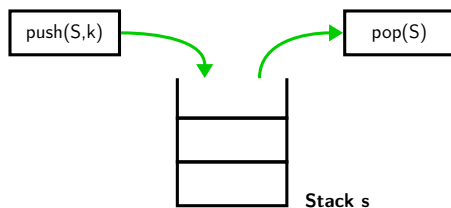


1 Grundlegende Datenstrukturen

1.1 Stacks

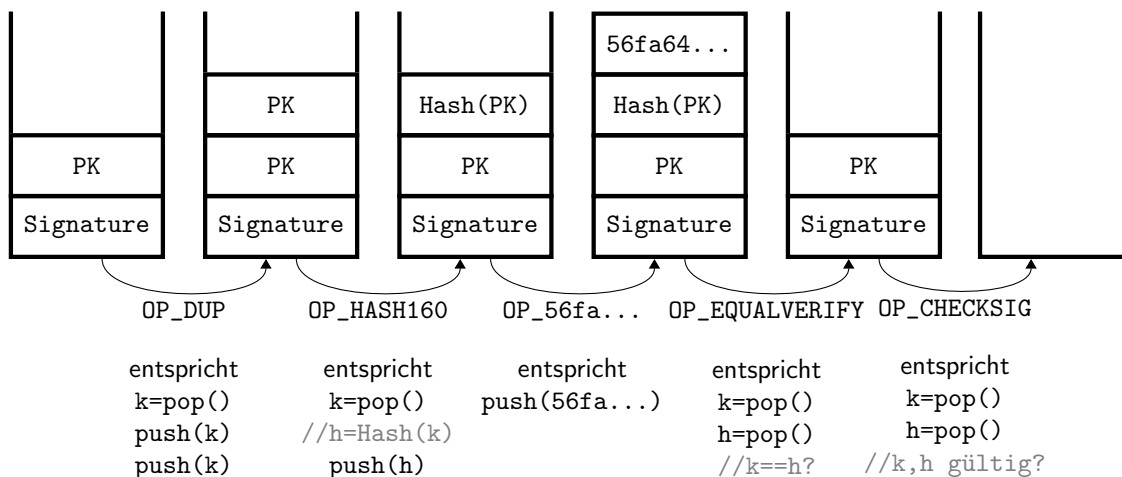
- **Abstrakter Datentyp Stack**

- new S()
 - * Erzeugt neuen (leeren) Stack
- s.isEmpty()
 - * Gibt an, ob Stack s leer ist
- s.pop()
 - * Gibt oberstes Element vom Stack s zurück und löscht es vom Stack
 - * Gibt Fehlermeldung aus, falls der Stack leer ist
- s.push(k)
 - * Schreibt k als neues oberstes Element auf Stack s
- Abstrakter Aufbau:
 - * **LIFO**-Prinzip - Last in, First out



- **Beispiel Bitcoin**

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



- Stacks als Array

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

- s.top zeigt immer auf oberstes Element
- pop() führt dazu, dass s.Top sich eins nach links bewegt
- push(k) führt dazu, dass s.Top sich eins nach rechts bewegt

- Stacks als Array - Methoden, falls maximale Größe bekannt

new(S) 1 S.A[] = ALLOCATE(MAX); 2 S.TOP = -1;	isEmpty(S) 1 IF S.top < 0 THEN 2 return true; 3 ELSE 4 return false;
pop(S) 1 IF isEmpty(S) THEN 2 error "underflow"; 3 ELSE 4 s.top = s.top - 1; 5 return S.A[S.top + 1];	push(S) 1 IF S.top == MAX - 1 THEN 2 error "overflow"; 3 ELSE 4 S.top = S.top + 1; 5 S.A[S.top] = k;

- Stacks mit variabler Größe - Einfach

- Falls push(k) bei vollem Array \Rightarrow Vergrößerung des Arrays
- Erzeugen eines neuen Arrays mit Länge + 1 und Umkopieren aller Elemente
- Durchschnittlich $\Omega(n)$ Kopierschritte pro push-Befehl

- **Stacks mit variabler Größe - Verbesserung**

- Idee:

- * Wenn Grenze erreicht, Verdopplung des Speichers und Kopieren der Elemente
 - * Falls weniger als ein Viertel belegt, schrumpfe das Array wieder

