Algorithmen und Datenstrukturen

Marc Fischlin, SS 2024

03
Grundlegende Datenstrukturen

Stacks

zur Erinnerung

Abstrakte Datentypen (ADTs) und Datenstrukturen

näher an der Anwendung

Beispiel:

Abstrakter Datentyp („was“)

Stack mit Operationen
`isEmpty,pop,push`

Übergang fließend; ADTs werden daher oft auch als Datenstruktur bezeichnet

Datenstruktur („wie“)

Stack-Operationen
als Array
oder verkettete Liste

näher „an der Maschine“

Abstrakter Datentyp Stack

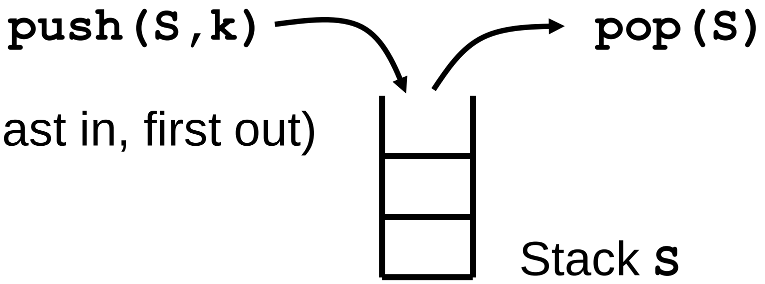
Bemerkung: auch Schreibweise
`S.push(k)` oder einfach **`pop()`**

`new(S)` - erzeugt neuen (leeren) Stack namens **`S`**

`isEmpty(S)` - gibt an, ob Stack **`S`** leer

`pop(S)` - gibt oberstes Element vom Stack **`S`** zurück und löscht es vom Stack (bzw. Fehlermeldung, wenn Stack leer)

`push(S,k)` - schreibt **`k`** als neues oberstes Element auf Stack **`S`** (bzw. Fehlermeldung, wenn Stack voll)



(LIFO – last in, first out)

Formale Erfassung z.B.
per algebraischer Spezifikation

Algebraische Spezifikation von Stacks

Stack mit Operationen **`new,isEmpty,push,pop`** muss folgende Regeln erfüllen:

(1) Wenn **`new(S)`**, dann unmittelbar **`isEmpty(S)`**, ergibt Ergebnis **`true`**.

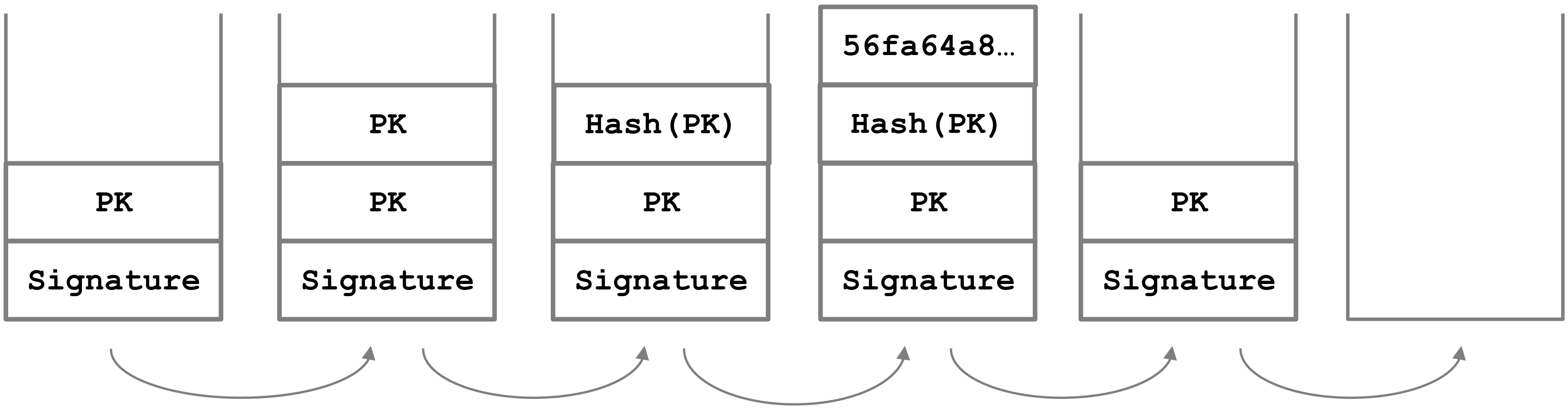
(2) Wenn **`push(S,k)`** und keine Fehlermeldung, dann unmittelbar **`pop(S)`**, ergibt Ergebnis **`k`**.

(3) …

Fokus dieses Teils der Vorlesung liegt auf Entwurf der Datenstrukturen; alle Lösungen erfüllen „natürliche“ Forderungen an solche Operationen.

Beispiel: Bitcoin

```
scriptPubKey:
OP_DUP OP_HASH160 56fa64a8bd7852d2c58095fa9a2fcd52d2c580b65d35549d
OP_EQUALVERIFY OP_CHECKSIG
```



OP_DUP	OP_HASH160	56fa64a8…	OP_EQUALVERIFY	OP_CHECKSIG
entspricht **k=pop()** **push(k)** **push(k)**	entspricht **k=pop()** //h=Hash(k) **push(h)**	entspricht **push(56fa…)**	entspricht **k=pop()** **h=pop()** //k==h?	entspricht **k=pop()** **h=pop()** //k,h gültig?

Stacks als Array (I)

Annahme: maximale Größe MAX
des Stacks vorher bekannt

	0	1	2	3	4	5	6	7	8
`S`	12	47	17	98	72				

`S.top`

zeigt auf oberstes Element

`pop()`

	0	1	2	3	4	5	6	7	8
`S`	12	47	17	98					

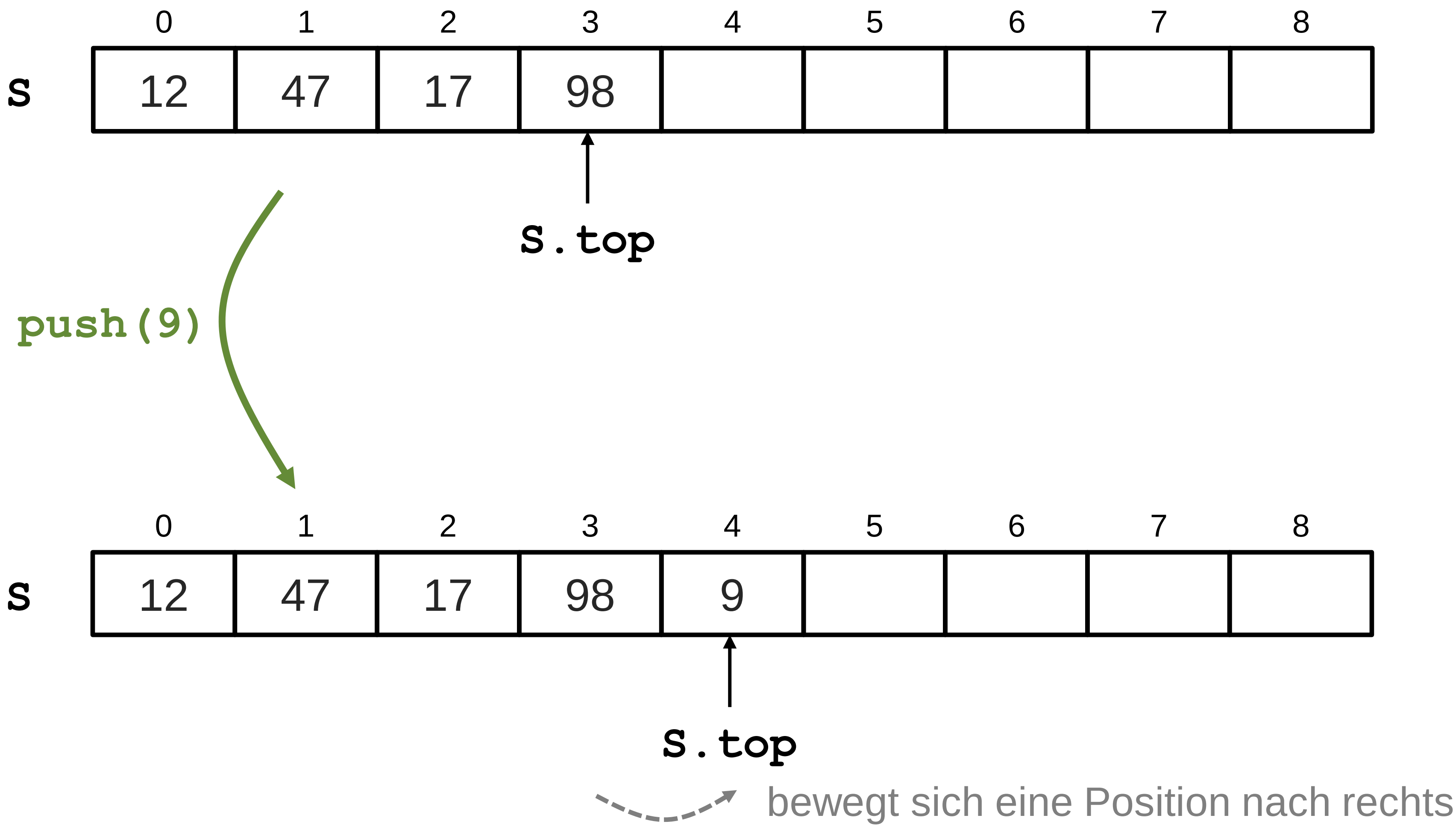
`S.top`

bewegt sich eine Position nach links

gibt 72 zurück

Stacks als Array (II)

Annahme: maximale Größe MAX
des Stacks vorher bekannt



Stacks als Array: Algorithmen

Annahme: maximale Größe MAX des Stacks vorher bekannt

	0	1	2	3	4	5	6	7	8
S	12	47	17	98	9				

S.top

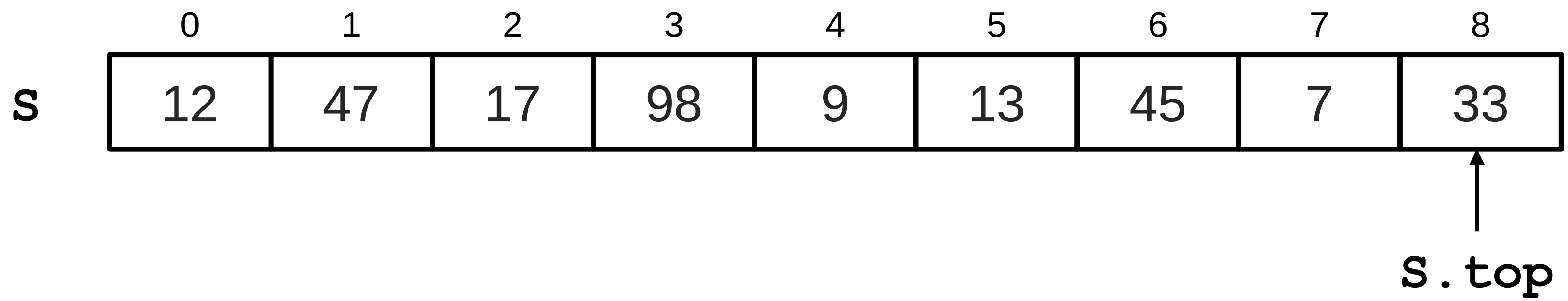
```
new(S)

S.A[]=ALLOCATE(MAX);
S.top=-1;
```

```
isEmpty(S)

IF S.top<0 THEN
    return true
ELSE
    return false;
```

```
pop(S)

IF isEmpty(S) THEN
    error 'underflow'
ELSE
    S.top=S.top-1;
    return S.A[S.top+1];
```

```
push(S,k)

IF S.top==MAX-1 THEN
    error 'overflow'
ELSE
    S.top=S.top+1;
    S.A[S.top]=k;
```

Stacks mit variabler Größe



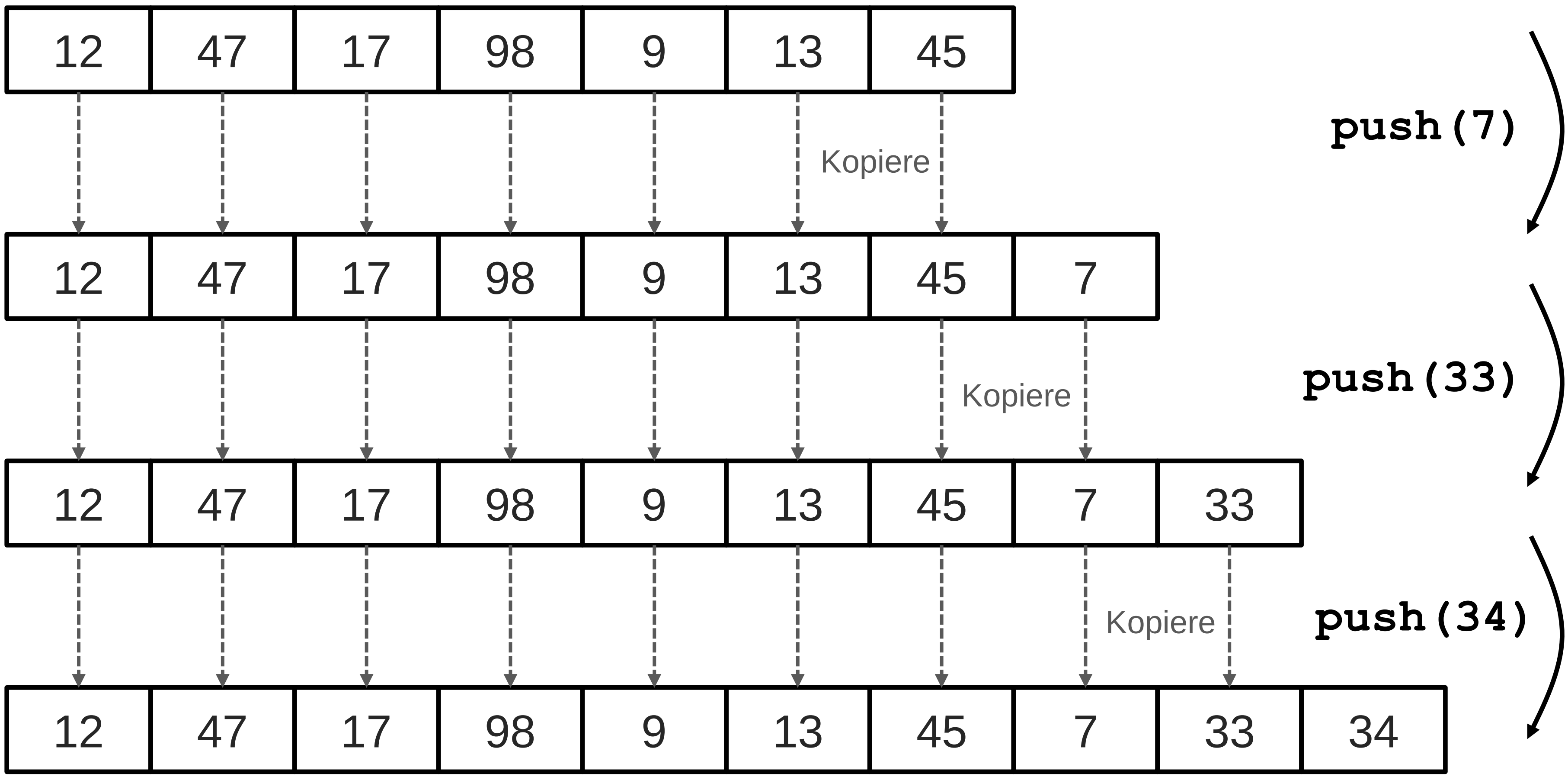
`push(14)`

Kopiere entweder in größeres, zusammenhängendes Array um,

oder verteile auf viele Arrays (→ Datenstruktur verkette Listen)

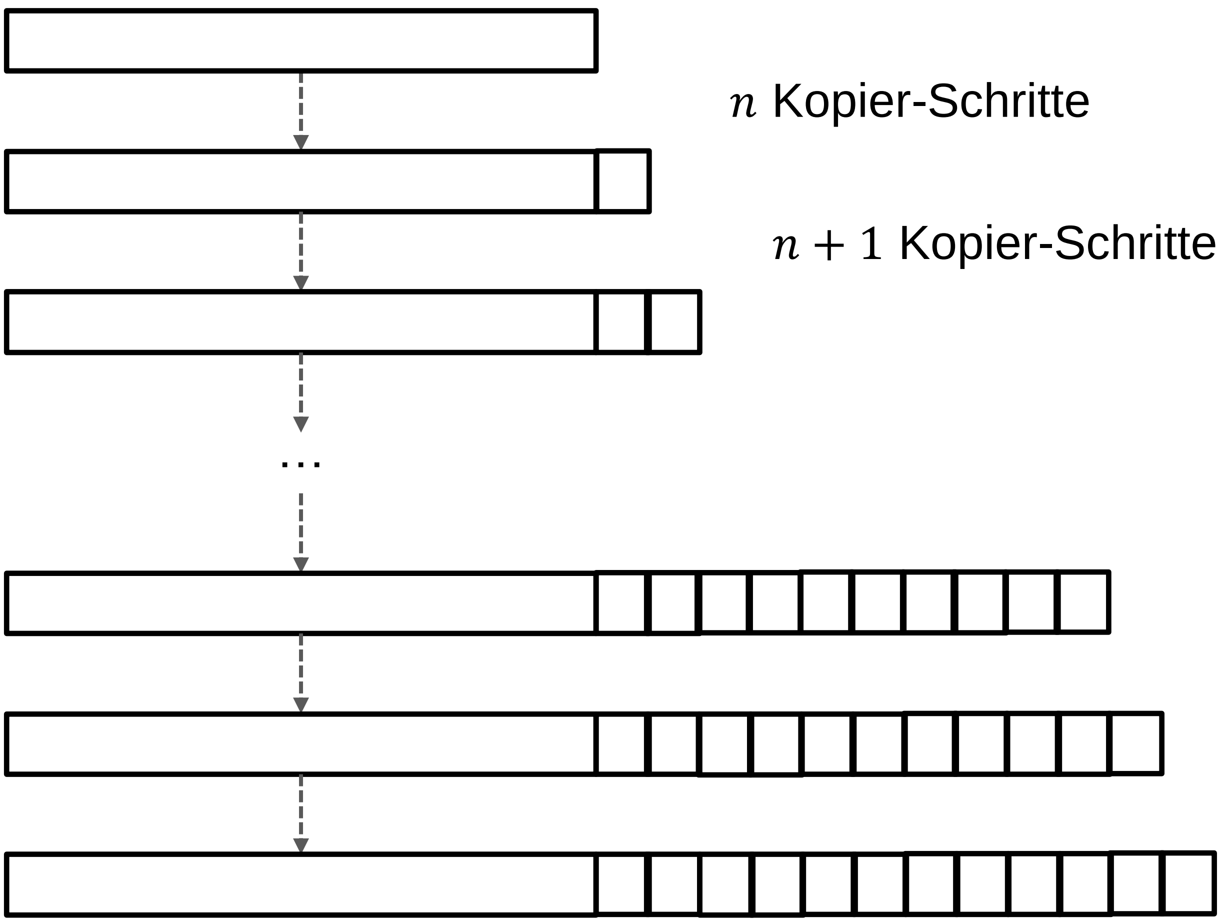
Einfache Feldarbeit

Erzeuge bei Bedarf neues Array mit zusätzlichem Eintrag und kopiere aktuellen Stack um



Laufzeit

$n = MAX$ Elemente in Liste und n weitere **`Push`**-Befehle



n Kopier-Schritte

$n + 1$ Kopier-Schritte

$2n - 2$ Kopier-Schritte

$2n - 1$ Kopier-Schritte

In Summe also
$n^2 + \sum_{i=0}^{n-1} i = \Omega(n^2)$
Kopier-Schritte

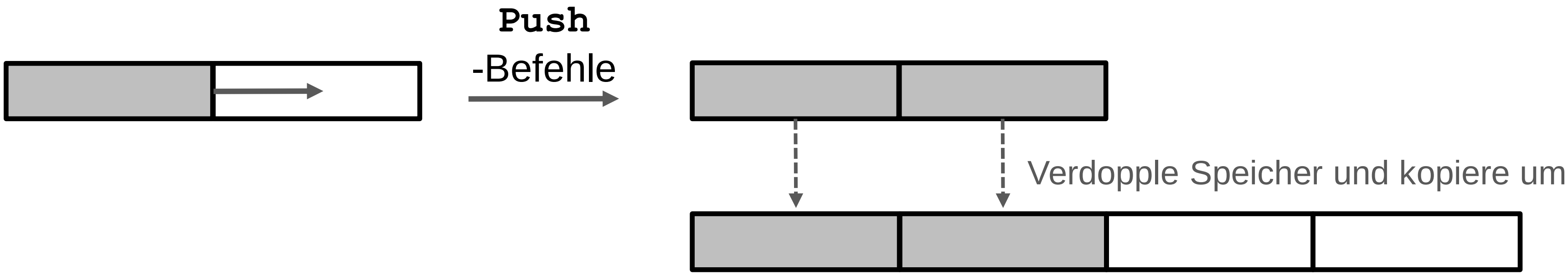
Somit durchschnittlich
$\Omega(n)$ Kopier-Schritte
pro **`Push`**-Befehl!

Verbesserung?

