

Algorithmen und Datenstrukturen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Zusammenfassung

Author: Ruben Deisenroth

Helper: J. Milkovits

Semester: SoSe 2020

v2.0.1 (SNAPSHOT)

Stand: 8. April 2021

Fachbereich: Informatik

Inhaltsverzeichnis

1 Einführung	3
1.1 Probleme in der Informatik	3
1.2 Definitionen für Algorithmen	3
1.3 Definitionen für Datenstrukturen	4
1.4 Pseudocode-Konventionen	4
1.5 Weitere wichtige Definitionen	4
2 Sortieren	5
2.1 Einführung - Das Sortierproblem	5
2.2 Arrays	5
2.3 Exkurs: Totale Ordnung	6
2.4 Vergleichskriterien von Suchalgorithmen	6
2.5 Analyse von Algorithmen (I)	7
2.6 Analyse von Algorithmen (II)	8
2.7 Analyse von Algorithmen (III)	9
2.8 Insertion Sort (Sortieren durch Einfügen)	12
2.9 Bubble Sort	14
2.10 Selection Sort	14
2.11 Divide-And-Conquer Prinzip	15
2.12 Merge Sort	16
2.13 Quicksort	18
2.14 Laufzeitanalyse von rekursiven Algorithmen	20
2.14.1 Analyse von Divide-And-Conquer Algorithmen	20
2.14.2 Substitutionsmethode	20
2.14.3 Rekursionsbaum	21
2.14.4 Mastertheorem	23
3 Grundlegende Datenstrukturen	25
3.1 Stacks	25
3.2 Verkettete Listen	28
3.2.1 Elementare Operationen auf Listen	29
3.3 Queues	30
3.4 Binäre Bäume	33
3.4.1 Abstrakter Datentyp Baum	35
3.5 Binäre Suchbäume	38
4 Advanced Data Structures	43
4.1 Rot-Schwarz-Bäume	43
4.2 AVL-Bäume	49
4.3 Splay-Bäume	51

4.4	Binäre Max-Heaps	53
4.5	B-Bäume	55
5	Randomized Data Structures	58
5.1	Skip Lists	58
5.2	Hashtables	60
5.3	Bloom-Filter	61
6	Graph Algorithms	63
6.1	Graphen	63
6.2	Breadth-First Search (BFS)	65
6.3	Depth-First Search(DFS)	67
6.4	Minimale Spannbäume	69
6.5	Kürzeste Wege in (gerichteten) Graphen	70
6.6	Maximaler Fluss in Graphen	72
7	Advanced Designs	75
7.1	Dynamische Programmierung	75
7.1.1	Stabzerlegungsproblem	75
7.2	Greedy-Algorithmus	79
7.2.1	Aktivitäten-Auswahl-Problem	79
7.3	Backtracking	81
7.4	Metaheuristiken	83
7.5	Amortisierte Analyse	91
8	NP	95
	Stichwortverzeichnis	102

1 Einführung

1.1 Probleme in der Informatik

Ein **Problem** im Sinne der Informatik:

- Enthält Beschreibung der Eingabe
- Enthält Beschreibung der Ausgabe
- Gibt selbst **keinen** Übergang von Ein und Ausgabe an



Abbildung 1: Modell Problem Informatik

z.B. Finde den kürzesten Weg zwischen 2 Orten

Eine **Probleminstanz** ist eine konkrete Eingabebelegung für die entsprechende Ausgabe gewünscht.

Für das obige Problem wäre das z.B. „Was ist der kürzeste Weg vom Audimax in die Mensa?“

1.2 Definitionen für Algorithmen

Definition – Algorithmus „Ein Algorithmus ist eine **endliche Folge** von Rechenschritten, die eine **Eingabe** in eine **Ausgabe** umwandelt.“^a

^aQuelle: Cormen et al., 4. Auflage

Anforderungen an Algorithmen:

Spezifizierung der Ein- und Ausgabe • Anzahl und Typen aller Elemente ist/sind definiert

Eindeutigkeit • Jeder Einzelschritt ist klar definiert und ausführbar
• Die Reihenfolge der Einzelschritte ist festgelegt.

Endlichkeit • Notation hat endliche Länge

Eigenschaften von Algorithmen:

Determiniertheit Für gleiche Eingabe folgt stets die gleiche Ausgabe (andere Zwischenzustände sind möglich)

Determinismus Für die gleiche Eingabe ist die Ausführung und Ausgabe stets identisch.

Terminierung Der Algorithmus läuft für jede endliche Eingabe nur endlich lange

Korrektheit Der Algorithmus berechnet stets die spezifizierte Ausgabe (falls dieser terminiert).

Effizienz Sparsamkeit im Ressourcenverbrauch (Zeit, Speicher, Energie, ...)

1.3 Definitionen für Datenstrukturen

Definition – Datenstruktur „Eine Datenstruktur ist eine Methode, Daten **abzuspeichern** und zu **organisieren** sowie den **Zugriff** auf die Daten und die **Modifikation** der Daten zu erleichtern.“^a

^aQuelle: Cormen et al., 4. Auflage

Datenstrukturen:

- Sind Organisationsformen für Daten
 - Beinhalten Strukturbestandteile und Nutzerdaten (Payload)
- z.B. Arrays, Listen, ...

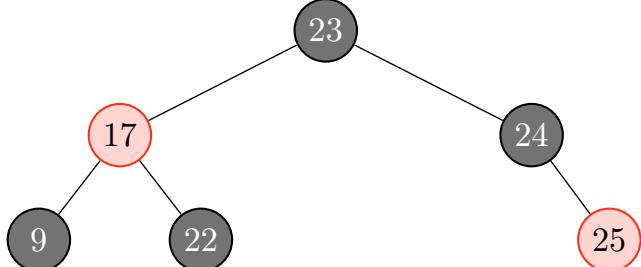


Abbildung 2: Beispiel Datenstruktur (Rot-Schwarz-Baum)

1.4 Pseudocode-Konventionen

- Blöcke werden durch Einrückungen hervorgehoben
- Blockkonstrukte sind „**for** x to y“, „**while**“, „**repeat-until**“ und „**if-else**“
- Kommentare erhalten das Prädikat „//“
- „**i = j = e**“ bedeutet, i und j erhalten den Wert von e
- Variablen sind immer lokal
- $A[i]$ bezeichnet das i-te Element im Array A
- $A[i..j]$ Array A im Bereich von $i - j$
- Attribute werden über einen „.“ abgerufen

1.5 Weitere wichtige Definitionen

Definition – short circuit evaluation (Kurzschlussauswertung) Strategie, bei der die Auswertung, nachdem die Gesamtlösung durch einen Teilausdruck eindeutig bestimmt wurde, abgebrochen wird.
z.B.: $1+1==2 \quad || \quad 1/0==0 \rightarrow \text{true}$

Definition – call-by-reference (Referenzparameter) Übergeben von Referenz auf ein Objekt. Dadurch sind Änderungen an diesem innerhalb der Routine möglich.

Definition – call-by-value (Wertparameter) indexcall-by-value Übergeben einer Kopie des Objekts. Das ursprüngliche Objekt kann so nicht mehr verändert werden, jedoch die Kopie. Die referenzen bleiben beim Kopieren gleich (z.B. bei Linked List)

2 Sortieren

2.1 Einführung - Das Sortierproblem

Definition – Das Sortierproblem

Ausgangspunkt:

D_1	D_2	\dots	D_n
-------	-------	---------	-------

Abbildung 3: Folge von Datensätzen D_1, D_2, \dots, D_n

Ziel: Datensätze so anzuordnen, dass die Schlüsselwerte sukzessive ansteigen (oder absteigen)

Bedingung: Schlüssel(werte) müssen vergleichbar sein

- zu sortierende Elemente heißen auch Schlüssel(werte)

Durchführung

- **Eingabe:** Sequenz von Schlüsselwerten $\{a_1, a_2, \dots, a_n\}$
- Eingabe ist eine **Instanz** des Sortierproblems
- **Ausgabe:** Permutation $\{a'_1, a'_2, \dots, a'_n\}$ derselben Folge mit Eigenschaft $a'_1 \leq \dots \leq a'_n$
- Algorithmus ist **korrekt**, wenn dieser das Problem für alle Instanzen löst

2.2 Arrays

Definition – Array

Reihung (Feld) fester Länge von Daten des gleichen Typs

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

Abbildung 4: beispielhafte Darstellung eines Arrays

A Bezeichnung des Arrays mit dem Namen „A“

$A[i]$ Zugriff auf das $(i + 1)$ -te Element des Arrays

Beispiel:

$$A[2] = 17$$

⇒ Arrays erlauben effizienten Zugriff auf Elemente: konstanter Aufwand

2.3 Exkurs: Totale Ordnung

Definition – Totale Ordnung Sei M eine nicht leere Menge und $\leq \subseteq M \times M$ eine binäre Relation auf M . Das Paar (M, \leq) heißt genau dann eine totale Relation auf der Menge M , wenn folgende Eigenschaften erfüllt sind:

- 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: $\forall x, y \in M : x \leq y \vee y \leq x$

Beispiele

- \leq Ordnung auf natürlichen Zahlen
- Lexikographische Ordnung \leq_{lex} ist eine totale Ordnung

2.4 Vergleichskriterien von Suchalgorithmen

Berechnungsaufwand	<ul style="list-style-type: none">• $\mathcal{O}(n)$
Effizienz	<ul style="list-style-type: none">• Best Case vs. Average Case vs Worst Case
Speicherbedarf	<ul style="list-style-type: none">• in-Place (in situ): zusätzlicher Speicher von der Eingabegröße unabhängig• out-of-place: Speichermehrbedarf von Eingabegröße abhängig
Stabilität	<ul style="list-style-type: none">• stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht
Anwendung als Auswahlfaktor	<ul style="list-style-type: none">• Hauptoperationen beim Sortieren: Vergleichen und Vertauschen• Anwendung spielt eine enorme Rolle:<ul style="list-style-type: none">– Verfahren mit vielen Vertauschungen und wenig Vergleichen, wenn Vergleichen teuer– Verfahren mit wenig Vertauschungen und vielen Vergleichen, wenn Umsortieren teuer

2.5 Analyse von Algorithmen (I)

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 | <ul style="list-style-type: none">• Rechenzeit (Anzahl der Einzelschritte)• Kommunikationsaufwand• Speicherplatzbedarf• Zugriffe auf Speicher |
|--------------------------|--|

- | | |
|-------------------------|---|
| Laufzeitfaktoren | <ul style="list-style-type: none">• Länge der Eingabe• Implementierung der Basisoperationen• Takt der CPU |
|-------------------------|---|

2.6 Analyse von Algorithmen (II)

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 / Initialisierungsaufwände
- 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 | <ul style="list-style-type: none">• Laufzeit abhängig von konkreter Implementierung und System• Theoretische Betrachtung ist frei von Abhängigkeiten und Seiteneffekten• Theoretische Betrachtung lässt direkte Vergleichbarkeit zu |
|-------------------------|---|

- | | |
|----------------|--|
| Aufwand | <ul style="list-style-type: none">• Wieviele Testreihen?• In welcher Umgebung?• Messen führt in der Ausführung zu hohem, praktischen Aufwand |
|----------------|--|

- | | |
|-----------------------------|--|
| Komplexitätsfunktion | <ul style="list-style-type: none">• Wachstumsverhalten ausreichend• Praktische Evaluation mit Zeiten nur für Auswahl von Systemen möglich• Theoretischer Vergleich (Funktionsklassen) hat ähnlichen Erkenntnisgewinn |
|-----------------------------|--|

2.7 Analyse von Algorithmen (III)

Übersicht

$$\Theta =$$

$$\mathcal{O} \leq$$

$$\Omega \geq$$

Θ -Notation

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

$$\Theta(g) = \{f : \exists c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}$$

↑ ↑
 Positive Konstanten Für alle n größer gleich n_0
 Funktion f

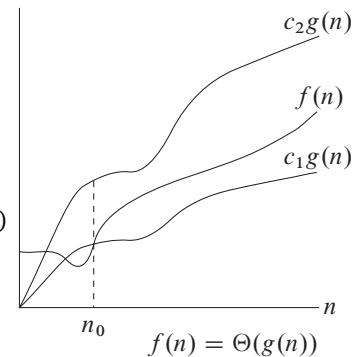


Abbildung 5: Veranschaulichung Θ -Notation

- $\Theta(g)$ 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 $f(n)$
- $f(n) = \Theta(g(n))$ gilt, wenn $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$ erfüllt sind
- z.B.: $f(n) = \frac{1}{2}n^2 - 3n$ | $f(n) \in \Theta(n^2)$?
- Aus $\Theta(n^2)$ folgt, dass $g(n) = n^2$

- **Vorgehen:**

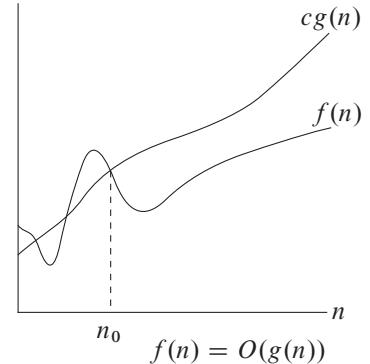
- Finden eines n_0 und c_1, c_2 , sodass
- $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ erfüllt ist
- Konkret: $c_1 * n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 * n^2$
- Division durch n^2 : $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$
- Ab $n = 7$ positives Ergebnis: $0,0714 \mid n_0 = 7$
- Deswegen setzen wir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

\mathcal{O} -Notation

- \mathcal{O} -Notation beschränkt eine Funktion asymptotisch von oben

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

↑ [] ↑ []
 Funktion f Positive Konstanten Für alle n größer gleich n_0 $f(n)$ wird von $c g(n)$ für hinreichend große n beschränkt



$f(n) = \mathcal{O}(g(n))$

- $\mathcal{O}(g)$ enthält alle f , die höchstens so schnell wie g wachsen
- **Schreibweise:** $f = \mathcal{O}(g)$
- $f(n) = \Theta(g) \rightarrow f(n) = \mathcal{O}(g) \mid \Theta(g(n)) \subseteq \mathcal{O}(g(n))$
- Ist f in der Menge $\Theta(g)$, dann auch in der Menge $\mathcal{O}(g)$
- z.B.: $f(n) = n + 2 \mid f(n) = \mathcal{O}(n)?$
- Ja $f(n)$ ist Teil von $\mathcal{O}(n)$ für z.B. $c = 2$ und $n_0 = 2$

Abbildung 6: Veranschaulichung \mathcal{O} -Notation

\mathcal{O} -Notation Rechenregeln

- | | |
|------------|---|
| Konstanten | <ul style="list-style-type: none"> • $f(n) = a$ mit $a \in \mathbb{R}$ konstante Funktion $\rightarrow f(n) = \mathcal{O}(1)$ • z.B. $3 \in \mathcal{O}(1)$ |
|------------|---|

- | | |
|------------------------|--|
| Skalare Multiplikation | <ul style="list-style-type: none"> • $f = \mathcal{O}(g)$ und $a \in \mathbb{R} \rightarrow a * f = \mathcal{O}(g)$ |
|------------------------|--|

- | | |
|----------|---|
| Addition | <ul style="list-style-type: none"> • $f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2) \rightarrow f_1 + f_2 = \mathcal{O}(\max\{g_1, g_2\})$ |
|----------|---|

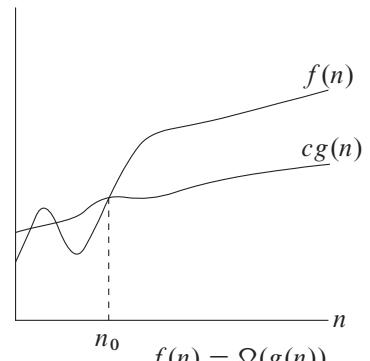
- | | |
|----------------|--|
| Multiplikation | <ul style="list-style-type: none"> • $f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2) \rightarrow f_1 * f_2 = \mathcal{O}(g_1 * g_2)$ |
|----------------|--|

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 $c g(n)$ für hinreichend große n unten beschränkt



$f(n) = \Omega(g(n))$

- Ω -Notation enthält alle f , die mindestens so schnell wie g wachsen
- **Schreibweise:** $f = \Omega(g)$

Abbildung 7: Veranschaulichung

Komplexitätsklassen

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

Tabelle 1: Komplexitätsklassen ($n =$ Länge der Eingabe)

Eingabegröße n	$\log_{10} n$	n	n^2	n^3	2^n
10	$1\mu\text{s}$	$10\mu\text{s}$	$100\mu\text{s}$	1ms	$\sim 1 \text{ ms}$
100	$2\mu\text{s}$	$100\mu\text{s}$	10ms	1s	$\sim 4 \times 10^{16} \text{ y}$
1000	$3\mu\text{s}$	1ms	1s	16min 40s	?
10000	$4\mu\text{s}$	10ms	1min 40s	$\sim 11,5\text{d}$?
100000	$5\mu\text{s}$	100ms	2h 64min 40s	$\sim 31,7\text{y}$?

