

Algorithmen und Datenstrukturen

Teil 2 – Datenstrukturen

Prof. Dr. Peter Jüttner

Fundamentale Datenstrukturen

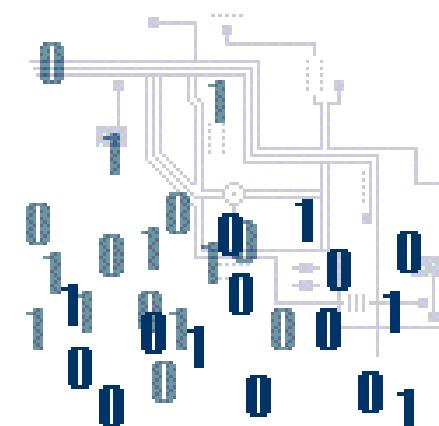
Einfache Datentypen

- **Der Datentyp, zu dem eine Konstante oder Variable gehört, legt die Werte fest, die die Konstante oder Variable annehmen kann und die Standard-Operationen (Funktionen und Operatoren), die darauf ausgeführt werden können.**
- **Der Datentyp legt auch den Speicherplatz fest, den eine Variable des Typs benötigt**
- **Programmiersprachen bieten in der Regel einen Satz vordefinierter Standard-Datentypen und darauf definierter Operationen an.**

Fundamentale Datenstrukturen

Einfache Datentypen in C (Wiederholung)

- **Ganzzahlige Datentypen mit Vorzeichen:** (char), short, int, long
- **Ganzzahlige Datentypen ohne Vorzeichen:** (unsigned char), unsigned short, unsigned int, long
- **Gleitkommazahlen:** float, double, long double
- **Buchstaben:** char
- **(Pointer: *auf Datentyp)**
- **(File)**



Fundamentale Datenstrukturen

Aufbau von Strukturen in C (Wiederholung)

- Aus bereits vorhandenen Datentypen können neue Datentypen mittels Arrays, Strukturen und Pointer aufgebaut werden.
- Beispiel:

```
struct punkt /* Aufbau einer Struktur aus Standardtypen */  
{ float x; /* X-Koordinate */  
    float y; /* Y-Koordinate */  
};
```

```
struct kreis /* Aufbau einer Struktur aus bestehenden Typen */  
{ punkt Mittelpunkt;  
    float radius;  
};
```

vor struct kann auch typedef verwendet werden (typedef struct ...)

Fundamentale Datenstrukturen

Aufbau von Strukturen in C (Wiederholung)

- Aus bereits vorhandenen Datentypen können neue Datentypen mittels Arrays, Strukturen und Pointer aufgebaut werden.
- Beispiel:

```
#define VECTORLAENGE 10
typedef struct vector /* Aufbau einer Struktur aus Standardtypen */
{ float feld[VECTORLAENGE];
};
```

Motivation

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Datentypen

Darstellung verschiedener Datentypen

- **Stack (Stapel, Keller)**
- **Listen**
- **Queue (Schlange)**

→ **Definition**

→ **Anwendung / Nutzen**

→ **Implementierung (z.T. Übung)**

Datentypen

Darstellung verschiedener Datentypen

- Die Namen der Datentypen und ihrer Funktionen der vorgestellten Datentypen sind „gebräuchlich“ aber nicht „normiert“, d.h. sie können in einem anderen Kontext (andere Vorlesung, Literatur) auch anders lauten

Datentypen

Stack (Stapel, Keller)

- Zugriff nur „von oben“
 - Ablegen oben
 - Wegnehmen oben
 - kein direkter Zugriff auf Elemente, die unterhalb des obersten Elements liegen



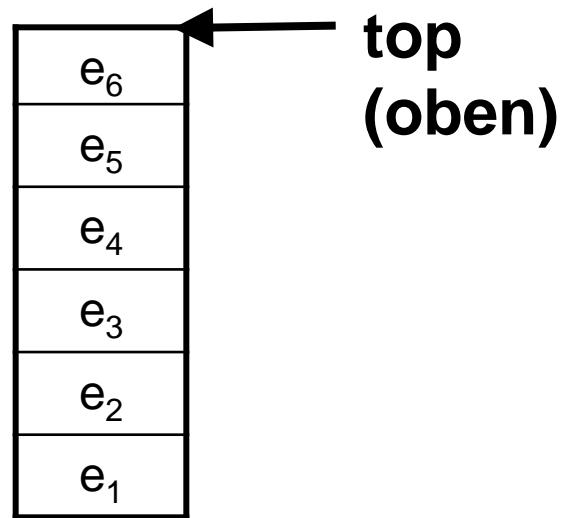
Datentypen

Stack

- **Definition**

Ein Stack (über einem bereits existierenden Datentyp T) besteht aus einer Folge von Elementen (vom Typ T), die nur an einem Ende der Folge gelesen oder beschrieben werden kann.

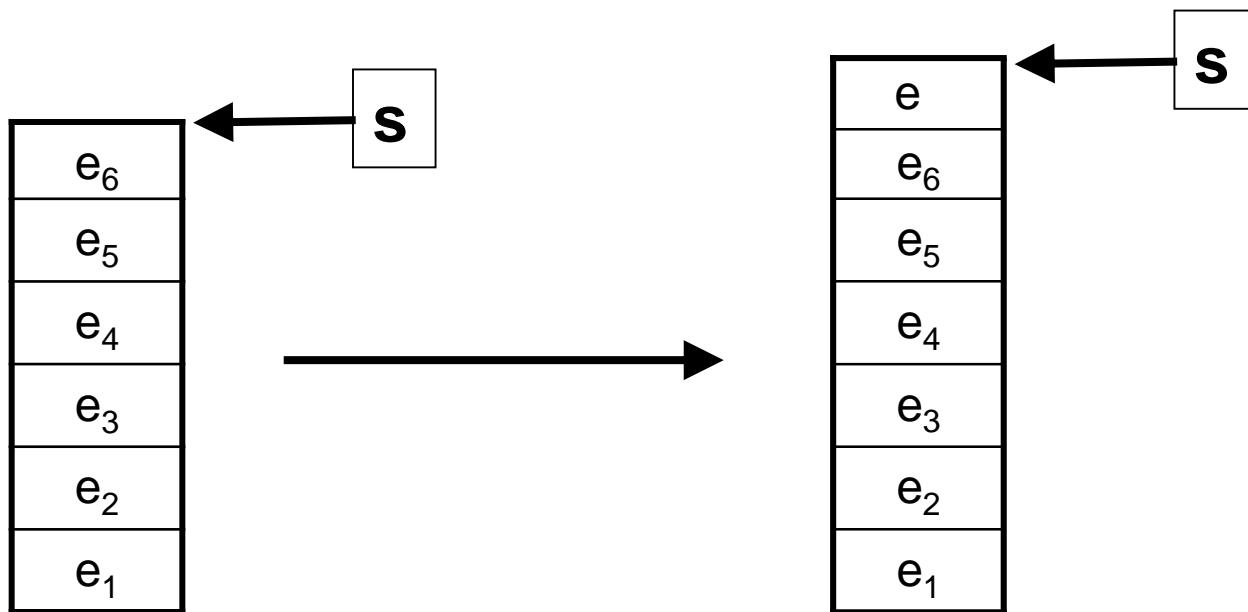
- **Beispiel: Stack mit 6 Elementen**



Datentypen

Stack

- **Funktionen**
 - **stack* Push (stack *s, T e)**
/* fügt oben ein Element an */

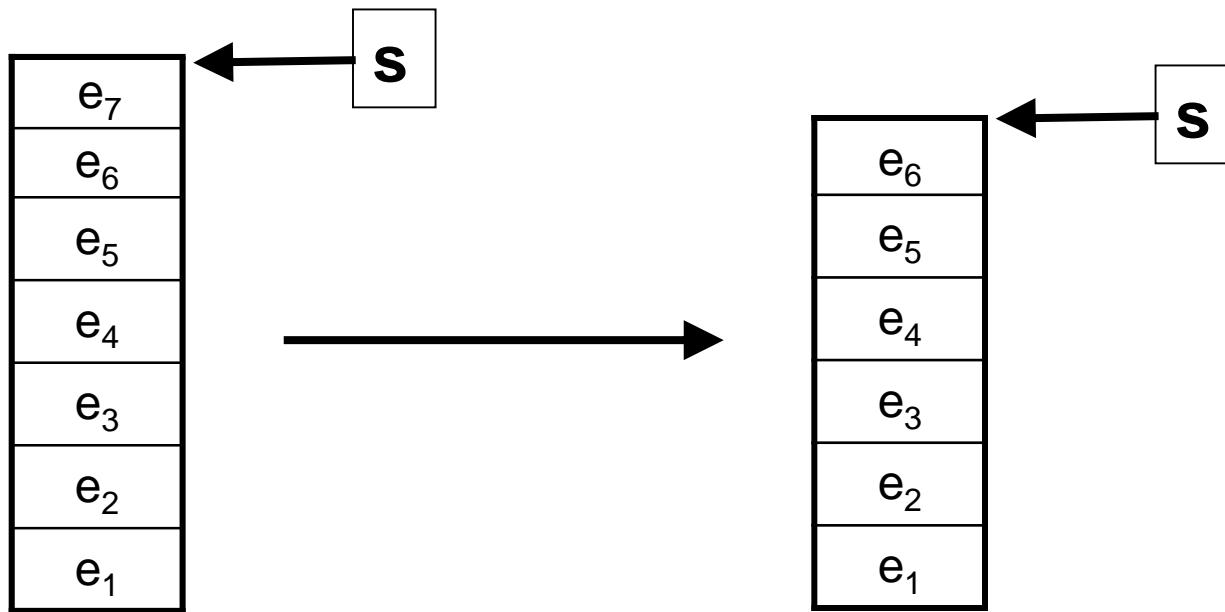


Datentypen

Stack

- **Funktionen**

- **stack* Pop (stack *s) /* löscht oberstes Element */
/* Voraussetzung: Stack ist nicht leer! */**

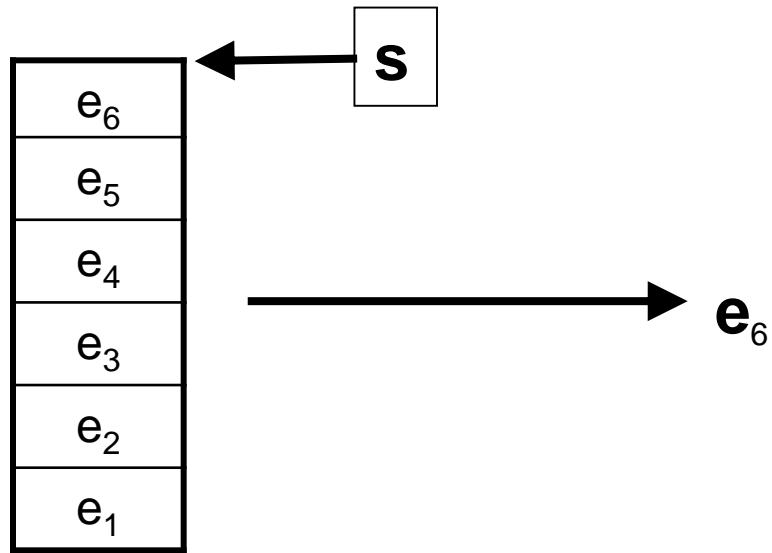


Datentypen

Stack

- **Funktionen**

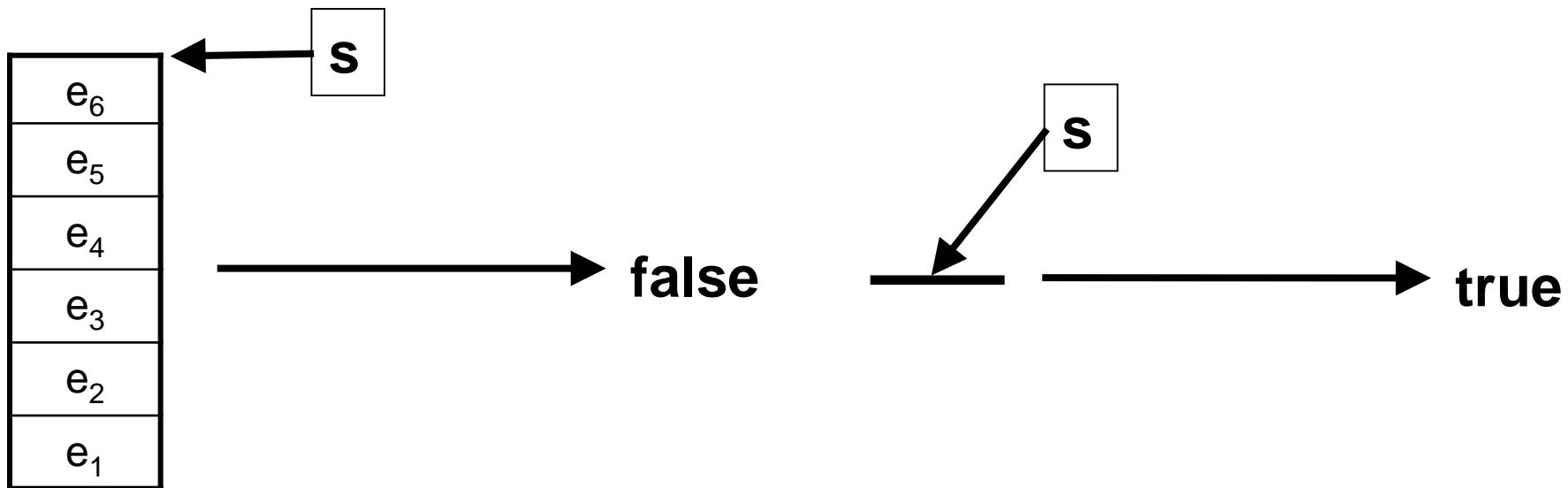
- T **Top (stack *s) /* liefert oberstes Element */**
/* Stack bleibt unverändert */
/* Voraussetzung: Stack ist nicht leer! */



Datentypen

Stack

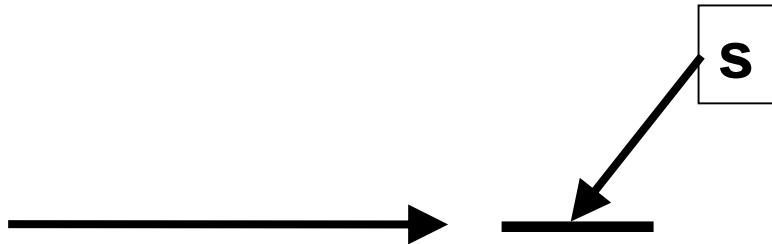
- **Funktionen**
 - **Bool Isempty (stack *s) /* Stack leer ? */
/* Stack bleibt unverändert */**



Datentypen

Stack

- **Funktionen**
 - **stack* emptystack() /* Erzeugt leeren Stack */**



Datentypen

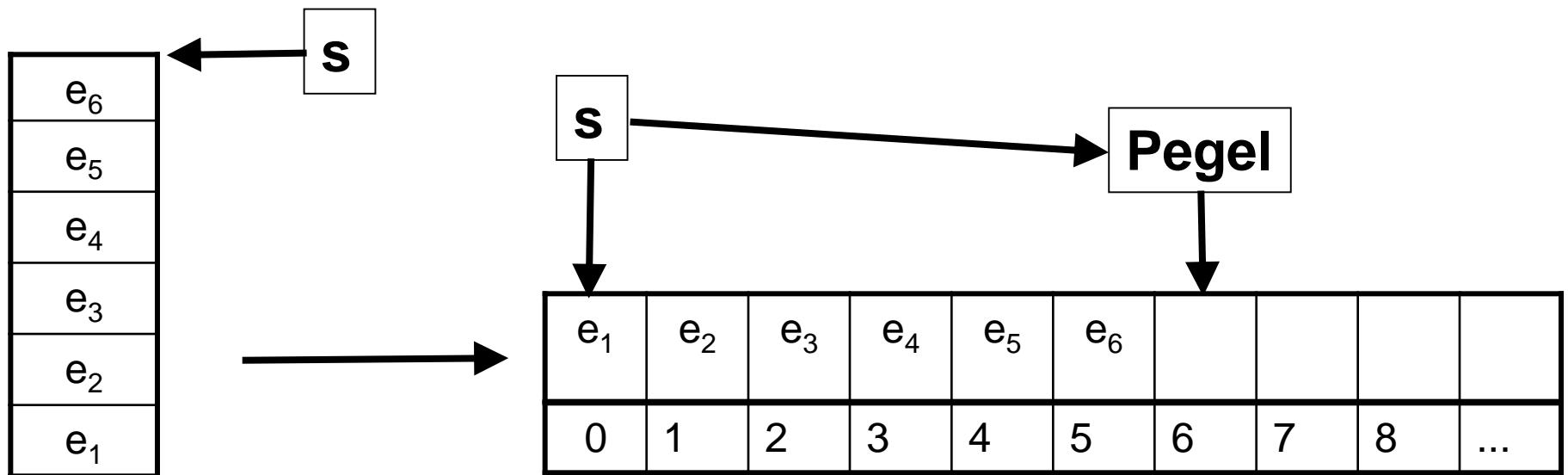
Stack

- Die Datenstruktur Stack kann durch ihre Eigenschaften formal vollständig beschrieben werden (**Axiomatische Beschreibung**)
- **Eigenschaften (Axiome)**
 - **isempty (emptystack()) = true**
 - **not isempty(s) \Rightarrow push(pop(s),top(s)) = s**
 - **isempty(push(s,e)) = false**
 - **top(push(s,e)) = e**
 - **top(emptystack()) \rightarrow Error**
 - **pop(emptystack()) \rightarrow Error**

Datentypen

Stack

- Implementierungen, z.B.
 - als Array (Stapel wird auf Array gekippt)



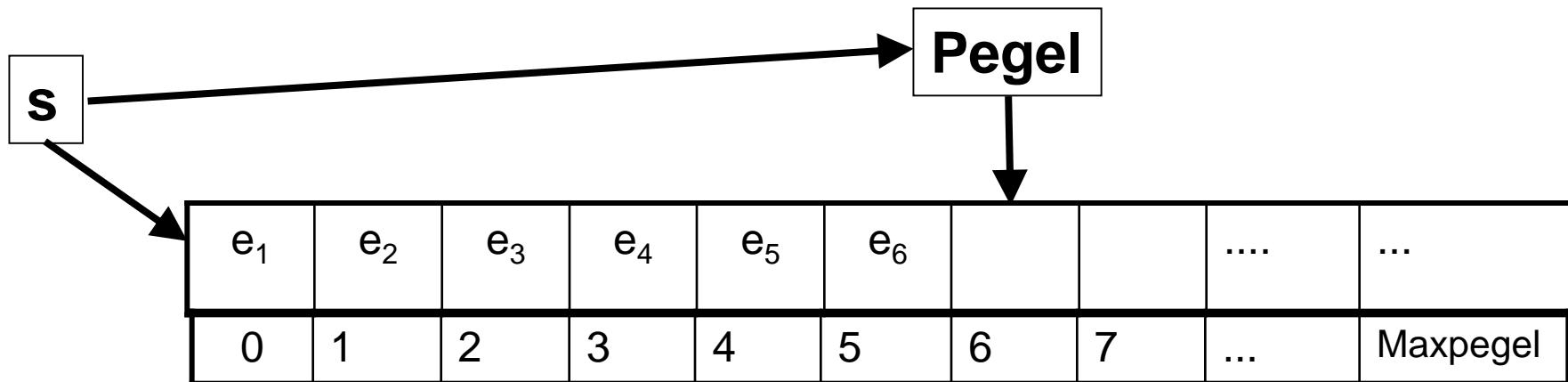
Datentypen

Stack

- **Implementierungen mittels eines Array → Datenstruktur**

```
const int Maxpegel = 1000; // Maximale Stackgröße
```

```
struct stack
{ int stackfield[Maxpegel]; / Array, das den Stack simuliert
  int pegel; // Füllstandspegel des Stack, erstes freies Element
};
```

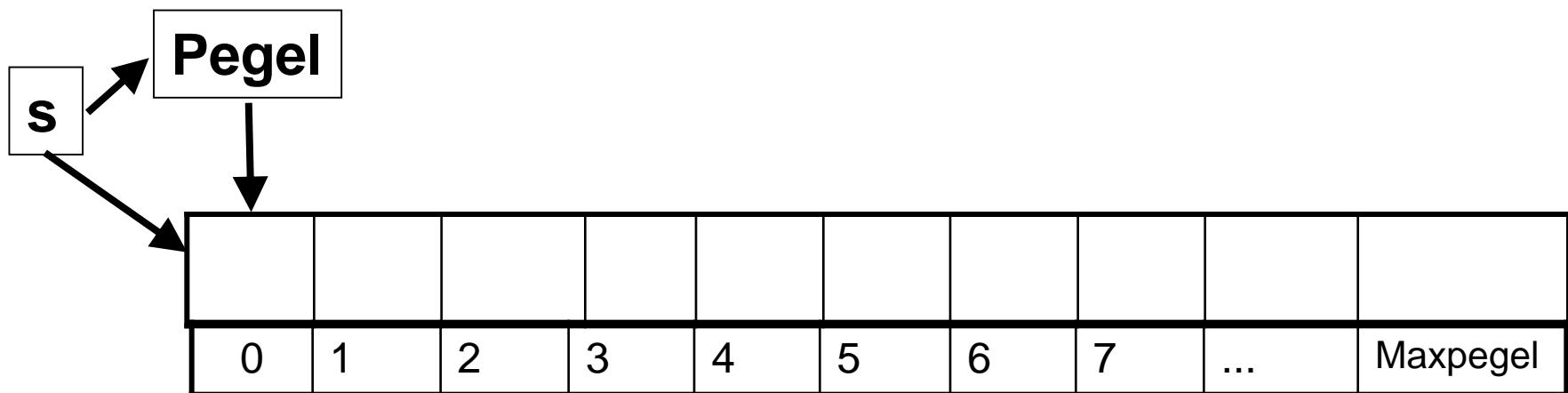


Datentypen

Stack

- **Implementierungen mittels eines Array → Funktionen**

```
stack* emptystack() // Erzeugt leeren Stack
{
    stack *stackpointer = new stack; // Dynamische Speicherplatzanforderung
    stackpointer->pegel = 0; // Stack ist noch leer
    return stackpointer;
};
```

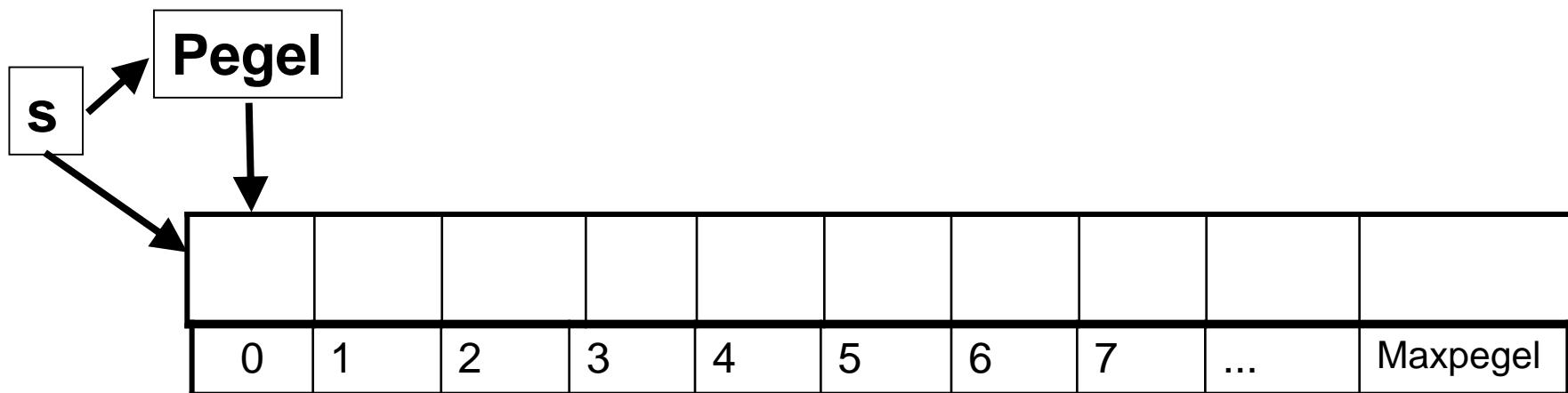


Datentypen

Stack

- **Implementierungen mittels eines Array → Funktionen**

```
int isempty(stack * stackpointer) // Stellt fest, ob der Stack leer ist
{ if (stackpointer == NULL)
    stackerror("Fehler: Isempty aufgerufen mit Nullpointer\n");
else return(stackpointer->pegel == 0);
};
```



