

Algorithmen und Datenstrukturen

Jonas Milkovits

Last Edited: 16. Mai 2020

Inhaltsverzeichnis

1	Einleitung	1
1.1	Probleme in der Informatik	1
1.2	Definitionen für Algorithmen	1
2	Sortieren	2
2.1	Einführung ins Sortieren	2
2.2	Analyse von Algorithmen - Teil 1	3
2.3	Analyse von Algorithmen - Teil 2	3
2.4	Analyse von Algorithmen - Teil 3	4
2.5	Insertion Sort	7
2.6	Bubble Sort	8
2.7	Selection Sort	10
2.8	Divide-And-Conquer-Ansatz	10
2.9	Merge Sort	10
2.10	Quicksort	12
2.11	Laufzeitanalyse von rekursiven Algorithmen	14
3	Grundlegende Datenstrukturen	17
3.1	Stacks	17
3.2	Verkettete Listen	18
3.3	Queues	20
3.4	Binäre Bäume	22
3.5	Binäre Suchbäume	25

1 Einleitung

1.1 Probleme in der Informatik

- Problem im Sinne der Informatik
 - Enthält eine Beschreibung der Eingabe
 - Enthält eine Beschreibung der Ausgabe
 - Gibt **keinen** Übergang von Eingabe und Ausgabe an
 - z.B.: Finde den kürzesten Weg zwischen zwei Orten
- Probleminstanzen
 - Probleminstanz ist eine konkrete Eingabenbelegung, für die entsprechende Ausgabe gewünscht ist
 - z.B.: Was ist der kürzeste Weg vom Audimax in die Mensa?

1.2 Definitionen für Algorithmen

- Begriff des Algorithmus
 - Endliche Folge von Rechenschritten, der eine Ausgabe in eine Eingabe verwandelt
- Anforderungen an Algorithmen
 - Spezifizierung der Eingabe und Ausgabe
 - Anzahl und Typen aller Elemente ist definiert
 - Eindeutigkeit
 - Jeder Einzelschritt ist klar definiert und ausführbar
 - Die Reihenfolge der Einzelschritte ist festgelegt
 - Eindlichkeit
 - Notation hat eine endliche Länge
- Eigenschaften von Algorithmen
 - Determiniertheit
 - Für gleiche Eingabe stets die gleiche Ausgabe (andere mögliche Zwischenzustände)
 - Determinismus
 - Für gleiche Eingabe stets identische Ausführung und Ausgabe
 - Terminierung
 - Algorithmus läuft für jede Eingabe nur endlich lange
 - Korrektheit
 - Algorithmus berechnet stets die spezifizierte Ausgabe (falls dieser terminiert)
 - Effizienz
 - Sparsamkeit im Ressourcenverbrauch (Zeit, Speicher, Energie,...)

2 Sortieren

2.1 Einführung ins Sortieren

- **Das Sortierproblem**

- Ausgangspunkt: Folge von Datensätzen D_1, D_2, \dots, D_n
- Zu sortierende Elemente heißen auch Schlüssel(werte)
- Ziel: Datensätze so anzuordnen, dass die Schlüsselwerte sukzessive ansteigen/absteigen
- Bedingung: Schlüsselwerte müssen vergleichbar sein
- Durchführung:
 - Eingabe: Sequenz von Schlüsselwerten $\langle a_1, a_2, \dots, a_n \rangle$
 - Eingabe ist eine **Instanz** des Sortierproblems
 - Ausgabe: Permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ derselben Folge mit Eigenschaft $a'_1 \leq \dots \leq a'_n$
- Algorithmus **korrekt**, wenn dieser das Problem für alle Instanzen löst

- **Exkurs: Totale Ordnung**

- Sei M eine nicht leere Menge und $\leq \subseteq M \times M$ eine binäre Relation auf M
- Das Paar (M, \leq) heißt genau dann totale Relation auf der Menge M , wenn Folgendes erfüllt ist:
 - Reflexivität: $\forall x \in M : x \leq x$
 - Transitivität: $\forall x, y, z \in M : x \leq y \wedge y \leq z \Rightarrow x \leq z$
 - Antisymmetrie: $\forall x, y \in M : x \leq y \wedge y \leq x \Rightarrow x = y$
 - Totalität: $\forall x, y \in M : x \leq y \vee y \leq x$
- z.B.: \leq Ordnung auf natürlichen Zahlen bildet eine totale Ordnung ($1 \leq 2 \leq 3 \dots$)
- z.B.: Lexikographische Ordnung \leq_{lex} ist eine totale Ordnung ($A \leq B \leq C \dots$)

- **Vergleichskriterien von Sortieralgorithmen**

- Berechnungsaufwand $O(n)$
- Effizient: Best Case vs Average Case vs Worst Case
- Speicherbedarf:
 - in-place (in situ): Zusätzlicher Speicher von der Eingabegröße unabhängig
 - out-of-place: Speichermehrbedarf von Eingabegröße abhängig
- Stabilität: Stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht
- Anwendung als Auswahlfaktor:
 - Hauptoperationen beim Sortieren: Vergleiche und Vertausche
 - Diese Operationen können sehr teuer oder sehr günstig sein, je nach Aufwand
 - Anpassung des Verfahrens abhängig von dem Aufwand dieser Operationen

2.2 Analyse von Algorithmen - Teil 1

- **Schleifeninvariante (SIV)**
 - Sonderform der Invariante
 - Am Anfang/Ende jedes Schleifendurchlaufs und vor/nach jedem Schleifendurchlauf gültig
 - Wird zur Feststellung der Korrektheit von Algorithmen verwendet
 - Eigenschaften:
 - Initialisierung: Invariante ist vor jeder Iteration wahr
 - Fortsetzung: Wenn SIV vor der Schleife wahr ist, dann auch bis Beginn der nächsten Iteration
 - Terminierung: SIV liefert bei Schleifenabbruch, helfende Eigenschaft für Korrektheit
 - Beispiel für Umsetzung: **Insertion Sort - SIV**
- **Laufzeitanalyse**
 - Aufstellung der Kosten und Durchführungsanzahl für jede Zeile des Quelltextes
 - Beachte: Bei Schleifen wird auch der Aufruf gezählt, der den Abbruch einleitet
 - Beispiel für Umsetzung: **Insertion Sort - Laufzeit**
 - Zusätzliche Überprüfung des **Best Case**, **Worst Case** und **Average Case**
- **Effizienz von Algorithmen**
 - Effizienzfaktoren
 - Rechenzeit (Anzahl der Einzelschritte)
 - Kommunikationsaufwand
 - Speicherplatzbedarf
 - Zugriffe auf Speicher
 - Laufzeit hängt von versch. Faktoren ab
 - Länge der Eingabe
 - Implementierung der Basisoperationen
 - Takt der CPU

2.3 Analyse von Algorithmen - Teil 2

- **Komplexität**
 - Abstrakte Rechenzeit $T(n)$ ist abhängig von den Eingabedaten
 - Übliche Betrachtungsweise der Rechenzeit ist asymptotische Betrachtung
- **Asymptotik**
 - Annäherung an einer sich ins Unendliche verlaufende Kurve
 - z.B.: $f(x) = \frac{1}{x} + x$ | Asymptote: $g(x) = x$ | ($\frac{1}{x}$ läuft gegen Null)
- **Asymptotische Komplexität**
 - Abschätzung des zeitlichen Aufwands eines Algorithmus in Abhängigkeit einer Eingabe
 - Beispiel für Umsetzung: **Insertion Sort - Laufzeit Θ**
- **Asymptotische Notation**
 - Betrachtung der Laufzeit $T(n)$ für sehr große Eingaben $n \in \mathbb{N}$
 - Komplexität ist unabhängig von konstanten Faktoren und Summanden
 - Nicht berücksichtigt: Rechnergeschwindigkeit / Initialisierungsauswände
 - Komplexitätsmessung via Funktionsklasse ausreichend
 - Verhalten des Algorithmus für große Problemgrößen

- Veränderung der Laufzeit bei Verdopplung der Problemgröße
- **Gründe für die Nutzung der theoretischen Betrachtung statt der Messung der Laufzeit**
 - Vergleichbarkeit
 - Laufzeit abhängig von konkreter Implementierung und System
 - Theoretische Betrachtung ist frei von Abhängigkeiten und Seiteneffekten
 - Theoretische Betrachtung lässt direkte Vergleichbarkeit zu
 - Aufwand
 - Wieviele Testreihen?
 - In welcher Umgebung?
 - Messen führt in der Ausführung zu hohem, praktischen Aufwand
 - Komplexitätsfunktion
 - Wachstumsverhalten ausreichend
 - Praktische Evaluation mit Zeiten nur für Auswahl von Systemen möglich
 - Theoretischer Vergleich (Funktionsklassen) hat ähnlichen Erkenntnisgewinn

2.4 Analyse von Algorithmen - Teil 3

• Θ -Notation

- Θ -Notation beschränkt eine Funktion asymptotisch von oben und unten
- Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$ (\mathbb{N} : Eingabelänge, \mathbb{R} : Zeit)

$$\Theta(g) = \{f : \exists c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

\uparrow \uparrow
 Funktion f Positive Konstanten Für alle n größer gleich n_0 $f(n)$ wird von $c_1 g(n)$ und $c_2 g(n)$ für hinreichend große n eingeschlossen

- $\Theta(g)$ enthält alle f , die genauso schnell wachsen wie g
- Schreibweise: $f \in \Theta(g)$ (korrekt), manchmal auch $f = \Theta(g)$
- $g(n)$ ist eine asymptotisch scharfe Schranke von $f(n)$
- $f(n) = \Omega(g(n))$ gilt, wenn $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$ erfüllt sind

