8 Binäre Suchbäume

Einführung

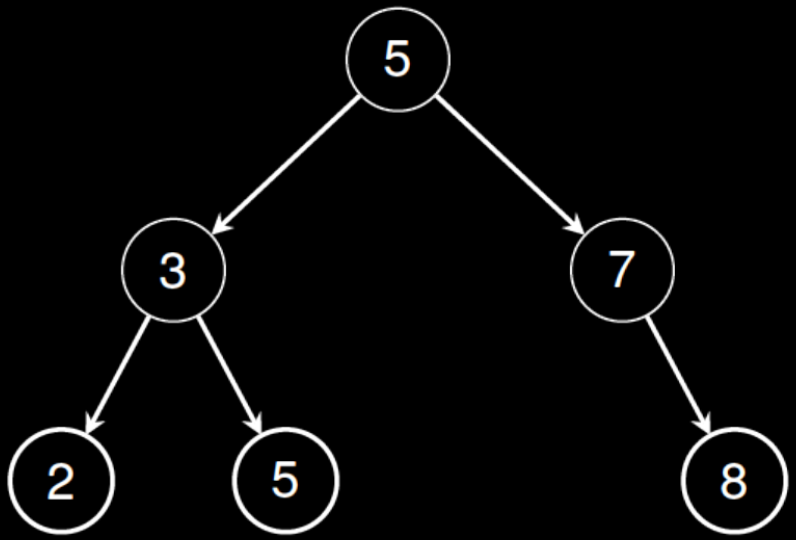
Definition

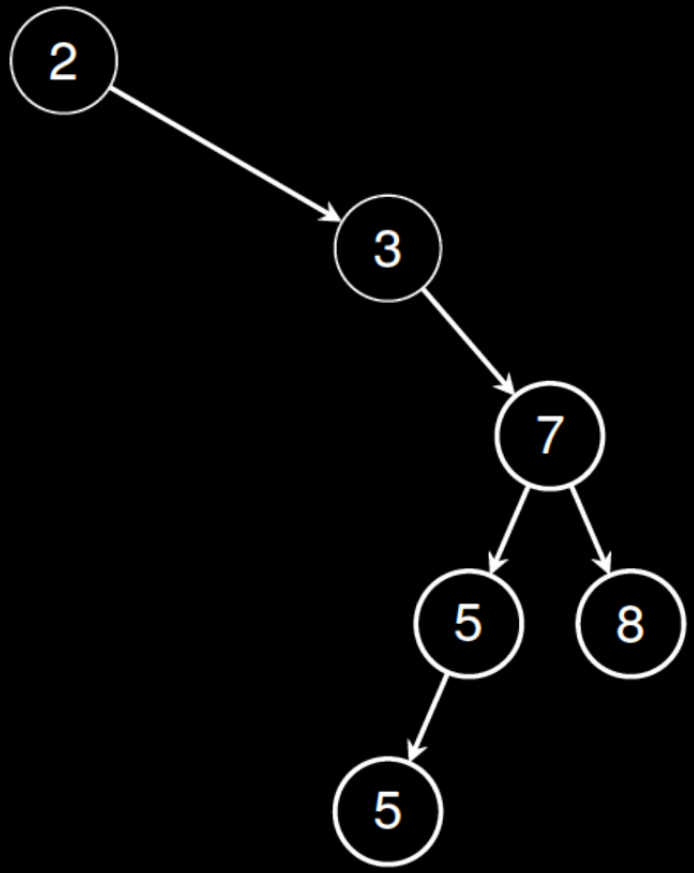
Ein binärer Suchbaum (binary search tree, BST) ist ein binärer Baum, für den die folgende Eigenschaft gilt (=binäre Suchbaum-Eigenschaft oder BST-Eigenschaft):

Sei *x* ein Knoten in einem binären Suchbaum.

- Ist *y* ein Knoten im linken Teilbaum von *x* , so gilt $y.key \leq x.key$.
- Ist *y* ein Knoten im rechten Teilbaum von *x* , so gilt $y.key \geq x.key$.

Beispiele





Bemerkungen

- Wir betrachten BST’s als verzeigerte Datenstruktur.
- Jeder Knoten ist ein Objekt mit
  - Schlüssel
  - Satellitendaten
  - Zeiger auf linkes Kind
  - Zeiger auf rechtes Kind
  - Zeiger auf Elternknoten

- Wir werden die BST-Eigenschaft für die Anwendung ein wenig strenger fassen

Notation

- *T* Der BST selber wird mit *T* bezeichnet.
- *x, y, z* Knoten.
- *x.key* Der Schlüssel des Knoten *x* .
- *x.left* Zeiger auf linkes Kind.
- *x.right* Zeiger auf rechtes Kind.
- *x.parent* Zeiger auf Elternknoten.
- *NIL* Leerer Zeiger.

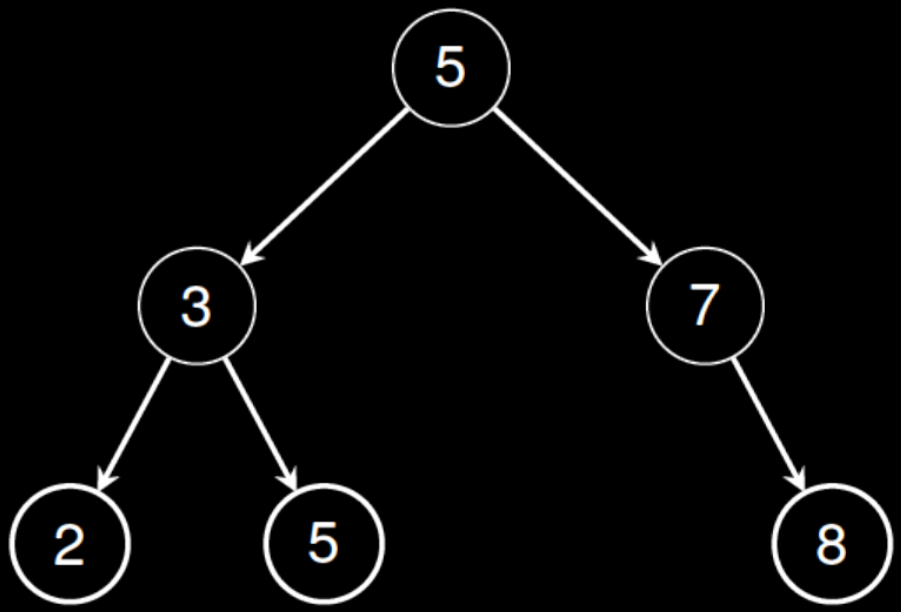
Traversierung

- Baum durchlaufen, durchsuchen
- Systematisches Untersuchen der Knoten in einer bestimmten Reihenfolge

Möglichkeiten einen Baum zu durchlaufen:

- Pre-order (Hauptreihenfolge) WLR
- In-order (Nebenreihenfolge) LWR
- Post-order (symmetrische Reihenfolge) LRW
- Level-order (Ebenen-Reihenfolge, nur mit zusätzlichen Zeigern möglich)

Traversierung: In-Order-Tree-Walk




```
1.  function IN-ORDER-TREE-WALK(x)
2.      if x ≠ NIL then
3.          IN-ORDER-TREE-WALK (x.left)
4.          print  x.key
5.          IN-ORDER-TREE-WALK (x.right)
6.      end if
7.  end function
```

Was gibt In-Order-Tree-Walk aus? 2 3 5 5 7 8

Traversierung: In-Order-Tree-Walk

Korrektheit von In-Order-Tree-Walk

Ergibt sich intuitiv aus der Definition von binären Suchbäumen: Wenn

1. der linke Teilbaum jedes Knoten x nur Elemente mit Schlüsseln $\leq x.key$ enthält

und

1. der rechte Teilbaum jedes Knoten x nur Elemente mit Schlüsseln $\geq x.key$ enthält

dann werden die Knoten in aufsteigender Reihenfolge besucht.

Traversierung: Laufzeit

Annahme:

- Sei *x* Wurzel eines Teilbaums mit *n* Knoten.
- Da jeder Knoten einmal betrachtet wird, hat der Algorithmus In-Order-Tree-Walk(x) die Laufzeit

$$T(n) = \Theta(n)$$

Traversierung: Laufzeit

Beweis:

- Sei $T(n)$ die Zeit, die In-Order-Tree-Walk beim Aufruf mit der Wurzel eines Teilbaums mit n Knoten benötigt.

$n = 0$: Für den Test ($x = NIL$) wird eine kleine konstante Zeit benötigt, so dass $T(0) = c$, mit $c > 0$.

$n > 0$: Sei x ein Knoten, dessen linker Teilbaum k Knoten enthalte. Dann enthält der rechte Teilbaum $n - k - 1$ Knoten.

Traversierung: Laufzeit

Beweis (Forts.)

- Die Laufzeit beträgt dann $T(n) = T(k) + T(n - k - 1) + d$

  $d > 0$ steht für die eigentliche Laufzeit von In-Order-Tree-Walk mit Ausnahme der Zeit für rekursive Aufrufe.

- Es soll nun bewiesen werden, dass

  $$T(n) = (c + d) \cdot n + c$$

  wobei gilt $T(n) = \Theta(n)$

Traversierung: Laufzeit

Für $n = 0$: $(c + d) \cdot 0 + c = c = T(0)$

Für $n > 0$: $T(n) = T(k) + T(n - k - 1) + d$

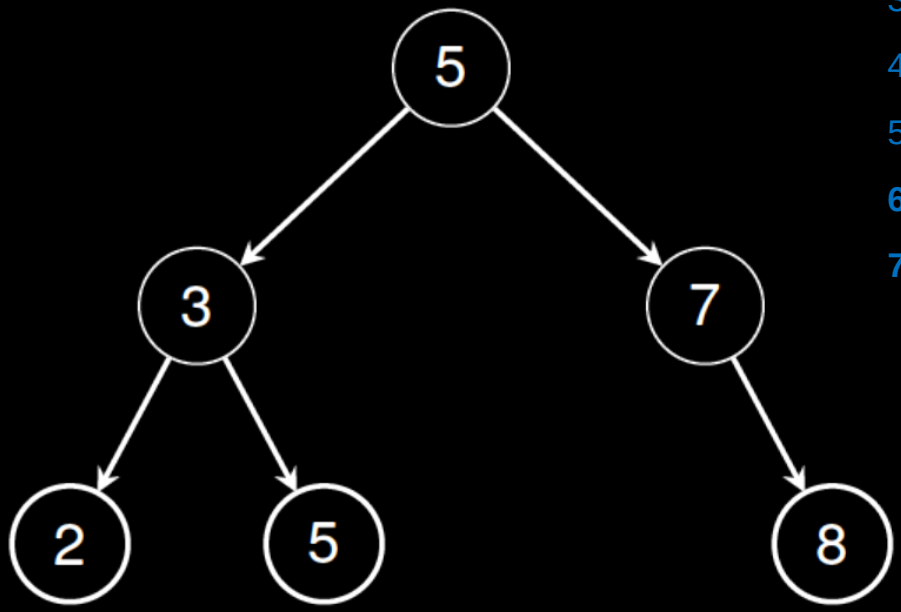
$= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d$

$= (c + d)k + c + (c + d)n - (c + d)k - (c + d) + c + d$

$= (c + d)n + c - (c + d) + c + d$

$= (c + d)n + c = \Theta(n)$

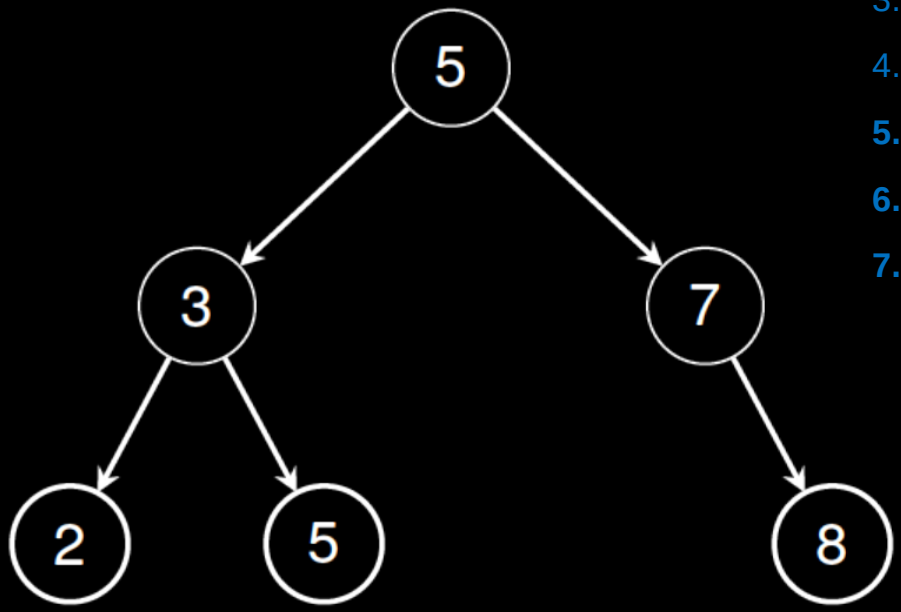
Traversierung: Pre-Order-Tree-Walk




```
1.  function PRE-ORDER-TREE-WALK(x)
2.      if x ≠ NIL then
3.          print  x.key
4.          PRE-ORDER-TREE-WALK (x.left)
5.          PRE-ORDER-TREE-WALK (x.right)
6.      end if
7.  end function
```

