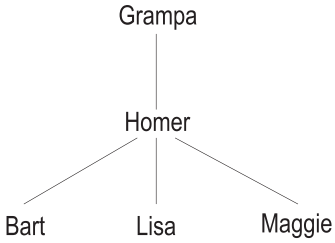
Teil VI

Bäume

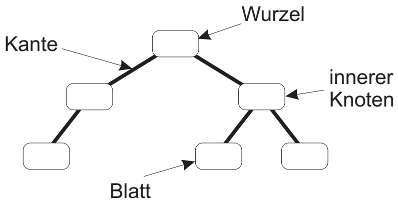
Überblick

Bäume

- bisherige Datenstrukturen: eindimensional bzw. linear
- oft aber hierarchische bzw. mehrdimensionale Strukturen notwendig
- Bäume sind weit verbreitete Datenstruktur zur hierarchischen Organisation von Daten / Informationen.
- Beispiele: Stammbäume, Dateibäume, Syntaxbäume, Entscheidungsbäume, Suchbäume

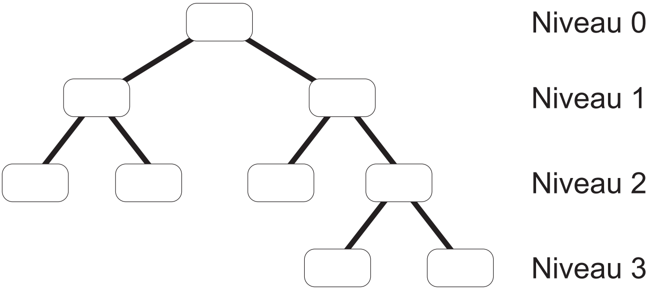


Begriffe im Baum



- Pfad = Folge von Knoten, die jeweils mit einer Kante verbunden sind.
- Def. 1: Baum = Graph, in dem zu jedem Knoten genau ein Pfad von der Wurzel existiert.
- Def. 2 (alternativ): Baum = zusammenhängender, zyklenfreier, gerichteter Graph.
- Folgerungen:
  - Wurzel = einziger Knoten ohne Vorgänger.
  - Alle übrigen Knoten haben genau einen Vorgänger.

Niveau und Höhe



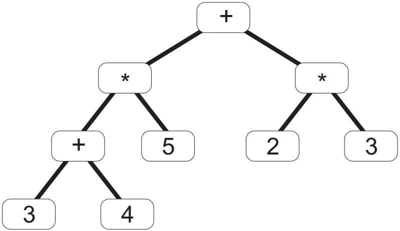
- Höhe := größtes Niveau + 1

Baum aus Term

Term

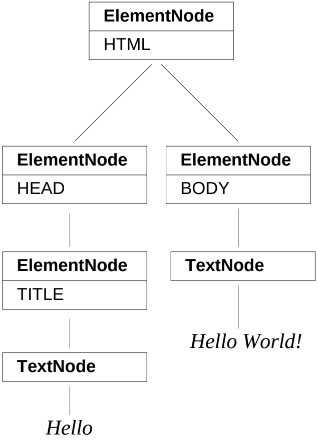
$$(3+4)*5+2*3$$

in Baumdarstellung:



Alternativen: Mehrere Knotentypen

```
<HTML>
   <HEAD>
      <TITLE>Hello
      </TITLE>
   </HEAD>
   <BODY>
      Hello World!
   </BODY>
</HTML>
```



Alternativen: Mehrere Knotentypen /2

- Verschiedene Knotentypen sinnvoll z.B.
  - bei Syntaxbäumen für Statements, Blöcke, Operanden, Operatoren etc.
  - bei hierarchischen Dokumentenstrukturen für verschiedene Dokumentenabschnitte wie Kapitel, Überschriften, Paragraphen, eingebettete Grafiken etc.
  - bei Dateiverwaltung für Laufwerke, Verzeichnis, Dateien, Verknüpfungen etc.
  - bei unterschiedlicher Struktur von internen Knoten (Verzweigung zu Kindknoten) und Blattknoten (Daten)

Implementierung von Bäumen

- Allgemein: Datenstruktur bestehend aus Baum, Baumknoten und den vom Baum organisierten Datenobjekten
- Unterscheidungen nach
  - Verzweigungsgrad des Baumes
  - geordnetem oder ungeordnetem Baum
  - Anzahl Knotentypen
  - Suchbäume
  - Verzeigerung der Daten
  - Vorgehensweise für Einfügen, Löschen, etc.

Binärer Baum in Java /1

- Baum für Elemente eines bestimmten Typs T

```
public class Tree<T> {
    TreeNode<T> root;
    ...
}
```

- Objekte des Typs T sind über Knoten verwaltet

```
private class TreeNode<T> {
    ...// siehe nächste Folie
}
```

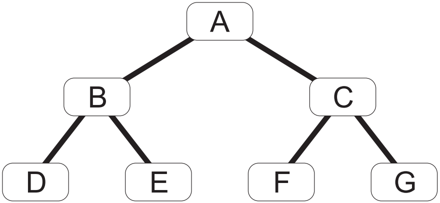
Wie bei Listen können die Knoten **`static`** oder **`private`** sein, je nach dem, ob sie ausserhalb des Baums sichtbar sein sollen oder nicht.

