Algorithmen und Datenstrukturen

7. DYNAMISCHE DATENSTRUKTUREN 1
SUCHBÄUME

Dynamische Datenstrukturen

- Unter dynamischen Datenstrukturen verstehen wir Datenstrukturen bei denen man
  - Elemente löschen und hinzufügen kann
  - Eine interne Ordnung (z.B. Sortierung) besitzen und
  - diese Ordnung unter Änderungen aufrecht erhalten kann

Beispiel: Lineare Datenstrukturen und Sortierung

- unsortierte Liste: Änderung einfach, Zugriff aufwändig
- Neusortierung einer Liste: Änderung schwierig, Zugriff einfach
- Trade-off: gesucht ist eine “intelligente Datenstruktur”, die Änderungen und Zugriff einfach (=effizient) halten

Viele dynamische Datenstrukturen nutzen Bäume als Repräsentation

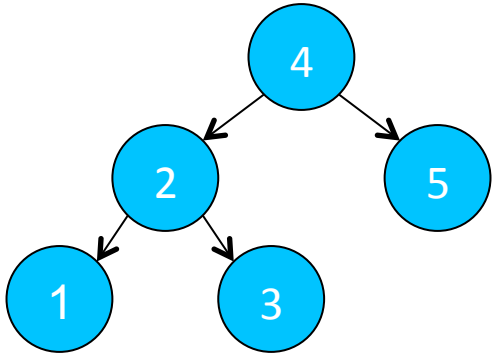
→ Kurzer Einschub zu Bäumen als Datenstruktur

Definition

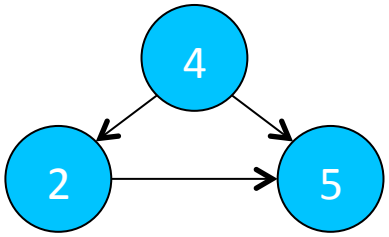
- Ein *Baumelement* e ist ein Tupel $e=(v,\{e_1,\ldots e_n\})$ mit
  - v ist der Wert von e
  - $\{e_1,\ldots,e_n\}$ sind die Nachfolger (oder Kinder) von e
- Ein Baum T ist ein Tupel $T=(r, \{e_1,\ldots,e_n\})$ mit
  - r ist der Wurzelknoten (ein Baumelement)
  - $\{e_1,\ldots,e_n\}$ sind die Knoten (Baumelemente) des Baumes mit
    - $r \in \{e_1,\ldots,e_n\}$
    - Für alle $e_i=(v_i,K_i)$, $e_j=(v_j,K_j) \in \{e_1,\ldots e_n\}$ gilt $K_i \cap K_j = \varnothing$
- Man spricht von einem *geordneten Baum*, wenn die Reihenfolge der Kinder $\{e_1,\ldots,e_n\}$ eines jeden Elements $e=(v,\{e_1,\ldots,e_n\})$ festgelegt ist (schreibe dann $(e_1,\ldots,e_n)$ statt $\{e_1,\ldots,e_n\}$)

Beispiel

$T=(v_4,\{v_1,v_2,v_3,v_4,v_5\})$
$v_1=(1,\{\})$
$v_2=(2,\{v_1,v_3\})$
$v_3=(3,\{\})$
$v_4=(4,\{v_2,v_5\})$
$v_5=(5,\{\})$

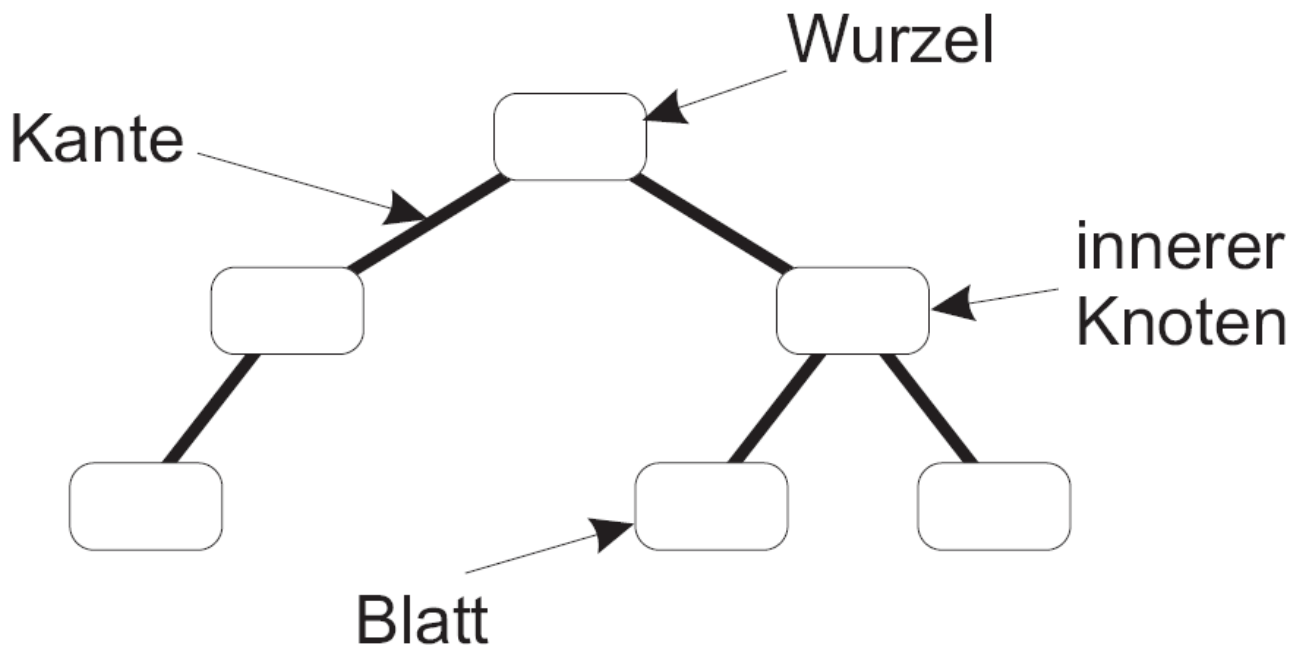


$T'=(v_4,\{v_2,v_4,v_5\})$
$v_2=(2,\{v_5\})$
$v_4=(4,\{v_2,v_5\})$
$v_5=(5,\{\})$



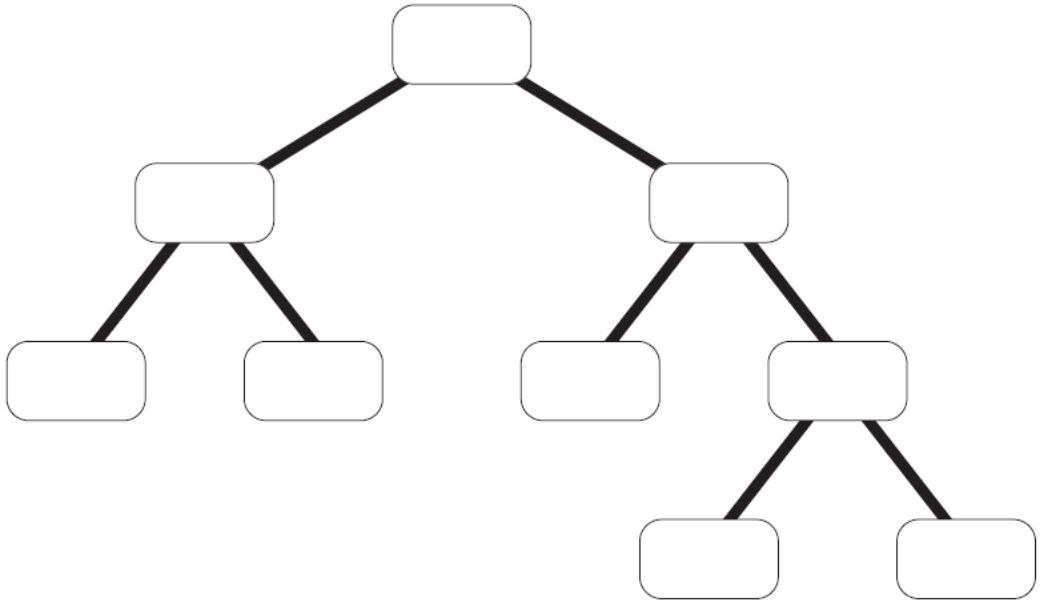
Kein Baum da v_4 und v_2 ein gemeinsames Kind haben!

Begriffe



- Pfad: Folge von durch Kanten verbundene Knoten
  - Zu jedem Knoten existiert **genau ein** Pfad von der Wurzel
- Baum: zusammenhängend, zyklenfrei

