Tafelübung 10

Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Verkettete Listen

Streutabellen
- Motivation und Grundlagen
- Hashfunktion und Belegungsfaktor
- Kollisionsauflösung
- Implementierung von `hashCode`

AVL-Bäume
- Wiederholung
- Balancefaktor
- Definition
- Rotationen

Halden
- Definition
- Operationen

Bäume malen
- ... mit Dia
- ... mit DOT

Verkettete Listen

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Verkettete Liste: Motivation

Szenario

Unser Programm soll Mitarbeiter verwalten. Im Laufe der Zeit werden neue Mitarbeiter eingestellt oder alte Mitarbeiter entlassen, d.h. die Mitarbeiterzahl variiert.

- Problem bei einer Umsetzung mittels Array:
  - Arrays liegen zusammenhängend im Speicher
  - $\leadsto$ Verkleinerung und insb. Vergrößerung i.A. nicht möglich
  - $\leadsto$ Arrays haben eine feste Größe

$\leadsto$ Verwendung einer Verketteten Liste:

- dynamische Datenstruktur $\leadsto$ Speicherung von „beliebig“ vielen Elementen
  - heißt: zu Beginn nicht bekannte und zur Laufzeit veränderliche Anzahl
- Realisierung: Referenzen zwischen den einzelnen Listenelementen
- $\leadsto$ Verkettung der Listenelemente
- $\leadsto$ Elemente können „verstreut“ im Speicher liegen

Verkettung: einfach/zweifach

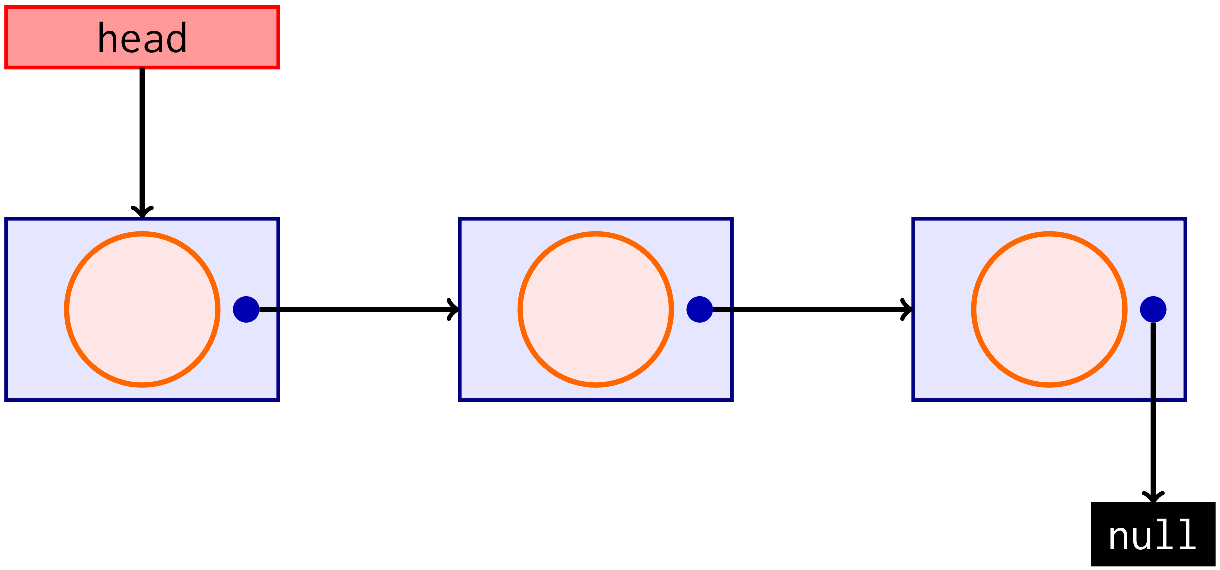
Einfach verkettete Liste

In einer einfach verketteten Liste kennt jedes Element seinen direkten Nachfolger.
$\leadsto$ Traversierung nur in eine Richtung möglich.

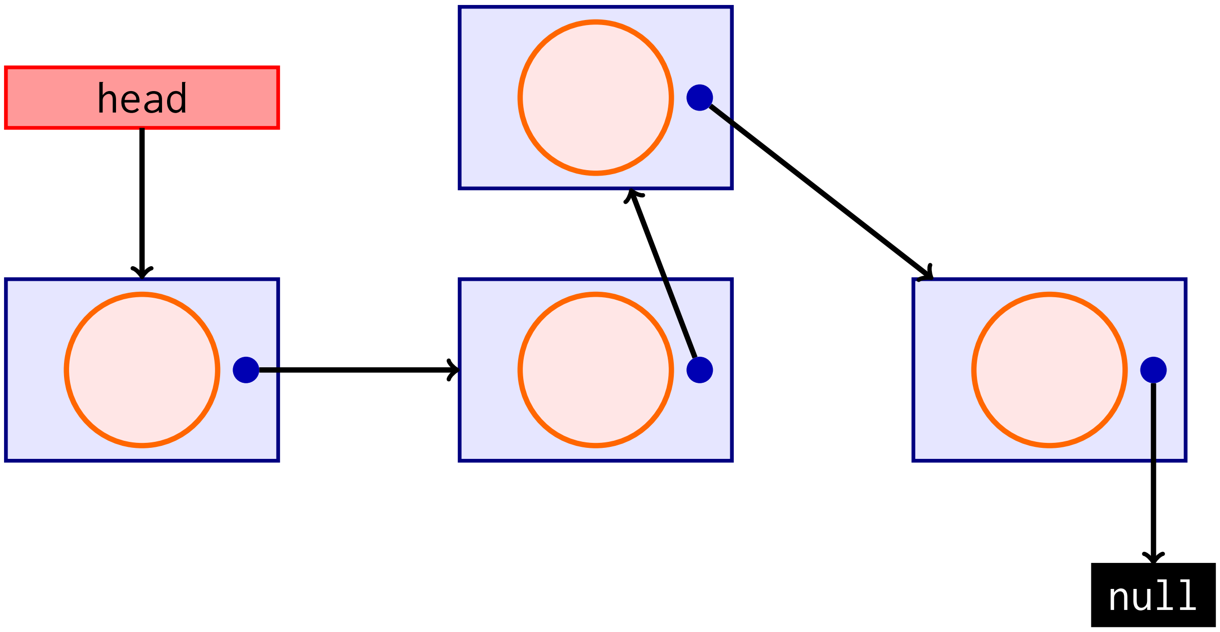
Zweifach verkettete Liste

In einer zweifach verketteten Liste kennt jedes Element zusätzlich seinen dir. Vorgänger.
$\leadsto$ Traversierung in beide Richtungen möglich.

(Einfach) Verkettete Liste: Idee

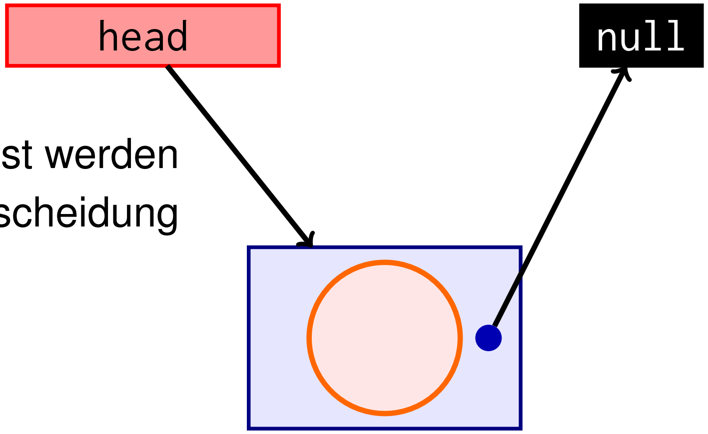


(Einfach) Verkettete Liste: Idee



Einfügen in leere Liste: Spezialfall

Einfügen in leere Liste: Spezialfall



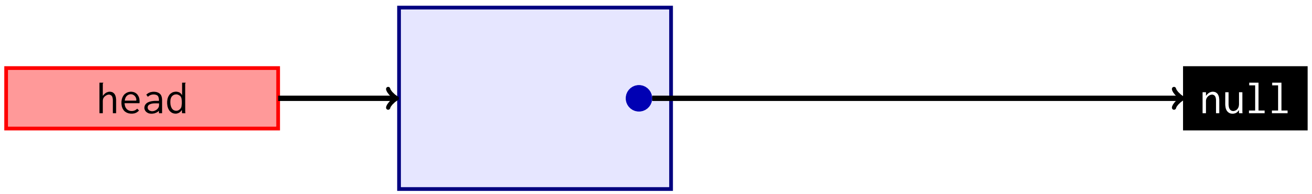
head muss angepasst werden
⇝ Fallunterscheidung

Einfügen in leere Liste mit Sentinel: Kein Spezialfall!

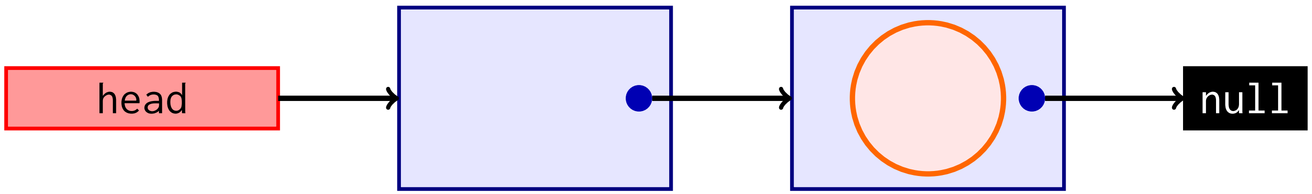
- Sentinel $\equiv$ *Wächter*
  - „leeres“ Element am Anfang der Liste
  - head zeigt *immer* auf Wächter
  - erstes „echtes“ Element wird hinter Wächter eingehängt

$\rightsquigarrow$ kein Spezialfall mehr für leere Liste ☺

Verkettete Liste mit Wächter



Verkettete Liste mit Wächter



Streutabellen

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Streutabellen: Motivation (I)

Szenario

Zu Mitarbeitern, die über ihren Namen identifizierbar sind, sollen Informationen wie Alter, Geschlecht, und weitere Daten gespeichert werden. Die Anwendung soll eine (möglichst schnelle) Suche nach Mitarbeitern durch ihren Namen ermöglichen.

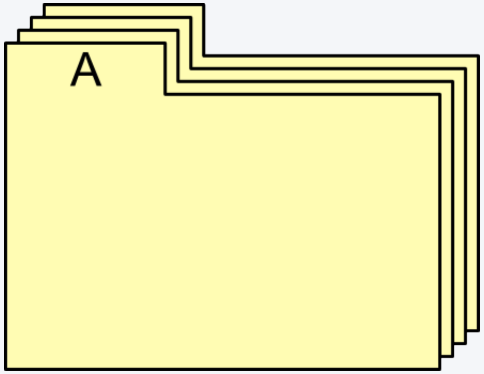
Naiver Ansatz

Alle Datensätze willkürlich in einer Liste speichern. Probleme:

- Suche mit linearem Aufwand verbunden
- im schlimmsten Fall müssen alle Elemente durchsucht werden
  - auch dann, wenn es gar keinen Mitarbeiter mit dem gesuchten Namen gibt!

Streutabellen: Motivation (II)

Besser: „Karteikasten mit Reitern“



- Idee einer sog. Indexstruktur
  - Streutabellen
  - sortierte Reihen
  - Suchbäume
  - ...

