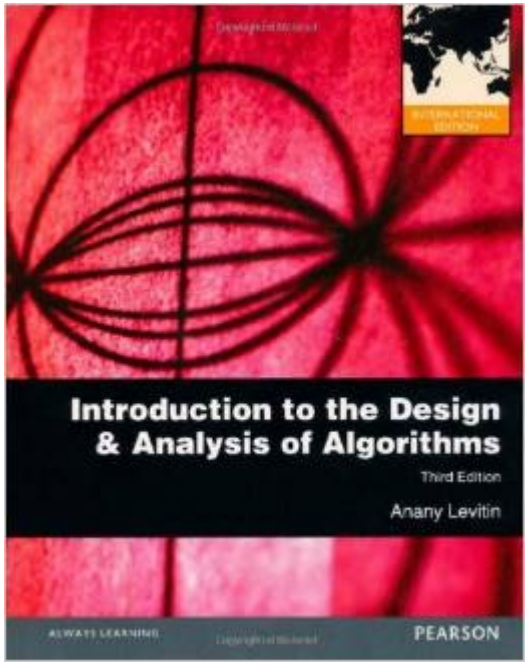
Algorithmen & Datenstrukturen

Transform-and-Conquer-Algorithmen

Literaturangaben

Diese Lerneinheit basiert größtenteils auf dem Buch „The Design and Analysis of Algorithms“ von Anany Levitin.

In dieser Einheit behandelte Kapitel:
6 Transform-and-Conquer
6.1 Presorting
6.3 Balanced Search Trees
6.6 Problem Reduction



Transform-and-Conquer

- Designtechniken, die ein Problem durch **Transformation** lösen
- Drei Varianten
  - **Vereinfachung**
    Transformation in eine einfachere/bequemere Form desselben Problems
  - **Änderung der Darstellung**
    Transformation in eine andersartige Darstellung
  - **Problemreduktion**
    Transformation in einen anderen Problemtyp, für den algorithmische Lösungen bereits bekannt sind

Vereinfachung des Problems: Vorsortieren

- Viele Probleme, bei denen mit **Listen** gearbeitet wird, werden einfacher, wenn die Listen **sortiert** sind:
  - **Suche** in Listen
  - **Medianbestimmung/Auswahlproblem**
  - **Prüfung auf Unterschiedlichkeit** der Elemente
- Außerdem
  - **Topologische Sortierung** hilft bei einigen Problemen mit DAGs
  - Vorsortierung ist bei vielen **geometrischen Problemen** nützlich

Wie schnell kann man sortieren?

- Algorithmus A verwendet Sortieralgorithmus S
  - Effizienz von A **abhängig** von Effizienz von S
- **Theorem** (siehe Levitin, Kap. 11.2):
  - Sei S ein **Sortieralgorithmus**,
  - S basiere auf **Vergleichen**,
  - zu sortierende Liste enthalte **n Elemente**, dann gilt:
  - S benötigt im schlechtesten Fall **mindestens:**

    $\lceil \log_2 n! \rceil \approx n \log_2 n$ **Vergleiche**
- **Hinweis**:
  - Ungefähr $n \log_2 n$ Vergleiche sind zum Sortieren von n Elementen auch **hinreichend** (siehe Mergesort)

Suche mit Vorsortierung

- **Problem**: Suche nach einem Schlüsselwert k in einem gegebenen Array A[0..n-1]
- **Algorithmus mit Vorsortierung**
  - **Phase 1**: Sortiere Array mit effizientem Sortieralgorithmus
  - **Phase 2**: Verwende binäre Suche
- **Effizienz**: $\Theta(n \log n) + O(\log n) = \Theta(n \log n)$
- Vorsortierung hier **sinnvoll**?
  - Warum sind Lexika, Telefonbücher usw. geordnet?

Prüfung auf Unterschiedlichkeit ohne Vorsortierung

- **Brute force-Algorithmus**: Vergleiche alle Elemente
- **Effizienz**: $O(n^2)$

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Prüfung auf Unterschiedlichkeit mit Vorsortierung

- **Phase 1**: Sortiere Array effizient
- **Phase 2**: Prüfe alle benachbarten Elemente auf Unterschiedlichkeit
- **Effizienz**: $\Theta(n \log n) + O(n) = \Theta(n \log n)$

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

```
//Solves the element uniqueness problem by sorting the array first
//Input: An array A[0..n − 1] of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] return false
return true
```

Suchen: Überblick

- **Problem**:
  - Gegeben sei eine (Multi-) **Menge M** und ein **Suchschlüssel S**
  - Finde S in M soweit vorhanden
- Welche **Aspekte** sind zu berücksichtigen?
  - **Größe** der Datenmenge (interne/externe Speicherung)
  - **Dynamik** der Daten (viele/wenige Änderungen)
- **Wichtigste Operationen**
  - find(S)
  - insert(S)
  - delete (S)