Datentypen

Stack

- **Implementierungen mittels eines Array → Funktionen**

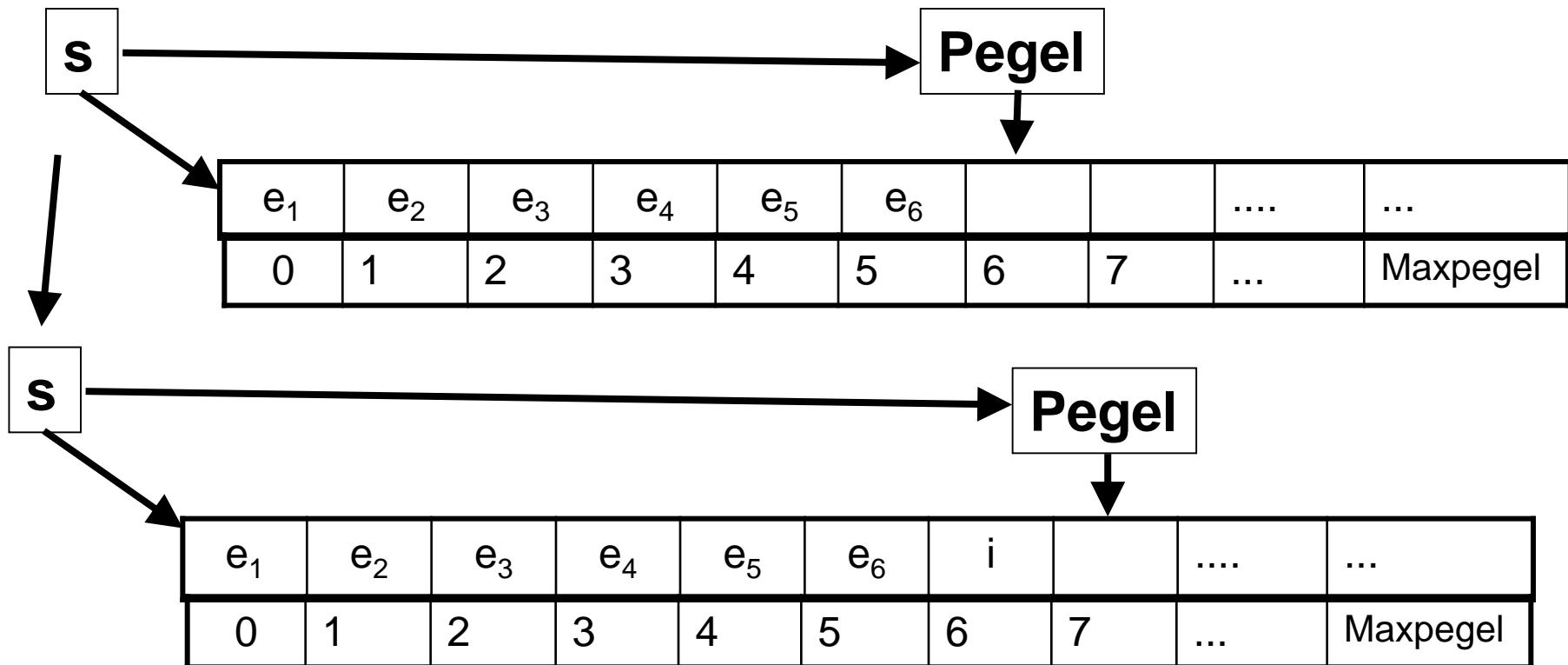
```
stack* push(stack *stackpointer, int i) // Legt ein Element auf den Stack
{ if (stackpointer == NULL)
    stackerror("Fehler: push aufgerufen mit Nullpointer!\n");
  else if (stackpointer->pegel == Maxpegel)
    stackerror("Fehler: push aufgerufen mit vollem Stack!\n");
  else
    { stackpointer->stackfield[stackpointer->pegel] = i;
      stackpointer->pegel++;
      return stackpointer;
    };
};
```

Datentypen

Stack

- **Implementierungen mittels eines Array → Funktionen**

```
stack* push(stack *stackpointer, int i) // Legt ein Element auf den Stack
```



Datentypen

Stack

- **Implementierungen mittels eines Array → Funktionen**

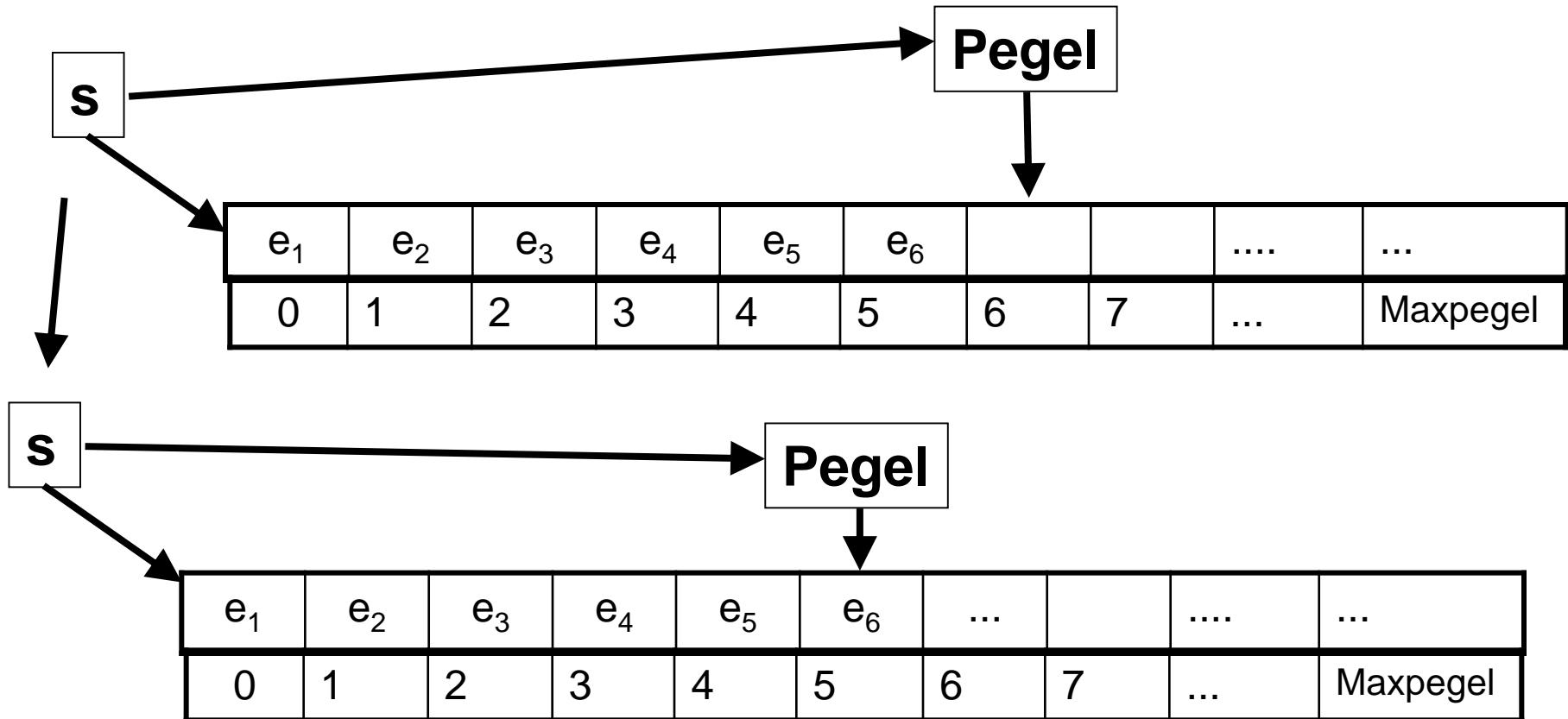
```
stack* pop(stack *stackpointer) // Entfernt ein Element vom Stack
{ if (stackpointer == NULL)
    stackerror("Fehler: pop aufgerufen mit Nullpointer!\n");
else if (stackpointer->pegel>0)
    { stackpointer->pegel--;
      return stackpointer;
    }
else stackerror("Fehler: pop aufgerufen mit leerem Stack!\n");
};
```

Datentypen

Stack

- **Implementierungen mittels eines Array → Funktionen**

```
stack* pop(stack *stackpointer) // Entfernt ein Element vom Stack
```

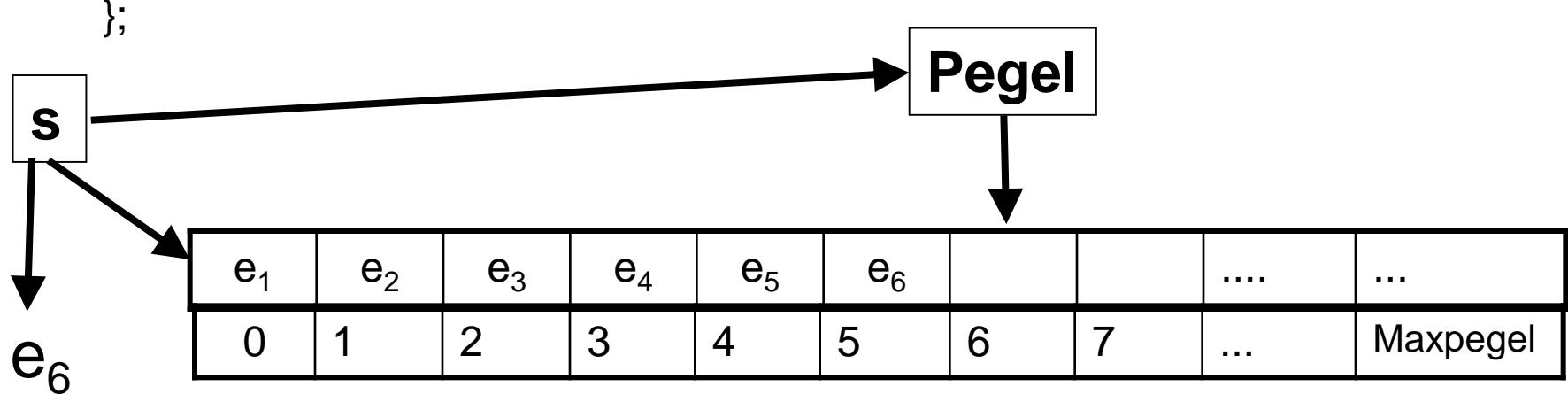


Datentypen

Stack

- **Implementierungen mittels eines Array → Funktionen**

```
int top(stack *stackpointer)
{ if (stackpointer == NULL)
    stackerror("Fehler: top aufgerufen mit Nullpointer!\n");
else if (stackpointer->pegel>0)
    return(stackpointer->stackfield[stackpointer->pegel-1]);
else stackerror("Fehler: top aufgerufen mit leerem Stack!\n");
};
```



Datentypen

Stack Anwendungen

- **Speicherverwaltung bei Laufzeitsystemen:**
 - Parameterübergabe
 - lokale/globale Variable
- **Verwaltung der Aufrufinformation bei geschachtelten und rekursiven Funktionen**
- **Umwandlung von rekursiven Programmen in nicht-rekursive**
- **LIFO-Speicher (Last-In-First-Out)**

Datentypen

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Datentypen

(einfach) Verkettete Liste (List)

- Zugriff nur „am hinteren Ende“
 - Anfügen und Wegnehmen hinten an der Liste
 - kein direkter Zugriff auf Elemente, die zwischen dem ersten und letzten Element liegen



Datentypen

(einfach) Verkettete Liste

- **Definition**

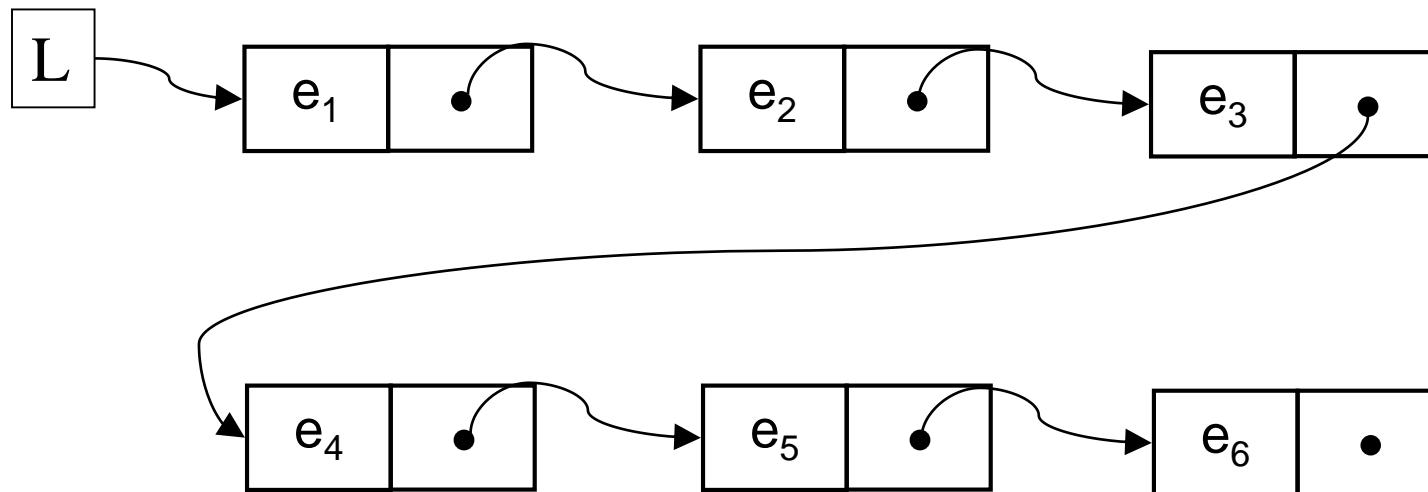
Eine einfach verkettete Liste (über einem bereits existierenden Datentyp T) besteht aus einer Folge von Listenelementen, die wiederum aus einem Element vom Typ T und aus einem Zeiger auf ein Listenelement bestehen. Der direkte (lesende und schreibende) Zugriff erfolgt am Ende der Liste¹⁾.

¹⁾Der Zugriff erfolgt wie bei einem Stack, daher lassen sich mittels Listen Stacks einfach implementieren. Mittels eines Stacks lässt sich auch eine Liste darstellen (eher ungewöhnlich)

Datentypen

(einfach) Verkettete Liste

- Beispiel: Liste mit 6 Elementen



- ohne Pfeil steht für leeren Pointer (in C NULL)

Datentypen

(einfach) Verkettete Liste

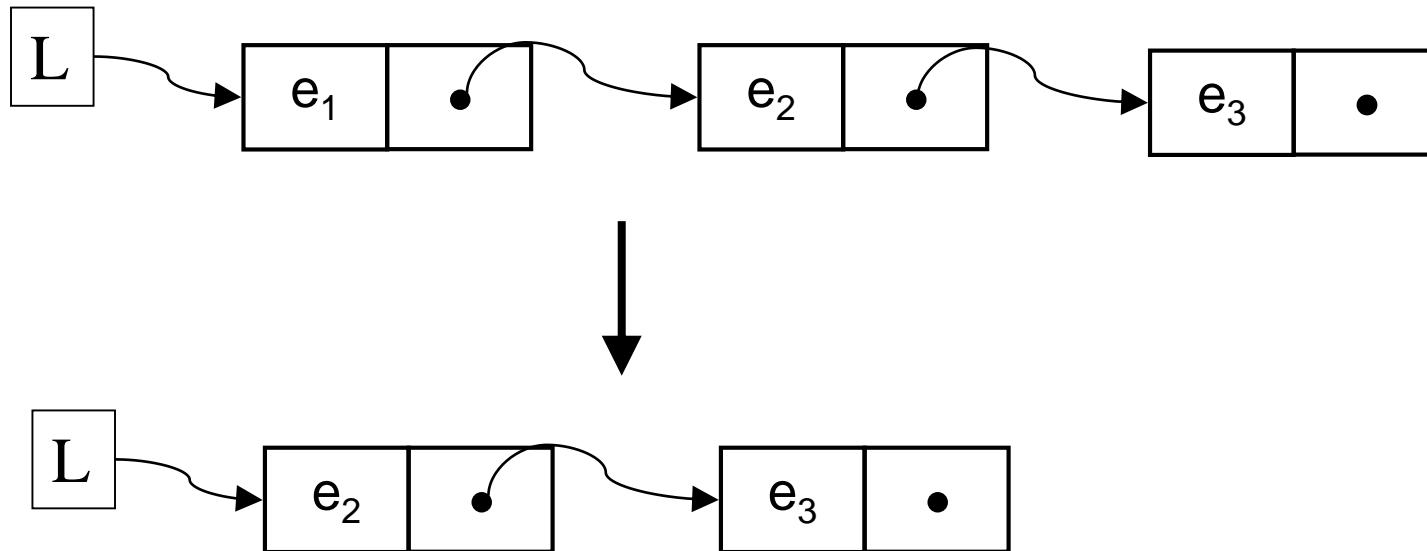
- Im Unterschied zu dem vorher besprochenen Datentyp Stack und noch zu besprechenden Typ Queue legen Listentypen einen bestimmte Implementierung (Struktur) über Pointer nahe.
- „Aufweichen“ der rein axiomatische Definition des Datentyps über seine Eigenschaften
- Trotzdem kann auch eine Liste „rein axiomatisch“ definiert werden.
- Im Unterschied zu Stack und Queue (die meist mittels Arrays implementiert werden) sind Listen als dynamischer Datentyp (theoretisch) nicht begrenzt.

Datentypen

(einfach) Verkettete Liste

- **Funktionen**

- **list* Tail (list *l) /* löschte erstes Element */**
- **/* Voraussetzung: Liste ist nicht leer! */**

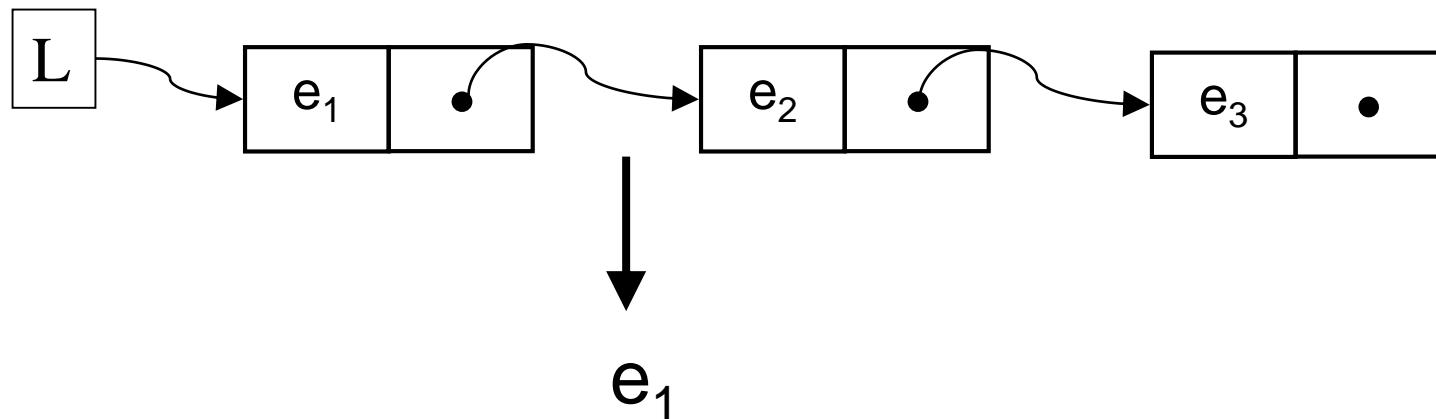


Datentypen

(einfach) Verkettete Liste

- **Funktionen**

- **T Head (list * l) /* liefert erstes Element */
/* Liste bleibt unverändert */
/* Voraussetzung: Liste ist nicht leer! */**

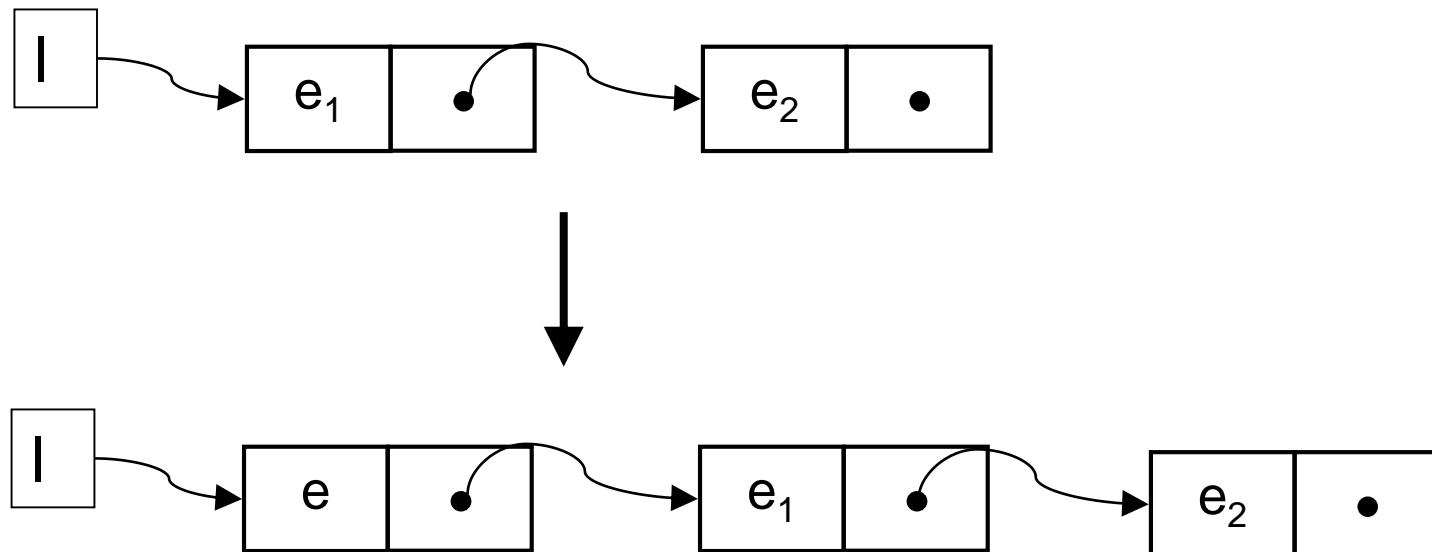


Datentypen

(einfach) Verkettete Liste

- **Funktionen**

- **list* Append (list *l, T e) /* fügt Element vorne an */**

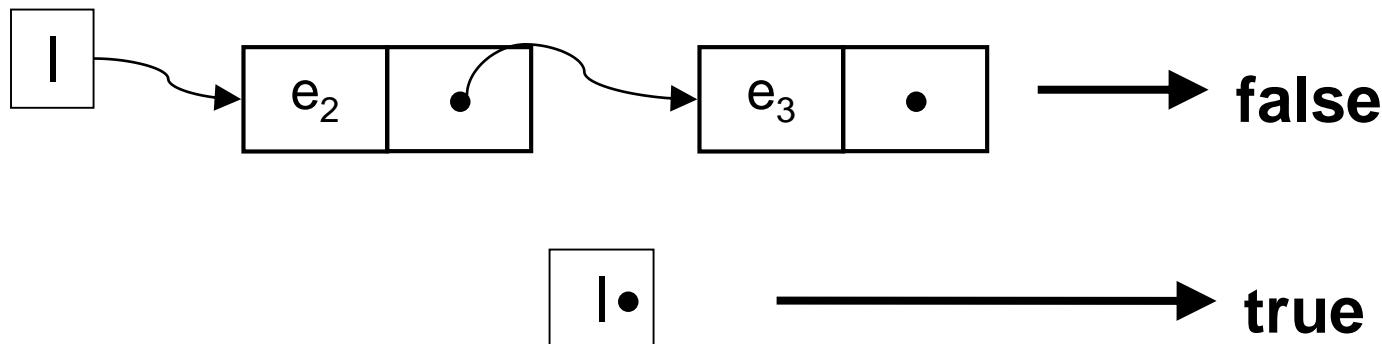


Datentypen

(einfach) Verkettete Liste

- **Funktionen**

- **Bool Isempty (list *l) /* Liste leer ? */
/* Liste bleibt unverändert */**



Datentypen

(einfach) Verkettete Liste

- **Funktionen**
 - **list* emptylist() /* Erzeugt eine leere Liste */**



Datentypen

(einfach) Verkettete Liste

- Die Datenstruktur Liste kann (wie Stack) durch ihre Eigenschaften formal vollständig beschrieben werden (Axiomatische Beschreibung)
- Die Beschreibung ergibt sich analog zu den Axiomen von Stack.