Was gibt Pre-Order-Tree-Walk aus? 5 3 2 5 7 8

Traversierung: Post-Order-Tree-Walk




```
1.  function POST-ORDER-TREE-WALK(x)
2.      if x ≠ NIL then
3.          POST-ORDER-TREE-WALK (x.left)
4.          POST-ORDER-TREE-WALK (x.right)
5.          print  x.key
6.      end if
7.  end function
```

Was gibt Post-Order-Tree-Walk aus? 2 5 3 8 7 5

Weitere Eigenschaften

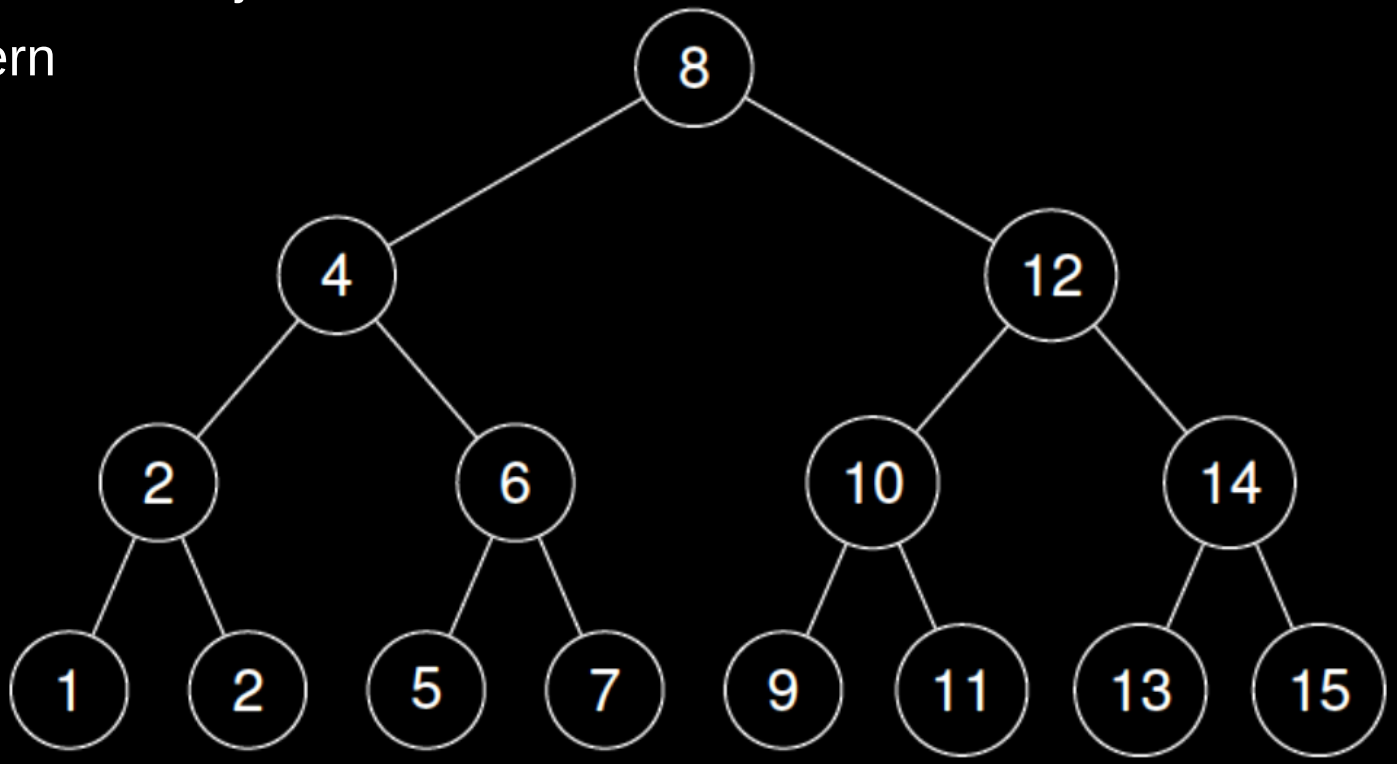
- Knotenanzahl eines Binärbaums
- Höhe eines Binärbaums

- Wichtig: Laufzeit der Algorithmen ist meist von der Höhe h eines Baums abhängig

Knotenanzahl

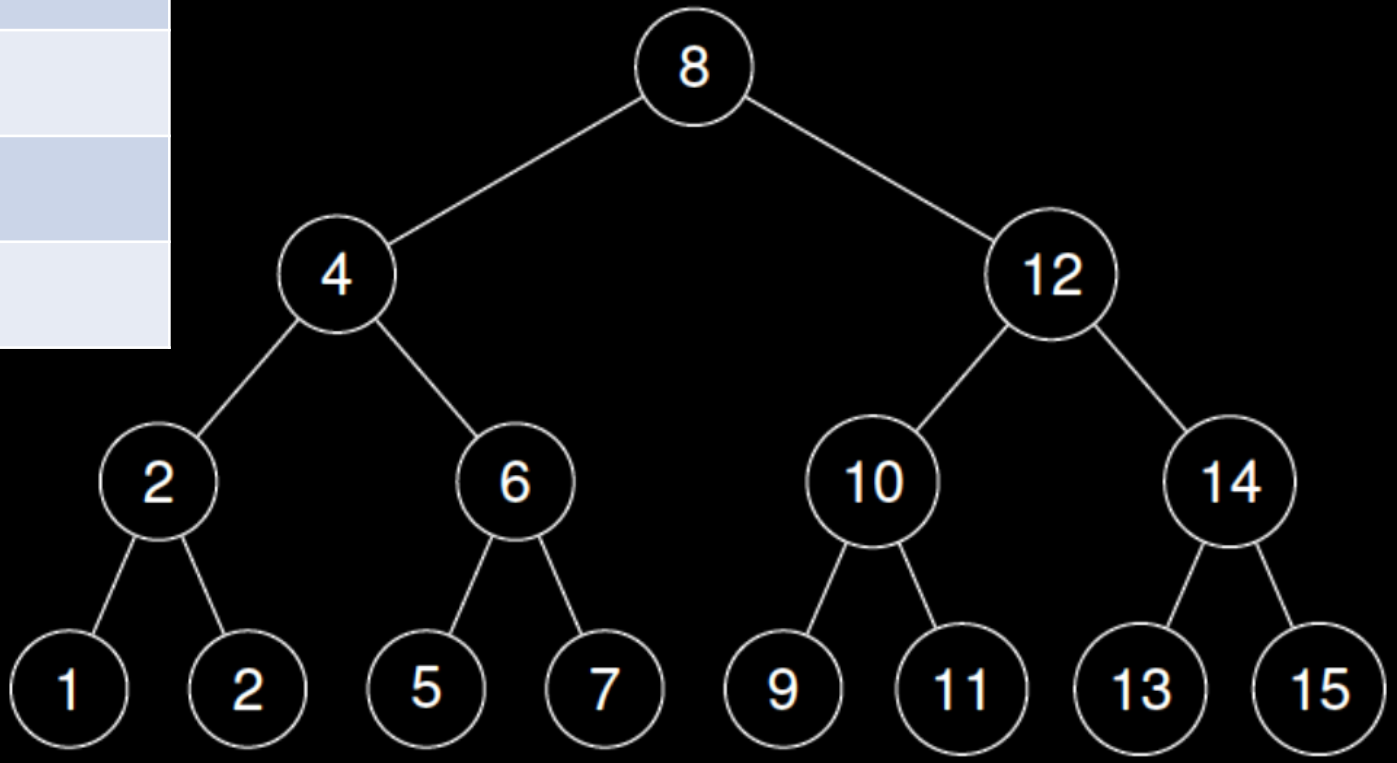
Beispiel

- Abzählen der Knoten in jeder Ebene
- Verallgemeinern
- Summieren



Knotenanzahl

Ebene	Anzahl Knoten
0	1
1	2
2	4
3	8



Knotenanzahl

Zusammenfassung

Ein Binärbaum der Höhe h hat maximal

$$n(h) = \underbrace{1 + 2 + 4 + \cdots + 2^h}_{geometrische\ Reihe}$$

$$= \sum_{i=0}^{h} 2^i$$

$$= 2^{h+1} - 1$$

Knoten.

Man nennt diese max. Knotenzahl auch die Kapazität des Baums.

Höhe

Beispiel

Anzahl Knoten	Min. Höhe	Max. Höhe
1	0	0
2	1	1
3	1	2
4	2	3
5	2	4

Höhe

Allgemein

- Sei *T* ein Binärbaum und *n* die Anzahl der Knoten von *T* , dann hat *T* eine maximale Höhe von

$$h_{max} = n - 1$$

und eine minimale Höhe von

$$h_{min} \leq \lceil \log_2 n \rceil$$

Höhe

Beweis für maximale Höhe h_{max}

- Die maximale Höhe h_{max} kann in einem Baum nur dann erreicht werden, wenn der Baum zu einer linearen Liste der Länge n entartet.
- Die maximale Höhe ist dann gleich der Pfadlänge $n - 1$

$$h_{max} = n - 1$$

Höhe

Beweis für minimale Höhe h_{min}

- Ein Baum hat die minimale Höhe, wenn er seine maximale Kapazität ausschöpft.

$$n = 2^{h+1} - 1$$
$$n + 1 = 2^{h+1}$$
$$\log_2(n+1) = h + 1$$
$$\log_2(n+1) - 1 = h$$

- Da die Höhe ganzzahlig ist, gilt:

