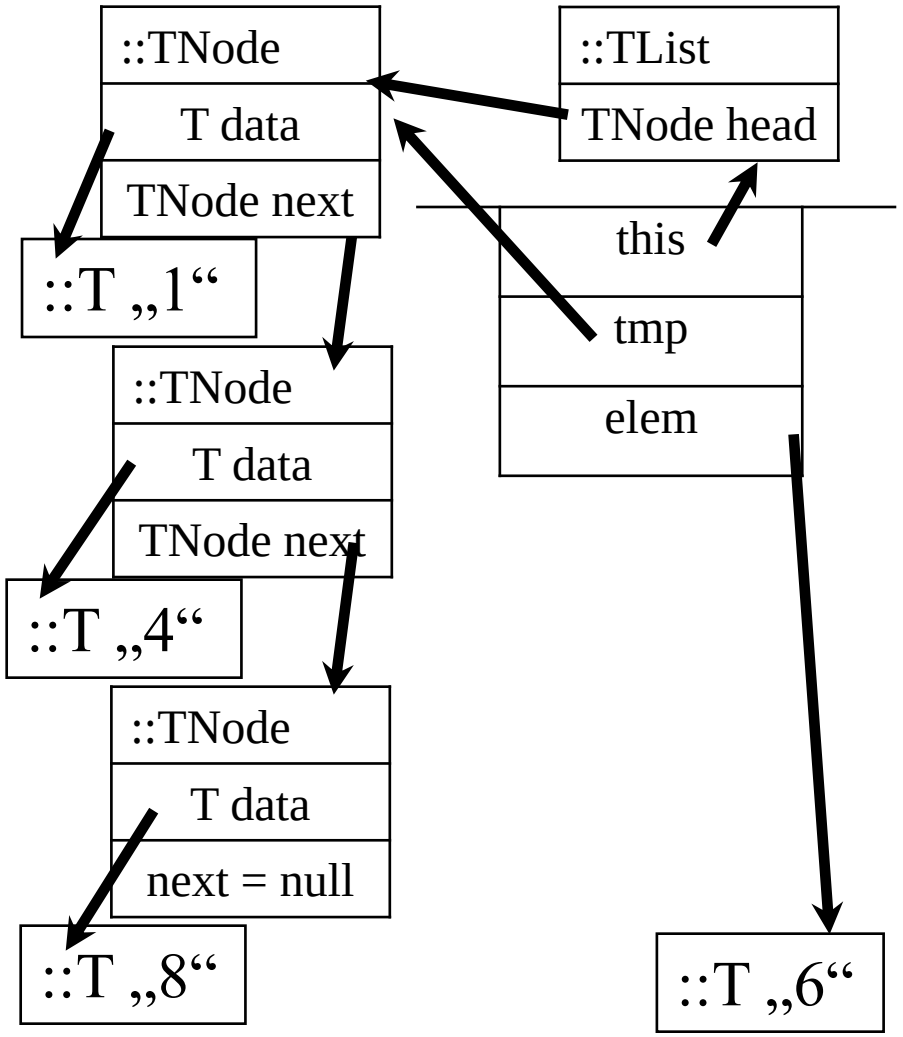
Typische Operationen auf Listen



- **Sortiertes Einfügen: Code**

```
public class TList { // [..]
    /**
     * Fügt elem sortiert in die
     * aufsteigend sortierte Liste ein.
     */
    public void insertSorted(T elem) {
        TNode tmp = head;
        // Spezialfall: Liste ist leer oder
        // neues Element ist kleinstes
        if( (head == null) ||
            (head.data.compareTo(elem) > 0))
        { insertFirst(elem); return; }
        // Allgemeinfall: an die richtige
        // Stelle in der Liste gehen
        while( (tmp.next != null) &&
               (tmp.next.data.compareTo(elem) <= 0))
            tmp = tmp.next;
        // neue Listenzelle erzeugen und einfügen
        tmp.next = new TNode(elem, tmp.next);
    }
}
```

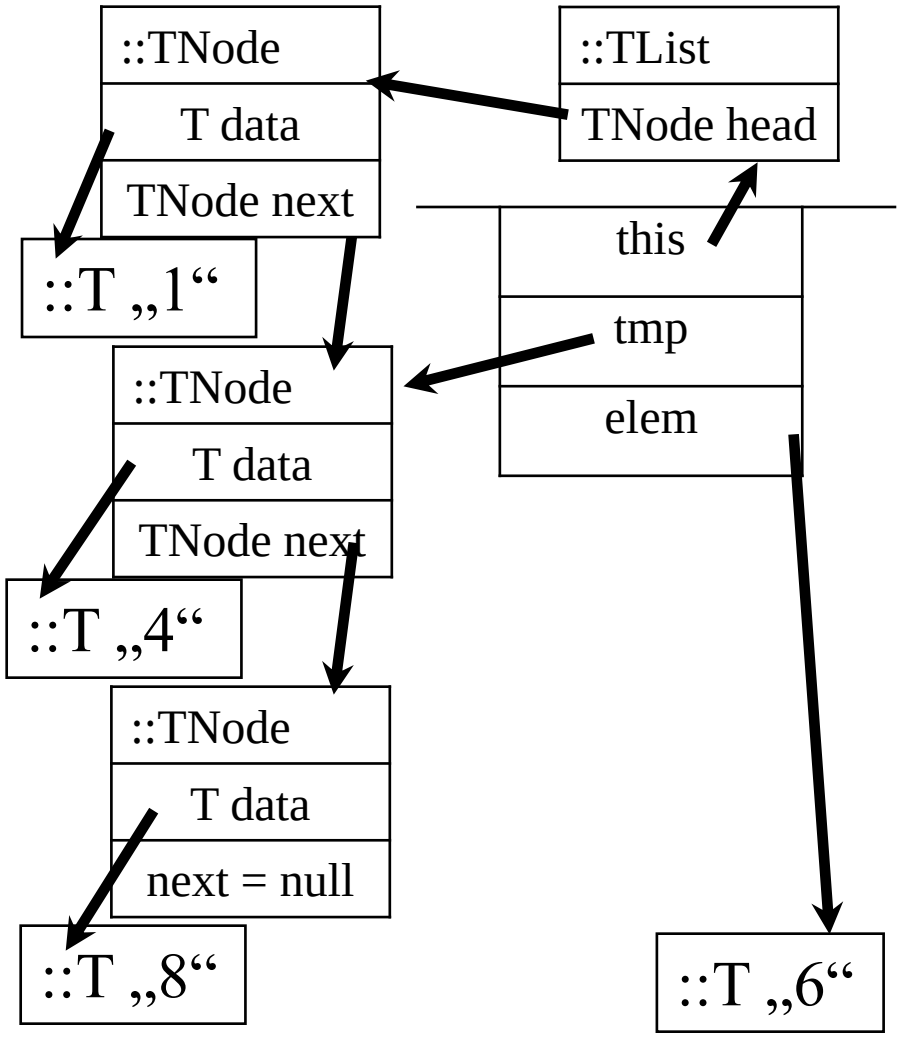
Typische Operationen auf Listen



- **Sortiertes Einfügen: Code**

```
public class TList { // [..]
    /**
     * Fügt elem sortiert in die
     * aufsteigend sortierte Liste ein.
     */
    public void insertSorted(T elem) {
        TNode tmp = head;
        // Spezialfall: Liste ist leer oder
        // neues Element ist kleinstes
        if( (head == null) ||
            (head.data.compareTo(elem) > 0))
        { insertFirst(elem); return; }
        // Allgemeinfall: an die richtige
        // Stelle in der Liste gehen
        while( (tmp.next != null) &&
               (tmp.next.data.compareTo(elem) <= 0))
            tmp = tmp.next;
        // neue Listenzelle erzeugen und einfügen
        tmp.next = new TNode(elem, tmp.next);
    }
}
```

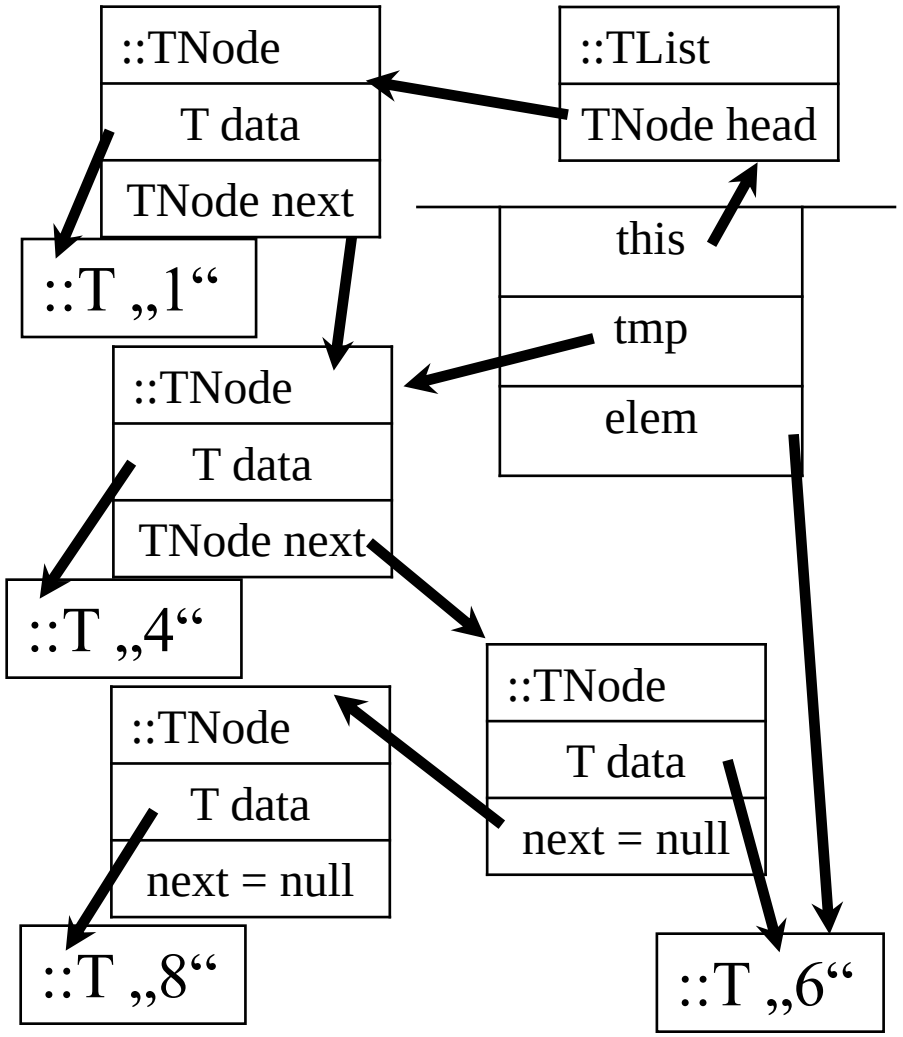
Typische Operationen auf Listen



- **Sortiertes Einfügen: Code**

```
public class TList { // [..]
    /**
     * Fügt elem sortiert in die
     * aufsteigend sortierte Liste ein.
     */
    public void insertSorted(T elem) {
        TNode tmp = head;
        // Spezialfall: Liste ist leer oder
        // neues Element ist kleinstes
        if( (head == null) ||
            (head.data.compareTo(elem) > 0))
        { insertFirst(elem); return; }
        // Allgemeinfall: an die richtige
        // Stelle in der Liste gehen
        while( (tmp.next != null) &&
               (tmp.next.data.compareTo(elem) <= 0))
            tmp = tmp.next;
        // neue Listenzelle erzeugen und einfügen
        tmp.next = new TNode(elem, tmp.next);
    }
}
```

Typische Operationen auf Listen



- **Sortiertes Einfügen: Code**

```
public class TList { // [..]
    /**
     * Fügt elem sortiert in die
     * aufsteigend sortierte Liste ein.
     */
    public void insertSorted(T elem) {
        TNode tmp = head;
        // Spezialfall: Liste ist leer oder
        // neues Element ist kleinstes
        if( (head == null) ||
            (head.data.compareTo(elem) > 0))
        { insertFirst(elem); return; }
        // Allgemeinfall: an die richtige
        // Stelle in der Liste gehen
        while( (tmp.next != null) &&
               (tmp.next.data.compareTo(elem) <= 0))
            tmp = tmp.next;
        // neue Listenzelle erzeugen und einfügen
        tmp.next = new TNode(elem, tmp.next);
    }
}
```

Typische Operationen auf Listen

- Rekursive Version von **`insertSorted`**
- Struktur
  - Vergleiche das erste Element der Liste (**`head.data`**) mit dem einzufügenden Element **`elem`**
  - Falls die Liste leer ist oder das neue Element kleiner ist, dann füge das neue Element als erstes in die Liste ein
  - Andernfalls füge das neue Element sortiert in die neue, um den Kopf verkürzte Liste **`tmpList`** ein (rekursiver Aufruf)
    - und hänge die neue Liste zuletzt wieder an den Kopf der ursprünglichen Liste an

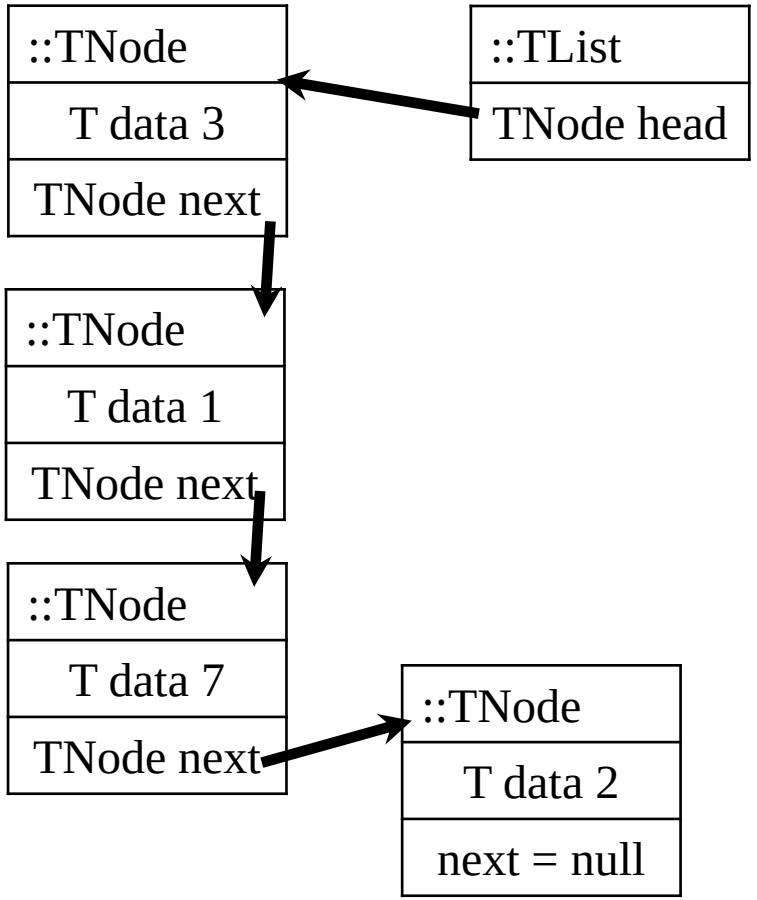
Typische Operationen auf Listen

- Da es für rekursive Aufrufe von Listenfunktionen ineffizient wäre, Knotenketten in Listen zu verpacken, benutzen wir eine private Hilfsmethode, die direkt auf einer Knotenkette arbeitet

```
public class TList { // [..]
    /**Fügt elem sortiert in die aufsteigend sortierte
     * Liste ein. Rekursive Version der Methode
     */
    public void insertSortedRec(T elem)
    { head=insertSortedNodeChain(head, elem); }
    private TNode insertSortedNodeChain (TNode chain,
                                         T elem) {
        // special case: chain is empty
        // or new elem is smallest
        if( (chain == null) ||
            (chain.data.compareTo(elem)>0))
        { return new TNode(elem, chain); }
        // recursion: advance in chain
        chain.next=insertSortedNodeChain(chain.next, elem);
        return chain;
    }
}
```