Binärer Baum in Java /2

```
static class TreeNode<T> {

   T element;
   TreeNode<T> left = null, right = null;

   public TreeNode(T e) { element = e; }
   public TreeNode<T> getLeft() { return left; }
   public TreeNode<T> getRight() { return right; }
   public T getElement() { return element; }

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

Traversierung

- systematisches Durchlaufen aller Knoten des Baums



- Inorder: $D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$
- Preorder: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$
- Postorder: $D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$
- Levelorder: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$

Preorder-Traversierung

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

```
static class TreeNode<T> {
   ...
   public void traverse() {
      if (element == null) return;
      System.out.print(" " + element);
      left.traverse();
      right.traverse();
   }
}
```

Inorder-Traversierung

- Inorder-Traversierung: Behandlung des linken Teilbaums, dann des aktuellen Knotens, dann rechter Teilbaum (L-**W**-R)
- Gibt Baum in sortierter Reihenfolge aus

```
public void traverse() {
   if (element == null) return;
   left.traverse();
   System.out.print(" " + element);
   right.traverse();
}
```

Postorder-Traversierung

- Postorder-Traversierung: Behandlung des linken und rechten Teilbaums, dann erst des aktuellen Knotens (L-R-**W**)

```
public void traverse() {
   if (element == null) return;
   left.traverse();
   right.traverse();
   System.out.print(" " + element);
}
```

Levelorder-Traversierung

- Ausgabe der Knoten ebenenweise beginnend bei der Wurzel
- Ordnung innerhalb der Baumebene: beginnend bei kleinstem Element
- Implementierung etwa mit Queue als Zwischenspeicher

```
algorithm Levelorder (k)
   Eingabe: Wurzelknoten k eines binären Baumes

   q := leere Warteschlange;
   enter (q, k); /* Wurzel in Warteschlange aufnehmen */
   while ¬ is_empty (q) do
      Knoten n := leave (q);
      System.out.print(" " + n.elem);
      enter (q, n.left); /* linken Knoten eintragen */
      enter (q, n.right) /* rechten Knoten eintragen */
   od
```

Dictionaries (Wörterbücher)

- ADT zum Abspeichern von Datensätzen, die Schlüssel enthalten.
  - Schlüssel müssen vergleichbar sein (mit "$\leq$" etc.)
- Operationen:
  - `search(k)` — Suche Element mit Schlüssel `k`;
    Rückgabe = Datensatz oder null
  - `insert(x)` — Einfügen von Element `x` (`x` ist Datensatz)
  - `delete(k)` — Löschen des Elements mit Schlüssel `k`
  - oder Varianten dieser Operationen
  - eventuell weitere Operationen: `min`, `max`, `successor`, `predecessor`
- Suchbäume sind eine spezielle Form von Dictionaries.

Arrays und Listen als Dictionaries

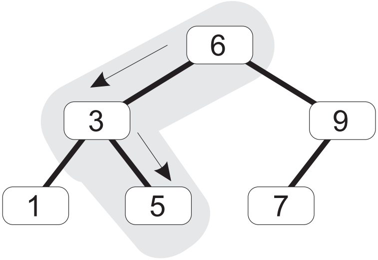
	Search	Insert	Delete *(ohne Search)*
Array unsortiert	$O(n)$	$O(1)$	$O(1)$
Array sortiert	$O(\log n)$	O(n)	O(n)
Liste unsortiert	$O(n)$	$O(1)$	$O(1)$ *(doppelt verkettet)* $O(n)$ *(einfach verkettet)*
Liste sortiert	$O(n)$	O(n)	$O(1)$ $O(n)$ *(einfach verkettet)*

Gesucht ist eine Datenstruktur, die für alle drei Funktionen $O(\log n)$ hat
$\leadsto$ 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 $k.key$
- Inorder-Traversierung eines binären Suchbaums = Ausgabe der sortierten Folge
- Es gibt mehrere binäre Suchbäume für denselben Datensatz; hängt von der Reihenfolge der Eingabe ab.

Suchen im Suchbaum



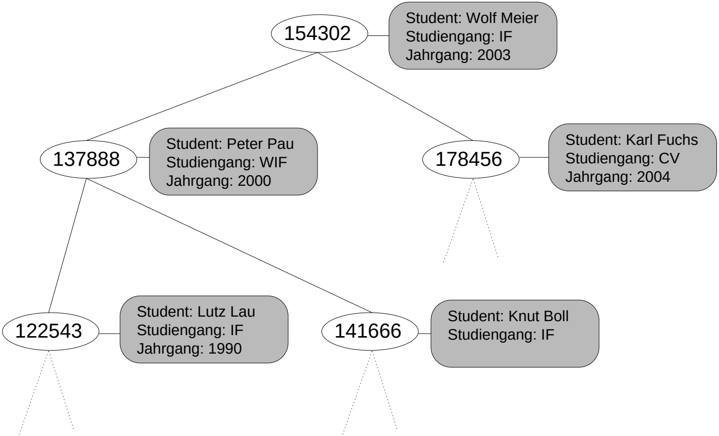
1. Vergleich des Suchschlüssels mit Schlüssel der Wurzel
2. wenn kleiner, dann in linken Teilbaum weitersuchen
3. wenn größer, dann in rechten Teilbaum weitersuchen
4. sonst $\rightsquigarrow$ gefunden

Implementierung von Suchbäumen

- Suchbaum erfordert das zusätzliche Abspeichern eines Schlüssels

```
public class Tree<K,T> {
    TreeNode<K,T> root;
    ...
}

public class TreeNode<K,T> {
    K key;
    T element;
    ...
}
```

Suchbaum: Schlüssel und Daten

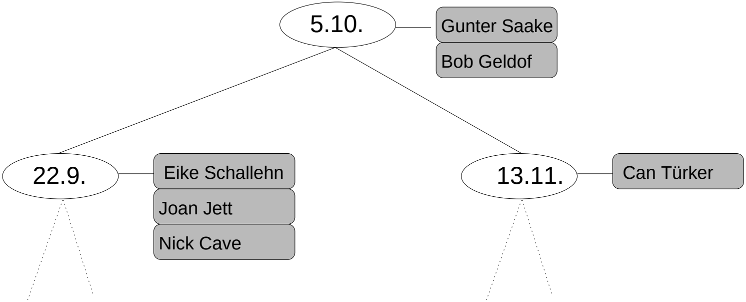


Nicht-eindeutige Suchschlüssel

- Nicht-eindeutiger Suchschlüssel erzwingt mengenwertige Einträge