Tabelle 2: Komplexitätsklassen-Ausführungszeit, falls eine Operation n genau $1\mu\text{s}$ dauert

Es gilt: $\log(n) < \sqrt{n} < n < n \cdot \log(n) < n^2 < n! < 2^n < n^n$

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 \mathcal{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 $\frac{n^2}{2} \neq \omega(n^2)$

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

2.8 Insertion Sort (Sortieren durch Einfügen)

Idee – Insertion-Sort

- 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

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] ein
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
```

Schleifeninvariante von Insertion Sort

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

Korrektheit von Insertion Sort

Initialisierung Beginn mit $j=1$, also Teilstück $A[0 \dots j-1]$ besteht nur aus einem Element $A[0]$. Dies ist auch das ursprüngliche Element und Teilstück ist sortiert.

Fortsetzung Zu zeigen ist, dass die Invariante bei jeder Iteration erhalten bleibt. Ausführungsblock der **for**-Schleife sorgt dafür, dass $A[j-1]$, $A[j-2], \dots$ je um Stelle nach rechts geschoben werden bis $A[j]$ korrekt eingefügt wurde. Teilstück $A[0 \dots j]$ besteht aus ursprünglichen Elementen und ist sortiert. Inkrementieren von j erhält die Invariante.

Terminierung Abbruchbedingung der **for**-Schleife, wenn $j > A.length - 1$. Jede Iteration erhöht j . Dann bei Abbruch ist $j = n$ und einsetzen in Invariante liefert das Teilstück $A[0 \dots n-1]$ welches aus den ursprünglichen Elementen besteht und sortiert ist. Teilstück ist gesamtes Feld.

⇒ Algorithmus **Insertion Sort** arbeitet damit korrekt.

Laufzeitanalyse von Insertion Sort

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

- Festlegung der Laufzeit für jede Zeile
- Jede Zeile besitzt gewissen Kosten c_i
- Jede Zeile wird x mal durchgeführt
- $\text{Laufzeit} = \text{Anzahl} * \text{Kosten}$ jeder Zeile
- Schleifen: Abbruchüberprüfung zählt auch
- t_j : Anzahl der Abfragen der **While**-Schleife
- Laufzeit: $T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_8(n - 1)$

Warum n in Zeile 1?

- Die Überprüfung der Fortführungsbedingung beinhaltet auch die letzte Überprüfung
- Quasi die Überprüfung, durch die die Schleife abbricht

Warum $\sum_{j=1}^{n-1}$ in Zeile 5?

- Aufsummierung aller einzelnen t_j über die Anzahl der Schleifendurchläufe
- Diese ist allerdings $n - 1$ und nicht n , da die Abbruchüberprüfung dort auch enthalten ist

Warum $t_j - 1$ in Zeile 6?

- Selbes Argument wie oben, bei t_j ist die Abbruchüberprüfung enthalten
- Deswegen wird die **while**-Schleife nur $t_j - 1$ -mal ausgeführt

Best Case

- zu sortierendes Feld ist bereits sortiert
- t_j wird dadurch zu 1, da die **While**-Schleife immer nur einmal prüft (Abbruch)
- Die zwei Zeilen innerhalb der **While**-Schleife werden nie ausgeführt
- Durch Umformen ergibt sich, dass die Laufzeit eine lineare Funktion in n ist

Worst Case

- zu sortierendes Feld ist umgekehrt sortiert
- t_j wird dadurch zu $j + 1$, da die **While**-Schleife immer die gesamte Länge prüft
- Durch Umformen ergibt sich, dass die Laufzeit eine quadratische Funktion in n ist (n^2)

Average Case

- im Mittel gut gemischt
- t_j wird dadurch zu $j/2$
- Die Laufzeit bleibt aber eine quadratische Funktion in n (n^2)

Asymptotische Laufzeitbetrachtung Θ

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

2.9 Bubble Sort

Idee – Bubble Sort

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

Code

```
BubbleSort(A)  
1   FOR i = 0 TO A.length - 2  
2       FOR j = A.length - 1 DOWNTO i + 1  
3           IF A[j] < A[j-1]  
4               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:** $\Theta(n^2)$

2.10 Selection Sort

Idee – Selection Sort

- 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)

```
Selection-Sort(A)  
1   FOR i = 0 TO A.length - 2  
2       k = i  
3       FOR j = i + 1 TO A.length - 1  
4           IF A[j] < A[k]  
5               k = j  
6       SWAP(A[i], A[k])
```

2.11 Divide-And-Conquer Prinzip

Idee – Divide-And-Conquer

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

- Anderer Ansatz im Gegensatz zu z.B. **InsertionSort** (inkrementelle Herangehensweise)
- Laufzeit ist im schlechtesten Fall immer noch besser als **InsertionSort**

2.12 Merge Sort

Idee – Merge Sort

- Divide: Teile die Folge aus n Elementen in zwei Teilfolgen von je $\frac{n}{2}$ Elementen 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)
```

```
1 IF p < r
2   q = ⌊(p+r)/2⌋           // Teilen in 2 Teilfolgen
3   MERGE-SORT(A, p, q)      // Sortieren der beiden Teilfolgen
4   MERGE-SORT(A, q+1, r)
5   MERGE(A, p, q, r)        // Vereinigung der beiden sortierten Teilfolgen
```

```
MERGE(A,p,q,r)
```

```
1 n1 = q - p + 1
2 n2 = r - q
3 Let L[0...n1] and R[0...n2] be new arrays
4 FOR i = 0 TO n1 - 1 // Auffüllen der neu erstellten Arrays
5   L[i] = A[p + i]
6 FOR j = 0 TO n2 - 1
7   R[j] = A[q + j + 1]
8 L[n1] = ∞ // Einfügen des Sentinel-Wertes
9 R[n2] = ∞
10 i = 0
11 j = 0
12 FOR k = p TO r // Eintragweiser Vergleich der Elemente
13   IF L[i] ≤ R[j]
14     A[k] = L[i] // Sortiertes Zurückschreiben in Original-Array
15     i = i + 1
16   ELSE
17     A[k] = R[j]
18     j = j + 1
```

(Teilarrays werden nicht parallel bearbeitet)

Korrektheit von MergeSort

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

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

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

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

Analyse von MergeSort

Ziel Bestimme Rekursionsgleichung für Laufzeit $T(n)$ von n Zahlen im schlechtesten Fall

Divide Berechnung der Mitte des Feldes: Konstante Zeit $\Theta(1)$

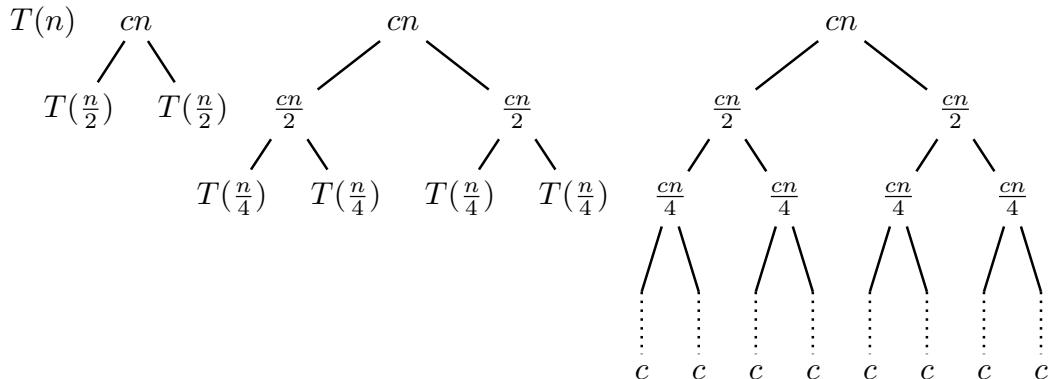
Conquer Rekursives Lösen von zwei Teilproblemen der Größe $\frac{n}{2}$: Laufzeit von $2 T(\frac{n}{2})$

Combine MERGE auf einem Teilstück der Länge n : Lineare Zeit $\Theta(n)$

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

Lösen der Rekursionsgleichung mithilfe eines Rekursionsbaums

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



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

2.13 Quicksort

Idee – Quicksort

Pivotelement Wahl eines Pivotelement x aus dem Array (=Mittelsäule bei Sortierung)

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

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

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

Code

QUICKSORT(A, p, r)

```
1 IF  $p < r$       // Überprüfung, ob Teilarray leer ist
2    $q = \text{PARTITION}(A, p, r)$ 
3    $\text{QUICKSORT}(A, p, q-1)$ 
4    $\text{QUICKSORT}(A, q+1, r)$ 
```

PARTITION(A, p, r)

```
1  $x = A[r]$       // Wahl des Pivotelements
2  $i = p - 1$     // Index  $i$  setzen
3 FOR  $j = p$  TO  $r - 1$  // Auffüllen des Teilarrays mit Elementen
4   IF  $A[j] \leq x$ 
5      $i = i + 1$ 
6      $\text{SWAP}(A[i], A[j])$ 
7  $\text{SWAP}(A[i+1], A[r])$  // Tausch des Pivotelements
8 RETURN  $i + 1$  // Neuer Index des Pivotelements
```

(Teilarrays werden nicht parallel bearbeitet)

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 i 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 mit Kosten $\Theta(n)$ zieht sich durch gesamte Rekursion
 - 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 $O(n \lg(n))$

2.14 Laufzeitanalyse von rekursiven Algorithmen

2.14.1 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)

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

2.14.2 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 $n' < n$.

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

2.14.3 Rekursionsbaum

Idee – Rekursionsbaum

Stellen das Ineinander-Einsetzen als Baum dar und Analyse der Kosten.
Ablauf:

1. Jeder Knoten stellt die Kosten eines Teilproblems dar
 - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar
 - Die Blätter stellen die Kosten der Basisfälle dar (z.B. $T(0)$)
2. Berechnen der Kosten innerhalb jeder Ebene des Baums
3. Die Gesamtkosten sind die Summe über die Kosten aller Ebenen

Rekursionsbaum ist nützlich um Lösung für Substitutionsmethode zu erraten

Beispiel

$$T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$$

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 $c(\frac{n}{4^i})^2, i = 0, \dots, \log_4 n - 1$
- Anzahl · Kosten = $3^i \cdot c(\frac{n}{4^i})^2 = (\frac{3}{16})^i \cdot cn^2$

Unterste Ebene

- $3^{\log_4(n)} = n \log_4(3)$ Knoten
- Jeder Knoten trägt $T(1)$ Kosten bei
- Kosten unten: $n^{\log_4(3)} \cdot T(1) = \Theta(n^{\log_4(3)})$

Addiere alle Kosten aller Ebenen

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + \dots + (\frac{3}{16})^{\log_4 n - 1}cn^2 + \Theta(n^{\log_4(3)}) \\ &= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(\frac{3}{16})^{\log_4 n} - 1}{\frac{3}{16} - 1} \cdot cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

- Verwendung einer unendlichen fallenden geometrischen Reihe als obere Schranke:

$$\begin{aligned} T(n) &= \sum_{i=0}^{-1} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - \frac{3}{16}} \cdot cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} \cdot cn^2 + \Theta(n^{\log_4 3}) = O(n^2) \end{aligned}$$

Jetzt

Zu zeigen $\exists d > 0 : T(n) \leq dn^2$

Substitutionsmethode

Induktionsanfang

$$\begin{aligned} T(n) &= 3 \cdot T(\left\lfloor \frac{1}{4} \right\rfloor) + c \cdot 1^2 \\ &= 3 \cdot T(0) + c = c \end{aligned}$$

Induktionsschritt

$$\begin{aligned} T(n) &\leq 3 \cdot T(\left\lfloor \frac{n}{4} \right\rfloor) + cn^2 \\ &\leq 3 \cdot d(\left\lfloor \frac{n}{4} \right\rfloor)^2 + cn^2 \\ &\leq 3d(\frac{n}{4})^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2, \text{ falls } d \geq \frac{16}{13}c \end{aligned}$$

2.14.4 Mastertheorem

Idee – Mastertheorem

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

1. Gilt $f(n) = O(n^{\log_b(a-\epsilon)})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b(a)})$
2. Gilt $f(n) = \Theta(n^{\log_b(a)})$, 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 $T(n) = \Theta(n \lg(n))$

Grundlegende Datenstrukturen	Fortgeschrittene Datenstrukturen	Randomisierte Datenstrukturen
Stacks	Rot-Schwarz-Bäume	Skip Lists
Verkettete Listen	AVL-Bäume	Hash Tables
Queues	Splay-Bäume	Bloom-Filter
Bäume	Heaps	
Binäre Suchbäume	B-Bäume	

Tabelle 3: Übersicht Datenstrukturen

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

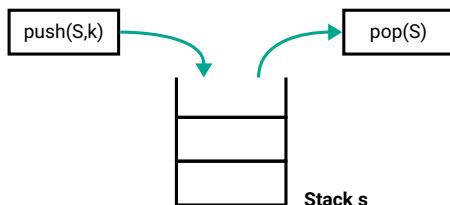
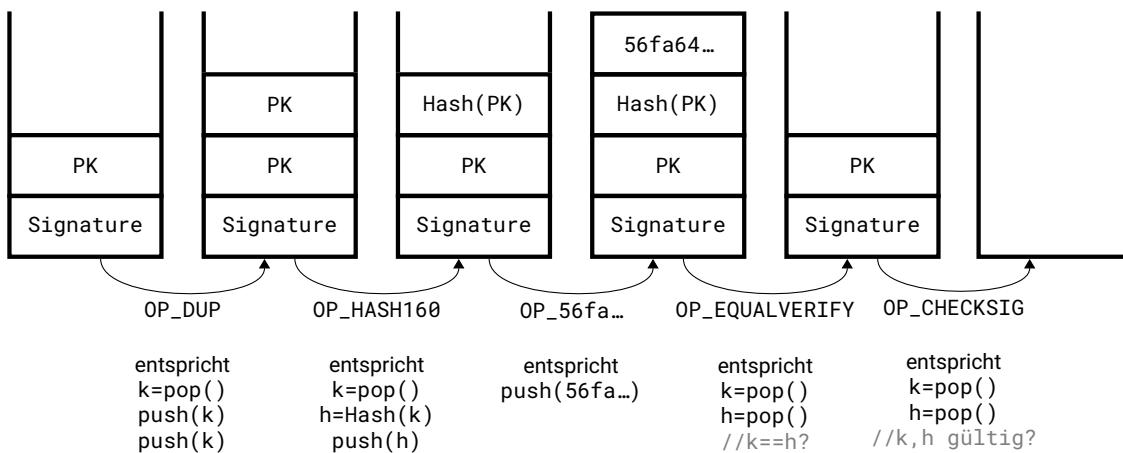


Abbildung 8: Abstrakter Aufbau eines Stacks

Beispiel Bitcoin