Datentypen

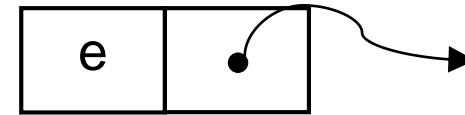
Implementierung einer einfach verketteten Liste → Datenstruktur

- **Verkettung über Pointer**

```
struct liste
{ int listenelement;
  liste *next;
};
```

Alternativen:

```
typedef struct liste
{ int listenelement;
  liste *next;
};
```



```
struct liste
{ int listenelement
  struct liste* next;
};
```

Datentypen

Implementierung einer einfach verketteten Liste →

Datenstruktur

Hinweise:

- In der angegebenen Implementierung wird mit den C++-Operatoren `new` und `delete` zur dynamischen Speicherplatzverwaltung gearbeitet, anstatt mit den C-Funktionen `malloc()` und `free()`
- Aus Gründen der Übersichtlichkeit wird bei der Anforderung von Speicherplatz auf dem Heap die Abfrage, ob die Anforderung erfolgreich war (Ergebnis ungleich `NULL`) weggelassen.

Datentypen

Implementierung einer einfach verketteten Liste → Funktionen

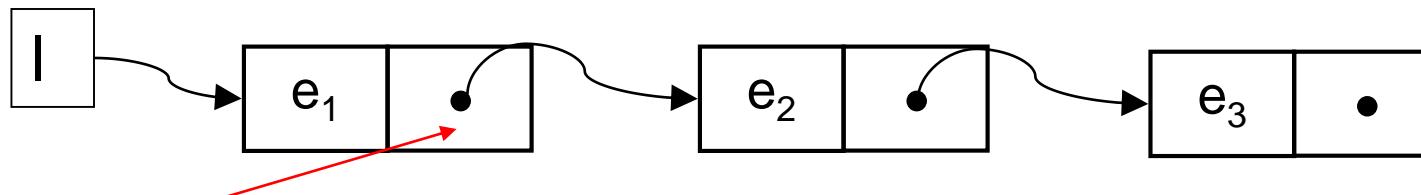
```
liste* tail(liste *listpointer)
{
    liste *hilfspointer;
    if (listpointer == NULL)
        listenderror("Fehler: tail aufgerufen mit Nullpointer!\n");
    else
        { hilfspointer = listpointer->next;
          delete listpointer; // Freigabe des Speichers
          return hilfspointer;
        };
};
```

Datentypen

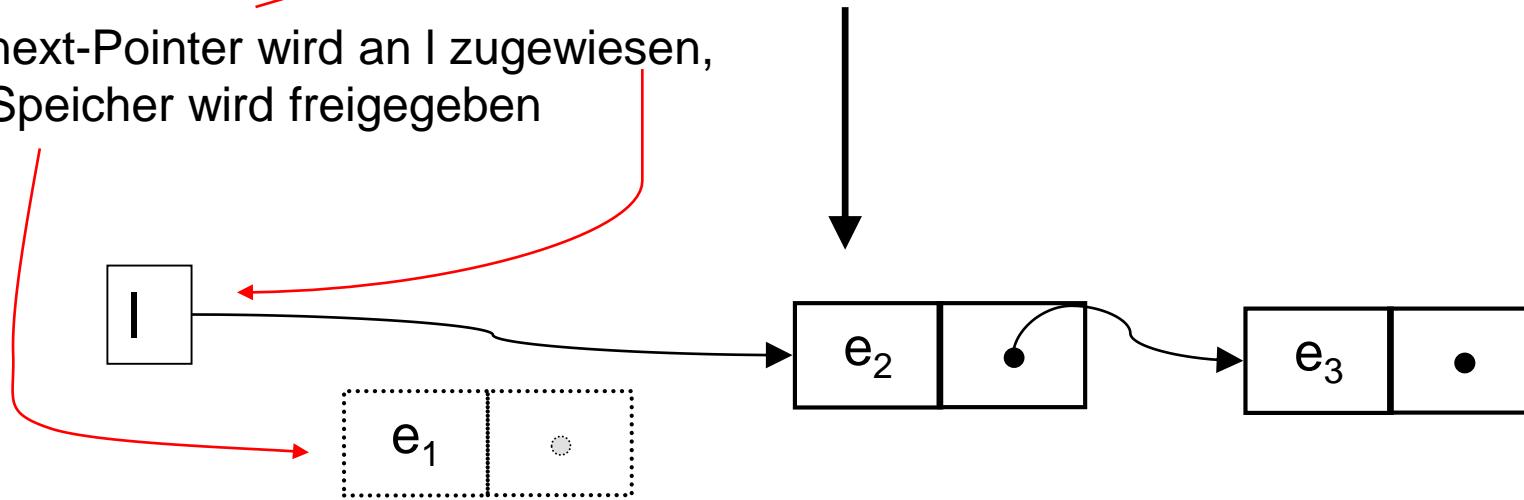
Implementierung einer einfach verketteten Liste → Funktionen

liste* tail(liste *listpointer)

...



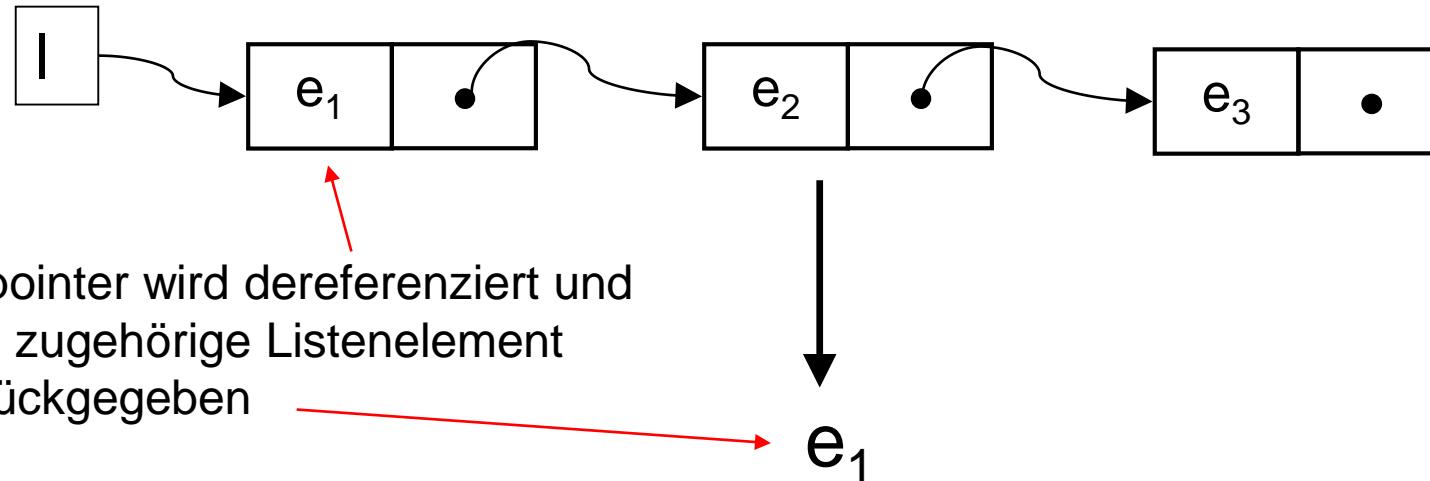
next-Pointer wird an I zugewiesen,
Speicher wird freigegeben



Datentypen

Implementierung einer einfach verketteten Liste → Funktionen

```
int head(liste *listpointer)
{ if (listpointer == NULL)
    listerror("Fehler: head aufgerufen mit Nullpointer!\n");
    else return listpointer->listenelement;
};
```

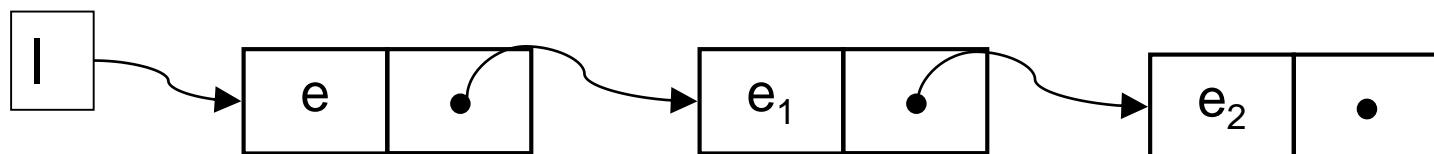
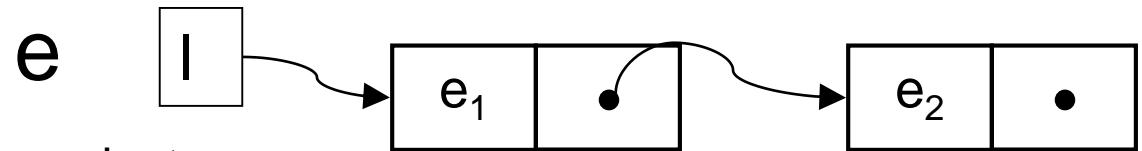


Datentypen

Implementierung einer einfach verketteten Liste → Funktionen

```
liste* append(liste *listpointer, int e)
{ liste *hilfspointer = new liste;
  hilfspointer->listenelement = e;
  hilfspointer->next = listpointer;
  return hilfspointer;
};
```

ein neues Listenelement wird angelegt für e, der next-Pointer wird auf die „alte“ Liste gesetzt und der Pointer auf das neue Listenelement zurückgegeben



Datentypen

Implementierung einer einfach verketteten Liste → Funktionen

```
liste* emptyliste() // Anlegen einer leeren Liste
{ return NULL;
};
```

```
int isempty(liste * listpointer) // Liste Leer
{ return(listpointer == NULL);
};
```

eine leere Liste besteht aus einem NULL Pointer, die Prüfung auf die leere Liste erfolgt durch einen Vergleich auf NULL

Datentypen

Listen Anwendungen

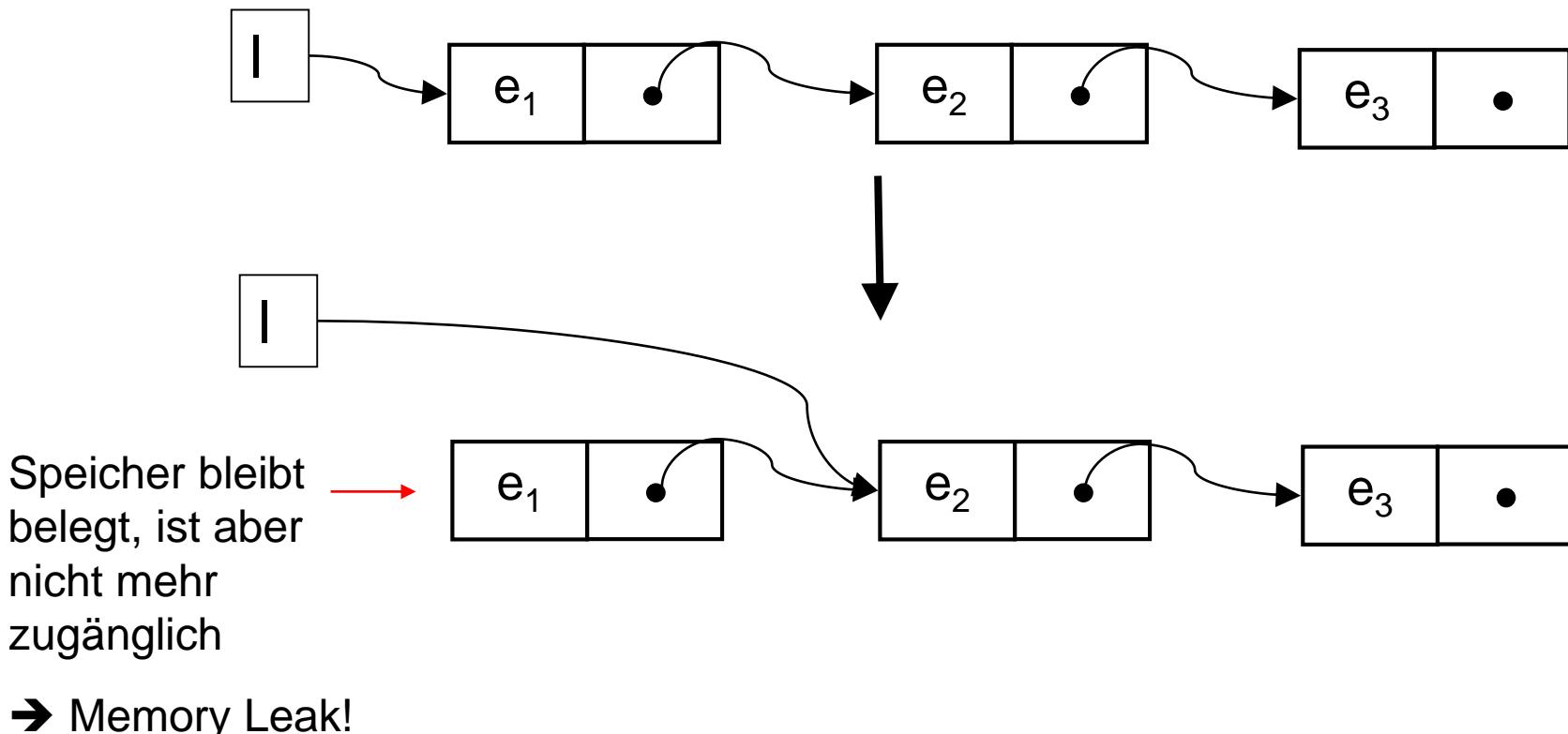
- **Implementierung von Stacks und Queues**
 - **Vorteile:**
 - **Dynamische Speicherverwaltung ermöglicht optimale Ausnutzung des Speichers**
 - **(theoretisch) keine Begrenzung der Größe**
 - **Nachteil: Zusätzlicher Speicherplatzverbrauch durch Verkettung über Pointer**
 - **Achtung: Speicherverwaltung muss sorgfältig durchgeführt werden (Memory Leaks!)*)**

*) dies gilt für alle dynamischen Datentypen, die über Pointer Speicherplatz selbst verwalten!

Datentypen

Memory Leaks („Speicherlecks“)

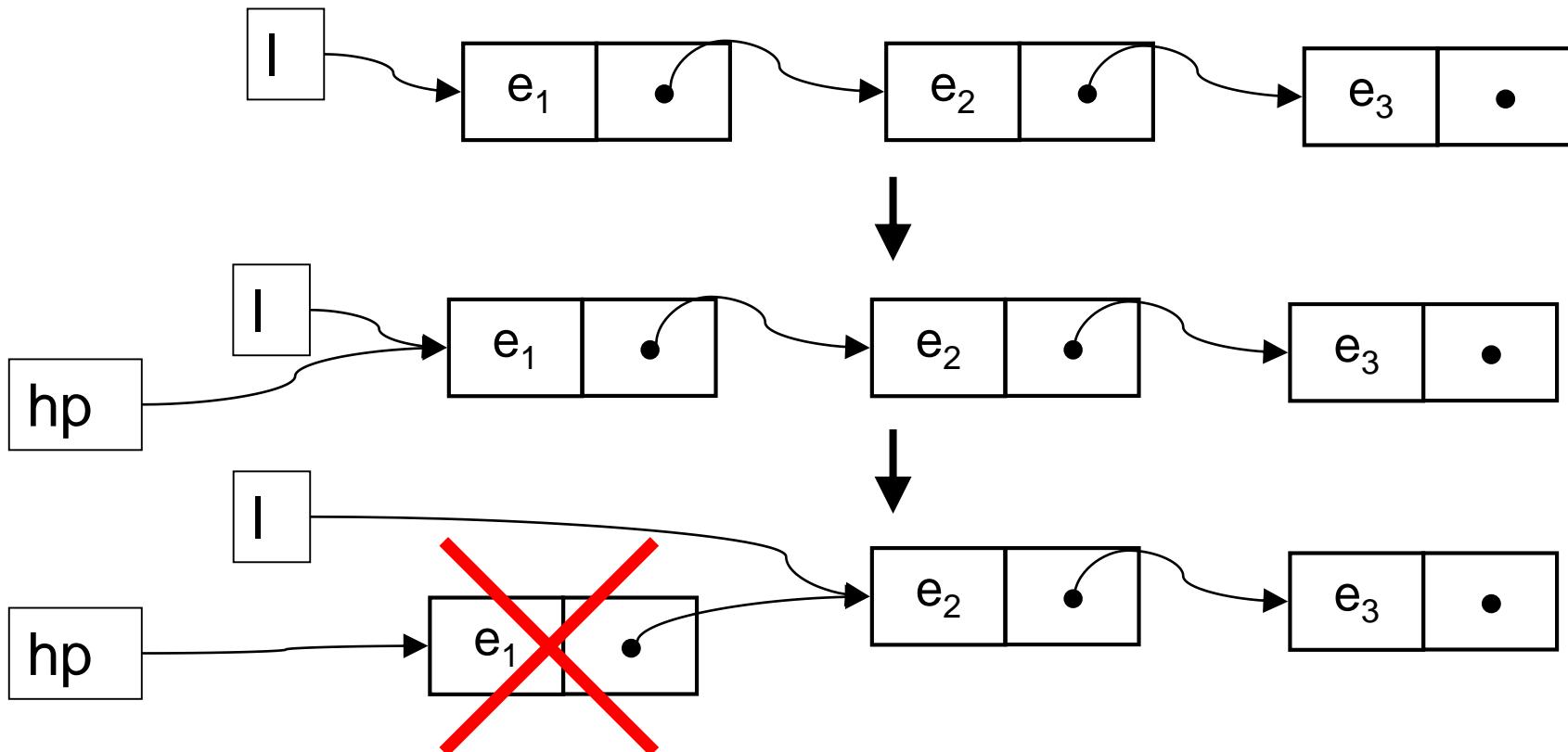
- Löschen des ersten Elements der Liste ohne Speicherfreigabe



Datentypen

Memory Leaks („Speicherlecks“)

- Löschen des ersten Elements der Liste mit Speicherfreigabe



Datentypen

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Datentypen

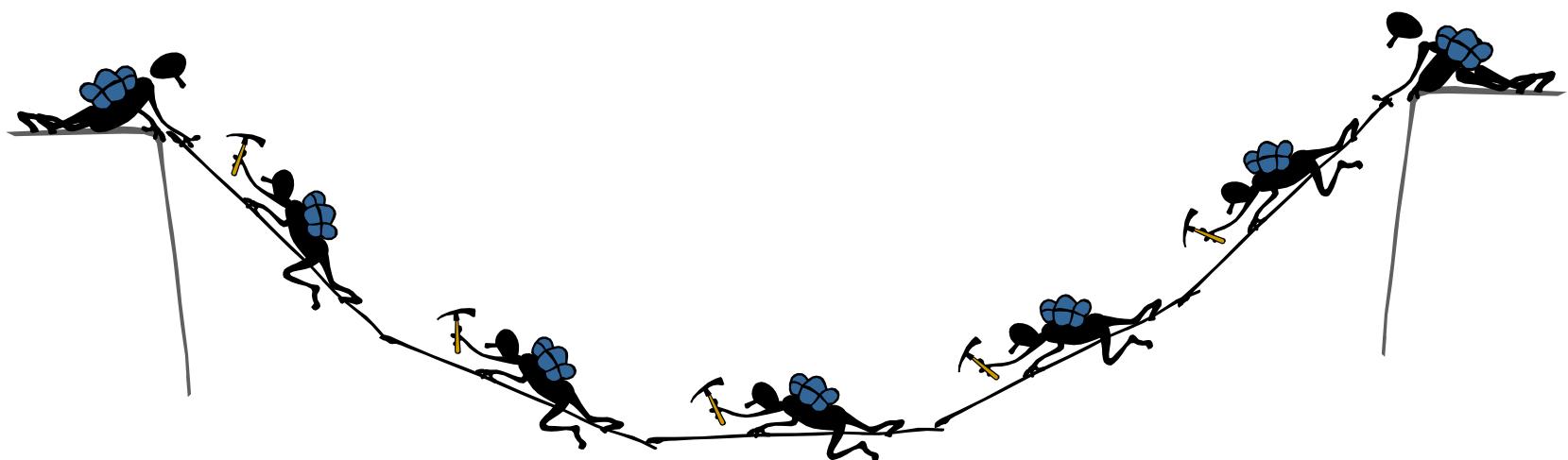
Listen Wiederholung

- **Arbeiten Sie in Gruppen (5 min)**
 - **Ermitteln Sie, was die Gruppe noch über das Thema „Listen“ weiß**
 - **Notieren Sie das Ergebnis auf einem persönlichen Zettel**
- **Suchen Sie einen Gesprächspartner aus einer anderen Gruppe und vergleichen Sie Ihr Ergebnis mit dem Ihres Partners, ergänzen Sie Ihre Aufzeichnungen ggf. (5 min)**
- **Wechseln Sie den Gesprächspartner und vergleichen Sie Ihr Ergebnis erneut.**

Datentypen

Doppelt verkettete Liste (List)

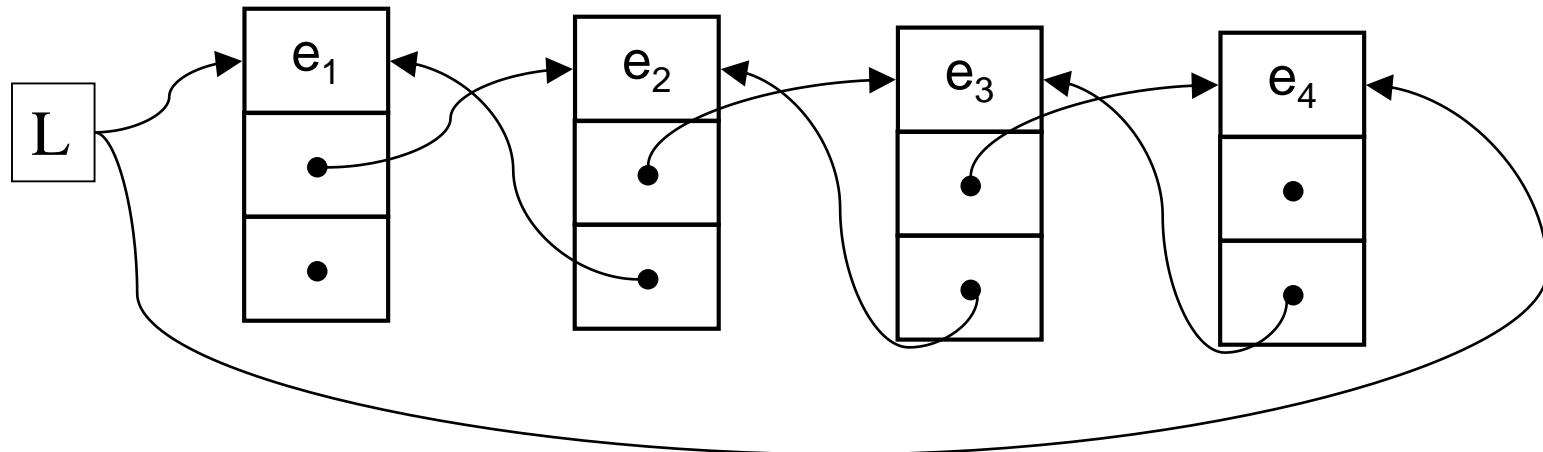
- Zugriff „am hinteren“ und am „vorderen Ende“
 - Anfügen und Wegnehmen an beiden Enden
 - kein direkter Zugriff auf Elemente, die zwischen dem ersten und letzten Element liegen



Datentypen

Doppelt verkettete Liste (List)

- Beispiel: Liste mit 4 Elementen



- ohne Pfeil steht für leeren Pointer (in C NULL)

Datentypen

Doppelt verkettete Liste (List)

- Spezieller Listenkopf zeigt auf den Anfang und das Ende der Liste.
- In jedem Listenelement zeigt ein Pointer auf das nächste Element (oder NULL) und auf das vorhergehende (oder NULL)
- vereinfacht den Zugriff
- ermöglicht die Implementierung von zweiköpfigen Queues

Datentypen

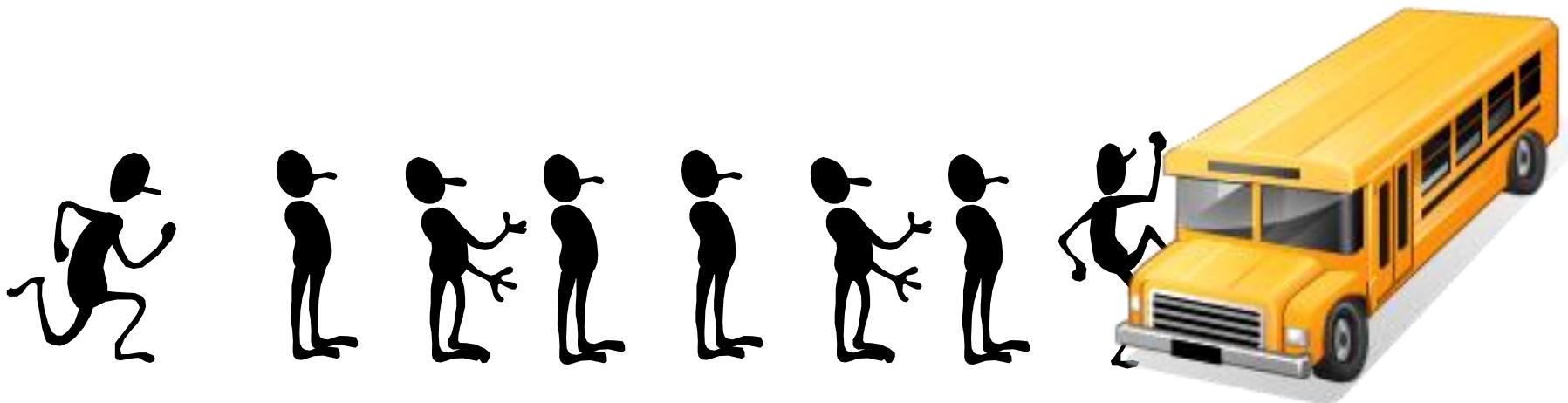
Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Datentypen

Queue (Schlange)

- Zugriff „an beiden Enden“
 - Anfügen nur hinten an der Schlange
 - Wegnehmen nur vorne
 - kein direkter Zugriff auf Elemente, die zwischen dem ersten und letzten Element liegen

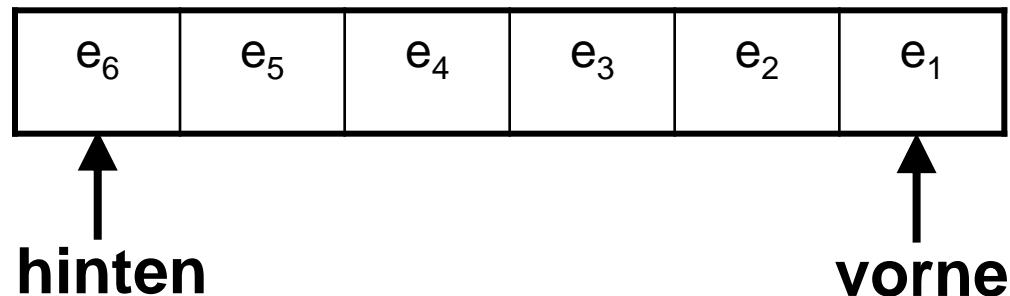


Datentypen

Queue

- **Definition**
Eine Queue (über einem bereits existierenden Datentyp T) besteht aus einer Folge von Elementen (vom Typ T), die nur an einem Ende („vorne“) gelesen bzw. gelöscht werden kann und am anderen Ende („hinten“) ergänzt.

