



Seminar Vortrag
Transform and Conquer

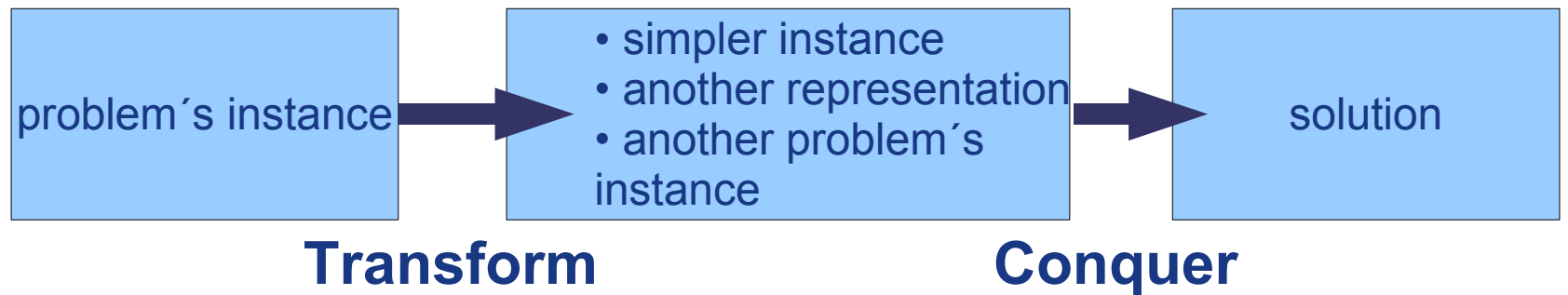
Alexander Sittig - winf8045

Themenübersicht

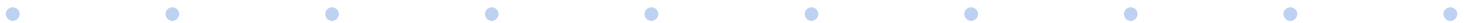
- Grundlagen
- Presorting
- Gaußsches Eliminationsverfahren
- Balancierte Bäume
 - AVL Bäume
 - 2-3 Bäume
- Heap
- Wörterbuch Problem



Grundlagen



Presorting



Presorting

- Array mit n Zahlen nicht sortiert
- Überprüfen das keine Zahl doppelt vorkommt

| | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 34 | 12 | 35 | 48 | 15 | 32 | 67 | 41 | 95 | 38 | 85 | 43 | 63 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|

- Man muss also n Zahlen mit n-1 Zahlen Vergleichen
- $\rightarrow n * (n-1)$ Vergleiche = $n^2 - n = O(n^2)$



Presorting

- Array mit n Zahlen sortiert
- Überprüfen das keine Zahl doppelt vorkommt

| | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 12 | 15 | 32 | 34 | 35 | 38 | 41 | 43 | 48 | 63 | 67 | 85 | 95 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|

- Man muss also n-1 Zahlen mit einer Zahl Vergleichen
- $\rightarrow (n-1) * 1$ Vergleiche = $(n - 1) = O(n)$



Presorting

Fazit:

Man benötigt $O(n^2)$ Zeit bei einer unsortierten Liste

Man benötigt $O(n)$ Zeit bei einer sortierten Liste

Das Sortieren einer Liste dauert mit z.B. Mergesort
 $O(n \log n)$

wenn man die unsortierte Liste erst sortiert und
dann überprüft braucht man

$O(n \log n) + O(n) = O(n \log n)$



Presorting

- Array mit n Zahlen nicht sortiert
- Suchen nach einer Zahl z

Brute-Force Ansatz:

alle n Element mit z vergleichen $\rightarrow O(n)$

Presorting Ansatz:

erst alle Elemente sortieren $\rightarrow O(n \log n)$ + Binäresuche
 $\rightarrow O(\log n) = O(n \log n) + O(\log n) = O(n \log n)$

Presorting ist von der Gesamtlaufzeit bei einem Suchvorgang langsamer – erst bei mehrere Suchvorgängen spart man Zeit ein



Gaußsches Eliminationsverfahren



Gaußsches Eliminationsverfahren

Anfang: n Gleichungen mit n Unbekannten

Form:

$$a_1 * x_1 + a_2 * x_2 + a_3 * x_3 = a_4$$

$$b_1 * x_1 + b_2 * x_2 + b_3 * x_3 = b_4$$

$$c_1 * x_1 + c_2 * x_2 + c_3 * x_3 = c_4$$

Wenn alle n Gleichungen linear unabhängig sind gibt es eine eindeutige Lösung



Gaußsches Eliminationsverfahren

- Ansatz: Umformen des Gleichungssystems in die Dreieckform

$$a_1 * x_1 + a_2 * x_2 + a_3 * x_3 = a_4$$

$$0 * x_1 + b_2 * x_2 + b_3 * x_3 = b_4$$

$$0 * x_1 + 0 * x_2 + c_3 * x_3 = c_4$$

In dieser Form kann die Lösung für x_3 sofort abgelesen werden



Gaußsches Eliminationsverfahren

- Beispiel:

