E7 Ergänzungen zu:

Algorithmen und Datenstrukturen (Kapitel 17 + 18)

Die Foliensammlung liefert in *skriptartiger Form* zusätzliche Informationen:

Es werden einige weitere Sortieralgorithmen vorgestellt sowie im Buch behandelte Themen durch zusätzliche Abbildungen erläutert, die aus Platzgründen nicht aufgenommen werden konnten.

Eine Bemerkung zur **Formulierung von Algorithmen** :

Neben der schematischen Darstellung des logischen Ablaufs durch Flussdiagrammsymbole und der programmiersprachen-unabhängigen Notation in Pseudocode existieren noch weitere Schemas.

Dazu gehören insbesondere sogenannte Struktogramme (Nassi-Schneidermann-Diagramme), die für jede der vorgestellten Strukturelemente der Programmierung spezifische Symbole vorsehen. Allerdings gehen wir nicht weiter auf diese ein, da sich die behandelten Algorithmen in Pseudocode oder durch direkte Implementierung in Java verständlicher darstellen lassen. Für komplexere Algorithmen werden Struktogramme schnell zu umfangreich und unübersichtlich, so dass sie in der Praxis eher selten Verwendung finden.

Quadratische Sortierverfahren: Zeitkomplexität $O(n^2)$

Sortieren durch Auswählen: Selection Sort

Strategie:

Suche nach kleinstem Element + Austausch mit erstem noch unsortierten Element

1. Finde Element mit **kleinstem** Schlüssel (= Wert) und vertausche es mit dem *ersten* Element der Folge
2. Finde Element mit **zweitkleinstem** Schlüssel und vertausche es mit dem *zweiten* Element der Folge
3. Finde Element mit **drittkleinstem** Schlüssel und vertausche es mit dem *dritten* Element der Folge

.....

i. Finde Element mit **i-kleinstem** Schlüssel und vertausche es mit dem ***i-ten*** Element der Folge

.....

n. Finde Element mit zweitgrößtem Schlüssel und vertausche es mit dem *vorletzten* Element der Folge

Resultat:

Nach jedem Schritt hat **vertauschtes Element** seine **Endposition erreicht** .....

..... nur noch **immer kleiner werdende Rest** der Folge ist unsortiert

$\Rightarrow$ nach **n** solchen Schritten ist gesamte Folge sortiert

Selection Sort

```
class Reihung  {
   private int[] s ;            // Bsp: Integers = Schlüssel
   private int counter ;     // Anzahl der Elemente

   public Reihung( int size )  {   s = new int[size];   counter = 0 ;   }

   public void insert( int x ) {   s[ counter ] = x ;      counter++ ;   }

   public void selectionSort( )  {
      int temp = 0 ;     int small = 0 ;
      for ( int i=0 ; i <  s.length ;  i++)  {
         small = i ;
         for ( int j = i+1 ;  j <  s.length ;  j++)  {
            if ( s[j] < s[small] )  {  small = j ;  }
          }
         if ( small !=  i )  {
            temp = s[ small ] ;   // kleinstes Element an
            s[ small ] = s[ i ] ;    // Position i setzen
            s[ i ] = temp ;
          }
       }
    }
}
```

Äußere Schleife:

Über das gesamte Array

Innere Schleife:

Suche nach dem kleinsten Element der Restfolge ab der **(i+1)**-ten Position bis zum **Ende** des Arrays

Wenn kleinstes Element schon am Anfang steht : small == i ⇒ *nix tun*

Ansonsten: Kleinstes Element mit Element an Position **i** vertauschen

Zeitkomplexität:

Wegen geschachtelter Schleifen ebenfalls **O(n²)**

Jedoch **quantitativ** besser als Bubblesort, da "intelligenter" ! **n² / 2**

Berechnung der Zeitkomplexität für n Datenelemente

BubbleSort

Äußere Schleife: wird **(n-1)** mal durchlaufen

Innere Schleife: wird **(n-1)** mal durchlaufen

⇒ Insgesamt: $(n-1) * (n-1) = n^2 - 2n + 1$ Iterationen

⇒ Zeitkomplexität ist **$n^2 - 2n + 1$**

⇒ **Asymptotische Zeitkomplexität (für lim n → ∞) = n^2**

⇒ **Zeitordnung = $O(n^2)$**

SelectionSort

Äußere Schleife: **n** -mal durchlaufen, dabei gilt:

Innere Schleife: 1.Durchgang **(n-1)** mal durchlaufen

2. Durchgang **(n-2)** mal durchlaufen

x.Durchgang **(n-x)** mal durchlaufen

⇒ Insgesamt: **(** (n-1) + (n-2) + ... + 2 + 1 **)** = $n * (n-1) / 2 = n^2 / 2 - n / 2$

⇒ Zeitkomplexität ist **$n^2 / 2 - n / 2$**

⇒ **Asymptotische Zeitkomplexität (für lim n → ∞) = $n^2 / 2$**

⇒ **Zeitordnung = $O(n^2)$**

Unterquadratische Sortierverfahren: Merge Sort

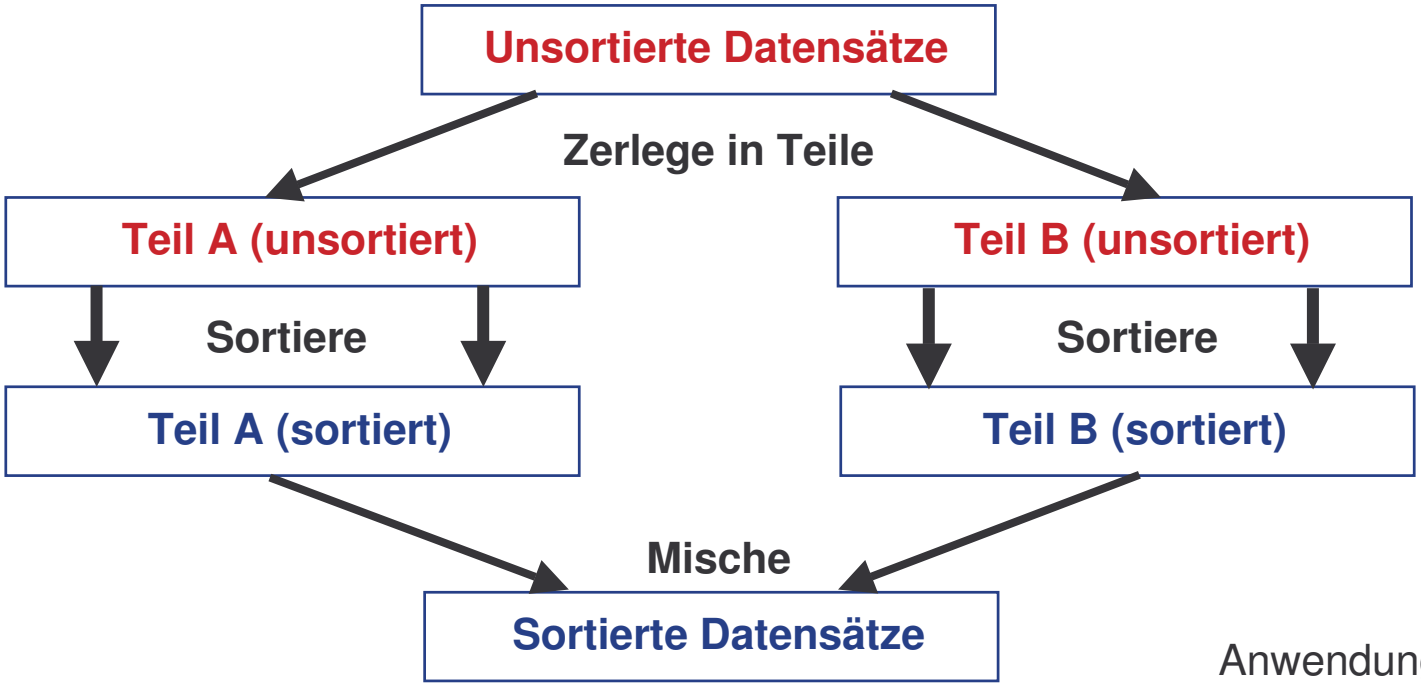
Sortieren durch Teilen & Mischen: Zeitkomplexität: Ø **$O(n \log_2 n)$**

Aufteilung der Folge in zwei Hälften ("teile und herrsche")

→ Hälften werden für sich sortiert + anschließend zusammen**gemischt**:

Dabei werden immer die vordersten Elemente der sortierten Hälften verglichen und das jeweils kleinste Element entnommen (**Merge**-Vorgang)

⇒ **Merge:** Aus zwei geordneten Teilhälften entsteht eine geordnete Liste aller Elemente

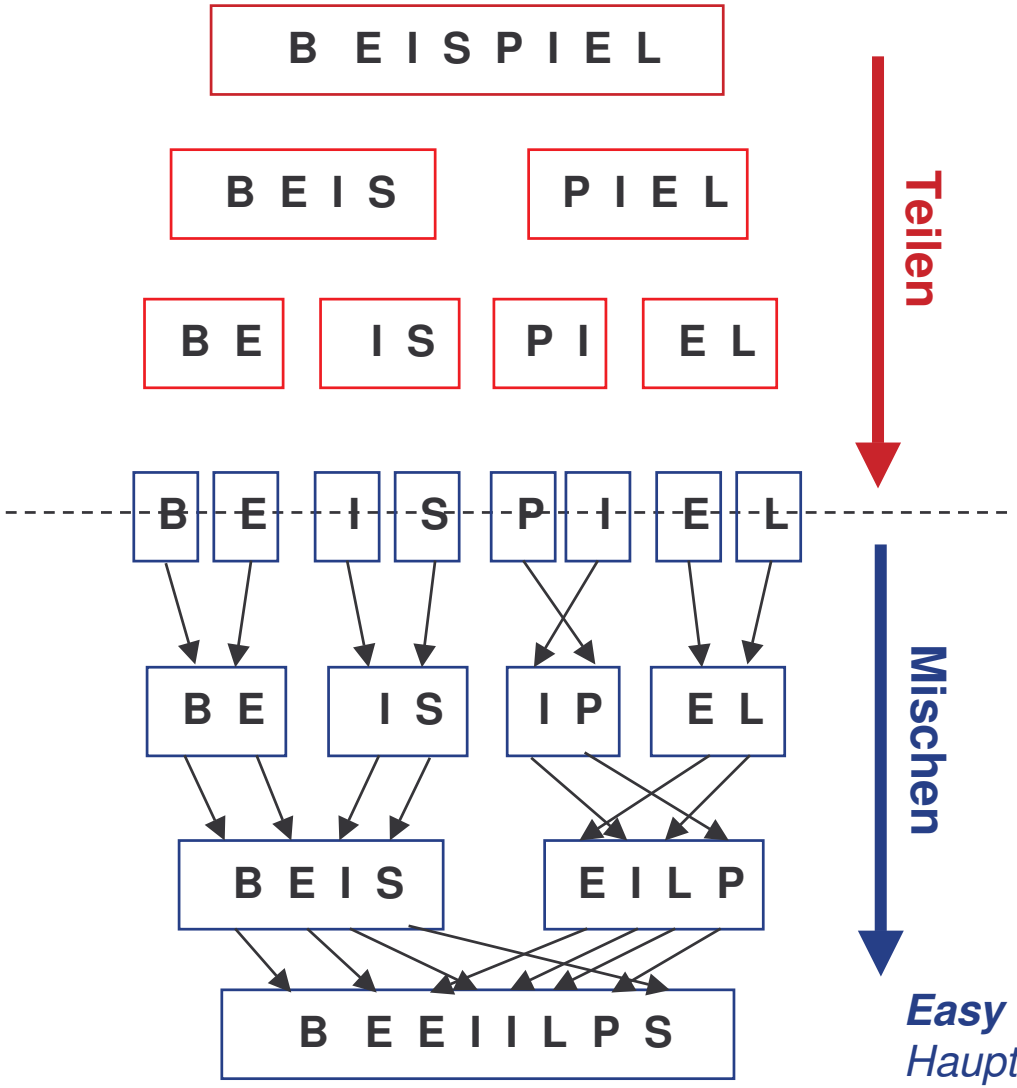


Anwendungsbereich:
Externes Sortieren

Merge Sort

Rekursives Verfahren :

Teilen des Datenbestands wird solange fortgesetzt, bis nur noch ein Element übrig ist



Mischvorgang:

Die **vordersten** Elemente aus zwei Teilmengen werden verglichen und das **Kleinste** wird entnommen.