```
scriptPubKey:  
OP_DUP OP_HASH160 56fa64a8bd7852d2c58095fa9a2fc...  
OP_EQUALVERIFY OP_CHECKSIG
```



Stacks als Array

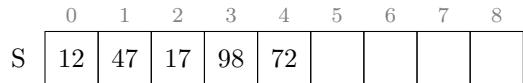


Abbildung 9: Beispiel: Stack als Array

`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)

```

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 – Stacks mit Variabler Größe

- 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)

```

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

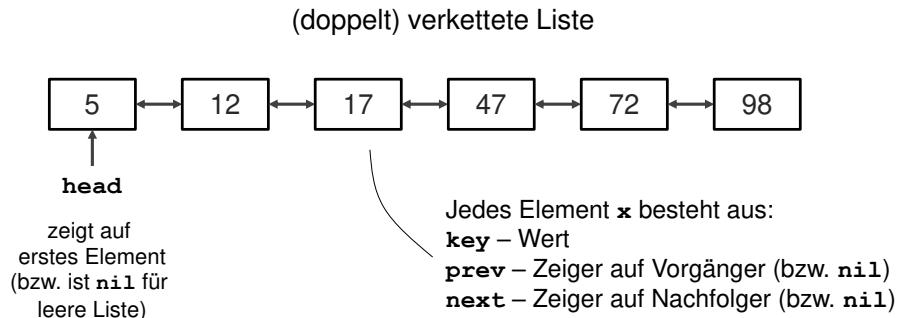


Abbildung 10: Aufbau Verkettete Liste

Verkettete Listen durch Arrays

Entspricht doppelter Verkettung zwischen 45 und 12

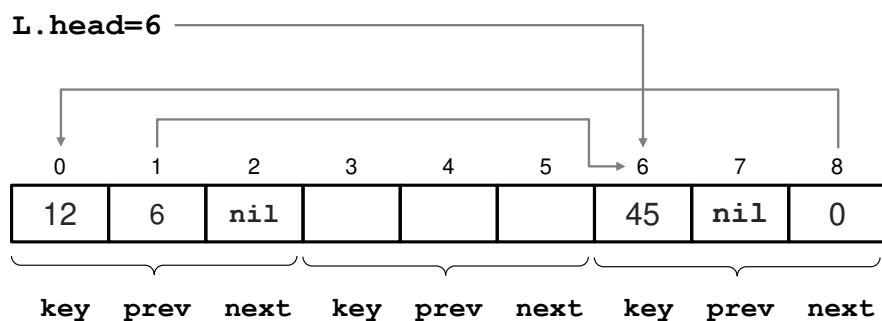


Abbildung 11: Beispiel Verkettete Liste durch Arrays

3.2.1 Elementare Operationen auf Listen

Suche nach Element

`search(L,k) // Returns pointer to k in L (or nil)`

```

1 current = L.head;
2 WHILE current != nil AND current.key != k DO
3     current = current.next;
4 return current;

```

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

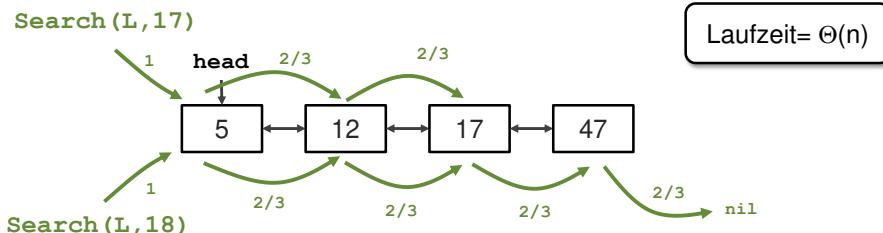


Abbildung 12: Grafische Darstellung einer Suche in Listen

Einfügen eines Elements am Kopf der Liste

`insert(L,x)`

```

1 insert(L,x)
2 x.next = l.head;
3 x.prev = nil;
4 IF L.head != nil THEN
5     L.head.prev = x;
6 L.head = x;

```

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

Löschen eines Elements aus Liste

`delete (L,x)`

```

1 IF x.prev != nil THEN
2     x.prev.next = x.next
3 ELSE
4     L.head = x.next;
5 IF x.next != nil THEN
6     x.next.prev = x.prev;

```

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

Vereinfachung per Wächter/Sentinels

Ziel ist die Eliminierung der Spezialfälle für Listenanfang/-ende

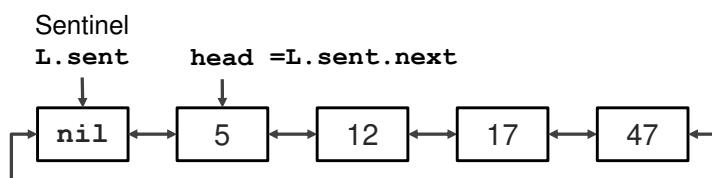


Abbildung 13: Beispiel Sentinel

Löschen mit Sentinels:

`deleteSent(L,x)`

```

1 x.prev.next = x.next;
2 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

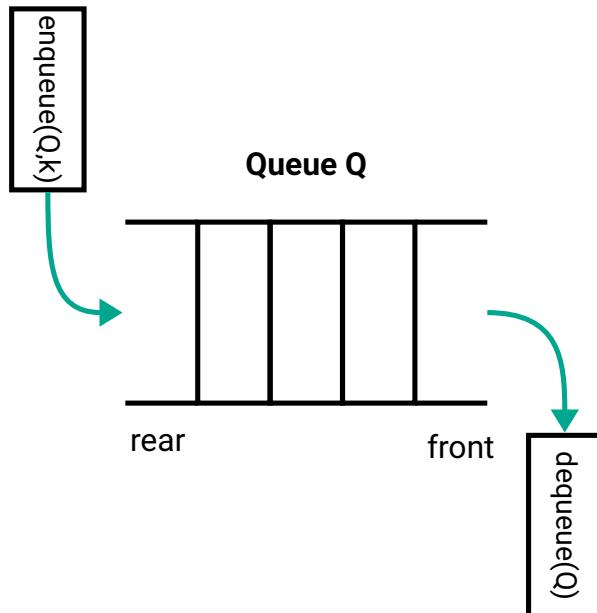


Abbildung 14: Beispiel FIFO

Queues als (virtuelles) zyklisches Array

Bekannt: Maximale Elemente gleichzeitig in Queue

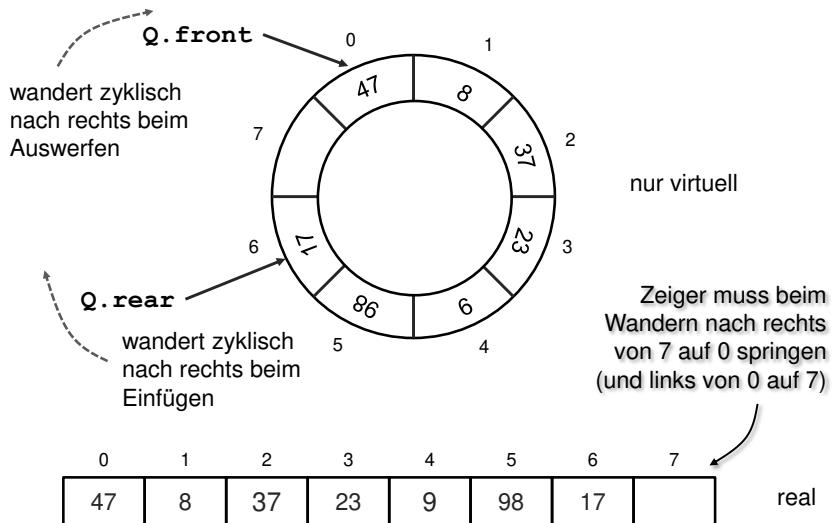


Abbildung 15: Beispiel Queue als Zyklisches Array

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:

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

```
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.isEmpty
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

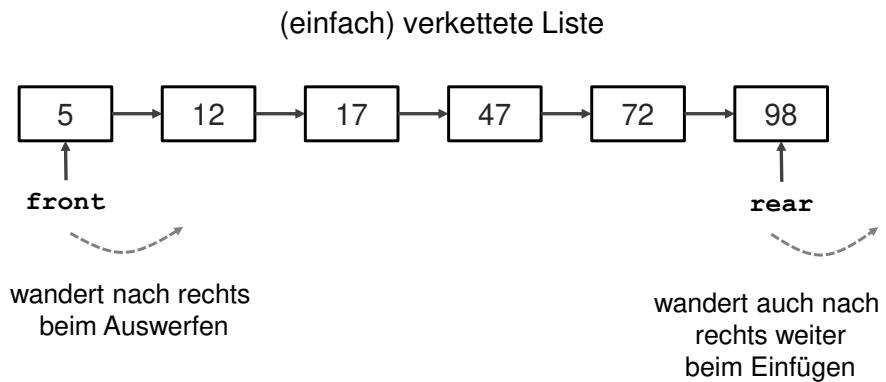


Abbildung 16: Beispiel Queue durch einfach verkettete Liste

Methoden:

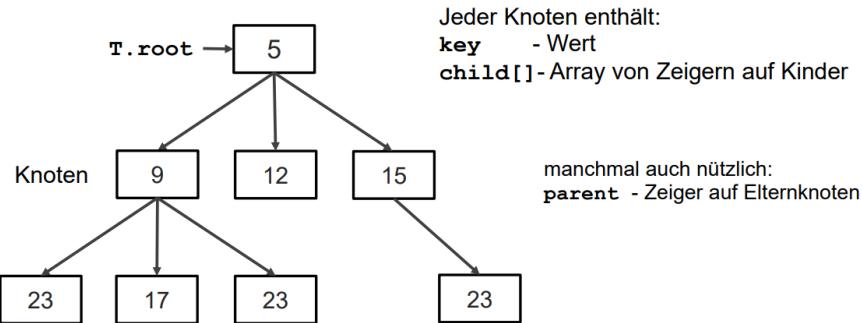
new(Q)	isEmpty(Q)
1 Q. front =nil; 2 Q. rear =nil;	1 IF Q. front ==nil THEN 2 return true ; 3 ELSE 4 return false ;
dequeue(Q)	enqueue(Q,k)
1 IF isEmpty (Q) THEN 2 error "underflow"; 3 ELSE 4 x=Q. front ; 5 Q. front =Q. front .next; 6 return x;	1 IF isEmpty (Q) THEN 2 Q. front =x; 3 ELSE 4 Q. rear .next=x; 5 x. next =nil; 6 Q. rear =x;

Laufzeit:

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

3.4 Binäre Bäume

Bäume durch verkettete Listen

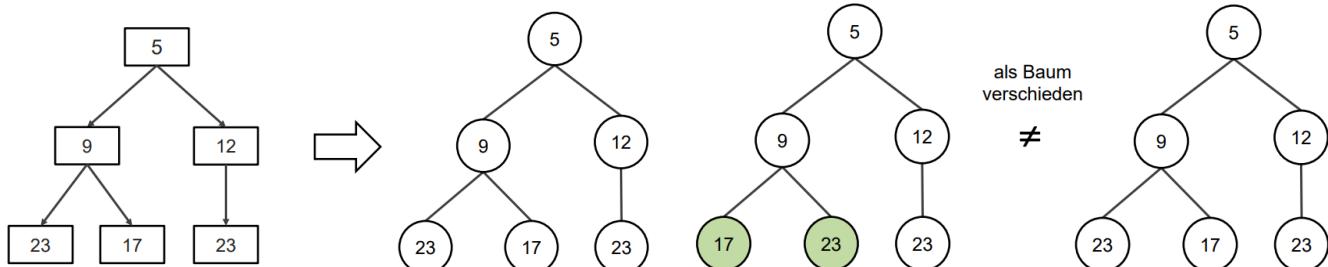


Baum-Bedingung: Baum ist leer oder...
 es gibt einen Knoten r („Wurzel“), so dass jeder Knoten v von der Wurzel aus per eindeutiger Sequenz von `child`-Zeigern erreichbar ist:
 $v = r.child[i1].child[i2] \dots .child[im]$

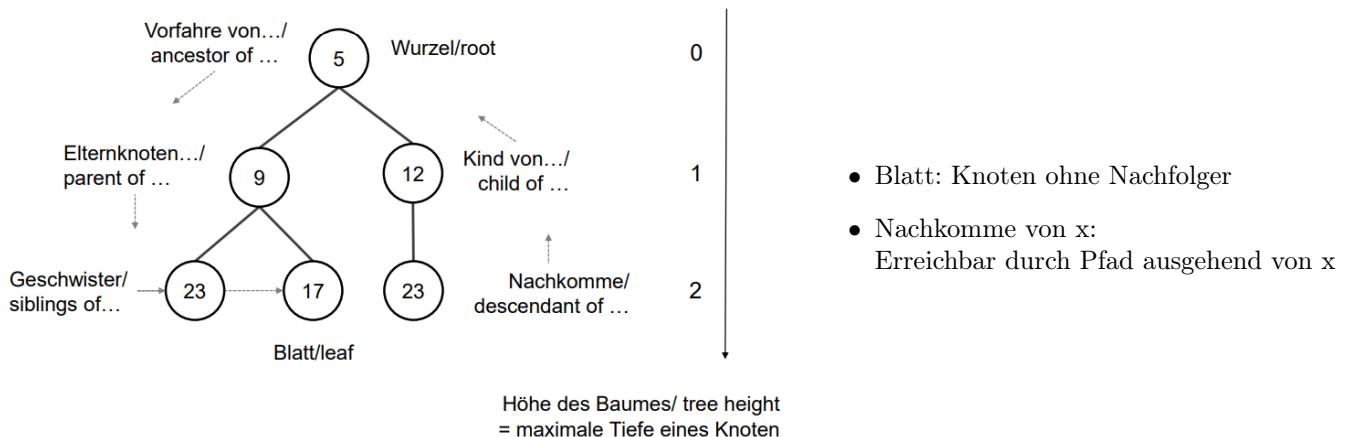
Abbildung 17: Binärbaum-Beispiel

Bäume sind „azyklisch“ (also „keine Schleifen zwischen Knoten“)

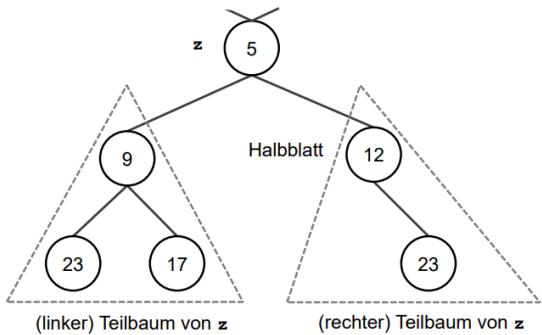
Darstellung als (ungerichteter) Graph



Allgemeine Begrifflichkeiten



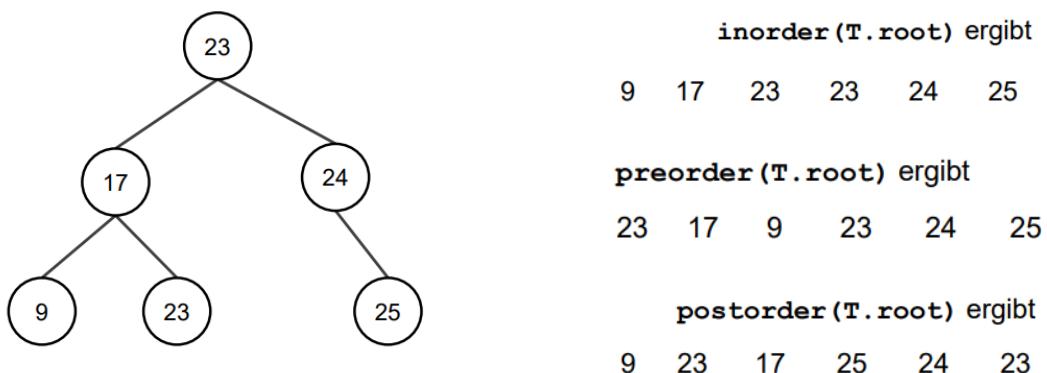
Begrifflichkeiten Binärbaum



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

Traversieren von Bäumen

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



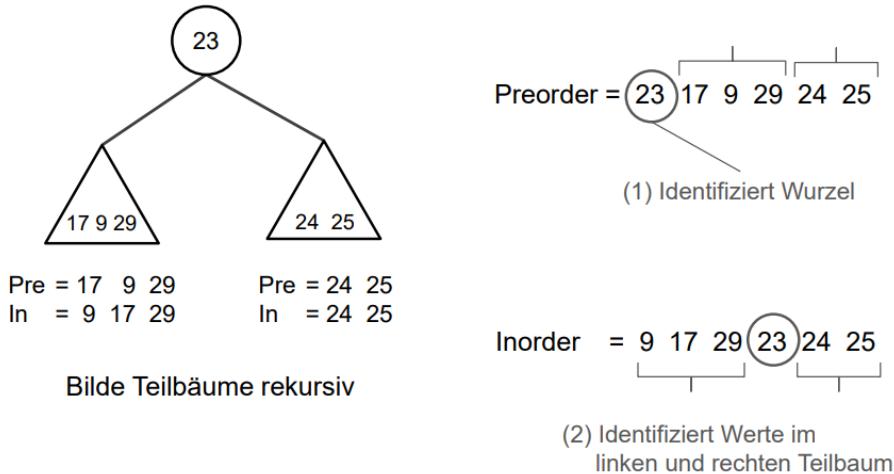
Code:

inorder(x)	preorder(x)	postorder(x)
<pre> 1 IF x != nil THEN 2 inorder(x.left); 3 print x.key; 4 inorder(x.right); </pre>	<pre> 1 IF x != nil THEN 2 print x.key; 3 preorder(x.left); 4 preorder(x.right); </pre>	<pre> 1 IF x != nil THEN 2 postorder(x.left); 3 postorder(x.right); 4 print x.key; </pre>

Anmerkung: bei Inorder im BST sind Zahlen einfach nur aufsteigend Sortiert

Eindeutige Bestimmbarkeit von Bäumen

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



3.4.1 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(x) | • Fügt Element x in Baum t hinzu |
| t.delete(x) | • Löscht x aus Baum t |

Suche nach Elementen Starte mit `search(T.root, k)` Code:

```
search(x,k)
1 IF x == nil THEN return nil;
2 IF x.key == k THEN return x;
3 y = search(x.left,k);
4 IF y != nil THEN return y;
5 return search(x.right,k);
```

Laufzeit = $\Theta(n)$ (Jeder Knoten maximal einmal, jeder Knoten im schlechtesten Fall)

Einfügen von Elementen

Hier wird als Wurzel eingefügt (Achtung: Erzeugt linkslastigen Baum) Code:

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

Laufzeit = $\Theta(1)$

Löschen von Elementen

Hier: Ersetze x durch Halbblatt ganz rechts

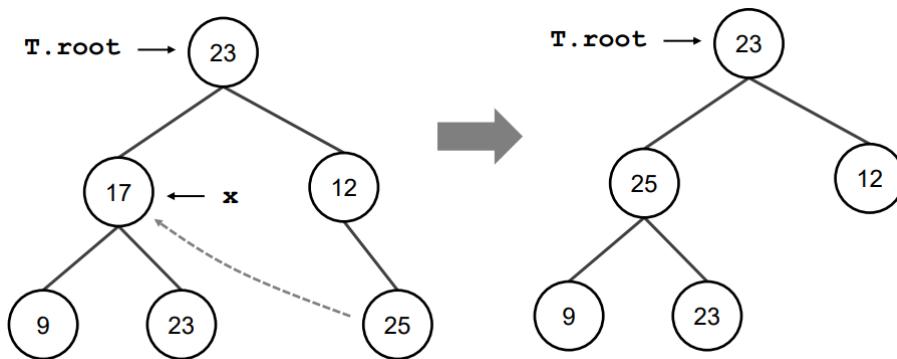


Abbildung 18: Löschen des Knoten 17

Connect-Algorithmus:

```
connect(T,y,w) // Connects w to y.parent
1  v = y.parent;
2  IF y != T.root THEN
3      IF y == v.right THEN
4          v.right = w;
5      ELSE
6          v.left = w;
7  ELSE
8      T.root = w;
9  IF w != nil THEN
10     w.parent = v;
```

Laufzeit = $\Theta(1)$

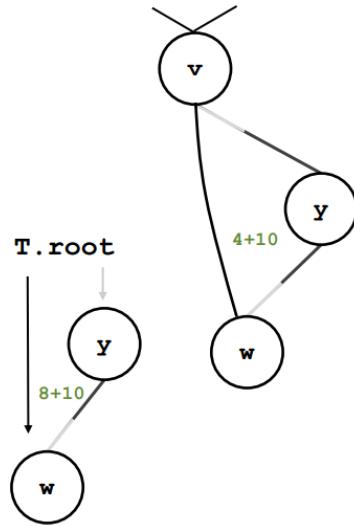


Abbildung 19: Beispiel Connect-Algorithmus
Binärbaum

Delete-Algorithmus:

```
delete(T,x) // assumes x in T
1  y = T.root;
2  WHILE y.right != nil DO
3      y = y.right;
4  connect(T,y,y.left);
5  IF x != y THEN
6      y.left = x.left;
7      IF x.left != nil THEN
8          x.left.parent = y;
9      y.right = x.right;
10     IF x.right != nil THEN
11         x.right.parent = y;
12     connect(T,x,y);
```

Laufzeit = $\Theta(h)$ (Höhe des Baumes, $h = n$ möglich)

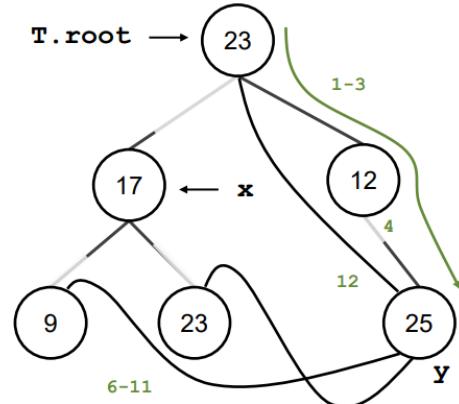


Abbildung 20: Beispiel Delete-Algorithmus
Binärbaum

3.5 Binäre Suchbäume

Definition – Binärer Suchbaum

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

Code:

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

Iterativer Code:

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

Laufzeit (beide) = $O(h)$ (Höhe)

search(T.root, 22)

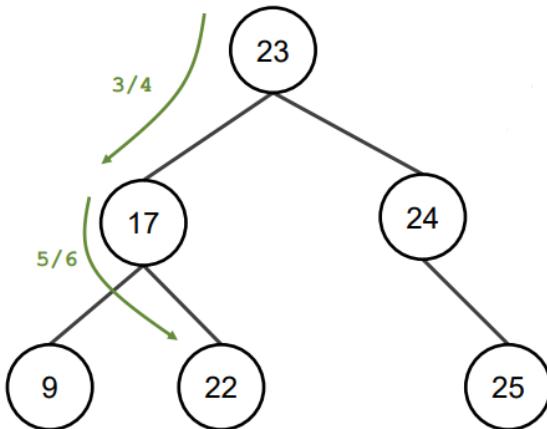


Abbildung 21: Beispiel Search-Algorithmus im Binärbaum

Einfügen im Binary Search Tree

Aufwendiger, da Ordnung erhalten werden muss Code:

```
insert (T,z) // z.left == z.right == nil;
```

```

1  x = T.root;
2  px = nil;
3  WHILE x != nil DO
4      px = x;
5      IF x.key > z.key THEN
6          x = x.left;
7      ELSE
8          x = x.right;
9      z.parent = px;
10     IF px == nil THEN
11         T.root = z;
12     ELSE
13         IF px.key > z.key THEN
14             px.left = z;
15         ELSE
16             px.right = z;

```

Laufzeit = $O(h)$

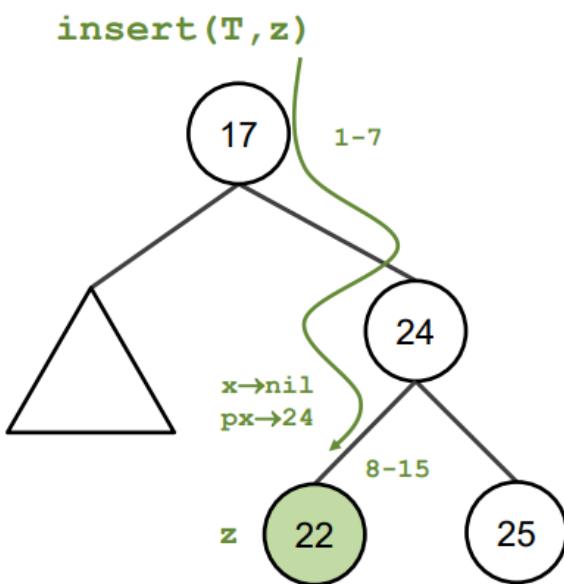


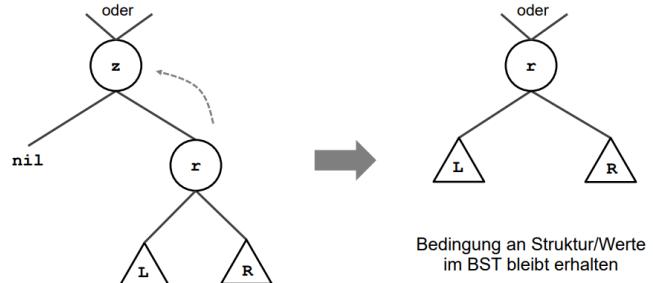
Abbildung 22: Beispiel einfügen in Binären Suchbaum

Löschen im BST

wir unterscheiden drei verschiedene Fälle:

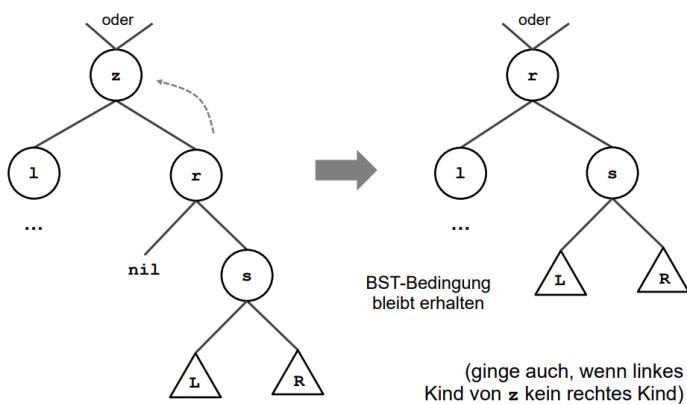
Löschen im BST (I)

zu löschernder Knoten z hat maximal ein Kind



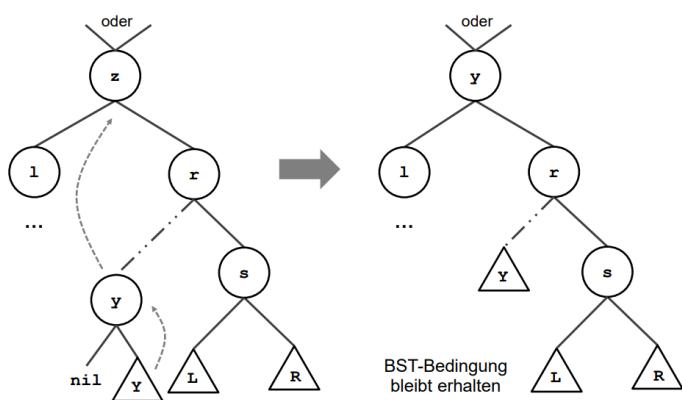
Löschen im BST (II)

rechtes Kind von Knoten z hat kein linkes Kind



Löschen im BST (III)

„kleinster“ Nachfahre vom rechten Kind von z



Code:

```
transplant(T,u,v) // Hängt Teilbaum v an Parent von u
```

```

1 IF u.parent == nil THEN
2     T.root = v;
3 ELSE
4     IF u == u.parent.left THEN
5         u.parent.left = v;
6     ELSE
7         u.parent.right = v;
8 IF v != nil THEN
9     v.parent = u.parent;
```