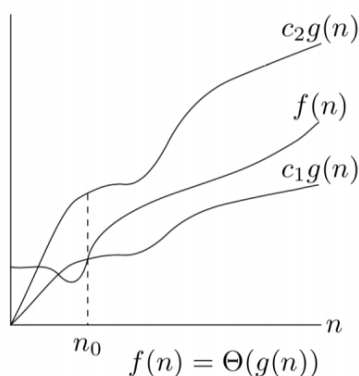


Abbildung 1: Veranschaulichung

- z.B.: $f(n) = \frac{1}{2}n^2 - 3n \mid f(n) \in \Theta(n^2)$
- Aus $\Theta(n^2)$ folgt, dass $g(n) = n^2$
- Vorgehen:
 - Finden eines n_0 und c_1, c_2 , sodass
 - $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ erfüllt ist
 - Konkret: $c_1 * n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 * n^2$
 - Division durch n^2 : $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$
 - Ab $n = 7$ positives Ergebnis: $0,0714 \mid n_0 = 7$
 - Deswegen setzen wir $c_1 = \frac{1}{14}$
 - Für $n \rightarrow \infty$: $0,5 \mid c_2 = 0,5$
 - Natürlich auch andere Konstanten möglich

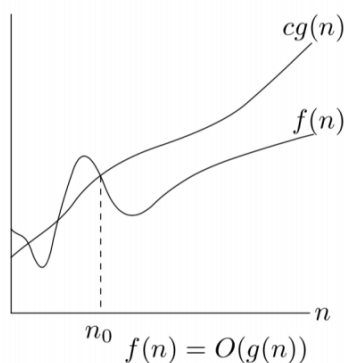
• O -Notation

- O -Notation beschränkt eine Funktion asymptotisch von oben

$$O(g) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

\uparrow \uparrow \uparrow
 Funktion f Positive Konstanten Für alle n größer gleich n_0 $f(n)$ wird von $cg(n)$ für hinreichend große n beschränkt

- $O(g)$ enthält alle f , die höchstens so schnell wie g wachsen
- Schreibweise: $f = O(g)$
- $f(n) = \Theta(g) \rightarrow f(n) = O(g) \mid \Theta(g(n)) \subseteq O(g(n))$
- Ist f in der Menge $\Theta(g)$, dann auch in der Menge $O(g)$



- z.B.: $f(n) = n + 2 \mid f(n) = O(n)$?
- Ja $f(n)$ ist Teil von $O(n)$ für z.B. $c = 2$ und $n_0 = 2$

Abbildung 2: Veranschaulichung

• O -Notation Rechenregeln

- Konstanten:
 - $f(n) = a$ mit $a \in \mathbb{R}$ konstante Funktion $\rightarrow f(n) = O(1)$
 - z.B. $3 \in O(1)$
- Skalare Multiplikation:
 - $f = O(g)$ und $a \in \mathbb{R} \rightarrow a * f = O(g)$
- Addition:
 - $f_1 = O(g_1)$ und $f_2 = O(g_2) \rightarrow f_1 + f_2 = O(\max\{g_1, g_2\})$
- Multiplikation:
 - $f_1 = O(g_1)$ und $f_2 = O(g_2) \rightarrow f_1 * f_2 = O(g_1 * g_2)$

• Ω -Notation

- Ω -Notation beschränkt eine Funktion asymptotisch von unten

$$\Omega(g) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

\uparrow \uparrow \uparrow
 Funktion f Positive Konstanten Für alle n größer gleich n_0 $f(n)$ wird von $cg(n)$ für hinreichend große n unten beschränkt

- Ω -Notation enthält alle f , die mindestens so schnell wie g wachsen

- Schreibweise: $f = \Omega(g)$

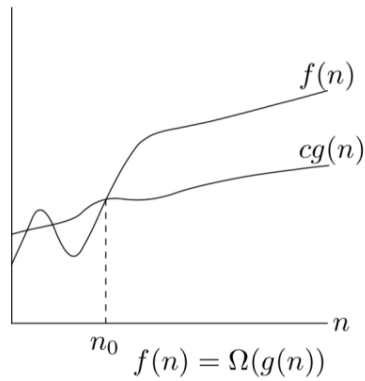


Abbildung 3: Veranschaulichung

• Komplexitätsklassen

- n ist hier die Länge der Eingabe

Klasse	Bezeichnung	Beispiel
$\Theta(1)$	Konstant	Einzeloperation
$\Theta(\log n)$	Logarithmisch	Binäre Suche
$\Theta(n)$	Linear	Sequentielle Suche
$\Theta(n \log n)$	Quasilinear	Sortieren eines Arrays
$\Theta(n^2)$	Quadratisch	Matrixaddition
$\Theta(n^3)$	Kubisch	Matrixmultiplikation
$\Theta(n^k)$	Polynomiell	
$\Theta(2^n)$	Exponentiell	Travelling-Salesman*
$\Theta(n!)$	Faktoriell	Permutationen

- Ausführungsdauer, falls eine Operation n genau $1\mu s$ dauert

Eingabe- größe n	$\log_{10} n$	n	n^2	n^3	2^n
10	$1\mu s$	$10\mu s$	$100\mu s$	1ms	$\sim 1ms$
100	$2\mu s$	$100\mu s$	10ms	1s	$\sim 4 \times 10^{16}y$
1000	$3\mu s$	1ms	1s	16min 40s	?
10000	$4\mu s$	10ms	1min 40s	$\sim 11,5d$?
10000	$5\mu s$	100ms	2h 46min 40s	$\sim 31,7y$?

- Asymptotische Notationen in Gleichungen

- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
- $\Theta(n)$ fungiert hier als Platzhalter für eine beliebige Funktion $f(n)$ aus $\Theta(n)$
- z.B.: $f(n) = 3n + 1$

• o-Notation

- o -Notation stellt eine echte obere Schranke dar
- Ausschlaggebend ist, dass es für alle $c \in \mathbb{R}_{>0}$ gelten muss
- Außerdem $<$ statt \leq
- z.B.: $2n = o(n^2)$ und $2n^2 \neq o(n^2)$

$$o(g) = \{f : \underbrace{\forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < cg(n)}\}$$

Gilt für **alle** Konstanten $c > 0$.

In O-Notation gilt es für eine Konstante $c > 0$

- **ω -Notation**

- ω -Notation stellt eine echte untere Schranke dar
- Ausschlaggebend ist, dass es für alle $c \in \mathbb{R}_{>0}$ gelten muss
- Außerdem $>$ statt \geq
- z.B.: $\frac{n^2}{2} = \omega(n)$ und $\frac{n^2}{2} \neq \omega(n^2)$

$$\omega(g) = \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

2.5 Insertion Sort

- **Idee**

- Halte die linke Teilfolge sortiert
- Füge nächsten Schlüsselwert hinzu, indem es an die korrekte Position eingefügt wird
- Wiederhole den Vorgang bis Teilfolge aus der gesamten Liste besteht

- **Code**

```
FOR j = 1 TO A.length - 1
  key = A[j]
  // Füge A[j] in die sortierte Sequenz A[0...j-1] ein
  i = j - 1
  WHILE i >= 0 and A[i] > key
    A[i + 1] = A[i]
    i = i - 1
  A[i + 1] = key
```

- **Schleifeninvariante von Insertion Sort**

- Zu Beginn jeder Iteration der **for**-Schleife besteht die Teilfolge $A[0 \dots j-1]$ aus den Elementen der ursprünglichen Teilfolge $A[0 \dots j-1]$ enthaltenen Elementen, allerdings in sortierter Reihenfolge.

- **Korrektheit von Insertion Sort**

- Initialisierung:
 - Beginn mit $j=1$, also Teilfeld $A[0 \dots j-1]$ besteht nur aus einem Element $A[0]$. Dies ist auch das ursprüngliche Element und Teilfeld ist sortiert.
- Fortsetzung:
 - Zu zeigen ist, dass die Invariante bei jeder Iteration erhalten bleibt. Ausführungsblock der **for**-Schleife sorgt dafür, dass $A[j-1]$, $A[j-2]$,... je um Stelle nach rechts geschoben werden bis $A[j]$ korrekt eingefügt wurde. Teilfeld $A[0 \dots j]$ besteht aus ursprünglichen Elementen und ist sortiert. Inkrementieren von j erhält die Invariante.
- Terminierung:
 - Abbruchbedingung der **for**-Schleife, wenn $j > A.length - 1$. Jede Iteration erhöht j . Dann bei Abbruch ist $j = n$ und einsetzen in Invariante liefert das Teilfeld $A[0 \dots n-1]$ welches aus den ursprünglichen Elementen besteht und sortiert ist. Teilfeld ist gesamtes Feld.
- Algorithmus **Insertion Sort** arbeitet damit korrekt.