$$1 * x_1 + 1 * x_2 + 1 * x_3 = 2$$

$$2 * x_1 + 1 * x_2 + 1 * x_3 = 3$$

$$1 * x_1 - 1 * x_2 + 3 * x_3 = 8$$

$$1 * x_1 + 1 * x_2 + 1 * x_3 = 2$$

$$0 * x_1 - 1 * x_2 - 1 * x_3 = -1 \quad | \text{II} - 2 * \text{I}$$

$$0 * x_1 - 2 * x_2 + 2 * x_3 = 6 \quad | \text{III} - 1 * \text{I}$$



Gaußsches Eliminationsverfahren

- Beispiel:

$$1 * x_1 + 1 * x_2 + 1 * x_3 = 2$$

$$0 * x_1 - 1 * x_2 - 1 * x_3 = -1$$

$$0 * x_1 - 0 * x_2 + 4 * x_3 = 8 \quad | \text{ III} - 2 * \text{ II}$$

Einsetzen

$$x_3 = 2$$

$$x_2 = 1 - x_3 = -1$$

$$x_1 = 2 + 1 - 2 = 1$$



Gaußsches Eliminationsverfahren

- 1. Schritt umwandeln in Dreiecksmatrix

for $i \leftarrow 1$ to $n-1$ do

for $j \leftarrow i+1$ to n do

for $k \leftarrow i$ to $n+1$ do

$$A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$$

- 2. Einsetzen

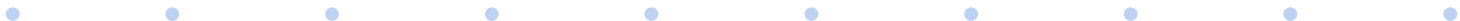
for $j \leftarrow n$ downto 1 do

$t \leftarrow 0$

for $k \leftarrow j+1$ to n do

$t \leftarrow t + A[j, k] * x[k]$

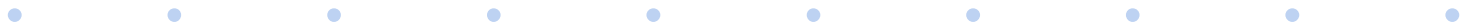
$$x[j] \leftarrow (A[j, n+1] - t) / A[j, j]$$



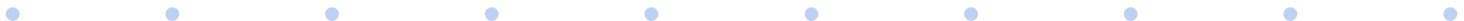
Gaußsches Eliminationsverfahren

Laufzeit Abschätzung

- Für 1. Schritt - umwandeln in Dreiecksmatrix
 - 3 geschachtelte von n abhängige Schleifen
 - Laufzeit $O(n^3)$
- Für 2. Schritt - Einsetzen
 - 2 geschachtelte von n abhängige Schleifen
 - Laufzeit $O(n^2)$
- Gesamtlaufzeit: $O(n^3) + O(n^2) = O(n^3)$

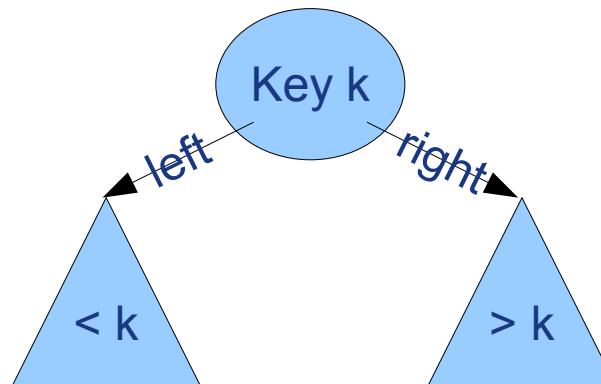


Suchbäume



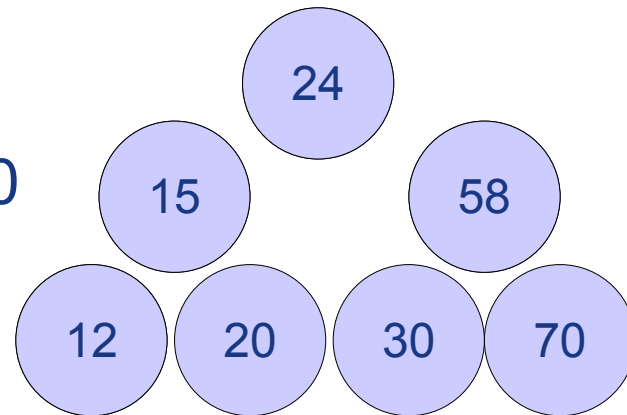
Binäre Suchbäume

- Jeder Knoten enthält einen Schlüssel k
- Jeder Knoten hat max. 2 Kind-Knoten
 - Im Linken Teilbaum sind alle Schlüssel $< k$
 - Im Rechten Teilbaum sind alle Schlüssel $> k$