$$h_{min} = \lceil \log_2 n + 1 \rceil - 1 \leq \lceil \log_2 n \rceil$$

Höhe

Kleine Übung

Vollziehen Sie die Ungleichung durch Tabellieren nach

n	$\lceil \log_2 n + 1 \rceil$	$\lceil \log_2 n \rceil$

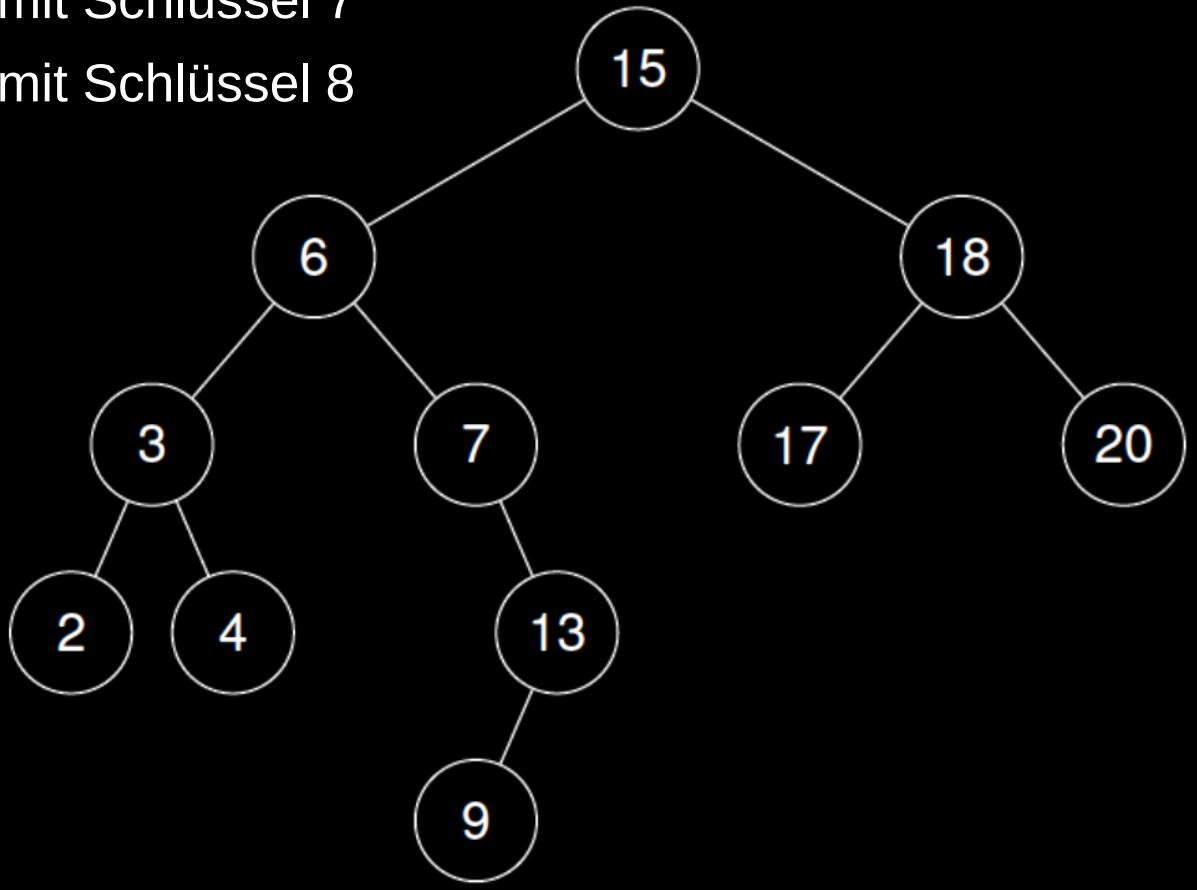
Operationen

- Suchen (rekursiv und nichtrekursiv)
- Minimum, Maximum
- Nachfolger, Vorgänger

Suchen

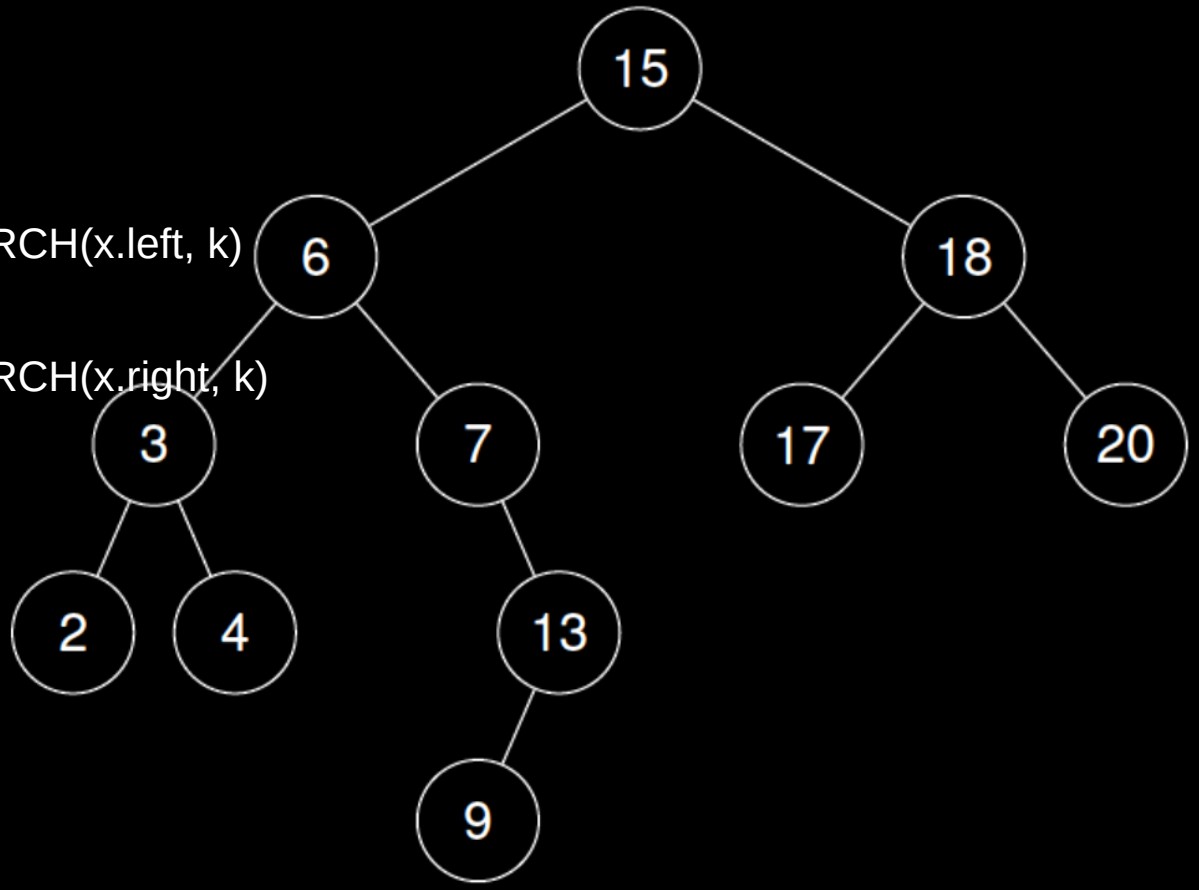
Überlegungen:

- Suche nach Knoten mit Schlüssel 7
- Suche nach Knoten mit Schlüssel 8



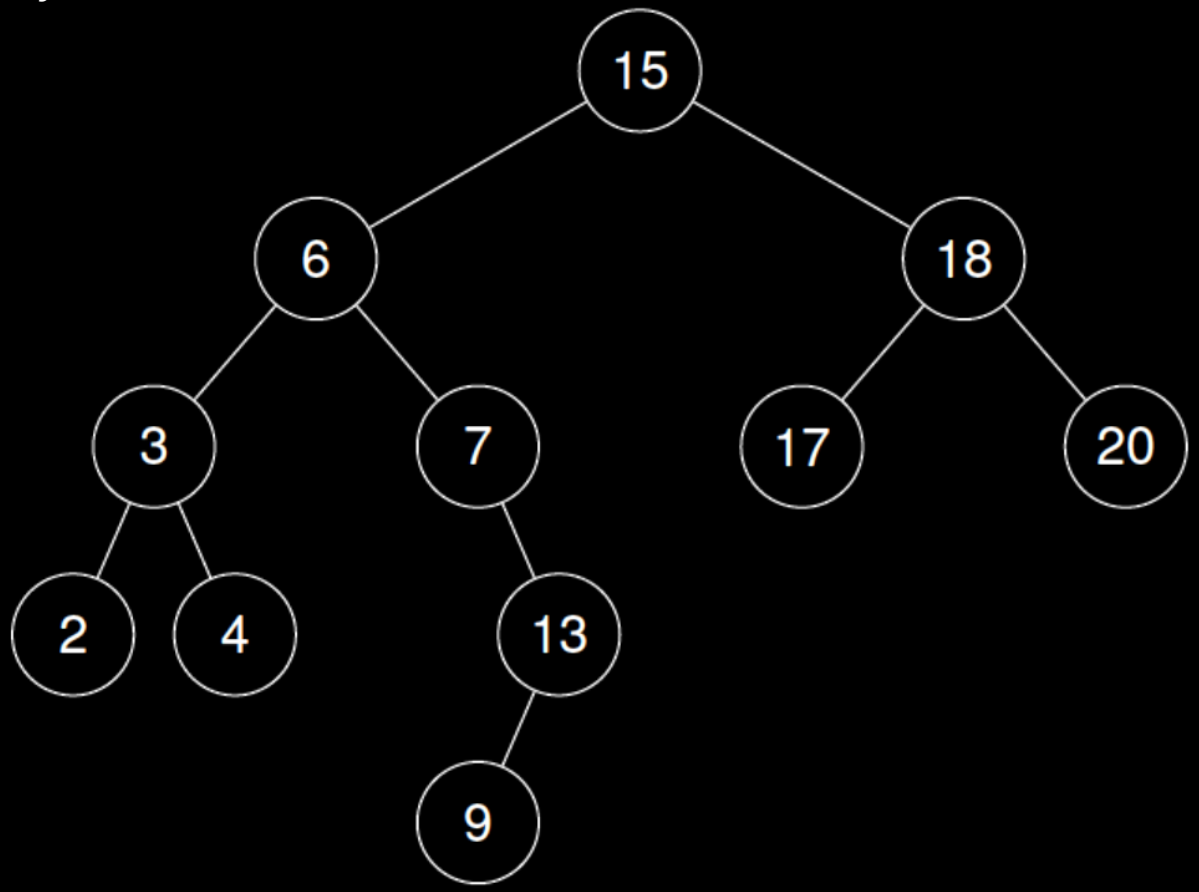
Rekursiver Algorithmus

```
1.  function TREE-SEARCH(x, k)
2.      if x = NIL  v k = x .key then
3.          return x
4.      end if
5.      if k < x .key then
6.          return TREE-SEARCH(x.left, k)
7.      else
8.          return TREE-SEARCH(x.right, k)
9.      end if
10. end function
```



Nichtrekursiver Algorithmus

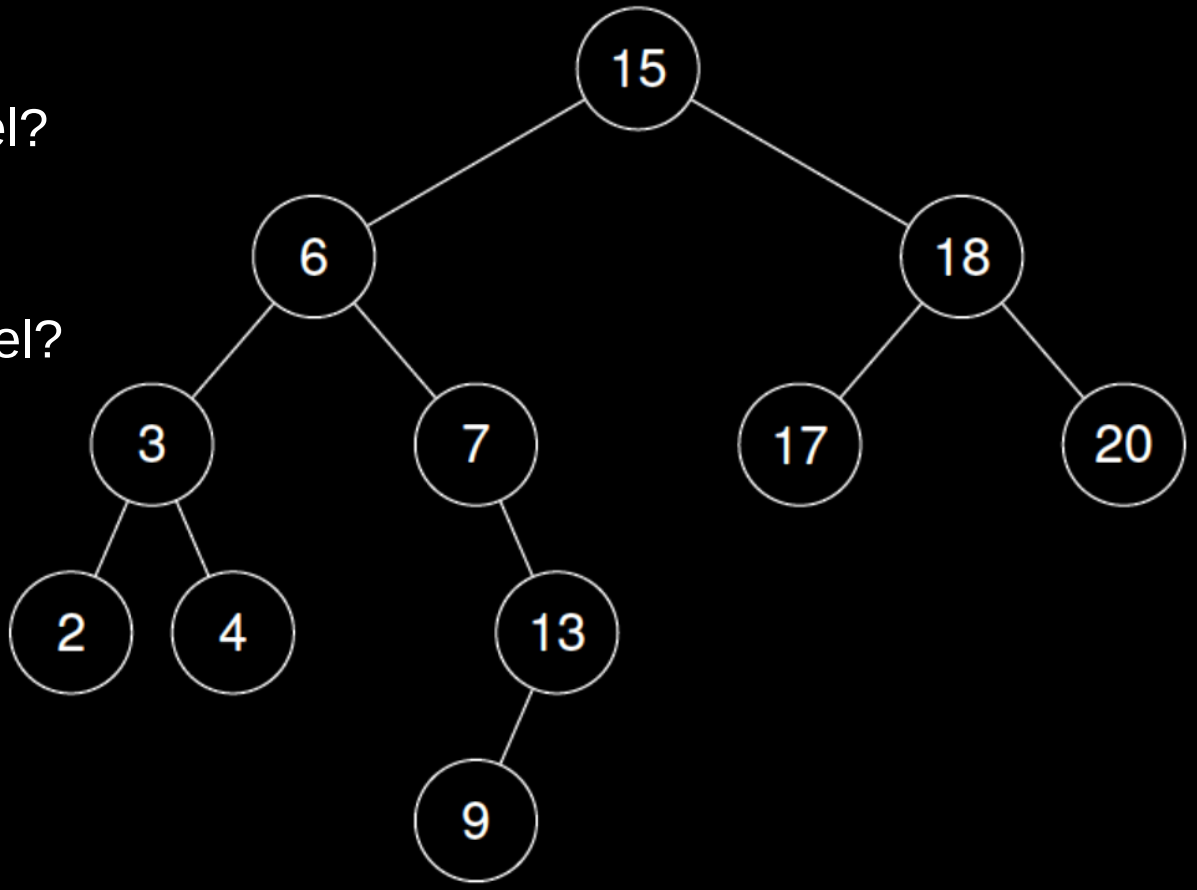
```
1.  function TREE-SEARCH(x, k)
2.      while x ≠ NIL ∧ k ≠ x.key do
3.          if k < x.key then
4.              x = x.left
5.          else
6.              x = x.right
7.          end if
8.      end while
9.      return x
10. end function
```



Suchen nach Minimum und Maximum

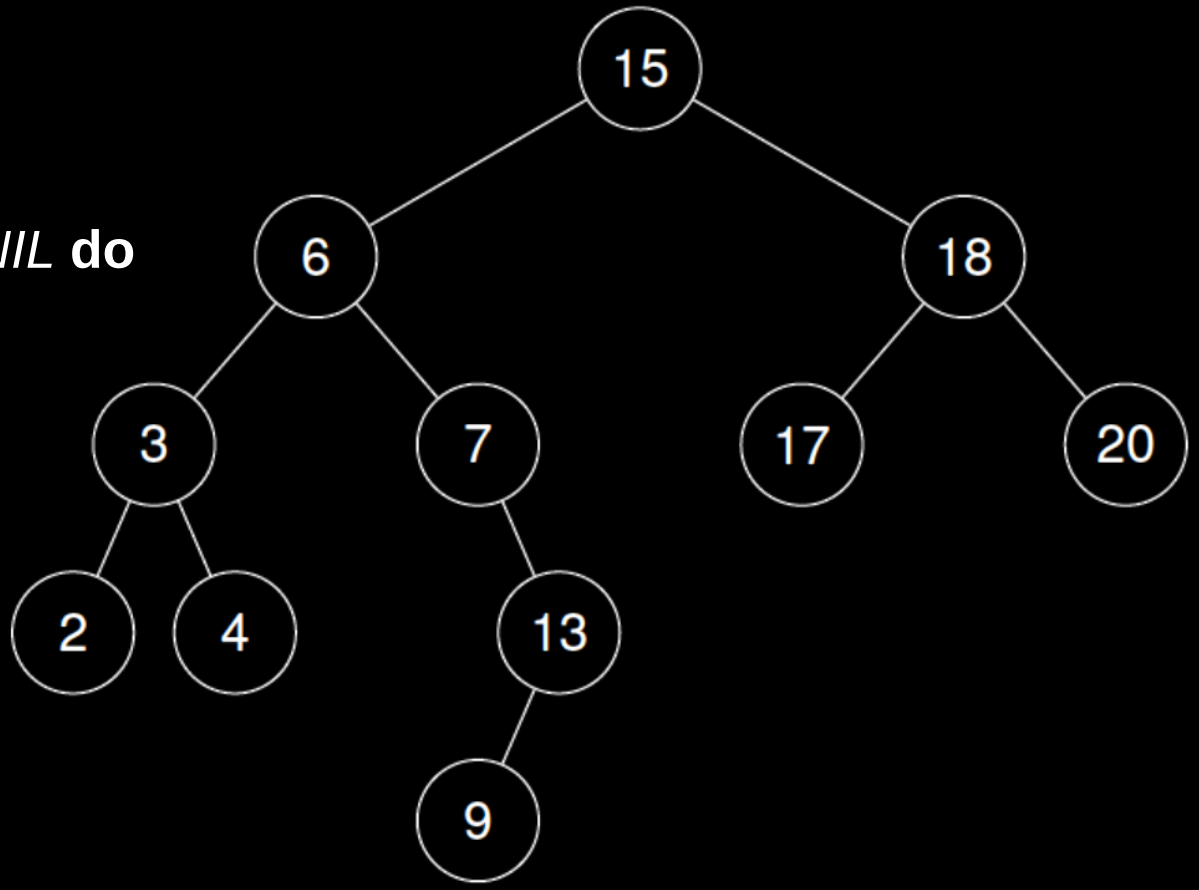
Überlegungen:

- Wie (wo) finde ich das Element mit minimalem Schlüssel?
- Wie (wo) finde ich das Element mit maximalem Schlüssel?



