universitätbonn

Objektorientierte Realisierung komplexer Datenstrukturen

Referenzvariablen als Zeigervariablen
Listen
Stapel und Warteschlangen
Generische Datenstrukturen → Generizität in Java und C++
Bäume
Graphen

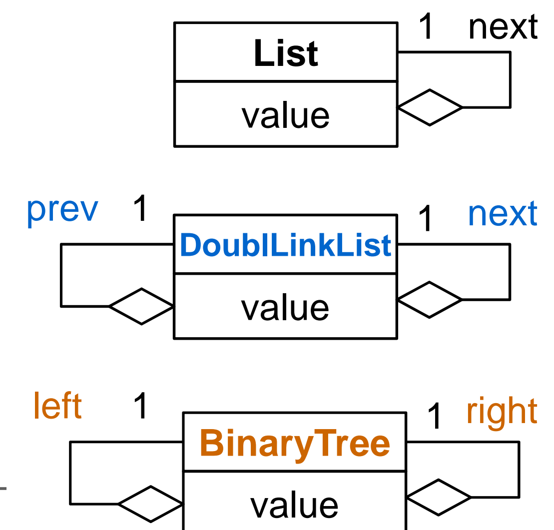
Stand: 20.07.2010

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

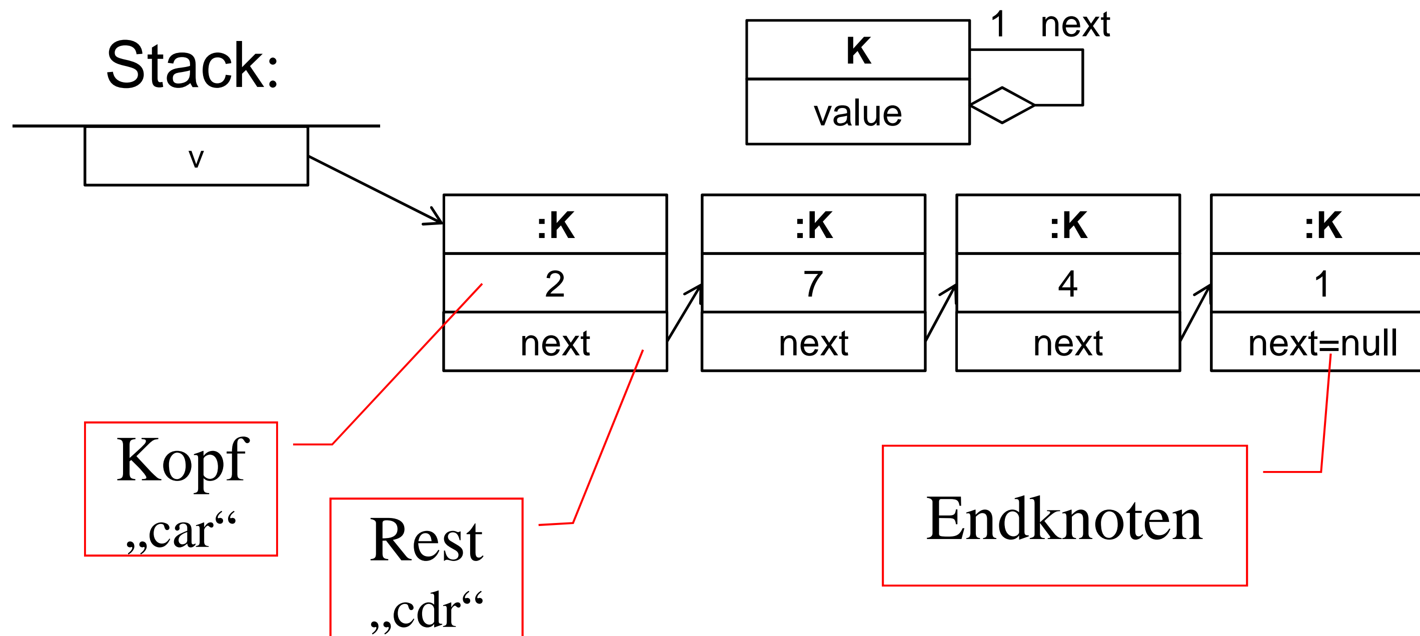


universitätbonn

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 **einfach verketteten Liste** verknüpfen

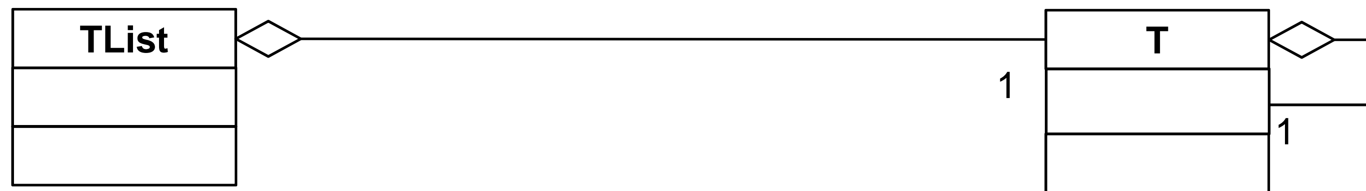


Listen (linked lists)

- Verzeigerte Listen sind 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

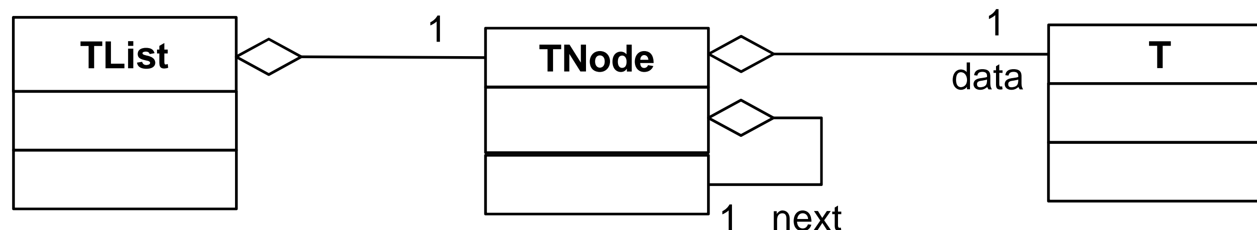
- Interne Verkettung
  - Die Datenobjekte stellen selbst die Verkettung her
  - Objekte vom Typ T sind dazu gemacht Listenelemente zu sein



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

Listenknoten

- Externe Verkettung
  - Die Datenobjekte vom Typ T wissen nicht dass sie in einer Liste sind
  - Listenzelle (TNode) enthält zwei Felder
    - Ein **Datenfeld** mit einer Referenz auf ein Objekt vom Typ T
    - Eine **Verkettungsfeld** mit einer Referenz auf den nächsten Knoten



- Containerklasse TList
  - Wie vorhin

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
}
```

Typische Operationen auf Listen ▸ Durchlauf durch Listen

- Beispiel
  - `toString()`-Methode

Typischer Listendurchlauf

Direkter Zugriff auf TNode-Felder

```
public class TList { // ...wie vorhin...
    /**
     * Liste wird von runden Klammern
     * begrenzt und die Listenelemente
     * durch Leerzeichen getrennt
     * ausgegeben.
     */
    public String toString() {
        // Initialize
        StringBuffer sb = new StringBuffer();
        sb.append("(");
        TNode 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

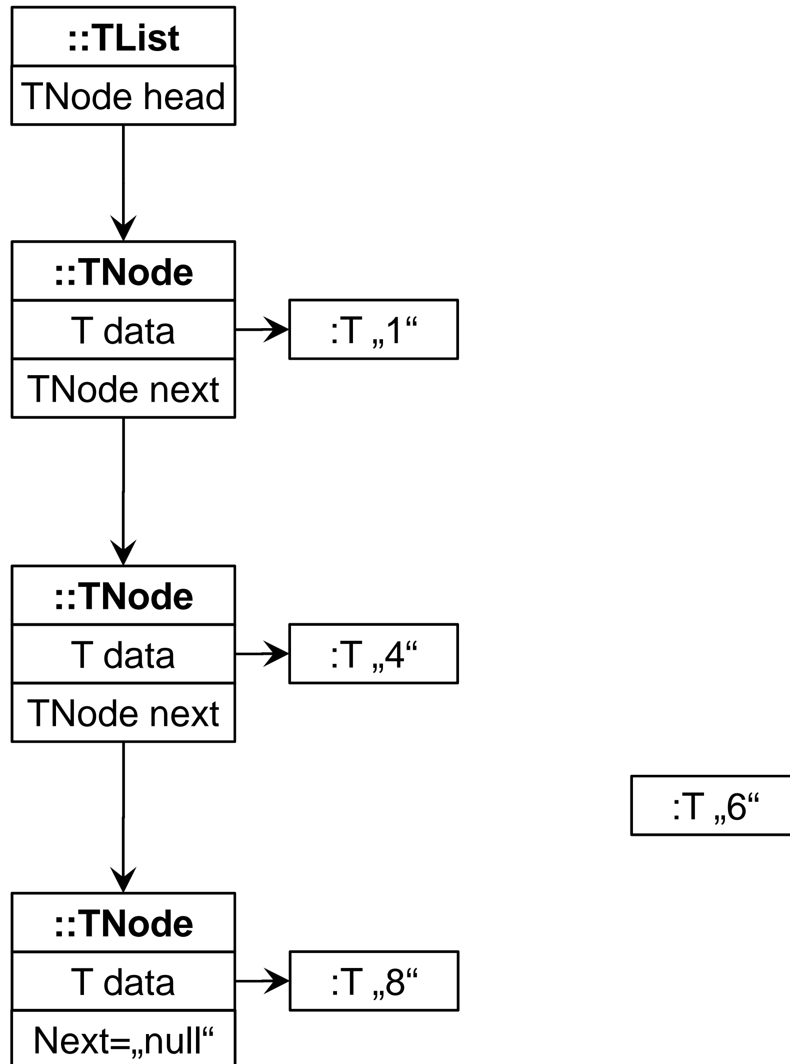
```
public class TList { // [..]
    /**
     * Fügt elem am Anfang der Liste ein.
     * Ist auch richtig für den
     * Spezialfall der leeren Liste.
     */
    public void insertFirst(T elem) {
        // Neue Listenzelle erzeugen,
        // deren next Feld auf head zeigt.
        TNode tmp = new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

Typische Operationen auf Listen ▸ Einfügen eines Elements 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:
Behandlung von Sonderfällen

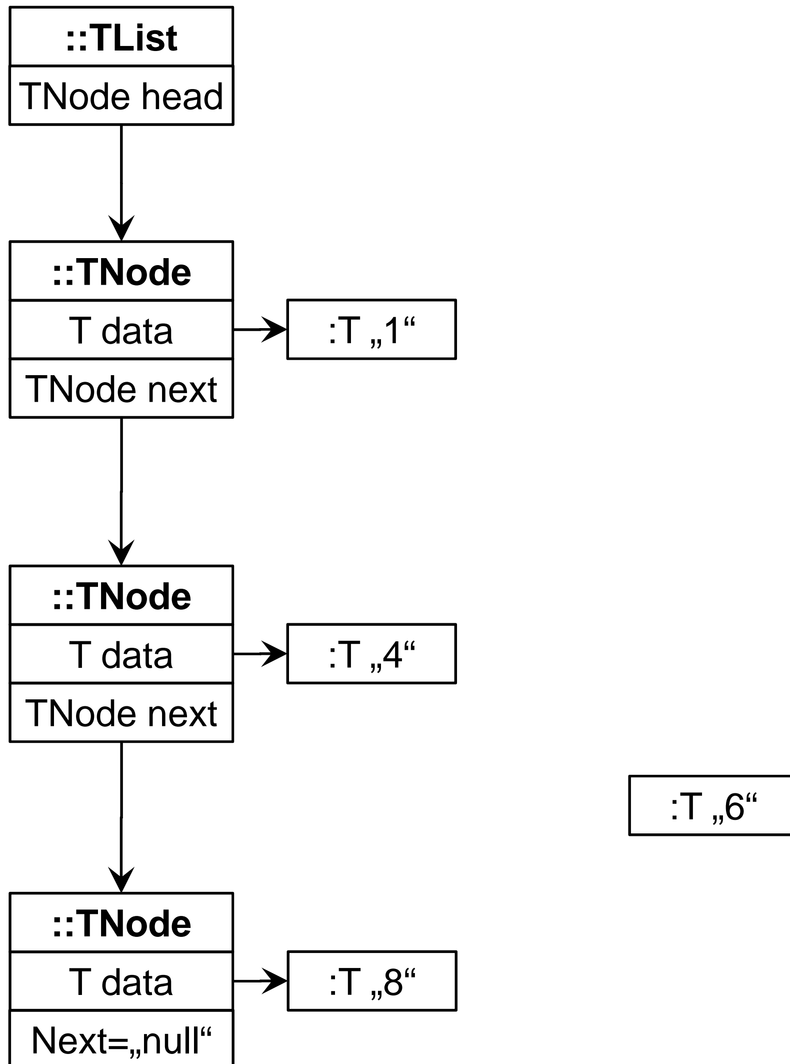
Typische Operationen auf Listen ▸ Sortiertes Einfügen



- Liste sei geordnet
  - Die Ordnung auf den Daten sei etwa durch Methode **`compareTo`** gegeben

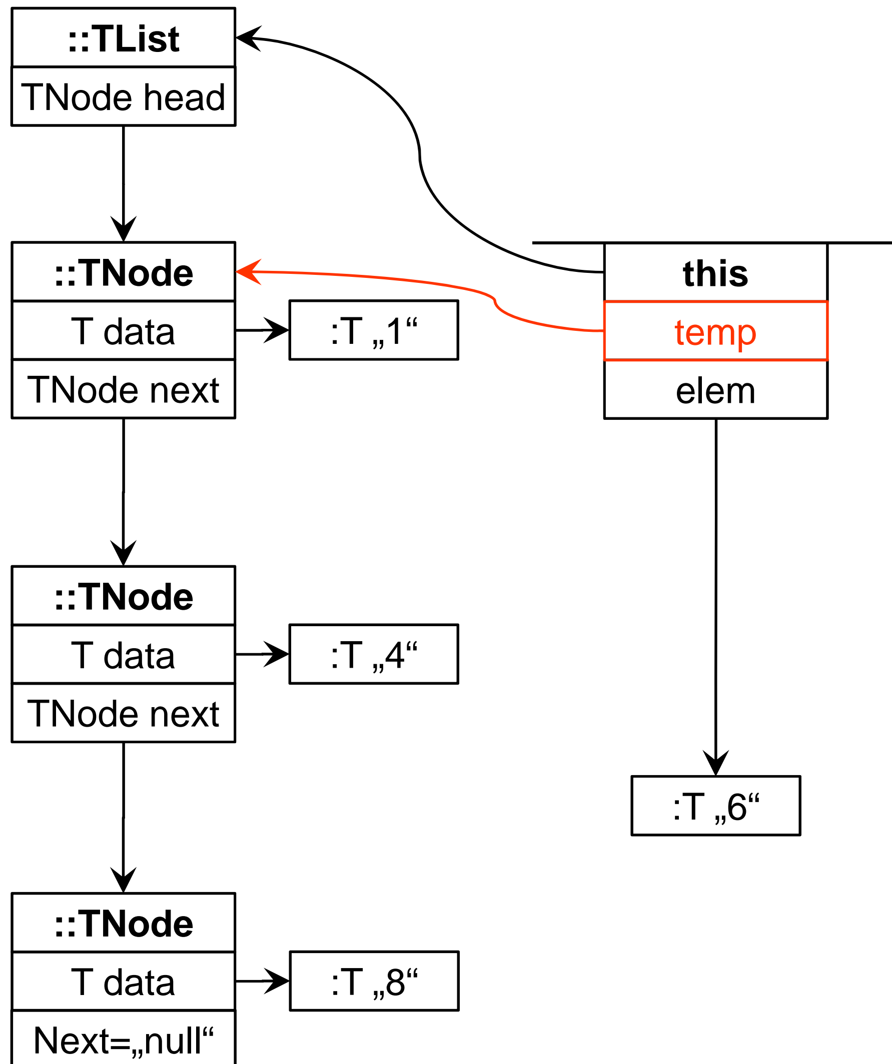
- Wie wird ein weiteres Datenobjekt unter Beibehaltung der Ordnung einsortiert?

Typische Operationen auf Listen ▸ Sortiertes Einfügen



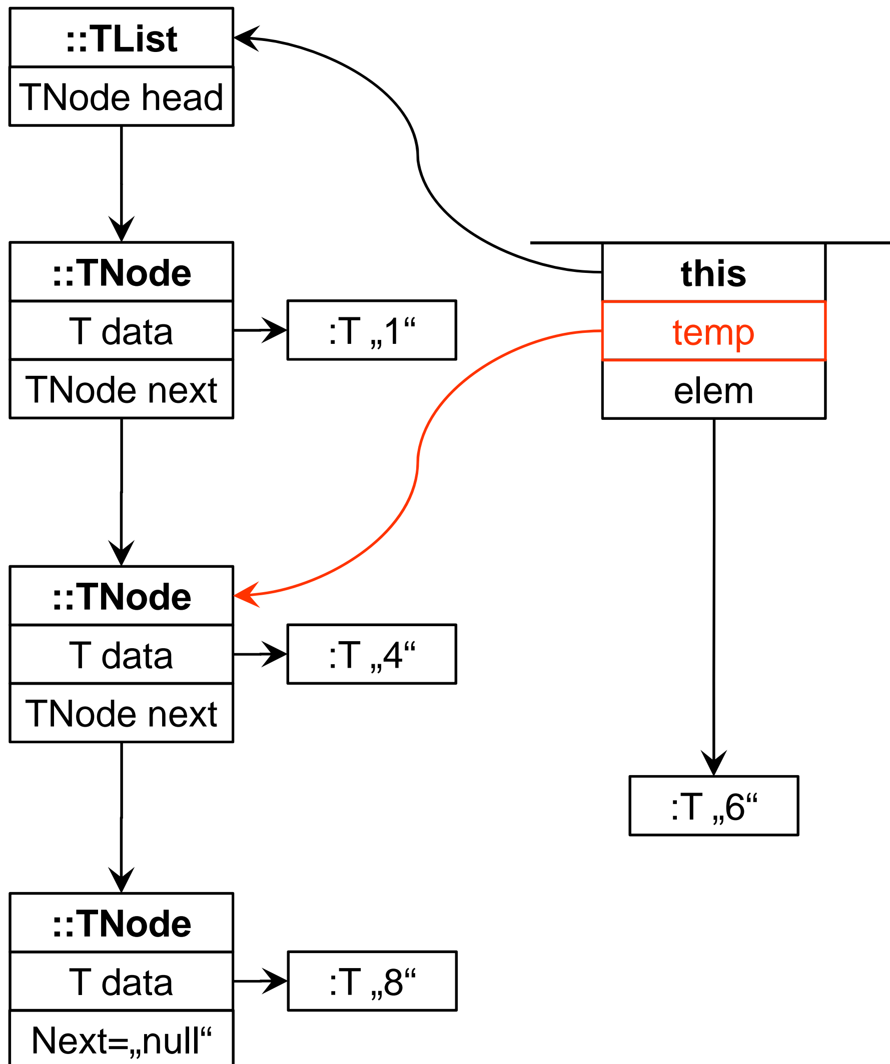