```
public class Tree<K,T> {
   TreeNode<K,T> root;
   ...
}

public class TreeNode<K,T> {
   K key;
   Set<T> elements;
   ...
}
```

Suchbaum: Schlüssel und Daten



Binärer Suchbaum

- Im folgenden: binärer Suchbaum ohne Betrachtung der Datenelemente
  $\rightarrow$ eigentliche Daten für Verständnis der Algorithmen unerheblich

```
public class Tree<K> {
    TreeNode<K> root;
    ...
}

public class TreeNode<K> {
    K key;
    TreeNode<K> left, right;
    ...
}
```

Weitere Implementierungsalternativen

- Häufige auftretendes Kriterium: Verbesserung der Speicher- und. Laufzeiteffizienz durch Speicherung in “flachen” Strukturen, z. B. Levelorder Speicherung eines binären Baumes in einem Array $\rightarrow$ siehe Abschnitt zu *Heapsort*
- Laufzeiteffizienz durch kleinere innere Knoten: bei Suchbäumen Verweise auf Daten häufig nur an Blattknoten $\rightarrow$ z.B. Binärbaum mit Verzweigung nach $\leq$ und $>$ als Schlüssel

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 $\rightarrow$ AVL- und Rot-Schwarz-Bäume
  - Für Sekundärspeicher: größere, angepasste Knoten günstiger $\rightarrow$ B-Bäume
  - Für Zeichenketten als Schlüssel: variable Schlüsselgröße $\rightarrow$ 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, right = null;

      ...

   }
}
```

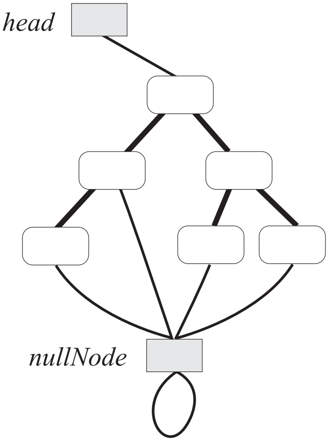
Implementierung eines binären Suchbaums /3

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

Implementierung eines binären Suchbaums /4

- Besonderheit der Implementierung:
  - “leere” Pseudoknoten `head` und `nullNode` zur Vereinfachung der Algorithmen
- Grundlegende Algorithmen
  - Suchen
  - Einfügen
  - Löschen

Implementierung mit Pseudoknoten



Implementierung mit Pseudoknoten /2

```
public class BinarySearchTree<K> {

   public BinarySearchTree () {
      head = new TreeNode<K>(null);
      nullNode = new TreeNode<K>(null);
      nullNode.setLeft(nullNode);
      nullNode.setRight(nullNode);
      head.setRight(nullNode);
   }
   ...
}
```

Implementierung mit Pseudoknoten /3

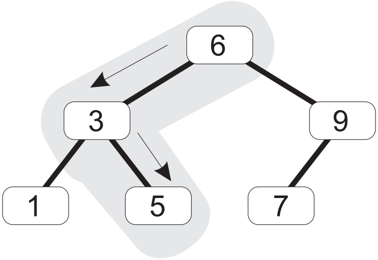
- Ziel: Verminderung von Sonderfällen
- `head`-Knoten:
  - Einfügen oder Löschen des Wurzelknotens würde spezielle Behandlung in der Baum-Klasse erfordern
- `null`-Knoten:
  - Erspart Testen, ob zum linken und rechten Teilknoten navigiert werden kann
  - im `nullNode` einfaches Beenden der Navigation (z.B. Rekursion) möglich

Implementierung mit Pseudoknoten /4

Erweiterung des Knotenvergleichs auf Pseudoknoten:

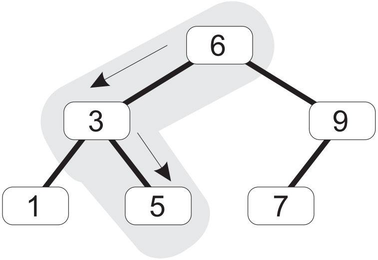
```
class TreeNode< ... > {
    ...
    public int compareKeyTo(K k) {
        return (key == null ? -1 :
                key.compareTo(k));
    }
    ...
}
```

Suchen im binären Suchbaum



1. Vergleich des Suchschlüssels mit Knotenschlüssel
2. wenn kleiner, dann in linken Teilbaum weiter suchen
3. wenn größer, dann in rechten Teilbaum weiter suchen
4. sonst $\rightsquigarrow$ gefunden

Suchen des kleinsten und des größten Elements



- Das kleinste Element steht “am weitesten links” im Baum.
- Das größte Element steht “am weitesten rechts” im Baum.

Binärer Suchbaum: Rekursives Suchen

```
protected TreeNode<K> recursiveFindNode
                              (TreeNode<K> n, K k) {
   if (n != nullNode) {
      int cmp = n.compareKeyTo (k);
      if (cmp == 0)
         return n;
      else if (cmp > 0)
         return
            recursiveFindNode (n.getLeft (), k);
      else
         return
            recursiveFindNode (n.getRight (), k);
   }
   else
      return null;
}
```

Binärer Suchbaum: Iteratives Suchen

```
protected TreeNode<K> iterativeFindNode (K k) {
   TreeNode<K> n = head.getRight();
   while (n != nullNode) {
      int cmp = n.compareKeyTo(k);
      if (cmp == 0)
         return n;
      else
         n = (cmp > 0 ? n.getLeft () : n.getRight ());
   }
   return null;
}
```

Spezialfall: Suchen des kleinsten Elementes

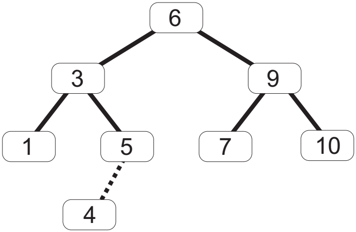
```
public K findMinElement () {
   TreeNode<K> n = head.getRight();
   while (n.getLeft () != nullNode)
      n = n.getLeft ();
   return n.getKey ();
}
```

Spezialfall: Suchen des größten Elementes

```
public K findMaxElement () {
   TreeNode<K> n = head.getRight();
   while (n.getRight () != nullNode)
      n = n.getRight ();
   return n.getKey ();
}
```

Einfügen

1. Finden der Einfügeposition: Abwärts suchen bis zu einem Knoten,
    - dessen Schlüsselwert größer als der einzufügende Schlüssel ist und der keinen linken Nachfolger hat, oder
    - dessen Schlüsselwert kleiner als der einzufügende Schlüssel ist und der keinen rechten Nachfolger hat.
2. Dann neuen Knoten als Blattknoten hinzufügen, sofern der Schlüssel nicht bereits existiert.



