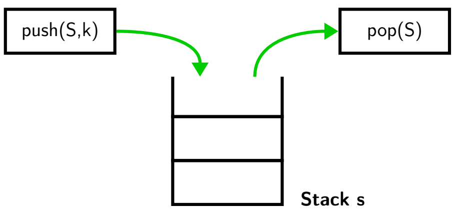
1 Grundlegende Datenstrukturen

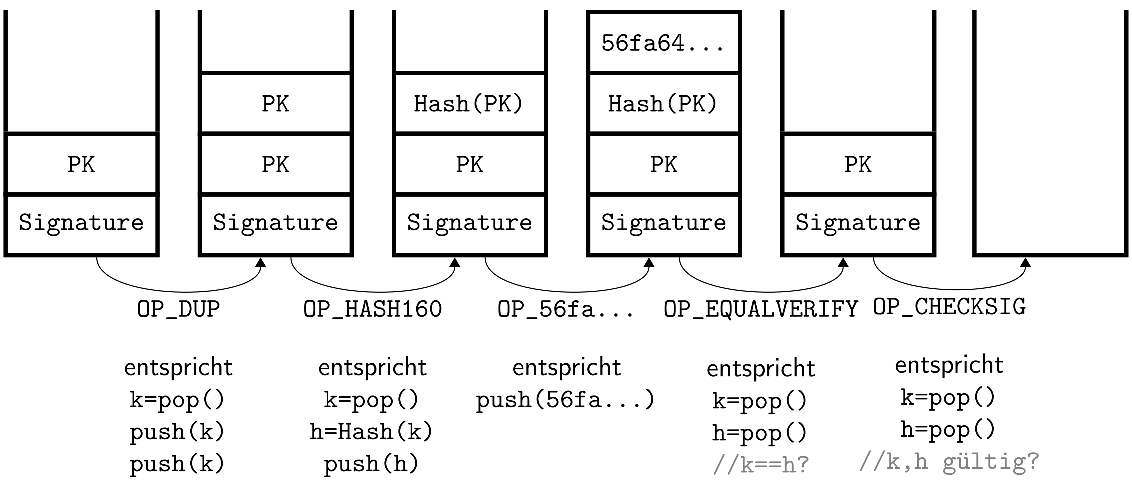
1.1 Stacks

- **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)

```
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)

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

- **Stacks mit variabler Größe - Einfach**
  - Falls `push(k)` bei vollem Array $\Rightarrow$ Vergößerung 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)