Typische Operationen auf Listen

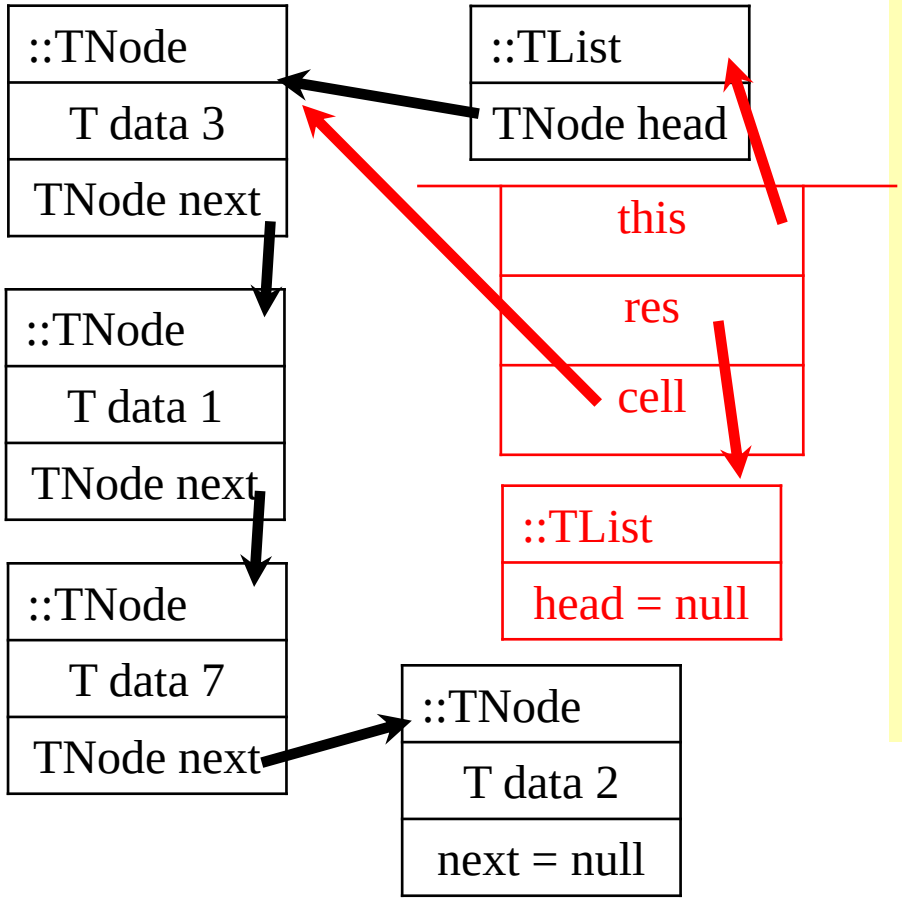
- **Konstruktives Invertieren**




```
public class TList { // [..]
    /**
     * Gibt die konstruktiv invertierte
     * Liste zurück.
     */
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;

        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

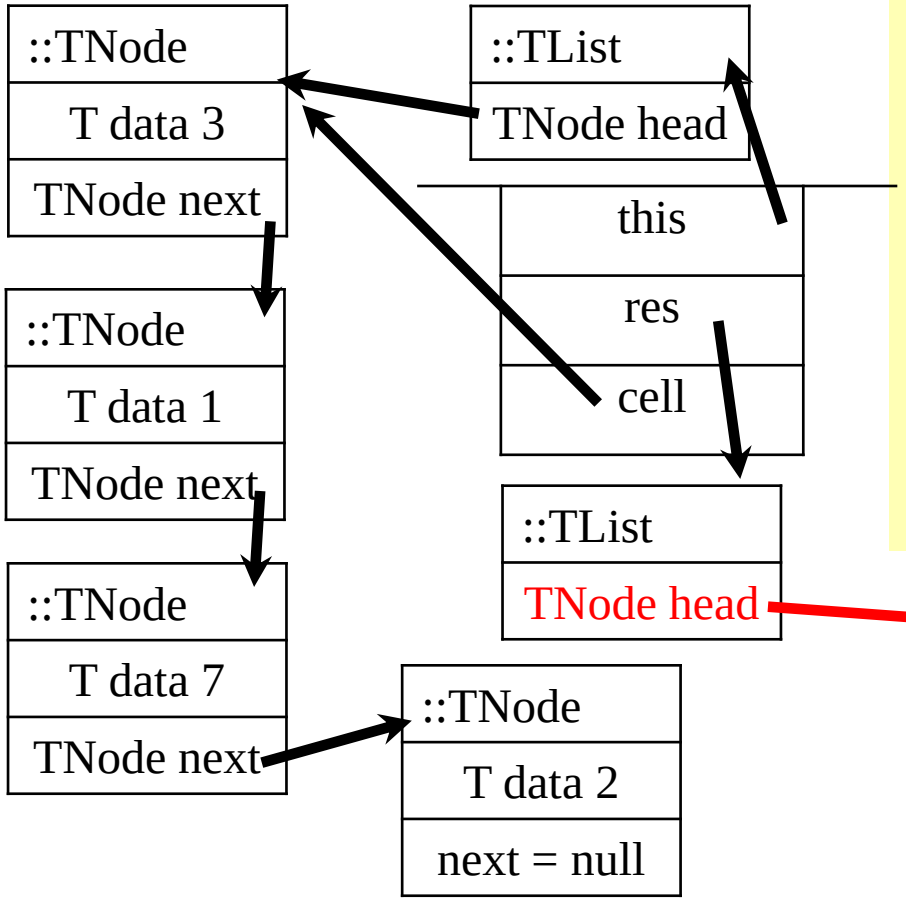
- **Konstruktives Invertieren**




```
public class TList { // [..]
    /**
     * Gibt die konstruktiv invertierte
     * Liste zurück.
     */
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;

        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

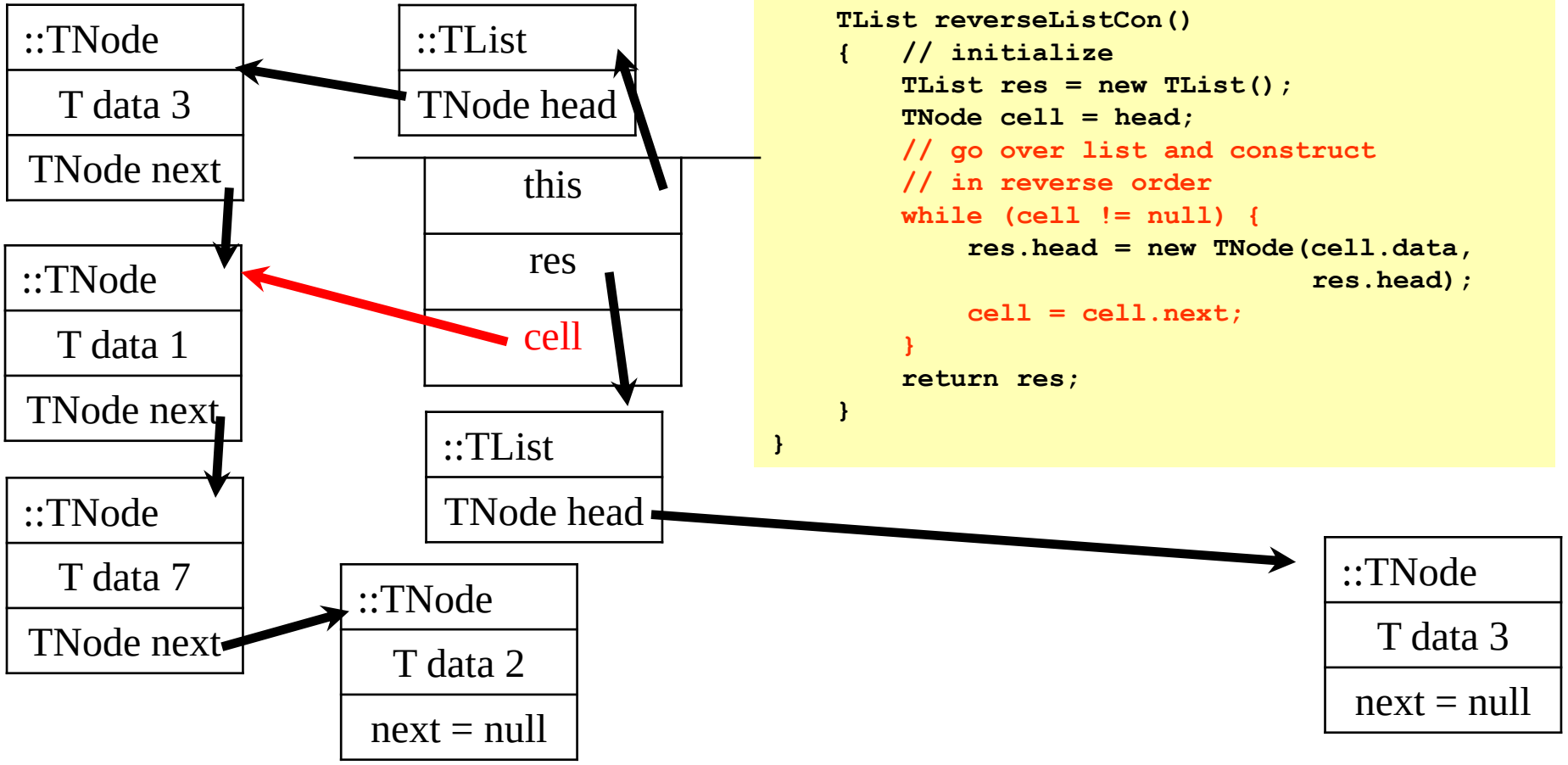
- **Konstruktives Invertieren**




```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

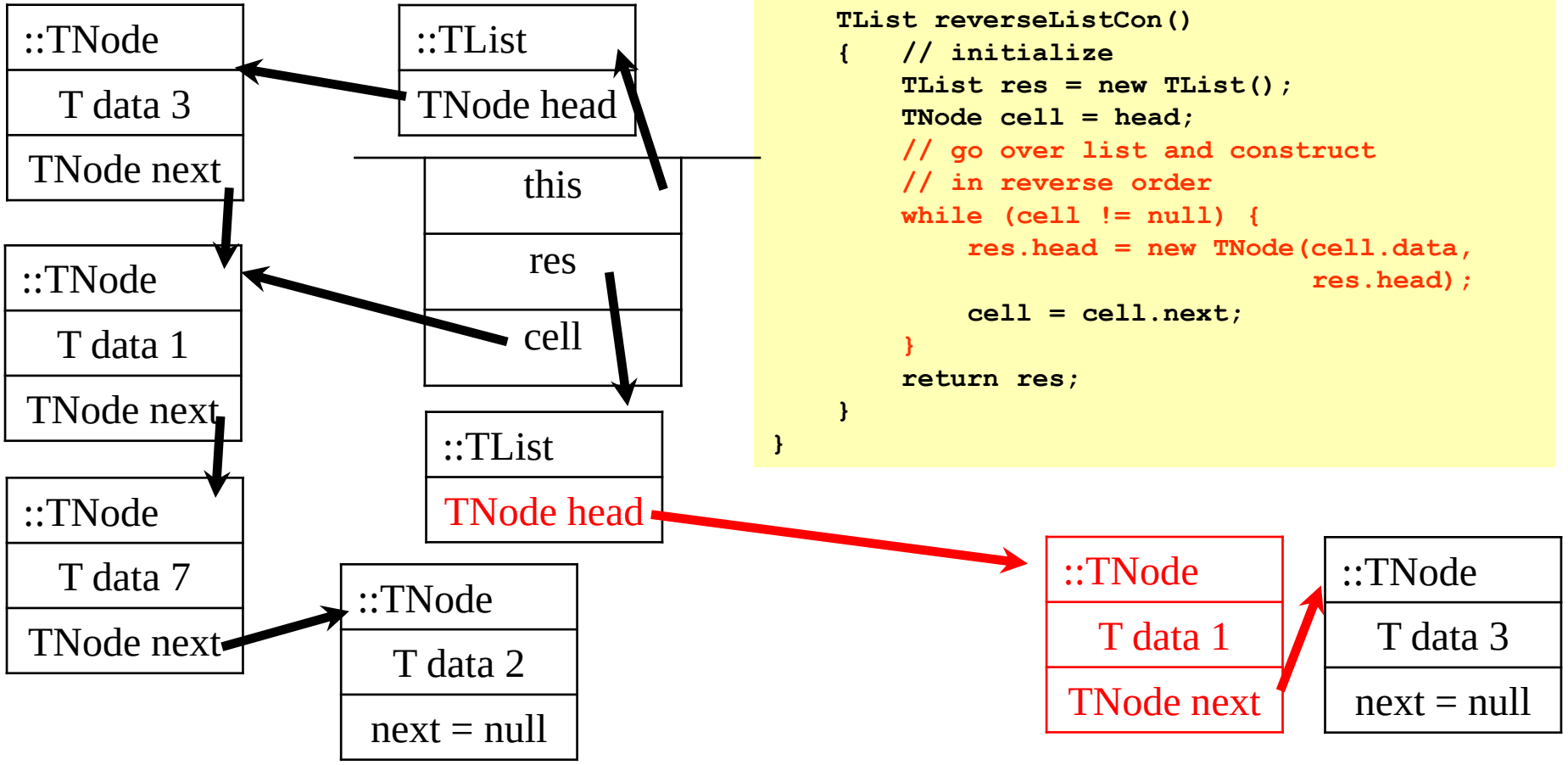
- **Konstruktives Invertieren**