Motivation für Balancierte Bäume

Binärer Baum für
Best Case
24, 15, 58, 12, 20, 30, 70



Suchen: $O(\log n)$
Einfügen: $O(\log n)$
Löschen: $O(\log n)$



Motivation für Balancierte Bäume

Binärer Baum

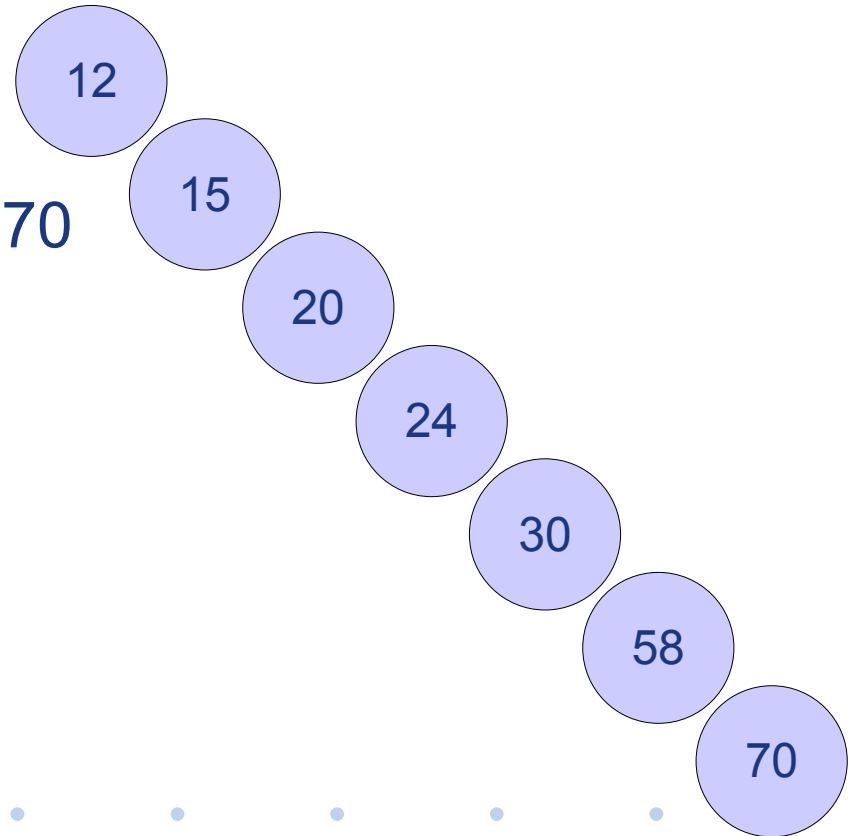
Worst Case

12, 15, 20, 24, 30, 58, 70

Suchen: $O(n)$

Einfügen: $O(n)$

Löschen: $O(n)$

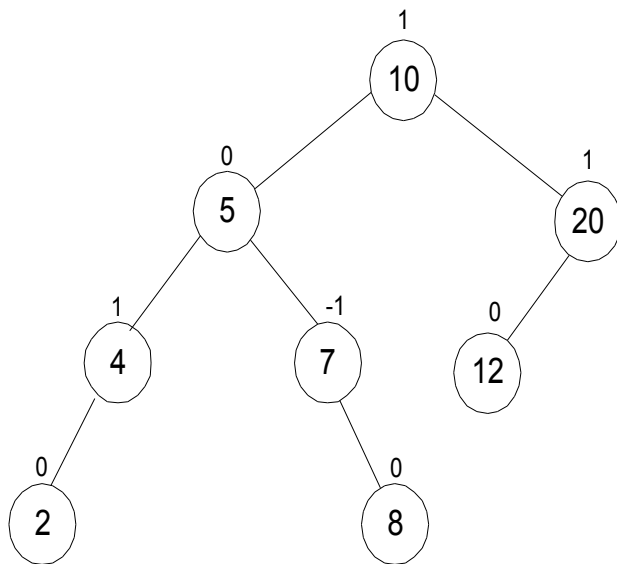


AVL-Bäume

- 1962 entwickelt von den russischen Wissenschaftlern G.M. Adelson-Velsky und E.M. Landis
- Definition:
 - Binärbaum
 - An jeden Knoten wird die Längendifferenz zwischen linken und rechten Teilbaum gespeichert – genannt “balance factor”
 - “balance factor” nur -1,0,1
 - Die Höhe des leeren Baums “-1”

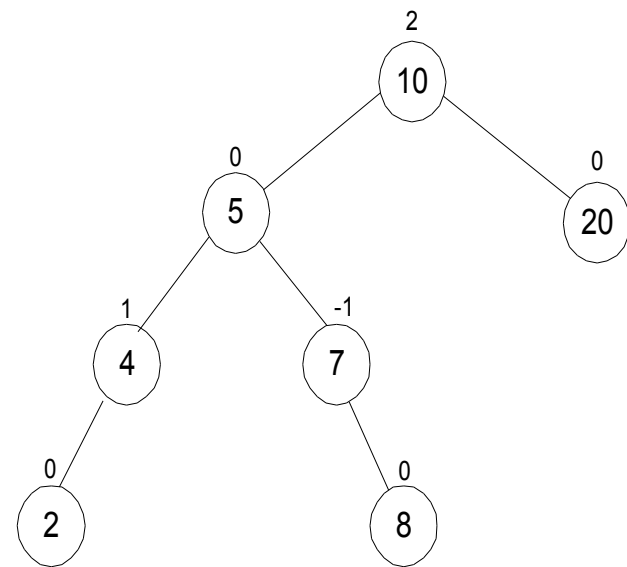


AVL-Bäume



(a)

AVL-Baum



(b)

Kein AVL-Baum

AVL-Bäume

Wie wird garantiert das der AVL Baum beim Einfügen ein AVL Baum bleibt?