```
delete(T,z)
```

```

1 IF z.left == nil THEN
2     transplant(T,z,z.left)
3 ELSE
4     IF z.right == nil THEN
5         transplant(T,z,z.left)
6     ELSE
7         y = z.right;
8         WHILE y.left != nil DO y = y.left;
9         IF y.parent != z THEN
10            transplant(T,y,y.right)
11            y.right = z.right;
12            y.right.parent = y;
13            transplant(T,z,y)
14            y.left = z.left;
15            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 (links- bzw. rechtslastiger Baum)
 - $h = n - 1$
 - 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 Speicherplatz

Suchbäume als Suchindex

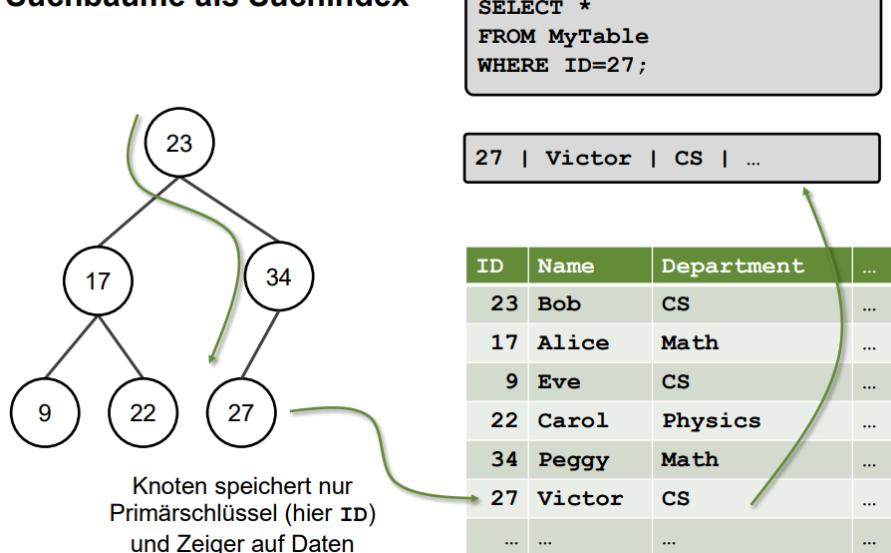


Abbildung 23: Suchbäume als Suchindex

4 Advanced Data Structures

4.1 Rot-Schwarz-Bäume

Definition – Rot-Schwarz-Baum

- Ist ein binärer Suchbaum mit den zusätzlichen Eigenschaften:
 - 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 oder Halbblatt die selbe Anzahl an 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 gilt 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 \Rightarrow Höhe $O(\log n)$

Alle folgenden Algorithmen arbeiten mithilfe eines Sentinels (zeigt auf sich selbst)

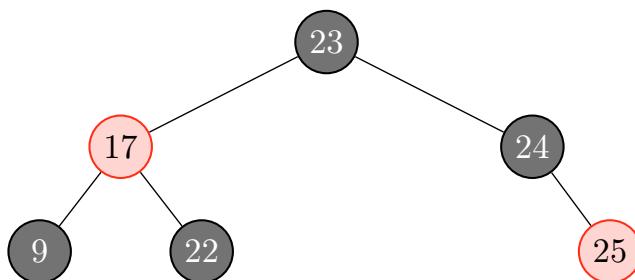


Abbildung 24: Beispielhafte Darstellung wie in 1.3

Einfügen

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

```

insert(T,z) //z.left==z.right==nil;
1  x=T.root; px=T.sent;
2  WHILE x != nil DO           //Bis zum passenden Blatt gehen
3      px=x;
4      IF x.key > z.key THEN
5          x=x.left;
6      ELSE
7          x=x.right;
8      z.parent=px;
9      IF px==T.sent THEN      // Einfügen
10         T.root=z
11     ELSE
12         IF px.key > z.key THEN
13             px.left=z;
14         ELSE
15             px.right=z;
16         z.color=red;          // ab hier anders als bei BST-Insert
17         fixColorsAfterInsertion(T,z);

```

Laufzeit: $\Theta(h)$ (h jedoch $\log n$)

Hilfsmethode `rotateLeft`

```

rotateLeft(T,x)
1  y = x.right;
2  x.right = y.left;
3  IF y.left != nil THEN
4      y.left.parent = x;
5  y.parent = x.parent;
6  IF x.parent == T.sent THEN
7      T.root = y;
8  ELSE
9      IF x == x.parent.left THEN
10         x.parent.left = y;
11     ELSE
12         x.parent.right = y;
13  y.left = x;
14  x.parent = y;

```

fixColorsAfterInsertion Beim Aufrufen werden zwei Bedingungen potentiell verletzt:

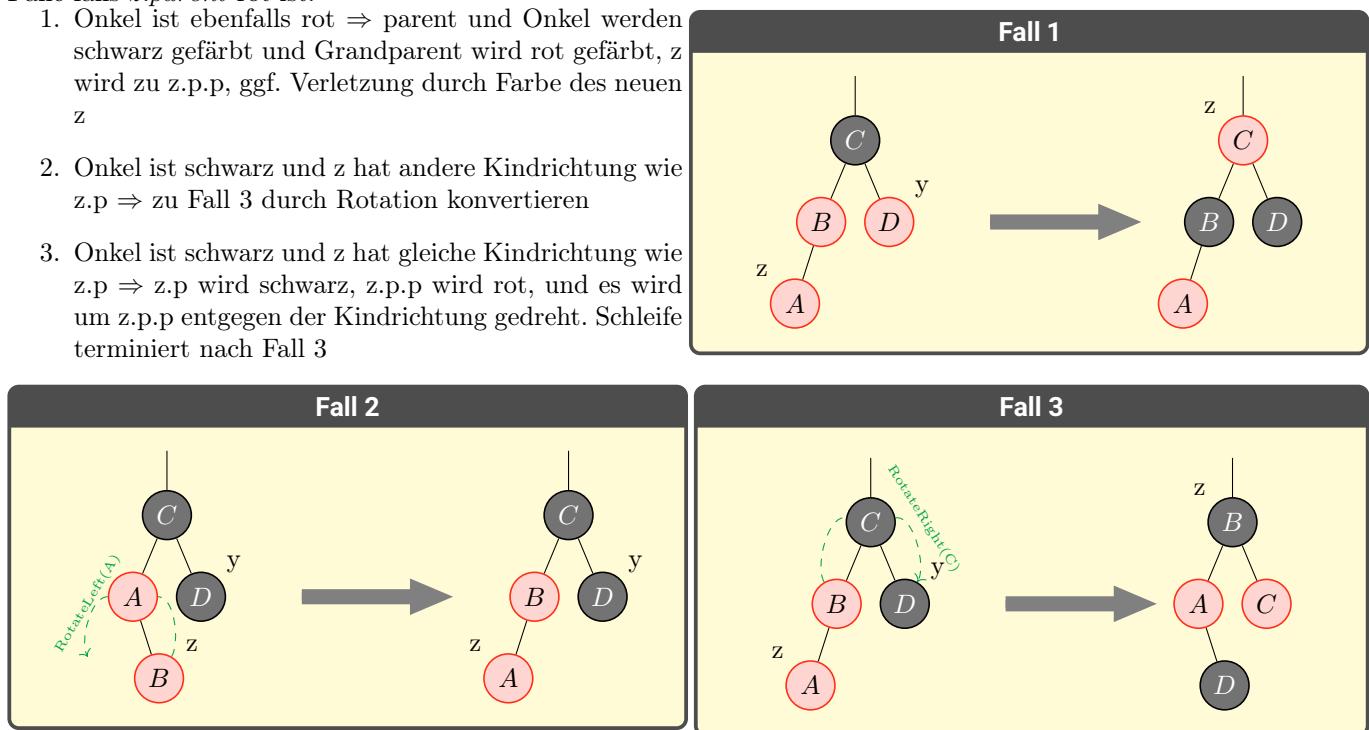
1. root ist nicht mehr schwarz
2. wenn eine Node rot ist müssen beide Kinder schwarz sein

Also müssen wir:

- nach Rotation die root des Baumes ggf auf schwarz setzen
- Überprüfen, ob RSB-Bedingung verletzt wurde
- Bloß wenn das parent auch rot ist kommen wir in Verlegenheit \Rightarrow wir müssen den Algorithmus starten

Fälle falls $z.parent$ rot ist:

1. Onkel ist ebenfalls rot \Rightarrow parent und Onkel werden schwarz gefärbt und Grandparent wird rot gefärbt, z wird zu z.p.p, ggf. Verletzung durch Farbe des neuen z
2. Onkel ist schwarz und z hat andere Kindrichtung wie z.p \Rightarrow zu Fall 3 durch Rotation konvertieren
3. Onkel ist schwarz und z hat gleiche Kindrichtung wie z.p \Rightarrow z.p wird schwarz, z.p.p wird rot, und es wird um z.p.p entgegen der Kindrichtung gedreht. Schleife terminiert nach Fall 3



fixColorsAfterInsertion(T,z)

```

1  WHILE z.parent.color == red DO           // solange der Elternknoten rot ist
2      IF z.parent == z.parent.parent.left THEN // Linkes Kind (if-Fall)
3          y = z.parent.parent.right;
4          IF y != nil AND y.color == red THEN // Fall 1
5              z.parent.color = black;
6              y.color = black;
7              z.parent.parent.color = red;
8              z = z.parent.parent;           // rekursiv nach oben weiterführen
9          ELSE
10             IF z == z.parent.right THEN // Fall 2
11                 z = z.parent;
12                 rotateLeft(T,z);
13                 z.parent.color = black;   // Fall 3
14                 z.parent.parent.color = red;
15                 rotateRight(T, z.parent.parent);
16             ELSE
17                 // Tauschen von rechts und links
18                 T.root.color = black;      // Setzen der Wurzel auf Schwarz

```

Löschen

- analog zum binären Suchbaum, aber neue Node erbt Farbe der alten Node
- Wenn „neue“ Node schwarz war \Rightarrow Fixup
- Verschiedene Fälle, die auch gegenseitig Voraussetzungen für einander sind

Hilfsroutine transplant:

```
transplant(T,u,v)
1  IF u.parent==nil THEN
2      T.root=v;
3  ELSE
4      IF u==u.parent.left THEN
5          u.parent.left=v;
6      ELSE
7          u.parent.right=v;
8  IF v != nil THEN
9      v.parent=u.parent;
```

```
delete(T,z)
1  y=z;
2  y-original-color=y.color;
3  IF z.left == nil;
4      x = z.right;
5      transplant(T,z,z.right);
6  ELSE IF z.right == nil;
7      x = z.left;
8      transplant(T,z,z.left);
9  ELSE
10     y TREE-MINIMUM(z.right);
11     y-original-color=y.color;
12     x=y.right;
13     IF y.p == z
14         x.p = y;
15     ELSE
16         transplant(T,y,y.right);
17         y.right=z.right;
18         y.right.p=y;
19     transplant(T,z,y);
20     y.left=z.left;
21     y.left.p=y;
22     y.color=z.color;
23 IF y-original-color == BLACK
24     deleteFixup(T,x);
```

Laufzeit: $O(h) = O(\log n)$

Delete kann die RSB-Bedingung verletzen. Das fixup hat vier Fälle:

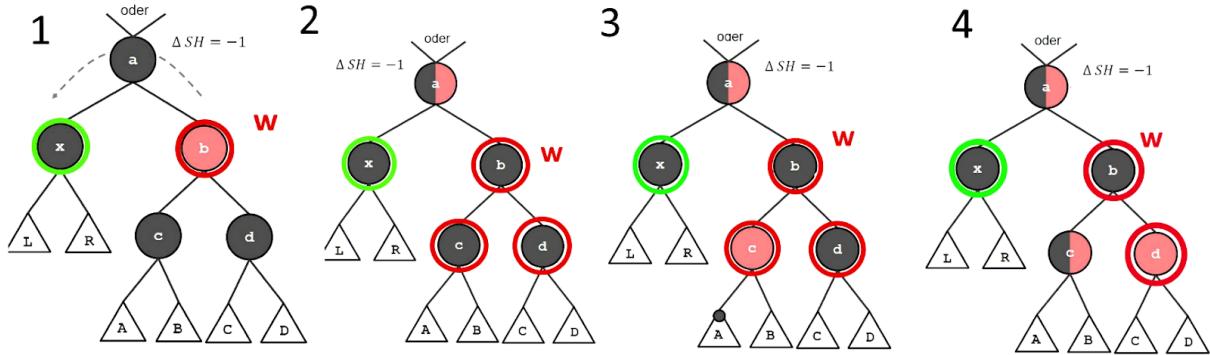


Abbildung 25: Fälle für Fixup bei delete in RSB

1. Knoten w (Bruder von Knoten x) ist rot

- Da w schwarze Kinder haben muss, können wir die Farben von w und $x.p$ wechseln und dann eine Linksdrehung auf $x.p$ ausführen, ohne eine der rot-schwarzen Eigenschaften zu verletzen.
- Das neue Geschwisterchen von x , das vor der Rotation eines der Kinder von w ist, ist jetzt schwarz. Wir haben also Fall 1 in Fall 2, 3 oder 4 konvertiert. Die Fälle 2, 3 und 4 treten auf, wenn der Knoten w schwarz ist, aber unterschiedliche Farben der Kinder.

2. w ist schwarz und beide Kinder von w sind schwarz

- entfernen eines Schwarz von x und w , wobei x nur ein Schwarz hat und w rot bleibt
- Um Entfernen aus x und w zu kompensieren ein zusätzliches Schwarz hinzufügen.
- w konnte entweder Schwarz oder rot sein.
- Wenn $x.p (= a)$ schwarz, dann Vaterknoten $\Delta SH = -1$; verfahren rekursiv mit $x.p (= a)$ als neuem x ; \Rightarrow wir wiederholen die while-Schleife mit $x.p$ als neuem Knoten x .

3. w ist schwarz, w 's linkes Kind ist rot und w 's rechtes Kind ist schwarz

- Farben von w und seinem linken Kind links ändern und dann eine Rechtsdrehung auf w ausführen
 - Das neue Geschwister w von x ist jetzt ein schwarzer Knoten mit einem roten rechten Kind
- \Rightarrow Fall 3 in Fall 4 transformiert.