```
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.mensize=s.memsize/2;
        RESIZE(S.A,S.memsize);
    return S.A[S.top+1];
```

push(S)

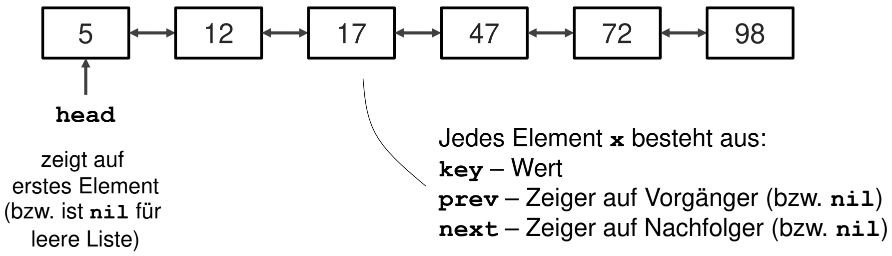
```
S.top=S.top+1;
S.A[S.top]=k;
IF S.top+1>=S.memsize THEN
    S.memsize=2*S.memsize;
    RESIZE(S.A,S.memsize);
```

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

1.2 Verkettete Listen

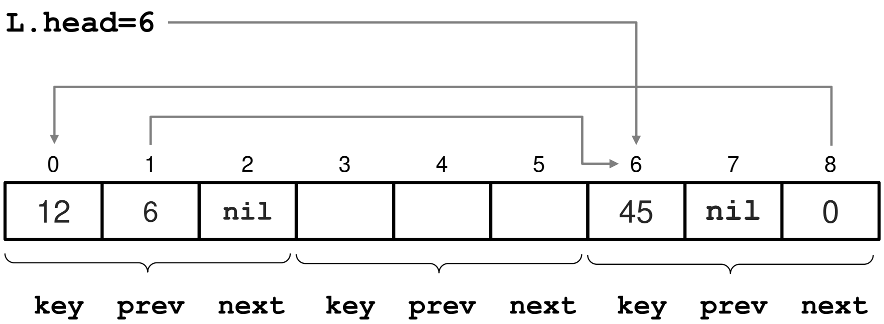
- **Aufbau**

(doppelt) verkettete Liste



- **Verkettete Listen durch Arrays**

  Entspricht doppelter Verkettung zwischen 45 und 12

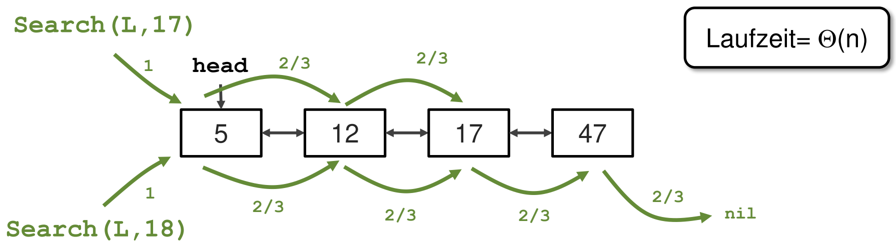


- **Elementare Operationen auf Listen**
  - *Suche nach Element*
    * Laufzeit beträgt im Worst Case $\Theta(n)$

      $\Rightarrow$ Keine Überprüfung, ob Wert bereits in Liste, sonst $\Theta(n)$
    * Code:

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;
```



- *Einfügen eines Elements am Kopf der Liste*
  * Laufzeit beträgt $\Theta(1)$, da Einfügen am Kopf
  * Code:

insert(L,x)

```
insert(L,x)
x.next = l.head;
x.prev = nil;
IF L.head != nil THEN
  L.head.prev = x;
L.head = x;
```

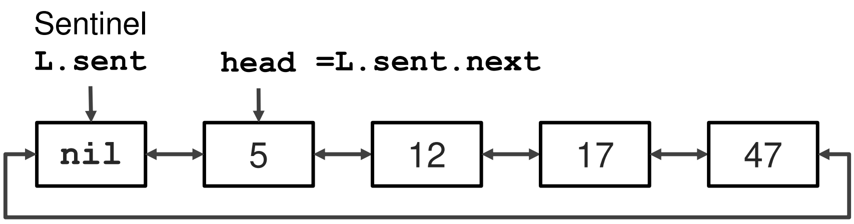
- *Löschen eines Elements aus Liste*
  * Laufzeit beträgt $\Theta(1)$, da hier Pointer auf Objekt gegeben

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

delete (L,x)

```
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;
```

- **Vereinfachung per Wächter/Sentinels**
  - Ziel ist die Eliminierung der Spezialfälle für Listenanfang/-ende



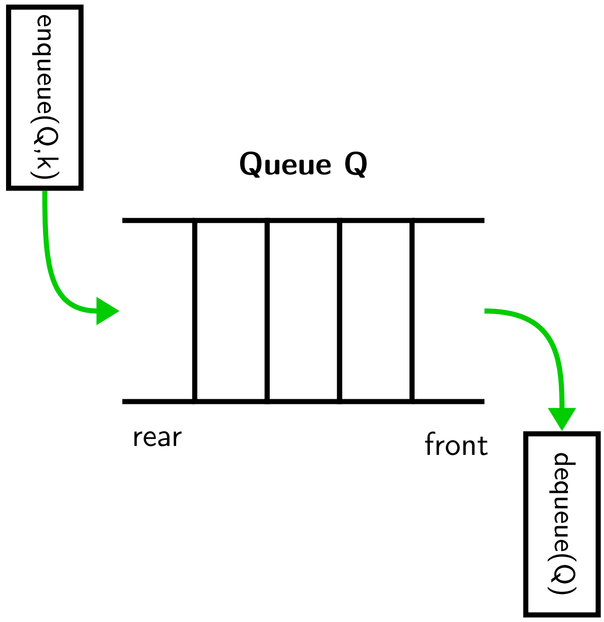
  - Löschen mit Sentinels:

deleteSent(L,x)

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

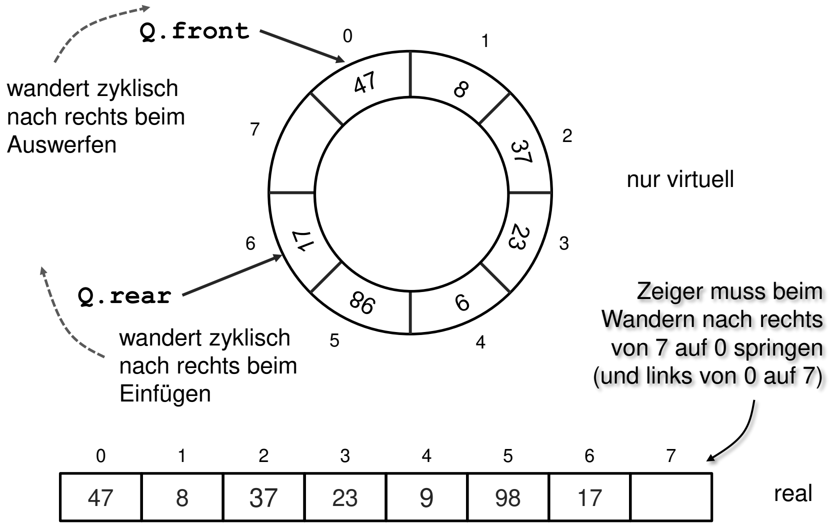
1.3 Queues

- **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)

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

isEmpty(Q)

```
return Q.empty;
```

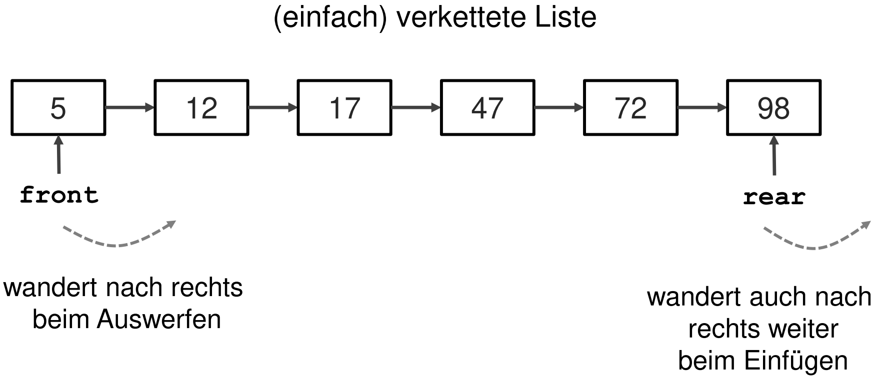
dequeue(Q)

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

enqueue(Q,k)

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

- **Queues durch einfach verkettete Listen**



Methoden:

isEmpty(Q)

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

new(Q)

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

dequeue(Q)

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

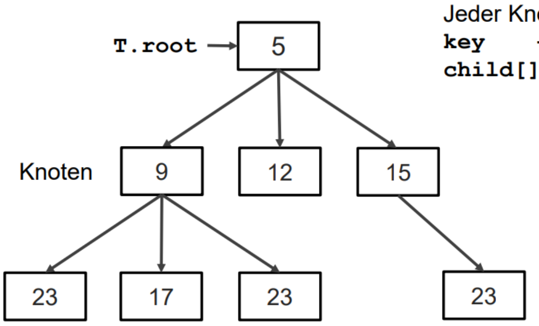
enqueue(Q,k)