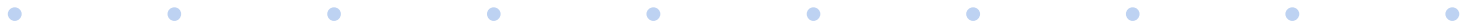
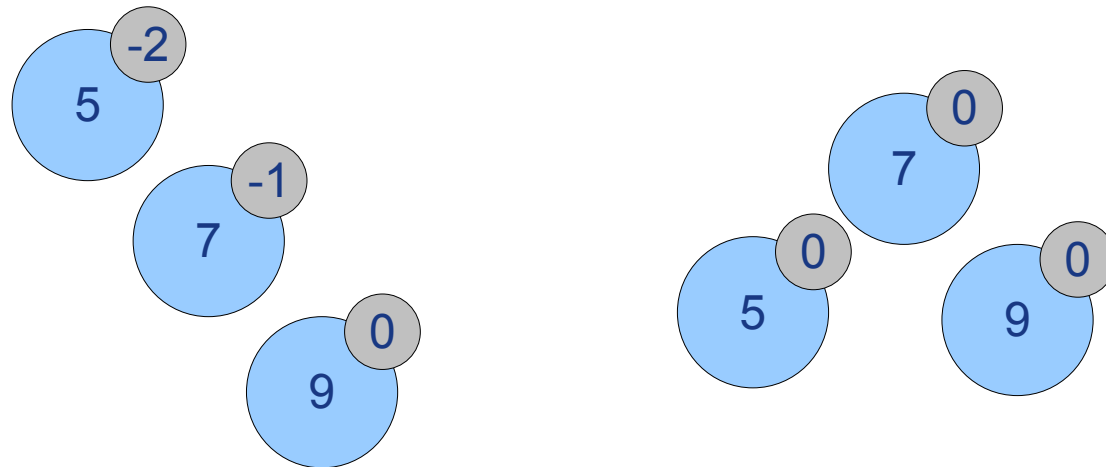
Nach dem Einfügen wird “balance factor” für die Knoten Neuberechnet

Sollte jetzt festgestellt werden das der AVL Baum unbalanciert ist gibt es verschiedene Möglichkeiten genannt *rotations* den Baum wieder auszubalancieren



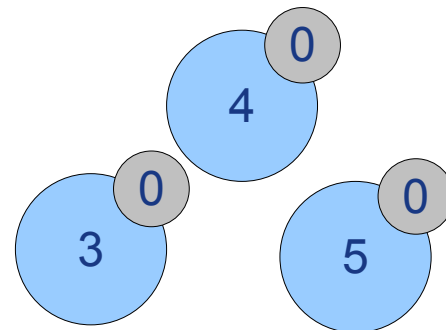
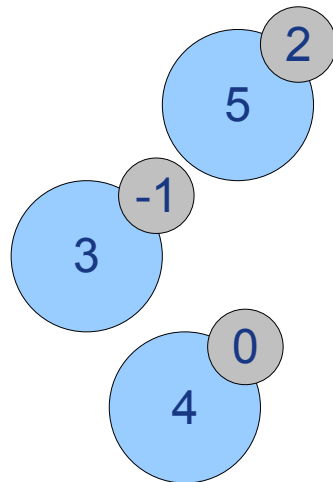
AVL-Bäume