⇝ Position eines beliebigen Elements kann schnell bestimmt werden

Hashing: Grundlagen

- Hashing:
  - Daten werden in einer Streu(wert)tabelle (*hash table*) abgelegt
    - Verwendung eines Arrays auf Grund des wahlfreien Zugriffs
  - eine Hashfunktion *h* bildet ein Datenelement auf einen Hashwert ab
    - benötigt dazu einen Schlüssel (*key*), der Element eindeutig identifiziert
    - Hashwert wird als Index in die Tabelle verwendet

  $\rightsquigarrow$ Element wird in entsprechendem *Bucket* der Tabelle gespeichert
- Suche nach einem Element mit bekanntem Schlüssel:
  - Index mittels Hashfunktion bestimmen
    - konstanter Aufwand
  - Nachschlagen an entsprechender Stelle
    - Aufwand abhängig von Organisationsform, s.u.

Beispiel

Beispiel: Tabelle mit 8 Buckets

k	13	3	42	18
$h(k)$	0	2	5	6

Bucket	*Inhalt*
0	13
1	
2	3
3	
4	
5	42
6	18
7	

Hashfunktion

Hashfunktion

Die Hashfunktion h bildet Elemente der Schlüsselmenge $\mathcal{K}$ auf Elemente der Index-Menge $\mathcal{I} \subset \mathbb{Z}$ ab, d.h. $h : \mathcal{K} \rightarrow \mathcal{I}$.

Problem: Kollisionen

Im Allgemeinen ist $|\mathcal{I}| \ll |\mathcal{K}|$. Eine Hashfunktion ist daher im Allgemeinen nicht injektiv, d.h. mehrere unterschiedliche Schlüssel werden auf denselben Index abgebildet. In diesem Fall spricht man von sog. Kollisionen.

Beispiel

Sei $\mathcal{K} := \mathbb{N}$, $\mathcal{I} := \{0, \ldots, 8\}$. Sei außerdem $h(k) := (5 \cdot k)\%9$ die verwendete Hashfunktion. Die beiden Schlüssel $k_1 := 2$ und $k_2 := 11$ führen zu einer Kollision, denn $h(k_1) = h(k_2) = 1$. k_1 und k_2 müssten also im selben Bucket landen.

„Gute“ Hashfunktionen

- „Qualität“ einer Hashfunktion ist immer abhängig von der Anwendung
  - z.B.: für *Kryptographie* andere Anforderungen als für *Datenbank*
- Entwurf einer guten Hashfunktion:
  - Wissen über Verwendung
  - Wissen über Verteilung der Schlüssel
- „Standard“-Kriterien für gute Hashfunktionen:
  - Abbildung der Hashfunktion sollte unbedingt surjektiv sein

  $\rightsquigarrow$ jeder Index sollte berechnet werden können
  - möglichst Gleichverteilung der berechneten Indizes

  $\rightsquigarrow$ möglichst geringe Wahrscheinlichkeit für Kollisionen
  - Berechung der Hashfunktion sollte möglichst effizient sein

Belegungsfaktor / Lastfaktor

Belegungsfaktor / Lastfaktor

Bezeichne $\tilde{\mathcal{K}} \subset \mathcal{K}$ die Menge der tatsächlich eingetragenen Schlüssel. Dann ist

$$BF := \frac{|\tilde{\mathcal{K}}|}{|\mathcal{I}|}$$

der Belegungsfaktor (*auch*: Lastfaktor) der Streutabelle. Er ist ein Maß dafür, wie „voll" die Tabelle ist. Der Belegungsfaktor kann größer als 100% sein (siehe unten).

Beachte

- niedriger Lastfaktor ($<$ 50%) $\leadsto$ Verschwendung von Speicherplatz
- hoher Lastfaktor ($>$ 80%) $\leadsto$ höhere Wahrscheinlichkeit für Kollisionen

$\leadsto$ höherer Suchaufwand

$\leadsto$ aufwändige Reorganisation notwendig

Kollisionsauflösung

- haben gesehen: Kollisionen sind „völlig normal“

$\leadsto$ Streutabelle muss mit diesen umgehen können

$\leadsto$ Kollisionsauflösung:

- durch Verkettung
  - mehr als ein Element pro Bucket speichern
- durch Sondieren
  - Elemente bei Kollisionen „weiterschieben“
- ...

Kollisionsauflösung durch Verkettung

Kollisionsauflösung durch Verkettung (*separate chaining*)

Jedes *Bucket* speichert mit Hilfe einer dynamischen Datenstruktur (Liste, Baum, weitere Streutabelle, ...) alle Elemente mit dem entsprechenden Hashwert.

Vorteil

Streutabelle kann „beliebig“ viele Elemente speichern.

Suche nach einem Element

- Bestimmung des *Buckets* mittels Hashfunktion
- Suche in der Menge der in diesem Bucket gespeicherten Elemente

$\rightsquigarrow$ Suche nur in einer Teilmenge aller Elemente nötig

Gefahr

Entartung bei schlechter Hashfunktion oder bei zu vielen Elementen. (Belegungsfaktor $\gg$ 100% ist möglich, sollte aber unbedingt vermieden werden!)

Beispiel: Verkettung

Beispiel: Kollisionsauflösung durch Verkettung

$$h(k) = (5 \cdot k)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13 , 22
3	15
4	8
5	
6	12 , 3 , 21
7	
8	7

Beispiel: „Schlechte“ Hashfunktion

Beispiel: „Schlechte“ Hashfunktion

$h(k) = (4 \cdot k)\%8,\quad$ 4 und 8 nicht teilerfremd

k	13	8	7	12	3	15	22	21
$h(k)$	4	0	4	0	4	4	0	4

Bucket	*Inhalt*
0	8, 12, 22
1	
2	
3	
4	13, 7, 3, 15, 21
5	
6	
7	
8	

Kollisionsauflösung durch Sondieren (1)

Kollisionsauflösung durch Sondieren

- jedes *Bucket* speichert nur ein Element
- falls eigentliches *Bucket* bereits belegt: Einfügen an anderer, freier Stelle
- Bestimmung der Stelle nach definierten Regeln:
  - lineares Sondieren mit Schrittweite d
    - zyklisch d *Buckets* weitergehen, bis freies Feld gefunden wird
  - quadratisches Sondieren
    - 1, 4, 9, 16, ... *Buckets* weitergehen, bis freies Feld gefunden wird
  - doppeltes Hashen mit weiterer Hashfunktion
  - ...

Kollisionsauflösung durch Sondieren (2)

Suche nach einem Element

- Bestimmung des Indizes mittels Hashfunktion
- Nachschlagen an entsprechender Stelle in Streutabelle:
  - falls *Bucket* gesuchtes Element enthält: gefunden!
  - falls *Bucket* leer: Element nicht in Tabelle enthalten
  - sonst: neuen Index zyklisch berechnen und weitersuchen

Nachteile

- Streutabelle kann nur begrenzte Anzahl an Elementen speichern
- Löschen von Elementen wird kompliziert

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	
3	
4	
5	
6	
7	
8	

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	
4	
5	
6	
7	
8	

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	
4	8
5	
6	
7	
8	

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	
4	8
5	
6	
7	
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \;\; h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	
4	8
5	
6	12
7	
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \;\; h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	
4	8
5	
6	12, 3_0
7	
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	
4	8
5	
6	12
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	15
4	8
5	
6	12
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13, 22_0
3	15
4	8
5	
6	12
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \;\; h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	15, 22_1
4	8
5	
6	12
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \;\; h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13
3	15
4	8
5	
6	12, 22_2
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	
1	
2	13, 22_3
3	15
4	8
5	
6	12
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	22_4
1	
2	13
3	15
4	8
5	
6	12
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \;\; h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	22_4
1	
2	13
3	15
4	8
5	
6	12, 21_0
7	3_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	22_4
1	
2	13
3	15
4	8
5	
6	12
7	3_1, 21_1
8	7

Beispiel: Quadratisches Sondieren

Beispiel: Quadratisches Sondieren

$$h_0(k) = (5 \cdot k)\%9, \quad h_i(k) = (h_0(k) + i^2)\%9$$

k	13	8	7	12	3	15	22	21
$h(k)$	2	4	8	6	6	3	2	6

Bucket	*Inhalt*
0	22_4
1	21_2
2	13
3	15
4	8
5	
6	12
7	3_1
8	7

Methode hashCode()

- Methode `hashCode()`:
  - wird von Oberklasse `Object` geerbt
    - kann in Unterklassen überschrieben werden
  - wandelt Objekt der Klasse in einen ganzzahligen Wert um
    - wird von Klassen der Java-API zur Bestimmung des Hashwerts benutzt
- folgende Bedingungen sollten (*müssen*) erfüllt sein: `hashCode()` ...
  - liefert für ein und dasselbe Objekt immer denselben Wert, außer das Objekt wurde zwischenzeitlich verändert
  - liefert für zwei nach `equals()` gleiche Objekte denselben Wert
  - liefert für zwei <u>un</u>gleiche Objekte nach Möglichkeit andere Werte

Verhalten bei „Standard-Klassen“

java.lang.Object

Implementierung JVM-abhängig; Berechnung aus interner Speicheradresse *möglich*.

java.lang.Integer

hashCode() liefert einfach den Wert des Integers zurück.

java.lang.Long

```
public int hashCode() {
    return (int)(value ^ (value >>> 32));
}
```

java.lang.String

$$\texttt{hashCode(s)} = \sum_{i=0}^{\texttt{length(s)}-1} \left(s_i \cdot 31^{\texttt{length(s)}-1-i}\right)$$

Achtung bei negativen Werten...

Achtung...

hashCode() kann negative Werte als Ergebnis liefern.
Dies führt bei naiver Verwendung zu Problemen:

```
bucket_idx = element.hashCode() % table_size;
// bucket_idx negativ, falls element.hashCode() negativ!
```

Mögliche Lösung

```
int hc = element.hashCode();
bucket_idx = ((hc % table_size) + table_size) % table_size;
// bucket_idx sicher positiv
```

AVL-Bäume

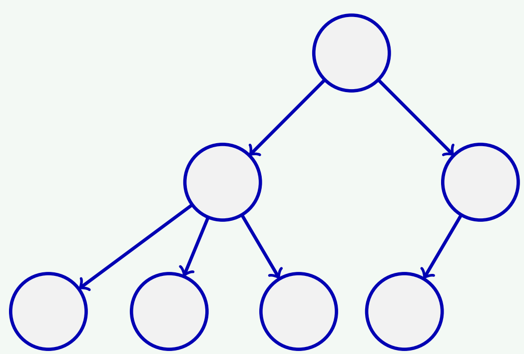
Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Allgemeine Bäume

- Baum: dyn. Datenstruktur zur Repräsentation hierarchischer Beziehungen
- besteht aus Knoten und Kanten
  - jeder Knoten ist entweder ein Blatt und hat keine Kinder oder ein innerer Knoten mit mindestens einem Kind
  - jeder Knoten hat genau einen Elternknoten oder ist die (einzige) Wurzel

Beispiel



Binärbäume und binäre Suchbäume

Binärbaum

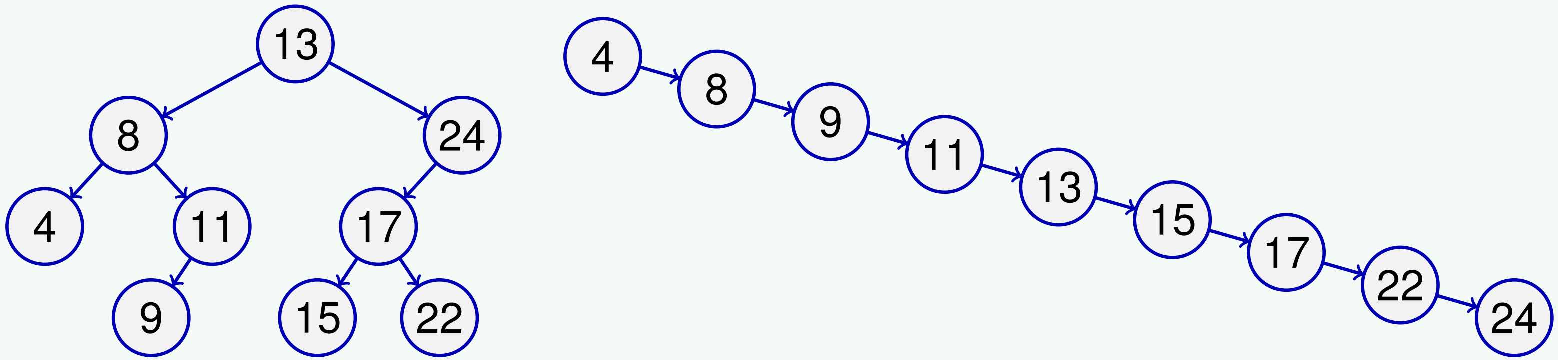
Ein Baum mit einem maximalen Verzweigungsfaktor von 2 heißt Binärbaum. Da jeder Knoten somit maximal zwei Kinder hat, spricht man auch vom linken und rechten Kind.