Triviale Lösung: reserviere „unendlich“ viel Speicher

Gesucht: Lösung, die maximal jeweils $O(\#Elemente)$ Zellen benötigt

Idee: Wenn Grenze erreicht, verdoppele Speicher und kopiere um



Schrumpfe und kopiere um, sofern weniger als ein Viertel belegt

Feldarbeit: Algorithmen

`RESIZE(S,m)`
reserviert neuen Speicher der Größe **`m`**,
kopiert **`S.A`** um, und lässt **`S.A`** auf neuen Speicher zeigen

```
new(S)

S.A[]=ALLOCATE(1);
S.top=-1;
S.memsize=1;
```

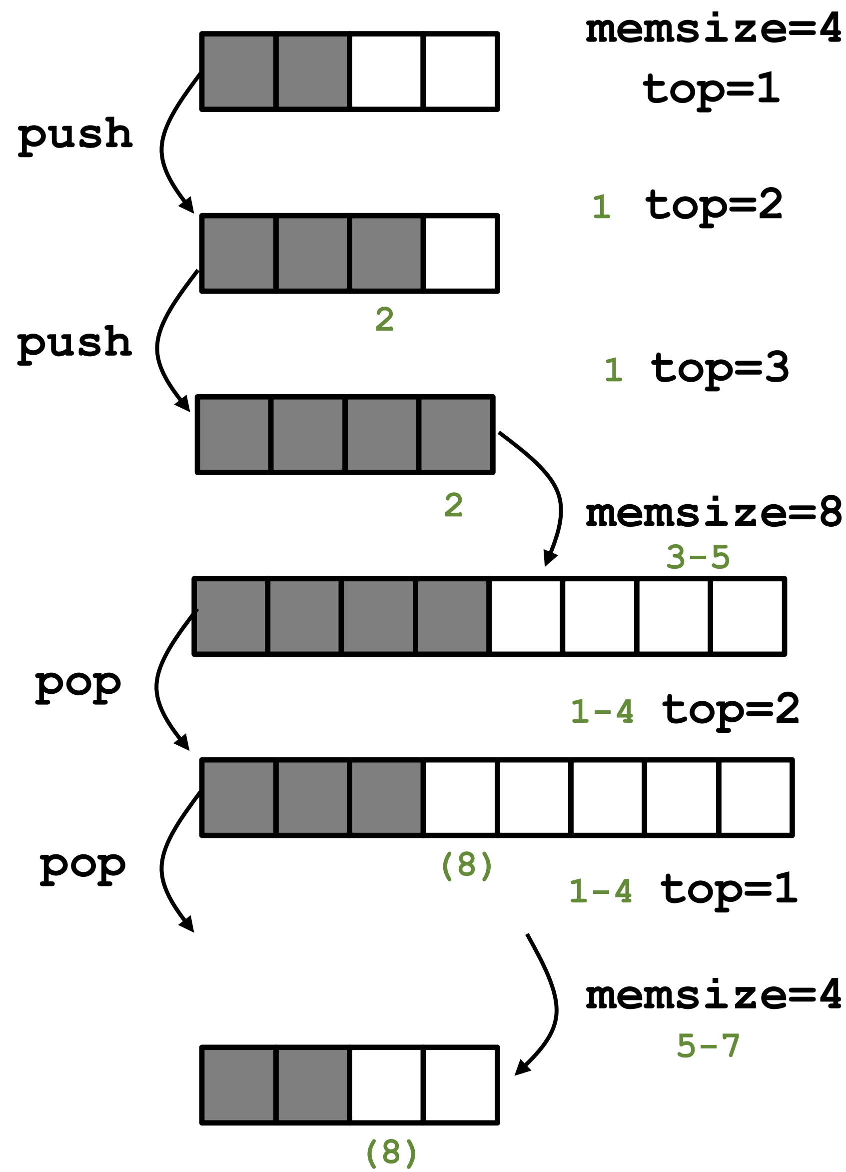
```
isEmpty(S)

IF S.top<0 THEN
    return true
ELSE
    return false;
```

```
pop(S)

IF isEmpty(S) THEN
  error ‘underflow‘
ELSE
  S.top=S.top-1;
  IF 4*(S.top+1)==S.memsize THEN
     S.memsize=S.memsize/2;
     RESIZE(S,S.memsize);
  return S.A[S.top+1];
```

```
push(S,k)

S.top=S.top+1;
S.A[S.top]=k;
IF S.top+1==S.memsize THEN
  S.memsize=2*S.memsize;
  RESIZE(S,S.memsize);
```

```
push(S,k)

1  S.top=S.top+1;
2  S.A[S.top]=k;
3  IF S.top+1==S.memsize THEN
4    S.memsize=2*S.memsize;
5    RESIZE(S,S.memsize);
```

```
pop(S)

1  IF isEmpty(S) THEN
2    error 'underflow'
3  ELSE
4    S.top=S.top-1;
5    IF 4*(S.top+1)==S.memsize THEN
6       S.memsize=S.memsize/2;
7       RESIZE(S,S.memsize);
8    return S.A[S.top+1];
```

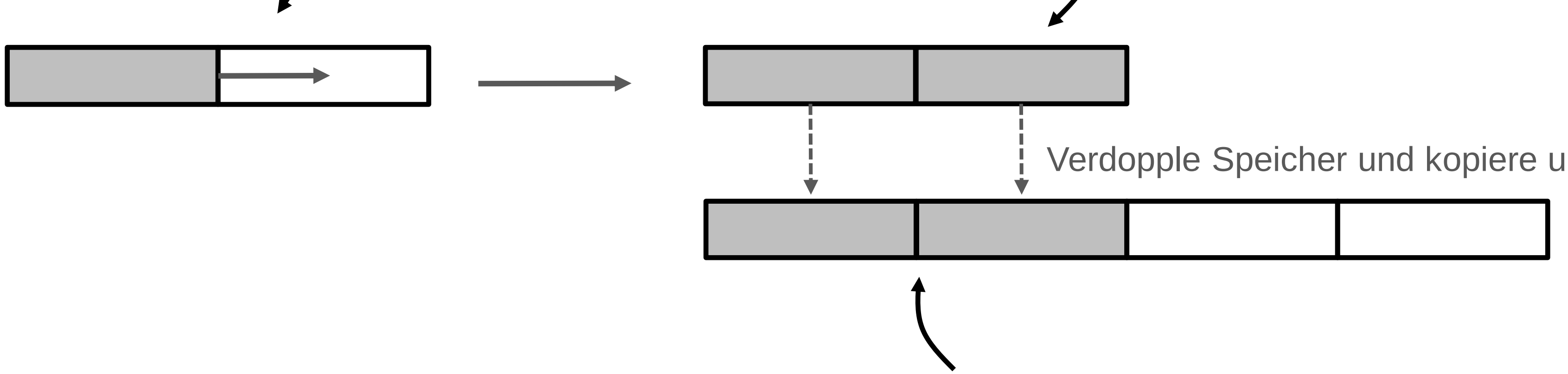


Analyse Laufzeit

Vergrößern (gilt analog für Verkleinern)

(1) n Elemente (unmittelbar nach letzter Vergrößerung)

(2) neue Speichergrenze wird nur erreicht, wenn dann mindestens n viele **`Push`**-Befehle



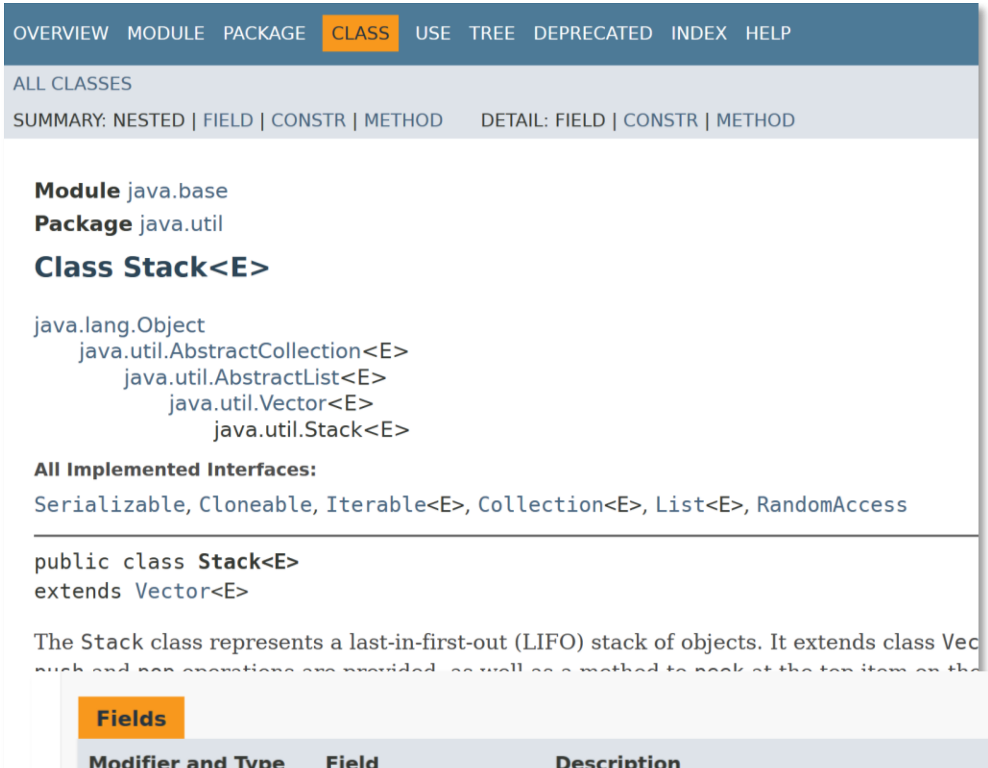
(3) Umkopieren kostet dann $O(n)$ Schritte

Im Durchschnitt für jeden der mindestens n Befehle $\Theta(1)$ Umkopierschritte!

Stacks in Java

Class Stack in Java 13 bis 22

Quelle: Wikipedia



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

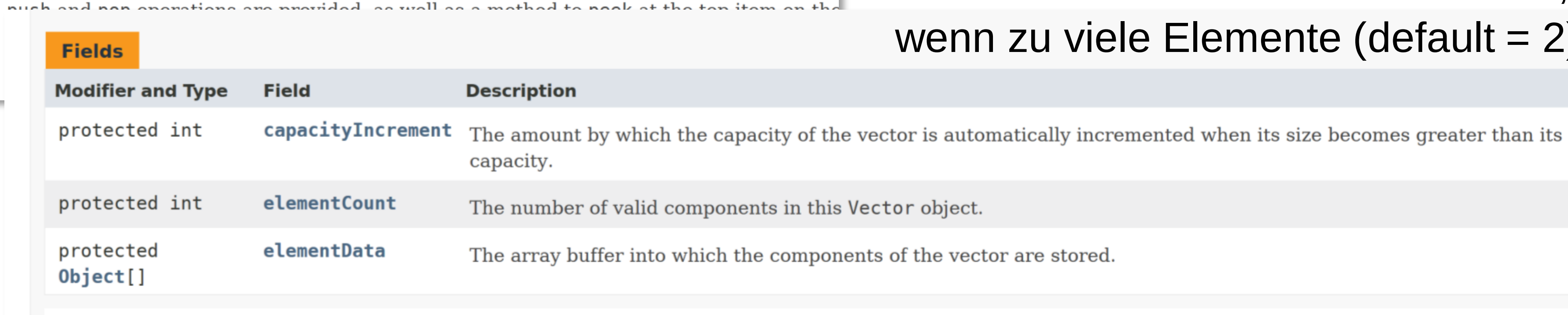
All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

```
public class Stack<E>
extends Vector<E>
```

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vec

capacityIncrement gibt an, wieviel **Vector** wachsen soll, wenn zu viele Elemente (default = 2)



Fields

Modifier and Type	Field	Description
protected int	capacityIncrement	The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
protected int	elementCount	The number of valid components in this Vector object.
protected Object[]	elementData	The array buffer into which the components of the vector are stored.

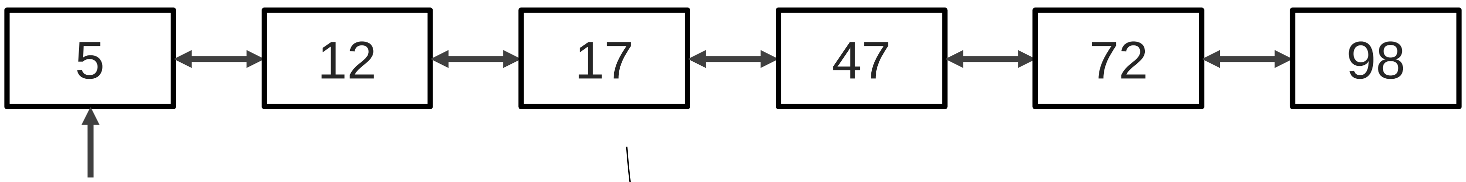
Geben Sie eine „schlechte“ Eingabe für die Java-Klasse **`Stack`** an, bei der wegen des fehlenden Schrumpfens viel Speicher verschwendet wird.

Stellen Sie die Operation **`clear`** für einen Stack in Pseudocode dar, die den Stack leert.

Verkettete Listen

Datenstruktur Verkettete Listen

(doppelt) verkettete Liste



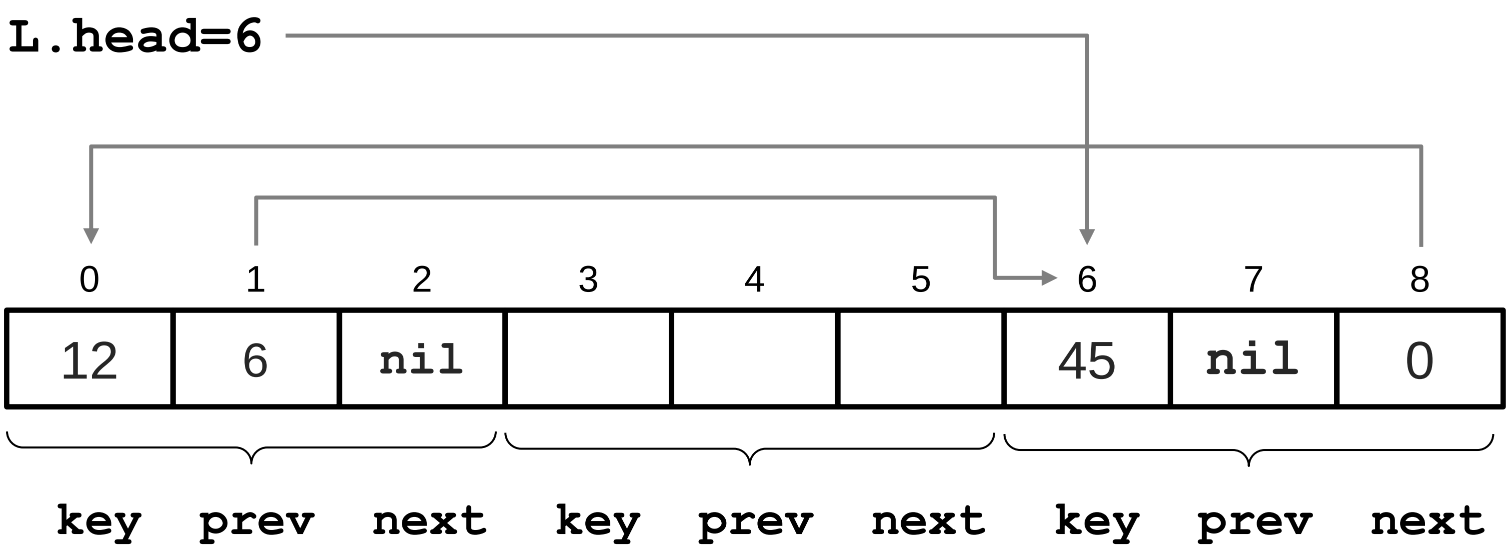
`head`

zeigt auf erstes Element (bzw. ist **`nil`** für leere Liste)

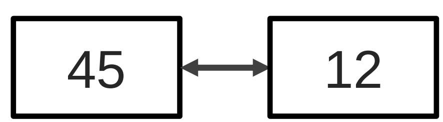
Jedes Element **`x`** besteht aus:
`key` – Wert
`prev` – Zeiger auf Vorgänger (bzw. **`nil`**)
`next` – Zeiger auf Nachfolger (bzw. **`nil`**)

evtl. schon als Datenstruktur implementiert, oder aber…

Datenstruktur Verkettete Listen durch Arrays



entspricht doppelt verketteter Liste

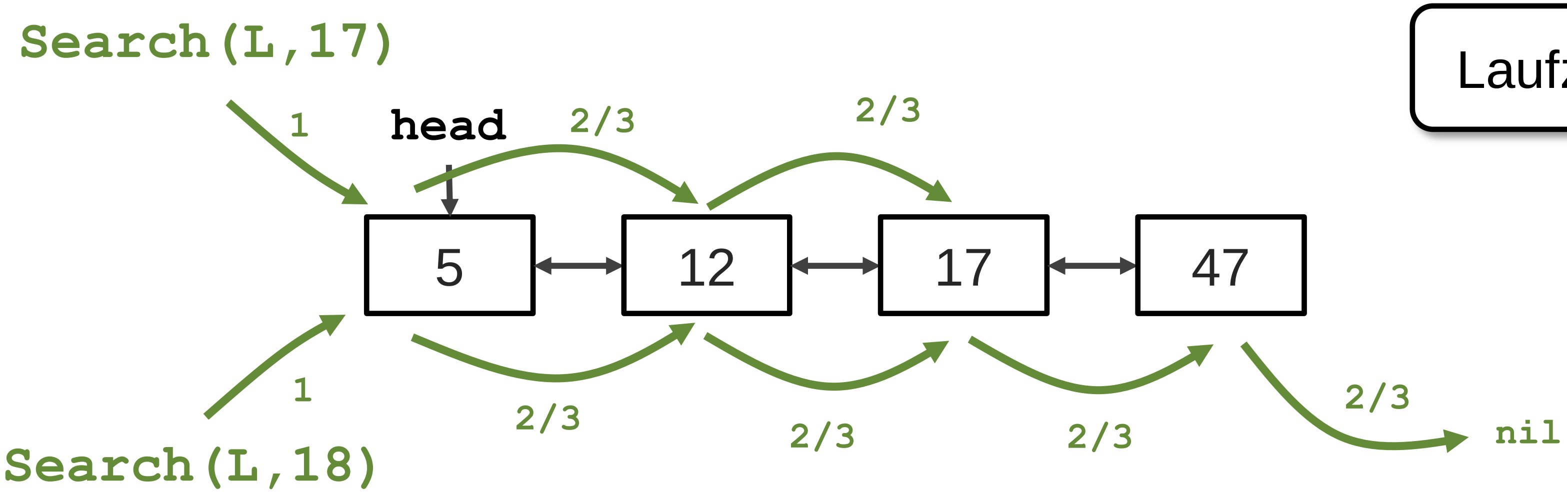


Elementare Operationen auf verketteten Listen

```
search(L,k)   //returns pointer to k in L (or nil)

current=L.head;

WHILE current != nil AND current.key != k DO
    current=current.next;
return current;
```

short circuit evaluation (wie in Java)

Laufzeit= $\Theta(n)$

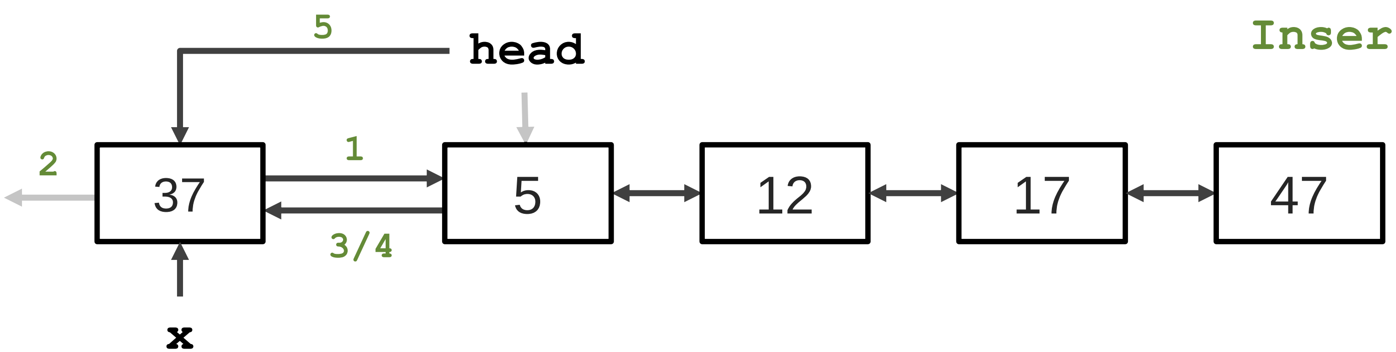


Elementare Operationen auf verketteten Listen

```
insert(L,x)    //inserts element x in L

x.next=L.head;
x.prev=nil;
IF L.head != nil THEN
   L.head.prev=x;
L.head=x;
```

call-by-reference
bzw. call-by-value
für Objekte wie in Java

Laufzeit= $\Theta(1)$



Insert(L,x)

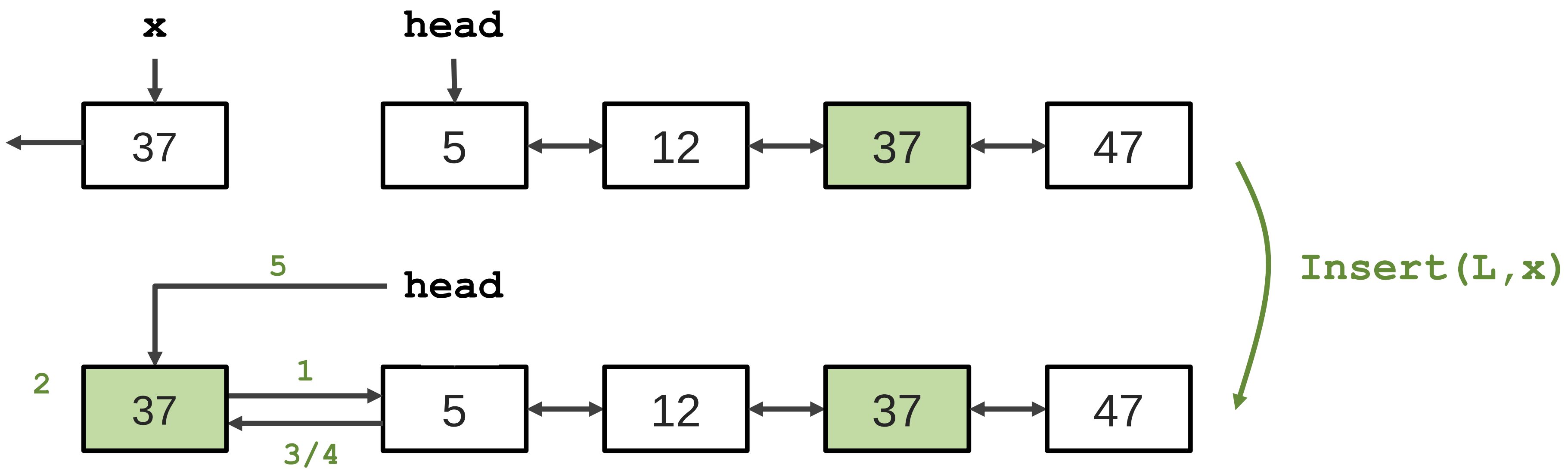
TECHNISCHE UNIVERSITÄT DARMSTADT

Cryptoplexity
Cryptography & Complexity Theory
Technische Universität Darmstadt
www.cryptoplexity.de

Elementare Operationen auf verketteten Listen

Achtung: Einfüge-Operation prüft nicht, ob Wert bereits in Liste

Wenn zuerst Suche nach Wert, dann wiederum Laufzeit Ω(n)!