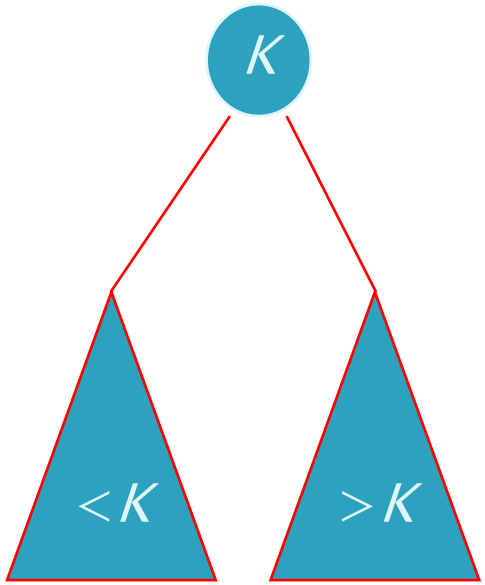
Taxonomie der Suchalgorithmen

- **Listenbasierte Suche**
  - Sequentielle Suche
  - Binäre Suche
  - Interpolationssuche
- **Baumbasierte Suche**
  - Binärer Suchbaum
  - Balancierte binäre Suchbäume
    - AVL-Bäume
    - Rot/Schwarz-Bäume
  - Balancierte Mehrwegbäume
    - 2-3 oder 2-3-4 Bäume
    - B-Bäume
- **Hashing**
  - Verkettung/separate chaining
  - Sondieren/open addressing

Transform-and-Conquer: Änderung der Darstellung

Binäre Suchbäume: Suchen und Einfügen

- Anordnung der Schlüssel gemäß der Eigenschaft binärer Suchbäume:



- **Suche** – gemäß Suchbaumeigenschaft
- **Einfügen** – Suche nach Schlüssel, Einfügen unterhalb des letzten gefundenen Blattes

- Beispiel: 6, 2, 1, 11, 13, 8, 10

Binäre Suchbäume: Löschen

Drei Fälle: Zu löschender Schlüssel steht in

- **einem Blatt**
  - direktes Löschen des Blattes möglich
- **einem Knoten mit einem Nachfolger**
  - Nachfolger mit Vorgänger (soweit vorhanden) verbinden
  - Knoten mit Schlüssel löschen
- **in einem Knoten mit zwei Nachfolgern**
  - Lösche Schlüssel aus Knoten
  - Trage Schlüssel des Ordnungsnachfolgers/ -vorgängers in den Knoten ein
  - Lösche Knoten des Ordnungsnachfolgers/ -vorgängers (gemäß erstem oder zweitem Fall)

Ordnungsnachfolger/ -vorgänger von k: Knoten mit nächstgrößtem/ -kleinstem Wert im Baum

Binäre Suchbäume: Effizienz

- Effizienz abhängig von der **Höhe h des Baums**:

$$\lfloor \log_2 n \rfloor \leq h \leq n-1$$

- Durchschnittliche Höhe (bei Zufallsdaten) etwa

$$3 \log_2 n$$

- Für alle drei Grundoperationen gilt:
  - Schlechtester Fall: O(n)
  - Mittlerer Fall $\approx \log n$
- **Bonus:**
  Inorder-Traversierung liefert sortierte Liste der Schlüssel

Balancierte Suchbäume

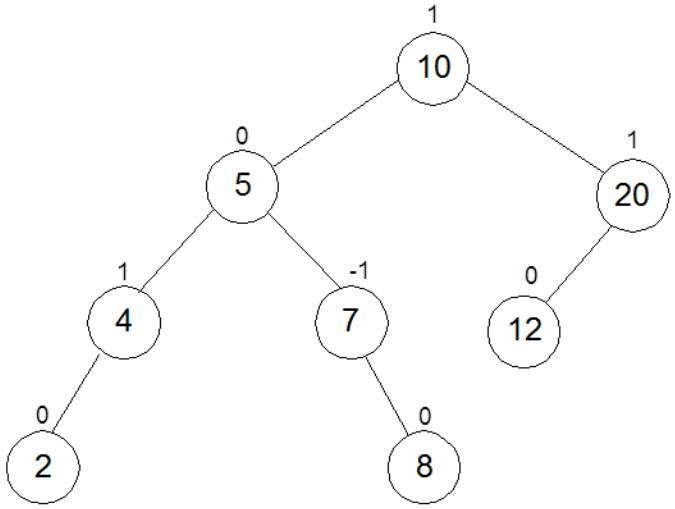
Attraktivität binärer Suchbäume durch **schlechten (linearen) Worst-Case** beeinträchtigt

