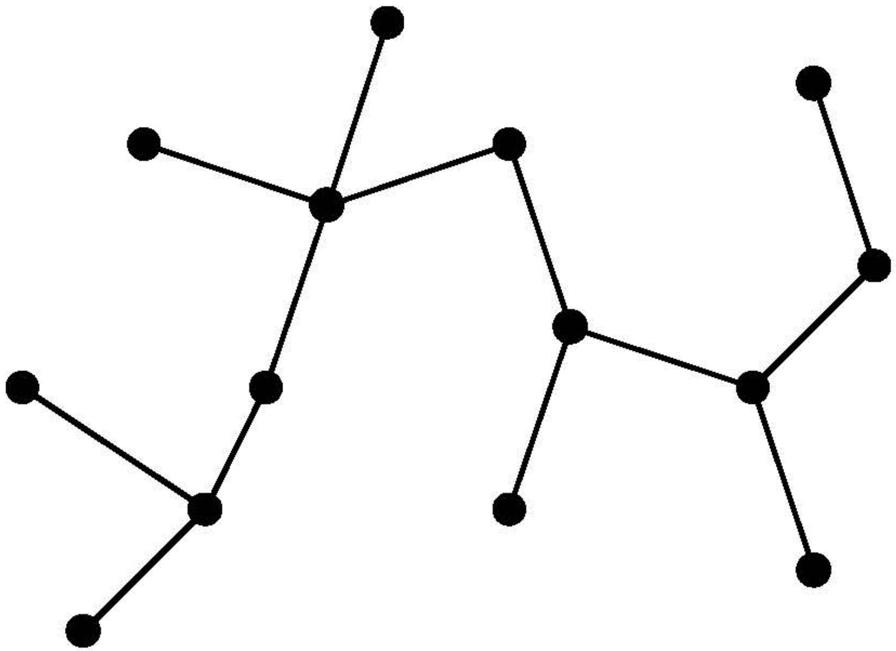
Binäre Bäume

Baum



(a)

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen Pfad verbunden

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn , ist G nicht mehr zusammenhängend

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante entfernt wird, ist G nicht mehr zusammenhängend

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante entfernt wird, ist G nicht mehr zusammenhängend
- G ist zusammenhängend und $|E|$ =

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante entfernt wird, ist G nicht mehr zusammenhängend
- G ist zusammenhängend und $|E| = |V|-1$

Baum

- $G = (V,E)$ ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante entfernt wird, ist G nicht mehr zusammenhängend
- G ist zusammenhängend und $|E| = |V|-1$
- G ist und $|G| = |V| - 1$

Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante entfernt wird, ist G nicht mehr zusammenhängend
- G ist zusammenhängend und $|E| = |V|-1$
- G ist azyklisch und $|G| = |V| - 1$

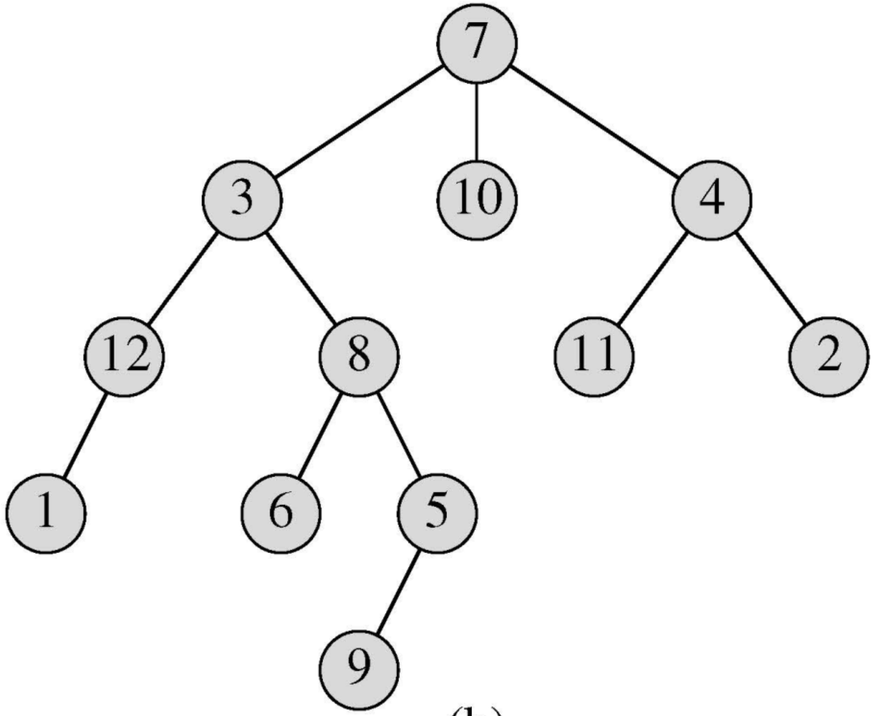
Baum

- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante entfernt wird, ist G nicht mehr zusammenhängend
- G ist zusammenhängend und |E| = |V|-1
- G ist azyklisch und |G| = |V| - 1
- G ist azyklisch, aber wenn , enthält G einen Zyklus