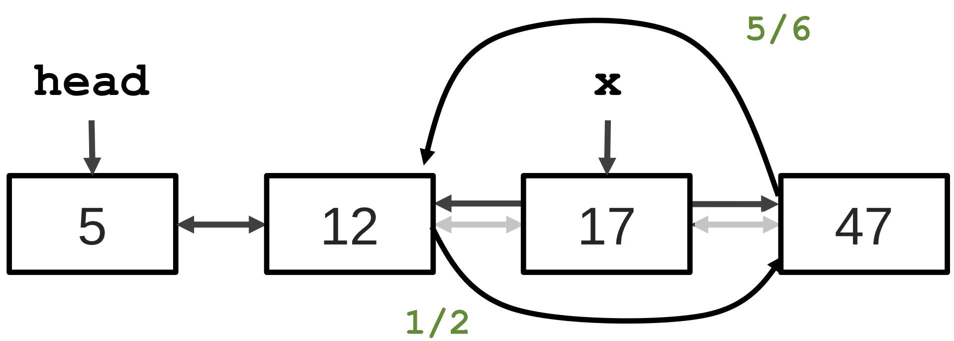


Elementare Operationen auf verketteten Listen

```
delete(L,x)    //deletes element x from L

1  IF x.prev != nil THEN
2        x.prev.next=x.next
3  ELSE
4        L.head=x.next;
5  IF x.next != nil THEN
6        x.next.prev=x.prev;
```

Laufzeit= $\Theta(1)$

Achtung: Löschen eines **Wertes** `k` kostet Zeit $\Omega(n)$

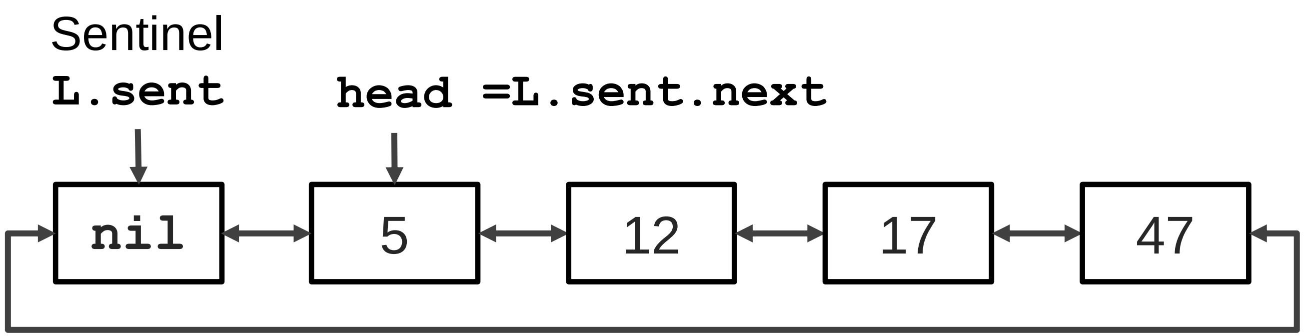


`Delete(L,x)`

Vereinfachung per Wächter/Sentinels

```
delete(L,x)   //deletes element x from L

IF x.prev != nil THEN
    x.prev.next=x.next
ELSE
    L.head=x.next;
IF x.next != nil THEN
    x.next.prev=x.prev;
```

Ziel:
eliminiere die
Spezialfälle für
Listenanfang/-ende



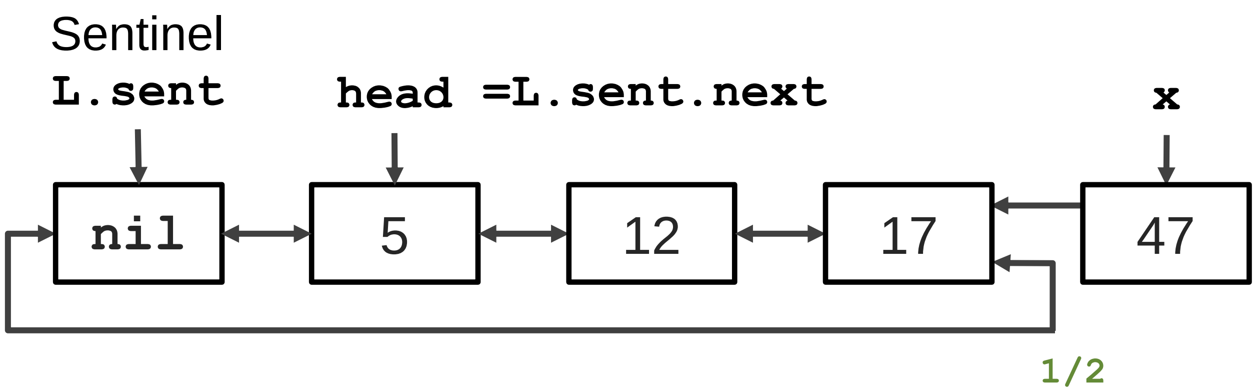
Sentinel ist „von außen“ nicht sichtbar

Leere Liste besteht nur aus Sentinel

Löschen mit Sentinels

```
deleteSent(L,x)
      // deletes x from L with sentinel

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

Andere Operationen
wie Einfügen
und Löschen
müssen auch
angepasst werden

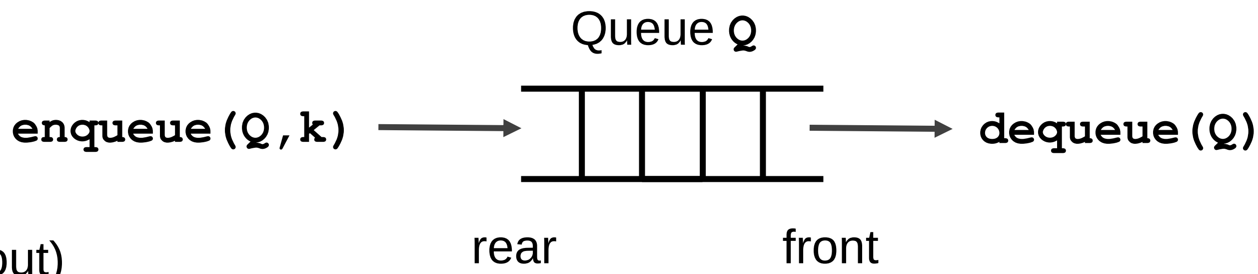


Delete(L,x)

Queues

Abstrakter Datentyp Queue

`new(Q)`	-	erzeugt neue (leere) Queue namens **`Q`**
`isEmpty(Q)`	-	gibt an, ob Queue **`Q`** leer
`dequeue(Q)`	-	gibt vorderstes Element der Queue **`Q`** zurück und löscht es aus Queue (bzw. Fehlermeldung, wenn Queue leer)
`enqueue(Q,k)`	-	schreibt **`k`** als neues hinterstes Element auf **`Q`** (bzw. Fehlermeldung, wenn Queue voll)



(FIFO – first in, first out)

Queues als Array? (I)

Wo ist **`front`**, wo **`rear`**?

	0	1	2	3	4	5	6	7
`Q`		17	98	9	23	37	8	47

`Q.rear`

zu Beginn auch rechts,
wandert dann nach links
beim Einfügen

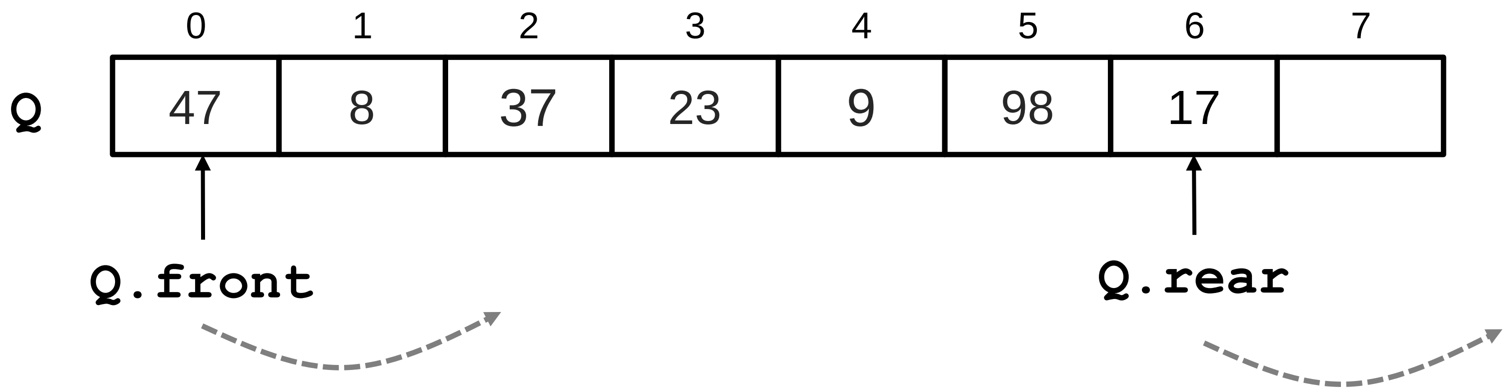
`Q.front`

zu Beginn ganz rechts,
wandert dann nach links
beim Auswerfen

Problem:
Selbst wenn maximale Anzahl Elemente, die gleichzeitig in der Queue sind, vorher bekannt, kann **`Q.rear`** die Array-Grenze links erreichen

Queues als Array? (II)

Wo ist **`front`**, wo **`rear`**?



zu Beginn ganz links,
wandert dann nach rechts
beim Auswerfen

zu Beginn auch links,
wandert dann nach rechts
beim Einfügen

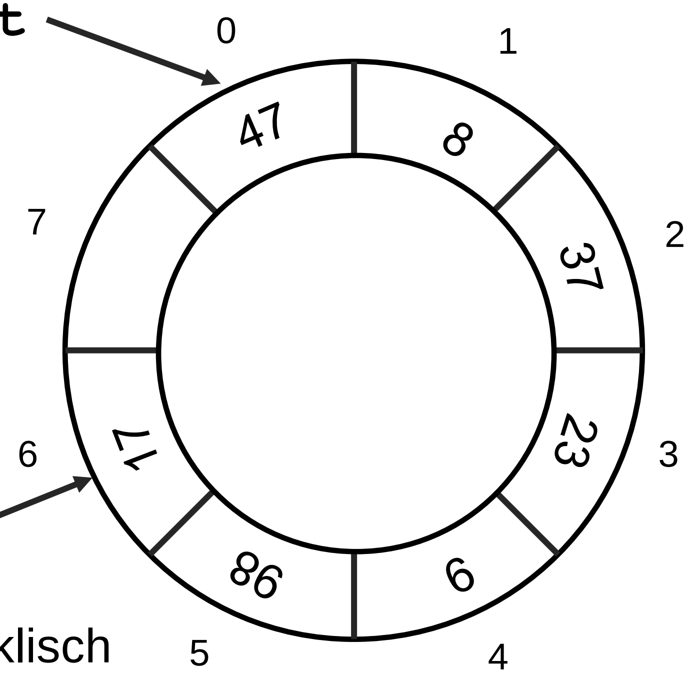
Problem:
Selbst wenn Array nach rechts unendlich lang,
wird Speicher links von **`Q.front`** verschwendet

Queues als (virtuelles) zyklisches Array

MAX Elemente
gleichzeitig in Queue

Q.front

wandert zyklisch
nach rechts beim
Auswerfen



nur virtuell

Q.rear

wandert zyklisch
nach rechts beim
Einfügen

Zeiger muss beim
Wandern nach rechts
von 7 auf 0 springen
(und links von 0 auf 7)

0	1	2	3	4	5	6	7
47	8	37	23	9	98	17	

real

Modulo-Operator

Modulo-Operator $\boldsymbol{x \; mod \; n}$ für $\boldsymbol{n > 0}$

Der Modulo-Operator $x \; mod \; n$ bildet eine ganze Zahl x auf die Zahl y zwischen 0 und $n - 1$ ab, so dass $y = x - i \cdot n$ für eine ganze Zahl i.

Beispiele:

$15 \bmod 5 = 0,$ weil $0 = 15 - 3 \cdot 5$

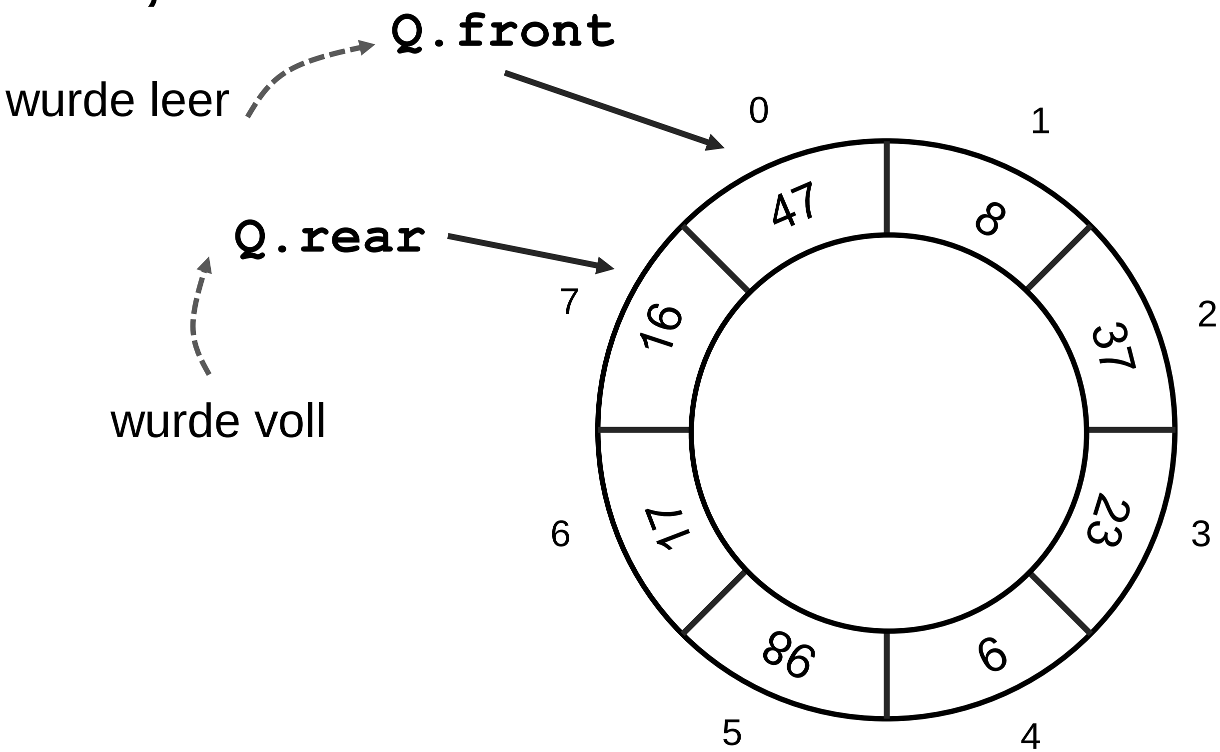
$(6 + 7) \; mod \; 8 = 5,$ weil $5 = 13 - 1 \cdot 8$

$-4 \; mod \; 7 = 3,$ weil $3 = -4 + 1 \cdot 7$

Achtung: In Java ist der %-Operator als Divisionsrest für negative Zahlen x anders definiert, dort ist z.B. $-4 \,\%\, 7 = -4$. Man kann dies unter Beachtung eventueller Überläufe abbilden durch $x \; mod \; n = ((x \,\%\, n) + n) \,\%\, n$.

Ein Bit, bitte

MAX Elemente
gleichzeitig in Queue



Ist das eine leere Schlange oder eine volle Schlange?

Speichere diese Information in Boolean **`empty`**
(alternativ: reserviere ein Element des Arrays als „Abstandshalter“)

Queues als zyklisches Array: Algorithmen

`Q` leer, wenn
`front==rear+1 mod MAX`
und **`empty==true`**

`Q` voll, wenn
`front==rear+1 mod MAX`
und **`empty==false`**

```
new(Q)

Q.A[]=ALLOCATE(MAX);
Q.front=0;
Q.rear=-1;
Q.empty=true;
```

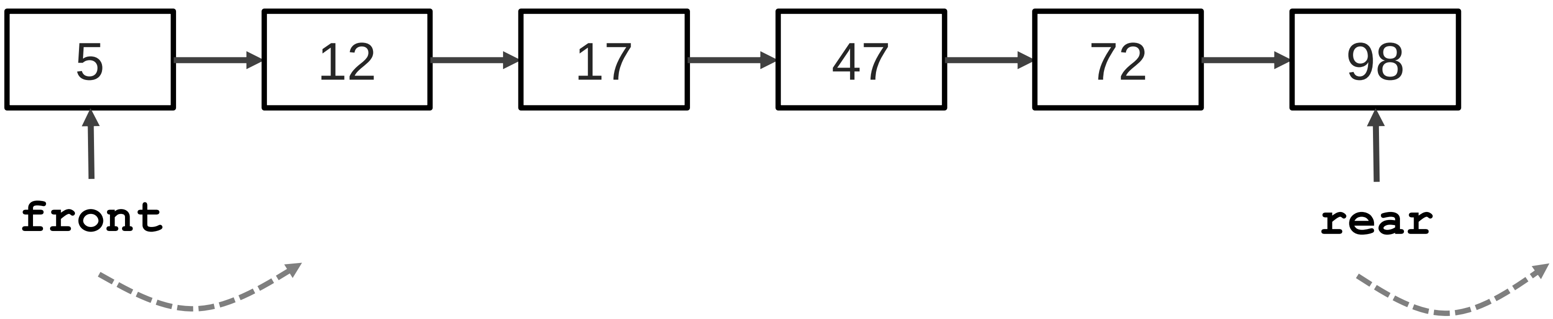
```
isEmpty(Q)

return Q.empty;
```

```
dequeue(Q)

IF isEmpty(Q) THEN
  error 'underflow'
ELSE
  Q.front=Q.front+1 mod MAX;
  IF Q.front==Q.rear+1 mod MAX
     THEN Q.empty=true;
  return Q.A[Q.front-1 mod MAX];
```

```
enqueue(Q,k)

IF Q.front==Q.rear+1 mod MAX
                AND !Q.empty THEN
  error 'overflow'
ELSE
  Q.rear=Q.rear+1 mod MAX;
  Q.A[Q.rear]=k;
  Q.empty=false;
```

Queues durch einfach (!) verkettete Listen

(einfach) verkettete Liste



wandert nach rechts
beim Auswerfen

wandert auch nach
rechts weiter
beim Einfügen

Queues durch Liste: Algorithmen

```
new(Q)

Q.front=nil;
Q.rear=nil;
```

```
isEmpty(Q)

IF Q.front==nil THEN
   return true
ELSE
   return false;
```

```
dequeue(Q)

IF isEmpty(Q) THEN
   error ‘underflow‘
ELSE
   x=Q.front;
   Q.front=Q.front.next;
   return x;
```

```
enqueue(Q,x)