```
public class TList { // [..]
     /**
      * Fügt elem sortiert in die
      * aufsteigend sortierte Liste ein.
      */
     public void insertSorted(T elem) {
          TNode tmp = head;
          // Spezialfall: Liste ist leer oder
          // neues Element ist kleinstes
          if( (head == 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



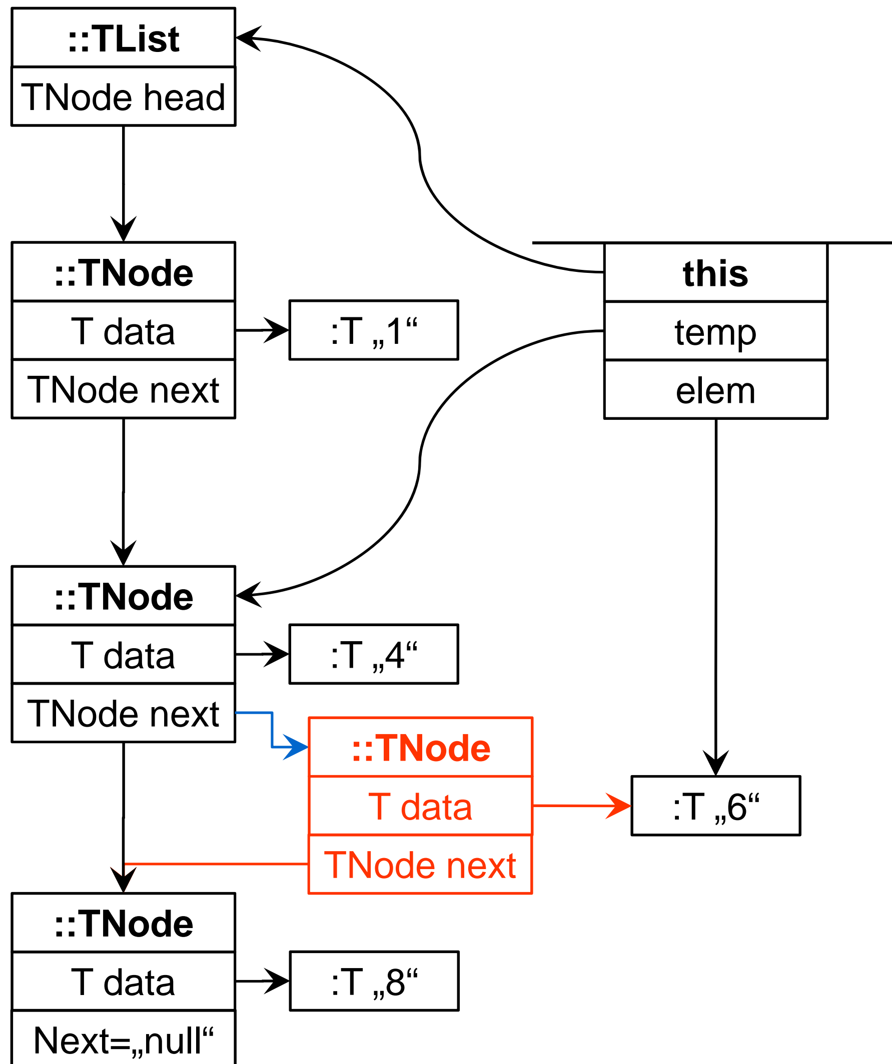

```
public class TList { // [..]
    /**
     * Fügt elem sortiert in die
     * aufsteigend sortierte Liste ein.
     */
    public void insertSorted(T elem) {
        TNode tmp = head;
        // Spezialfall: Liste ist leer oder
        // neues Element ist kleinstes
        if( (head == null) ||
            (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




```
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




```
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 (Rekursiv)

- 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
  - Sonst füge das neue Element sortiert in die um den Kopf verkürzte Liste **`tmpList`** ein (rekursiver Aufruf)
  - … und hänge die neue Liste zuletzt wieder an den Kopf der ursprünglichen Liste an

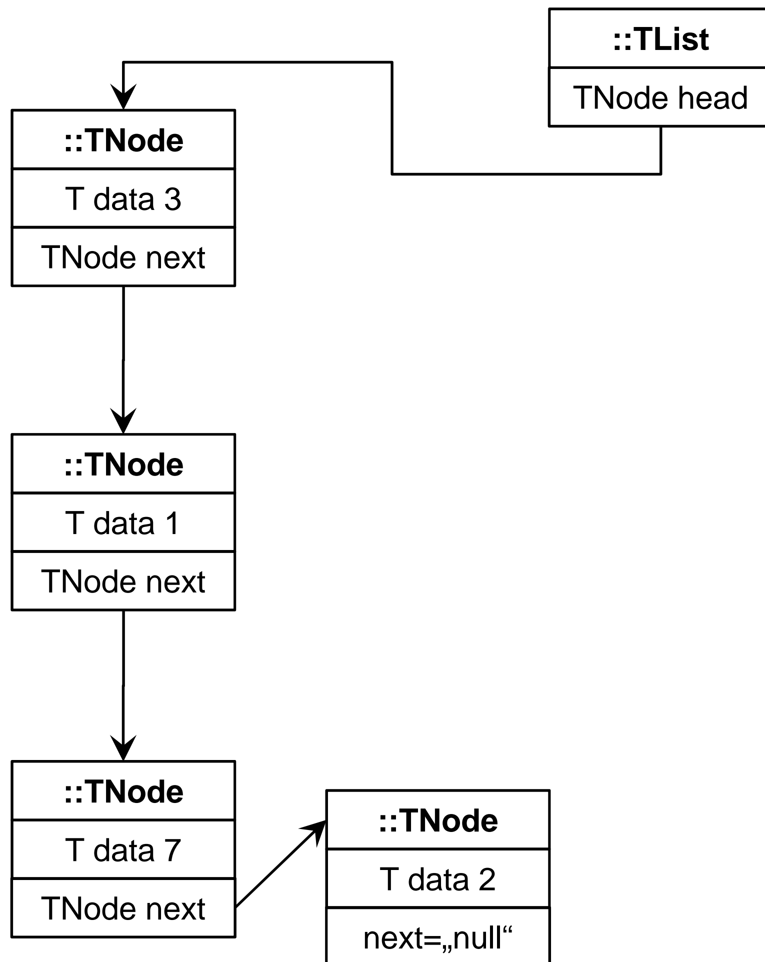
Typische Operationen auf Listen ▸ Sortiertes Einfügen (Rekursiv)

- 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

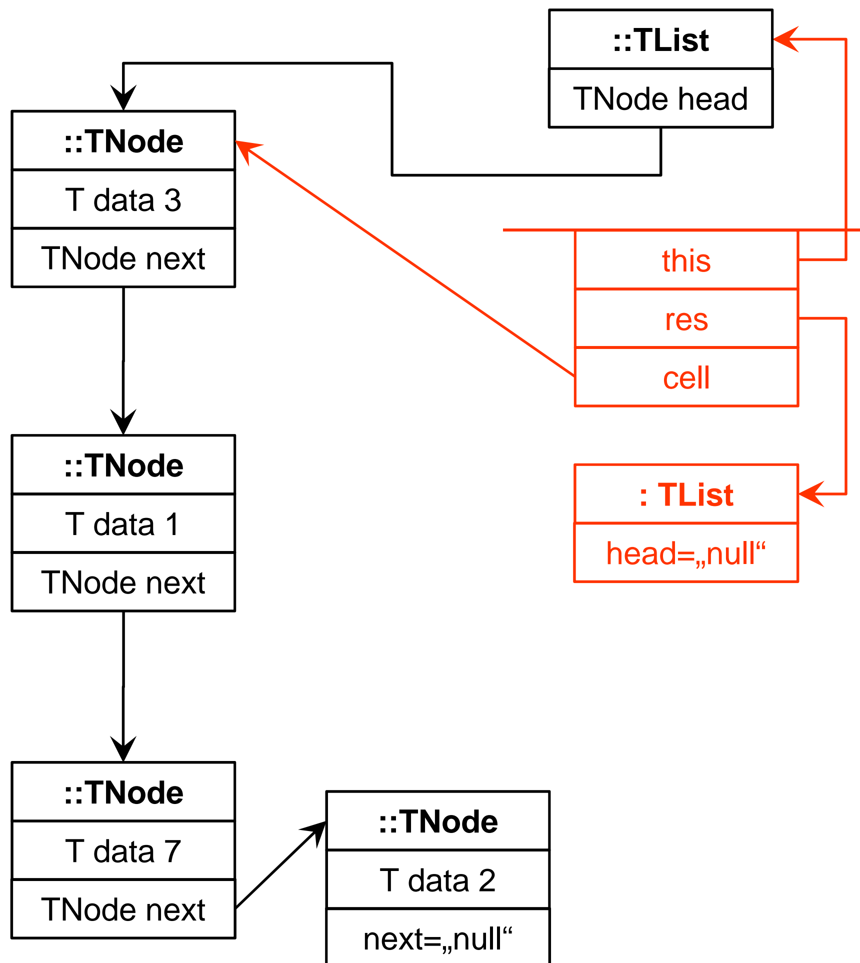
▸ Konstruktives Invertieren




