

Kapitel 2: Elementare Datenstrukturen

- 1. Elementare und strukturierte Datentypen**
- 2. Stapel und Warteschlangen**
- 3. Listen**
- 4. Bäume**

**Folien nur für den eigenen Gebrauch, Weiterleitung
an Dritte und Online-Stellen nicht erlaubt.**

2.1 Elementare und Strukturierte Datentypen

■ Abstrakter Datentyp (ADT) / Datenstruktur:

- Ein oder Mehrere Objekt(e) (Beschreibung der Daten) und
- Operationen (Manipulation der Daten)

■ Beispiel: Datum

■ Objekt D: Tag.Monat.Jahr	1.11.2002
■ Objekt T: Tage	17 Tage
■ Operationen:	
◆ Addition: $D \times T \rightarrow D$	1.11.2002 + 5 Tage \rightarrow 6.11.2002
◆ Subtraktion 1: $D \times T \rightarrow D$	1.11.2002 - 5 Tage \rightarrow 27.10.2002
◆ Subtraktion 2: $D \times D \rightarrow T$	1.11.2002 - 27.10.2002 \rightarrow 5 Tage
◆ IstFeiertag: $D \rightarrow \{\text{wahr}, \text{falsch}\}$	IstFeiertag(1.11.2017) \rightarrow falsch (in HH)

■ Entwurf von Datentypen (konstruktive Methode):

- Definition der Objekte (bestehend aus elementaren Datentypen)
- Definition aller Operationen (Operanden, Ergebnis, Spezifikation)

Elementare und strukturierte Datentypen

■ **Elementare Datentypen:** ADTs, die typischerweise (in einer Programmiersprache) zur Verfügung stehen:

- INTEGER: ganze Zahlen
- REAL: reelle Zahlen
- BOOLEAN: Wahrheitswerte {TRUE, FALSE}
- CHAR: Zeichen
- STRING: Zeichenkette

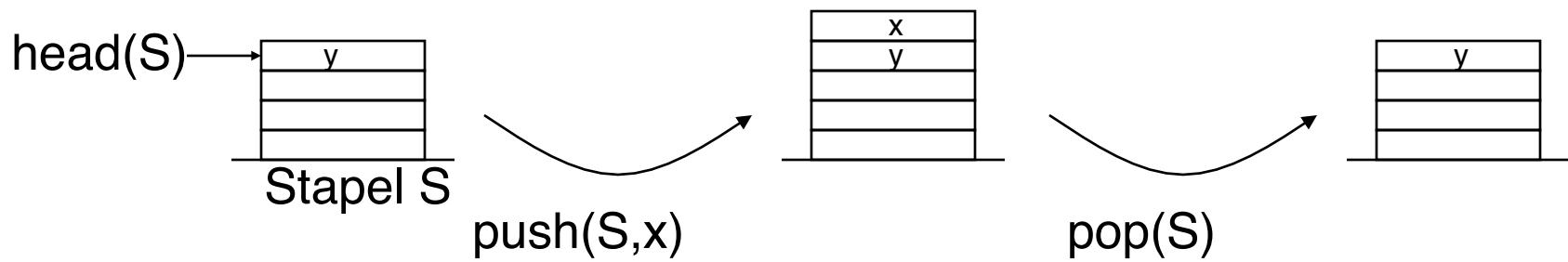
■ **Strukturierter Datentyp:** aus elementaren (oder strukturierten!) Datentypen zusammengesetzte Datentypen, wichtige Strukturierungsmethoden:

- ARRAY: über natürliche Zahlen indizierte Menge
- RECORD/STRUCT: Gruppierung ggf. verschiedener Datentypen
- ENUM: konstante Wertemenge
- UNION: Vereinigung verschiedener Datentypen

■ **Referenzen:** Verweise auf andere Daten (Zeiger, Adressen)

2.2 Stapel

- **LIFO-Prinzip (last-in first-out):** Speicherung mit Zugriffsmöglichkeit nur auf dem zuletzt gespeicherten Objekt
- Stapel (Stack): linearer Speicher mit folgenden Operationen
 - **head(S):** Wert des ‚obersten‘ Elements
 - **push(S,x):** Lege x oben auf den Stapel
 - **pop(S):** Entferne oberstes Element vom Stapel



- Implementierung: Sequenzielle oder verkettete Speicherung

Stapel: Sequentielle Speicherung

■ Datenstruktur:

stack S : Array mit Elementen 1,...,MAXE
S.top : Index des ‚obersten Elements‘

■ EMPTY-STACK (S)

```
if( S.top == 0 ) return TRUE
else             return FALSE
```

■ PUSH(S, x)

```
if( S.top == MAXE ) error „Überlauf!“
else S.top = S.top+1
      S[S.top] = x
```

■ POP(S)

```
if( EMPTY-STACK(S) ) error „Unterlauf!“
else S.top = S.top-1
      return S[S.top+1]
```

■ HEAD(S)

```
if( EMPTY-STACK(S) ) error „Unterlauf!“
else return S[S.top]
```

Stack.java: Beispielimplementierung 1/2

```
import java.util.ArrayList;
public class Stack {
    private int count = 0;
    private ArrayList<String> stack = new ArrayList<>(100);

    /**
     * prüft ob der Stack leer ist
     * @return true, falls Stack leer; false, sonst
     */

    public boolean emptystack() {
        return count == 0;
    }

    /**
     * Liefert das oberste Element des Stacks,
     * ohne den Stack dabei zu verändern
     * @return das oberste Element des Stacks
     */

    public String head() {
        if (count > 0) {
            return stack.get(count - 1);
        } else {
            throw new IndexOutOfBoundsException();
        }
    }
}
```



Stack.java: Beispielimplementierung 2/2

```
/**  
 * Schiebt ein Element auf den Stack  
 * @param s das Element, das auf den Stack geschoben werden soll  
 */  
public void push(String s) {  
    if (count < 100) {  
        stack.add(count, s);  
        count++;  
    } else {  
        throw new IndexOutOfBoundsException();  
    }  
}  
  
/**  
 * Gibt das oberste Element des Stacks zurück und entfernt es dabei vom Stack  
 * @return das oberste Element des Stacks  
 */  
public String pop() {  
    if (count > 0) {  
        count--;  
        return stack.get(count);  
    } else {  
        throw new IndexOutOfBoundsException();  
    }  
}  
}
```



Stapel: Anwendung

- Problem: Erkennung wohlgeformter Klammerausdrücke
 - finde zu jeder schließenden Klammer die zugehörige öffnende
 - Eingabe Array brackstr[1,...,nofbrack]
 - Ausgabe Array pairno[1,...,nofbrack]
(Index der öffnenden/schließenden Klammer)

Beispiel:

index:	1	2	3	4	5	6	7	8	9	10	11	12
brackstr:	(()	()	(()	()))
pairno:	12	3	2	5	4	11	8	7	10	9	6	1

- Lösung: speichere Index öffnender Klammern in einem Stack

Stapel: Anwendung

```
■ BRACKETS( brackstr, pairno )
// brackstr: array[1,...,nofbrackstr] of char (Eingabe)
// pairno: array[1,...,nofbrackstr] of integer (Ausgabe)

S = INIT();           // initialisiere Stack S
for p = 1 to nofbrackstr
    if( brackstr[p] == '(' ) PUSH(S,p)
    else
        if( EMPTY-STACK(S) ) error „Opening bracket missing!“
        else pairno[p] = HEAD(S)
                pairno[HEAD(S)] = p
                POP(S)
    if( not EMPTY-STACK(S) ) error „Closing bracket missing!“
    else return „Format correct.“
```