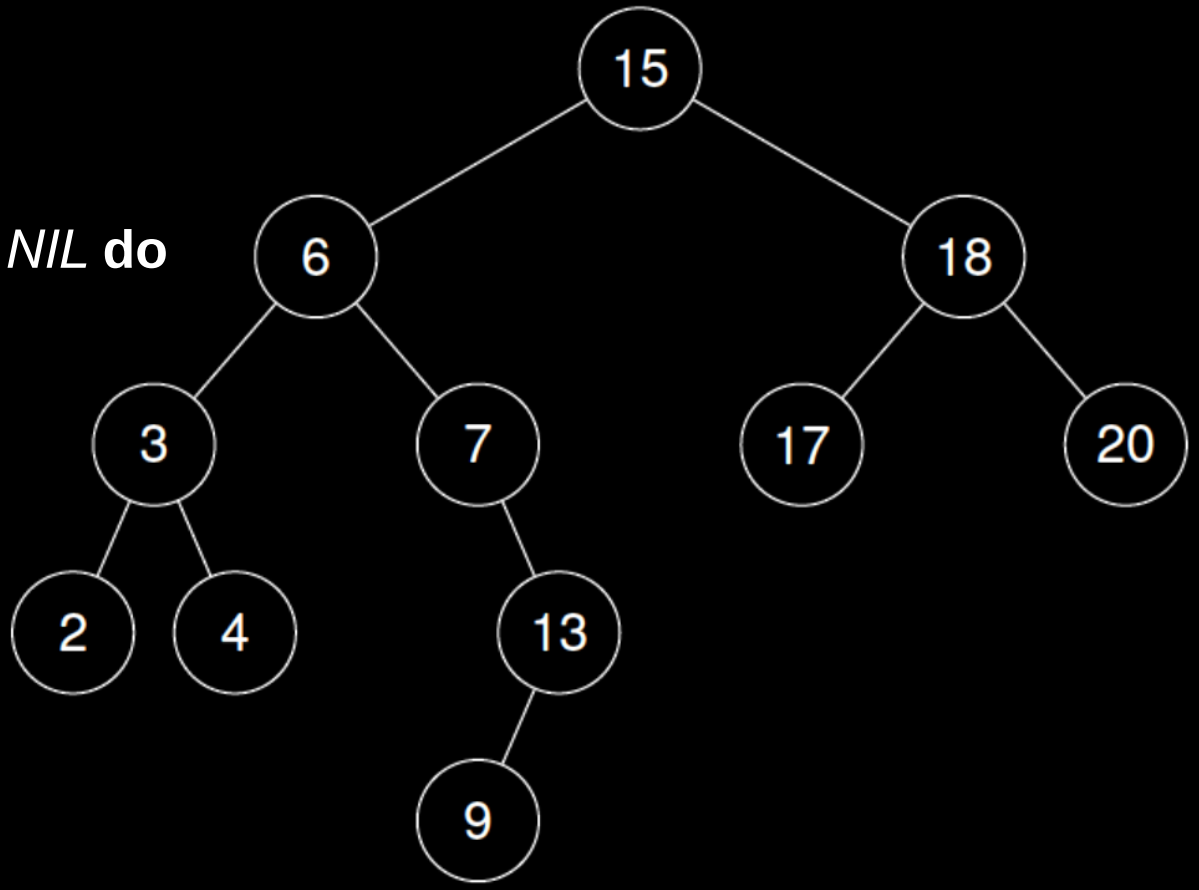
Suche nach Minimum

```
1.  function TREE-MINIMUM(x)
2.      if x = NIL then
3.          return NIL
4.      end if
5.      while x .left ≠ NIL do
6.          x = x .left
7.      end while
8.      return x
9.  end function
```



Suche nach Maximum

```
1.  function TREE-MAXIMUM(x)
2.      if x = NIL then
3.          return NIL
4.      end if
5.      while x .right ≠ NIL do
6.          x = x .right
7.      end while
8.      return x
9.  end function
```



Nachfolger und Vorgänger

Nachfolger (successor)

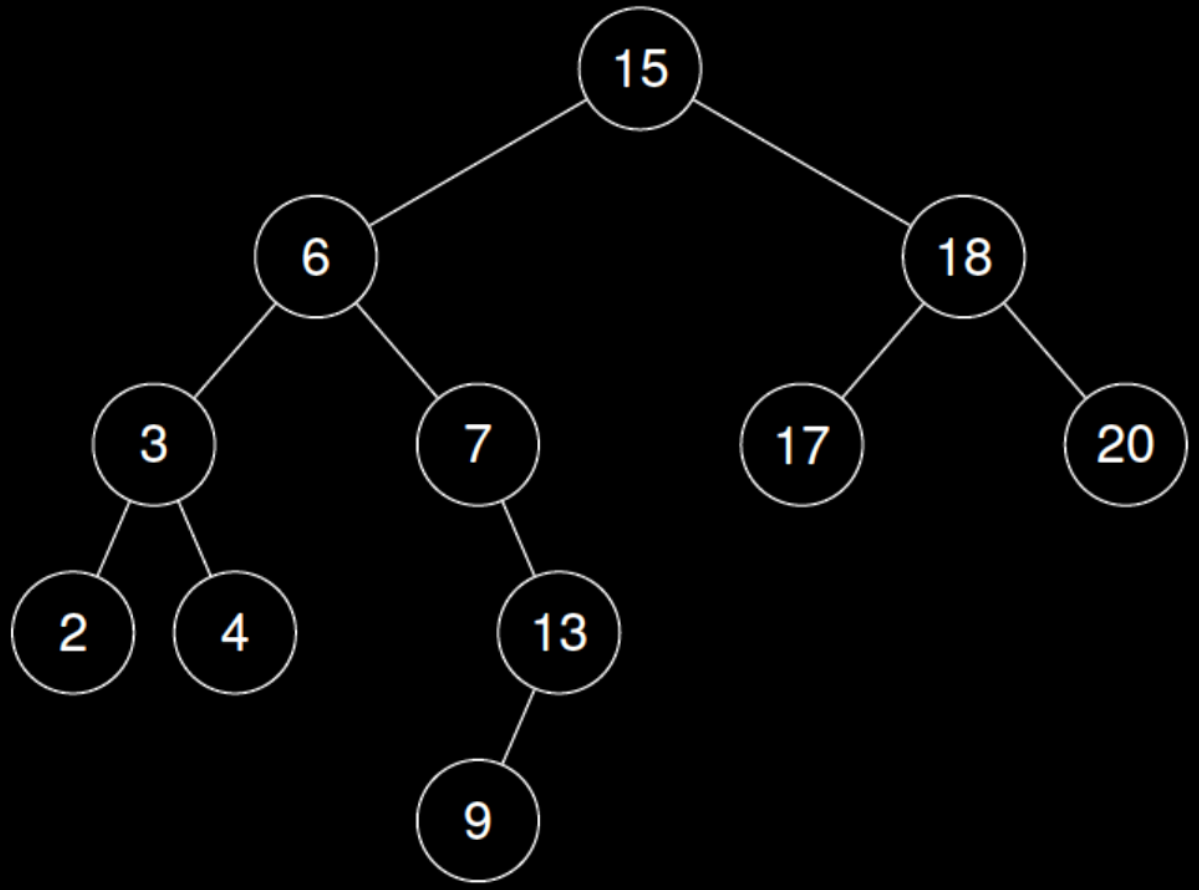
- Der Nachfolger eines Knotens *x* ist der Knoten *y* mit minimalem Schlüssel $k > x.key$. In anderen Worten: Der Nachfolger eines Knotens ist der Knoten mit dem nächstgrößeren Schlüssel im Baum.

Vorgänger (predecessor)

- Der Vorgänger eines Knotens *x* ist der Knoten *y* mit maximalem Schlüssel $k < x.key$. In anderen Worten: Der Vorgänger eines Knotens ist der Knoten mit dem nächstkleineren Schlüssel im Baum.

Beispiel: Nachfolger

x	Tree-Successor(x)
3	4
6	7
15	17
13	15
17	18
4	6



Nachfolger

Beobachtungen

- *x* hat rechten Teilbaum:
  Nachfolger von *x* ist das Minimum im rechten Teilbaum, das ist der am weitesten links stehende Knoten im rechten Teilbaum.
- *x* hat keinen rechten Teilbaum:
  Falls *x* einen Nachfolger *y* hat, so ist *y* der tiefste Knoten, dessen linkes Kind im Baum vor *x* liegt.
  Anders formuliert: Man geht im Baum so weit nach oben, bis man auf einen Knoten trifft, der linkes Kind seines Vaters ist. Dabei muss man *x* selbst mit berücksichtigen.

Nachfolger

Beobachtungen

- Interessant: Man braucht keine Schlüssel-Vergleiche auszuführen, die gesamte Information steckt in der Struktur des Baumes.
- Wie viele Kinder hat der Nachfolger von x?
  - 0/1, falls *x* 2 Kinder hat. Dann ist Minimum *y* im rechten Teilbaum Nachfolger. *y* hat entweder kein Kind oder nur einen rechten Teilbaum.
  - 1/2 sonst, da x im Baum unter dem Nachfolger y liegt.

- Wir nehmen an, dass alle Schlüssel paarweise verschieden sind.

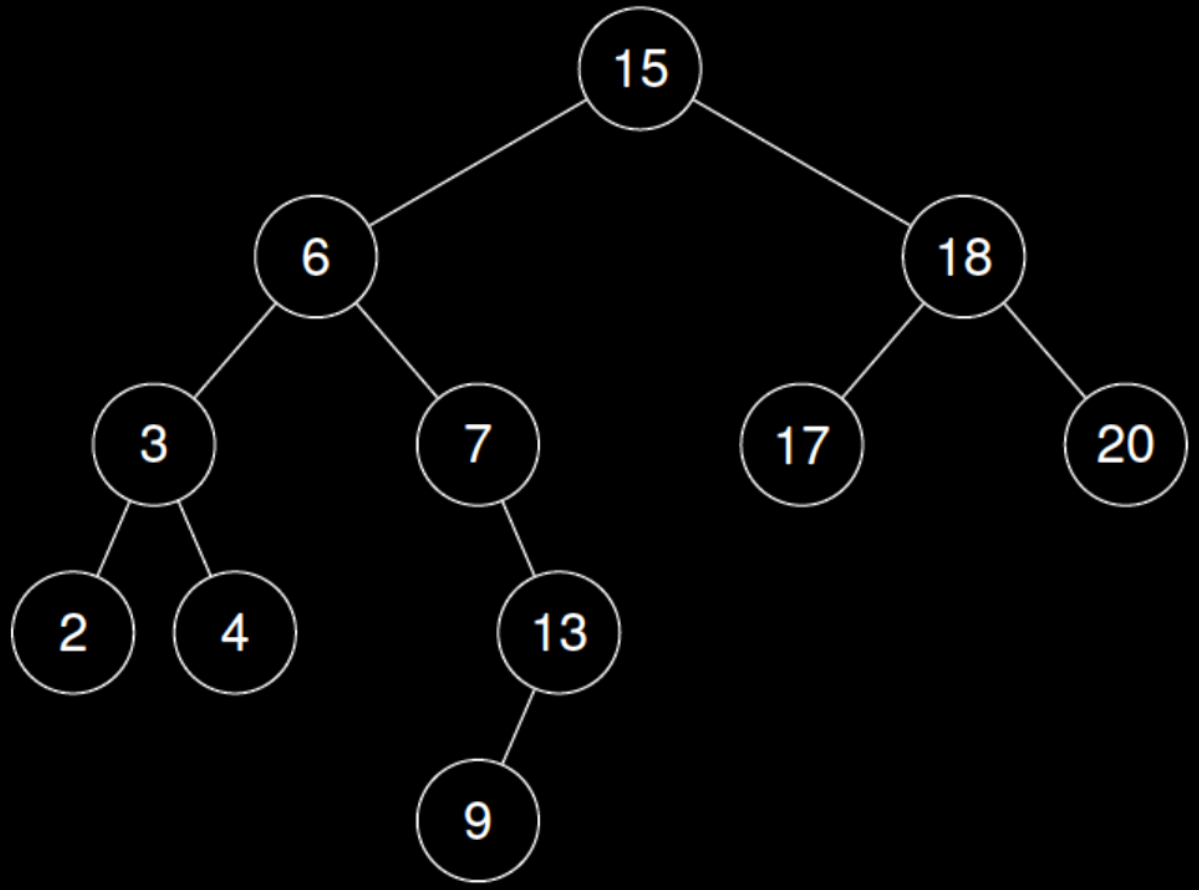
- Falls das nicht gilt, wird als Nachfolger von *x* der Knoten *y* definiert, den der Algorithmus zurückliefert.