```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

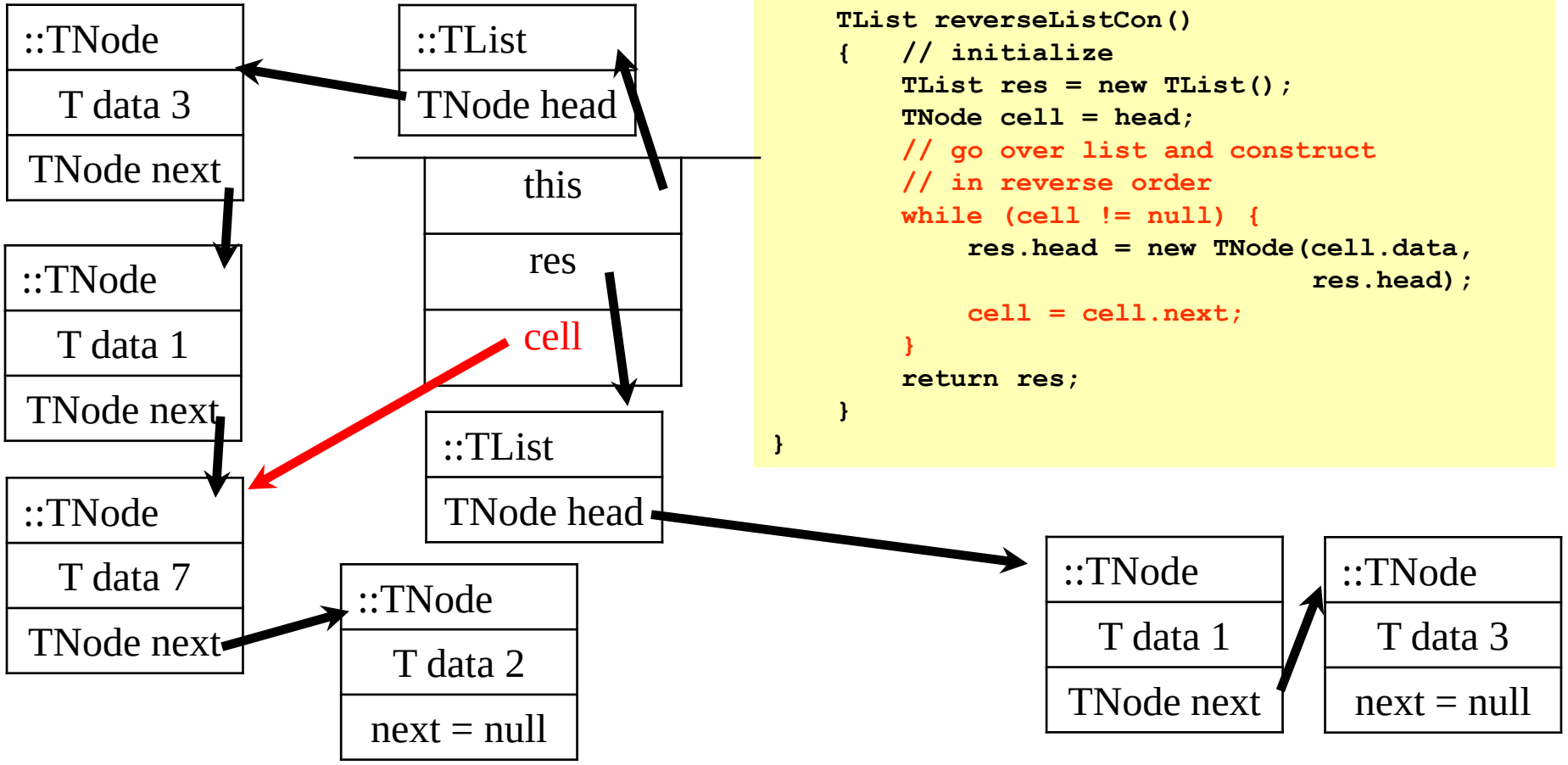
- **Konstruktives Invertieren**




```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

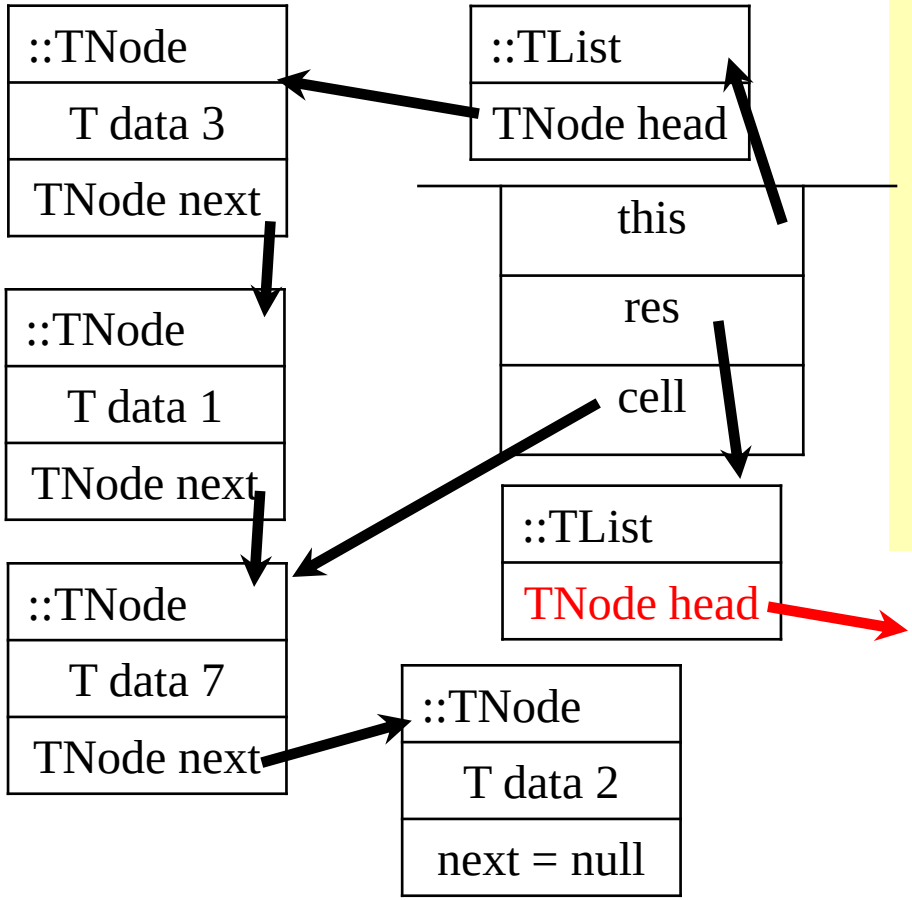
- **Konstruktives Invertieren**



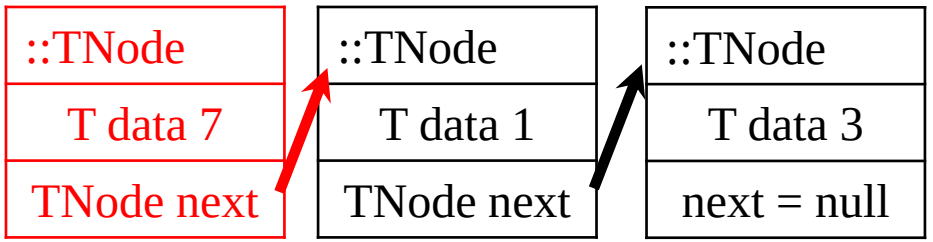

```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

- **Konstruktives Invertieren**

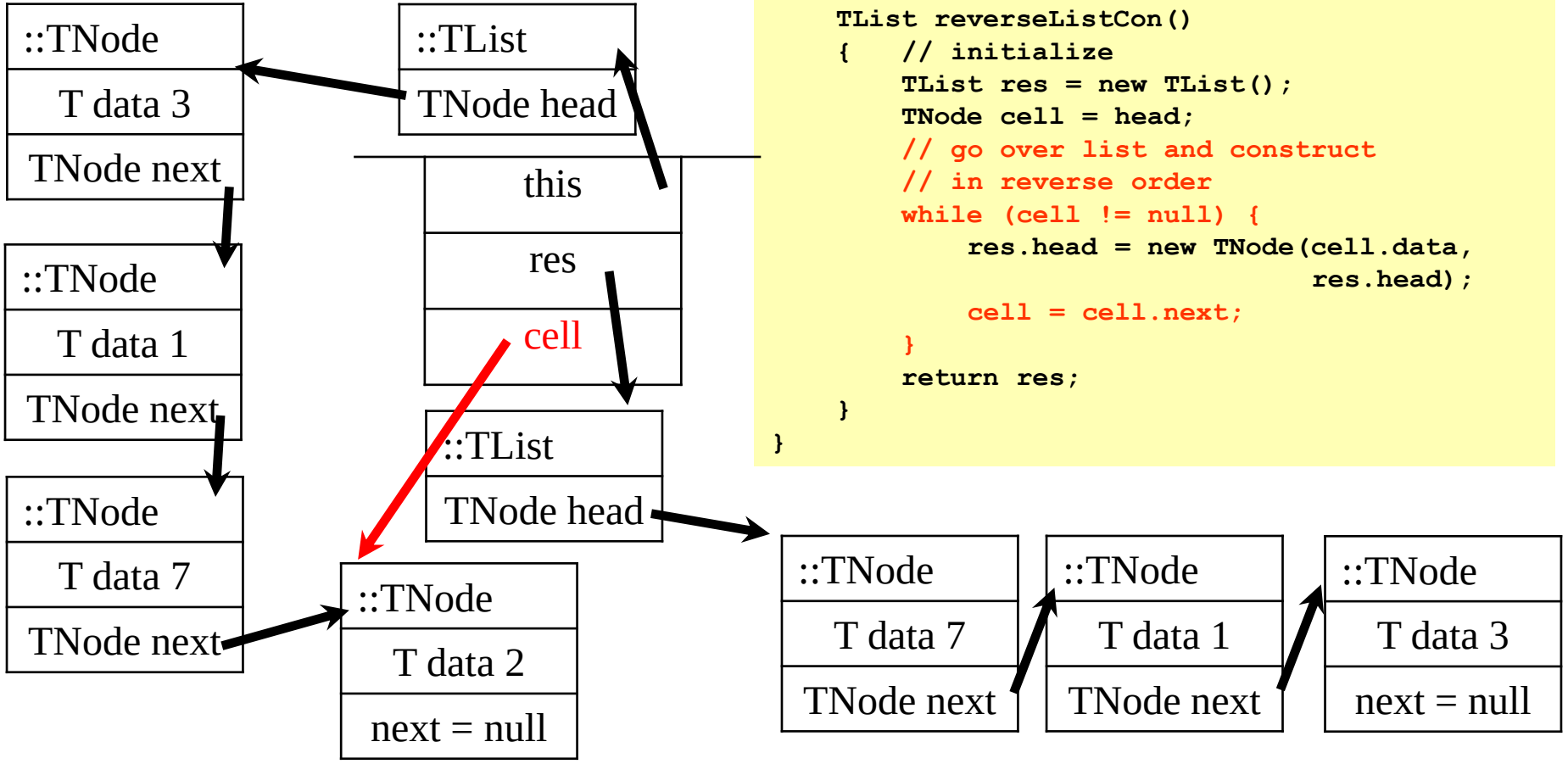



```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```



Typische Operationen auf Listen

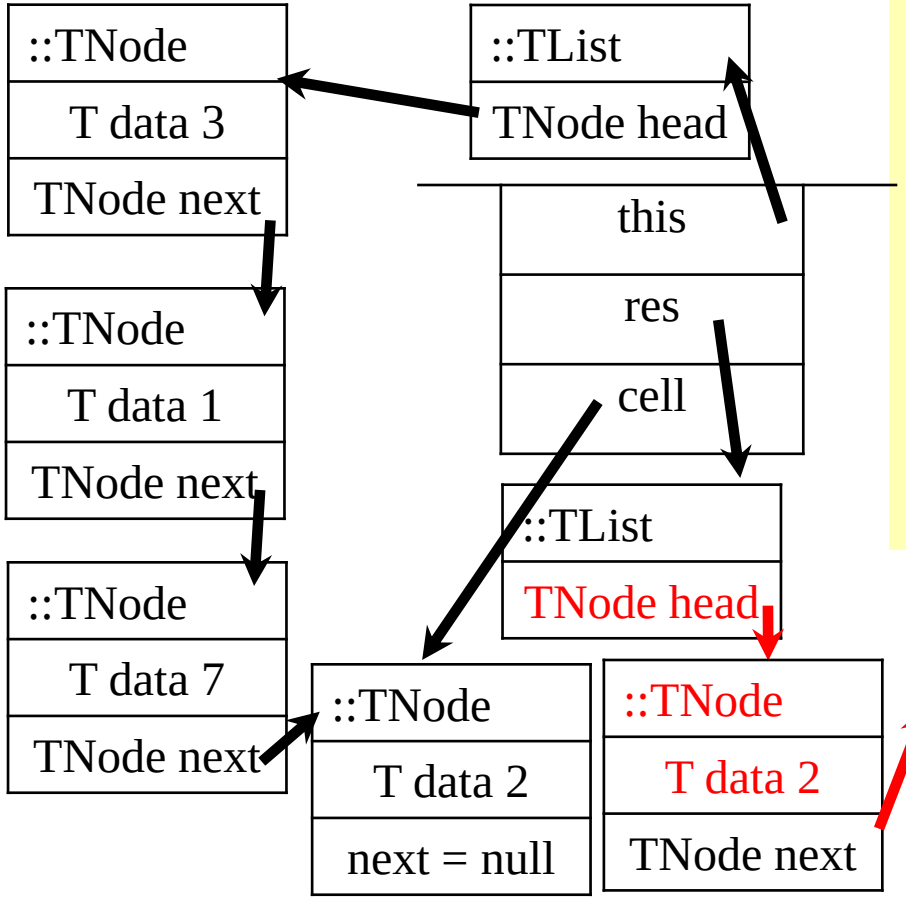
- **Konstruktives Invertieren**



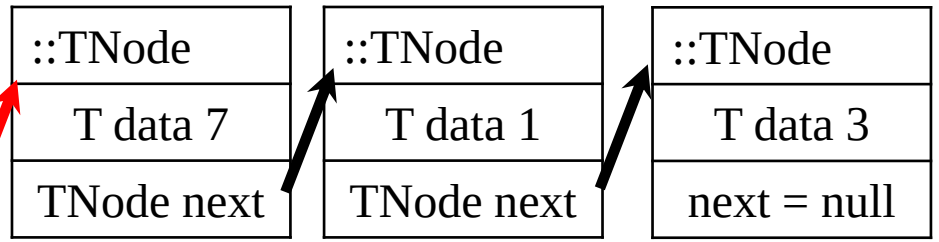