IF isEmpty(Q) THEN
    Q.front=x;
ELSE
    Q.rear.next=x;
x.next=nil;
Q.rear=x;
```

*Worst Case

Anzahl Operationen

Stack

Operation	Laufzeit*
Push	$\Theta(1)$
Pop	$\Theta(1)$

Queue

Operation	Laufzeit*
Enqueue	$\Theta(1)$
Dequeue	$\Theta(1)$

Verkettete Liste

Operation	Laufzeit*
Einfügen	$\Theta(1)$
Löschen	$\Theta(1)$
Suchen	$\Theta(n)$

Laufzeit Löschen eines Wertes $\Omega(n)$

Geben Sie die Suchoperation für einen Wert **`k`** bei einer Liste mit Sentinels an.

Wie kann man mit Hilfe zweier Stacks eine Queue implementieren?

Wie kann man mit Hilfe zweier Queues einen Stack implementieren?

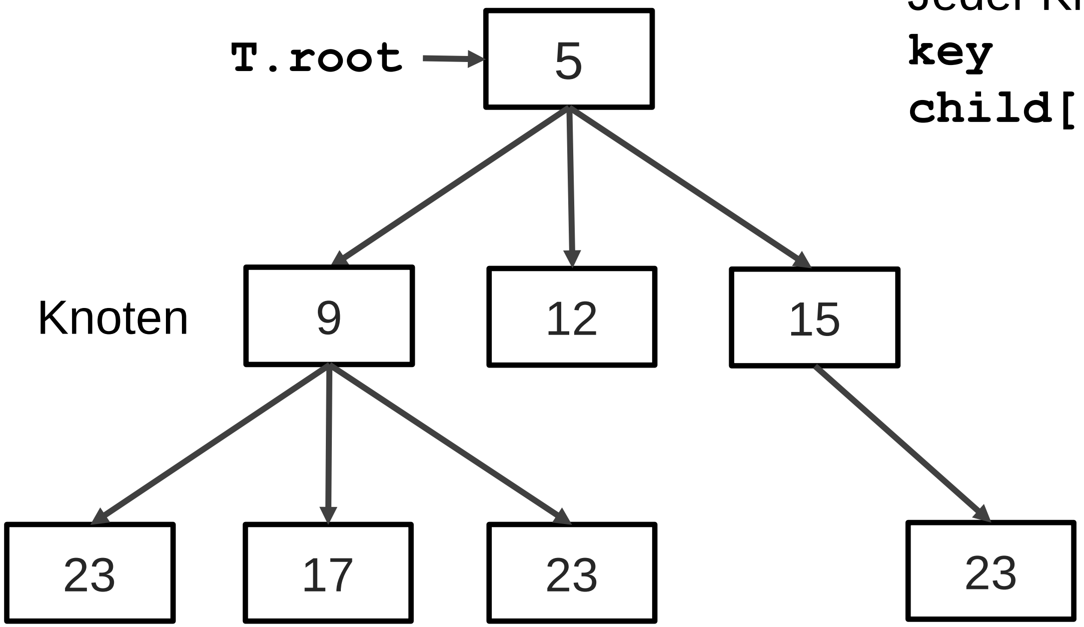
Binäre Bäume

Verkettete Liste

Operation	Laufzeit*
Einfügen	$\Theta(1)$
Löschen	$\Theta(1)$
Suchen	$\Theta(n)$

Geht das besser?

Bäume durch verkettete Listen



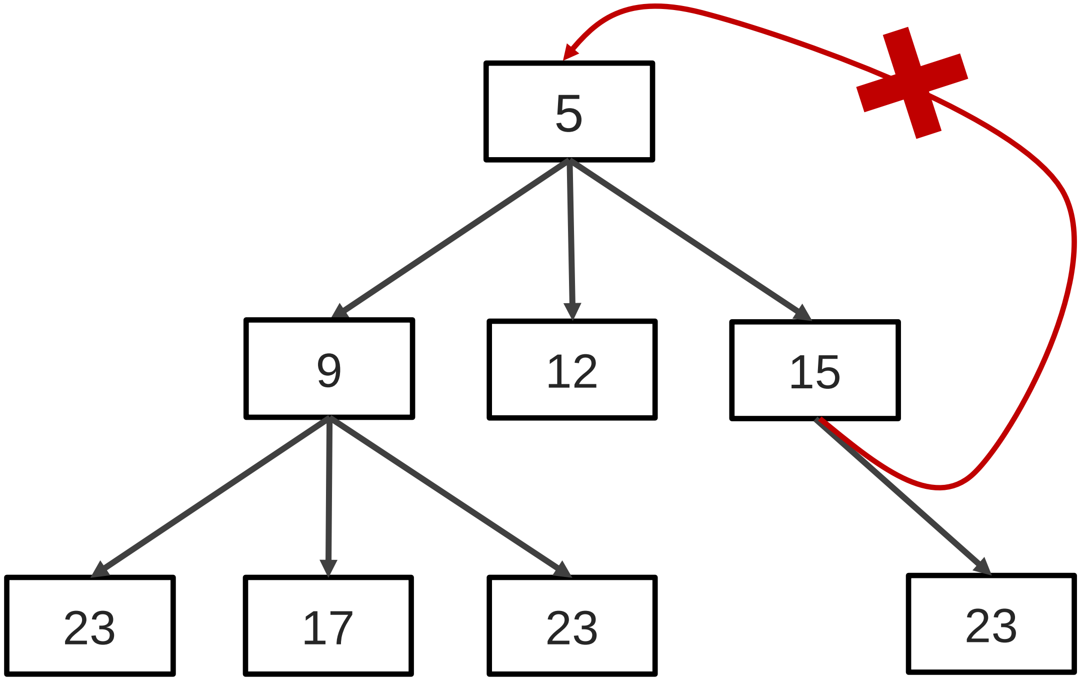
Jeder Knoten enthält:
`key` - Wert
`child[]` - Array von Zeigern auf Kinder

manchmal auch nützlich:
`parent` - Zeiger auf Elternknoten

Baum-Bedingung: Baum ist leer oder…
es gibt einen Knoten **`r`** („Wurzel“), so dass jeder Knoten **`v`** von der Wurzel aus per eindeutiger Sequenz von **`child`**-Zeigern erreichbar ist:

`v = r.child[i1].child[i2].….child[im]`

Eigenschaften von Bäumen

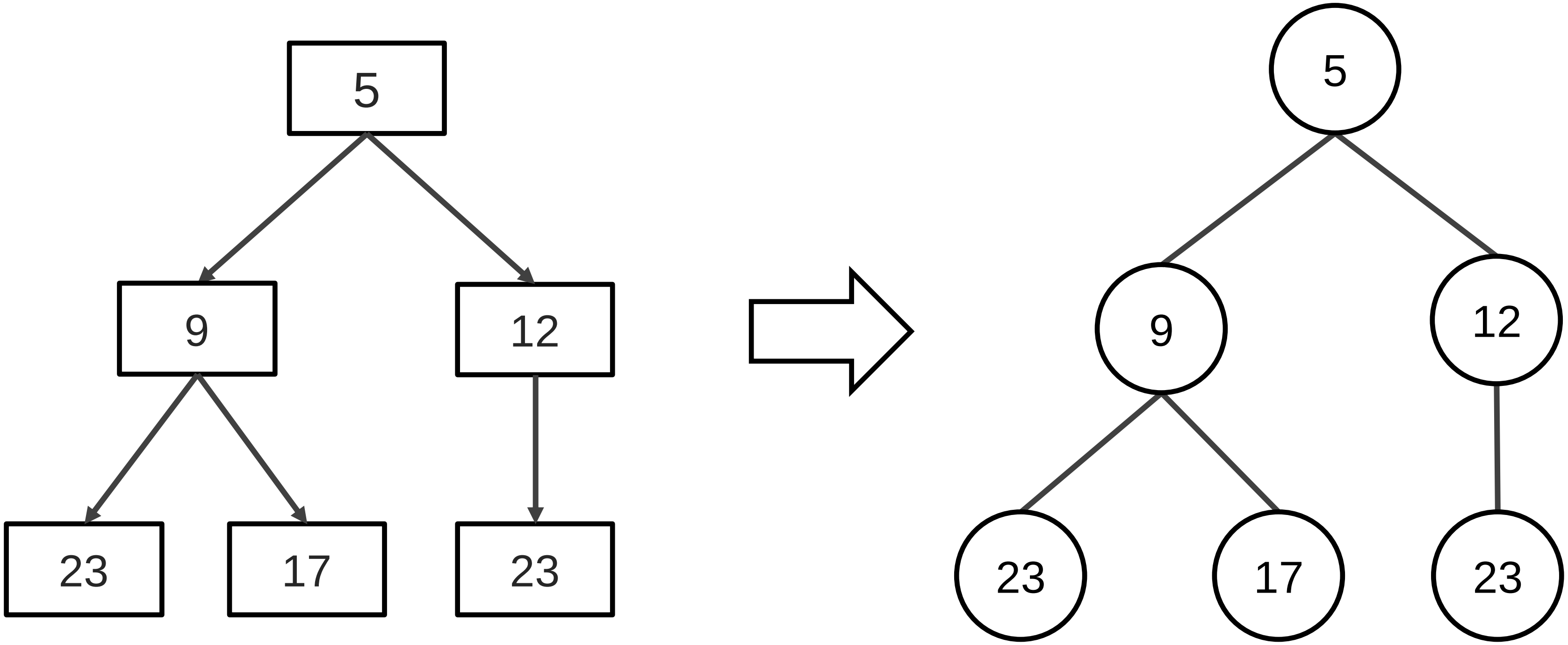


Bäume sind „azyklisch“

Für nicht-leeren Baum gibt es genau $\#Knoten - 1$ viele Einträge $\neq$**`nil`** über alle Listen **`child[]`**

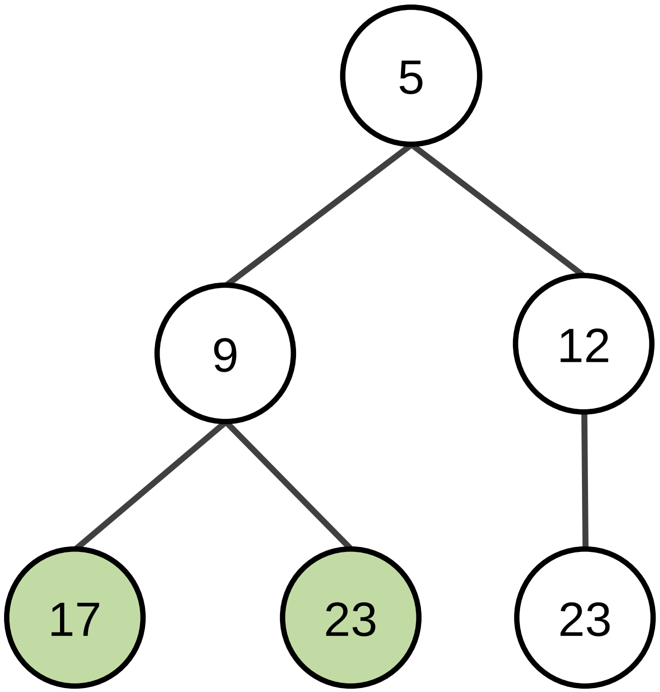
Darstellung als (ungerichteter) Graph

später mehr zu Graphen



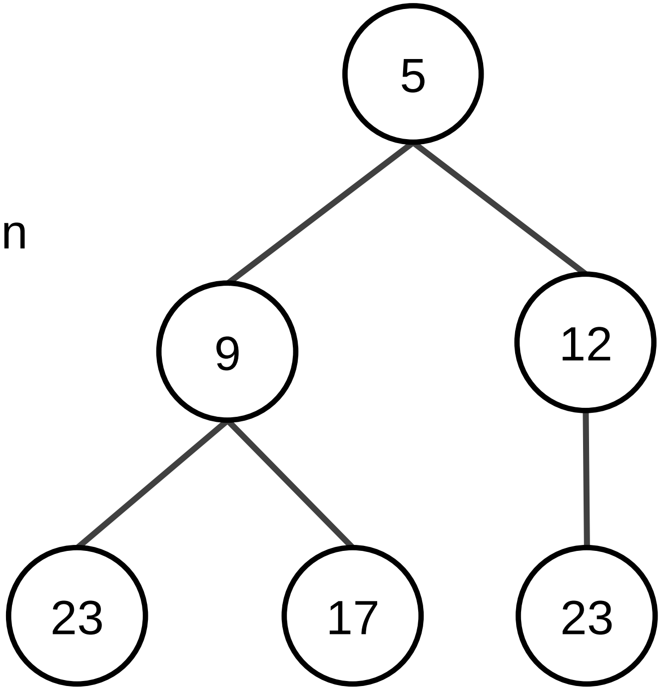
Achtung: in beiden Darstellungen ist die Reihenfolge
in **`child[]`** quasi durch die Anordnung der Knoten dargestellt

Darstellung als (ungerichteter) Graph



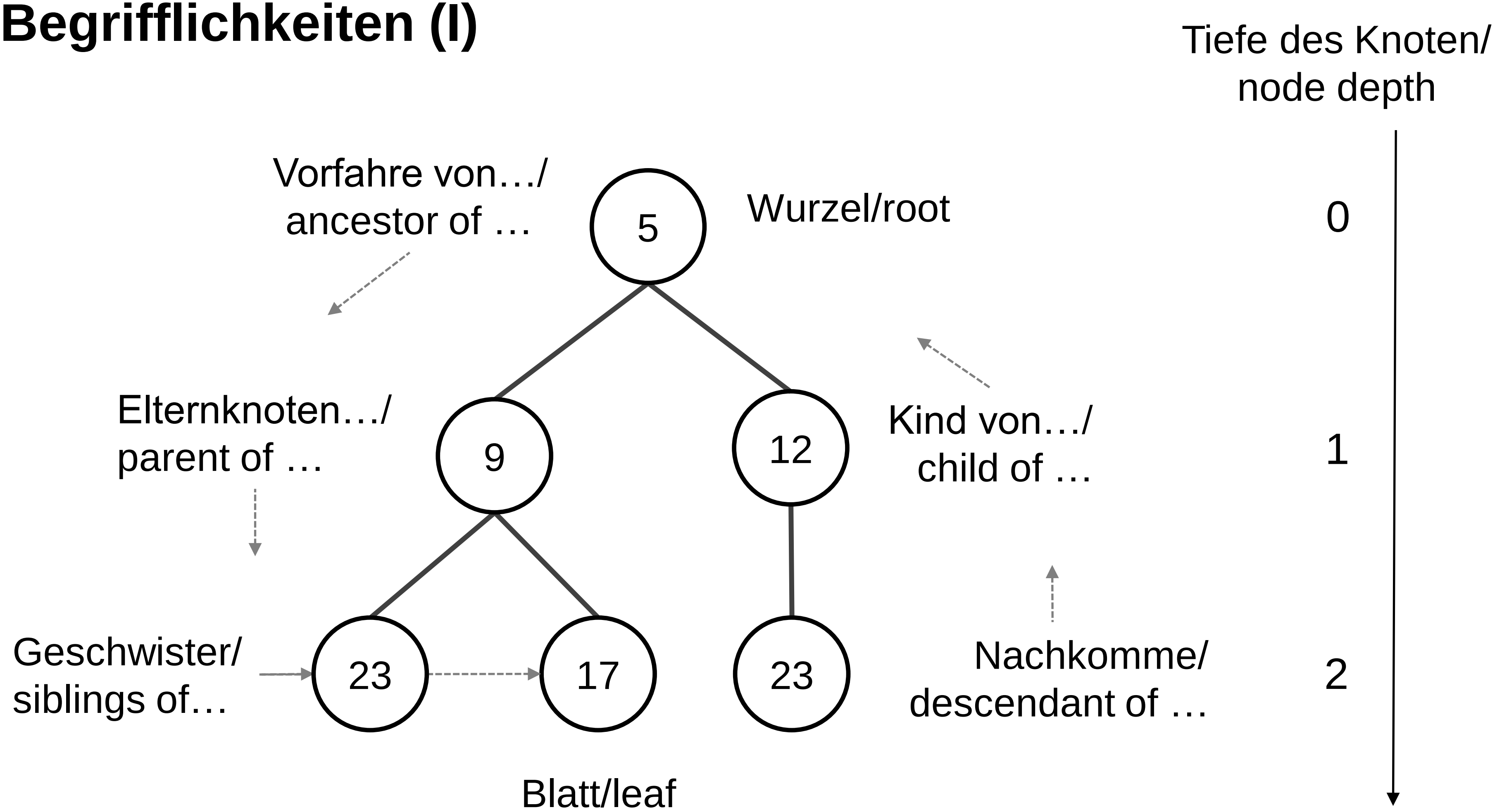
als Baum verschieden

≠



Achtung: in beiden Darstellungen ist die Reihenfolge
in **`child[]`** quasi durch die Anordnung der Knoten dargestellt

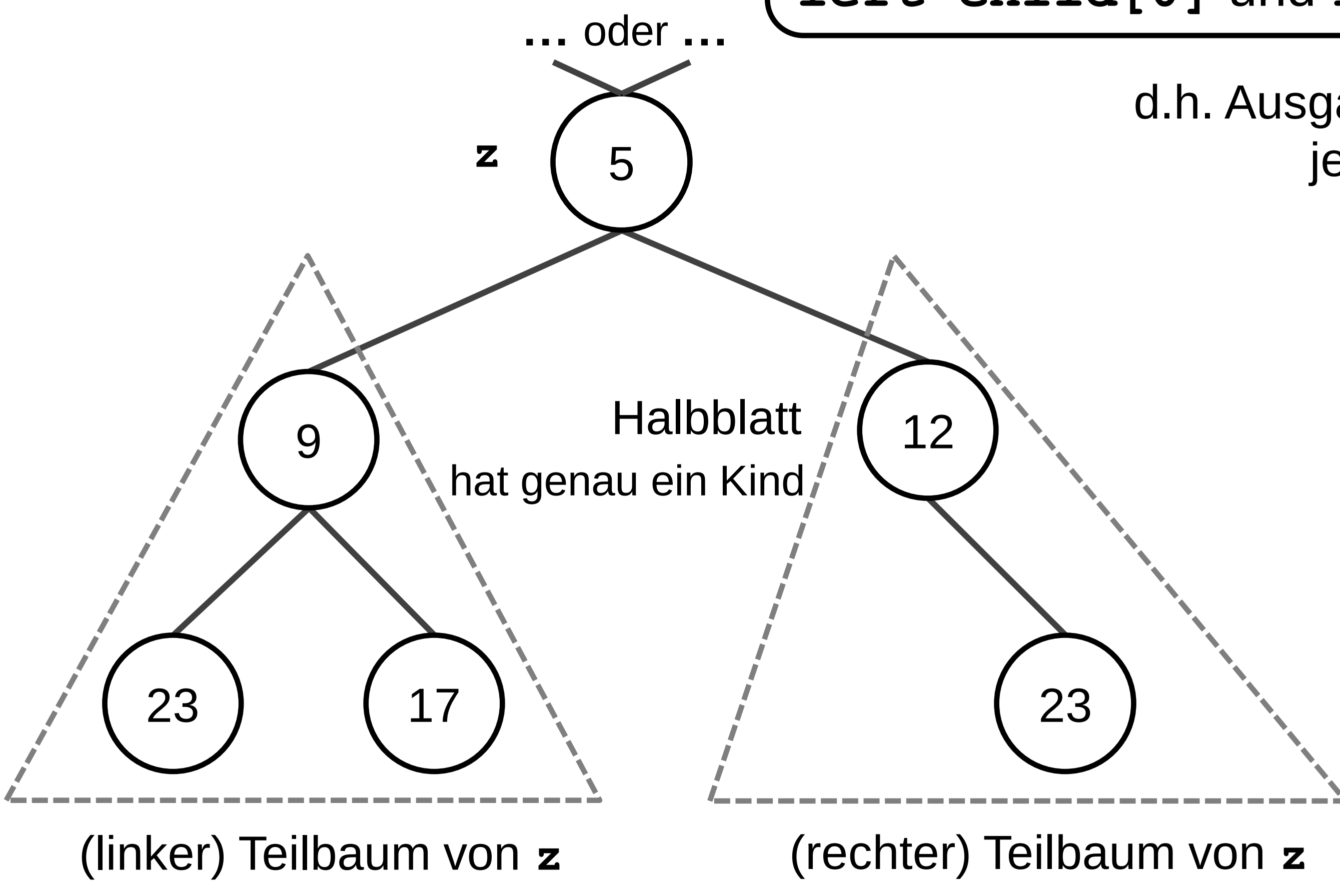
Begrifflichkeiten (I)



Höhe des Baumes/ tree height
= maximale Tiefe eines Knoten

Begrifflichkeiten (II)

Binärbaum:
jeder Knoten hat maximal zwei Kinder,
`left=child[0]` und **`right=child[1]`**



d.h. Ausgangsgrad/outdegree jedes Knotens ist ≤ 2

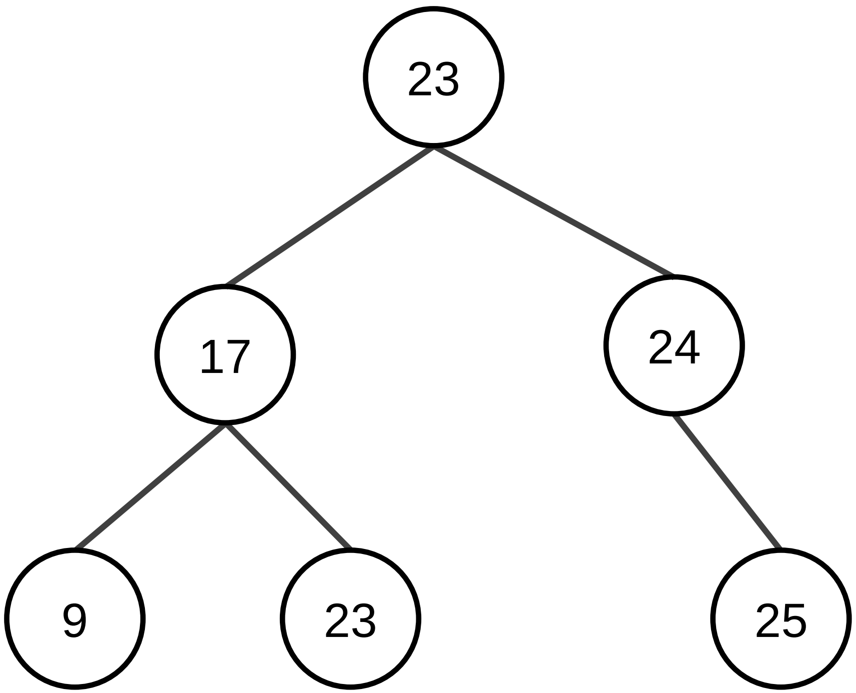
markiere Knoten auch graphisch als linkes oder rechtes Kind

Höhe leerer Baum hier per Konvention -1

Höhe (nicht-leerer) Baum = max { Höhe aller Teilbäume der Wurzel } +1

Inorder-Traversieren von Binärbäumen

Beispielanwendung:
Serialisierung



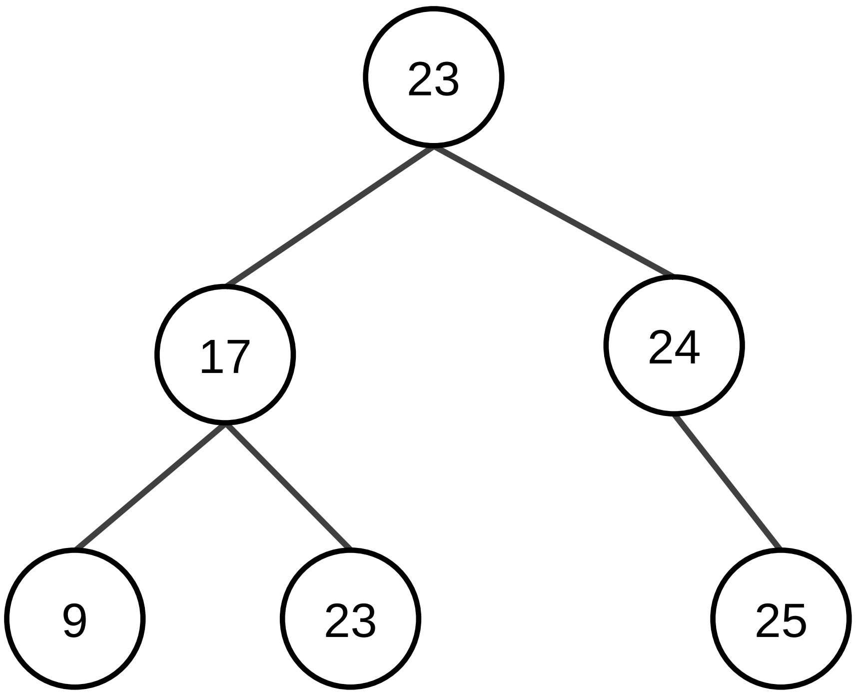