Baum

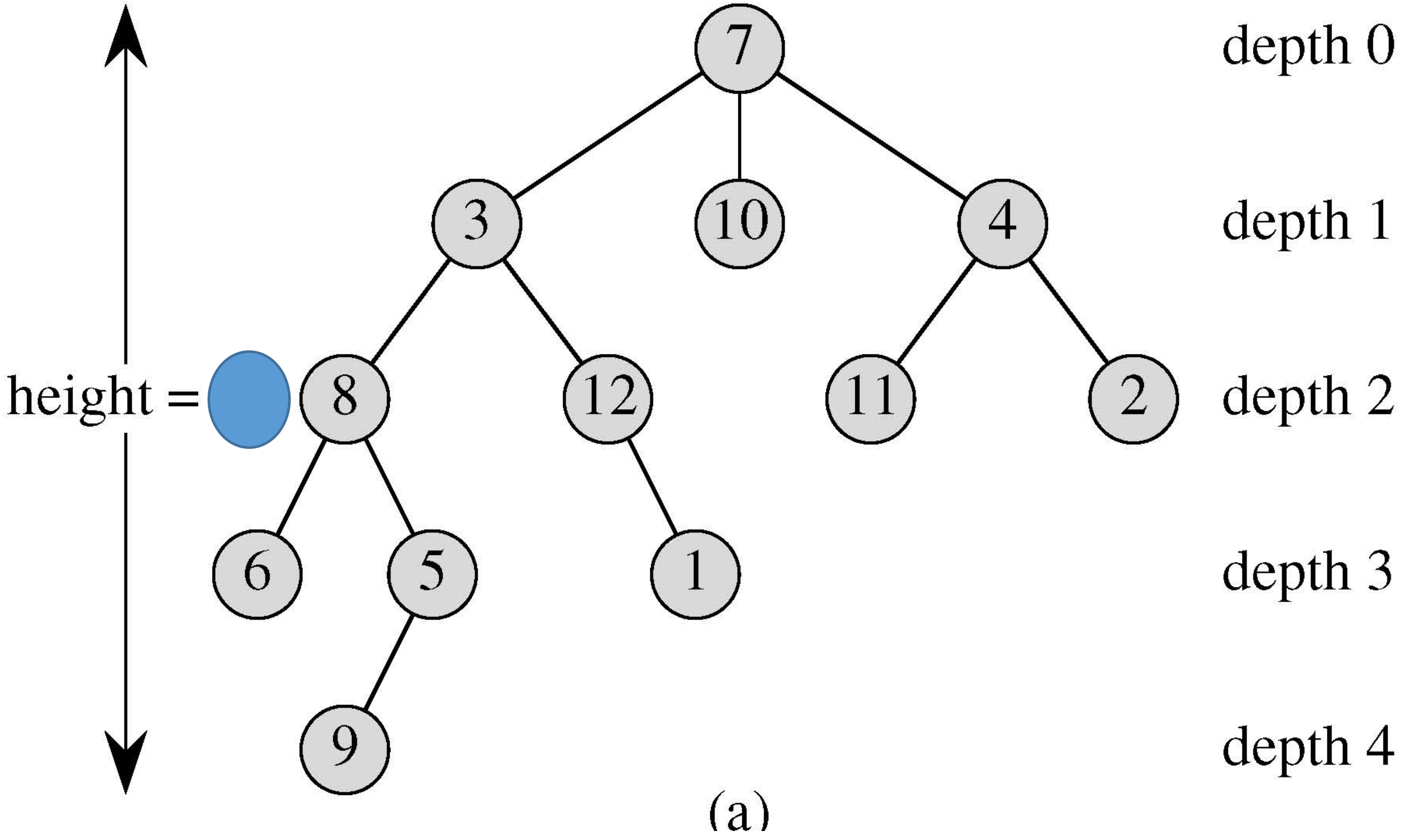
- G = (V,E) ungerichtet, azyklisch, zusammenhängend
- Je zwei Knoten in G sind durch einen einfachen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante entfernt wird, ist G nicht mehr zusammenhängend
- G ist zusammenhängend und $|E| = |V|-1$
- G ist azyklisch und $|G| = |V| - 1$
- G ist azyklisch, aber wenn eine Kante zu G hinzugefügt wird, enthält G einen Zyklus

Baum mit Wurzel

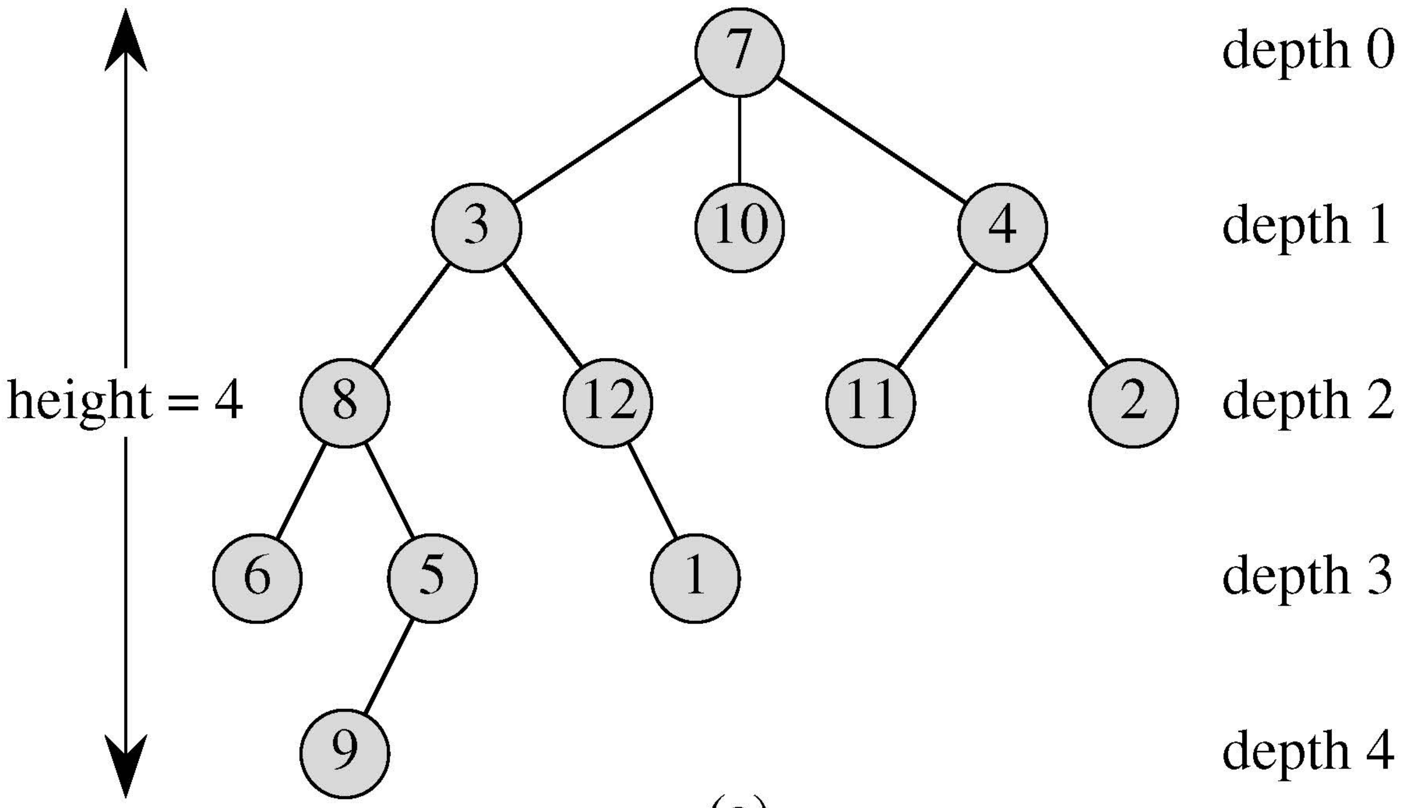


(b)