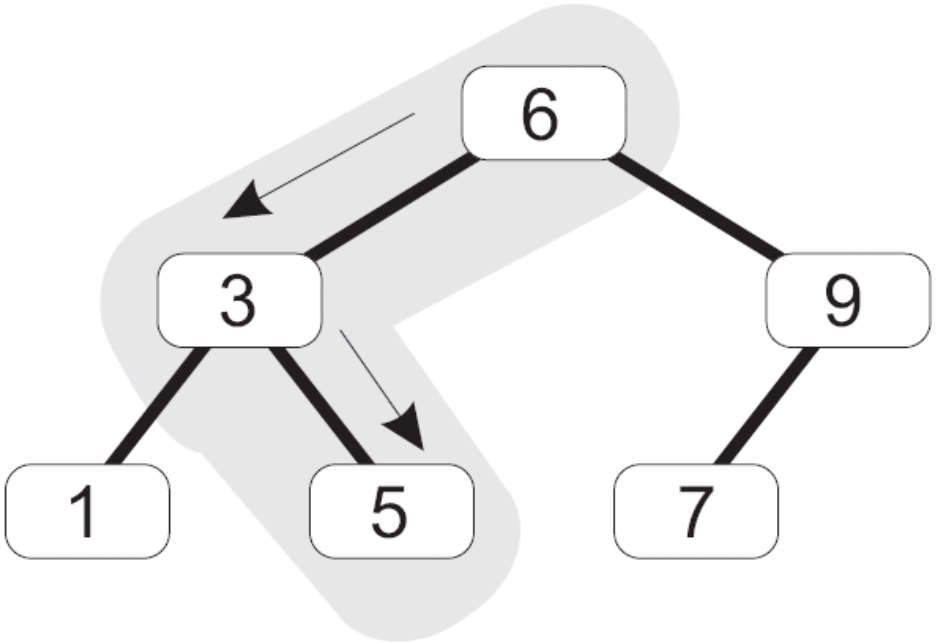
Begriffe



- Höhe := größtes Niveau + 1

Länge des Pfades

Anwendung: Baum als Datenindex



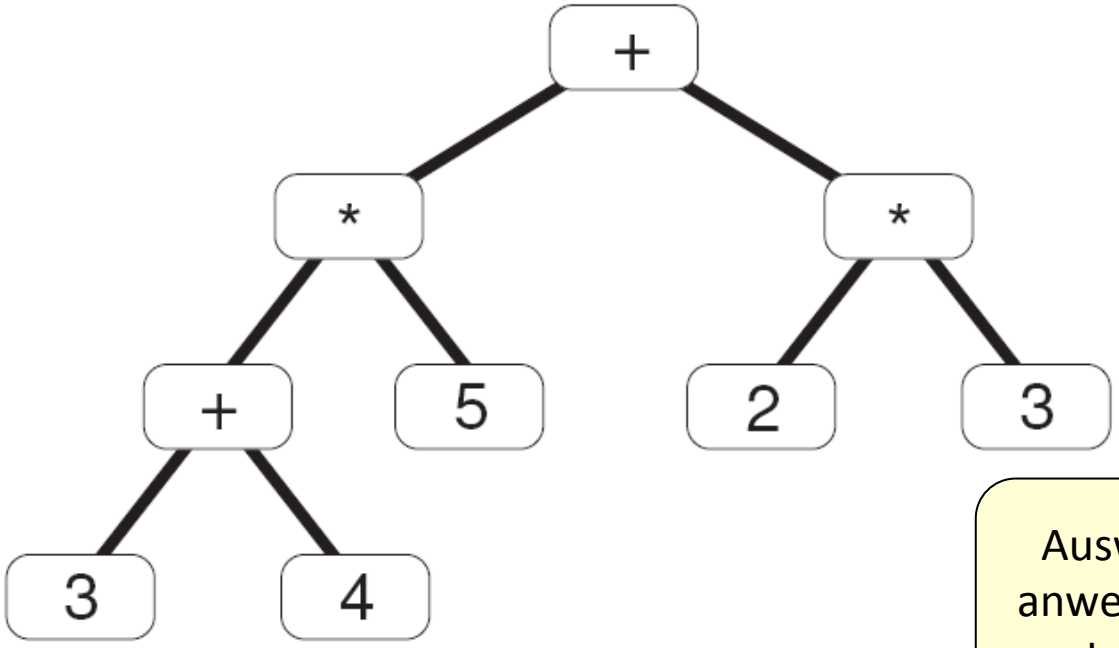
Alternative zu Listen und Arrays

Beispiel: Suche nach „5“

→ Suchbaum

Anwendung: Baum aus Termen

- Term (3+4) * 5 + 2 * 3
- in Baumdarstellung:



Auswertung: Operator anwenden auf Werte der beiden Teilbäume

Atomare Operationen auf Bäumen

Lesen:

- root(): Wurzelknoten eines Baums
- get(e): Wert eines Baumelements e
- children(e): Kinderknoten eines Elements e
- parent(e): Elternknoten eines Elements e

Schreiben:

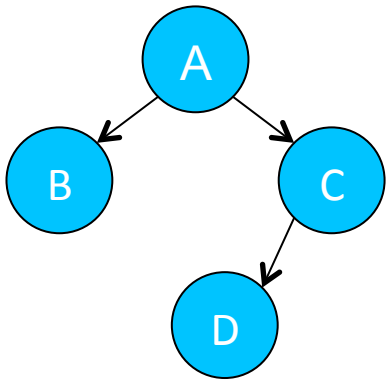
- set(e,v): Wert des Elements e auf v setzen
- addChild(e,e'): Füge Element e' als Kind von e ein (falls geordneter Baum nutze addChild(e,e',i) für Index i)
- del(e): Lösche Element e (nur wenn e keine Kinder hat)

Spezialfall: Binärer Baum als Datentyp

```
class TreeNode<K extends Comparable<K>>{

        K key;
        TreeNode<K> left = null;
        TreeNode<K> right = null;

        public TreeNode(K e) { key = e; }
        public TreeNode<K> getLeft() { return left; }
        public TreeNode<K> getRight()  { return right; }
        public K getKey() { return key; }

        public void setLeft(TreeNode<K> n) {left = n;}
        public void setRight(TreeNode<K> n) {right = n;}

          ...

 }
```

Beispiel

```
TreeNode<Character> root = new TreeNode<Character>('A');
TreeNode<Character> node1 = new TreeNode<Character>('B');
TreeNode<Character> node2 = new TreeNode<Character>('C');
TreeNode<Character> node3 = new TreeNode<Character>('D');

root.setLeft(node1);
root.setRight(node2);
node2.setLeft(node3);
```

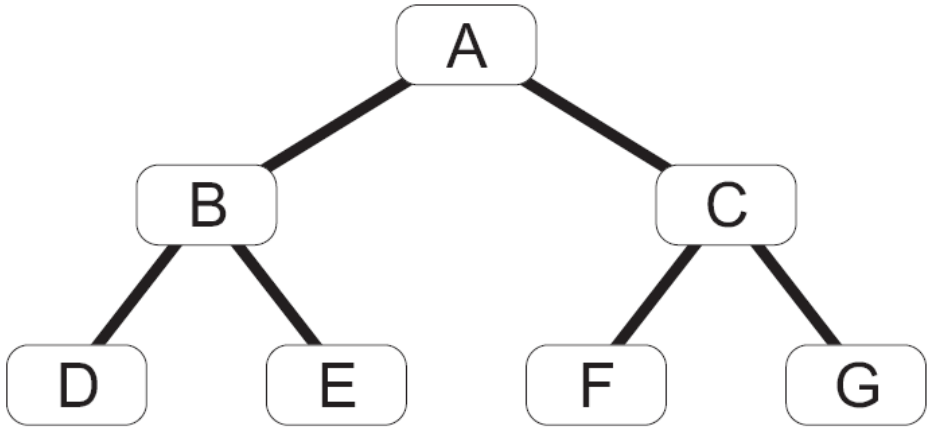


Traversierung

- Bäume können visuell gut dargestellt werden
- Manchmal ist jedoch eine *Serialisierung* der Elemente eines Baumes nötig
- Eindeutige Aufzählung aller Elemente eines (binären) Baumes durch
  - Preorder-Aufzählung
  - Inorder-Aufzählung
  - Postorder-Aufzählung
  - Levelorder-Aufzählung

Traversierung

- Systematisches Durchlaufen aller Knoten des Baumes



Preorder: A → B → D → E → C → F → G
Inorder: D → B → E → A → F → C → G
Postorder: D → E → B → F → G → C → A
Levelorder: A → B → C → D → E → F → G

Traversierung von (Binär-) Bäumen

- Traversierung durch Methoden
  - Preorder
  - Inorder
  - Postorder
  - Levelorder
- Traversierung mit Hilfe von Iteratoren
  - Erfordert Zwischenspeicherung des Traversierungszustands

Preorder-Traversierung

- Preorder-Traversierung: Behandlung (Ausgabe) des aktuellen Knotens zuerst, dann linker und rechter Teilbaum (W-L-R)

```
class TreeNode<K extends Comparable<K>> {
 ...
 public void print_preorder() {
   System.out.print(" " + key);
   if(left != null) left.print_preorder();
   if(right != null) right.print_preorder();
 }
```

Inorder-Traversierung

- Inorder-Traversierung: Behandlung des linken Teilbaums zuerst, dann des aktuellen Knotens, dann des rechten Teilbaums (L-W-R)
- Gibt damit einen Suchbaum in sortierter Reihenfolge aus

```
class TreeNode<K extends Comparable<K>> {
  ...
  public void print_inorder() {
      if(left != null) left.print_inorder();
      System.out.print(" " + key);
      if(right != null) right.print_inorder();
  }
```

Postorder-Traversierung

- Postorder-Traversierung: Behandlung des linken und rechten Teilbaums, dann erst des aktuellen Knotens (L-R-W)
- Kann beispielsweise genutzt werden um einen Baum aus Termen, entsprechend der Priorität der Operatoren, auszuwerten

```
class TreeNode<K extends Comparable<K>> {
  ...
  public void print_postorder() {
      if(left != null) left.print_postorder();
      if(right != null) right.print_postorder();
      System.out.print(" " + key);
  }
```

Levelorder-Traversierung

- Levelorder-Traversierung: *siehe Traversierung mit Iteratoren*

Traversierung mit Iteratoren

- Iteratoren erlauben
  - Schrittweise Abarbeitung
  - Verwendung von Standardschleifen für Baumdurchlauf

```
TreeNode<Integer> root = ...;

for(Integer i : root)
    System.out.print(i);
```

- Erfordert Zwischenspeicherung des Bearbeitungszustands
- Unterstützung verschiedener Iteratoren

Traversierung mit Iteratoren (2)

```
class TreeNode<K extends Comparable<K>> implements Iterable<K> {
  ...
  public Iterator<K> iterator() {
      return new MyIterator<K>(this);
  }
```

Inorder-Iterator

```
class InorderIterator<K extends Comparable <K>>
            implements Iterator<K> {

    java.util.Stack<TreeNode<K>> st =
         new java.util.Stack<TreeNode<K>>();

    public InorderIterator(TreeNode<K> root) {
         TreeNode<K> node = root;
         while (node != null) {
             st.push(node);
             node = node.getLeft();
         }
    }
   ...
```

Inorder-Iterator (2)

```
    ...
    public boolean hasNext() {
        return !st.isEmpty();
    }

   public K next(){
       TreeNode<K> node = st.pop();
       K obj = node.getKey();
       node = node.getRight();
       while (node != null) {
          st.push(node);
          node = node.getLeft();
       }
       return obj;
   }

}
```

rechten Knoten holen

linker Knoten auf den Stack

Preorder-Iterator

```
class PreorderIterator<K extends Comparable <K>>
        implements Iterator<K> {

    java.util.Stack<TreeNode<K>> st =
        new java.util.Stack<TreeNode<K>>();

    public PreorderIterator(TreeNode<K> root){
       st.push(root);
    }
  ...
```

Wurzelknoten auf den Stack legen

Preorder-Iterator (2)

```
 ...
  public boolean hasNext() {
          return !st.isEmpty();
  }

   public K next(){
       TreeNode<K> node = st.pop();
       K obj = node.getKey();
       node = node.getRight();
       if(node != null) {
          st.push(node);
       }
       node = node.getLeft();
       if(node != null) {
          st.push(node);
       }
       return obj;
   }
}
```

rechter Knoten auf den Stack

linker Knoten auf den Stack

Postorder-Iterator

Postorder-Implementierung mit Iteratoren überspringen wir (relativ unschöne Implementierung)

Levelorder-Iterator

```
class LevelorderIterator<K extends Comparable <K>>
        implements Iterator<K> {

  java.util.Queue<TreeNode<K>> q =
        new java.util.LinkedList<TreeNode<K>>();

  public LevelorderIterator(TreeNode<K> root){
      if(root != null) q.add (root);
  }

  ...
```

Wurzelknoten in die Warteschlange (queue) einfügen

hier:
queue als LinkedList
(beliebig)