Binärer Suchbaum

Ein Binärer Suchbaum ist ein Binärbaum, der die Suchbaumeigenschaft erfüllt: Für jeden Knoten mit Wert X im Baum gilt, dass (bzgl. einer beliebigen, aber festen Ordnungsrelation) der linke Teilbaum nur kleinere Werte als X und der rechte Teilbaum nur größere Werte als X beinhaltet.

Entartete Suchbäume

Beispiel: Zwei Suchbäume mit denselben Werten...



Beobachtung

- ein Suchbaum ist nicht zwangsläufig balanciert
- Ausprägung eines Suchbaums hängt von Einfügereihenfolge ab

Problem? Problem!

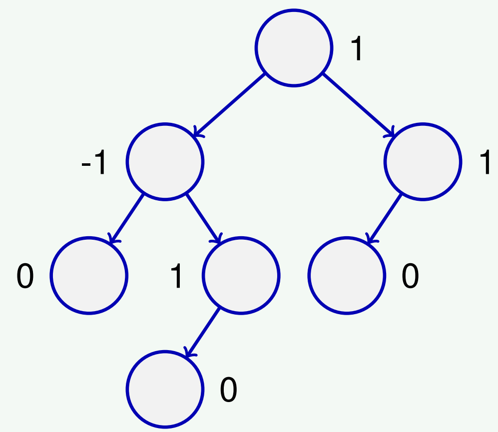
Je schlechter ein Suchbaum balanciert ist, desto aufwändiger sind i.A. die typischen Suchbaum-Operationen, d.h. das Suchen, Einfügen und Löschen von Knoten.

Balancefaktor

Balancefaktor

Der Balancefaktor eines Knotens bezeichnet die Differenz zwischen der Höhe des linken Teilbaums und der des rechten Teilbaums. Blätter haben einen Balancefaktor von 0.

Beispiel

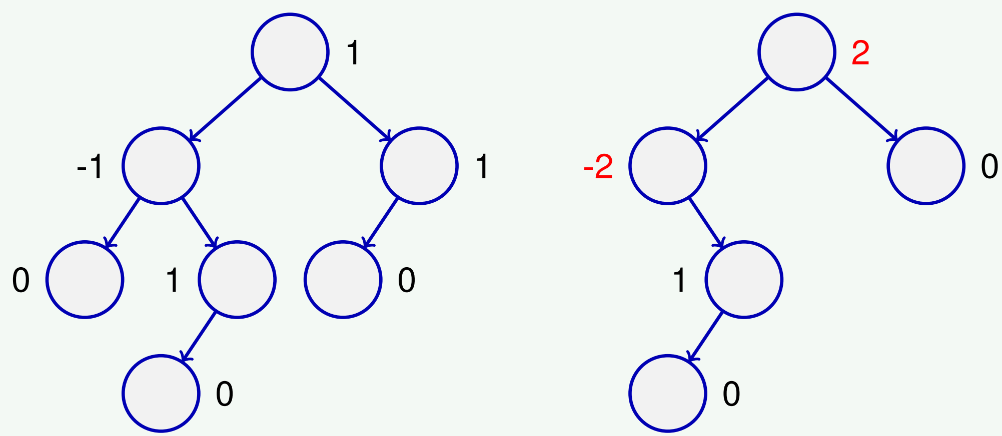


AVL-Bäume

AVL-Baum

Ein AVL-Baum ist eine spezielle Form eines binären Suchbaums, welcher immer möglichst gut balanciert ist. Dazu stellt der AVL-Baum beim Einfügen und Löschen von Knoten sicher, dass die Balancefaktoren aller Knoten betragsmäßig stets ≤ 1 sind.

Beispiel (links: AVL-Baum, rechts: kein AVL-Baum)



AVL-Bäume: Operationen

Operationen auf einem AVL-Baum

- die Operationen *Suchen*, *Einfügen* und *Löschen* funktionieren prinzipiell wie auf einem klassischen Suchbaum
- aber: wenn ein Element hinzugefügt oder gelöscht wird, dann kann sich die Höhe eines (Teil-)Baums um 1 verändern

$\rightsquigarrow$ alle Balancefaktoren müssen neu berechnet werden

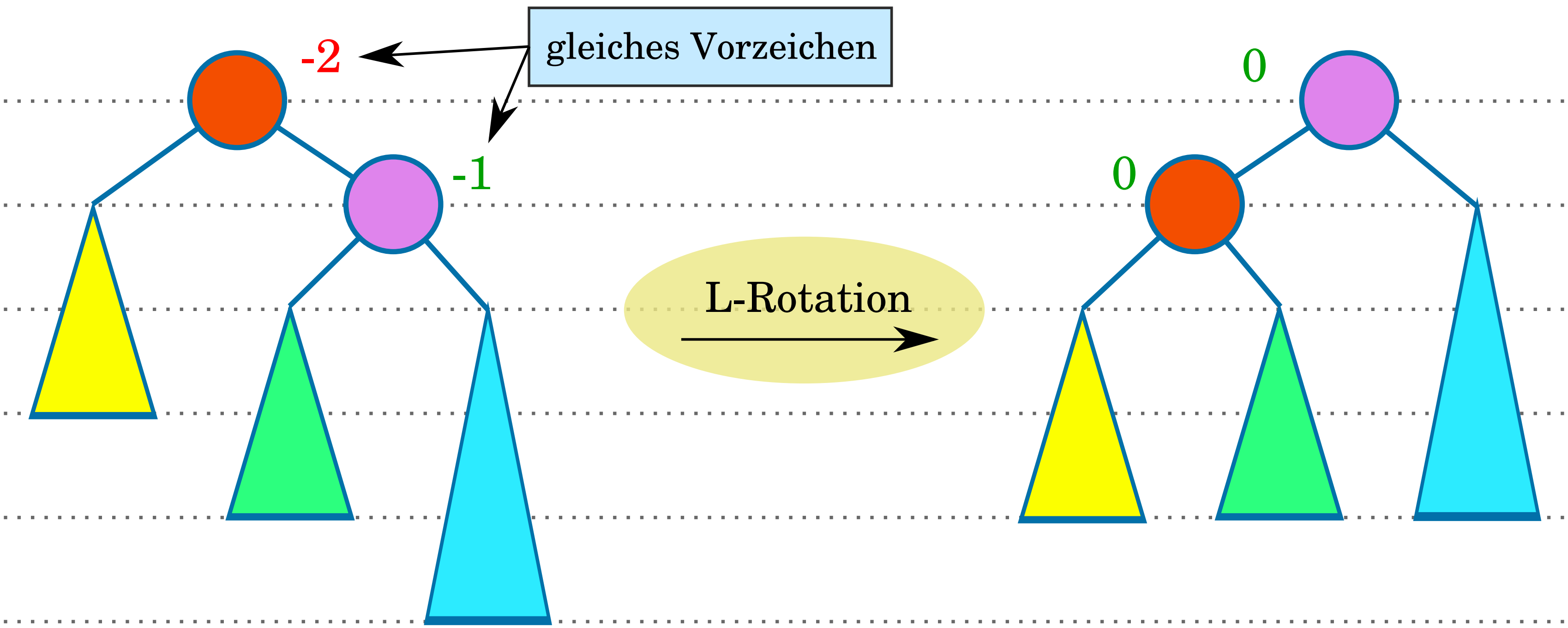
$\rightsquigarrow$ falls mindestens ein Balancefaktor betragsmäßig > 1 ist

  $\rightsquigarrow$ Rebalancierung durch Einzel- oder Doppel-Rotationen erforderlich

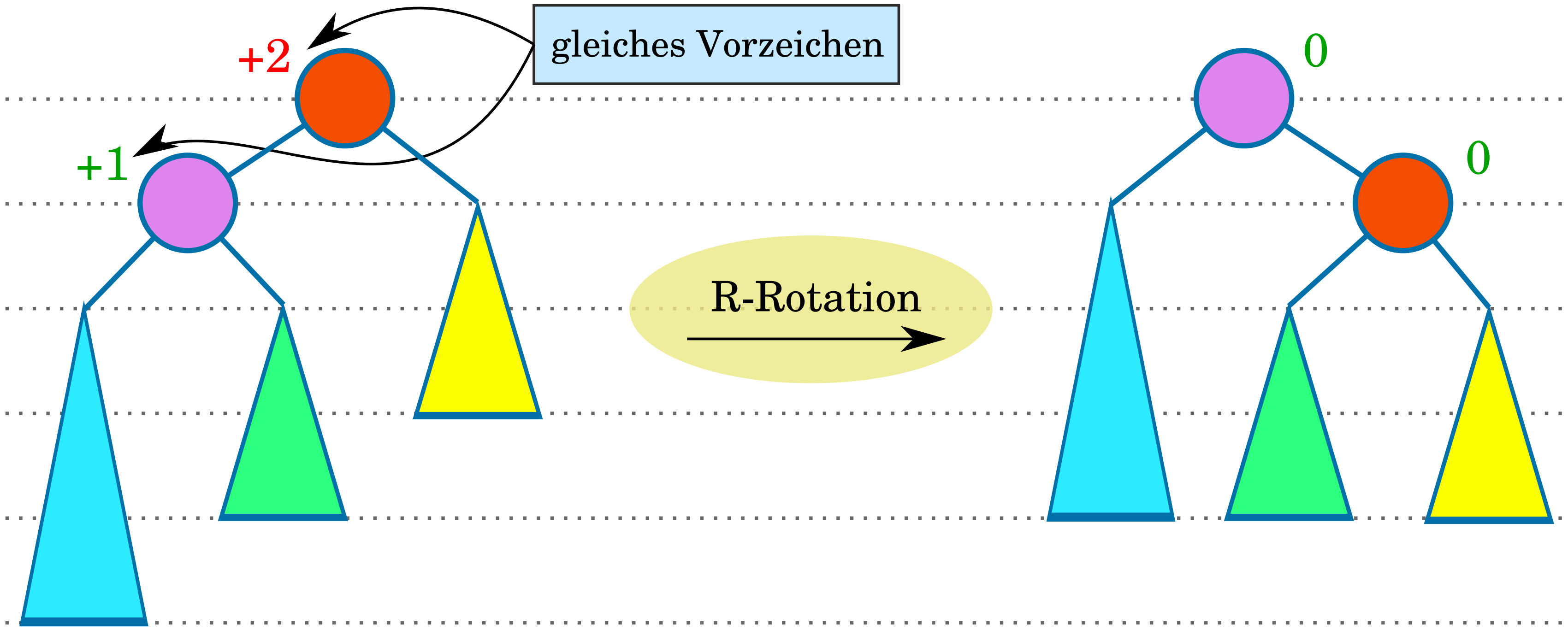
Achtung

Die Balance muss nach *jeder* einzelnen Einfüge- oder Lösch-Operation wiederhergestellt werden – auch, wenn mehrere Werte hintereinander eingefügt/gelöscht werden.