```
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

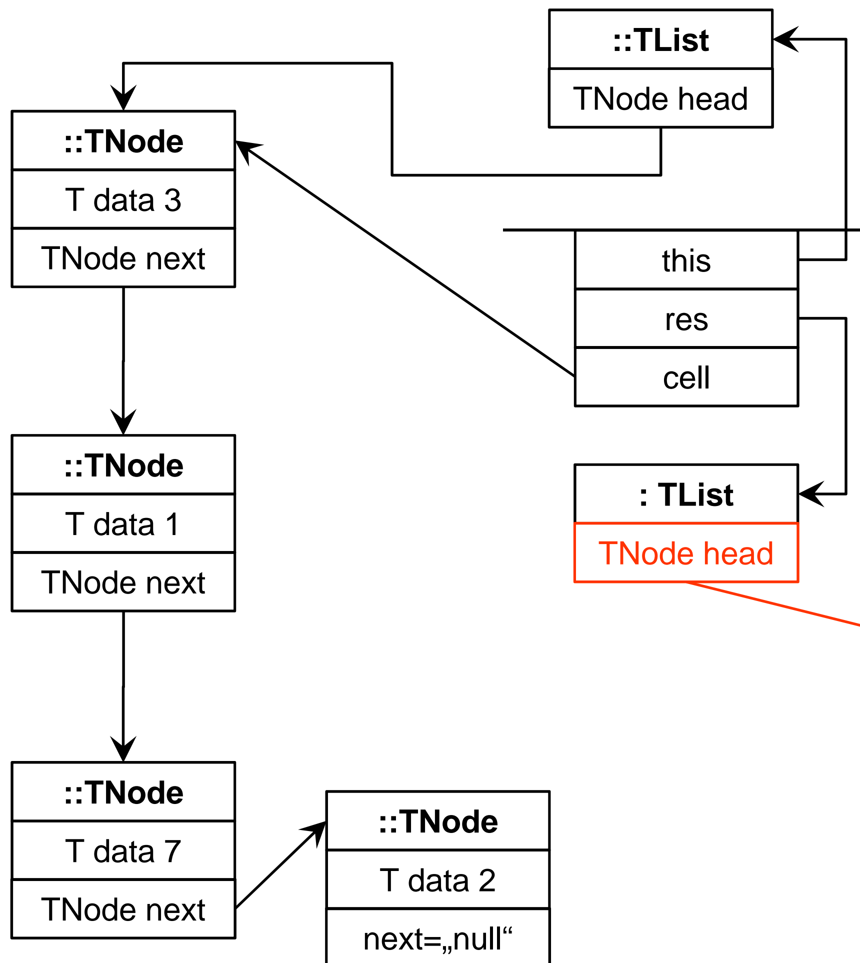
Konstruktives Invertieren




```
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

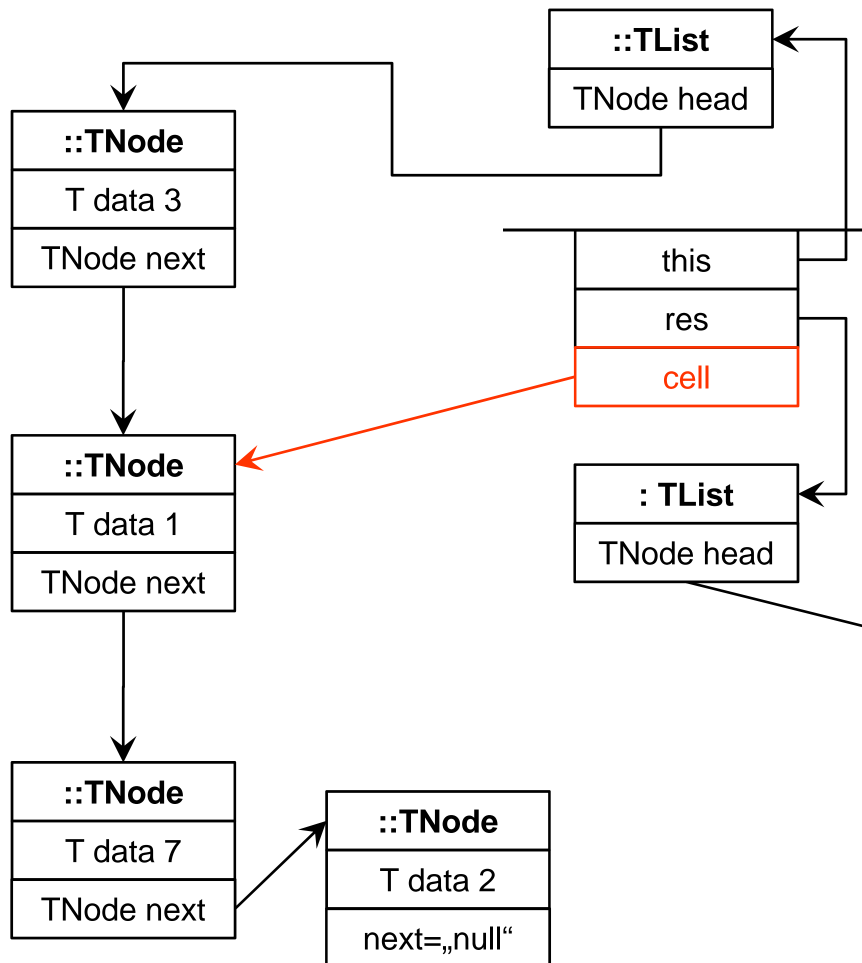
Konstruktives Invertieren




```
public class TList { // [..]
     /**
      * 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

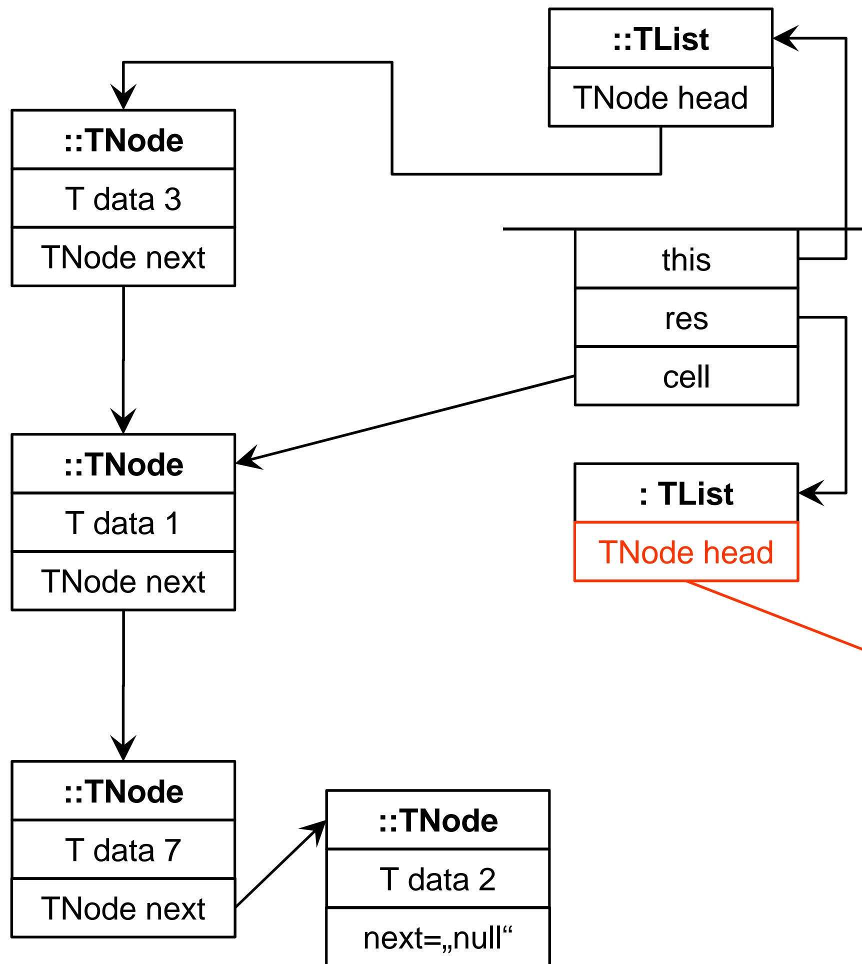
Konstruktives Invertieren




```
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

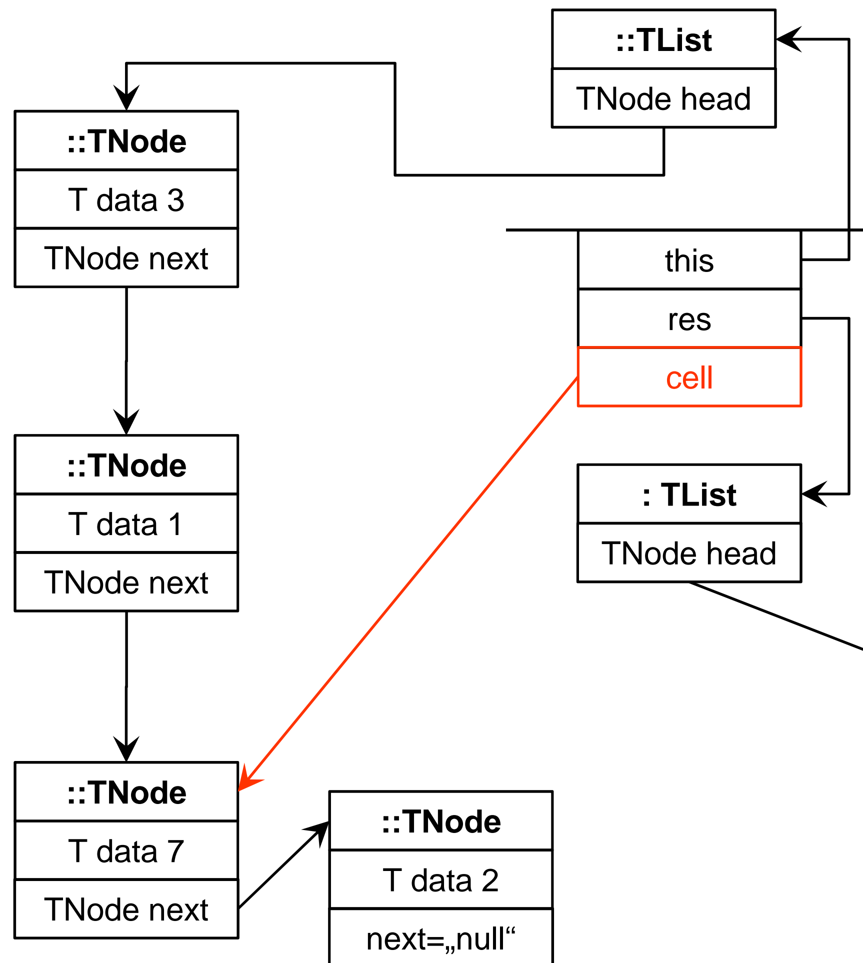



```
public class TList { // [..]
    /**
     * Gibt die konstruktiv invertierte
     * Liste zurück.
     */
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;

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

::TNode
T data 1
TNode next

::TNode
T data 3
next=„null“

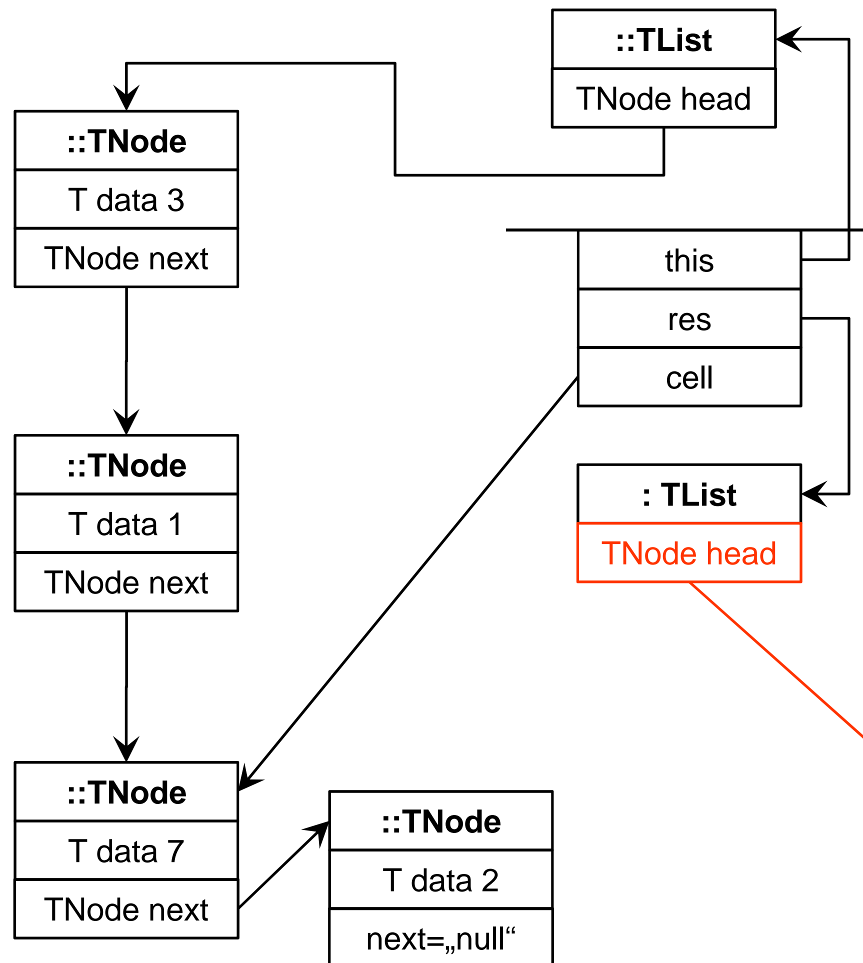
Typische Operationen auf Listen
▸ Konstruktives Invertieren




```
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

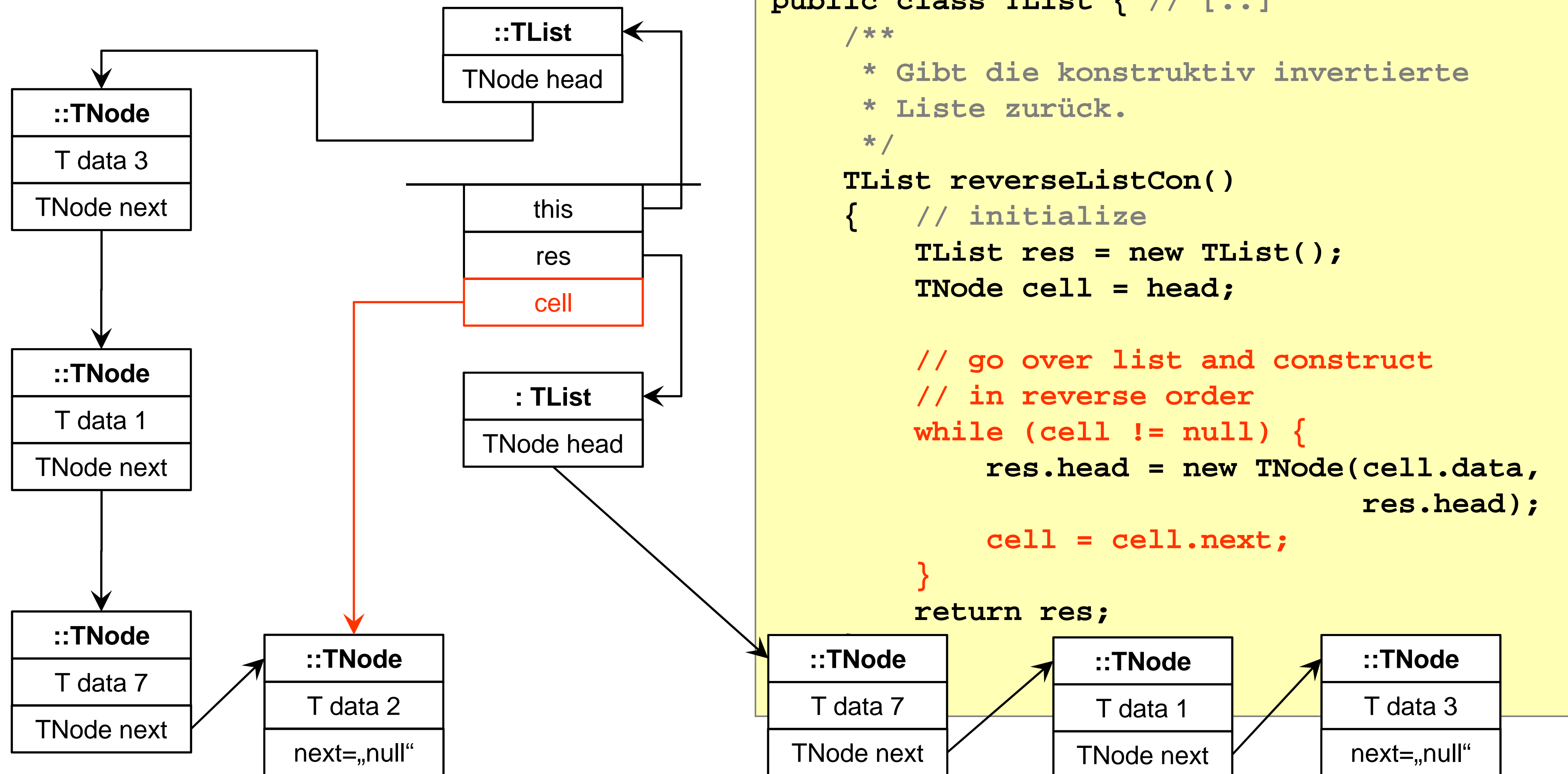
Konstruktives Invertieren