Nachfolger

```
1.  function TREE-SUCCESSOR(x)
2.      if x = NIL then
3.          return NIL
4.      else if x .right ≠ NIL then
5.          return TREE-MINIMUM(x .right )
6.      end if
7.      y = x .parent
8.      while y ≠ NIL ∧ x = y .right do
9.          x = y
10.         y = y .parent
11.     end while
12.     return y
13. end function
```

Beispiel: Vorgänger

x	Tree-Predecessor(x)
3	2
6	4
15	13
13	9
17	15
4	3



Vorgänger

Beobachtungen

- Das Problem ist symmetrisch zum Nachfolger-Problem
  - *x* hat linken Teilbaum:
    Vorgänger von *x* ist das Maximum im linken Teilbaum, das ist der am weitesten rechts stehende Knoten im linken Teilbaum.
  - *x* hat keinen linken Teilbaum:
    Falls *x* einen Vorgänger *y* hat, so ist *y* der tiefste Knoten, dessen rechtes Kind im Baum vor *x* liegt. Anders formuliert: Man geht im Baum so weit nach oben, bis man auf einen Knoten trifft, der rechtes Kind seines Vaters ist. Dabei muss man *x* selbst mit berücksichtigen.

Vorgänger

Beobachtungen

- Man braucht auch hier keine Schlüssel-Vergleiche auszuführen. Die gesamte Information steckt in der Struktur des Baumes.

- Auch hier gilt bei nicht paarweise verschiedenen Schlüsseln, dass als Vorgänger von *x* der Knoten *y* definiert wird, den der Algorithmus zurückliefert.

Vorgänger

```
1.  function TREE-PREDECESSOR(x)
2.  if x = NIL then
3.      return NIL
4.  else if x .left ≠ NIL then
5.      return TREE-MAXIMUM(x .left )
6.  end if
7.  y = x .parent
8.  while y ≠ NIL ∧ x = y .left do
9.      x = y
10.     y = y .parent
11. end while
12. return y
13. end function
```

Laufzeit der Operationen

Alle bisher betrachteten Algorithmen verfolgen den Baum

- abwärts (Minimim, Maximum)
- aufwärts (Vorgänger, Nachfolger)

Die Laufzeit ist also von der Höhe abhängig. Es gilt:

$$T(n) = O(h)$$
$$= O(log(n))$$

Modifikationsoperationen im Baum

Überblick

- Einfügen und Löschen als modifizierende Operationen
- Im Unterschied zu vorherigen Datenstrukturen: Suchbaumeigenschaft muss erhalten bleiben
- Einfügen? Leicht, wenn einfach „unten“ eingefügt wird
- Löschen? Ist wohl etwas schwieriger

Modifikation: Einfügen

Funktionsweise

- Der Prozedur TREE-INSERT() wird die Wurzel *T* eines BST sowie ein Knoten *z* übergeben.
- Die Felder *z.left* und *z.right* müssen mit *NIL* initialisiert sein.
- *z.parent* wird vom Algorithmus geändert.
- Der Wert des Schlüssels (das Feld *z.key*) dient zum Einordnen in den BST

Modifikation: Einfügen

Variablen

- *z* Knoten, der eingefügt wird
- *x* Laufvariable zum Suchen der Einfügeposition
- *y* Knoten, an den eingefügt wird. Vater von *z* .

Modifikation: Einfügen

```
1.  function TREE-INSERT(T, z)
2.      y = NIL
3.      x = T .root
4.      while x ≠ NIL do
5.          y = x
6.          if z .key < x .key then
7.              x = x .left
8.          else
9.              x = x .right
10.         end if
11.     end while
            ...
```

Erläuterung:

Z. 4-11 Suchen der Einfügeposition. Man steigt so lange ab, bis man mit *y* einen Blattknoten erreicht hat.

Ergebnis: *y* ist Vater des neuen Knotens oder *NIL*. Dann ist der neue Knoten Wurzel des Baums.

Modifikation: EInfügen

```
12.     z .parent = y
13.     if y = NIL then
14.         T.root = z
15.     else if z .key < y .key then
16.         y.left = z
17.     else
18.         y.right = z
19.     end if
20. end function
```

Erläuterung:

Z. 12 *y* wird als Vater von *z* gesetzt.

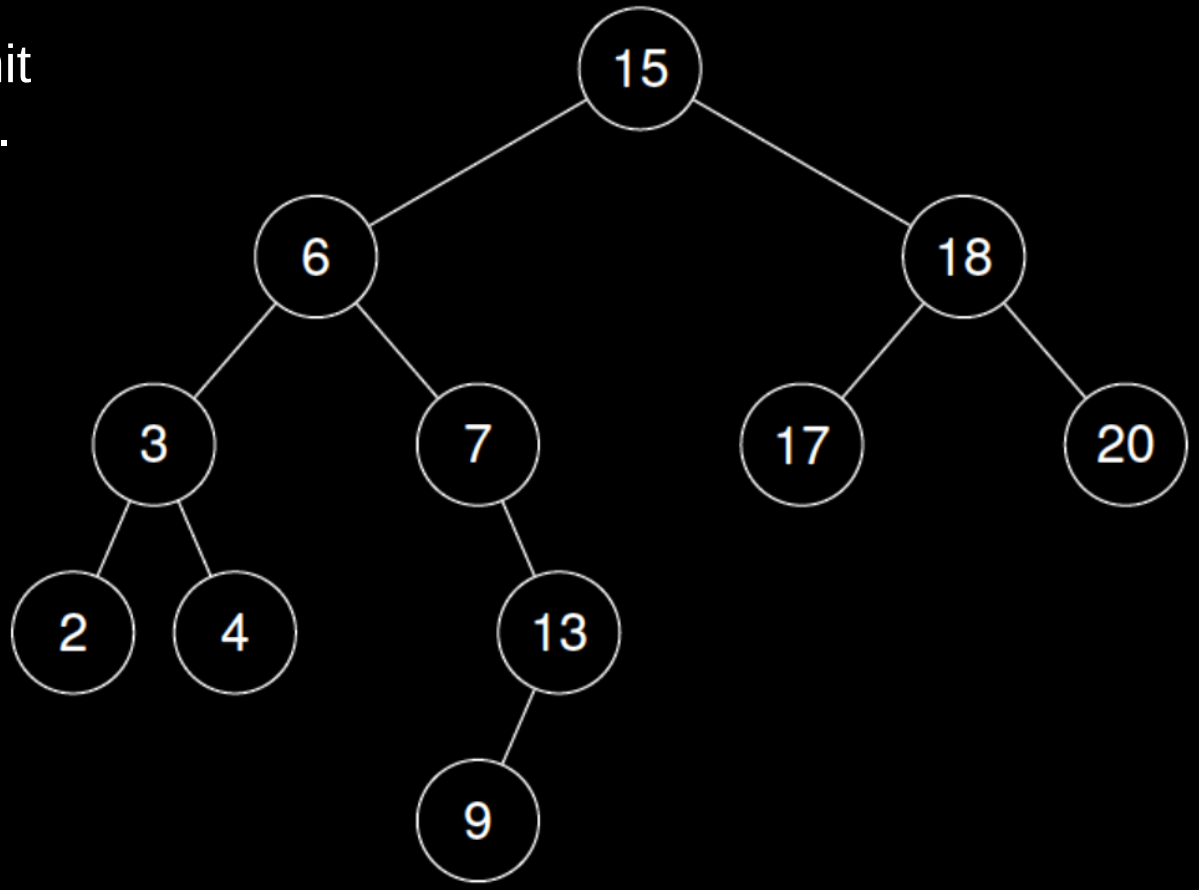
Z. 13-14 Falls *y* == *NIL* ist der neue Knoten Wurzel.

Z. 16-20 *z* wird an *y* als linkes oder rechtes Kind angehängt

Modifikation: Einfügen

Kleine Übung

- Fügen Sie in diesen Baum den Knoten mit dem Schlüssel 8 ein. Protokollieren Sie alle Variablen im Ablauf des Algorithmus.



Modifikation: Einfügen

- Bemerkungen

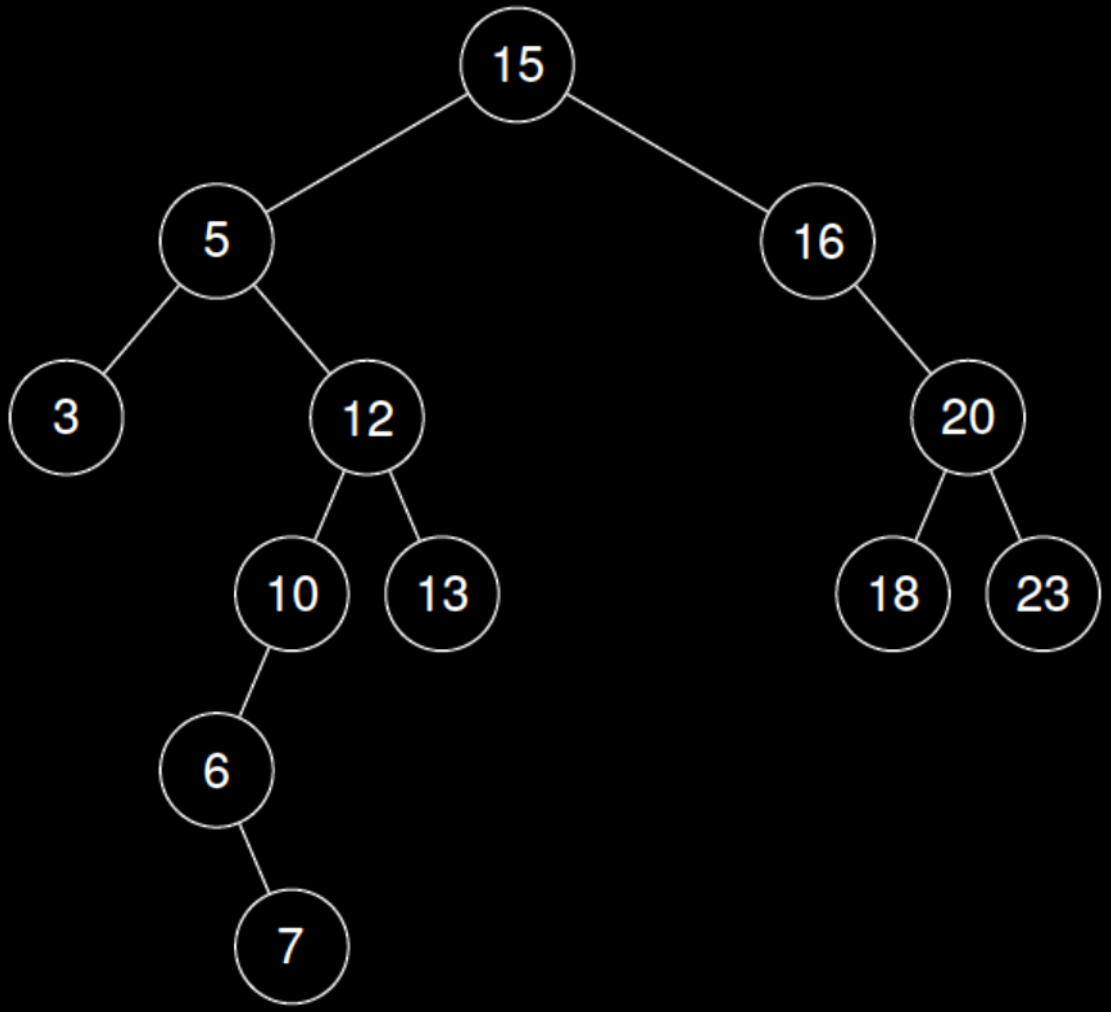
- Struktur des Baumes wird durch die Einfügereihenfolge bestimmt
- Dabei entstehende Bäume heißen *natürlich*.
- Es gibt $n!$ viele Möglichkeiten (Reihenfolgen), Knoten in Bäume einzufügen. Es gibt aber nicht $n!$ viele Bäume.
  - Günstig: Vollständiger Baum $\Rightarrow T(n) = O(\log n)$.
  - Ungünstig: Lineare Liste $\Rightarrow T(n) = O(n)$.
- Wichtige Fragen:
  - Was liegt dazwischen und wie häufig kommen welche Varianten vor?
  - Was ist der „mittlere“ Fall?
  - Ergebnis: Die erwartete Höhe eines natürlichen BST mit n Knoten ist $O(\log n)$.