• Laufzeitanalyse von Insertion Sort

```

INSERTION-SORT (A)
1 FOR j = 1 TO A.length - 1
2   key = A[j]
3   // Füge A[j] in die
   //sortierte Sequenz A[0..j-1]
4   i = j - 1
5   WHILE i ≥ 0 and A[i] > key
6     A[i+1] = A[i]
7     i = i - 1
8   A[i+1] = key

```

Laufzeit:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_8(n-1)$$

Zeile	Kosten	Anzahl
1	c_1	n
2	c_2	$n-1$
3	0	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=1}^{n-1} t_j$
6	c_6	$\sum_{j=1}^{n-1} (t_j - 1)$
7	c_7	$\sum_{j=1}^{n-1} (t_j - 1)$
8	c_8	$n-1$

- Festlegung der Laufzeit für jede Zeile
- Jede Zeile besitzt gewissen Kosten c_i
- Jede Zeile wird x mal durchgeführt
- $\text{Laufzeit} = \text{Anzahl} * \text{Kosten}$ jeder Zeile
- Schleifen: Abbruchüberprüfung zählt auch
- t_j : Anzahl der Abfragen der **While**-Schleife

- Warum n in Zeile 1?
 - Die Überprüfung der Fortführungsbedingung beinhaltet auch die letzte Überprüfung
 - Quasi die Überprüfung, durch die die Schleife abbricht
- Warum $\sum_{j=1}^{n-1}$ in Zeile 5?
 - Aufsummierung aller einzelnen t_j über die Anzahl der Schleifendurchläufe
 - Diese ist allerdings $n-1$ und nicht n , da die Abbruchüberprüfung dort auch enthalten ist
- Warum $t_j - 1$ in Zeile 6?
 - Selbes Argument wie oben, bei t_j ist die Abbruchüberprüfung enthalten
 - Deswegen wird die **while**-Schleife nur $t_j - 1$ -mal ausgeführt
- **Best Case**
 - zu sortierendes Feld ist bereits sortiert
 - t_j wird dadurch zu 1, da die **While**-Schleife immer nur einmal prüft (Abbruch)
 - Die zwei Zeilen innerhalb der **While**-Schleife werden nie ausgeführt
 - Durch Umformen ergibt sich, dass die Laufzeit eine lineare Funktion in n ist
- **Worst Case**
 - zu sortierendes Feld ist umgekehrt sortiert
 - t_j wird dadurch zu $j+1$, da die **While**-Schleife immer die gesamte Länge prüft
 - Durch Umformen ergibt sich, dass die Laufzeit eine quadratische Funktion in n ist (n^2)
- **Average Case**
 - im Mittel gut gemischt
 - t_j wird dadurch zu $j/2$
 - Die Laufzeit bleibt aber eine quadratische Funktion in n (n^2)

• Asymptotische Laufzeitbetrachtung Θ

- $T(n)$ lässt sich als quadratische Funktion $an^2 + bn + c$ betrachten
- Terme niedriger Ordnung sind für große n irrelevant
- Deswegen Vereinfachung zu n^2 und damit $\Theta(n^2)$

2.6 Bubble Sort

• Idee

- Vergleiche Paare von benachbarten Schlüsselwerten
- Tausche das Paar, falls rechter Schlüsselwert kleiner als linker

• Code

```

FOR i = 0 TO A.length - 2
  FOR j = A.length - 1 DOWNT0 i + 1
    IF A[j] < A[j-1]
      SWAP(A[j], A[j-1])

```

- **Analyse von Bubble Sort**
 - Anzahl der Vergleiche:
 - Es werden stets alle Elemente der Teilfolge miteinander verglichen
 - Unabhängig von der Vorsortierung sind **Worst** und **Best Case** identisch
 - Anzahl der Vertauschungen:
 - **Best Case**: 0 Vertauschungen
 - **Worst Case**: $\frac{n^2-n}{2}$ Vertauschungen
 - Komplexität:
 - **Best Case**: $\Theta(n)$
 - **Average Case**: $\Theta(n^2)$
 - **Worst Case**: $\Theta(n^2)$

2.7 Selection Sort

- Idee
 - Sortieren durch direktes Auswählen
 - **MinSort**: "wähle kleines Element in Array und tausche es nach vorne"
 - **MaxSort**: "wähle größtes Element in Array und tausche es nach vorne"
- Code - MinSort

```
FOR i = 0 TO A.length - 2
  k = i
  FOR j = i + 1 TO A.length - 1
    IF A[j] < A[k]
      k = j
  SWAP(A[i], A[k])
```

2.8 Divide-And-Conquer-Ansatz

- Anderer Ansatz im Gegensatz zu z.B. **InsertionSort** (inkrementelle Herangehensweise)
- Laufzeit ist im schlechtesten Fall immer noch besser als **InsertionSort**
- Prinzip: Zerlege das Problem und löse es direkt oder zerlege es weiter
- **Divide**:
 - Teile das Problem in mehrere Teilprobleme auf
 - Teilprobleme sind Instanzen des gleichen Problems
- **Conquer**:
 - Beherrsche die Teilprobleme rekursiv
 - Falls Teilprobleme klein genug, löse sie auf direktem Weg
- **Combine**:
 - Vereine die Lösungen der Teilprobleme zu Lösung des ursprünglichen Problems

2.9 Merge Sort

- Idee
 - **Divide**: Teile die Folge aus n Elementen in zwei Teilfolgen von je $\frac{n}{2}$ Elemente auf
 - **Conquer**: Sortiere die zwei Teilfolgen rekursiv mithilfe von **MergeSort**
 - **Combine**: Vereinige die zwei sortierten Teilfolgen, um die sortierte Lösung zu erzeugen
- Code

```
MERGE-SORT (A,p,r)
IF p < r
  q =  $\lfloor (p+r)/2 \rfloor$  // Teilen in 2 Teilfolgen
  MERGE-SORT(A,p,q) // Sortieren der beiden Teilfolgen
  MERGE-SORT(A,q+1,r)
  MERGE(A,p,q,r) // Vereinigung der beiden sortierten Teilfolgen
```

```

MERGE(A,p,q,r) // Geteiltes Array an Stelle q
n1 = q - p + 1
n2 = r - q
Let L[0...n1] and R[0...n2] be new arrays
FOR i = 0 TO n1 - 1 // Auffüllen der neu erstellten Arrays
    L[i] = A[p + i]
FOR j = 0 TO n2 - 1
    R[j] = A[q + j + 1]
L[n1] = ∞ // Einfügen des Sentinel-Wertes
R[n2] = ∞
i = 0
j = 0
FOR k = p TO r // Eintragweiser Vergleich der Elemente
    IF L[i] ≤ R[j]
        A[k] = L[i] // Sortiertes Zurückschreiben in Original-Array
        i = i + 1
    ELSE
        A[k] = R[j]
        j = j + 1

```

• Korrektheit von MergeSort

• Schleifeninvariante

Zu Beginn jeder Iteration der `for`-Schleife (Letztes `for` in Methode `MERGE`) enthält das Teilfeld $A[p \dots k-1]$ die $k-p$ kleinsten Elemente aus $L[0 \dots n_1]$ und $R[0 \dots n_2]$ in sortierter Reihenfolge. Weiter sind $L[i]$ und $R[i]$ die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.

• Initialisierung

Vor der ersten Iteration gilt $k=p$. Daher ist $A[p \dots k-1]$ leer und enthält 0 kleinste Elemente von L und R . Wegen $i=j=0$ sind $L[i]$ und $R[i]$ die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.

• Fortsetzung

Müssen zeigen, dass Schleifeninvariante erhalten bleibt. Dafür nehmen wir an, dass $L[i] \leq R[j]$. Dann ist $L[i]$ kleinstes Element, welches noch nicht zurück kopiert wurde. Da Array $A[p \dots k-1]$ die $k-p$ kleinsten Elemente enthält, wird der Array $A[p \dots k]$ die $k-p+1$ kleinsten Elemente enthalten, nachdem der Wert nach der Durchführung von $A[k]=L[i]$ kopiert wurde. Die Erhöhung der Variablen k und i stellt die Schleifeninvariante für die nächste Iteration wieder her. Wenn $L[i] > R[j]$ dann analoges Argument in der `ELSE`-Anweisung.

• Terminierung

Beim Abbruch gilt $k=r+1$. Durch die Schleifeninvariante enthält $A[p \dots r]$ die kleinste Elemente von $L[0 \dots n_1]$ und $R[0 \dots n_2]$ in sortierter Reihenfolge. Alle Elemente außer der Sentinels wurden komplett zurück kopiert. `MergeSort` ist außerdem ein stabiler Algorithmus.

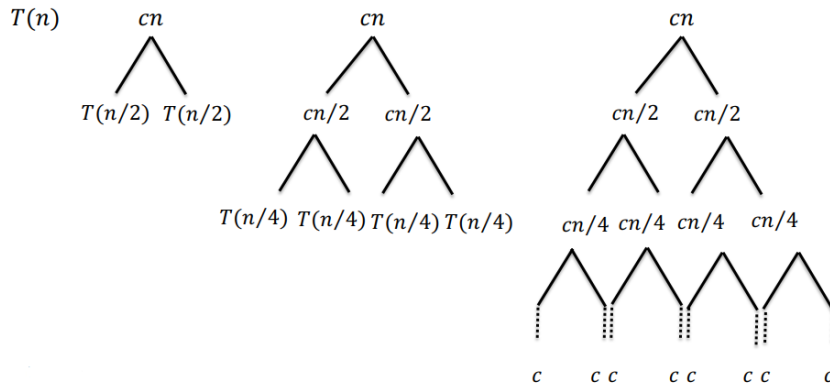
• Analyse von MergeSort

- Ziel: Bestimme Rekursionsgleichung für Laufzeit $T(n)$ von n Zahlen im schlechtesten Fall
- Divide: Berechnung der Mitte des Feldes: Konstante Zeit $\Theta(1)$
- Conquer: Rekursives Lösen von zwei Teilproblemen der Größe $\frac{n}{2}$: Laufzeit von $2 T(\frac{n}{2})$
- Combine: `MERGE` auf einem Teilfeld der Länge n : Lineare Zeit $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 2 T(\frac{n}{2}) + \Theta(n) & \text{falls } n > 1 \end{cases}$$

- Lösen der Rekursionsgleichung mithilfe eines Rekursionsbaums

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T(n/2) + cn & \text{falls } n > 1 \end{cases}$$



- Verwenden der Konstante c statt $\Theta(1)$
- cn stellt den Aufwand an der ersten Ebene dar
- Der addierte Aufwand jeder Stufe (aller Knoten) ist auch cn
- Die Anzahl der Ebenen lässt sich mithilfe von $\lg(n) + 1$ bestimmen (2-er Logarithmus)
- Damit ergibt sich für die Laufzeit: $cn \cdot \lg(n) + cn$
- Für $\lim_{n \rightarrow \infty}$ wird diese zu $n \cdot \lg(n)$
- Laufzeit beträgt damit $\Theta(n \cdot \lg(n))$
- Laufzeit von MergeSort ist in jedem Fall gleich

2.10 Quicksort

- Idee

- **Pivotelement:**

Wahl eines Pivotelement x aus dem Array

- **Divide:**

Zerlege den Array $A[p \dots r]$ in zwei Teilarrays $A[p \dots q-1]$ und $A[q+1 \dots r]$, sodass jedes Element von $A[p \dots q-1]$ kleiner oder gleich $A[q]$ ist, welches wiederum kleiner oder gleich jedem Element von $A[q+1 \dots r]$ ist. Berechnen Sie den Index q als Teil vom **Partition** Algorithmus.