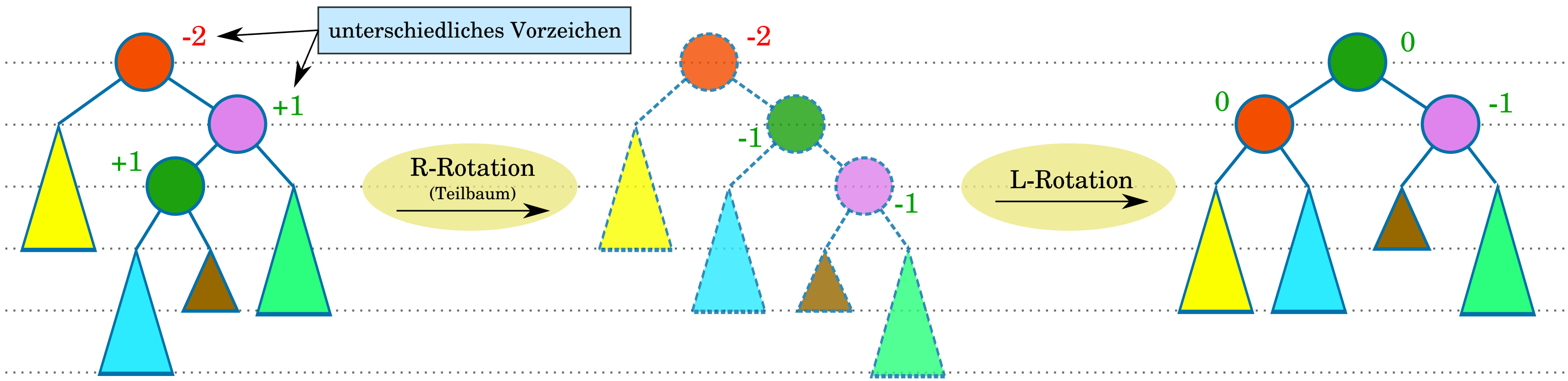
Einzel-Rotation: L-Rotation



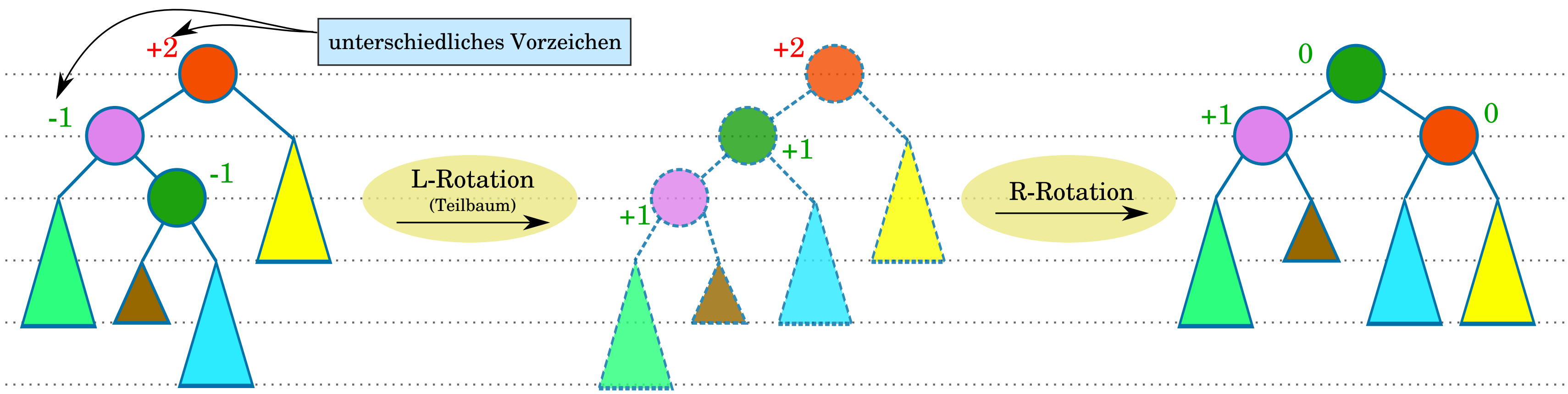
Einzel-Rotation: R-Rotation



Doppel-Rotation: RL-Rotation



Doppel-Rotation: LR-Rotation



Tafel-Beispiel

Tafel-Beispiel

Fügen Sie die folgenden Werte in einen anfangs leeren AVL-Baum ein:

- 10, 16, 20, 7, 2, 15, 11

Löschen Sie anschließend die 2 wieder aus dem Baum.

Halden

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Heaps (Halden)

- Heap $\equiv$ dynamische Datenstruktur
  - partiell geordneter Binärbaum
    - Max-Heap:
      - Wurzel jedes Teilbaums ist größer als andere Knoten des Teilbaums
    - Min-Heap:
      - Wurzel jedes Teilbaums ist kleiner als andere Knoten des Teilbaums

$\leadsto$ schneller Zugriff auf das größte bzw. kleinste Element

  $\leadsto$ Verwendung als Prioritätswarteschlange

  - Implementierung als „klassischer" Binärbaum möglich
    - aber: effiziente Speicherung in Array möglich (s.u.)

Links-Vollständigkeit

- in AuD versteht man unter Heaps links-vollständige Bäume:
  - alle „Ebenen“ (bis auf die unterste) sind voll besetzt
  - auf unterster „Ebene“ sitzen alle Knoten soweit links wie möglich

$\leadsto$ lückenlose Darstellung in einem Array möglich (sog. Feld-Einbettung)

Berechnung der Indizes

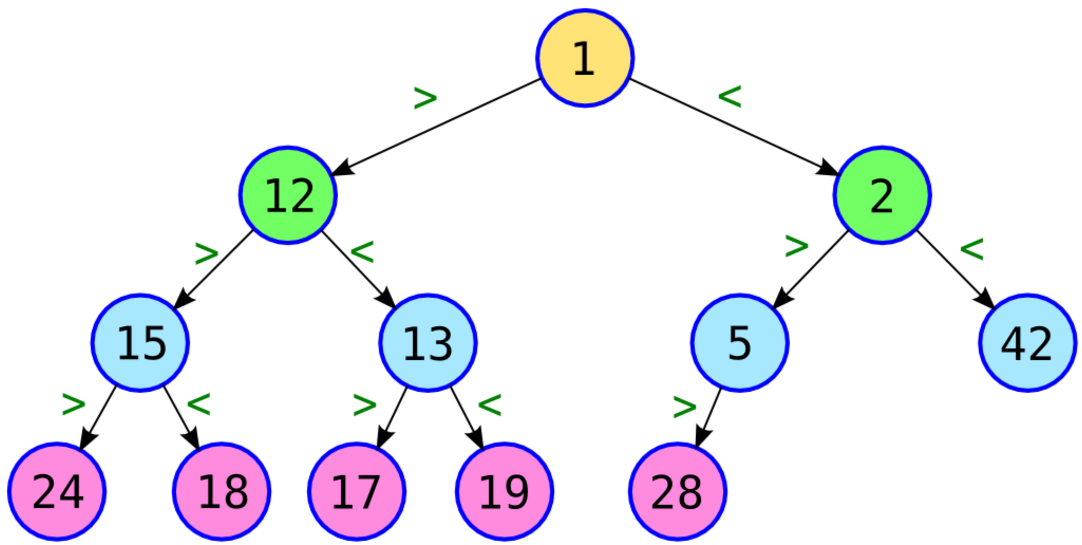
- Wurzel steht an Position 0
- Kinder von Knoten an Position i stehen an Stelle $2 \cdot i + 1$ und $2 \cdot i + 2$
- Elternknoten von Knoten an Position i steht an Stelle $\lfloor \frac{i-1}{2} \rfloor$

Hinweis

Nicht nur Halden können linksvollständig sein, sondern jeder beliebige Baum.

Beispiel

Beispiel für einen Min-Heap und seine Feld-Einbettung



1	12	2	15	13	5	42	24	18	17	19	28

Heap: Operationen

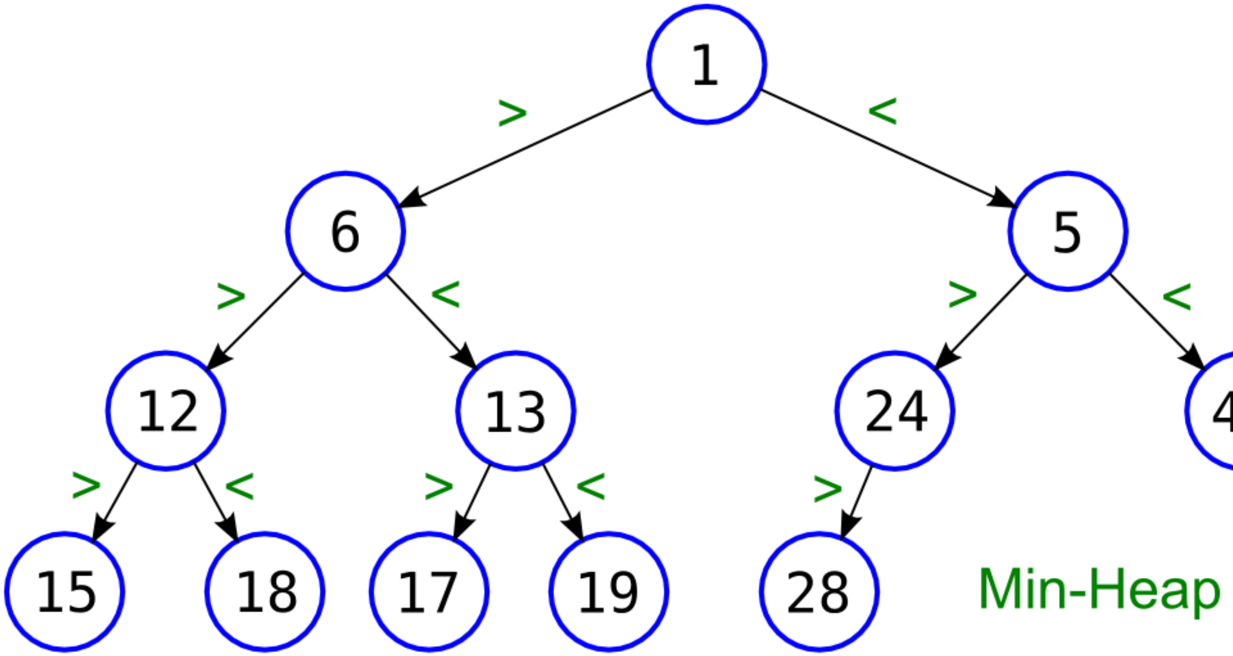
Einfügen eines neuen Elements

- Element an die nächste freie Position in der untersten Ebene einfügen
  - falls Ebene voll: Element wird erster Knoten einer neuen Ebene
- solange Halden-Eigenschaft in einem Teilbaum verletzt ist:
  - Element entsprechend der Halden-Eigenschaft nach oben wandern lassen