Einfügen 1: Einfügeposition suchen

```
public boolean insert (K k) {
   System.out.println("insert: " + k);
   TreeNode<K> parent = head, child = head.getRight();
   while (child != nullNode) {
      parent = child;
      int cmp = child.compareKeyTo(k);
      if (cmp == 0)
         return false;
      else if (cmp > 0)
         child = child.getLeft ();
      else
         child = child.getRight ();
   }
   ...
```

Einfügen 2: neuen Knoten verlinken

```
    ...
    TreeNode<K> node = new TreeNode<K>(k);
    node.setLeft(nullNode); node.setRight(nullNode);
    if (parent.compareKeyTo(k) > 0)
       parent.setLeft (node);
    else
       parent.setRight (node);
    return true;
}
```

Löschen

- zu löschendes Element suchen: Knoten k
- drei Fälle
  1. k ist Blatt: Löschen
  2. k hat einen Sohn: Sohn hochziehen
  3. k hat zwei Söhne: Tausche mit weitest links stehenden Knoten des rechten Teilbaumes und entferne diesen nach Regel 1 bzw. 2.

Löschen 1: Knoten suchen

```
public boolean remove (K k) {
    TreeNode<K> parent = head, node = head.getRight(),
                                    child = null, tmp = null;
    while (node != nullNode) {
        int cmp = node.compareKeyTo(k);
        if (cmp == 0)
            break;
        else {
            parent = node;
            node = (cmp > 0 ?
                node.getLeft() : node.getRight());
        }
    }
    if (node == nullNode)
        return false;
    ...
```

Löschen 2: Nachrücker finden /1

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

Löschen 2: Nachrücker finden /2

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

Löschen 3: Baum reorganisieren

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

Komplexität

- Komplexität der Operationen: $O(h)$ für Baum der Höhe h
- Höhe eines ausgeglichenen Baumes: $h = \log_2 n$ für n Knoten

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

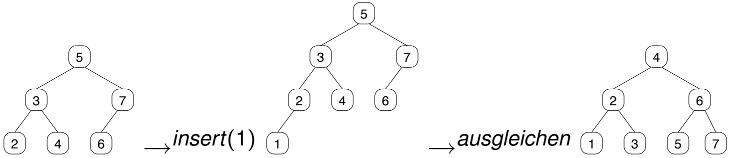
```
for (int i=0; i < 10; i ++) tree.insert(i);
```

  - Vermeidung durch spezielle Algorithmen zum Einfügen und Löschen

Ausgeglichene Bäume

Ein Binärbaum mit n Knoten heißt ausgeglichen, wenn er eine Höhe von $\log_2 n$ hat.

- Wie verhindert man, daß Suchbäume entarten??
- Jeweils ausgleichen ist zu aufwendig:



- Bei dieser Einfügung müsste beim Ausgleichen jeder Knoten bewegt werden!

Lösungsideen

- abgeschwächtes Kriterium für ausgeglichene Höhe
  - Beispiel: AVL-Bäume
- ausgeglichene Höhe, aber unausgeglichener Verzweigungsgrad
  - Beispiele: (2, 4)-Bäume bzw. Rot-Schwarz-Bäume, B-Bäume

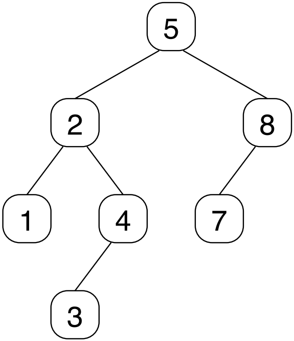
AVL-Bäume

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

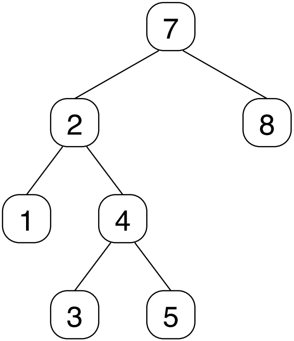
  für jeden (inneren) Knoten gilt: Höhe des linken und rechten Teilbaums differieren maximal um 1.

- Bemerkung: es reicht nicht, dieses nur für die Wurzel zu fordern!