Tiefe von Knoten um Bäumen

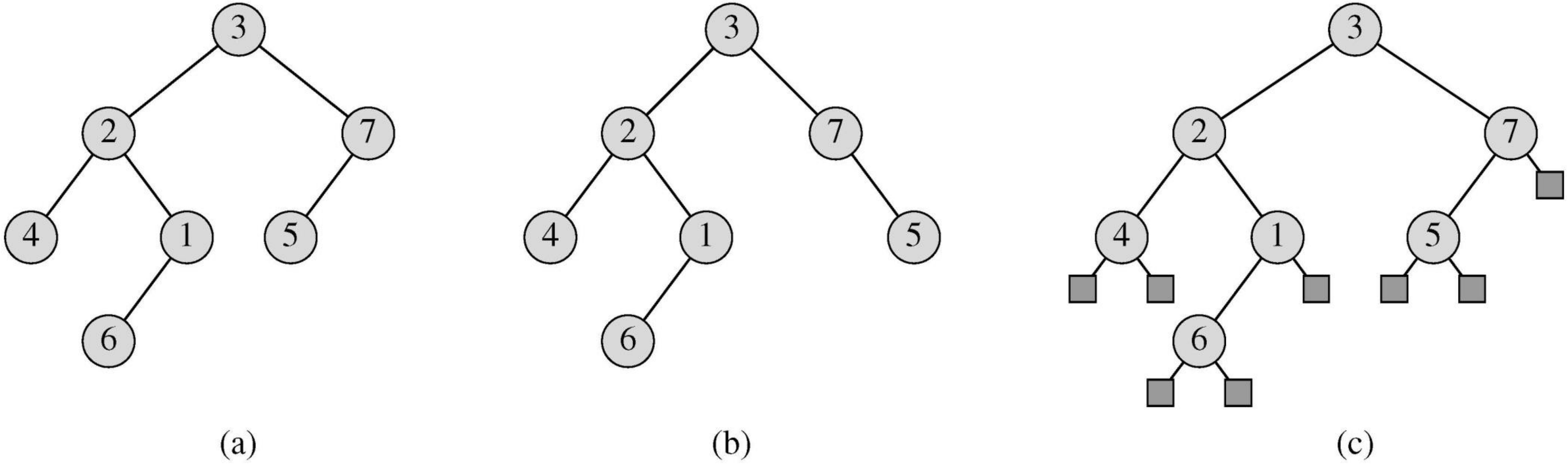


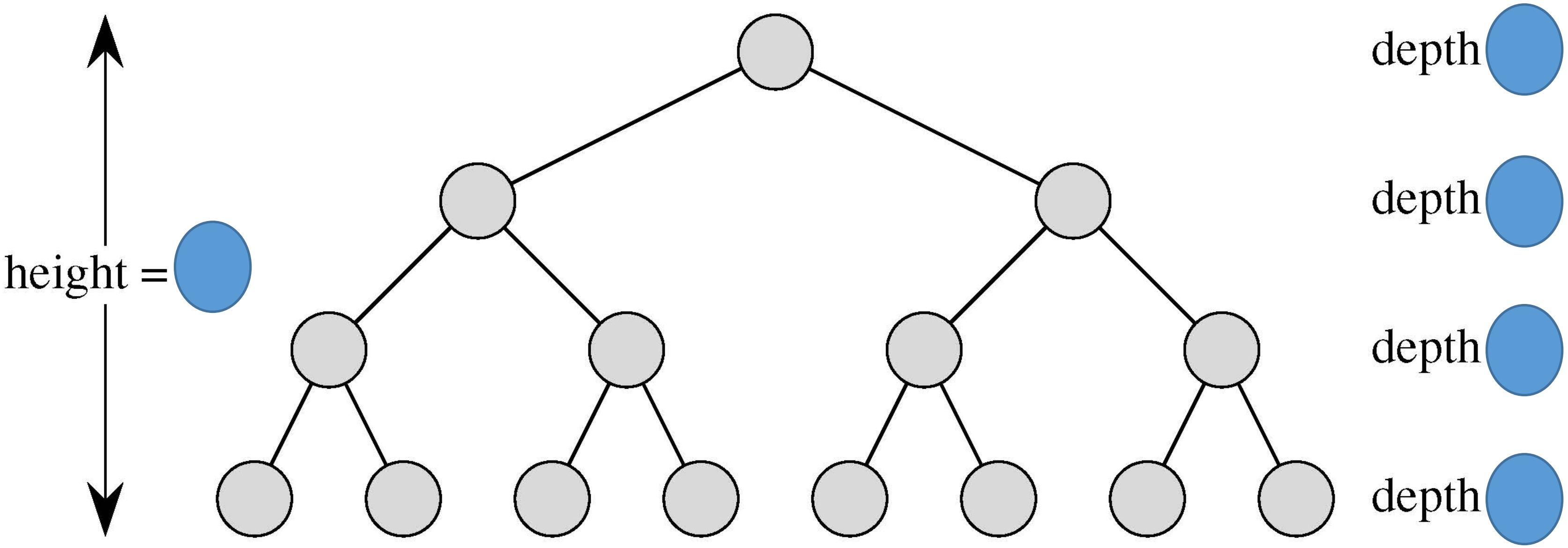
Tiefe von Knoten um Bäumen



(a)