Zwei Lösungsansätze

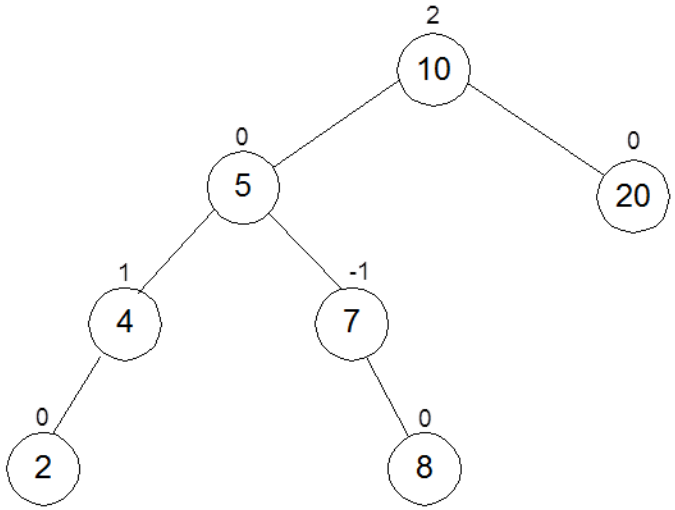
- Baum **rebalancieren**, wenn er zu unausgeglichen ist
  - AVL-Bäume
  - Rot/Schwarz-Bäume
- **Mehr als einen Schlüssel** pro Knoten zulassen
  - 2-3-Bäume/2-3-4-Bäume
  - B-Bäume

Balancierte Suchbäume: AVL-Bäume

- **AVL-Baum**
  - Binärer Suchbaum
  - Für *alle* Knoten gilt: **Unterschied der Höhe** des linken und rechten Unterbaums beträgt **höchstens 1**



(a)
AVL-Baum

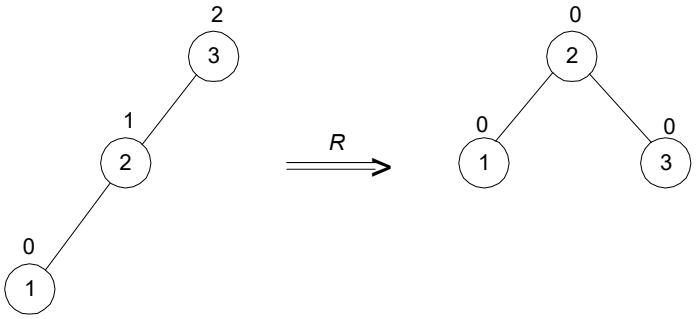


(b)
Kein AVL-Baum

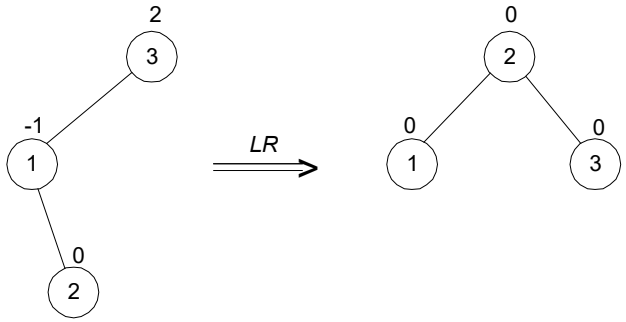
Rotationen

Falls nach Einfügen eines Knotens die Balance-Eigenschaft verletzt ist:

- Transformiere den Unterbaum des unbalancierten Knotens durch **Rotation** (**vier Varianten** möglich)
- Rotiere immer den Unterbaum mit dem unbalancierten Knoten als Wurzel, der **dem eingefügten Blatt am nächsten** ist