Stapel: Anwendung

Beispiel:

index: 1 2 3 4 5 6 7 8 9 10 11 12

brackstr: (() () (() ()))

pairno: 0 0 0 0 0 0 0 0 0 0 0 0

↑
p

2
1
Stack

p=3



Stapel: Anwendung

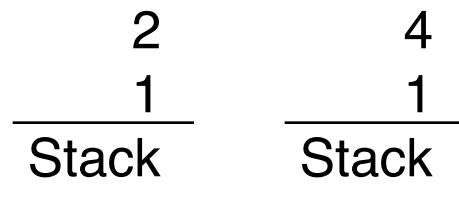
Beispiel:

index: 1 2 3 4 5 6 7 8 9 10 11 12

brackstr: (() () (() ()))

pairno: 0 3 2 0 0 0 0 0 0 0 0 0

↑
p



p=3

p=5



Stapel: Anwendung

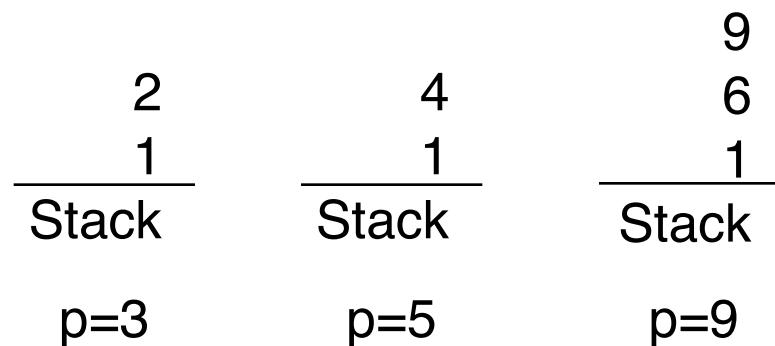
Beispiel:

index: 1 2 3 4 5 6 7 8 9 10 11 12

brackstr: (() () (() ()))

pairno: 0 3 2 5 4 0 8 7 0 0 0 0

↑
p

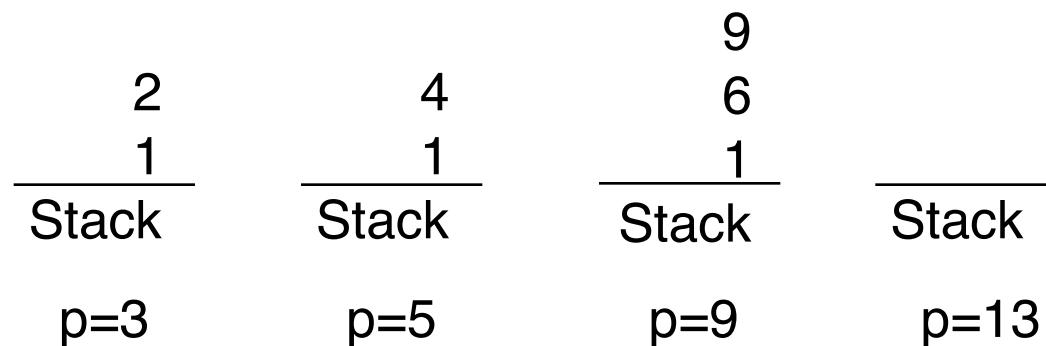


Stapel: Anwendung

Beispiel:

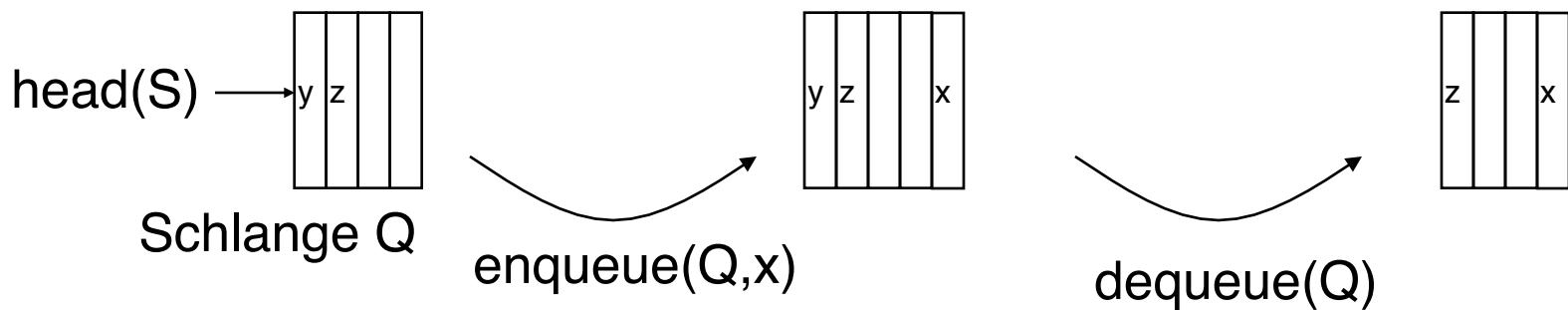
index:	1	2	3	4	5	6	7	8	9	10	11	12
brackstr:	(()	()	(()	()))
pairno:	12	3	2	5	4	11	8	7	10	9	6	1

↑
p



Schlange

- **FIFO-Prinzip** (first-in first-out): Speicherung mit Zugriffsmöglichkeit nur auf dem zuerst gespeicherten Objekt
- Schlange (Queue): linearer Speicher mit folgenden Operationen
 - **head(Q)**: Wert des ‚vordersten‘ Elements
 - **enqueue(Q,x)**: Füge x am Ende der Schlange an
 - **dequeue(Q)**: Entferne vorderstes Element der Schlange



- Implementierung: Sequenzielle oder verkettete Speicherung

Schlange: Sequentielle Speicherung

■ Datenstruktur:

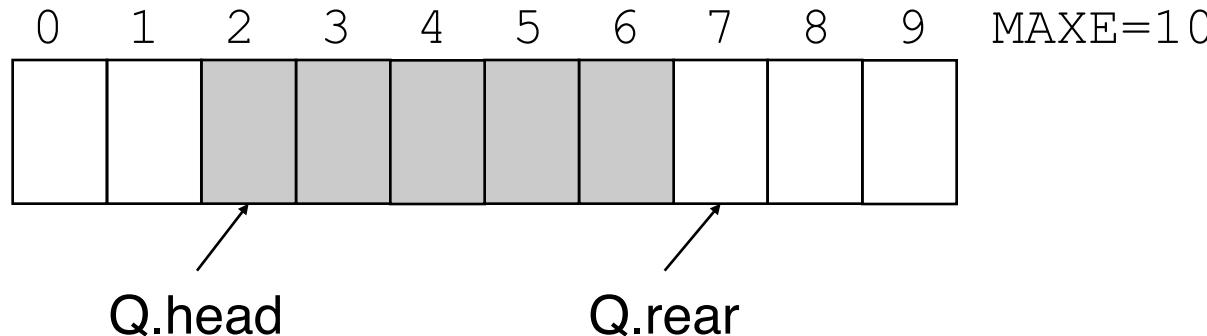
queue Q : Array 0,..,MAXE
Q.head : Position des ersten Elements
Q.rear : Pos. des ersten freien Elements (hinter der Schlange)
// ACHTUNG: Implementierung im Cormen mit Array 1..n

■ INIT (Q)

Q.head = Q.rear = 1

■ EMPTY-QUEUE (Q: queue)