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

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

1.4 Binäre Bäume

- **Bäume durch verkettete Listen**



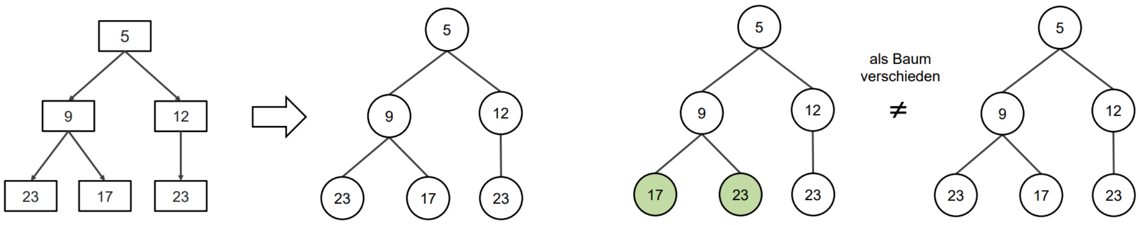
Jeder Knoten enthält:
`key` - Wert
`child[]` - Array von Zeigern auf Kinder

manchmal auch nützlich:
`parent` - Zeiger auf Elternknoten

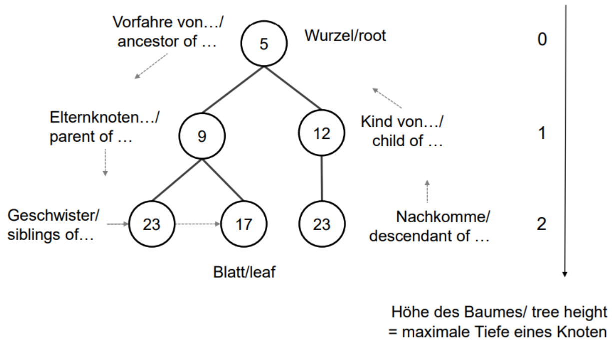
Baum-Bedingung: Baum ist leer oder…
es gibt einen Knoten **`r`** („Wurzel"), so dass jeder Knoten **`v`** von der Wurzel aus per eindeutiger Sequenz von **`child`**-Zeigern erreichbar ist:
`v = r.child[i1].child[i2].….child[im]`

Bäume sind „azyklisch"(also „keine Schleifen zwischen Knoten")

- **Darstellung als (ungerichteter) Graph**



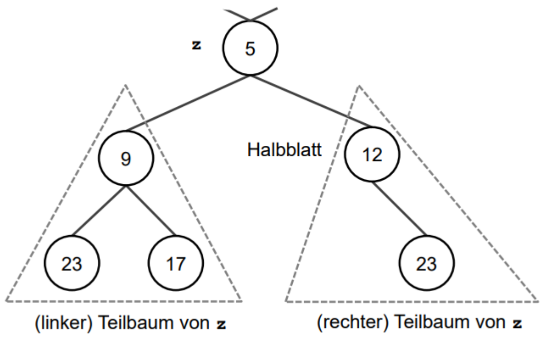
- **Allgemeine Begrifflichkeiten**



* Blatt: Knoten ohne Nachfolger

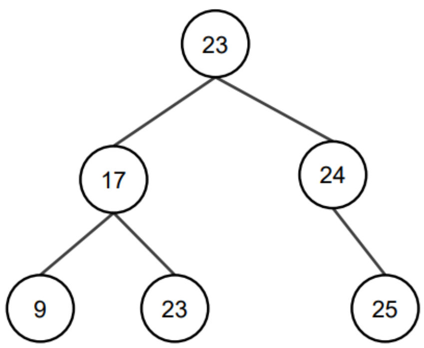
* Nachkomme von x:
  Erreichbar durch Pfad ausgehend von x