```
inorder(x)

IF x != nil THEN
     inorder(x.left);
     print x.key;
     inorder(x.right);
```

Bei Bedarf mit „Wrapper"
`inorderTree(T)=inorder(T.root)`

`inorder(T.root)` ergibt

9 17 23 23 24 25

Inorder-Traversieren von Binärbäumen: Laufzeit




```
inorder(x)

IF x != nil THEN
    inorder(x.left);
    print x.key;
    inorder(x.right);
```

$T(n)$ = Laufzeit bei n Knoten

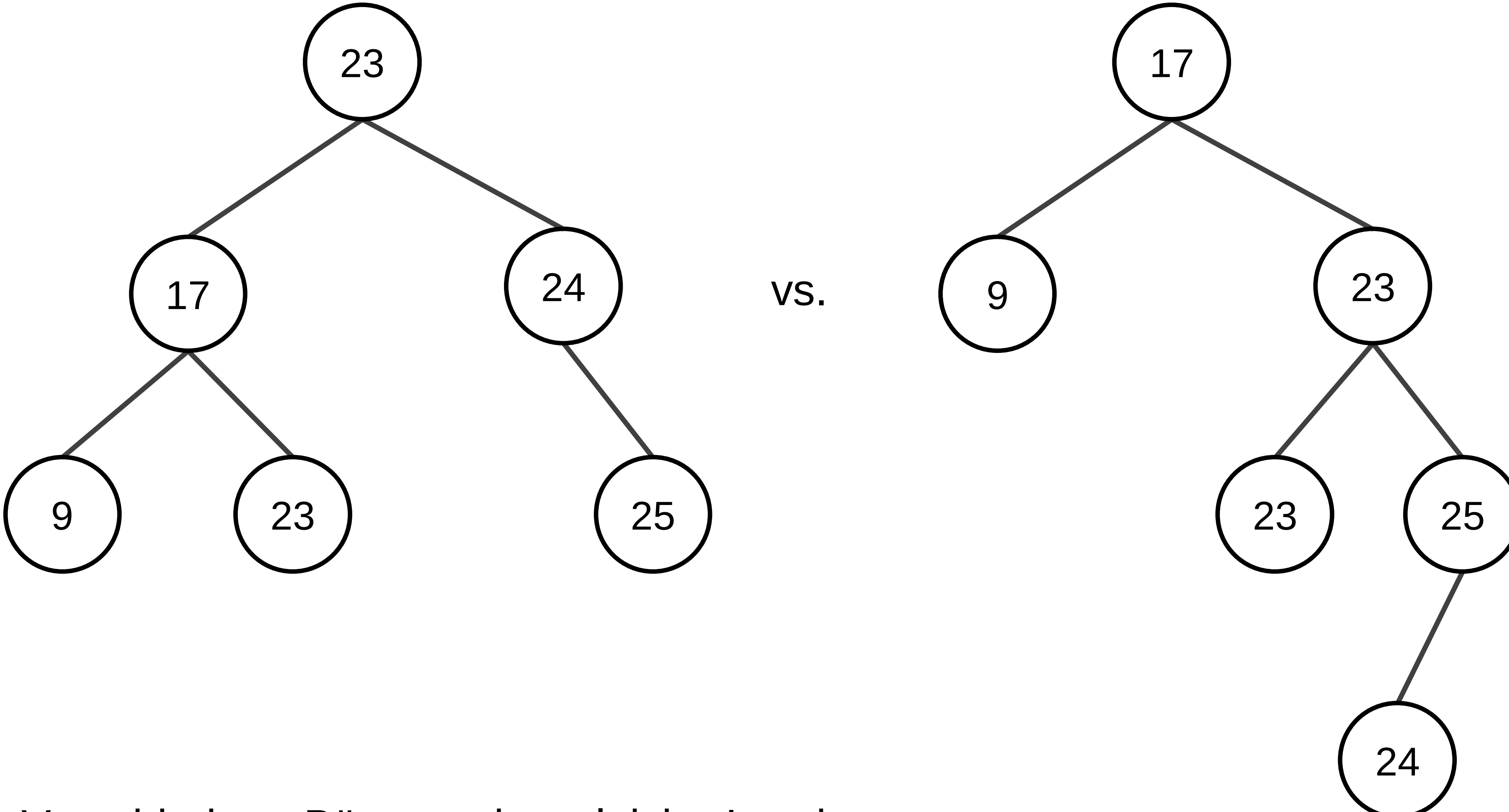
Behauptung: $T(n) = O(n)$,
genauer $T(n) \leq (c+d)n + c$

Gilt mit $T(0) = c$ bei leerem Baum

Rekursion mit k Knoten im linken Teilbaum und $n-k-1$ im rechten:

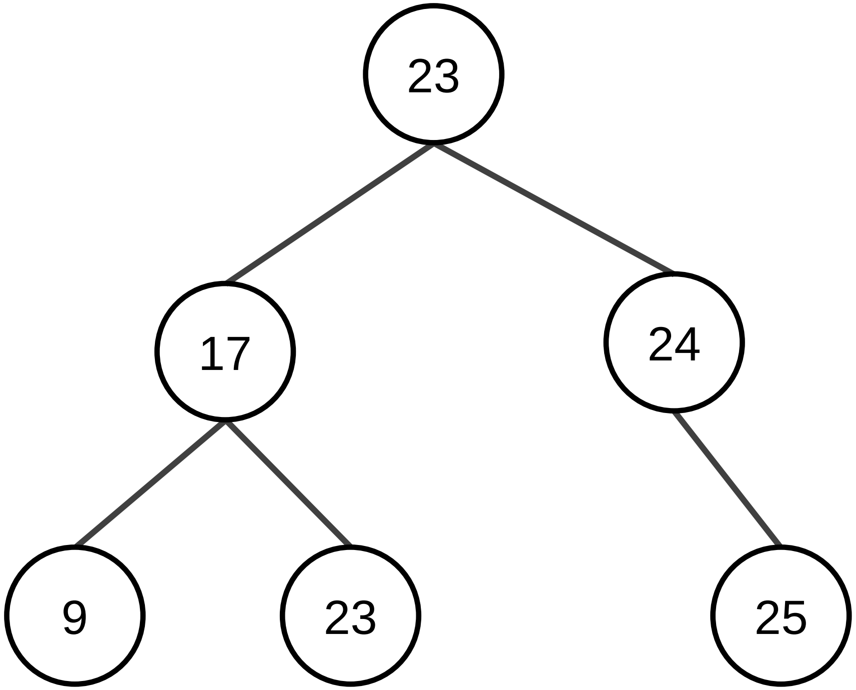
$$T(n) = T(k) + T(n-k-1) + d$$
$$\leq (c+d)k + c \;+\; (c+d)(n-k-1) + c \;+ d \;\leq (c+d)n + c$$

Inorder $\not\Rightarrow$ Baum



Verschiedene Bäume, aber gleiche Inorder

Pre- und Postorder-Traversieren von Binärbäumen (I)

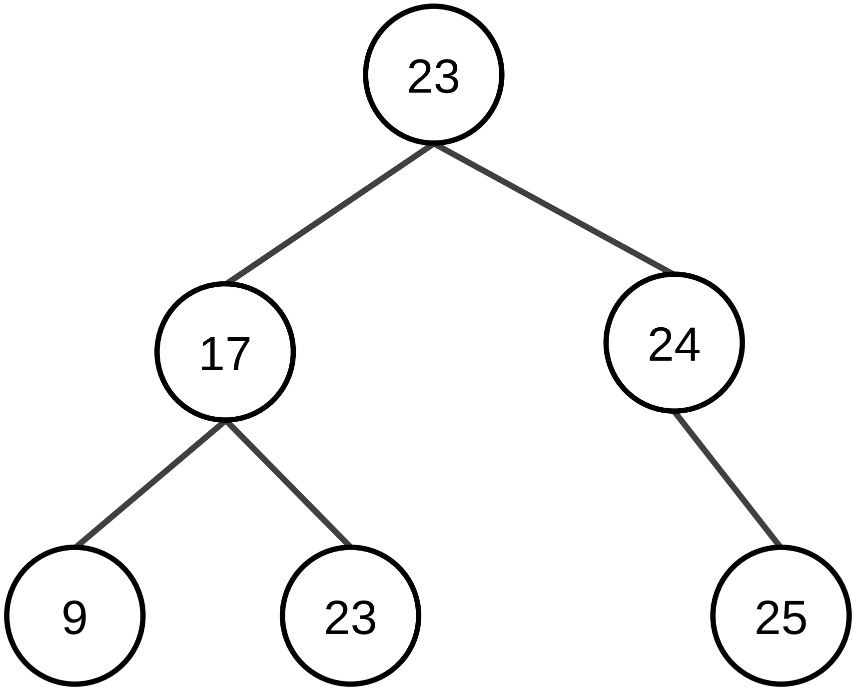



```
preorder(x)

IF x != nil THEN
      print x.key;
      preorder(x.left);
      preorder(x.right);
```

`preorder(T.root)` ergibt

23 17 9 23 24 25

Pre- und Postorder-Traversieren von Binärbäumen (II)




```
preorder(x)

IF x != nil THEN
      print x.key;
      preorder(x.left);
      preorder(x.right);
```

```
postorder(x)

IF x != nil THEN
      postorder(x.left);
      postorder(x.right);
      print x.key;
```

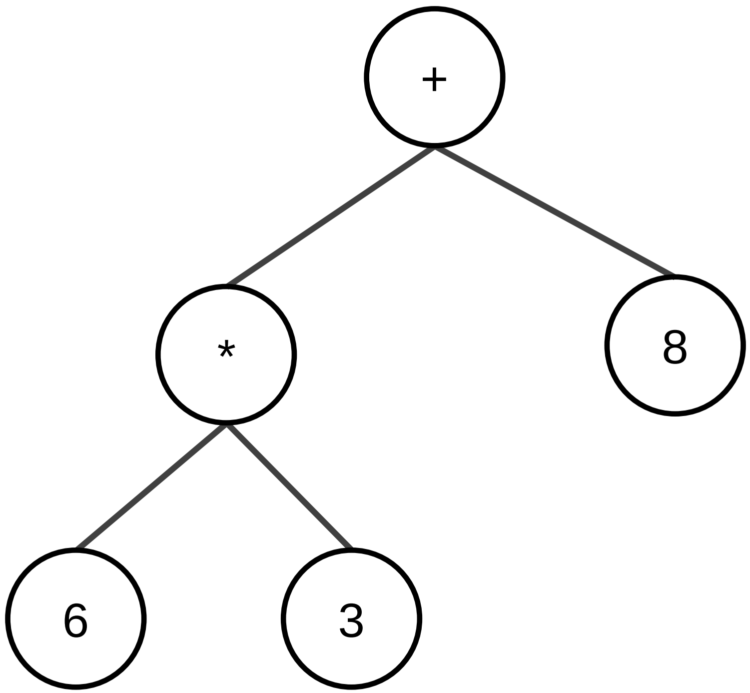
`preorder(T.root)` ergibt

23 17 9 23 24 25

`postorder(T.root)` ergibt

9 23 17 25 24 23

Beispiel Preorder-Traversierung




```
preorder(x)

IF x != nil THEN
      print x.key;
      preorder(x.left);
      preorder(x.right);
```

`preorder(T.root)` ergibt

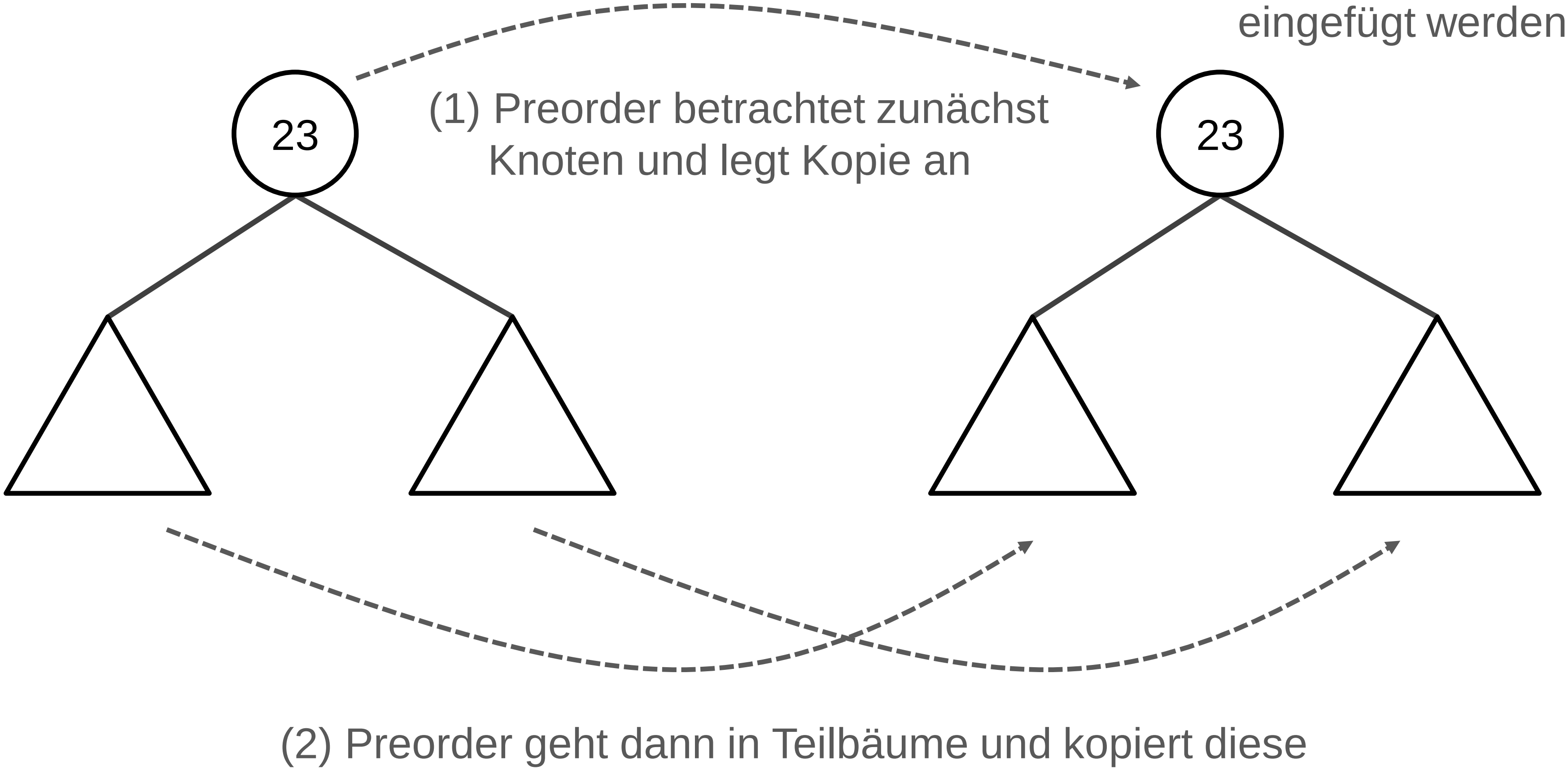
(+ (* 6 3) 8)

`inorder(T.root)` ergibt

6 * 3 + 8

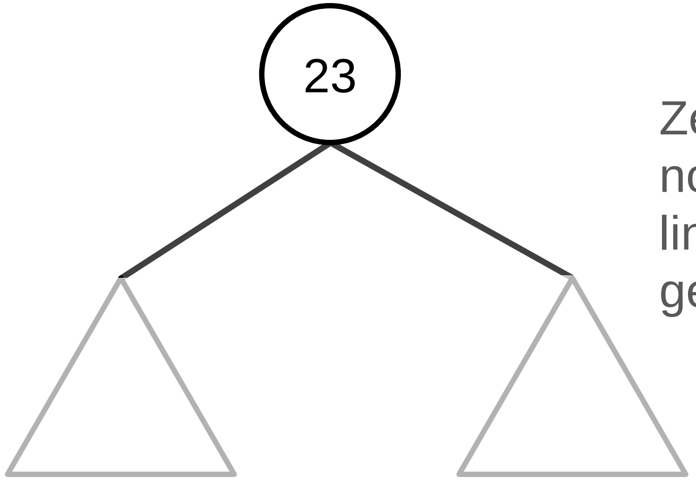
Preorder-Traversieren für Kopieren

Unterbäume können in kopierten Knoten eingefügt werden



Postorder-Traversieren für Löschen

(2) Postorder betrachtet Knoten erst danach und löscht dann den kompletten Knoten

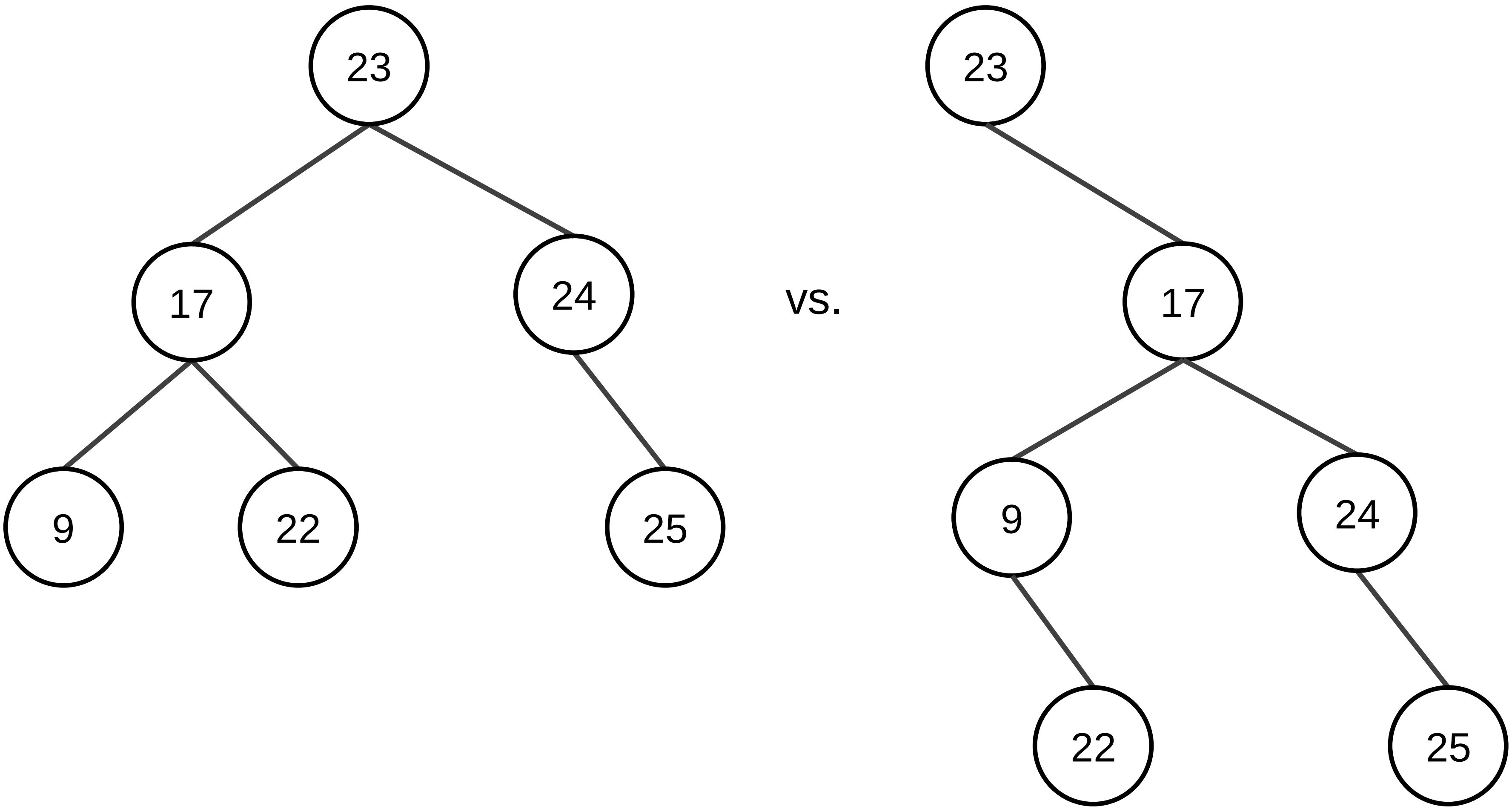


Zeiger auf rechten Teilbaum noch vorhanden, wenn linker Teilbaum bereits gelöscht wurde

(1) Postorder geht zuerst in Teilbäume und löscht diese

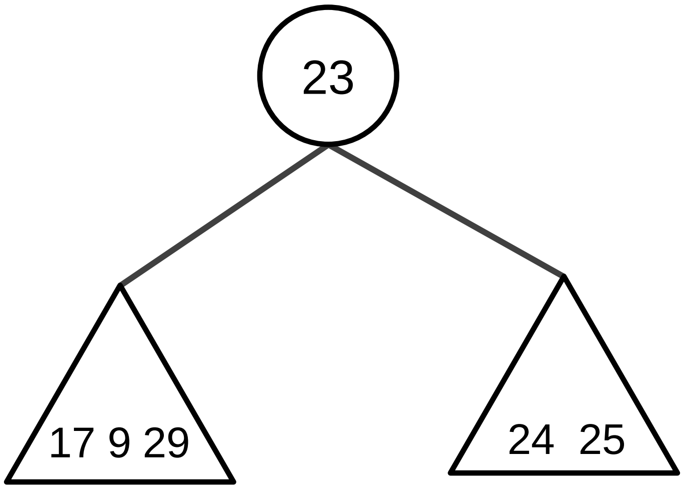
Preorder $\not\Rightarrow$ Binärbaum

gilt entsprechend auch für Postorder



Verschiedene Bäume, aber gleiche Preorder

Preorder + Inorder + eindeutige Werte ⇒ Binärbaum



Pre = 17 9 29
In = 9 17 29

Pre = 24 25
In = 24 25

Bilde Teilbäume rekursiv

Preorder = 23 17 9 29 24 25

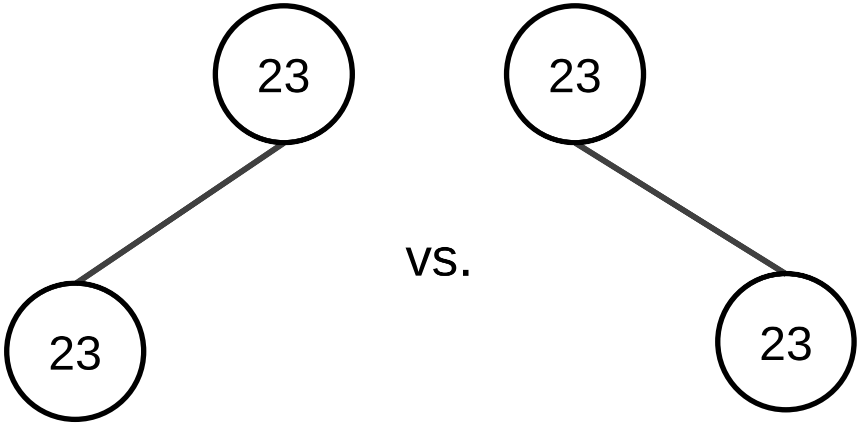
(1) Identifiziert Wurzel

Inorder = 9 17 29 23 24 25

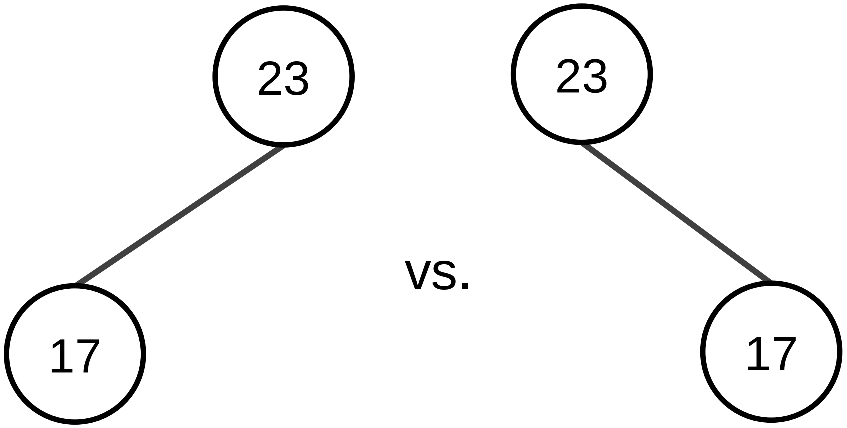
(2) Identifiziert Werte im
linken und rechten Teilbaum

Gilt analog für Postorder

Inorder und eindeutige Werte sind notwendig



Haben gleiche
Pre-, Post- und Inorder



Haben gleiche
Pre- und Postorder

Zeigen Sie: In einem nicht-leerem Binärbaum mit n Knoten gibt es genau $n+1$ viele Einträge **`child[i]=nil`**.

Geben Sie zu dem linken Baum auf Folie 55 (Preorder$\not\Rightarrow$Binärbaum) einen anderen Baum mit gleicher Postorder an.

Abstrakter Datentyp Baum

`new(T)` - erzeugt neuen Baum namens **`T`**

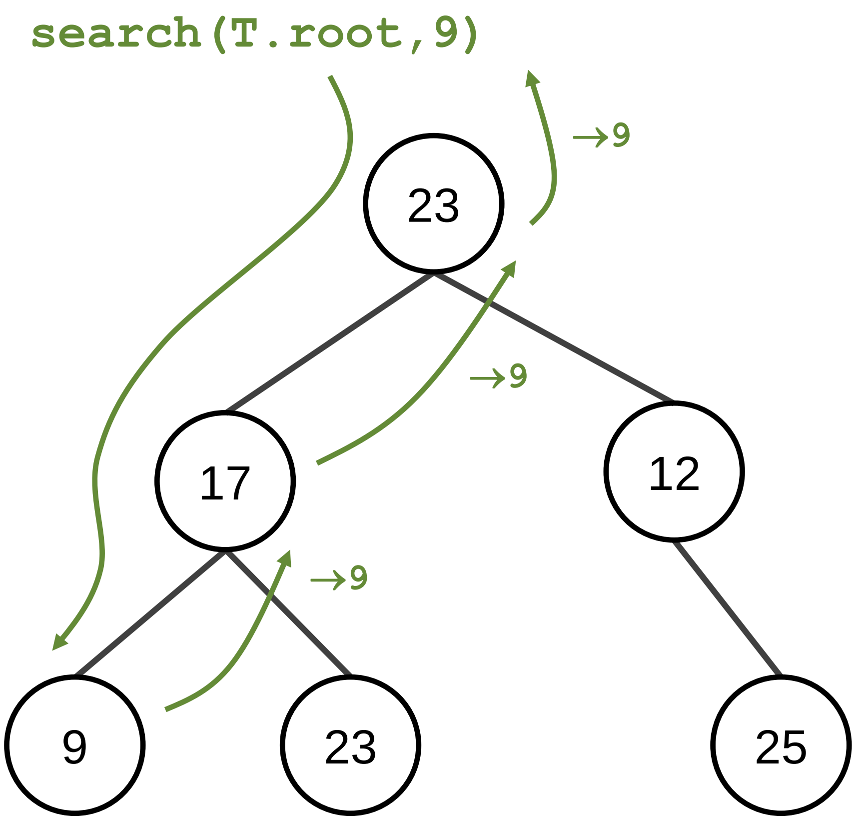
`search(T,k)` - gibt Element **`x`** in Baum **`T`** mit **`x.key==k`** zurück (bzw. **`nil`**)

`insert(T,x)` - fügt Element **`x`** in Baum **`T`** hinzu

`delete(T,x)` - löscht **`x`** aus Baum **`T`**

oft weitere Baum-Operationen wie Wurzel, Höhe, Traversieren,….

Suchen



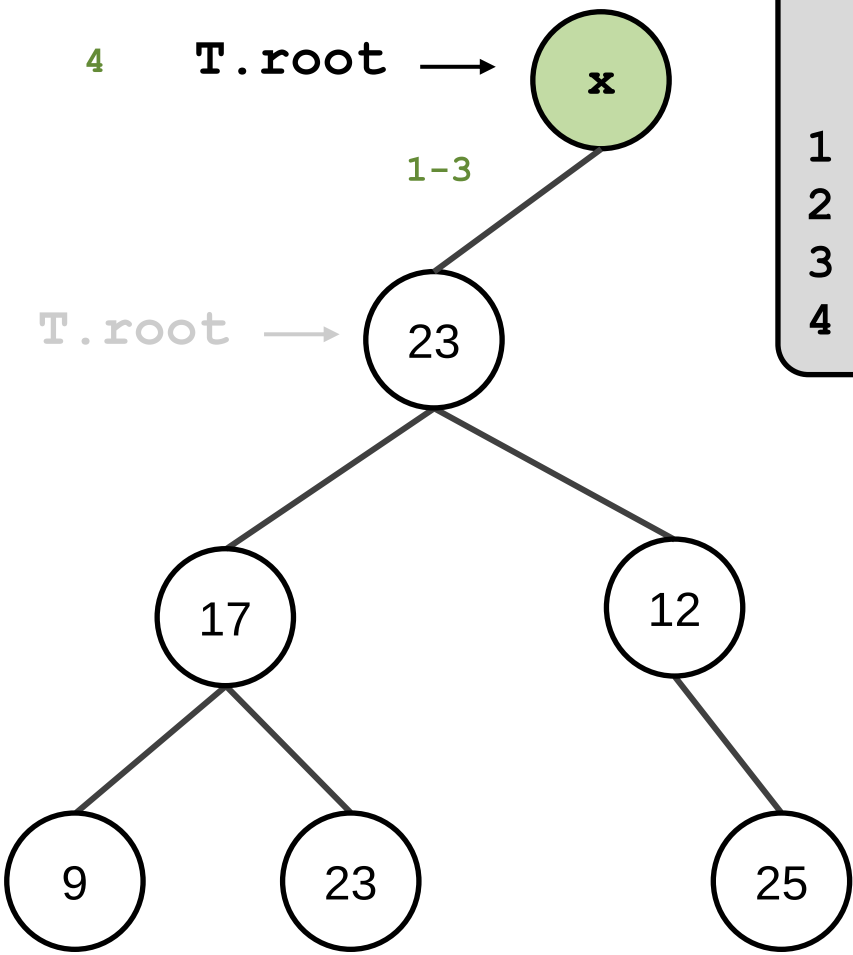