- Methoden:

RESIZE(A, m) reserviert neuen Speicher der Größe m und kopiert A um

new(S)

```
1 S.A[ ]=ALLOCATE(1);
2 S.TOP=-1;
3 S.memsize=1;
```

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.A,S.memsize);
8   return S.A[S.top+1];
```

isEmpty(S)

```
1 IF S.top<0 THEN
2   return true;
3 ELSE
4   return false;
```

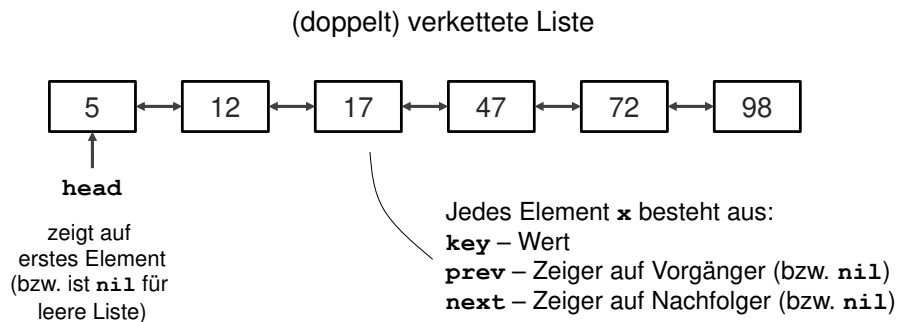
push(S)

```
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.A,S.memsize);
```

- Im Durchschnitt für jeder der mindestens n Befehle $\Theta(1)$ Umkopierschritte

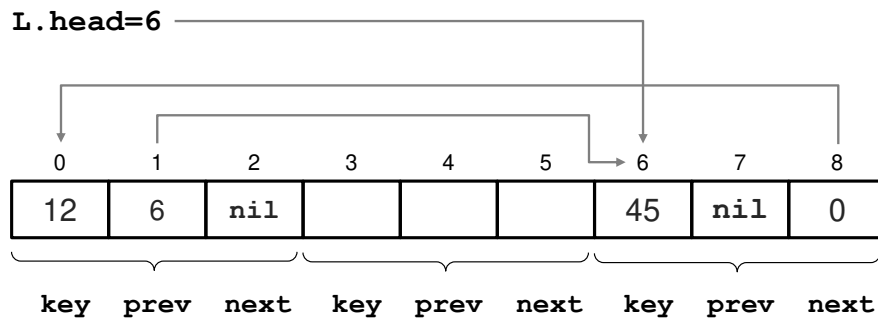
1.2 Verkettete Listen

- Aufbau



- Verkettete Listen durch Arrays

Entspricht doppelter Verkettung zwischen 45 und 12



- Elementare Operationen auf Listen

– Suche nach Element

* Laufzeit beträgt im Worst Case $\Theta(n)$

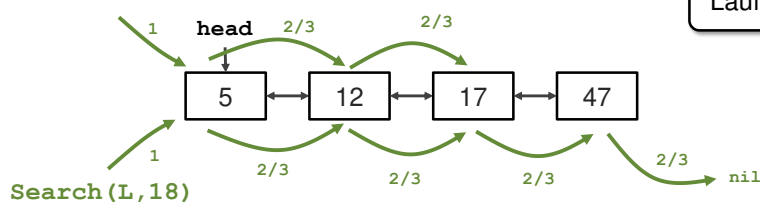
⇒ Keine Überprüfung, ob Wert bereits in Liste, sonst $\Theta(n)$

* Code:

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

```
1 current = L.head;  
2 WHILE current != nil AND current.key != k DO  
3   current = current.next;  
4 return current;
```

`Search(L,17)`



– Einfügen eines Elements am Kopf der Liste

* Laufzeit beträgt $\Theta(1)$, da Einfügen am Kopf

* Code:

```
insert(L,x)
1 insert(L,x)
2 x.next = l.head;
3 x.prev = nil;
4 IF L.head != nil THEN
5   L.head.prev = x;
6 L.head = x;
```

– Löschen eines Elements aus Liste

* Laufzeit beträgt $\Theta(1)$, da hier Pointer auf Objekt gegeben

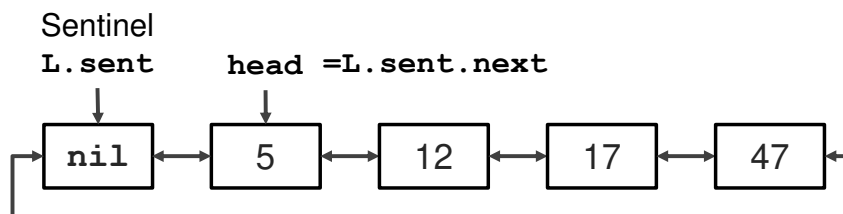
Löschen eines Wertes k mithilfe von Suche beträgt $\Omega(n)$

* Code:

```
delete (L,x)
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;
```

• Vereinfachung per Wächter/Sentinels

– Ziel ist die Eliminierung der Spezialfälle für Listenanfang/-ende



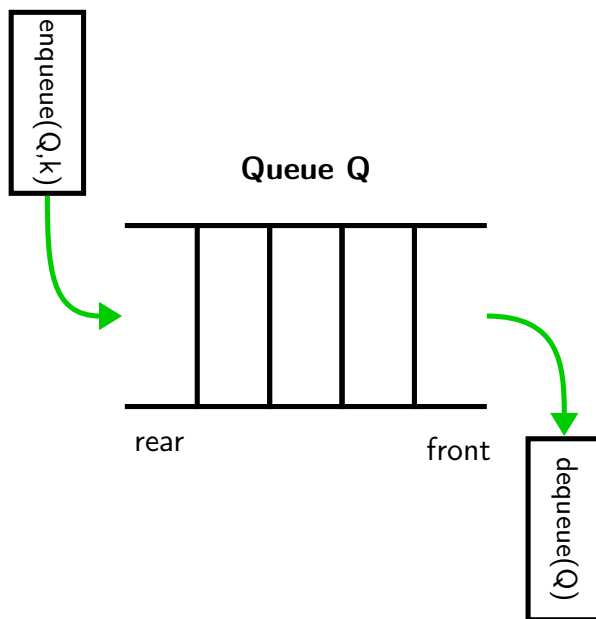
– Löschen mit Sentinels:

```
deleteSent(L,x)
1 x.prev.next = x.next;
2 x.next.prev = x.prev;
```

1.3 Queues

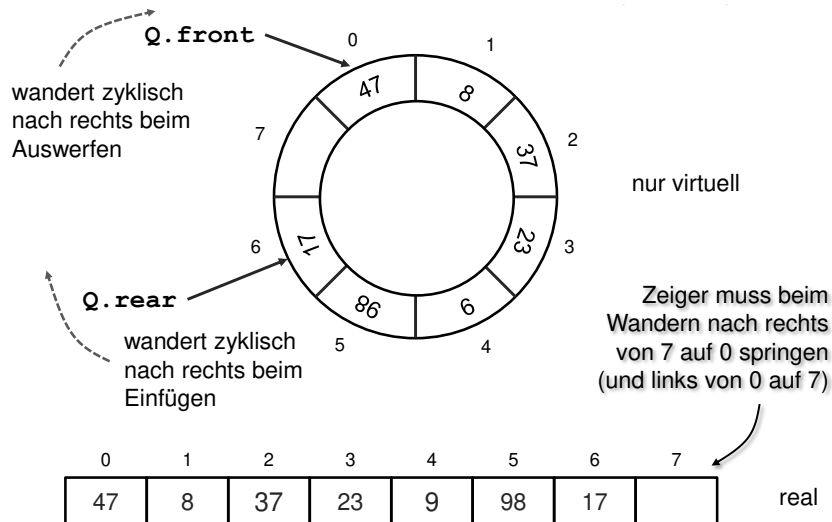
- Abstrakter Datentyp Queue

- `new Q()`
 - * Erzeuge neue (leere) Queue
- `q.isEmpty()`
 - * Gibt an, ob Queue q leer ist
- `q.dequeue()`
 - * Gibt vorderstes Element aus q zurück und löscht es auf Queue
 - * Fehlermeldung, falls Queue leer ist
- `q.enqueue(k)`
 - * Schreibt k als neues hinterstes Element auf q
 - * Fehlermeldung, falls Queue voll ist
- Abstrakter Aufbau:
 - * **FIFO**-Prinzip / First in, First out



• Queues als (virtuelles) zyklisches Array

Bekannt: Maximale Elemente gleichzeitig in Queue



- Problem, falls `Q.rear` und `Q.front` auf selbes Element zeigen
 - * Speichere Information, ob Schlange leer oder voll, in boolean `empty`
 - * Alternativ: Reserviere ein Element des Arrays als Abstandshalter
- Methoden für zyklisches Array

`new(Q)`

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

`isEmpty(Q)`