- **Beispiel: Queue mit 6 Elementen**

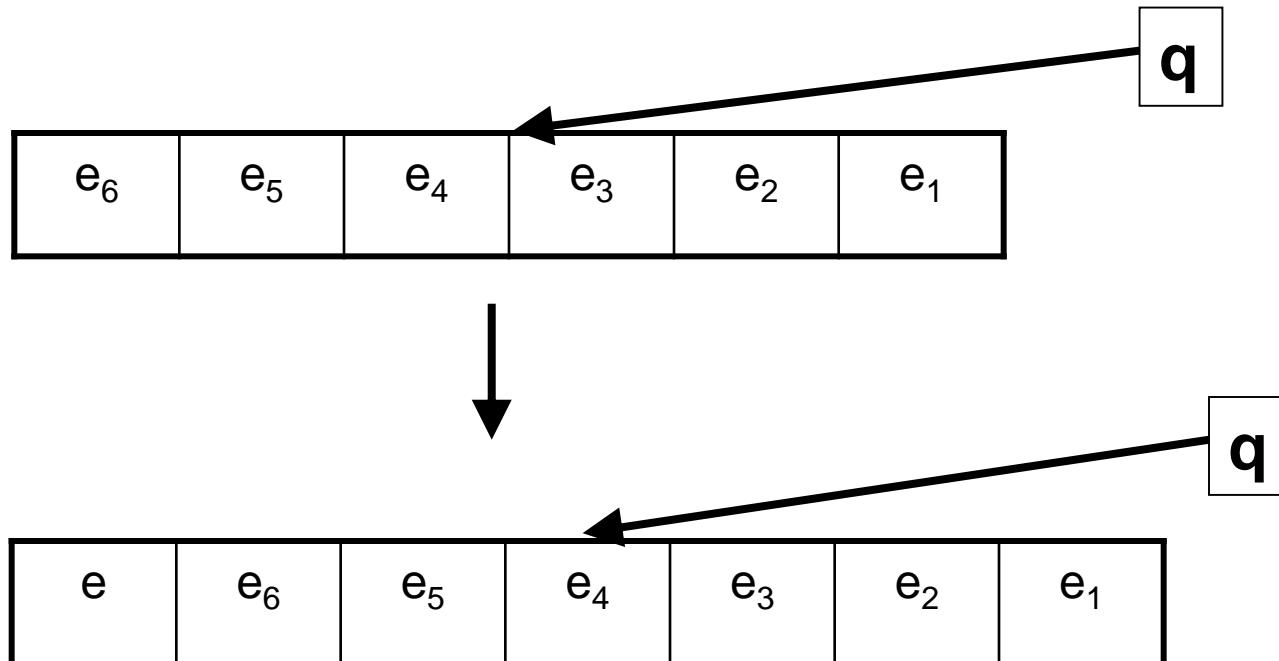


Datentypen

Queue

- **Funktionen**

- **queue* Append (queue *q, T e)**
/* fügt hinten ein Element an */

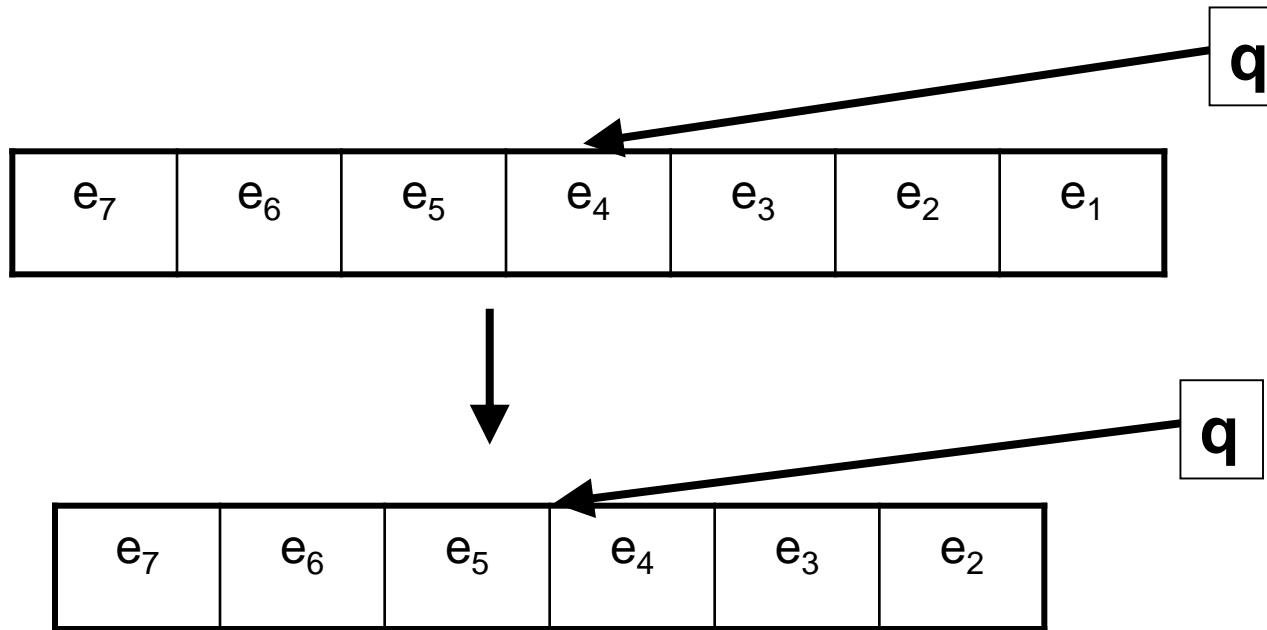


Datentypen

Queue

- **Funktionen**

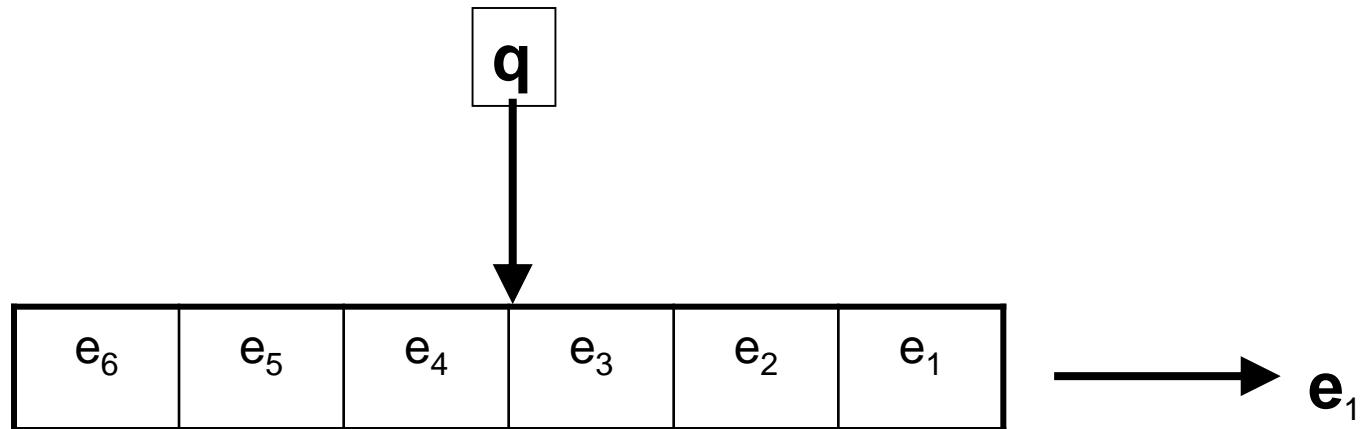
- **queue* Rest (queue *q) /* löschte erstes Element */**
- **/* Voraussetzung: Queue ist nicht leer! */**



Datentypen

Queue

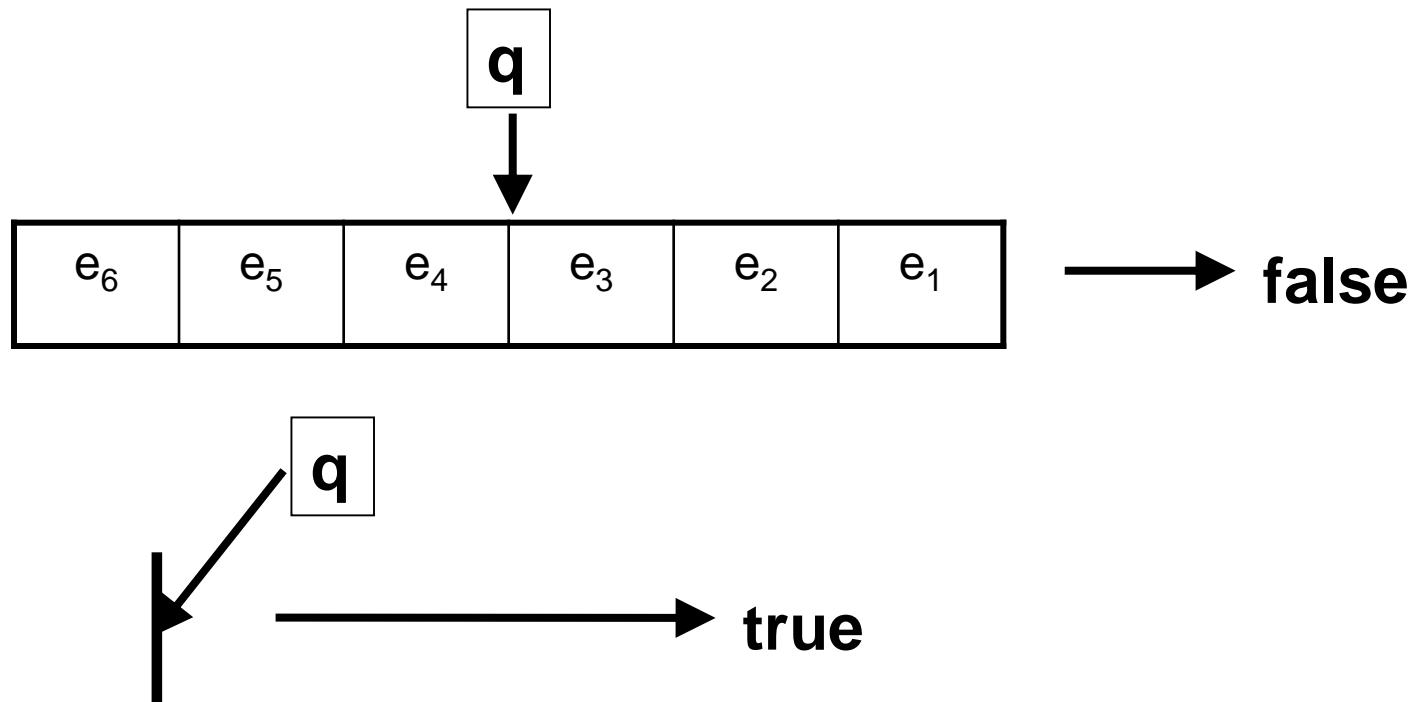
- **Funktionen**
 - **T Top (queue *q) /* liefert erstes Element */
/* Queue bleibt unverändert */
/* Voraussetzung: Queue ist nicht leer! */**



Datentypen

Queue

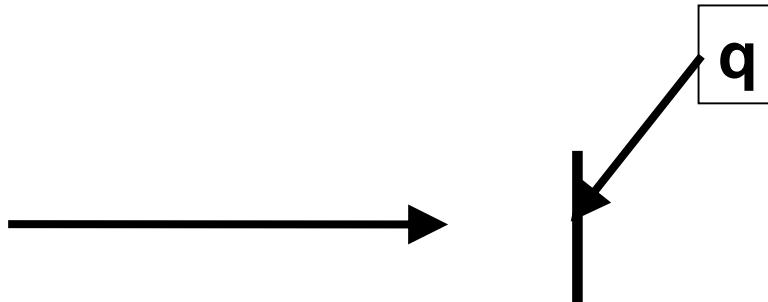
- **Funktionen**
 - **Bool Isempty (queue * q) /* Queue leer ? */
/* Queue bleibt unverändert */**



Datentypen

Queue

- **Funktionen**
 - **queue* emptyqueue() /* Erzeugt eine leere Queue */**



Datentypen

Queue

- **Die Datenstruktur Queue kann (wie Stack) durch ihre Eigenschaften formal vollständig beschrieben werden (Axiomatische Beschreibung)**

Datentypen

Queue

- **Eigenschaften (Axiome)**
 - **isempty(emptyqueue()) = true**
 - **rest append(q,e) = emptyqueue(), falls isempty(q)
append(rest(q),e), sonst**
 - **top append(q,e) = e, falls isempty(q)
top(q), sonst**
 - **isempty.append(q,e)) = false**
 - **top(emptyqueue()) → Error**
 - **rest(emptyqueue()) → Error**

Übung

Wiederholung Rekursion

- **Bearbeiten Sie in Gruppen folgende Themen. Präsentieren Sie Ihr Ergebnis an der Tafel**
 - Was ist Rekursion, welche Bedeutung hat Rekursion in der Programmierung (Gruppe 1)
 - Geben Sie Beispiele rekursiv zu lösender Probleme (Gruppe 2)



Übung

Wiederholung Rekursion

- Bearbeiten Sie in Gruppen folgende Themen. Präsentieren Sie Ihr Ergebnis an der Tafel

- Die Fakultät einer positiven ganzen Zahl lässt sich als rekursive C-Funktion darstellen. Erläutern Sie diese Funktion und erklären Sie den Aufruf der Fakultät für die Zahl 4. (Gruppe 3)
- Die Fibonacci-Zahl $\text{fib}(x)$ für eine natürliche Zahl x ist folgendermaßen rekursiv definiert:

$$\text{fib}(0) = 0, \text{fib}(1) = 1, \text{fib}(x, x>1) = \text{fib}(x-1)+\text{fib}(x-2)$$

Definieren Sie eine C-Funktion, die die Fibonacci-Zahl für einen Parameter berechnet. Erläutern Sie die Funktion und erklären Sie den Aufruf $\text{fib}(3)$ (Gruppe 4)



Übung

Wiederholung Rekursion

- **Bearbeiten Sie in Gruppen folgende Themen. Präsentieren Sie Ihr Ergebnis an der Tafel**
 - In einer verketteten Liste lässt sich rekursiv ein Element löschen. Geben Sie eine C-Funktion `liste* loesche(liste* l, int e)` an, die das Element e aus einer Liste löscht (falls vorhanden), auf die l zeigt. `loesche` soll dabei nur die bekannten Funktionen der Liste (`head`, `tail`, `append`, ...) verwenden. Erläutern Sie die Funktion und erklären Sie den Aufruf an einem Beispiel (Gruppe 5)



Übung

Implementierung einer Queue in C

Beschreibung:

Die Datenstruktur Queue wird definiert durch ihr Signatur (Schnittstellen der Funktionen) und die dazugehörenden Eigenschaften (Axiome).

Aufgabe:

Entwerfen und Implementieren Sie eine Queue mit den beschriebenen Funktionen basierend auf einem Array oder einer verketteten Liste. Implementieren Sie zusätzlich eine Funktion, die interaktiv das Testen der Queuefunktionen erlaubt.

Zeit:



Datentypen

Queue Anwendungen

- **Verwaltung von Betriebsmitteln (z.B. Drucker, Prozessor) in Betriebssystemen:**
 - Prozesse, die auf ein Betriebsmittel warten werden in eine Warteschlange eingehängt
 - Verschiedene Warteschlangen, z.B. individuell für jedes Betriebsmittel oder auch prioritätsgesteuert.
- **FIFO (First-in-First-Out) Speicher**

Datentypen

Zum Schluss dieses Abschnitts ...

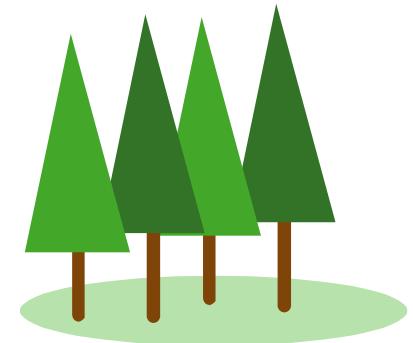
Noch Fragen ??

Datentypen

Bäume

- **Definitionen:**

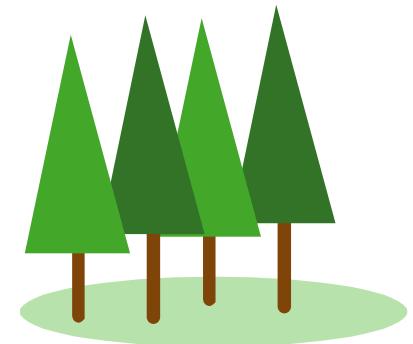
Ein Baum besteht aus einer endlichen Menge von Knoten K und einer endlichen Menge von gerichteten Kanten P (dargestellt als Pfeil) zwischen Knoten aus K. Es gibt maximal eine Kante von einem Knoten zu einem anderen. Auf jeden Knoten zeigt genau eine Kante (Ausnahme Wurzelknoten)



Datentypen

Bäume

- **Definitionen (Fortsetzung)::**
Ein Baum hat einen ausgezeichneten Knoten, die Wurzel, die nicht Endknoten einer Kante ist.

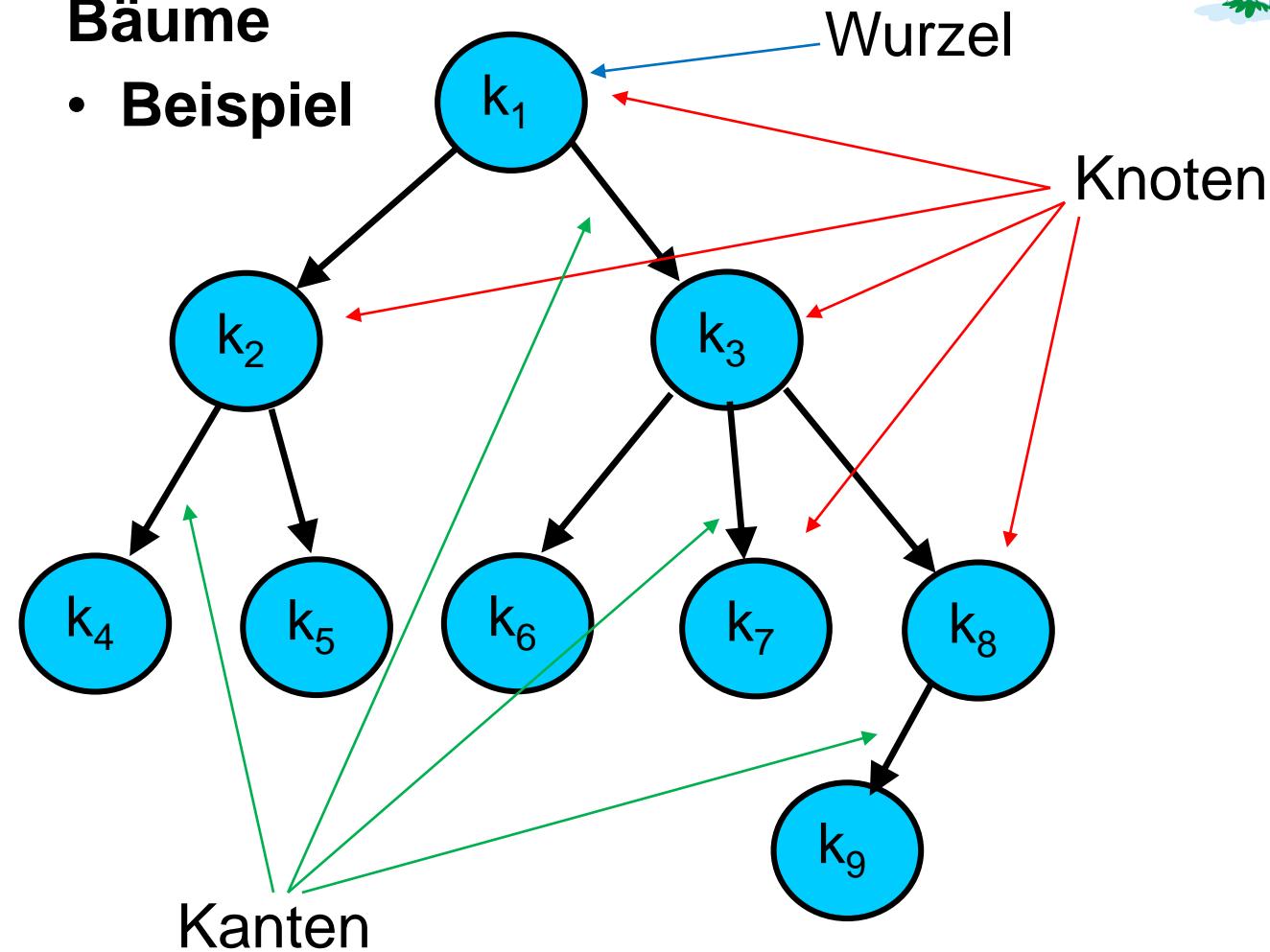




Datentypen

Bäume

- Beispiel



Datentypen

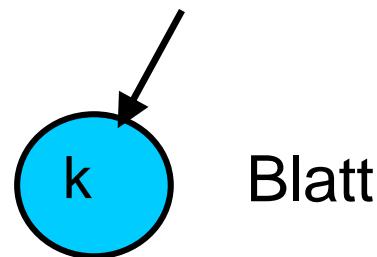
Bäume

- **Definition (Fortsetzung):**
Gibt es eine Kante von einem Knoten k_1 zu einem Knoten k_2 , dann ist k_1 der Elternknoten von k_2 . k_2 ist Kind von k_1 .
Ein Knoten k_2 ist erreichbar von einem Knoten k_1 , wenn es eine Folge von Knoten $k_1, k_x, \dots, k_{x+n}, k_2$ gibt, so dass k_1 Elternknoten von k_x ist, k_{x+i} Elternknoten von K_{x+i+1} für alle i von 0 bis $(n-1)$ und k_{x+n} ist Elternknoten von K_2 .