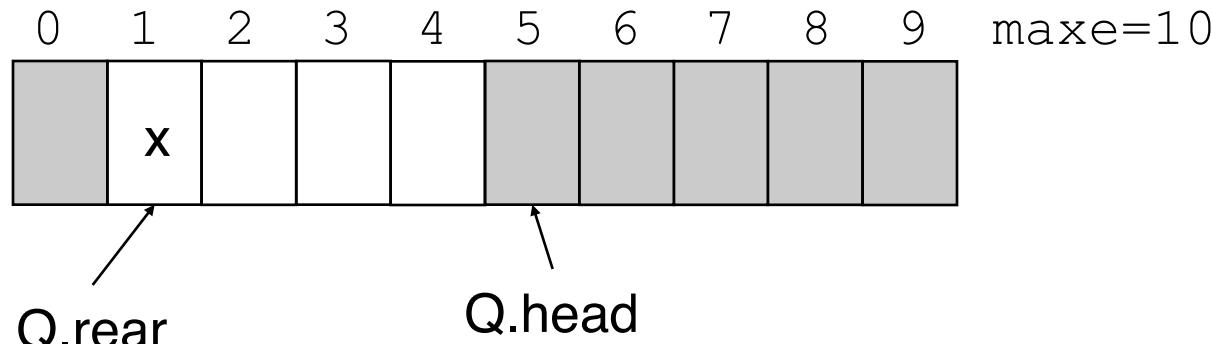
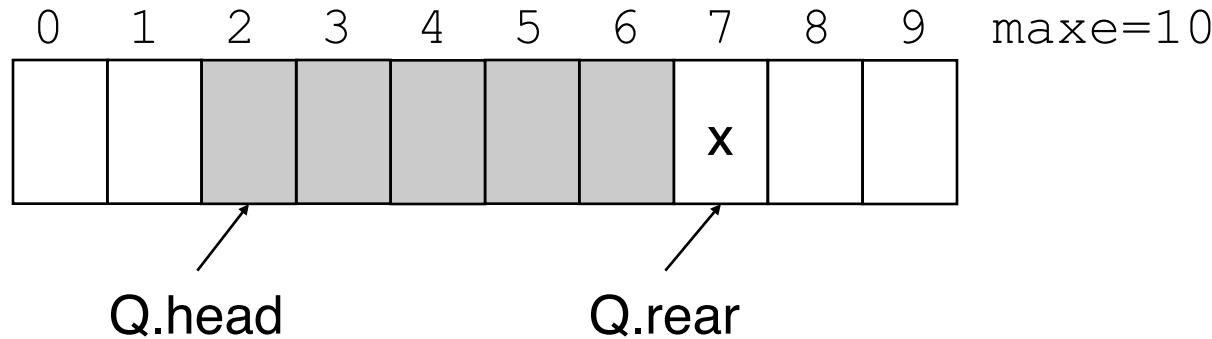
```
if( Q.head == Q.rear ) return TRUE  
else return FALSE
```



Schlange: Sequentielle Speicherung

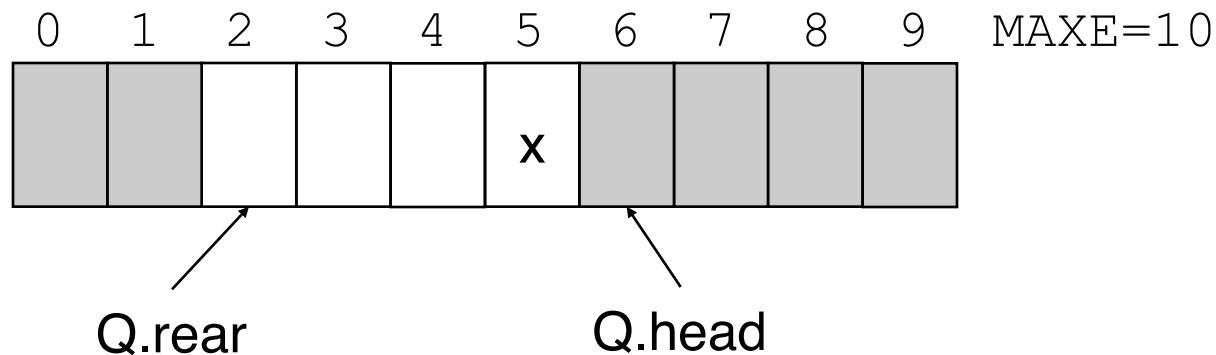
■ ENQUEUE (Q , x)

```
if ( (Q.rear +1) mod MAXE == Q.head ) error „Overflow!“  
else Q[Q.rear] = x  
    Q.rear = (Q.rear +1) mod MAXE
```



Schlange: Sequentielle Speicherung

```
■ DEQUEUE ( Q )  
if( Q.head == Q.rear ) error „Underflow!“  
else x = Q[Q.head]  
    Q.head = (Q.head + 1) mod MAXE  
return x
```



2.3 Lineare Listen

■ Lineare Liste:

- Endliche Folge von Elementen eines Grundtyps
- Elemente haben eine Ordnung: $a_1, a_2, a_3, \dots, a_n$
- Grundtyp ist von untergeordneter Bedeutung (hier Integer)
- Nomenklatur: $L = \langle a_1, a_2, a_3, \dots, a_n \rangle$; leere Liste: $\langle \rangle$

■ Grundoperationen:

- Einfügen(x, p, L): einfügen von x an Stelle p in L
 $\langle a_1, \dots, a_p, a_{p+1}, \dots, a_n \rangle \rightarrow \langle a_1, \dots, a_p, x, a_{p+1}, \dots, a_n \rangle$
- Entfernen(p, L): entfernen des p -ten Elements
 $\langle a_1, \dots, a_{p-1}, a_p, a_{p+1}, \dots, a_n \rangle \rightarrow \langle a_1, \dots, a_{p-1}, a_{p+1}, \dots, a_n \rangle$
- Suchen(x, L): Position von Element mit Wert x
 $\langle a_1, \dots, a_{p-1}, x, a_{p+1}, \dots, a_n \rangle \rightarrow p$
- Zugriff(p, L): Wert des p -ten Elements
 $\langle a_1, \dots, a_{p-1}, x, a_{p+1}, \dots, a_n \rangle \rightarrow x$

Lineare Listen

■ weiterführende Operationen:

■ Verketten(L_a, L_b): verbindet zwei Listen zu einer
 $\langle a_1, \dots, a_n \rangle \text{ II } \langle b_1, \dots, b_m \rangle \rightarrow \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle$

■ Leer(L): wahr, falls $L = \langle \rangle$

■ Länge(L): Anzahl der Elemente in L

$$\langle a_1, \dots, a_n \rangle \rightarrow n$$

■ Anhängen(L, x): hängt ein neues Element x an L an

$$\langle a_1, \dots, a_n \rangle \rightarrow \langle a_1, \dots, a_n, x \rangle$$

■ Kopf(L), rest(L): zerlegt eine Liste in erstes Element und Rest

$$\text{kopf: } \langle a_1, \dots, a_n \rangle \rightarrow a_1 \quad \text{rest: } \langle a_1, \dots, a_n \rangle \rightarrow \langle a_2, \dots, a_n \rangle$$

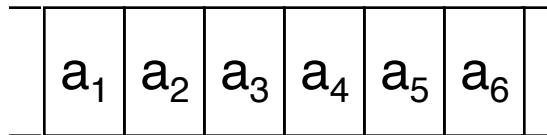
■ Entfernen2(x, L): Entfernen(Suchen(x, L), L)

:

:

Lineare Listen

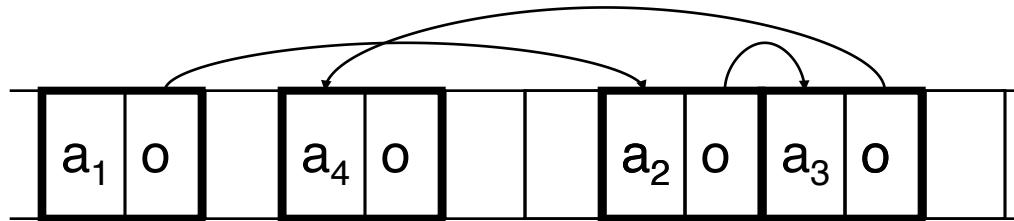
- Wie können lineare Listen am effizientesten realisiert werden?
- Variante 1: Sequenzielle Speicherung



Speicher

- Vorteil: schneller Zugriff Nachteil: langsames Einfügen

- Variante 2: verkettete Speicherung



Speicher

- Vorteil: schnelles Einfügen Nachteil: langsamer Zugriff, höherer Speicherbedarf

Lineare Listen: Sequenzielle Speicherung

■ Datenstruktur:

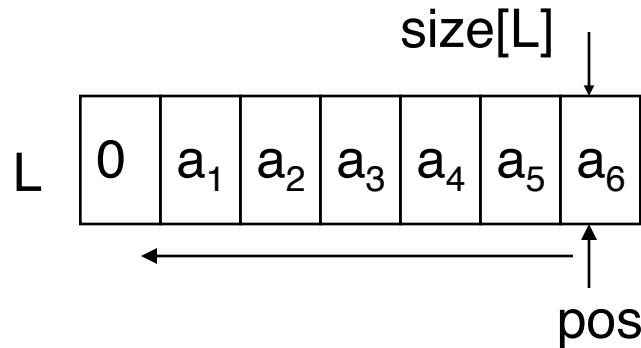
list L : Array 0,..,MAXE; Speicherung ab Position 1
L.size : Anzahl Elemente in der Liste

Zugriff	direkt über Index p: L.element[p]	O(1)
Suchen	durchlaufe die Liste bis x gefunden wurde	O(N)
Einfügen	verschiebe Elemente p+1,...,n um eine Position nach hinten	O(N)
Entfernen	verschiebe Elemente p+1,...,n um eine Position nach vorne	O(N)
Verketten	füge die Elemente von Liste 2 hinter Liste 1 ein	O(N)

Lineare Listen: Sequentielle Speicherung

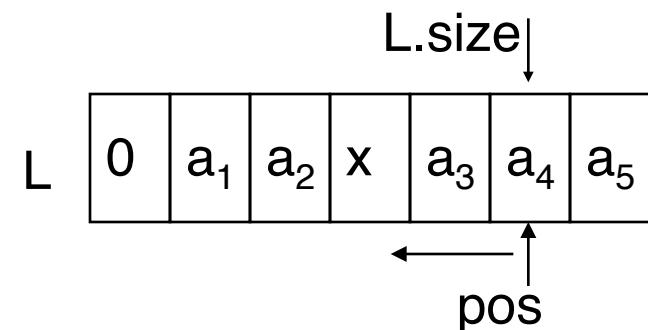
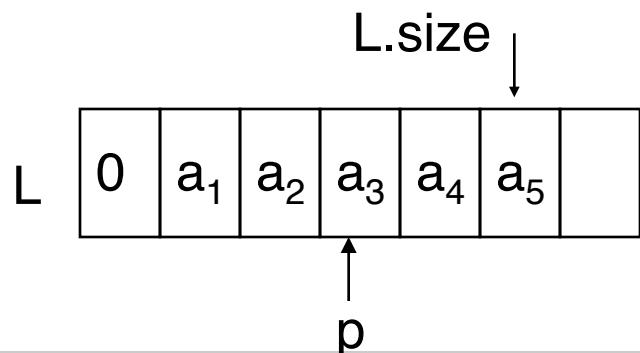
■ LIST-SEARCH (x, L)

```
L[0] = x  
pos = L.size  
while L[pos] ≠ x  
    pos = pos - 1  
return pos
```



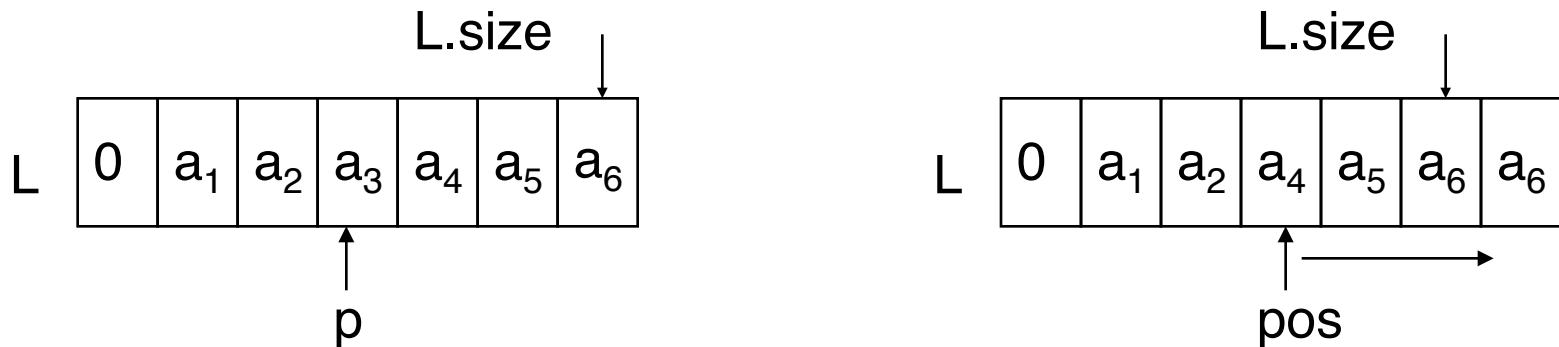
Lineare Listen: Sequentielle Speicherung

```
■ LIST-INSERT(x, p, L)
  if( L.size == MAXE ) error "Overflow!"
  else
    if( p > L.size + 1 or p < 1 ) error "Invalid position!"
    else
      for pos = L.size downto p
        L[pos+1] = L[pos]
      L[p] = x
      L.size = L.size +1
```



Lineare Listen: Sequentielle Speicherung

```
■ LIST-DELETE (p, L)
  if ( L.size == 0 ) error "Empty list!"
  else
    if ( p > L.size or p < 1 ) error "Invalid position!"
    else
      L.size = L.size -1
      for pos = p to L.size
        L[pos] = L[pos+1]
```



Lineare Listen: Verkettete Speicherung

■ Datenstruktur:

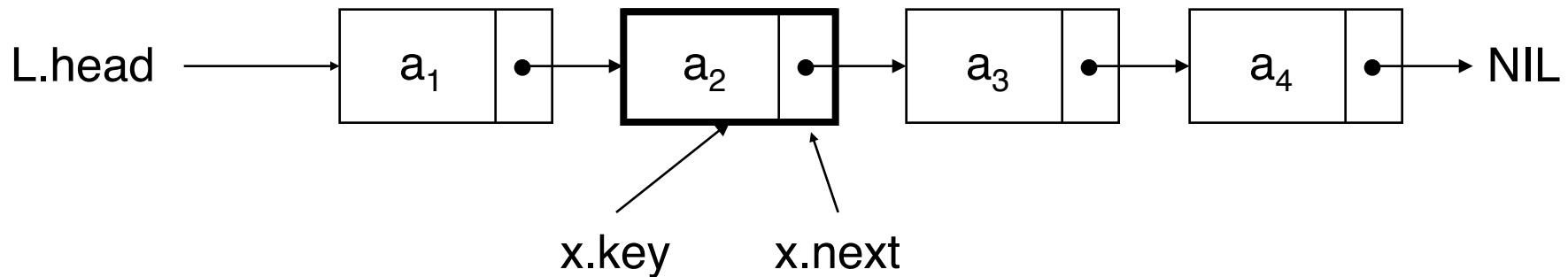
List node x : Listenelement

x.key : Schlüssel des Elements x

x.next : Zeiger auf das Nachfolger-Element von x

L.head : Zeiger auf das erste Listenelement

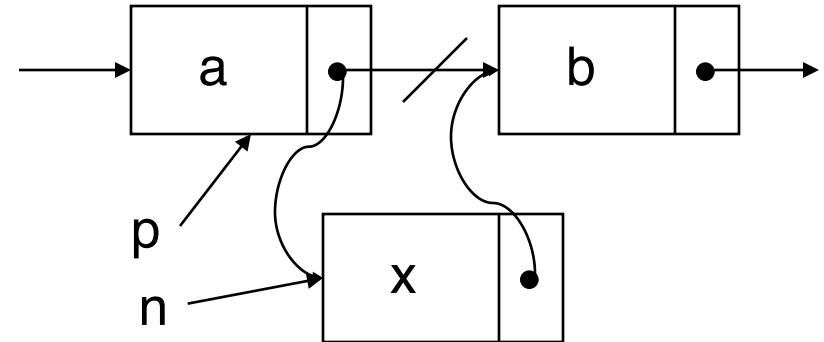
einfach verkettete Liste



Lineare Listen: Verkettete Speicherung

■ Durchlaufen der Liste (für Suchen, Einfügen, Entfernen)

```
p = L.head  
while p ≠ NIL  
    //mache etwas mit p.key  
    p = p.next
```



■ Einfügen von x hinter Element p:

```
new(n)  
n.key  = x  
n.next = p.next  
p.next = n
```

■ Entfernen hinter Position p:

```
d = p.next  
p.next = p.next.next  
delete(d)
```

■ Achtung: besondere Regeln zum Einfügen/Entfernen am Listenanfang

Verkettete Liste: Implementierung in Python 1/2

```
class Node:  
    def __init__(self):  
        self.next = None  
        self.key = ""
```

Hilfsklasse

```
class IntegerVerketteteListe:  
    def __init__(self):  
        self._head = None
```

Initialisierung

```
def Get(self, idx):  
    if idx < 0:  
        raise IndexError  
    p = self._head  
    i = 0  
    while p is not None:  
        if i is idx:  
            return p.key  
        i += 1  
        p = p.next  
    raise IndexError
```

Liefere
Inhalt an
Indexposition

```
def Insert(self, idx, e):  
    if idx < 0:  
        raise IndexError  
    if idx is 0:  
        n = Node()  
        n.key = e  
        n.next = self._head  
        self._head = n  
        return  
    p = self._head  
    i = 0  
    while p is not None:  
        if i is idx - 1:  
            n = Node()  
            n.key = e  
            n.next = p.next  
            p.next = n  
            return  
        i += 1  
        p = p.next  
    raise IndexError
```

Einfügen
von e an
Index idx

Verkettete Liste: Implementierung in Python 2/2

```
def Delete(self, idx):    Löschen an      if __name__ == "__main__":
    if idx < 0:          Index idx        print("* New list *")
        raise IndexError
    if self._head is None:
        return IndexError
    if idx is 0:
        self._head = self._head.next
        return
    p = self._head
    i = 0
    while p is not None:
        if i is idx - 1:
            p.next = p.next.next
            return
        i += 1
        p.next = p.next
    raise IndexError

def PrintList(self):
    print("++ List start +++")
    p = self._head
    while p is not None:
        print(p.key)
        p = p.next
    print("++ List end +++")
```

Löschen an Index idx

Main-Funktion zur Ansteuerung

Funktion zur Ausgabe



Lineare Listen: Doppelt verkettete Speicherung

Datenstruktur:

List node x : Listenelement

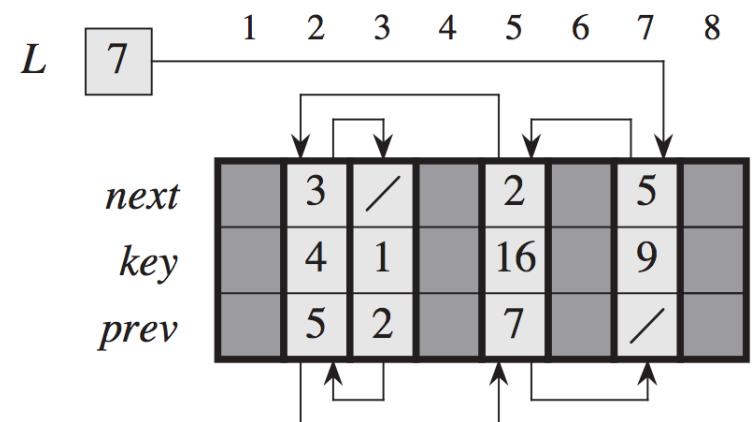
$x.key$: Schlüssel des Elements x

$x.next$: Zeiger auf das Nachfolger-Element von x

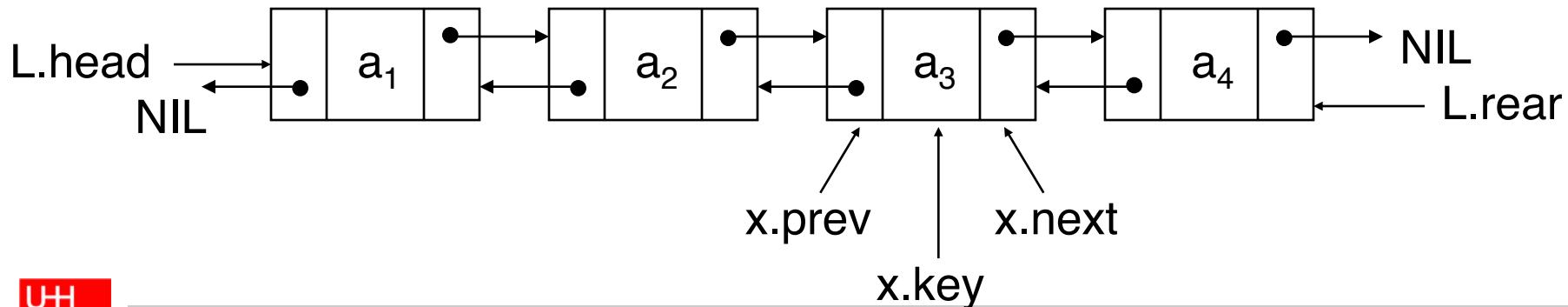
$x.prev$: Zeiger auf das Vorgänger-Element von x

$L.head$: Zeiger auf das erste Listenelement

$L.rear$: Zeiger auf das letzte Listenelement



doppelt verkettete Liste



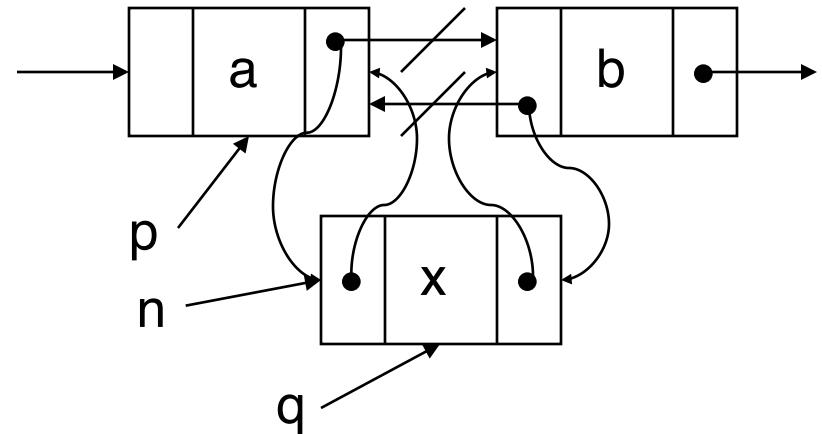
Doppelt-verkettete Listen

- Durchlaufen der Liste vorwärts und rückwärts möglich

```
p = L.head  
while p ≠ NIL  
    // mache etwas mit p.key  
    p = p.next
```

rückwärts:

head → rear, next → prev



- Einfügen hinter Position p:

```
new(n)  
n.key = x  
n.next = next[p]  
n.prev = p  
n.next.prev = n  
n.prev.next = n
```