Modifikation: Löschen

Beispiel 1

Löschen des Knotens z

mit z.key = 13

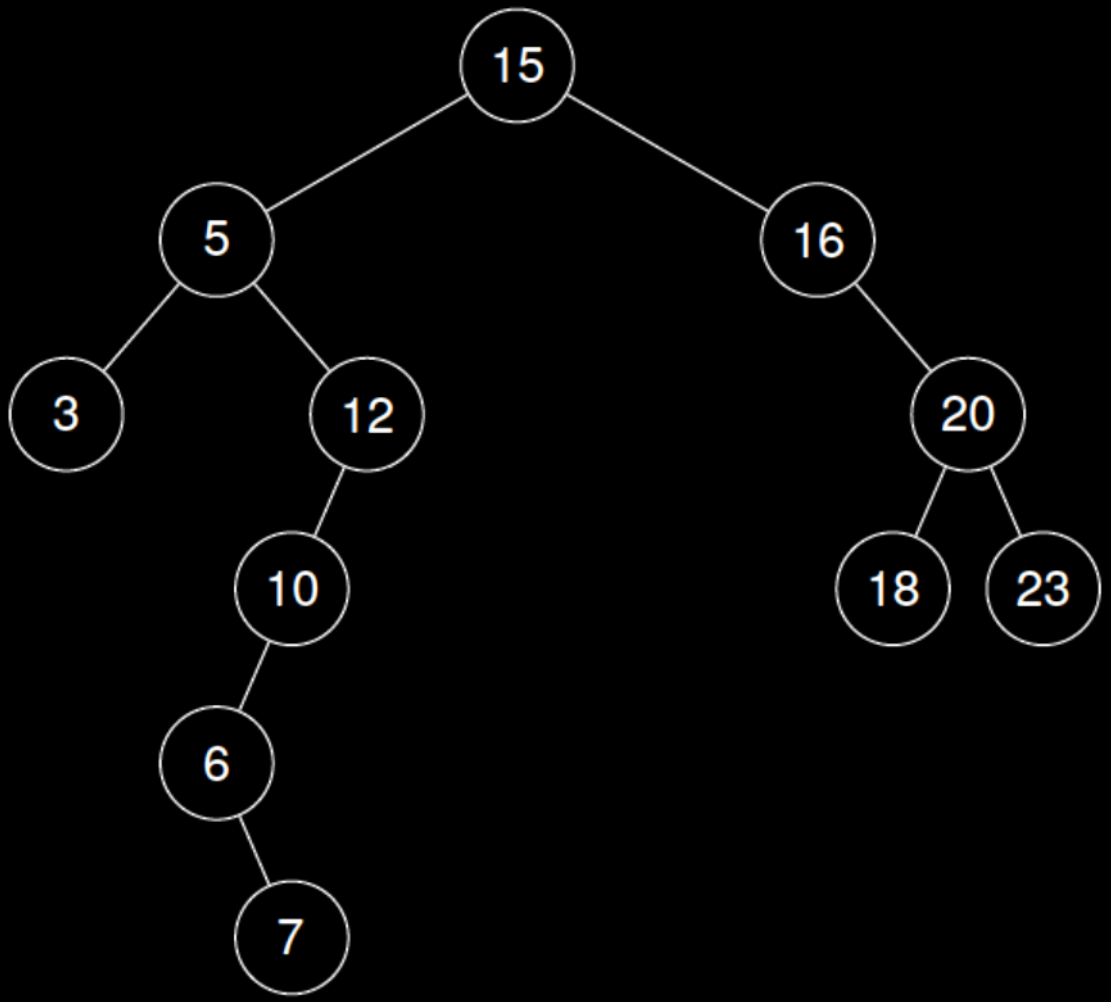


Modifikation: Löschen

Beispiel 1

Löschen des Knotens z

mit z.key = 13

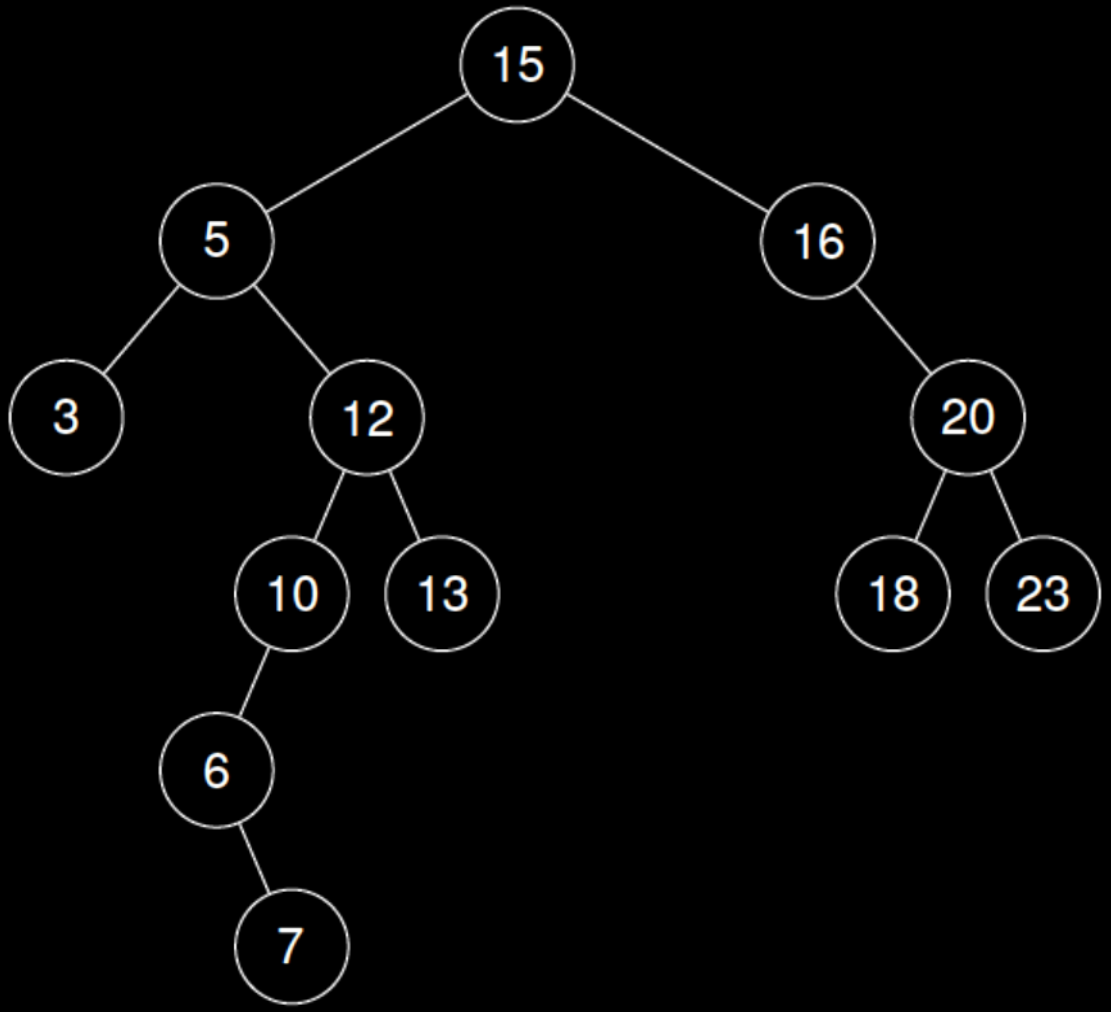


Modifikation: Löschen

Beispiel 2

Löschen eines Knotens z

mit z.key = 16

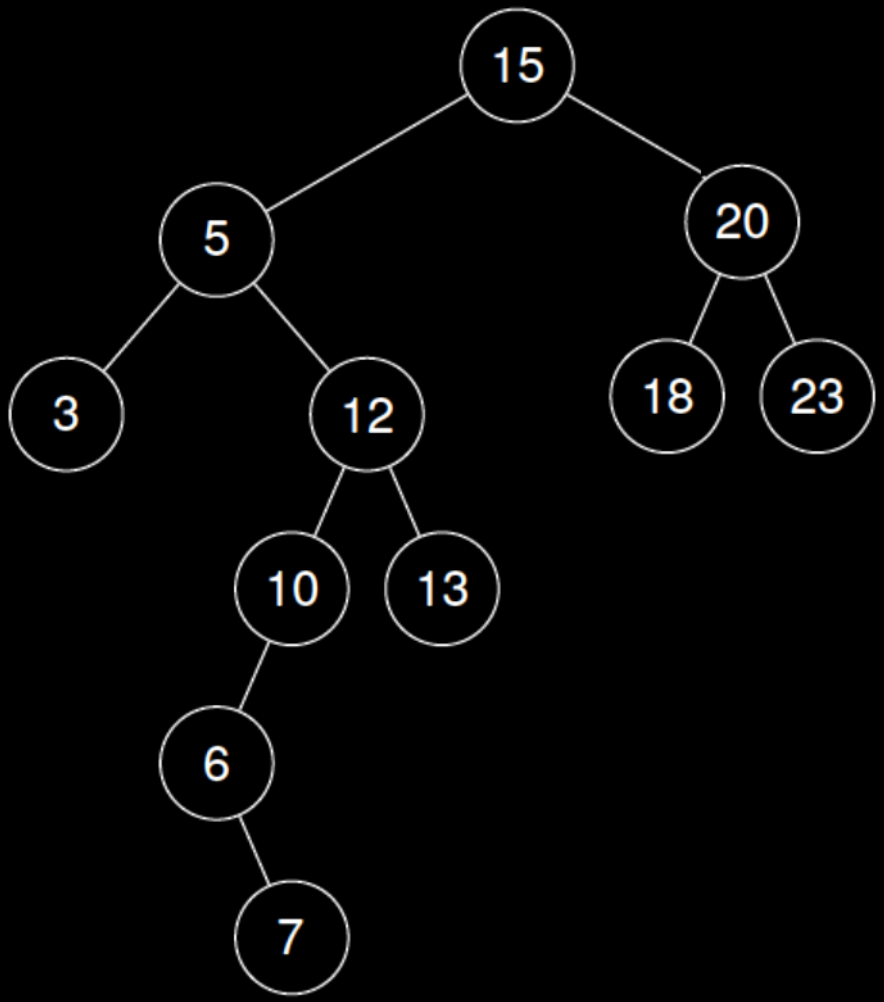


Modifikation: Löschen

Beispiel 2

Löschen eines Knotens z

mit z.key = 16

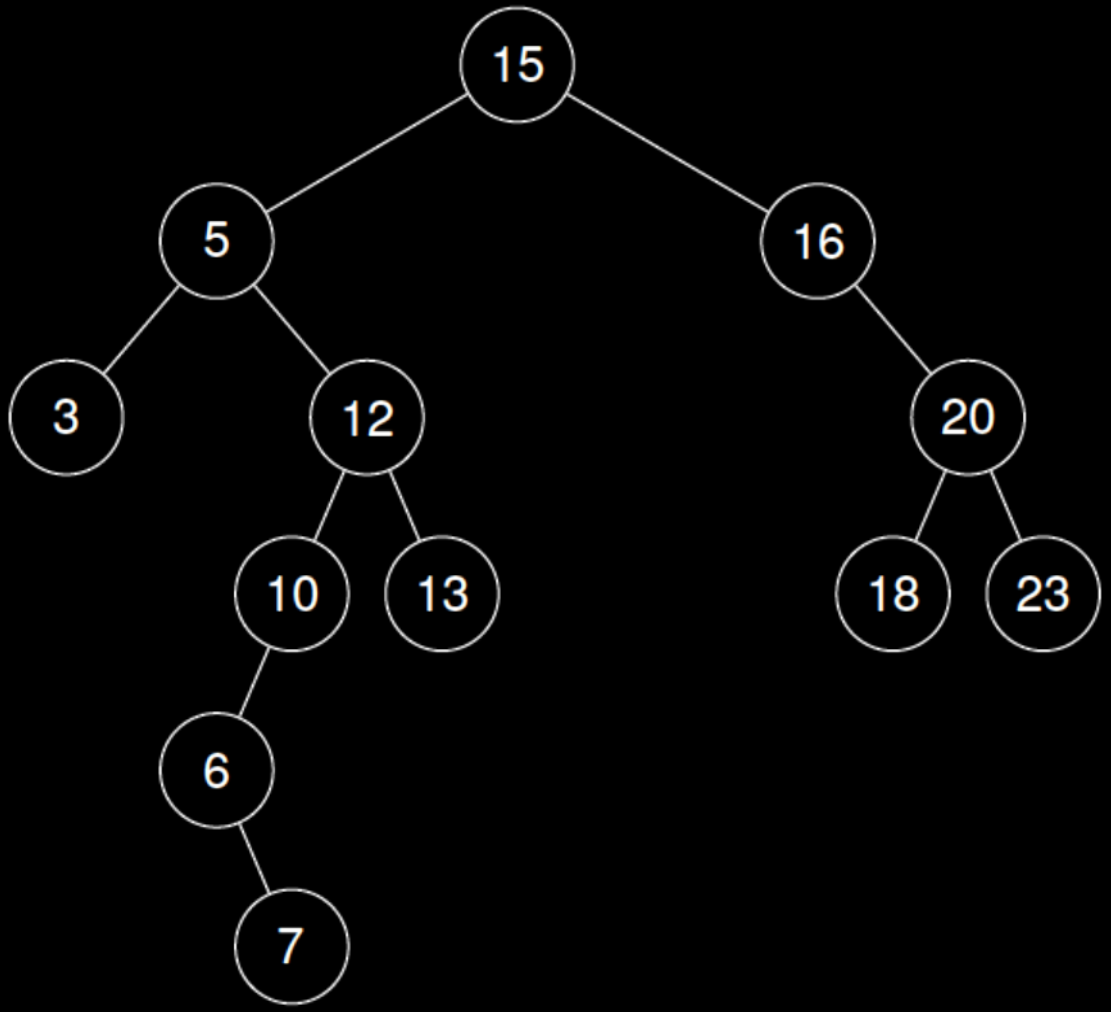


Modifikation: Löschen

Beispiel 3

Löschen des Knotens z

mit z.key = 5

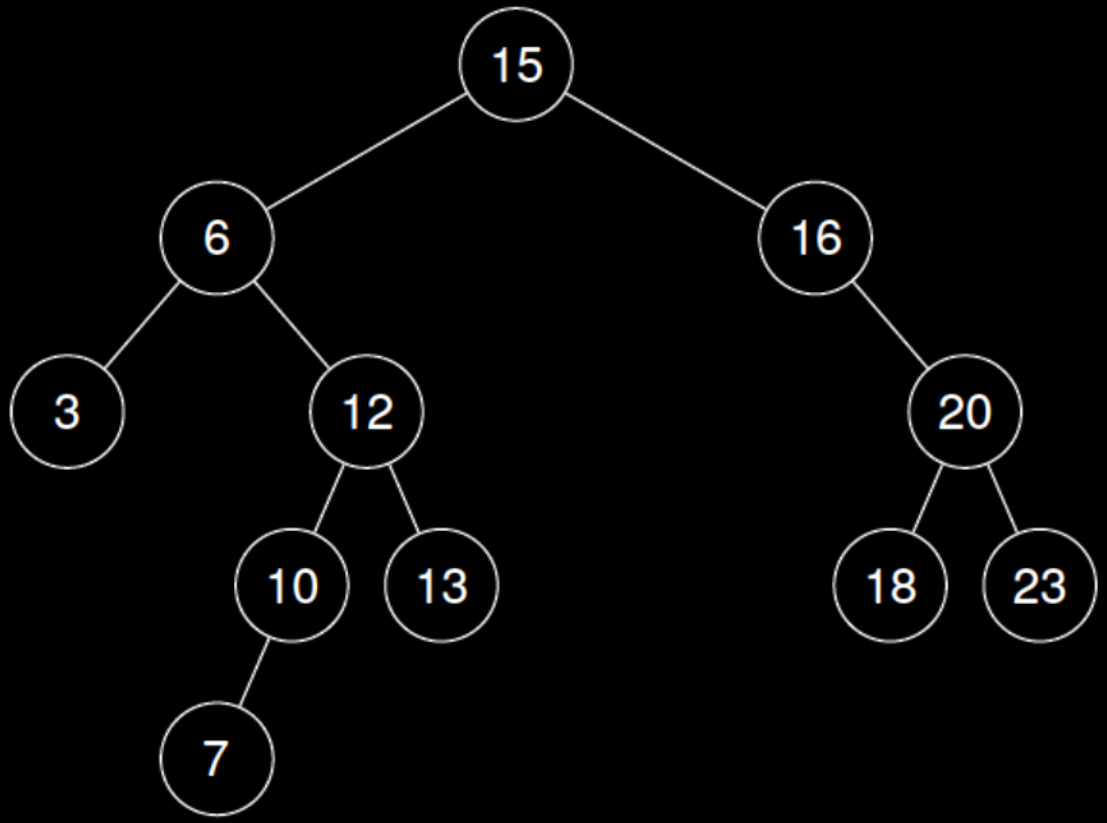


Modifikation: Löschen

Beispiel 3

Löschen des Knotens z

mit z.key = 5



Modifikation: Löschen

Beobachtung

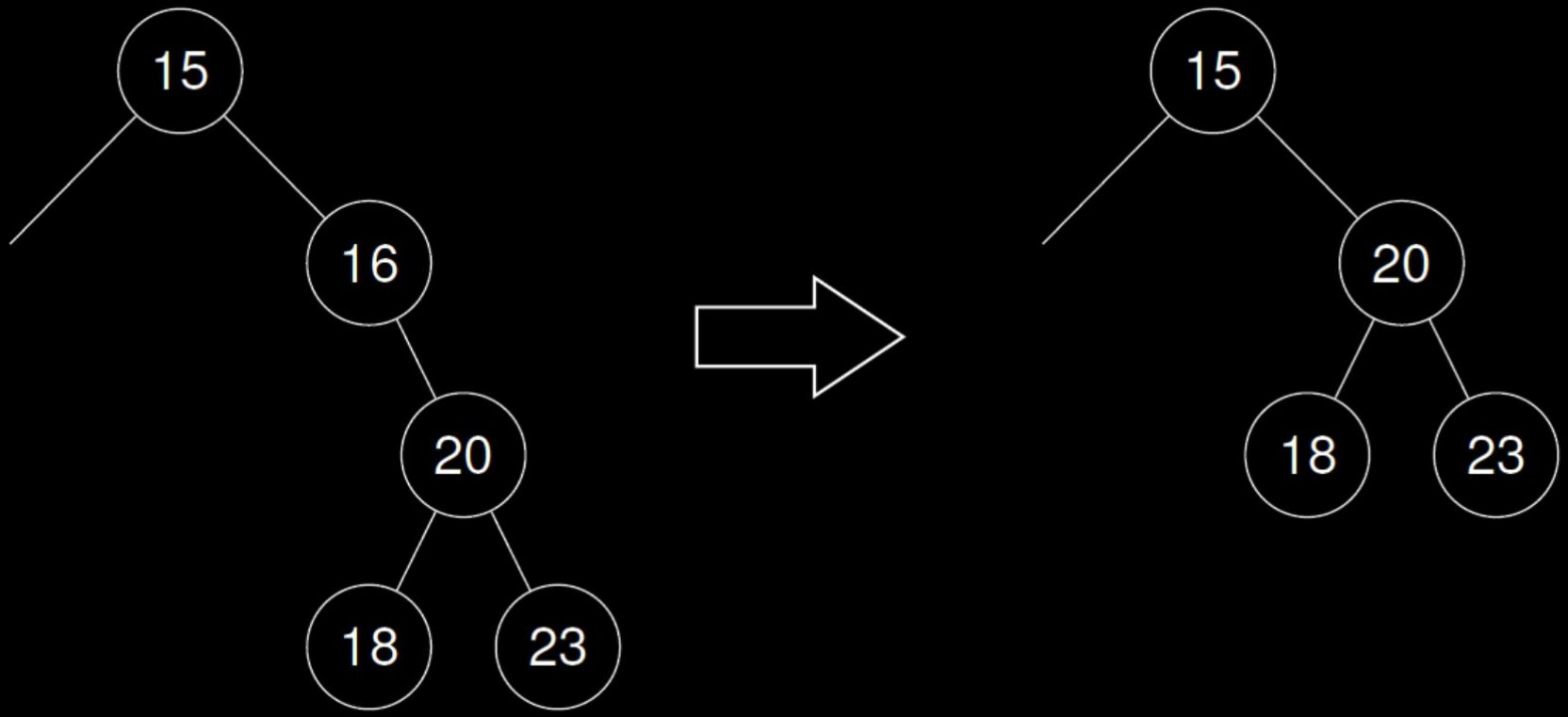
- Das Vorgehen beim Löschen hängt offenbar von der Anzahl der Kinder des zu löschenden Knotens ab.
- Ein Knoten ohne Kinder (Blattknoten) kann direkt entfernt werden.
- Ein Knoten mit einem Kind kann durch Herauslösen entfernt werden.
- Es gibt viele Arten, wie man Knoten mit zwei Kindern entfernen kann (z.B: Ersetzen durch Vorgänger oder Nachfolger)

Modifikation: Löschen

z ist Blattknoten

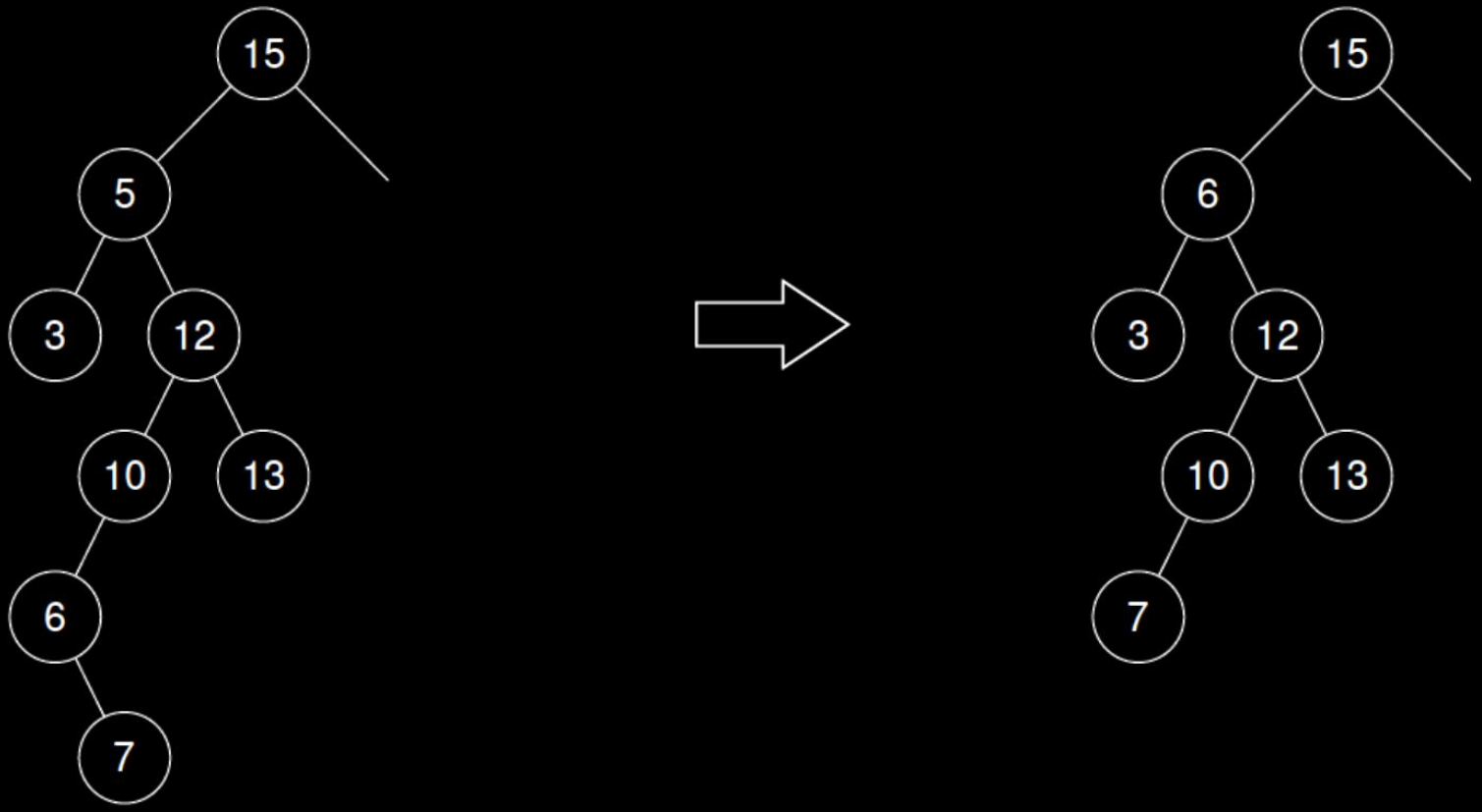
Modifikation: Löschen

z hat 1 Kind



Modifikation: Löschen

z hat 2 Kinder



Modifikation: Löschen

- Funktionsweise

- Hat der zu löschende Knoten *z* höchstens 1 Kind, so ist das Löschen einfach.
- Hat *z* kein linkes Kind, so wird *z* durch sein rechtes Kind ersetzt, auch wenn dies *NIL* ist. Ist das rechte Kind *NIL*, dann liegt der Fall vor, dass *z* keine Kinder hat. Ist das rechte Kind nicht *NIL*, dann liegt der Fall vor, dass *z* genau ein Kind hat.
- Hat *z* genau ein Kind (das muss jetzt das linke Kind sein), so wird *z* durch sein linkes Kind ersetzt.
- Ansonsten (*z* hat 2 Kinder) bestimmt man den Nachfolger *y* von *z* im rechten Teilbaum, der ja sicher kein linkes Kind hat. *y* wird herausgelöst und ersetzt *z*