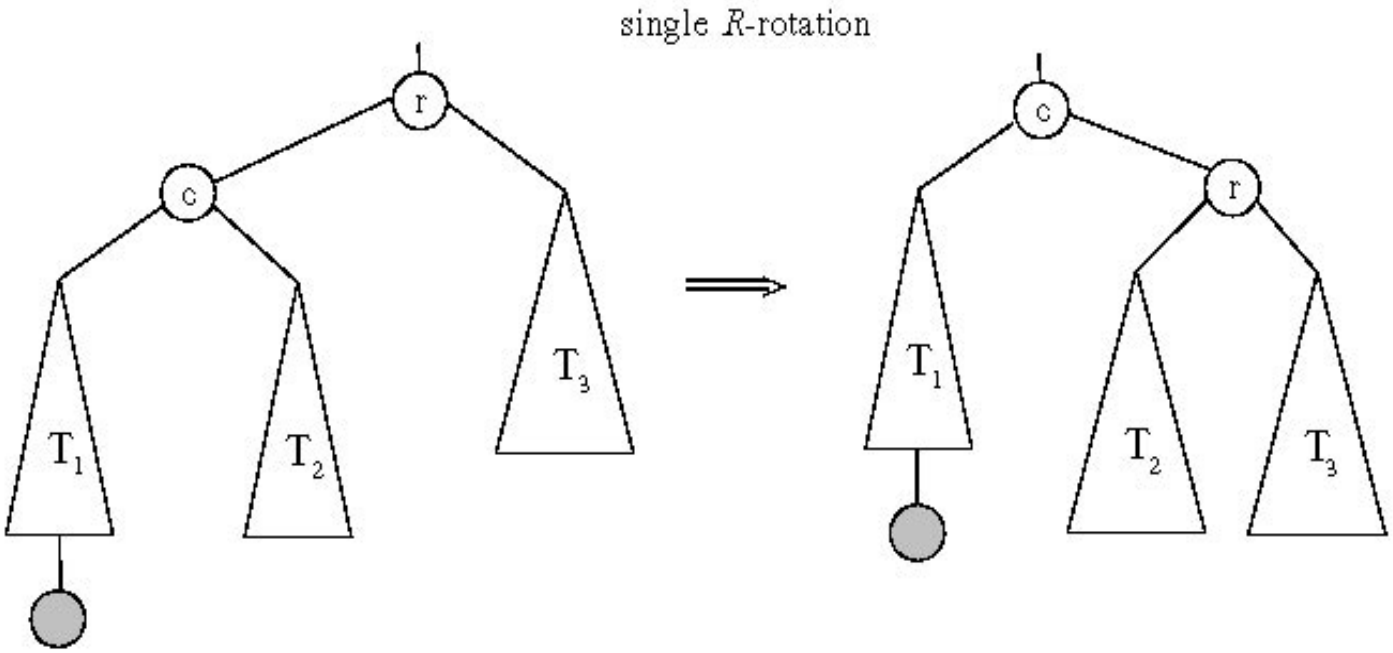


Einfache R-Rotation

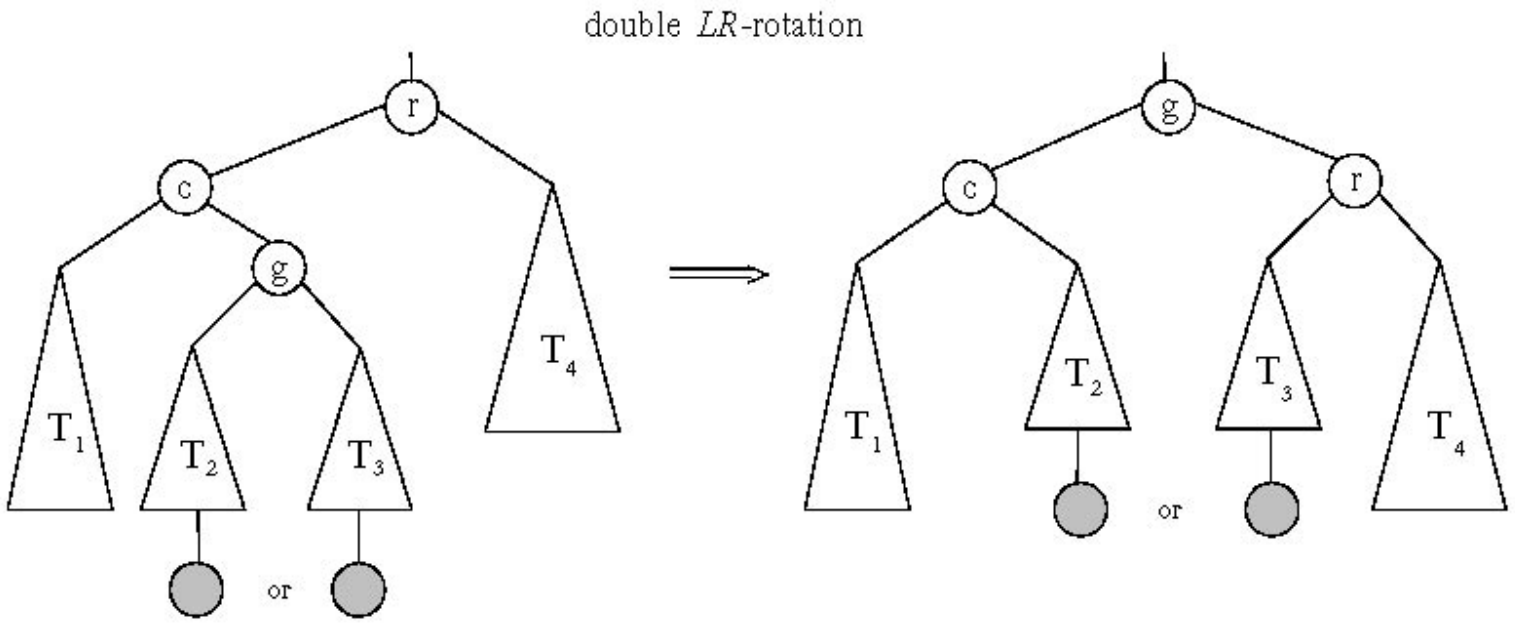


Doppelte LR-Rotation

Allgemeiner Fall: Einfache R-Rotation