Levelorder-Iterator (2)

```
 ...


   public K next(){
        TreeNode<K> node = q.getFirst();
        K obj = node.getKey();
        if (node.getLeft() != null)
             q.addLast(node.getLeft());
        if (node.getRight() != null)
             q.addLast(node.getRight());
        return obj;
   }

   ...
}
```

zuerst linken, dann rechten Knoten in die Warteschlange einfügen

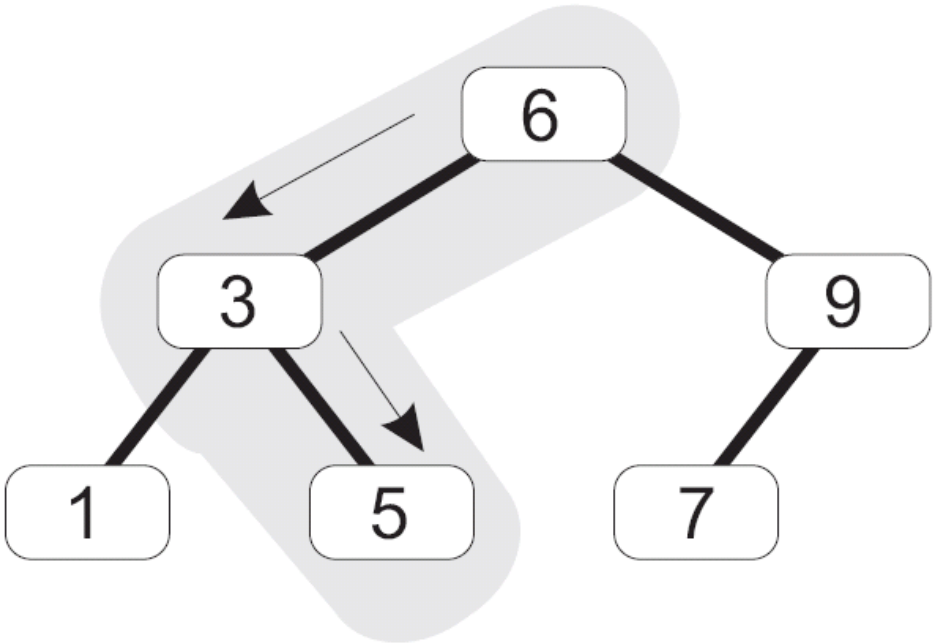
Algorithmen und Datenstrukturen

7.1. BINÄRE SUCHBÄUME

Suchbäume

- Anwendung von Bäumen zur effizienten Suche
- Prinzip:
  - Pro Knoten: Schlüssel und Datenelement
  - Ordnung der Knoten anhand der Schlüssel
- Binärer Suchbaum
  - Knoten k enthält einen Schlüsselwert: k.key
  - Alle Schlüsselwerte im linken Teilbaum k.left sind kleiner als k.key
  - Alle Schlüsselwerte im rechten Teilbaum k.right sind größer als k.key

Suchen im Suchbaum



1. Vergleich des Suchschlüssels mit Schlüssel der Wurzel
2. Wenn kleiner, dann in linken Teilbaum weitersuchen
3. Wenn größer, dann in rechtem Teilbaum weitersuchen
4. Sonst gefunden/nicht gefunden

Nochmal: Binärer Baum als Datentyp

```
class TreeNode<K extends Comparable<K>>{

        K key;
        TreeNode<K> left = null;
        TreeNode<K> right = null;

        public TreeNode(K e) {key = e; }
        public TreeNode<K> getLeft() {return left; }
        public TreeNode<K> getRight()  {return right; }
        public K getKey() {return key; }

        public void setLeft(TreeNode<K> n) {left = n;}
        public void setRight(TreeNode<K> n) {right = n;}

           ...

 }
```

Operationen auf Suchbäumen

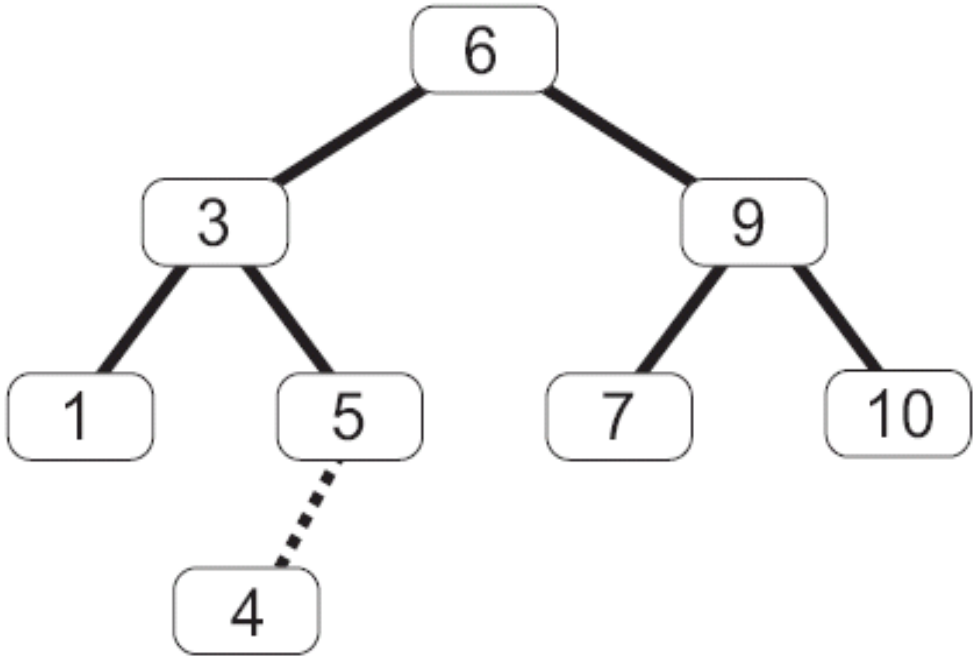
- Suche von Elementen

- Einfügen von Elementen
- Entfernen von Elementen

} Voraussetzung für Suche: Erhaltung der Ordnung der Schlüssel

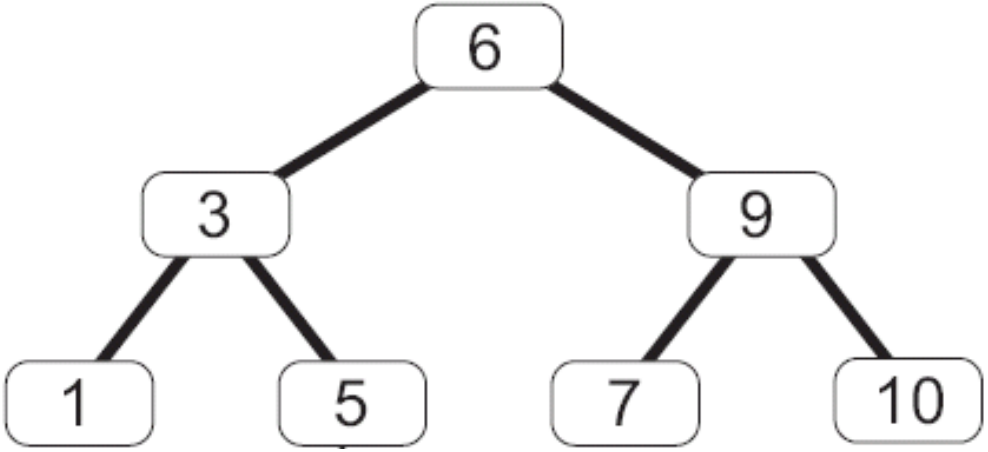
Einfügen

- Finden der Einfügeposition
  - Knoten, dessen Schlüsselwert größer als der einzufügende Schlüssel ist und der keinen linken Nachfolger hat
  - Knoten, dessen Schlüsselwert kleiner als der einzufügende Schlüssel ist und der keinen rechten Nachfolger hat



Löschen

- zu löschendes Element suchen: Knoten k
- drei Fälle:
  1. k ist Blatt: löschen
  2. k hat ein Kind: Kind „hochziehen“
  3. k hat zwei Kinder: Tausche mit weitest links stehenden Kind des rechten Teilbaums, da dieser in der Sortierreihenfolge der nächste Knoten ist
     - Entferne diesen nach den Regeln 1. oder 2.



Beispiel:
Lösche 5, dann 3, dann 6

Implementierung eines binären Suchbaums

- Binärer Suchbaum:
  - Häufig verwendete Hauptspeicherstruktur
  - Insbesondere geeignet für Schlüssel fester Größe, z.B. numerische int, float und char[n]
  - Gewährleistet $O(\log_2 n)$ für Suchen, Einfügen und Löschen, vorausgesetzt Baum ist **balanciert**
- Später:
  - Gewährleistung der Balancierung durch spezielle Algorithmen
  - Für Sekundärspeicher: größere, angepasste Knoten günstiger
    - B-Bäume
  - Für Zeichenketten als Schlüssel: variable Schlüsselgröße
    - sogenannte Tries

Implementierung eines binären Suchbaums (2)

```
public class BinarySearchTree<K extends Comparable<K>>
             implements Iterable<K> {

    ...

    static class TreeNode<K extends Comparable<K>> {
         K key;
         TreeNode<K> left = null;
         TreeNode<K> right = null;

         ...

    }
 }
```

Implementierung eines binären Suchbaums (3)

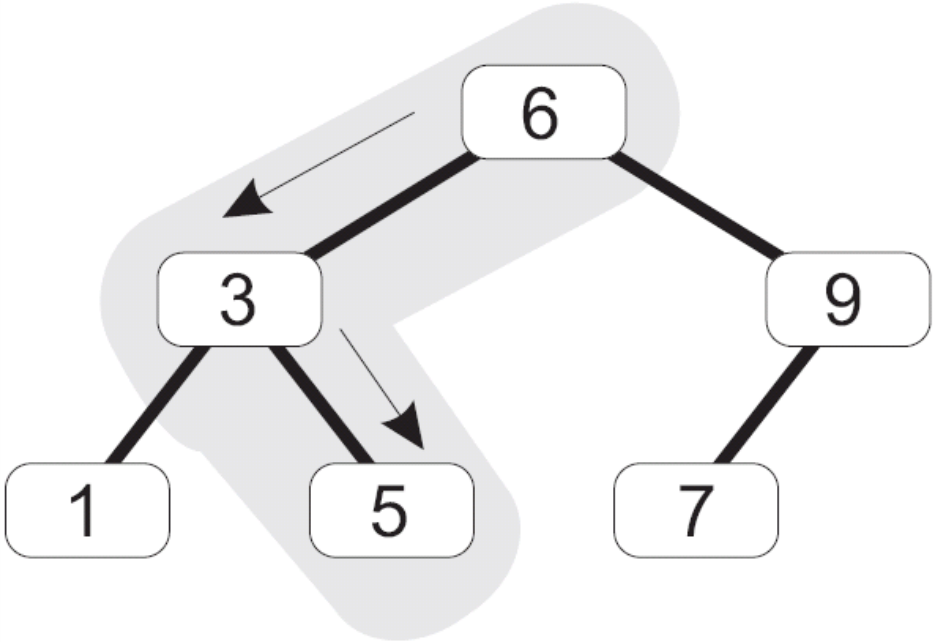
- Schlüssel müssen Comparable-Interface, d.h. compareTo()-Methode, implementieren, da Suchbaum auf Vergleichen der Schlüssel basiert
- Baum selbst implementiert Iterable-Interface, d.h. iterator()-Methode, um Traversierung des Baums über Iterator zu erlauben
  - Siehe Baumtraversierung
- TreeNode und alles weitere als innere Klassen implementiert
  - erlaubt Zugriff auf Attribute und Methoden der Baumklasse

Implementierung Binärer Suchbaum

- Konventionen
  - root = null: leerer Baum
  - node.getLeft() = null: kein linkes Kind (genauso für node.getRight())

```
public class BinarySearchTree<K extends Comparable<K>>
           implements Iterable<K> {

        private TreeNode<K> root;
        ...

        public SimpleBinarySearchTree(){
             root = null;
        }

        ...

 }
```

Nochmal: Suchen im binären Suchbaum



1. Vergleich des Suchschlüssels mit Schlüssel der Wurzel
2. wenn kleiner, dann in linken Teilbaum weitersuchen
3. wenn größer, dann in rechten Teilbaum weitersuchen
4. sonst
   - ➔ gefunden/nicht gefunden

Binärer Suchbaum: Knotenvergleich

```
class  TreeNode<...> {
          ...

     public int compareTo(TreeNode<K> other) {
           return (this.key == null) ? -1 :
               this.key.compareTo(other.key);
     }
          ...


 }
```