```
search(x,k)

IF x==nil THEN return nil;
IF x.key==k THEN return x;
y=search(x.left,k);
IF y != nil THEN return y;
return search(x.right,k);
```

starte mit **`search(T.root,k)`**

Laufzeit $= \Theta(n)$

Jeder Knoten wird maximal einmal besucht, im schlechtesten Fall aber auch jeder Knoten

Einfügen




```
insert(T,x)
  //x.parent==x.left==x.right==nil;

1  IF T.root != nil THEN
2       T.root.parent=x;
3       x.left=T.root;
4  T.root=x;
```

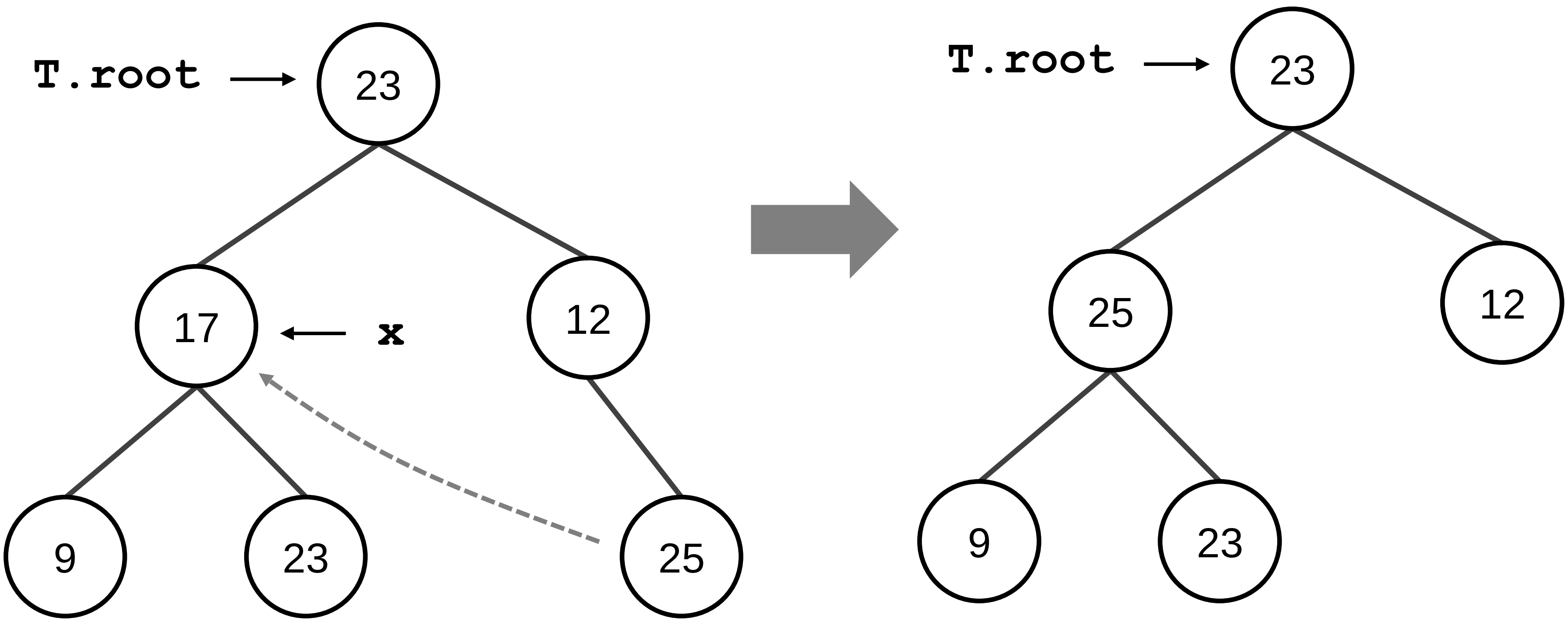
Laufzeit $= \Theta(1)$

Achtung: erzeugt linkslastigen Baum!!!

Löschen

Sonderfälle beachten:
(Halb-)Blatt ist selbst **x** oder Wurzel

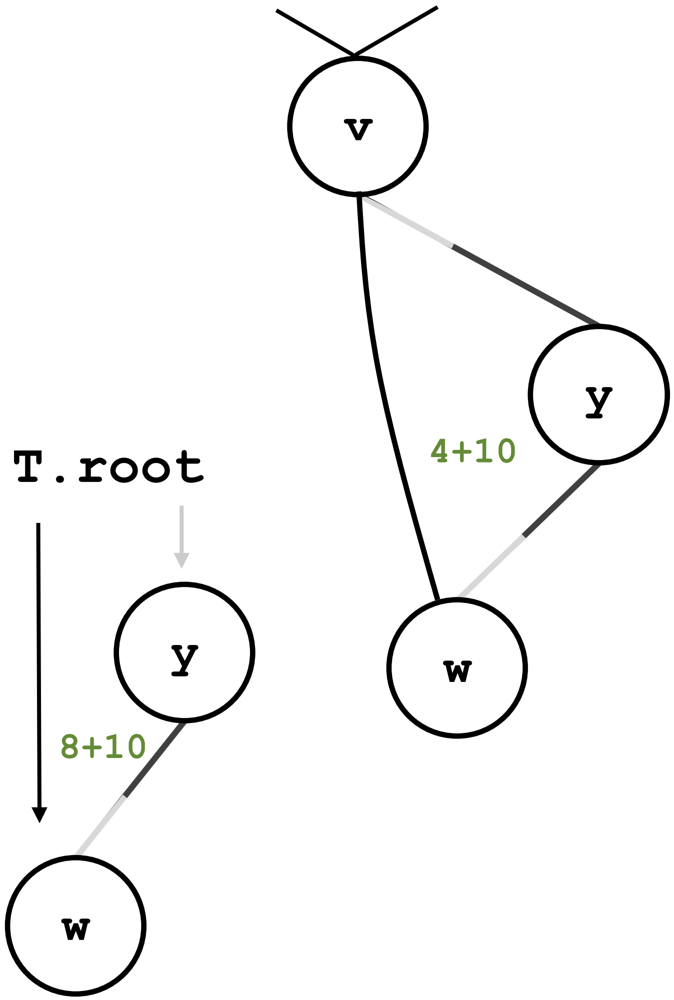
Idee:
Ersetze **x** durch (Halb-)Blatt ganz rechts



Es gibt natürlich auch andere Möglichkeiten

Löschen: Transplantation

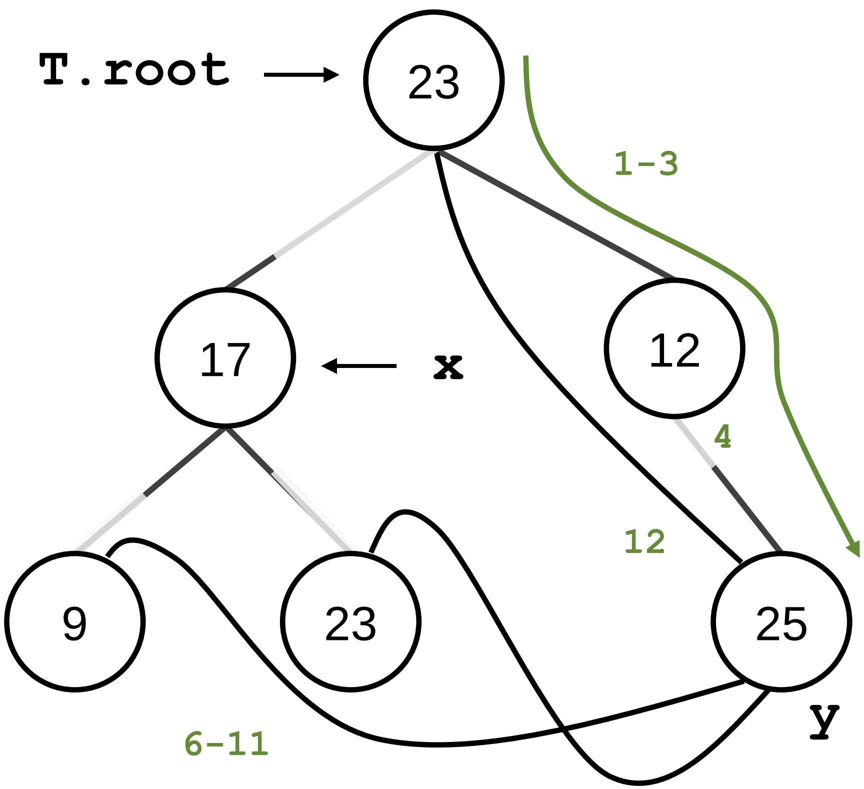
Laufzeit = $\Theta(1)$




```
transplant(T,y,w)
  //transplant w to y.parent

1  v=y.parent;

2  IF y != T.root THEN
3     IF y == v.right THEN
4        v.right=w;
5     ELSE
6        v.left=w;
7  ELSE
8     T.root=w;

9  IF w != nil THEN
10    w.parent=v;
```

(**w** muss dabei nicht an **y** hängen)

Löschen: Algorithmus




```
delete(T,x) //assumes x in T

1  y=T.root;
2  WHILE y.right!=nil DO
3     y=y.right;

4  transplant(T,y,y.left);

5  IF x != y THEN
6     y.left=x.left;
7     IF x.left != nil THEN
8        x.left.parent=y;
9     y.right=x.right;
10    IF x.right != nil THEN
11       x.right.parent=y;

