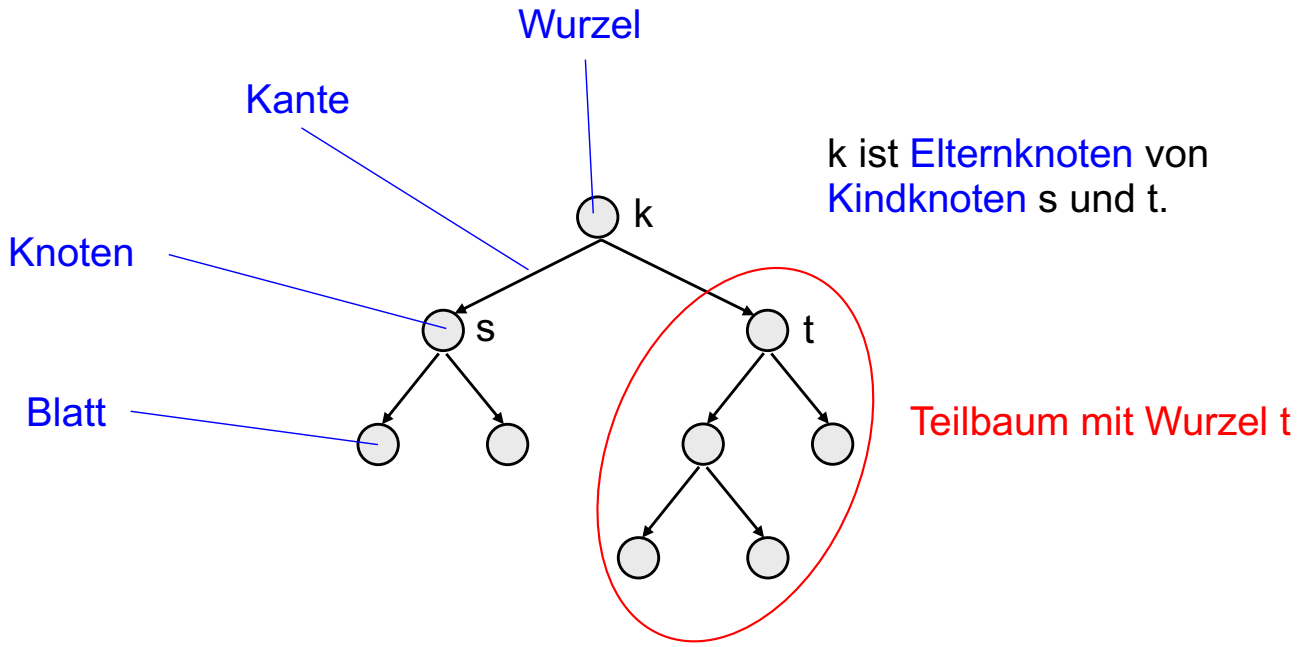
3. Binäre Suchbäume

- Begriffe und Eigenschaften (Wiederholung PROG 2)
- Binäre Suchbäume (Wiederholung PROG 2)
- Iterative Traversierung mit Elternzeigern

Begriffe und Eigenschaften

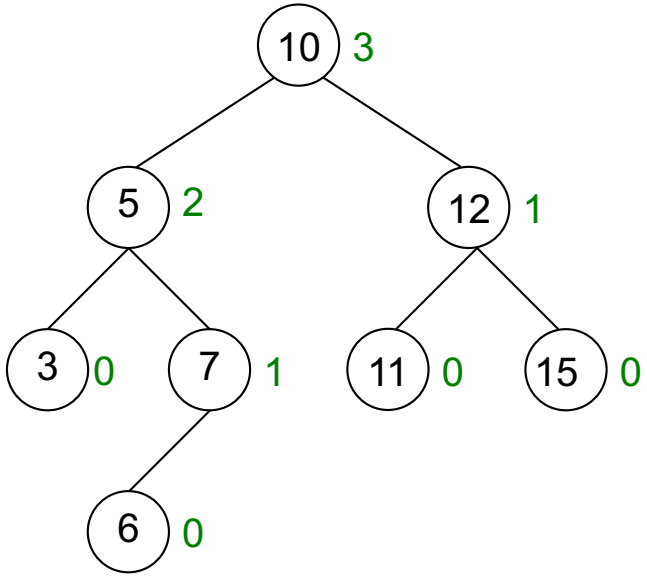


- Ein Baum besteht aus einer Menge von Knoten und einer Menge von gerichteten Kanten (von Elternknoten zu Kindknoten)
- Zyklen sind nicht erlaubt.
- Es gibt genau einen Knoten, der keine eingehende Kante hat: die sogenannte Wurzel.
- Alle anderen Knoten haben genau eine eingehende Kante.
- Knoten ohne ausgehende Kante heißen Blätter.
- Kanten sind meistens nach unten gerichtet. Kantenrichtung wird dann oft weggelassen.

Höhe eines Baums

Höhe = Maximale Anzahl von Kanten von seiner Wurzel zu einem Blatt.

Beispiel



Höhe für jeden Teilbaum

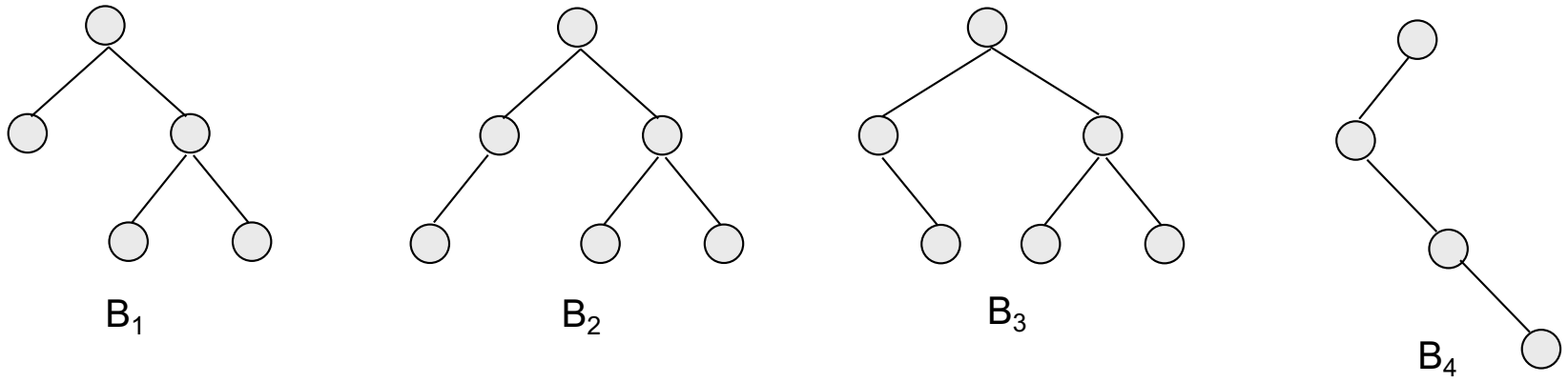
Beachte

- Ein Baum, der nur aus einem Knoten besteht, besitzt die Höhe 0.
- Aus technischen Gründen wird die Höhe eines leeren Baums (d.h. Anzahl Knoten = 0) als -1 definiert.