- **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
  * Hohe (nicht-leerer) Baum: max{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)

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

preorder(x)

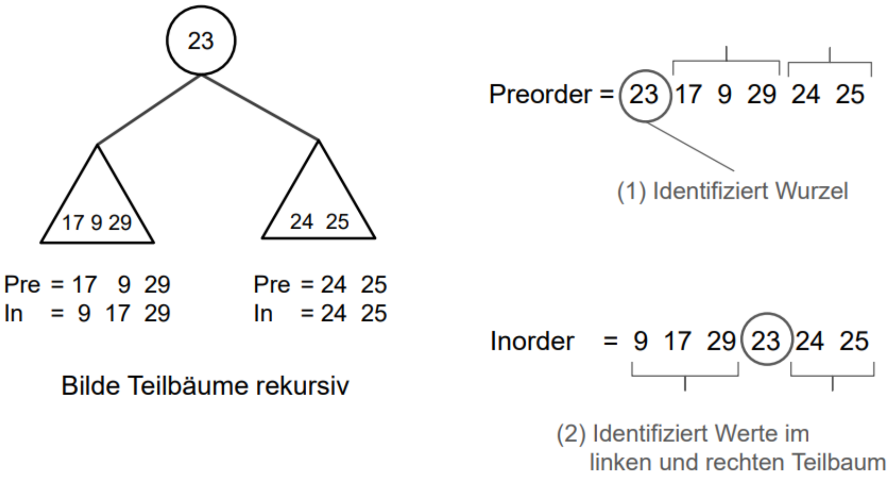
```
IF x != nil THEN
    print x.key;
    preorder(x.left);
    preorder(x.right);
```

postorder(x)

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

- **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)

```
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);
```

– *Einfügen von Elementen:*

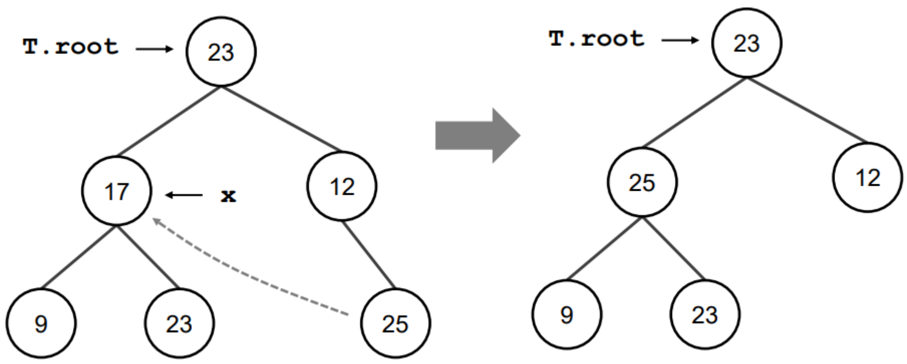
  * Laufzeit = $\Theta(1)$
  * Hier wird als Wurzel eingefügt (Achtung: Erzeugt linkslastigen Baum)
  * Code:

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

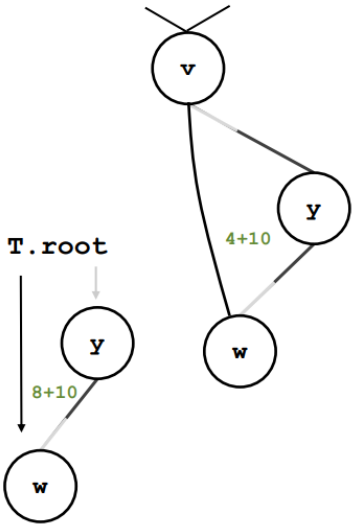
```
IF T.root != nil THEN
  T.root.parent = x;
  x.left = T.root;
T.root = x;
```