- **Conquer:**

Sortieren beider Teilarrays $A[p \dots q-1]$ und $A[q+1 \dots r]$ durch rekursiven Aufruf von Quicksort.

- **Combine:**

Da die Teilarrays bereits sortiert sind, ist keine weitere Arbeit nötig um diese zu vereinigen. $A[p \dots r]$ ist nun sortiert.

- Code

```

QUICKSORT(A,p,r)
IF p < r    // Überprüfung, ob Teilarray leer ist
    q = PARTITION(A,p,r)
    QUICKSORT(A,p,q-1)
    QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
x = A[r]    // Wahl des Pivotelements
i = p - 1   // Index i setzen
FOR j = p TO r - 1 // Auffüllen des Teilarrays mit Elementen
    IF A[j] ≤ x
        i = i + 1
        SWAP(A[i], A[j]) /

```

```

SWAP(A[i+1], A[r]) // Tausch des Pivotelements
RETURN i + 1 // Neuer Index des Pivotelements

```

• Korrektheit von Quicksort

- Schleifeninvariante:

Zu Beginn jeder Iteration der **for**-Schleife gilt für den Arrayindex k folgendes:

1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$
2. Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$
3. Ist $k = r$, so gilt $A[k] = x$

- Initialisierung:

Vor der ersten Iteration gilt $i = p - 1$ und $j = p$. Da es keine Werte zwischen p und j gibt und es auch keine Werte zwischen $i + 1$ und $j - 1$ gibt, sind die ersten beiden Eigenschaften trivial erfüllt. Die Zuweisung in $x = A[r]$ sorgt für die Erfüllung der dritten Eigenschaft.

- Fortsetzung:

Zwei mögliche Fälle durch **IF** $A[j] \leq x$. Wenn $A[j] > x$, dann inkrementiert die Schleife nur den Index j . Dann gilt Bedingung 2 für $A[j-1]$ und alle anderen Einträge bleiben unverändert. Wenn $A[j] \leq x$, dann wird Index i inkrementiert und die Einträge $A[i]$ und $A[j]$ getauscht und schließlich der Index j erhöht. Wegen des Vertauschens gilt $A[i] \leq x$ und Bedingung 1 ist erfüllt. Analog gilt $A[j-1] > x$, da das Element welches mit $A[j-1]$ vertauscht wurde wegen der Invariante gerade größer als x ist.

- Terminierung:

Bei der Terminierung gilt, dass $j = r$. Daher gilt, dass jeder Eintrag des Arrays zu einer der drei durch die Invariante beschriebenen Mengen gehört.

• Performanz von Quicksort

- Abhängig von der **Balanciertheit** der Teilarrays

- Definition Balanciert: ungefähr gleiche Anzahl an Elementen
- Teilarrays balanciert: Laufzeit asymptotisch so schnell wie **MergeSort**
- Teilarrays unbalanciert: Laufzeit kann so langsam wie **InsertionSort** laufen

- Zerlegung im **schlechtesten Fall**

- Partition zerlegt Problem in ein Teilproblem mit $n - 1$ Elementen und eins mit 0 Elementen
- Unbalancierte Zerlegung zieht sich durch gesamte Rekursion
- Zerlegung kostet $\Theta(n)$
- Aufruf auf Feld der Größe 0: $T() = \Theta(1)$
- Laufzeit (rekursiv):
 - $T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n)$
 - Insgesamt folgt: $T(n) = \Theta(n^2)$

- Zerlegung im **besten Fall**

- Problem wird so balanciert wie möglich zerlegt
- Zwei Teilprobleme mit maximaler Größe von $\frac{n}{2}$
- Zerlegung kostet $\Theta(n)$
- Laufzeit (rekursiv):
 - $T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$
 - Laufzeit beträgt: $O(n \lg(n))$
- Solange die Aufteilung konstant bleibt, bleibt die Laufzeit $O(n \lg(n))$

2.11 Laufzeitanalyse von rekursiven Algorithmen

- **Analyse von Divide-And-Conquer Algorithmen**

- $T(n)$ ist Laufzeit eines Problems der Größe n
- Für kleines Problem benötigt die direkte Lösung eine konstante Zeit $\Theta(1)$
- Für sonstige n gilt:
 - Aufteilen eines Problems führt zu a Teilproblemen
 - Jedes dieser Teilprobleme hat die Größe $\frac{1}{b}$ der Größe des ursprünglichen Problems
 - Lösen eines Teilproblems der Größe $\frac{n}{b}$: $T(\frac{n}{b})$
 - Lösen a solcher Probleme: $a T(\frac{n}{b})$
 - $D(n)$: Zeit um das Problem aufzuteilen (Divide)
 - $C(n)$: Zeit um Teillösungen zur Gesamtlösung zusammenzufügen (Combine)

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n \leq c \\ a T(\frac{n}{b}) + D(n) + C(n) & \text{sonst} \end{cases}$$

- **Substitutionsmethode**

- Idee: Erraten einer Schranke und Nutzen von Induktion zum Beweis der Korrektheit
- Ablauf:
 1. Rate die Form der Lösung (Scharfes Hinsehen oder kurze Eingaben ausprobieren/einsetzen)
 2. Anwendung von vollständiger Induktion zum Finden der Konstanten und Beweis der Lösung

- **Beispiel**

- Betrachten von MergeSort:
 - $T(1) \leq c$
 - $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn$

- Ziel:

Obere Abschätzung $T(n) \leq g(n)$ mit $g(n)$ ist eine Funktion, die durch eine geschlossene Formel dargestellt werden kann.

Wir "raten": $T(n) \leq 4cn \lg(n)$ und nehmen dies für alle $n' < n$ an und zeigen es für n .

- Induktion:

- \lg steht hier für \log_2
- $n = 1$: $T(1) \leq c$
- $n = 2$: $T(2) \leq T(1) + T(1) + 2c$
 $\leq 4c \leq 8c$

$$T(2) = 4c * 2 \lg(2) = 8c$$

- Hilfsbehauptungen:

- (1): $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$
- (2): $\lfloor \frac{n}{2} \rfloor \leq \frac{n}{2} \leq \frac{2}{3}n$
- (3): $\log_c(\frac{a}{b}) = \log_c(a) - \log_c(b)$
- (4): $\log_c(a * b) = \log_c(a) + \log_c(b)$

- Induktionsschritt:

- Annahme: $n > 2$ und sei Behauptung wahr für alle $n' < n$.

$$\begin{aligned} T(n) &\leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn \\ &\leq 4c \lfloor \frac{n}{2} \rfloor \lg(\lfloor \frac{n}{2} \rfloor) + 4c \lceil \frac{n}{2} \rceil \lg(\lceil \frac{n}{2} \rceil) + cn \end{aligned}$$

$$\begin{aligned} \text{(HB)} &\leq 4c \cdot \lg(\frac{2}{3}n) \cdot (\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil) + cn \\ &\leq 4c \cdot \lg(\frac{2}{3}n) \cdot n + cn \end{aligned}$$

$$\begin{aligned} \text{(HB)} &\leq 4cn \cdot (\lg(\frac{2}{3}) + \lg(n)) + cn \\ &= 4cn \cdot \lg(n) + 4cn \cdot \lg(\frac{2}{3}) \\ &= 4cn \cdot \lg(n) + cn(1 + 4 \cdot (\lg(2) - \lg(3))) \\ &\leq 4cn \cdot \lg(n) \\ &\Rightarrow \Theta(n \lg(n)) \end{aligned}$$

- **Rekursionsbaum**

- Idee: Stellen das Ineinander-Einsetzen als Baum dar und Analyse der Kosten
- Ablauf:
 1. Jeder Knoten stellt die Kosten eines Teilproblems dar
 - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar
 - Die Blätter stellen die Kosten der Basisfälle dar (z.B. $T(0)$)
 2. Berechnen der Kosten innerhalb jeder Ebene des Baums
 3. Die Gesamtkosten sind die Summe über die Kosten aller Ebenen
- Rekursionsbaum ist nützlich um Lösung für Substitutionsmethode zu erraten
- **Beispiel:** $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$
 - $\Rightarrow T(n) = 3T(\frac{n}{4}) + cn^2$ ($c > 0$)
 - Je Abstieg verringert sich die Größe des Problems um den Faktor 4.
 - Erreichen der Randbedingung ist vonnöten, die Frage ist wann dies geschieht.
 - Größe Teilproblem bei Level i : $\frac{n}{4^i}$
 - Erreichen Teilproblem der Größe 1, wenn $\frac{n}{4^i} = 1$, d.h. wenn $i = \log_4(n)$
 \Rightarrow Baum hat also $\log_4 n + 1$ Ebenen
 - Kosten pro Ebene:
 - Jede Ebene hat 3-mal so viele Knoten wie darüber liegende
 - Anzahl der Knoten in Tiefe i ist 3^i
 - Kosten $c(\frac{n}{4^i})^2$, $i = 0 \dots \log_4 n - 1$
 - Anzahl \cdot Kosten $= 3^i \cdot c(\frac{n}{4^i})^2 = (\frac{3}{16})^i \cdot cn^2$
 - Unterste Ebene:
 - $3^{\log_4(n)} = n \log_4(3)$ Knoten
 - Jeder Knoten trägt $T(1)$ Kosten bei
 - Kosten unten: $n^{\log_4(3)} \cdot T(1) = \Theta(n^{\log_4(3)})$
 - Addiere alle Kosten aller Ebenen:
 - $T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + \dots + (\frac{3}{16})^{\log_4 n - 1}cn^2 + \Theta(n^{\log_4(3)})$
 $= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4(3)})$
 $= \frac{(\frac{3}{16})^{\log_4 n} - 1}{\frac{3}{16} - 1} \cdot cn^2 + \Theta(n^{\log_4(3)})$
 (Verwendung der geometrischen Reihe)
 - Verwendung einer unendlichen fallenden geometrischen Reihe als obere Schranke:
 $T(n) = \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{\log_4(3)})$
 $< \sum_{i=0}^{\infty} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{\log_4(3)})$
 $= \frac{1}{1 - \frac{3}{16}} \cdot cn^2 + \Theta(n^{\log_4(3)})$
 $= \frac{16}{13} \cdot cn^2 + \Theta(n^{\log_4(3)}) = O(n^2)$
 - Jetzt **Substitutionsmethode**:
 - Zu zeigen: $\exists d > 0 : T(n) \leq dn^2$
 - Induktionsanfang:
 $T(n) = 3 \cdot T(\lfloor \frac{n}{4} \rfloor) + c \cdot 1^2$
 $= 3 \cdot T(0) + c = c$
 - Induktionsschritt:
 $T(n) \leq 3 \cdot T(\lfloor \frac{n}{4} \rfloor) + cn^2$
 $\leq 3 \cdot d(\lfloor \frac{n}{4} \rfloor)^2 + cn^2$
 $\leq 3d(\frac{n}{4})^2 + cn^2$
 $= \frac{3}{16}dn^2 + cn^2$
 $\leq dn^2$, falls $d \geq \frac{16}{13}c$