Binärbäume

- Ein Binärbaum ist ein geordneter Baum (Reihenfolge der Kinder ist wichtig), bei dem jeder Knoten maximal 2 Kinder hat.
- Die beiden Kinder werden linkes und rechtes Kind bzw. linker und rechter Teilbaum.
- Hat ein Knoten nur ein Kind, dann muss es entweder ein linkes oder ein rechtes Kind sein.

Beispiele

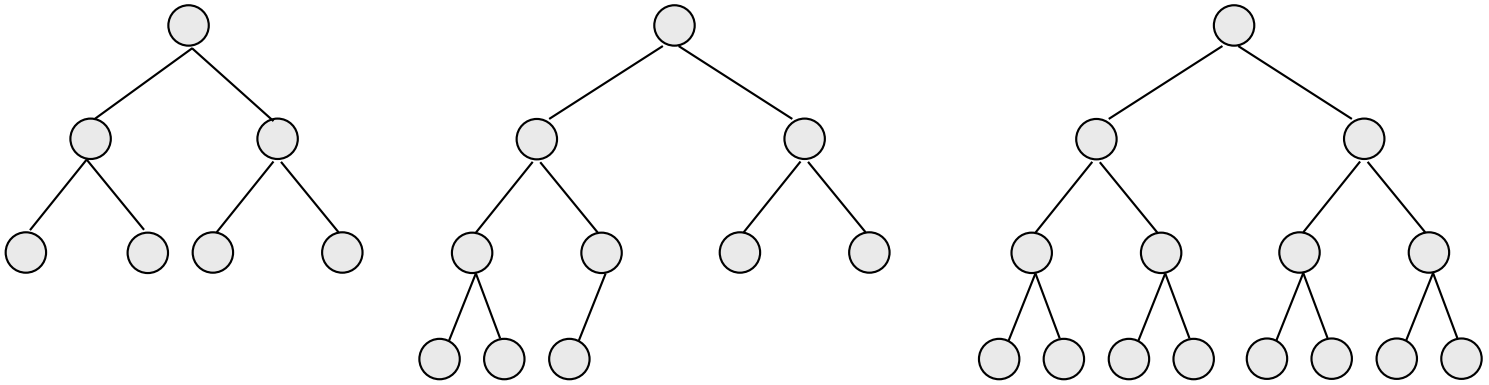


Bäume B_2 und B_3 sind strukturell unterschiedlich

Vollständiger Binärbaum

Ein vollständiger Binärbaum ist ein Binärbaum, bei der jede Ebene (bis auf die letzte) vollständig gefüllt und die letzte Ebene von links nach rechts gefüllt ist.

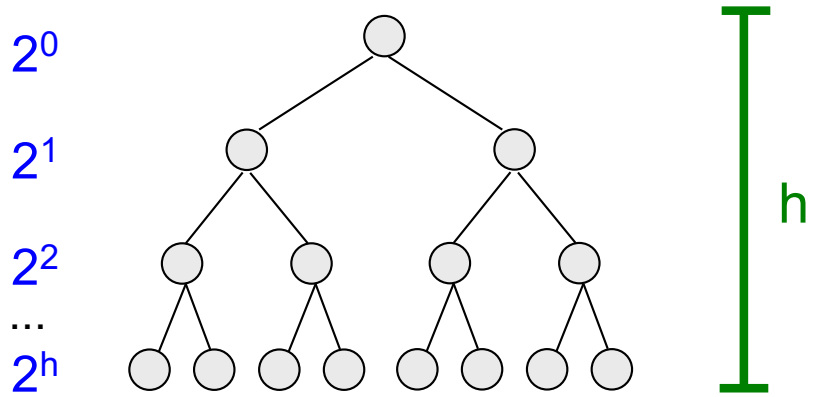
Beispiele



Eigenschaften

- Ein vollständiger Binärbaum mit n Knoten hat die Höhe $h = \lfloor \log_2 n \rfloor$.
- Vollständige Binärbäume sind (bei einer gegebenen Knotenzahl) Binärbäume mit einer minimalen Höhe.

Herleitung der Höhe eines vollständigen Binärbaums



Vollständiger Binärbaum mit n Knoten und der Höhe h mit vollständig gefüllter letzter Ebene.

$$2^0 + 2^1 + 2^2 + \ldots + 2^h = n$$

$$\Rightarrow \quad 2^{h+1} - 1 = n$$

$$\Rightarrow \quad h = \log_2(n+1) - 1$$

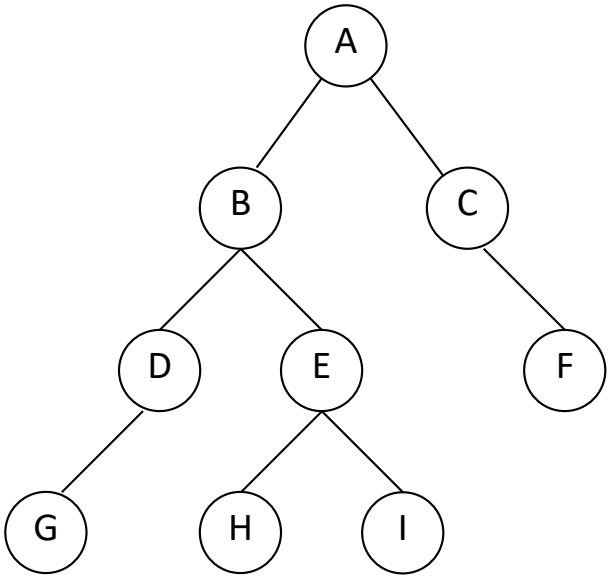
$$\Rightarrow \quad h = \lfloor \log_2 n \rfloor .$$

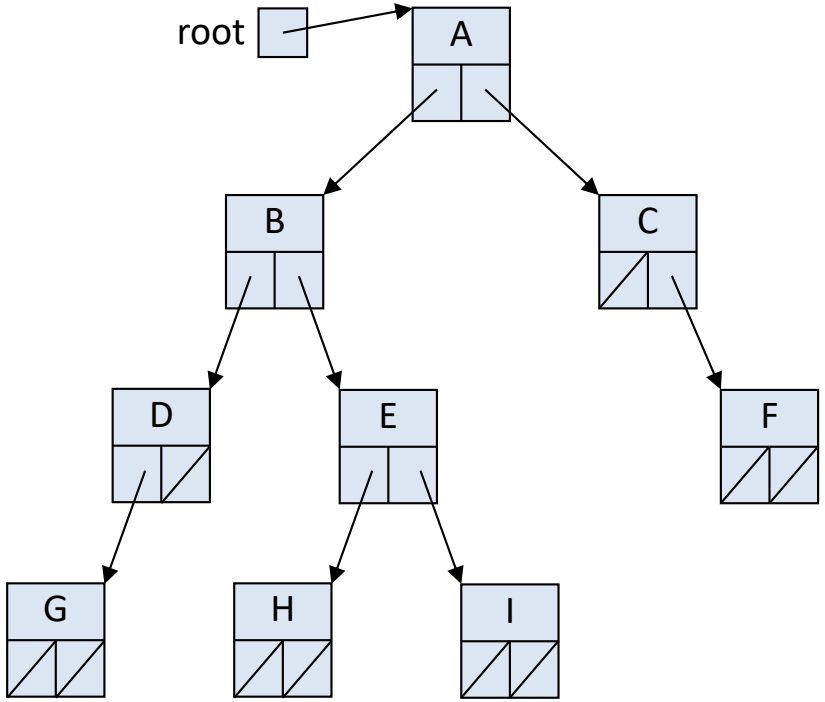
(Bei nicht vollständig gefüllter letzter Ebene Herleitung ähnlich.)