```
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;
```

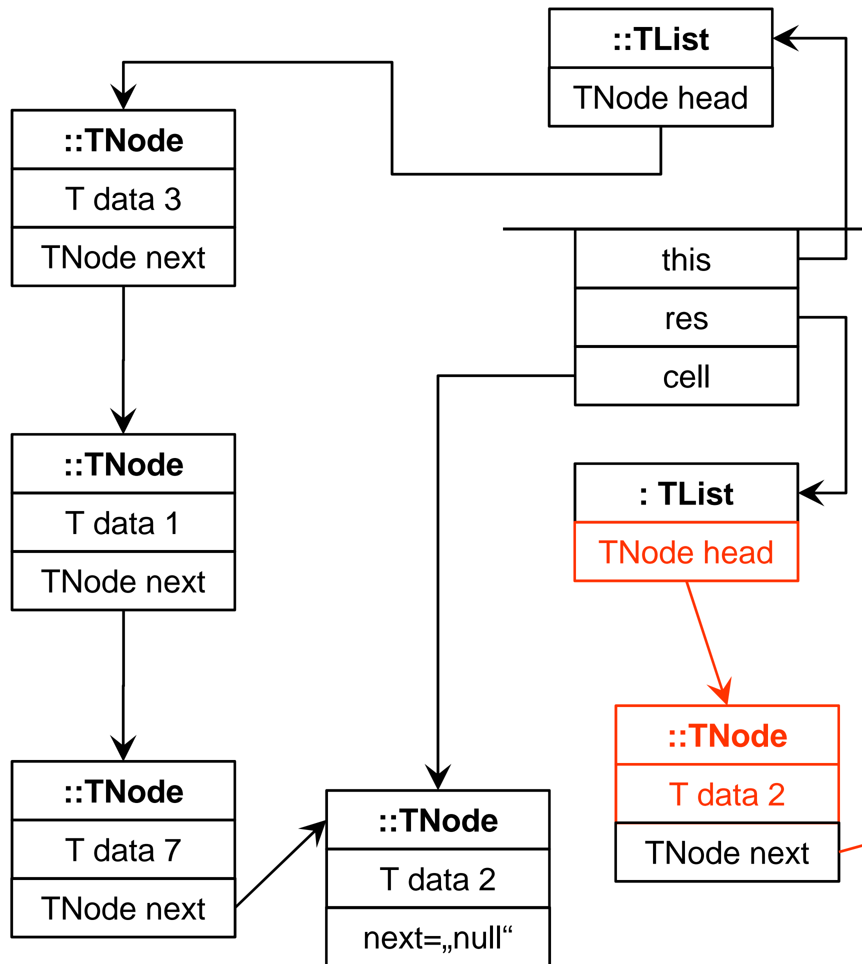
::TNode	::TNode	::TNode
T data 7	T data 1	T data 3
TNode next	TNode next	next=„null“

Typische Operationen auf Listen
Konstruktives Invertieren

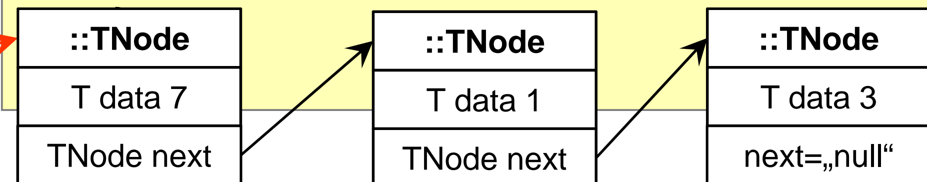



```
public class TList { // [..]
    /**
     * Gibt die konstruktiv invertierte
     * Liste zurück.
     */
    TList reverseListCon()
    {   // initialize
        TList res = new TList();
        TNode cell = head;

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

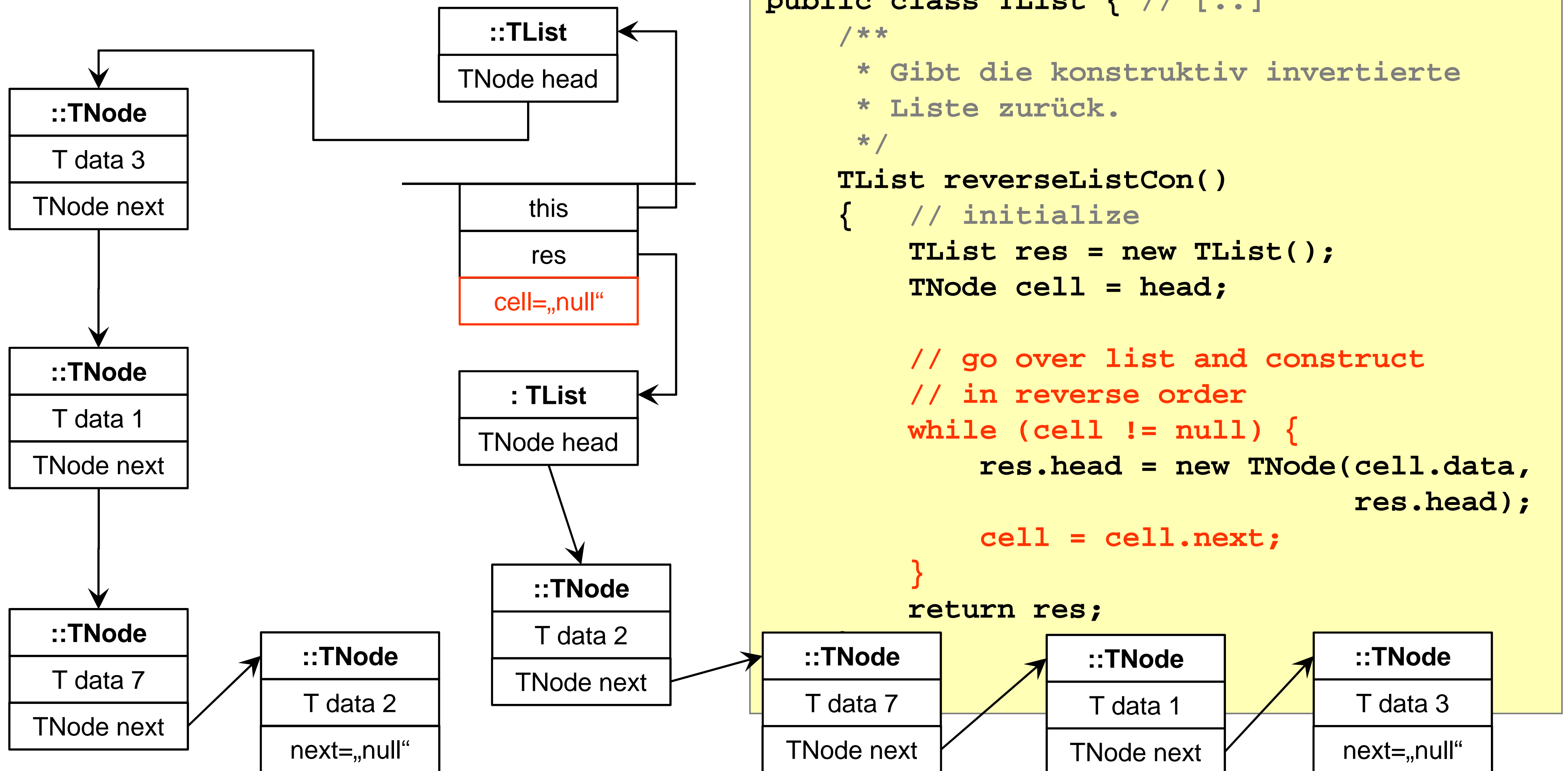
Typische Operationen auf Listen
Konstruktives Invertieren




```
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;
```



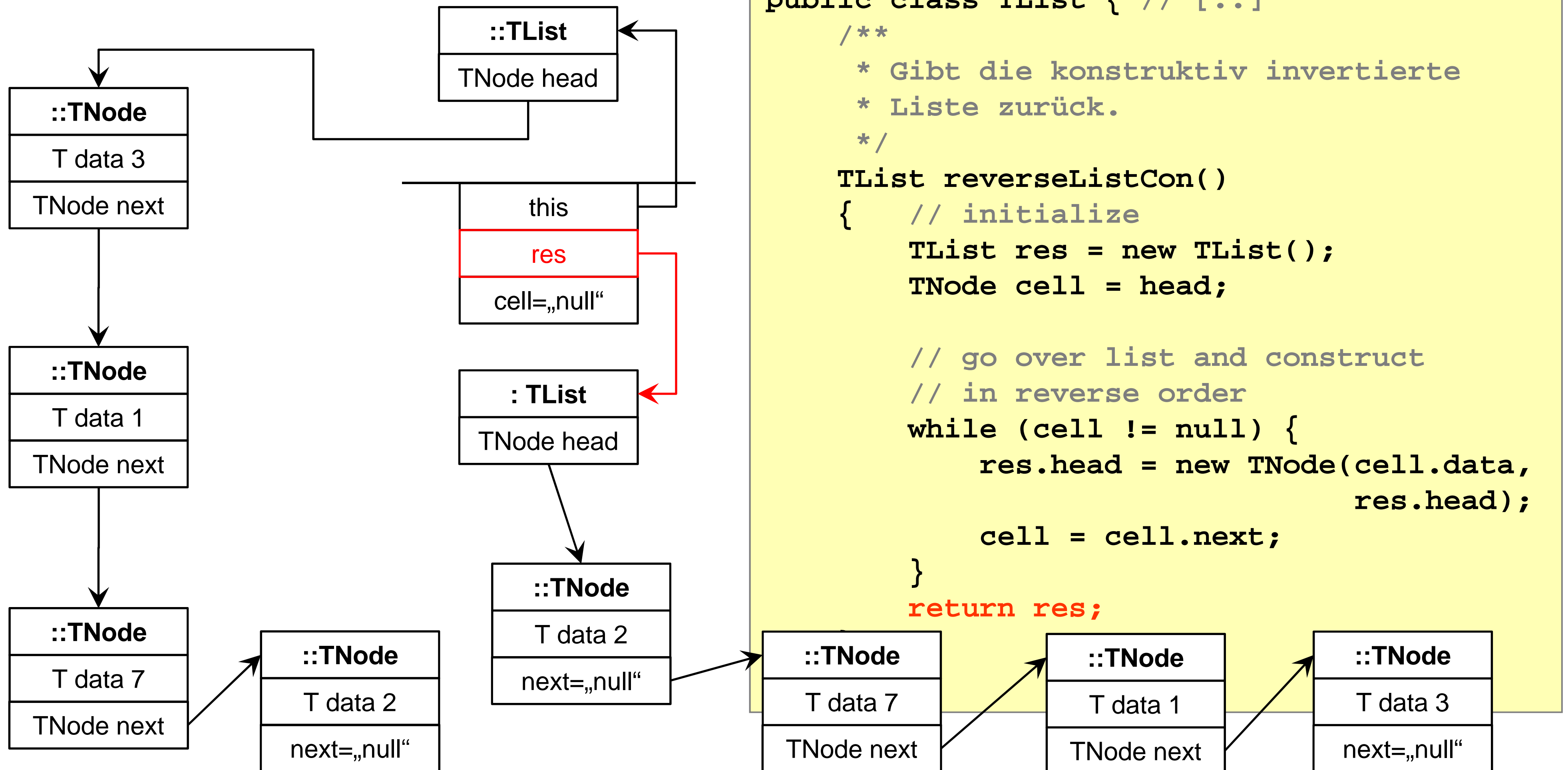
Typische Operationen auf Listen
Konstruktives Invertieren




```
public class TList { // [..]
    /**
     * 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

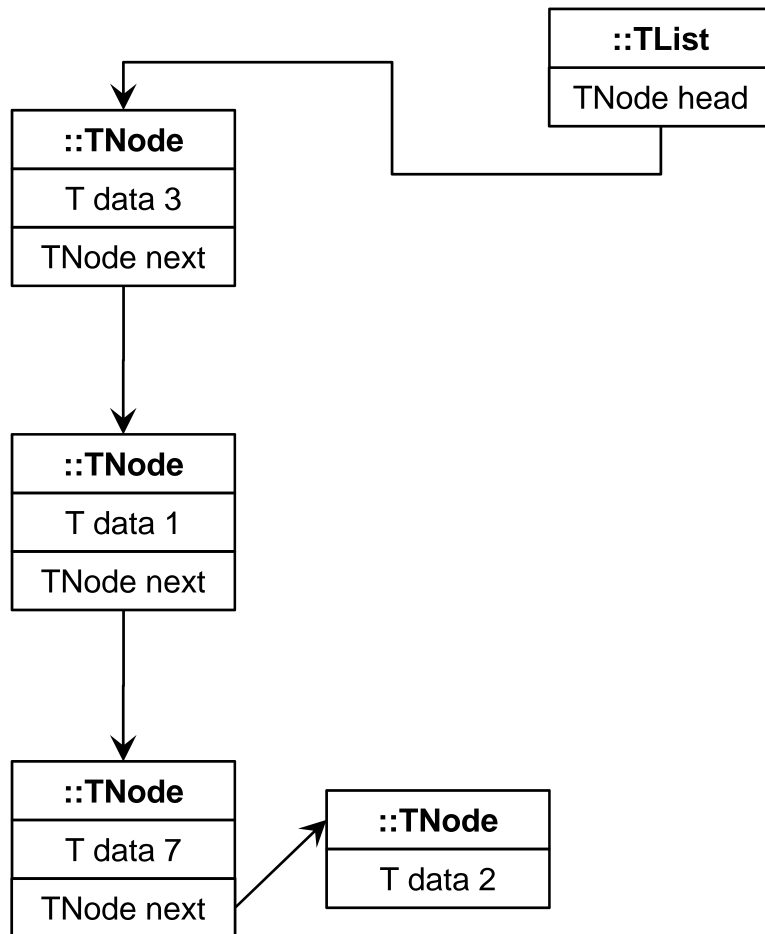
Konstruktives Invertieren




