Algorithmen und Datenstrukturen

J. Milkovits

Last Edited: 29. Juli 2020

Inhaltsverzeichnis

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
  - Endlichkeit
    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 $D_1, D_2, ..., D_n$
  - 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 $< a_1, a_2, ..., a_n >$
    - Engabe ist eine **Instanz** des Sortierproblems
    - Ausgabe: Permutation $< a'_1, a'_2, ..., a'_n >$ derselben Folge mit Eigenschaft $a'_1 \leq ... \leq a'_n$
  - Algorithmus **korrekt**, wenn dieser das Problem für alle Instanzen löst
- **Exkurs: Totale Ordnung**
  - Sei M eine nicht leere Menge und $\leq \subseteq MxM$ 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 $(1 \leq 2 \leq 3...)$
  - z.B.: Lexikographische Ordnung $\leq_{lex}$ ist eine totale Ordnung $(A \leq B \leq C...)$
- **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

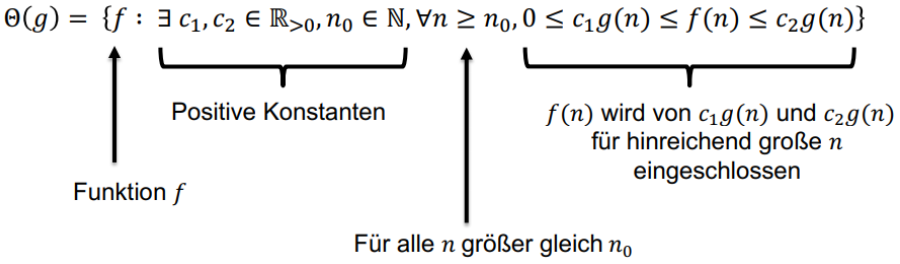
- **Laufzeitanalyse**
  - Aufstellung der Kosten und Durchführungsanzahl für jede Zeile des Quelltextes
  - Beachte: Bei Schleifen wird auch der Aufruf gezählt, der den Abbruch einleitet
  - Beispiel für Umsetzung: Insertion Sort - Laufzeit
  - Zusätzliche Überprüfung des `Best Case`, `Worst Case` und `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 Betrachung 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 mögliche
    - 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 $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ ($\mathbb{N}$: Eingabelänge, $\mathbb{R}$: Zeit)



  - $\Theta(g)$ 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

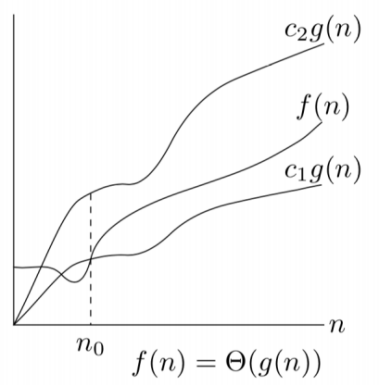
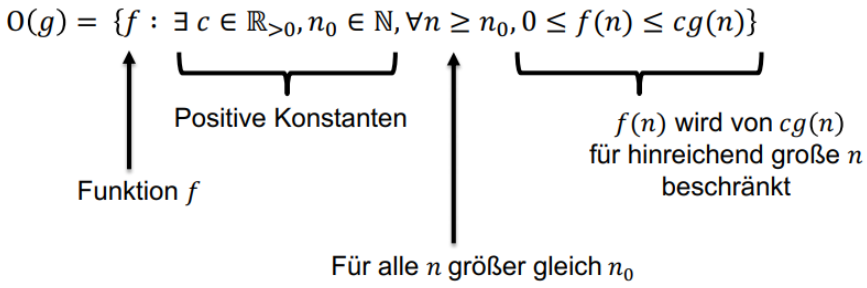


Abbildung 1: Veranschaulichung

  - 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: $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c2$
    - Ab $n = 7$ positives Ergebnis: $0,0714 \mid n_0 = 7$
    - Deswegen setzen wir $c_1 = \frac{1}{14}$
    - Für $n \to \infty :\ 0,5 \mid c_2 = 0,5$
    - Natürlich auch andere Konstanten möglich

- **O-Notation**
  - O-Notation beschränkt eine Funktion asymptotisch von oben



  - $O(g)$ 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 $O(g)$

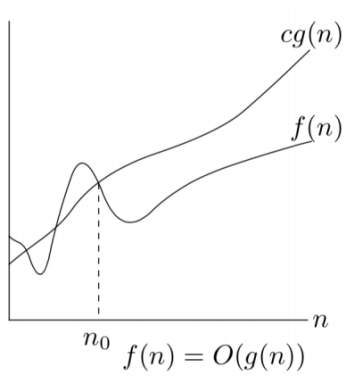


Abbildung 2: Veranschaulichung

  - 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 $n_0 = 2$

- **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 $f_1 = O(g_2) \rightarrow f_1 * f_2 = O(g_1 * g_2)$

- **Ω-Notation**
  - Ω-Notation beschränkt eine Funktion asymptotisch von unten

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

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: $f = \Omega(g)$

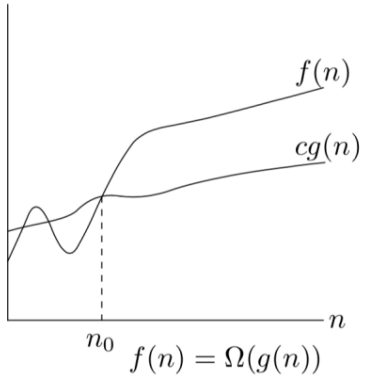


Abbildung 3: Veranschaulichung

- **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	1µs	10µs	100µs	1ms	~1ms
100	2µs	100µs	10ms	1s	~4x10^{16}y
1000	3µs	1ms	1s	16min 40s	?
10000	4µs	10ms	1min 40s	~11,5d	?
10000	5µs	100ms	2h 46min 40s	~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...j-1]` aus den Elementen der ursprünglichen Teilfolge `A[0...j-1]` enthaltenen Elementen, allerdings in sortierter Reihenfolge.
- **Korrektheit von `Insertion Sort`**
  - *Initialisierung:*
    - Beginn mit `j=1`, also Teilfeld `A[0...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...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...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
- $Laufzeit = Anzahl * Kosten$ 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 letze 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 DOWNTO 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 = ⌊(p+r)/2⌋ // 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
n_1 = q - p + 1
n_2 = r - q
Let L[0...n_1] and R[0...n_2] be new arrays
FOR i = 0 TO n_1 - 1 // Auffüllen der neu erstellten Arrays
    L[i] = A[p + i]
FOR j = 0 TO n_2 - 1
    R[j] = A[q + j + 1]
L[n_1] = ∞ // Einfügen des Sentinel-Wertes
R[n_2] = ∞
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...k-1]` die `k-p` kleinsten Elemente aus `L[0...`n_1`]` und `R[0...`n_2`]` in sortierter Reihenfolge. Weiter sind `L[i]` und `R[i]` die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.
  - *Initialisierung*

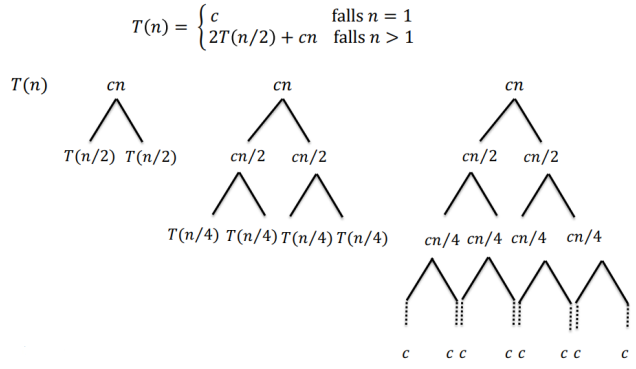
    Vor der ersten Iteration gilt `k=p`. Daher ist `A[p...k-1]` leer und enthält 0 kleinste Elemente von `L` und `R`. Wegen `i=j=0` sind `L[i]` und `R[i]` die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.
  - *Fortsetzung*

    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...k-1]` die `k-p` kleinsten Elemente enthält, wird der Array `A[p...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...r]` die kleinste Elemente von `L[0...`n_1`]` und `R[0...`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



    - 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 Azahl 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 $\lim_{n\to\infty}$ wird diese zu $n \cdot lg(n)$
    - Laufzeit beträgt damit $\Theta(n \cdot lg(n))$
    - Laufzeit von `MergaSort` ist in jedem Fall gleich

2.10 Quicksort

- **Idee**
  - *Pivotelement:*

    Wahl eines Pivotelement `x` aus dem Array
  - *Divide:*

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

    Sortieren beider Teilarrays `A[p...q-1]` und `A[q+1...r]` durch rekursiven Aufruf von Quicksort.
  - *Combine:*

    Da die Teilarrays bereits sortiert sind, ist keine weitere Arbeit nötig um diese zu vereinigen. `A[p...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)

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

- **Subsitutionsmethode**
  - 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 $n' < n$.

      $$\begin{aligned}
      \text{T(n)} &\leq T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + cn \\
      &\leq 4c \lfloor \tfrac{n}{2} \rfloor\ lg(\lfloor \tfrac{n}{2} \rfloor) + 4c \lceil \tfrac{n}{2} \rceil\ lg(\lceil \tfrac{n}{2} \rceil) + cn \\
      \text{(HB)} &\leq 4c \cdot lg(\tfrac{2}{3}n) \cdot (\lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil + cn \\
      &\leq 4c \cdot lg(\tfrac{2}{3}n) \cdot n + cn \\
      \text{(HB)} &\leq 4cn \cdot (lg(\tfrac{2}{3}) + lg(n)) + cn \\
      &= 4cn \cdot lg(n) + 4cn \cdot lg(\tfrac{2}{3}) \\
      &= 4cn \cdot lg(n) + cn(1 + 4 \cdot (lg(2) - lg(3))) \\
      &\leq 4cn \cdot lg(n) \\
      &\Rightarrow \Theta(n\ lg(n))
      \end{aligned}$$

- **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 Subsitutionsmethode zu erraten
  - **Beispiel:** $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$
    - *Vorüberlegungen:*
      - $\Rightarrow T(n) = 3T(\frac{n}{4}) + cn^2 \ (c > 0)$
      - 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 soviele Knoten wie darüber liegende
      - Anzahl der Knoten in Tiefe i ist 3^i
      - Kosten $c(\frac{n}{4^i})^2 \ , \ i = 0...log_4 n - 1$
      - Anzahl $\cdot$ Kosten $= 3^i \cdot c(\frac{n}{4^i})^2 = (\frac{3}{16})^i \cdot cn^2$
    - *Unterste Ebene:*
      - $3^{log_4(n)} = nlog_4(3)$ 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:*
      - $\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + ... + (\frac{3}{16})^{log_4 n - 1} cn^2 + \Theta(n^{log_4(3)}) \\ &= \sum_{i=0}^{log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{log_4^3}) \\ &= \frac{(\frac{3}{16}^{log_4 n}) - 1}{\frac{3}{16} - 1} \cdot cn^2 + \Theta(n^{log_4 3}) \end{aligned}$
        (Verwendung der geometrischen Reihe)
      - Verwendung einer unendlichen fallenden geometrischen Reihe als obere Schranke:
        $\begin{aligned} T(n) &= \sum_{i=0}^{log_4 n - 1} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{log_4 3}) \\ &< \sum_{i=0}^{\infty} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{log_4 3}) \\ &= \frac{1}{1 - \frac{3}{16}} \cdot cn^2 + \Theta(n^{log_4 3}) \\ &= \frac{16}{13} \cdot cn^2 + Theta(n^{log_4 3}) = O(n^2) \end{aligned}$
    - *Jetzt **Subsitutionsmethode:***
      - Zu zeigen: $\exists d > 0 : T(n) \leq dn^2$
      - Induktionsanfang:
        $\begin{aligned} T(n) &= 3 \cdot T(\lfloor \tfrac{1}{4} \rfloor) + c \cdot 1^2 \\ &= 3 \cdot T(0) + c = c \end{aligned}$
      - Induktionsschritt:
        $\begin{aligned} T(n) &\leq 3 \cdot T(\lfloor \tfrac{n}{4} \rfloor) + cn^2 \\ &\leq 3 \cdot d(\lfloor \tfrac{n}{4} \rfloor)^2 + cn^2 \\ &\leq 3d(\tfrac{n}{4})^2 + cn^2 \\ &= \tfrac{3}{16} dn^2 + cn^2 \\ &\leq dn^2, \text{ falls } d \geq \tfrac{16}{13} c \end{aligned}$

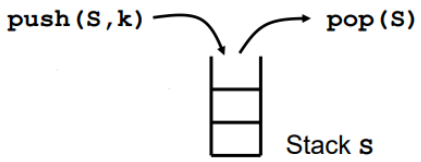
- **Mastertheorem**
  - *Idee:*

    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 $T(n) = a\ T(\frac{n}{b}) + f(n)$ defininiert, 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** $\Rightarrow f(n) = \Omega(n^{log_b(a+0.1)})$
      - $af(\frac{n}{b}) = 3f(\frac{n}{4}) = 3(\frac{n}{4})\ lg(\frac{n}{4}) \leq \frac{3}{4} n\ lg(n)$
      - Damit ist auch die Randbedingung erfüllt und $T(n) = \Theta(n\ lg(n))$

3 Grundlegende Datenstrukturen

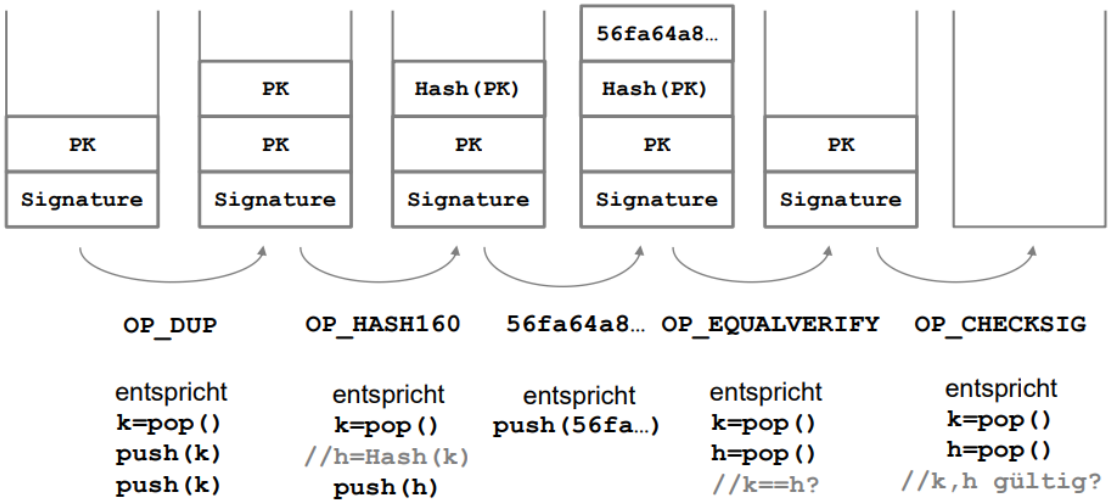
3.1 Stacks

- **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

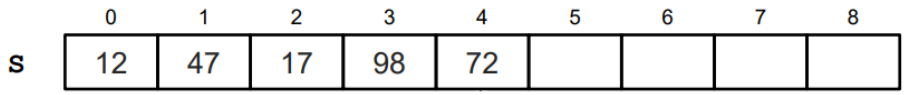


- **Beispiel Bitcoin**

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



- **Stacks als Array**



  - `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 $\Rightarrow$ Vergößerung 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

```
new(S)

1  S.A[]=ALLOCATE(1);
2  S.top=-1;
3  S.memsize=1;
```

```
isEmpty(S)

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

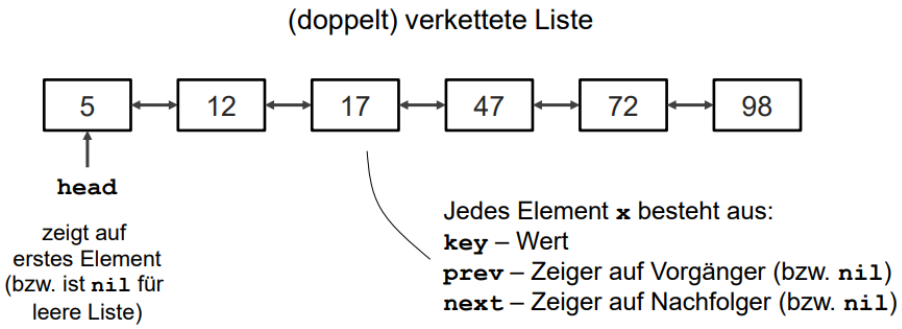
```
pop(S)

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

```
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);
```

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

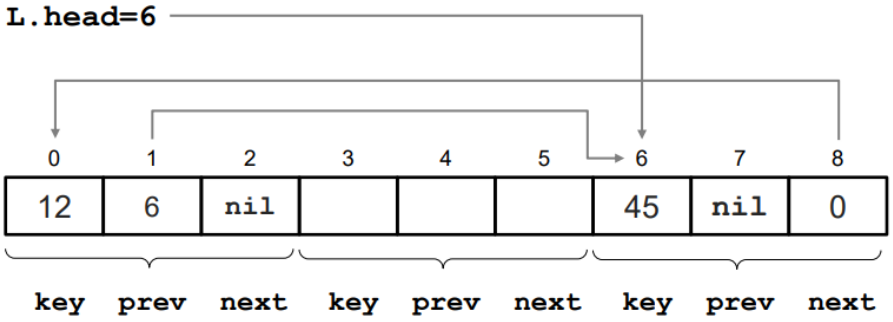
3.2 Verkettete Listen

- **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:

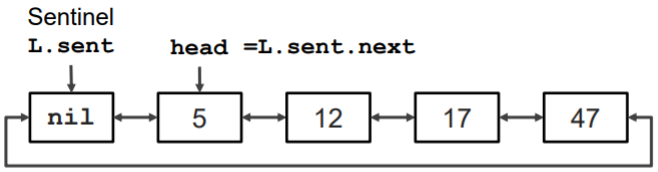
```
insert(L,x)
x.next = l.head;
x.prev = nil;
IF L.head != nil THEN
    L.head.prev = x;
L.head = x;
```

  - *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



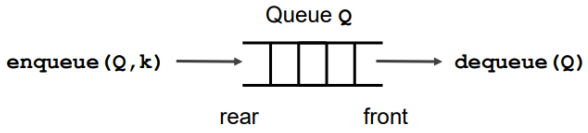
Sentinel ist „von außen“ nicht sichtbar Leere Liste besteht nur aus Sentinel

  - Löschen mit Sentinels:

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

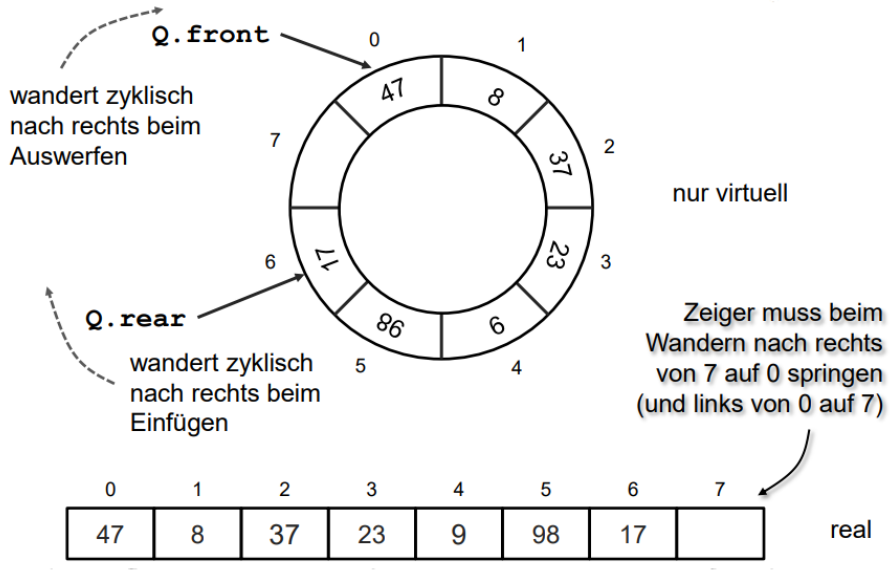
3.3 Queues

- **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;
```

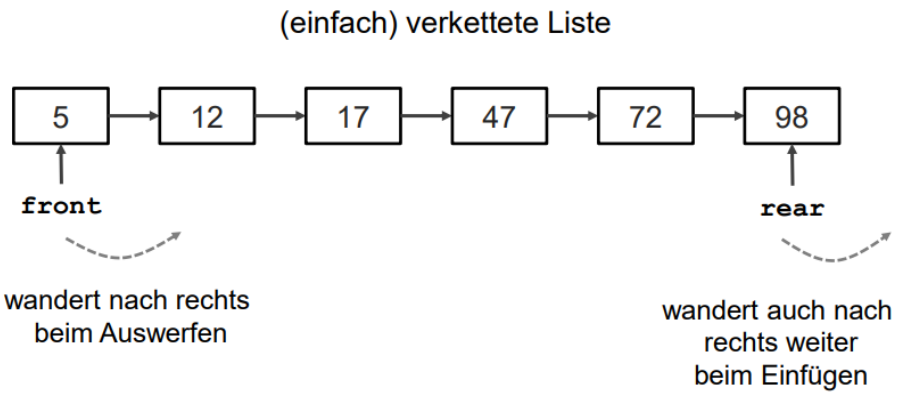
Q voll, wenn **front==rear** und **empty==false**

```
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**



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;
```

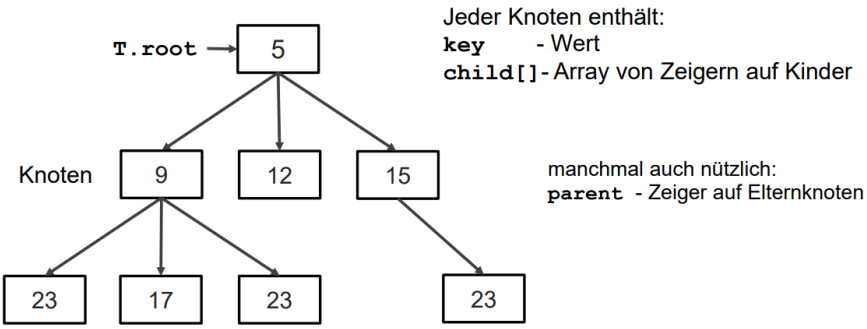
```
dequeue(Q)

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

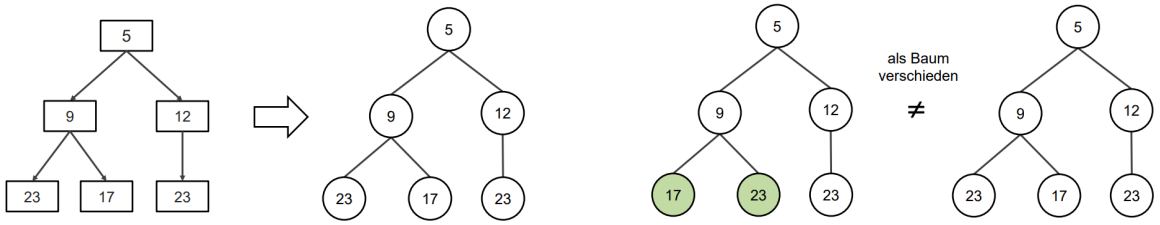
- **Bäume durch verkettete Listen**



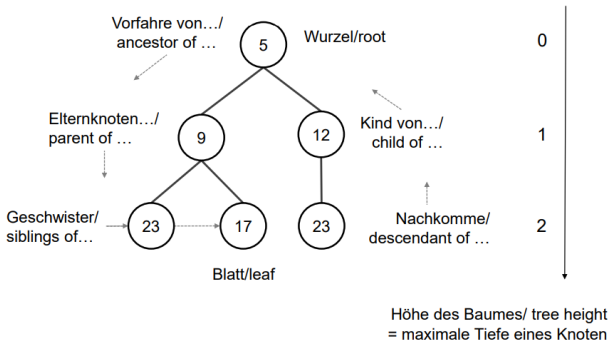
Baum-Bedingung: Baum ist leer oder…
es gibt einen Knoten `r` („Wurzel"), so dass jeder Knoten `v` von der Wurzel aus per eindeutiger Sequenz von `child`-Zeigern erreichbar ist:
`v = r.child[i1].child[i2]. … .child[im]`

Bäume sind "azyklisch" (Keine rückführende Spur")

- **Darstellung als (ungerichteter) Graph**

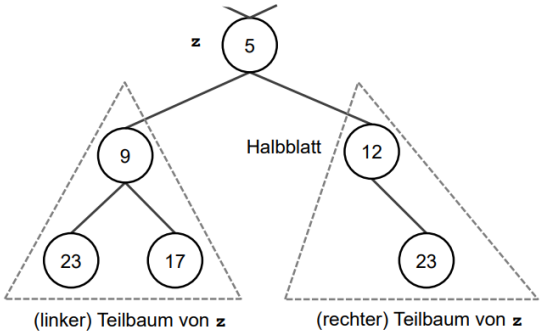


- **Allgemeine Begrifflichkeiten**



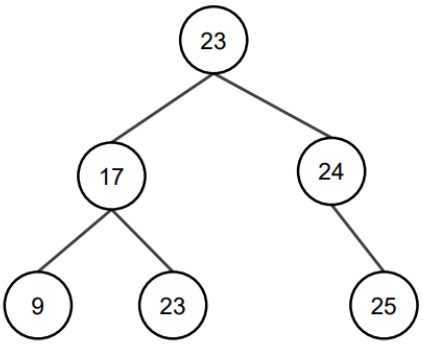
- Blatt: Knoten ohne Nachfolger
- Nachkomme von x:
  Erreichbar durch Pfad ausgehend von x

- **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
- Hohe (nicht-leerer) Baum:
  max{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

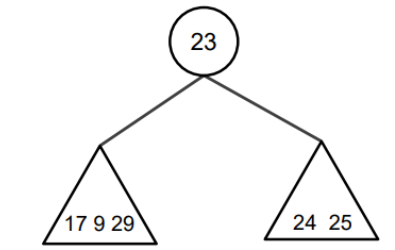
Code:

```
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;
```

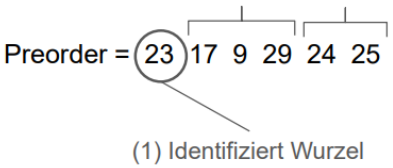
- **Eindeutige Bestimmbarkeit von Bäumen**
  - Nur In-,Pre-,Postorder reichen nicht zur eindeutigen Bestimmbarkeit von Bäumen
    $\Rightarrow$ Preorder/Postorder + Inorder + eindeutige Werte sind notwendig

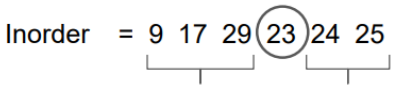


Pre = 17 9 29
In = 9 17 29

Pre = 24 25
In = 24 25

Bilde Teilbäume rekursiv





(2) Identifiziert Werte im linken und rechten Teilbaum

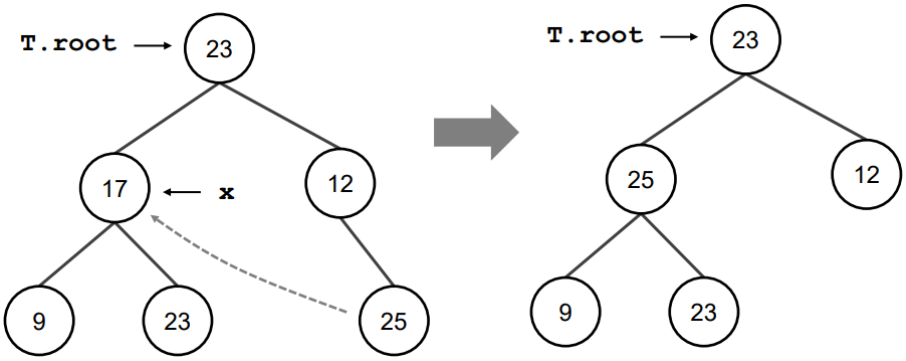
- **Abstrakter Datentyp Baum**
  - *Abstrakter Aufbau:*
    - `new T()`
      - Erzeugt neuen Baum namens `t`
    - `t.search(k)`
      - Gibt Element `x` in Baum `t` mit `x.key == k` zurück
    - `t.insert(k)`
      - Fügt Element `x` in Baum `t` hinzu
    - `t.delete(x)`
      - Löscht `x` aus Baum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

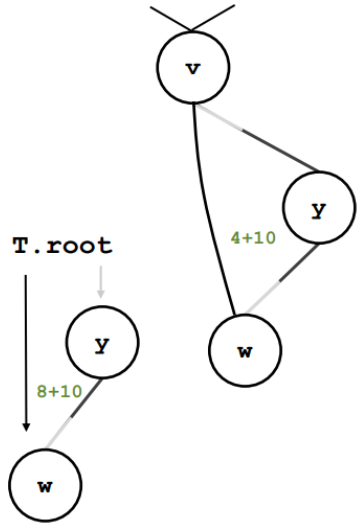
  - *Einfügen von Elementen:*
    - Laufzeit $= \Theta(1)$
    - Hier wird als Wurzel eingefügt (Achtung: Erzeugt linkslastigen Baum)
    - Code:

```
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;
```

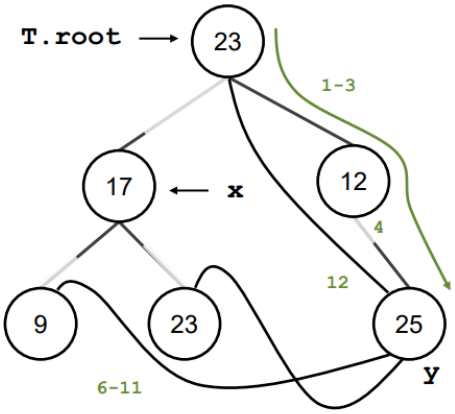
  - *Löschen von Elementen:*
    - Laufzeit $= \Theta(h)$ (Höhe des Baumes, $h = n$ möglich)
    - Hier: Ersetze x durch Halbblatt ganz rechts



- `Connect`-Algorithmus:



- `Delete`-Algorithmus:



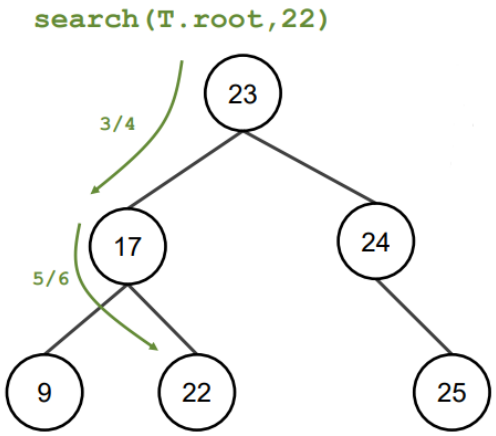
- Laufzeit $= \Theta(1)$

```
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.key` $\leq$ `z.key`
    Wenn y Knoten im rechten Teilbaum von z, dann `y.key` $\geq$ `z.key`
  - Preorder/Postorder + eindeutige Werte $\Rightarrow$ Eindeutige Identifizierung
- **Suchen im Binären Suchbaum**



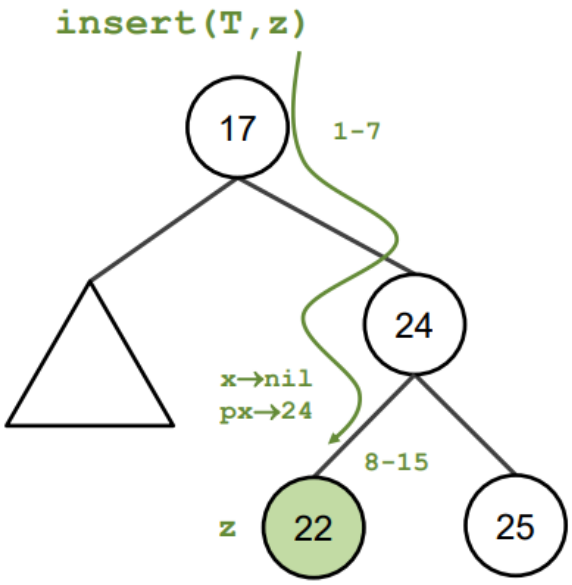
  - Laufzeit $= O(h)$ (Höhe)
  - Code:

```
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;
```

- **Einfügen im Binary Search Tree**

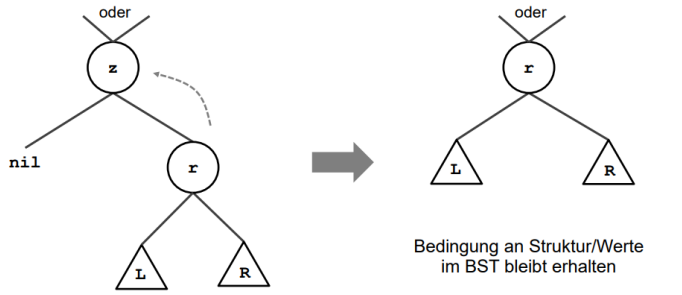


  - Laufzeit $= O(h)$
  - Aufwendiger, da Ordnung erhalten werden muss
  - Code:

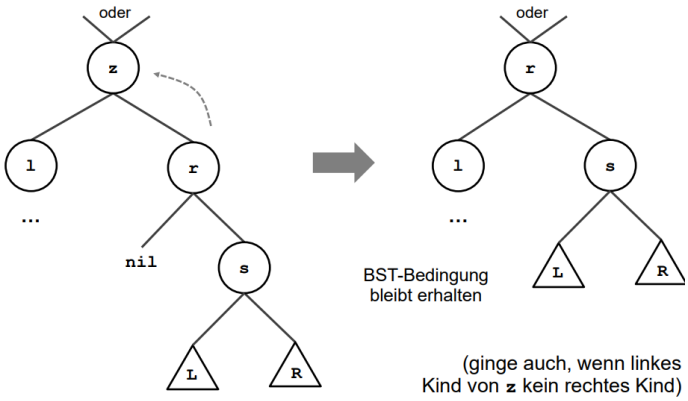
```
insert (T,z) // z.left == z.right == nil;
x = T.root;
px = nil;
WHILE x != nil DO
    px = x;
    IF x.key > z.key THEN
        x = x.left;
    ELSE
        x = x.right;
z.parent = px;
IF px == nil THEN
    T.root = z;
ELSE
    IF px.key > z.key THEN
        px.left = z;
    ELSE
        px.right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