Mit den beidenTeilmengen **fortsetzen**, bis all ihre Elemente geordnet sind.

⇒ Zwei Teilmengen **vereinigt** !

⇓

Mit **größeren Teilmengen** fortsetzen, bis alle Teilmengen gemergt sind.

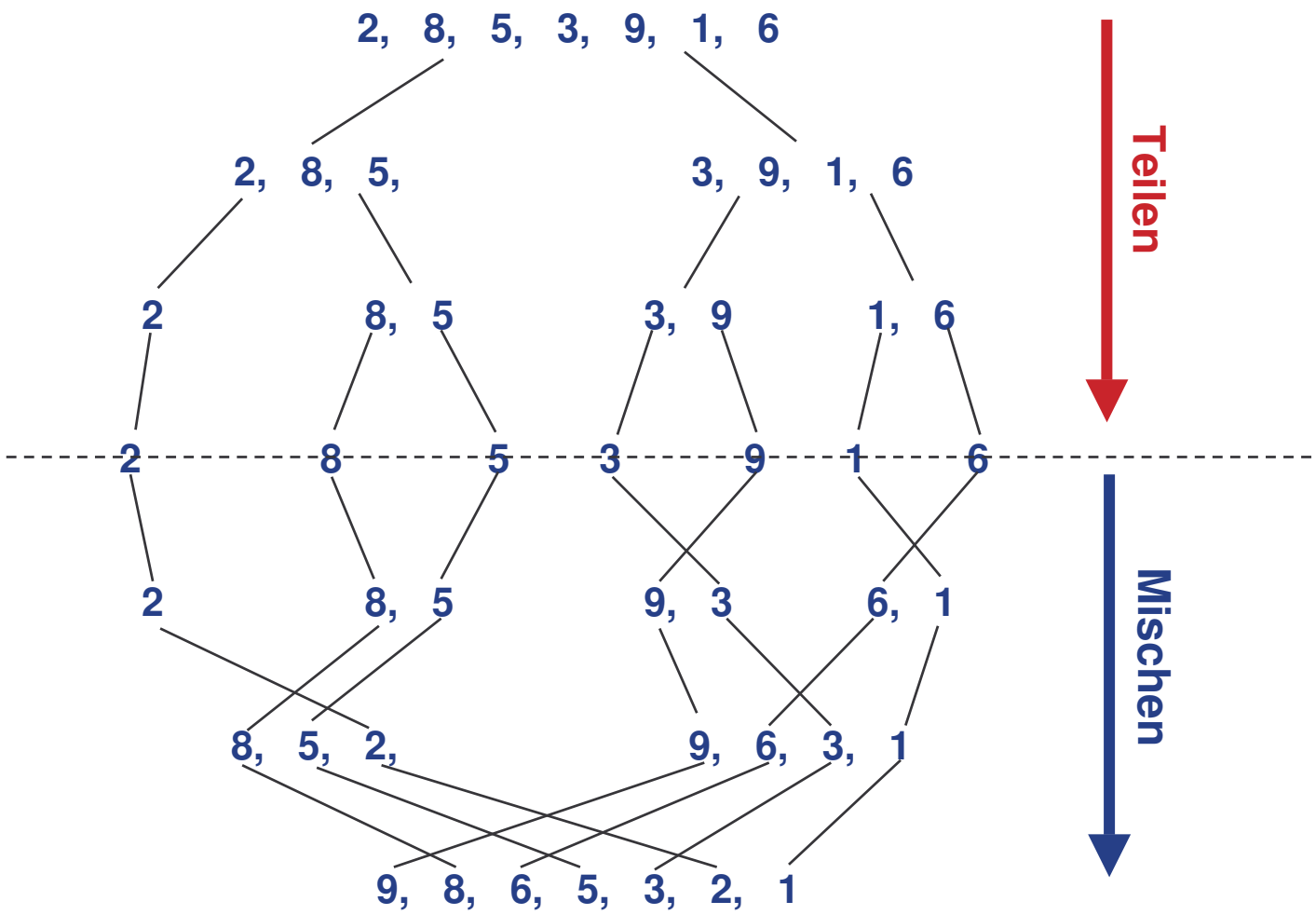
< Mischen beendet >

Easy split / hard join: *Aufteilen ist trivial, Hauptarbeit beim Zusammensetzen verrichtet ...*

Merge Sort

Rekursives Verfahren:

Teilen des Datenbestands wird solange fortgesetzt, bis nur noch ein Element übrig ist



hier: absteigende Sortierung

Merge Sort (8)

Rekursiver Aufruf von ***Sortiere*** bewirkt immer feinere **Zerlegung** der Ursprungsmenge, bis jeder Teil nur noch aus **einem** Element besteht

Rekursiver Sortiervorgang:

***Sortiere*(von Lo bis Hi):**

Wenn Lo >= Hi **dann** kein weiterer Split, da nur noch 1 Element

Andernfalls (wenn Lo < Hi)

split = (Lo + Hi) / 2

Sortiere(von Lo bis split)

Sortiere(von split + 1 bis Hi)

AbmischeTeile(Teil1(Lo bis split) und Teil2(split+1 bis Hi))

Abmischen / Merge: *// nur grobe Näherung*

***AbmischeTeile*(B und C):**

j = 0; k = 0 ; Array **Temp** (mit Größe B + Größe C)

Wiederhole von i=0 bis n= (Größe B + Größe A) - 1

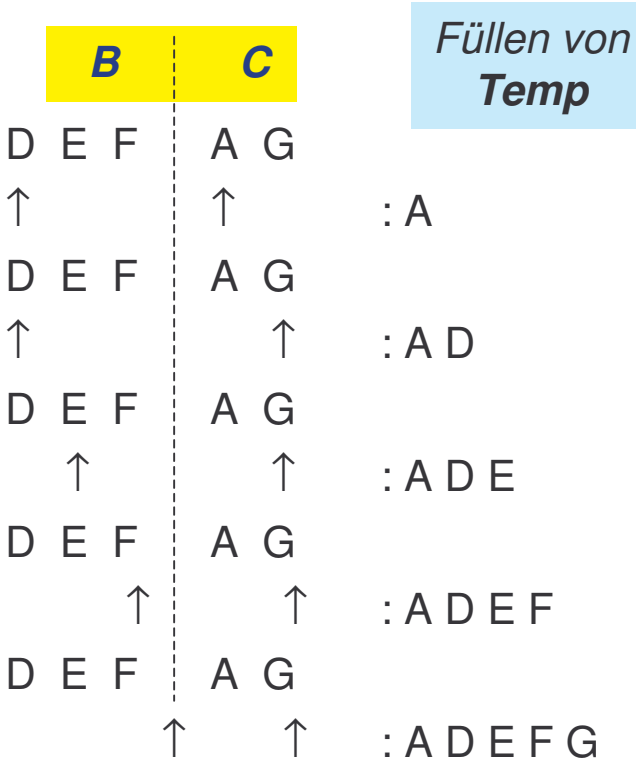
Wenn B[j] < C[k] **dann**

Temp[i] = B[j] ; j = j + 1

Andernfalls

Temp[i] = C[k] ; k = k + 1

// Problem des Indexüberlaufs nicht erfasst!



Merge Sort (9)

```
class Merge  {
 //  int[] b = new int[s.length] ;  // Hilfsarray!
  public void MergeSort( int[] s ) {
      msort ( s, 0, s.length-1 ) ;  // gesamtes Array s sortieren
   }
  private void msort( int[] s, int Lo, int Hi )  {
    if ( Lo < Hi ) {    // andernfalls nur ein Element!
       int split = ( Lo + Hi ) / 2 ;
       msort(  s, Lo, split ) ;
       msort(  s, split + 1, Hi ) ;
       int j =0 ;   int l = Lo ;   int h = split + 1 ;
       int n = Hi - Lo + 1 ;
       while( l<=split &&  h <= Hi )  {
          if ( s[l] < s[h] )  { b[j] = s[l] ;    j++ ;   l++ ; }
          else                { b[j] = s[h] ;   j++;   h++ ; }
        }
       while ( l <= split )  { b[j] = s[l] ;   j++ ;   l++ ;  }
       while ( h <= Hi )     { b[j] = s[h] ;  j++ ;   h++ ; }
       for( j=0 ; j< n ; j++ )   {  s[Lo + j] = b[j] ;  }
   }
  }
}
```

Berechnung Teilungsindex

Rekursive Selbstaufrufe für die Teilhälften

Hilfsarray für Hälften

Vergleich der Elemente zweier Hälften. Ablage in sortierter Reihenfolge in Hilfsarray b

Rechte Hälfte schon leer

Linke Hälfte schon leer

MergeSort ist etwas langsamer als QuickSort, aber unproblematisch in Bezug auf teilsortierte Folgen, bei denen Quicksort zT schlechte Resultate zeigt (worst case).

Hat jedoch höhere Speicherkomplexität als Quicksort.

Theorie zur bestmöglichen Zeitkomplexität von Vergleichsverfahren

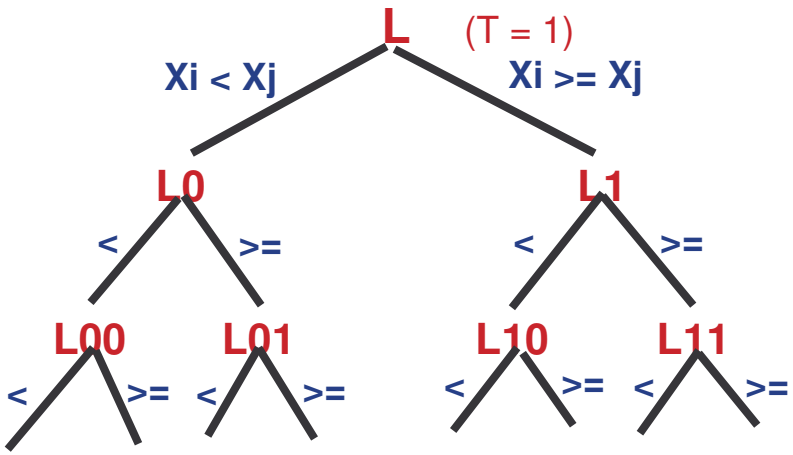
Prinzip der bisherigen Sortierverfahren: Vergleichen + Vertauschen

Zur sortieren sei Liste **L** = { X1, X2, ..., **Xi**, ... , **Xj**, ..., Xn } mit **n** Elementen

Sortiervorgang = Vergleich zweier Elemente Xi und Xj + Vertauschaktion durchführen

⇒ Liste wird durch Vertauschaktion verändert - in **neue** Liste übergeführt

⇒ Gesamtheit aller möglichen Veränderungen = **Entscheidungsbaum :**



Tiefe **T** des Entscheidungsbaums entspricht der Anzahl der **Entscheidungen E + 1,** die zu einer speziellen Sortierung führen =

Anzahl der dazu im Mittel erforderlichen **Iterationen** des Algorithmus

Unterste Ebene des Entscheidungsbaums umfasst **alle möglichen** umsortierten Versionen der Ausgangsliste L : (Endknoten)

n-Listenelemente ⇒ **n!** Permutationen

= Anzahl möglicher Sortierungen

⇒ **Entscheidungsbaum umfasst n! Endknoten**

Binärbaum mit Tiefe **T hat** maximal $2^{(T-1)}$ Elemente in der Tiefe **T** :