Implementierung von Binärbäumen

- Implementierung als verkettete Struktur: Jeder Knoten hat jeweils eine Referenz für das linke und das rechte Kind.

```
class Node<K,V> {
        K key;
        V value;
        Node<K,V>  left;        // linkes Kind
        Node<K,V>  right;       // rechtes Kind
}
```



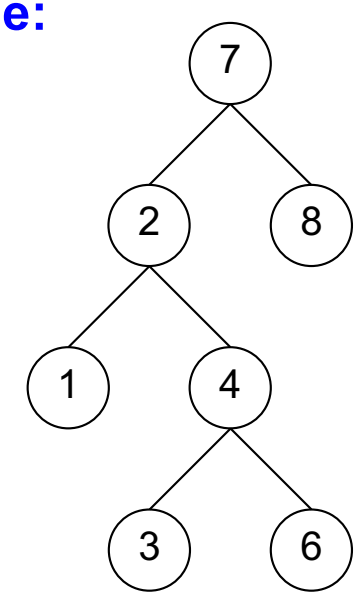


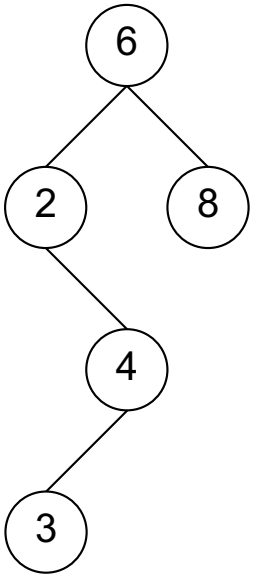
Einfachheitshalber sind im Baum nur die Schlüssel dargestellt.

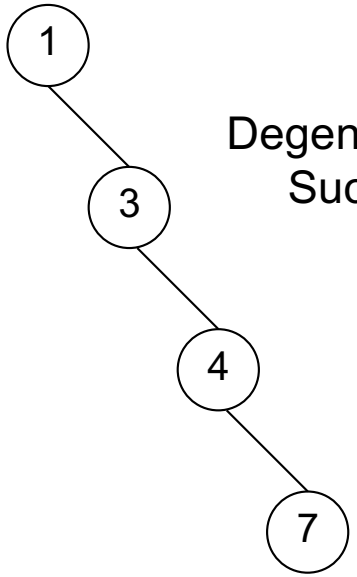
Binäre Suchbäume

- Ein binärer Suchbaum ist ein Binärbaum, bei dem für alle Knoten k folgende Eigenschaften gelten:
  - Alle Schlüssel im linken Teilbaum sind kleiner als k
  - Alle Schlüssel im rechten Teilbaum sind größer als k
- Beachte, dass die Zahlen hier eindeutig sein müssen. Es ist aber auch möglich, dass gleiche Zahlen mehrfach vorkommen dürfen, was kleine Änderungen in den Algorithmen erfordert.

Beispiele:







Degenerierter Suchbaum

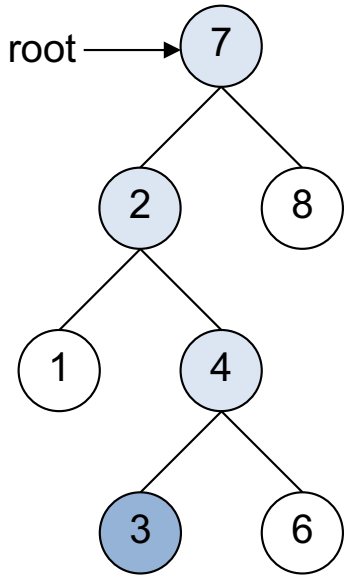
Klasse BinarySearchTree

```
public class BinarySearchTree<K extends Comparable<? super K>, V> {

    private static class Node<K, V> {
        private K key;
        private V value;
        private Node<K, V>  left;
        private Node<K, V> right;
        private Node(K k, V v) {
            key = k;
            value = v;
            left = null;
            right = null;
        }
    }

    private Node<K, V> root = null;

    // ...
}
```

Suchen in binären Suchbäumen

```
public V search(K key) {
        return searchR(key, root);
}

private V searchR(K key, Node<K,V> p) {
        if (p == null)
                return null;
        else if (key.compareTo(p.key) < 0)
                return searchR(key, p.left);
        else if (key.compareTo(p.key) > 0)
                return searchR(key, p.right);
        else
                return p.value;
}
```

Beispiel



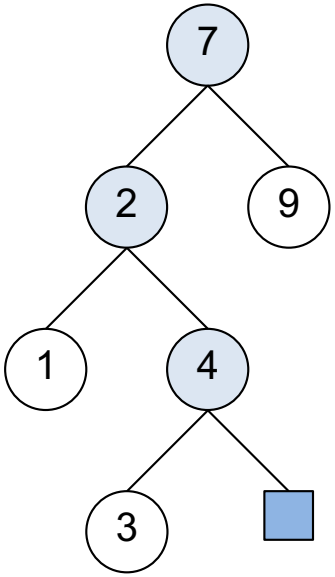
searchR(3, root)
liefert true

Einfügen in binären Suchbäumen (1)

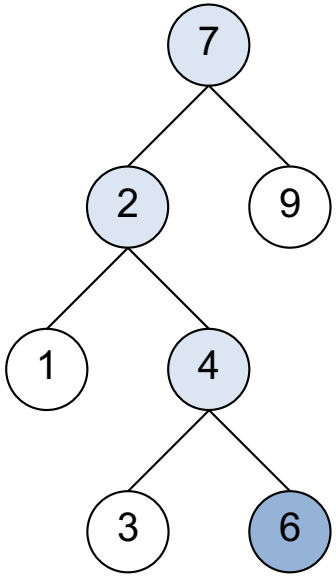
Idee

- Um eine Zahl x einzufügen, wird zunächst nach x gesucht.
- Falls x nicht bereits im Baum vorkommt,
  endet die Suche erfolglos bei einer null-Referenz.
- An dieser Stelle wird dann ein neuen Knoten mit Eintrag x eingefügt.