Datentypen

Bäume

- **Definition (Fortsetzung):**
Ist k_1 Elternknoten von k_2 , so ist k_2 von k_1 aus auch erreichbar.
Knoten, die keine Kinder haben, heißen Blatt



Datentypen

Bäume

- **Definition (Fortsetzung):**
Ist ein Knoten l erreichbar von einem Knoten k, so ist k Vorgänger von l und l Nachfolger von k.
Jeder Knoten ist auf nur genau eine Weise von der Wurzel aus erreichbar.
- **Der Grad g(k) eines Knotens k ist die Anzahl seiner Kinder**

Datentypen

Bäume

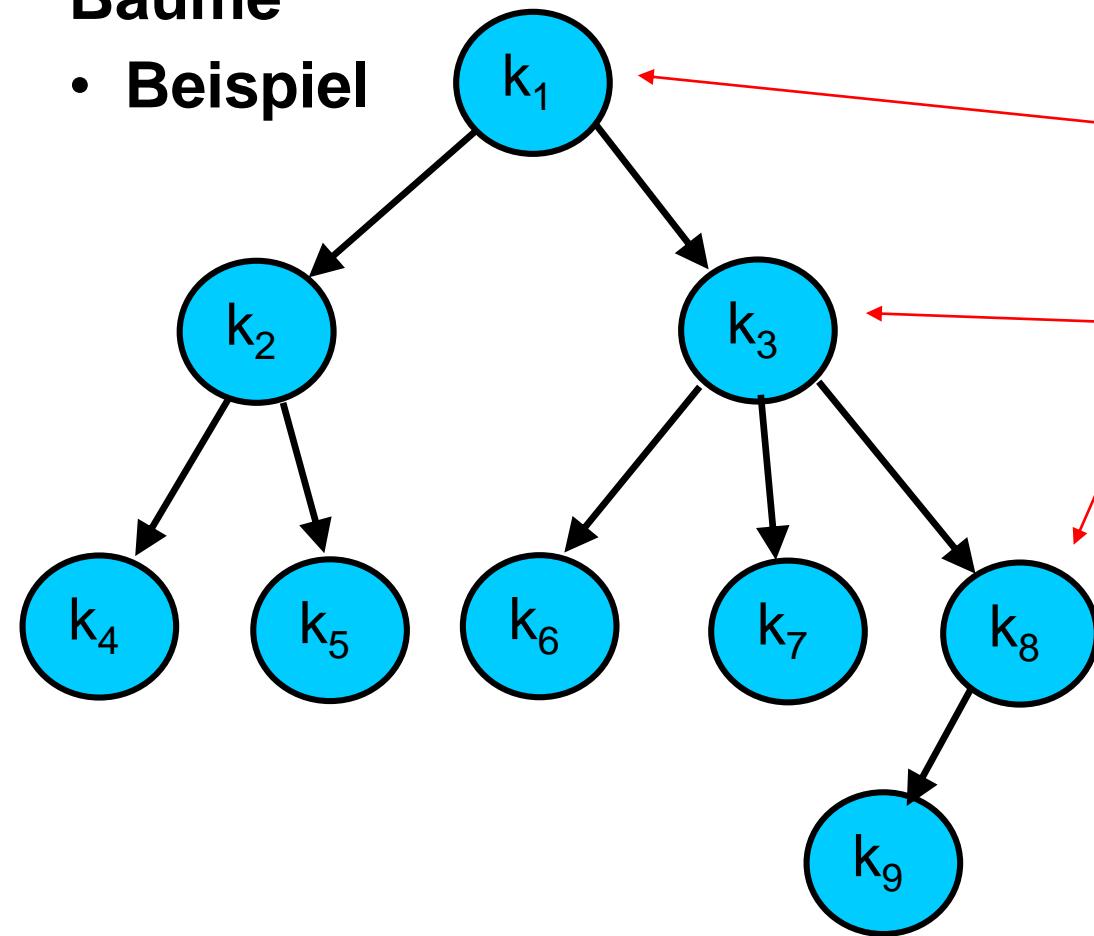
- Die Tiefe (oder Höhe) $t(k)$ eines Knotens k ist definiert als
 - 0, falls k die Wurzel des Baums ist
 - $1 + t(k^*)$, wenn k^* Elternknoten von k ist.



Datentypen

Bäume

- **Beispiel**



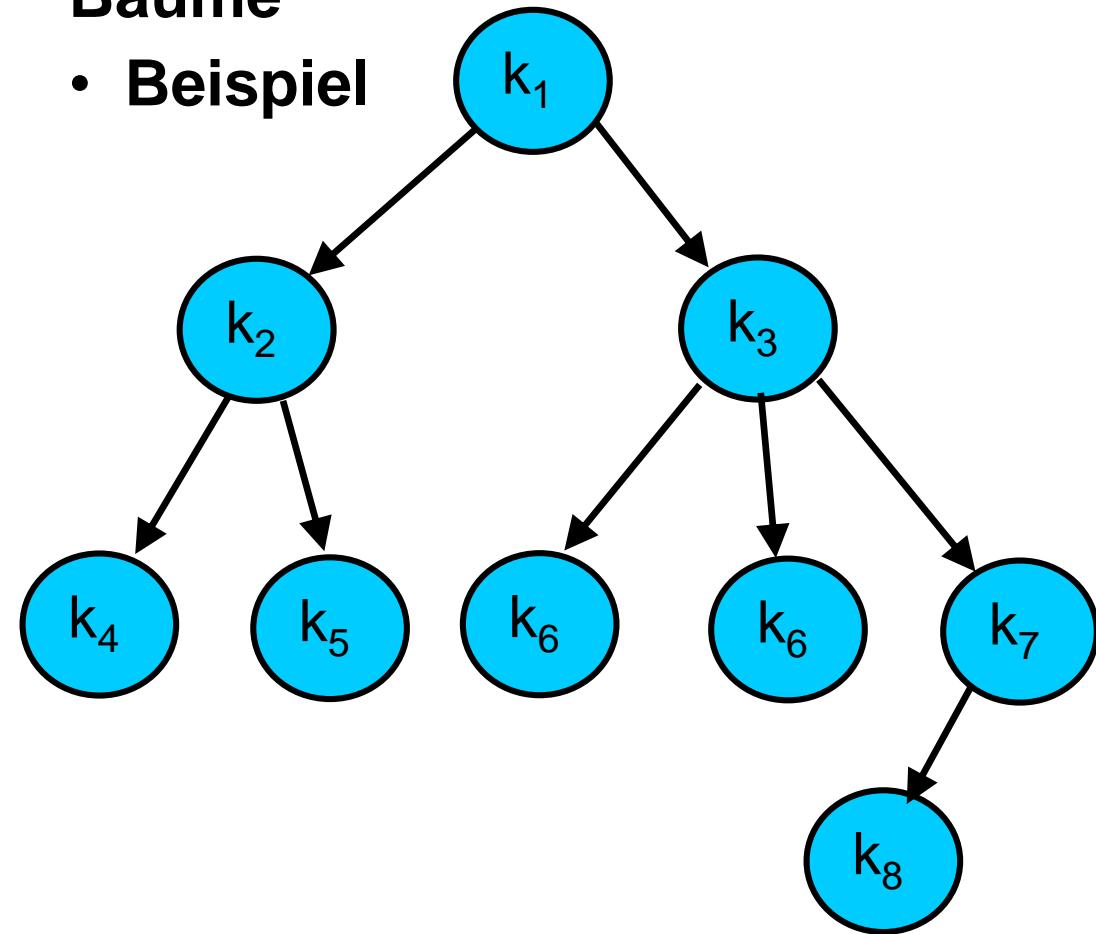
- k₁ ist die Wurzel des Baums, k₂ und k₃ sind Kinder von k₁. k₁ ist vom Grad 2
- k₉ ist erreichbar von k₃ (über k₈)
- k₈ ist nicht erreichbar von k₂
- t(k₇) = 2
- k₄, k₅, k₆, k₇, k₉ sind Blätter



Datentypen

Bäume

- **Beispiel**

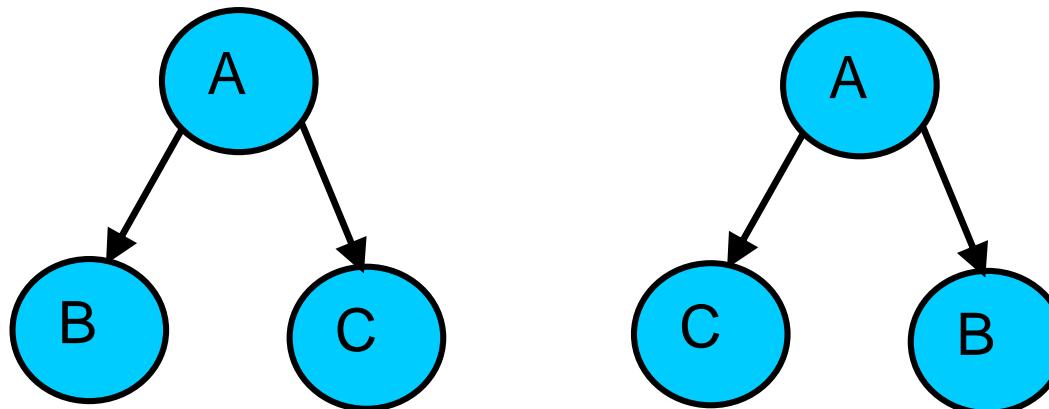


- Bäume werden (meist) grafisch dargestellt
- In der Software wachsen die Bäume nicht in den Himmel, sondern meist von der Wurzel nach unten!

Datentypen

Bäume

- **Definition (Fortsetzung):**
Ein Baum heißt geordnet, wenn die Reihenfolge der Verzweigungen in einem Knoten festgelegt ist. Ist dies nicht der Fall heißt der Baum ungeordnet

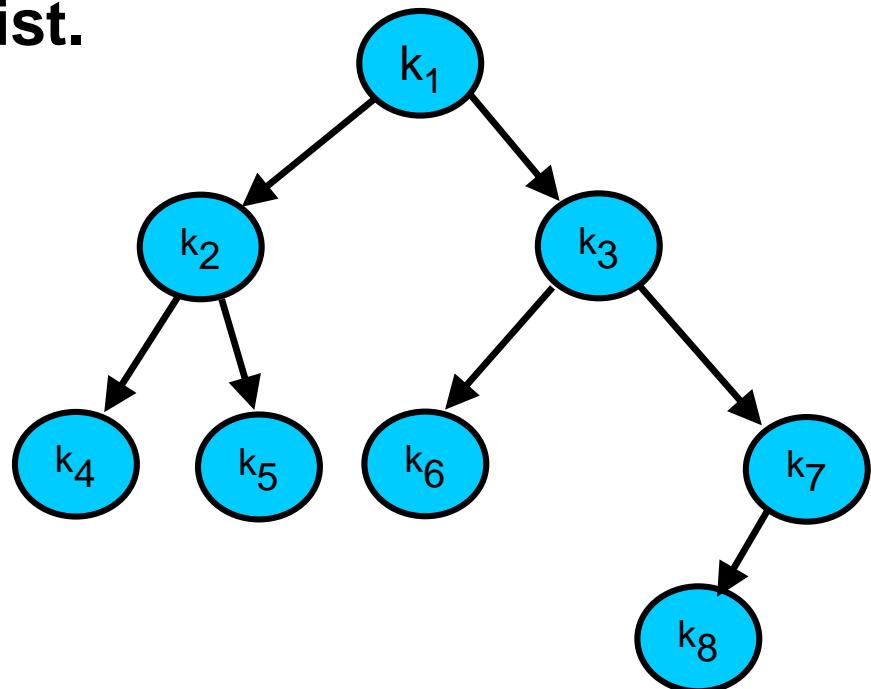


→ in diesem Sinn sind die beiden Bäume gleich als ungeordnete und verschieden als geordnete Bäume

Datentypen

Bäume

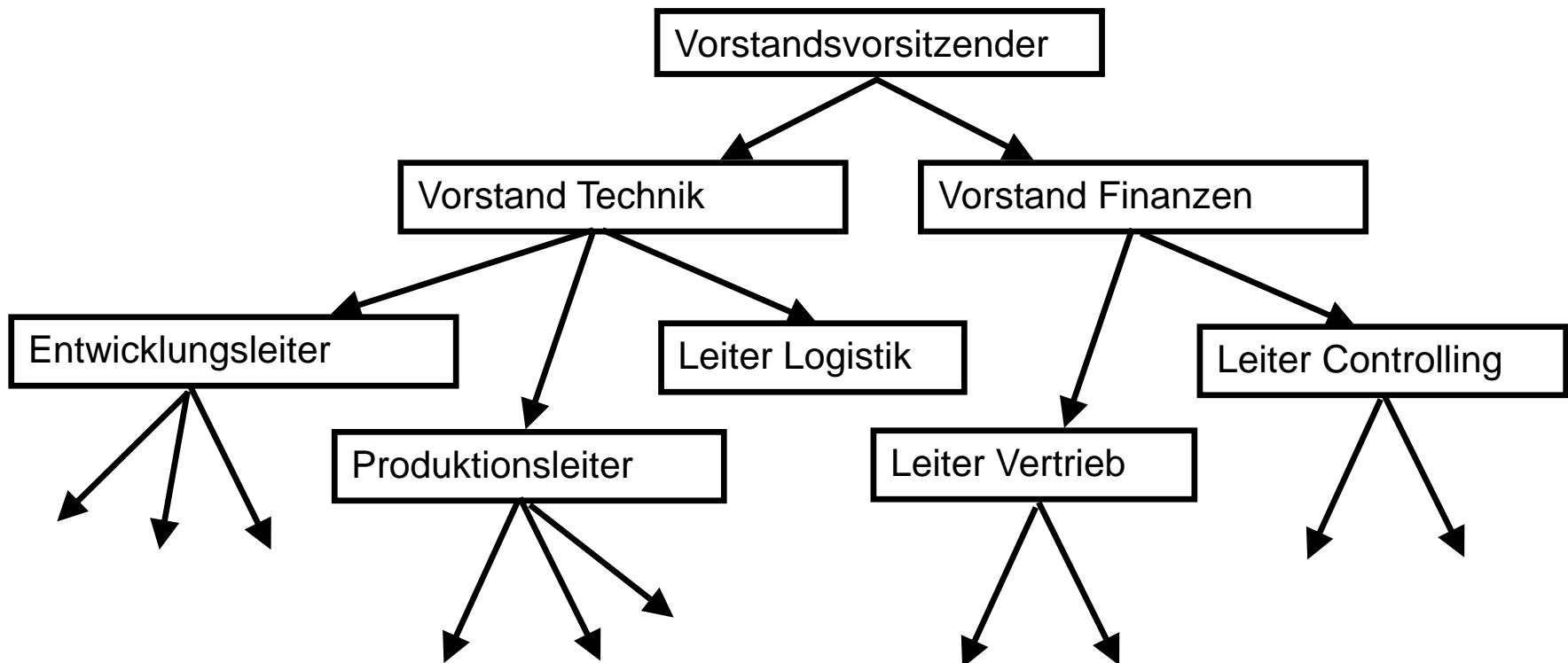
- **Definition (Fortsetzung):**
Ein geordneter Baum heißt binär, wenn er leer ist oder der Grad aller seiner Knoten kleiner oder gleich 2 ist.



Datentypen

Bäume

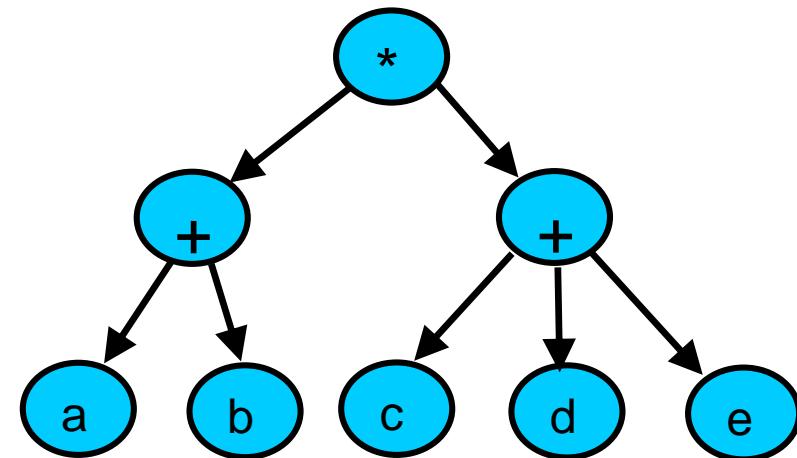
- Anwendungen
 - Darstellung von Unternehmensstrukturen



Datentypen

Bäume

- Anwendungen
 - Darstellung von Inhaltsverzeichnissen in Kapitel, Unterkapitel, Abschnitte
 - Darstellung von statischen Programmstrukturen (Funktionen, Blöcke, Anweisungen) → Compilerbau
 - Darstellung mathematischer Ausdrücke, z.B. $(a+b)*(c+d+e)$



Datentypen

Bäume

- **Anwendungen (Fortsetzung)**
 - **Darstellung von Codes**
 - **Stammbäume**
 - **Suchen**
 - **Sortieren**



Datentypen

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Datentypen

Binäre Bäume

- **Eigenschaften (Satz 1):**
 - **Sei T ein nichtleerer binärer Baum mit Höhe h . Dann gilt:**
 1. **Für $0 \leq i \leq h$ gilt, dass T maximal 2^i Knoten der Tiefe i besitzt.**
 2. **T besitzt minimal $h + 1$ und maximal $2^{h+1} - 1$ Knoten.**
 3. **Für die Anzahl n der Knoten in T gilt $\log(n + 1) - 1 \leq h \leq n - 1$.**



Datentypen

Binäre Bäume

- **Eigenschaften (Satz 1 → Beweis):**
 1. **Vollständige Induktion nach i:**
 - i=0: T besitzt genau einen Knoten der Tiefe 0 ($=2^0$), nämlich die Wurzel.**
 - i, $i < h \rightarrow i+1$: T besitzt maximal 2^i Knoten der Tiefe i. Jeder dieser Knoten kann maximal 2 Kinder haben \rightarrow es gibt maximal $2 * 2^i = 2^{i+1}$ Knoten der Tiefe i+1**

Datentypen

Binäre Bäume

- **Eigenschaften (Satz 1 → Beweis):**
 2. zum einen gilt $n \geq h+1$ (für die Höhe $h+1$ sind mindestens $h+1$ Knoten nötig.
zum anderen gilt:

$$n = \sum_{i=0}^h \text{Anzahl der Knoten mit Tiefe } i \leq 2^i$$

$$\sum_{i=0}^h 2^i = 2^{h+1}-1 \quad \text{qed.}$$

Datentypen

Binäre Bäume

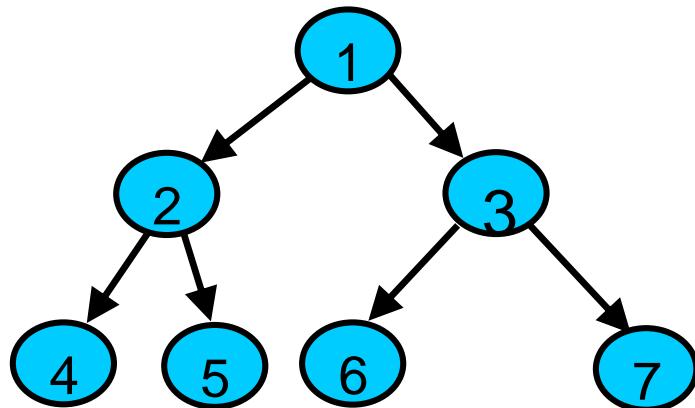
- **Eigenschaften (Satz 1 → Beweis):**
 3. Nach 2. gilt $h \leq n - 1$. Andererseits gilt nach 2. auch $n \leq 2^{h+1} - 1$, d.h. $\log(n+1) \leq h+1$. qed.

Datentypen

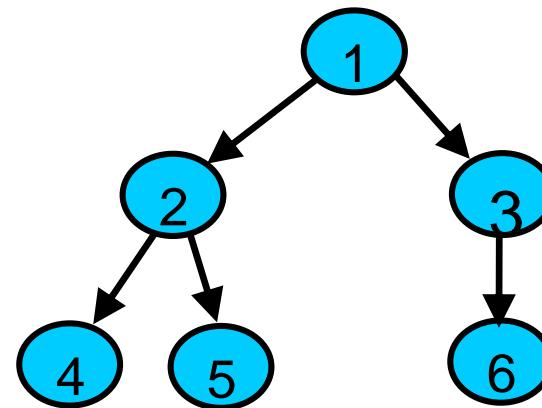
Bäume

- **Definition (Fortsetzung):**
Ein nichtleerer binärer Baum mit Höhe h heißt voll, wenn er $2^{h+1} - 1$ Knoten besitzt.

voller binärer Baum



nicht voller binärer Baum



Datentypen

Bäume

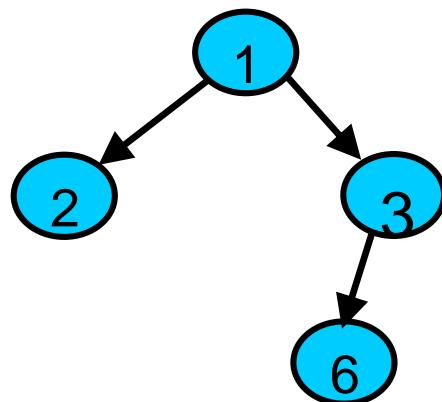
- **Definition (Fortsetzung):**

Sei T_1 ein binärer Baum der Höhe h mit $n > 0$ Knoten. T_2 sei ein voller binärer Baum ebenfalls mit Höhe h . In T_2 seien die Knoten stufenweise von links nach rechts durchnummert (siehe vorheriges Beispiel). T_1 heißt dann vollständig, wenn er aus T_2 durch Wegstreichen der Knoten der Nummern $n+1, n+2, \dots, 2^{h+1}-1$ erzeugt werden kann.