- **Löschen im BST**
  - *Verschiedene Fälle:*

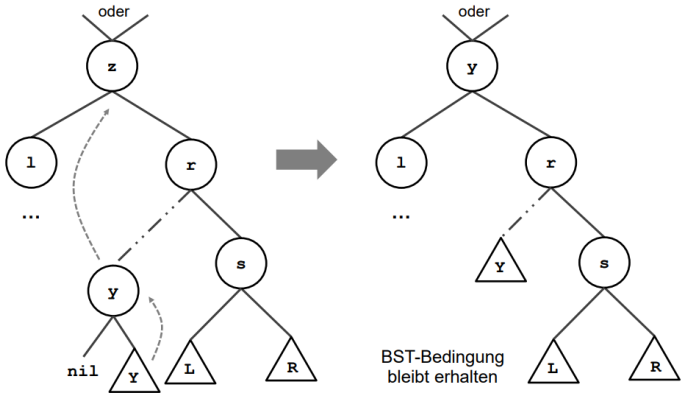
Löschen im BST (I) zu löschender Knoten z hat maximal ein Kind



Löschen im BST (II) rechtes Kind von Knoten z hat kein linkes Kind



Löschen im BST (III) „kleinster" Nachfahre vom rechten Kind von z



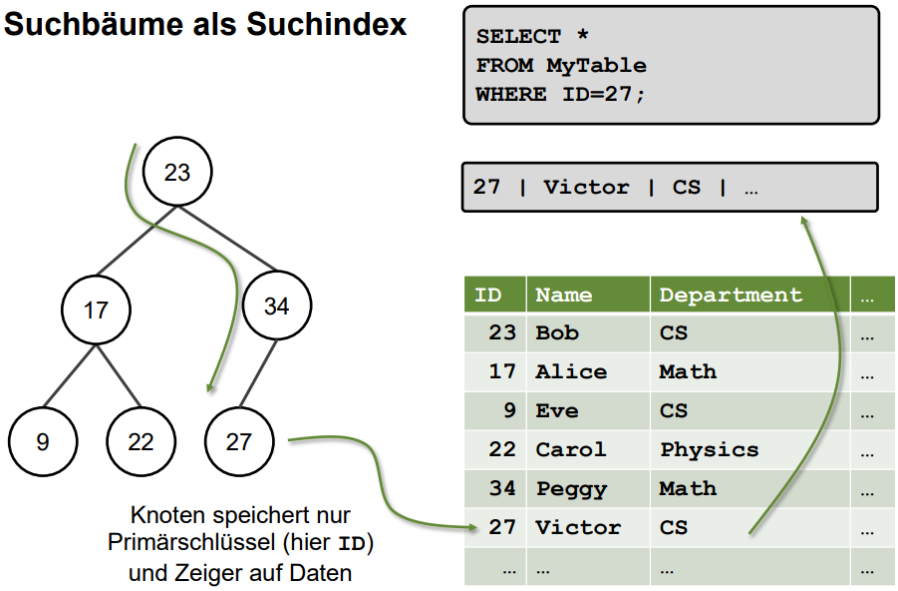
  - *Code*

```
// 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



4 Advanced Data Structures

4.1 Rot-Schwarz-Bäume

- **Definition**
  - Binärer Suchbaum mit Zusatzeigenschaften
  - Zusatzeigenschaften:
    - Jeder Knoten hat die Farbe rot oder schwarz
    - Die Wurzel ist schwarz
    - Wenn ein Knoten rot ist, sind seine Kinder schwarz ("Nicht-Rot-Rot-Regel")
    - Für jeden Knoten hat jeder Pfad zu einem Blatt die selbe Anzahl an gleichen schwarzen Knoten
  - Halbblätter im RBT sind schwarz
  - Schwarzhöhe eines Knoten:
    Eindeutige Anzahl von schwarzen Knoten auf dem Weg zu einem Blatt im Teilbaum des Knoten
  - Für leeren Baum gibt Schwarzhöhe = 0 ($SH(nil) = 0$)
  - Höhe eines Rot-Schwarz-Baums
    - $h \leq 2 \cdot log_2(n+1)$ (n Knoten)
    - In jedem Unterteilbaum gleiche Anzahl schwarzer Knoten
    - Maximal zusätzlich gleiche Anzahl roter Knoten auf diesem Pfad
    - Einigermaßen ausbalanciert ⇒ Höhe $O(log\ n)$
  - Alle folgenden Algorithmen arbeiten mithilfe eines Sentinels (zeigt auf sich selbst)
- **Einfügen**
  - Laufzeit: $\Theta(h)$ (h jedoch $log\ n$)

1. Finde Elternknoten wie im BST (BST-Einfüge Algorithmus)
2. Färbe den neuen Knoten rot
3. Wiederherstellen der Rot-Schwarz-Bedingung

```
fixColorsAfterInsertion(T,z)

WHILE z.parent.color == red DO                  // solange der Elternknoten rot ist
    IF z.parent == z.parent.parent.left THEN    // Linkes Kind (if-Fall)
        y = z.parent.parent.right;
        IF y != nil AND y.color == red THEN     // Fall 1
            z.parent.color = block;
            y.color = black;
            z.parent.parent.color = red;
            z = z.parent.parent;                // rekursiv nach oben weiterführen
        ELSE                                    // Fall 2
            IF z == z.parent.right THEN         // Zwischenfall (2.1)
                z = z.parent;
                rotateLeft(T,z);
            z.parent.color = black;
            z.parent.parent.color = red;
            rotateRight(T, z.parent.parent);
    ELSE                                        // Rechtes Kind (else-Fall)
        // Tauschen von rechts und links
    T.root.color = black;                       // Setzen der Wurzel auf Schwarz
```

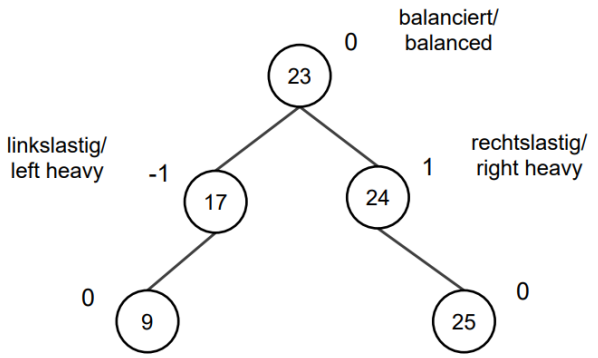
- Hilfsmethode `rotateLeft`

```
rotateLeft(T,x)

y = x.right;
x.right = y.left;
IF y.left != nil THEN
    y.left.parent = x;
y.parent = x.parent;
IF x.parent == T.sent THEN
    T.root = y;
ELSE
    IF x == x.parent.left THEN
        x.parent.left = y;
    ELSE
        x.parent.right = y;
y.left = x;
x.parent = y;
```

- **Löschen**
  - Laufzeit: $O(h) = O(log\ n)$
  - analog zum binären Suchbaum, aber neue Node erbt Farbe der alten Node
  - Wenn neueNode schwarz war $\Rightarrow$ Fixup
  - Verschiedene Fälle, die auch gegenseitig Voraussetzungen füreinander sind
  - Da das Ganze jedoch etwas umfangreicher ist, findet es sich nicht hier in der Zusammenfassung
- **Worst-Case-Laufzeiten**
  - Einfügen: $\Theta(log\ n)$
  - Löschen: $\Theta(log\ n)$
  - Suchen: $\Theta(log\ n)$

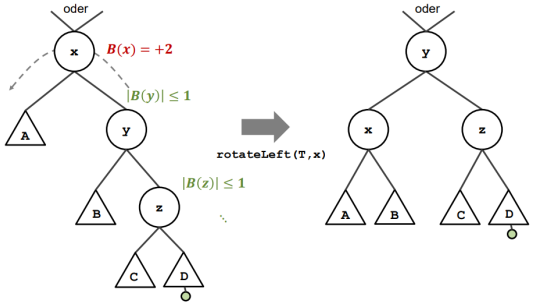
4.2 AVL-Bäume

- **Definition:**
  - $h \leq 1.441 \cdot log\ n$ (optimierte Konstanten - 1,441 vs 2 (RBT))
  - Binärer Suchbaum
  - Allerdings Balance in jedem Knoten nur $-1, 0, 1$
  - Balance für x: $B(x)$ = Höhe(rechter Teilbaum) - Höhe(linker Teilbaum)

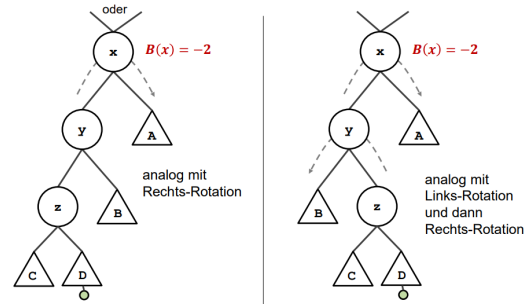


- **AVL vs. Rot-Schwarz**
  - *AVL:*
    - Einfügen und Löschen verletzen in der Regel öfter die Baum-Bedingung
    - Aufwendiger zum Rebalancieren
  - *Rot-Schwarz:*
    - Suchen dauert evtl. länger
  - *Konklusion:*
    - AVL geeigneter, wenn mehr Such-Operationen und weniger Einfügen und Löschen
  - Gemeinsamkeiten:
  - AVL $\subset$ Rot-Schwarz
  - AVL Baum $\Rightarrow$ Rot-Schwarz-Baum mit Höhe $\lceil \frac{h+1}{2} \rceil$
  - Für jede Höhe $h \geq 3$ gibt es einen RBT, der kein AVL-Baum ist (AVL $\neq$ RBT)
- **Einfügen**
  - Einfügen funktioniert wie beim Binary Search Tree mit Sentinel
  - Erfordert danach jedoch Rebalancieren weiter oben im Baum
  - Rebalancieren: (verschiedene Fälle)

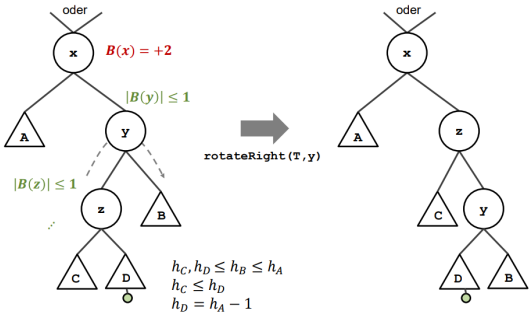
Rebalancieren: Fall I



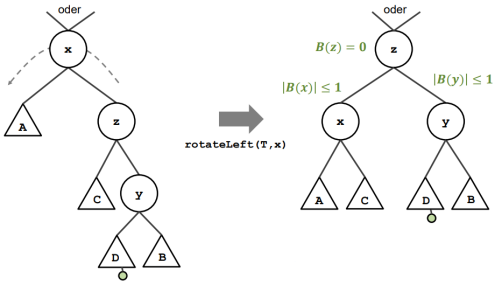
Rebalancieren: Fälle III+IV



Rebalancieren: Fall II (erste Rotation)



Rebalancieren: Fall II (zweite Rotation)



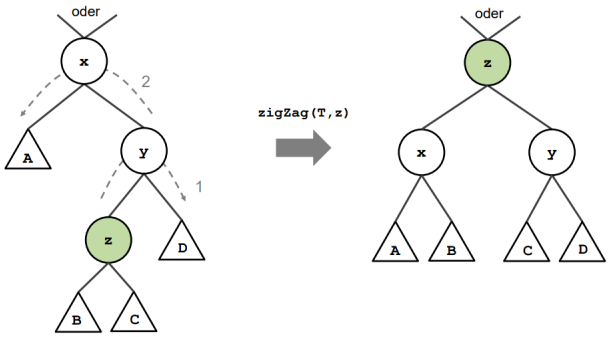
- **Löschen**
  - Analog zum binären Suchbaum
  - Rebalancieren bis eventuell in die Wurzel notwendig
- **Worst-Case-Laufzeiten**
  - Einfügen: $\Theta(log\ n)$
  - Löschen: $\Theta(log\ n)$
  - Suchen: $\Theta(log\ n)$
  - theoretisch bessere Konstanten als RBT
  - in Praxis aber nur unwesentlich schneller

4.3 Splay-Bäume

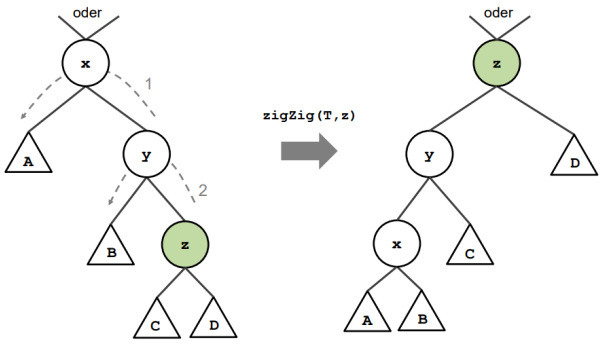
- **Definition**
  - selbst-organisierende Listen
  - Ansatz: Einmal angefragte Werte werdeb wahrs. noch öfter angefragt
  - Angefragte Werte nach oben schieben
  - Splay-Bäume sind Untermenge von BST
- **Splay-Operationen**
  - Suchen oder Einfügen: Spüle gesuchten oder neu eingefügten Knoten an die Wurzel
  - Splay: (Folge von Zig-,Zig-Zig-, Zig-Zag-Operationen)

```
splay(T,z)

WHILE z != T.root DO
    IF z.parent.parent == nil THEN
        zig(T,z);
    ELSE
        IF z == z.parent.parent.left.left OR
           z == z.parent.parent.right.right THEN
           zigZig(T,z);
        ELSE
            zigZag(T,z);
```

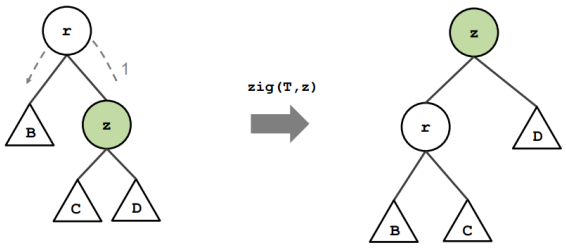
Zig-Zag-Operation =Rechts-Links- oder Links-Rechts-Rotation



Zig-Zig-Operation =Links-Links- oder Rechts-Rechts-Rotation



Zig-Operation =einfache Links- oder Rechts-Rotation

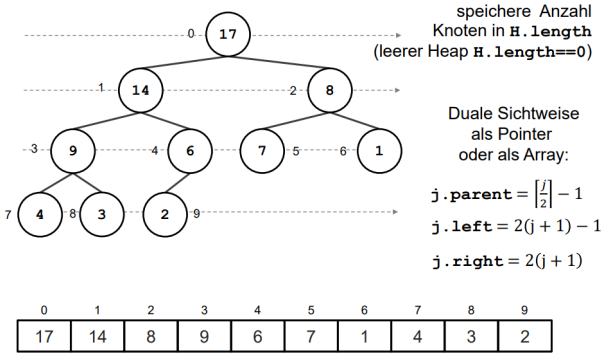


- **Suchen**
  - Laufzeit: $O(h)$
  - Suche des Knotens wie im BST
  - Hochspülen des gefundenen Knotens (alternativ zuletzt besuchter Knoten, falls nicht gefunden)
- **Einfügen**
  - Laufzeit: $O(h)$
  - Suche der Position wie im BST
  - Einfügen und danach hochspülen des eingefügten Knotens
- **Löschen**
  - Laufzeit: $O(h)$

  1. Spüle gesuchten Knoten per Splay-Operation nach oben
  2. Lösche den gesuchten Knoten (Wenn einer der beiden entstehenden Teilbäume leer, dann fertig)
  3. Spüle den größten Knoten im linken Teilbaum nach oben (kann kein rechtes Kind haben)
  4. Hänge rechten Teilbaum an größten Knoten aus 3. an
- **Laufzeit Splay-Bäume**
  - Amortisierte Laufzeit: Laufzeit pro Operation über mehrere Operationen hinweg
  - Worst-Case-Laufzeit pro Operation: $O(log_n\ n)$

4.4 Binäre Max-Heaps

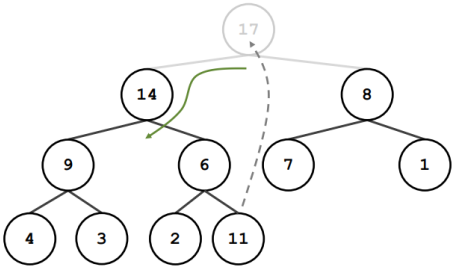
- **Definition**
  - Heaps sind keine BSTs
  - Eigenschaften binäre Max-Heaps:
    - bis auf das unterste Level vollständig und dort von links gefüllt ist
    - Für alle Knoten gilt: `x.parent.key` $\geq$ `x.key`
    - Maximum des Heaps steht damit in der Wurzel
  - $h \leq log\ n$, da Baum fast vollständig
- **Heaps durch Arrays**



- **Einfügen**
  - Idee: Einfügen und danach Vertauschen nach oben, bis Max-Eigenschaft wieder erfüllt ist
  - Laufzeit: $O(h) = O(log\ n)$

```
insert(H,k) // als unbeschränktes Array

H.length = H.length + 1;
H.A[H.length-1] = k;

i = H.length - 1;
WHILE i > 0 AND H.A[i] > H.A[i.parent]
    SWAP(H.A, i, i.parent);
    i = i.parent;
```

- **Lösche Maximum**
  1. Ersetze Maximum durch letztes"Blatt
  2. Vertausche Knoten durch Maximum der beiden Kinder (`heapify`)