AVL-Eigenschaft am Beispiel



AVL

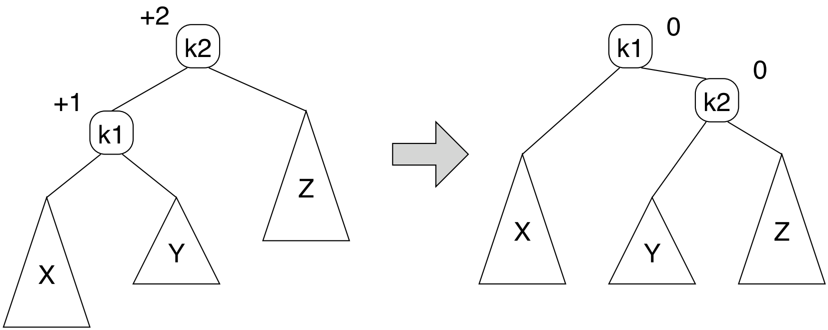


nicht AVL

Einfügen in AVL-Bäume

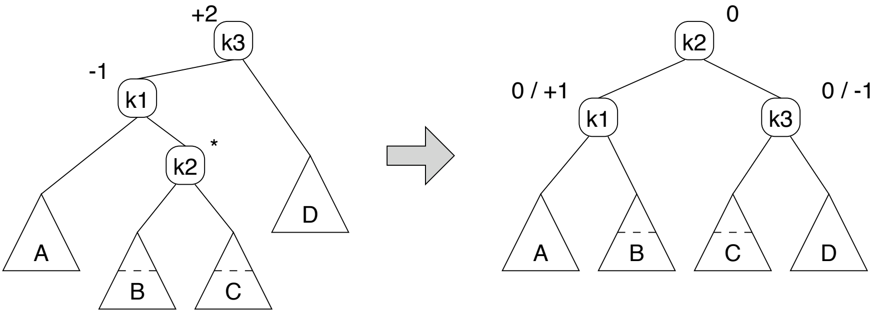
- Einfügen eines Schlüssels mit üblichen Algorithmus
- Danach kann (in einem oder mehreren Knoten) die AVL-Eigenschaft verletzt sein:
  - $\mathit{Balance} = \mathit{left}.\mathit{height} - \mathit{right}.\mathit{height}$
  - AVL-Eigenschaft: $\mathit{balance} \in \{-1, 0, +1\}$
  - nach Einfügen: $\mathit{balance} \in \{-2, -1, 0, +1, +2\}$
- Reparieren mittels *Rotation* und *Doppelrotation*

Einfache Rotation



Rotation mit linkem Kind nach rechts, analoge Operation nach links (spiegelbildlich)

Doppelrotation

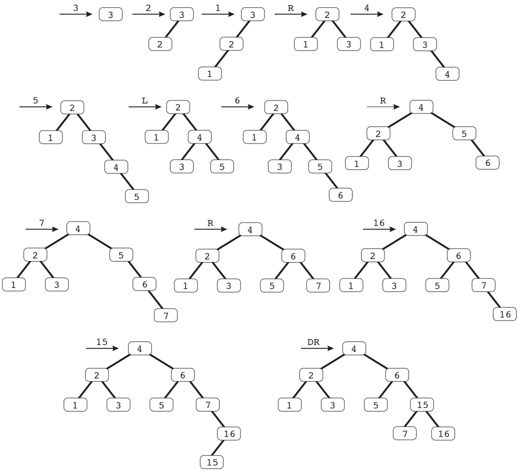


Doppelrotation mit linkem Kind nach rechts, analoge Operation nach links (spiegelbildlich)

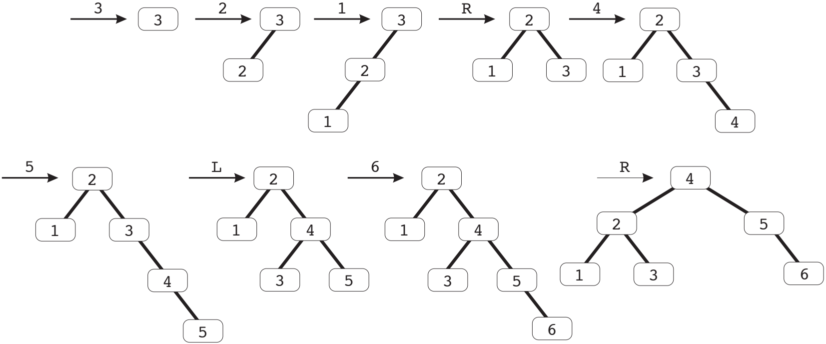
Rotationen am Beispiel

```
insert 3, 2, 1
      → einfache Rotation nach rechts (2,3)
insert 4, 5
      → einfache Rotation nach links (4,3)
insert 6
      → einfache Rotation (Wurzel) nach links (4,2)
insert 7
      → einfache Rotation nach links (6,5)
insert 16, 15
      → Doppelrotation nach links (7,15,16)
insert 13 + 12 + 11 + 10
      → jeweils einfache Rotationen
insert 8, 9
      → Doppelrotation nach rechts
```

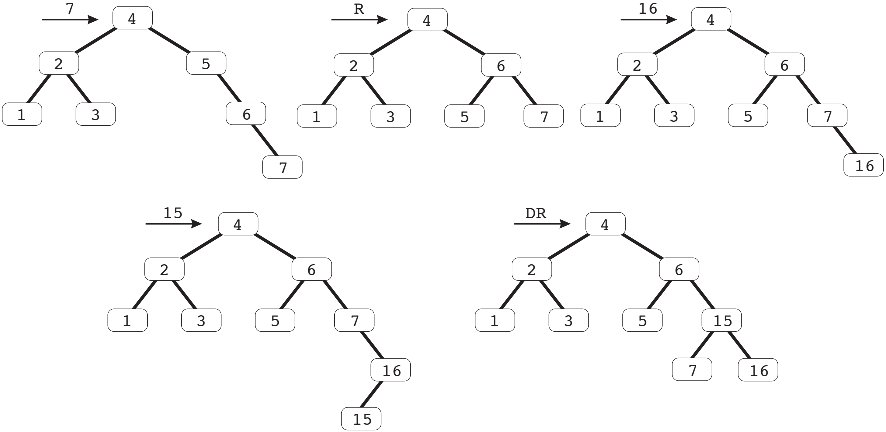
Beispielrotationen (gesamt)



Beispielrotationen I



Beispielrotationen II



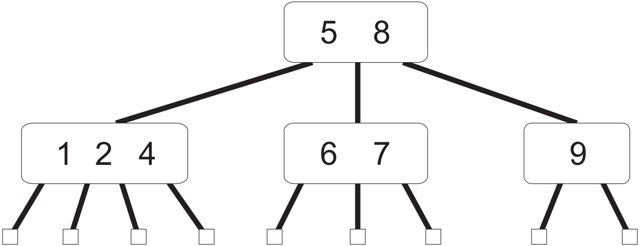
Rotationen in Einzelfällen

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

(2, 4)-Bäume und Rot-Schwarz-Bäume

- Idee: ausgeglichene Bäume mit variablen Verzweigungsgrad
- Ausgeglichenheit wird durch Einfügeoperation gewähleistet
- Implementierung durch Binärbäume