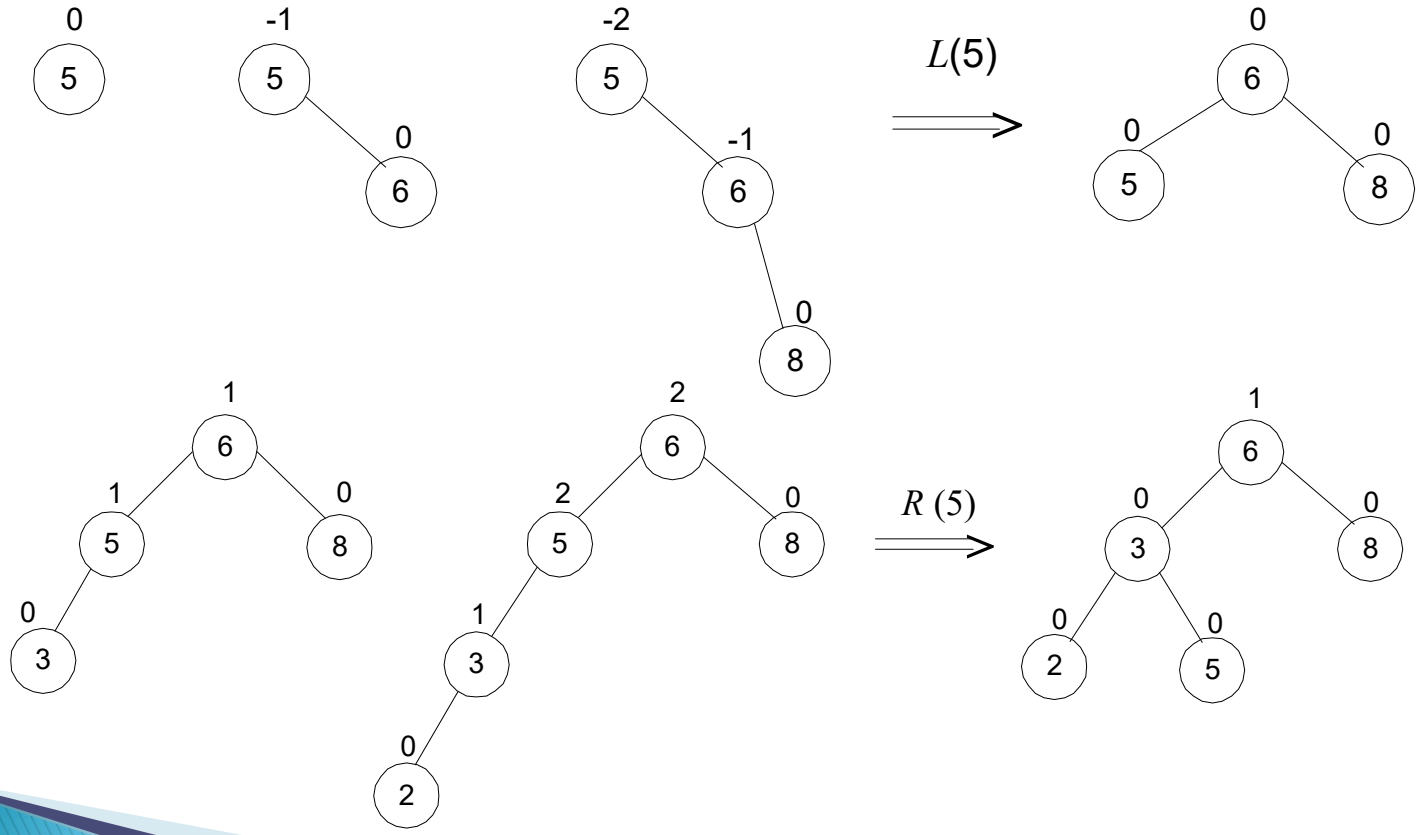


Allgemeiner Fall: Doppelte LR-Rotation

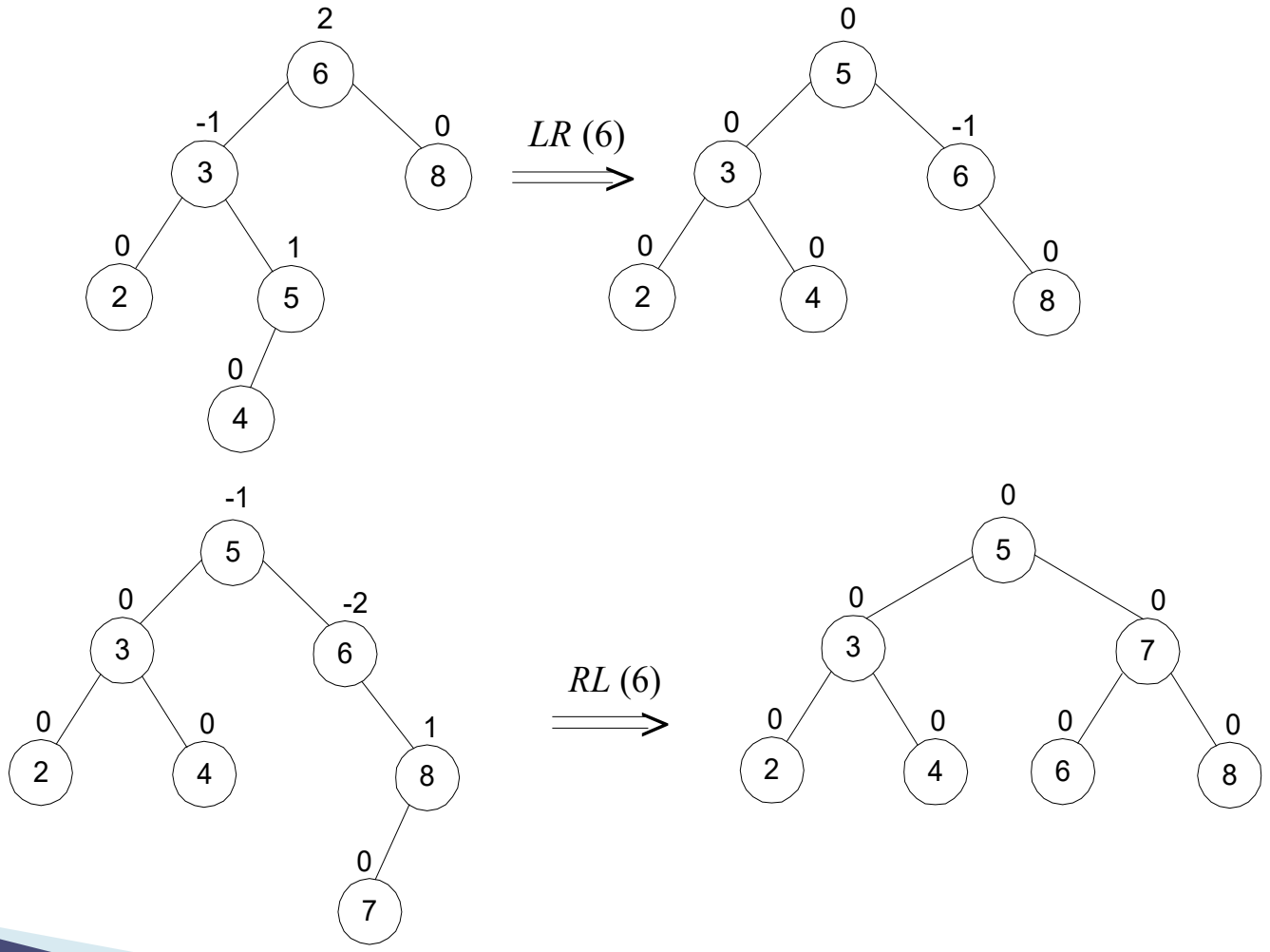


AVL-Baum-Konstruktion: Beispiel (Teil 1)

Konstruiere schrittweise einen AVL-Baum mit den Elementen 5, 6, 8, 3, 2, 4, 7



AVL-Baum-Konstruktion: Beispiel (Teil 2)