- Entfernen an Position q:

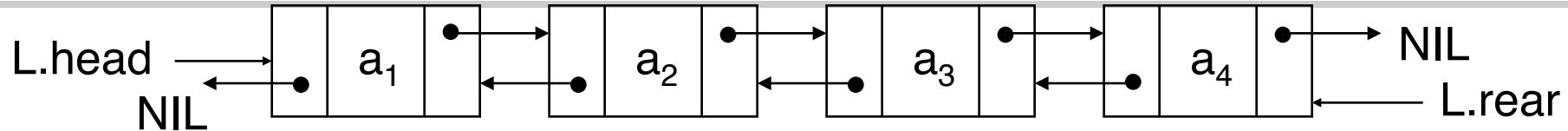
```
q.prev.next = q.next  
q.next.prev = q.prev  
delete(q)
```

Lineare Listen: (Doppelt-)verkettete Speicherung

Zugriff	durchlaufe die Liste bis Position p	O(N) [O(1)]
Suchen	durchlaufe die Liste bis x gefunden wurde	O(N)
Einfügen	erzeuge neues Element, ‚verbiege‘ Zeiger	O(N) [O(1)]
Entfernen	‚verbiege‘ Zeiger, lösche das Element	O(N) [O(1)]
Verketten	next-Zeiger des letzten Elements der Liste 1 zeigt auf erstes Element der Liste 2	O(1)

[] : falls pos-Zeiger bereits an Position p

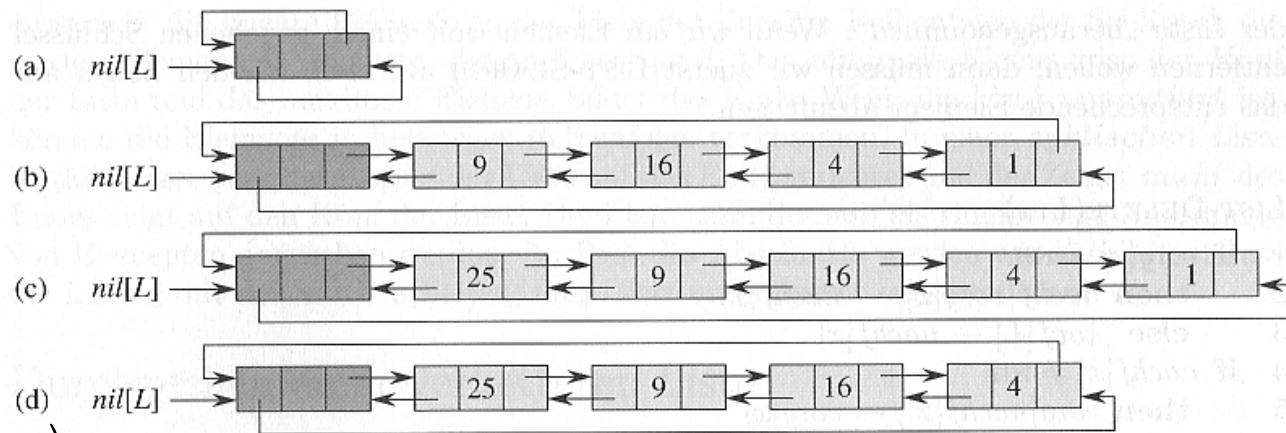
Doppelt-verkettete Liste ohne Wächter



- Spezieller Code für das Einfügen und Löschen am Listenanfang / -ende notwendig:
- ```
LIST-INSERT(L, x, p) // fügt Element x hinter p ein
 x.prev = p
 if p == NIL // Listenanfang
 x.next = L.head
 if(L.head == NIL) L.rear = x // Listenanfang/-ende
 else L.head.prev = x
 L.head = x
 else // Listenmitte oder -ende
 x.next = p.next
 p.next = x
 if(x.next == NIL) L.rear = x // Listenende
 else x.next.prev = x
 L.size = L.size + 1
```

# Doppelt-verkettete Liste mit Wächter (Sentinel)

- Wächter  $L.\text{nil}$ : spezielles Listenelement, repräsentiert Listenanfang und -ende
  - $L.\text{nil}.\text{next}$ : Listenanfang ( $= L.\text{head}$ )
  - $L.\text{nil}.\text{prev}$ : Listenende ( $= L.\text{rear}$ )
  - $L.\text{nil}.\text{key}$ : NIL (speichert keinen Wert)



- **LIST-SEARCH( $L, x$ )**

```
p = L.nil.next
while(p ≠ L.nil and
 p.key ≠ x)
 p = p.next[p]
return p
```

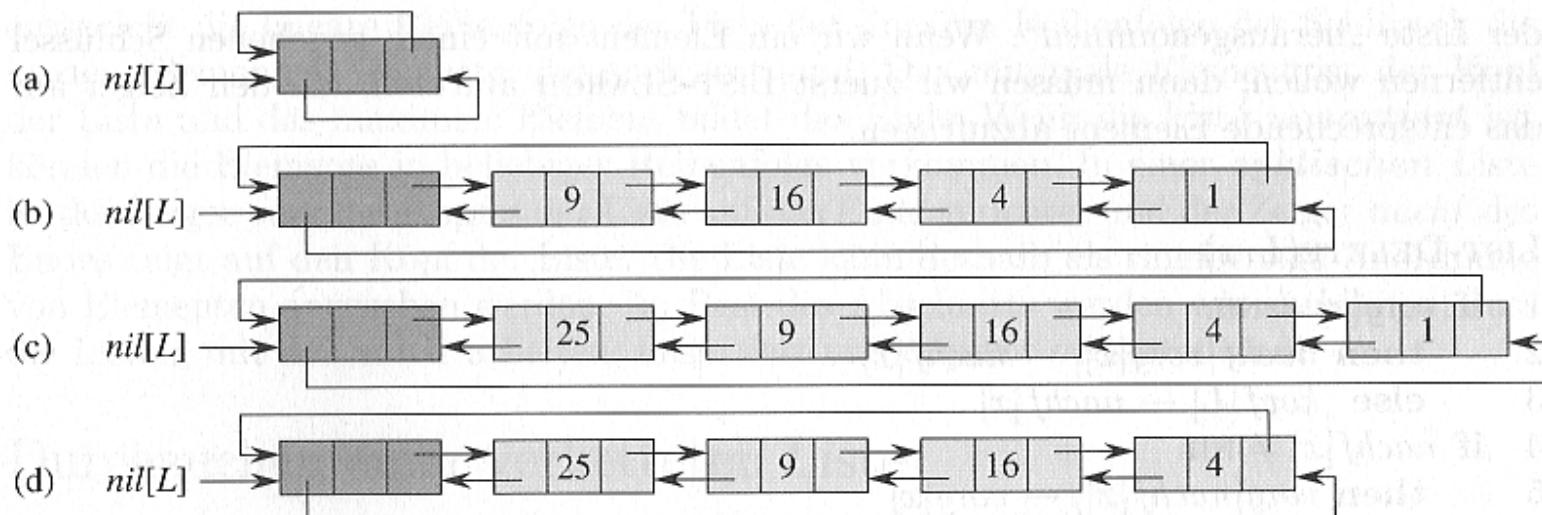
# Doppelt-verkettete Liste mit Wächter

■ LIST-INSERT ( $L, x, p$ )

```
x.next = p.next
x.prev = p
x.prev.next = x
x.next.prev = x
```