Single left rotation - L-Rotation



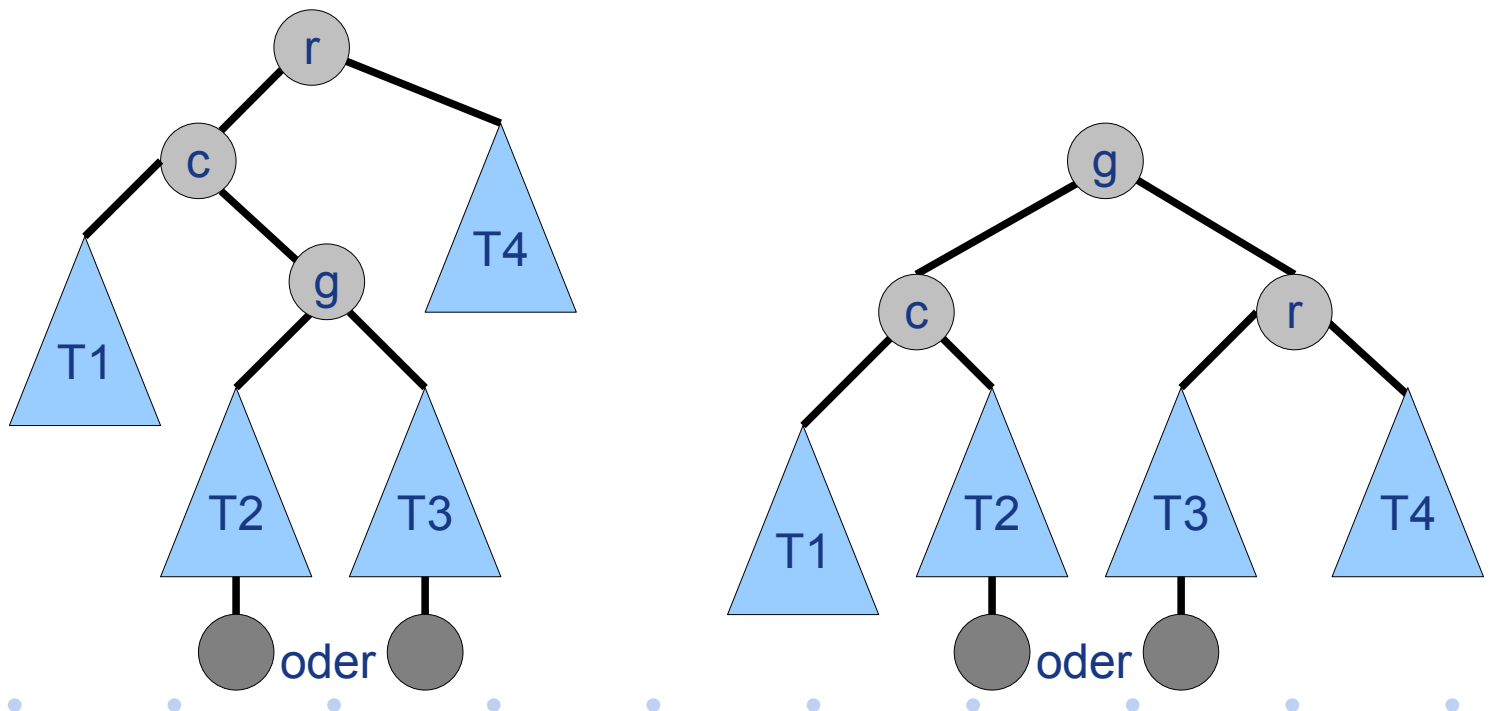
AVL-Bäume

double left-right rotation - LR-Rotation



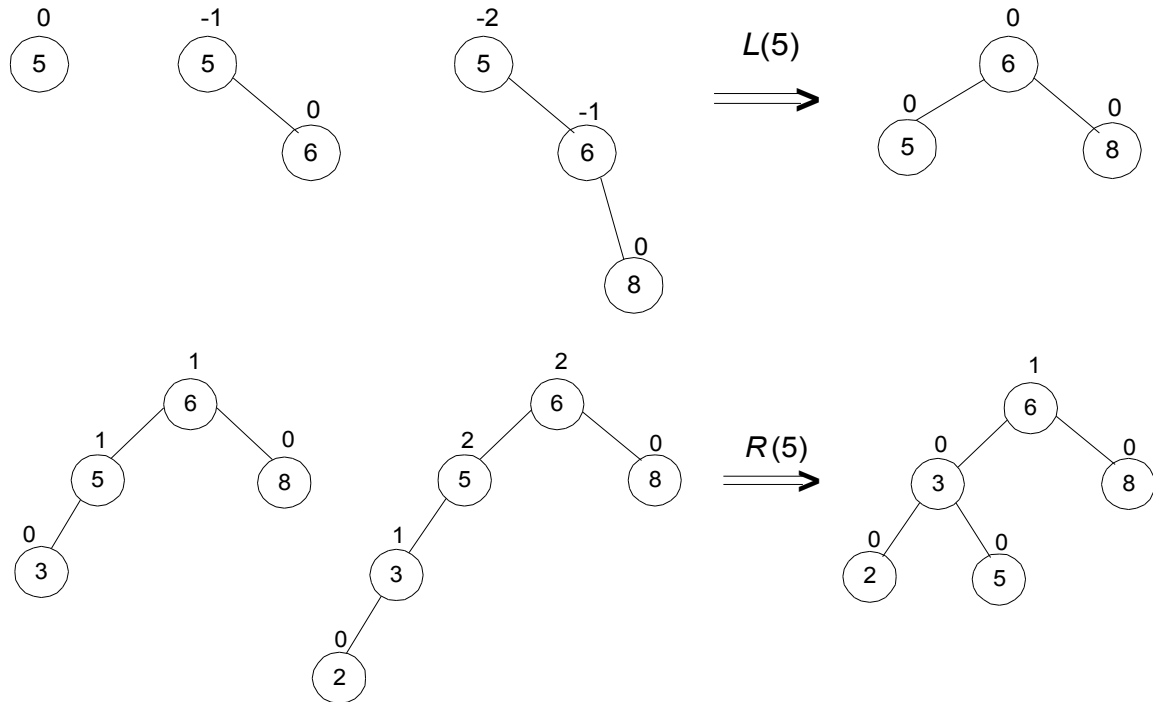
AVL-Bäume

Double left-right rotation - LR-Rotation

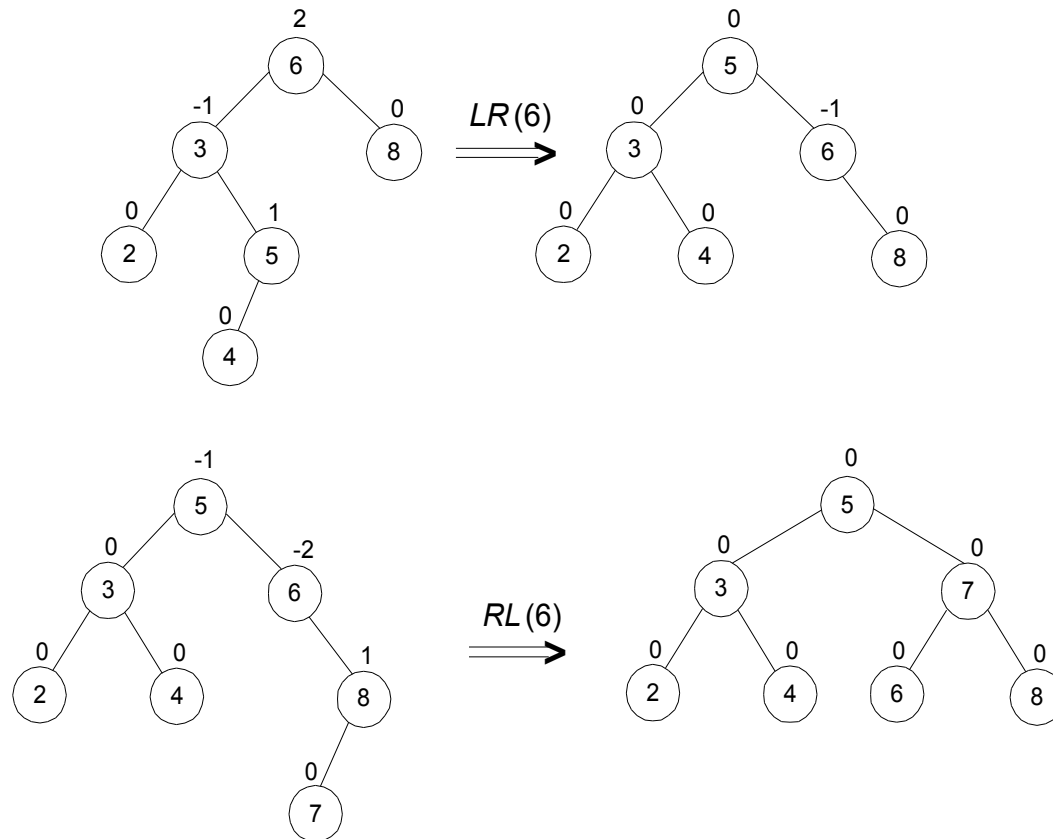


AVL-Bäume

- Beispiel: 5,6,8,3,2,4,7



AVL-Bäume



AVL-Bäume

Laufzeitabschätzung:

Die Höhe: $h \leq 1.44 \log_2 (n + 2) - 1.3277$

Es gibt also eine obere Schranke für die max. Höhe eines AVL Baums mit n Elementen