```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

- **Konstruktives Invertieren**

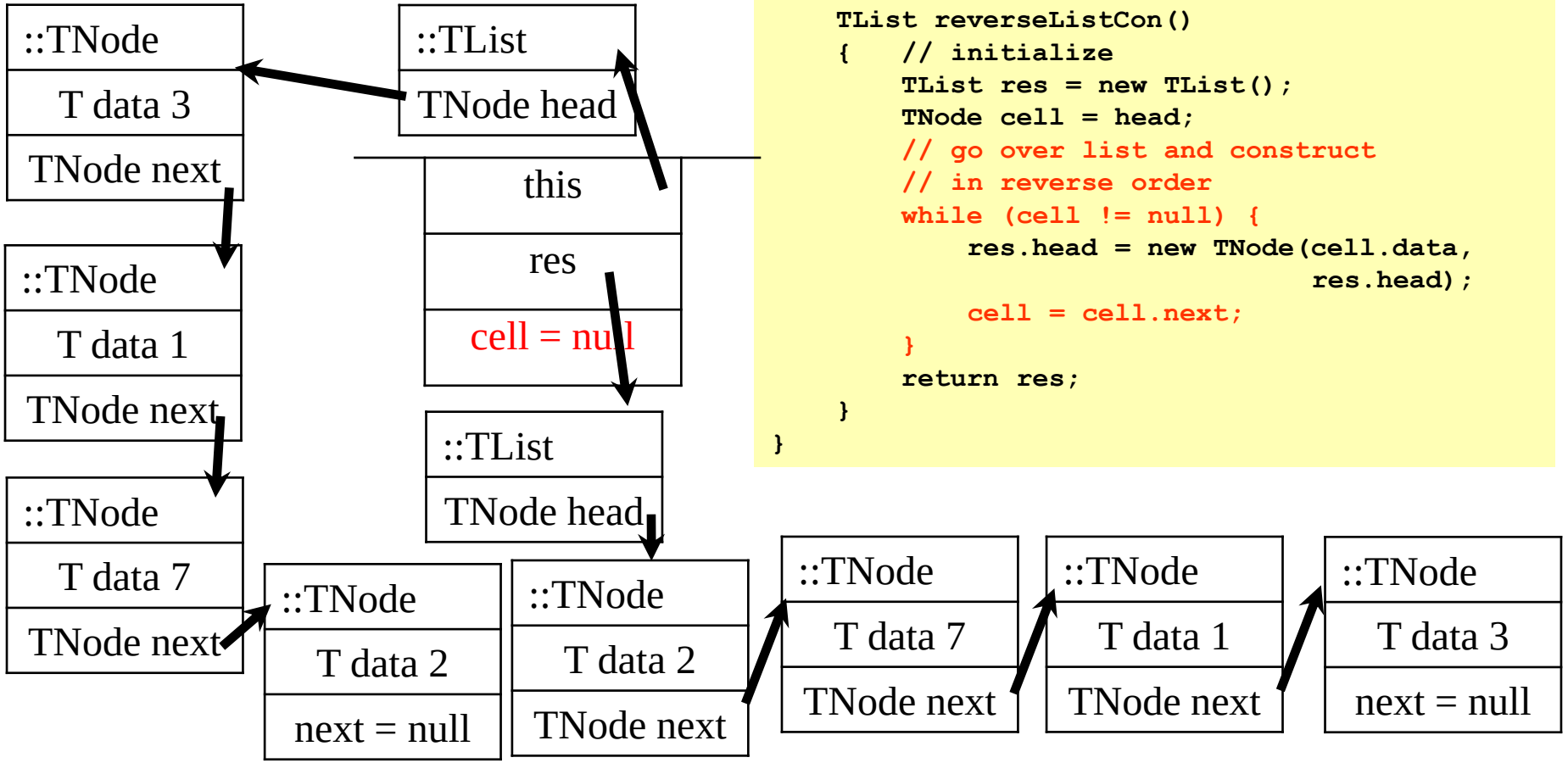



```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```



Typische Operationen auf Listen

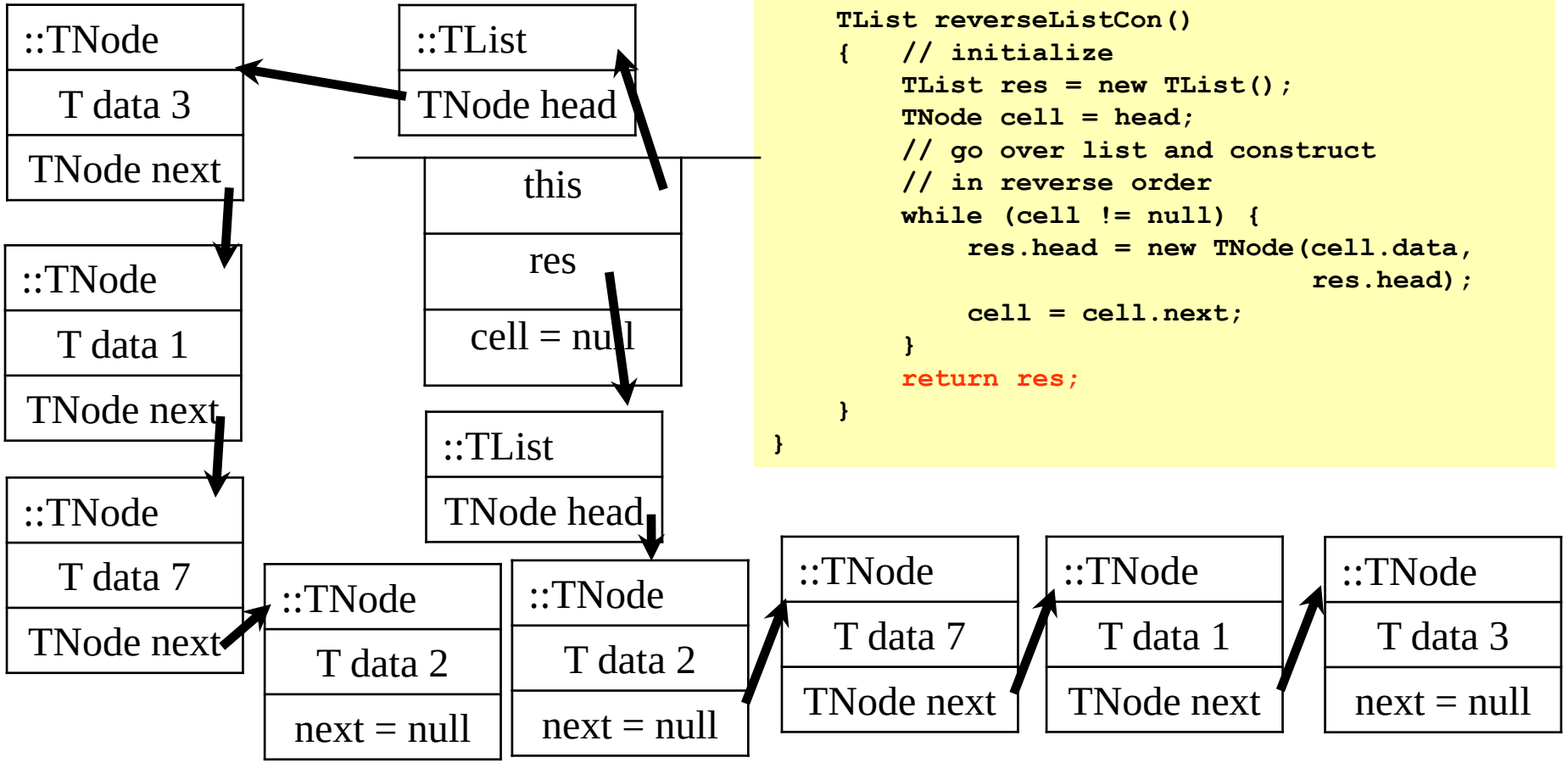
- **Konstruktives Invertieren**




```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

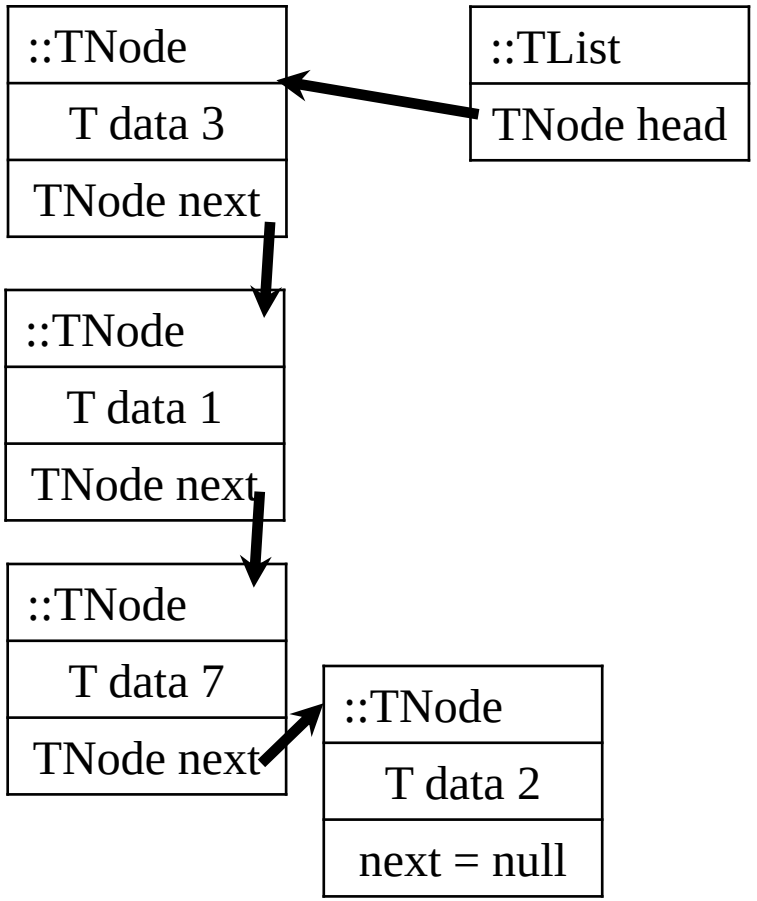
- **Konstruktives Invertieren**




```
public class TList { // [..]
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;
        // go over list and construct
        // in reverse order
        while (cell != null) {
            res.head = new TNode(cell.data,
                                 res.head);
            cell = cell.next;
        }
        return res;
    }
}
```

Typische Operationen auf Listen

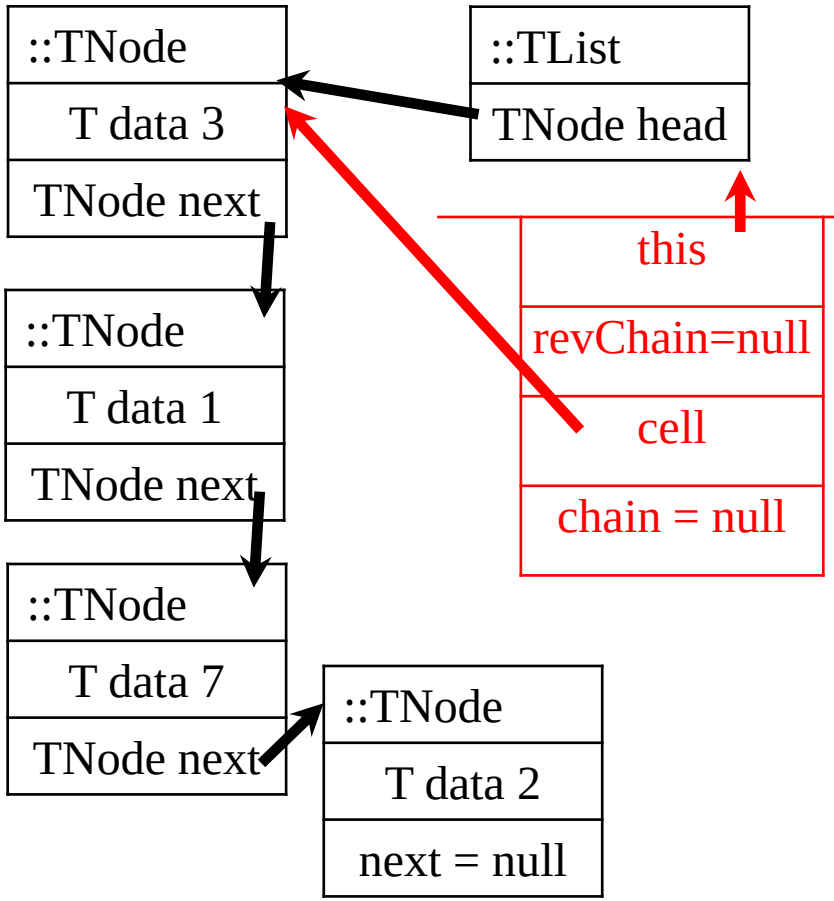
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

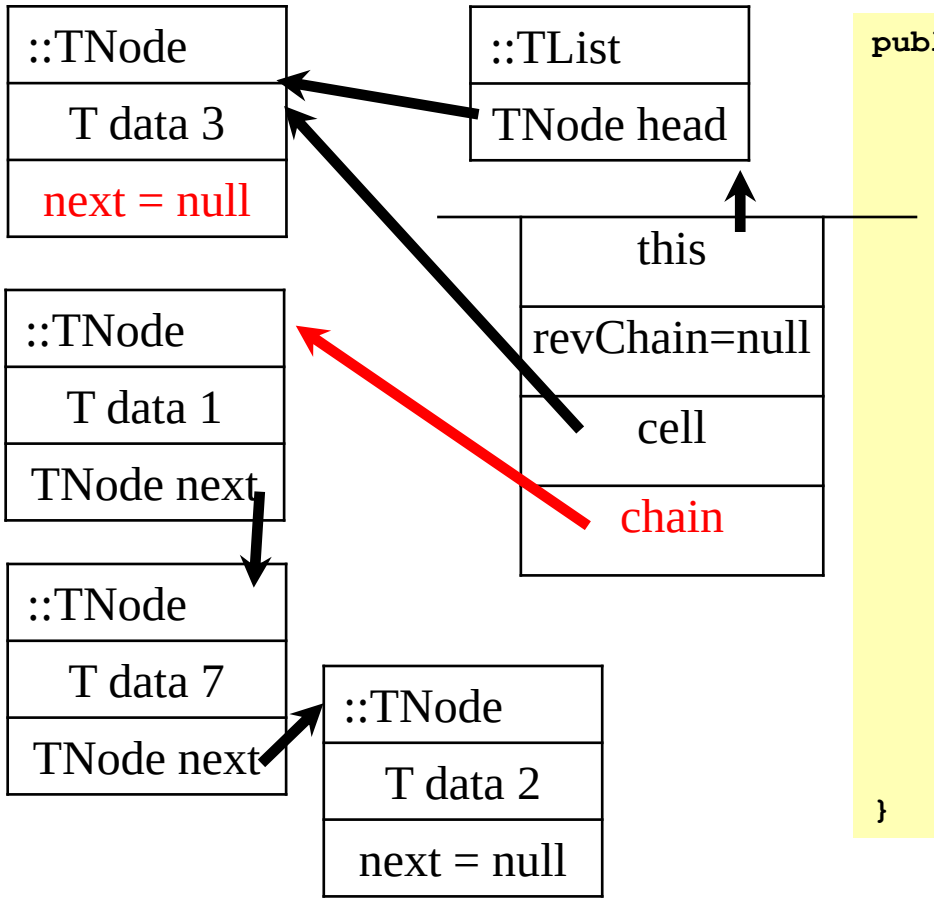
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