  1. Ist *y* rechtes Kind von *z* , können wir *z* so ersetzen, da es keine Kinder hat
  2. Ansonsten liegt *y* mitten im rechten Teilbaum. *y* wird zunächst durch sein eigenes rechtes Kind *x* ersetzt. Anschließend wird *z* durch *y* ersetzt.

Modifikation: Löschen

Die Funktion TRANSPLANT erledigt das Ersetzen/Umhängen von Teilbäumen.

```
1.  function TRANSPLANT(T, u, v)
2.      if u.parent = NIL then
3.          T .root = v
4.      else if u = u.parent .left then
5.          u.parent .left = v
6.      else
7.          u.parent .right = v
8.      end if
9.      if v ≠ NIL then
10.         v.parent = u.parent
11.     end if
12. end function
```

Erläuterung:

Z. 2-3 Behandelt den Fall, dass *u* die Wurzel des Baums ist.

Z. 4-5 Hängt *v* als linkes Kind ein.

Z. 6-7 Hängt *v* als rechtes Kind ein.

Z. 9-10 Wenn *v* nicht *NIL* ist, wird *v.parent* aktualisiert.

Achtung: *v.left* und *v.right* werden durch die Funktion nicht akualisiert. Dies muss ggfls. die aufrufende Funktion erledigen.

Modifikation: Löschen

```
1.  function TREE-DELETE(T, z)
2.      if z.left = NIL then
3.          TRANSPLANT(T , z , z.right )
4.      else if z.right = NIL then
5.          TRANSPLANT(T , z , z.left )
6.      else
7.          y = TREE-MINIMUM(z.right )
8.          if y.parent != z then
9.              TRANSPLANT(T , y , y.right )
10.             y.right = z.right
11.             y.right.parent = y
12.         end if
13.     TRANSPLANT(T , z , y )
14.     y.left = z.left
15.     y.left .parent = y
16.   end if
17. end function
```

Erläuterung:

Z. 2-3 Behandelt den Fall, dass *z* kein linkes Kind hat.

Z. 4-5 Behandelt den Fall, dass *z* kein rechtes Kind (aber ein linkes Kind) hat.
Z. 7 Suche des Nachfolgers *y* von *z* im rechten Teilbaum.