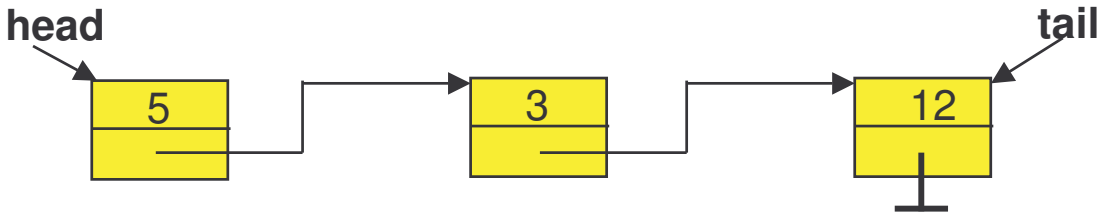
⇒ $\mathbf{2^{(T-1)} \geq n!}$ ⇒

$\mathbf{E = (T-1) >= } \; ld(n!) > ld((n/2)^{n/2})$

$= \mathbf{(n/2)\, ld(\,(n/2)\,)}$

⇒ **E hat Ordnung ≥ O(n ld(n))**

Unsortierte Listen: Einfügen von Elementen → **Am Listenende**



Reihenfolge der Knoten entspricht nicht Sortierreihenfolge ihrer Datenfelder

Einfügen am Ende

Annahme: Liste am Anfang leer ⇒ **head + tail** mit **null** initialisiert

1. Neuen Knoten **p** erzeugen und Wert darin speichern
2. Mittels tail.next auf den neuen Knoten verweisen:

   tail.next = p ;

   Dadurch ist dieser nun ans Ende der Liste gehängt
3. Variable tail auf den neuen Knoten setzen,

   da dieser nun das neue Ende der Liste darstellt:

   tail = p ;

Fälle:

a) Liste noch **leer** (head == null) :

neuer Knoten p ist einziger Knoten ⇒ head = tail = p

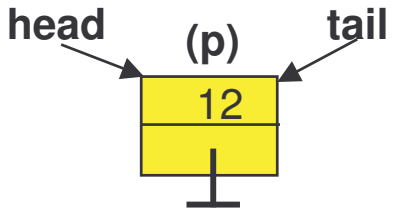
b) Liste **nicht** mehr **leer** (head != null) :

neuer Knoten p ans Ende der Liste ⇒ tail.next = p

```
class List{
  // kein Zugriff auf Pointer! Private!
  private Node  head =  null ;
  private Node  tail = nul l;

   public void add( int x )  {
          Node p = new Node( x ) ;
          if ( head == null )
                { head = p ; }
          else
                { tail.next = p ; }
          tail = p ;
    }
}
```

Initialzustand

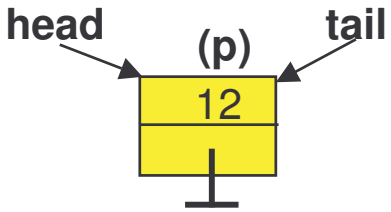
add(12);



Unsortierte Listen: Einfügen von Elementen → **Am Listenende**

Initialzustand

add(12);



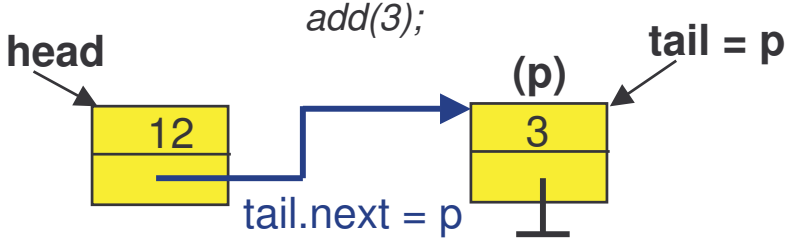
Zuerst:

Alten tail verwenden, um auf neuen Knoten zu verweisen:
tail.next = p ;

Dann:

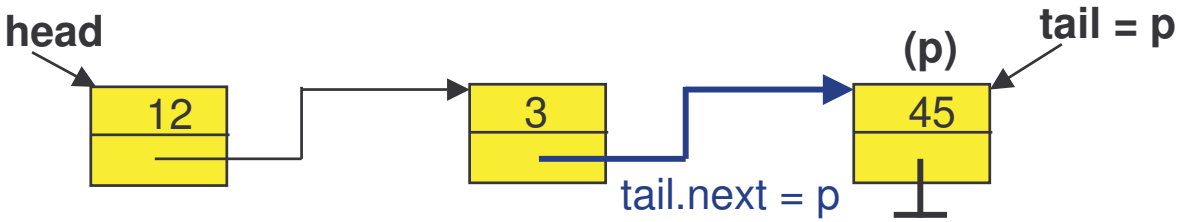
Neuen tail erzeugen, indem tail nun auf neuen Knoten p zeigt:
tail = p ;

add(3);




```
class List{

    private Node  head =  null ;
    private Node  tail = null ;

     public void add( int x )  {
        Node p = new Node( x ) ;
        if ( head == null )
            {  head = p ;  }
        else
            {  tail.next = p ;  }
         tail = p ;
     }
}
```

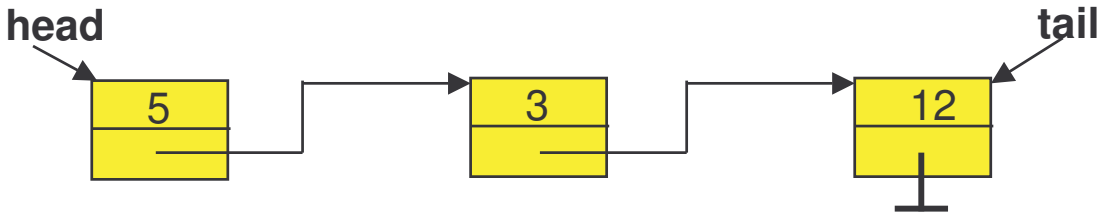
add(45);



Objektvariable **p** ist **lokale** Variable der Methode append :

"Lebt" nur während Ausführung der Methode - im Gegensatz zu den globalen Variablen head und tail !

Unsortierte Listen: Einfügen von Elementen → **Am Listenanfang**



Einfügen am Anfang

Vorteil: man braucht sich **nicht** mit Listenende beschäftigen

Zeiger tail hierzu nicht erforderlich!

1. Neuen Knoten **p** erzeugen und Wert darin speichern
2. Der neue Knoten hat als Nachfolger den bisherigen Listenanfang (head)

   p.next = head ;
3. Der neue Kopf der Liste ist nun der neue Knoten:

   Knoten **p wird zum neuen Listenanfang**

   head = p ;

Vorteil: Keine Unterscheidungen hinsichtlich Listenlänge (leer / nicht leer) erforderlich

Bei ungeordneten Listen ist es **egal**, wo man neues Element einfügt !

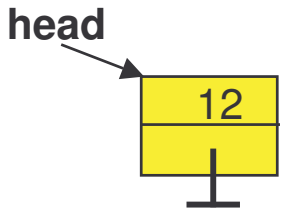
```
class List{

    private Node  head =  null ;
    // private Node  tail = null ;

     public void addFirst( int x )  {
        Node p = new Node(x) ;
        p.next = head ;
        head = p ;
      }
}
```

Weiterer Vorteil gegenüber Arrays:

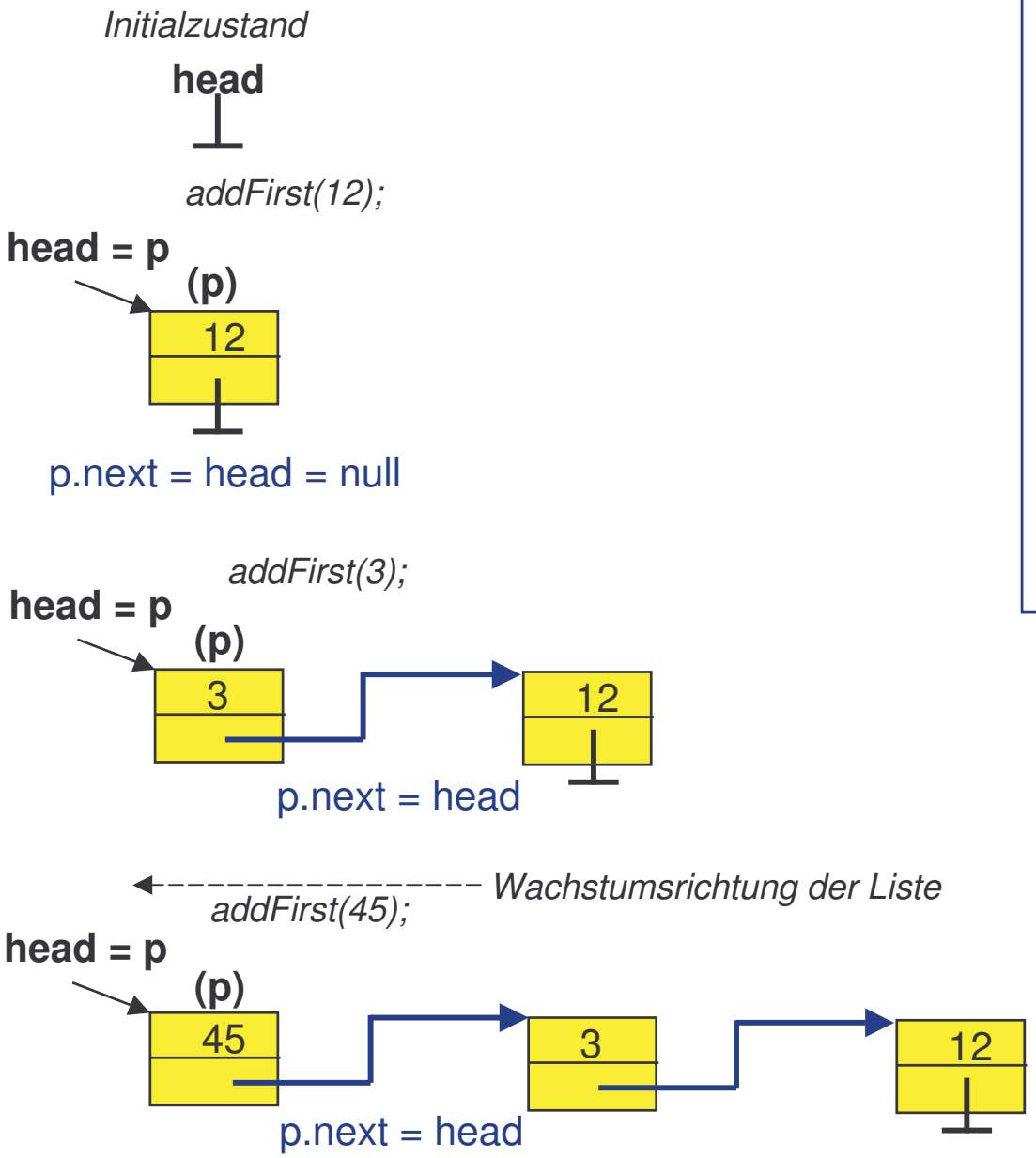
Einfügen neuer Elemente am Anfang erfordert kein Verschieben von Elementen !

Initialzustand

addFirst(12);



Unsortierte Listen: Einfügen von Elementen → **Am Listenanfang**




```
class List{

    private Node  head =  null ;
    // private Node  tail = null ;

     public void addFirst( int x )  {
        Node p = new Node(x) ;
        p.next = head ;
        head = p ;
      }
}
```

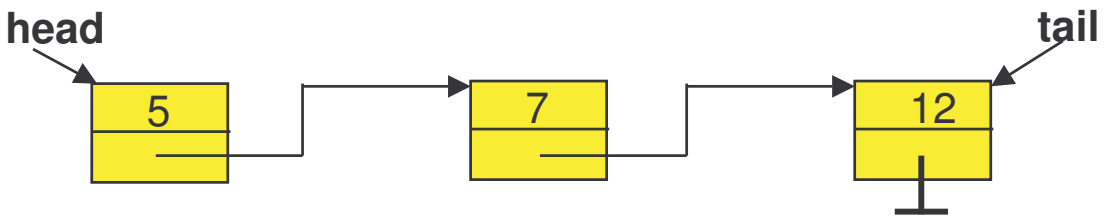
Zuerst:

Next-Zeiger des **neuen** Knotens **p** auf **alten head** zeigen lassen:
p.next = head ;

Dann:

Neuen head erzeugen, indem head nun auf den **neuen** Knoten **p** zeigt:
head = p ;

Sortierte Listen



Reihenfolge der Knoten **entspricht** Sortierreihenfolge ihrer Datenfelder

Suchen

Vorteil der sortierten Liste:

Sortierte Reihenfolge erlaubt sortierte Ausgabe der Werte ohne extra Sortieroperation

Schnelleres Aufsuchen und Finden von Elementen: Liste muß nicht "dumm" bis zum Ende durchlaufen werden, um Element zu finden

Abbrechen der Suche, wenn man auf Elemente stößt, die größer sind als das gesuchte Element