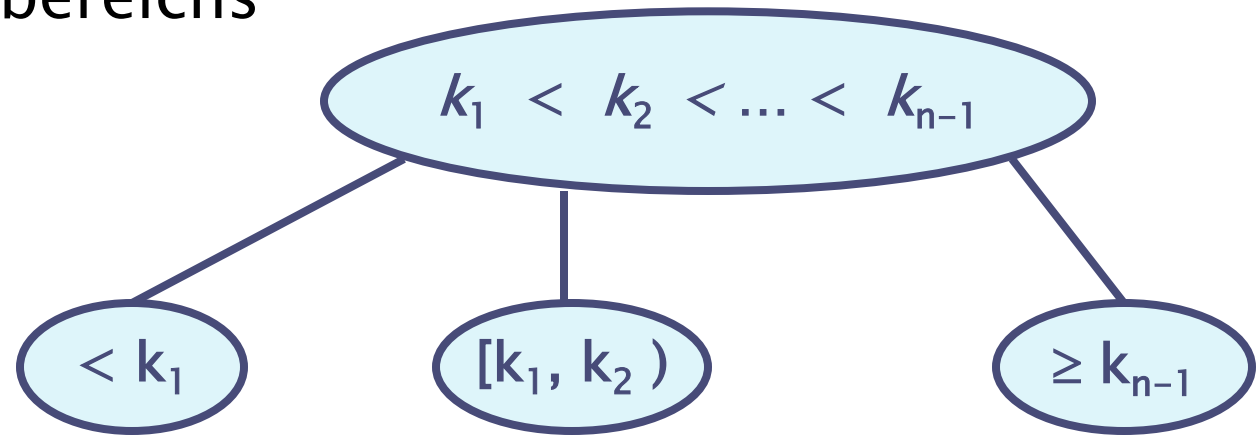


Analyse von AVL-Bäumen

- **Höhe**
  - $h \leq 1.4404 \log_2 (n + 2) - 1.3277$
  - Durchschnittliche Höhe für große n: $1{,}01 \log_2 n + 0.1$ (empirisch ermittelt)
- **Operationen**
  - Suchen und Einfügen: O(log n)
  - Löschen etwas komplizierter, aber auch in O(log n) möglich
- **Nachteile**
  - Häufige Rotationen
  - Operationen komplex
- Ähnlicher Ansatz: **Rot-Schwarz-Bäume** (Höhe der Unterbäume darf um den Faktor 2 unterschiedlich sein)

Mehrweg-Suchbäume

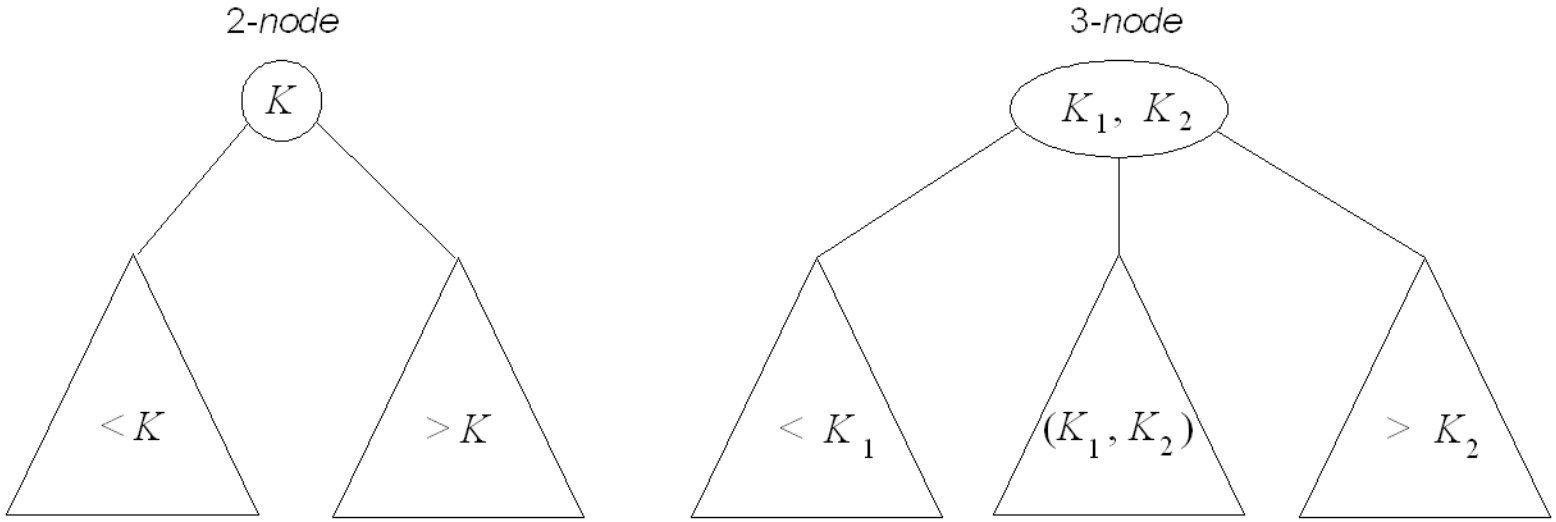
- **Mehrweg-Suchbaum**: Suchbaum, dessen Knoten **mehr als einen** Schlüssel enthalten können
- **n-Knoten**: Knoten eines Suchbaums, der **n-1 geordnete Schlüssel** enthält
  - **n Kinder** des n-Knotens formen **n Intervalle** des Schlüsselbereichs