Beispiel 1: füge 6 ein

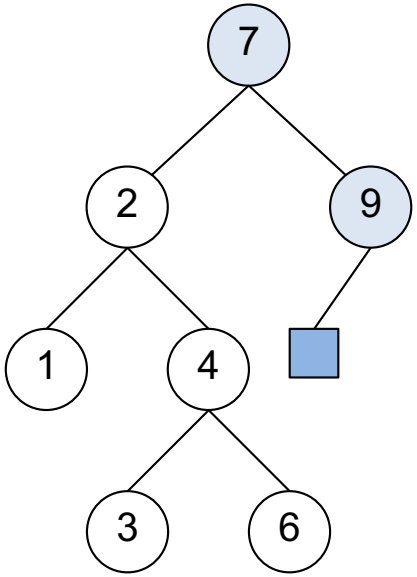


Suche von 6 endet bei null

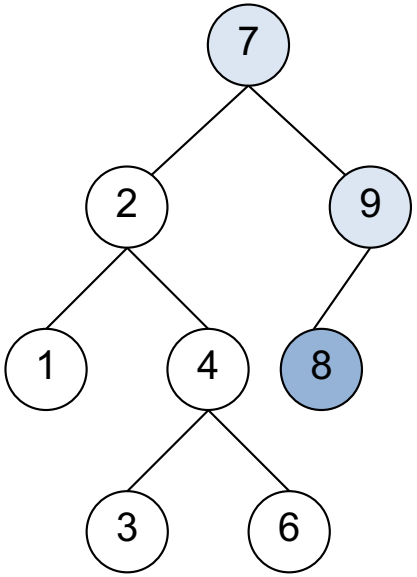


Ersetzte null durch neuen Knoten mit Eintrag 6

Beispiel 2: füge 8 ein



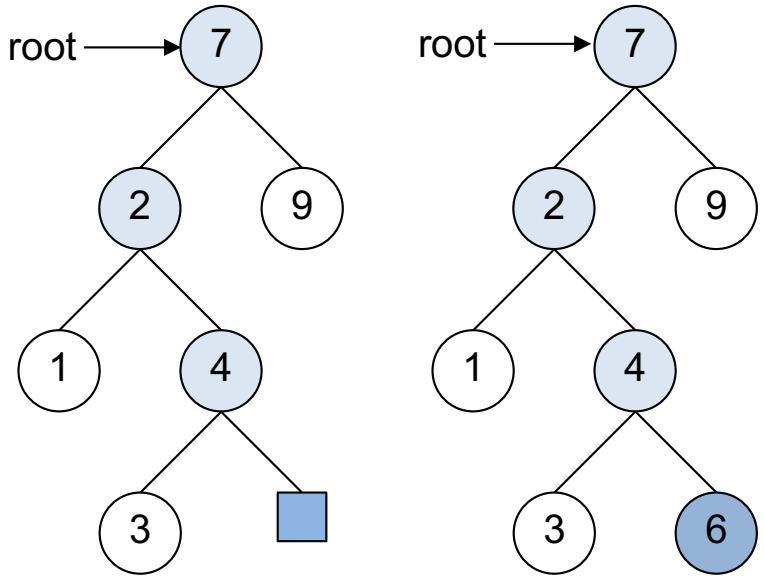
Suche von 8 endet bei null



Ersetzte null durch Knoten mit Eintrag 8

Einfügen in binären Suchbäumen (2)

```
private V oldValue; // Rückgabeparameter

public V insert(K key, V value) {
        root = insertR(key, value, root);
        return oldValue;
}

private Node<K,V> insertR(K key, V value, Node<K,V> p) {
        if (p == null) {
                p = new Node(key, value);
                oldValue = null;
        }
        else if (key.compareTo(p.key) < 0)
                p.left = insertR(key, value, p.left);
        else if (key.compareTo(p.key) > 0)
                p.right = insertR(key, value, p.right);
        else {    // Schlüssel bereits vorhanden:
                oldValue = p.value;
                p.value = value;
        }
        return p;
}
```



root = insert(6, root);

Löschen in binären Suchbäumen (1)

Idee

- Um eine Zahl x zu löschen, wird zunächst nach x gesucht.
  Es sind dann 4 Fälle zu unterscheiden:

- Fall „Nicht vorhanden“:

  x kommt nicht vor. Dann ist nichts zu tun.

- Fall „Keine Kinder“:

  x kommt in einem Blatt vor (keine Kinder):
  dann kann der Knoten einfach entfernt werden.

- Fall „Ein Kind“:

  Der Knoten, der x enthält, hat genau ein Kind:
  s. nächste Folie

- Fall „Zwei Kinder“:

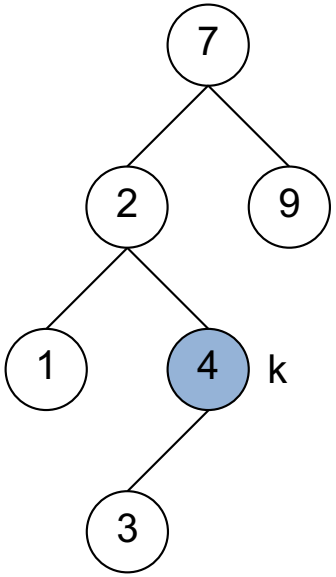
  Der Knoten, der x enthält, hat zwei Kinder:
  s. übernächste Folie

Löschen in binären Suchbäumen (2)

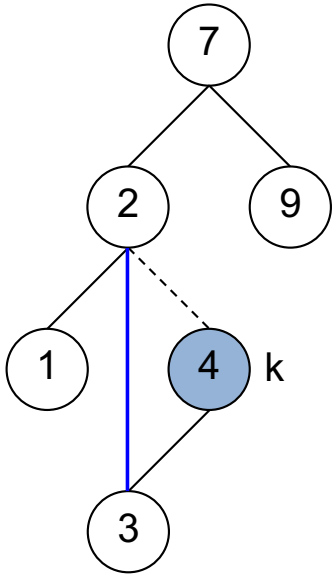
Fall: der zu löschende Knoten k hat ein Kind

- Überbrücke den Knoten k, indem der Elternknoten von k auf das Kind von k verzeigert wird (Bypass).

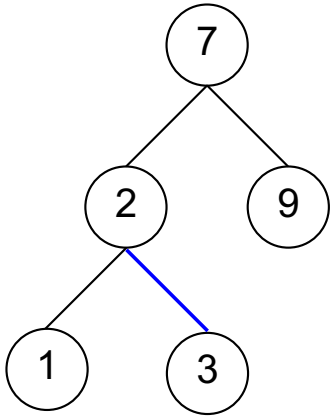
Beispiel: lösche Knoten k mit Inhalt 4



Suche 4



Knoten 4 wird überbrückt (Bypass)



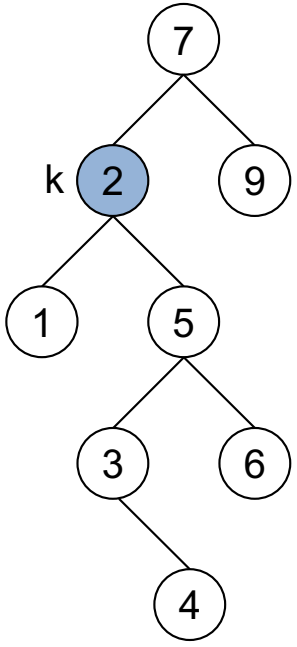
Knoten 4 wird gelöscht

Löschen in binären Suchbäumen (3)

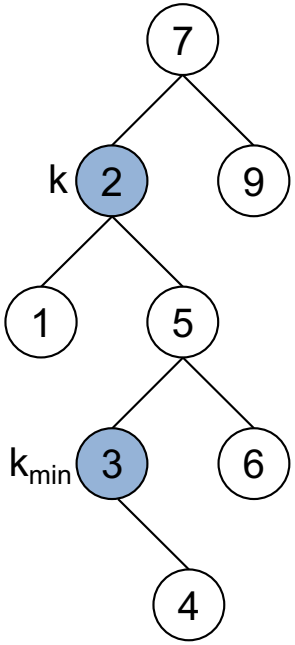
Fall: der zu löschende Knoten k hat zwei Kinder

- Ersetze den Knoten k durch den kleinsten Knoten k_{min} im rechten Teilbaum von k.
- Lösche dann k_{min}.
- Da der Knoten k_{min} kein linkes Kind haben kann, kann das Löschen von k_{min} wie im Fall „Ein Kind“ bzw. „Keine Kinder“ behandelt werden.