Binärer Suchbaum: Suchen

```
protected  TreeNode<K>  findNode(TreeNode<K> k){
    /* k wird gesucht */

    TreeNode<K> current = root;

    while(current != null){
        if(current.compareTo(k) == 0)
            return current;
        else if(current.compareTo(k) > 0)
            current = current.getLeft();
        else current = current.getRight();
    }

    return null;
}
```

Binärer Suchbaum: Einfügen

- Einfügen prinzipiell in 2 Schritten
  1. Schritt: Einfügeposition suchen
     - ➔ Blattknoten mit nächstkleinerem oder nächstgrößerem Schlüssel
  2. Schritt: neuen Knoten erzeugen und als Kindknoten des Knoten aus Schritt 1 verlinken

- In Schritt 1: wenn Schlüssel bereits existiert, nicht erneut einfügen

Binärer Suchbaum: Einfügen

```
public boolean insert(K k){
    TreeNode<K> newNode = new TreeNode<K>(k);

    TreeNode<K> parent = null;
    TreeNode<K> child = root;
    while(child != null) {
        parent = child;
        int cmp = child.getKey().compareTo(newNode);
        if (cmp == 0)
            return false;  // Knoten bereits vorhanden
        else if (cmp > 0)
            child = child.getLeft();
        else
            child = child.getRight();
    }

    ...
```

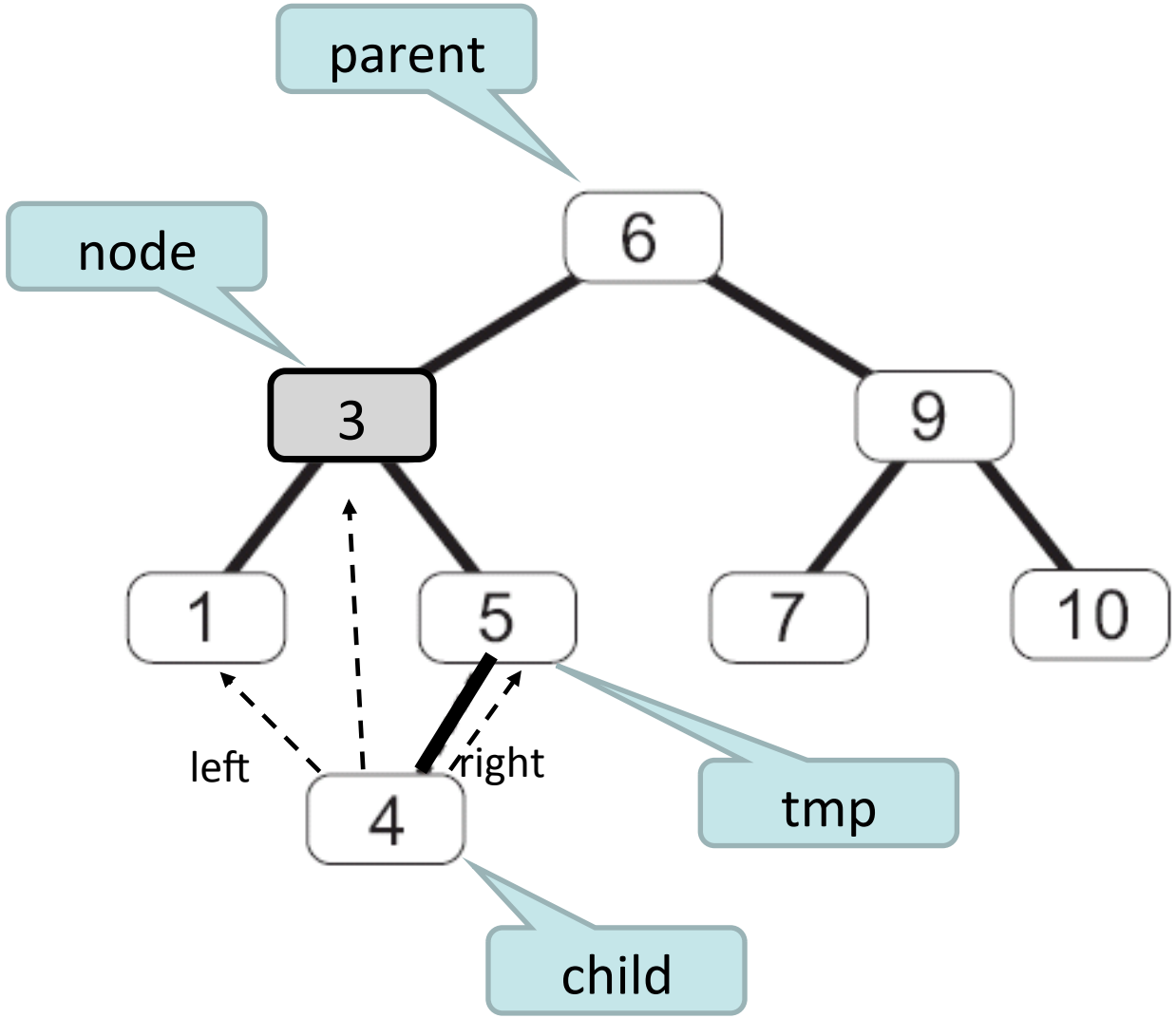
Binärer Suchbaum: Einfügen (2)

```
    ...

    // Spezialfall leerer Baum
    if(parent == null){
        root = newNode;
        return true;
    }

    // Allgemeiner Fall
    // Hänge newNode an korrekte Position an
    if(parent.compareTo(newNode) > 0)
        parent.setLeft(newNode);
    else parent.setRight(newNode);

    return true;
}
```

Binärer Suchbaum: Löschen

- Schlüssel löschen in 3 Schritten
  1. Schritt: zu löschenden Knoten finden
  2. Schritt: Nachrückerknoten finden
     - Fall 1: externer Knoten (=Blatt) ohne Kinder
       - → Ersetzen durch nullNode
     - Fall 2a: nur ein rechter Kindknoten
       - → Ersetzen durch rechten Kindknoten
     - Fall 2b: nur ein linker Kindknoten
       - → Ersetzen durch linken Kindknoten
     - Fall 3: interner Knoten mit Kindern rechts und links
       - → Ersetzen durch Knoten mit kleinstem (alternativ größtem) Schlüssel im rechten (alternativ linken) Teilbaum
  3. Schritt: Baum reorganisieren

Beispiel: Löschen im binären Suchbaum



Löschen 1: Knoten suchen

```
public boolean remove(K k){
    TreeNode<K> parent = null;
    TreeNode<K> node = root;
    TreeNode<K> child = null;
    TreeNode<K> tmp = null;

    while(node != null) {
        int cmp = node.compareTo(k);
        if (cmp == 0)
            break;
        else {
            parent = node;
            node = (cmp > 0 ? node.getLeft() : node.getRight());
        }
    }
    if (node == null)
        return false;
      ...
```

Löschposition gefunden

Knoten k nicht im Baum

Löschen 2: Nachrücker finden (1)

```
...

if (node.getLeft() == null  && node.getRight() == null)
    child = null;
else if (node.getLeft() == null)
    child = node.getRight();
else if (node.getRight() == null)
    child = node.getLeft();

...
```

Fall 1

Fall 2a

Fall 2b

Löschen 2: Nachrücker finden (2)

Fall 3

```
...
else {
    child = node.getRight();
    tmp = node;
    while (child.getLeft() != null) {
        tmp = child;
        child = child.getLeft();
    }
    child.setLeft(node.getLeft());
    if (tmp != node) {
        tmp.setLeft(child.getRight());
        child.setRight(node.getRight());
    }
}
...
```

aus dem rechten Teilbaum

… den kleinsten Knoten entnehmen

linken Teilbaum von node einhängen

größeren Knoten von child links in tmp einhängen

tmp != node, d.h. nur wenn child kein direkter Nachfolger von Knoten k ist

rechten Teilbaum von node einhängen

Löschen 3: Baum reorganisieren

```
    ...

    //Spezialfall: zu löschender Knoten ist Wurzel
    if(node == root){
        root = child;
        return true;
    }

    // Allgemeiner Fall
    if (parent.getLeft() == node)
        parent.setLeft(child)
    else
        parent.setRight(child);
    return true;
}
```

Verbindung vom parent Knoten zum child Knoten herstellen

Entartung von Bäumen

- Ungünstige Einfüge- oder Löschreihenfolge führt zu extremer Unbalanciertheit
- Extremfall: Baum wird zur Liste
- Dann: Operationen mit Komplexität O(n)
- Beispiel:

```
for(int i = 0; i < 10; i++)
    tree.insert(i);
```

- Vermeidung: durch spezielle Algorithmen zum Einfügen und Löschen
  - z.B. mit Rot-Schwarz-Bäume, AVL-Bäume

Algorithmen und Datenstrukturen

7.1 BINÄRE SUCHBÄUME

ZUSAMMENFASSUNG

Zusammenfassung

- Binäre Suchbäume
  - Operationen
    - Suchen von Elementen
    - Einfügen von Elementen
    - Löschen von Elementen
  - Implementierung

Algorithmen und Datenstrukturen

7.2. AVL-BÄUME

Erinnerung

- Komplexität von Operationen auf Bäumen
  - Hängt von der Höhe ab!
    - O(h) für Baum der Höhe h
- Höhe eines **ausgeglichenen** Baums:
  - ➔ h = ld n für n Knoten

Erinnerung:
ld n = $\log_2$ n

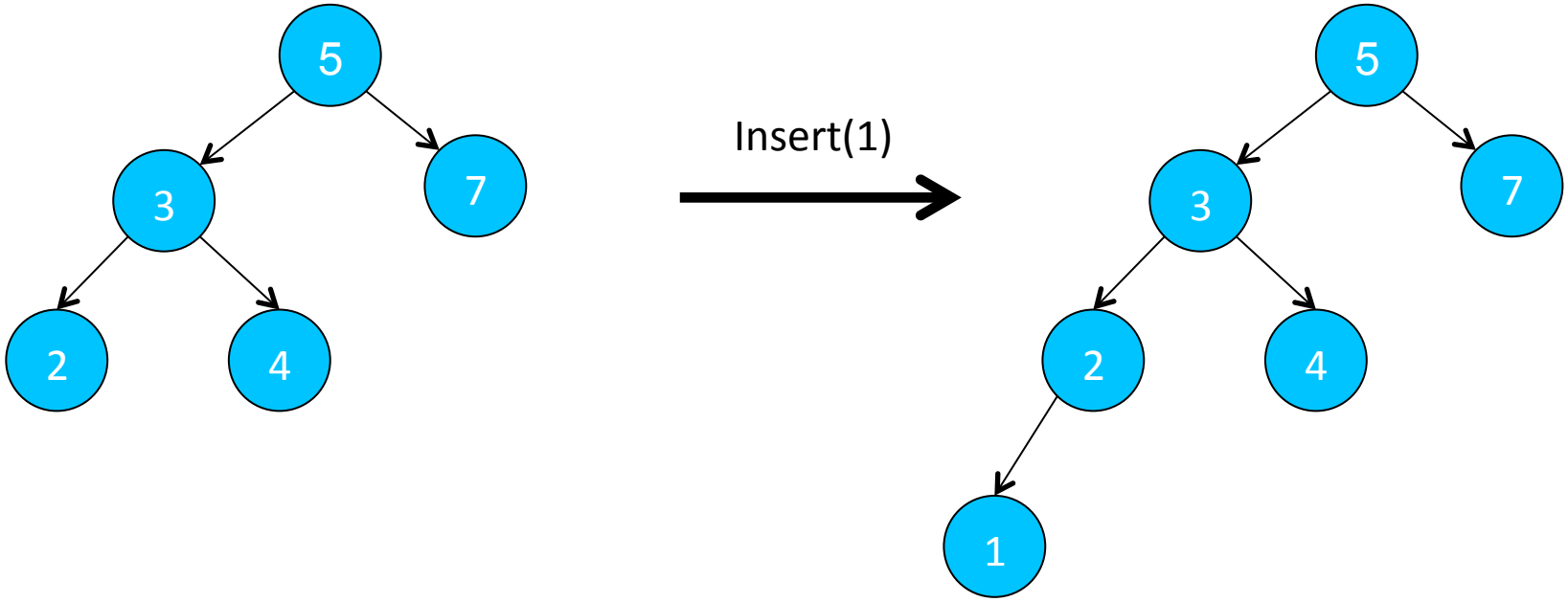
Ausgeglichener oder balancierter Baum:

1. Rechter und linker Teilbaum eines jeden Knotens unterscheiden sich in der Höhe höchstens um 1 und
2. Je zwei Wege von der Wurzel zu einem Blattknoten unterscheiden sich in der Länge um höchstens 1

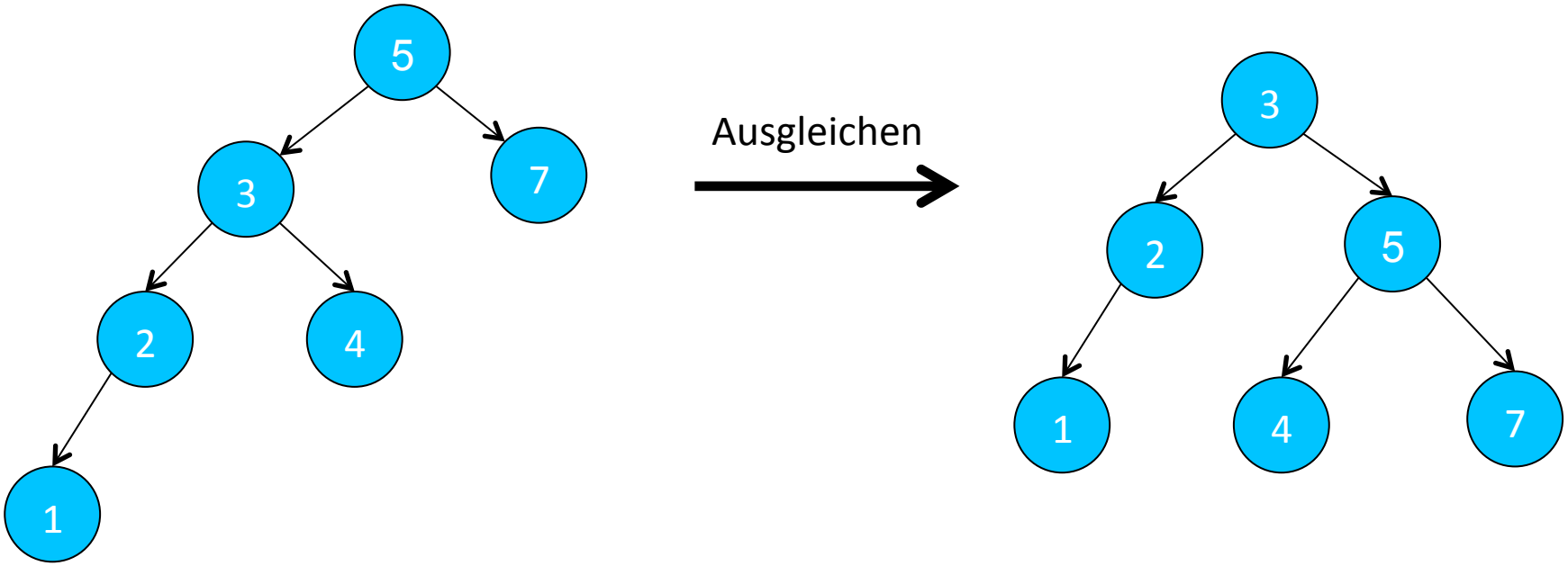
- Erfordert Ausgleichen nach Einfügen und Löschen

Ausgeglichene Bäume

- Wie verhindert man, dass Suchbäume entarten?
- Möglichkeit: jeweils ausgleichen?



Ausgeglichene Bäume (2)



Lösungsidee

1. Abgeschwächtes Kriterium für ausgeglichene Höhe
   - Beispiel: AVL Bäume
2. Ausgeglichene Höhe, aber unausgeglichener Verzweigungsgrad
   - Direkte Realisierung als Mehrwegbäume
     - Beispiel: B-Bäume
   - Kodierung als binäre Bäume
     - Beispiel: Rot-Schwarz-Bäume

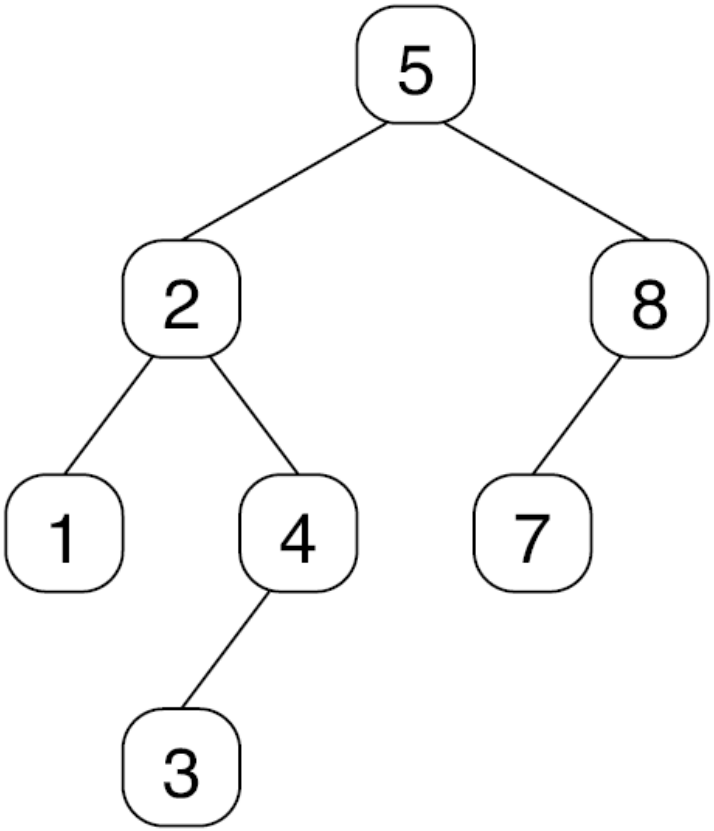
AVL-Bäume

- AVL für Adelson-Velskii und Landis (russische Mathematiker)
- Binäre Suchbäume mit AVL-Kriterium:

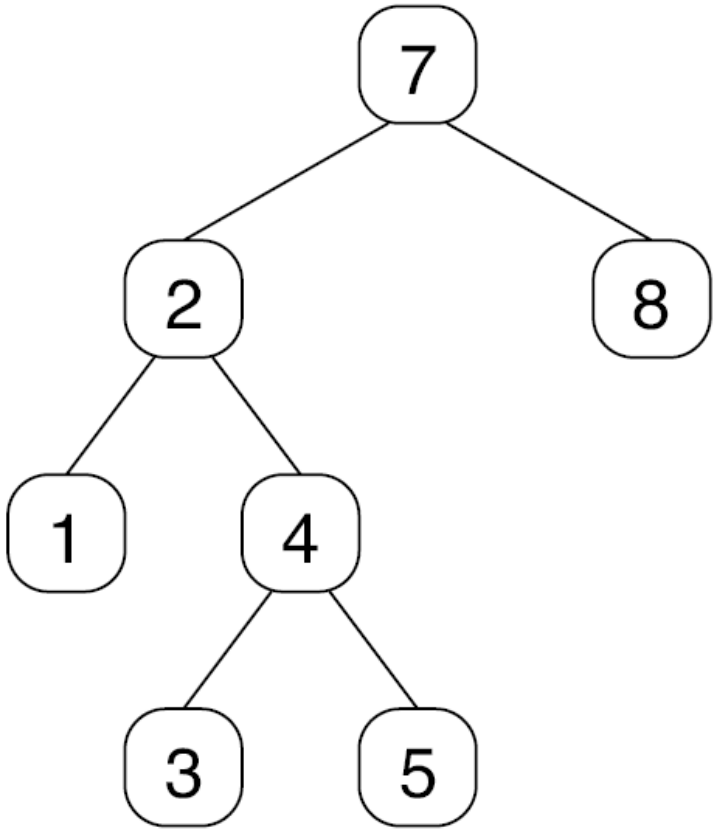
> für jeden (inneren) Knoten gilt: Höhe des linken und rechten Teilbaums differieren maximal um 1

- Bemerkung: es reicht nicht dieses nur für die Wurzel zu fordern!
  - Beide Teilbäume der Wurzel könnten entartet sein

AVL-Eigenschaft am Beispiel



AVL



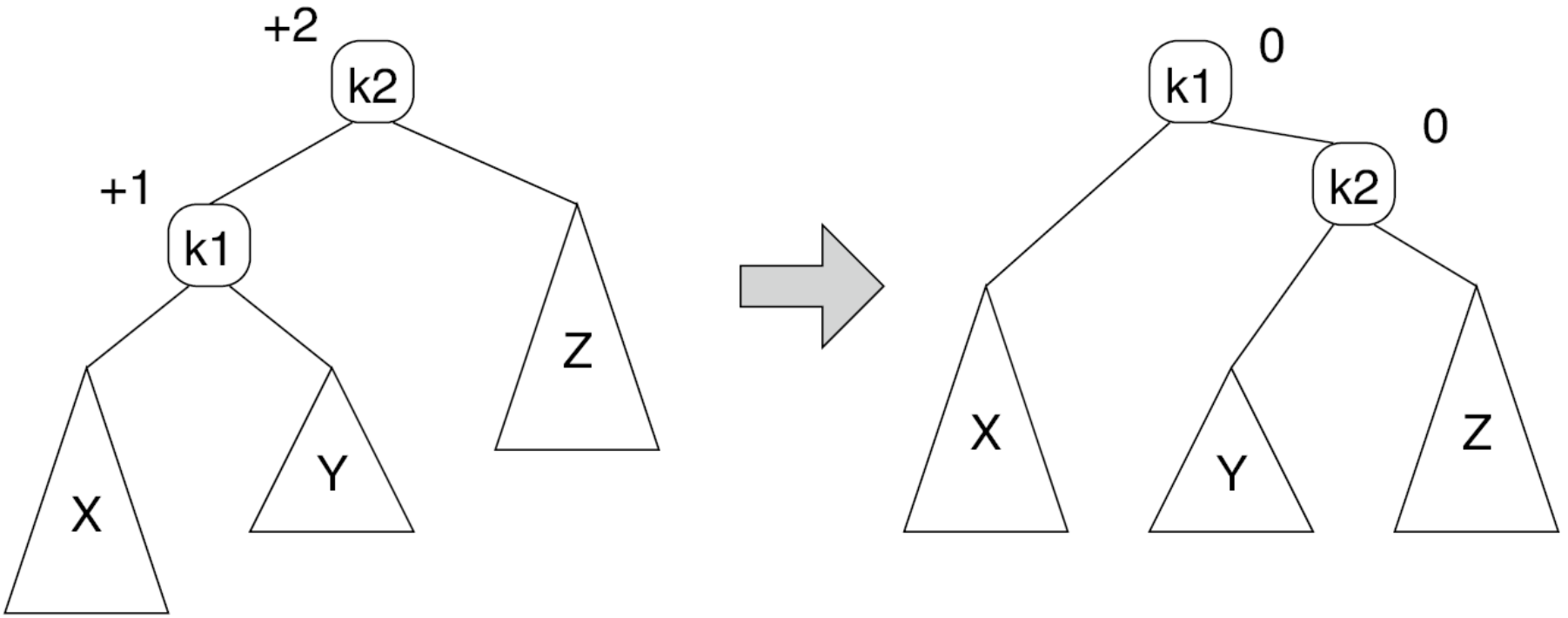
nicht AVL

Fallunterscheidung beim Einfügen

- Verletzung der AVL-Eigenschaft tritt ein bei
  1. Einfügen in linken Teilbaum des linken Kindes
  2. Einfügen in rechten Teilbaum des linken Kindes
  3. Einfügen in linken Teilbaum des rechten Kindes
  4. Einfügen in rechten Teilbaum des rechten Kindes