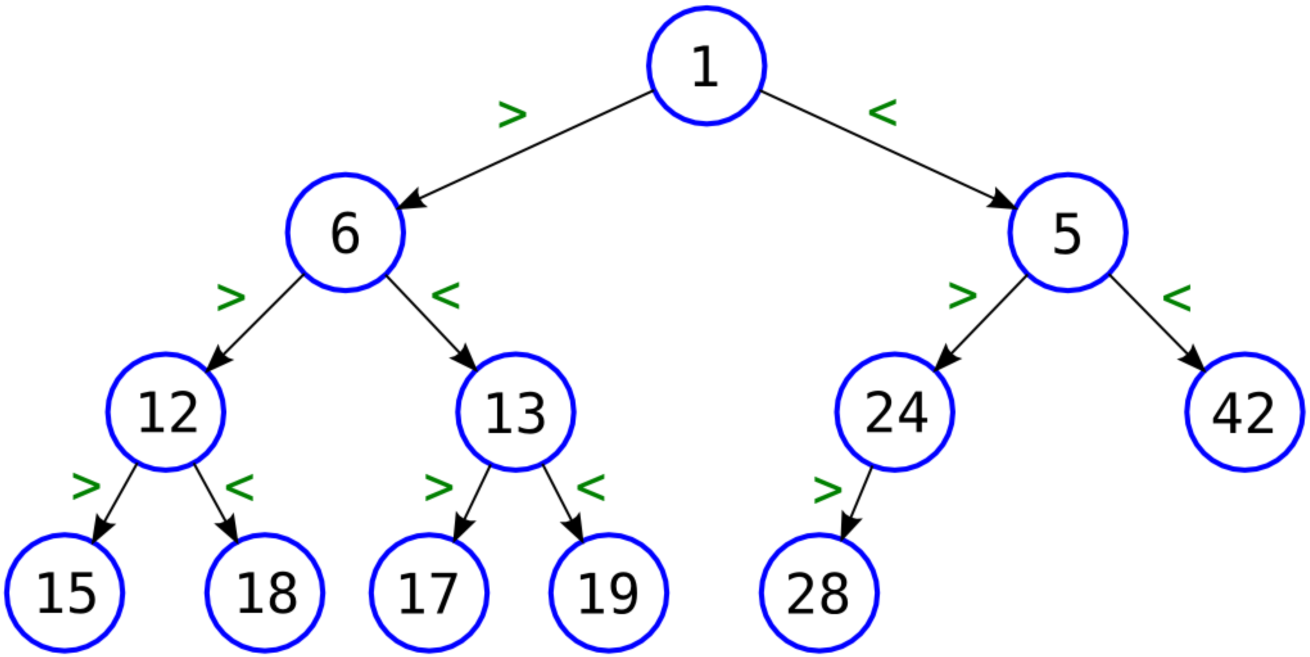
Löschen eines Elements

- zu löschendes Element mit dem letzten Element ersetzen
  - letztes Element der untersten Ebene
- solange Halden-Eigenschaft in einem Teilbaum verletzt ist:
  - Element entsprechend der Halden-Eigenschaft nach unten wandern lassen
    - Min-Heap: Tauschen mit kleinerem Kind
    - Max-Heap: Tauschen mit größerem Kind