– *Löschen von Elementen:*

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



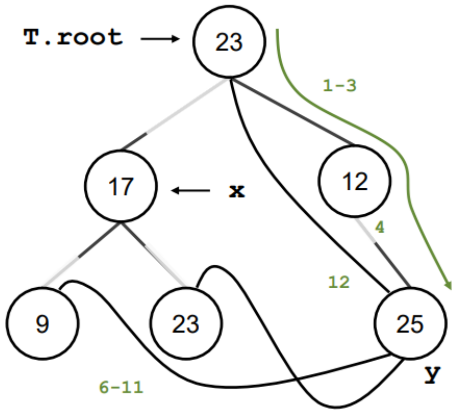
* Connect-Algorithmus:



· Laufzeit $= \Theta(1)$

```
connect(T,y,w)   //   Connects   w   to y.parent
v = y.parent;
IF y != T.root THEN
    IF y == v.right THEN
        v.right = w;
    ELSE
        v.left = w;
ELSE
    T.root = w;
IF w != nil THEN
    w.parent = v;
```

* Delete-Algorithmus:



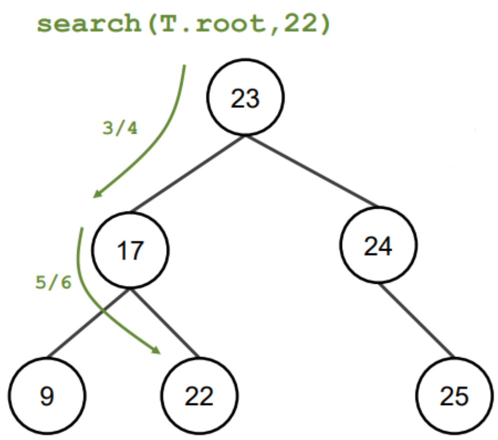

```
delete(T,x) // assumes x in T
y = T.root;
WHILE y.right != nil DO
    y = y.right;
connect(T,y,y.left);
IF x != y THEN
    y.left = x.left;
    IF x.left != nil THEN
        x.left.parent = y;
    y.right = x.right;
    IF x.right != nil THEN
        x.right.parent = y;
    connect(T,x,y);
```

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

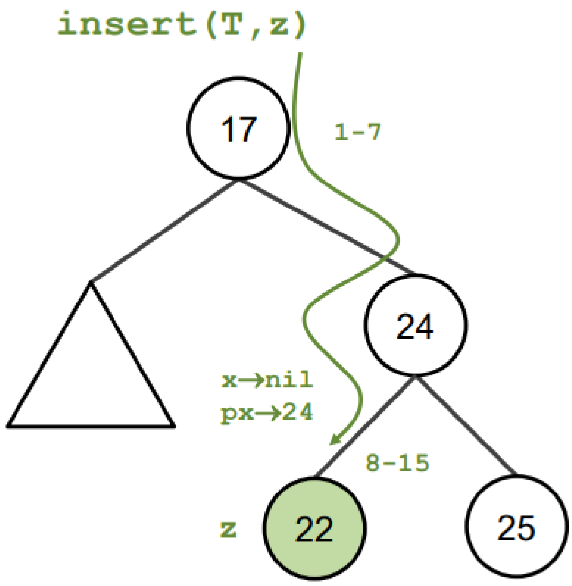
```
1 IF x == nil OR x.key == k THEN
2   return x;
3 IF x.key > k THEN
4   return search(x.left,k);
5 ELSE
6   return search(x.right,k);
```

* Iterativer Code:

iterative-search(x,k)