Binäre Bäume



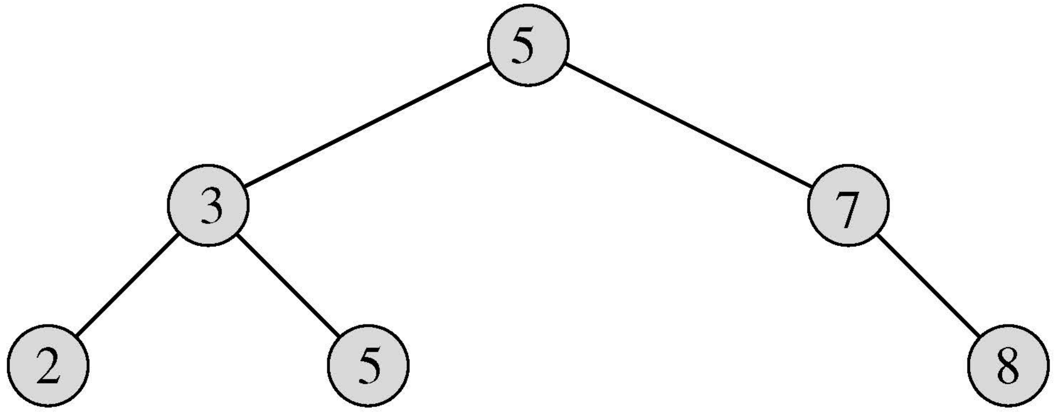

height =
depth
depth
depth
depth

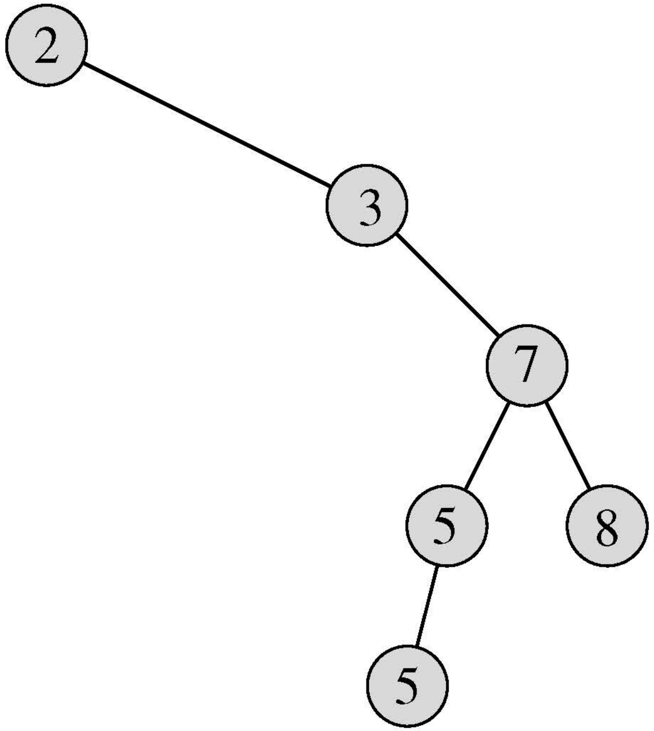
Verallgemeinerte verkettete Liste

Array

Array

Binäre Suchbäume von unterschiedlicher Effizienz





Was macht dieser Algorithmus?

```
INORDER-TREE-WALK(x)
if x ≠ NIL
   then INORDER-TREE-WALK(left[x])
        print key[x]
        INORDER-TREE-WALK(right[x])
```

Ausgeben der Knoten in der richtigen Reihenfolge

```
INORDER-TREE-WALK(x)
if x ≠ NIL
   then INORDER-TREE-WALK(left[x])
        print key[x]
        INORDER-TREE-WALK(right[x])
```

Was macht dieser Algorithmus?

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else  return TREE-SEARCH(right[x], k)
```

Sucht rekursiv nach Knoten x mit key[x] = k

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else  return TREE-SEARCH(right[x], k)
```

Komplexität?

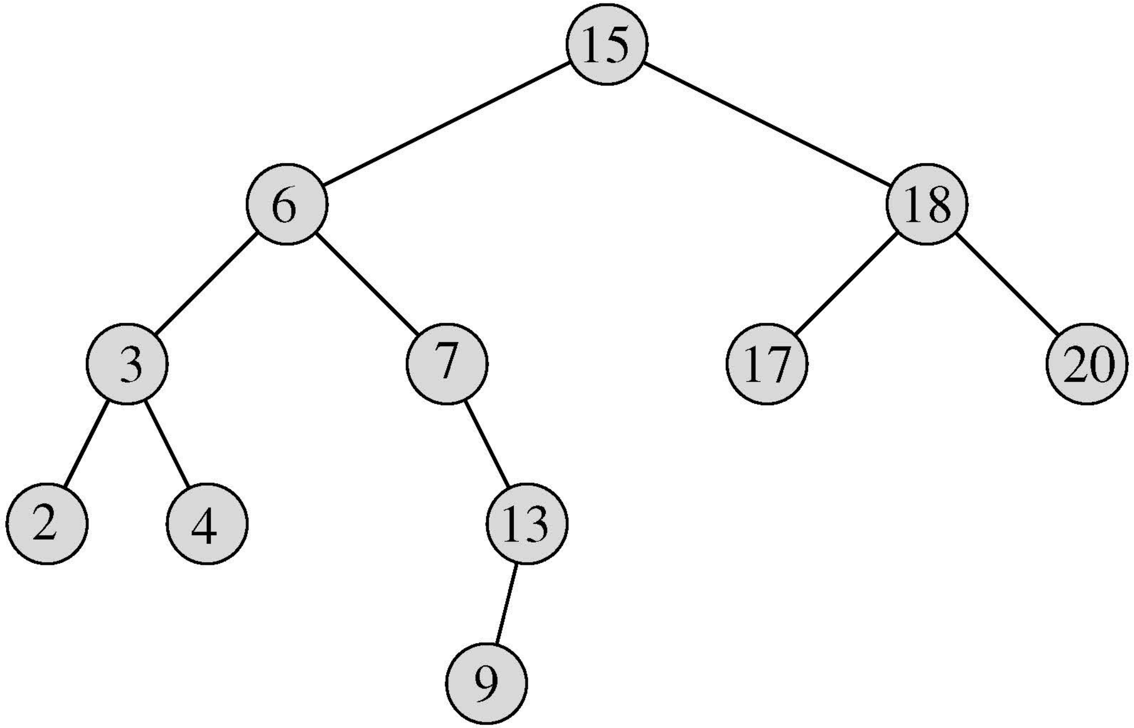
```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else return TREE-SEARCH(right[x], k)
```

Komplexität

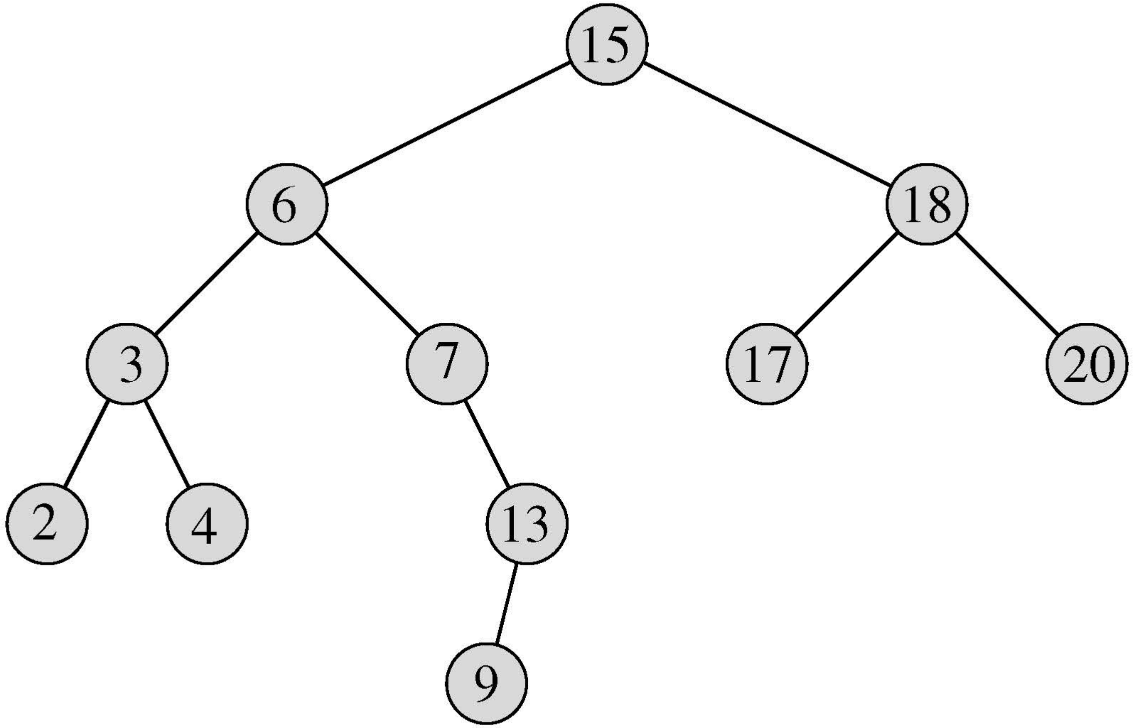
Komplexität = O(Höhe von x)