  - Knoten eines binären Suchbaums sind **2-Knoten**

2-3-Bäume

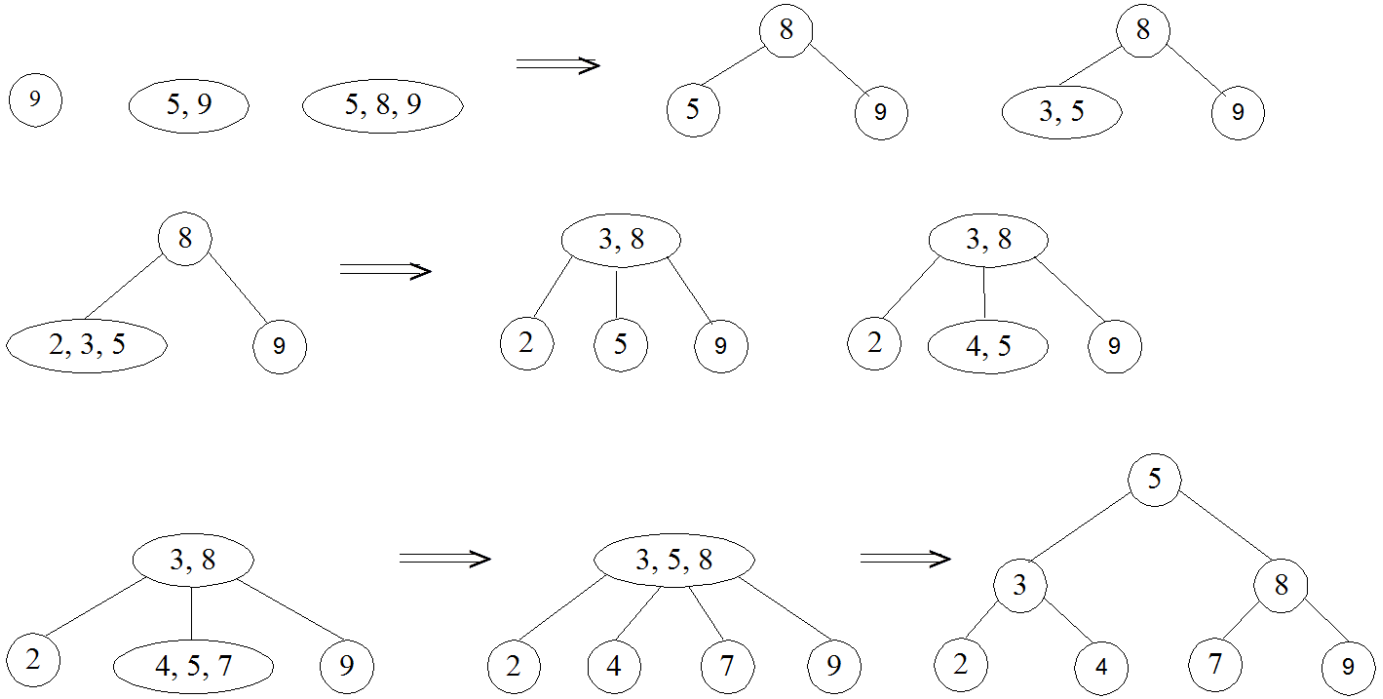
- **Definiton**: Ein 2-3-Baum ist ein Suchbaum der
  - **2-Knoten** oder **3-Knoten** besitzen kann und
  - **vollständig höhenbalanciert** ist (alle Blätter befinden sich auf der gleichen Ebene)



Einfügen in 2-3-Bäume

- **Einfügen** in 2-3-Baum:
  - Einfügen **in bestehendes Blatt** (außer beim ersten Schlüssel)
  - Falls Blatt bereits 3-Knoten → **spalte in zwei Knoten** und reiche **mittleres Element zum Elternknoten** (soweit vorhanden)

Füge ein:
9, 5, 8, 3, 2, 4, 7



Analyse von 2-3-Bäumen

- **Höhe h** eines 2-3-Baumes:
  - $\log_3 (n + 1) - 1 \leq h \leq \log_2 (n + 1) - 1$
- Suche, Einfügen und Löschen in **$\Theta(\log n)$**
- Idee erweiterbar auf Bäume mit **noch mehr Schlüsseln pro Knoten**
  - 2-3-4-Bäume
  - B-Bäume

Transform-and-Conquer: Problemreduktion

- Transformiere
  - das **gegebene Problem**
  - in ein **andersartiges Problem**
  - mit **bekannter algorithmischer Lösung**
- Praktisch **nur sinnvoll** wenn

Aufwand für Transformation

+

Aufwand zur Lösung des anderen Problems

<

Aufwand zur Lösung des originalen Problems

Beispiele für Problemreduktion

Originalproblem	gelöst mittels…
Kleinstes gemeinsames Vielfaches	Größter gemeinsamer Teiler
Anzahl der Pfade der Länge n in einem Graphen	Berechnung der n-ten Potenz der Adjazenzmatrix
Maximierungsproblem	Minimierungsproblem
Verschiedene Denksportaufgaben	Graphen Probleme