```
extract-max(H)

IF isEmpty(H) THEN return error 'underflow';
ELSE
    max = H.A[0];
    H.A[0] = H.A[H.length - 1];
    H.length = H.length - 1;
    heapify(H, 0);
    return max;
```

```
heapify(H, i)

maxind = i;
IF i.left < H.length AND H.A[i]<H.A[i.left] THEN
    maxind = i.left;
IF i.right < H.length AND H.A[maxind]<H.A[i.right] THEN
    maxind = i.right;

IF maxind != i THEN
    SWAP(H.A, i, maxind);
    heapify(H, maxind);
```

- **Heap-Konstruktion aus Array**
  - Blätter sind für sich triviale Max-Heaps
  - Bauen von Max-Heaps für Teilbäume mithilfe Rekursion per `heapify`
  - (Array nicht unbedingt in richtiger Reihenfolge)

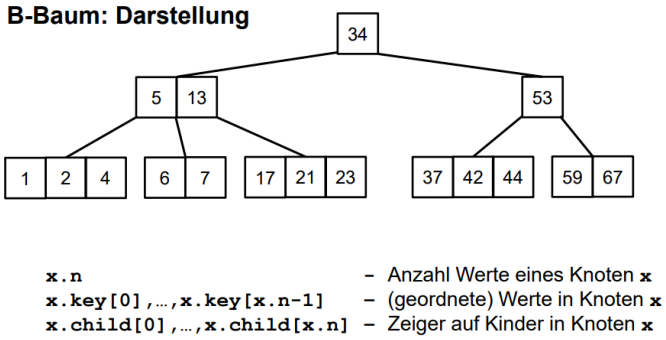
```
buildHeap(H.A) // Array in H.A

H.length = A.length;
FOR i = ceil((H.length-1)/2) - 1 DOWNTO 0 DO
    heapify(H.A,i);
```

- **Heap-Sort**
  - Idee: Bauen des Heaps aus Array und dann Extraktion des Maximums

```
heapSort(H.A)

buildHeap(H.A)                  // Bauen des Heaps
WHILE !isEmpty(H) DO
    PRINT extract-max(H);       // Ausgabe des Maximums bis Heap leer ist
```

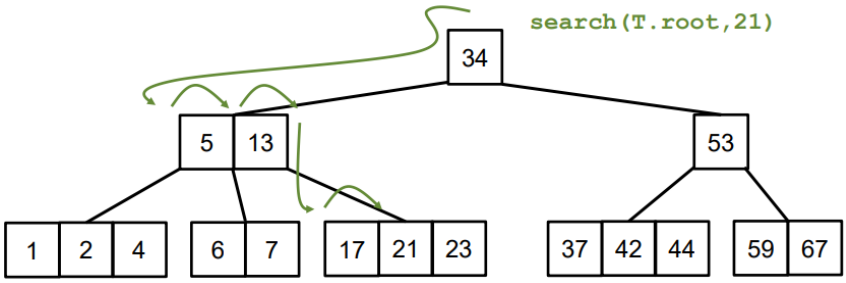
4.5 B-Bäume

- **Definition**
  - Jeder B-Baum hat einen angebenen Grad also z.B. $t = 2$
  - Eigenschaften:
    - Wurzel zwischen $[1, ..., 2t - 1]$ Werte
    - Knoten zwischen $[t - 1, ...2t - 1]$ Werte
    - Werte innerhalb eines Knotens aufsteigend geordnet
    - Blätter haben alle die gleiche Höhe
    - Jeder innere Knoten mit n Werten hat $n + 1$ Kinder, sodass gilt: $k_0 \leq key[0] \leq k_1 \leq key[1] \leq ... \leq k_{n-1} \leq key[n - 1] \leq k_n$



  - Höhe B-Baum: $h \leq log_t \frac{n+1}{2}$ (Grad t und n Werte)
  - B-Baum wird für größere t flacher

- **Suche**

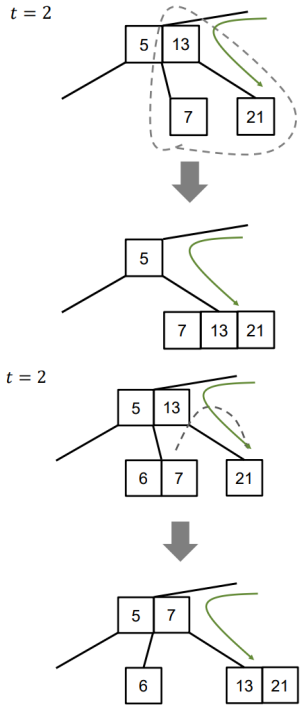



```
search(x, k)

WHILE x != nil DO
    i = 0;
    WHILE i < x.n AND x.key[i] < k DO
        i++;
    IF i < x.n AND x.key[i] == k THEN
        return(x, i);
    ELSE
        x = x.child[i];
return nil;
```

- **Einfügen**
  - Einfügen erfolgt immer in einem Blatt
  - Falls das Blatt voll ist, muss jedoch gesplittet werden
  - ⇒ Beim Durchlaufen des Baumes an jeder notwendigen (voll) Position splitten
  - Splitten:
    - Bricht volle Node auf und fügt mittleren Wert zur Elternnode hinzu
    - Aus den anderen Werten entstehen nun jeweils eigene Kinder
    - An der Wurzel splitten erzeugt neue Wurzel und erhöht Baumhöhe um eins
  - Ablauf zusammengefasst:
    1. Start bei Wurzel, falls kein Platz mehr splitten
    2. Durchlaufen des Baumes bis zur richtigen Position und immer, falls voll, splitten
    3. Einfügen der Node (fertig)

```
insert(T, z)

Wenn Wurzel schon 2t-1 Werte, dann splitte Wurzel
Suche rekursiv Einfügeposition:
    Wenn zu besuchendes Kind 2t-1 Werte, splitte es erst
Füge z in Blatt ein
```

- **Löschen**
  - Wenn Blatt noch mehr als t-1 Werte, kann der Wert einfach entfernt werden
  - Allerdings durchlaufen wir hier den Baum auch wieder von oben und stellen gewisse Voraussetzungen her
  - Durchlaufen des Baumes von oben und Anwendung der folgenden Algorithmen



Allgemeines Verschmelzen:

- Kind und alle rechten/linken Geschwisterknoten nur $t-1$ Werte
- Wenn Elternknoten vorher min. t Werte

  ⇒ keine Änderung oberhalb notwendig

Allgemeines Rotieren/Verschieben:

- Kind nur $t-1$ Werte
- Geschwister jedoch mehr als $t-1$ Werte
- keine Änderung oberhalb notwendig

  - Code:

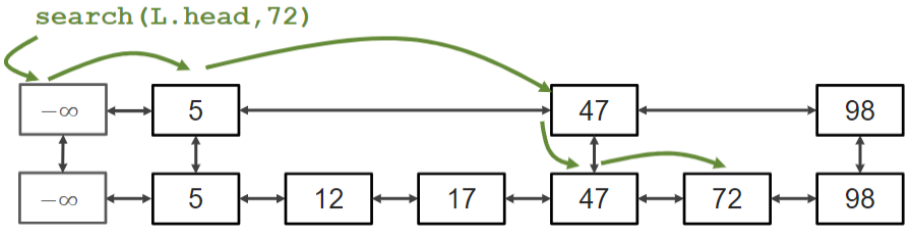
```
delete(T, k)

Wenn Wurzel nur 1 Wert und beide Kinder t-1 Werte,
verschmelze Wurzel und Kinder (reduziert Höhe um 1)
Suche rekursiv Löschposition:
    Wenn zu besuchendes Kind nur t-1 Werte,
    verschmelze es oder rotiere/verschiebe
Entferne Wert k im inneren Knoten/Blatt
// Ohne Probleme, aufgrund vorheriger Anpassung
```

- **Laufzeiten**
  - Einfügen: $\Theta(log_t\ n)$
  - Löschen: $\Theta(log_t\ n)$
  - Suchen: $\Theta(log_t\ n)$
  - Nur vorteilhaft wenn Daten blockweise eingelesen werden
  - O-Notation versteckt hier konstanten Faktor t für Suche innerhalb eines Knotens

5 Randomized Data Structures

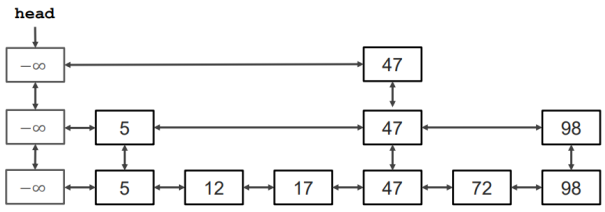
5.1 Skip Lists

- **Idee**
  - Einfügen von "Express-Liste" mit einigen Elementen
  - Beginne mit Suche in der Express-Liste mit weniger Elementen
  - Falls das suchende Element kleiner als nächstes Element in Express-Liste ⇒ weiter nach rechts
  - Falls nicht ⇒ Eine Stufe nach unten wandern und dort weiter suchen



  - Verbesserung: Zusätzliche Stufen an Express-Listen
  - Anwendung:
    - Gut für parallele Verarbeitung z.B. Multicore-Systeme (Einfügen und Löschen)
    - Dafür logarithmische Laufzeit nur im Durchschnitt
  - Auswahl von Elementen:
    - Abhängig von einer gewählten Wahrscheinlichkeit p
    - Element kommt mit Wahrscheinlichkeit p in übergeordnete Liste
    - Höhe: $h = O(log_{\frac{1}{p}} n)$
    - Anzahl Elemente: $n \Rightarrow pn \Rightarrow p^2n \Rightarrow$ ... (unten nach oben)

- **Implementierung**



L.head – erstes/oberstes Element der Liste
L.height – Höhe der Skiplist
x.key – Wert
x.next – Nachfolger
x.prev – Vorgänger
x.down – Nachfolger Liste unten
x.up – Nachfolger Liste oben
nil – kein Nachfolger / leeres Element

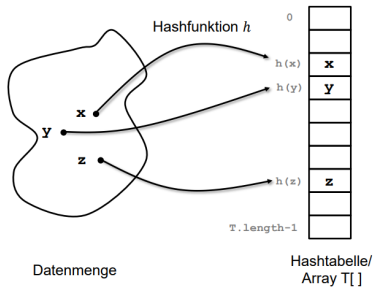
- **Suche**
  - Laufzeit ist von Expresslisten abhängig

```
search(L, k)
current = L.head;
WHILE current != nil DO
    IF current.key == k THEN
        return current;
    IF current.next != nil AND current.next.key <= k THEN
        current = current.next;
    ELSE
        current = current.down;
return nil;
```

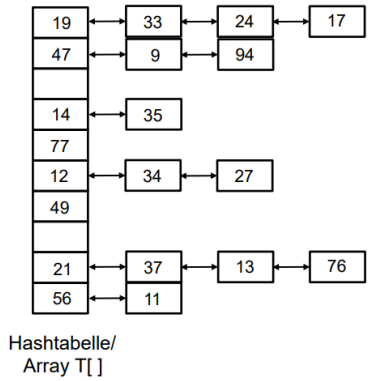
- **Einfügen**
  - Füge auf unterster Ebene ein
  - Evtl. auf höheren Ebenen mit zufälliger Wahl mithilfe von p auf jeder Ebene
- **Löschen**
  - Entferne Vorkommen des Elements aus allen Ebenen
- **Laufzeiten**
  - Einfügen: $\Theta(log_{\frac{1}{p}} n)$
  - Löschen: $\Theta(log_{\frac{1}{p}} n)$
  - Suchen: $\Theta(log_{\frac{1}{p}} n)$
  - (Im Durchschnitt)
  - O-Notation versteckt konstanten Faktor $\frac{1}{p}$
  - Speicherbedarf im Durchschnitt: $\frac{n}{1-p}$

5.2 Hashtables

- **Idee**



- Hashfunktion sollte gut verteilen
- $h(x)$ sollte uniform sein
- Unabhängig im Intervall $[0, T.length - 1]$ verteilt
- Einfügen mit konstant vielen Array-Operationen



- Kollisionsauflösung z.B. mithilfe von LinkedLists
- Neue Elemente werden vorne angefügt
- Konstante Anzahl an Array-Operationen
- Soviele Schritte wie die Liste lang ist
- Uniforme Hashfunktion

  $\Rightarrow \frac{n}{T.length}$ Einträge pro Liste

- **Hash-Funktionen**
  - Universelle Hash-Funktion:
    - Wähle zufällige $a, b \in [0, p-1]$, p $prim$, $a \neq 0$
    - $h_{a,b}(x) = ((a \cdot x + b) \ mod \ p) \ mod \ T.length$
  - Krypthographische Hash-Funktionen:
    - MD5, SHA-1, SHA-2, SHA-3
    - $h(x) = MD5(x) \ mod \ T.length$

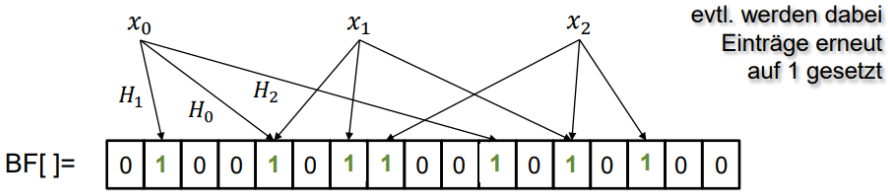
- **Hashtables vs. Bäume**
  - Hashtables:
    - nur Suche nach bestimmten Wert möglich
    - meist größer als zu erwartende Anzahl Einträge
  - Bäume:
    - schnelles Traversieren zu Nachbarn möglich
    - Bereichssuche möglich
- **Laufzeiten**
  - Wählt mal $T.length = n$ ergibt sich konstante Laufzeit
  - Einfügen: $\Theta(1)$
  - Löschen: $\Theta(1)$
  - Suchen: $\Theta(1)$
  - (Im Durchschnitt, beim Einfügen sogar im Worst-Case)
  - Speicherbedarf i.d.R. höher als n, meist ca. $1,33 \cdot n$

5.3 Bloom-Filter

- **Idee**
  - Speicherschonende Wörterbucher mit kleinem Fehler
  - z.B. Vermeidung von schlechten Passwörtern
    1. Abspeichern aller schlechten Passwörter in kompakter Form
    2. Prüfe, ob eingegebenes Passwort im Bloom-Filter
  - z.B. Erkennen von schädlichen Websites (Chrome früher)
- **Erstellen**
  - n Elemente $x_0, ..., x_{n-1}$
  - m Bits-Speicher z.B. als Bit-Array
  - k gute Hash-Funktionen $H_0, ..., H_{k-1}$ mit Bildbereich $0, 1, ..., m-1$
  - Empfohlene Wahl: $k = \frac{m}{n} \cdot ln2$ (Fehlerrate von ca. 2^{-k})
  - Code:

```
initBloom(X, BF, H) // H Array of hash functions

FOR i = 0 TO BF.length - 1 DO
    BF[i] = 0;
FOR i = 0 TO X.length - 1 DO
    FOR j = 0 TO H.length - 1 DO
        BF[H[j](X[i])] = 1;
```

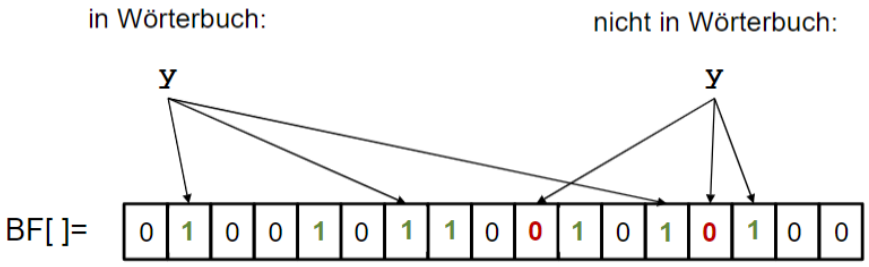
1. Initialisiere Array mit 0-Einträgen
2. Schreibe für jedes Element in jede Bit-Position $H_0(x_i), ..., H_{k-1}(x_i)$ eine 1



- **Suche**