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else  return TREE-SEARCH(right[x], k)
```

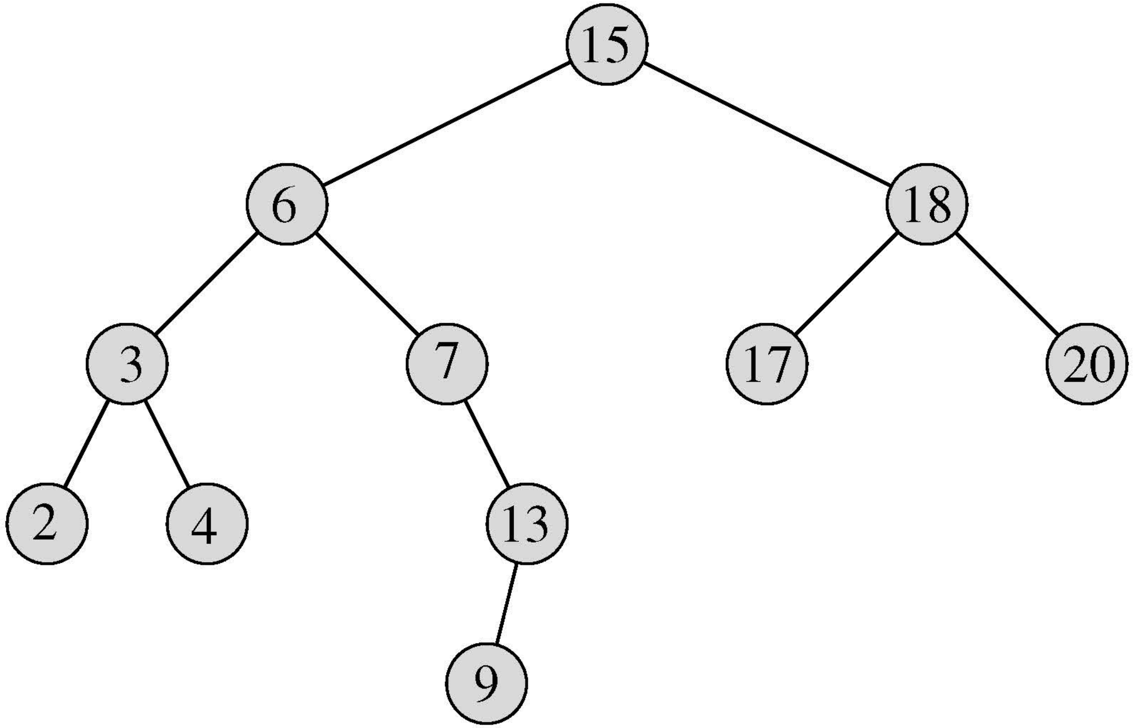
Suche nach 13



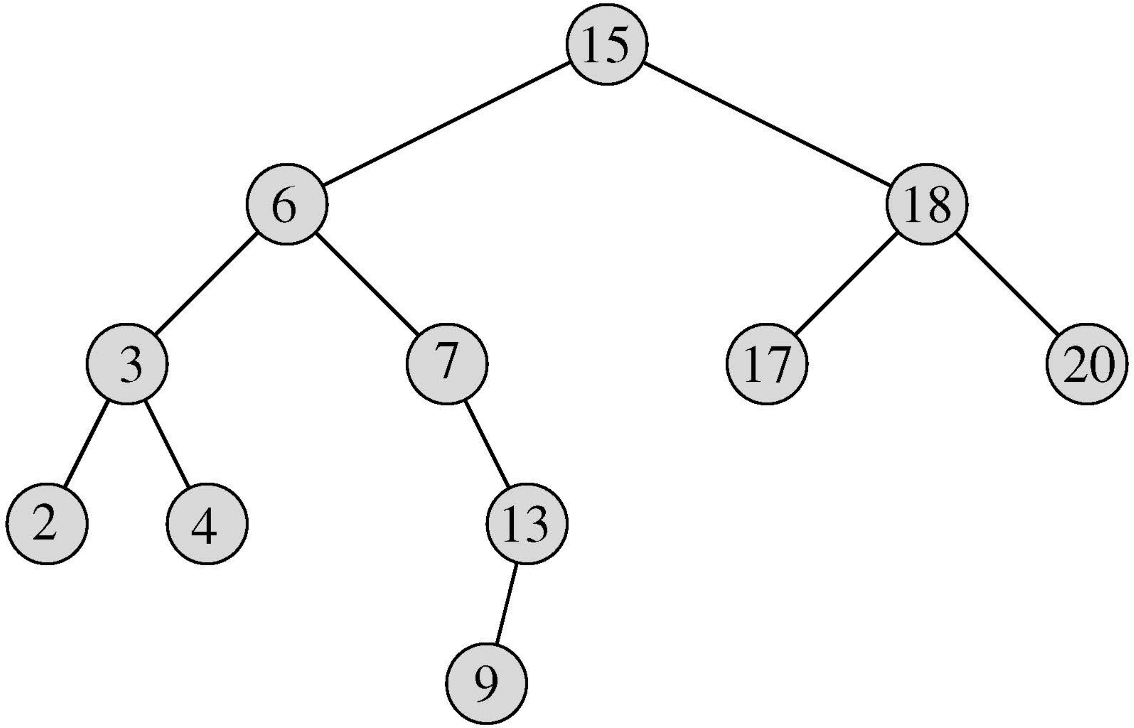
Suche nach 13



Suche nach 19



Suche nach 19



Was macht dieser Algorithmus?

ITERATIVE-TREE-SEARCH(x, k)

while $x \neq$ NIL and $k \neq key[x]$
    do if $k < key[x]$
        then $x \leftarrow left[x]$
        else $x \leftarrow right[x]$
return x

Sucht iterativ nach Knoten x mit key[x] = k

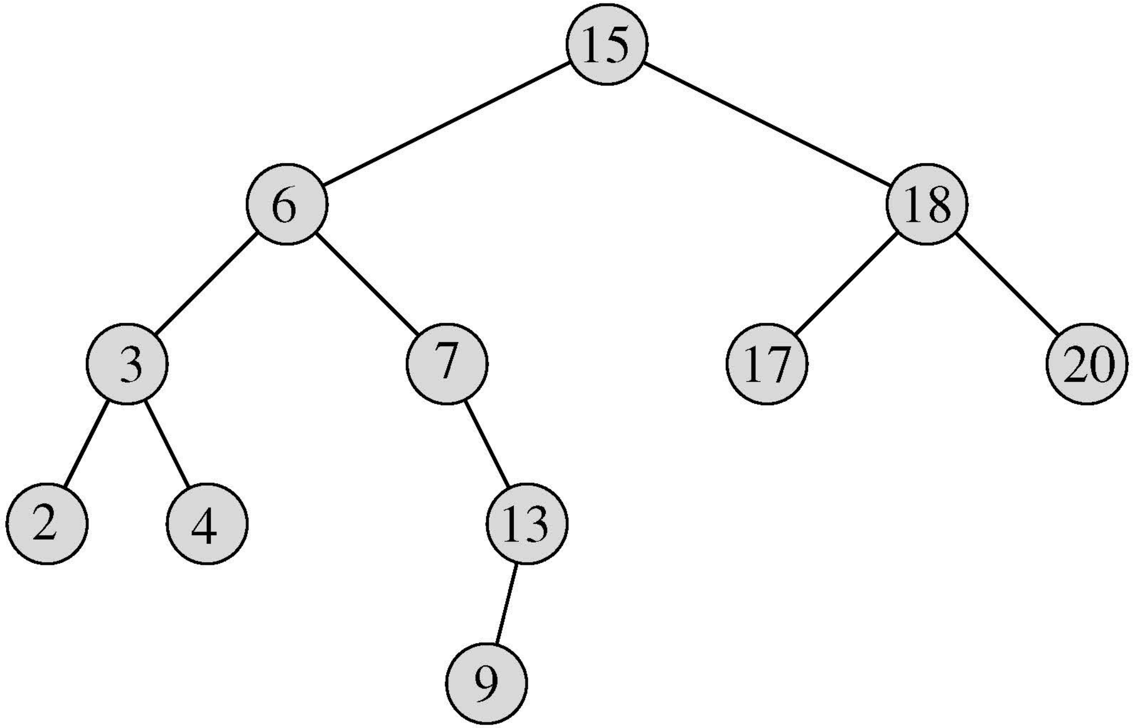
ITERATIVE-TREE-SEARCH(x, k)

while $x \neq \text{NIL}$ **and** $k \neq key[x]$
    do if $k < key[x]$
        then $x \leftarrow left[x]$
        else $x \leftarrow right[x]$
return x

Komplexität = O(Höhe von x)