Aber : Für wirklich effizientes Suchen in großen Datenbeständen gibt es geeignetere Datenstrukturen als Listen → binäre Suchbäume

Lineare Liste zum Durchsuchen nicht gut geeignet, da kein wahlfreier Zugriff ⇒

Kein Vorgehen analog Binary Search bei Arrays möglich!

```
class SortedList {

    private Node  head ;
    private Node  tail ;

    public boolean contains( int x )  {
        Node p = head ;

        while ( p! = null  &&  p.data <  x )
            {    p = p.next ;    }

        if ( p != null &&  p.data == x ) return true;
        else      return false ;
    }
}
```

Lineares Suchen in verketteter Liste


```
class List  {

      private Node  head ;
      private Node  tail ;

      public boolaen contains( int x )  {
           Node p = head ;

           while (p != null  &&  p.data !=  x )
              {     p = p.next ;      }

           if( p = null )  return false ;
           return true;
         }
}
```

```
class SortedList  {

      private Node  head ;
      private Node  tail ;

      public boolean contains( int x )  {
           Node p = head ;

           while ( p! = null  &&  p.data <  x )
                {    p = p.next ;    }

            if ( p != null &&  p.data == x ) return true;
            else      return false ;
         }
}
```

Zeitbedarf: **hoch**, da ungeordnet !

neg. Fall → n Vergleiche

pos. Fall → ∅ n /2 Vergleiche

⇒ **Komplexität bleibt O(n)**

(wächst linear mit Zahl der Elemente)

Zeitbedarf: **hoch**, da linear !

neg. Fall → ∅ n / 2 Vergleiche

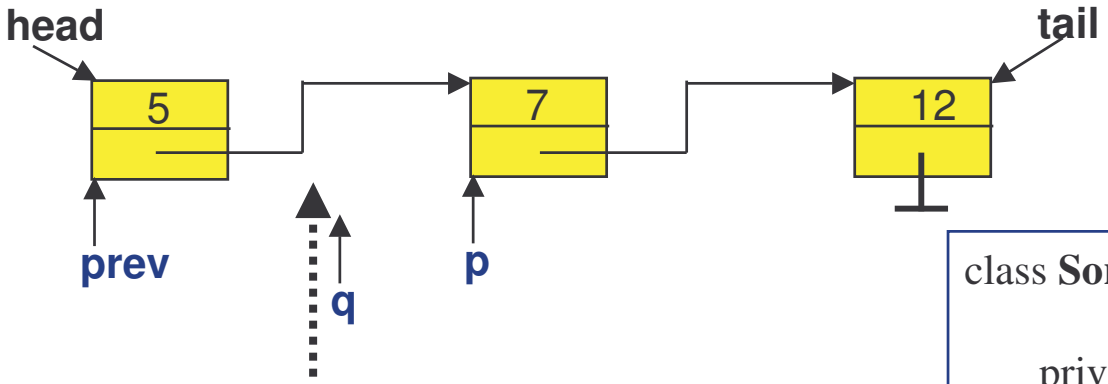
(kann **früher** abgebrochen werden)

pos. Fall → ∅ n /2 Vergleiche

⇒ **Komplexität bleibt O(n)**

(wächst linear mit Zahl der Elemente)

Sortierte Listen: Einfügen von Elementen



Einfügen erfordert nur zwei Zeiger-operationen

Kein Verschieben von Listen-elementen nötig !!

Einfügen in sortierte Liste:

Neues Element muß an der **richtigen Position** eingefügt werden (nicht einfach am Anfang oder Ende wie bei unsortierten Listen)

⇒ **Vor Einfügen die Einfügeposition suchen!**

1. Zeiger **p** auf den ersten Knoten setzen, dessen Wert größer ist als einzufügendes Element + Zeiger **prev** auf den Vorgänger von p setzen
2. Eingefügt werden muß dann **zwischen p und prev:** Neuen Knoten **q** erzeugen + **einhängen**:

   prev.next = q ; und **q.next = p ;**

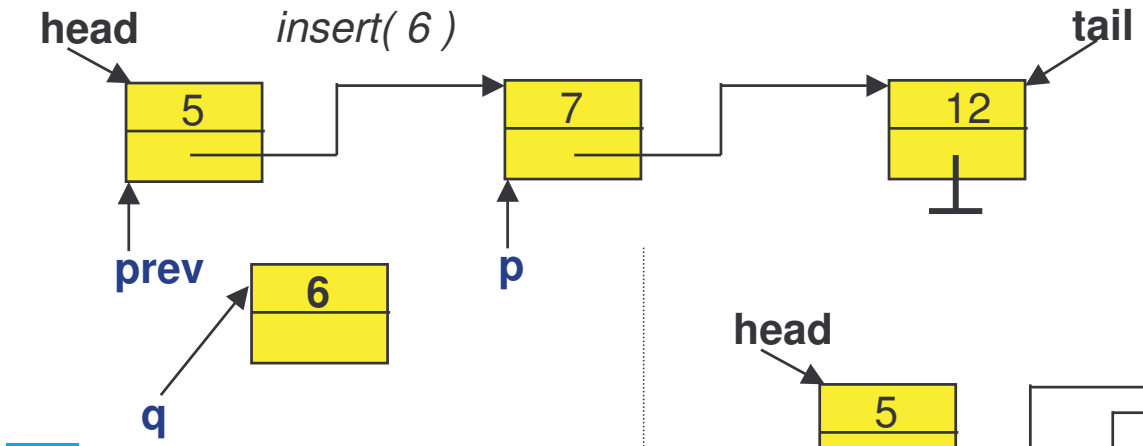
Sonderfälle :

a) Wenn p == null dann wird **q** am Ende eigehängt

b) Wenn p == head dann wird **q** zum neuen Kopf

```
class SortedList {

    private Node head ;
    private Node tail ;

    public void insert( int x ) {
        Node p = head ;
        Node prev = null ;
        Node q = new Node( x ) ;

        while ( p!= null && p.data < x ) {
            prev = p ;   p = p.next ;
        }
        q.next = p ;

        if ( p == head )   head = q ;
        else               prev.next = q ;
        if( p == null )    tail = q ;
    }
}
```

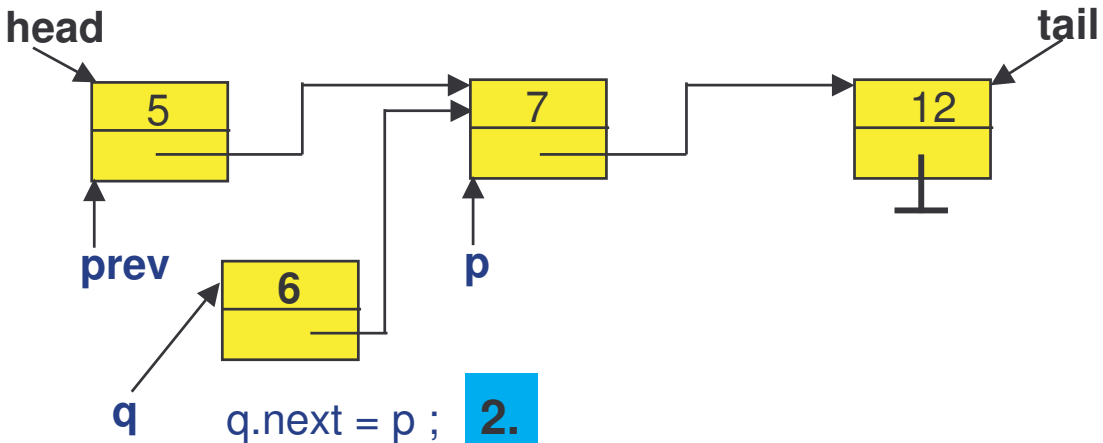
Sortierte Listen: Einfügen von Elementen



Lokale Variablen: q, p, prev

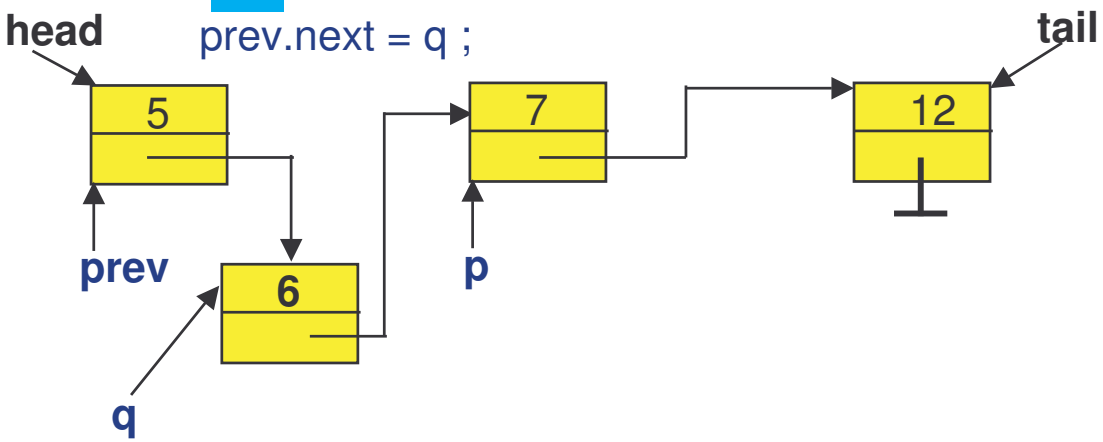
Globale Variablen: head, tail

1. Node q = new Node(6) ;

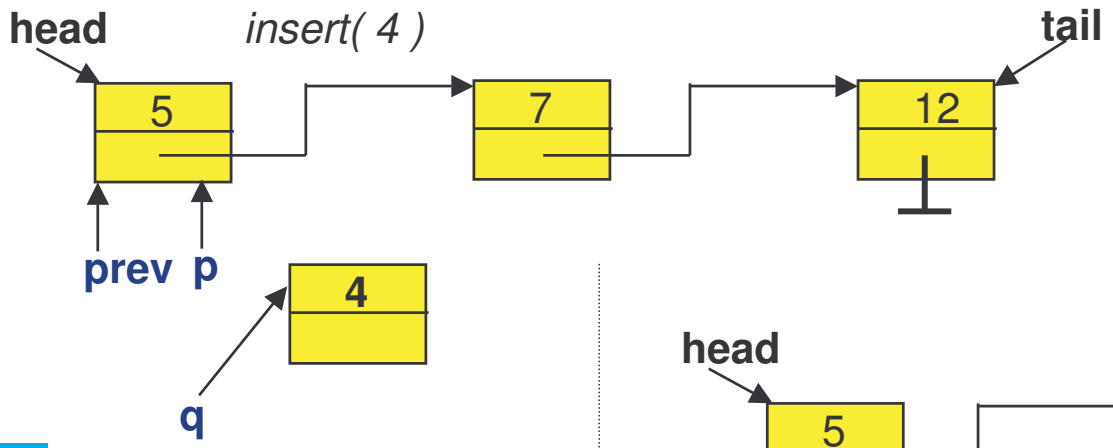


q.next = p ; **2.**

3. prev.next = q ;



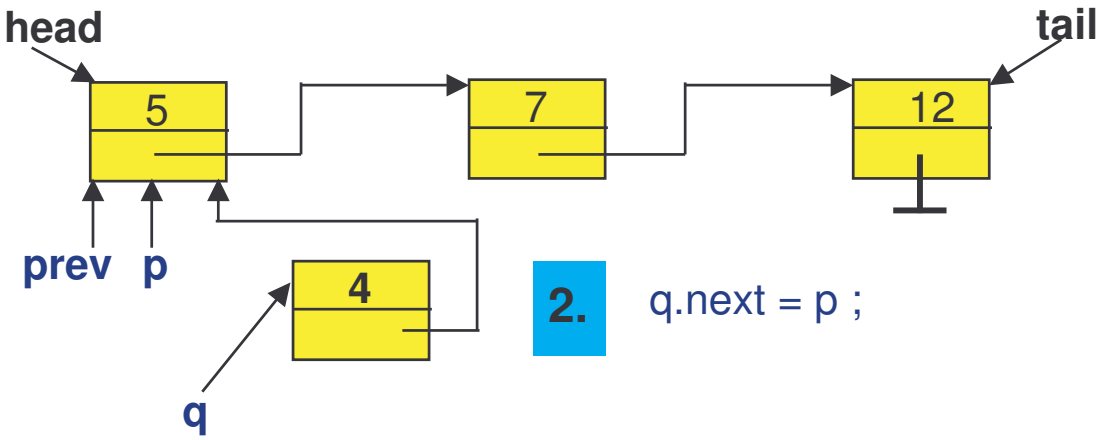
Sonderfall Einfügen am Anfang: p == head