-> Bei der Suche in einem AVL-Baum ist die Anzahl der Vergleiche durch die Höhe des Baumes beschränkt -> Suche $O(\log n)$

-> Beim Einfügen ist die Anzahl der Rotationen auch begrenzt -> Einfügen $O(\log n)$

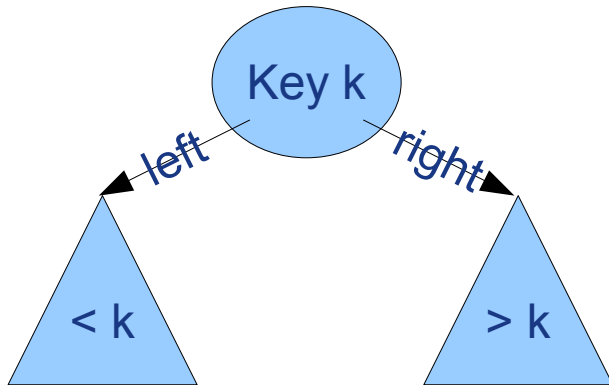
2-3-Bäume

- 1970 entwickelt von John Hopcraft
- Definition:
 - Kein Binärbaum - Mehrwegbaum
 - Speichert Informationen in Knoten und Blättern
 - Es gibt Knoten mit einer Infokomponente und zwei Kindknoten diese sind gleich der Knoten in einem Binär-Baum
 - Es gibt Knoten mit zwei Infokomponenten und drei Kindknoten
 - Es wird nur in den Blättern eingefügt