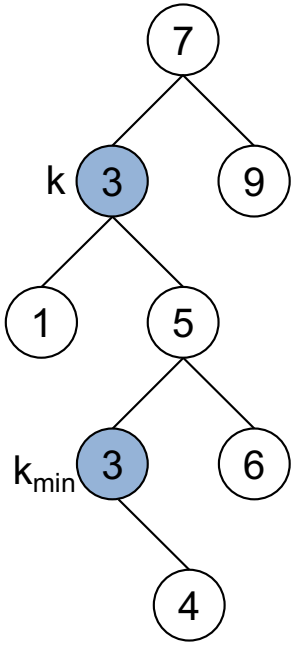
Beispiel: lösche Knoten k mit Inhalt 2



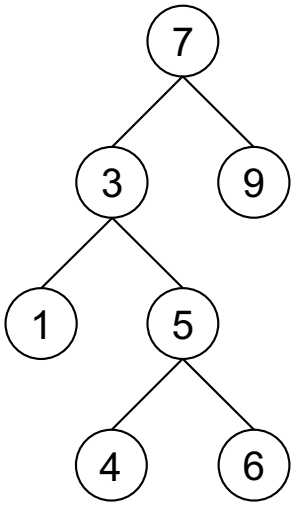
Suche k mit Inhalt 2



Suche kleinsten Knoten k_{min} in rechten Teilbaum von k



Knoten k wird durch k_{min} ersetzt



k_{min} wird gelöscht

Löschen in binären Suchbäumen (4)

```
private V oldValue;       // Rückgabeparameter

public V remove(K key) {
        root = removeR(k, root);
        return oldValue;
}

private Node<K,V> removeR(K key, Node<K,V> p) {
        if (p == null) {  oldValue = null; }
        else if(key.compareTo(p.key) < 0)
                p.left = removeR(key, p.left);
        else if (key.compareTo(p.key) > 0)
                p.right = removeR(key, p.right);
        else if (p.left == null || p.right == null) {
                // p muss gelöscht werden
                // und hat ein oder kein Kind:
                oldValue = p.value;
                p = (p.left != null) ? p.left : p.right;
        } else {
                // p muss gelöscht werden und hat zwei Kinder:
                MinEntry<K,V> min = new MinEntry<K,V>();
                p.right = getRemMinR(p.right, min);
                oldValue = p.value;
                p.key = min.key;
                p.value = min.value;
        }
        return p;
}
```

```
private Node<K,V> getRemMinR(Node<K,V> p, MinEntry<K,V> min) {
        assert p != null;
        if (p.left == null) {
                min.key = p.key;
                min.value = p.value;
                p = p.right;
        }
        else
                p.left = getRemMinR(p.left, min);
        return p;
}

private static class MinEntry<K, V> {
        private K key;
        private V value;
}
```

- getRemMinR löscht im Baum p den Knoten mit kleinstem Schlüssel und liefert Schlüssel und Daten des gelöschten Knotens über min zurück
- MinEntry ist ein Hilfsdatentyp für den Rückgabe-Parameter min von getRemMinR

Analyse

Worst-Case

- Binärer Suchbaum mit n Knoten kann zu einem Baum der Höhe n-1 entarten.
- Beispiel: Einfügen einer sortierten Folge 1, 2,3 …, n.
- Damit: $T_{max}(n) = O(n)$

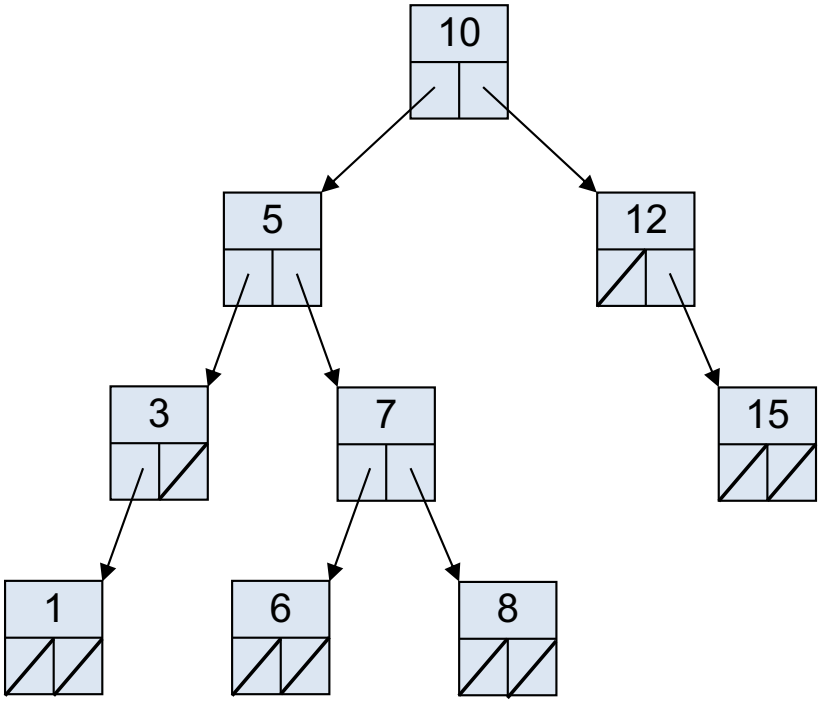
Average-Case

- Einfügen einer zufälligen Folge von n Elementen sorgt im Schnitt für eine logarithmische Höhe:
- Damit: $T_{mit}(n) = O(\log_2 n)$

3. Binäre Suchbäume

- Begriffe und Eigenschaften (Wiederholung PROG 2)
- Binäre Suchbäume (Wiederholung PROG 2)
- Iterative Traversierung mit Elternzeigern