```
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

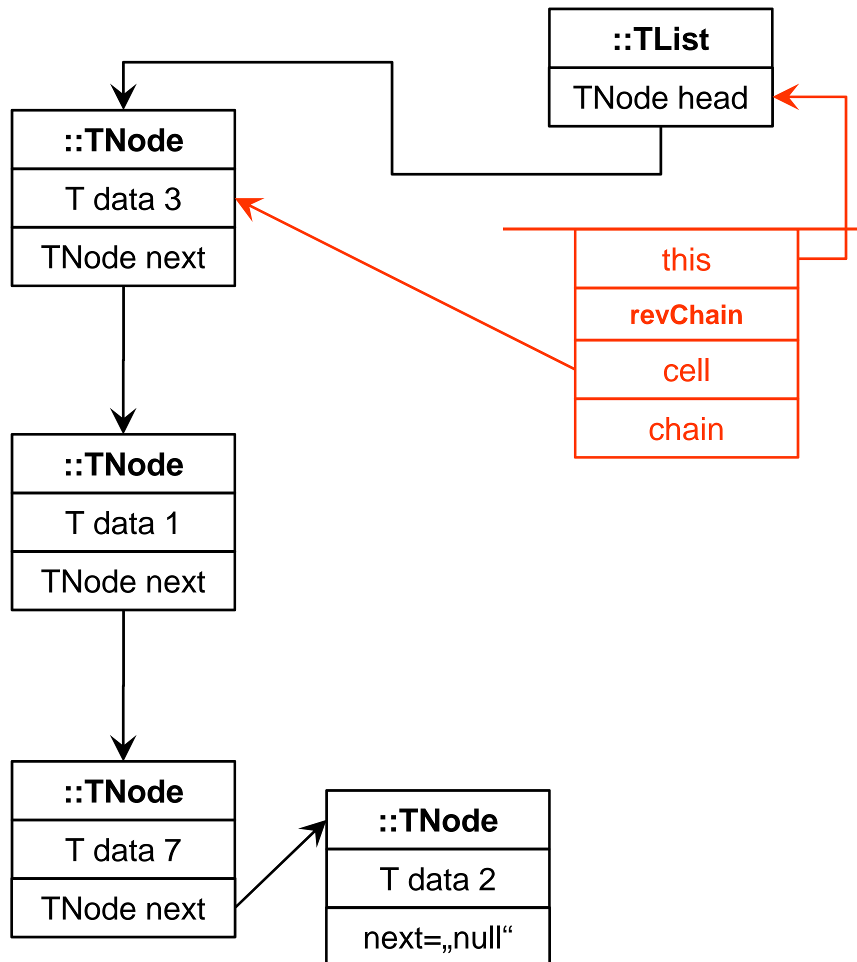
Destruktives Invertieren




```
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

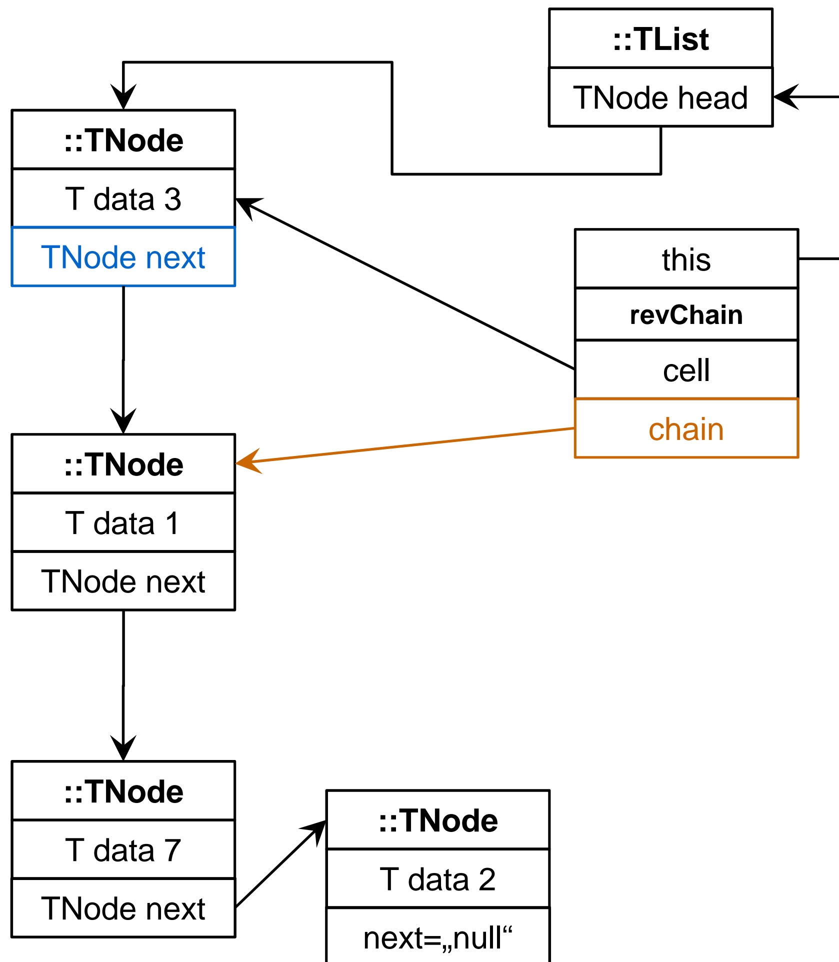
‣ Destruktives Invertieren




```
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

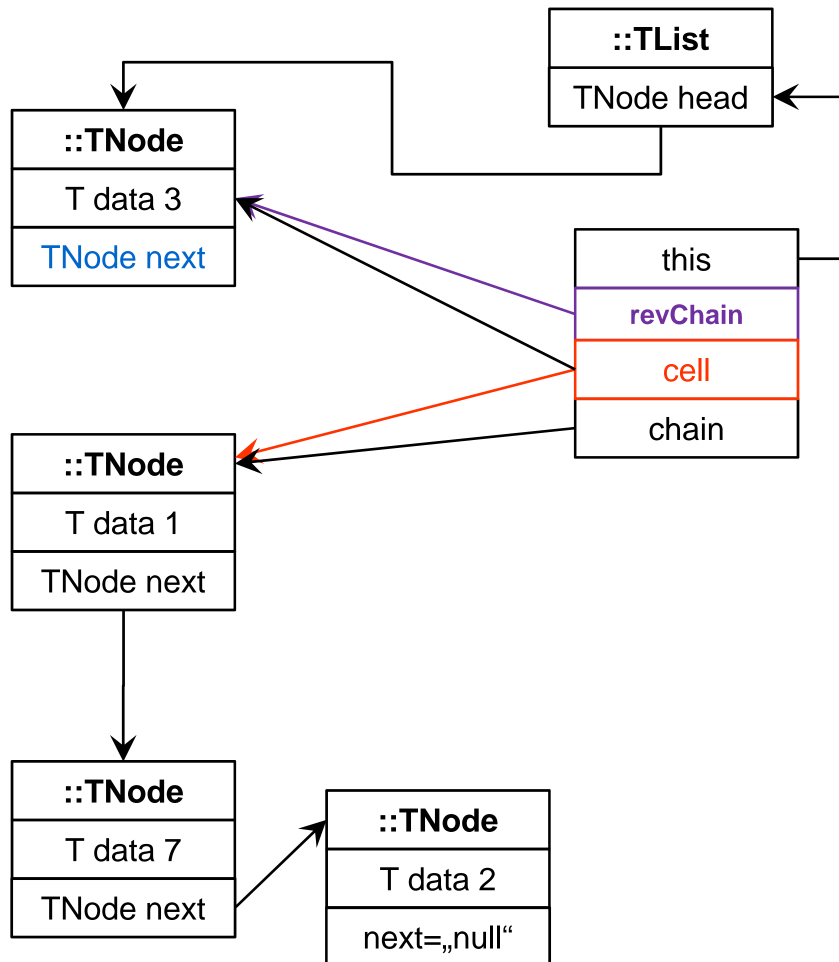
Destruktives Invertieren




```
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

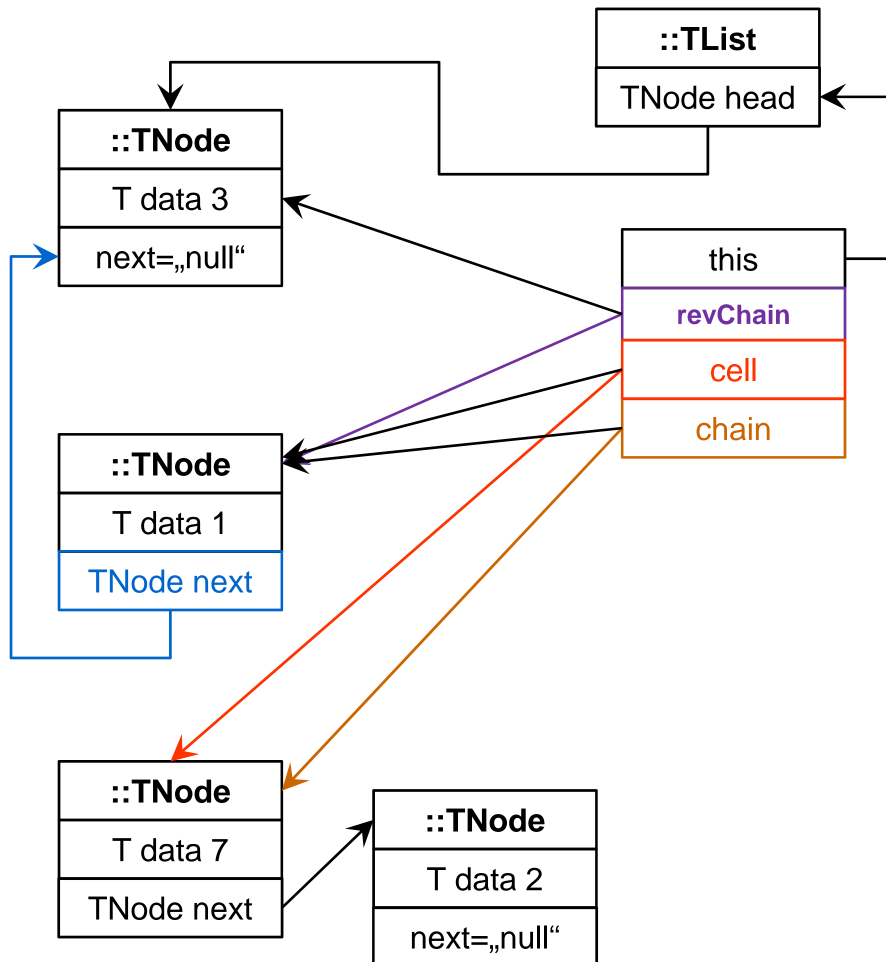
Destruktives Invertieren




```
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;
    }
}
```

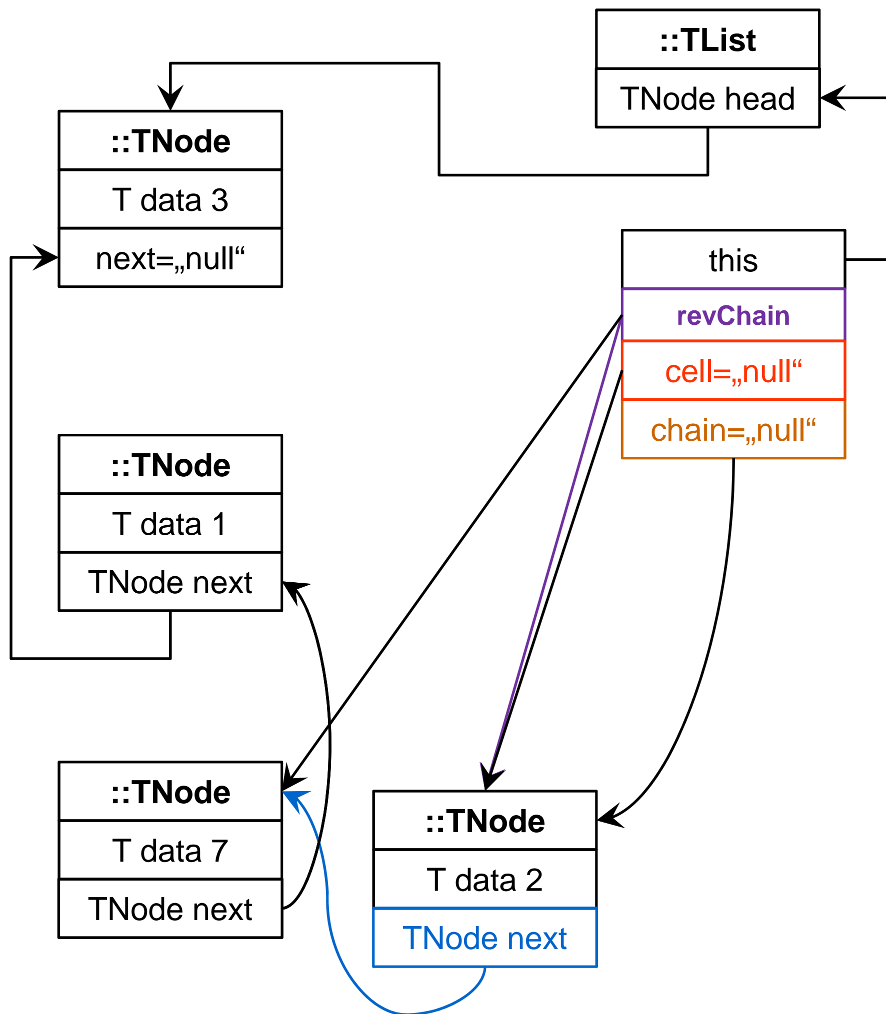
Typische Operationen auf Listen

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;
    }
}
```