(2, 4)-Bäume



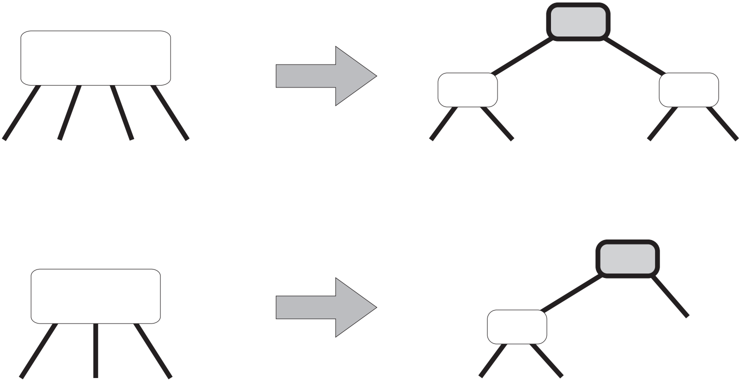
(2, 4)-Bäume enthalten neben binären Knoten (2-Knoten) auch 3-Knoten und 4-Knoten.

Operationen in (2, 4)-Bäumen

- Suche analog zu binären Suchbäumen
- Einfügen:
  - erfolglose Suche liefert Blattknoten *b*
  - ist *b* ein 2- oder 3-Knoten: Einfügen
  - ist *b* ein 4-Knoten: Aufteilen ("split"), mittleres Element nach oben ziehen
  - Splitten kann sich bis zur Wurzel fortpflanzen! (bottom-up)
- Alternativ: Beim Einfügen werden vorsorglich alle 4-Knoten auf dem Pfad gesplittet (top-down)

Binäre Repräsentation von (2, 4)-Bäumen

Rot-Schwarz-Bäume (red-black trees):



B-Bäume

- B-Baum von Bayer und McCreight
  - B steht für balanciert, breit, buschig, Bayer, nicht für binär
- dynamischer, balancierter Suchbaum
- Idee: Baumhöhe vollständig ausgeglichen, aber Verzweigungsgrad variiert

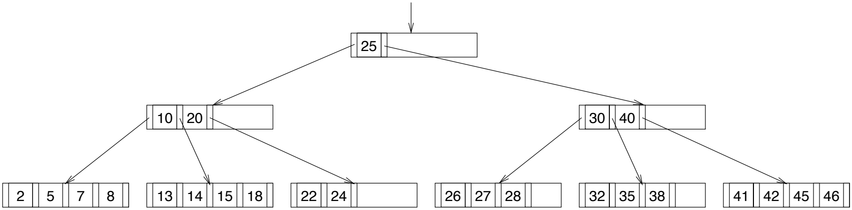
B-Baum-Kriterium: Jede Seite außer der Wurzelseite enthält zwischen m und $2m$ Schlüsselwerte

Eigenschaften eines B-Baumes

- m heißt *Ordnung* des B-Baums
- i Schlüsselwerte ($m \leq i \leq 2m$) im inneren Knoten $\rightarrow i + 1$ Unterbäume
- Höhe des B-Baums bei minimaler Füllung: $log_m(n)$
- n Datensätze $\Rightarrow$ in $log_m(n)$ Seitenzugriffen von der Wurzel zum Blatt

Suchen in B-Bäumen

- Startend auf Wurzelseite Eintrag im B-Baum ermitteln, der den gesuchten Schlüsselwert *w* überdeckt



Komplexität der Operationen

- Aufwand beim Einfügen, Suchen und Löschen im B-Baum immer $O(log_m(n))$ Operationen (dies entspricht genau der "Höhe" des Baumes)
- beliebt für sehr große Datenbestände (mit großer Knotengröße):
  - Konkret : Seiten der Größe 4 KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger: zwischen 50 und 100 Indexeinträge pro Seite; Ordnung dieses B-Baumes 50
  - 1.000.000 Datensätze: $\log_{50}(1.000.000) = 4$ Seitenzugriffe im schlechtesten Fall
- optimiert Anzahl der von der Festplatte in den Hauptspeicher zu transferierenden Blöcke!
- Mehr dazu in der Datenbanken-Vorlesung

Praktischer Einsatz der vorgestellten Verfahren

- AVL in Ausbildung und Lehrbüchern
- Rot-Schwarz-Bäume: Implementierung von Sets und Maps in Java-Bibliothek[1]
- B-Bäume sind “überall im Einsatz” (einfache Einfüge-Algorithmen; Knotengröße an Seitengröße von Hintergrundspeichern optimal anpassbar)

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/TreeMap.html

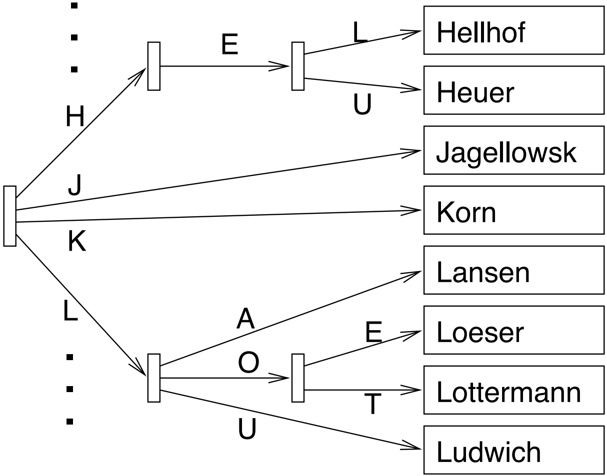
Digitale Bäume

- spezielle (Mehrweg-) Suchbäume für Schlüssel, die Zeichenketten (mit variabler Länge) über festem Alphabet sind.
- Verzweigungsstruktur ist *unabhängig* von den gespeicherten Schlüsseln
- Verzweigung nach jeweils erstem Buchstaben
- können unausgeglichen werden
- Beispiele: Tries, Patricia-Bäume