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

- **Einfügen im Binary Search Tree**



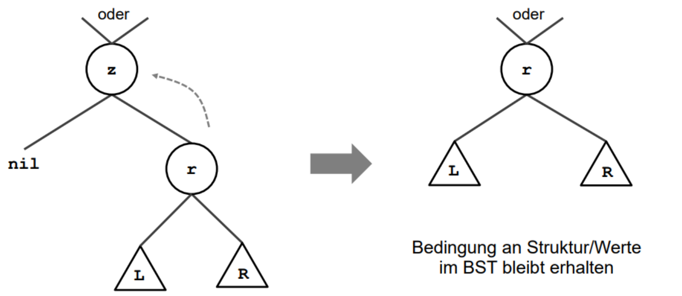
  * Laufzeit = $O(h)$
  * Aufwendiger, da Ordnung erhalten werden muss
  * Code:

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

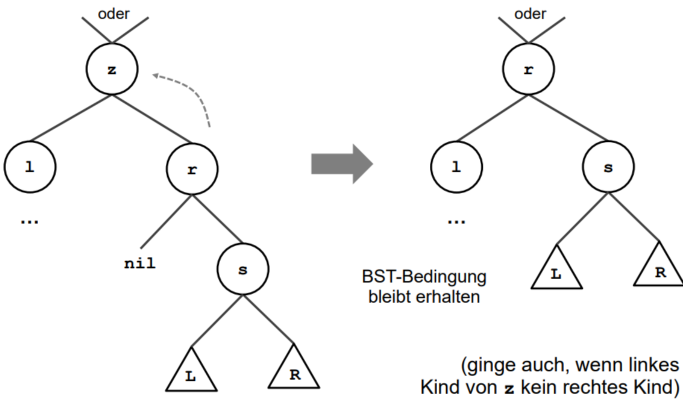
```
x = T.root;
px = nil;
WHILE x != nil DO
  px = x;
  IF x.key > z.key THEN
      x = x.left;
  ELSE
      x = x.right;
z.parent = px;
IF px == nil THEN
  T.root = z;
ELSE
  IF px.key > z.key THEN
      px.left = z;
  ELSE
      px.right = z;
```

- **Löschen im BST**
  - *Verschiedene Fälle:*

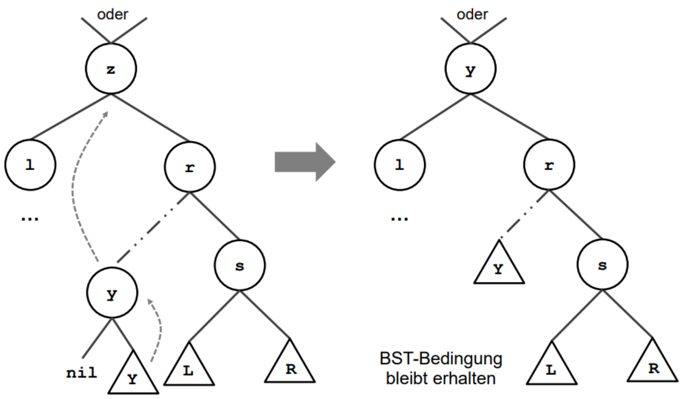
Löschen im BST (I) zu löschender Knoten **z** hat maximal ein Kind



Löschen im BST (II) rechtes Kind von Knoten **z** hat kein linkes Kind



Löschen im BST (III) „kleinster" Nachfahre vom rechten Kind von **z**



- *Code*

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

```
IF u.parent == nil THEN
    T.root = v;
ELSE
    IF u == u.parent.left THEN
        u.parent.left = v;
    ELSE
        u.parent.right = v;
IF v != nil THEN
    v.parent = u.parent;
```

delete(T,z)

```
IF z.left == nil THEN
    transplant(T,z,z.left)
ELSE
    IF z.right == nil THEN
        transplant(T,z,z,left)
    ELSE
        y = z.right;
        WHILE y.left != nil DO y = y.left;
        IF y.parent != z THEN
            transplant(T,y,y.right)
            y.right = z.right;
            y.right.parent = y;
        transplant(T,z,y)
        y.left = z.left;
        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 (links- bzw. rechtslastiger Baum)
    - $h = n - 1$
    - 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