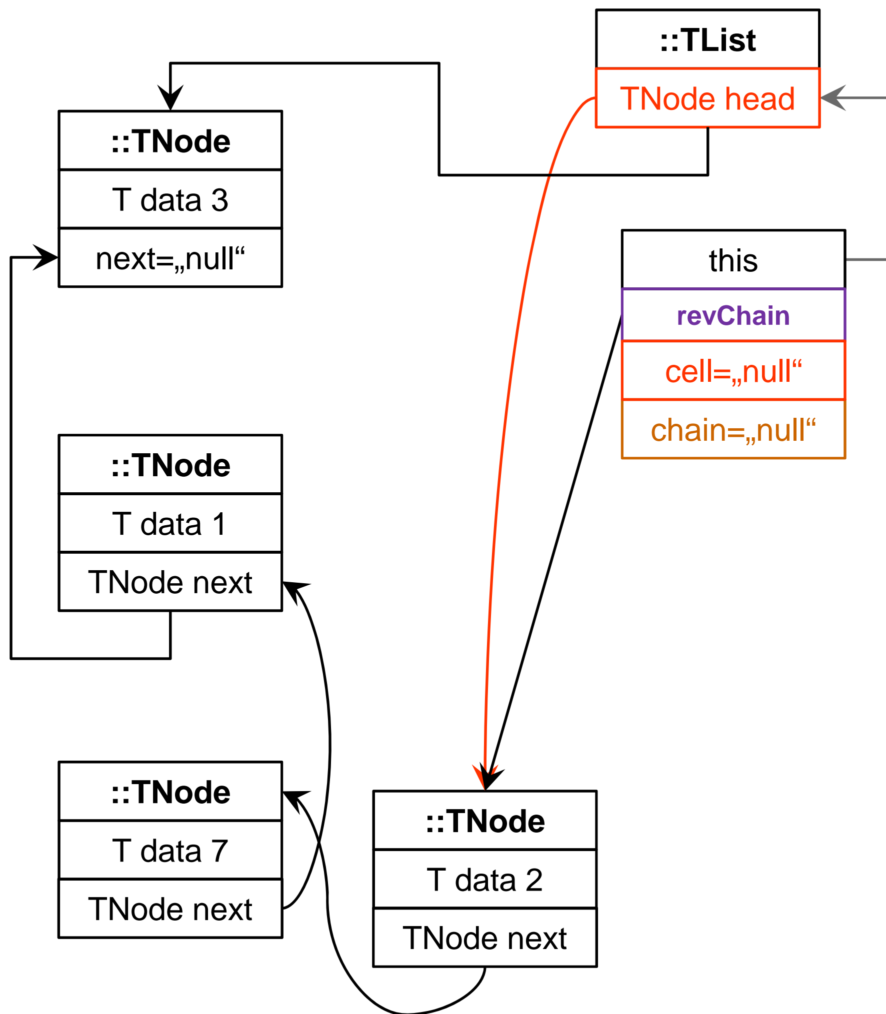
Typische Operationen auf Listen

Destruktives Invertieren




```java
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;
    }
}
```

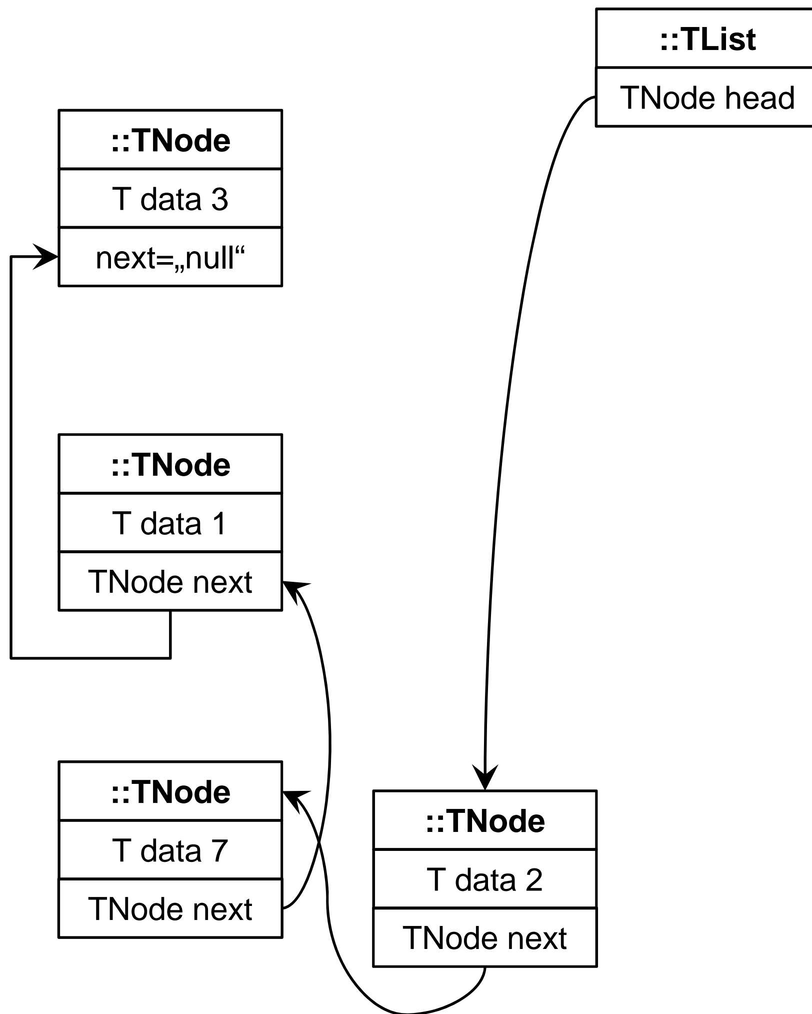
Typische Operationen auf Listen
Destruktives Invertieren




```
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

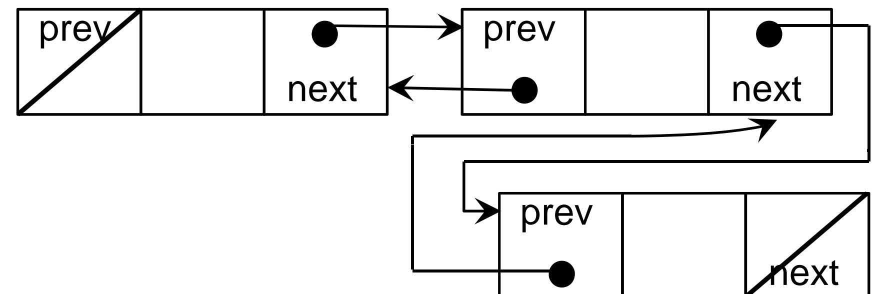



```
public class TList { // [..]
    /**
     * Invertiert die Liste destruktiv
     */
    public void reverseList() {
        TNode revChain = null; // reversed chain
        TNode cell = head;     // cell to
                               // transfer
        TNode chain;           // chain to be
                               // 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

Referenzvariablen versus Zeigervariablen

- Gemeinsamkeiten
  - Mit beiden sehr effiziente Realisierungen möglich
  - Mit 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
- Unterschiede
  - Referenzvariablen sind abstrakter als Zeigervariablen
  - Keine „Pointerarithmetik" mit Referenzvariablen möglich
    - Etwa „refvar + 1"
    - Erhöht Zuverlässigkeit und Sicherheit des Codes

Referenzvariablen versus Zeigervariablen

- In Java ist Speicherbedarf einer Referenzvariable nicht vollständig festgelegt
  - Nur Mindestgröße von 32 Bit
  - Erst die Realisierung der Java Virtual Machine legt die Größe fest
    - 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

- 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

Generische Datentypen
▸ Motivation

```
public class TNode {
    T data;
    TNode next;
    ...
}
```

Wie können wir Tlist, TStack, TQueue, etc. für Elemente eines anderen Typs als T wiederverwenden?

- Erste Idee
  - Kopieren und im Editor T durch gewünschten Datentyp ersetzen
- Vorteil
  - Leicht möglich
- Nachteil
  - Verschiedener Quellcode für alle benötigten Varianten von Listen / Stapel / Schlangen / …
    - Nachträgliche Modifikationen müssen für jede Variante nachgeführt werden
    - Wartbarkeit des Codes wesentlich eingeschränkt
  - Fazit: Kopieren und destruktive Änderungen nur vertretbar, wenn höchstens 1 bis 2 derartige Kopien benötigt werden

Generische Datentypen
▸ Motivation

```
public class TNode {
    T data;
    TNode next;
    ...
}
```

Bessere Lösungen

(1) Manuell: Verwende gemeinsamen Obertyp aller relevanten Ts und füge manuell Typkonversionen (casts) in den Code ein

- Überall wo wirklich Methoden von T gebraucht werden

(2) Sprachunterstützt: Verwende T als Typ-Parameter

- Wenn die Sprache dies zulässt → In Java seit Version 1.5

- Vorteile von (2) über (1)
  - Bessere statische Typ-Sicherheit
  - Bessere Lesbarkeit

Generische Datentypen ▸ Manuell

```
public class TNode {
     T data;
     TNode next;
     ...
}
```

- Beispiel für (1) – Manuelle „Generizität“

Wir erhalten eine „generische“ Klasse List (mit der generischen Knotenklasse Node) wenn wir in unseren Klassen TList und TNode den Typ T durch Object ersetzen:

```
public class List {
     private Node head; // Kopf der Liste
     // ...
}
public class Node {
     Object data;          // Datenelement
     Node next;            // Zeiger auf Listenzelle
     // ...
     Object getData() {…} // Datenelement herausgeben
}
```

Soweit, so gut. Aber überall wo wir die Listenelemente als T-Instanzen nutzen wollen, müssen wir manuell casts auf T einfügen… → nächste Folie:

Generische Datentypen ▸ Manuell versus Sprachunterstützt

- Manuell: „Object“ und casts

```
List list = new ArrayList();        // List: s. vorherige Seite
list.add("str");
String s = (String)list.get(0);     // cast erforderlich
…
Integer i = (Integer)list.get(0);   // cast + Laufzeit-Fehler
```

- Sprachunterstützt: Mit Typparametern

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

Genererische Datentypen ▸ Abgrenzung

- Keine Generizität

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

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

zu spezifisch, nicht wiederverwendbar für Daten eines anderen Typs

- Manuelle „Generizität"

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

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

zu allgemein, mit viel manuellem Aufwand verbunden (Code müsste lauter Casts enthalten), fehleranfällig

- Sprachgestützte Generizität: Ein Datentyp ist generisch, wenn er hinsichtlich des Typs seiner Elemente parametrisiert ist.

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

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

generisch, anwendbar auf beliebige Typen T.

T wird erst bei der Variablendeklaration und Objekt-Instanziierung festgelegt

Generische Datentypen ▸ Beispiel

Das Paket java.util enthält folgende Definitionen

- List<E> abstrahiert Listen und den Listendurchlauf:

```
public interface List<E> {
        void add(E x);
        ...
        Iterator<E> iterator();
}
```

- Iterator<E> abstrahiert den Durchlauf durch jegliche Datenstrukturen:

```
public interface Iterator<E> {
        E next();
        boolean hasNext();
}
```

Genererische Datentypen ‣ Beispiel Hastable

```
public class Hashtable<Key, Data> {
 ...
 private static class Entry<Key, Data> {
   Key key;
   Data value;
   int hash;
   Entry<Key, Data> next;
   ...
 }

 private Entry<Key,Data>[] table;
 ...
 public Data get(Key key) {
     int hash = key.hashCode() ;
     for (Entry<Key,Data> e = table[hash & hashMask];
          e != null; e = e.next) {
          if ((e.hash == hash) && e.key.equals(key) ) {
             return e.value;
          }
    }
    return null;
  }
}
```

Zur Erinnerung: Bitweises UND

Generische Typdeklarationen und Parametrisierte Typen

Generische Typdeklration

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

T ist ein „formaler Typparameter“ (wie der formale Parameter einer Prozedur)

- `Node<T>` ist eine Schablone für die Instanzierung von „Parametrisierten Typen“

Parametrisierter Typ

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

T wird bei der Variablendeklaration und Objekterzeugung durch „Typargument“ ersetzt (wie bei einem Prozeduraufruf)

- Entsteht durch „Typinstanziierung“

Typargumente können beliebige Objekttypen sein (auch parametrisierte!)

```
Node<List<Integer>> n =
      new Node<List<Integer>>();
```

Es muss aber ein Objekttyp sein („Integer“ statt „int“)

Generische Datentypen ▸ Es gibt keine Subtypbeziehung zwischen Typinstanzen

Legale Typbenutzung

```
Node<Super> n = new Node<Super>();
Node<Sub>   n = new Node<Sub>();
```

- Genau gleicher Typ auf rechter und linker Seite der Zuweisung

Illegale Benutzung

```
Node<Super> n = new Node<Sub>();
```

- Sei `Sub` ein Subtyp von `Super`
- Dann ist `Node<Sub>` kein Subtyp von `Node<Super>`
- Prinzip: Unterschiedliche Instanzen eines generischen Typs sind nie Subtypen voneinander

Generische Datentypen ▸ Es gibt keine Subtypbeziehung zwischen Typinstanzen

- Beispiel für Problem

Aus Gilad Bracha's „Generics Tutorial“
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

```
List<String> ls = new ArrayList<String>(); // 1: OK
List<Object> lo = ls;                       // 2: Illegal in Java!
```

- Zeile 1 is legal (den ArrayList ist ein Subtyp von List).
- Bei Zeile 2 stellt sich die Frage: "Ist eine 'Liste von Strings' auch eine 'Liste von Objekten' "?
- Instinktiv würden die meisten sagen: "Na klar!"
- Wenn das wahr wäre könnten wir wie folgt weitermachen:

```
lo.add(new Object());     // 3: Objekt in lo (und in ls!) einfügen
String s = ls.get(0);     // 4: Versuch einem String ein (beliebiges)
                          //    Objekt zuzuweisen!
```

- `ls` darf daher <u>nicht</u> an `lo` zugewiesen werden

Generische Datentypen ▸ Beschränkte Generizität

- Dilemma
  - T kann irgendeinen Typ darstellen
  - Wir wissen nicht welchen
  - Wir können also auch nicht auf seine Eigenschaften / Operationen zugreifen
  - Manchmal bräuchten wir das aber:

```
interface MyClass<T> {
  T getMember();
}

class Controller {
  void printMember(MyClass<T1 extends Number> c) {
    System.out.println(c.getMember().intValue());
  }
}
```

Der Compiler muss wissen, dass c eine getMember-Methode hat, sonst erlaubt er uns nicht sie aufrufen.