```
ITERATIVE-TREE-SEARCH(x, k)
  while x ≠ NIL and k ≠ key[x]
      do if k < key[x]
            then x ← left[x]
            else  x ← right[x]
  return x
```

Finde Minimum



```
TREE-MINIMUM(x)
    while
        do
    return x
```

TREE-MINIMUM(x)

```
while left[x] ≠ NIL
    do x ← left[x]
return x
```

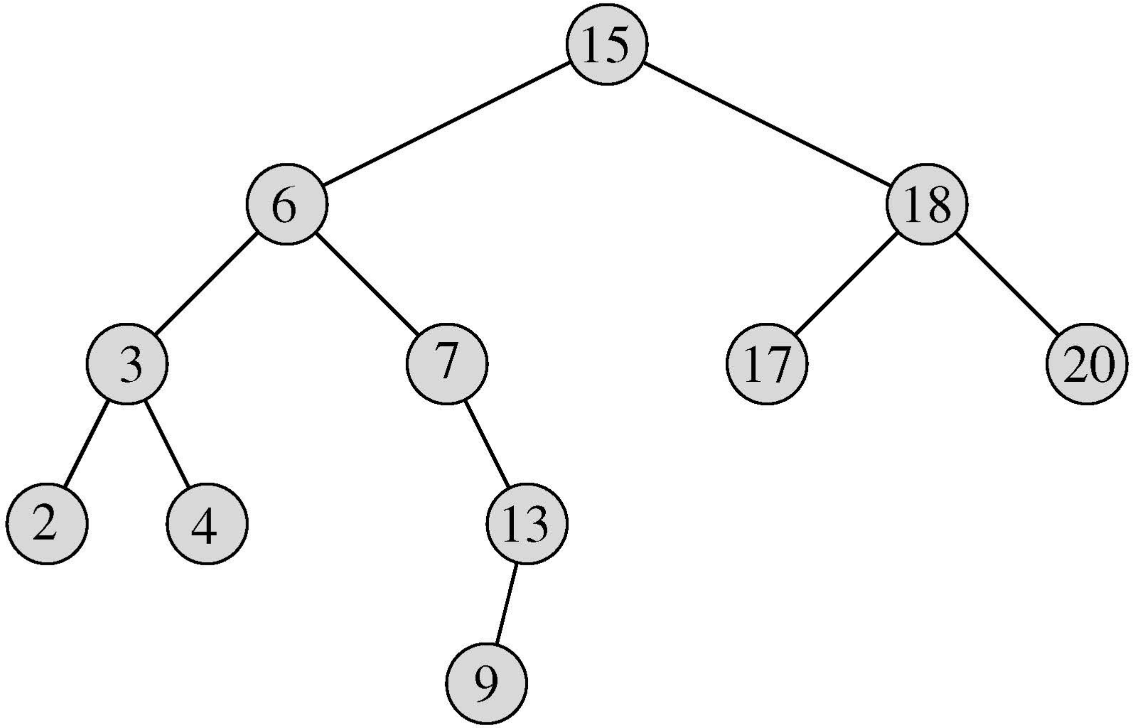
TREE-MAXIMUM(x)