Traversierungsreihenfolge bei binären Suchbäumen

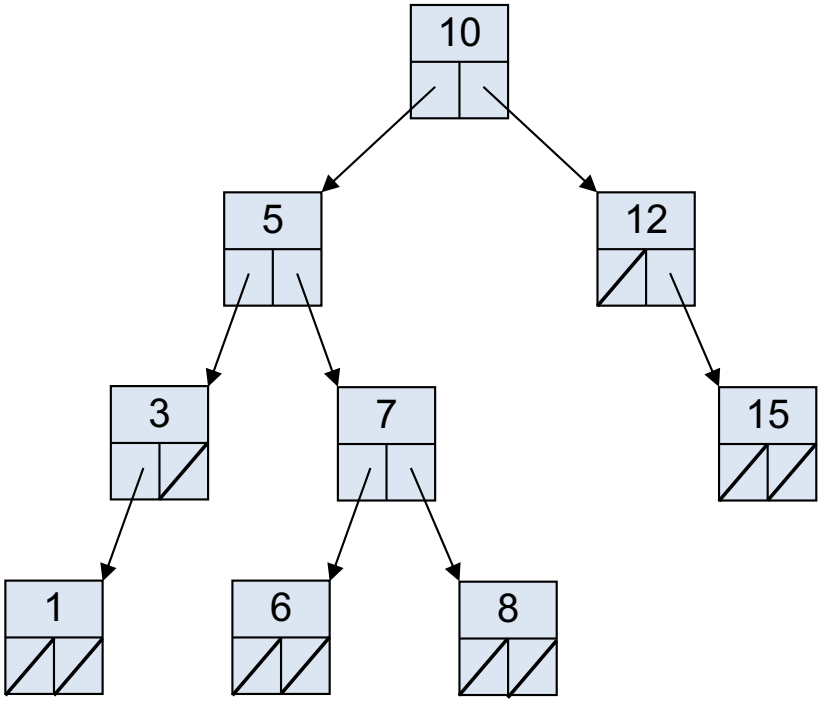


PrePrder	Knoten – linkes Kind – rechtes Kind	10, 5, 3, 1, 7, 6, 8, 12, 15
PostOrder	linkes Kind – rechtes Kind – Knoten	1, 3, 6, 8, 7, 5, 15, 12, 10
InOrder	linkes Kind – Knoten – rechtes Kind	1, 3, 5, 6, 7, 8, 10, 12, 15 sortierte Reihenfolge!

Iterative InOrder-Traversierung

Problem

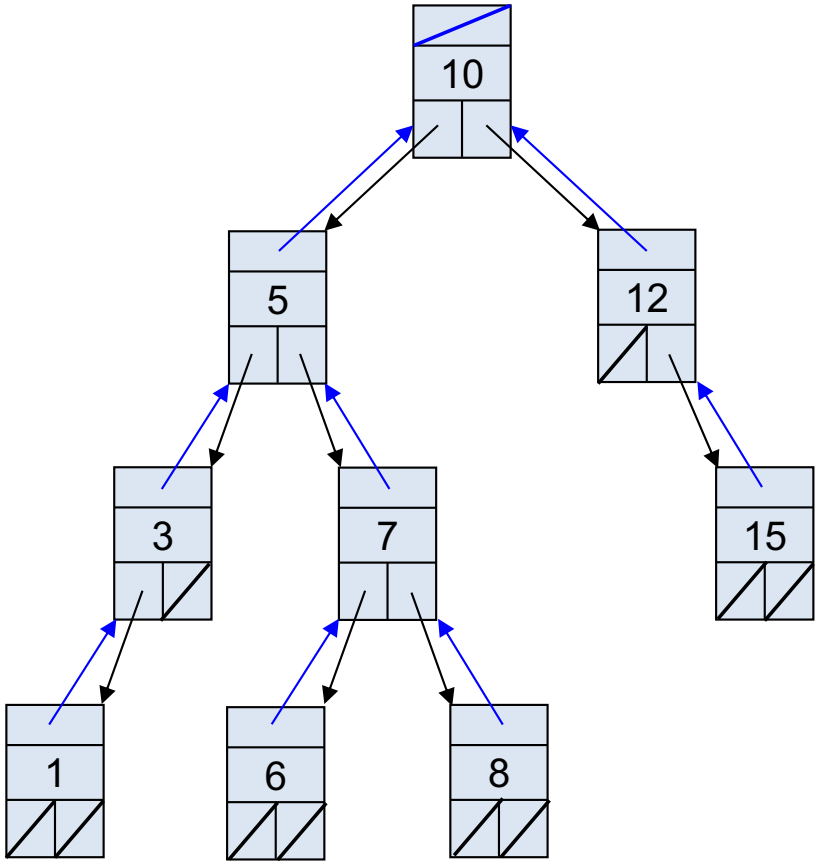
- In binären Suchbäumen gibt es für einen Knoten im allgemeinen keinen effizienten Zugriff auf seinen InOrder-Vorgänger bzw. -Nachfolger.
- Daher ist mit der bisher besprochenen Datenstruktur für Suchbäume keine effiziente Vorwärts- bzw. Rückwärtstraversierung mit Iteratoren machbar.



- Wie erreicht man den Nachfolger von 8?
- Wie erreicht man den Vorgänger von 6?

Bäume mit Elternzeigern

- Erweitere jeden Knoten um einen Zeiger auf den Elternknoten.



Klasse BinarySearchTree mit Elternzeiger (1)

- Erweitere Klasse Node um Elternzeiger parent:

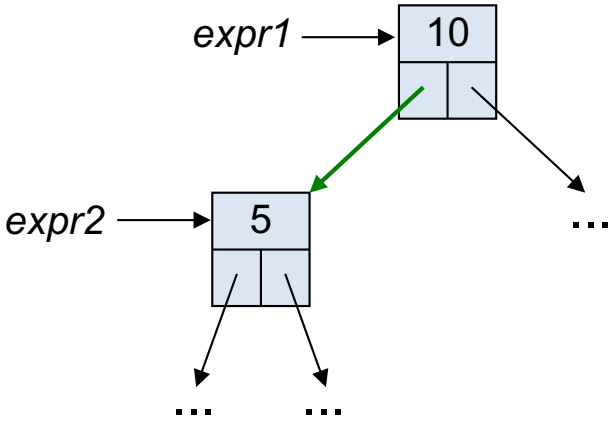
```
public class BinarySearchTree<K extends …, V> {

    private static class Node<K, V> {
        private Node<K, V> parent;   // Elternzeiger
        private K key;
        private V value;
        private Node<K, V>  left;
        private Node<K, V> right;

        private Node(K k, V v) {
            key = k;
            value = v;
            left = null;
            right = null;
            parent = null;
        }
    }

    private Node<K, V> root = null;

    // ...
}
```

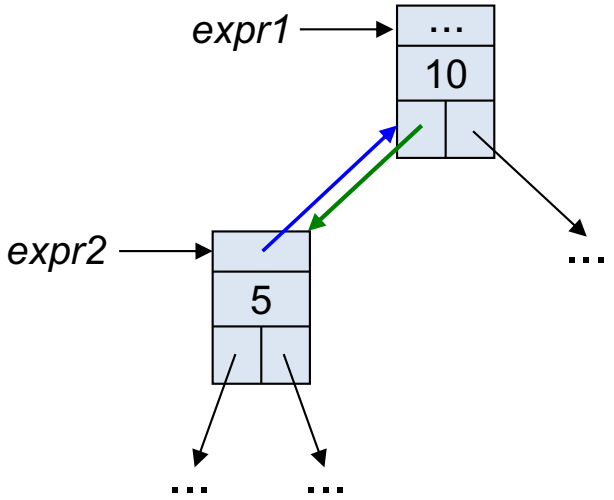
Klasse BinarySearchTree mit Elternzeiger (2)

- Überall, wo die Struktur des Baums geändert wird, wird zusätzlich der parent-Zeiger neu gesetzt:

```
expr1.left = expr2;
```

```
expr1.left = expr2;
if (expr1.left != null)
    expr1.left.parent = expr1;
```





expr1 und *expr2* sind beliebige Ausdrücke vom Typ Node.