```
searchBloom(BF, H, y)

result = 1;
FOR j = 0 TO H.length - 1 DO
    result = result AND BF[H[j](y)];
return result;
```

  - Gibt an, dass y im Wörterbuch, falls alle k Einträge für y in $BF = 1$ sind



  - Eventuell "false positives" (1, obwohl y nicht im Wörterbuch)
    - Passiert, falls die Einträge vorher von anderen Werten getroffen wurden
    - Daher gute Hashfunktionen und Filtergröße nicht zu klein

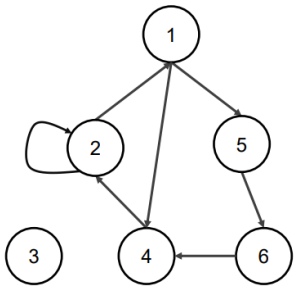
6 Graph Algorithms

6.1 Graphen

- **(Endlicher) gerichteter Graph**
  - (endlicher) gerichteter Graph $G = (V, E)$
  - besteht aus (endlicher) Knotenmenge V
  - besteht aus (endlicher) Kantenmenge $E \subseteq VxV$
  - $(u, v) \in E$: Kanten von Knoten u zu v
  - Kanten haben eine Richtung
- **Ungerichtete Graphen**
  - (endlicher) ungerichteter Graph $G = (V, E)$
  - besteht aus (endlicher) Knotenmenge V
  - besteht aus (endlicher) Kantenmenge $E \subseteq VxV$, sodass $(u, v) \in E \Leftrightarrow (v, u) \in E$
  - Kanten haben keine Richtung
- **Pfadfinder**
  - Knoten v ist von Knoten u erreichbar, wenn es einen Pfad gibt
  - u ist immer von u per leerem Pfad (k=1) erreichbar
  - Länge des Pfades $= k - 1 =$ Anzahl Kanten
- **Zusammenhänge**
  - Ungerichtet: Zusammenhängend wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist
  - Gerichtet: **Stark** zusammenhängend, wenn obiges auch gemäß Kantenrichtung gilt
- **Bäume und Subgraphen**
  Graph G ist ein Baum, wenn V leer ist oder wenn es einen Knoten in V gibt,
  von dem aus jeder andere Knoten eindeutig erreichbar ist (Wurzel).
  Graph $G' = (V', E')$ ist Subgraph von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$.

- **Darstellung von Graphen**
  - Als Adjazentmatrix (1, wenn Kante von i zu j / 0, wenn keine Kante)
  - Bei ungerichteten Graphen ist Matrix spiegelsymmetrisch zur Hauptdiagonalen
  - Speicherbedarf: $\Theta(|V^2|)$

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$



  - Auch darstellbar als Array mit verketteten Listen
  - Speicherbedarf: $\Theta(|V| + |E|)$
- **Gewichtete Graphen**
  - gewichteter gerichteter Graph $G = (V, E)$
  - besitzt zusätzlich Funktion $w : E \to R$
  - Abspeichern des Werts einer Kante $w((u, v))$

6.2 Breadth-First Search (BFS)

- **Idee**
  - Besuche zuerst alle unmittelbaren Nachbarn, dann deren Nachbarn, usw.
  - Anwendung: Webcrawling, Garbage Collection,..
- **Algorithmus**

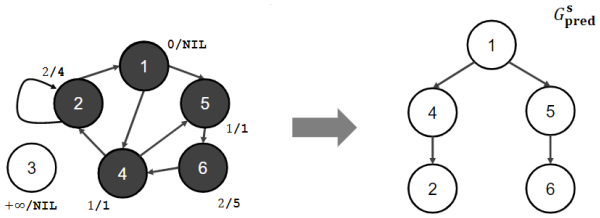
```
BFS(G,s) //G=(V,E) s = source node in V

FOREACH u in V-{s} DO
    u.color = WHITE;          // Weiß = noch nicht besucht
    u.dist = +∞               // Setzen der Distanzen auf Unendlich
    u.pred = nil;             // Setzen der Vorgänger auf nil
s.color = GRAY;               // Anfang bei Startnode
s.dist = 0;
s.pred = nil;
newQueue(Q);
enqueue(Q,s);
WHILE !isEmpty(Q) DO
u = dequeue(Q);
FOREACH v in adj(G,u) DO
    IF v.color == WHITE THEN
        v.color == GRAY;
        v.dist = u.dist+1;
        v.pred = u;
        enqueue(Q,v);
u.color = BLACK;              // Knoten abgearbeitet
```

  - Laufzeit: $O(|V| + |E|)$
  - Nach Algorithmus steht in v die kürzeste Distanz von s nach v

- **Kürzeste Pfade ausgeben**

```
print-path(G,s,v) // Assumes that BFS(G,s) has already been executed

IF v == s THEN
    print s;
ELSE
    IF v.pred == nil THEN
        print 'no path from s to v'
    ELSE
        print-path(G,s,v.pred);
        print v;
```

- **Abgeleiteter BFS-Baum**



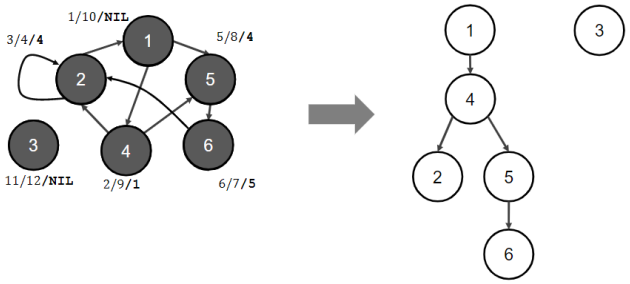
  - Subgraph $G^s_{pred} = (V^s_{pred}, E^s_{pred})$ von G:
    - $V^s_{pred} = \{v \in V | v.pred \neq nil\} \cup \{s\}$
    - $E^s_{pred} = \{(v.pred, v) | v \in V^s_{pred} - \{s\}\}$
  - G^s_{pred} enthält alle von s aus erreichbaren Knoten in G
  - Außerdem handelt es sich hier nur um kürzeste Pfade

6.3 Depth-First Search(DFS)

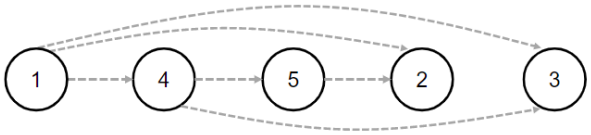
- **Idee**
  - Besuche zuerst alle noch nicht besuchten Nachfolgeknoten
  - "Laufe so weit wie möglich weg vom aktuellen Knoten"
- **Algorithmus**

```
DFS(G)

FOREACH u in V DO
    u.color = WHITE;
    u.pred = nil;
time = 0;               // time hier als globale Variable
FOREACH u in v DO
    IF u.color == WHITE THEN
        DFS-VISIT(G,u)  // Start eines rekursiven Aufrufs
DFS-VISIT(G,u)

time = time + 1;
u.disc = time;          // discovery time
u.color = GRAY;
FOREACH v in adj(G,u) DO
    IF v.color == WHITE THEN
        v.pred = u;
        DFS-VISIT(G,v);
u.color = BLACK;
time = time + 1;
u.finish = time;        // finish time
```

- **DFS-Wald = Menge von DFS-Bäumen**
  - Subgraph $G_{pred} = (V, E_{pred})$ von G
  - besteht aus $E_{pred} = (v.pred, v) | v \in V, v.pred \neq nil$
  - DFS-Baum gibt nicht unbedingt den kürzesten Weg wieder

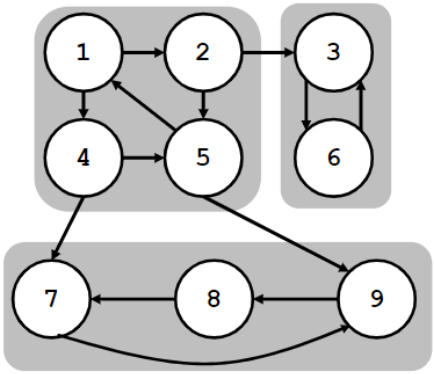


- **Kantenarten**
  - Baumkanten: alle Kanten in G_{pred}
  - Vorwärtskanten: alle Kanten in G zu Nachkommen in G_{pred}, die nicht Baumkante
  - Rückwärtskanten: alle Kanten in G zu Vorfahren in G_{pred}, die nicht Baumkante
  - Kreuzkanten: alle anderen Kanten in G (inkl. Schleifen)
- **Anwendungen DFS**
  - Job Scheduling (Job X muss vor Job Y beendet sein)
  - Topologisches Sortieren
    - nur für dag (directed acyclic graph)
    - Kanten immer nur nach rechts
    - Sortierung aber nicht eindeutig




```
TOPOLOGICAL-SORT(G)

newLinkedList(L);
run DFS(G) but, each time a node is finished, insert in front of L
return L.head
```

- **Starke Zusammenhangskomponenten**
  - Knotenmenge $C \subseteq V$, so dass
    es zwischen zwei Knoten $u, v \in C$ einen Pfad von u nach v gibt
    und es keine Menge $D \subseteq V$ mit $C \subsetneq D$ gibt, für die obiges auch gilt.



Eigenschaften:

- Verschiedene SCC's sind disjunkt
- Zwei SCC's sind nur in eine Richtung verbunden

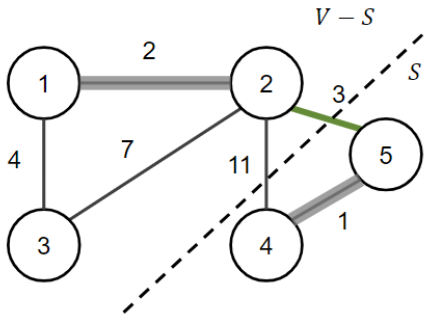
  - Algorithmus:
    - DFS zweimal laufen lassen
      Einmal auf Graph G
      Einmal auf Graph $G^T = (V, E^T)$ (transponiert)
    - Dadurch bleiben die SCC's gleich, die Kanten drehen sich aber jeweils um
    - Code:

```
SCC(G)

run DFS(G)
compute G^T
run DGS(G^T) but visit vertices in main loop
    in descending finish time from 1
output each DFS tree from above as one SCC
```

6.4 Minimale Spannbäume

- **Definition**
  - Verbindung aller Knoten miteinander
  - Minimaler Spannbaum ⇒ Minimales Gewicht
- **Allgemeiner Algorithmus**

```
genericMST(G,w)

A = ∅
WHILE A does not form a spanning tree for G DO
    find safe edge {u,v} for A
    A = A ∪{{u,v}}
return A
```



Terminologie:

- Schnitt (S, V-S) partioniert Knoten in zwei Mengen
- {u,v} überbrückt Schnitt, wenn $u \in S$ und $v \in V - S$
- Schnitt respektiert $A \subseteq E$, wenn keine Kante {u,v} aus A den Schnitt überbrückt
- {u,v} leichte Kante für (S, V-S), wenn w({u,v}) minimal für alle den Schnitt überbrückenden Kanten
- {u,v} sicher für A, wenn $A \cup \{\{u, v\}\}$ Teilmenge eines MST

- **Algorithmus von Kruskal**
  - Lässt parallel mehrere Unterbäume eines MST wachsen
  - In Worten: Suchen der "kleinsten" Kante und Zusammenfügen von Mengen, falls noch nicht geschehen
  - Laufzeit: $O(|E| \cdot log|E|)$

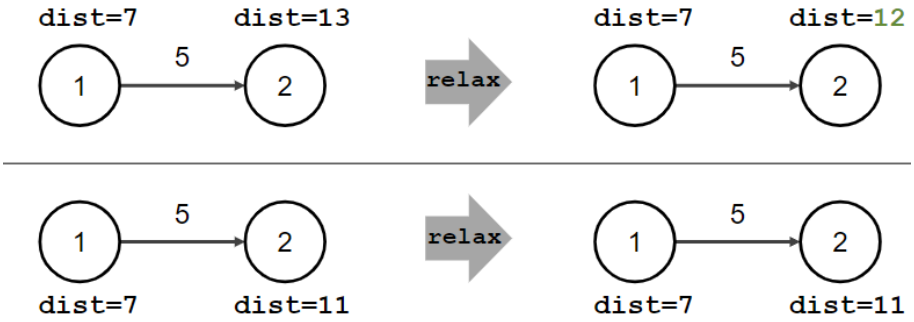
```
MST-Kruskal(G,w)

A = ∅
FOREACH v in V DO
    set(v) = {v};        // Menge mit sich selbst
Sort edges according to weight in nondecreasing order
FOREACH {u,v} in E according to order DO
    IF set(u) != set(v) THEN    // Mengen noch nicht verbunden
        A = A ∪ {{u,v}};
        UNION(G,u,v);   // Zusammenführen der Mengen aller Knoten aus den Sets
return A;
```

- **Algorithmus von Prim**
  - Konstruiert einen MST Knoten für Knoten
  - Fügt immer leichte Kante zu zusammenhängender Menge hinzu
  - Laufzeit: $O(|E| + |V| \cdot log|V|)$

```
MST-Prim(G,w,r) // r is given root

FOREACH v in V DO
    v.key = +∞;
    v.pred = nil;
r.key = -∞
Q = V;
WHILE !isEmpty(Q) DO
    u = EXTRACT-MIN(Q);     // smallest key value
    FOREACH v in adj(u) DO
        IF v∈Q and w({u,v})<v.key THEN
            v.key = w({u,v});
            v.pred = u;
```

6.5 Kürzeste Wege in (gerichteten) Graphen

- **Definition**
  - SSSP - Single-Source Shortest Path
  - Von Quelle s ausgehend die kürzesten Pfad zu allen anderen Knoten
  - Kürzester Pfad: Minimales Gewicht von einem zum anderen Knoten
  - BFS findet nur minimale Kantenwege (nicht Gewichtswege)
  - MST minimiert das Gesamtgewicht des Baumes (nicht zu einzelnen Kanten)
  - Negative Kantengewichte sind erlaubt, aber keine Zyklen mit negativem Gesamtgewicht
- **Gemeinsame Idee für Algorithmen - Relax**
  - Verringere aktuelle Distanz von Knoten v, wenn durch Kante (u, v) kürzer erreichbar




```
relax(G,u,v,w)

IF v.dist > u.dist + w((u,v)) THEN
    v.dist = u.dist + w((u,v));
    v.pred = u;
```

- **Bellman-Ford-Algorithmus**
  - Laufzeit: $\Theta(|E| \cdot |V|)$

```
Bellman-Ford-SSSP(G,s,w)

initSSSP(G,s,w);
FOR i = 1 TO V-1 DO
    FOREACH (u,v) in E DO
        relax(G,u,v,w);
FOREACH (u,v) in E DO   // Prüfung ob negativer Zyklus
    IF v.dist > u.dist+w((u,v)) THEN
        return false;
return true;

initSSSP(G,s,w)

FOREACH v in V DO
    v.dist = ∞;
    v.pred = nil;
s.dist = 0;
```

- **TopoSort für dag**
  - Erhalten des kürzesten Pfades durch das topologische Sortieren
  - Laufzeit: $\Theta(|E| + |V|)$

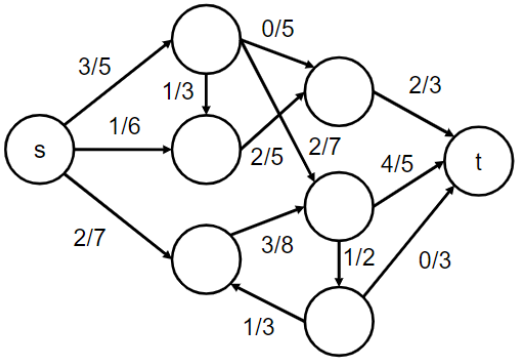
```
TopoSort-SSSP(G,s,w)     // G muss dag sein

initSSSP(G,s,w);
execute topological sorting
FOREACH u in V in topological order DO
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