Heap: Beispiel

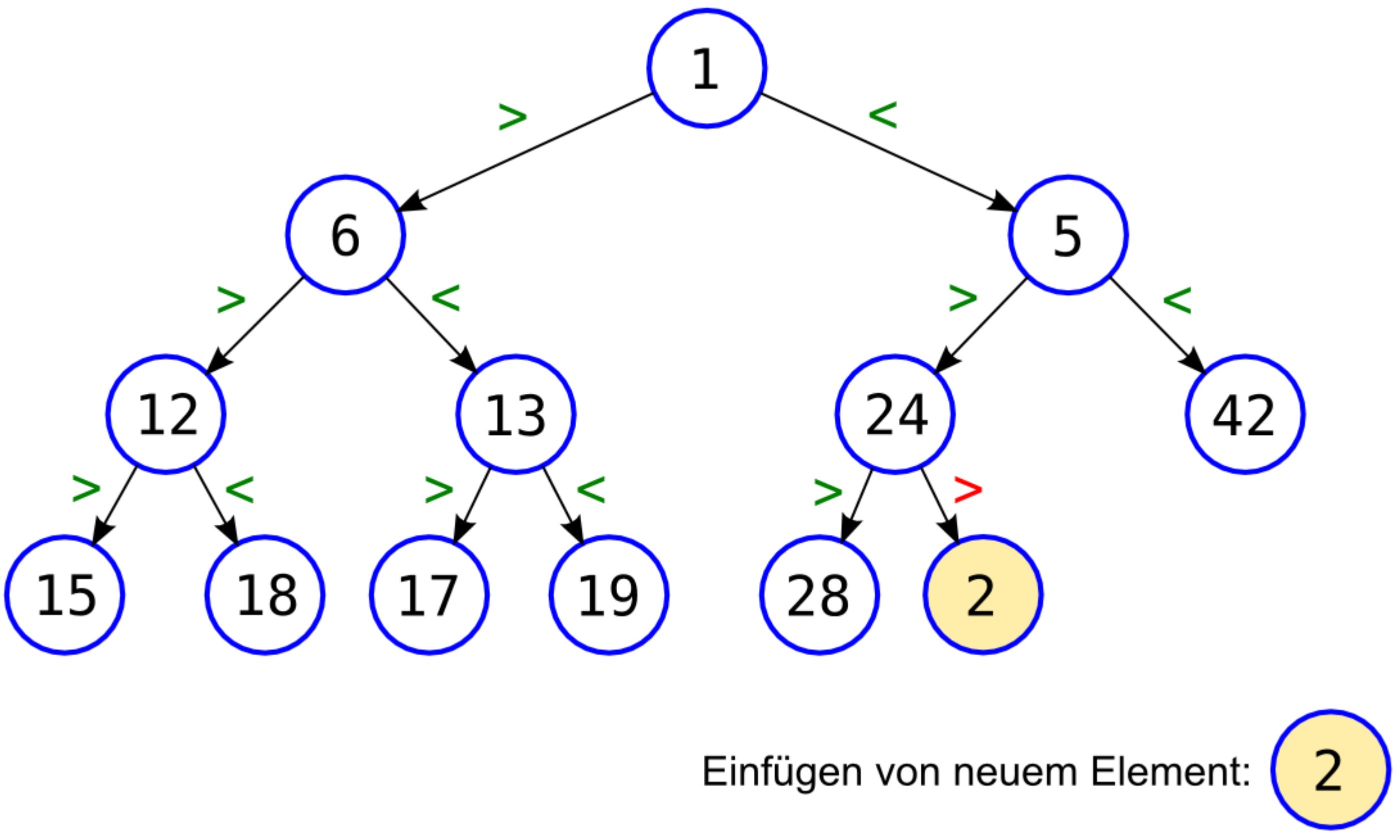


Heap: Einfügen

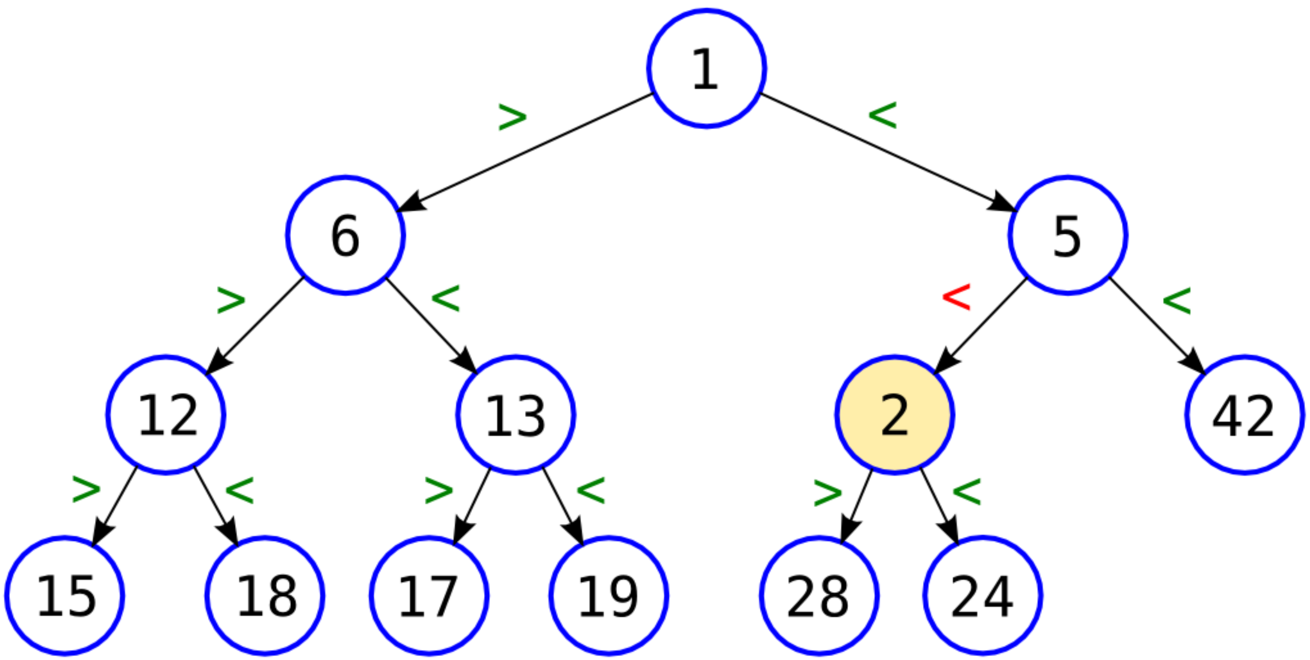


Einfügen von neuem Element: 2

Heap: Einfügen

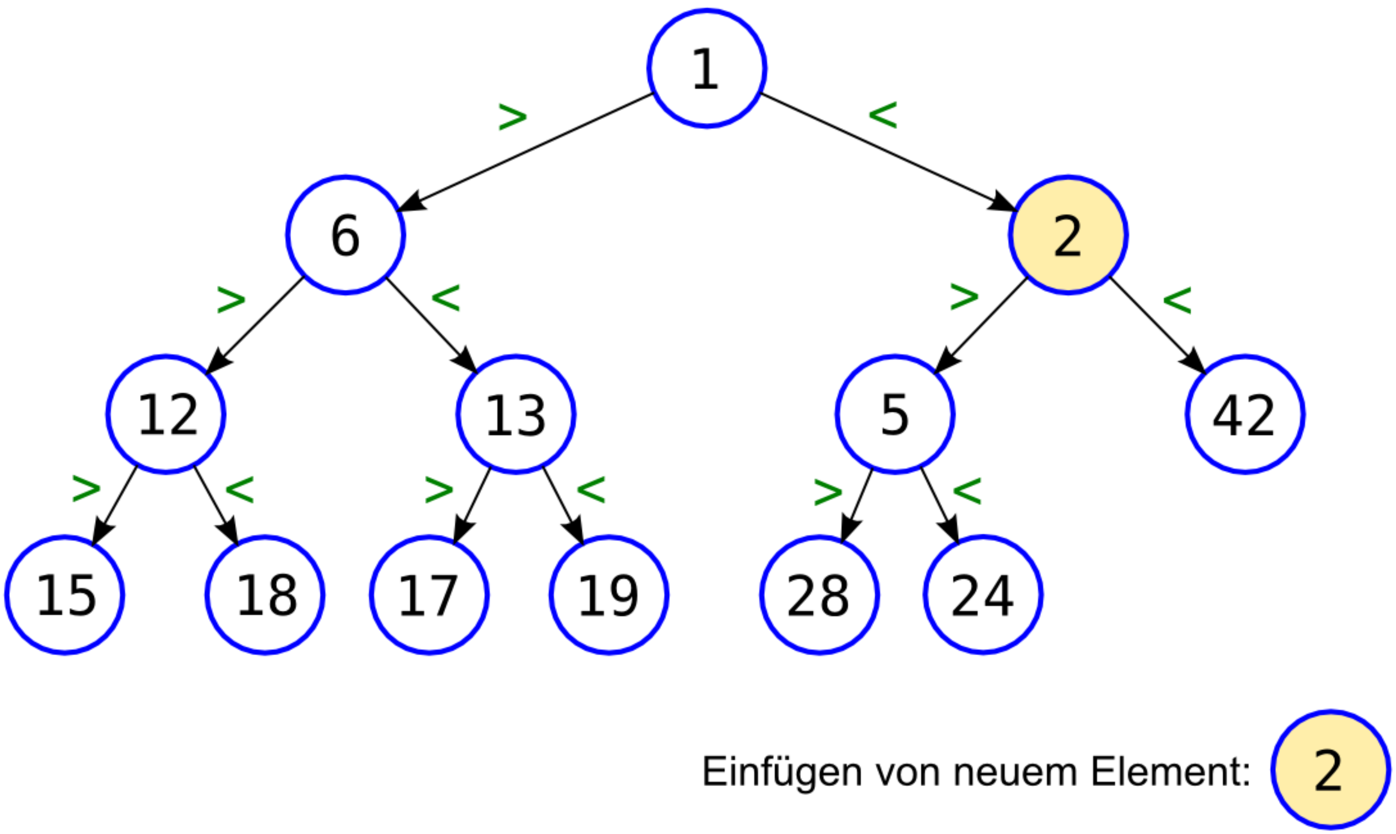


Heap: Einfügen

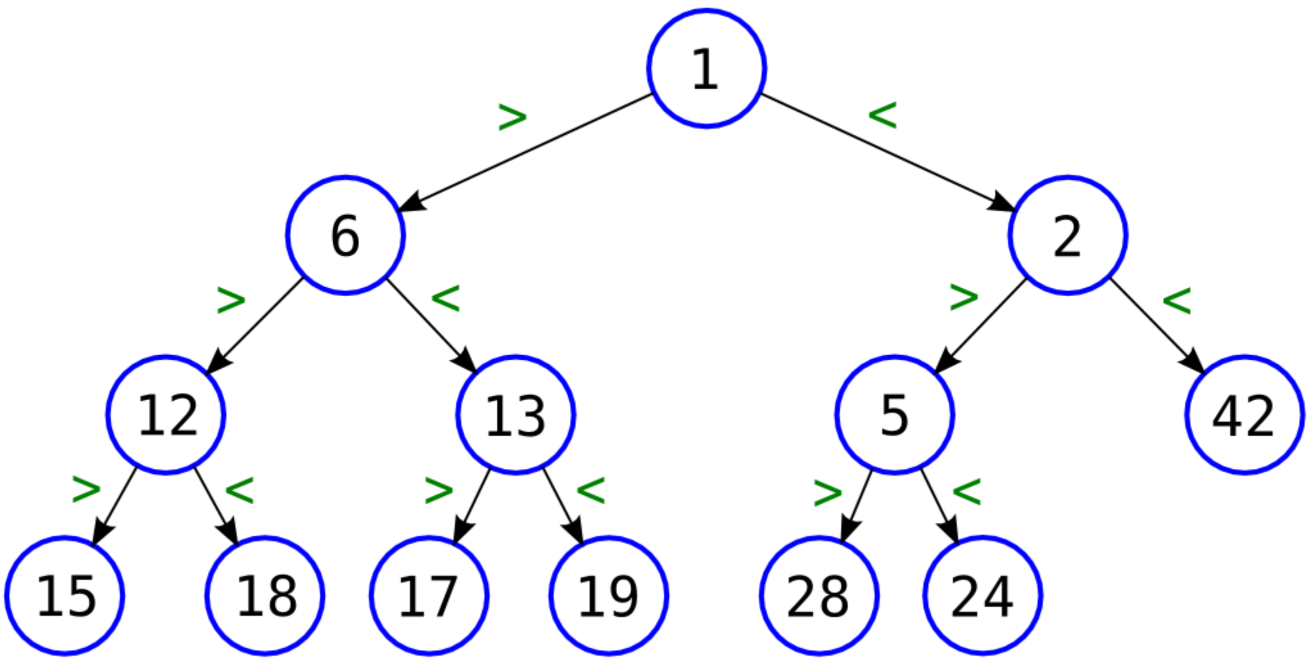


Einfügen von neuem Element: 2

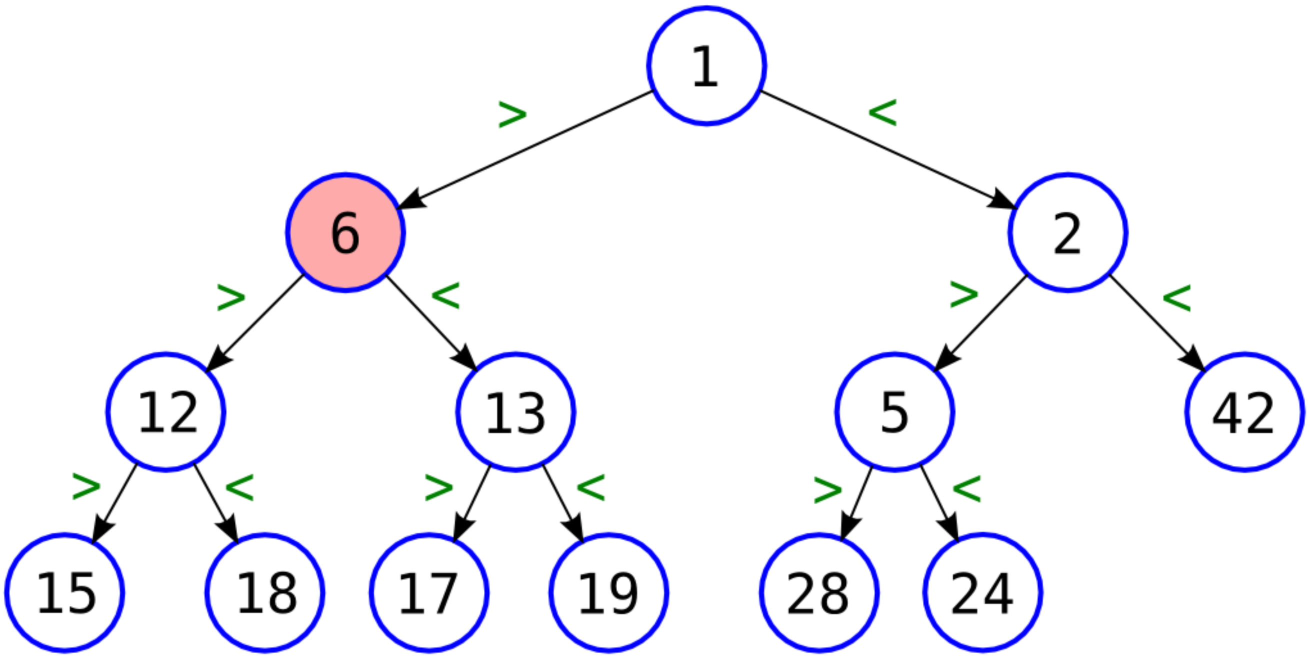
Heap: Einfügen



Heap: Einfügen

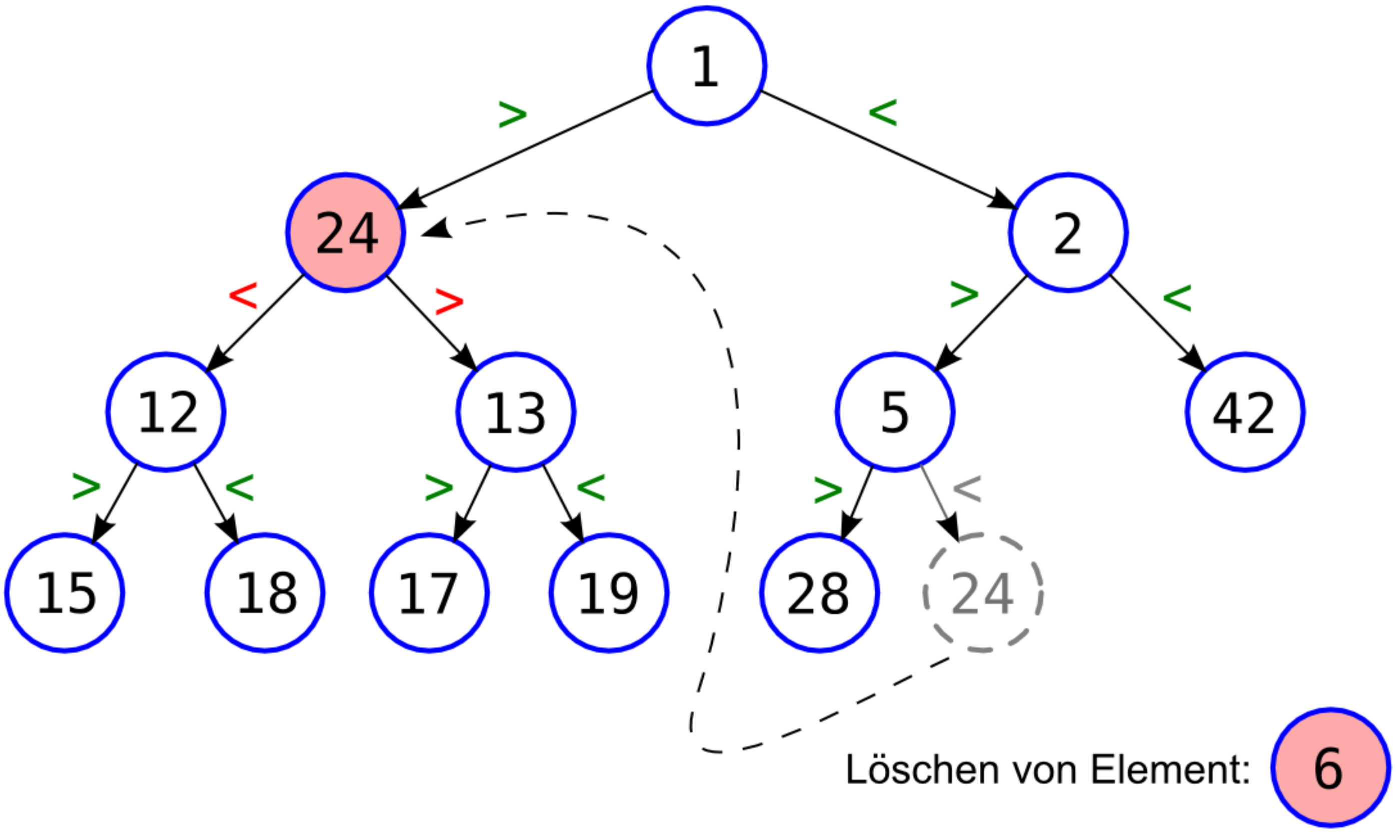


Heap: Löschen

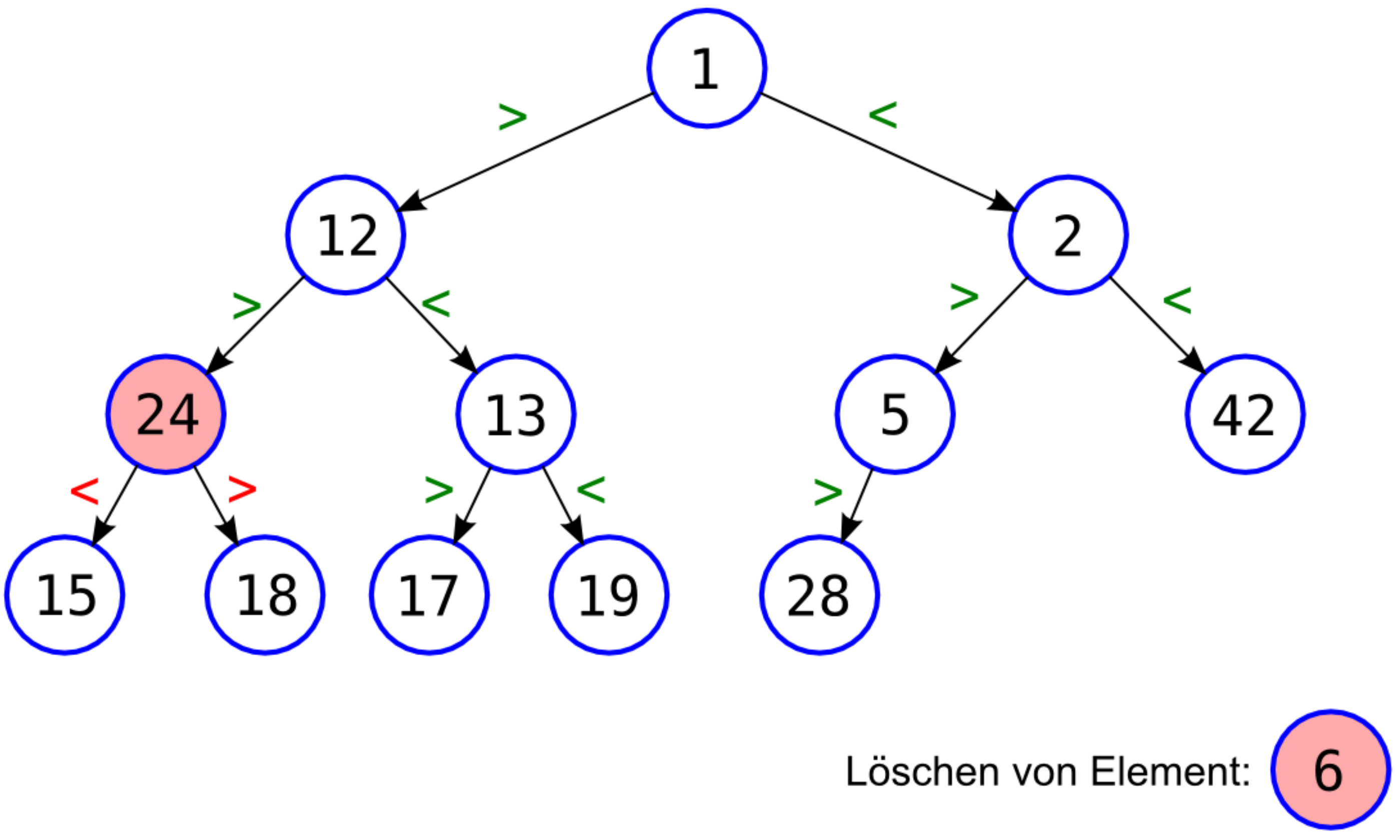


Löschen von Element: 6

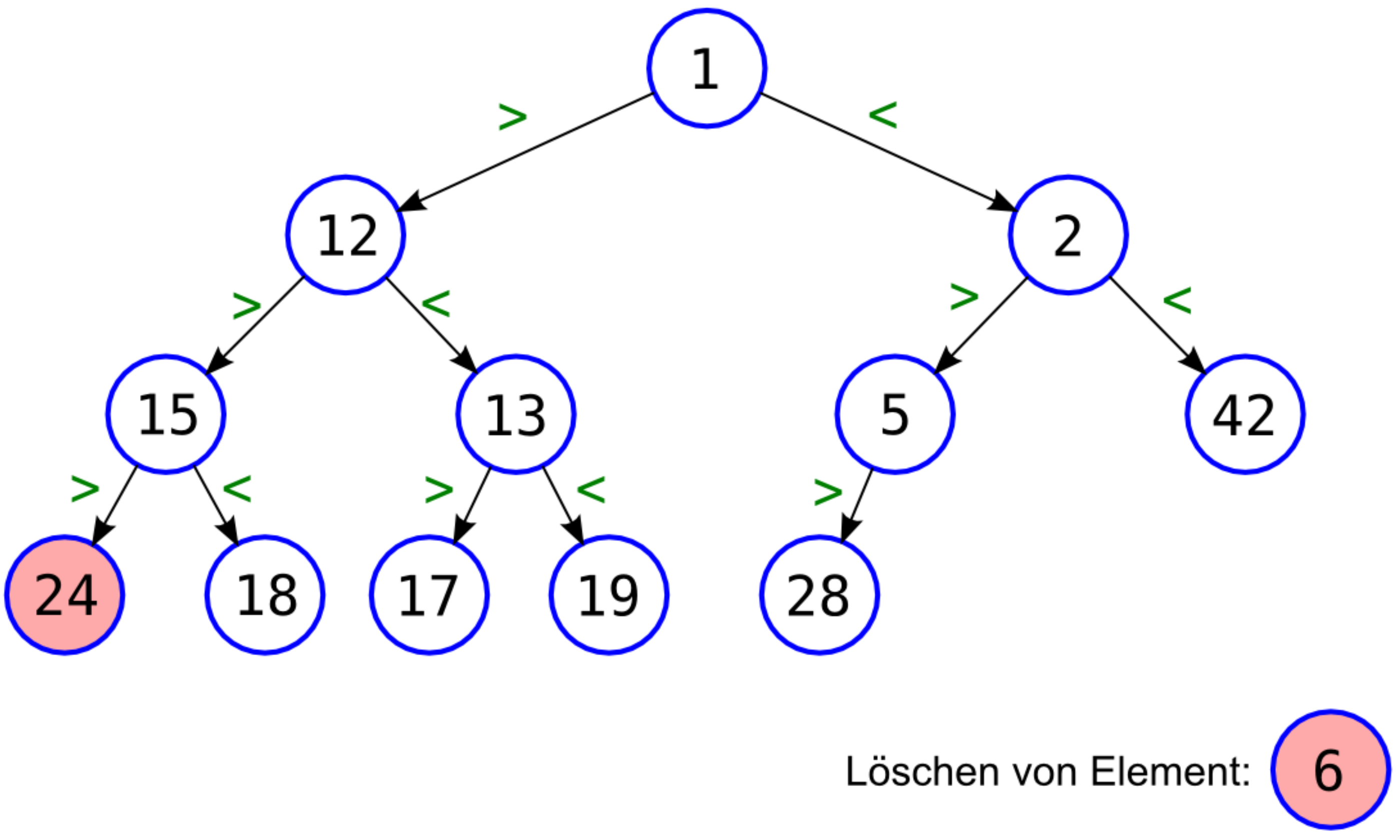
Heap: Löschen



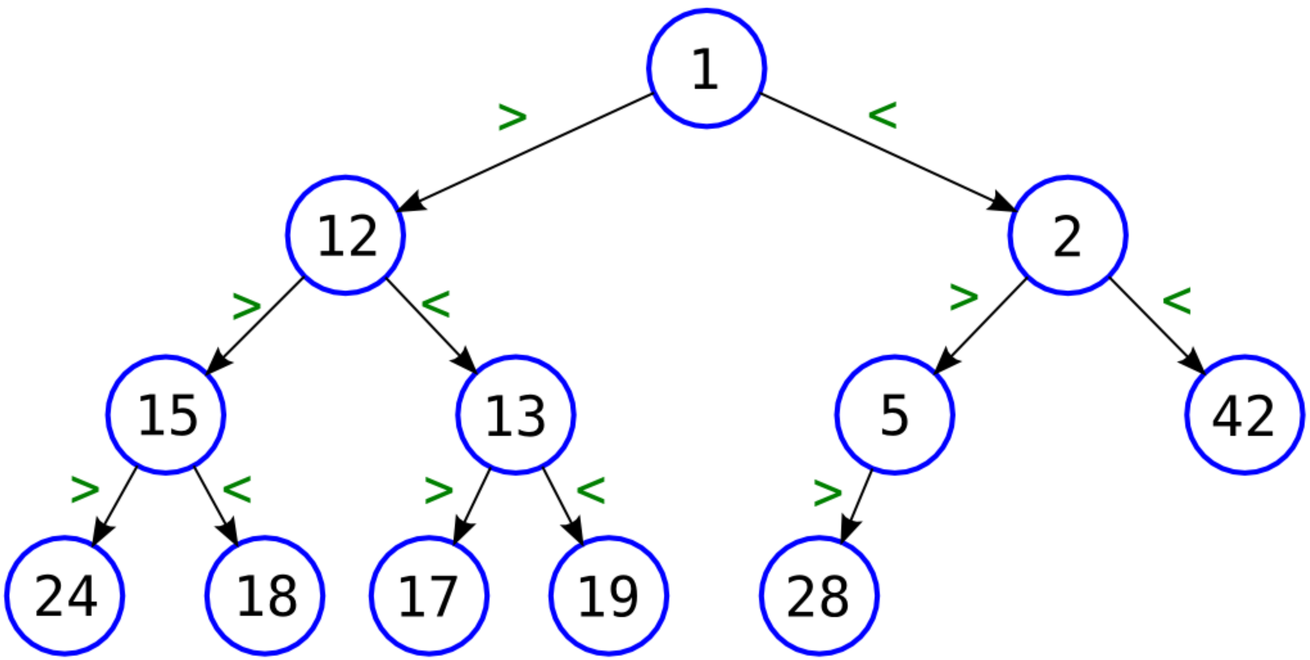
Heap: Löschen



Heap: Löschen



Heap: Löschen



Bäume malen

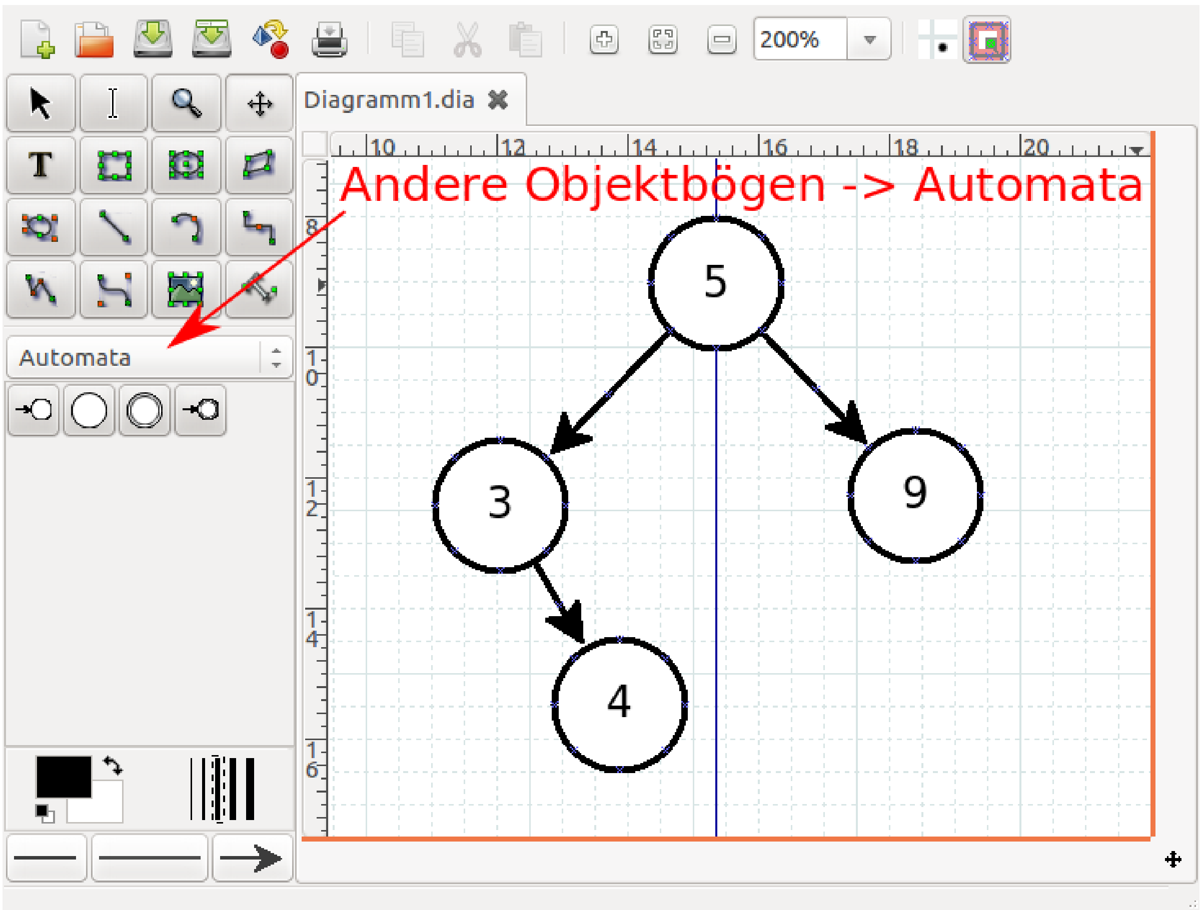
Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Dia

- Open-Source-Software (Lizenz: *GNU GPL*)
  - im CIP bereits installiert
  - Versionen für diverse Systeme vorhanden
- kann Diagramme verschiedenster Art erzeugen
  - Ablaufdiagramme
  - UML-Diagramme
  - ...
- eignet sich auch für die Darstellung von Bäumen und Graphen
- unterstützt den PDF-Export
- hat ein paar Schwächen, aber erfüllt seinen Zweck
- Download und weitere Informationen unter `http://dia-installer.de/`

Passenden „Objektbogen“ wählen



Bäume malen mit Dia

- für Knoten „Intermediate State“ auswählen
  - Knotenbeschriftung nicht vergessen
- für Kanten „Linie (L)“ auswählen

Hinweis

- auch andere Objektbögen sind denkbar
  - „Automata“ ist aber für die Darstellung von Graphen und Bäumen gut geeignet

DOT