Generische Datentypen ‣ Beschränkte Generizität

- Prinzip
  - Angabe von Grenzen der Typvariable
    - <TypeVar **extends** Type> Wert von TypeVar muss Subtyp von Type sein
    - <TypeVar **super** Type> Wert von TypeVar muss Obertyp von Type sein
  - Bei Angabe einer oberen Schranke weiß man, worauf man sich mindestens verlassen kann
    - Z.B. Existenz von getMember() in Subtypen von Number

```
interface MyClass<T> {
  T getMember();
}

class Controller {
  void printMember(MyClass<T1 extends Number> c) {
    System.out.println(c.getMember().intValue());
  }
}
```

So teilen wir dem Compiler mit, dass der Typ von C unbekannt aber garantiert ein Subtyp von Number ist.

Generische Methoden

- Die folgende Funktion scan implementiert den generischen Listendurchlauf.

```
public class List {
    private Node head; // Kopf der Liste
    // ...
    void scan () {
        for (Node cursor=head; cursor!=null; cursor=cursor.next) {
            // Aktion (zur Zeit leer)
            // etwa System.out.println(cursor.data.toString());
        }
    }
}
```

In `Object` definierte Methode `toString()`

- Sie funktioniert völlig unabhängig von welchem Typ die Datenelemente der Knoten sind

Generische Methoden

- Im allgemeinen sind die Methoden für generischen Datentyp in einem Interface definiert
  - Nicht schon in der Urklasse Object

- Erläutern Einsatz am Beispiel von Vergleichen
  - Sowohl für Typparameter als auch bei der Realisierung mittels `Object`, casts und instanceof-Tests
    - die vom Programmierer manuell eingefügt werden

Generische Vergleiche

- Viele Algorithmen zum Suchen und Sortieren können als generische Algorithmen implementiert werden, die durch einen Vergleichsoperator parametrisiert sind
- In Java wird dazu ein Interface **`Comparable`** im Paket **`java.lang`** zur Verfügung gestellt, in dem eine Vergleichsmethode **`compareTo`** wie folgt spezifiziert ist:
  - **`compareTo`** liefert eine Zahl kleiner als 0, gleich 0, oder größer als 0 zurück, je nachdem, ob das Objekt, auf dem **`compareTo`** läuft, kleiner, gleich, oder größer als das Argument-Objekt ist
    - „3-wertiger Vergleich“
    - Vgl. **`String`**-Klasse
  - Wenn das Objekt, mit dem verglichen wird, nicht den gleichen Typ besitzt, dann wird ein Ausnahmeobjekt geworfen
    - Dies ist die Spezifikation im Interface
    - Eine Implementierung hat diese Spezifikation zu erfüllen

Generische Vergleiche (manuell)

- Beispiel

```
public class List {
    private Node head; // Kopf der Liste
    /**
     * Returns true iff m is smaller than all elements
     * of type Comparable in this list.
     */
    public boolean isMin(Comparable m) {
        for (Node cursor = head;
                  cursor !=null;
                  cursor = cursor.next) {
            if (cursor.data instanceof Comparable) {
                Comparable o =
                    (Comparable) cursor.data;
                if (o.compareTo(m) < 0) return false;
            }
        }
        return true;
    }
}
```

Referenztyp einer Schnittstellen-Klasse

Test mit **`instanceof`** Operator

Generische Vergleiche mit Typparametern

- Beispiel

```
public class List<T> {
    private Node<T> head; // Kopf der Liste
    /**
     * Returns true iff m is smaller than all elements
     * in this list.
     */
    public boolean isMin(Comparable<T> m) {
        for (Node<T> cursor = head;
                     cursor !=null;
                     cursor = cursor.next) {
            if (m.compareTo(cursor.data) < 0)
                return false;
        }
        return true;
    }
}
```

Generische Vergleiche in Java Standard-Bibliotheken

- In den Java Standard-Bibliotheken wird häufig das Interface Comparator zugrunde gelegt
  - Wie Comparable, spezifiziert jedoch eine totale Ordnung
- Die Methoden zum Finden eines Minimums (Maximums, Sortieren, …) sind 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.

Sprachunterstützte Generizität in Java: Generelle Informationen

- Erst seit Java 5.0 in Sprache definiert
  - D.h. JDK 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 aus und eliminiert sie
    - Einfügen von casts, 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

Generizität in C++: „Template classes“

Gemeinsamkeiten und Unterschiede zu Java

Template classes in C++

- Verwendung von Typ-Parametern findet auch in anderen Sprachen (zunehmend) Verwendung
  - Etwa Einführung von Template-Klassen in C++ und ihre Verwendung in Standard-Bibliotheken wie der „Standard Template Library“ (STL)
- Folgendes ist das Fragment einer generischen Knotenklasse in **C++** (**kein Java-Code**)

```
template<class T> class Node {
     T data;      // generic data field
     Node* next; // pointer to next Node
  public:
     T get_data() { return(data); }
     set_data (T d) { data = d; }
}
```

  - Instanzierung zu Knoten über `int` **in C++** mit: `Node<int> n;`
  - Die Java-Syntax wurde absichtlich ähnlich gestaltet

Template classes in C++

- Zur Übersetzzeit werden die verschiedene Ausprägungen des „generischen Programms" vom Compiler generiert
  - **Warum**: In **C++** gibt es keine gemeinsame Oberklasse wie **`Object`**
  - **Vorteil**: Keine Indirektion über Virtual Function Table notwendig
    - Gerade bei relativ einfachen Operationen etwa auf Listen kann Indirektion zu einer wesentlichen Erhöhung (grob: Verdoppelung) der Laufzeit führen
  - **Nachteil**: Übersetzer erzeugt (i.A.) separaten Code für jede Typ-Instanz
    - Aufblähung des übersetzten Codes (code bloat)

Template classes in C++

- In C++ können Templates tief ineinander verschachtelt werden
  - „Rekursion“ von Templates möglich
  - Auch Fallunterscheidungen möglich
- Damit muss der Compiler zur Auflösung der Templates evtl. beliebig komplizierte Berechnungen durchführen
  - Die auch nicht zu terminieren brauchen
  - Rekursion + Fallunterscheidung ergibt „Turingvollständigkeit“
    - Alle theoretisch berechenbare Funktionen können mit Templates dargestellt werden!
- Berechnungen mit Templates werde in der Praxis genutzt
  - „C++ Template Metaprogramming“

universitätbonn

Bäume

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

Grundkonzepte von Bäumen

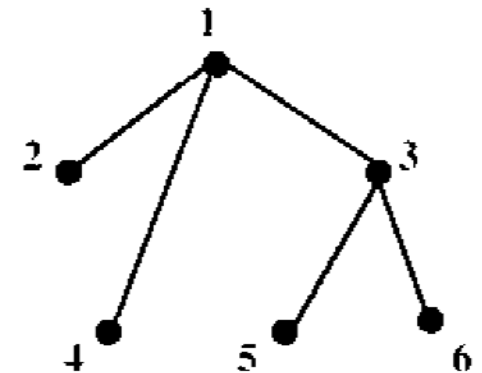
- Bäume (trees) können als eine Verallgemeinerung von Listen angesehen werden
  - Bei einer Liste hat jeder Knoten einen Nachfolger
  - Bei einem Baum hat jeder Knoten potentiell mehrere „Nachfolger“
  - Kinderknoten (child, children) genannt
  - Wie bei einer Liste hat jeder Knoten (bis auf den Kopfknoten)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

Grundkonzepte von Bäumen

- Ein Knoten ohne Elternknoten wird Wurzel (root) genannt
  - Baum wird i.A. eine Datenstruktur genannt, die genau eine Wurzel besitzt
  - Sonst wird von einem „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 Wurzel hat somit die Ebene 0
- 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 hier dargestellten Baum
  - ist Knoten 3 der Elternknoten von Knoten 5 und 6
  - sind die Knoten 2, 3 und 4 Kinder von 1
  - liegt der Knoten 1 auf Ebene 0,
    die Knoten 2, 3 und 4 auf Ebene 1 und
    die Knoten 5 und 6 auf Ebene 2
  - ist die Höhe des Baumes gleich 2

Grundkonzepte von Bäumen

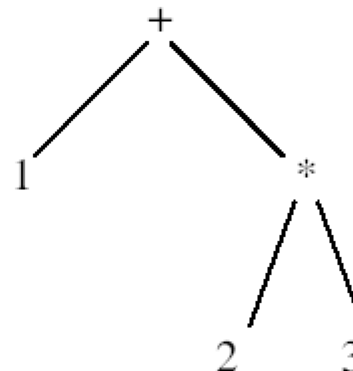
- Der **Verzweigungsgrad** (out degree) eines Knotens ist die Anzahl seiner Kinder
- Ein **Binärbaum** (binary tree) ist ein Baum, dessen Knoten höchstens den Verzweigungsgrad 2 haben
  - Der Baum des vorigen Beispiels ist kein Binärbaum, da die Wurzel Verzweigungsgrad 3 hat

Anwendungen von Bäumen

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

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)



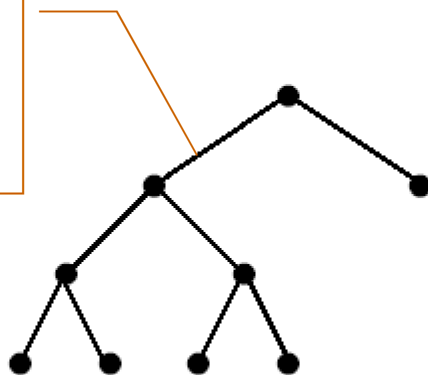
Grundkonzepte von Bäumen

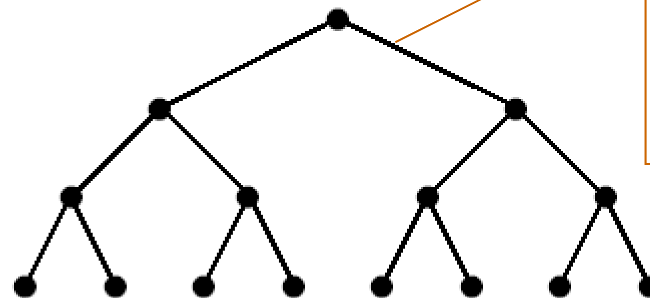
- **Bemerkung:** Wir definieren hier Bäume als Verallgemeinerungen von Listen
  - Man kann Bäume aber auch als einen *Spezialfall* einer anderen für die Informatik sehr wichtigen Datenstruktur definieren, nämlich der eines Graphen
    - Ein (gerichteter) **Graph** (directed graph) ist ein Paar (V,E) 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

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

Beispiel eines vollen Baumes





Beispiel eines vollständigen Baumes

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*

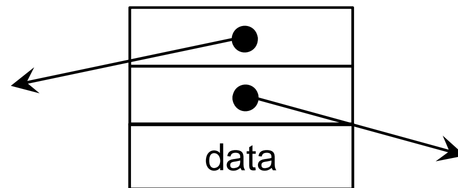
In einem vollständigen Baum ist die Anzahl der Blätter gleich der Anzahl der inneren Knoten minus 1.

Eigenschaften von Bäumen: Bemerkungen

- Vorherige Zusammenhänge zwischen Höhe eines Baumes und Anzahl der Knoten bzw. Blätter sind ein Grund für die Bedeutung von Bäumen, z.B. in Datenbanksystemen
- Bei „Suchbäumen“ wächst die Höhe nur „logarithmisch“ mit der Anzahl der Blätter
  - 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




```
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;
    }
}
```

Das Paket „tree“ kapselt die Klasse „Node“

Da wir im Paket tree sind, können wir den gleichen Namen für unsere Klasse benutzen wie für Listenknoten (keine Namenskollision)

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 Tree-Objekt t mit t.root=null

```
/**
 * Class for a generic binary tree.
 */
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);
    }
}
```

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 that
         * operates 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
  - Zweck: Ausgabe der Daten

```
package tree;

public class NodePrintAction implements NodeActionInterface {

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

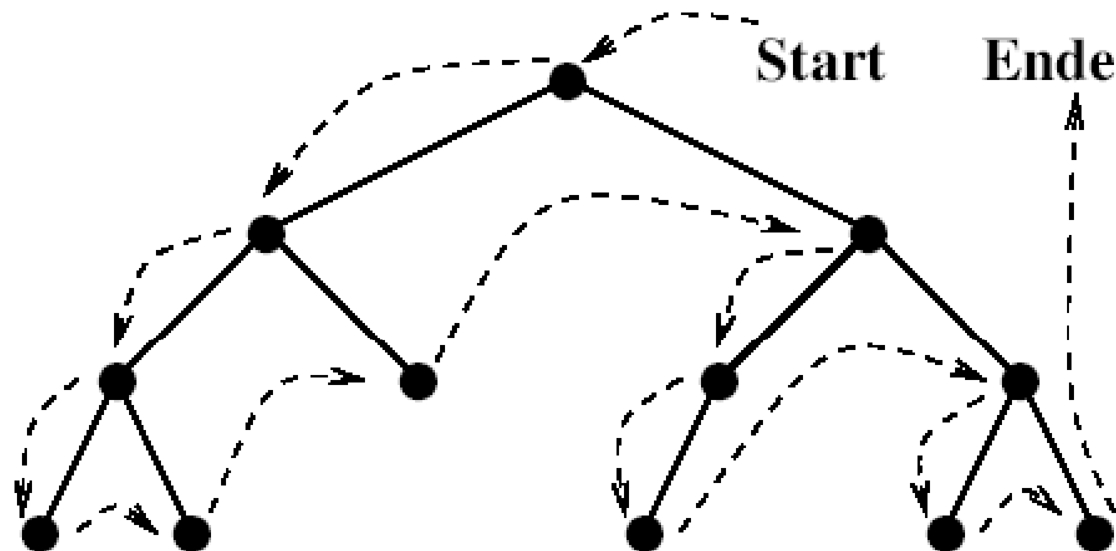
Baumdurchläufe

- Beispielklasse, die NodeActionInterface implementiert
  - Zweck: die inneren Knoten sollen gezählt werden
  - Wieder ein fold-Operator

```
package tree;

public class CountInnerNode 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 ++;
    }
}
```

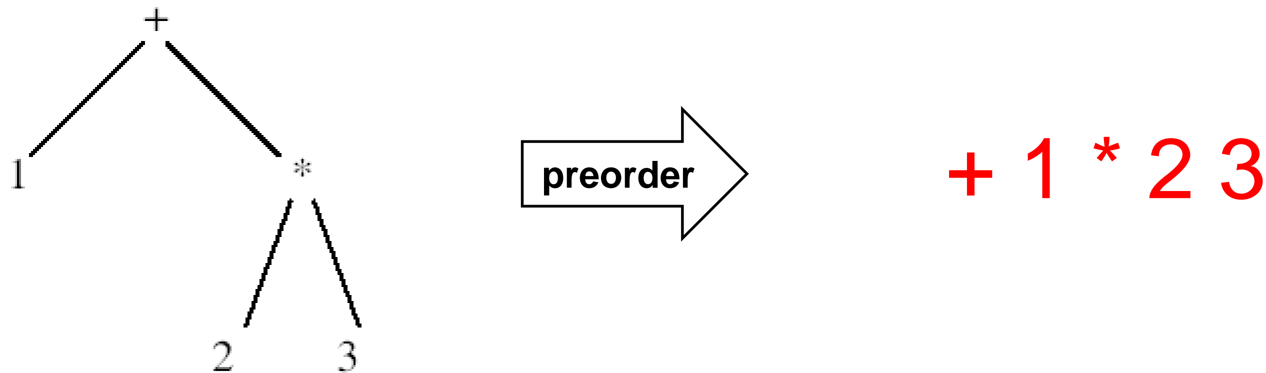
Baumdurchläufe: Präorder

- Das abstrakte Verfahren zum Durchlauf in Präorder lautet:
  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

- Ein Baumdurchlauf für den Strukturbaum zu „1+2*3“ in Präorder mit der Operation *Drucke Symbol* erzeugt die Ausgabe: + 1 * 2 3



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

Baumdurchläufe: Präorder (rekursiv)

```
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) return;                   // empty tree?