- **Dijkstra-Algorithmus**
  - Voraussetzung: Keine negativen Kantengewichte
  - Laufzeit: $\Theta(|V| \cdot log|V| + |E|)$

```
Dijkstra-SSSP(G,s,w)

initSSSP(G,s,w);
Q = V;
WHILE !isEmpty(Q) DO
    u = EXTRACT-MIN(Q);      // smallest distance
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

6.6 Maximaler Fluss in Graphen

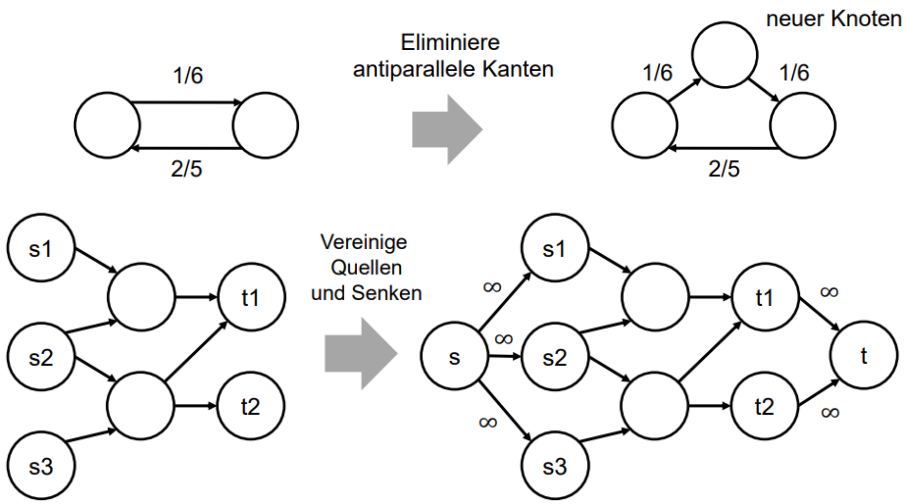
- **Idee**



  - Kanten haben Flusswert und maximale Kapazität
  - Jeder Knoten (außer s und t) haben den gleichen eingehenden und ausgehenden Fluss
  - Ziel: Finde maximalen Fluss von s nach t
  - s: Source/Quelle
  - t: Target/Senke

- **Flussnetzwerk:**
  Ein Flussnetzwerk ist ein gewichteter, gerichteter Graph $G = (V, E)$ mit Kapazität c, so dass $c(u, v) \geq 0$ für $(u, v) \in E$ und $c(u, v) = 0$ für $(u, v) \notin E$, mit zwei Knoten $s, t \in V$, so dass jeder Knoten von s aus erreichbar ist und t von jedem Knoten aus erreichbar ist. Damit gilt $|E| \geq |V| - 1$.
- **Fluss:**
  Ein Fluss $f : VxV \to \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t erfüllt $0 \leq f(u, v) \leq c(u, v)$ für alle $u, v \in V$, sowie für alle $u \in V - \{s, t\}$:
  $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$ (ausgehend = eingehend)
- **Wert eines Flusses**
  Der Wert $|f|$ eines Flusses $f : VxV \to \mathbb{R}$ für ein Flussnetzwerk G ist:
  $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, s)$

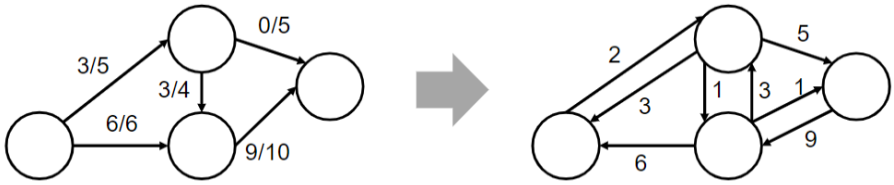
- **Transformationen**



- **Restkapazitätsgraph**
  - Wird für Ford-Fulkerson benötigt
  - Restkapazität $c_f(u,v)$:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{falls } (u,v) \in E \\ f(v,u) & \text{falls } (v,u) \in E \\ 0 & \text{sonst} \end{cases}$$

  - $G_f = (V, E_f)$ mit $E_f = \{(u,v) \in VxV | c_f(u,v) > 0\}$



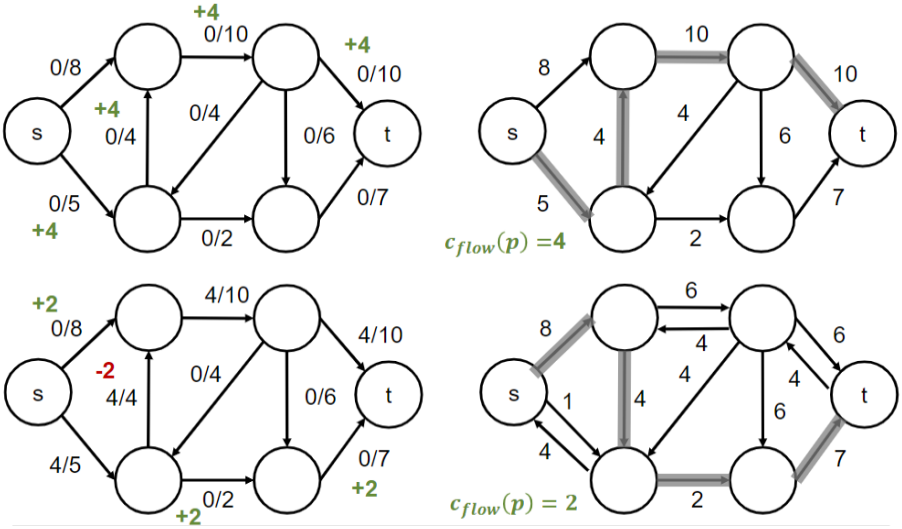
  - Suche eines Pfades von s nach t und Erhöhung aller Flüsse um niedrigsten möglichen Wert auf Pfad

- **Ford-Fulkerson-Algorithmus**
  - Idee: Suche Pfad von s nach t, der noch **erweiterbar** ist
  - Suche dieses Pfades im Restkapazitätsgraphen G_f (mögliche Zu- und Abflüsse)
  - Code:

```
Ford-Fulkerson(G,s,t,c)

FOREACH e in E do e.flow = 0;
WHILE there is path p from s to t in G_flow DO
    c_flow(p) = min {c_flow(u,v) : (u,v) in p}
    FOREACH e in p DO
        IF e in E THEN
            e.flow = e.flow + c_flow(p);
        ELSE
            e.flow = e.flow - c_flow(p);
```

  - Die Pfadsuche erfolgt z.B. per BFS oder DFS
  - Laufzeit: $O(|E| \cdot u \cdot |f^*|)$
    ($O(|V| \cdot |E|^2)$ Mit Verbesserung nach Edmonds-Karp)
    (wobei f^* maximaler Fluss und Fluss um bis zu $\frac{1}{u}$ pro Iteration wächst)

- Beispiel:



7 Advanced Designs

7.1 Dynamische Programmierung

- **Anwendung**

  Anwendung, wenn sich Teilprobleme überlappen

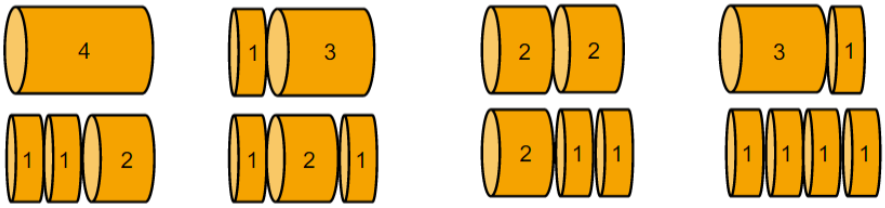
  1. Wir charakterisieren die Struktur einer optimalen Lösung
  2. Wir definieren den Wert einer optimalen Lösung rekursiv
  3. Wir berechnen den Wert einer optimalen Lösung (meist bottom-up Ansatz)
  4. Wir konstruieren eine zugehörige optimale Lösung aus berechneten Daten

- **Stabzerlegungsproblem**

  Ausgangsproblem: Stangen der Länge n cm sollen so zerschnitten werden, dass der Erlös r_n maximal ist, indem die Stange in kleinere Stäbe geschnitten wird.

Länge i	0	1	2	3	4	5	6	7	8	9	10
Preis p_i	0	1	5	8	9	10	17	17	20	24	30

  Beispiel: Gesamtstange hat Länge 4. Welchen Erlös kann man max. erhalten?



  Optimaler Erlös: zwei 2cm lange Stücke $(5 + 5 = 10)$

- *Aufteilung der Eisenstange:*
  - Stange mit Länge n kann auf 2^{n-1} Weisen zerlegt werden
  - Position i: Distanz vom linken Ende der Stange
  - Aufteilung in k Teilstäbe $(1 \leq k \leq n)$
  - optimale Zerlegung: $n = i_1 + i_2 + ... + i_k$
  - maximaler Erlös: $r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}$
  - z.B.: $r_4 = 10$ (siehe oben)

- *Rekursive Top-Down Implementierung:*

```
CUT-ROD(p,n)    // p Preis-Array, n Stangenlänge

IF n== 0
    return 0;
q = −∞;
FOR i = 1 TO n  // nicht Start bei 0, sonst kein Rekursionsschritt
    q = max(q, p[i] + CUT-ROD(p, n - i));
return q;
```

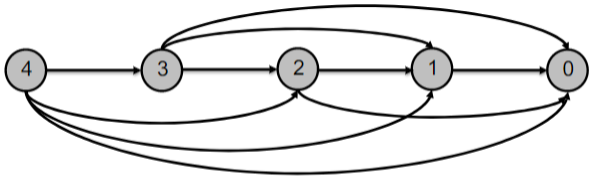
- *Stabzerlegung via Dynamischer Programmierung:*
  - Ziel:
    Mittels dynamischer Programmierung wollen wir `CUT-ROD` in einen effizienten Algorithmus verwandeln.
  - Bemerkung:
    Naiver rekursiver Ansatz ist **ineffizient**, da dieser immer wieder diesselben Teilprobleme löst.
  - Ansatz:
    Jedes Teilproblem nur einmal lösen. Falls die Lösung eines Teilproblems nochmal benötigt wird, schlagen wir diese nach.
  - Dynamische Programmierung wird zusätzlichen Speicherplatz benutzen um Laufzeit einzusparen.
  - Reduktion der exponentiellen Laufzeit auf polynomielle.
- *Rekursive Top-Down mit Memoisation:*
  - Idee: Speicherung der Lösungen der Teilprobleme
  - Laufzeit: $\Theta(n^2)$

```
MEMOIZED-CUT-ROD(p, n)

Let r[0...] be new array
FOR i = 0 TO n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)


MEMOIZED-CUT-ROD-AUX(p, n, r)       // r new Array

IF r[n] ≥ 0                         // Abfrage ob vorhanden
    return r[n]
IF n == 0
    q = 0
ELSE
    q = −∞
    FOR i = 1 to n
    q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q                            // Abspeichern
return q
```

- *Bottom-Up Ansatz:*
  - Laufzeit: $\Theta(n^2)$
  - Sortieren der Teilprobleme nach ihrer Größe und lösen in dieser Reihenfolge
  - Immer alle kleineren Teilprobleme bei bestimmten Wert bereits gelöst

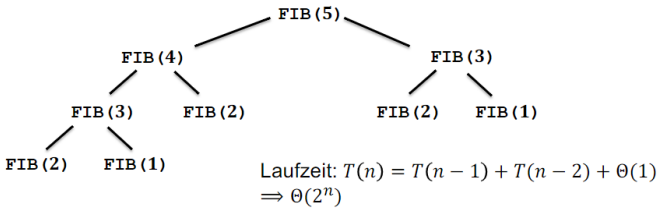
```
BOTTOM-UP-CUT-ROD(p, n)

Let r[0...n] be a new array
r[0] = 0
FOR j = i TO n
    q = −∞
    FOR i = 1 TO j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

  - Teilproblemgraph ($i \to j$ bedeutet, dass Berechnung von r_i den Wert r_j benutzt)



- **Fibonacci-Zahlen**
  - $F_1 = F_2 = 1$
  - $F_n = F_{n-1} + F_{n-2}$
  - *Naiver rekursiver Algorithmus:*

```
FIB(n)

IF n ≤ 2
    f = 1;
ELSE
    f = FIB(n-1) + FIB(n-2);
return f;
```



Gleiche Teilprobleme werden wieder mehrmals gelöst

- *Rekursiver Algorithmus mit Memoisation*
  - Wieder Abspeichern von Teilproblemen um Laufzeit einzusparen
  - Laufzeit: $\Theta(n)$

```
MEMOIZED-FIB(n)

Let m[0...n-1] be a new array
FOR i = 0 TO n - 1
    m[i] = 0
return MEMOIZED-FIB-AUX(n, m)


MEMOIZED-FIB-AUX(n, m)

IF m[n-1] != 0
    return m[n-1];       // Auslesen von gespeicherten Werten
IF n ≤ 2
    f = 1;
ELSE
    f = MEMOIZED-FIB-AUX(n-1, m) + MEMOIZED-FIB-AUX(n-2, m);
m[n-1] = f;
return f;
```

- *Bottom-Up Algorithmus*
  - Hier wieder Berechnen aller Teilprobleme von unten beginnend

```
BOTTOM-UP-FIB(n)

Let m[0...n-1] be a new array
FOR i = 1 TO n
    IF n ≤ 2
        f = 1;
    ELSE
        f = BOTTOM-UP-FIB(n-1) + BOTTOM-UP-FIB(n-2);
    m[i] = f;
return m[n-1];
```

7.2 Greedy-Algorithmus

- **Idee**
  - Trifft stets die Entscheidung, die in diesem Moment am besten erscheint
  - Trifft **lokale** optimale Entscheidung (evtl. nicht global die Beste)
- **Aktivitäten-Auswahl-Problem**
  - *Definition*
    - 11 anstehende Aktivitäten $S = \{a_1, ..., a_{11}\}$
    - Startzeit s_i und Endzeit f_i, wobei $0 \leq s_i < f_i < \infty$
    - Aktivität a_i findet im halboffenen Zeitintervall $[s_i, f_i)$ statt
    - Zwei Aktivititäten sind kompatibel, wenn sich deren Zeitintervalle nicht überlappen

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Aktivitäten: $\{a_3, a_9, a_{11}\}$
Aktivitäten: $\{a_1, a_4, a_8, a_{11}\}$
Aktivitäten: $\{a_2, a_4, a_9, a_{11}\}$

- *Ansatz mittels dynamischer Programmierung*
  - Menge von Aktivitäten, die starten nachdem a_i endet und enden, bevor a_j startet
    $S_{ij} = \{a \in S, a = (s, f) : s \geq f_i, f < s_j\}$
  - Definiere maximale Menge A_{ij} von paarweise kompatiblen Aktivitäten in S_{ij}.
    $c[i, j] = |A_{ij}|$
  - Optimale Lösung für Menge S_{ij} die Aktivitäten a_k enthält:
    $c[i, j] = max_{a_k \in S_{ij}}\{c[i, k] + c[k, j] + 1\}$ (0, falls $S_{ij} = \emptyset$)
- *Greedy-Wahl*
  - lokal die beste Wahl
  - Auswahl der Aktivität mit geringster Endzeit (möglichst viele freie Ressourcen)
  - Also hier Teilprobleme, die nach a_1 starten
  - $S_k = \{a_i \in S : s_i \geq f_k\}$: Menge an Aktivitäten, die starten, nachdem a_k endet
  - *Optimale-Teilstruktur-Eigenschaft*
    Wenn a_1 in optimaler Lösung enthalten ist, dann besteht optimale Lösung zu ursprünglichem Problem aus Aktivität a_1 und allen Aktivitäten zur einer optimalen Lösung des Teilproblems S_1
- *Rekursiver Greedy-Algorithmus*
  - Voraussetzung: Aktivitäten sind monoton steigend nach der Endzeit sortiert
  - Laufzeit: $\Theta(n)$

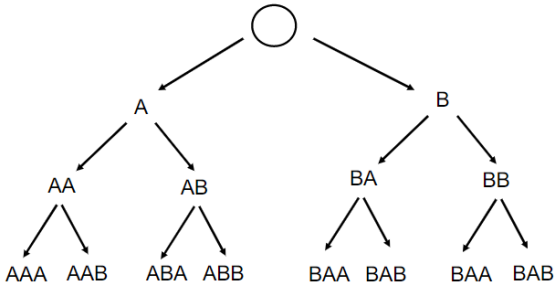
```
RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)
// s Anfangszeitenarray, f Endzeitenarray,
// k Index von Teilproblem, n Größe Anfangsproblem
m = k + 1;
WHILE m ≤ n and s[m] < f[k]  // Suche nach erster Kompatibilität
    m = m + 1;
IF m ≤ n
    // Ausgabe des Elements und Berechnung weiterer Aktivitäten
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)
ELSE
    return ∅
```

- *Iterativer Greedy-Algorithmus*
  - Voraussetzung: Aktivitäten sind monoton steigend nach der Endzeit sortiert
  - Laufzeit: $\Theta(n)$

```
GREEDY-ACTIVITY-SELECTOR(s,f)

n = s.length;
A = {a_1};
k = 1;          // Index des zuletzt hinzugefügten Elements in A
FOR m = 2 TO n
    IF s[m] ≥ f[k]           // Findet zuerst endende Aktivität in Menge
        A = A ∪ {a_m};   // Fügt diese hinzu, falls kompatibel
        k = m;
return A;
```

7.3 Backtracking

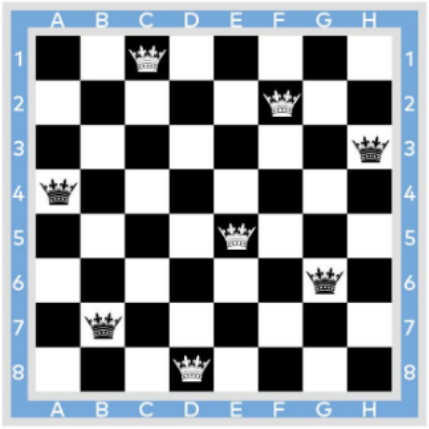
- **Suchbaum - Baum der Möglichkeiten**
  - Darstellung aller für ein Problem bestehenden Möglichkeiten
  - Problem: Dreimal hintereinander der selbe Buchstabe (A,B)



- **Backtracking - Idee**
  - Lösung finden via Trial and error
  - Schrittweises Herantasten an die Gesamtlösung
  - Falls Teillösung inkorrekt → Schritt zurück und andere Möglichkeit
  - Voraussetzung:
    - Lösung setzt sich aus Komponenten zusammen (Sudoku, Labyrinth,..)
    - Mehrere Wahlmöglichkeiten für jede Komponente
    - Teillösung kann getestet werden

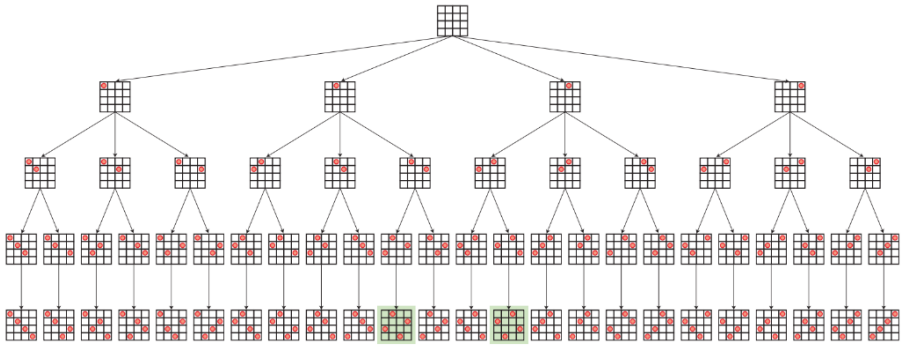
- **Damenproblem**

  Auf einem Schachbrett der Größe $n \cdot n$ sollen n Damen so positioniert werden, dass sie sich gegenseitig nicht schlagen können. Wie viele Möglichkeiten gibt es, n Damen so aufzustellen, dass keine Damen eine andere schlägt.