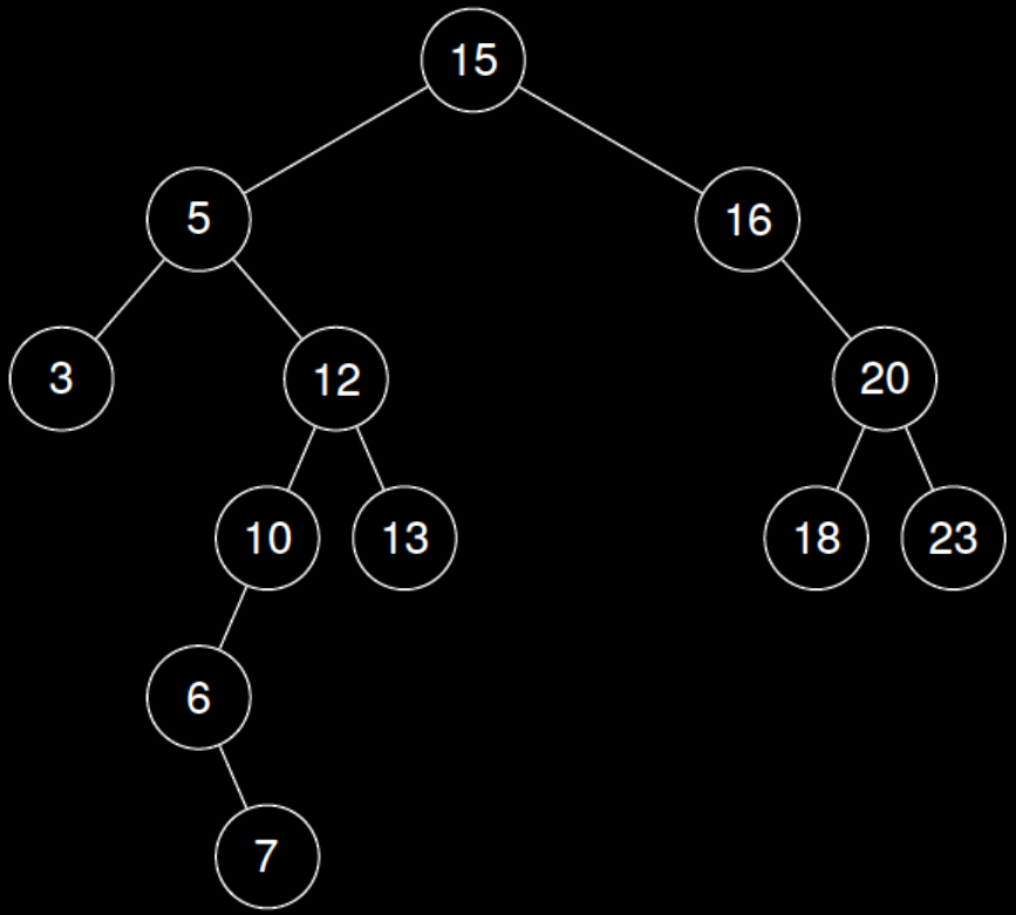
Z. 8-11 Wenn *y* nicht rechtes Kind von *z* ist, so wird *y* zunächst durch sein rechtes Kind ersetzt und anschließend *z*'s rechtes Kind an *y* gehängt. Danach:

Z. 13-15 *y* ersetzt *z* und *z*'s linkes Kind wird an *y* gehängt.

Modifikation: Löschen

Kleine Übung

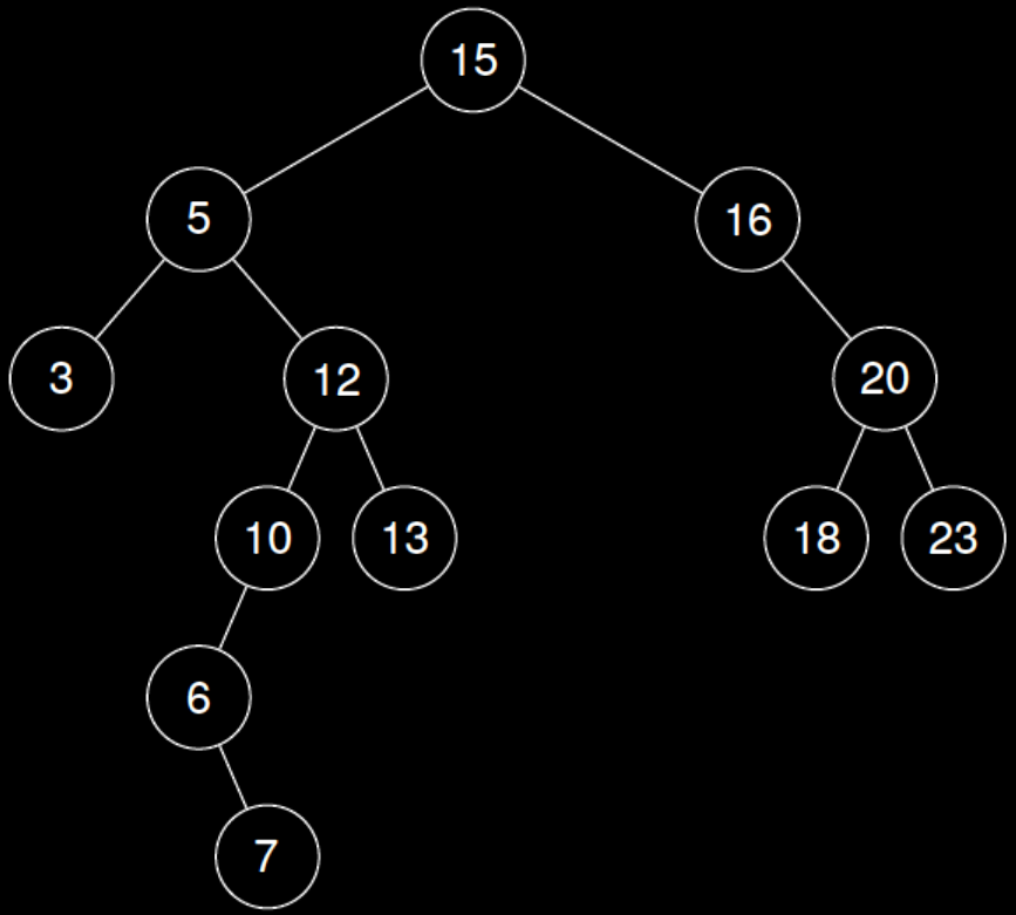
- Löschen Sie aus diesem Baum den Knoten mit Schlüssel 3.
- Protokollieren Sie alle Variablen im Ablauf des Algorithmus.



Modifikation: Löschen

Kleine Übung

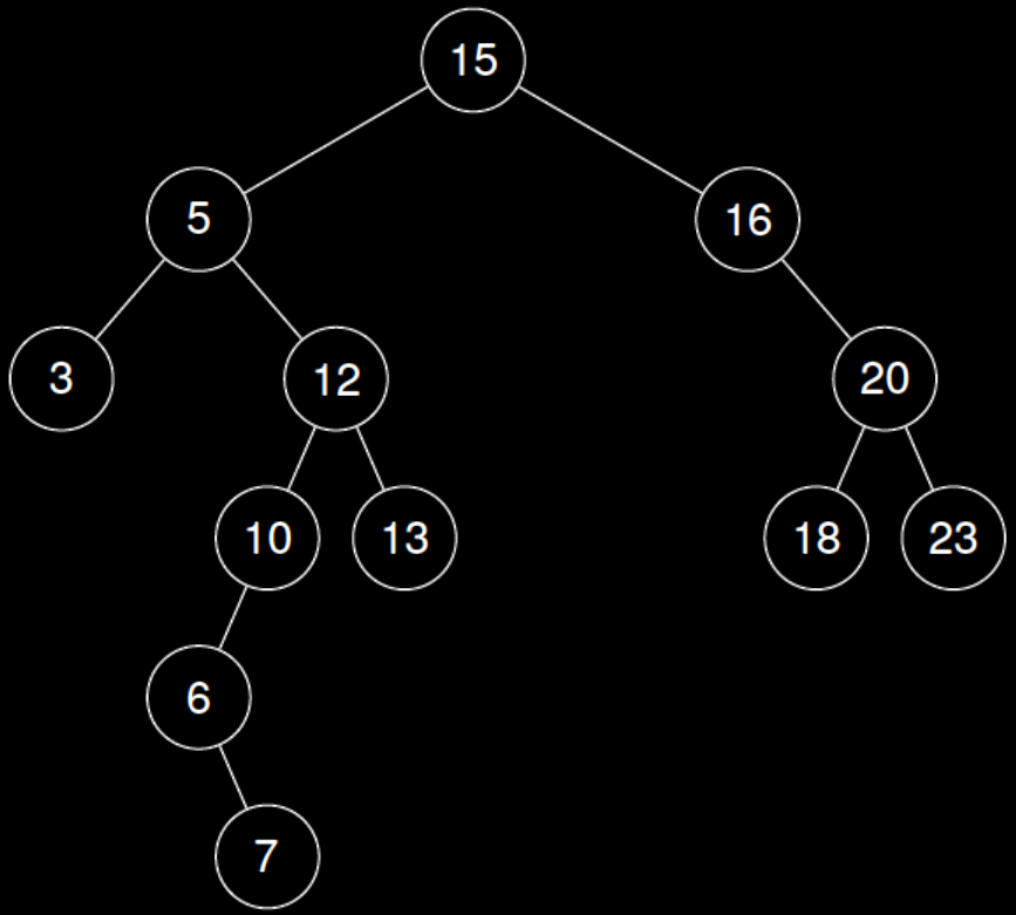
- Löschen Sie aus diesem Baum den Knoten mit Schlüssel 16.
- Protokollieren Sie alle Variablen im Ablauf des Algorithmus.



Modifikation: Löschen

Kleine Übung

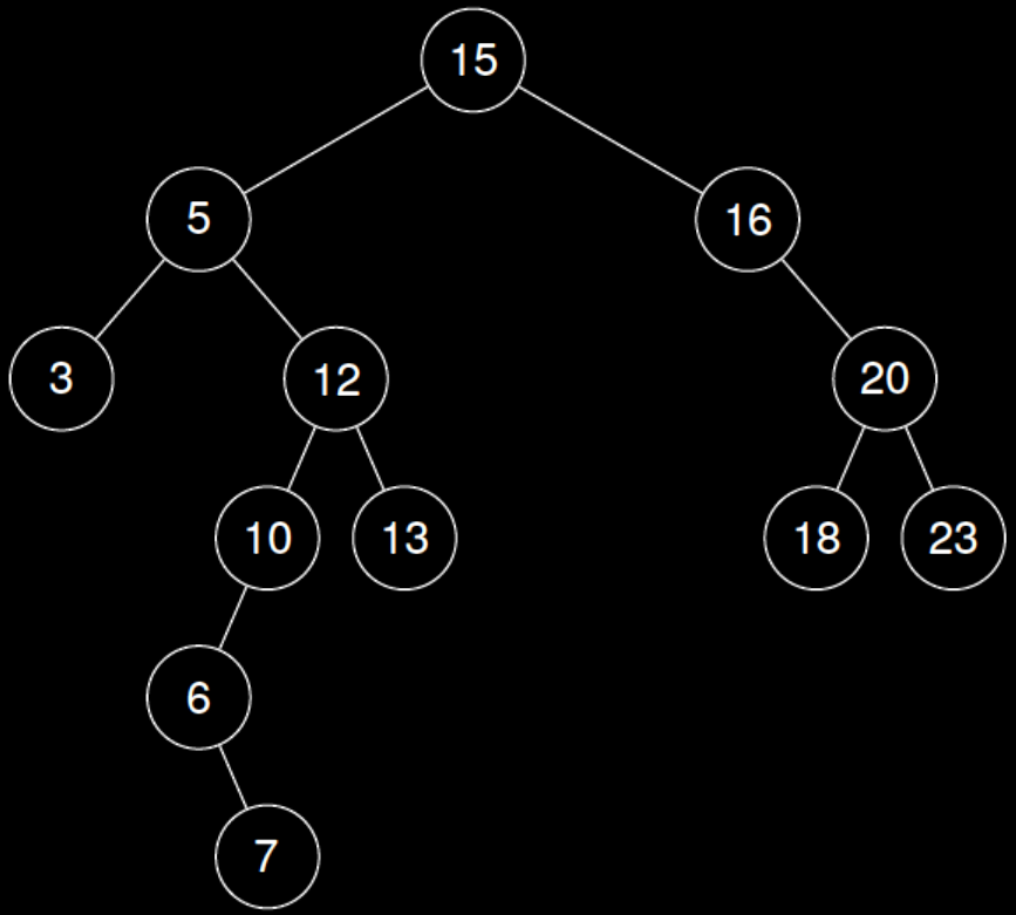
- Löschen Sie aus diesem Baum den Knoten mit Schlüssel 5.
- Protokollieren Sie alle Variablen im Ablauf des Algorithmus.



Modifikation: Löschen

Kleine Übung

- Löschen Sie aus diesem Baum den Knoten mit Schlüssel 15.
- Protokollieren Sie alle Variablen im Ablauf des Algorithmus.



Binäre Suchbäume

Zusammenfassung

- Alle Wörterbuchoperationen konnten effizient realisiert werden.
  - Ungünstigste Laufzeit: $T(n) = O(n)$
  - Günstigste Laufzeit: $T(n) = O(h) = O(\log n)$

- Wir wissen wenig über die Struktur (v.a. Höhe) eines Baums im allgemeinen Fall.
- Man kann zeigen, dass die erwartete Höhe eines BST im mittleren Fall $O(\log n)$ ist.
- Dennoch: Hinzufügen und Löschen kann im Verlauf der Lebenszeit eines Baums zu ungünstigen, nicht balancierten Bäumen führen.

- Im nächsten Kapitel wird eine Variante der BST vorgestellt, die zu „einigermaßen“ balancierten Bäumen führt.