```
while right[x] ≠ NIL
    do x ← right[x]
return x
```

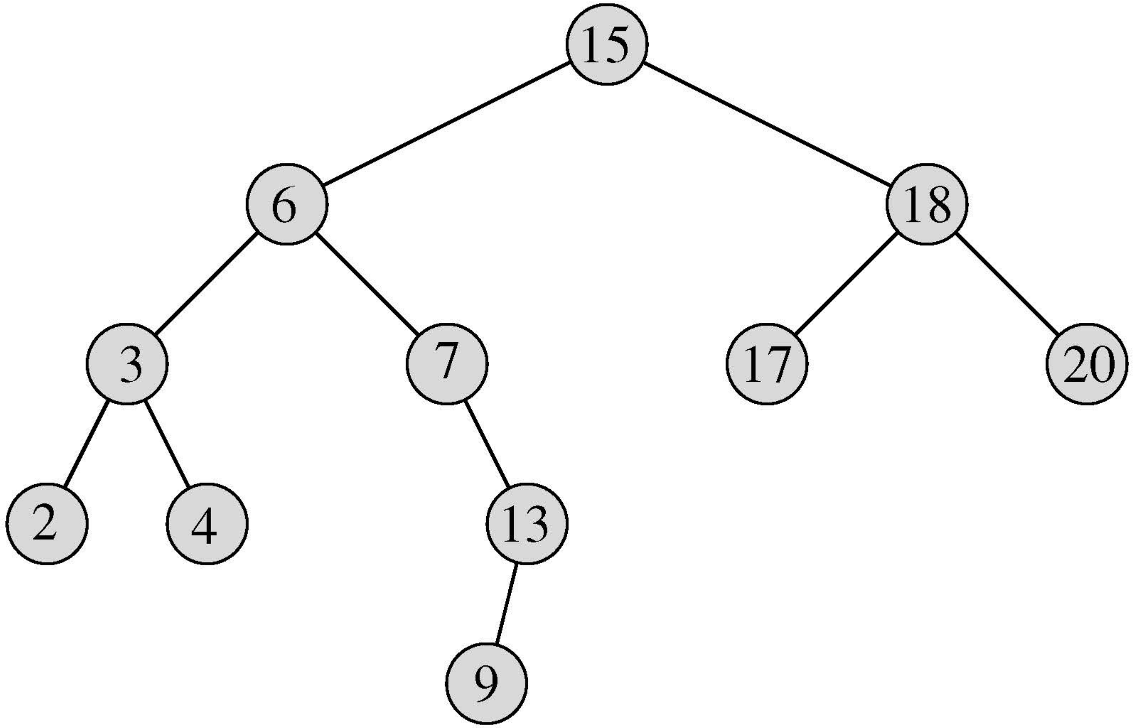
```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and
      do x ← y
         y ← p[y]
return y
```

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
      do x ← y
         y ← p[y]
return y
```