  - $n = 8$: 4 Milliarden Positionierungen
  - Optimierte Suche: In jeder Zeile/Spalte nur eine Dame
  - Reduziert Problem auf 40.000 Positionierungen (ohne Diagonale)

```
PLACE-QUEENS(Q,r) // Q Array, r Index der ersten leeren Zeile

IF r == n
    return Q
ELSE
    FOR j = 0 TO n - 1 // Mögliche Positionierungen
        legal = true;
        FOR i = 0 TO r - 1  // Evaluation der mgl. Bedrohungen
            IF (Q[i] == j) OR (Q[i==j + r - i]) OR (Q[i] == j - r + i)
                legal = false;
        IF legal == true
            Q[r] = j;
            PLACE-QUEENS(Q, r + 1)
```

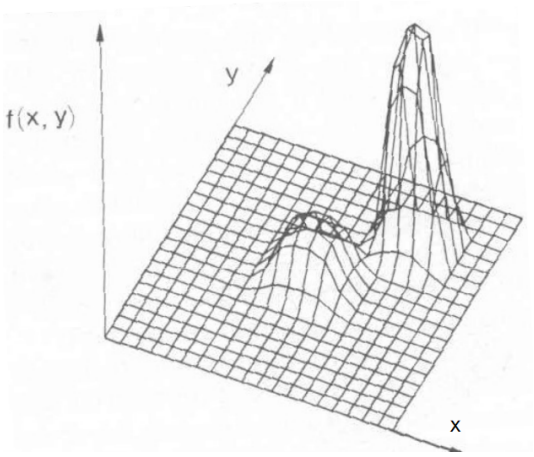


- **Allgemeiner Backtracking-Algorithmus**

```
BACKTRACKING(A, s)

IF alle Komponenten richtig gesetzt
    return true;
ELSE
    WHILE auf aktueller Stufe gibt es Wahlmöglichkeiten
        wähle einen neuen Teillösungsschritt
        Teste Lösungsschritt gegen vorliegende Einschränkungen
        IF keine Einschränkung THEN
            setze die Komponente
        ELSE
            Auswahl(Komponente) rückgängig machen
        BACKTRACKING(A, s + 1)
```

7.4 Metaheuristiken

- **Optimierungsproblem**



  - Lösungsstrategien:
    - Exakte Methode
    - Approximationsmethode
    - Heuristische Methode
  - Einschränkungen
    - Antwortzeit
    - Problemgröße

    ⇒ exkludieren oft exakte Methoden

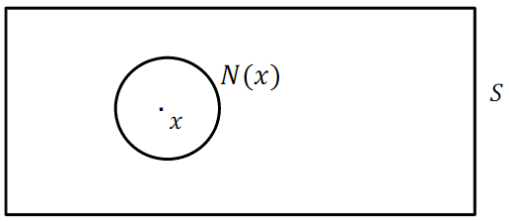
- **Heuristik**
  - Technik um Suche zur Lösung zu führen
  - Metaheuristik (Higher-Level-Strategie)
    - soll z.B. Hängenbleiben bei lokalem Maxima verhindern
  - *Leiten einer Suche*
    1. Finde eine Lösung (z.B. mit Greedy-Algorithmus)
    2. Überprüfe die Qualität der Lösung
    3. Versuche eine bessere Lösung zu finden
       - Herausfinden in welcher Richtung bessere Lösung evtl. liegt
       - ggf. Wiederholung dieses Prozesses
  - *Finden einer besseren Lösung*
    - Modifikation der Lösung durch erlaubte Operationen
    - Dadurch erhalten wir Nachbarschaftslösungen

      ⇒ Suche nach besseren Lösungen in der Nachbarschaft

- **Rucksackproblem**

	1	2	3	4	5	6	7	8	9
Wert	79	32	47	18	26	85	33	40	45
Größe	85	26	48	21	22	95	43	45	55

  - Rucksack hat eine Kapazität von 101, 9 verschiedene Gegenstände
  - Ziel: Höchster Wert der Gegenstände im Rucksack
  - Beispiellösung: 3 + 5 (Wert 73, Größe 70)
  - Nachbarschaftslösungen:
    - 2,3 und 5: Wert 105, Größe 96
    - 1,3 und 5: Wert 152, Größe 155 (problematisch)
    - 3: Wert 47, Größe 48



Nachbarschaft:

- Suchraum S kann sehr groß sein
- Einschränkung des Suchraums in der Nähe des Punktes
- Distanzfunktion $d : SxS \to \mathbb{R}$
- Nachbarschaft: $N(x) = \{y \in S : d(x, y) \leq \epsilon\}$

- **Zufällige Suche**
  - *Idee und Ablauf*
    - Suche nach globalem Optimum
    - Anwenden der Technik auf **aktuelle** Lösung im Suchraum
    - Wahl einer neuen zufälligen Lösung in jeder Iteration
    - Falls die neue Lösung besseren Wert liefert ⇒ neue **aktuelle** Lösung
    - Terminierung, falls keine weiteren Verbesserungen oder Zeit vorbei
  - *Code*

```
RANDOM-SEARCH

best <- irgendeine initiale zufällige Lösung
REPEAT
    S <- zufällige Lösung
    IF (Quality(S) > Qualityy(best)) THEN
        best <- S
UNTIL best ist die ideale Lösung oder Zeit ist vorbei
return best
```

  - *Nachteile*
    - Potentiell lange Laufzeit
    - Laufzeit abhängig von der initialien Konfiguration
  - *Vorteile*
    - Algorithmus **kann** beim globalen Optimum terminieren
- **Bergsteigeralgorithmus**
  - *Idee und Ablauf*
    - Nutzung einer iterativen Verbesserungstechnik
    - Anwenden der Technik auf **aktuelle** Lösung im Suchraum
    - Auswahl einer neuen Lösung aus Nachbarschaft in jeder Iteration
    - Falls diese besseren Wert liefert, überschreiben der **aktuellen** Lösung
    - Falls nicht, Wahl einer anderen Lösung aus Nachbarschaft
    - Terminierung, falls keine weiteren Verbesserungen oder Zeit vorbei
  - *Code*

```
HILL-CLIMBER
T <- Distribution von möglichen Zeitintervallen
S <- irgendeine initiale zufällige Lösung
best <- S
REPEAT
    time <- zufälliger Zeitpunkt in der Zukunft aus T
    REPEAT
        wähle R aus der Nachbarschaft von S
        IF Quality(R) > Quality(S) THEN
            S <- R
    UNTIL S ist ideale Lösung oder time ist erreicht oder totale Zeit erreicht
    IF Quality(S) > Quality(best) THEN
        best <- S
    S <- irgendeine zufällige Lösung
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

- *Nachteile*
  - Algorithmus terminiert in der Regel bei lokalem Optimum
  - Keine Auskunft, inwiefern sich lokale Lösung von Globaler unterscheidet
  - Optimum abhängig von Initialkonfiguration
- *Vorteile*
  - Einfach anzuwenden

- **Iterative lokale Suche**
  - *Idee und Ablauf*
    - Suche nach anderen lokalen Optima bei Fund eines lokalen Optimas
    - Lösungen nur in der Nähe der "Homebase"
    - Entscheidung, ob neue oder alte Lösung
    - Bergsteigeralgo zu Beginn, danach aber großen Sprung um anderes Optimum zu finden
  - *Code*

```
ITERATIVE-LOCAL-SEARCH
T <- Distribution von möglichen Zeitintervallen
S <- irgendeine initiale zufällige Lösung
H <- S      // Wahl des Homebasepunktes
best <- S
REPEAT
    time <- zufälliger Zeitpunkt in der Zukunft aus T
    REPEAT
        wähle R aus der Nachbarschaft von S
        IF Quality(R) > Quality(S) THEN
            S <- R
    UNTIL S ist ideale Lösung oder time ist erreicht oder totale Zeit erreicht
    IF Quality(S) > Quality(best) THEN
        best <- S
    H <- NewHomeBase(H,S)
    S <- Perturb(H)
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

  - Perturb:
    - ausreichend weiter Sprung (außerhalb der Nachbarschaft)
    - Aber nicht soweit, dass es eine zufällige Wahl ist
  - NewHomeBase:
    - wählt die neue Startlösung aus
    - Annahme neuer Lösungen nur, wenn die Qualität besser ist

- **Simulated Annealing**
  - *Idee und Ablauf*
    - Wenn neue Lösung besser, dann wird diese immer gewählt
    - Wenn neue Lösung schlechter, wird diese mit gewisser Wahrscheinlichkeit gewählt $Pr(R, S, t) = e^{\frac{Quality(R)-Quality(S)}{t}}$
    - Der Bruch ist negativ, da R schlechter ist als S
  - *Code*

```
SIMULATED-ANNEALING
t <- Temperatur, initial eine hohe Zahl
S <- irgendeine initiale zufällige Lösung
best <- S
REPEAT
    wähle R aus der Nachbarschaft von S
    IF Quality(R) > Quality(S) oder zufälliges
                Z ∈ [0,1] < e^((Quality(R)-Quality(S))/t) THEN
        S <- R
    dekrementiere t
    IF Quality(S) > Quality(best) THEN
        best <- S
UNTIL best ist die ideale Lösung oder Temperatur ≤ 0
return best
```

  - *Tabu-Search*
    - *Idee und Ablauf*
      - Speichert alle bisherigen Lösungen und Liste und nimmt diese nicht nochmal
      - Kann sich jedoch von der optimalen Lösung entfernen
      - Tabu List hat maximale Größe, falls voll, werden älteste Lösungen gelöscht
    - *Code*

```
TABU-SEARCH
l <- maximale Größe der Tabu List
n <- Anzahl der zu betrachtenden Nachbarschaftslösungen
S <- irgendeine initiale zufällige Lösung
best <- S
L <- { } Tabu List der Länge l
Füge S in L ein
REPEAT
    IF Length(L) > l THEN
        Entferne ältestes Element aus L
    wähle R aus Nachbarschaft von S
    FOR n - 1 mal DO
        Wähle W aus Nachbarschaft von S
        IF W ∉ L und (Quality(W) > Quality(R)) oder R ∈ L) THEN
            R <- W
    IF R ∉ L THEN
        S <- R
        Füge R in L ein
    IF Quality(S) > Quality(best) THEN
        best <- S
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

- **Populationsbasierte Methode**
  - Bisher: Immer nur Betrachtung einer einzigen Lösung
  - Hier: Betrachtung einer Stichprobe von möglichen Lösungen
  - Bei der Bewertung der Qualität spielt die Stichprobe die Hauptrolle
  - z.B. Evolutionärer Algorithmus
- **Evolutionärer Algorithmus**
  - *Idee und Ablauf*
    - Algorithmus aus der Klasse der Evolutionary Computation
    - *generational Algorithmus:* Aktualisierung der gesamten Stichprobe pro Iteration
    - *steady-state Algorithmus:* Aktualisierung einzelner Kandidaten der Probe pro Iteration
    - *Resampling-Technik:* Generierung neuer Strichproben basierend auf vorherigen Resultaten
  - *Abstrakter Code (Allgemeiner Breed und Join)*

```
ABSTRACT-EVOLUTIONARY-ALGORITHM
P <- generiere initiale Population
best <- ⊡ // leere Menge
REPEAT
    AssesFitness(P)
    FOR jedes individuelle P_i ∈ P DO
        IF best = ⊡ oder Fitness(P_i) > Fitness(best) THEN
            best <- P_i
    P <- Join(P, Breed(P))
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

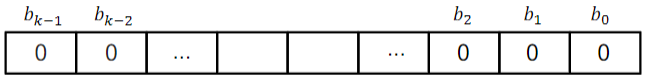
    - *Breed:* Erstellung neuer Stichprobe mithilfe Fitnessinformation
    - *Join:* Fügt neue Population der Menge hinzu
  - *Initialisierung der Population*
    - Initialisierung durch zufälliges Wählen der Elemente
    - Beeinflussung der Zufälligkeit bei Vorteilen möglich
    - Diversität der Population (alle Elemente in Population einzigartig)
    - Falls neue zufällige Wahl eines Individuums
      - Entweder Vergleich mit allen bisherigen Individuen ($O(n^2)$)
      - Oder besser: Nutzen eines Hashtables zur Überprüfung auf Einzigartigkeit ($O(n)$)
  - *Evolutionsstrategien - Ideen*
    - Generiere Population zufällig
    - Beurteile Qualität jedes Individuums
    - Lösche alle bis auf die μ besten Individuen
    - Generie $\frac{\lambda}{\mu}$-viele Nachfahren pro bestes Individuum
    - Join Funktion: Die Nachfahren ersetzen die Individuen