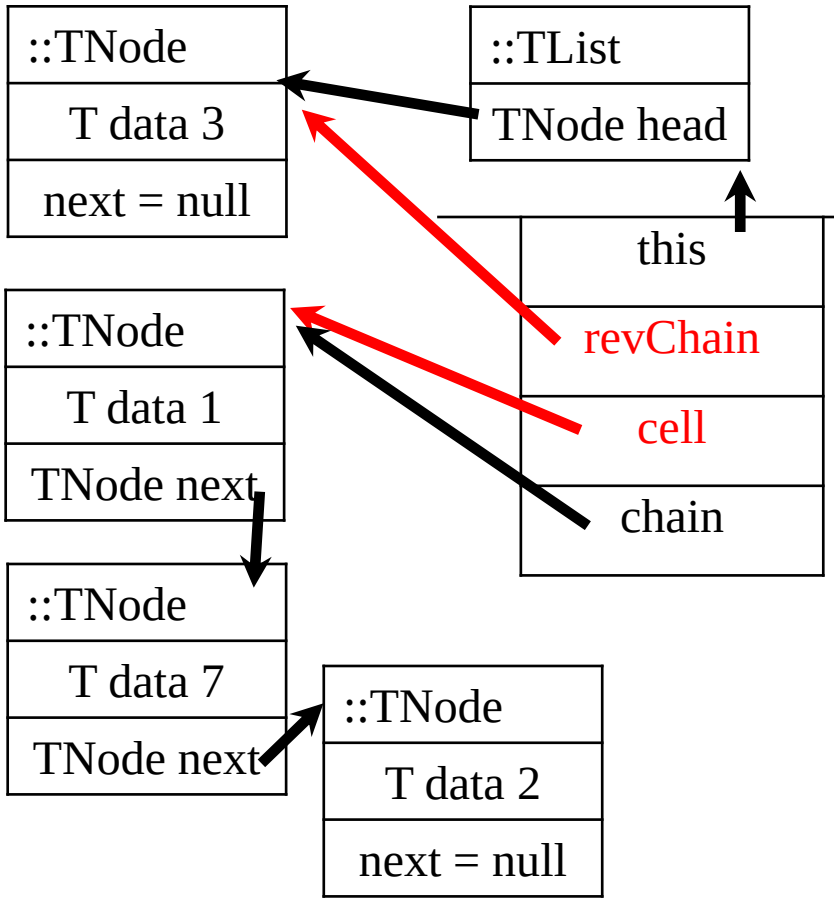
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

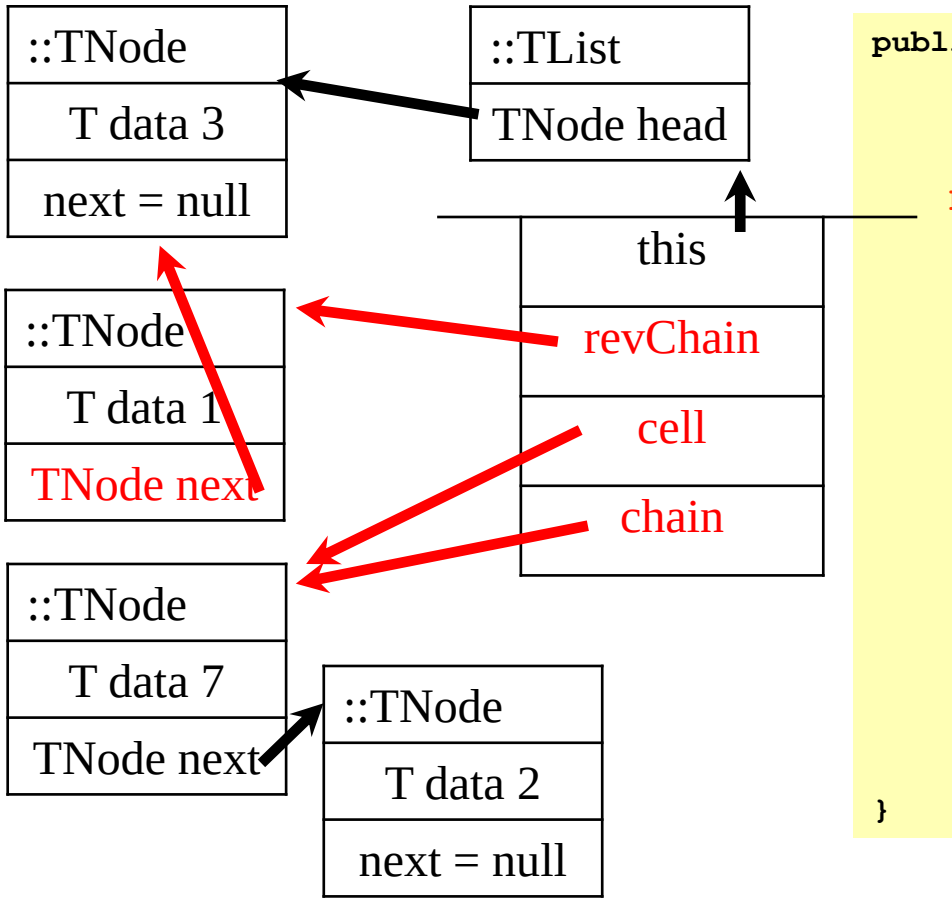
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

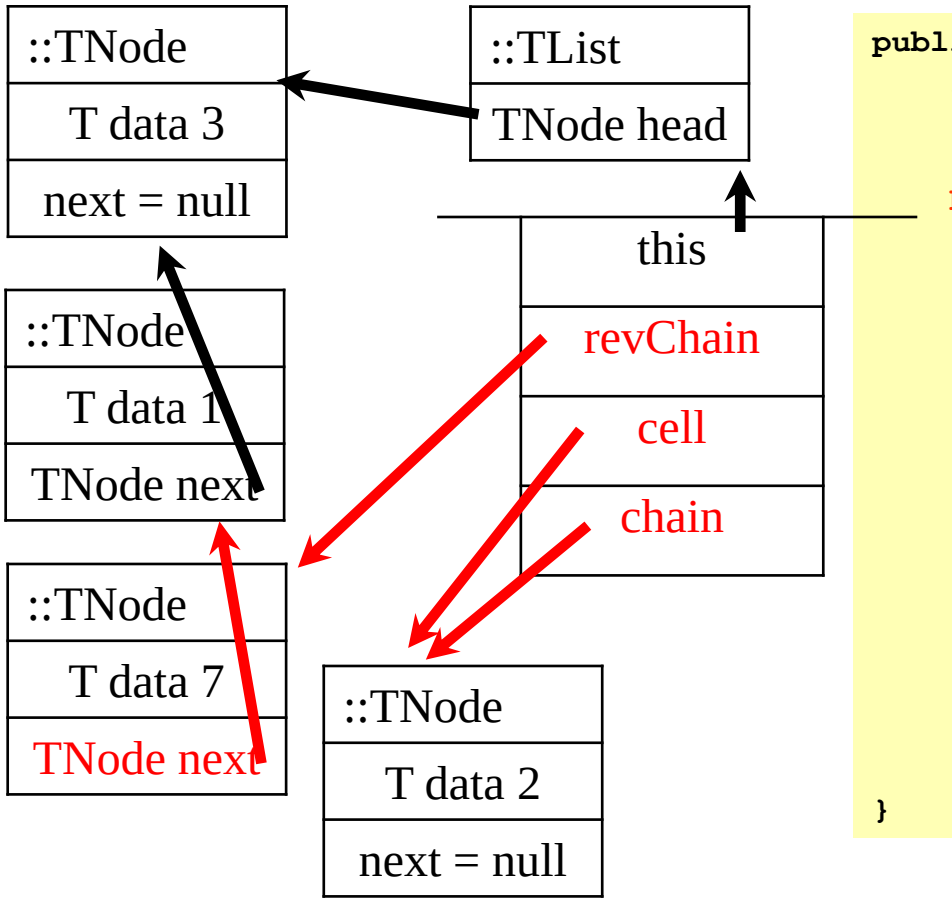
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

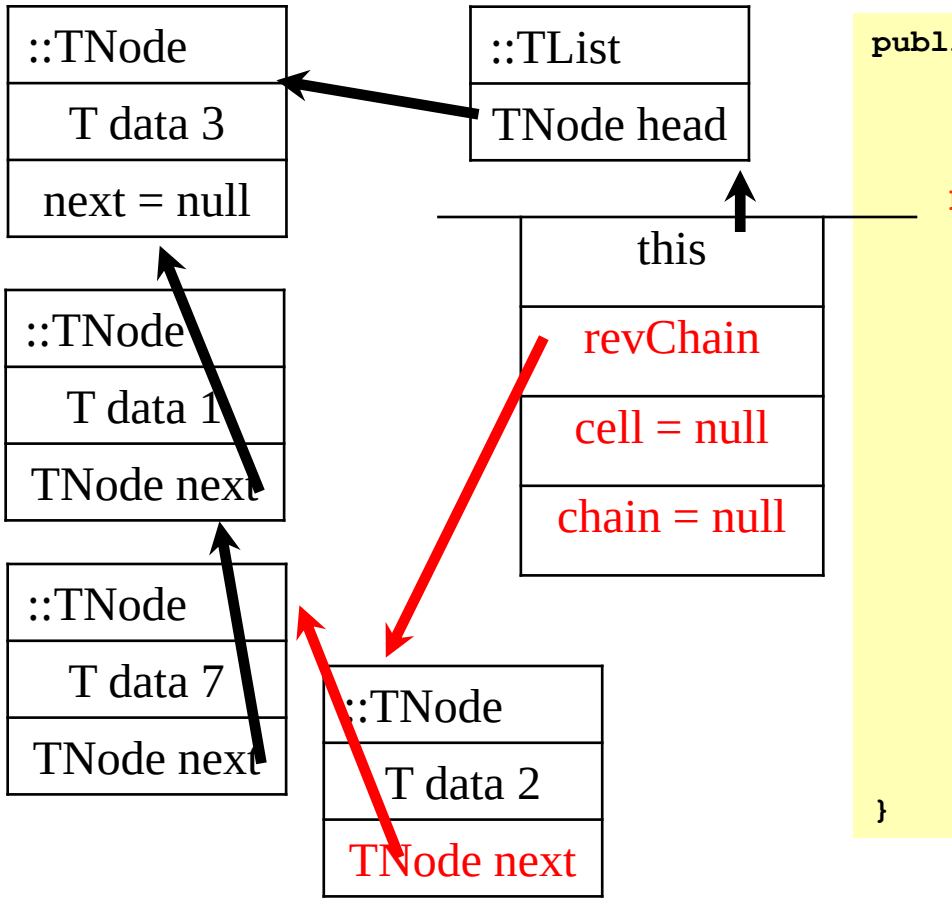
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

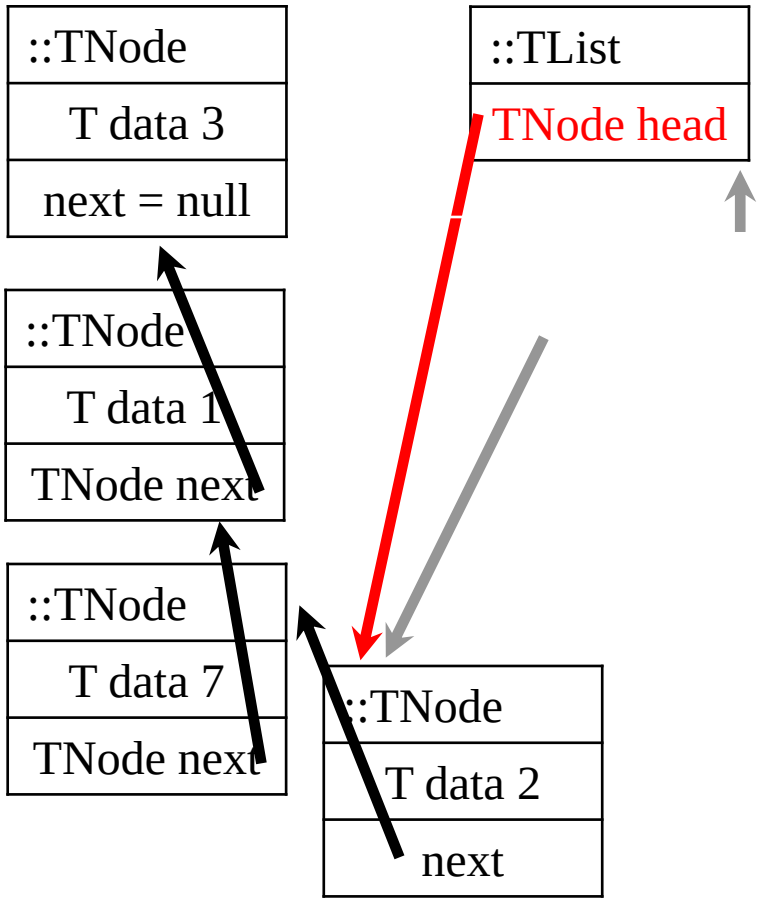
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

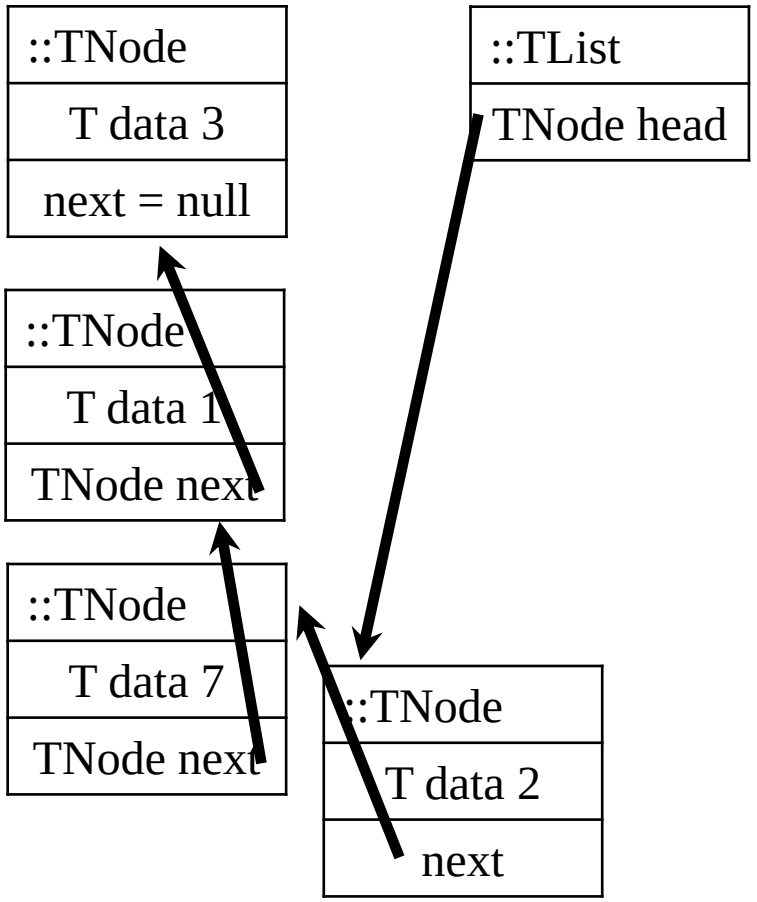
- **Destruktives Invertieren**




```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Typische Operationen auf Listen

- **Destruktives Invertieren**

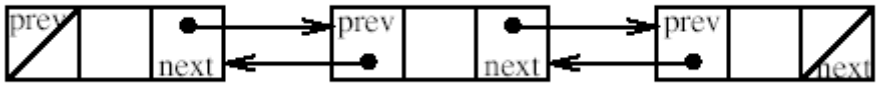