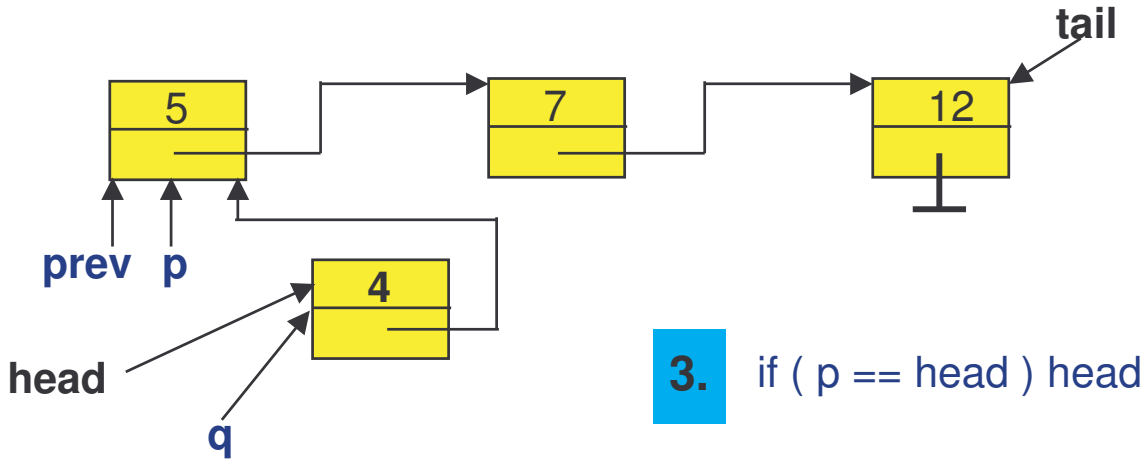
Lokale Variablen: q, p, prev

Globale Variablen: head tail

1. Node q = new Node(4) ;

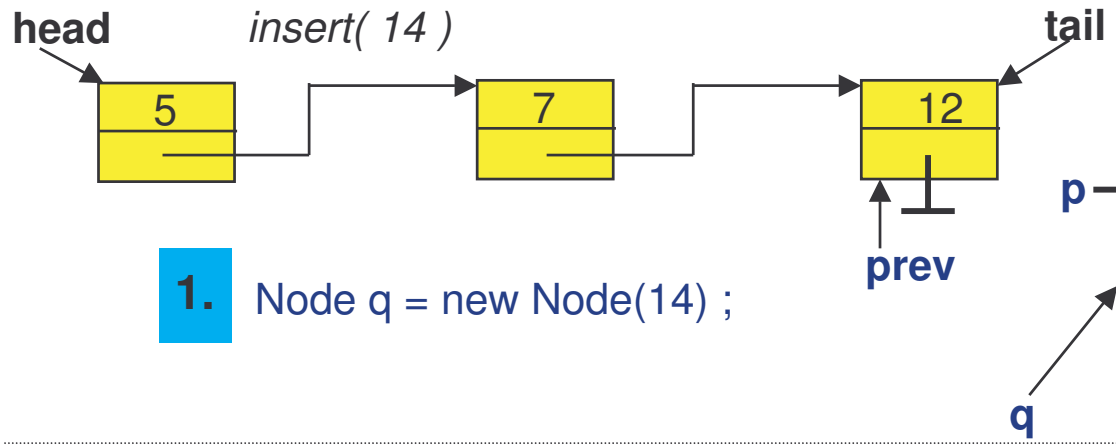


2. q.next = p ;



3. if (p == head) head = q ;

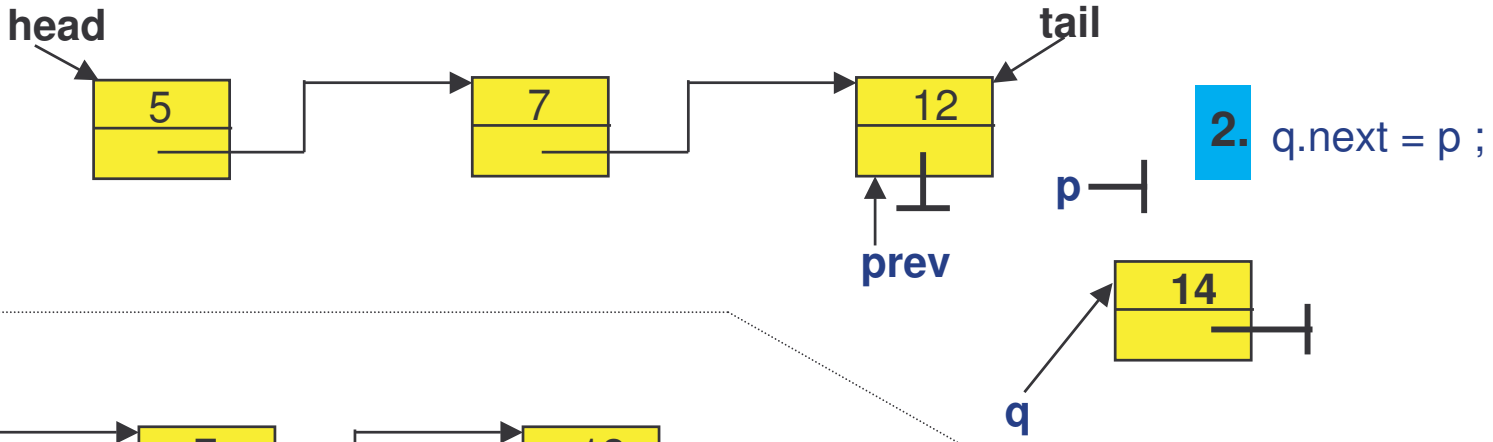
Sonderfall Einfügen am Ende: p == null



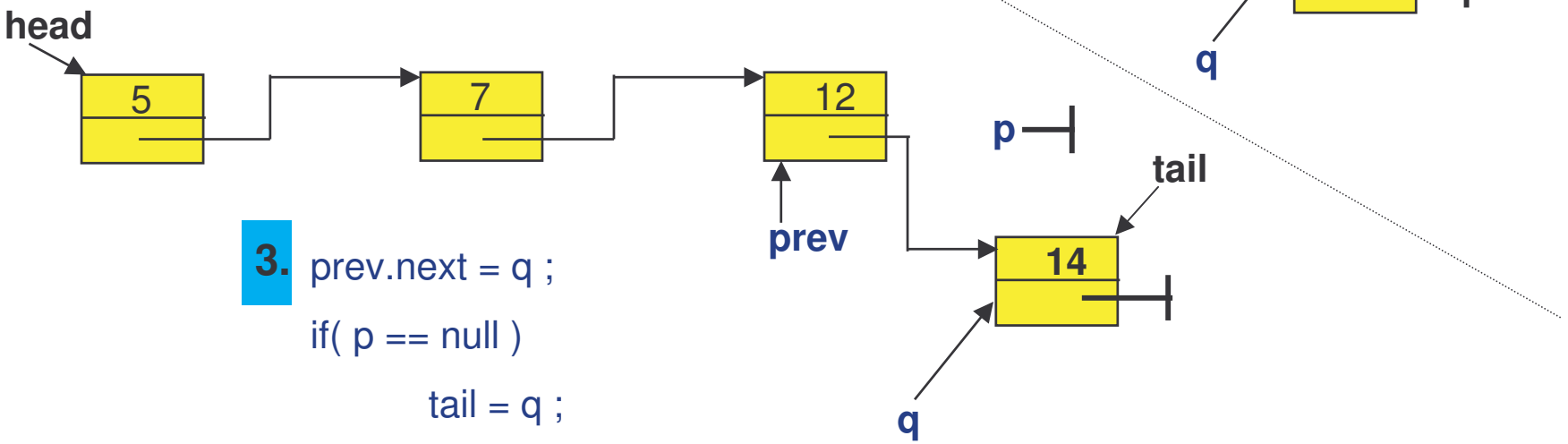
Lokale Variablen: q, p, prev

Globale Variablen: head tail

1. Node q = new Node(14) ;



2. q.next = p ;



3. prev.next = q ;

if(p == null)

tail = q ;

Sortierte Listen: Einfügen von Elementen

Sonderfälle:

1. **p ist der erste Knoten der Liste : p == head**

   ⇒ Es gibt keinen Vorgänger prev

   ⇒ q wird neuer Listenkopf: **head = q ;**

2. **Leere Liste : head = null und p == null**

   ⇒ q wird neuer Listenkopf : **head = q ;**

   ⇒ q hat noch keinen Nachfolger : q.next = null

3. **q muß ans Ende der Liste :**

   Schleife läuft bis ans Ende der Liste

   ⇒ p hat dort den Wert null

   prev zeigt auf den letzten Knoten

   ⇒ q kommt ans Ende, wird neuer tail :

   prev.next = q ; q.next = p ; // = null

   tail = q;

Effektivität der Suche nach Elementen:

In unsortierter Liste muß im schlimmsten Fall die gesamte Liste durchlaufen werden (wenn Element nicht vorhanden). In sortierter Liste im Mittel die halbe Liste, bis die Schleife abbricht .....

```
class SortedList {

    private Node head = null ;

    public void insert( int x ) {
        Node p = head ;
        Node prev = null ;
        Node q = new Node( x ) ;

        while ( p != null && p.data < x ) {
            prev = p ;  p = p.next ;
        }
        q.next = p ;

        if ( p == head)     head = q ;
        else                prev.next = q ;
        if( p == null )     tail = q ;
    }
}
```

Knoten: **Speichern von Objekten**


```
// Datenobjektklasse
class Mitarbeiter  {
   private String name ;
   private int gebJahr ;

   public String getName( )    {  return name ;  }
   public int getGebJahr( )    {  return gebJahr ;  }

   public Mitarbeiter( String n, int g )  {    name = n ;   gebJahr = g ;  }
}

// Knotenklasse = Behälter für Mitarbeiter-Objekte
class Node {
     public  Mitarbeiter  data ;
     public  Node  next ;

     public Node( Mitarbeiter m ) {
          data =  m ;
          next =  null ;
      }
      // Vergleichsoperationen:
      public boolean isOlder(  Mitarbeiter m1, Mitarbeiter m2  )  {
              return ( m1.getGebJahr( ) < m2.getGebJahr( )  ) ;
      }
      public boolean hasSameAge(  Mitarbeiter m1, Mitarbeiter m2  )  {
             return ( m1.getGebJahr( ) == m2.getGebJahr( )  ) ;
      }
}
```

In Knoten **Node** können als Datenelemente **Objekte** (zB Typ Mitarbeiter) gespeichert werden

aber :

Für Verwendung in **sortierten** Listen müssen Datenelemente nach einem **Größenkriterium** verglichen werden:

a) **Kriterium** muß definiert werden

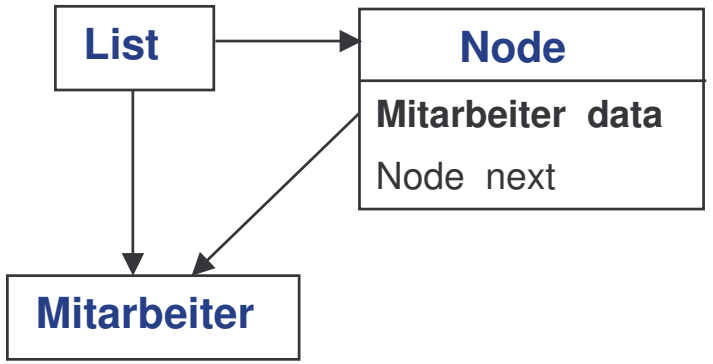
b) **Vergleichsoperationen** müssen speziell für jeweilige Klasse der Objekte implementiert werden

hier :

Sortierte Anordnung nach gebJahr mittels Methode isOlder().
Suche nach Elementen mit gleichem gebJahr mittels Methode hasSameAge().