Datentypen

Bäume

- **Definition (Fortsetzung):**
Sei T_1 ein (binärer) Baum. T_1 heißt ausgeglichen, wenn in jedem Knoten die Höhen der Teilbäume sich um höchstens 1 voneinander unterscheiden*).



ausgeglichener binärer Baum

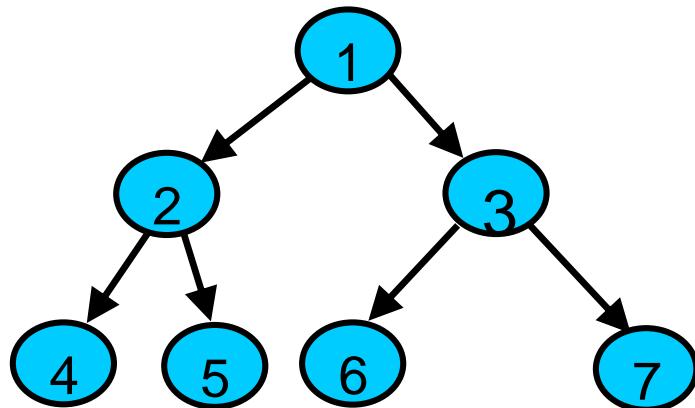
*)diese Bäume werden auch AVL-Bäume bezeichnet nach ihren Erfindern Adelson-Velski und Landis

Datentypen

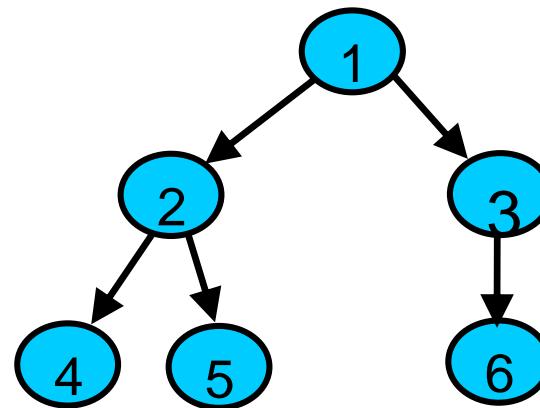
Bäume

- Beispiel
Voller und davon abgeleiterter vollständiger Baum

voller binärer Baum



nicht voller binärer Baum



Datentypen

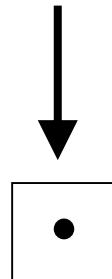
Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Datentypen

Binärbäume

- **Funktionen**
 - **b_tree* emptytree () /* erzeugt leeren Baum */**

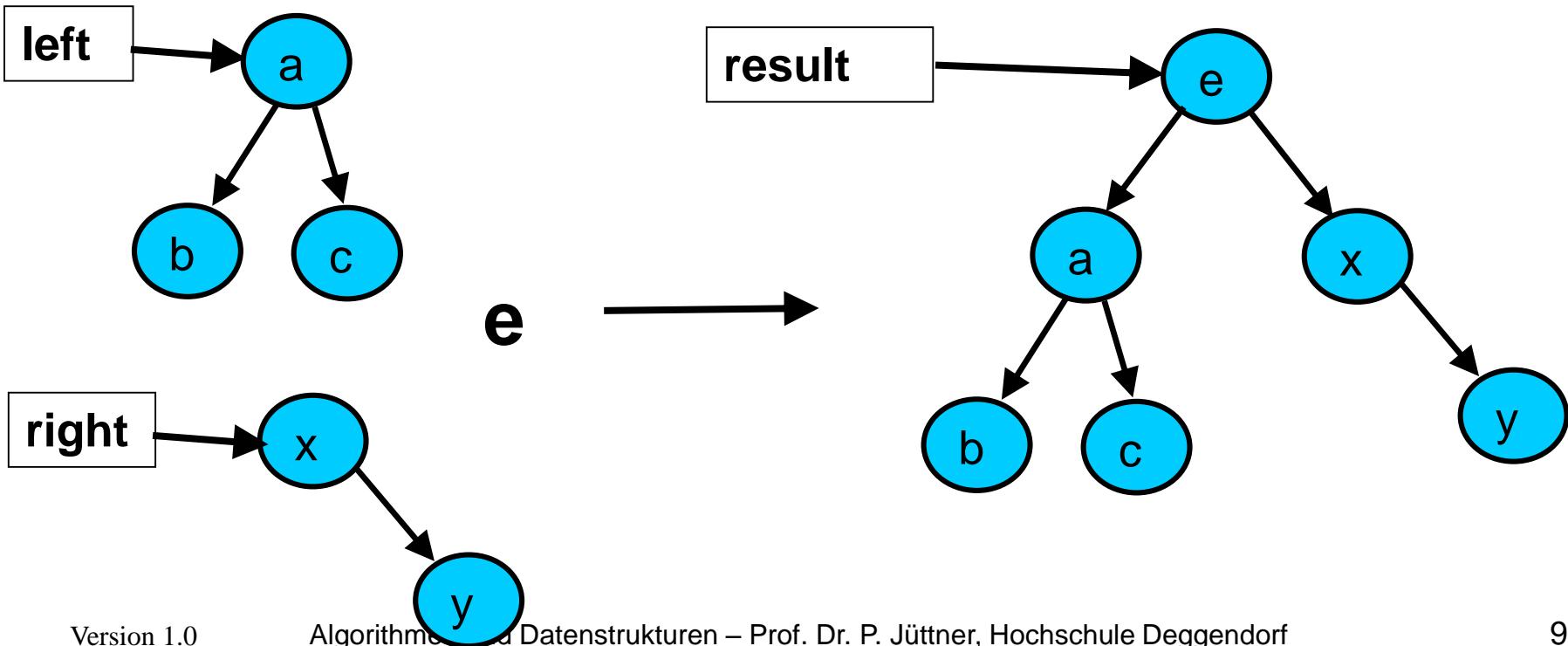


Datentypen

Binärbäume

- **Funktionen**

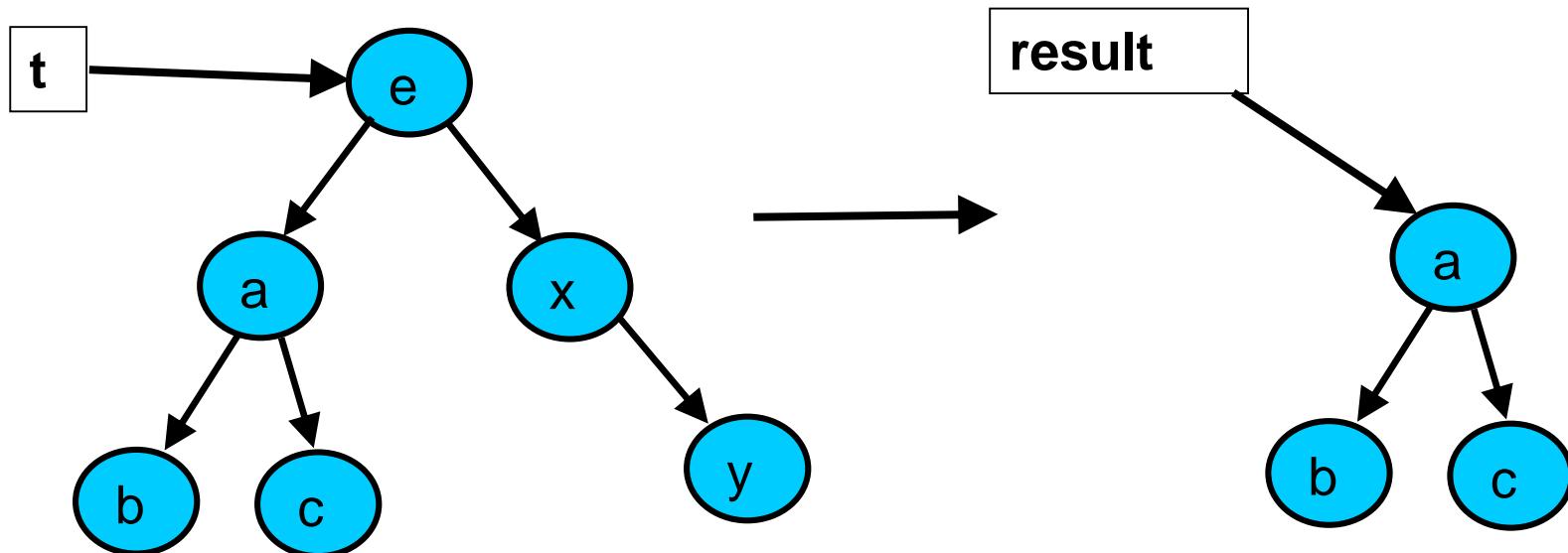
- **b_tree* b_tree_new (b_tree *left, T e, b_tree *right)**
/* erzeugt aus zwei Teilbäumen und einem Knotenelement einen neuen Baum */



Datentypen

Binärbäume

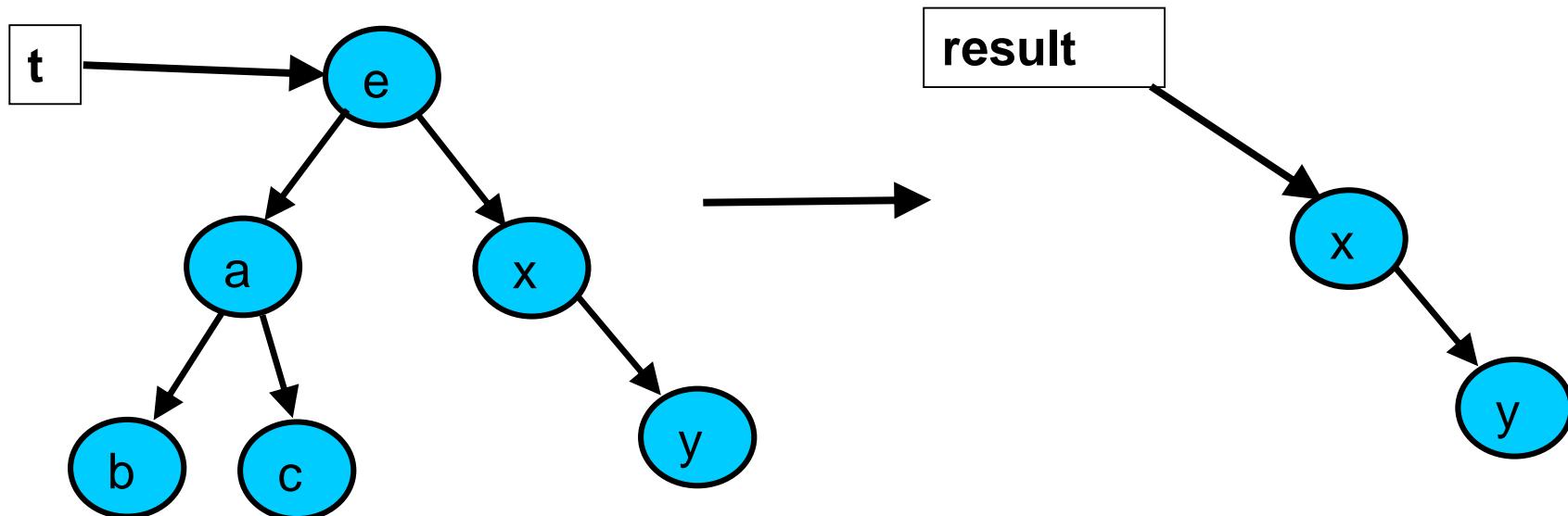
- **Funktionen**
 - **b_tree* left (b_tree *t)**
/* gibt den linken Teilbaum zurück */



Datentypen

Binärbäume

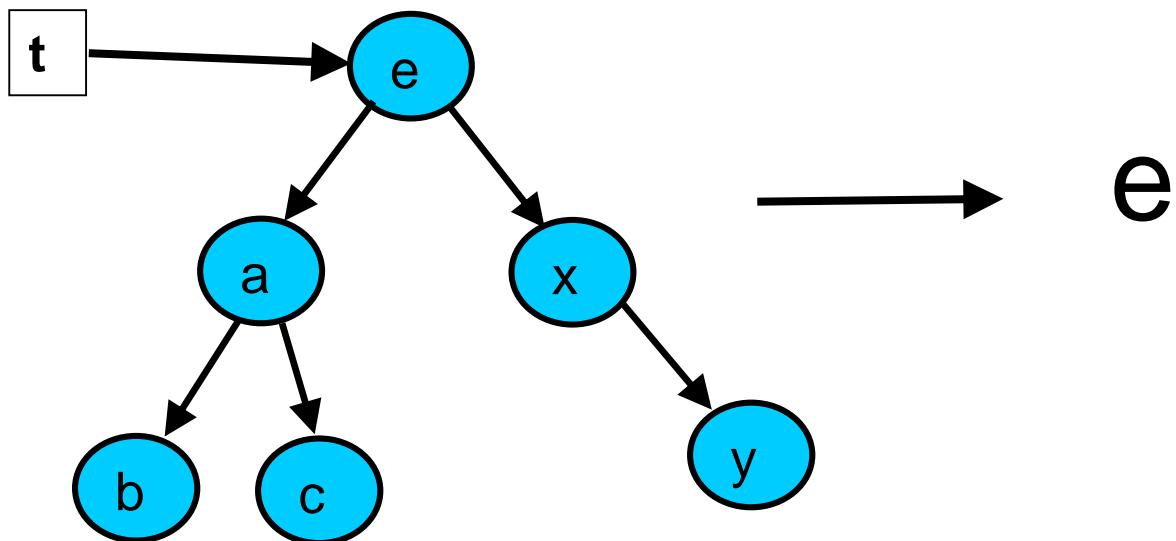
- **Funktionen**
 - **b_tree* right (b_tree *t)**
/* gibt den rechten Teilbaum zurück */



Datentypen

Binärbäume

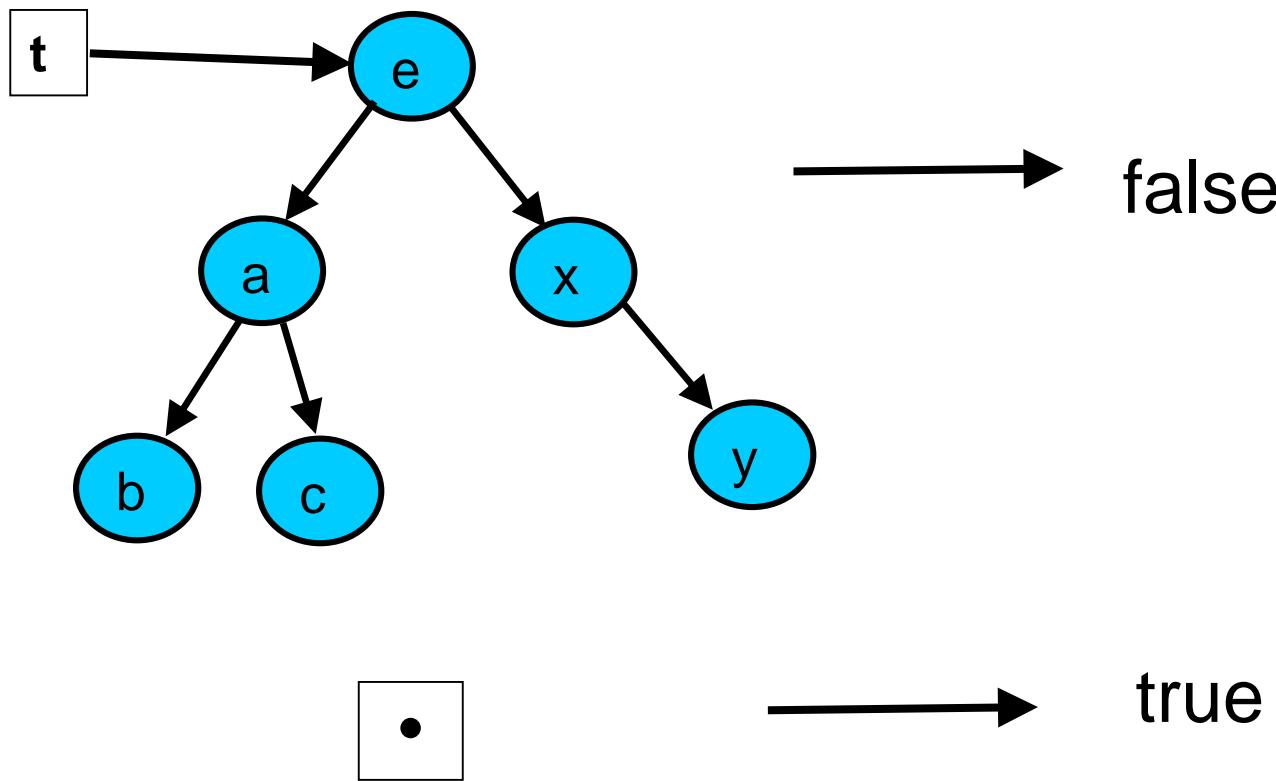
- **Funktionen**
 - T element (b_tree *t)
/* gibt den Inhalt der Wurzel zurück */



Datentypen

Binärbäume

- **Funktionen**
 - **bool ismempty (b_tree *t) /* Baum leer ? */**



Datentypen

Zum Schluss dieses Abschnitts ...

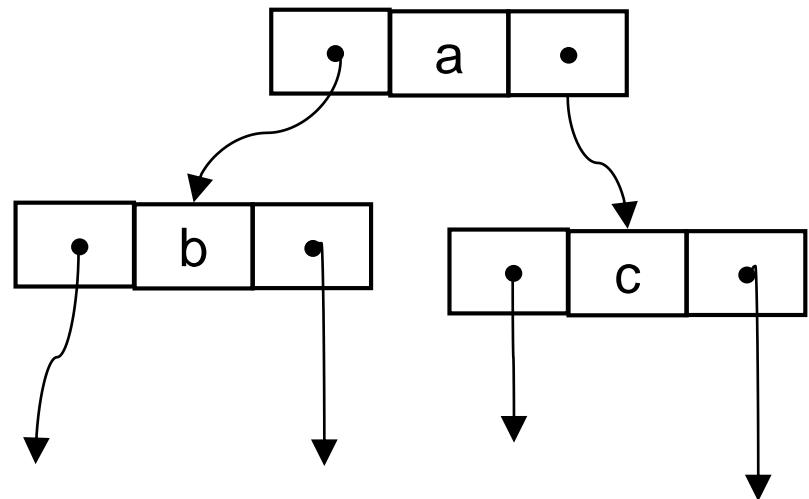
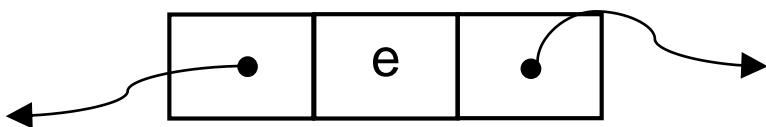
Noch Fragen ??

Datentypen

Implementierung eines Binärbaums → Datenstruktur

- Verknüpfung über Pointer

```
struct b_tree
{ int treeelement; // Knotenelement
  b_tree *left; // linker Teilbaum
  b_tree *right; // rechter Teilbaum
};
```



Datentypen

Implementierung eines Binärbaums → Funktionen

```
b_tree* b_tree_new (b_tree *left, int e, b_tree *right)
/* Erzeugt dynamisch aus zwei Teilbäumen und einem Element einen neuen
Baum, das Ergebnis wird an einem Pointer zurückgegeben */
{ b_tree* t_pointer = (b_tree*) malloc(sizeof(b_tree));
  t_pointer->left = left;
  t_pointer->right = right;
  t_pointer->treeelement = e;
  return t_pointer;
};
```

Speicherplatzanforderung für die neue Wurzel,
Einhängen des linken und rechten Teilbaums

Datentypen

Implementierung eines Binärbaums → Funktionen

```
b_tree* left(b_tree *treepointer)
{ if (treepointer == NULL)
    b_treeerror("Fehler: left aufgerufen mit Nullpointer!\n");
  else return copytree(treepointer->left);
};
```

Datentypen

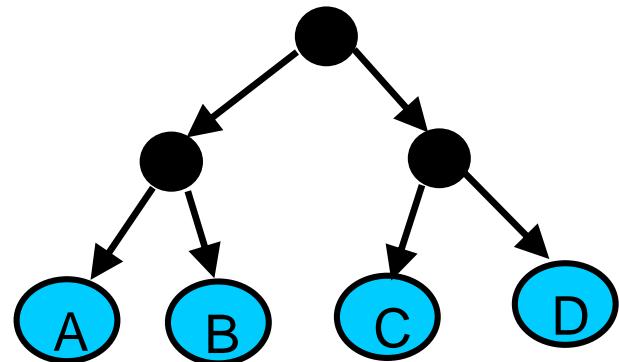
Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Datentypen

Bäume

- weitere „Baumtypen“
 - **n-näre Bäume**, d.h. Bäume, die in jedem Knoten n-fach verzweigen (unärer Baum = verkettete Liste)
 - **beblätterte Bäume**, d.h. Bäume, die Information nur an den Blättern tragen. Die Knoten, die keine Blätter sind, haben nur Verzweigungen
 - ...



Datentypen

Anwendungen von Bäumen

- **Filesystem auf einem Rechner**
- **Teileliste einer Maschine**
- **Suchbäume (binäre Bäume)**

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

- Betrachtet werden Bäume über einem Typ T mit einer linearen Ordnung \leq , d.h. für alle $a, b, c \in T$ gilt:
 - $a \leq a$ (**reflexiv**)
 - aus $a \leq b$ und $b \leq c$ folgt $a \leq c$ (**transitiv**)
 - aus $a \leq b$ und $b \leq a$ folgt $a = b$ (**antisymmetrisch**)
 - es gilt immer $a \leq b$ oder $b \leq a$ (**totale Ordnung**)

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

- Sei a_1, a_2, \dots, a_n eine Folge von n Elementen aus T , so soll durch Sortieren die Folge so neu geordnet werden, dass gilt:

$$a_i \leq a_j \leq a_k \leq \dots \leq a_l ,$$

wobei die Menge $\{a_1, a_2, \dots, a_n\}$ der Menge $\{a_i, a_j, a_k, \dots, a_l\}$ entspricht.

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

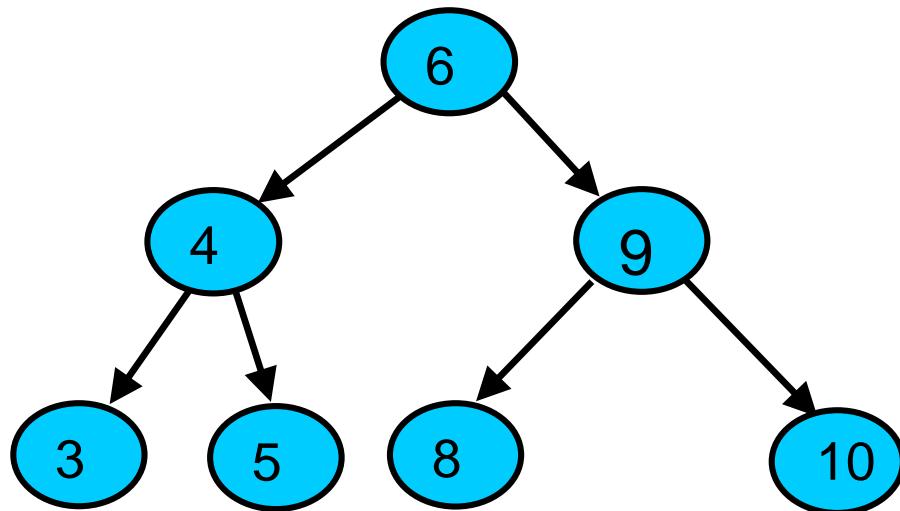
Ein binärer Baum ist geordnet, wenn

- er entweder leer ist oder
- alle Knoten des linken geordneten Teilbaums kleiner oder gleich sind als die Wurzel und
- alle Knoten des rechten geordneten Teilbaums größer oder gleich sind als die Wurzel

Ein geordneter binärer Baum wird als Suchbaum bezeichnet

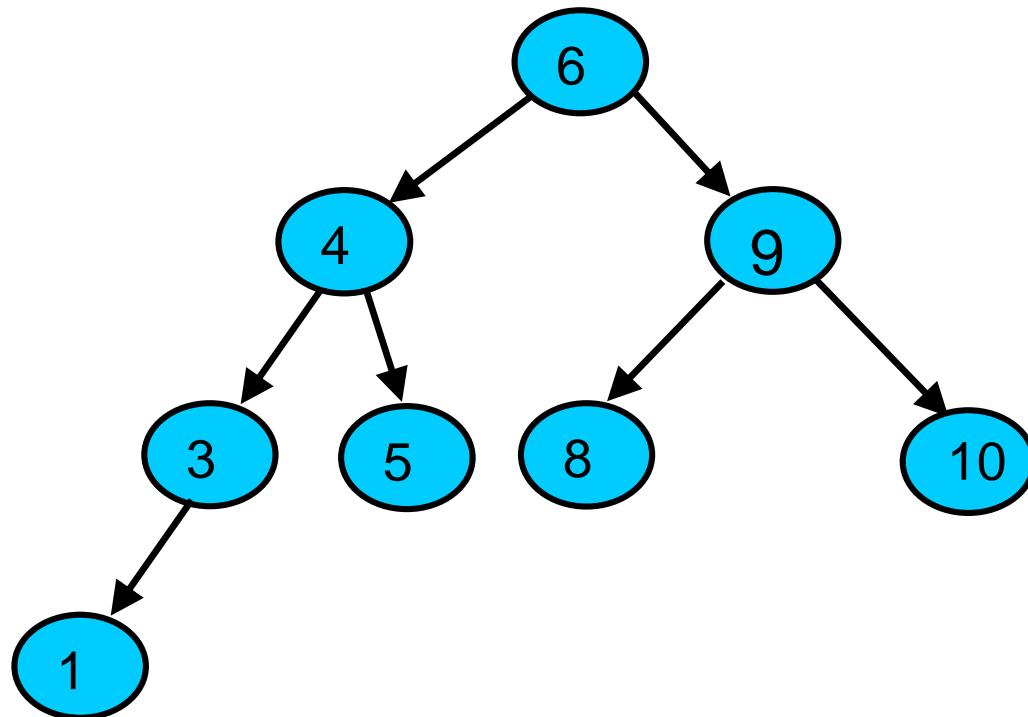
Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren
Suchbäume, Beispiele