```
1 return Q.empty;
```

`dequeue(Q)`

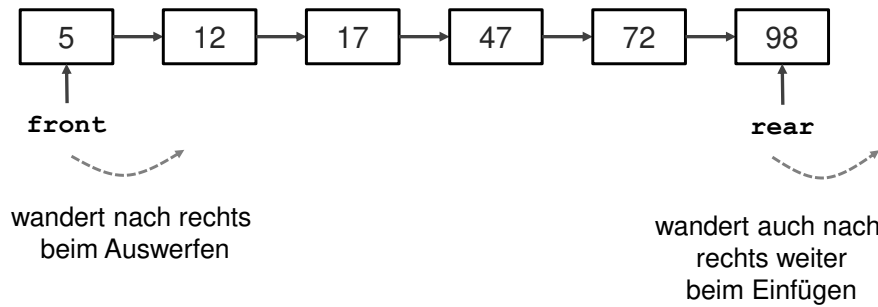
```
1 IF isEmpty(Q) THEN
2     error "underflow";
3 ELSE
4     Q.front = Q.front + 1 mod MAX;
5     IF Q.front == Q.rear THEN
6         Q.empty = true;
7     return Q.A[Q.front - 1 mod MAX];
```

`enqueue(Q,k)`

```
1 IF Q.rear == Q.front AND !Q.isEmpty
2 THEN error "overflow";
3 ELSE
4     Q.A[Q.rear] = k;
5     Q.rear = Q.rear + 1 mod MAX;
6     Q.empty = false;
```

- Queues durch einfach verkettete Listen

(einfach) verkettete Liste



Methoden:

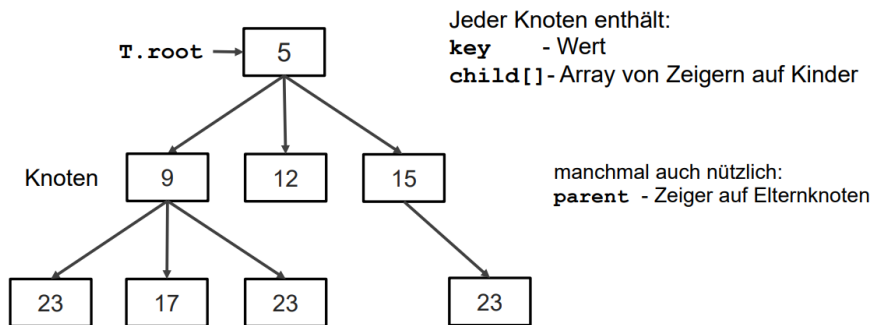
<pre> new(Q) 1 Q.front=nil; 2 Q.rear=nil; </pre>	<pre> isEmpty(Q) 1 IF Q.front==nil THEN 2 return true; 3 ELSE 4 return false; </pre>
<pre> dequeue(Q) 1 IF isEmpty(Q) THEN 2 error "underflow"; 3 ELSE 4 x=Q.front; 5 Q.front=Q.front.next; 6 return x; </pre>	<pre> enqueue(Q,x) 1 IF isEmpty(Q) THEN 2 Q.front=x; 3 ELSE 4 Q.rear.next=x; 5 x.next=nil; 6 Q.rear=x; </pre>

- Laufzeit

- Enqueue: $\Theta(1)$
- Dequeue: $\Theta(1)$

1.4 Binäre Bäume

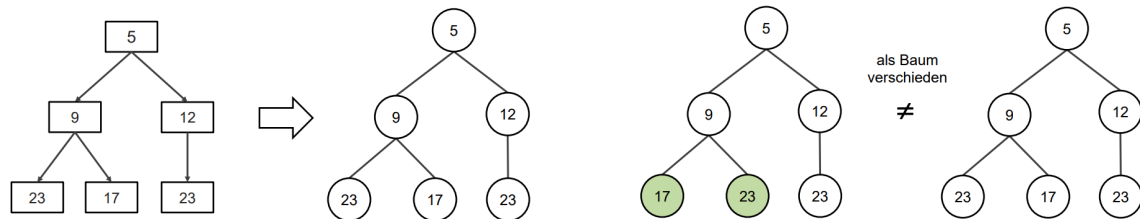
- Bäume durch verkettete Listen



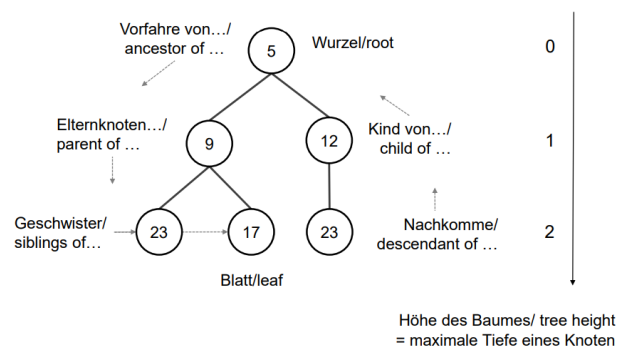
Baum-Bedingung: Baum ist leer oder...
es gibt einen Knoten r („Wurzel“), so dass jeder Knoten v von der Wurzel aus
per eindeutiger Sequenz von **child**-Zeigern erreichbar ist:
 $v = r.child[i_1].child[i_2] \dots child[i_m]$

Bäume sind "azyklisch" (Keine "rückführende Spur")

- Darstellung als (ungerichteter) Graph

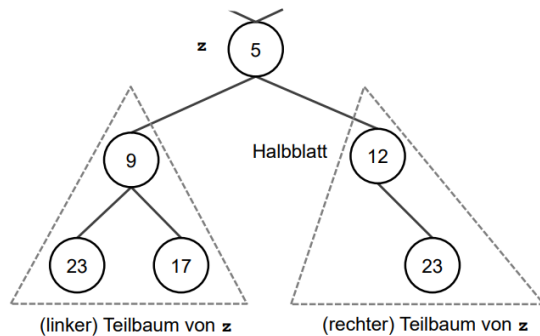


- Allgemeine Begrifflichkeiten



- * Blatt: Knoten ohne Nachfolger
- * Nachkomme von x:
Erreichbar durch Pfad ausgehend von x

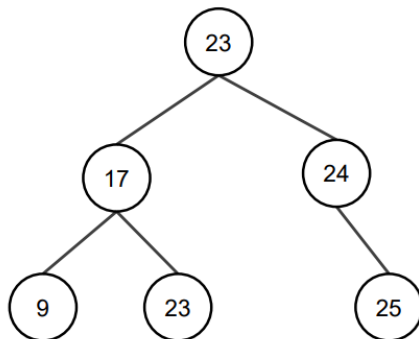
• Begrifflichkeiten Binärbaum



- * Jeder Knoten hat maximal zwei Kinder
left=child[0] und right=child[1]