■ LIST-DELETE ( $L, x$ )

```
x.prev.next = x.next
x.next.prev = x.prev
```



## 2.4 Bäume

---

- Datenstruktur (Binär-)Baum:
  - Knoten mit einer endlichen Anzahl (2) Nachfolger
- Nomenklatur:
  - Knoten: Element eines Baums
  - (direkter) Vorgänger: vorheriges Element im Baum (eindeutig!)  
(Vater, Elter, parent)
  - (direkter) Nachfolger: nachfolgendes Element im Baum  
(Tochter/Sohn, Kind, child)
  - Vorfahre: Knoten auf dem Weg zur Wurzel
  - Nachfahre: Knoten auf dem Weg zu einem Blatt
  - geordneter Baum: Nachfolger haben eine feste Reihenfolge  
(left, right bei Binärbäumen)
  - Wurzel: Knoten ohne Vorgänger
  - innerer Knoten: mit Nachfolger
  - Blatt / externer Knoten: Knoten ohne Nachfolger
  - Ordnung: Anzahl direkter Nachfolger eines Knotens
  - Pfad ( $v_1, \dots, v_k$ ): Folge von Knoten mit  $v_i = \text{parent}(v_{i-1})$

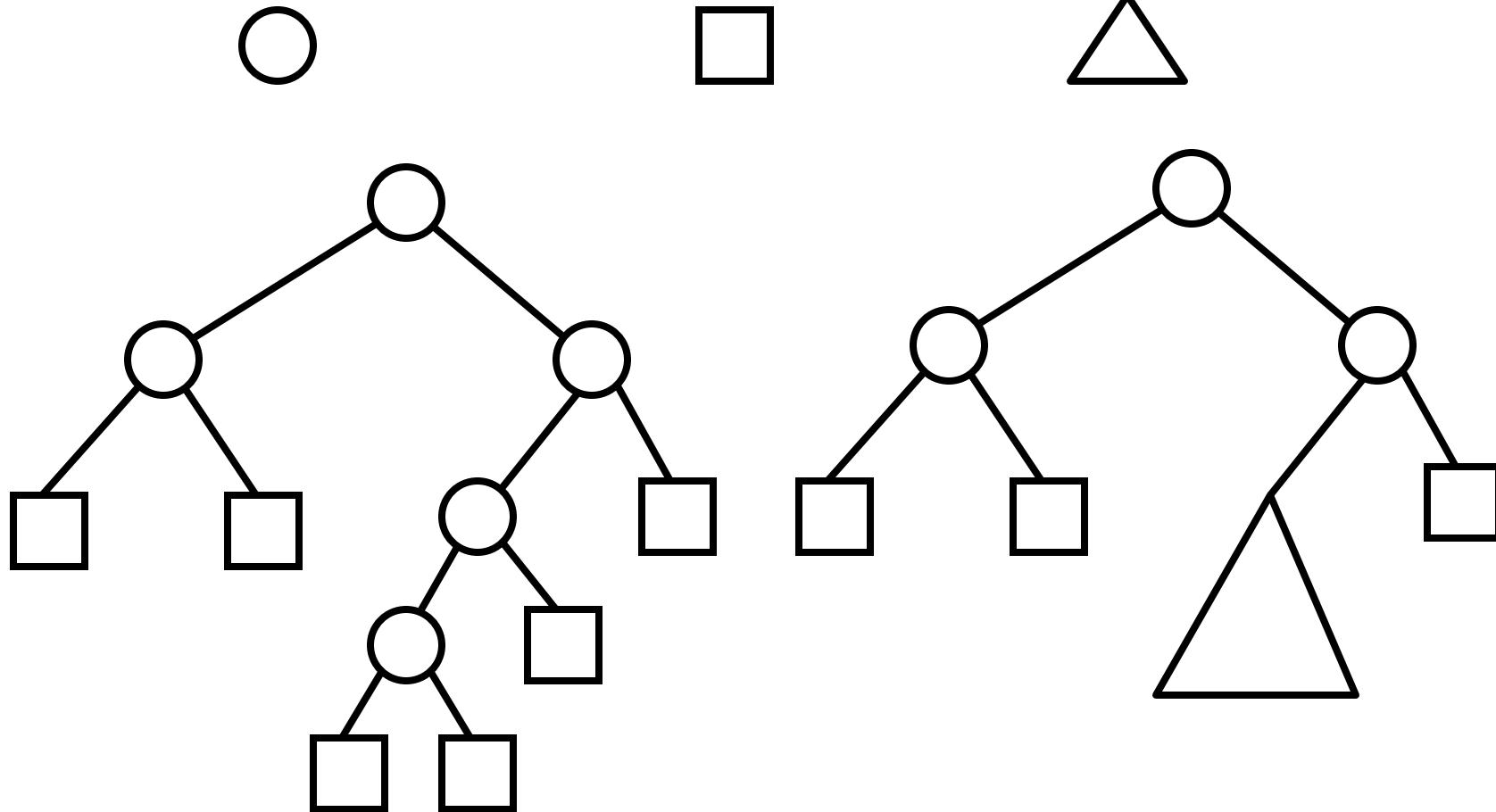
# Bäume

## ■ Nomenklatur:

■ innerer Knoten

Blatt

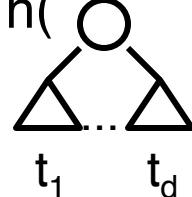
Teilbaum



# Bäume: Höhen und Tiefen

- Höhe eines (Teil-)Baums: (rekursive Definition)

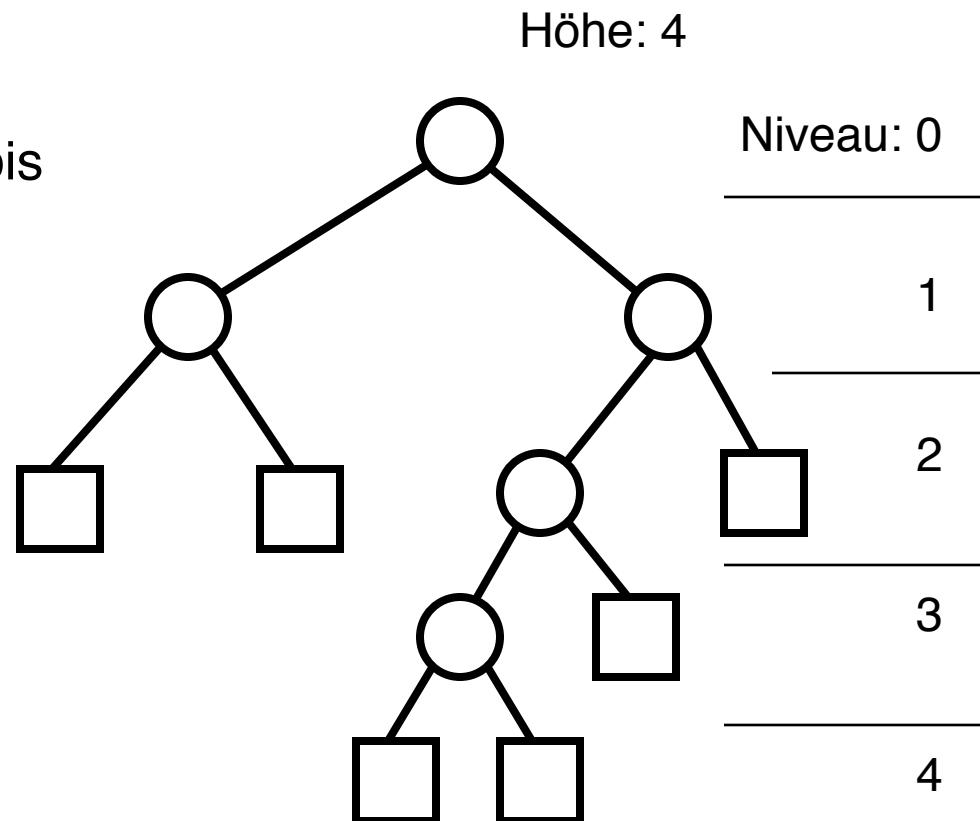
- $h(\square) = 0; h(\text{---}) = \max\{h(t_1), \dots, h(t_d)\} + 1$