```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to transfer
        TNode chain;           // chain to be reversed
        while (cell != null) {
            // set chain to remaining node chain
            chain = cell.next;
            // reverse next pointer in cell
            cell.next = revChain;
            // add cell to revChain
            revChain = cell;
            // set cell to next cell of chain
            cell = chain;
        }
        head = revChain;
    }
}
```

Doppelt verkettete Listen

- Doppelt verkettete Listen bestehen aus Listenzellen mit zwei Zeigern
  - ein Zeiger **`prev`** auf die vorherige Listenzelle,
  - ein Zeiger **`next`** auf die nächste Listenzelle

```
public class TDNode
{
    T data;
    TDNode prev;  // Vorgängerknoten
    TDNode next;  // Nachfolgerknoten

    // Konstruktoren
    public TDNode(T a){
        data=a; prev = null; next = null;
    }
    public TDNode(T a, TDNode p, TDNode n) {
        data = a;
        prev = p;
        next = n;
    }
    // evtl. weitere Methoden, siehe unten
}
```



Doppelt verkettete Listen

- Container-Klasse für doppelt verkettete Listen

```
public class TDList
{
    private TDNode head;
    private TDNode last;

    // Konstruktoren
    public TDList() {
        head = null;
        last = null;
    }
    public TDList(T a) {
        head = new TDNode(a);
        last = head;
    }
}
```

Doppelt verkettete Listen

- Vorteile doppelt verketteter Listen
  - Einfügen am Ende sehr viel schneller möglich
    - Kein Durchlaufen durch die ganze Liste
      - Geht auch bei einfach verketteter Liste, wenn Container-Klasse zusätzlich Referenz auf Ende der Liste enthält
- Nachteile doppelt verketteter Listen
  - Höherer Speicherbedarf
    - Zwei Referenzen pro Listenzelle statt nur einer
  - Mehr Aufwand bei Listenmanipulation
    - Zwei Referenzen sind zu ändern statt nur einer

Vergleich: Listen und Arrays

- **Listen**
  - Vorteile
    - Einfügen neuer Elemente leicht möglich
    - Löschen leicht möglich
  - Nachteile
    - „Durchhangeln“ durch viele Elemente der Liste, um ein spezielles zu erreichen
      - Etwa das „Fünfte in der Liste“
    - Zusätzlicher Speicherbedarf
      - Eine Referenz pro Listenelement
- **Arrays**
  - Nachteile
    - Einfügen neuer Elemente erfordert „umkopieren“
    - Löschen von Elementen erfordert ebenfalls ein „umkopieren“
  - Vorteile
    - Wahlfreier Zugriff
    - Speicherbedarf nur für Daten
      - Nur insgesamt pro Array noch Speicher-Bedarf für ein `length`-Feld

Bemerkungen zur Verwendung von Referenzvariablen als Zeigervariablen

- Referenzvariablen nur etwas abstrakter als Zeigervariablen
  - Mit beiden sehr effiziente Realisierungen möglich
  - Jedoch in beiden ein „goto“ auf Datenstrukturen realisiert
    - Etwa Zyklen in Listen möglich
      - Terminierung von Listenalgorithmen deshalb nicht beweisbar
  - In funktionalen Sprachen können Listen abstrakter definiert werden
    - Diese schließen dann zum Beispiel Zyklen aus
- Keine „Pointerarithmetik“ mit Referenzvariablen möglich
  - Etwa „refvar + 1“
    - Erhöht Zuverlässigkeit und Sicherheit des Codes
- In Java ist Speicherbedarf einer Referenzvariable nicht vollständig festgelegt
  - Selbst im compilierten Java-Byte-Code noch nicht
    - Nur Mindestgröße von 32bit
    - Erst die Realisierung der Java Virtual Machine muss Festlegung treffen
  - Daher ist Java-Byte-Code auf 32bit- und 64bit-Architekturen ohne recompilieren lauffähig

universitätbonn

Stapel und Warteschlangen (Stacks and Queues)

Abstrakter Datentyp Stapel

- **Stapel** (Keller, Stack) sind Datenstrukturen zum Zwischenspeichern von Elementen, die nach dem **last-in**, **first-out** (**lifo**) Prinzip arbeiten
  - Die Elemente, die als letzte auf dem Stapel abgelegt wurden (mittels einer Operation push) sind die ersten, die zurückgegeben werden (mittels der Operationen top bzw. pop)
- Zum Beispiel arbeitet der von uns betrachtete **Laufzeitstapel** (Stack) zur Speicherung von Prozedur-Rahmen nach diesem Prinzip

Abstrakter Datentyp Stapel

- Intuition des LIFO-Prinzips kann wie folgt formalisiert werden

Definition 7.8.1. *Ein* ***Stapel*** *(stack) von Elementen vom Typ* `T` *ist ein abstrakter Datentyp, welcher drei Operationen unterstützt, die im allgemeinen* `push`*,* `top` *und* `pop` *genannt werden, und der das Element* ***emptystack*** *enthält.*

Die (partielle) Funktion `top` *hat einen Stack als Eingabeparameter und gibt ein Element zurück, die partielle Funktion* `pop` *hat einen Stack als Ein- und Ausgabeparameter. Die Funktion* `push` *hat einen Stack und ein Element als Eingabeparameter und gibt einen Stack zurück.*

Die Semantik von `top`*,* `pop` *und* `push` *wird (implizit) durch folgende Gleichungen gegeben, wobei* s *ein Stapel (von Elementen vom Typ* `T`*) und* `e` *ein Element vom Typ* `T` *sind:*

$$\begin{aligned} top(push(s, e)) &= e \\ pop(push(s, e)) &= s \end{aligned}$$

Funktionen top und pop partiell, nicht für alle Stapel definiert (nicht für emptystack)

„Algebraischer“ abstrakter Datentyp (da durch Gleichungen spezifiziert)

Abstrakter Datentyp Stapel

- Implementierung von Stapeln (Stacks)
  - Darstellung als Containerklasse, die
    - eine einfache Liste zur Speicherung der Elemente enthält
    - und neben Konstruktoren bloß Methoden enthält, welche auf das erste Element (den **Kopf**, bzw. den **Stack-Top**) zugreifen
      - Im wesentlichen sind das die Methoden **`insertFirst`** (oder **`push`**) und **`takeFirst`** (oder **`top`**)
- Bemerkung: Andere „gute“ Implementierungen für Stapel möglich
  - Array-basiert

Abstrakter Datentyp Stapel

Implementierung von Stacks: Code

```
public class TStack {
    private TNode top; //Stack-Top
    public TStack() { top = null; }

    /**
     * Legt c auf den Stack
     */
    public void push(T c) { top = new TNode(c, top); }

    /**
     * Gibt den Stack-Top zurück und
     * reduziert den Stack um Top
     * Anforderung: Stack ist nicht leer.
     */
    public T pop() throws EmptyStackException {
        if (top == null) throw new
                              EmptyStackException();
        T res = top.data;     // Stack-Top lesen
        top = top.next;       // Stack-Top verschieben
        return res;
    }

    public boolean empty() { return (top==null); }
}
```

Funktion pop ist partielle Funktion; daher Exception für Parameterwerte, für die pop nicht definiert ist

Warteschlangen (Queues)

- Ein Stapel verwirklicht einen Zwischenspeicher für Elemente nach dem last-in, first-out (lifo) Prinzip
- Das duale Konzept eines Zwischenspeichers, der nach dem first-in, first-out (fifo) Prinzip arbeitet, ist im abstrakten Datentyp einer **Warteschlange** (queue) verwirklicht
  - Die Elemente, die mittels einer Methode **`append`** als erste in eine Warteschlange eingereiht wurden, sind auch die ersten, die mittels einer **`get`**-Methode wieder gewonnen werden

Implementierung von Queues

```
public class TQueue {
    private TNode head, last;
    /**
     * Returns true iff the queue is empty
     */
    public boolean empty()
    { return head == null; }
    /**
     * Appends t to the queue
     */
    public void append(T t) {
        TNode p = new TNode(t);
        if ( last == null )
            head = p;
        else
            last.next = p;
        last = p;
    }
```

```
    /**
     * Returns the first element of the queue
     * and removes it from the queue.
     * @exception EmptyQueueException queue is empty
     */
    public T get() throws EmptyStackException {
        if (head == null)
            throw new EmptyStackException();

        TNode p = head;   // Remember first element
        head = head.next; // Remove it from queue
        if (head == null) // update last?
            last = null;
        return p.data;
    }
}
```

universitätbonn

Generische Datentypen und Generische Programmierung

TList versus generische Listen

- Idee zur Wiederverwendung unserer TList (TStack, TQueue, …)
  - Ersetze im Editor das T durch gewünschten Datentyp
- Vorteil:
  - Leicht möglich
  - Effizienter Code
    - Kein Mehraufwand zur Laufzeit zur Codeauswahl
      - Geschah zur „Editierzeit“ durch „Replace-and-Save“ im Editor
- Nachteil
  - Verschiedener Source-Code für alle bislang benötigten Instanzen von Listen
    - Nachträgliche Modifikationen müssen für alle Instanzen nachgeführt werden
    - Wartbarkeit des Codes wesentlich eingeschränkt

TList versus generische Listen

- Bei TList also Wartbarkeit des Codes erschwert
  - Nur empfehlenswert, wenn Effizienz sehr wichtig ist und nur sehr wenige (eine oder zwei) Instanzen benötigt werden
- Bessere Lösungen
  - (1) Verwende das T als sprachunterstützten Typ-Parameter
    - Wenn die Sprache dies zulässt
      - In Java seit Java 1.5
  - (2) Verwende gemeinsame Basisklasse für T
    - Und Laufzeitanpassungen
- Bemerkung
  - In Java wird (1) auf (2) durch Compiler reduziert

Generische Datentypen (1)

```
public class Node {
     MyData data;
     Node next;
     setData(MyData) …
}
```

```
Node n = new Node();
```

zu spezifisch, nicht wiederverwendbar für Daten eines anderen Typs

```
public class Node {
     Object data;
     Node next;
     setData(Object) …
}
```

```
Node n = new Node();
```

Sehr allgemein, nicht leicht verwendbar (Code muss lauter Casts enthalten)

- Ein Datentyp ist generisch, (1) wenn er hinsichtlich des Typs seiner Elemente parametrisiert ist, oder (2) über Elementen vom Typ Object definiert ist.

```
public class Node<T> {
     T data;
     Node next;
     setData(T) …
}
```

```
Node<MyData> n = new Node<MyData>();
```

generisch, wiederverwendbar für beliebige Typen T.

T wird erst bei der Variablendeklaration und Objekt-Instanziierung festgelegt

Generische Datentypen (seit Java 5)

- Typ-Parameter für Methoden, Klassen, Interfaces
  - Unterstützt List<E>, etc
- “Bounded polymorphism”
  - Wildcards (“?”)
  - void add(List<? extends Number>)
    - Listen von Elements, die Subtyp von Number sind
- Vorteile von (1) über (2)
  - Bessere statische Typ-Sicherheit
  - Bessere Lesbarkeit
    - Syntaktisch ähnlich zu C++-Templates

Generische Datentypen in Java: Beispiel

- Über Objekt (Typ 1)

```
List list = new ArrayList();
list.add("str");
String s = (String)list.get(0);     // <- cast necessary
…
Integer i = (Integer)list.get(0);   // <- cast + runtime error
```

- Mit Typparametern (Typ 2)