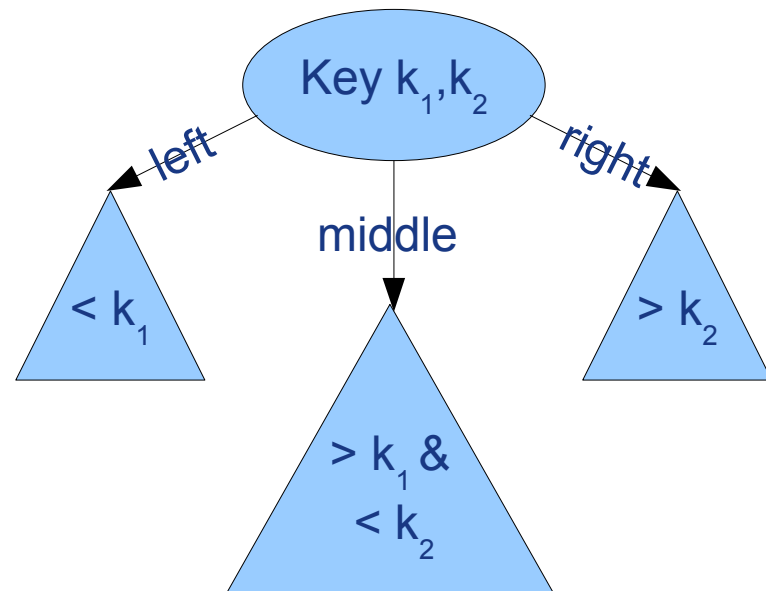
2-3-Bäume

Knoten-2



Knoten-3

$$k_1 < k_2$$

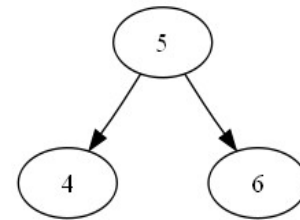


2-3-Bäume

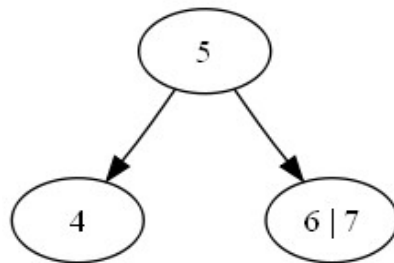
- Beispiel: 4,5,6,7,8



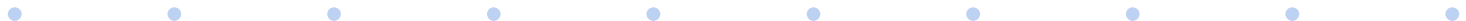
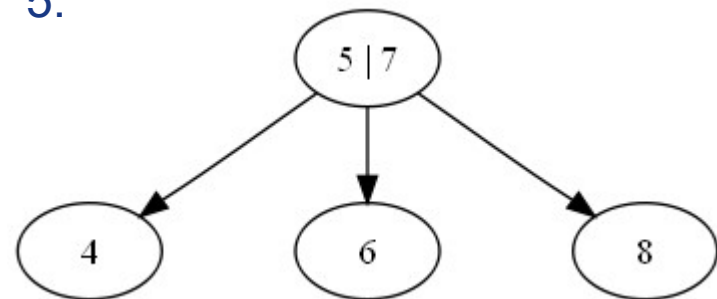
3.



4.



5.



2-3-Bäume

Laufzeitabschätzung:

Die Höhe: $h \leq \log_2 (n + 1) - 1$

- > Bei der Suche in einem 2-3 Baum haben wir max. $2 \cdot (\log_2 (n + 1) - 1)$ Vergleiche, wenn nur Knoten vom Type3 besuchen -> $O(\log n)$
- > Beim Einfügen ist die Anzahl der zusätzlichen Operation für das Teilen von Knoten des Type3 durch die Höhe des Baumes beschränkt -> $O(\log n)$

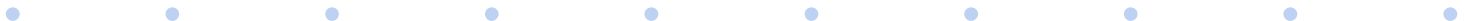


Heap



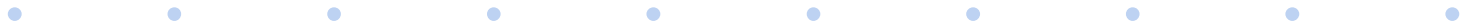
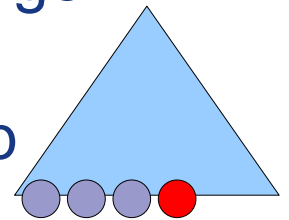
Heap

- Genutzt als z.B. Vorrangwarteschlangen
- Definition:
 - Binärbaum
 - Ein Schlüssel pro Knoten
 - Beide Kindknoten sind $<$ oder \leq dem Schlüssel
 - Der Baum ist komplett \rightarrow es gibt nur auf der untersten Ebene rechts Knoten die keine zwei Kindknoten besitzen