- * Ausgangsgrad jedes Knoten ist ≤ 2
- * Höhe leerer Baum per Konvention -1
- * Höhe (nicht-leerer) Baum:
 $\max\{\text{Höhe aller Teilbäume der Wurzel}\} + 1$
- * Halbblatt: Knoten mit nur einem Kind

• Traversieren von Bäumen

- Darstellung eines Baumes mithilfe einer Liste der Werte aller Knoten
- Laufzeit bei n Knoten: $T(n) = O(n)$
- Nutzung der Preorder für das Kopieren von Bäumen
 - (a) Preorder betrachtet Knoten und legt Kopie an
 - (b) Preorder geht dann in Teilbäume und kopiert diese
- Nutzung der Postorder für das Löschen von Bäumen
 - (a) Postorder geht zuerst in Teilbäume und löscht diese
 - (b) Betrachten des Knoten erst danach und dann Löschung dieses



inorder(T.root) ergibt

9 17 23 23 24 25

preorder(T.root) ergibt

23 17 9 23 24 25

postorder(T.root) ergibt

9 23 17 25 24 23

Code:

inorder(x)

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

preorder(x)

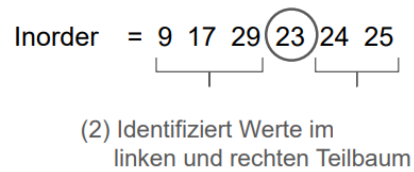
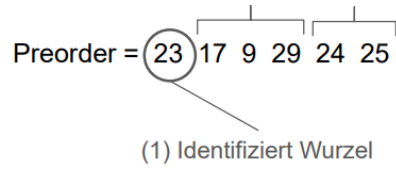
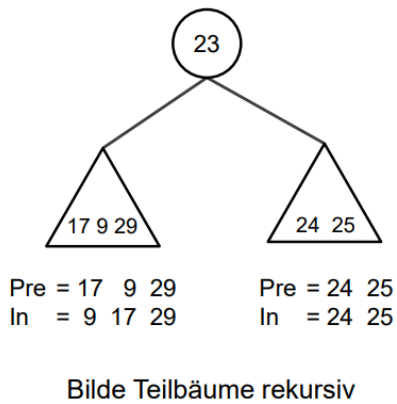
```
1 IF x != nil THEN
2   print x.key;
3   preorder(x.left);
4   preorder(x.right);
```

postorder(x)

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

• Eindeutige Bestimmbarkeit von Bäumen

- Nur In-,Pre-,Postorder reichen nicht zur eindeutigen Bestimmbarkeit von Bäumen
 \Rightarrow Preorder/Postorder + Inorder + eindeutige Werte sind notwendig



• Abstrakter Datentyp Baum

– Abstrakter Aufbau:

- * new T()
 - Erzeugt neuen Baum namens t
- * t.search(k)
 - Gibt Element x in Baum t mit $x.key == k$ zurück
- * t.insert(k)
 - Fügt Element x in Baum t hinzu
- * t.delete(x)
 - Löscht x aus Baum t

– Suche nach Elementen:

- * Laufzeit = $\Theta(n)$ (Jeder Knoten maximal einmal, jeder Knoten im schlechtesten Fall)
- * Starte mit search(T.root, k)
- * Code:

```
search(x,k)
1 IF x == nil THEN return nil;
2 IF x.key == k THEN return x;
3 y = search(x.left,k);
4 IF y != nil THEN return y;
5 return search(x.right,k);
```

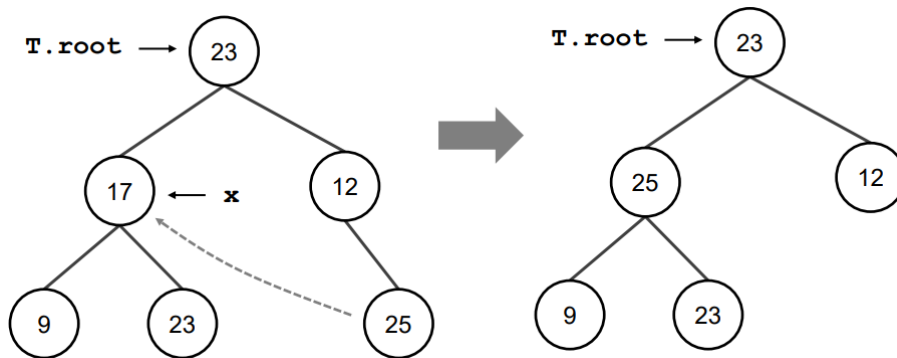
– Einfügen von Elementen:

- * Laufzeit = $\Theta(1)$
- * Hier wird als Wurzel eingefügt (Achtung: Erzeugt linkslastigen Baum)
- * Code:

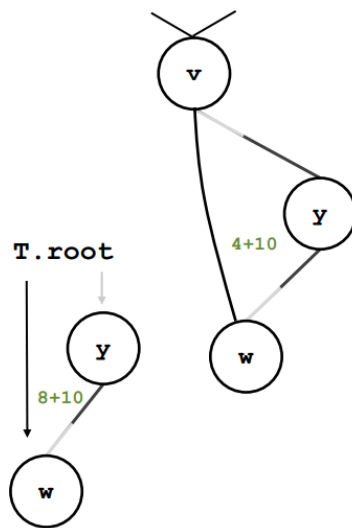
```
insert(T,x) // x.parent == x.left == x.right == nil;  
1 IF T.root != nil THEN  
2   T.root.parent = x;  
3   x.left = T.root;  
4 T.root = x;
```

– Löschen von Elementen:

- * Laufzeit = $\Theta(h)$ (Höhe des Baumes, $h = n$ möglich)
- * Hier: Ersetze x durch Halbblatt ganz rechts



* Connect-Algorithmus:



· Laufzeit = $\Theta(1)$

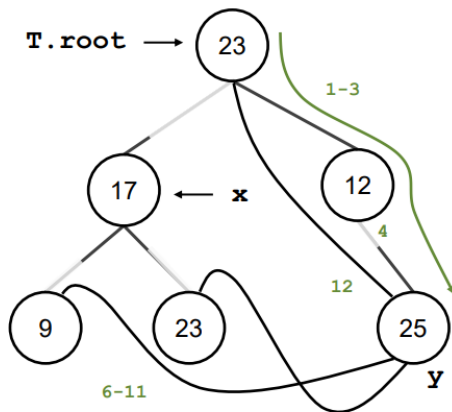
connect(T,y,w) // Connects w to y.parent