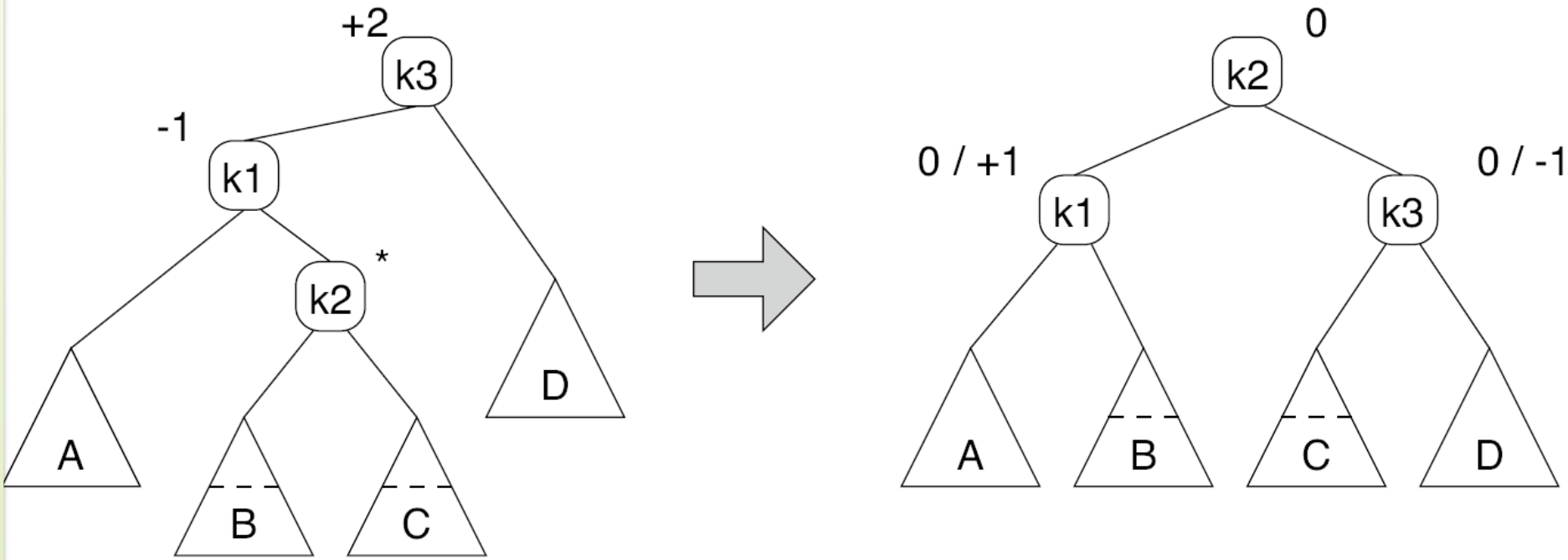
1 und 4 sowie 2 und 3 sind symmetrische Problemfälle

Einfache Rotation



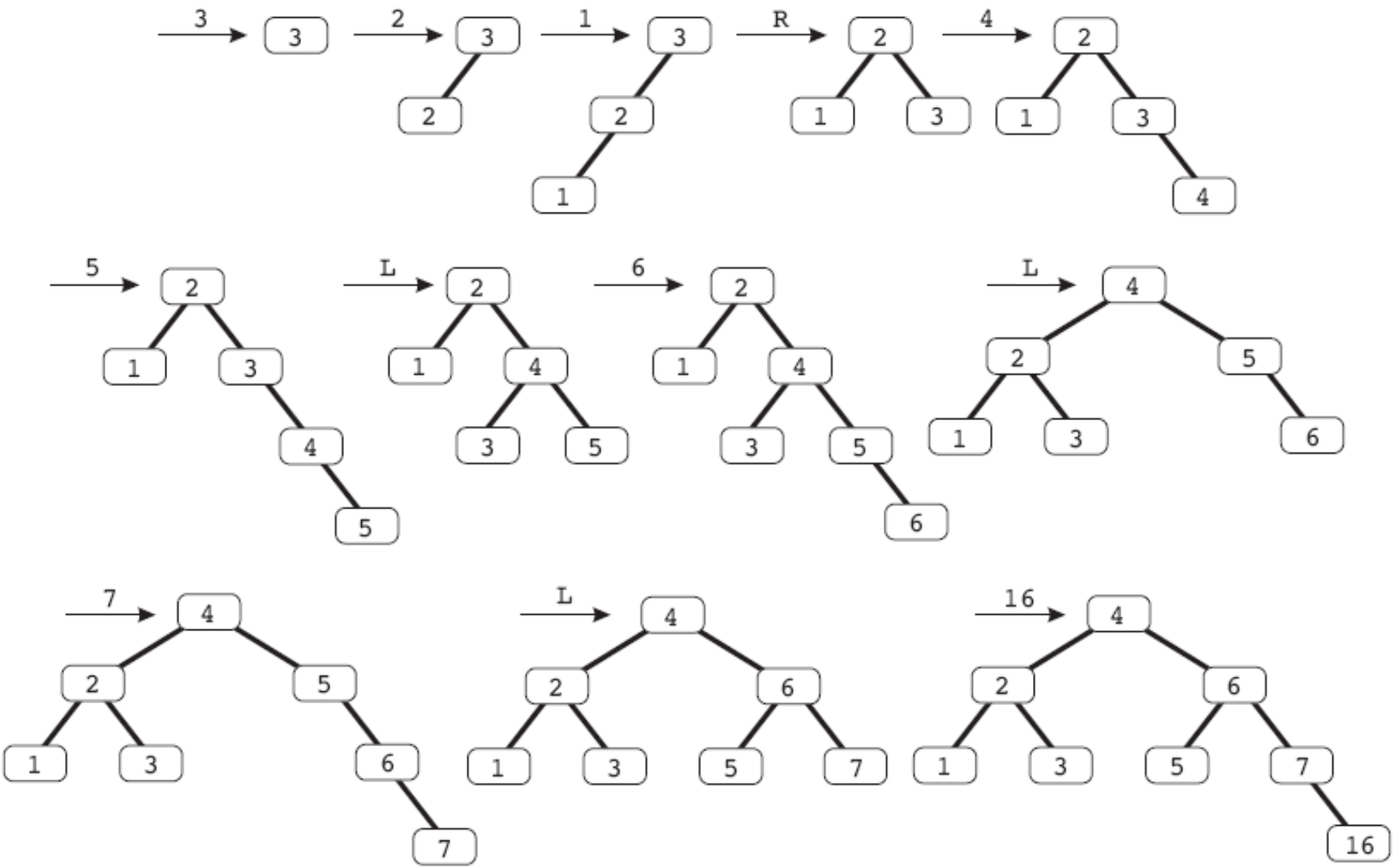
- Rotation mit linkem Kind nach rechts
  - Analog für das spiegelbildliche Problem von rechts nach links

Doppelrotation

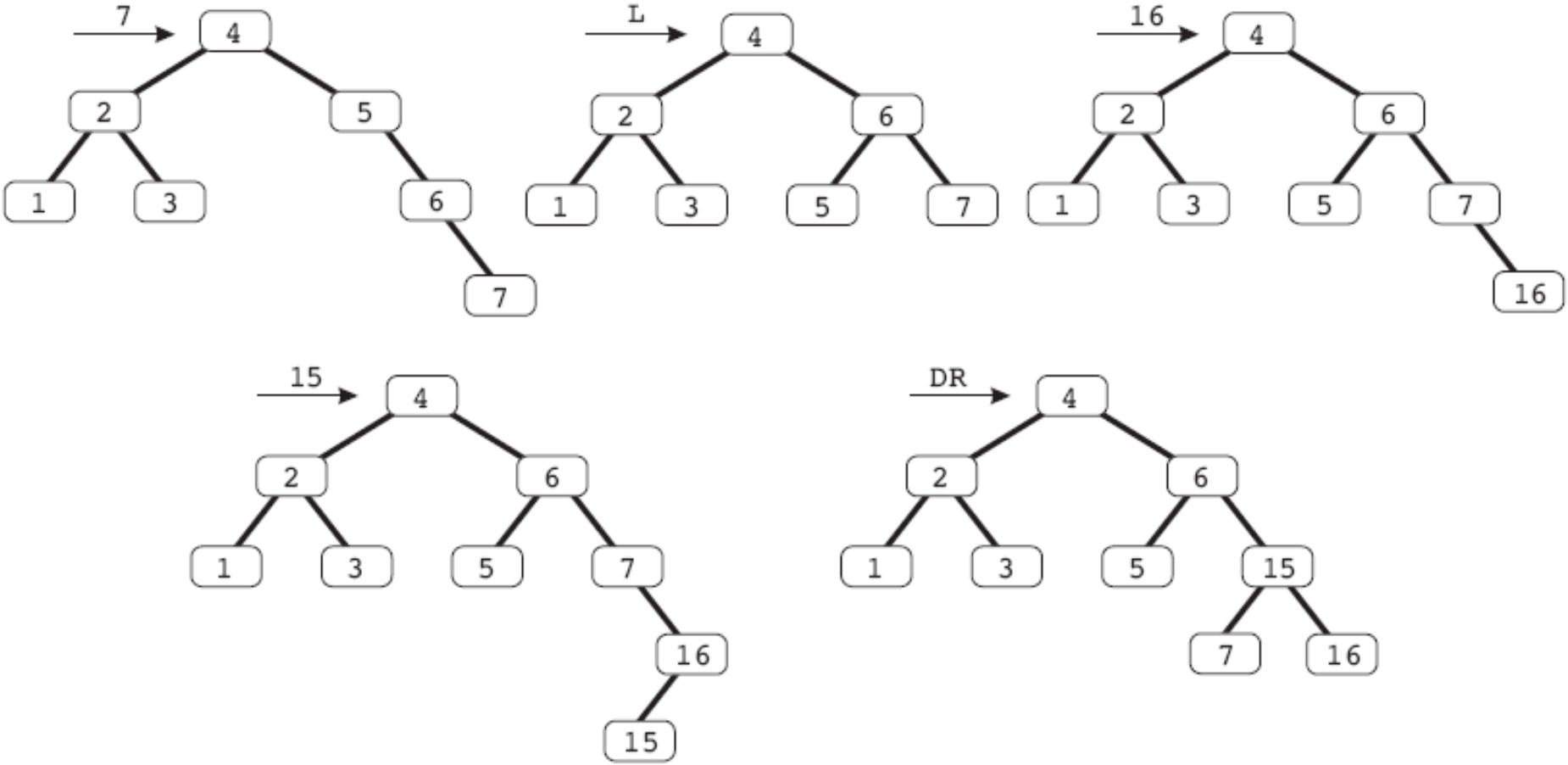


- Doppelrotation mit linkem Kind nach rechts
  - Analog für das spiegelbildliche Problem nach links