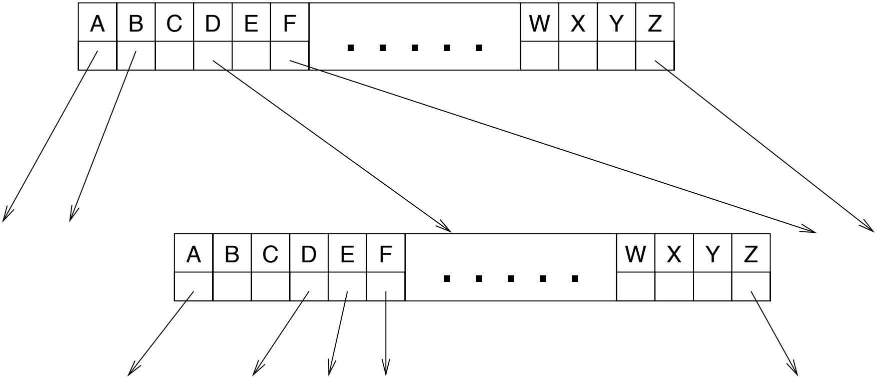
- ‘digital’ von ‘Finger’ (10 Finger → Digitalbaum für numerische Zeichenketten)

Tries

- von “Information Retrieval”, aber wie *try* gesprochen

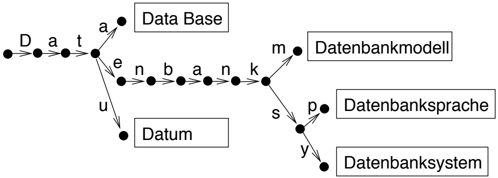


Trie-Knoten



Probleme mit Tries

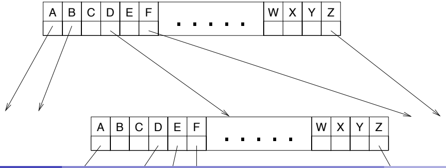
- lange gemeinsame Teilworte, insbes. beispielsweise bei künstlichen Schlüsseln (Teilekennzahlen), Pfadnamen, URLs, Bitfolgen



- nicht vorhandene Buchstaben und Buchstabenkombinationen

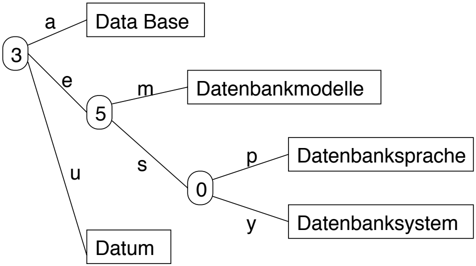
$\rightsquigarrow$ fast leere Knoten

$\rightsquigarrow$ sehr unausgeglichene Bäume

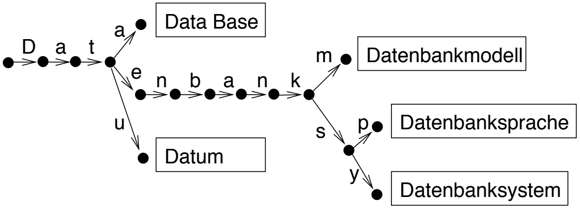


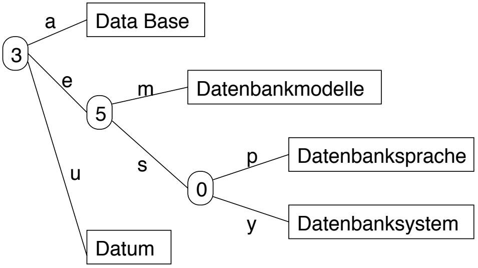
Patricia-Bäume

- *Practical Algorithm To Retrieve Information Coded In Alphanumeric* (Patricia)
- Umgehen die Probleme der Tries
- Prinzip: Angeben, wieviele Zeichen übersprungen werden können, bis die nächste Verzweigung ansteht.

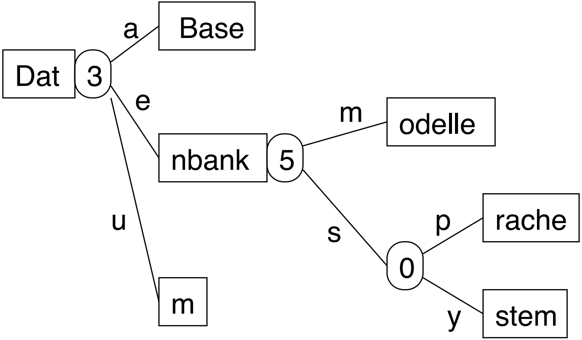


Trie und Patricia-Baum im Vergleich





Präfix-Bäume



Bemerkungen zu digitalen Bäumen

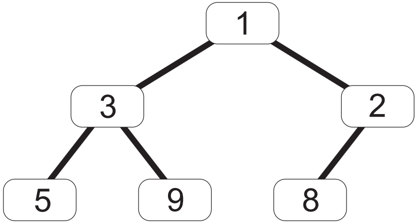
- nur bei Gleichverteilung ungefähr ausgeglichen (welcher Vorname beginnt mit dem Buchstaben ‘Q’?)
- Einsatz insbesondere für Information Retrieval, Textindizierung (Suchmaschinen), Bitfolgen

Prioritätswarteschlange

- Bspw. für Prozesse im Computer
- Element mit jeweils höchster Priorität wird als erstes entnommen
- Nutzt als Datenstruktur einen Heap [dt.: Halde].
- Heaps sind Binärbäume mit der zusätzlichen Heap-Eigenschaft:
  - Baum ist vollständig, d.h., die Blattebene ist von links nach rechts gefüllt.
  - Der Schlüssel eines jeden Knotens ist kleiner (oder gleich) als die Schlüssel seiner Kinder.