4. w ist schwarz und das rechte Kind von w ist rot

- w erbt die Farbe von $x.p$
 - $x.p$ wird schwarz
 - $w.right$ wird schwarz
- \Rightarrow Linksdrehung von $x.p$
- x als Wurzel festlegen, wird der Whileloop beendet, wenn die Schleifenbedingung getestet wird.

```
deleteFixup(T,z)
1 WHILE x != T.root and x.color == BLACK
2   IF x==x.p.left
3     w=x.p.right;
4       IF w.color == RED
5         w.color=BLACK;                                // Case 1
6         x.p.color=RED;
7         rotateLeft(T,x.p);
8         w=x.p.right;
9       IF w.left.color == w.right.color == BLACK    // Case 2
10      w.color = RED;
11      x=x.p;
12    ELSE IF w.right.color == BLACK                 // Case 3
13      w.left.color = BLACK;
14      w.color=RED;
15      rotateRight(T,w);
16      w=x.p.right;
17      w.color=x.p.color;                           // Case 4
18      x.p.color=BLACK;
19      w.right.color=BLACK;
20      rotateLeft(T,x.p);
21      x=T.root;
22  ELSE
23    // same as above with "right" and "left" exchanged
24  x.color=BLACK;
```

Worst-Case-Laufzeiten

Einfügen $\Theta(\log n)$

Löschen $\Theta(\log n)$

Suchen $\Theta(\log n)$

4.2 AVL-Bäume

Definition – AVL-Baum

Binärer Suchbaum, aber für Balance in **jedem** Knoten nur $-1, 0$, oder 1 erlaubt.

Balance für x : $B(x) = \text{Höhe}(\text{rechter Teilbaum}) - \text{Höhe}(\text{linker Teilbaum})$

$h \leq 1.441 \cdot \log n$ (optimierte Konstanten - 1,441 vs 2 (RBT))

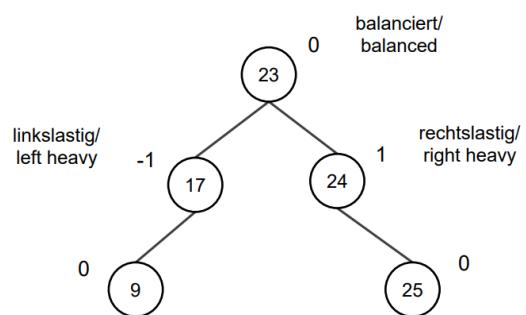


Abbildung 26: Beispiel Balance AVL Baum

- AVL**
- Einfügen und Löschen verletzen in der Regel öfter die Baum-Bedingung
 - Aufwendiger zum Rebalancieren

- Rot-Schwarz**
- Suchen dauert evtl. länger

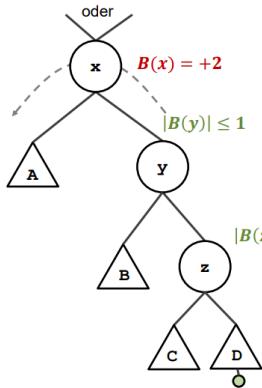
- Konklusion**
- AVL geeigneter, wenn mehr Such-Operationen und weniger Einfügen und Löschen

- Gemeinsamkeiten**
- $\text{AVL} \subset \text{Rot-Schwarz}$
 - $\text{AVL-Baum} \Rightarrow \text{Rot-Schwarz-Baum mit Höhe } \lceil \frac{h+1}{2} \rceil$
 - Für jede Höhe $h \geq 3$ gibt es einen RBT, der kein AVL-Baum ist ($\text{AVL} \neq \text{RBT}$)

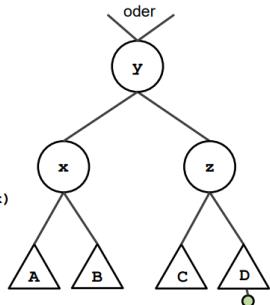
Einfügen

- Einfügen funktioniert wie beim Binary Search Tree mit Sentinel
- Erfordert danach jedoch Rebalancieren weiter oben im Baum
- Rebalancieren: (verschiedene Fälle)

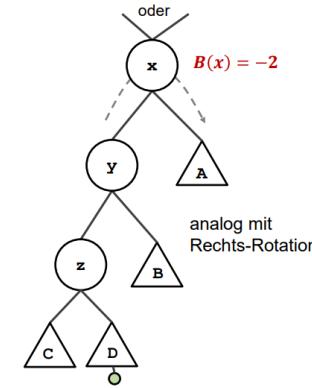
Rebalancieren: Fall I



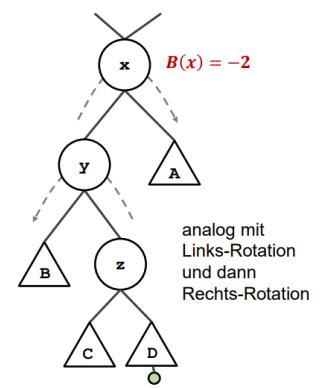
`rotateLeft(T, x)`



Rebalancieren: Fälle III+IV

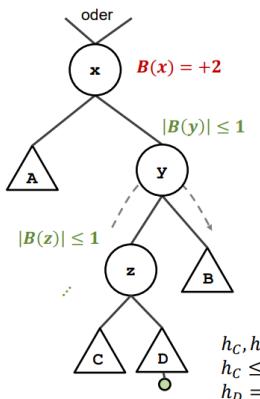


analog mit Rechts-Rotation

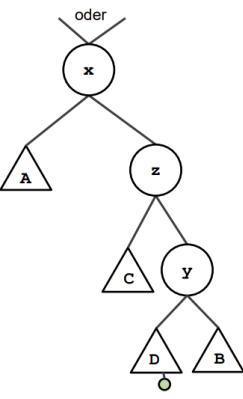


analog mit Links-Rotation und dann Rechts-Rotation

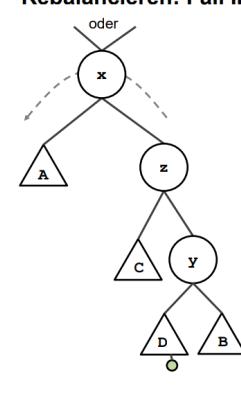
Rebalancieren: Fall II (erste Rotation)



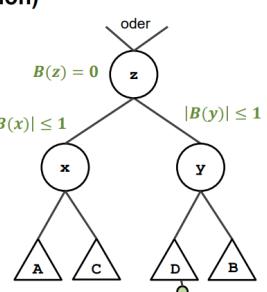
`rotateRight(T, y)`



Rebalancieren: Fall II (zweite Rotation)



`rotateLeft(T, x)`



Löschen

- Analog zum binären Suchbaum
- Rebalancieren (eventuell bis zur 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 – Splay-Baum

- selbst-organisierender Baum
- Ansatz: Einmal angefragte Werte werden wahrs. noch öfter angefragt
- Angefragte Werte nach oben schieben
- Splay-Bäume sind eine Untermenge von Rot-Schwarz-Bäumen (\subseteq)

Splay-Operationen

- Suchen oder Einfügen: Spüle gesuchten oder neu eingefügten Knoten an die Wurzel
- Splay: Folge von Zig-,Zig-Zig-, Zig-Zag-Operationen

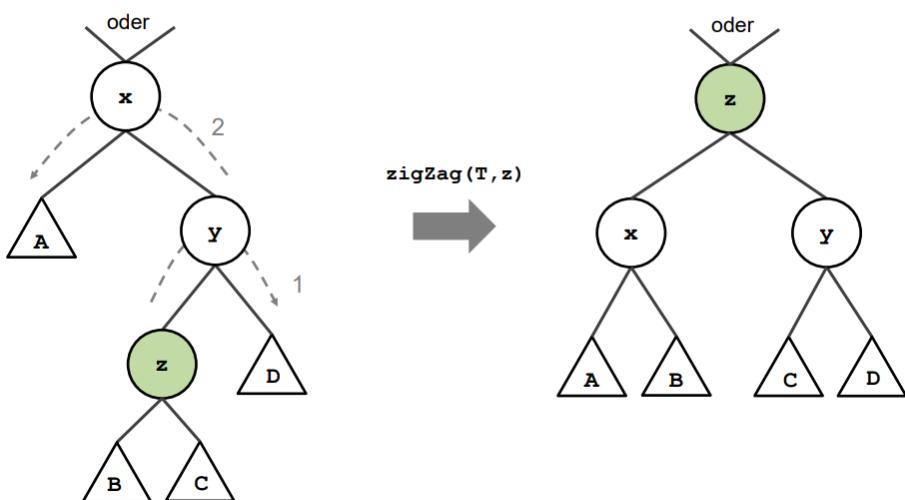
splay(T,z)

```

1  WHILE z != T.root DO
2      IF z.parent.parent == nil THEN
3          zig(T,z);
4      ELSE
5          IF z == z.parent.parent.left.left OR
6              z == z.parent.parent.right.right THEN
7              zigZig(T,z);
8          ELSE
9              zigZag(T,z);

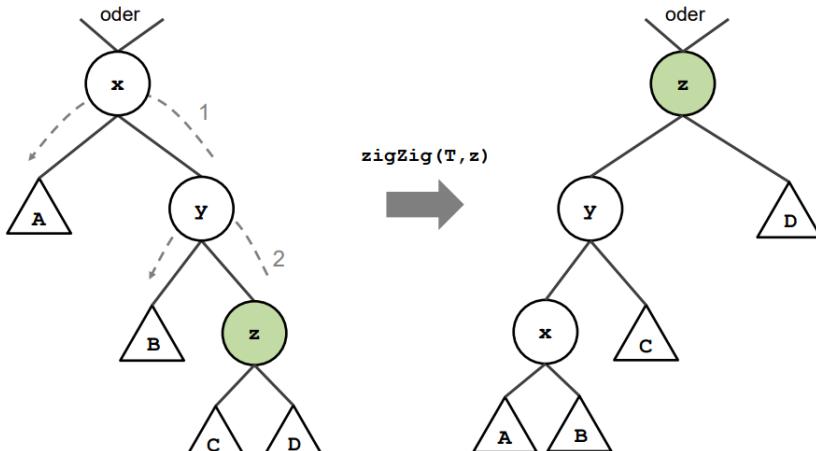
```

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



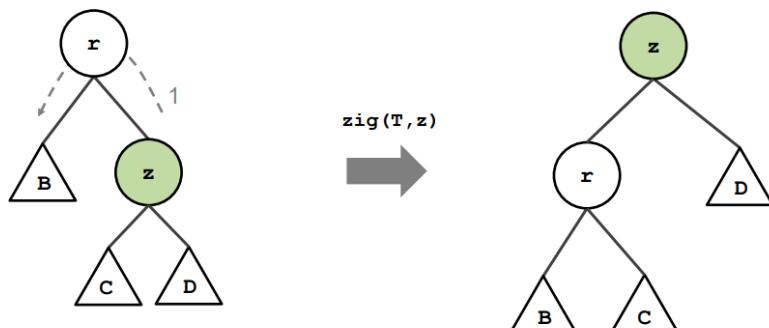
Zig-Zig-Operation

=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

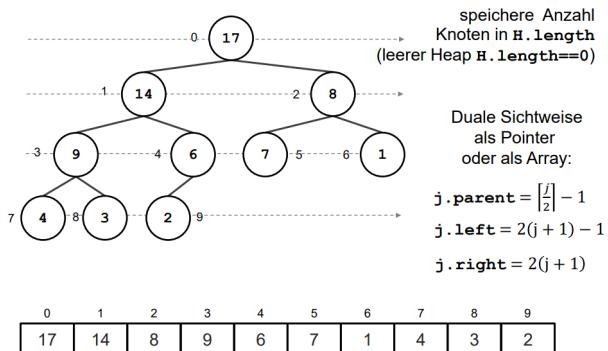
- Amortisierte Laufzeit: Laufzeit pro Operation über mehrere Operationen hinweg
- Worst-Case-Laufzeit pro Operation: $O(\log_n n)$

4.4 Binäre Max-Heaps

Definition – Binäre Max-Heaps

- Heaps sind keine BSTs
- Eigenschaften von binären 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 Einfügen und danach Vertauschen nach oben, bis Max-Eigenschaft wieder erfüllt ist

Code:

```
insert(H,k) // als unbeschränktes Array
1  H.length = H.length + 1;
2  H.A[H.length-1] = k;
3  i = H.length - 1;
4  WHILE i > 0 AND H.A[i] > H.A[i.parent]
5      SWAP(H.A, i, i.parent);
6      i = i.parent;
```

Laufzeit: $O(h) = O(\log n)$

Lösche Maximum

1. Ersetze Maximum durch „letztes“ Blatt
2. Vertausche Knoten durch Maximum der beiden Kinder (**heapify**)

```
extract-max(H)
1 IF isEmpty(H) THEN return error "underflow";
2 ELSE
3     max = H.A[0];
4     H.A[0] = H.A[H.length - 1];
5     H.length = H.length - 1;
6     heapify(H, 0);
7     return max;
```

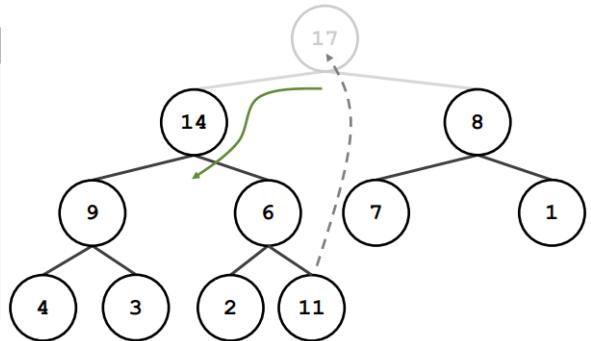


Abbildung 27: Beispiel Löschen des Maximums im Binären Max-Heap

```
heapify(H, i)
1 maxind = i;
2 IF i.left < H.length AND H.A[i]<H.A[i.left] THEN
3     maxind = i.left;
4 IF i.right < H.length AND H.A[maxind]<H.A[i.right] THEN
5     maxind = i.right;
6 IF maxind != i THEN
7     SWAP(H.A, i, maxind);
8     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
1 H.length = A.length;
2 FOR i = ceil((H.length-1)/2) - 1 DOWNT0 0 DO
3     heapify(H.A, i);
```

Heap-Sort

- Idee: Bauen des Heaps aus Array und dann (wiederholte) Extraktion des Maximums

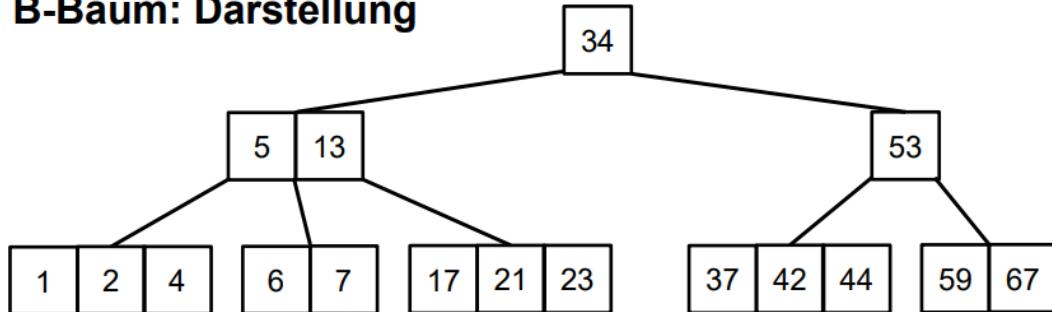
```
heapSort(H.A)
1 buildHeap(H.A)           // Bauen des Heaps
2 WHILE !isEmpty(H) DO
3     PRINT extract-max(H); // Ausgabe des Maximums bis Heap leer ist
```