Beispiel



Beispiel


7
4
2
5
1
3
6
7
L
4
2
6
1
3
5
7
16
4
2
6
1
3
5
7
16
15
4
2
6
1
3
5
7
16
15
DR
4
2
6
1
3
5
15
7
16

Rotationen in Einzelfällen

- Verletzung der AVL-Eigenschaft tritt ein bei
  - Einfügen in linken Teilbaum des linken Kindes
    - Rotation mit linkem Kind
  - Einfügen in rechten Teilbaum des linken Kindes
    - Doppelrotation mit linkem Kind
  - Einfügen in linken Teilbaum des rechten Kindes
    - Doppelrotation mit rechtem Kind
  - Einfügen in rechten Teilbaum des rechten Kindes
    - Rotation mit rechtem Kind

Implementation

Baumimplementierung ähnlich wie beim binären Suchbaum:

```
public class AvlTree<K extends Comparable<K>> {

    private AvlTreeNode<K> root;

    public AvlTree(){
        root = null;
    }

    ...
}
```

Neuer Knotentyp

Implementation

Knotenimplementierung:

```
class AvlTreeNode<K extends Comparable<K>>{
     K key;
     private AvlTreeNode<K> left;
     private AvlTreeNode<K> right;
     private AvlTreeNode<K> parent;

     public AvlTreeNode(K key){
          this.left = null;
          this.right = null;
          this.parent = null;
          this.key = key;
     }

     public AvlTreeNode<K> getLeft() { return left; }
     public AvlTreeNode<K> getRight()  { return right; }
     public K getKey() { return key; }
     public void setLeft(AvlTreeNode<K> n) {left = n;}
     public void setRight(AvlTreeNode<K> n) {right = n;}
     public void setParent(AvlTreeNode<K> n) {parent = n;}

     public int compareTo(AvlTreeNode<K> other){
          return (this.key == null) ? -1 : this.key.compareTo(other.key);
     }

     ...
```

Für jeden Knoten merken wir uns nun den Elternknoten

Wie beim binären Suchbaum

Implementation

Knotenimplementierung (Hilfsmethoden):

```
...

public int balance(){
    int leftHeight = this.left == null ? 0 : this.left.height();
    int rightHeight = this.right == null ? 0 : this.right.height();
    return leftHeight - this.rightHeight;
}

public int height(){
    int leftHeight = this.left == null ? 0 : this.left.height();
    int rightHeight = this. right == null ? 0 : this.right.height();
    return Math.max(leftHeight,rightHeight) + 1;
}

...
```

Positive Balance, falls linker Teilbaum größer als rechter Teilbaum
Negativ, falls rechter Teilbaum größer als linker Teilbaum
0, falls ausgeglichen

Länge des längsten Pfades zu einem Blatt

Implementation

Knotenimplementie ce):

Rückgabe ist neuer Wurzelknoten (könnte sich ändern)

Teste, ob die Balance für den aktuellen Knoten (this) verletzt ist und repariere gegebenenfalls

Die nächsten zwei nachfolgenden Knoten auf dem Pfad der Einfügung; wir arbeiten uns vom Punkt der Einfügung (ein Blatt) von unten nach oben vor

```
public AvlTreeNode<K> repair(AvlTreeNode<K> child, AvlTreeNode<K> grandchild){
     AvlTreeNode<K> newRoot = this;
     AvlTreeNode<K> newRootChild = child;
     if(this.balance() > 1 && child.balance() > 0){
          child.parent = this.parent;
          this.left = child.right;
          if(this.left != null)
               this.left.parent = this;
          child.right = this;
          this.parent = child;
          if(child.parent != null)
               if(child.parent.left == this)
                    child.parent.left = child;
               else child.parent.right = child;
          newRoot = child;
          newRootChild = grandchild;
     }

     ...
```

Fall 1:
Einfügung im linken Teilbaum des linken Kindes

Implementation

Knotenimplementierung (Reparatur der Balance):

```
        ...
        if(this.balance() > 1 && child.balance() < 0){
             grandchild.parent = this.parent;
             this.left = grandchild.right;
             if(this.left != null) this.left.parent = this;
             grandchild.right = this;
             this.parent = grandchild;
             child.right = grandchild.left;
             if(child.right != null) child.right.parent = child;
             grandchild.left = child;
             child.parent = grandchild;
             if(grandchild.parent != null)
                  if(grandchild.parent.left == this)
                       grandchild.parent.left = grandchild;
                  else grandchild.parent.right = grandchild;
             newRoot = grandchild;
             newRootChild = child;
        }

        ...

        if(newRoot.parent != null)
             return newRoot.parent.repair(newRoot, newRootChild);
        return newRoot;
   }
```

Fall 2:
Einfügung im rechtenTeilbaum des linken Kindes

Fall 3/4 analog

Fahre rekursiv mit Elternknoten fort (falls existent)

Implementation

Aufruf in AvlTree.insert(K k):

```
public boolean insert(K k){
     AvlTreeNode<K> parent = null;
     AvlTreeNode<K> child = root;
     while (child != null) {
            parent = child;
            int cmp = child.getKey().compareTo(k);
            if (cmp == 0)
                return false;
            else if (cmp > 0)
                child = child.getLeft();
            else
                child = child.getRight();
     }
     AvlTreeNode<K> node = new AvlTreeNode<K>(k);

     // Spezialfall leerer Baum
     if(parent == null){
          root = node;
          return true;
     }

     ...
```

Methode nahezu identisch mit der Version des binären Suchbaums

Implementation

Aufruf in AvlTree.insert(K k):

```
        ...

        // Allgemeiner Fall
        // Hänge node an korrekte Position an
        node.setParent(parent);
        if (parent.compareTo(node) > 0)
              parent.setLeft(node);
        else
              parent.setRight(node);

        this.root = parent.repair(node, null);

        return true;
}
```

Nach der Einfügung Balance ggfs. reparieren und Wurzelknoten aktualisieren

Algorithmen und Datenstrukturen

7.2 AVL-BÄUME
ZUSAMMENFASSUNG

Zusammenfassung

- AVL-Kriterium:
  - für jeden (inneren) Knoten gilt: Höhe des linken und rechten Teilbaums differieren maximal um 1
- Wiederherstellung des AVL-Kriteriums nach Einfüge- und Löschoperationen durch Rotationen

Algorithmen und Datenstrukturen

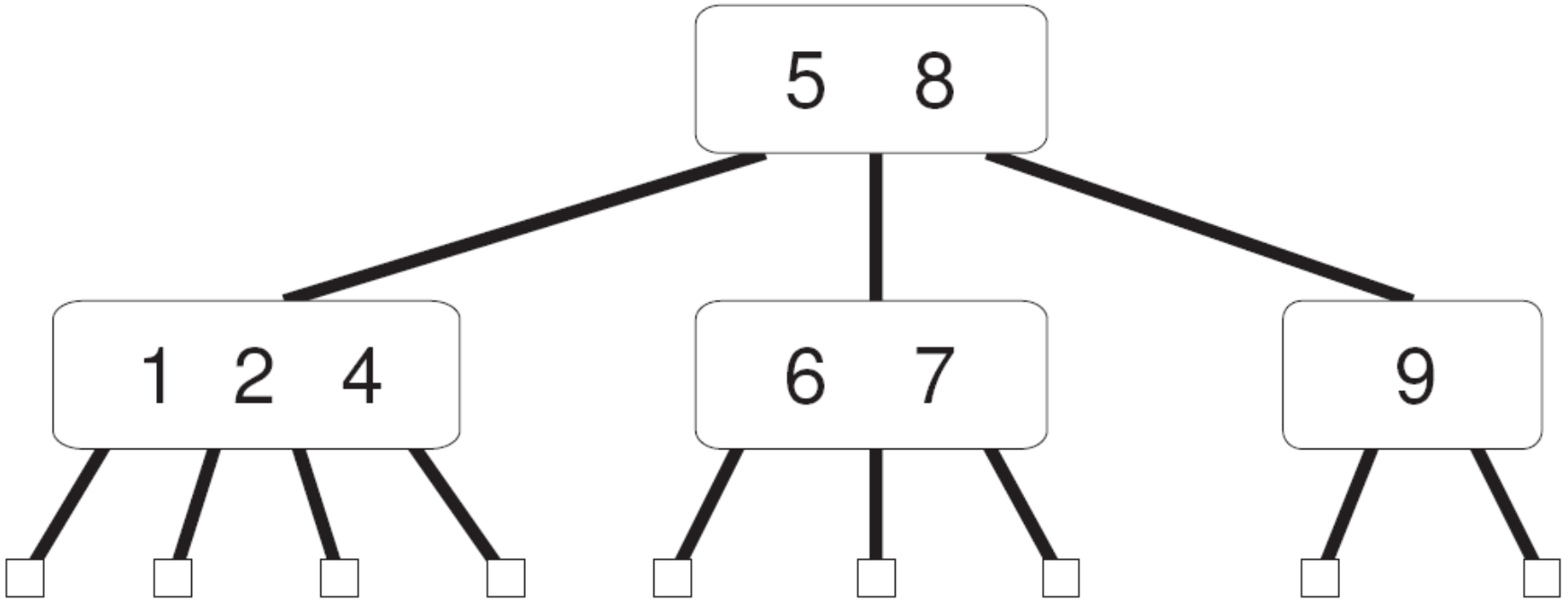
7.3. B-BÄUME

B-Bäume

- Idee: perfekt ausgeglichene Bäume mit variablem Verzweigungsgrad und variabler Anzahl Schlüssel
- Ausgeglichenheit wird bei der Einfügeoperation gewährleistet

- Parameter d (auch *Ordnung* genannt) gibt maximale Anzahl Kinder eines Knotens an
- Jeder Knoten mit K Kindern enthält K-1 Schlüssel

B-Baum der Ordnung d=4



neben binären Knoten (2-Knoten) auch 3-Knoten und 4-Knoten

B-Bäume der Ordnung d=4 heißen auch 2-3-4-Bäume

B-Baum: Definition 1/2

Sei d>3. Ein B-Baum T der Ordnung d ist ein Baum mit den folgenden Eigenschaften:

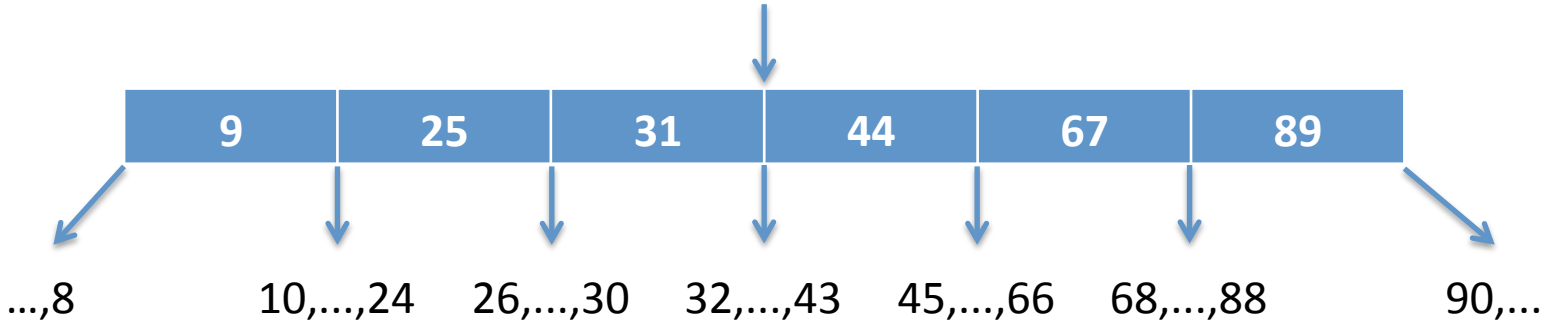
1. Jeder Knoten hat maximal d Kinder
2. Jeder **innere** Knoten hat wenigstens d/2 Kinder
3. Der Wurzelknoten hat wenigstens 2 Kinder (ausser der Baum ist leer)
4. Für jeden Knoten (ausser Blattknoten) gilt: hat der Knoten D Kinder, so enthält er (D-1) Schlüssel
5. Der Baum T ist perfekt ausgeglichen (alle Blätter sind auf derselben Ebene)

B-Baum: Definition 2/2

Sei d>3 und T ein B-Baum der Ordnung d.