```
List<String> list = new ArrayList<String>();
list.add("str");
String s = list.get(0);
…
Integer i = list.get(0);            // <- compile-time error
```

Generische Datentypen mit Typparametern in Java: Einige generelle Informationen

- Erst seit Java 5.0 in Sprache definiert
  - D.h. Java 1.5
- Änderungen in JDK Standardbibliotheken:
  - Collections und Iterators sind generisch
- Keine Änderungen am Class File Format oder der JVM
  - Compiler führt Typcheck für Typparameter und eliminiert sie
    - Einfügen von cast etc.
  - Parametrisierte Typen werden nicht Makro-expandiert (wie dies bei C++-templates der Fall ist!)
  - Einige Einschränkungen (arrays, instanceof) wegen beschränkter Laufzeitunterstützung*

* **Gilad Bracha**, „Generics in the Java Programming Language“ java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Generische Vergleiche in Java Standard-Bibliotheken

- Bemerkungen
  - In den Java Standard-Bibliotheken wird häufig das Interface Comperator zugrunde gelegt
    - Ähnlich zu Comparable, spezifiziert jedoch, dass die Ordnung total ist
  - Die Methoden zum Finden eines Minimums (Maximums, Sortieren, …) werden als statische Methoden deklariert
    - Mit der generischen Liste (bzw. Collection) als erstem Element
      - public static <T extends Object & Comparable<? super T>> T **min**(Collection<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements. All elements in the collection must implement the Comparable interface. Furthermore, all elements in the collection must be *mutually comparable* (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the collection).This method iterates over the entire collection, hence it requires time proportional to the size of the collection.
      - **sort(List<T>, Comparator<? super T>)** - Static method in class java.util.Collections Sorts the specified list according to the order induced by the specified comparator.

universitätbonn

Funktionen höherer Stufe in Java

Simulationen Funktionen höherer Stufen in Java

Aktionsobjekte

universitätbonn

Bäume

Listen und Bäume
Graphen und Bäume,
elementare Eigenschaften von Binärbäumen
Implementierung
Generische Baumdurchläufe

Grundkonzepte von Bäumen

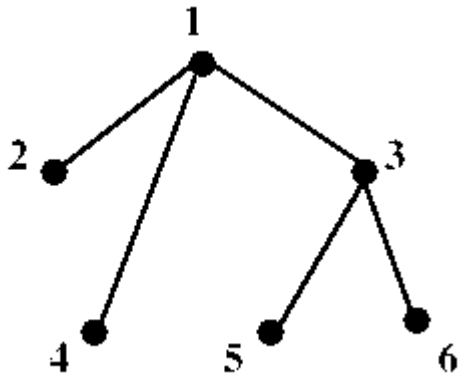
- Bäume (trees) können als eine Verallgemeinerung von Listen angesehen werden
  - Bei einer Liste hat jeder Knoten einen Nachfolger
  - Bei einem Baum hat jeder Knoten potentiell mehrere „Nachfolger“
  - Kinderknoten (child, children) genannt
  - Wie bei einer Liste hat jeder Knoten (bis auf den Kopfknoten) genau einen Vorgänger
  - Vorgänger eines Kindknotens wird Elternknoten (parent) genannt

Grundkonzepte von Bäumen

- Ein Knoten ohne Elternknoten wird Wurzel (root) genannt
  - Bei endlichen Bäumen gibt es stets Wurzeln
  - Als Baum wird i.A. Datenstruktur genannt, die genau eine Wurzel besitzt
  - Sonst wird von „Wald“ (forest) gesprochen
- Knoten ohne Kinderknoten heißen Blätter oder Terminalknoten (leaf, leaves)
- Knoten mit Kinderknoten heißen auch innere Knoten (inner nodes)
- Jedem Knoten ist eine **Ebene** (level) im Baum zugeordnet
  - Die Ebene eines Knotens ist die Länge des Pfades von diesem Knoten bis zur Wurzel
- Die **Höhe** (height) eines Baums ist die maximale Ebene, auf der sich Knoten befinden

Grundkonzepte von Bäumen

- Bäume kann man dadurch visualisieren, dass Verbindungen zwischen Eltern und ihren Kindern eingezeichnet werden
  - In der Informatik zeichnet man Bäume üblicherweise von der Wurzel abwärts
- Bei dem rechts dargestellten Baum ist der Knoten 3 ein Elternknoten der Knoten 5 und 6
  - Die Knoten 2, 3 und 4 sind Kinder von 1
  - Bei diesem Baum liegt der Knoten mit Etikett 1 auf Ebene 0, die Knoten 2, 3 und 4 auf Ebene 1 und die Knoten 5 und 6 auf Ebene 2
  - Die Höhe des Baumes ist 2



Grundkonzepte von Bäumen

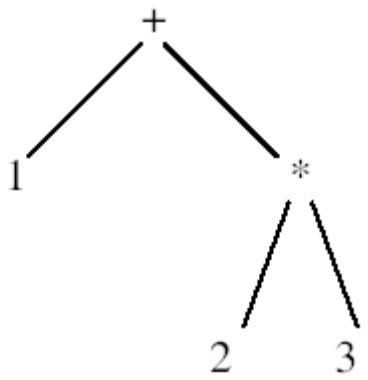
- Anwendungen sind vielfältig. Beispiele:
  - Organisationshierarchien in Unternehmen und Behörden
  - Aufrufstruktur von rekursiven Algorithmen wie etwa divide-and-conquer-Verfahren
  - Mögliche Züge in einem Zweipersonenspiel (z.B. Schach, Mühle)
  - Struktur eines mathematischen oder (programmiersprachlichen) Ausdrucks
  - Hierarchische Unterteilung von geometrischen Objekten oder vom Räumen
  - Struktur von Sequenzen von Entscheidungen in strategischen Untersuchungen
  - ...

Grundkonzepte von Bäumen

- Der **Verzweigungsgrad** (out degree) eines Knotens ist die Anzahl seiner Kinder
- Ein **Binärbaum** (binary tree) ist ein Baum, dessen Knoten höchstens den Verzweigungsgrad 2 haben
  - Der Baum des vorigen Beispiels ist kein Binärbaum, da die Wurzel Verzweigungsgrad 3 hat

Grundkonzepte von Bäumen

- Ausdrücke können durch Strukturbäume (parse trees) gut repräsentiert werden
  - Rekursive Definition spiegelt sich in Eltern-Kinderknoten-Verhältnis wieder
  - Beispiel: Strukturbaum von 1+2*3
    - Oder von 1+(2*3)

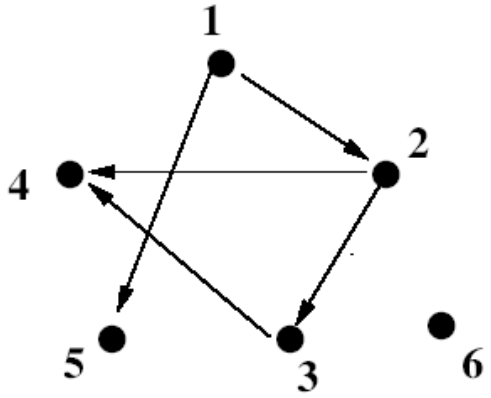


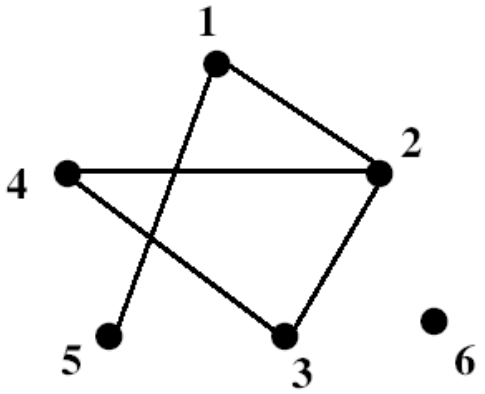
Grundkonzepte von Bäumen

- **Bemerkung:** Wir definieren hier Bäume als Verallgemeinerungen von Listen
  - Man kann Bäume aber auch als einen *Spezialfall* einer anderen für die Informatik sehr wichtigen Datenstruktur definieren, nämlich der eines Graphen
    - Ein (gerichteter) **Graph** (directed graph) ist ein Paar (V,E) bestehend aus einer nicht leeren Menge V von Ecken oder **Knoten** (vertices, nodes) und einer Menge E von **Kanten** (edges)
      - Dabei ist die Menge der Kanten E eine binäre Relation auf *V,* also $E \subseteq V \times V$
        - Ein Knoten kann ein **Etikett** (label) und weitere Informationen enthalten
      - Die Kanten können als Verbindungen zwischen den als kleine Kreise dargestellten Ecken visualisiert werden
      - Die Richtung der Kanten wird i. a. durch einen Pfeil angegeben

Graphen und Bäume

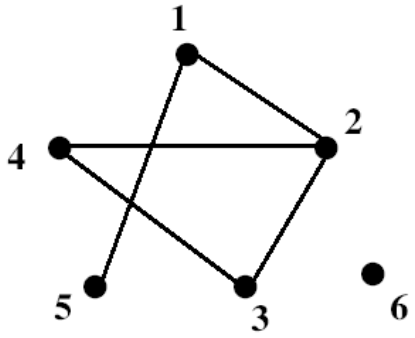
- Ungerichtete Graphen:
  - alle Kanten paarweise $((v_1,v_2) \in E \Leftrightarrow (v_2,v_1) \in E)$
  - es reicht nur eines der Paare anzugeben
- Beispiele gerichteter und ungerichteter Graphen
  - V = {1,2,3,4,5,6}, E = {(1,2), (2,3), (3,4), (2,4), (1,5)}
  - a) als gerichteter Graph, b) als ungerichteter Graph





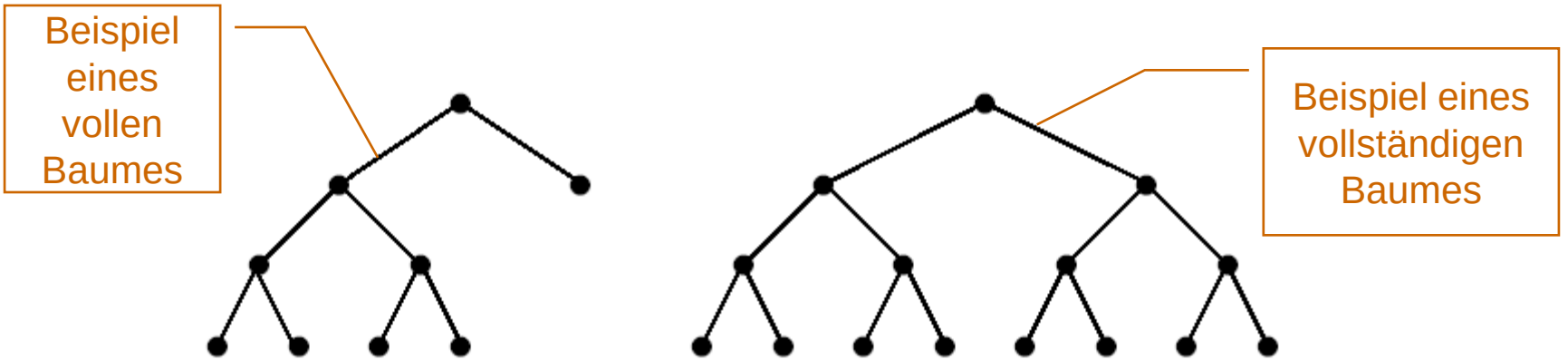
Graphen und Bäume

- Ein Pfad (path) ist eine Folge von verschiedenen Knoten, die durch Kanten verbunden sind
- Fakt: Ein (ungerichteter) Graph ist genau dann ein Baum, wenn es zwischen je zwei beliebigen Knoten genau einen Pfad gibt
  - Ist der Graph von letzter Folie ein Baum?
  - Beweis der Aussage (Übung)



Eigenschaften von Bäumen

- Ein Binärbaum heißt **voll**, falls alle inneren Knoten den Verzweigungsgrad 2 haben
- Ein voller Binärbaum heißt **vollständig**, falls alle Blätter den gleichen Level haben



Eigenschaften von Bäumen

Lemma: *Ein Baum mit n Knoten hat n-1 Kanten*

- **Beweis:** Jede Kante verbindet einen Knoten mit dem Elternknoten. Außer dem Wurzelknoten ist jeder Knoten durch eine Kante mit seinem Elternknoten verbunden. Also muss die Anzahl Kanten genau um eins kleiner sein als die Anzahl Knoten.

Eigenschaften von Bäumen

Lemma: *Ein vollständiger Binärbaum der Höhe n hat 2^n Blätter*

Beweis: Induktion über Höhe des Baumes

Lemma: *Ein vollständiger Binärbaum der Höhe n hat 2^{n+1} -1 Knoten*

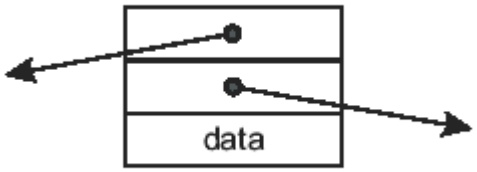
In einem vollständigen Baum ist die Anzahl der Blätter gleich der Anzahl der inneren Knoten minus 1.

Eigenschaften von Bäumen

- **Bemerkung**: Diese (und ähnliche) Zusammenhänge zwischen Höhe eines Baumes und Anzahl der Knoten bzw. Blätter ist ein Grund für die Bedeutung von Bäumen z.B. in Datenbanksystemen
  - Bei „Suchbäumen“ wächst die Höhe nur „logarithmisch“ mit der Anzahl der Blätter
    - Etwa Anfragen an Suchmaschinen, die Milliarden von Seiten im World Wide Web abdecken, liefern durch die Verwendung von Bäumen (zur Strukturierung der Daten) trotzdem sehr schnell eine Antwort
    - Auch wenn das WWW wächst, wird die Zeit für Suchanfragen nur „logarithmisch“ wachsen

Implementierung von Bäumen

- Generische Binärbäume: Knotenklasse



Paket tree kapselt die Klasse Node

```
package tree;

/**
 * Class for Nodes of a generic binary tree.
 */
class Node {
  Node left;
  Node right;
  Object data;
  // constructor
  Node(Object a) {
     data=a;
     left = right = null;}
}
```

Da wir im Paket tree sind, können wir den gleichen Namen wie bei Listenknoten für unsere Klasse nehmen

Implementierung von Bäumen

- Generische Binärbäume durch Referenz auf Wurzelknoten
- Node kann für weitere Datenstrukturen verwendet werden
- Ein leerer Tree t wird nicht durch t=null repräsentiert, sondern durch ein existierendes Objekt t mit t.root=null

```
 * Class for a generic binary tree.
 */
public class Tree {
  protected Node root;

  /**
   * Constructor for empty tree.
   */
  Tree() { root=null;}
  /**
   * Constructs a tree with new root node rn.
   */
   Tree(Node rn) {
    root = rn;
   }
  /**
   * Checks whether this tree is empty.
   */
   public boolean isEmpty() {
     return (root==null);
   }
  // weitere Methoden siehe unten
}
```

Baumdurchläufe

- Schnittstellen-Klasse für generische Baumdurchläufe
- In folgenden Beispielen wählen wir ein Aktionsobjekt der Klasse NodeActionInterface als Parameter, das eine Funktion action(Node) kapselt

```
package tree;
 /**
  * Interface consisting of functions
  * which operate on the nodes of a tree.
  */
  interface NodeActionInterface {
   /**
    * Abstract function whose realizations
    * operate on tree nodes.
    */
    public void action(Node n);
  // evtl. weitere Funktionen
}
```

Wenn p ein formaler Parameter vom Typ
NodeActionInterface
ist und n ein Parameter oder eine Variable vom Typ Node, dann können wir also in den Methoden zum Baumdurchlauf Anweisungen der Form
p.action(n);
verwenden.

Baumdurchläufe

- Beispielklasse, die NodeActionInterface implementiert

```
package tree;

public class NodePrintAction
  implements NodeActionInterface {

  /**
   * Sample implementation of action.
   */
  public void action(Node n) {
    System.out.print(n.data.toString());
  }
}
```

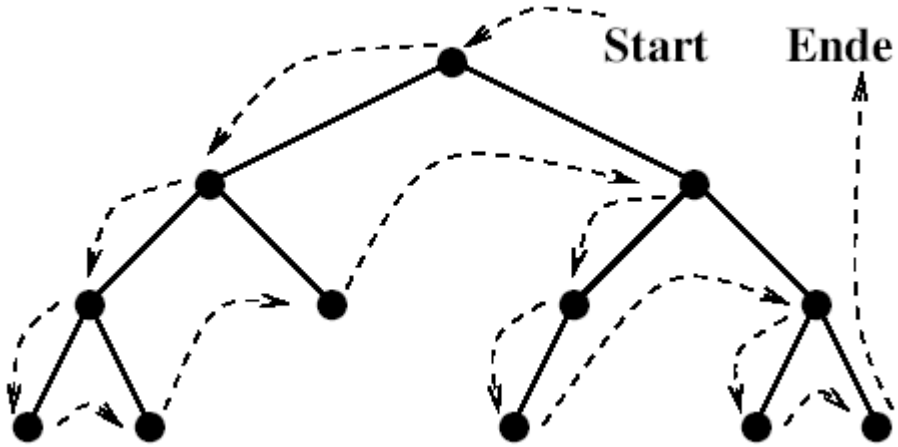
Baumdurchläufe

- Weiteres Beispiel einer Klasse, die NodeActionInterface implementiert
  - Zweck: die inneren Knoten sollen gezählt werden
  - Wieder ein fold-Operator

```
package tree;
public class CountInnerNodes
      implements NodeActionInterface {
  public int counter; // state field
                      // to count nodes
  /**
   * action: count inner nodes
   */
  public void action(Node n) {
    if (n.left!=null || n.right!=null) counter++;
  }
}
```

Baumdurchläufe: Präorder

- Das abstrakte Verfahren zum Durchlauf in Präorder lautet folgendermaßen:
  1. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)
  2. Durchlaufe den linken Teilbaum
  3. Durchlaufe den rechten Teilbaum



Baumdurchläufe: Präorder

```
package tree;

public class Tree {
  protected Node root;
  // ...