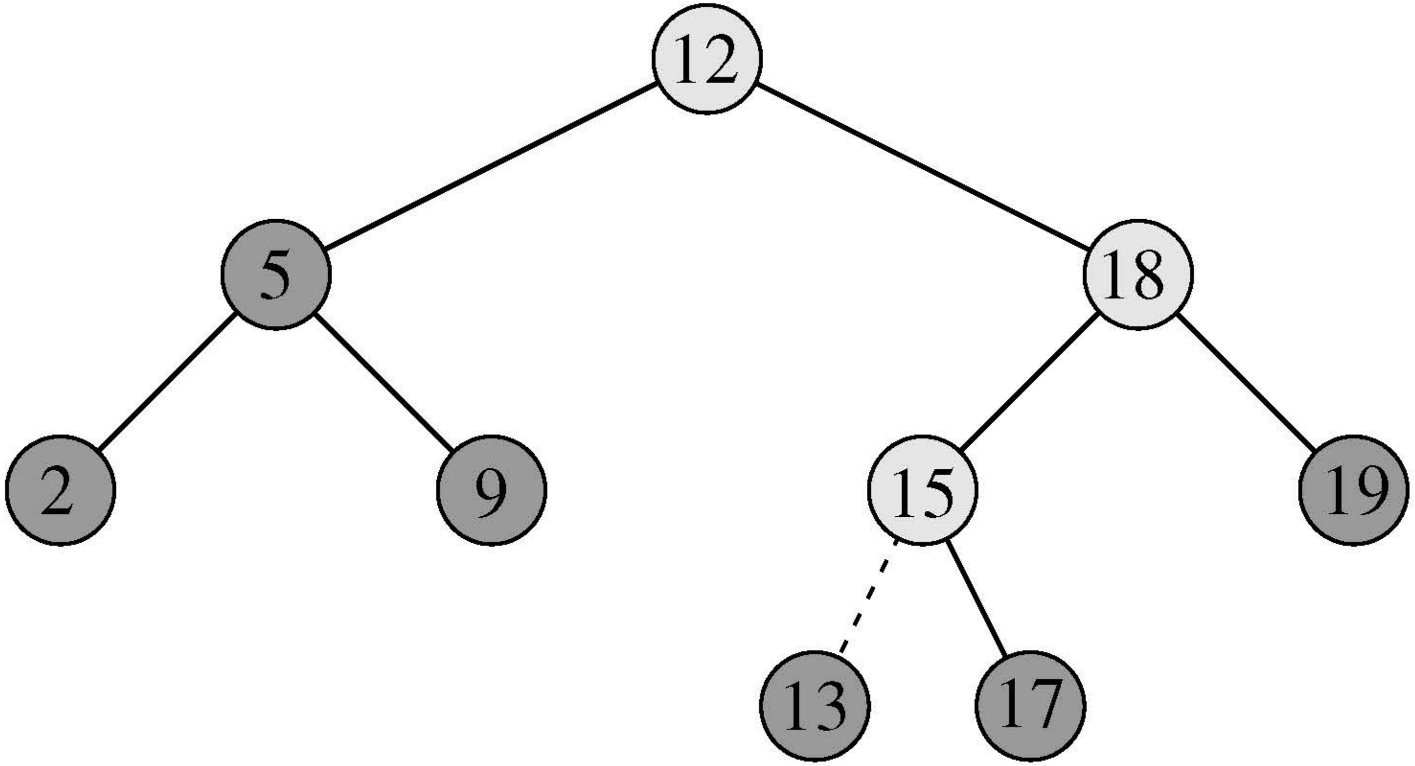
Nachfolger von 4



Nachfolger von 15



Einfügen von 13



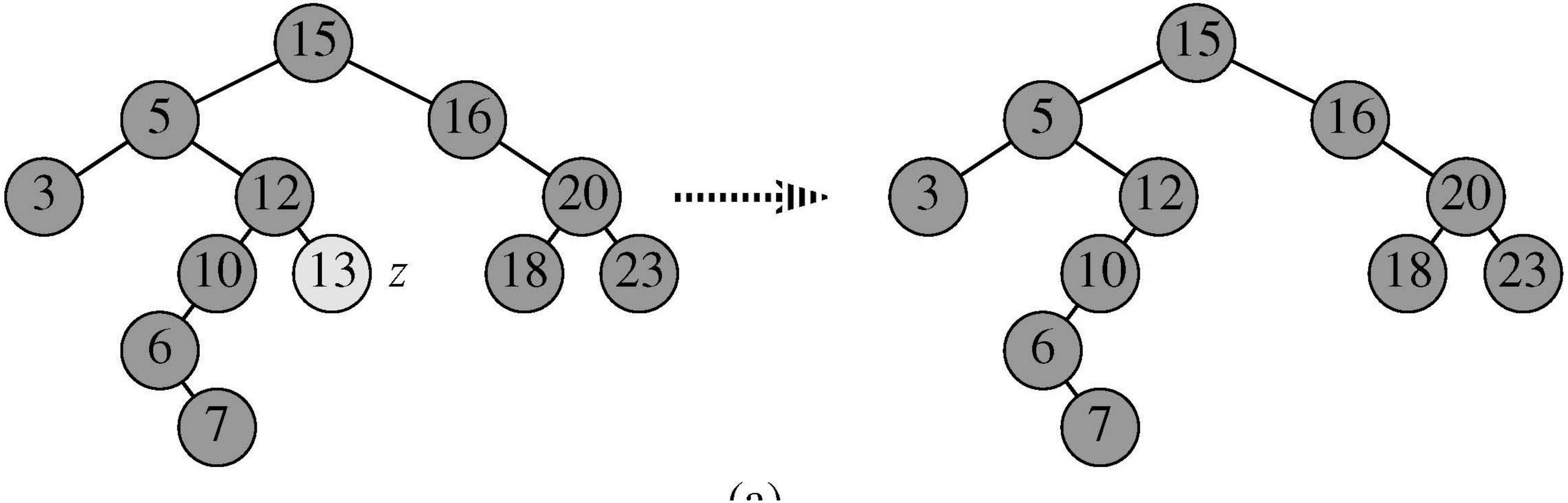
Einfügen

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if
            then x ← left[x]
            else  x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z
   else  if key[z] < key[y]
            then left[y] ← z
            else  right[y] ← z
```

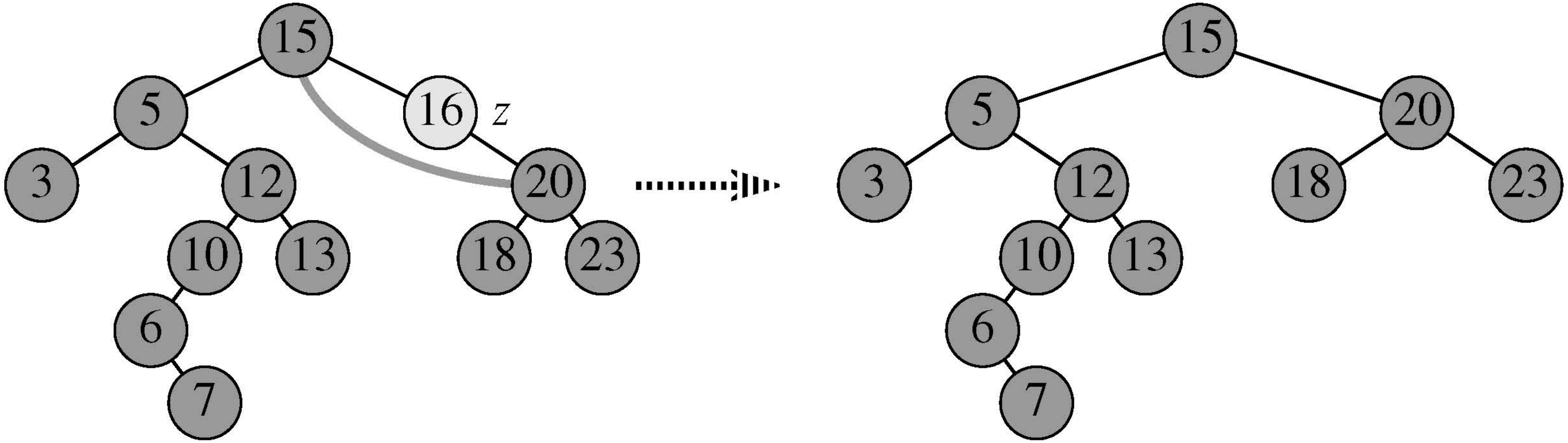
Einfügen

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                    ▷ Tree T was empty
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
```

Löschen von Blatt 13

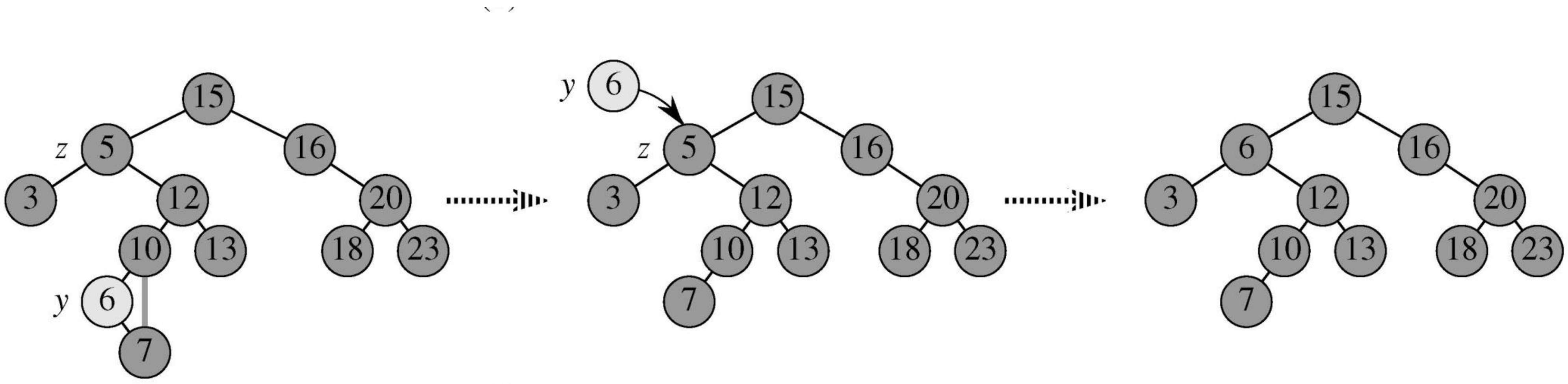


Löschen von Knoten 16 mit einem Kind



(b)

Löschen von Knoten 5 mit zwei Kindern



(c)

Löschen

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else  y ←
if left[y]
    then x ← left[y]
    else  x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then
    else  if y = left[p[y]]
              then left[p[y]] ← x
              else  right[p[y]] ← x
if y ≠ z
    then

return y
```

Löschen

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
if left[y]
    then x ← left[y]
    else  x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then
    else  if y = left[p[y]]
              then left[p[y]] ← x
              else  right[p[y]] ← x
if y ≠ z
    then

return y
```

Löschen

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else  x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then
    else  if y = left[p[y]]
             then left[p[y]] ← x
             else  right[p[y]] ← x
if y ≠ z
    then

return y
```

Löschen

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```