T ist *aufsteigend sortiert*, wenn für jeden Knoten N mit Schlüsseln N_1, …,N_D gilt:

1. $N_1<...<N_D$
2. Seien $N'_1,...,N'_{D+1}$ die Kinder von N (falls N kein Blatt ist). Dann gilt:
   1. Alle Schlüssel in N'_1 sind kleiner als N_1
   2. Alle Schlüssel in N'_{D+1} sind größer als N_D
   3. Für alle i=2,...,D gilt: Alle Schlüssel in N'_i sind größer als N_{i-1} und kleiner als N_i



Implementation: B-Baumknoten 1

```
class BTreeNode<K extends Comparable<K>>{

     private int numKeys = 0;
     private int order;
     private List<K> keys;
     private List<BTreeNode<K>> children;
     private BTreeNode<K> parent;


     public BTreeNode(int numKeys, int order){
          this.numKeys = numKeys;
          this.order = order;
          this.keys = new ArrayList<K>();
          for(int i = 0; i < this.numKeys; i++)
               this.keys.add(null);
          this.children = new ArrayList<BTreeNode<K>>();
          for(int i = 0; i < this.numKeys+1; i++)
               this.children.add(null);
     }

     public K setKey(int idx, K key){
          if(idx < this.numKeys)
               return this.keys.set(idx, key);
          return null;
     }

     ...
```

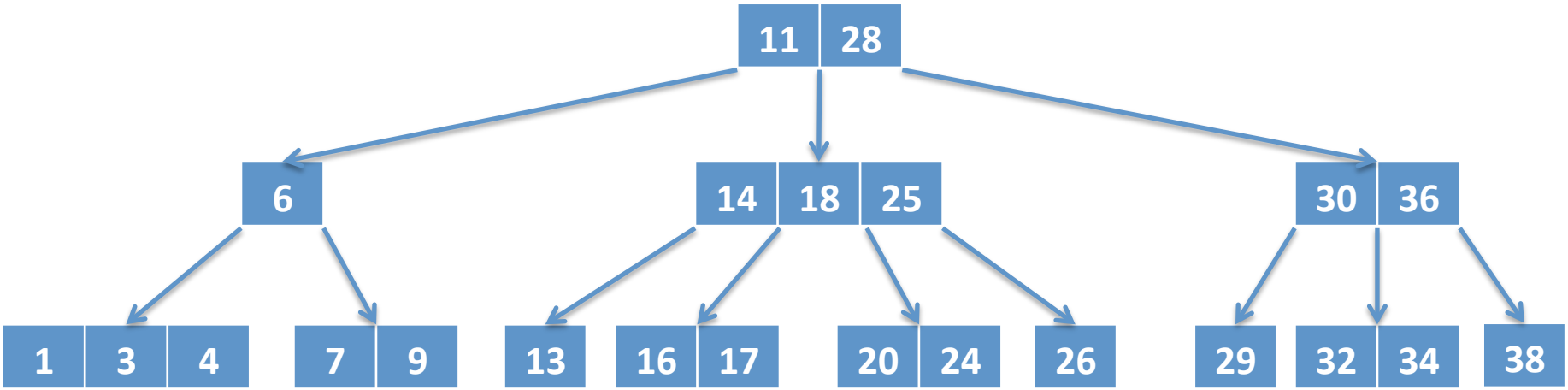
Implementation: B-Baumknoten 2

```
class BTreeNode<K extends Comparable<K>>{

     ...

     public void setParent(BTreeNode<K> parent){
          this.parent = parent;
     }

     public BTreeNode<K> setChild(int idx, BTreeNode<K> child){
          if(idx < this.numKeys+1)
               return this.children.set(idx, child);
          return null;
     }

     public BTreeNode<K> getChild(int idx){
          if(idx < this.numKeys+1)
               return this.children.get(idx);
          return null;
     }

     public boolean isLeaf(){
          return this.children.get(0) == null;
     }
     ...

}
```

Implementation: B-Baum 1

```
public class BTree<K extends Comparable<K>> {

     private BTreeNode<K> root;
     private int order;


     public BTree(int order){
          this.order = order;
     }

     ...

}
```

Suche in B-Bäumen



Suche ähnlich wie bei binären Suchbäumen

1. Aktueller Knoten ist Wurzelknoten
2. Enthält der aktuelle Knoten den gesuchten Schlüssel?
   a. Falls ja: fertig
   b. Falls nein: aktueller Knoten wird Kind im passenden Intervall

Schritt 2 kann durch binäre Suche realisiert werden

Implementation: B-Baumknoten 3

```
class BTreeNode<K extends Comparable<K>>{

     ...
     private int numKeys = 0;
     private List<K> keys;
     ...

     public int indexOf(K key){
          int l = 0;
          int r = this.numKeys-1;
          int c;
          while(l <= r){
               c = (int) Math.floor((l+r)/2);
               int cmp = this.keys.get(c).compareTo(key);
               if(cmp < 0 )
                    l = c+1;
               else if(cmp > 0)
                    r = c-1;
               else return c;
          }
          return l;
     }

     ...

}
```

Gibt den Index des Schlüssels zurück (wenn er enthalten ist) oder
den Index des nächstgrößeren Elements
(kann dann auch rechts außerhalb der Liste sein)

Implementation: B-Baumknoten 4

```
class BTreeNode<K extends Comparable<K>>{

     ...

     public boolean containsKey(K key){
          int idx = this.indexOf(key);
          return idx < this.numKeys &&
               this.keys.get(this.indexOf(key)).compareTo(key) == 0;
     }

     ...

}
```

Implementation: B-Baum 2

```
public class BTree<K extends Comparable<K>> {

     ...

     public boolean containsKey(K key){
          if(this.root == null)
               return false;
          BTreeNode<K> node = this.root;
          while(!node.isLeaf()){
               if(node.containsKey(key))
                    return true;
               node = node.getChild(node.indexOf(key));
          }
          return node.containsKey(key);
     }

     ...

}
```

Einfügen in B-Bäumen

- Erfolglose Suche liefert Blattknoten b
- Ist b ein Knoten mit weniger als (d-1) Schlüsseln: Einfügen!
- Ist b ein Knoten mit (d-1) Schlüsseln: Aufteilen (split!), mittleres Element nach oben ziehen
- Splitten kann sich bis zur Wurzel fortpflanzen (bottom-up)!

Implementation: B-Baumknoten 5

```
class BTreeNode<K extends Comparable<K>>{

     ...

     public BTreeNode<K> getRoot(){
          if(this.parent == null)
               return this;
          return this.parent.getRoot();
     }

     private int getLevel(){
          if(this.parent == null)
               return 0;
          return this.parent.getLevel()+1;
     }

     ...

}
```

Einige Hilfsmethoden

Implementation: B-Baumknoten 6

```
class BTreeNode<K extends Comparable<K>>{

     ...

     public BTreeNode<K> insert(K key){
          return insert(key,null,null);
     }

     public BTreeNode<K> insert(K key, BTreeNode<K> lo, BTreeNode<K> hi){
          if(this.indexOf(key) >= this.keys.size())
               this.keys.add(key);
          else
               this.keys.add(this.indexOf(key), key);
          this.numKeys++;
          if(!this.isLeaf()){
               this.children.set(this.indexOf(key), hi);
               this.children.add(this.indexOf(key), lo);
               lo.parent = this;
               hi.parent = this;
          }
          if(this.numKeys <= this.order-1)
               return this.getRoot();

          ...
```

Implementation: B-Baumknoten 7

```
public BTreeNode<K> insert(K key, BTreeNode<K> lo, BTreeNode<K> hi){
     ...
     // split
     BTreeNode<K> left = new BTreeNode<K>
          ((int) Math.ceil(new Double(this.numKeys-1)/2),this.order);
     BTreeNode<K> right = new BTreeNode<K>
          ((int) Math.floor(new Double(this.numKeys-1)/2),this.order);
     int i;
     for(i = 0; i < Math.ceil(new Double(this.numKeys-1)/2); i++){
          left.setKey(i, this.keys.get(i));
          if(!this.isLeaf())
               left.setChild(i, this.children.get(i));
     }
     if(!this.isLeaf())
          left.setChild(i, this.children.get(i));
     K median = median = this.keys.get(i);
     i++;
     int j;
     for(j = 0; j < Math.floor(new Double(this.numKeys-1)/2);j++){
          right.setKey(j, this.keys.get(i+j));
          if(!this.isLeaf())
               right.setChild(j, this.children.get(i+j));
     }
     if(!this.isLeaf())
          right.setChild(j, this.children.get(i+j));

     ...
```

Implementation: B-Baumknoten 8

```
        public BTreeNode<K> insert(K key, BTreeNode<K> lo, BTreeNode<K> hi){
            ...

            //Spezialfall Knoten = Wurzel
            if(this.parent == null){
                BTreeNode<K> newRoot = new BTreeNode<K>(1,this.order);
                newRoot.setParent(null);
                newRoot.setKey(0, median);
                newRoot.setChild(0, left);
                newRoot.setChild(1, right);
                left.parent = newRoot;
                right.parent = newRoot;
                return newRoot;
            }
            // Fahre rekursiv fort
            return this.parent.insert(median, left, right);
        }

        ...
    }
```

Implementation: B-Baum 3

```
public class BTree<K extends Comparable<K>> {

     ...

     public boolean insert(K key){
          // Spezialfall: leerer Baum
          if(this.root == null){
               this.root = new BTreeNode<K>(1,this.order);
               this.root.setParent(null);
               this.root.setKey(0, key);
               return true;
          }
          BTreeNode<K> node = root;
          while(!node.isLeaf()){
               if(node.containsKey(key))
                    return false;
          node = node.getChild(node.indexOf(key));
          }
          this.root = node.insert(key);
          return true;
     }
     ...

}
```

Algorithmen und Datenstrukturen

7.3 B-BÄUME
ZUSAMMENFASSUNG

Zusammenfassung

- B-Bäume
  - Knoten mit variabler Anzahl Werte und variabler Anzahl Kinder
  - Einfügeoperation einfach möglich durch Splitten von zu vollen Knoten