Klasse BinarySearchTree mit Elternzeiger (3)

- Analog:

```
expr1.right = expr2;
```

```
expr1.right = expr2;
if (expr1.right != null)
    expr1.right.parent = expr1;
```

- Beachte, dass der parent-Zeiger des root-Knotens den Wert null bekommt:

```
root = expr;
```

```
root = expr;
if (root != null)
    root.parent = null;
```

Beispiel: insert-Methode mit Elternzeiger

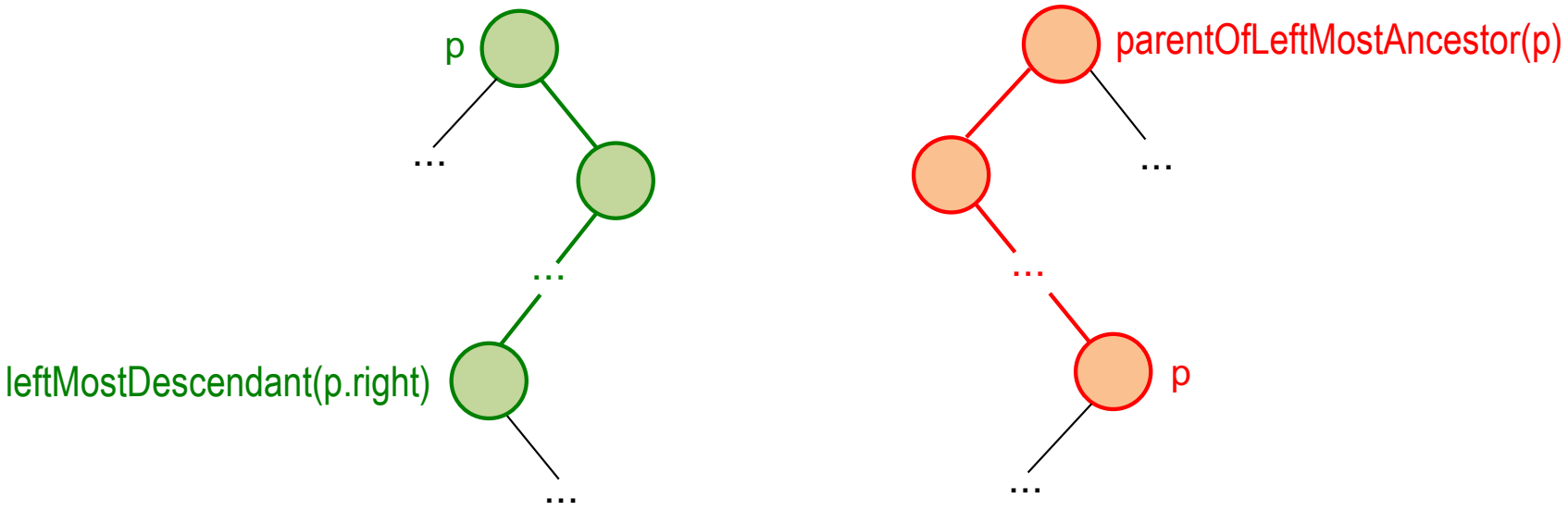
```
public V insert(K key, V value) {
        root = insertR(key, value, root);
        if (root != null)
                root.parent = null;
        return oldValue;
}
```

```
private Node<K,V> insertR(K key, V value, Node<K,V> p) {
        if (p == null) {
                p = new Node(key, value);
                oldValue = null;
        } else if (key.compareTo(p.key) < 0) {
                p.left = insertR(key, value, p.left);
                if (p.left != null)
                        p.left.parent = p;
        } else if (key.compareTo(p.key) > 0) {
                p.right = insertR(key, value, p.right);
                if (p.right != null)
                        p.right.parent = p;
        } else {    // Schlüssel bereits vorhanden:
                oldValue = p.value;
                p.value = value;
        }
        return p;
}
```

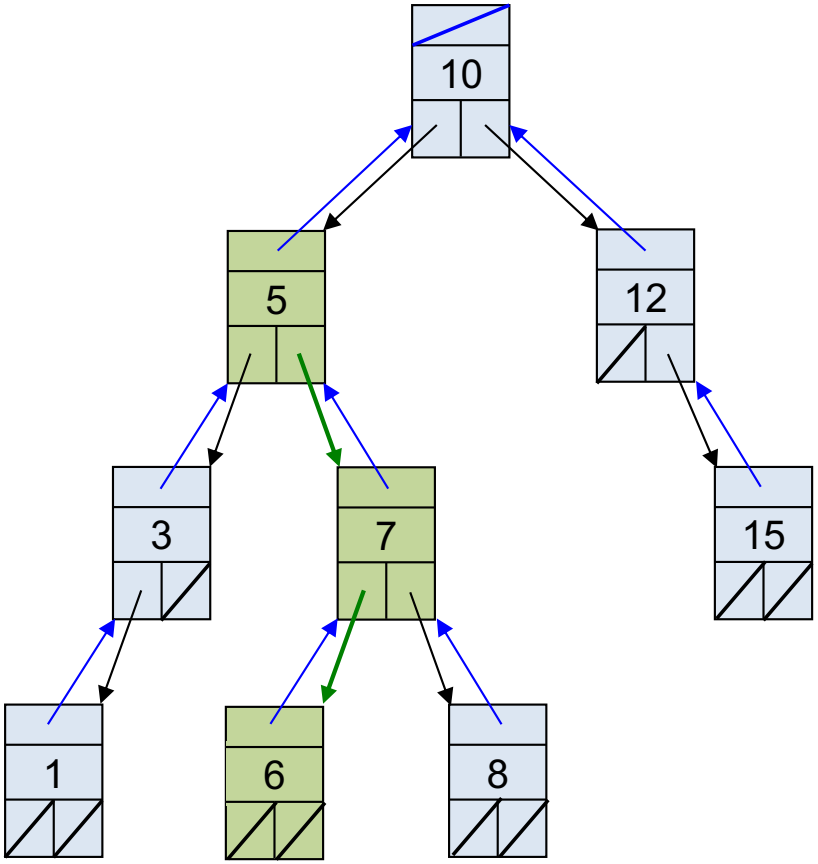
Iterative Traversierung von Bäumen

- Hier: nur Traversierung zum InOrder-Nachfolger.
  Traversierung zum InOrder-Vorgänger geht analog.
- Sei p der aktuelle Knoten. Gesucht ist der Nachfolger von p.
  Wir unterscheiden zwei Fälle:

```
if (p.right != null)
      p = leftMostDescendant(p.right);
else
      p = parentOfLeftMostAncestor(p);
```