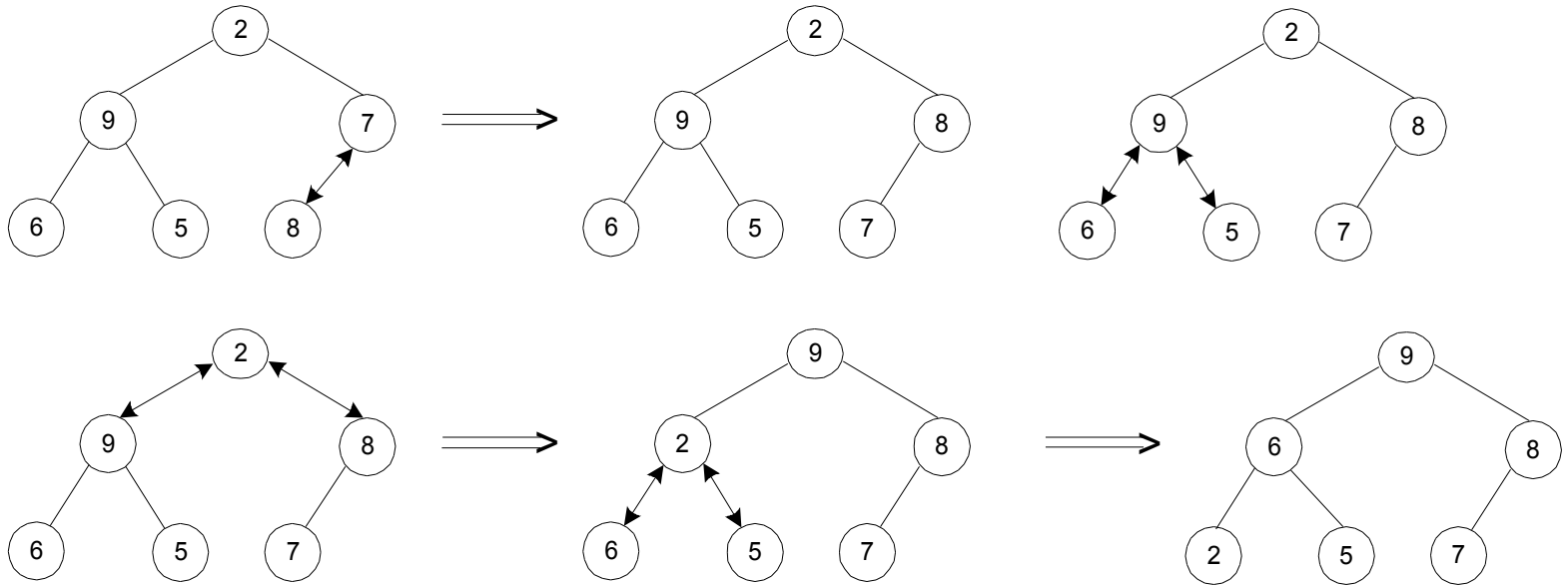
Heap Konstruktion

- Start: Initialisieren der Struktur mit den Schlüsseln in der gegebenen Reihenfolge
- 1: Überprüfe beginnend mit rechten Knoten in der letzten Reihe die Heap Bedingung($k < \text{Elternknoten}.k$)
 - Sollte die Bedingung nicht erfüllt sein, vertausche parent und den aktuellen Knoten. Und prüfe weiter bis Bedingung erfüllt oder Wurzel erreicht
 - Wiederhole 1 für ehemals Elternknoten



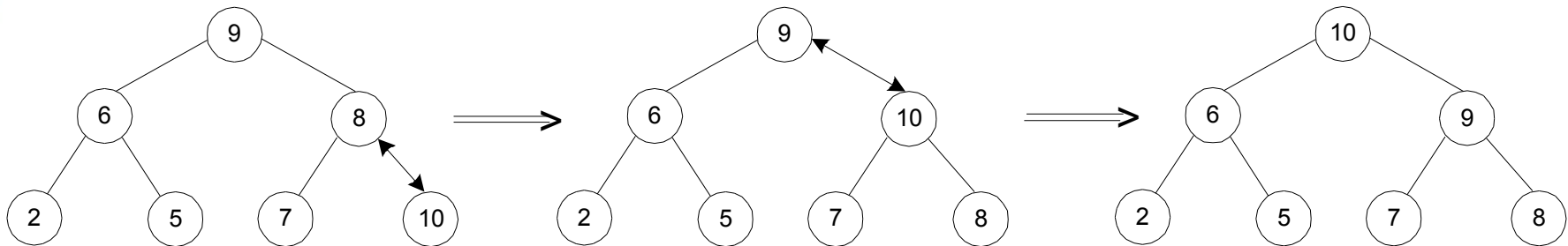
Heap

- Beispiel: 2, 9, 7, 6, 5, 8



Heap

Einfügen eines neuen Elements in den Heap
Beispiel Einfügen der 10 in den eben erstellten
Heap



Heap

Laufzeitabschätzung:

Einfügen: Beim Einfügen wird die Anzahl der Tausch Operationen durch die Höhe des Baumes beschränkt und da laut Definition der Baum ein dichter Baum ist $\rightarrow O(\log n)$

Erstellen: 1. Schritt Aufbau des Baum $\rightarrow O(n)$

2. Schritt Überprüfung der Heap Bedingung
Beschränkt durch die Höhe des Baums $\rightarrow O(n)$



Heapsort

- Array mit n Zahlen nicht sortiert
- 1. Schritt: Konstruiere einen Heap für die n -Elemente des Arrays
- 2. Schritt: Speichere Wurzel des Heaps, entferne Wurzel des Heap. Wiederhole Schritt 2 solange bis der Heap leer ist.



Wörterbuch Problem

Folgende Eigenschaften werden von Dictionaries gefordert:

- Die Einträge sind Schlüssel-Wert Paare
- Die Funktionen sollen effizient implementiert sein
 - > $O(\log n)$
 - Suchen
 - Einfügen
 - Löschen



UML – Diagramm 2-3 Baum