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

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

Baumdurchläufe: Präorder (nicht-rekursiv)

```
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 preorder(NodeActionInterface p) {
        java.util.Stack stack = new java.util.Stack();
        stack.push(root);                              // initialize
        while (!stack.isEmpty()) {
             Object tmp = stack.pop();
             if (tmp != null &&                        // empty tree?
                  tmp instanceof Node)                 // avoid instanceof error
             {
                  Node tmpn = (Node) tmp;
                  p.action(tmpn);                      // visit node: apply function
                  stack.push(tmpn.right);              // visit right subtree
                  stack.push(tmpn.left);               // visit left subtree
             }
        }
   }}
```

Für Erläuterungen der nicht-rekursiven Version siehe die nächsten 2 Skript-Seiten.

Baumdurchläufe: Präorder (nicht-rekursiv)

- Wollen wir keinen rekursiven Aufruf für die Tiefensuche verwenden, brauchen wir einen Stack, auf dem wir uns die zu behandelnden Knoten für später merken
  - Rekursive Aufrufe verwenden den Laufzeitstapel

Baumdurchläufe: Präorder (nicht-rekursiv)

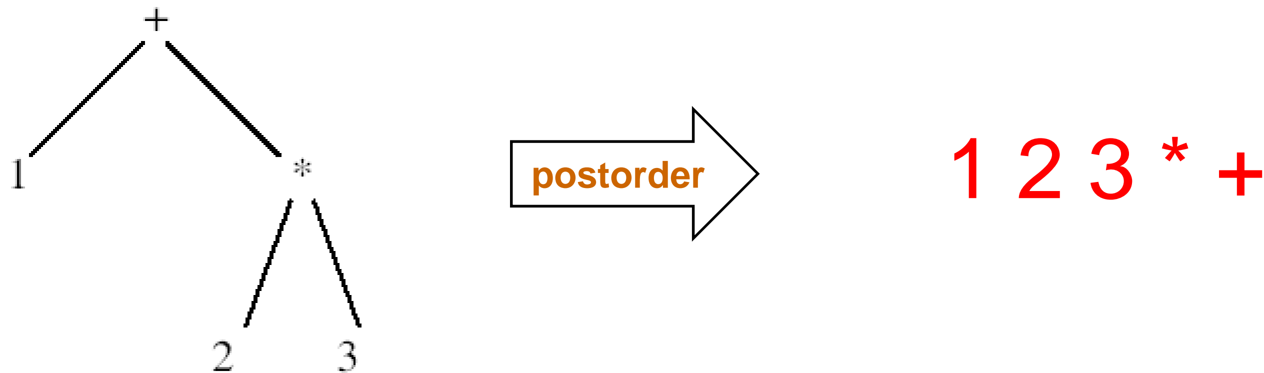
- Wir verwenden hier die Klasse java.util.Stack (in ihrer nicht-generischen Fassung aus dem JDK 1.4)
  - Den Stack von Elementen vom Typ Object benutzen wir für Elemente vom Typ Node
  - Da die Methode pop() ein Ergebnis vom Typ Object zurückliefert, überprüfen wir mittels des instanceof-Operators, ob es sich um einen Node handelt und spezialisieren es dann per Typumwandlung (‚cast') zu einem Node
  - Da wir vor dem Aufruf der Methode pop() testen, ob der Stack leer ist, kann die Ausnahme EmptyStackException nicht vorkommen
    - Java-Compiler können eine solche Analyse 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 Klasse EmptyStackException, stellt eine ungeprüfte Ausnahme dar, da sie eine Unterklasse von RuntimeException ist

Baumdurchläufe: Postorder

- Das abstrakte Verfahren zum Durchlauf in Postorder lautet:
  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

- Ein Baumdurchlauf für den Strukturbaum zu „1+2*3“ in Postorder mit der Operation *Drucke Symbol* erzeugt die Ausgabe: 1 2 3 * +



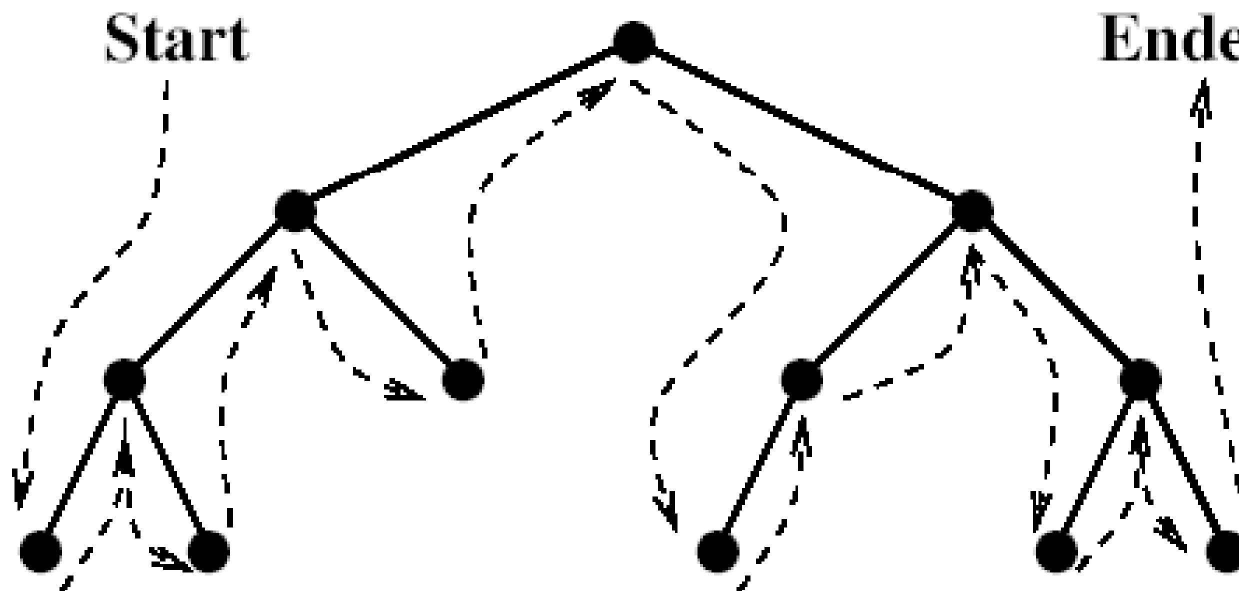
- Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Postorder entspricht der **umgekehrten polnischen Notation** (‚reverse Polish notation') für Ausdrücke.

Baumdurchläufe: Präorder (rekursiv)

```
package tree;

public class Tree {

    protected Node root;
    // …

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

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

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

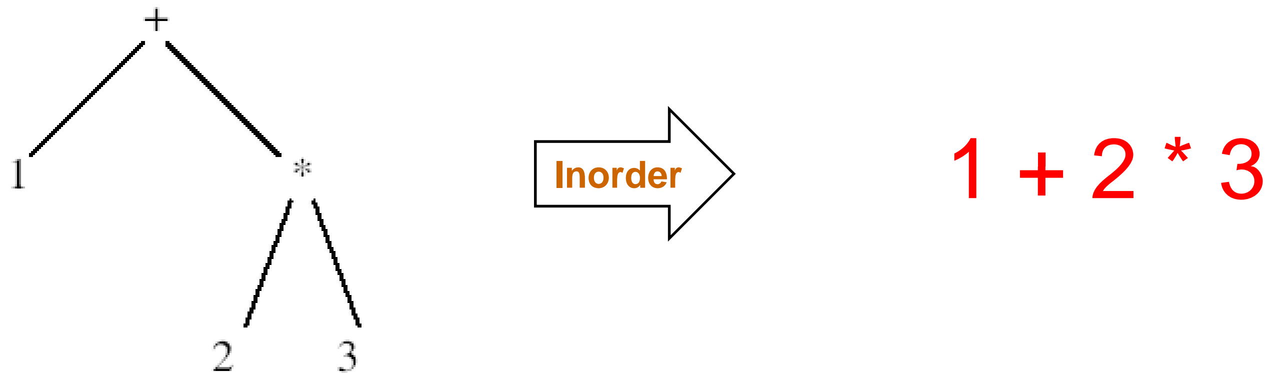
Baumdurchläufe: Inorder

- Das abstrakte Verfahren zum Durchlauf in Inorder lautet:
    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

- Ein Baumdurchlauf für den Strukturbaum zu „1+2*3“ in Inorder mit der Operation *Drucke Symbol* erzeugt die Ausgabe: 1 + 2 * 3



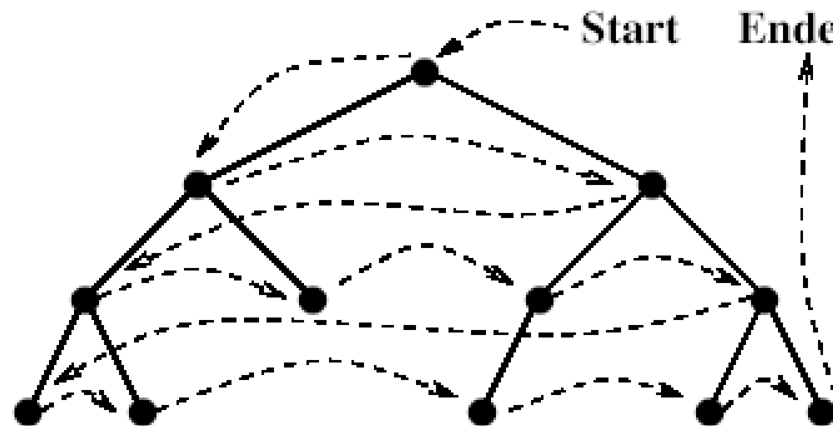
- Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Inorder entspricht der normalen Operator-Schreibweise (wenn zusätzlich die Teilausdrücke noch geklammert werden)

Baumdurchläufe: Inorder (rekursiv)

```
package tree;

public class Tree {

    protected Node root;
    // …

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

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

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

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 Levelorder geht man nicht zuerst in die Tiefe (depth first), sondern die Strategie heißt Breite zuerst (breadth first)
- Dies kommt besonders bei **Suchbäumen** (search trees) 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 hat sie eine Länge von $n/2$ bei n Knoten (= Blätter eines vollständigen Binärbaums)
  - Bei den „depth first" Methoden 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 1 tiefer als der zu durchlaufende Baum selbst. Damit ist die maximale Tiefe $1+\log_2 n$.

Baumdurchläufe: Inorder (rekursiv)

```
package tree;

public class Tree {

    protected Node root;
    // …

    /**
     * Applies the function p.action to any node of the tree in preorder sequence.
     */
    public void levelorder(NodeActionInterface p) {
        Queue queue = new Queue();
        queue.append(root);
        while (!queue.isEmpty()) {
            Object tmp = queue.get();
             if (tmp != null &&                           // empty tree?
                 tmp instanceof Node)                     // avoid instanceof error
             {
                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 p) {
        traversePreorder(root,p);
    }

    private void traversePreorder(Node n, NodeActionInterface  p)  {

        if (n == null) return;              // Trivial case

        p.action(n);                        // Action

        traversePreorder(n.left, p);        // Recursion
        traversePreorder(n.right, p);       // Recursion
    }
}
```

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 der Test

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

  unnötig (außer zu Beginn)
- Lösung: zweite Mantelprozedur

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

  mit Aufruf nur, wenn Baum nicht leer.

Speicheroptimierung durch Mantelprozedur (jacket)

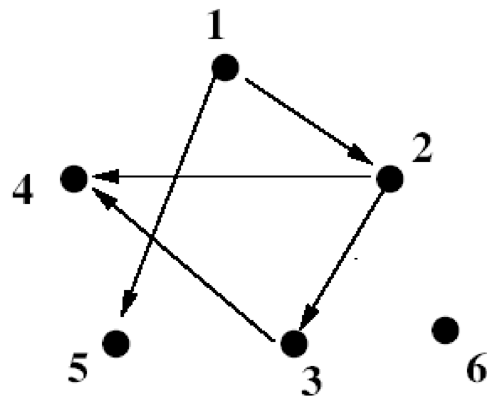
```
package tree;

public class Tree {
    protected Node root;
    // …

    public void preorder(NodeActionInterface p) {
        if (root == null) return;                          // empty tree
        else preorderNonEmpty(root, p);
    }

    private void preorderNonEmpty(Node n, NodeActionInterface  p)  {
        p.action(n);                                        // Action

        if (n.left != null) preorderNonEmpty(n.left, p);    // Recursion
        if (n.left != null) preorderNonEmpty(n.right, p);   // Recursion
    }
}
```

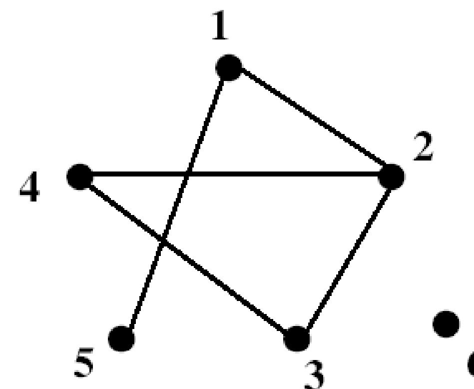
universitätbonn

Graphen

Graphen

- 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 hier gezeigte Graph ein Baum?
  - Beweis der Aussage (Übung)