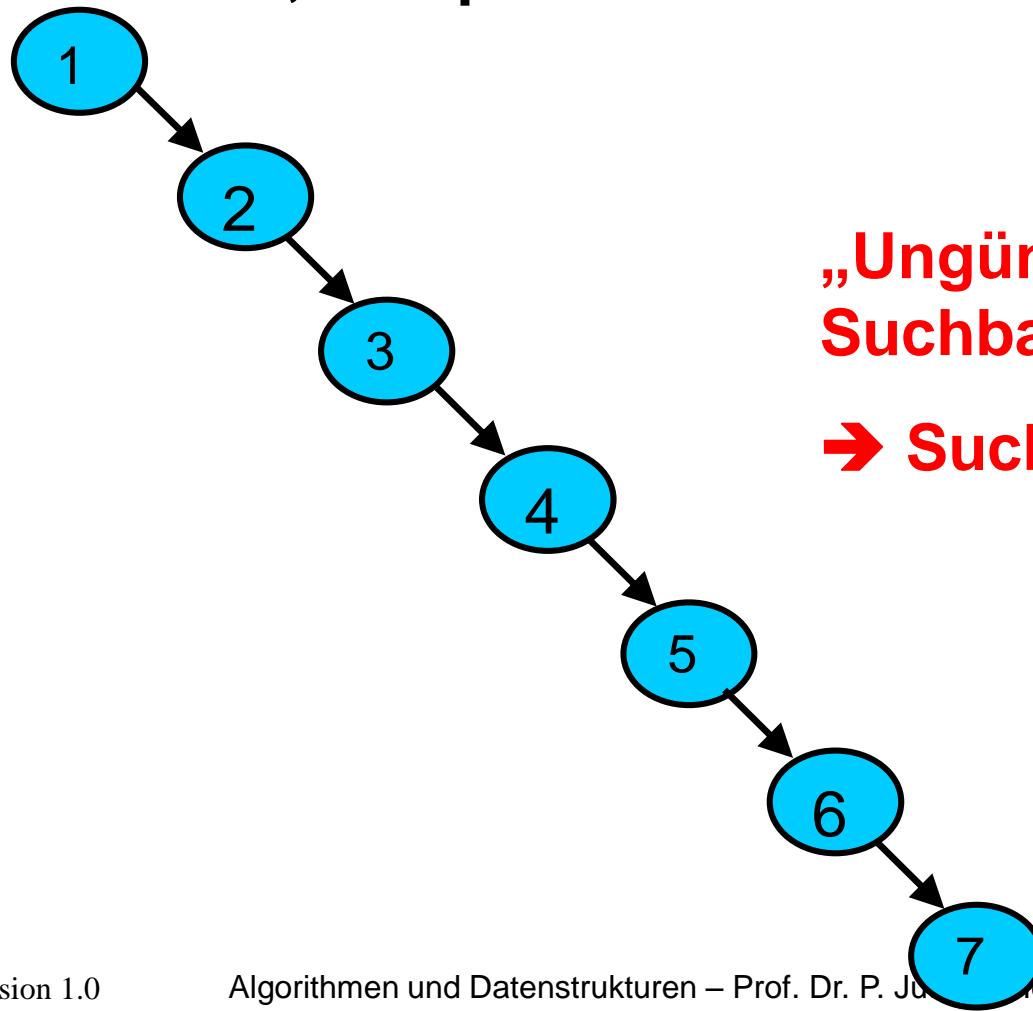
Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren
Suchbäume, Beispiele



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren
Suchbäume, Beispiele



„Ungünstiger“
Suchbaum
→ Suchaufwand $O(n)$

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

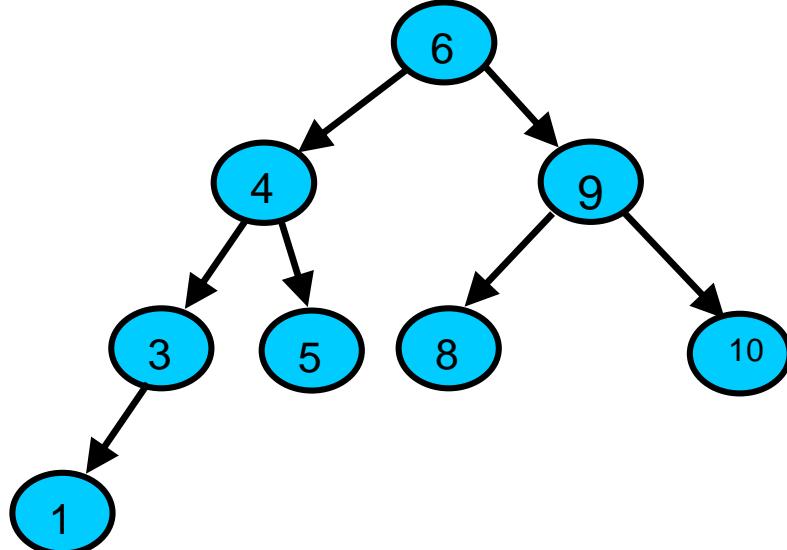
- **Suchen eines Elements x in einen Baum b**
- **Einfügen eines neuen Elements in den Baum**
- **Löschen eines Elements aus dem Baum**
- **Zusammenfügen von zwei Bäumen**
- **Bereinigen „ungünstiger“ Suchbäume**

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

- Suchen eines Elements 9 in einen Baum b:



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

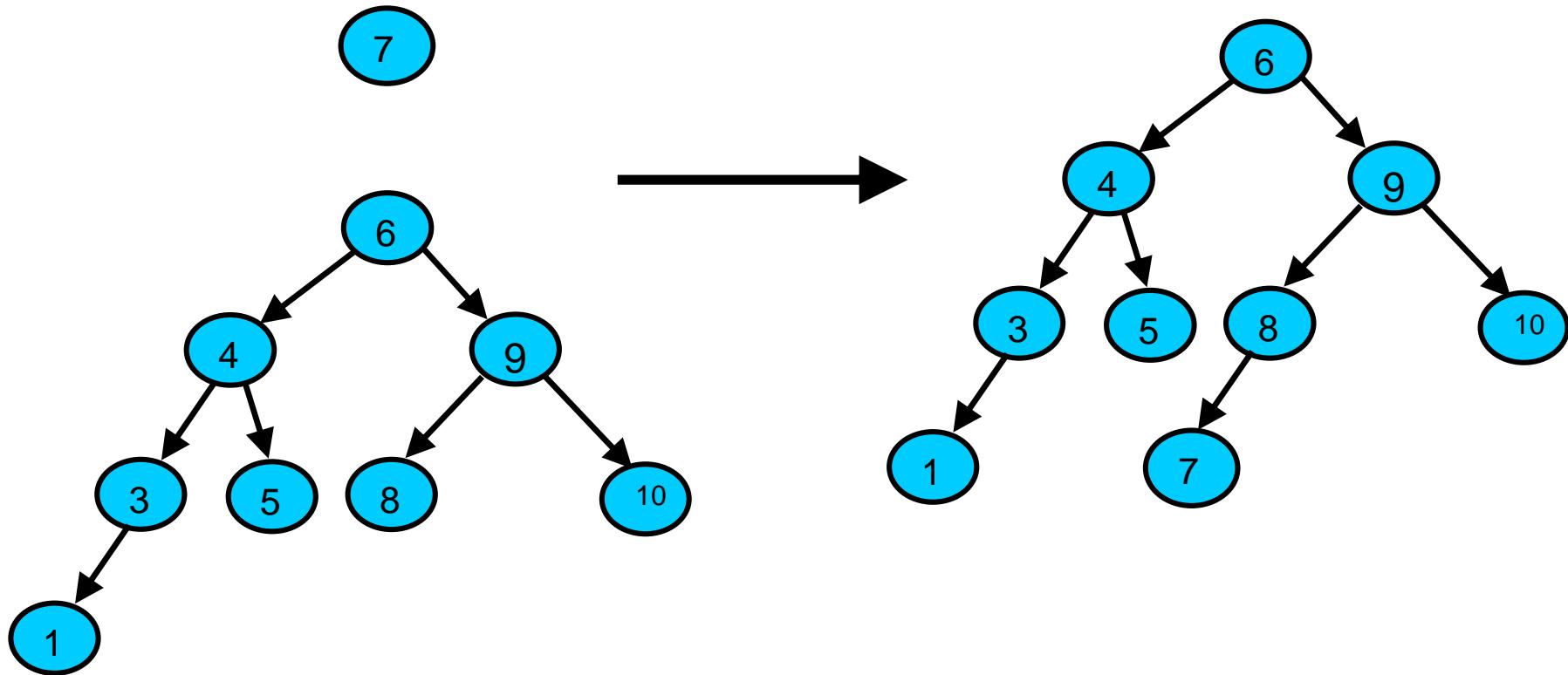
- Suchen eines Elements x in einen Baum b (rekursive Lösung):
 - Falls b leer, false
 - Falls $x = \text{Wurzel} \rightarrow$ true
 - Falls $x < \text{Wurzel} \rightarrow$ suche im linken Teilbaum
 - Falls $x > \text{Wurzel} \rightarrow$ suche im rechtem Teilbaum

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

- Einfügen eines neuen Elements x in einen Baum b :



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

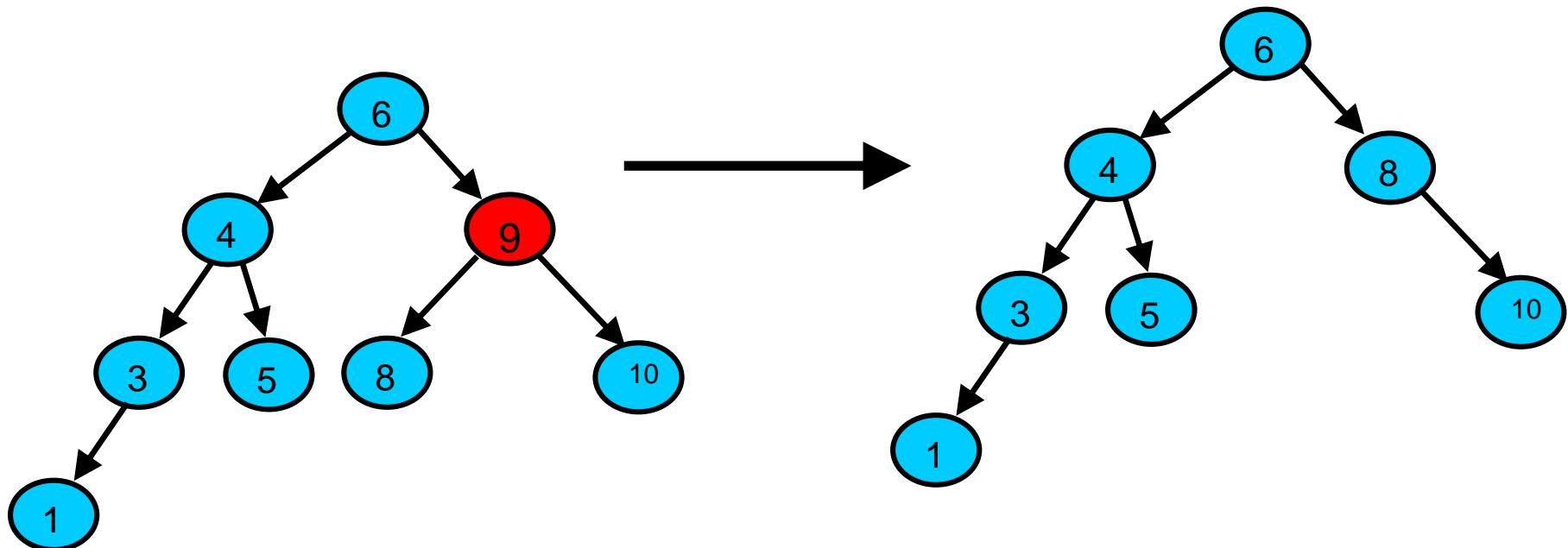
- Einfügen eines neuen Elements x in einen Baum b (rekursive Lösung):
 - Falls b leer, erzeuge neuen Baum mit x als einzigen Knoten
 - Falls x gleich Wurzel von b → x ist bereits im Baum
 - Falls $x <$ Wurzel von b → füge x im linken Teilbaum ein.
 - Falls $x >$ Wurzel von b → füge x im rechten Teilbaum ein.

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

- Löschen eines Elements x aus einem Baum b :



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

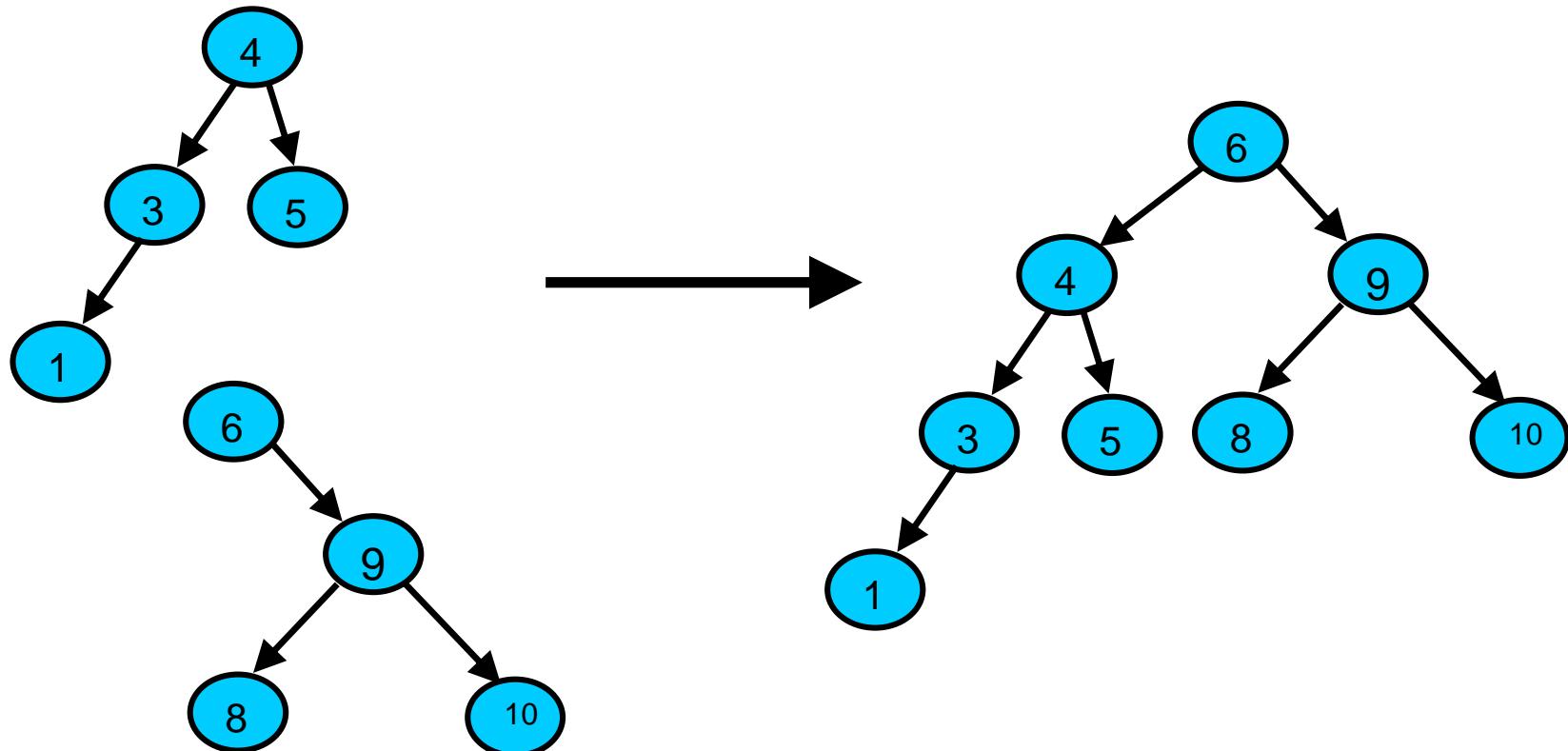
- Löschen eines Elements x in einen Baum b (rekursive Lösung):
 - Falls b leer, fertig
 - Falls $x = \text{Wurzel}$ und linker Teilbaum ist leer → Ergebnis ist rechter Teilbaum
 - Falls $x = \text{Wurzel}$ und rechter Teilbaum ist leer → Ergebnis ist linker Teilbaum
 - Falls $x = \text{Wurzel}$ und weder rechter noch linker Teilbaum sind leer → bilde neuen Baum aus linkem und rechtem Teilbaum
 - Falls $x < \text{Wurzel}$ → lösche x aus linkem Teilbaum
 - Falls $x > \text{Wurzel}$ → lösche x aus rechtem Teilbaum

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

- Zusammenfügen von zwei Suchbäumen a und b:



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

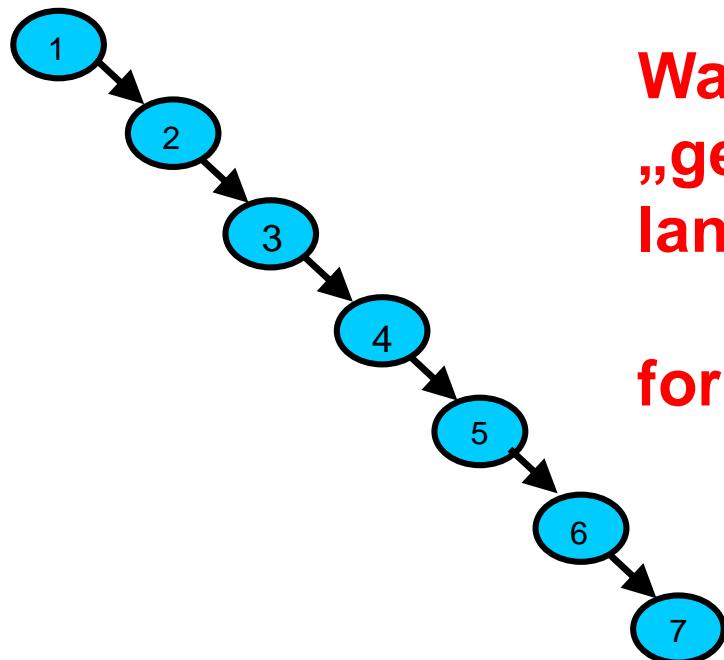
- **Zusammenfügen von zwei Suchbäumen a und b:**
(rekursive Lösung):
 - **Falls a leer, Ergebnis ist b**
 - **Falls b leer, Ergebnis ist a**
 - **Nimm die Wurzel von b, füge diese in a ein, danach füge a mit dem linken Teilbaum von b zusammen und dann mit dem rechten Teilbaum von b**

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

- Bereinigen „ungünstiger“ Suchbäume :



**Wann ist ein Suchbaum ungünstig ?
„gefühlt: wenn ein Ast ungewöhnlich lang wird“ Folge?**

formales Kriterium: ??

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

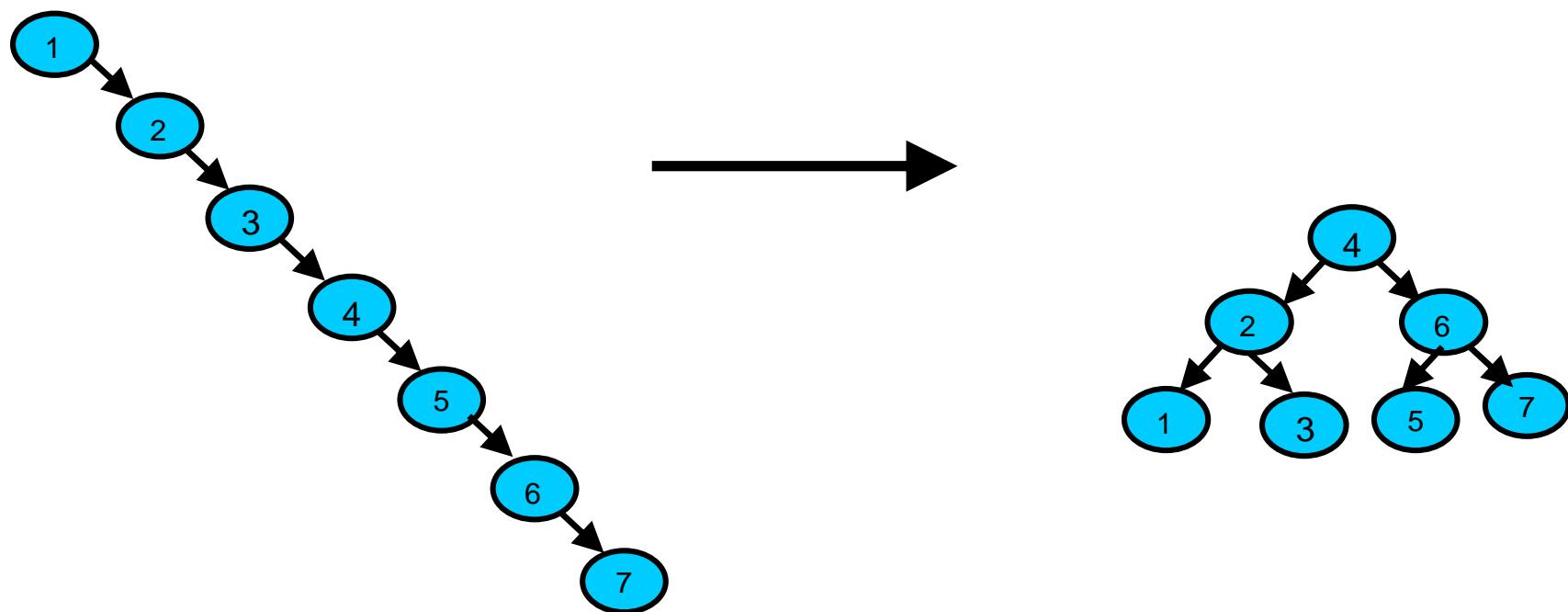
- **Bereinigen „ungünstiger“ Suchbäume**
 - **ungünstige Suchbäume erhöhen den Aufwand bei allen Operationen**
 - **$O(\log n) \rightarrow O(n)$**

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

- Bereinigen „ungünstiger“ Suchbäume :
 - „Umbauen des Baums“



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf Suchbäumen

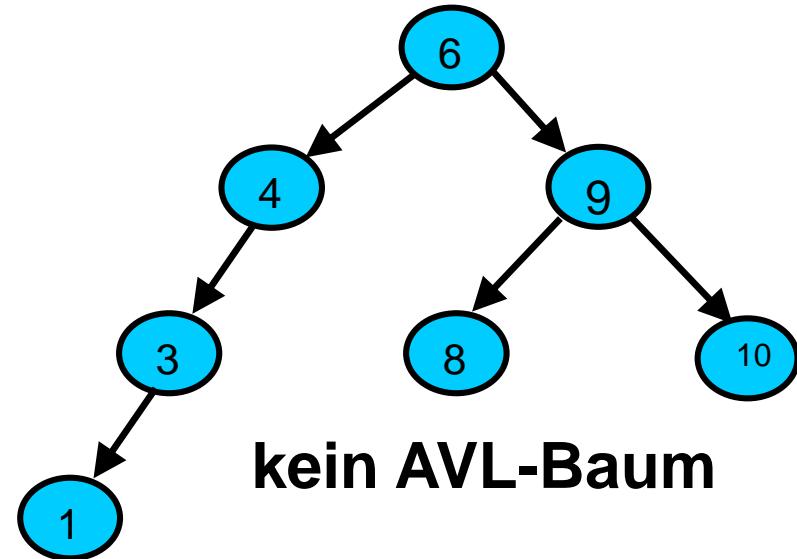
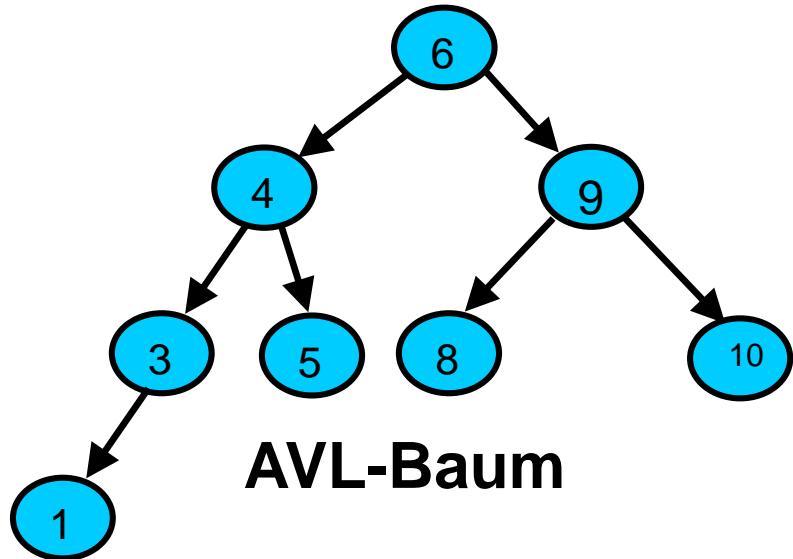
- **Bereinigen „ungünstiger“ Suchbäume**
 - **Bereinigung nach jeder Operation, die den Baum verändert ist aufwändig**
 - **Nutzung von AVL-Bäumen**
 - **Nutzung von B-Bäumen**