- **Mastertheorem**

- Idee:

Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f(n)$ eine positive Funktion und $T(n)$ über den nicht-negativen ganzen Zahlen über die Rekursionsgleichung $T(n) = a T(\frac{n}{b}) + f(n)$ definiert, wobei wir $\frac{n}{b}$ so interpretieren, dass damit entweder $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ gemeint ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken (a und b werden aus $f(n)$ gelesen):

1. Gilt $f(n) = O(n^{\log_b(a-\epsilon)})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b(a)})$
2. Gilt $f(n) = O(n^{\log_b(a)})$, dann gilt $T(n) = \Theta(n^{\log_b(a)} \lg(n))$
3. Gilt $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ für eine Konstante $\epsilon > 0$ und $a f(\frac{n}{b}) \leq c f(n)$ für eine Konstante $c < 1$ und hinreichend großen n , dann ist $T(n) = \Theta(f(n))$

- Erklärung:

- In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^{\log_b(a)}$ verglichen
 1. Wenn $f(n)$ polynomial kleiner ist als $n^{\log_b(a)}$, dann $T(n) = \Theta(n^{\log_b(a)})$
 2. Wenn $f(n)$ und $n^{\log_b(a)}$ die gleiche Größe haben, gilt $T(n) = \Theta(n^{\log_b(a)} \lg(n))$
 3. Wenn $f(n)$ polynomial größer als $n^{\log_b(a)}$ und $a f(\frac{n}{b}) \leq c f(n)$ erfüllt, dann $T(n) = \Theta(f(n))$
- (polynomial größer/kleiner: um Faktor n^ϵ asymptotisch größer/kleiner)

- Nicht abgedeckte Fälle:

- Wenn einer dieser Fälle eintritt, kann das Mastertheorem nicht angewendet werden
 1. Wenn $f(n)$ kleiner ist als $n^{\log_b(a)}$, aber nicht polynomial kleiner
 2. Wenn $f(n)$ größer ist als $n^{\log_b(a)}$, aber nicht polynomial größer
 3. Regularitätsbedingung $a f(\frac{n}{b}) \leq c f(n)$ wird nicht erfüllt
 4. a oder b sind nicht konstant (z.B. $a = 2^n$)

- Beispiel:

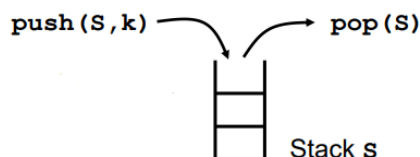
- $T(n) = 9T(\frac{n}{3}) + n$
 - $a = 9, b = 3, f(n) = n$
 - $\log_b(a) = \log_3(9) = 2$
 - $f(n) = n = O(n^{\log_b(a-\epsilon)})$
 $= O(n^{2-\epsilon})$
 - Ist diese Gleichung für ein $\epsilon > 0$ erfüllt? $\Rightarrow \epsilon = 1$
 - **1. Fall** $\Rightarrow T(n) = \Theta(n^2)$
- $T(n) = T(\frac{2n}{3}) + 1$
 - $a = 1, b = \frac{3}{2}, f(n) = 1$
 - $\log_{\frac{3}{2}} 1 = 0$
 - $f(n) = 1 = O(n^{\log_b(a)})$
 $= O(n^0)$
 $= O(1)$
 - **2. Fall** $\Rightarrow T(n) = \Theta(1 * \lg(n)) = \Theta(\lg(n))$
- $T(n) = 3(T(\frac{n}{4}) + n \lg(n))$
 - $a = 3, b = 4, f(n) = n \lg(n)$
 - $n^{\log_b(a)} = n^{\log_4(3)} \leq n^{0.793}$
 - $\epsilon = 0.1$ im Folgenden
 - $f(n) = n \lg(n) \geq n \geq n^{0.793+0.1} \geq n^{0.793}$
 - **3. Fall** $\Rightarrow f(n) = \Omega(n^{\log_b(a+0.1)})$
 - $a f(\frac{n}{b}) = 3 f(\frac{n}{4}) = 3(\frac{n}{4}) \lg(\frac{n}{4}) \leq \frac{3}{4} n \lg(n)$
 - Damit ist auch die Randbedingung erfüllt und $T(n) = \Theta(n \lg(n))$

3 Grundlegende Datenstrukturen

3.1 Stacks

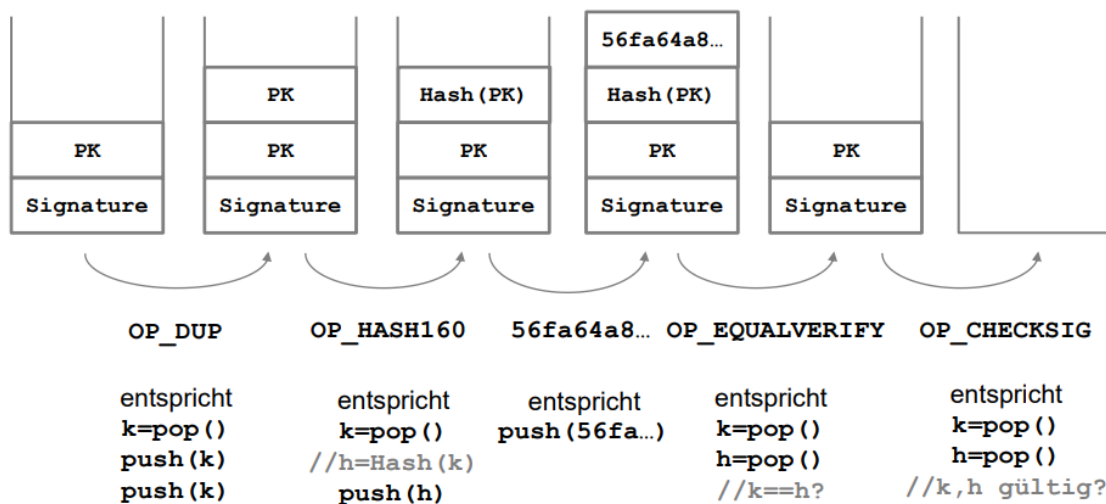
- Abstrakter Datentyp Stack

- `new S()`
 - Erzeugt neuen (leeren) Stack
- `s.isEmpty()`
 - Gibt an, ob Stack `s` leer ist
- `s.pop()`
 - Gibt oberstes Element vom Stack `s` zurück und löscht es vom Stack
 - Gibt Fehlermeldung aus, falls der Stack leer ist
- `s.push(k)`
 - Schreibt `k` als neues oberstes Element auf Stack `s`
- Abstrakter Aufbau:
 - LIFO-Prinzip - Last in, First out



- Beispiel Bitcoin

```
scriptPubKey:
OP_DUP OP_HASH160 56fa64a8bd7852d2c58095fa9a2fcd52d2c580b65d35549d
OP_EQUALVERIFY OP_CHECKSIG
```



- Stacks als Array

	0	1	2	3	4	5	6	7	8
s	12	47	17	98	72				

- `s.top` zeigt immer auf oberstes Element
- `pop()` führt dazu, dass `s.Top` sich eins nach links bewegt
- `push(k)` führt dazu, dass `s.Top` sich eins nach rechts bewegt
- Stacks als Array - Methoden, falls maximale Größe bekannt

```

new(S)

1 S.A[]=ALLOCATE(MAX);
2 S.top=-1;

```

```

isEmpty(S)

1 IF S.top<0 THEN
2   return true
3 ELSE
4   return false;

```

```

pop(S)

1 IF isEmpty(S) THEN
2   error 'underflow'
3 ELSE
4   S.top=S.top-1;
5   return S.A[S.top+1];

```

```

push(S,k)

1 IF S.top==MAX-1 THEN
2   error 'overflow'
3 ELSE
4   S.top=S.top+1;
5   S.A[S.top]=k;

```

- Stacks mit variabler Größe - Einfach

- Falls push(k) bei vollem Array \Rightarrow Vergrößerung des Arrays
- Erzeugen eines neuen Arrays mit Länge + 1 und Umkopieren aller Elemente
- Durchschnittlich $\Omega(n)$ Kopierschritte pro push-Befehl