- *Algorithmus der Evolutionsstrategie*

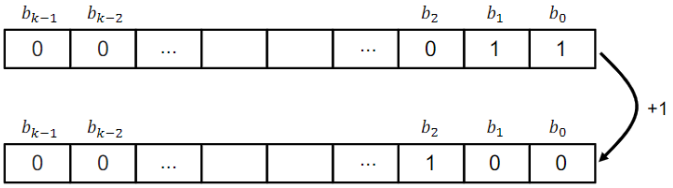
```
(μ, λ)-EVOLUTION-STRATEGY

μ <- Anzahl der Eltern (initiale Lösung)
λ <- Anzahl der Kinder
P <- {}
FOR λ-oft DO
    P <- {neues zufälliges Individuum}
best <- ⊡
REPEAT
    FOR jedes individuelle P_i ∈ P DO
        AssesFitness(P_i)
        IF best = ⊡ oder Fitness(P_i) > Fitness(best) THEN
            best <- P_i
    Q <- die μ Individueen deren Fitness() am Größten ist
    P <- {}
    FOR jedes Element Q_j ∈ Q DO
        FOR λ/μ-oft DO
            P <- P ∪ {MUTATE(Q_j)}
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

7.5 Amortisierte Analyse

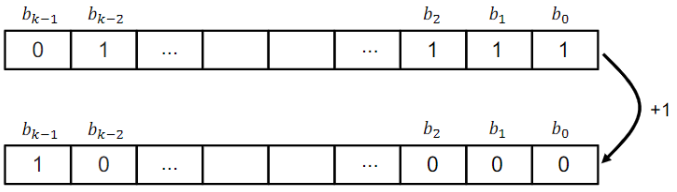
- **Kosten von Operationen**
  - Bisher: Betrachtung von Algorithmen, die Folge von Operationen auf Datenstrukturen ausführen
  - Abschätzung der Kosten von n Operationen im Worst-Case
  - Dies liefert die obere Schranke für die Gesamtkosten der Operationenfolge
  - Nun: **Amortisierte Analyse**: Genauere Abschätzung des Worst Case
  - Voraussetzung: Nicht alle Operationen in der Operationenfolge gleich teuer
  - z.B. eventuell abhängig vom aktuellen Zustand der Datenstruktur
  - Amortisierte Analyse garantiert die mittlere Performanz jeder Operation im Worst-Case
- **Beispiel Binärzähler**
  - *Eigenschaften*
    - k-Bit Binärzähler hier als Array
    - Codierung der Zahl als $x = \sum_{i=0}^{k-1} 2^i b_i$
    - Initialer Array für $x = 0$:



  - *Inkrementieren eines Binärzählers*
    - Erhöhe x um 1
    - Beispiel: $x = 3$
    - `INCREMENT` kostet 3, da sich drei Bitpositionen ändern

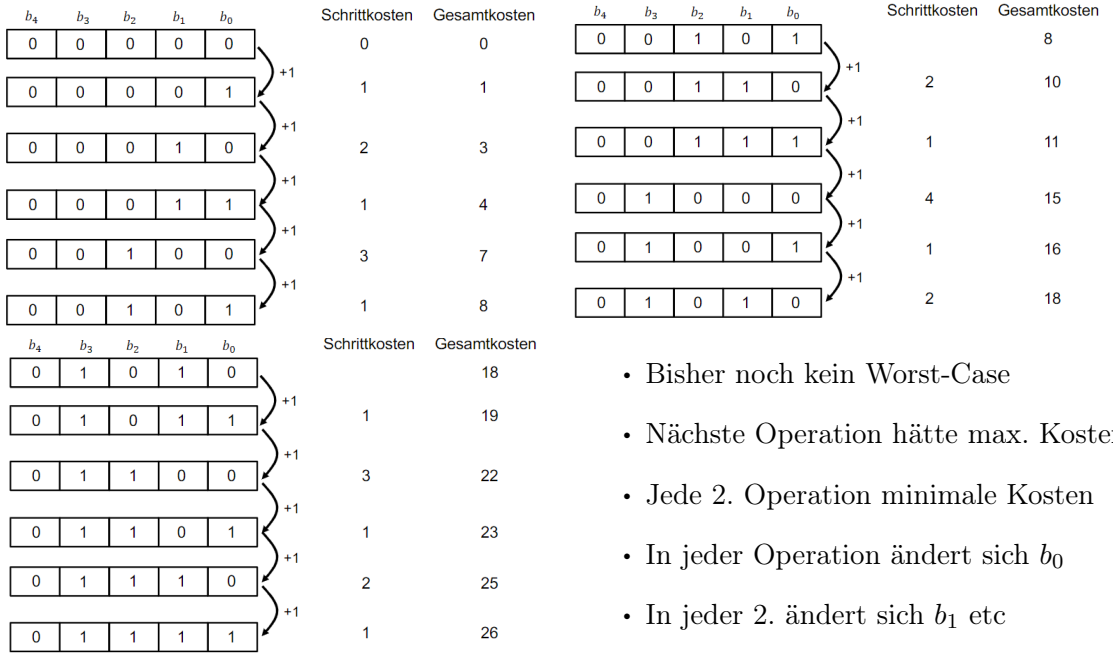


  - *Teuerste INCREMENT-Operation*
    - `INCREMENT` flippt $k - 1$ Bits von 1 zu 0 und 1 Bit von 0 auf 1
    - Kosten nicht konstant, stark abhängig von Datenstruktur



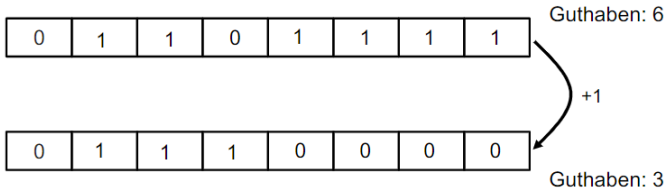
  - *Traditionelle Worst-Case Analyse*
    - Worst-Case Kosten von n `INCREMENT`-Operationen auf k-Bit Binärzähler
    - Anfangswert $x = 0$
    - Schlimmster Kostenfall: $INCREMENT$-Operation hat k Bitflips
    - n-mal inkrementieren sorgt für Kosten: $T(n) \leq n \cdot k \in O(kn)$

- **Aggregat Methode - Beispiel Binärzähler**
  - *Eigenschaften*
    - Methode für Amortisierte Analyse
    - Sequenz von n-Operationen kostet Zeit $T(n)$
    - Durchschnittliche Kosten pro Operation $\frac{T(n)}{n}$
    - Ziel: $T(n)$ genau berechnen, **ohne** jedes Mal Worst-Case anzunehmen
    - Ansatz: Aufsummation der **tatsächlich** anfallenden Kosten aller Operationen
  - *Durchführung*



    - Bisher noch kein Worst-Case
    - Nächste Operation hätte max. Kosten
    - Jede 2. Operation minimale Kosten
    - In jeder Operation ändert sich b_0
    - In jeder 2. ändert sich b_1 etc
  - *Genauere Kostenanalyse*
    - Nun in der Lage $T(n)$ genau auszurechnen
    - Bei n Operationen ändert sich das Bit b_i genau $\lfloor \frac{n}{2^i} \rfloor$-mal
    - Bits b_i mit $i > log_2\ n$ ändern sich nie
    - Über alle k Bits aufsummieren liefert:
      $T(n) = \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor = n \sum_{i=0}^{k-1} \frac{1}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2n \in O(n)$
    - Obere Schranke: $T(n) \leq 2n$
    - Kosten jeder `INCREMENT`-Operation im Durchschnitt: $\frac{2n}{n} = 2 \in O(1)$

- **Account Methode - Beispiel Binärzähler**
  - *Eigenschaften*
    - Besteuerung einer Operationen, so dass sie Kosten anderer Operationen mittragen
    - Zuweisung von höherer Kosten (Amortisierte Kosten), als ihre tatsächlichen Kosten sind
    - **Guthaben**: Differenz zwischen amortisierten und tatsächlichen Kosten
    - Nutzung dieses Guthabens für Operationen bei denen amortisiert $<$ tatsächlich
    - Guthaben darf nicht negativ werden:
      Summe amortisierte Kosten $>$ Summe tatsächliche Kosten
  - *Wahl der Amortisierten Kosten - Binärzähler*
    - Setzen eines Bits von $0 \rightarrow 1$ zahlt 2 Einheiten ein / Bezeichnung f_i
    - Setzen eines Bits von $1 \rightarrow 0$ zahlt 0 Einheiten ein / Bezeichnung e_i
    - Tatsächliche Kosten t_i: Anzahl der Bitflips bei der $i - ten$ `INCREMENT`-Operation
      $t_i = e_i + f_i$
    - Amortisierte Kosten betragen: $a_i = 0 \cdot e_i + 2 \cdot f_i$
  - *Kostenbeispiel*
    - Jede Bitflip Operation kostet zusätzlich 1 Einheit
    - Setzen Bit $0 \rightarrow 1$: Zahlt 2 ein, kostet aber 1 $\rightarrow$ +1 Guthaben
    - Setzen Bit $1 \rightarrow 0$: Zahlt 0 ein, kostet aber 1 $\rightarrow$ -1 Guthaben



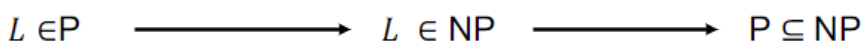
  - *Obere Schranken der Kosten*
    - Guthaben auf dem Konto entspricht der Anzahl der auf 1 gesetzten Bits
    - Kosten: $T(n) \sum_{i=1}^{n} t_i \leq v \sum_{i=1}^{n} a_i$
    - Nun Abschätzung dieser Formel zum Erhalten einer oberen Schranke
    - Beobachtung: Bei jeder `INCREMENT` höchstens ein neues Bit von 0 auf 1
    - Für alle i gilt damit $f_i \leq 1$
    - Amortisierte Kosten jeder Operation höchstens $2 \cdot f_i \leq 2$
    - Insgesamt: $T(n) = \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \leq 2n \in O(n)$

- **Potential-Methode - Beispiel Binärzähler**
  - *Eigenschaften*
    - Betrachtung welchen Einfluss die Operationen auf die Datenstruktur haben
    - Potentialfunktion $\phi(i)$: Hängt vom aktuellen Zustand der Datenstruktur nach i-ter Operation ab
    - Ausgangspotential sollte vor jeglicher Operation nicht negativ sein $\phi(0) \geq 0$
  - *Amortisierte Kosten*
    - Amortisierte Kosten der i-ten Operation: (Summe tatsächliche Kosten + Potentialänderung)
      $a_i = t_i + \phi(i) - \phi(i-1)$
    - Summe der amortisierten Kosten:
      $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n}(t_i + \phi(i) - \phi(i-1)) = \sum_{i=1}^{n} t_i + \phi(n) - \phi(0)$
    - Wenn für jedes i gilt $\phi(i) \geq \phi(0)$:
      Summe der amor. Kosten ist gültige obere Schranke an Summe der tatsächlichen Kosten
  - *Potential-Methode anhand des Binärzählers*
    - $\phi(i)$: Anzahl der 1-en im Array nach i-ter `INCREMENT`-Operation
      $\rightarrow \phi(i)$ nie negativ und $\phi(0) = 0$
    - Angenommen i-te Operation setzt e_i Bits von 1 auf 0, dann hat diese Operation Kosten $t_i \leq e_i + 1$
    - Neues Potential: $\phi(i) \leq \phi(i-1) - e_i + 1 \Leftrightarrow \phi(i) - \phi(i-1) \leq e_i$
    - Amortisierte Kosten der i-ten `INCREMENT`-Operation:
      $a_i = t_i + \phi(i) - \phi(i-1) \leq e_i + 1 + 1 - e_i = 2$
    - Insgesamt: $T(n) = \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \leq 2n \in O(n)$

8 NP

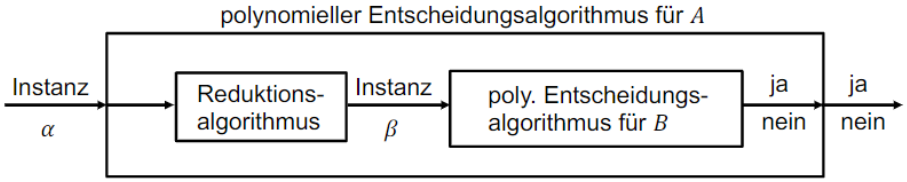
- **Berechnungsprobleme**
  - Sind alle Probleme in polynomieller Zeit lösbar? ($O(n^k)$)
  - Nein ⇒ Manche nur in superpolynomieller Zeit lösbar
  - Polynomielle Probleme: *"einfach"*
  - Superpolynomielle Probleme: *"hart"*
- **Klasse P**
  - Klasse aller Polynomialzeitprobleme
  - Problem ist *effizient lösbar* gdw. es in polynomieller Zeit lösbar ist
  - Gilt für Polynome beliebigen Grades (auch n^k)
  - Zeitkomplexität n^k mit großem k bedenklich, jedoch fast nie notwendig
  - n beschreibt die Länge der Eingabe
  - Beispiele: Binäre Addition, Kürzeste Wege, Sortieren,...
- **Klasse NP**
  - Enthält *"einfach zu verifizierende"* Probleme (polynomieller Zeit)
  - Enthält Probleme mit *"kurzem Beweis"* (Länge polynomiell in Länge der Instanz)
  - Also: Klasse aller Probleme, deren Lösung in Polynomialzeit verifizierbar ist
  - Beispiele: Soduko, 3D-Matching,...
  - Beispiel: *Faktorisierungsproblem*
    - Jede nicht Primzahl kann eindeutig als Primzahlprodukt geschrieben werden
    - $35 = 5 \cdot 7$, $117 = 3 \cdot 3 \cdot 13$,...
    - Faktorsieren auf klassischen Computern schwer
    - $n \longrightarrow^{schwer} p, q$
    - $n, p, q \longrightarrow^{leicht}$ ist $n = p \cdot q$?
  - *Rucksackproblem* auch in polynomieller Laufzeit verifizierbar
  - *Hamilton-Kreis-Problem*
    - Hamiltonischer Kreis: Zyklus, der alle Knoten, aber nicht unbedingt alle Kanten enthält
    - Entscheidungsalgorithmus listet alle möglichen Permutationen der Knoten aus G auf
    - Prüfung bei jeder Permutation, ob es ein Hamiltonischer Kreis ist
    - Laufzeit:
      - Kodierung via Adjazenzmatrix: m Knoten ⇒ Matrix mit $n = m \ x \ m$ Einträgen
      - $m!$ mögliche Permutationen der Knoten
      - $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$
      - ⇒ superpolynomielle Laufzeit (liegt **nie** in $O(n^k)$)
    - **Allerdings:** Einfacher, wenn nur Beweis verifiziert werden muss
      ⇒ Test, ob es sich um Permutation der Knoten handelt
      ⇒ Test, ob alle angegebenen Kanten auf Kreis im Graphen existieren
      ⇒ Verifikationsalgorithmus V mit quadratischer Laufzeit
    - Verifikationsalgorithmus: $V(x, y) = 1/0$ (1, falls Kreis/ 0, falls nicht)
    - Damit: Hamilton-Kreis ∈ NP

- **Entscheidungsproblem vs Optimierungsproblem**
  - Optimierungsproblem: Lösung nimmt bestimmten Wert an
  - Entscheidungsproblem: Binäre Antwort (Ja/Nein)
  - Bei NP Betrachtung von Entscheidungsproblemen
  - Optimierungsproblem oft in verwandtes Entscheidungsproblem umwandelbar
  - Verwandtes Entscheidungsproblem: dem zu optimierenden Wert wird eine Schranke auferlegt
- **P versus NP**



  - Für viele wichtige Probleme ist jedoch unbekannt, ob sie in P (effizient) lösbar sind
  - Unbekannt ob $P \neq NP$
  - Intuitive Frage: Ist das Finden eines Beweises schwieriger als dessen Überprüfung?
    $\Rightarrow$ Ja, also $P \neq NP$ gilt
  - In den letzten 50 Jahren kein Beweis für $P = NP$
  - Eines der wichtigsten offenen Probleme der theoretischen Informatik
  - Konsequenzen eines Beweises von $P = NP$:
    - $P = NP$: **dramatisch**, vieles bisher schwieriges lösbar (Rucksack, Kryptographie)
    - $P \neq NP$: **nicht dramatisch**, mgl. interessante Konsequenzen in Kryptographie
- **NP-Vollständigkeit**
  - Problem befindet sich in NP
  - Problem ist so "schwer" wie jedes Problem in NP
  - Beweis: Zeigen, dass kein effizienter Algorithmus existiert
  - Werkzeug: Reduktionen (zum Vergleich verschiedener Probleme)
  - *NP-Härte/NP-Schwere:*
    - Klassifikation von Problemen als schwierig, trotz fehlender genauer Zuordnung
    - *Starke Indikatoren*, dass Problem L nicht in P ist:
      - L ist mindestens so schwierig, wie alle anderen Probleme in NP
      - Daraus folgt, dass L nur in P, wenn $P = NP$ (unwahrscheinlich)
  - *Definitionen*
    - Problem L ist **NP-schwer**, wenn $L' \leq_p L$ für alle $L' \in NP$
    - Problem L ist **NP-vollständig**, wenn L sowohl NP-schwer als auch in NP ist
    - z.B.: Hamilton-Kreis ist NP-vollständig

- **Reduktionen**
  - *Reduktionsidee*
    - Betrachte Problem A, das wir in polynomieller Zeit lösen wollen
    - Bereits bekannt: Problem B (in polynomieller Zeit lösbar)
    - Benötigt wird Prozedur, die Instanzen der Probleme ineinander überführt
      $\Rightarrow$ Transformation benötigt polynomielle Zeit
      $\Rightarrow$ Antworten sind gleich

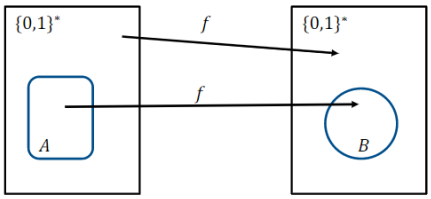


  - *Beispiel:*
    - Intuitiv: Reduktion von A auf B, wenn Umformulierung möglich
      $\Rightarrow$ Jede Instanz A kann leicht in Instanz von B umformuliert werden
      $\Rightarrow$ Lösung der Instanz B liefert Lösung von Instanz A
    - Reduktion: Lösen von linearen Gleichungen auf quadratische Gleichnungen
      - Lineare Gleichung $ax + b = 0 \Rightarrow x = \frac{-b}{a}$
      - Quadratische Gleichung $ax^2 + bx + 0 = 0 \Rightarrow x = \frac{-b}{a}, x = 0$
      - Quadratische Gleichung liefert also auch Lösung für lineare Gleichung
  - *Formale Definition:*

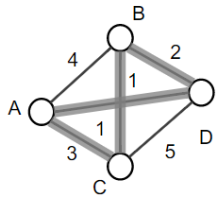
    A lässt sich auf B in **polynomieller Zeit reduzieren**, mit Schreibweise $A \leq_p B$, wenn eine in polynomieller Zeit berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ existiert, sodass für alle $x \in \{0,1\}^*$ gilt:

    $$x \in A \text{ genau dann, wenn } f(x) \in B$$

    Illustration der Polynomialzeitreduktion:

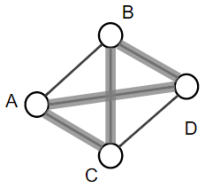


- **Travelling-Salesman Problem**
  - *Beschreibung*
    - Reisender plant Rundreise durch mehrere Städte
    - Start und Ziel ist eine vorgegebene Stadt
    - Jede Stadt nur einmal besucjhen
    - Ziel: Minimale Reiselkosten



Eine optimale Route mit Kosten 7 verläuft von A → D → B → C → A

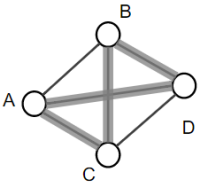
  - *Problem:*
    - Anzahl der Rundreisen $(n-1)!$
    - Stark nach oben explodierende Zahlen
    - Brute-Force für große n praktisch unmöglich
    - Es existiert kein effizienter Algorithmus, der das TSP effizient löst
    - TSP ist **NP-vollständig**
  - *Beweis NP-Vollständigkeit*
    - Zeigen: TSP gehört zu NP und TSP ist NP-schwer
  - *TSP gehört zu NP*
    - Gegeben: Instanz des Problems TSP, Folge der n Knoten der Tour (Zertifikat)
    - Verifikationsalgorithmus überprüft, ob Folge jeden Knoten genau einmal enthält
    - Außerdem Aufsummieren der Kantenkosten und überprüfen, ob diese maximal k ist
    - Verifikation läuft in polynomieller Laufzeit $\Rightarrow$ gehört zu NP
  - *TSP ist NP-schwer*
    - Wir zeigen $HAM-KREIS \leq_p TSP$
    - Start: Instanz von $HAM-KREIS$ mit $G=(V,E)$
    - Konstruiere Instanz von TSP
      $\Rightarrow G'=(V,E')$ mit $E'=\{(i,j): i,j \in V \text{ und } i \neq j\}$
    - Definiere Kostenfunktion $c(i,j)=0$,falls $(i,j) \in E$ / $c(i,j)=1$, falls $(i,j) \notin E$
    - Instanz von TSP ist $< G', c, 0 >$ (Konstruktion in polynomieller Zeit) (0: Kosten von 0)
    - **Zeige jetzt:** G besitzt hamiltonischen Kreis $\Leftrightarrow$ G' enthält Tour mit Kosten ≤ 0
    - $\Rightarrow$ Graph G besitzt einen hamiltonischen Kreis h



Jede Kante von h gehört zu E und daher besitzt laut Kostenfunktion der Graph G' die Kosten 0.

Damit ist h eine Tour in G' mit den Kosten 0.

    - $\Leftarrow$ Graph G besitzt eine Tour h' mit Kosten kleiner gleich 0



Die Kosten der Kanten in E' haben die Werte 0 und 1. Die Kosten der Tour betragen exakt 0 und jede Kante muss die Kosten 0 haben.

Damit hat h' nur Kanten von E.

Damit folgt, dass h' ein Hamiltonischer Kreis des Graphen G ist.