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Suchbäumen, AVL Bäume

- benannt nach den russischen Mathematikern Adelson-Velskii und Landis (1962)
- Kriterium: für jeden (inneren) Knoten gilt: Höhe des linken und rechten Teilbaums differieren maximal um 1



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Suchbäumen, AVL Bäume

- **Minimale Anzahl der Knoten eines AVL-Baums mit Höhe n**

$$A(0) = 0$$

$$A(1) = 1$$

$$A(n) = A(n-1) + A(n-2) + 1 \text{ für } (n>1)$$

→ Ergebnis ähnlich Fibonacci-Zahlen*)

$$\text{Es gilt } A(n) \leq 1,44 * \log(n + 2) - 0,328$$

*) deshalb werden diese Bäume manchmal auch Fibonacci-Bäume genannt

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

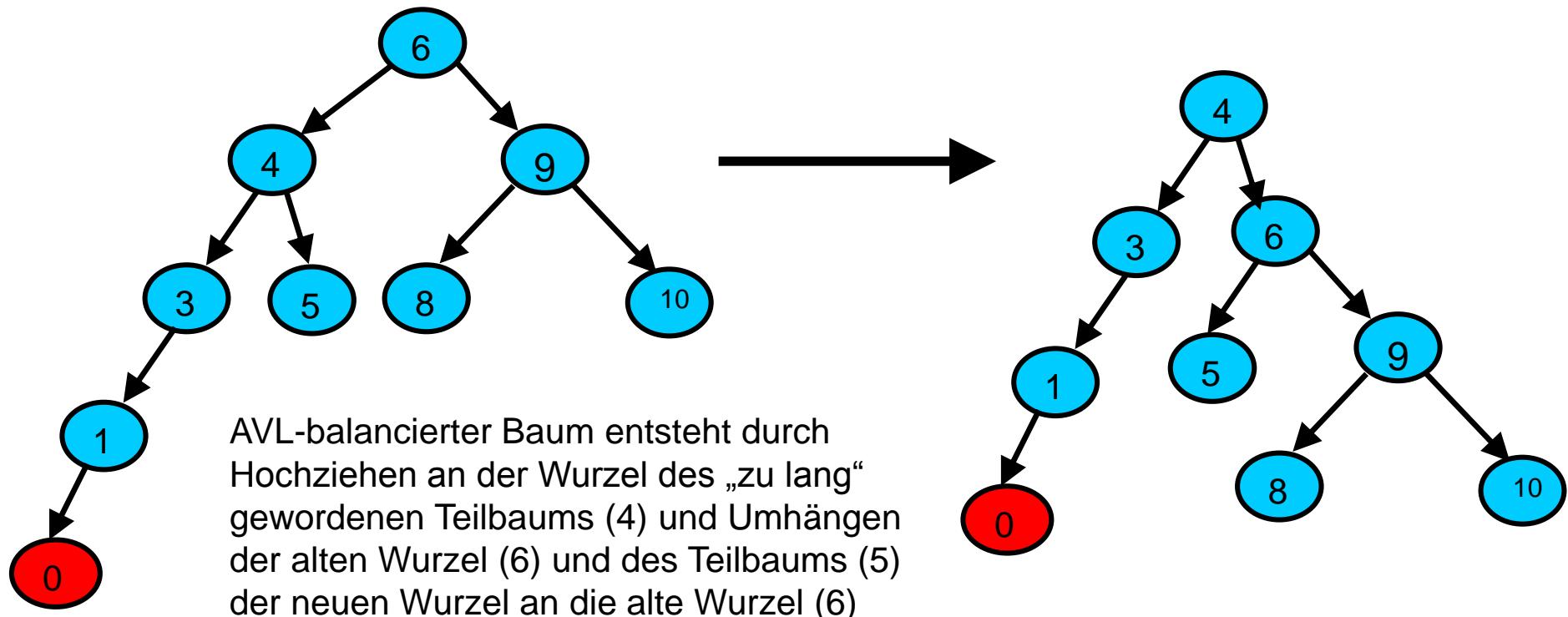
- **Einfügen, Löschen, Suchen wie zuvor**
- **Ändernde Operationen können die AVL-Eigenschaft zerstören**
→ Reparieren mittels
Eine Rotation oder Doppelrotation beim Einfügen
Eine oder mehrere (Doppel)Rotationen beim Löschen
- **Operationen auf AVL-Bäumen erfordern $O(\log n)$ Aufwand**

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

- Rotation nach Einfügen

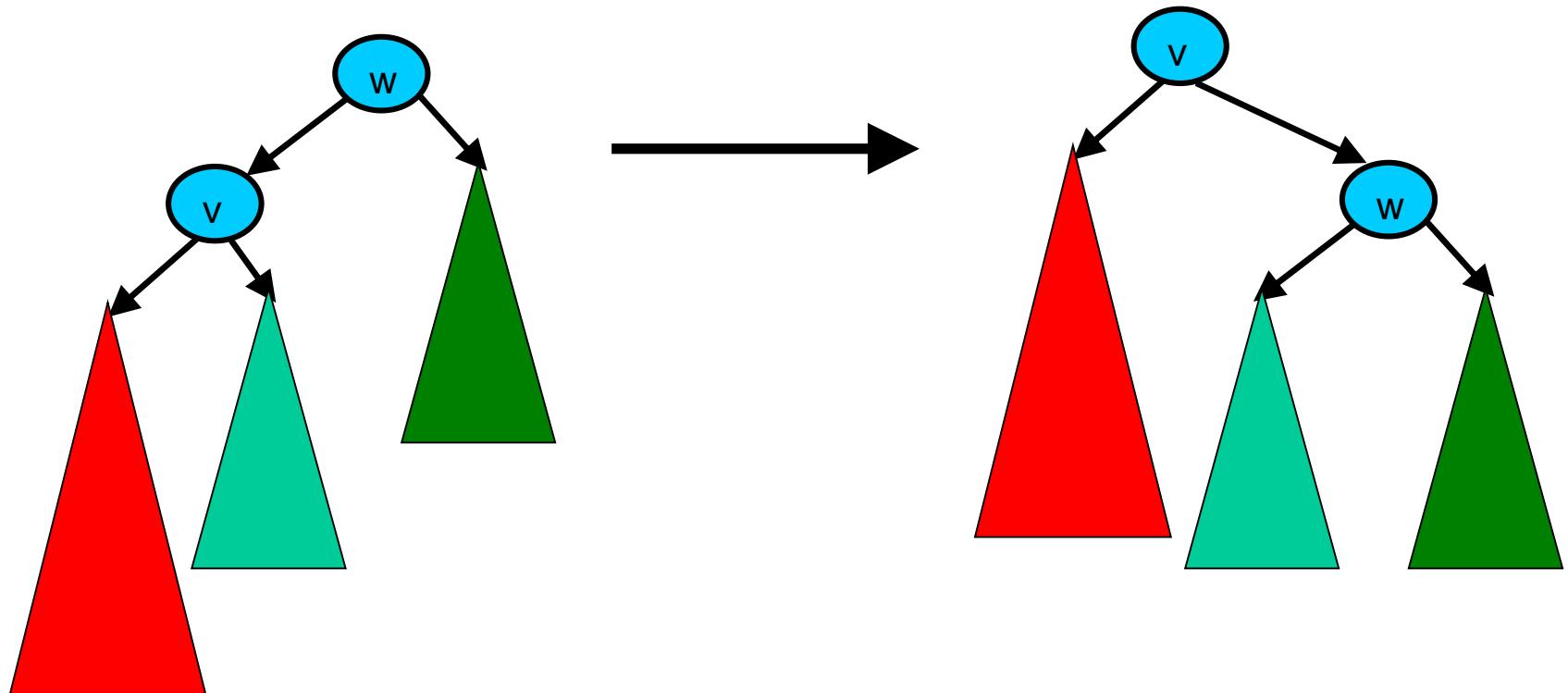


Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

- Rotation nach Einfügen (roter Teil wird „zu lang“)

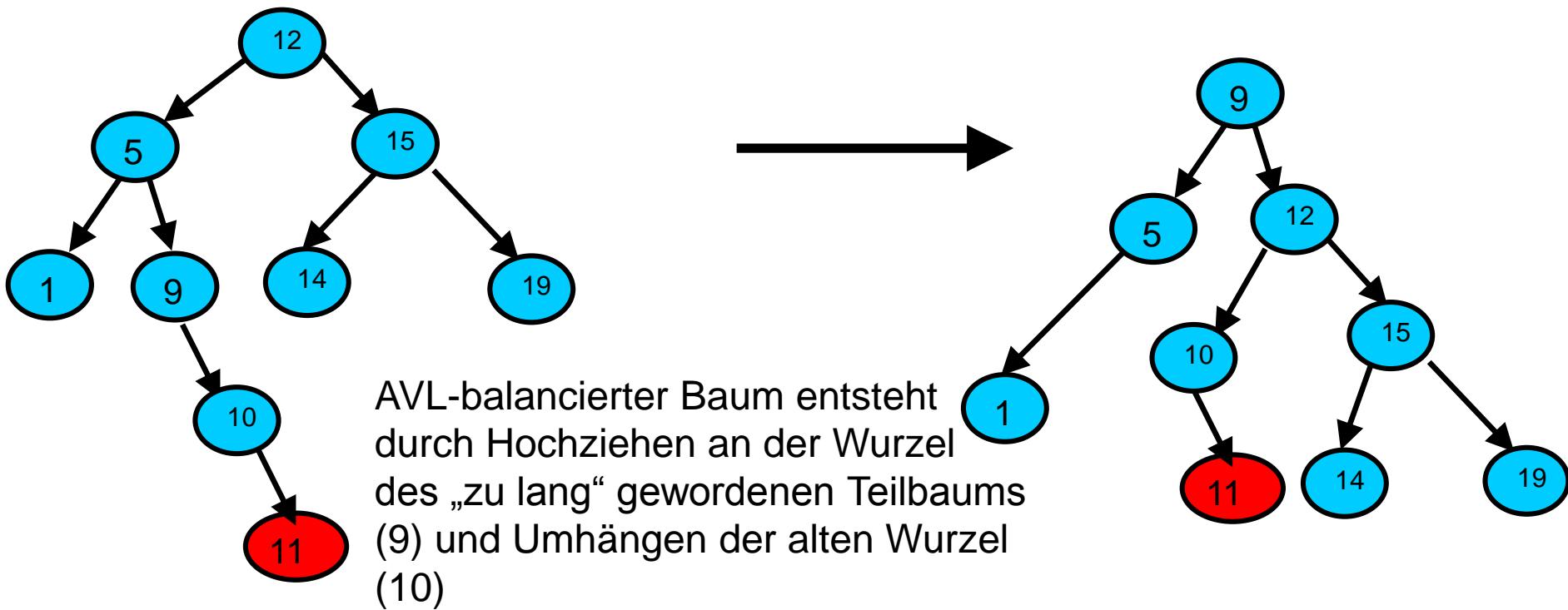


Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

- Doppelrotation nach Einfügen

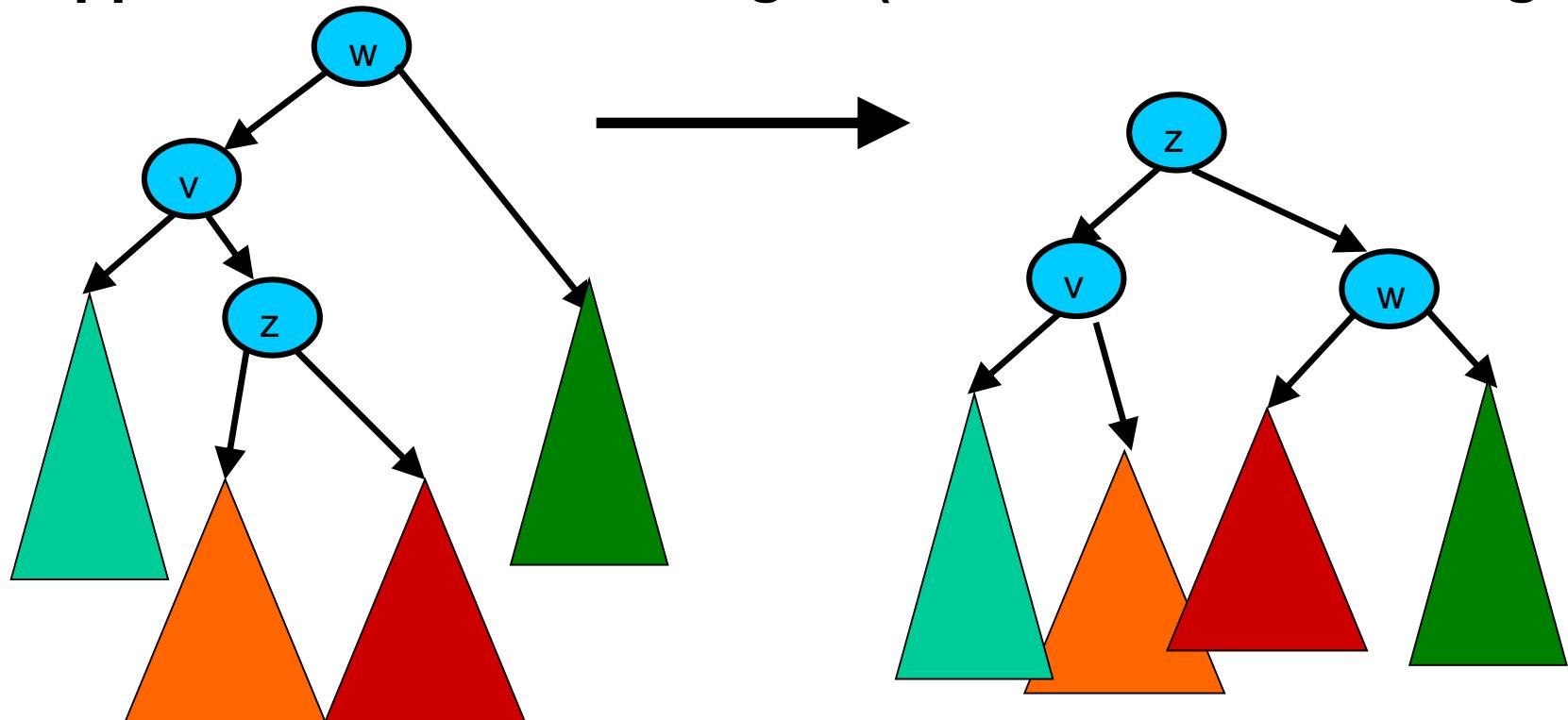


Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

- Doppelrotation nach Einfügen (roter Teil wird „zu lang“)

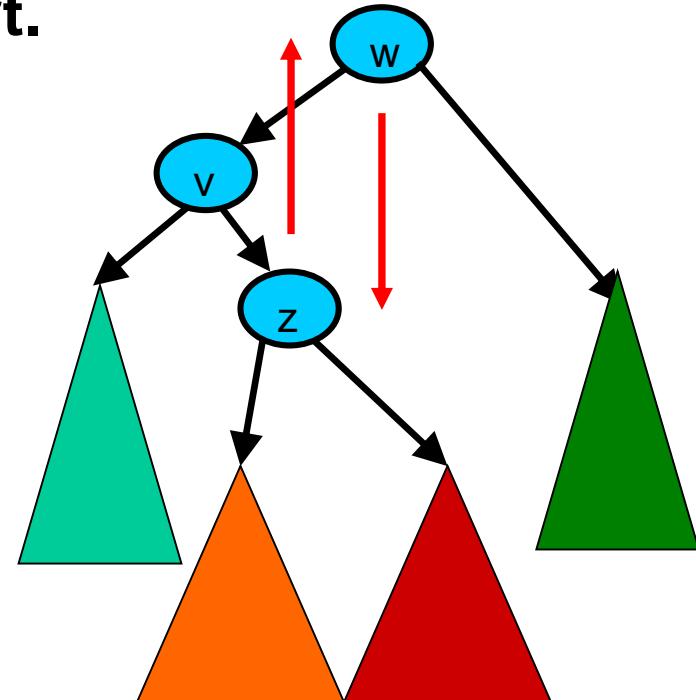


Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

Anmerkung Rotationen. Bei den Rotationen ändert sich die „horizontale Position“ eines Knotens nicht, nur die vertikale Position wird verändert.



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

- **Zusammenfassung Rotationen auf oberstem Niveau:**
 1. Einfügen in linken Teilbaum des linken Kindes
 2. Einfügen in rechten Teilbaum des linken Kindes
 3. Einfügen in linken Teilbaum des rechten Kindes
 4. Einfügen in rechten Teilbaum des rechten Kindes

Anmerkung 1. und 4. sind symmetrisch, 2. und 3. sind symmetrisch

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Operationen auf AVL Bäumen

- Sonderfälle: Rotation von Teilbäumen
 - Einfügen in linken Teilbaum des linken Kindes
→ Rotation mit linkem Kind
 - Einfügen in rechten Teilbaum des linken Kindes
→ Doppelrotation mit linkem Kind
 - Einfügen in linken Teilbaum des rechten Kindes
→ Doppelrotation mit rechtem Kind
 - Einfügen in rechten Teilbaum des rechten Kindes
→ Rotation mit rechtem Kind

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Implementierung von AVL-Bäumen

```
typedef struct AVLBaum
{ AVLBaum *left;
  int knoten;
  AVLBaum *right;
  int balance;
};
```

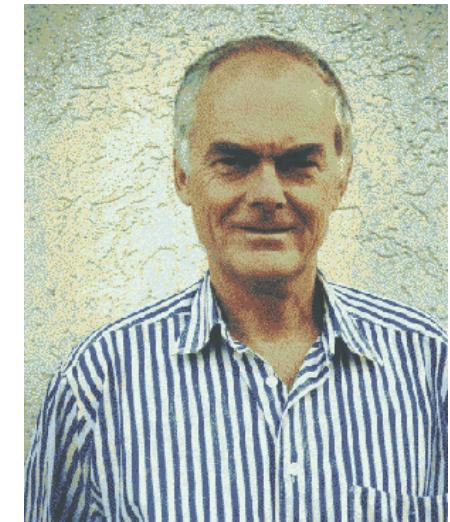
Balance wird in jedem Knoten mitgeführt um schnell entscheiden zu können, ob eine Rotation notwendig ist.
(Ohne diese Information muss der Baum bei jedem Einfügen komplett auf die AVL Bedingung geprüft werden.)

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Suchbäumen, B-Bäume

- benannt 1978 nach ihrem Erfinder R. Bayer (B kann auch stehen für balanciert, breit, buschig, aber NICHT für binär)
- B-Bäume sind dynamische balancierte Mehrweg-suchbäume (d.h. nicht binär)



Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren

Suchbäumen, B Bäume → Motivation

- **Vollständig ausgeglichener Mehrwegbaum:**
 - alle Wege von der Wurzel bis zu den Blättern gleich lang,
 - jeder Knoten gleich viele Einträge
- Kriterium nur mit sehr hohem Aufwand einzuhalten
- Modifikation zu B-Bäumen

Datentypen

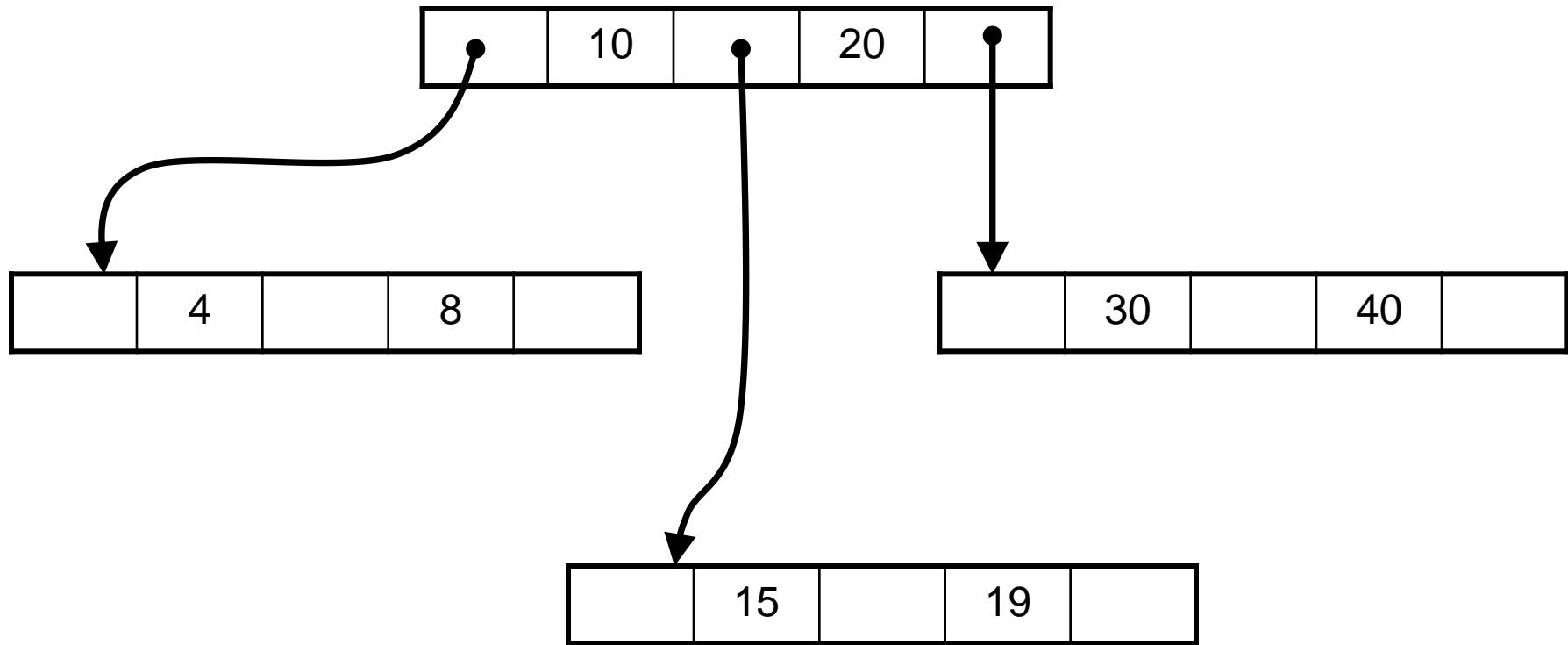
Anwendungen binärer Bäume → Suchen, Sortieren

Suchbäumen, B Bäume → Definition

- **Jede Seite (= Knoten) enthält höchstens $2m$ Elemente.**
- **Jede Seite, außer der Wurzel, enthält mindestens m Elemente.**
- **Jede Seite ist entweder ein Blatt ohne Nachfolger oder hat $i + 1$ Nachfolger, wobei i die Anzahl ihrer Elemente ist.**
- **Alle Blattseiten liegen auf der gleichen Stufe, d.h. die Höhe vom Blatt zur Wurzel ist für alle Blätter gleich.**

Datentypen

Anwendungen binärer Bäume → Suchen, Sortieren
Suchbäumen, B Bäume → Beispiel



Datentypen

Zum Schluss dieses Abschnitts ...

Noch Fragen ??