4.5 B-Bäume

Definition – B-Baum

- Jeder B-Baum hat einen angegebenen Grad also z.B. $t = 2$
- Eigenschaften:
 - Wurzel zwischen $[1, \dots, 2t - 1]$ Werte
 - Knoten zwischen $[t - 1, \dots, 2t - 1]$ 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:
 $k_0 \leq key[0] \leq k_1 \leq key[1] \leq \dots \leq k_{n-1} \leq key[n - 1] \leq k_n$

B-Baum: Darstellung



x.n

- Anzahl Werte eines Knoten **x**

x.key[0], ..., x.key[x.n-1]

- (geordnete) Werte in Knoten **x**

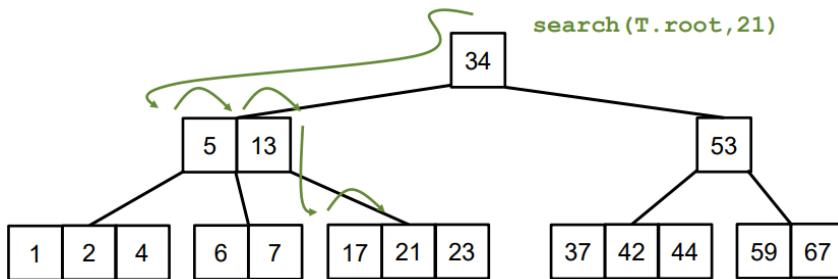
x.child[0], ..., x.child[x.n]

- Zeiger auf Kinder in Knoten **x**

- 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)

```

1 WHILE x != nil DO
2   i = 0;
3   WHILE i < x.n AND x.key[i] < k DO
4     i++;
5   IF i < x.n AND x.key[i] == k THEN
6     return(x, i);
7   ELSE
8     x = x.child[i];
9   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 vollen 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)

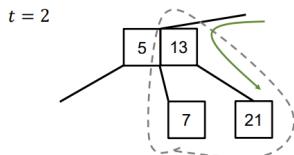
insert(T, z)

```

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

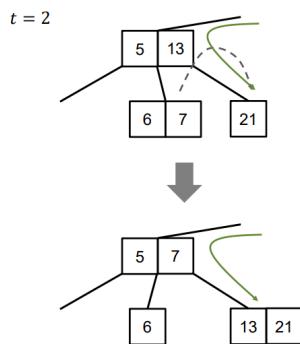
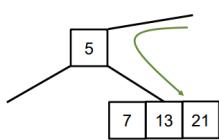
Löschen

- 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



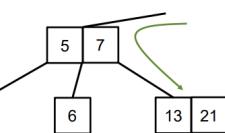
Allgemeines Verschmelzen:

- Kind und rechter und linker Geschwisterknoten (sofern existent) 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)
1 Wenn Wurzel nur 1 Wert und beide Kinder t-1 Werte,
2 verschmelze Wurzel und Kinder (reduziert Höhe um 1)
3 Suche rekursiv Löschposition:
4   Wenn zu besuchendes Kind nur t-1 Werte,
5     verschmelze es oder rotiere/verschiebe
6   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
- \mathcal{O} -Notation versteckt hier konstanten Faktor t für Suche innerhalb eines Knotens

5 Randomized Data Structures

5.1 Skip Lists

Idee – skip Lists

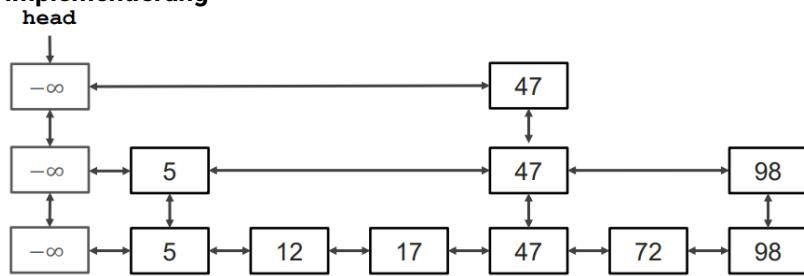
- Einfügen von „Express-Liste“ mit einigen Elementen
- Beginne mit Suche in der Express-Liste mit weniger Elementen
- Falls das suchende Element kleiner als nächstes Element in Express-Liste \Rightarrow weiter nach rechts
- Falls nicht \Rightarrow Eine Stufe nach unten wandern und dort weiter suchen

Mögliche 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: $n \Rightarrow pn \Rightarrow p^2n \Rightarrow \dots$ (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

Abbildung 28: Beispiel Skip List

Suche

Laufzeit ist von Expresslisten abhängig

```
search(L, k)
1 current = L.head;
2 WHILE current != nil DO
3     IF current.key == k THEN
4         return current;
5     IF current.next != nil AND current.next.key <= k THEN
6         current = current.next;
7     ELSE
8         current = current.down;
9 return nil;
```

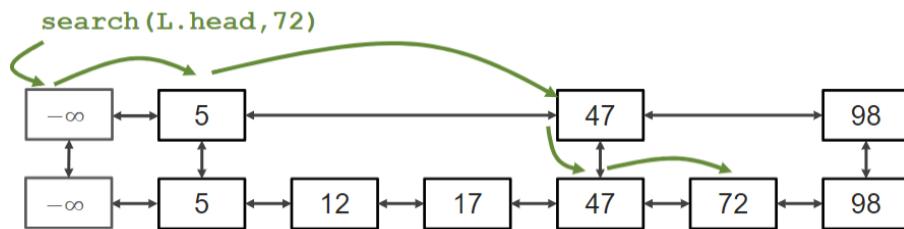


Abbildung 29: Beispiel Suche in einer Skip List

Einfügen

- Füge auf unterster Ebene ein
- Evtl. auf höheren Ebenen mit zufälliger Wahl mithilfe von p auf jeder Ebene
- falls ein Element nicht auf die nächst höhere Ebene gelangt, gelangt es auch nicht auf andere höhere Ebenen (Abbruch des Auswahlprozesses)

Löschen

- Entferne Vorkommen des Elements aus allen Ebenen

Laufzeiten

Einfügen $\Theta(\log_{\frac{1}{p}} n)$

Löschen $\Theta(\log_{\frac{1}{p}} n)$

Suchen $\Theta(\log_{\frac{1}{p}} n)$

- O -Notation versteckt konstanten Faktor $\frac{1}{p}$

- Speicherbedarf im Durchschnitt: $\frac{n}{1-p}$

5.2 Hashtables

Idee – Hashtable

- 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

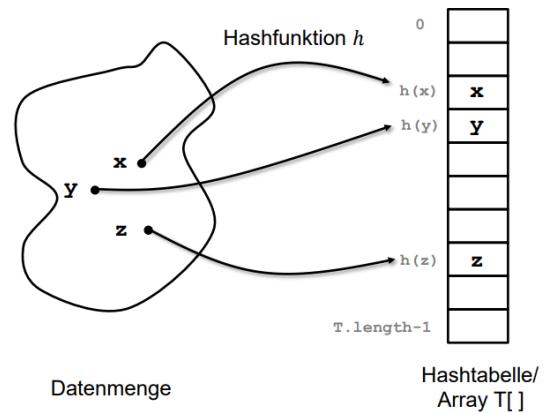


Abbildung 30: Beispiel Hashfunktion

Hash-Funktionen

Universelle Hash-Funktion

- Wähle zufällige $a, b \in [0, p - 1]$, p prim, $a \neq 0$
- $h_{a,b}(x) = ((a \cdot x + b) \text{ mod } p) \text{ mod } T.length$

Kryptographische Hash-Funktionen

- MD5, SHA-1, SHA-2, SHA-3
- $h(x) = MD5(x) \text{ mod } T.length$

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

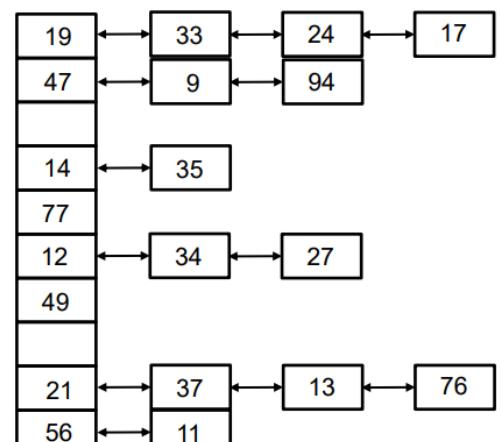
Laufzeiten

Einfügen $\Theta(1)$

Löschen $\Theta(1)$

Suchen $\Theta(1)$

- Für $T.length = n$ ergibt sich konstante Laufzeit
- (Im Durchschnitt, beim Einfügen sogar im Worst-Case)
- Speicherbedarf i.d.R. höher als n , meist ca. $1,33 \cdot n$



Hashtabelle/
Array $T[]$

Abbildung 31: Beispiel Hashtabelle

5.3 Bloom-Filter

Idee – Bloom-Filter

Speicherschonende Wörterbücher mit kleinem Fehlerpotenzial 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, \dots, x_{n-1}
- m Bits-Speicher z.B. als Bit-Array
- k gute Hash-Funktionen H_0, \dots, H_{k-1} mit Bildbereich $0, 1, \dots, m - 1$
- Empfohlene Wahl: $k = \frac{m}{n} \cdot \ln 2$ (Fehlerrate von ca. 2^{-k})

Code:

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

```

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

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

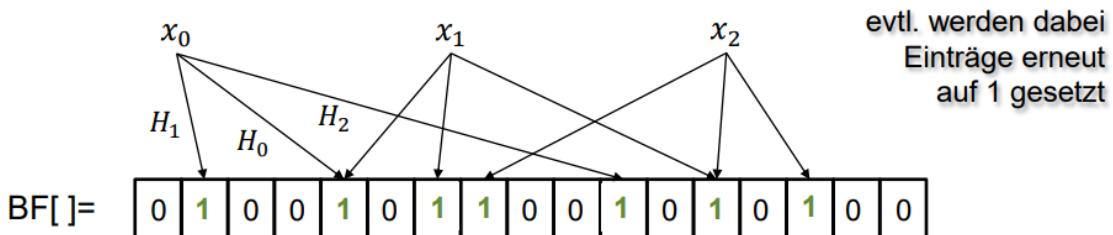


Abbildung 32: Beispiel Bloom Filter

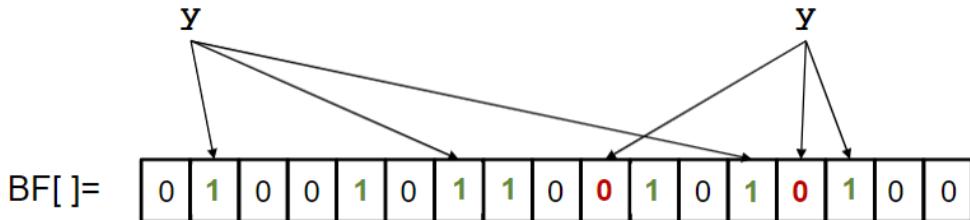
Suche

```
searchBloom(BF, H, y)
```

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

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

in Wörterbuch:



nicht in Wörterbuch:

Abbildung 33: Beispiel Suche im Bloom Filter

- 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

Definition – (Endlicher) gerichteter Graph

- (endlicher) gerichteter Graph $G = (V, E)$
- besteht aus (endlicher) Knotenmenge V
- besteht aus (endlicher) Kantenmenge $E \subseteq V \times V$
- $(u, v) \in E$: Kanten von Knoten u zu v
- Kanten haben eine Richtung

Definition – Ungerichteter Graph

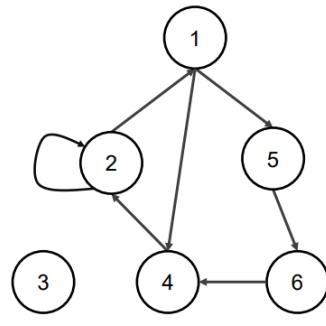
- (endlicher) ungerichteter Graph $G = (V, E)$
- besteht aus (endlicher) Knotenmenge V
- besteht aus (endlicher) Kantenmenge $E \subseteq V \times V$, sodass $(u, v) \in E \Leftrightarrow (v, u) \in E$
- Kanten haben keine Richtung

Darstellung von Graphen

- Als Adjazenzmatrix (1, wenn Kante von i zu j bzw. 0, wenn keine Kante)
 - Bei ungerichteten Graphen ist Matrix spiegelsymmetrisch zur Hauptdiagonalen
- \Rightarrow Speicherbedarf: $\Theta(|V|^2|)$

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

(a) Darstellung als Adjazenzmatrix



(b) Grafische Darstellung

Abbildung 34: Beispielhafte Darstellung eines Graphen

- Auch darstellbar als Array mit verketteten Listen
- \Rightarrow Speicherbedarf: $\Theta(|V| + |E|)$

Pfadfinder

- Knoten v ist von Knoten u erreichbar, wenn es von u aus einen Pfad über n Knoten nach v gibt
- u ist immer von u per leerem Pfad ($k=1$) erreichbar
- Länge des Pfades = $k - 1$ = Anzahl Kanten

Definition – Zusammenhängende Graphen

- 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 $E' \subseteq E$.

Definition – Gewichtete Graphen

- gewichteter gerichteter oder ungerichteter Graph $G = (V, E)$
- besitzt zusätzlich Funktion $w : E \rightarrow R$
- Angabe des Gewichts einer Kante u nach v durch $w((u, v))$

6.2 Breadth-First Search (BFS)

Idee – Breadth-First Search

- 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

```

1  BFS(G, s) //G=(V, E) s = source node in V
2  FOREACH u in V-{s} DO
3      u.color = WHITE;           // Weiß = noch nicht besucht
4      u.dist = +∞;             // Setzen der Distanzen auf Unendlich
5      u.pred = nil;            // Setzen der Vorgänger auf nil
6  s.color = GRAY;              // Anfang bei Startnode
7  s.dist = 0;
8  s.pred = nil;
9  newQueue(Q);
10 enqueue(Q, s);
11 WHILE !isEmpty(Q) DO
12     u = dequeue(Q);
13     FOREACH v in adj(G, u) DO
14         IF v.color == WHITE THEN
15             v.color == GRAY;
16             v.dist = u.dist+1;
17             v.pred = u;
18             enqueue(Q, v);
19     u.color = BLACK;          // Knoten abgearbeitet

```

Farben:

- **WHITE**: Knoten noch nicht besucht
- **GRAY**: Knoten in Queue für nächsten Schritt
- **BLACK**: Knoten ist fertig

- 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
```

```

1  IF v == s THEN
2      print s;
3  ELSE
4      IF v.pred == nil THEN
5          print "no path from s to v"
6      ELSE
7          print-path(G,s,v.pred);
8          print v;
```