Sortierte Liste: Speichern von Objekten


```
class SortedList  {

      private Node  head =  null ;

      public void insert(  Mitarbeiter m  )  {
           Node p = head ;
           Node prev = null ;
           Node q = new Node( m ) ;

           while ( p != null  &&  p.isOlder( m,  p.data )   )  {
                    prev = p ;    p = p.next ;
            }
            q.next = p ;

            if ( p == head )      head = q ;
            else                  prev.next = q ;
        }

       public boolean contains(  Mitarbeiter m  )  {
           Node  p = head ;

           while (  p != null  &&  p.isOlder(  m ,  p.data )  )
                 p = p.next ;

            if (  p != null  &&  p.hasSameAge( m , p.data )  )
                     return  true ;
            else   return  false ;
        }
}
```



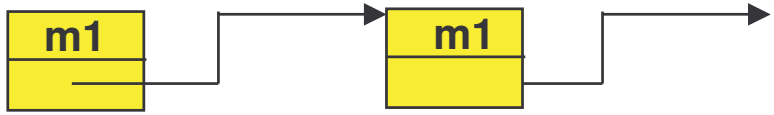
Coding praktisch unverändert .....

Aber :

Sortieren impliziert **Vergleichs-operationen** auf gespeicherten Objekten: Diese müssen definiert und implementiert werden

→ gleich / kleiner / größer / ....

Zum **Vergleich der Datenobjekte** vom Typ Mitarbeiter muß dafür **definierte Vergleichsoperation** der Knotenklasse aufgerufen werden.



Anwendung verkettete Liste: **Stack als verkettete Liste**

Nachteil der Implementierung als Array : Fest vorgegebene Größe ⇒ kann überlaufen !

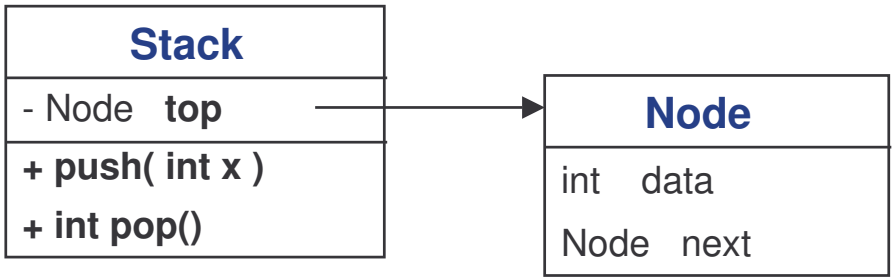
Vorteil der Impl als verkettete Liste : Beliebig viele Elemente hinzufügbar (bis RAM erschöpft)

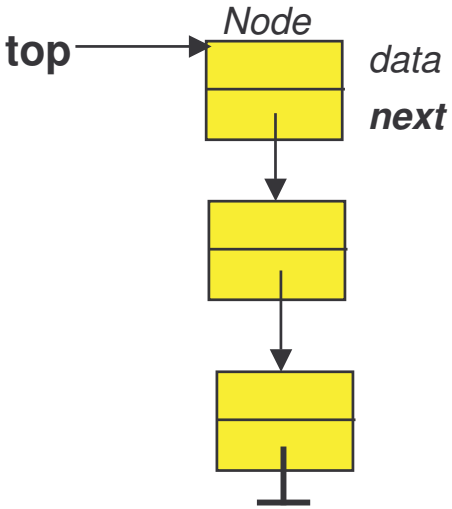
Darstellung mittels verketteter Liste :

- Zeiger **top** zeigt auf den obersten Knoten
- next-Zeiger liefern darunterliegende Elemente
- Operation **push:** Einfügen eines neuen Knotens am **Anfang** der Liste

  Operation **pop:** Entfernen eines Knotens vom **Anfang** der Liste

Gespeicherte Werte könnten jeden möglichen Datentyp haben - auch Objekte .....





Stack als verkettete Liste - Implementierung

Einfügen push() :

1. Neuen Knoten **p** mit einzufügendem Wert erzeugen
2. Nachfolger des neuen Knotens ist bisheriger Anfang top :

   p.next = top ;
3. Neuer Anfang top ist nun der neue Knoten p :

   top = p ;

Zugriff nur über die öffentliche Schnittstelle! Kein direkter Zugriff auf top-Pointer!

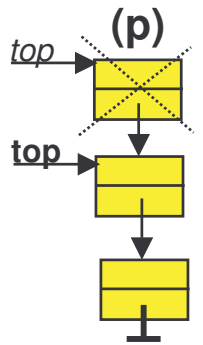
Entfernen pop() :

1. Wenn Stack **leer** ist (top == null) Ausnahme auslösen
2. Wenn Stack nicht leer ist, muß top-Element entfernt werden :
   - 2.1. Lokalen Hilfszeiger **p** auf top setzen: Node p = top ;
   - 2.1. Nachfolger des bisherigen top-Elements wird top :

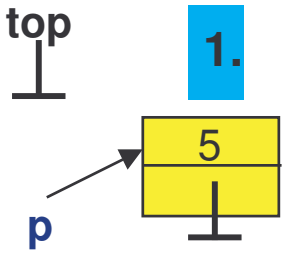
     top = top.next ;
   - 2.2. Wert des bisherigen top-Elements wird zurückgegeben mittels Hilfszeigers p: **return p.data ;**
3. Das **alte** top-Objekt wird nun nicht mehr referenziert

   ⇒ wird von Garbage Collection nach Methodenende **entsorgt** (nachdem lokaler Zeiger p freigegeben ist)

```
class Stack {

    private Node top = null ;

    public void push( int x ) {
        Node p = new Node( x ) ;
        p.next = top ;
        top = p ;
    }

    public int pop( ) {
        if( top == null )
            throw new RuntimeException(
                                "Empty Stack") ;
        else {
            Node p = top ;
            top = top.next ;
            return p.data ;
        }
    }
}
```



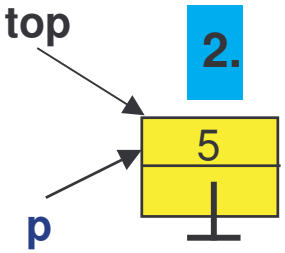
Stack als verkettete Liste: push



push(5)

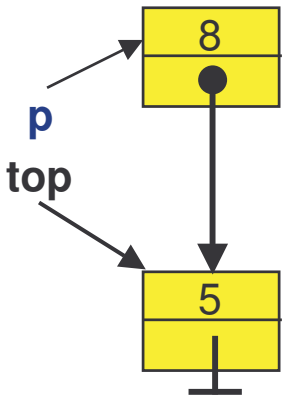
Node p = new Node(5) ;

p.next = top; // = null !



top = p ;

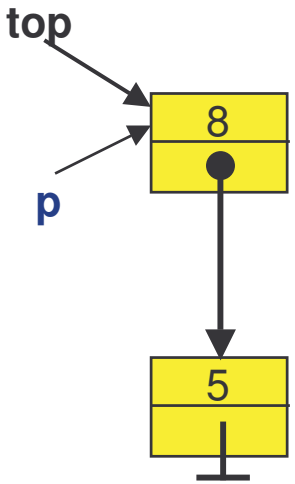
```
class Stack  {

     private Node  top =  null ;

      public void push( int x )  {
          Node p =  new Node( x ) ;
          p.next = top ;
          top = p ;
        }
       .....................................
}
```



push(8)

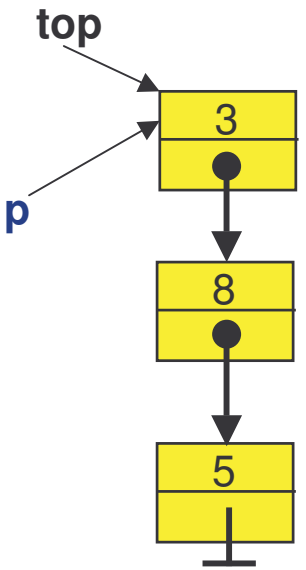
1. Node p = new Node(8) ;

p.next = top;



2. top = p ;

Stack als verkettete Liste: pop



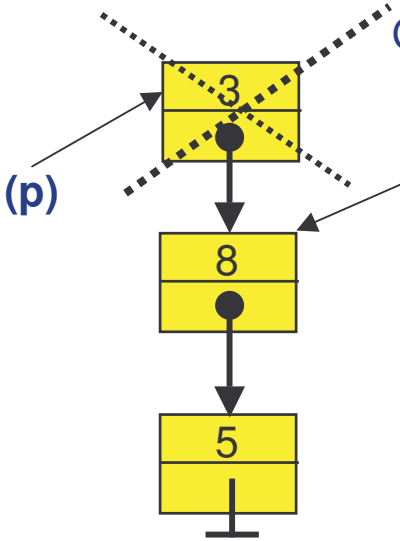
pop( )

1. Node p = top ;

.....

return p.val ;

```
class Stack  {
      private Node  top =  null ;

       public int pop( )  {
            if( top == null )
                throw new RuntimeException(
                                    "Empty Stack") ;
            else {
              Node p =  top ;
              top = top.next ;
              return p.val ;
             }
          }        .....................................
}
```



2. top = top.next ;

Referenz **p** ist **lokale Variable** der Methode **pop** ⇒
Nach Methodenende verschwindet **p** und somit **letzte** Referenz auf Element mit Wert **3** ⇒
Garbage Collection entfernt oberstes Element !

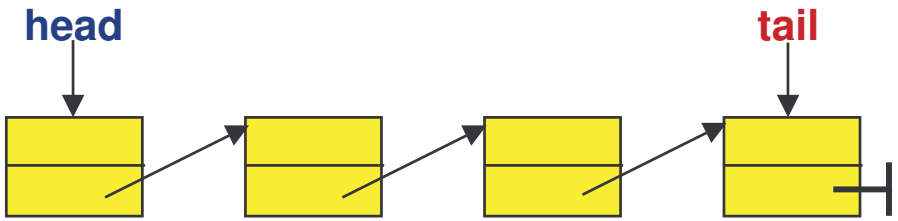
Stack innerhalb JDK: **java.util.Stack** für Elemente vom Typ **Object**.
Methoden empty(), push(), pop(), peek(), search()

Anwendung verketteter Listen: **Queue als verkettete Liste**

Nachteil der Implementierung als Array : Fest vorgegebene Größe ⇒ kann überlaufen

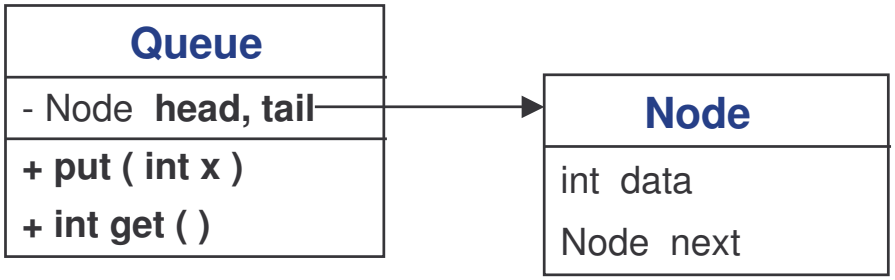
Vorteil der Impl als verkettete Liste : Beliebig viele Elemente hinzufügbar (bis RAM erschöpft)

Darstellung mittels verketteter Liste:



- Zeiger **head** zeigt auf den Anfang der Schlange
- Zeiger **tail** zeigt auf das Ende der Schlange
- Verkettung läuft von Anfang Richtung Ende
- Operation **put :** Anhängen eines neuen Elements ans Ende
- Operation **get :** Entfernen des ältesten Elements vom Anfang

Sonderfall: Schlange ist leer ⇒ es kann nichts entnommen werden:
Ausnahme auslösen



***tail**-Pointer praktisch um bei get()-Operation direkten Zugriff auf das letzte Element zu haben .....*

Queue als verkettete Liste

Einfügen (put) :

1. Neuen einzufügenden Knoten **p** erzeugen
2. ***Leere Schlange*** (head == p) :

   ⇒ neuer Knoten p wird neuer head: **head = p ;**

   Nichtleere Schlange: neuen Knoten p ans Ende hängen :