```

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;

```

* Delete-Algorithmus:



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

```

1 y = T.root;
2 WHILE y.right != nil DO
3     y = y.right;
4 connect(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    connect(T,x,y);

```

1.5 Binäre Suchbäume

- **Definition**

- Totale Ordnung auf den Werten

- Für alle Knoten z gilt:

- Wenn x Knoten im linken Teilbaum von z , dann $x.key \leq z.key$

- Wenn y Knoten im rechten Teilbaum von z , dann $y.key \geq z.key$

- Preorder/Postorder + eindeutige Werte \Rightarrow Eindeutige Identifizierung

- **Suchen im Binären Suchbaum**

- * Laufzeit = $O(h)$ (Höhe)

- * Code:

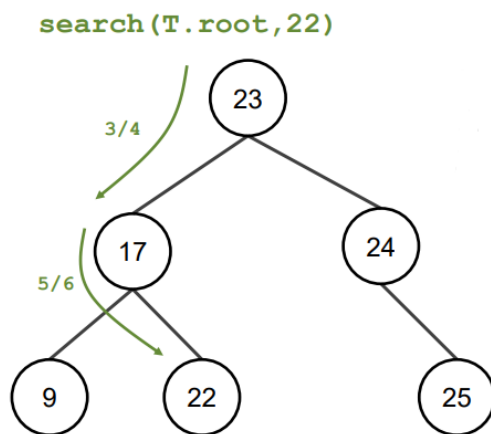
```
search(x,k) // 1. Aufruf x = root
```

```
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);
```

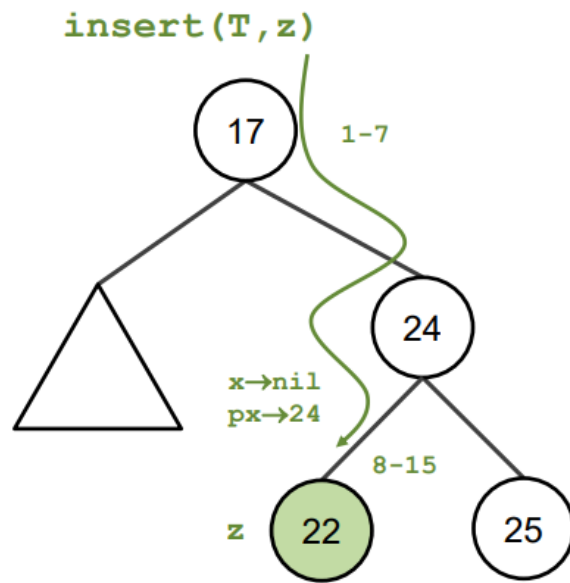
- * Iterativer Code:

```
iterative-search(x,k)
```

```
1 WHILE x != nil AND x.key != k DO
2   IF x.key > k THEN
3     x = x.left;
4   ELSE
5     x = x.right;
6 return x;
```



- Einfügen im Binary Search Tree



- * Laufzeit = $O(h)$
- * Aufwendiger, da Ordnung erhalten werden muss
- * Code:

```

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

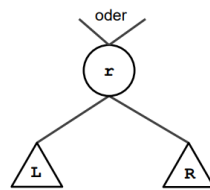
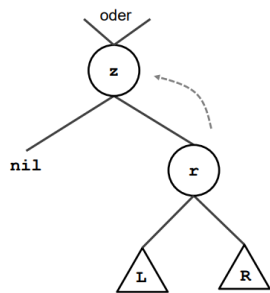
1  x = T.root;
2  px = nil;
3  WHILE x != nil DO
4      px = x;
5      IF x.key > z.key THEN
6          x = x.left;
7      ELSE
8          x = x.right;
9  z.parent = px;
10 IF px == nil THEN
11     T.root = z;
12 ELSE
13     IF px.key > z.key THEN
14         px.left = z;
15     ELSE
16         px.right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

• Löschen im BST

– Verschiedene Fälle:

Löschen im BST (I)

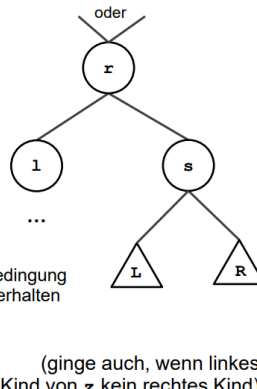
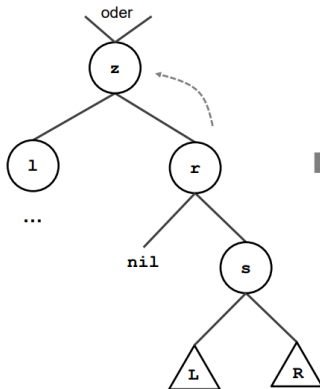
zu löschender Knoten z hat maximal ein Kind



Bedingung an Struktur/Werte
im BST bleibt erhalten

Löschen im BST (II)

rechtes Kind von Knoten z hat kein linkes Kind

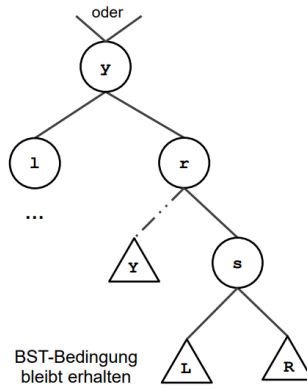
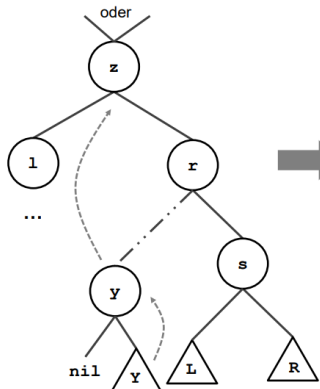


BST-Bedingung
bleibt erhalten

(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

– Code

```
transplant(T,u,v) // Hängt Teilbaum v an Parent von u
```

```
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;
```

```
delete(T,z)
```

```
1 IF z.left == nil THEN
2   transplant(T,z,z.left)
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;
```

– Laufzeit = $O(h)$

– Laufzeit ist damit besser, wenn viele Suchoperationen und h klein relativ zu n

- **Höhe eines BST**

- *Best Case:*

- * Vollständiger Baum (Alle Blätter gleiche Tiefe)
 - * $h = O(\log_2 n)$
 - * Laufzeit = $O(\log_2 n)$

- *Worst Case:*

- * Degenerierter Baum (lineare Liste)
 - * $h = n - 1$
 - * Laufzeit = $\Theta(n)$

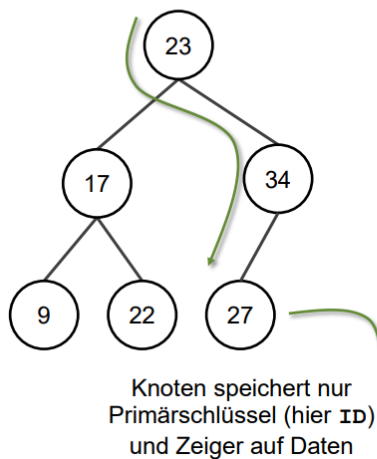
- *Durchschnittliche Höhe:*

- * Erwartete Höhe: $\Theta(\log_2 n)$

- **Suchbäume als Suchindex**

- Knoten speichert nur Primärschlüssel und Zeiger auf Daten
 - Zusätzliche Indizes möglich, kosten aber Speicherplatzbedarf

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	...
...