- Tiefe eines Knotens  $k$ :

- Anzahl der Kanten von  $k$  bis zur Wurzel

- $t(\text{Wurzel}) = 0;$   
 $t(x) = t(\text{parent}(x)) + 1$



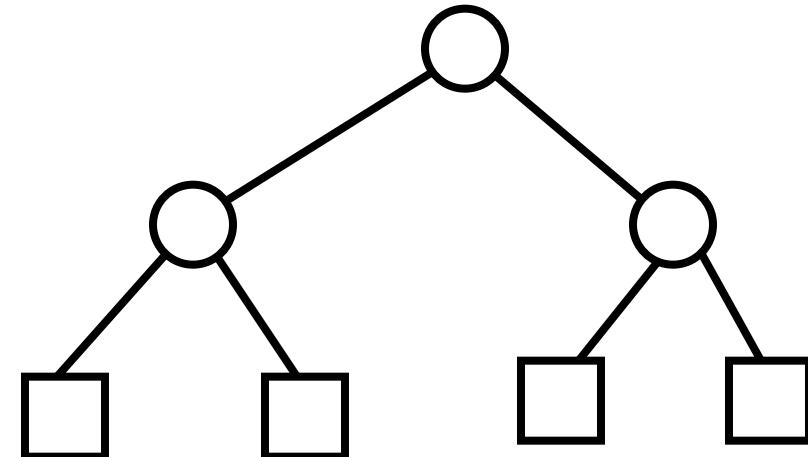
- Niveau / Ebene  $i$ :

- alle Knoten der Tiefe  $i$

# Bäume: Zahlen

## ■ Vollständiger Binärbaum der Höhe $h$ :

- auf jedem Niveau die maximal mögliche Anzahl Knoten
- alle Blätter auf Niveau  $h$



## ■ Für einen vollst. Binärbaum der Höhe $h$ gilt:

- Anzahl Knoten auf Ebene  $i$ :  $2^i$

- Anzahl der Blätter:  $2^h$

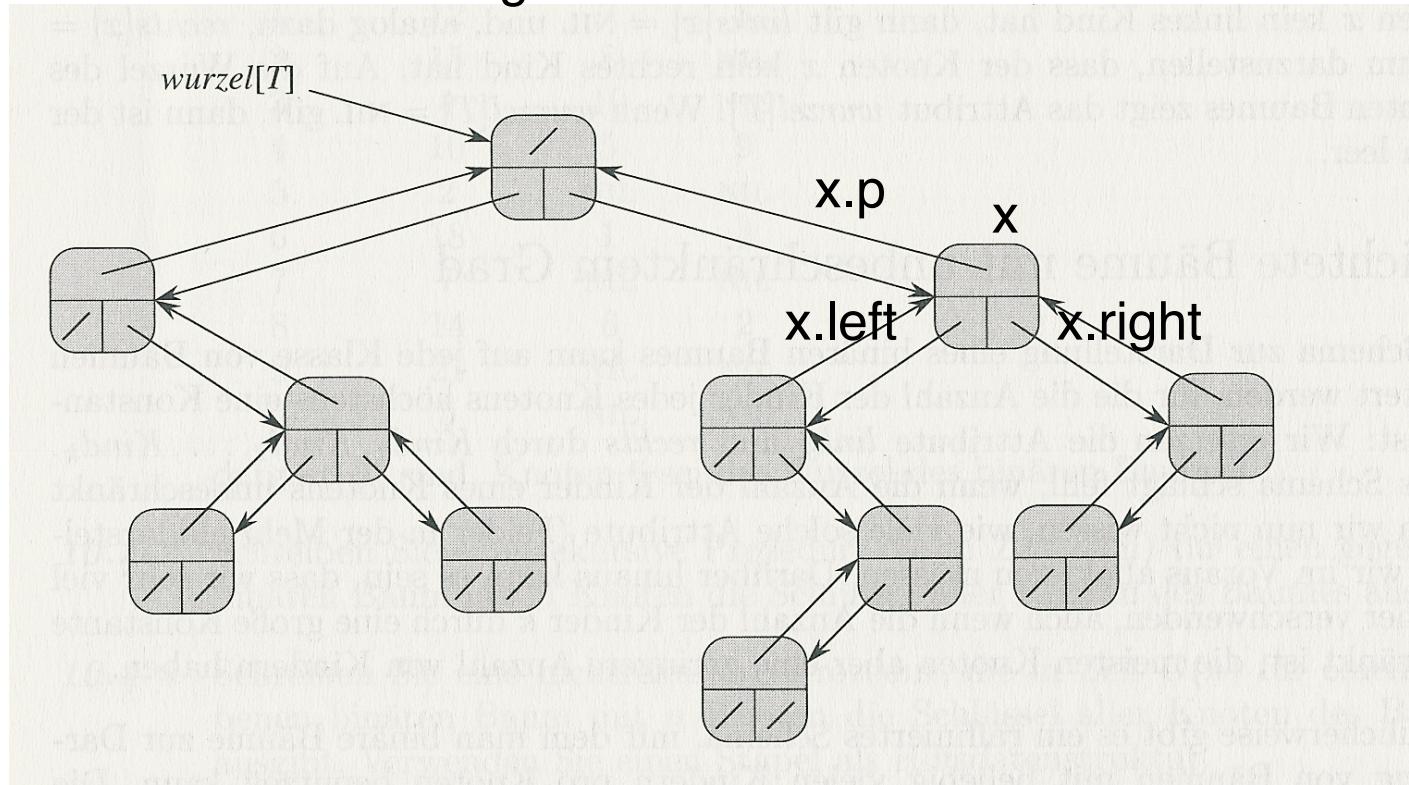
- Anzahl der Knoten:  $|K| = \sum_{i=0}^h 2^i = 2^{h+1} - 1$

- zur Speicherung von IKI Knoten benötigt man einen Binärbaum der Höhe:

$$h = \log_2\left(\frac{|K| + 1}{2}\right) = \log_2(|K| + 1) - 1 = \Theta(\log |K|)$$

# Darstellung von binären Bäumen

- $T.\text{root}$  : Wurzel des Baums  $T$
- Sei  $x$  ein Knoten des Baumes, dann ist
  - $x.p$  : Vorgänger / Elter von  $x$
  - $x.\text{left}$  : linker Nachfolger / Kind von  $x$
  - $x.\text{right}$  : rechter Nachfolger / Kind von  $x$



# Darstellung von Bäumen mit unbeschränktem Grad

- Vermeidung von Nachfolger Listen/Arrays durch *left-child/right-sibling* Repräsentation:

- $x.p$  : Vorgänger / Elter von  $x$
- $x.left-child$  : links-stehender Nachfolger von  $x$
- $x.right-sibling$  : rechtes Geschwister von  $x$