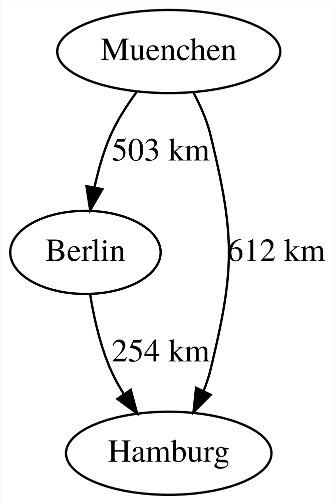
- Beschreibungssprache zur Definition von Graphen
  - Beispiel für eine sog. *Domain Specific Language*
- Graph wird in Form einer textuellen Beschreibung erstellt
  - ein Compiler (`dot` aus dem `graphviz`-Paket) generiert daraus eine Grafik
- Nachteil:
  - höhere Einarbeitungszeit als bei Dia
- Vorteile:
  - Graphen/Bäume können sehr schnell erzeugt werden
  - Graphen/Bäume können durch Programm erzeugt werden
  - automatische Anordnen der Knoten
  - automatisches *Routing*
  - ...

DOT: Beispiel

Beispiel-Graph: distanzen.dot

```
digraph {
  Berlin;
  M [label="Muenchen"];
  H [label="Hamburg"];
  M -> Berlin [label="503 km"];
  Berlin -> H [label="254 km"];
  M -> H [label="612 km"];
}
```

Ergebnis



PDF-Datei aus Beschreibung erzeugen

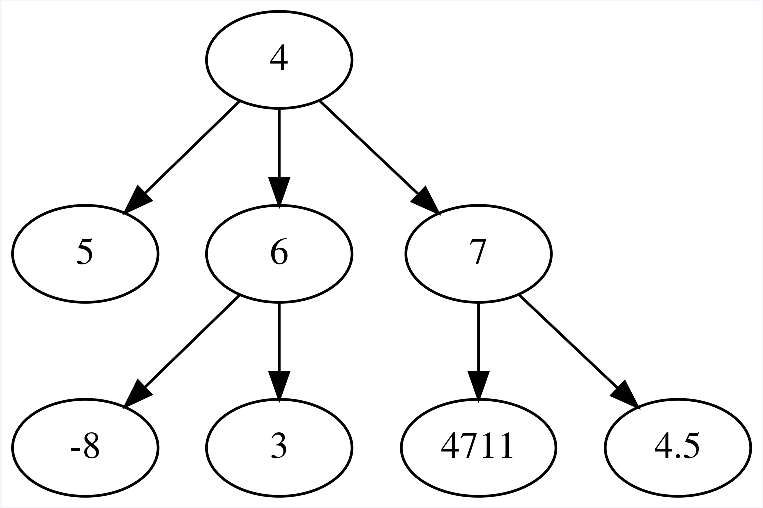
```
$> dot distanzen.dot -Tpdf -o distanzen.pdf
```

Allgemeiner Baum in DOT (I)

Allgemeiner Baum in DOT

```
digraph {
  4 -> 5;
  4 -> 6;
  4 -> 7;
  6 -> -8;
  6 -> 3;
  7 -> 4711;
  7 -> 4.5;
}
```

Ergebnis



Syntax

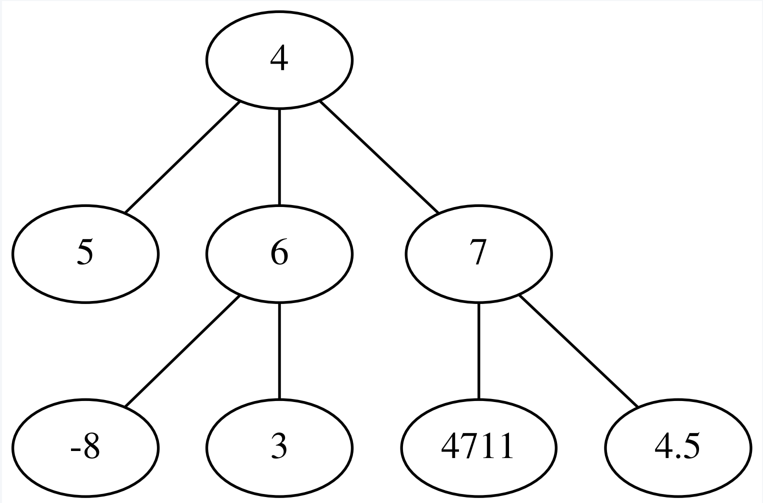
- `digraph` $\leadsto$ gerichteter Graph/Baum
- `4 -> 5` $\leadsto$ gerichtete Kante von Knoten 4 zu Knoten 5

Allgemeiner Baum in DOT (II)

Ungerichteter Baum in DOT

```
graph {
    4 -- 5;
    4 -- 6;
    4 -- 7;
    6 -- -8;
    6 -- 3;
    7 -- 4711;
    7 -- 4.5;
}
```

Ergebnis



Syntax

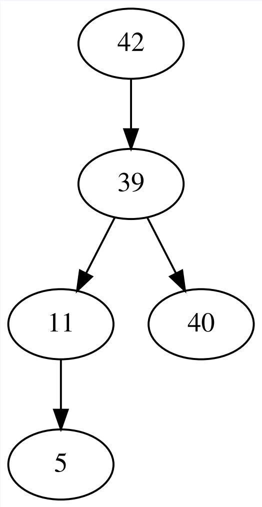
- `graph` $\rightsquigarrow$ ungerichteter Graph/Baum