  /**
   * Applies the function p.action to any node
   * of the tree in preorder sequence.
   */
  public void preorder(NodeActionInterface p){
    if (root == null)          // empty tree?
      return;

    // init
    Tree leftTree =
      new Tree(root.left);    // left subtree
    Tree rightTree =
      new Tree(root.right);   // right subtree

    // work
    p.action(root);      // visit node: apply function
    leftTree.preorder(p);      // left recursion
    rightTree.preorder(p);     // right recursion
  }

}
```

Ein Baumdurchlauf für den oben gegebenen Strukturbaum (zu 1+2*3) in Präorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

+ 1 * 2 3

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Präorder entspricht der **polnischen Notation** (Polish notation) für Ausdrücke.

Baumdurchläufe: Präorder (nicht-rekursive Version)

- Wollen wir keinen rekursiven Aufruf für die Tiefensuche verwenden, brauchen wir einen Stack, auf dem wir uns die zu behandelnden Knoten für später merken
  - Der rekursive Aufruf verwendet den Laufzeitstapel
- Wir wollen an dieser Stelle die generische Klasse Stack aus dem Paket java.util heranziehen
  - Den generischen Stack von Elementen vom Typ Object benutzen wir also für Elemente vom Typ Node
  - Da die Methode pop ein Objekt vom Typ Object zurückliefert, überprüfen wir mittels des instanceof-Operators, ob es sich tatsächlich um einen Node handelt und spezialisieren es dann auch syntaktisch zu einem Node
  - Da wir vor dem Aufruf der Methode pop testen, ob der Stack leer ist, kann die Ausnahme EmptyStackException nicht vorkommen
    - Die üblichen Java-Compiler können eine solche Analyse jedoch nicht durchführen und verlangen auch in solchen Fällen eine explizite Ausnahmebehandlung - es sei denn, es handelt sich wie hier um eine ungeprüfte Ausnahme (unchecked exception), die ohnehin nicht unbedingt behandelt werden muss
      - Dazu gehört auch die Klasse EmptyStackException, da sie eine Unterklasse von RuntimeException ist

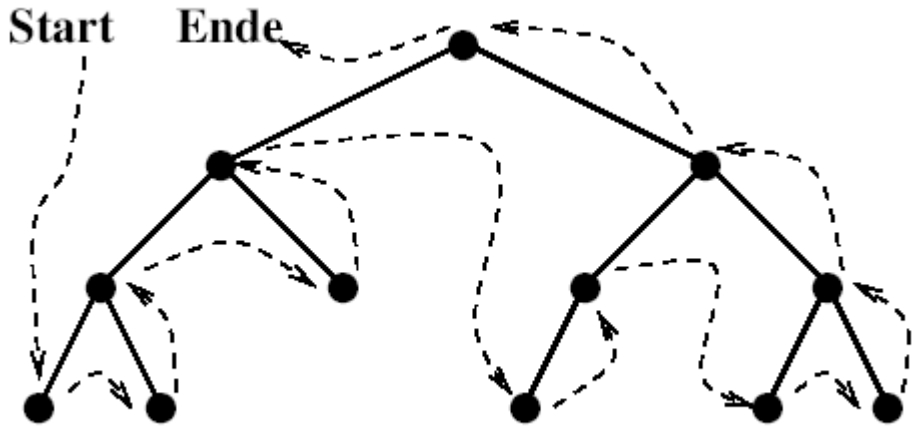
Baumdurchläufe: Präorder (nicht-rekursive Version)

```
package tree;

public class Tree {
  protected Node root;
  // ...
```

```
  /**
   * Applies the function p.action to any node
   * of the tree in preorder sequence.
   * Non-recursive version of the method.
   */
  public void preorderNonRecursive(NodeActionInterface p){
    java.util.Stack stack =
      new java.util.Stack();

    stack.push(root);          // Initialize
    while (!stack.isEmpty()) {
      Object tmp =  stack.pop();
      if (tmp != null && tmp instanceof Node)
        // empty tree?
        // instanceof error must not happen
        {
          Node tmpn = (Node) tmp;
          p.action(tmpn);      // visit node:
                               // apply function
          stack.push(tmpn.right); // right subtree
          stack.push(tmpn.left);  // left subtree
        }
    }
  }
}
```

Baumdurchläufe: Postorder

- Das abstrakte Verfahren zum Durchlauf in Postorder lautet folgendermaßen:
  1. Durchlaufe den linken Teilbaum
  2. Durchlaufe den rechten Teilbaum
  3. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)



Baumdurchläufe: Postorder

```
package tree;
public class Tree {
  protected Node root;

  // ...

  /**
   * Applies the function p.action
   * to any node of the tree in postorder
   * sequence.
   */
  public void postorder(NodeActionInterface p){
    if (root == null)          // empty tree?
      return;

    // init
    Tree leftTree =
      new Tree(root.left);    // left subtree
    Tree rightTree =
      new Tree(root.right);   // right subtree

    // work
    leftTree.postorder(p);    // left recursion
    rightTree.postorder(p);   // right recursion
    p.action(root);   // visit node: apply function
  }
}
```

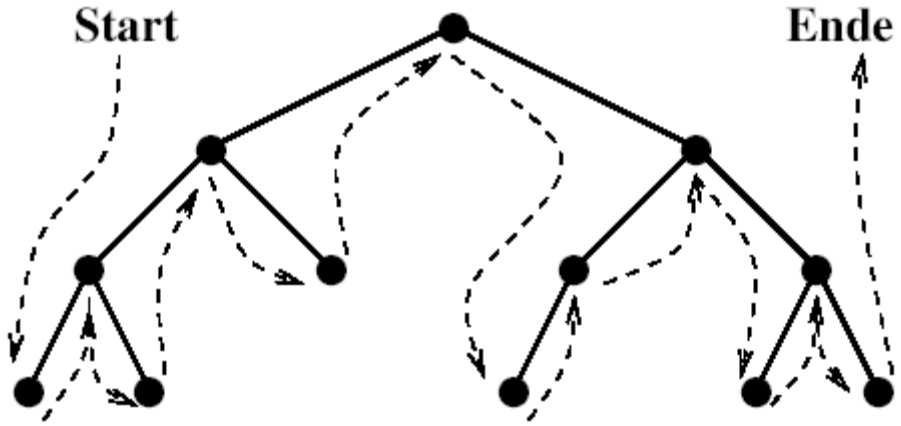
Ein Baumdurchlauf für den oben gegebenen Strukturbaum in Postorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

1 2 3 * +

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Postorder entspricht der **umgekehrten polnischen Notation** (reverse Polish notation) für Ausdrücke.

Baumdurchläufe: Inorder

- Das abstrakte Verfahren zum Durchlauf in Inorder lautet folgendermaßen:
  1. Durchlaufe den linken Teilbaum
  2. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)
  3. Durchlaufe den rechten Teilbaum



Baumdurchläufe: Inorder

```
package tree;
public class Tree {
  protected Node root;
  // ...

 /**
   * Applies the function p.action
   * to any node of the tree in inorder
   * sequence.
   */
  public void inorder(NodeActionInterface p){
    if (root == null)        // empty tree?
      return;

    // init
    Tree leftTree =
      new Tree(root.left);  // left subtree
    Tree rightTree =
      new Tree(root.right); // right subtree

    // work
    leftTree.inorder(p);  // left recursion
    p.action(root);    // visit node:
                       // apply function
    rightTree.inorder(p) // right recursion

  }
}
```

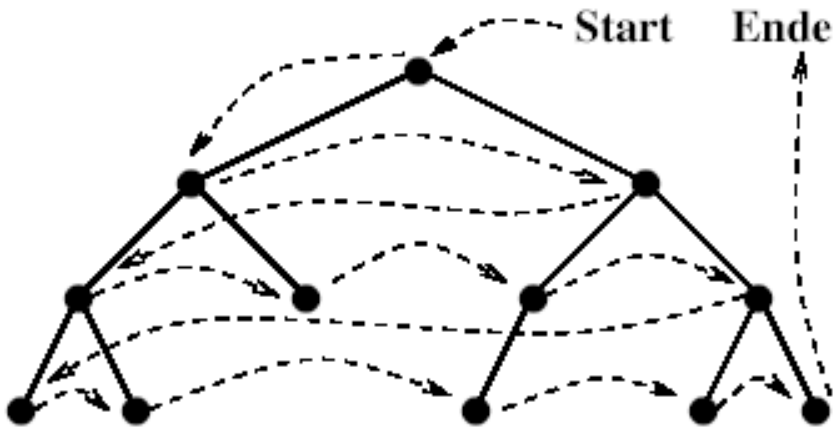
Ein Baumdurchlauf für den oben gegebenen Strukturbaum in Inorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

1 + 2 * 3

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Inorder entspricht der normalen Operator-Schreibweise (wenn zusätzlich die Subausdrücke noch geklammert werden)

Baumdurchläufe: Levelorder

- Beim Durchlaufen eines Baumes Schicht für Schicht (levelorder) geht man wie folgt vor
  - Starte bei der Wurzel (Ebene 0)
  - Bis die Höhe des Baumes erreicht ist, setze die Ebene um eins höher und gehe von links nach rechts durch alle Knoten dieser Ebene



Baumdurchläufe: Levelorder

- Bei diesem Verfahren geht man nicht zuerst in die Tiefe, sondern die Strategie heißt **Breite zuerst** (breadth first)
  - Dies kommt besonders bei **Suchbäumen** (search tree) zum Einsatz, deren Knoten Spielpositionen und deren Kanten Spielzüge darstellen (z. B. für Schach, Dame etc.)
    - Wir suchen dann im Baum eine Gewinnstellung, die in möglichst wenigen Zügen erreichbar ist
    - Solche Suchbäume sind sehr tief und werden daher nur begleitend zur Suche Schicht für Schicht generiert, bis der gesuchte Knoten gefunden wurde
    - Der Einfachheit halber werden wir aber in der Folge von einem bereits generierten Baum ausgehen

Baumdurchläufe: Levelorder

- Um der Breite nach durch einen Baum zu wandern, müssen wir uns alle Knoten einer Ebene merken
- Diese Knoten speichern wir für späteren Zugriff in einer Warteschlange (queue) ab
- Die Warteschlange kann sehr lang werden
  - Im schlimmsten Fall erhält sie eine Länge von $n/2$ bei n Knoten (die Anzahl der Blätter eines vollständigen Binärbaums)
  - Bei den rekursiven Methoden („depth first“) preorder, inorder oder postorder wird der (implizit oder explizit) verwendete Stack maximal so groß wie die Tiefe des bei der Rekursion betrachteten Baumes
  - Dieser ist - wie wir unten genauer zeigen werden - um eins tiefer als der zu durchlaufende Baum selbst. Damit ist die maximale Tiefe $1+\log_2 n$.

Baumdurchläufe: Levelorder

```
package tree;
public class Tree {
  protected Node root;
  // ...

  /**
   * Applies the function p.action
   * to any node of the tree in levelorder
   * sequence.
   */
  public void levelorder(NodeActionInterface p){
    Queue queue = new Queue();
    queue.append(root);
    while( !queue.isEmpty()) {
      Object tmp = queue.get();
      if( tmp != null && tmp instanceof Node)
        // empty tree?
        // instanceof error must not happen
        {
          Node tmpn = (Node) tmp;

          p.action(tmpn.data);        // apply function
          queue.append(tmpn.left);    // left subtree
          queue.append(tmpn.right);   // right subtree
        }
    }
  }

}
```

Speicherbedarf in rekursiven Baumdurchläufen

- Betrachte Rekursion in Klassenmethoden von Tree
- Aufwand dazu:
  - Vor Rekursion: Erzeugung von zwei neuen Bäumen erforderlich:

    ```
    Tree leftTree = new Tree(root.left);
    Tree rightTree = new Tree(root.right);
    ```
- Für jeden (nicht leeren) Knoten werden zwei Bäume generiert und wieder zerstört
- Auf Laufzeitstapel Speicher für *2n* Referenzvariablen und *2n* Knotenvariablen (muss nicht gleichzeitig existieren) wobei *n* = Anzahl der Knoten

Speicheroptimierung durch Mantelprozedur (jacket)

```
package tree;
public class Tree {
    protected Node root;
    // ...
    public void preorder(NodeActionInterface f){
        traversePreorder(root, f);
    }
    private void traversePreorder(Node n, NodeActionInterface  f)  {
      // trivial case
      if (n == null) return;
      // Action
      f.action(n);
      // Recursion
      traversePreorder(n.left, f);
      traversePreorder(n.right, f);   }}
```

Zeitbedarf in rekursivem Baumdurchläufe

- Wir haben je Knoten und für jeden nicht existenten Nachfolger eines Knoten einen Prozeduraufruf
- In vollständigem Binärbaum der Tiefe k mit $n=2^k-1$ Knoten ist die Anzahl dieser Aufrufe mit leeren Teilbaum $2 \cdot 2^{k-1} = n+1$, also sogar größer als die Anzahl der nichttrivialen Aufrufe!
- Lösung: Test vor jedem rekursiven Aufruf

  ```
  if (n.left!=null) traversePreorder(n.left, f);
  ```
- Dann ist aber Test

  ```
  if (n == null) return;
  ```

  unnötig (außer zu Beginn)
- Lösung: zweite Mantelprozedur

  ```
  traversePreorderNonEmpty(root, f);
  ```

  mit Aufruf nur, wenn Baum nicht leer.

Effizienter Präorder Baumdurchlauf

```
package tree;
public class Tree {
  protected Node root;
  // ...
  public void preorder(NodeActionInterface f){
    if (root == null) return; // empty tree
    else traversePreorderNonEmpty(root, f);
  }
  private void traversePreorderNonEmpty (Node n, NodeActionInterface f) {
    // Non-trivial case!
    // Action
    f.action(n);
    // Recursion
    if (n.left != null) traversePreorderNonEmpty(n.left, f);
    if (n.right != null) traversePreorderNonEmpty(n.right, f);
  }
}
```