- Stacks mit variabler Größe - Verbesserung

- Idee:
 - Wenn Grenze erreicht, Verdopplung des Speichers und Kopieren der Elemente
 - Falls weniger als ein Viertel belegt, schrumpfe das Array wieder

- Methoden:

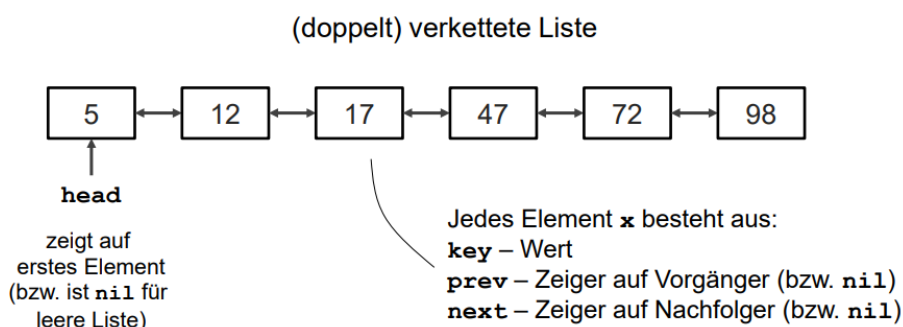
RESIZE(A,m) reserviert neuen Speicher der Größe m und kopiert A um

<pre> new(S) 1 S.A[]=ALLOCATE(1); 2 S.top=-1; 3 S.memsize=1; </pre>	<pre> isEmpty(S) 1 IF S.top<0 THEN 2 return true 3 ELSE 4 return false; </pre>
<pre> pop(S) 1 IF isEmpty(S) THEN 2 error 'underflow' 3 ELSE 4 S.top=S.top-1; 5 IF 4*(S.top+1)==S.memsize THEN 6 S.memsize=S.memsize/2; 7 RESIZE(S.A,S.memsize); 8 return S.A[S.top+1]; </pre>	<pre> push(S,k) 1 S.top=S.top+1; 2 S.A[S.top]=k; 3 IF S.top+1>=S.memsize THEN 4 S.memsize=2*S.memsize; 5 RESIZE(S.A,S.memsize); </pre>

- Im Durchschnitt für jeder der mindestens n Befehle $\Theta(1)$ Umkopierschritte

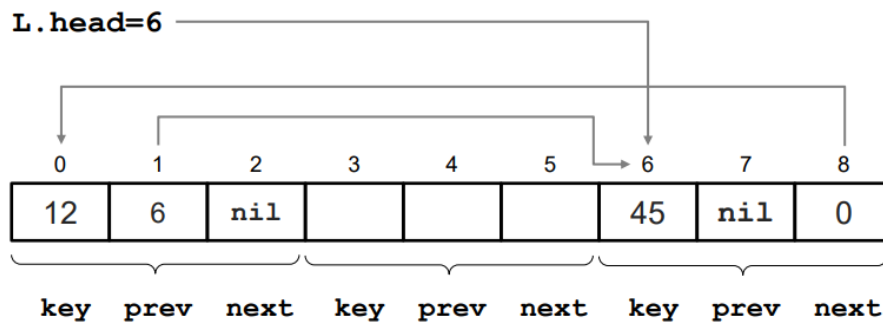
3.2 Verkettete Listen

- Aufbau



- Verkettete Listen durch Arrays

Entspricht doppelter Verkettung zwischen 45 und 12



- Elementare Operationen auf Listen

- Suche nach Element

- Laufzeit beträgt im Worst Case $\Theta(n)$
 \Rightarrow Keine Überprüfung, ob Wert bereits in Liste, sonst $\Theta(n)$

- Code:

```
search(L,k)    // Returns pointer to k in L (or nil)
current = L.head;
WHILE current != nil AND current.key != k DO
    current = current.next;
return current;
```

- Einfügen eines Elements am Kopf der Liste

- Laufzeit beträgt $\Theta(1)$, da Einfügen am Kopf

- Code:

```
insert(L,x)
x.next = l.head;
x.prev = nil;
IF L.head != nil THEN
    L.head.prev = x;
L.head = x;
```

- Löschen eines Elements aus Liste

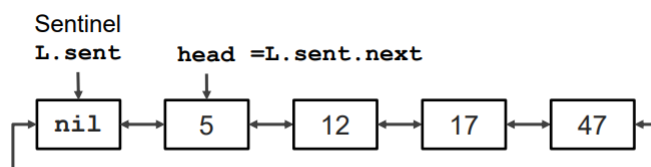
- Laufzeit beträgt $\Theta(1)$, da hier Pointer auf Objekt gegeben
 Löschen eines Wertes k mithilfe von Suche beträgt $\Omega(n)$

- Code:

```
delete (L,x)
IF x.prev != nil THEN
    x.prev.next = x.next
ELSE
    L.head = x.next;
IF x.next != nil THEN
    x.next.prev = x.prev;
```

- Vereinfachung per Wächter/Sentinels

- Ziel ist die Eliminierung der Spezialfälle für Listenanfang/-ende



Sentinel ist „von außen“ nicht sichtbar

Leere Liste besteht nur aus Sentinel

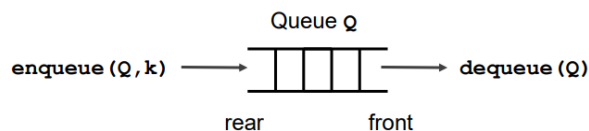
- Löschen mit Sentinels:

```
deleteSent(L,x)
x.prev.next = x.next;
x.next.prev = x.prev;
```

3.3 Queues

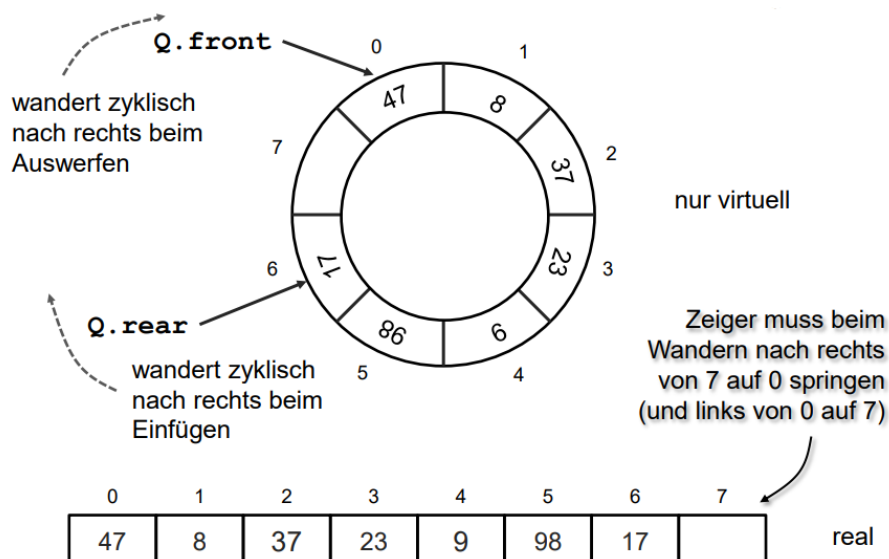
- **Abstrakter Datentyp Queue**

- `new Q()`
 - Erzeuge neue (leere) Queue
- `q.isEmpty()`
 - Gibt an, ob Queue q leer ist
- `q.dequeue()`
 - Gibt vorderstes Element aus q zurück und löscht es auf Queue
 - Fehlermeldung, falls Queue leer ist
- `q.enqueue(k)`
 - Schreibt k als neues hinterstes Element auf q
 - Fehlermeldung, falls Queue voll ist
- Abstrakter Aufbau:
 - **FIFO-Prinzip** / First in, First out



- **Queues als (virtuelles) zyklisches Array**

Bekannt: Maximale Elemente gleichzeitig in Queue



- Problem, falls `Q.rear` und `Q.front` auf selbes Element zeigen
 - Speichere Information, ob Schlange leer oder voll, in boolean `empty`
 - Alternativ: Reserviere ein Element des Arrays als Abstandshalter
- Methoden für zyklisches Array

Q leer, wenn `front==rear`
und `empty==true`

```
new(Q)
1 Q.A[]=ALLOCATE(MAX);
2 Q.front=0;
3 Q.rear=0;
4 Q.empty=true;
```

Q voll, wenn `front==rear`
und `empty==false`

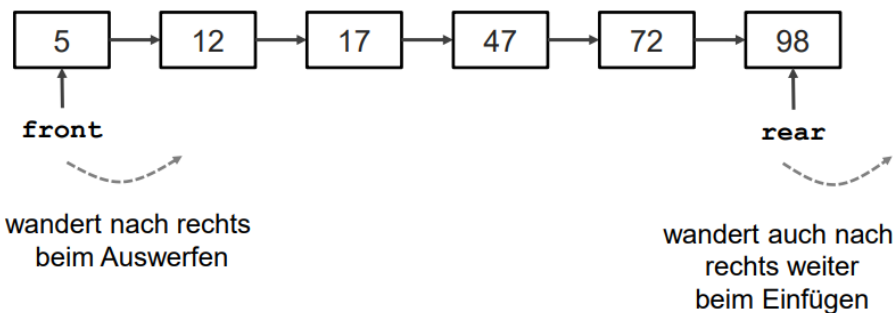
```
isEmpty(Q)
1 return Q.empty;
```

```
dequeue(Q)
1 IF isEmpty(Q) THEN
2   error 'underflow'
3 ELSE
4   Q.front=Q.front+1 mod MAX;
5   IF Q.front==Q.rear THEN
6     Q.empty=true;
7   return Q.A[Q.front-1 mod MAX];
```

```
enqueue(Q,k)
1 IF Q.rear==Q.front AND !Q.empty
2 THEN error 'overflow'
3 ELSE
4   Q.A[Q.rear]=k;
5   Q.rear=Q.rear+1 mod MAX;
6   Q.empty=false;
```