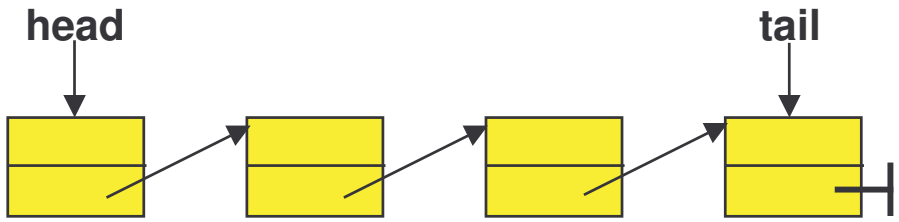
   tail.next = p ;
3. Nach Anhängen ist neuer Knoten p nun das neue Ende der Schlange ⇒ **tail = p ;**

Entfernen (get) :

1. Wenn Schlange leer ist (head == null) Ausnahme
2. Hilfszeiger **p** auf den Kopf setzen (soll entfernt werden)

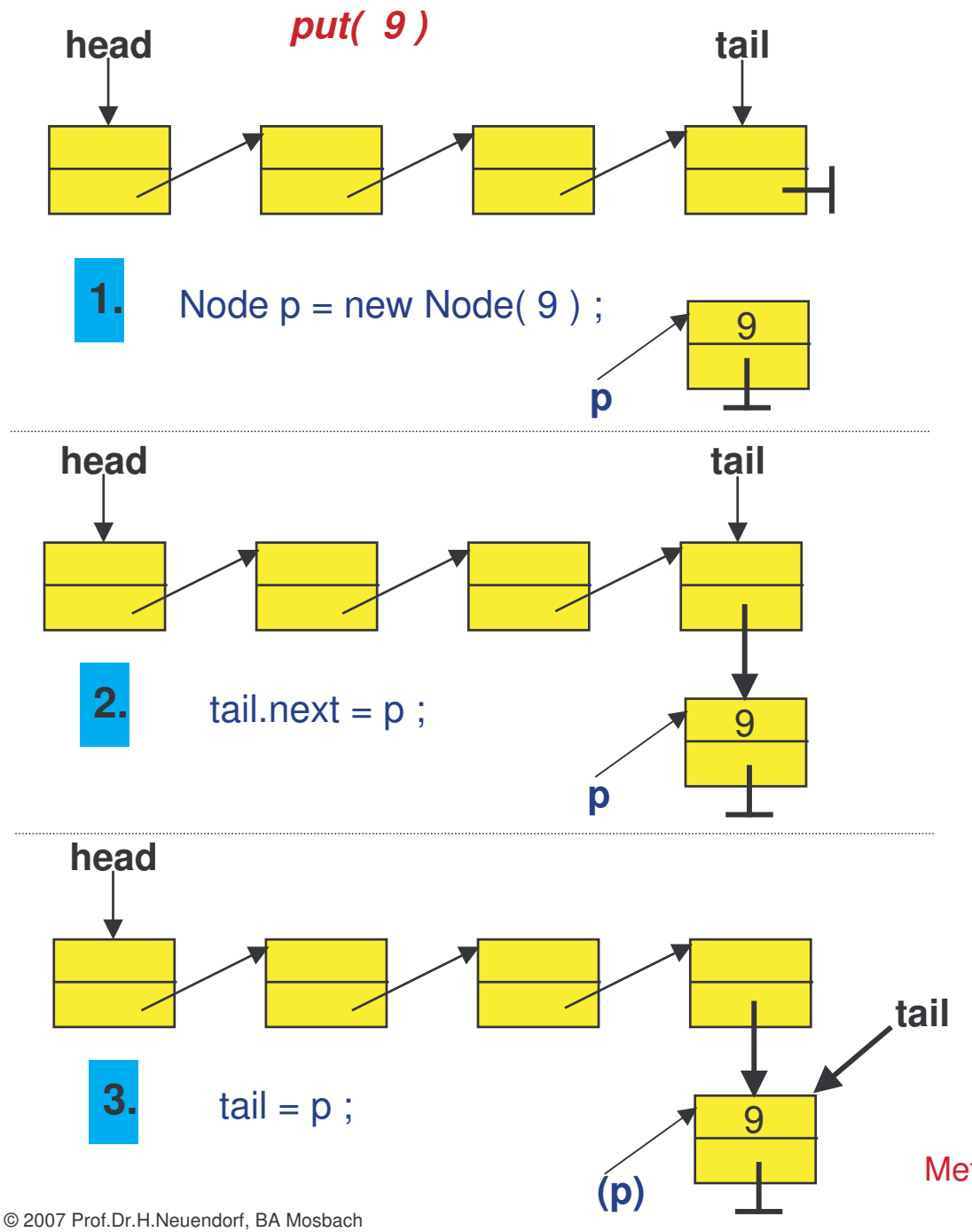
   Node p = head ; (Rückgabe von p.val)
3. Neuer Kopf ist der Nachfolger des bisherigen Kopfes:

   head = head.next ;
4. Wenn letztes Element der Schlange entfernt wird dann auch tail = null ; (head + tail nun beide null)




```
class Queue {
    private Node head = null ;
    private Node tail = null ;

    public void put( int x ) {
        Node p = new Node( x ) ;
        if ( head == null) head = p ;
        else  tail.next = p ;
        tail = p ;
    }

    public int get( ) {
        if( head == null ) throw new
                    RuntimeException(
                        "Empty Queue") ;
        else {
            Node p = head ;
            head = head.next ;
            if ( head == null ) tail = null ;
            return p.data ;
        }
    }
}
```

Queue als verkettete Liste



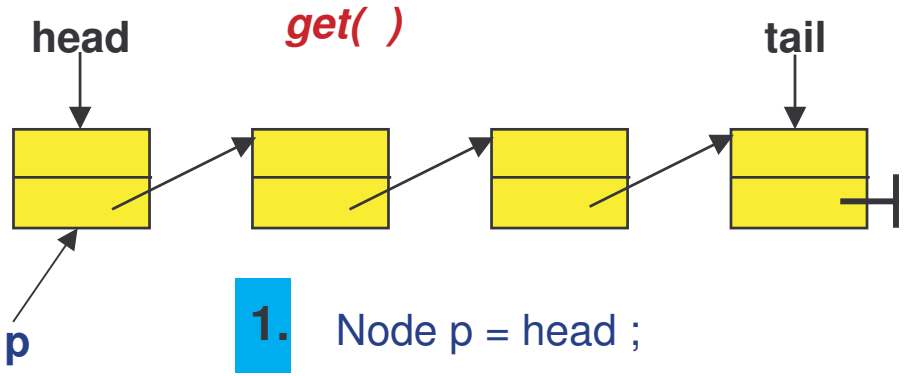

```
class Queue  {

      private Node  head =  null ;
      private Node tail = null ;

       public void put( int x )  {
          Node p =  new Node( x ) ;
          if ( head ==  null)  head = p ;
          else   tail.next = p ;
          tail = p ;
        }
      // ..........................................
}
```

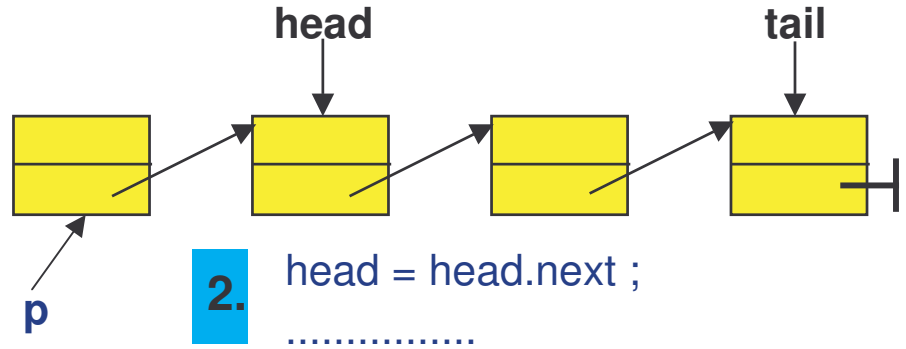
Sonderfall: Leere Queue

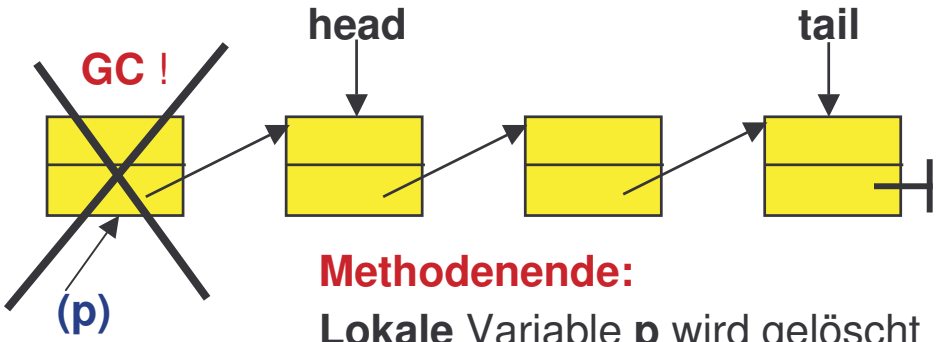
- **head** ist noch null
- head zeigt nach Erzeugen von neuem Knoten **p** auf den neuen Knoten p: **head = p ;**

Methodenende: Lokale Variable **p** wird gelöscht

Queue als verkettete Liste







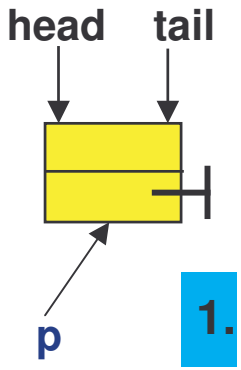
Methodenende:

Lokale Variable **p** wird gelöscht

⇒ altes Anfangs-Objekt wird nicht mehr referenziert ⇒ **G**arbage **C**ollection!

```
class Queue  {

      private Node  head =  null ;
      private Node tail = null ;

      public int get( )  {
            if( head == null ) throw new
                        RuntimeException(
                              "Empty Queue") ;
            else {
              Node p =  head ;
              head =  head.next ;
              if ( head == null ) tail = null ;
              return  p.data ;
             }
          }
       //  .....................................
}
```

Queue als verkettete Liste

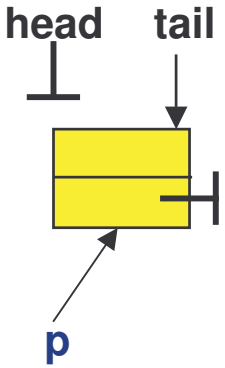


get( )

Sonderfall:

Queue enthält **nur** noch **ein** Element / Knoten

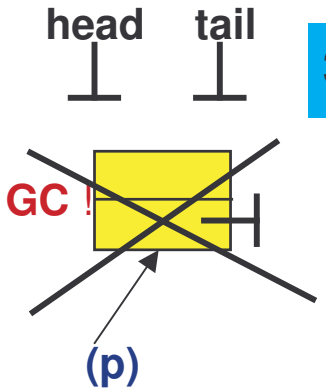
1. Node p = head ;



2. head = head.next ; // = null !

.....

return p.val ;



3. if(head == null) **tail = null ;**

Methodenende:

Lokale Variable **p** wird gelöscht

⇒ Objekt wird nicht mehr referenziert ⇒ **G**arbage **C**ollection !

⇒ Es bleibt eine **leere Queue** zurück - head + tail sind null !

```
class Queue  {

    private Node  head =  null ;
    private Node tail = null ;

    public int get( )  {
        if( head == null ) throw new
                    RuntimeException(
                        "Empty Queue") ;
        else {
          Node p =  head ;
          head =  head.next ;
          if ( head == null ) tail = null ;
          return  p.data ;
        }
      }
    //  .....
}
```

Datenabstraktion

Grundprinzip = Information Hiding ↔ **Ziel = Reduktion von Komplexität:**

Implementierungsdetails sollten soweit wie möglich **"versteckt"** werden

Verwendung von Datenstukturen:

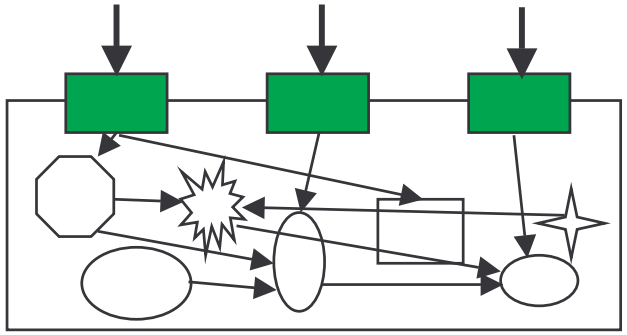
Zugriff auf komplexe Datenstrukturen →

- nur über wenige öffentliche, einfach handhabbare Methoden = einfache Schnittstelle
- ohne tatsächliche interne Datenstrukturen/ Implementierungsdetails offenzulegen !