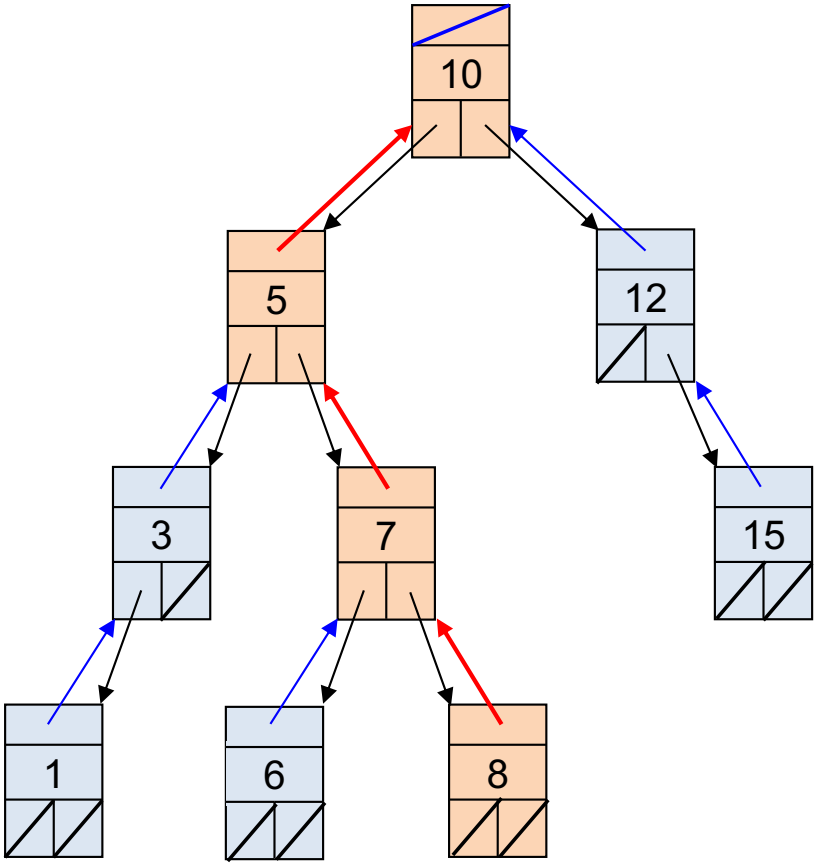
leftMostDescendant



InOrder-Nachfolger von p = 5:

leftMostDescendant(p.right) = 6

parentOfLeftMostAncestor

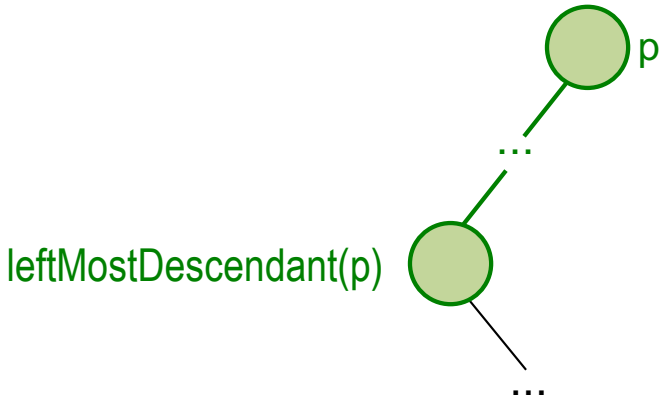


InOrder-Nachfolger von p = 8:

parentOfLeftMostAncestor(p) = 10

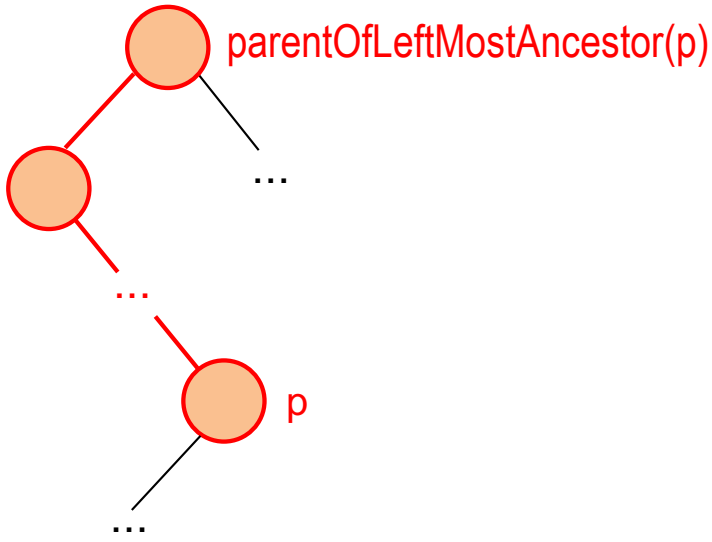
Implementierung von leftMostDescendant

```
private Node<K,V> leftMostDescendant(Node<K,V> p) {
      assert p != null;
      while (p.left != null)
            p = p.left;
      return p;
}
```



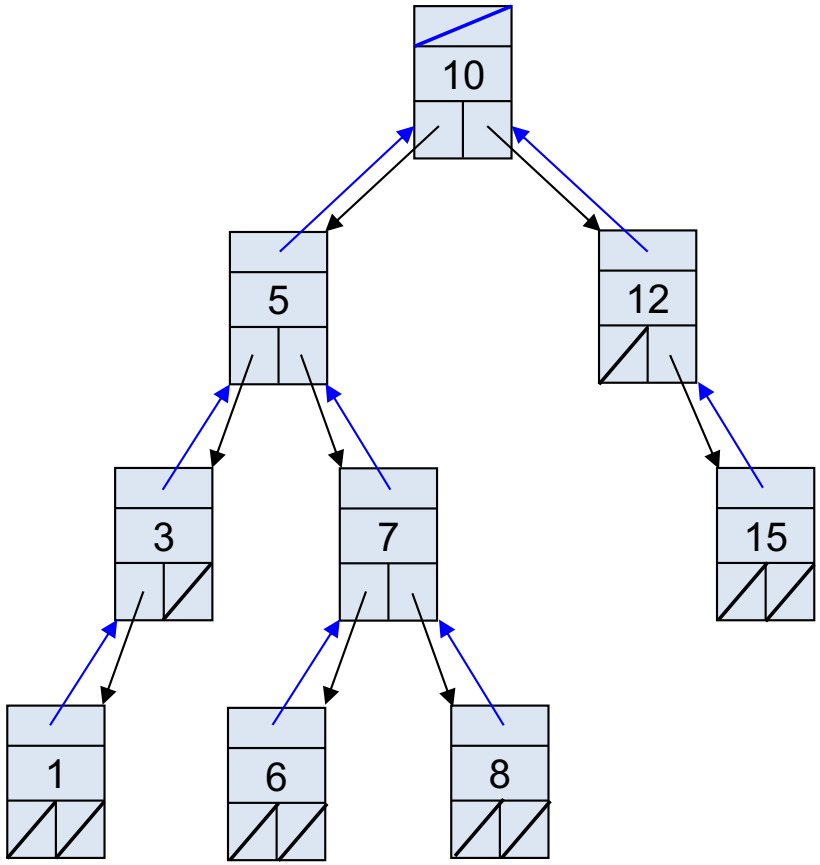
Implementierung von parentOfLeftMostAncestor

```
private Node<K,V> parentOfLeftMostAncestor(Node<K,V> p) {
      assert p != null;
      while (p.parent != null && p.parent.right == p)
            p = p.parent;
      return p.parent;        // kann auch null sein
}
```



Traversierungsschleife für InOrder-Nachfolger

```
// Erster Knoten:
Node<K,V> p = null;
if (root != null)
        p = leftMostDescendant(root);

while( p != null) {
        System.out.print(p.key + ", ");
        if (p.right != null)
                p= leftMostDescendant(p.right);
        else
                p = ParentOfLeftMostAncestor(p);
}
```



- Traversierungsschleife ergibt:
  1, 3, 5, 6, 7, 8, 10, 12, 15