- Queues durch einfach verkettete Listen

(einfach) verkettete Liste



Methoden:

```
new(Q)
1 Q.front=nil;
2 Q.rear=nil;
```

```
isEmpty(Q)
1 IF Q.front==nil THEN
2   return true
3 ELSE
4   return false;
```

```
dequeue(Q)
1 IF isEmpty(Q) THEN
2   error 'underflow'
3 ELSE
4   x=Q.front;
5   Q.front=Q.front.next;
6   return x;
```

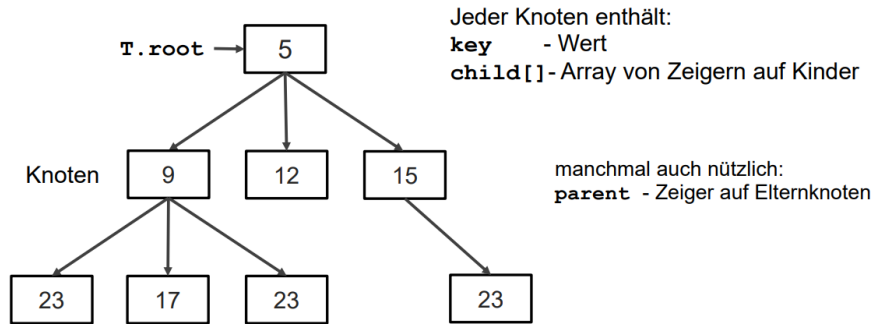
```
enqueue(Q,x)
1 IF isEmpty(Q) THEN
2   Q.front=x;
3 ELSE
4   Q.rear.next=x;
5   x.next=nil;
6   Q.rear=x;
```

- Laufzeit

- Enqueue: $\Theta(1)$
- Dequeue: $\Theta(1)$

3.4 Binäre Bäume

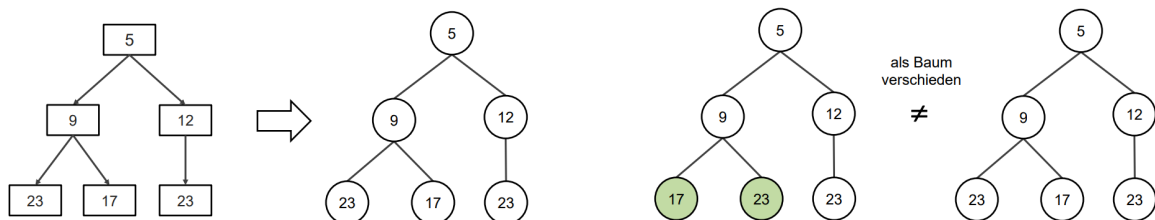
- Bäume durch verkettete Listen



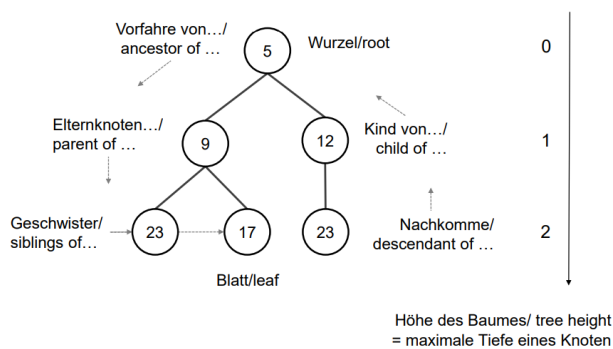
Baum-Bedingung: Baum ist leer oder...
es gibt einen Knoten r („Wurzel“), so dass jeder Knoten v von der Wurzel aus
per eindeutiger Sequenz von **child**-Zeigern erreichbar ist:
 $v = r.child[i_1].child[i_2] \dots child[i_m]$

Bäume sind "azyklisch" (Keine rückführende Spur")

- Darstellung als (ungerichteter) Graph

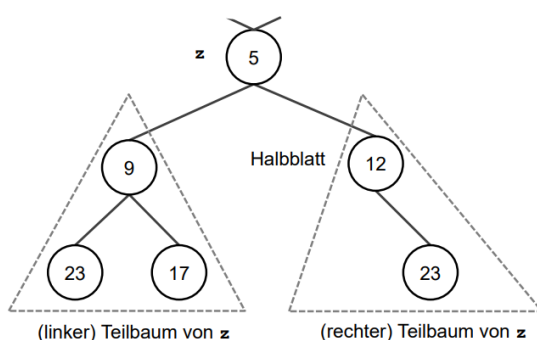


- Allgemeine Begrifflichkeiten



- Blatt: Knoten ohne Nachfolger
- Nachkomme von x :
Erreichbar durch Pfad ausgehend von x

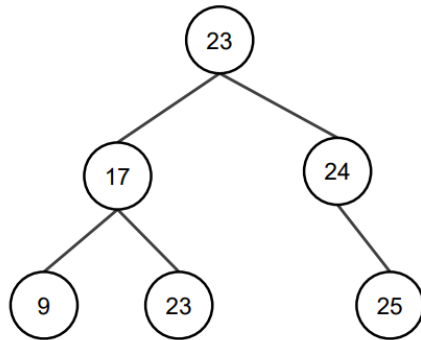
- Begrifflichkeiten Binärbaum



- Jeder Knoten hat maximal zwei Kinder
 $left=child[0]$ und $right=child[1]$
- Ausgangsgrad jedes Knoten ist ≤ 2
- Höhe leerer Baum per Konvention -1
- Höhe (nicht-leerer) Baum:
 $\max\{\text{Höhe aller Teilbäume der Wurzel}\} + 1$
- Halbblatt: Knoten mit nur einem Kind

• Traversieren von Bäumen

- Darstellung eines Baumes mithilfe einer Liste der Werte aller Knoten
- Laufzeit bei n Knoten: $T(n) = O(n)$
- Nutzung der Preorder für das Kopieren von Bäumen
 1. Preorder betrachtet Knoten und legt Kopie an
 2. Preorder geht dann in Teilbäume und kopiert diese
- Nutzung der Postorder für das Löschen von Bäumen
 1. Postorder geht zuerst in Teilbäume und löscht diese
 2. Betrachten des Knoten erst danach und dann Löschung dieses



inorder(T.root) ergibt

9 17 23 23 24 25

preorder(T.root) ergibt

23 17 9 23 24 25

postorder(T.root) ergibt

9 23 17 25 24 23

Code:

```

inorder(x)
IF x != nil THEN
    inorder(x.left);
    print x.key;
    inorder(x.right);
  
```

```

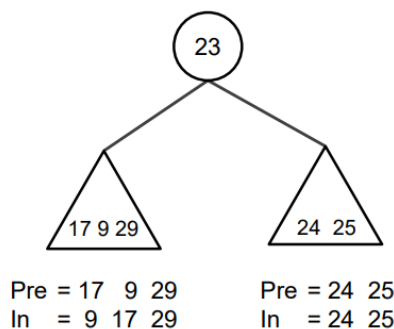
preorder(x)
IF x != nil THEN
    print x.key;
    preorder(x.left);
    preorder(x.right);
  
```

```

postorder(x)
IF x != nil THEN
    postorder(x.left);
    postorder(x.right);
    print x.key;
  
```

• Eindeutige Bestimmbarkeit von Bäumen

- Nur In-,Pre-,Postorder reichen nicht zur eindeutigen Bestimmbarkeit von Bäumen
 ⇒ Preorder/Postorder + Inorder + eindeutige Werte sind notwendig



Bilde Teilbäume rekursiv

Preorder = 23 17 9 29 24 25

(1) Identifiziert Wurzel

Inorder = 9 17 29 23 24 25

(2) Identifiziert Werte im linken und rechten Teilbaum

- **Abstrakter Datentyp Baum**

- Abstrakter Aufbau:

- `new T()`
 - Erzeugt neuen Baum namens `t`
- `t.search(k)`
 - Gibt Element `x` in Baum `t` mit `x.key == k` zurück
- `t.insert(k)`
 - Fügt Element `x` in Baum `t` hinzu
- `t.delete(x)`
 - Löscht `x` aus Baum `t`

- Suche nach Elementen

- Laufzeit = $\Theta(n)$ (Jeder Knoten maximal einmal, jeder Knoten im schlechtesten Fall)
- Starte mit `search(T.root, k)`
- Code:

```
search(x, k)
IF x == nil THEN return nil;
IF x.key == k THEN return x;
y = search(x.left, k);
IF y != nil THEN return y;
return search(x.right, k);
```

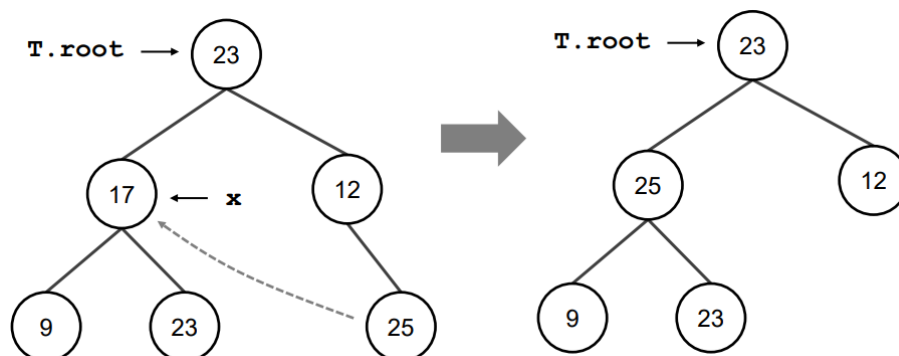
- Einfügen von Elementen

- Laufzeit = $\Theta(1)$
- Hier wird als Wurzel eingefügt (Achtung: Erzeugt linkslastigen Baum)
- Code:

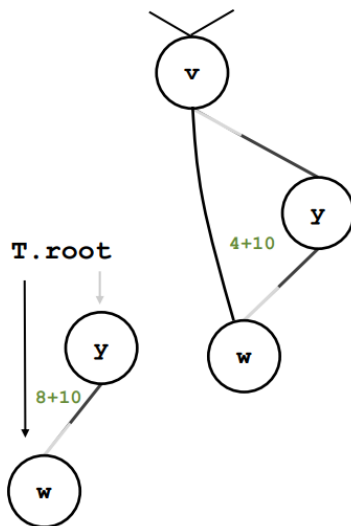
```
insert(T, x) // x.parent == x.left == x.right == nil;
IF T.root != nil THEN
    T.root.parent = x;
    x.left = T.root;
T.root = x;
```

- Löschen von Elementen

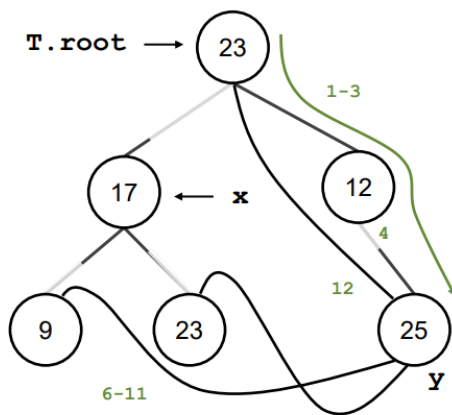
- Laufzeit = $\Theta(h)$ (Höhe des Baumes, $h = n$ möglich)
- Hier: Ersetze `x` durch Halbblatt ganz rechts



- Connect-Algorithmus:



- Delete-Algorithmus:



- Laufzeit = $\Theta(1)$

```

connect(T,y,w) // Connects w to y.parent
v = y.parent;
IF y != T.root THEN
    IF y == v.right THEN
        v.right = w;
    ELSE
        v.left = w;

ELSE
    T.root = w;

IF w != nil THEN
    w.parent = v;

```

```

delete(T,x) // assumes x in T
y = T.root;
WHILE y.right != nil DO
    y = y.right;

connect(T,y,y.left);

if x != y THEN
    y.left = x.left;
    IF x.left != nil THEN
        x.left.parent = y;
    y.right = x.right;
    IF x.right != nil THEN
        x.right.parent = y;
    connect(T,x,y);

```

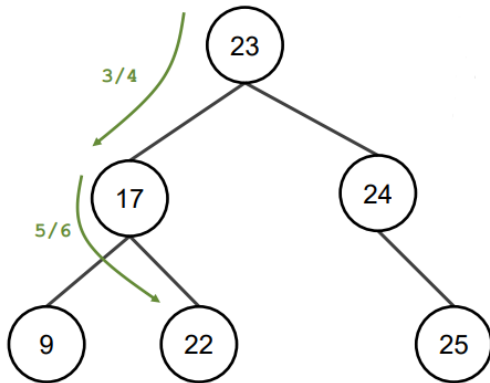
3.5 Binäre Suchbäume

- Definition

- Totale Ordnung auf den Werten
- Für alle Knoten z gilt:
 Wenn x Knoten im linken Teilbaum von z , dann $x.\text{key} \leq z.\text{key}$
 Wenn y Knoten im rechten Teilbaum von z , dann $y.\text{key} \geq z.\text{key}$
- Preorder/Postorder + eindeutige Werte \Rightarrow Eindeutige Identifizierung

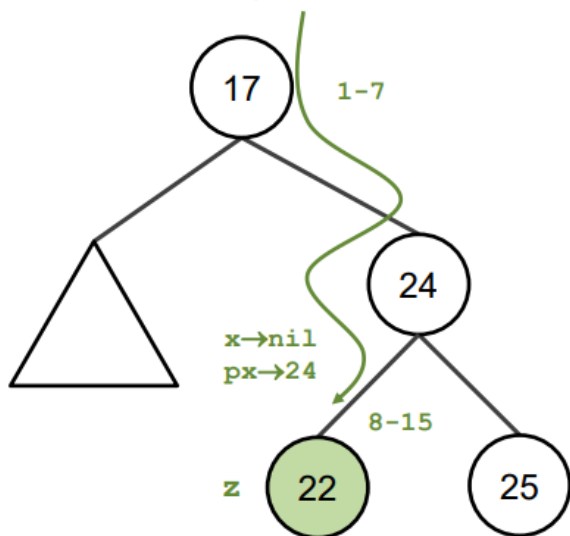
- Suchen im Binären Suchbaum

`search(T.root, 22)`



- Einfügen im Binary Search Tree

`insert(T, z)`



- Laufzeit = $O(h)$ (Höhe)

- Code:

```
search(x,k) // 1. Aufruf x = root
IF x == nil OR x.key == k THEN
    return x;
IF x.key > k THEN
    return search(x.left,k);
ELSE
    return search(x.right,k);
```

- Iterativer Code:

```
iterative-search(x,k)
WHILE x != nil AND x.key != k DO
    IF x.key > k THEN
        x = x.left;
    ELSE
        x = x.right;
return x;
```

- Laufzeit = $O(h)$

- Aufwenderiger, da Ordnung erhalten werden muss

- Code:

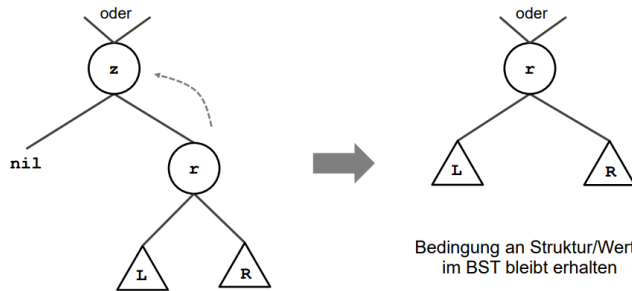
```
insert (T,z) // z.left == z.right == nil;
x = T.root;
px = nil;
WHILE x != nil DO
    px = x;
    IF x.key > z.key THEN
        x = x.left;
    ELSE
        x = x.right;
z.parent = px;
IF px == nil THEN
    T.root = z;
ELSE
    IF px.key > z.key THEN
        px.left = z;
    ELSE
        px.right = z;
```

• Löschen im BST

- Verschiedene Fälle:

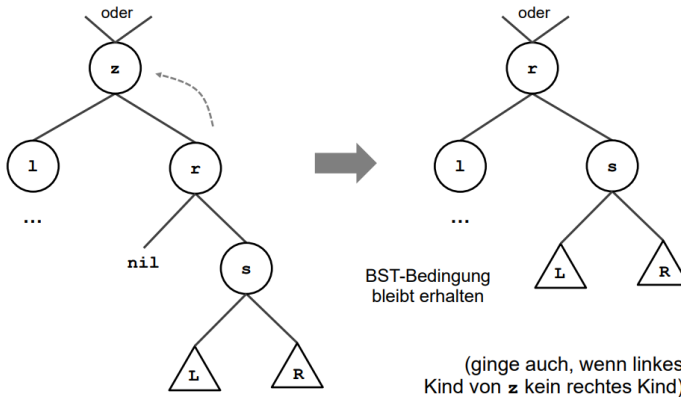
Löschen im BST (I)

zu löschender Knoten z hat maximal ein Kind



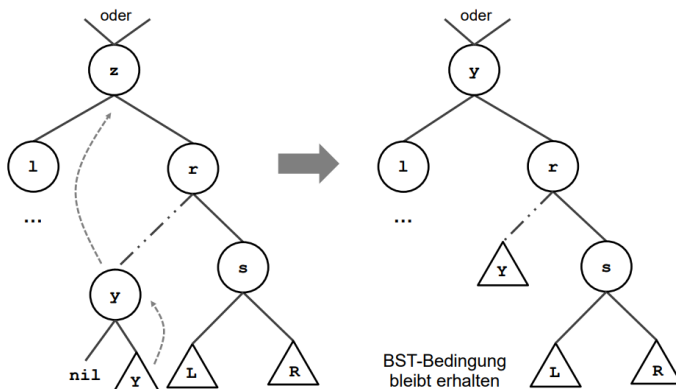
Löschen im BST (II)

rechtes Kind von Knoten z hat kein linkes Kind



Löschen im BST (III)

„kleinster“ Nachfahre vom rechten Kind von z



- Code (Transplantation)

// Hängt Teilbaum v an Parent von u

transplant(T,u,v)

IF u.parent == nil THEN

 T.root = v;

ELSE

 IF u == u.parent.left THEN

 u.parent.left = v;

 ELSE

 u.parent.right = v;

IF v != nil THEN

 v.parent = u.parent;

delete(T,z)

IF z.left == nil THEN

 transplant(T,z,z.left)

ELSE

 IF z.right == nil THEN

 transplant(T,z,z.left)

 ELSE

 y = z.right;

 WHILE y.left != nil DO y = y.left;

 IF y.parent != z THEN

 transplant(T,y,y.right)

 y.right = z.right;

 y.right.parent = y;

 transplant(T,z,y)

 y.left = z.left;

 y.left.parent = y;

- Laufzeit = $O(h)$
- Laufzeit ist damit besser, wenn viele Suchoperationen und h klein relativ zu n

- **Höhe eines BST**

- Best Case:
 - Vollständiger Baum (Alle Blätter gleiche Tiefe)
 - $h = O(\log_2 n)$
 - Laufzeit = $O(\log_2 n)$
- Worst Case:
 - Degenerierter Baum (lineare Liste)
 - $h = n - 1$
 - Laufzeit = $\Theta(n)$
- Durchschnittliche Höhe:
 - Erwartete Höhe: $\Theta(\log_2 n)$

- **Suchbäume als Suchindex**

- Knoten speichert nur Primärschlüssel und Zeiger auf Daten
- Zusätzliche Indizes möglich, kosten aber Speicherplatzbedarf

Suchbäume als Suchindex