12    transplant(T,x,y);
```

Laufzeit $= \Theta(h)$

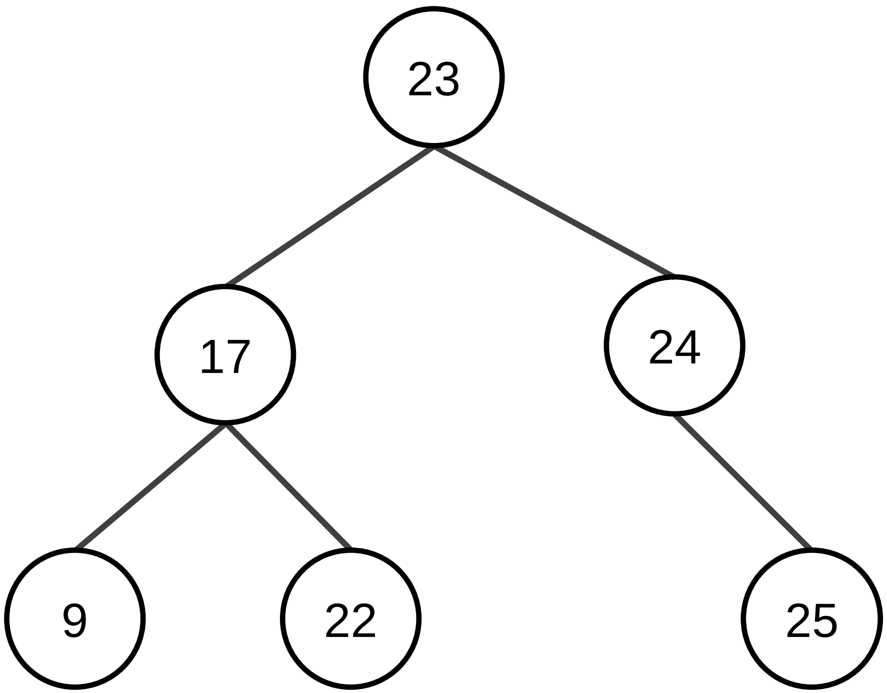
h Höhe des Baumes, $h = n$ möglich

Binäre Suchbäume

Operation	Laufzeit*
Einfügen	$\Theta(1)$
Löschen	$\Theta(h)$
Suchen	$\Theta(n)$

Geht das besser?

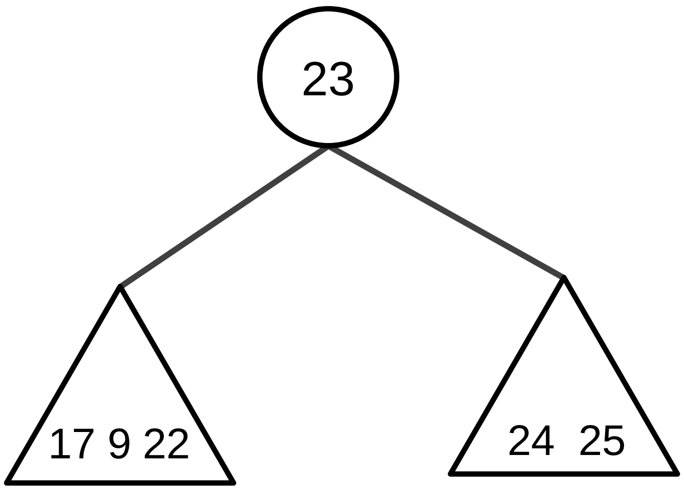
Binäre Suchbäume (Binary Search Tree, BST)



Wir nehmen wieder totale Ordnung auf den Werten an

Binärer Suchbaum:
Binärbaum, so dass für alle Knoten **`z`** gilt:
Wenn **`x`** Knoten im linken Teilbaum von **`z`**, dann **`x.key <= z.key`**
Wenn **`y`** Knoten im rechten Teilbaum von **`z`**, dann **`y.key >= z.key`**

Preorder + eindeutige Werte ⇒ Binärer Suchbaum



Pre = 17 9 22

Pre = 24 25

Bilde Teilbäume rekursiv

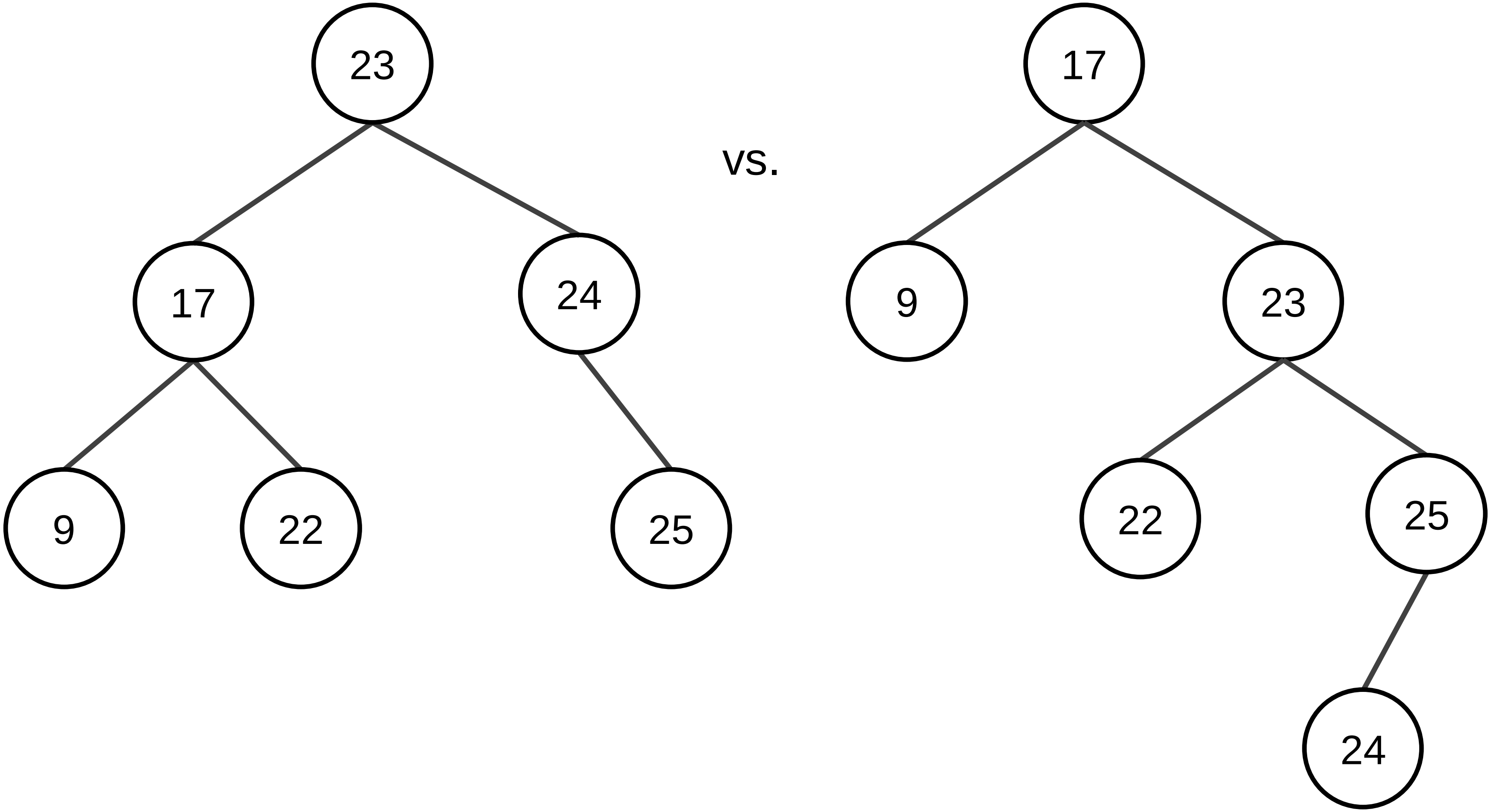
(1) Identifiziert Wurzel

Preorder = 23 17 9 22 24 25

(2) Identifiziert Werte im linken und rechten Teilbaum

Gilt analog für Postorder

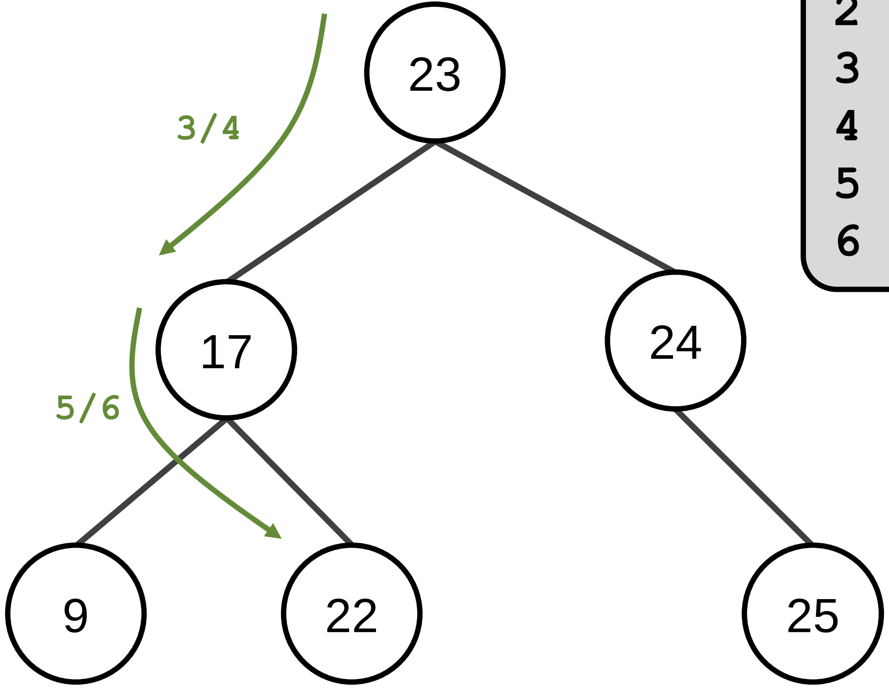
Inorder + eindeutige Werte $\not\Rightarrow$ Binärer Suchbaum



Beide Suchbäume haben gleiche Inorder

Suchen im Binären Suchbaum

`search(T.root,22)`




```
search(x,k) //1.Aufruf x=root

1  IF x==nil OR x.key==k THEN
2       return x;
3  IF x.key > k THEN
4       return search(x.left,k)
5  ELSE
6       return search(x.right,k);
```

Laufzeit = $O(h)$

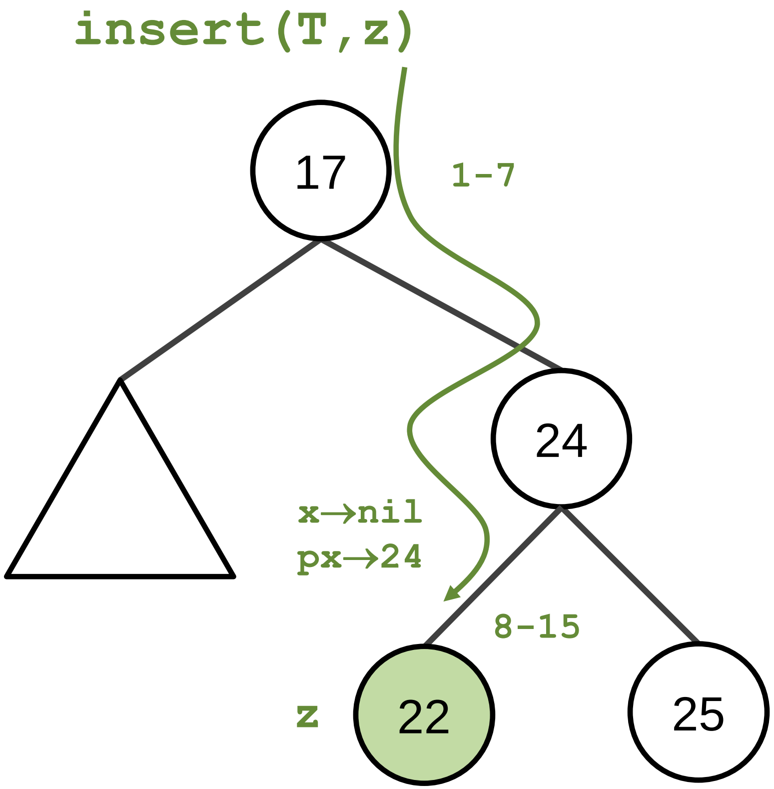
h Höhe des Baumes

Iterative Suche im Binären Suchbaum

```
search(x,k) //1.Aufruf x=root

IF x==nil OR x.key==k THEN
     return x;
IF x.key > k THEN
     return search(x.left,k)
ELSE
     return search(x.right,k);
```

```
iterative-search(x,k)  //Aufruf x=root

WHILE x != nil AND x.key != k DO
    IF x.key > k THEN
       x=x.left
    ELSE
       x=x.right;
return x;
```

Einfügen im BST

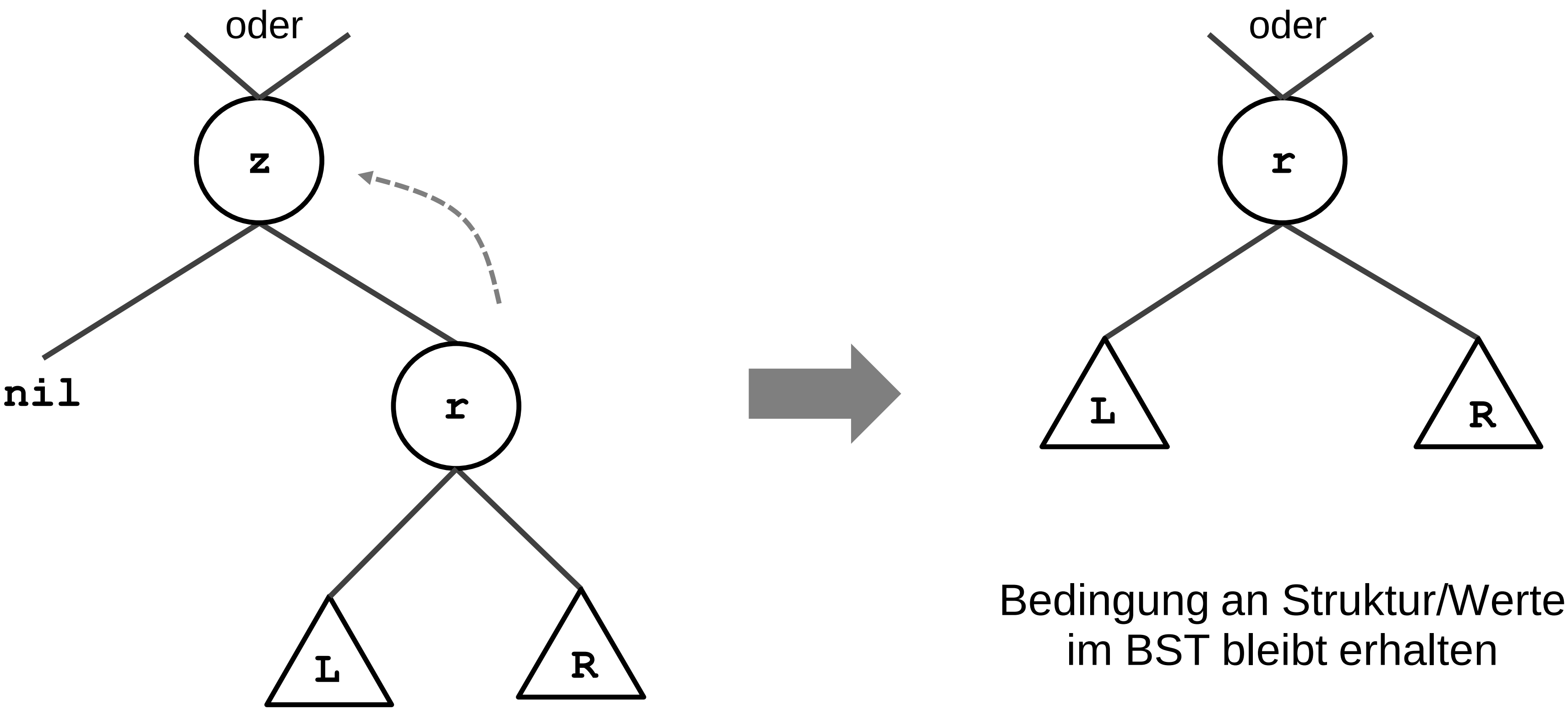



```
insert(T,z)
 //may insert z again
 //z.left==z.right==nil;

1  x=T.root; px=nil;

2  WHILE x != nil DO
3       px=x;
4       IF x.key > z.key THEN
5          x=x.left
6       ELSE
7          x=x.right;
8  z.parent=px;
9  IF px==nil THEN
10      T.root=z
11 ELSE
12      IF px.key > z.key THEN
13         px.left=z
14      ELSE
15         px.right=z;
```

Laufzeit = $O(h)$

Löschen im BST (I)

zu löschender Knoten **z** hat maximal ein Kind

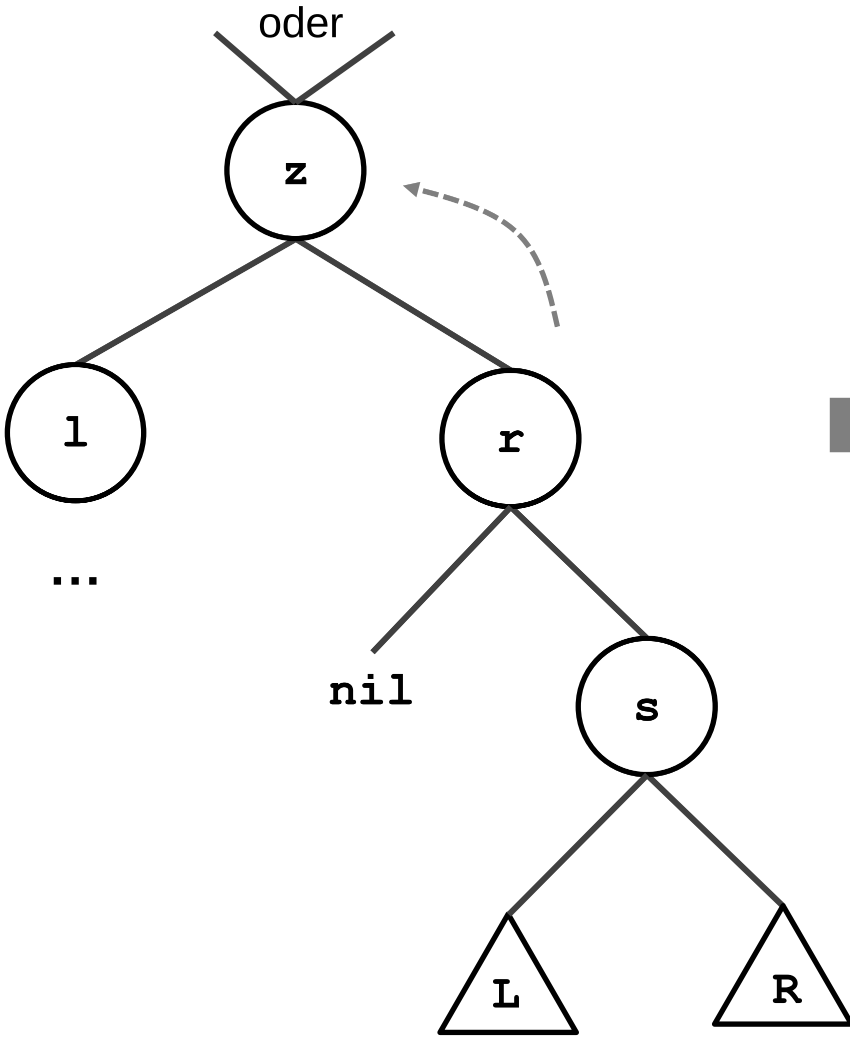


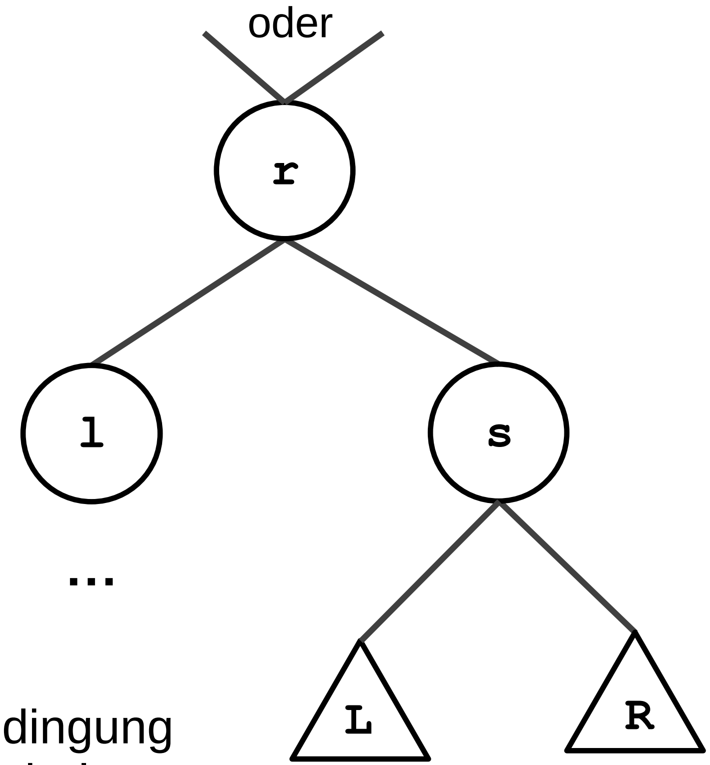
Bedingung an Struktur/Werte
im BST bleibt erhalten

analog, wenn auch/oder rechtes Kind = **nil**

Löschen im BST (II)

rechtes Kind von Knoten **z** hat kein linkes Kind



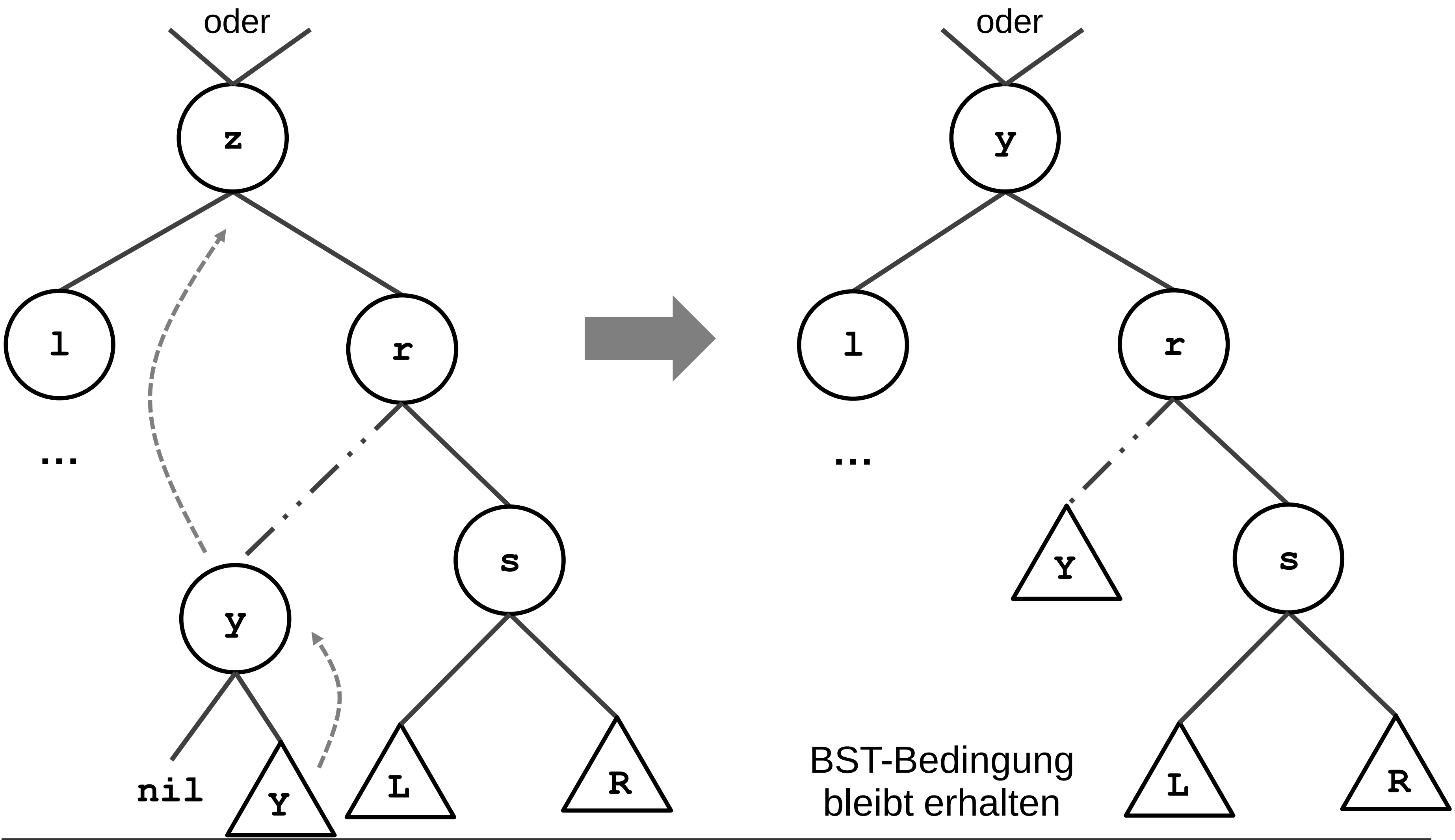


BST-Bedingung bleibt erhalten

(ginge auch, wenn linkes Kind von **z** kein rechtes Kind)

Löschen im BST (III)

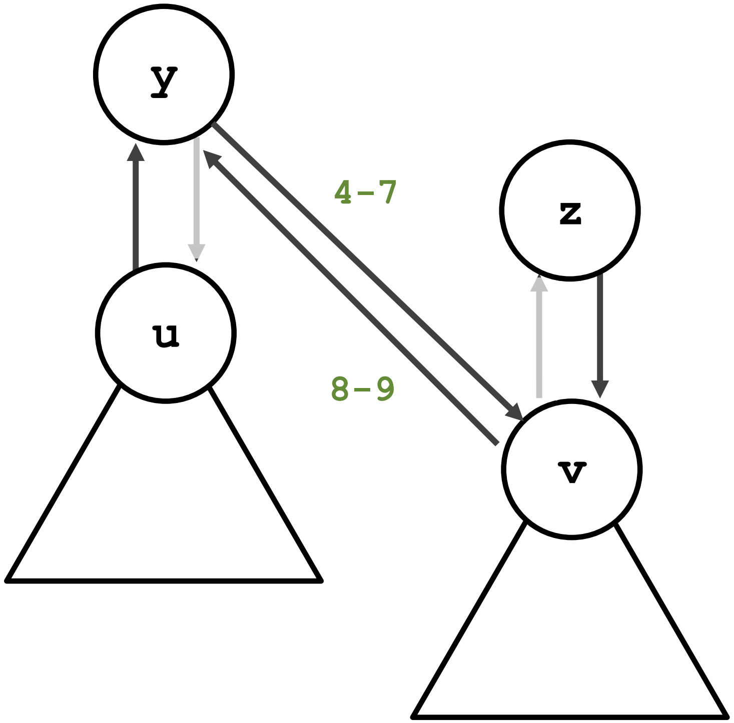
„kleinster“ Nachfahre vom rechten Kind von **z**



BST-Bedingung bleibt erhalten

Löschen: Transplantation

hängt Teilbaum **v** an Elternknoten von **u**

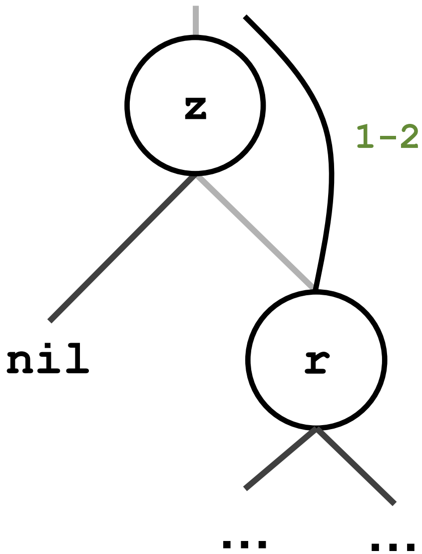