- `4 -- 5` $\rightsquigarrow$ ungerichtete Kante von Knoten 4 zu Knoten 5

Binärer Suchbaum in DOT (I)

Binärer Suchaum in DOT

```
digraph {
    42 -> 39;
    39 -> 11;
    39 -> 40;
    11 -> 5;
}
```

Ergebnis



Problem? Problem!

Bei Knoten mit nur einem Kind ist nicht erkennbar, ob es sich um ein linkes oder ein rechtes Kind handelt. Außerdem *kann* es passieren, dass DOT Knoten umsortiert und dabei linkes und rechtes Kind vertauscht.

Binärer Suchaum in DOT (II)

Lösung

- an den erforderlichen Stellen unsichtbare Knoten und Kanten einfügen

$\rightsquigarrow$ überall da, wo ein Knoten „verrutscht“ ist

- dafür sorgen, dass linkes und rechtes Kind nicht vertauscht werden

Syntax

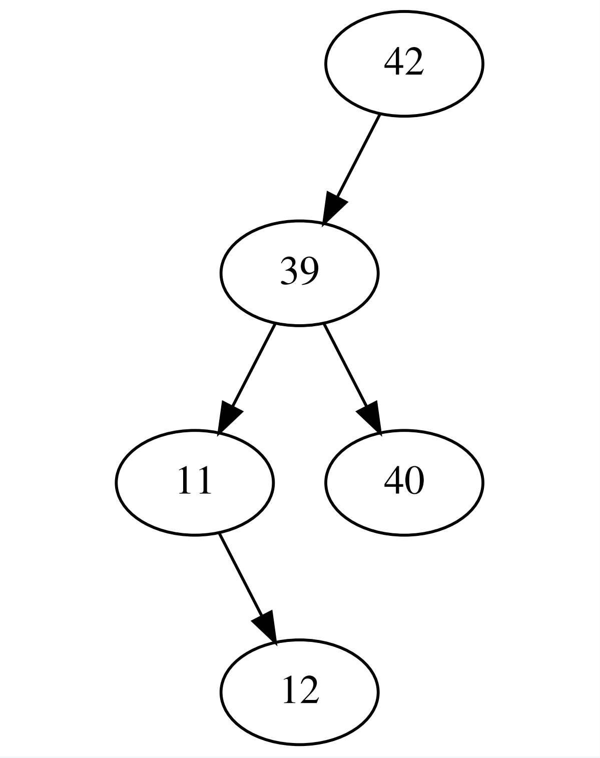
- `R42 [style=invis]` erzeugt einen unsichtbaren Knoten („rechtes Kind von 42“)
- `42 -> R42 [style=invis]` erzeught eine unsichtbare Kante von Knoten 42 zu dem unsichtbaren rechten Kind
- `graph [ordering="out"]` sorgt dafür, dass die Knoten in der Reihenfolge ihrer Definition angeordnet werden

Binärer Suchaum in DOT (III)

Besser

```
digraph {
  graph [ordering="out"]
    // links/rechts nicht vertauschen
  42 -> 39;
  R42 [style=invis];
    // unsichtbares rechtes Kind von 42
  42 -> R42 [style=invis];
    // unsichtbare Verbindung
  39 -> 11;
  39 -> 40;
  L11 [style=invis];
    // unsichtbares linkes Kind von 11
  11 -> L11 [style=invis];
    // unsichtbare Verbindung
  11 -> 12;
}
```

Ergebnis



Fragen? Fragen!

(hilft auch den anderen)