Ein Heap

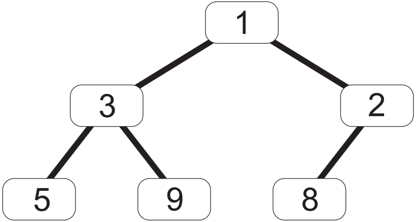
Wurzel enthält Element höchster Priorität



Heap als Array

- Wurzel an der Position 1
  - deren Kinder an den Positionen 2 und 3
  - Knoten der nächsten Ebene auf die Positionen 4, 5, 6 und 7 usw.
- Nummerierung der Knoten des Heaps in Levelorder
  - Kinder des i-ten Knotens auf den Positionen $2i$ (für das linke Kind) und $2i + 1$ (für das rechte Kind)
  - Elternknoten eines Knotens j an der Position $\lfloor j/2 \rfloor$

Heap als Array

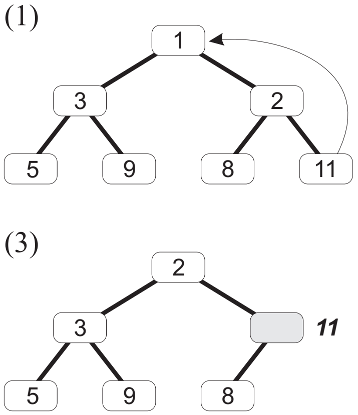


1	3	2	5	9	8
[1]	[2]	[3]	[4]	[5]	[6]

Entfernen aus Heap

1. Entnehme die Wurzel
2. Schiebe „letztes“ Element des Feldes an die Wurzelposition
3. Lasse es jeweils in Richtung des „kleineren“ Elementes „durchsickern“ (eventuell bis auf Blattebene), um die Heap-Eigenschaft zu erhalten.

Entfernen der Wurzel im Heap

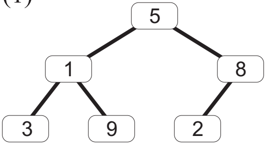


Aufbau eines Heap

1. Betrachte das unsortierte Feld als Baum ohne Heap-Eigenschaft
2. Stelle Heap-Eigenschaft her:
    - Betrachte die ersten $n/2$ Elemente „von hinten her“
    - Lasse jedes Element in den Heap „einsickern“

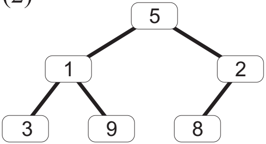
Aufbau eines Heaps

(1)



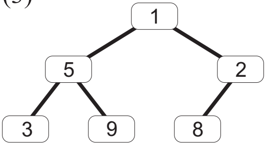
5	1	$\boxed{8}$	3	9	2

(2)



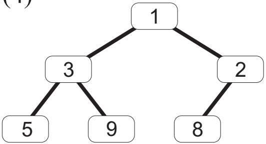
5	$\boxed{1}$	2	3	9	8

(3)



$\boxed{1}$	5	2	3	9	8

(4)



$\boxed{1}$	3	2	5	9	8

Heap-Sort

- Nutzung eines Heaps zum Sortieren:
  - Baum wird in Array gespeichert
  - Heap wird schrittweise ausgelesen und in den freiwerdenden Array-Positionen gespeichert.

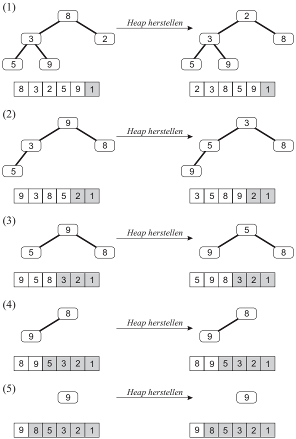
- Average- und Worst-Case-Komplexität: $O(n \log n)$

Heap-Sort: Algorithmus

```
algorithm HeapSort (F)
   Eingabe: zu sortierende Folge F der Länge n

   Überführe F in einen Heap;
   l := n; /* Position des letzten Elementes */
   while l > 1 do
      Vertausche F[1] und F[l];
      Versenke F[1] im Heap F[1...l-1];
      l := l - 1
   od
```

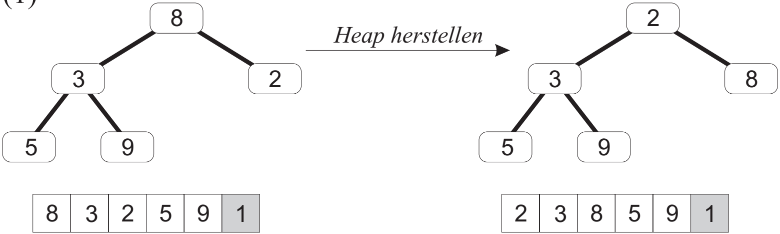
Komplexität: O(n log n), da Versenken O(log n) hat und n-mal aufgerufen wird.

Bsp.: Heapsort



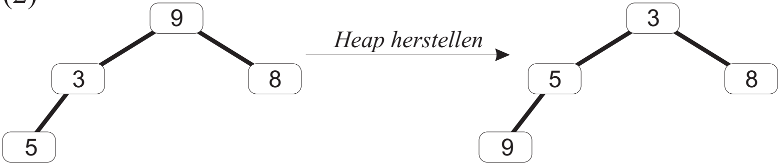
Bsp.: Heapsort I

(1)



Bsp.: Heapsort II

(2)



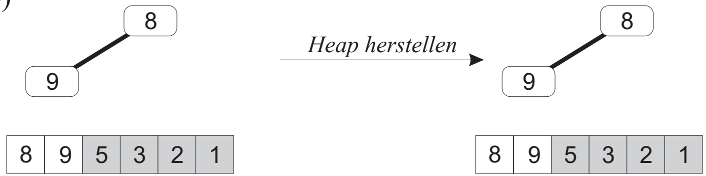
9	3	8	5	2	1

3	5	8	9	2	1

Bsp.: Heapsort III

Bsp.: Heapsort IV

(4)



Bsp.: Heapsort V

(5)