```
transplant(T,u,v)

1  IF u.parent==nil THEN
2     T.root=v
3  ELSE
4     IF u==u.parent.left THEN
5        u.parent.left=v
6     ELSE
7        u.parent.right=v;
8  IF v != nil THEN
9     v.parent=u.parent;
```

zur Erinnerung

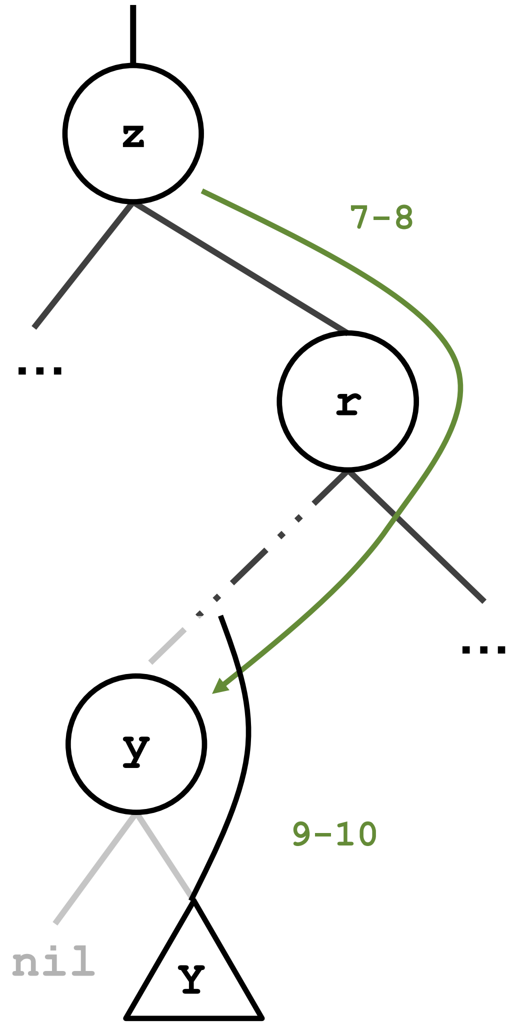
Laufzeit = $\Theta(1)$

Löschen: Algorithmus (I)



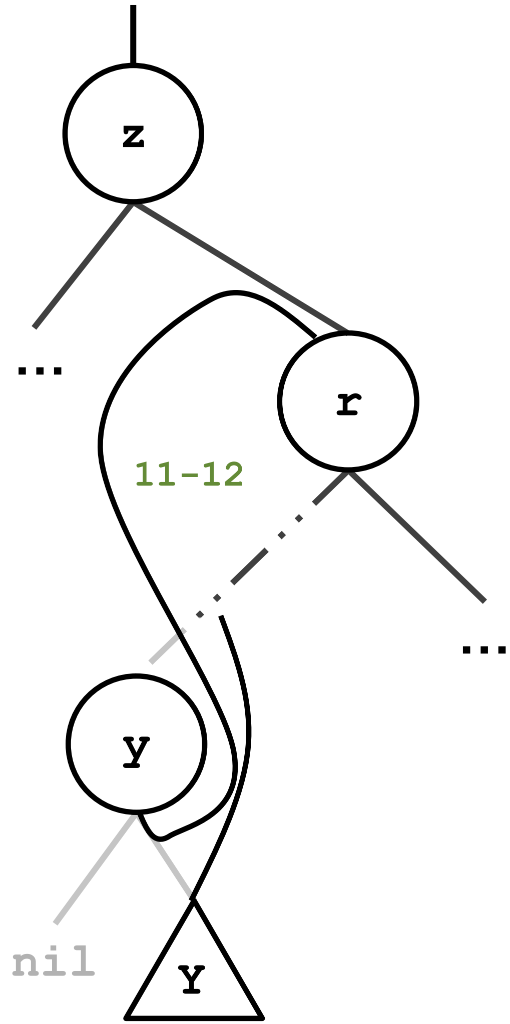

```
delete(T,z)

1  IF z.left==nil THEN
2     transplant(T,z,z.right)
3  ELSE
4     IF z.right==nil THEN
5        transplant(T,z,z.left)
6     ELSE
7        y=z.right;
8        WHILE y.left != nil DO y=y.left;

9        IF y.parent != z THEN
10          transplant(T,y,y.right);
11          y.right=z.right;
12          y.right.parent=y;
13       transplant(T,z,y);
14       y.left=z.left;
15       y.left.parent=y;
```

Löschen: Algorithmus (II)




```
delete(T,z)

1  IF z.left==nil THEN
2     transplant(T,z,z.right)
3  ELSE
4     IF z.right==nil THEN
5        transplant(T,z,z.left)
6     ELSE
7        y=z.right;
8        WHILE y.left != nil DO y=y.left;

9        IF y.parent != z THEN
10          transplant(T,y,y.right);
11          y.right=z.right;
12          y.right.parent=y;
13       transplant(T,z,y);
14       y.left=z.left;
15       y.left.parent=y;
```

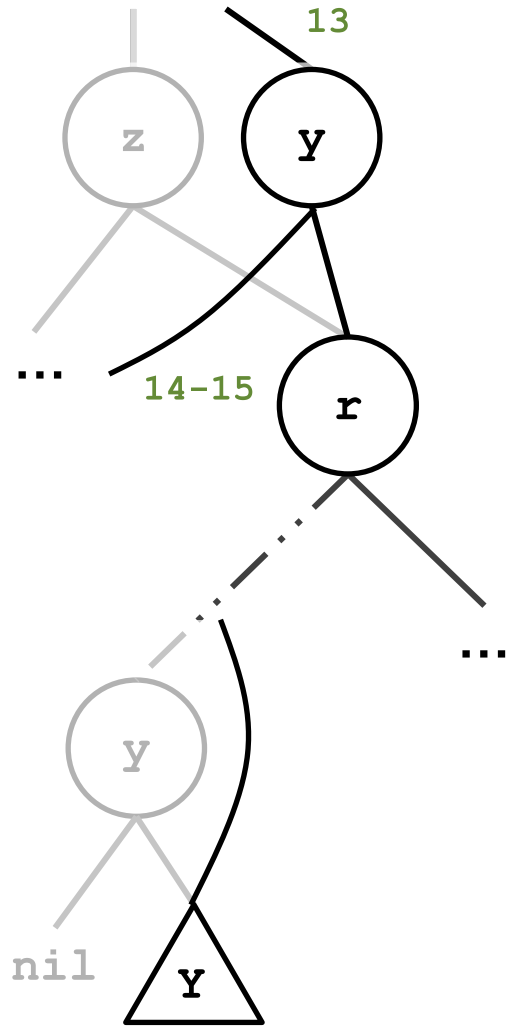
Löschen: Algorithmus (III)




```
delete(T,z)

1  IF z.left==nil THEN
2     transplant(T,z,z.right)
3  ELSE
4     IF z.right==nil THEN
5        transplant(T,z,z.left)
6     ELSE
7        y=z.right;
8        WHILE y.left != nil DO y=y.left;

9        IF y.parent != z THEN
10          transplant(T,y,y.right);
11          y.right=z.right;
12          y.right.parent=y;
13       transplant(T,z,y);
14       y.left=z.left;
15       y.left.parent=y;
```

Löschen: Algorithmus (IV)

Laufzeit = $O(h)$




```
delete(T,z)

1  IF z.left==nil THEN
2     transplant(T,z,z.right)
3  ELSE
4     IF z.right==nil THEN
5        transplant(T,z,z.left)
6     ELSE
7        y=z.right;
8        WHILE y.left != nil DO y=y.left;

9        IF y.parent != z THEN
10          transplant(T,y,y.right);
11          y.right=z.right;
12          y.right.parent=y;
13       transplant(T,z,y);
14       y.left=z.left;
15       y.left.parent=y;
```

Höhe Laufzeit

Binärer Suchbaum

Operation	Laufzeit*
Einfügen	O(h)
Löschen	O(h)
Suchen	O(h)

Verkettete Liste

Operation	Laufzeit*
Einfügen	$\Theta(1)$
Löschen	$\Theta(1)$
Suchen	$\Theta(n)$

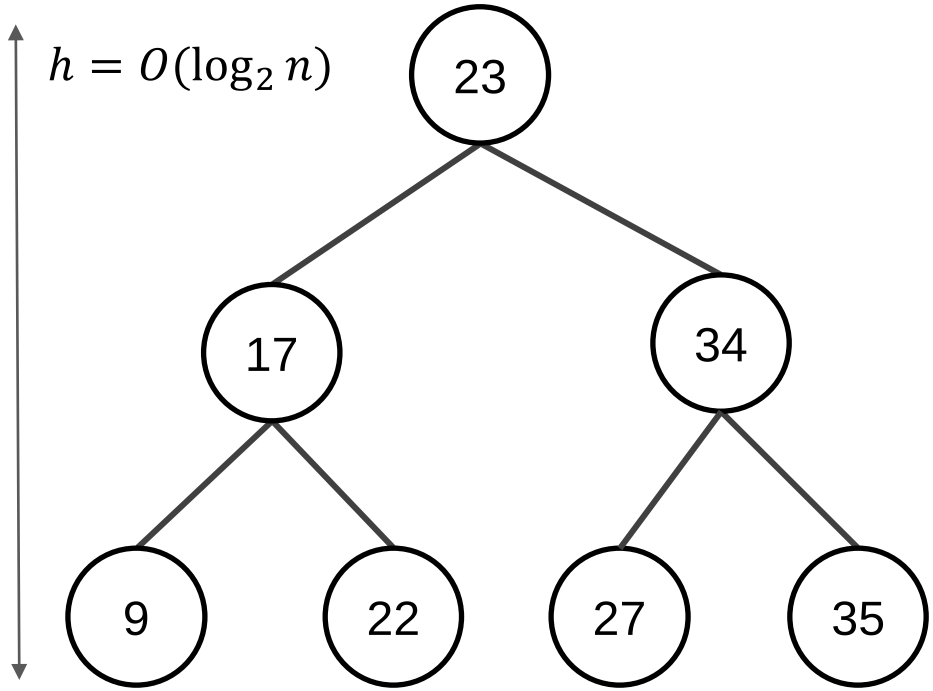
besser, wenn
viele Such-Operationen
und h klein relativ zu n

Höhe eines BST

Best-Case:

Laufzeit = $O(\log_2 n)$

$h = O(\log_2 n)$

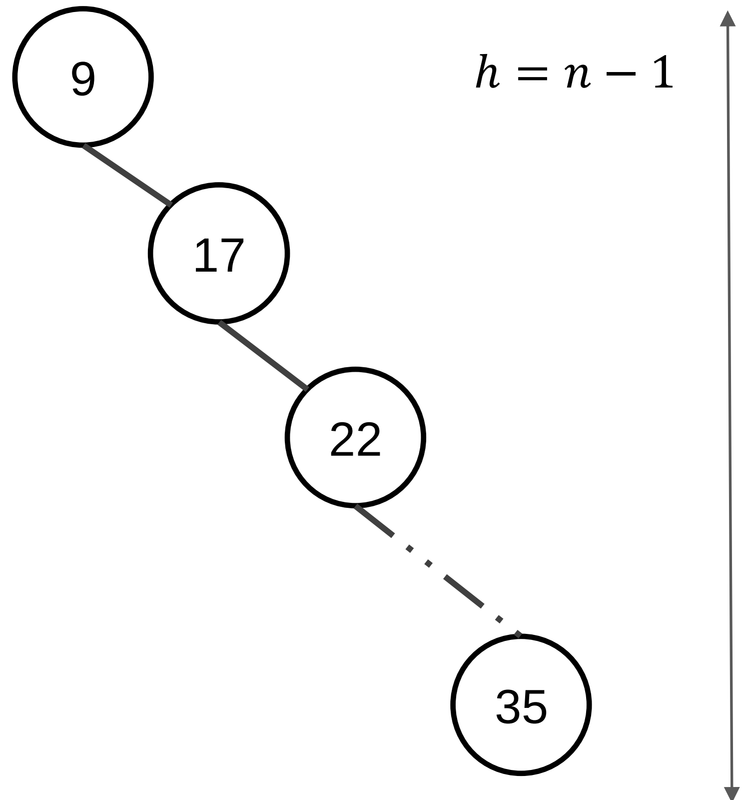


vollständig: alle Blätter haben gleiche Tiefe

Worst-Case:

Laufzeit = $\Omega(n)$

$h = n - 1$



degeneriert: lineare Liste

Durchschnittliche Höhe?

Analyse ohne Einfügen und Löschen

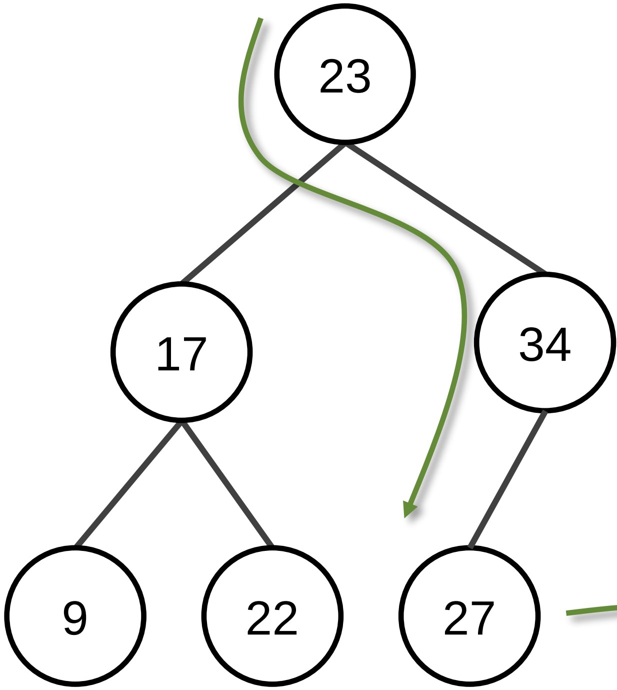
```
randomlyBuiltTree(D) //D data set

T=newTree();
WHILE D != ∅  DO
    Pick d uniformly from D;
    insert(T,newNode(d));
    remove d from D;
return T;
```

Die erwartete Höhe $E[h]$ des Baumes **`T`** erzeugt durch **`randomlyBuiltTree(D)`** für eine Datenmenge **`D`** mit n Werten ist $E[h] = \Theta(\log_2 n)$.

Suchbäume als Suchindex

```
SELECT *
FROM MyTable
WHERE ID=27;
```

```
27 | Victor | CS | …
```



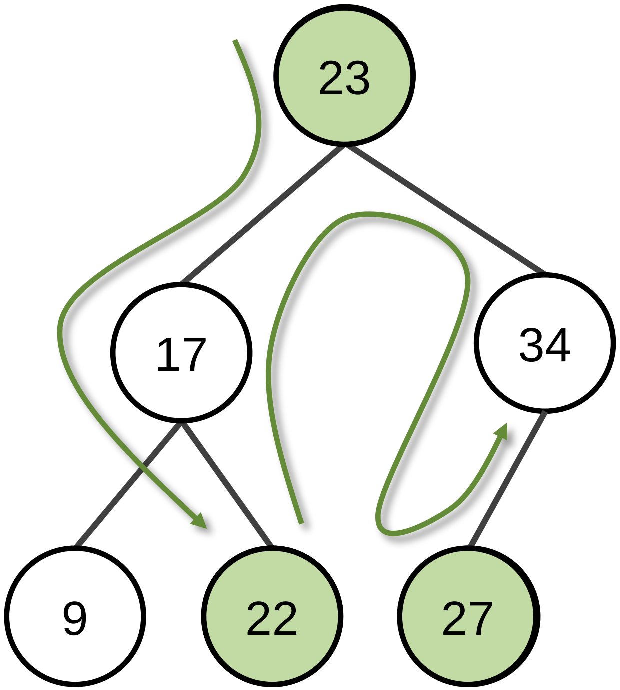
ID	Name	Department	…
23	Bob	CS	…
17	Alice	Math	…
9	Eve	CS	…
22	Carol	Physics	…
34	Peggy	Math	…
27	Victor	CS	…
…	…	…	…

Knoten speichert nur Primärschlüssel (hier **`ID`**) und Zeiger auf Daten

Bereichssuche

```
SELECT *
FROM MyTable
WHERE ID BETWEEN 20 AND 30;
```

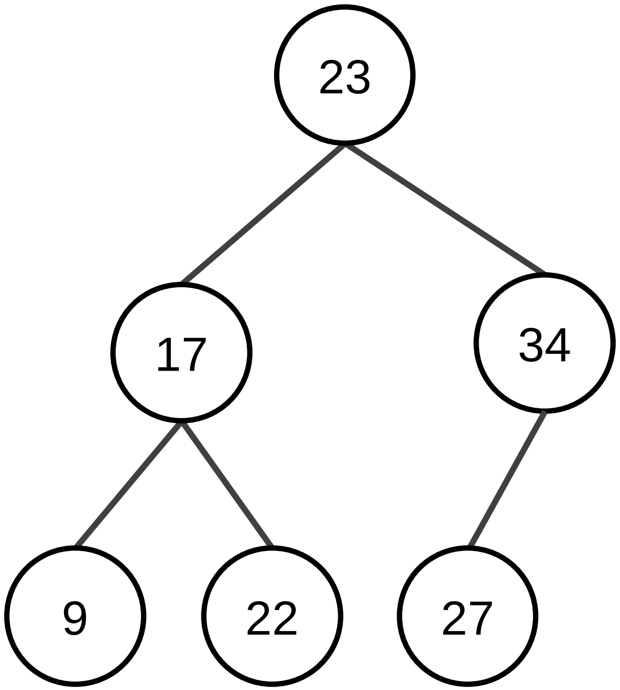
```
22 | Carol  | Physics | …
23 | Bob    | CS      | …
27 | Victor | CS      | …
```

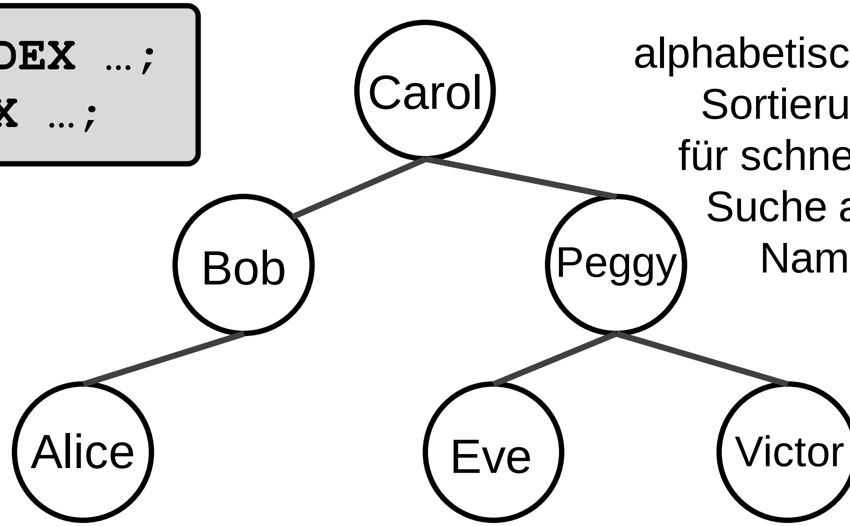


ID	Name	Department	…
23	Bob	CS	…
17	Alice	Math	…
9	Eve	CS	…
22	Carol	Physics	…
34	Peggy	Math	…
27	Victor	CS	…
…	…	…	…

Sekundärindizes

```
CREATE INDEX …;
DROP INDEX …;
```





alphabetische Sortierung für schnelle Suche auf Namen

ID	Name	Department	…
23	Bob	CS	…
17	Alice	Math	…
9	Eve	CS	…
22	Carol	Physics	…
34	Peggy	Math	…
27	Victor	CS	…
…	…	…	…

Zusätzliche Indizes kosten Speicherplatz, daher nur sinnvoll, wenn oft gesucht wird

Geben Sie Algorithmen für das Maximum und das Minimum im binären Suchbaum an. Welche Laufzeiten haben sie?

Beschreiben Sie eine Modifikation der Einfüge-Operation, die keine doppelten Einträge erzeugt.

Geben Sie einen Algorithmus an, der für Eingabe k alle Werte $\leq k$ eines binären Suchbaumes ausgibt.