Vorteil: Interne Implementierungsdetails können jederzeit geändert werden, solange Schnittstelle konstant bleibt ⇒ Verwender / Clients werden nicht invalidiert

⇒ **Datenabstraktion:** ***" Wie"*** der Implementierung wird versteckt

Nur das ***" Was"*** der Datenhaltung wird exponiert in der öffentlichen Schnittstelle .....



Bsp: **Stack-Klasse**

- Feste Schnittstelle zum Gebrauch:

  public int pop( ) , public void push(int x)

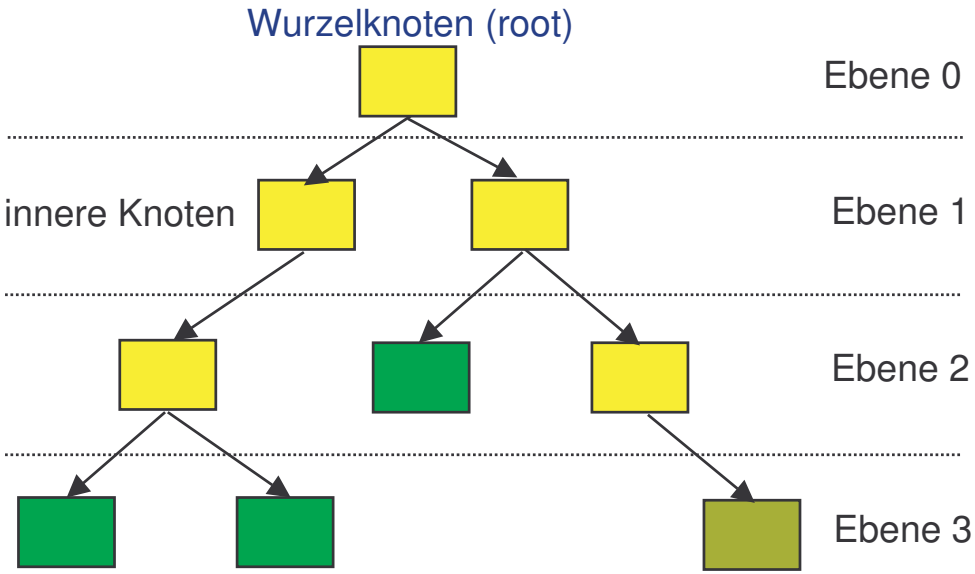
- Implementierung mittels Array oder verketteter Liste

  = Implementierungsdetail, soll von Außen nicht sichtbar sein, kann **ausgetauscht** werden !

Baumstrukturen (Hierarchische Datenstruktur)

Baum = Verzweigte Struktur aus Knoten + Folge von Nachfolgern (≠ Graph)

Binärbaum: Jeder Knoten hat maximal **zwei** Nachfolger - Jeder Knoten hat **einen** Vorgänger



Blätter (Blattknoten) = Elemente ohne Nachfolger

Tiefe T = Zahl der Ebenen = 4

Gewicht = Zahl der Knoten = 9

Zu jedem Knoten führt ausgehend von Wurzel ein **eindeutiger** Weg

Definitionen:

Baum ist "voll", wenn außer der letzten Ebene alle seine Ebenen voll besetzt sind :

Ebene k → max. 2^k - Elemente

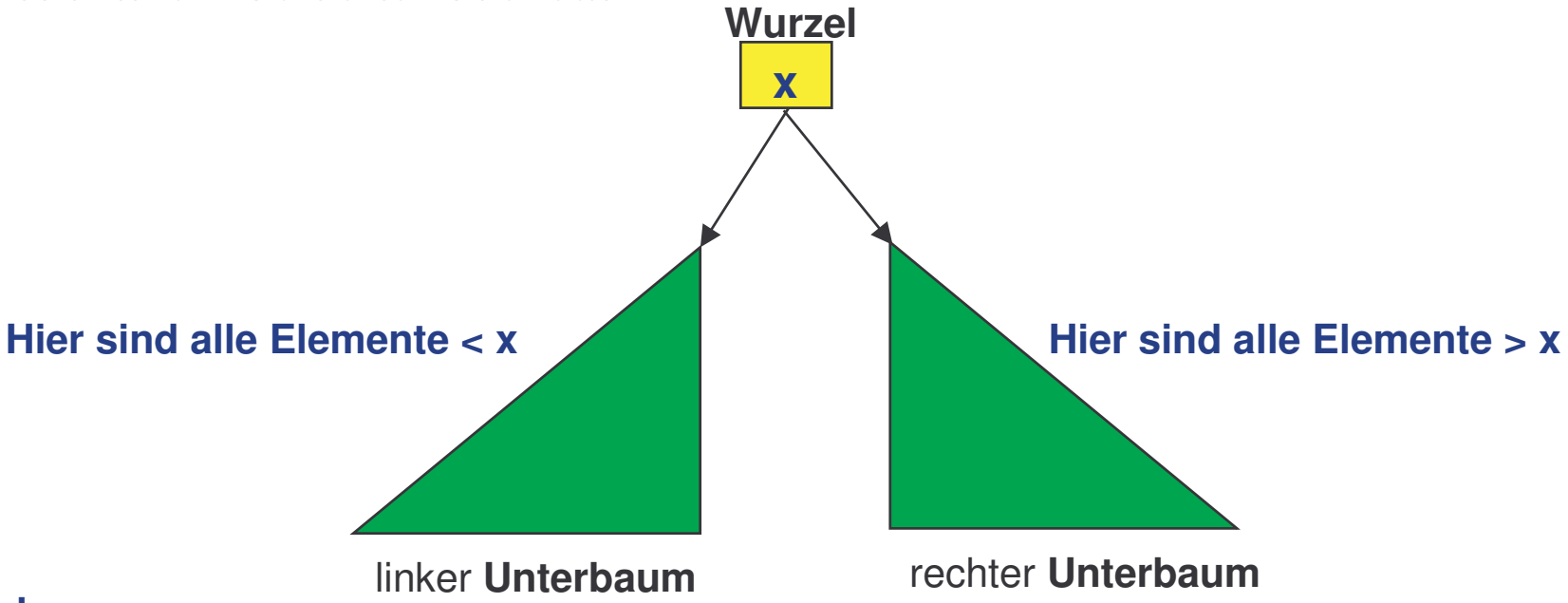
Ein Binärbaum ist **streng sortiert**, wenn für jeden Knoten für dessen beide Unterbäume gilt :

1. Alle Knoten im **linken** Unterbaum haben **kleinere** Schlüssel *und.....*

2. Alle Knoten im **rechten** Unterbaum haben **größere** Schlüssel ....

.... als ihr Vorgängerknoten

Baumstrukturen: Sortierter Suchbaum



Suchvorgang

An jeder **Verzweigung** wird geprüft :

Gesuchtes Element **==** Knoten-Wert $\Rightarrow$ Suche beendet, gefunden !

Gesuchtes Element **>** Knoten-Wert $\Rightarrow$ Suche im **rechten** Unterbaum fortsetzen

Gesuchtes Element **<** Knoten-Wert $\Rightarrow$ Suche im **linken** Unterbaum fortsetzen

Ausgeglichener voller Baum hat Gesamt-Elementzahl: $n = 2^T - 1$

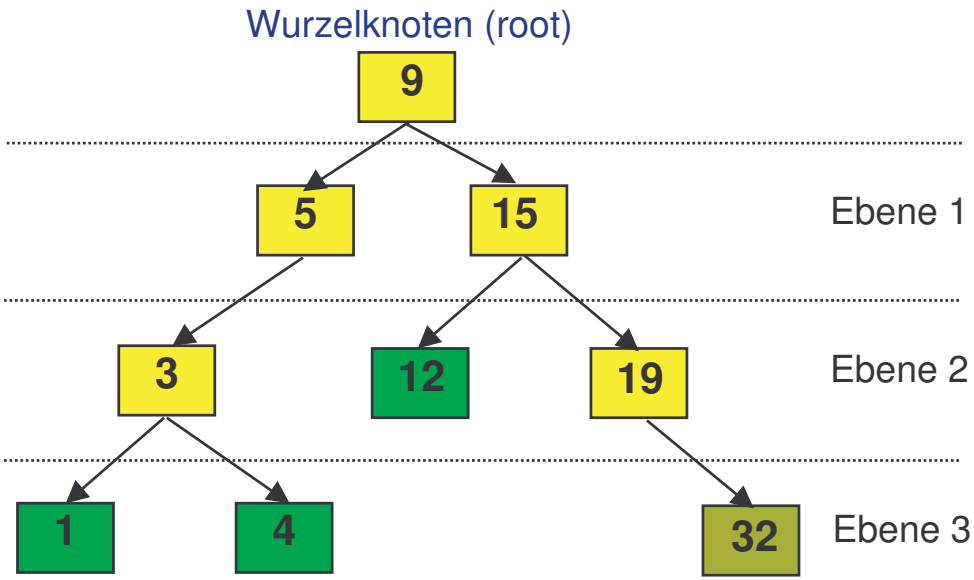
Anzahl der Ebenen T somit: $T = ld(n +1)$ **= Anzahl der erforderlichen Suchschritte**

Suchen in sortierten Binärbäumen

Sortierte Bäume erlauben sehr effektives Suchen !

Grund: Beim Durchlauf **reduziert** sich an jeder Verzeigung die Anzahl der noch zu prüfenden Elemente

⇒ **deutlich weniger zu durchlaufende Einträge als bei linearen Strukturen !!**



Maximale Gesamtknotenzahl $n = 2^T - 1$

⇒ Tiefe eines ausgeglichenen, vollen Baums:

$T = ld(n + 1)$

Vergleich lineare Liste : "Tiefe" = n (unverzeigt)

Zu durchsuchende Elemente im Mittel :

lineare Liste = **n / 2** **Baum = ld(n + 1)**

"Gute" Bäume:

Haben gute = regelmäßige Verteilung der **n** Elemente auf die einzelnen Ebenen

Ebenen sind **möglichst voll** besetzt

⇒ Tiefe möglichst klein: **T = ld(n+1)**

⇒ **Suchvorgang sehr effektiv**

"Schlechte" Bäume:

Haben schlechte = unregelmäßige, unausgeglichene Verteilung der **n** Elemente auf einzelne Ebenen

Ebenen sind schwach besetzt

⇒ Tiefe sehr groß: Extremfall = **n**

Entartung des Baums zur linearen Liste

⇒ Suchen nicht effektiver als bei Listen

Suchen in sortierten Binärbäumen

Aufwandsvergleich im Mittel:

Einträge	**Liste**	**Suchbaum**
7	3.5	3
31	15.5	5
127	63.5	7
1023	511.5	10
16383	**8191.5**	**14**

Suchvorgang = Durchlaufen des Baums von der Wurzel bis zum gesuchten Knoten oder Blatt

Verzweigung: An jedem Knoten wird gesuchter Schlüssel mit Schlüssel / Wert des Knotens verglichen:

Suchwert = Knotenwert ⇒ Gesuchter Wert gefunden!
Suchwert > Knotenwert ⇒ weiter im rechten Teilbaum
Suchwert < Knotenwert ⇒ weiter im linken Teilbaum

Darstellung von Binärbäumen: (ähnlich wie bei Listen)

→ **Baum besteht aus Knoten**

→ **Knoten** besteht aus:

a) einen Inhalt (Wert, Objekt = **data**) +

b) **zwei Nachfolgern:**

→ wiederum vom Typ **Knoten** (**links, rechts**)

⇒ Es handelt sich um eine **rekursive Datenstruktur**

⇒ Jeder Knoten hat nun **zwei Nachfolger**

```
class Knoten  {
   public int data ;
   public Knoten  links, rechts ;

   public Knoten(  int n ) {
         data = n ;
         links = null ;
         rechts = null ;
      }
}
```

Realisierung des Binärbaums durch Klasse Knoten

Maximale Gesamtknotenzahl:

$n = 2^T - 1$