Abgeleiteter BFS-Baum

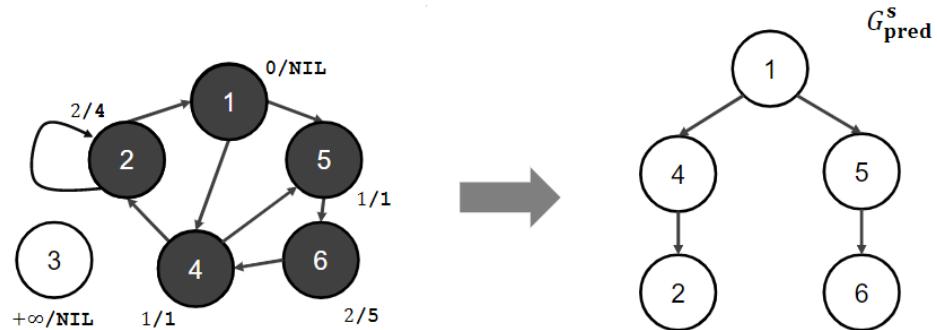


Abbildung 35: Beispiel BFS-Baum

- Subgraph $G_{pred}^s = (V_{pred}^s, E_{pred}^s)$ von G :
 - $V_{pred}^s = \{v \in V | v.pred \neq \text{nil}\} \cup \{s\}$
 - $E_{pred}^s = \{(v.pred, v) | v \in V_{pred}^s - \{s\}\}$
- G_{pred}^s 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 – Depth-First Search

- Besuche zuerst alle noch nicht besuchten Nachfolgeknoten

- "Laufe so weit wie möglich weg vom aktuellen Knoten"

Algorithmus

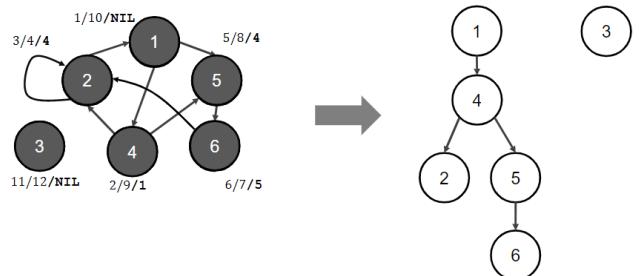
```
DFS(G)
1  FOREACH u in V DO
2      u.color = WHITE;
3      u.pred = nil;
4      time = 0;           // time hier als globale Variable
5  FOREACH u in v DO
6      IF u.color == WHITE THEN
7          DFS-VISIT(G,u) // Start eines rekursiven Aufrufs
```

DFS-VISIT(G,u)

```
1  time = time + 1;
2  u.disc = time;           // discovery time
3  u.color = GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color == WHITE THEN
6          v.pred = u;
7          DFS-VISIT(G,v);
8  u.color = BLACK;
9  time = time + 1;
10 u.finish = time;        // finish time
```

DFS-Wald = Menge von DFS-Bäumen

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



Kantenarten

Baumkanten alle Kanten in G_{pred}

Abbildung 36: Beispiel DFS-Wald

Vorwärtskanten alle Kanten in G zu Nachkommen in G_{pred} , die keine Baumkante sind

Rückwärtskanten alle Kanten in G zu Vorfahren in G_{pred} , die keine Baumkante sind (inkl. Schleifen)

Kreuzkanten alle anderen Kanten in G

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

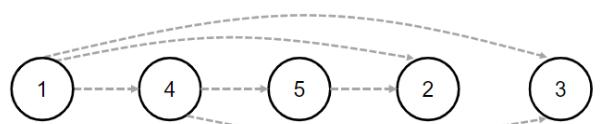


Abbildung 37: Beispiel Topologisches Sortieren

TOPOLOGICAL-SORT(G)

```

1 new LinkedList(L);
2 run DFS(G) but, each time a node is finished, insert in front of L
3 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

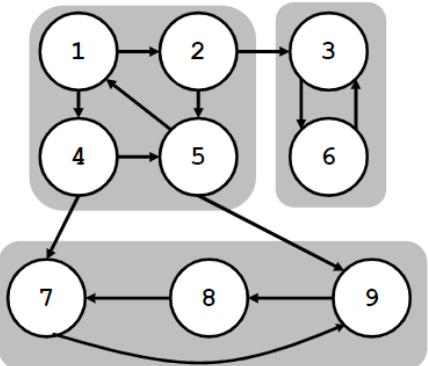


Abbildung 38: Beispiel Starke Zusammenhangskomponenten

Code:

SCC(G)

```

1 run DFS(G)
2 compute  $G^T$ 
3 run DFS( $G^T$ ) but visit vertices in main loop
4     in descending finish time from 1
5     output each DFS tree from above as one SCC

```

6.4 Minimale Spannbäume

Definition – Minimaler Spannbaum

- Verbindung aller Knoten miteinander
- Minimaler Spannbaum \Rightarrow Minimales Gewicht

Allgemeiner Algorithmus

genericMST(G, w)

```

1 A =  $\emptyset$ 
2 WHILE A does not form a spanning tree for G DO
3     find safe edge  $\{u, v\}$  for A
4     A = A  $\cup \{\{u, v\}\}$ 
5 return A

```

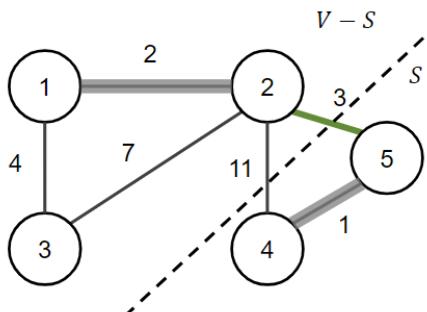


Abbildung 39: Beispiel Ausführung genericMST()

Terminologie:

- Schnitt $(S, V-S)$ partitioniert 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

Idee – Algorithmus von Kruskal Suchen der „kleinsten“ Kante und Zusammenfügen von Mengen, falls Mengen ungleich sind

- Lässt parallel mehrere Unterbäume eines MST wachsen
- Laufzeit: $O(|E| \cdot \log |E|)$

MST-Kruskal(G, w)

```

1 A =  $\emptyset$ 
2 FOREACH v in V DO
3     set(v) = {v};           // Menge mit sich selbst
4 Sort edges according to weight in nondecreasing order
5 FOREACH  $\{u, v\}$  in E according to order DO
6     IF set(u) != set(v) THEN    // Mengen noch nicht verbunden
7         A = A  $\cup \{\{u, v\}\}$ ;
8         UNION(G, u, v);        // Zusammenführen der Mengen aller Knoten aus den Sets
9 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

```

1  FOREACH v in V DO
2      v.key = +∞;
3      v.pred = nil;
4  r.key = -∞
5  Q = V;
6  WHILE !isEmpty(Q) DO
7      u = EXTRACT-MIN(Q);      // smallest key value
8      FOREACH v in adj(u) DO
9          IF v∈Q and w({u,v})<v.key THEN
10             v.key = w({u,v});
11             v.pred = u;

```

6.5 Kürzeste Wege in (gerichteten) Graphen

Definition – SSSP

- SSSP - Single-Source Shortest Path
- Von Quelle s ausgehend die kürzesten Pfade zu allen anderen Knoten
- Kürzester Pfad: Pfad mit minimalem Gesamtgewicht 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)

```

1  IF v.dist > u.dist + w((u,v)) THEN
2      v.dist = u.dist + w((u,v));
3      v.pred = u;

```

Bellman-Ford-Algorithmus

Laufzeit: $\Theta(|E| \cdot |V|)$

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

```

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

initSSSP(G,s,w)

```

1  FOREACH v in V DO
2      v.dist = ∞;
3      v.pred = nil;
4  s.dist = 0;
```

TopoSort für dag

Idee – TopoSort für dag Erhalten des kürzesten Pfades durch das topologische Sortieren

Laufzeit: $\Theta(|E| + |V|)$

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

```

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

Dijkstra-Algorithmus

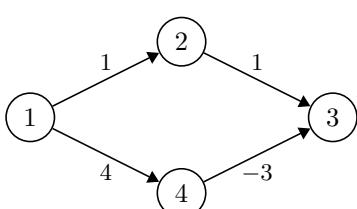
Voraussetzung: Keine negativen Kantengewichte

Laufzeit: $\Theta(|V| \cdot \log|V| + |E|)$

Dijkstra-SSSP(G,s,w)

```

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



- Beispiel für Problem mit negativen Kantengewichten bei Dijkstra: Dijksta würde Pfad 1-2-3 liefern, da das Kantengewicht 4 größer als der andere Pfad ist.

6.6 Maximaler Fluss in Graphen

Idee – Maximaler Fluss im Graphen

- 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

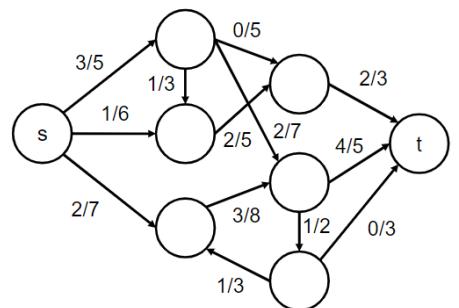


Abbildung 40: Beispiel Flussnetzwerk

Definition – 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 $|E| \geq |V| - 1$.

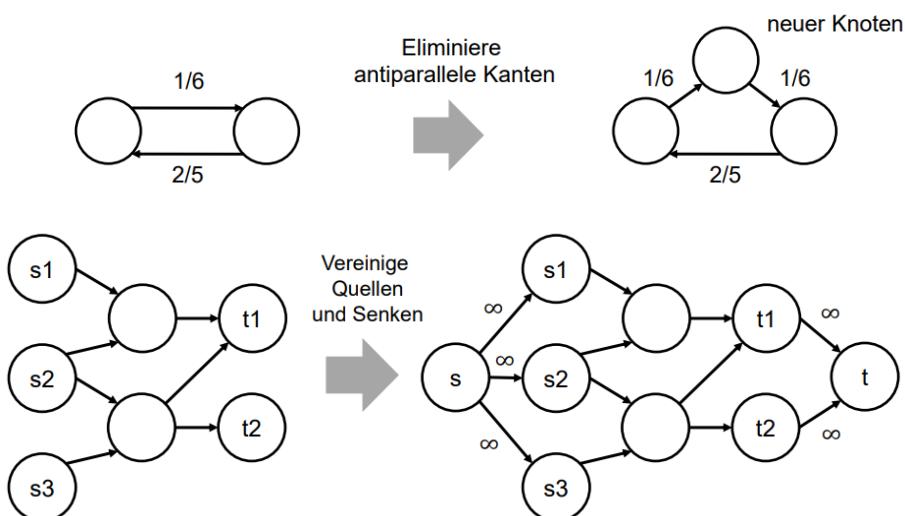
Definition – Fluss

Ein Fluss $f : V \times V \rightarrow \mathbb{R}$ 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)

Definition – Wert eines Flusses Der Wert $|f|$ eines Flusses $f : V \times V \rightarrow \mathbb{R}$ 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 $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$ Suche eines Pfades von s nach t und Erhöhung aller Flüsse um

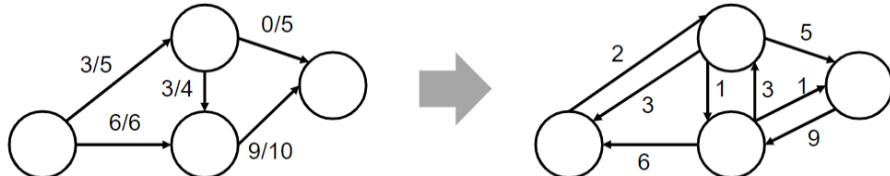


Abbildung 41: Beispiel Restkapazitätsgraph

niedrigsten möglichen Wert auf Pfad

Ford-Fulkerson-Algorithmus

Idee – Ford-Fulkerson-Algorithmus

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

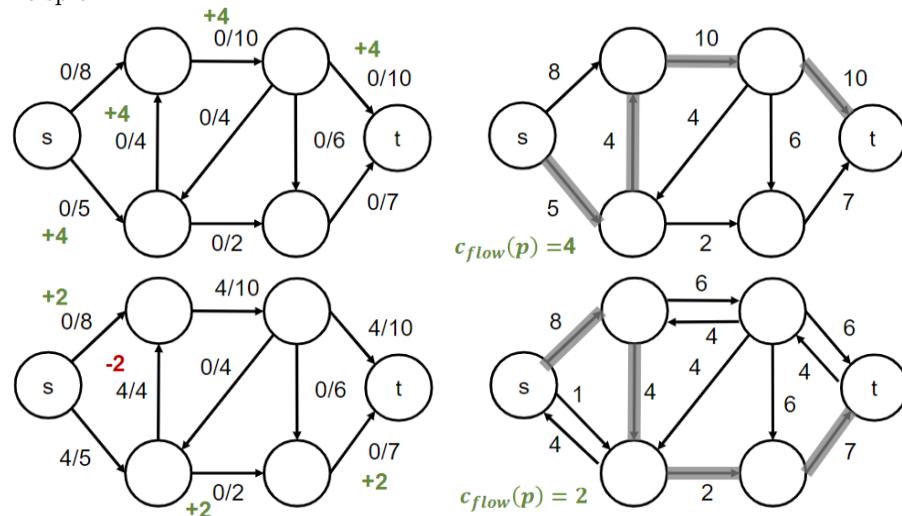
Code:

```

Ford-Fulkerson(G,s,t,c)
1 FOREACH e in E do e.flow = 0;
2 WHILE there is path p from s to t in Gflow DO
3   cflow(p) = min {cflow(u,v) : (u,v) in p}
4   FOREACH e in p DO
5     IF e in E THEN
6       e.flow = e.flow + cflow(p);
7     ELSE
8       e.flow = e.flow - cflow(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:

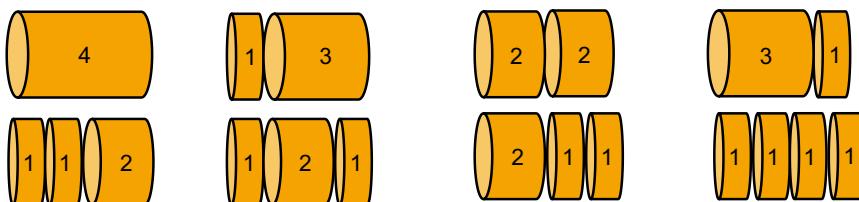
1. Wir charakterisieren die Struktur einer optimalen Lösung
2. Wir definieren den Wert einer optimalen Lösung rekursiv
3. Wir berechnen den Wert einer optimalen Lösung (meist bottom-up Ansatz)
4. Wir konstruieren eine zugehörige optimale Lösung aus berechneten Daten

7.1.1 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 Stange

- 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: $n = i_1 + i_2 + \dots + i_k$
- maximaler Erlös: $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$
- z.B.: $r_4 = 10$ (siehe oben)

Rekursive Top-Down Implementierung

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

```

1 IF n == 0
2   return 0;
3 q = -∞;
4 FOR i = 1 TO n // nicht Start bei 0, sonst kein Rekursionsschritt
5   q = max(q, p[i] + CUT-ROD(p, n - i));
6 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 dieselben Teilprobleme löst.

Ansatz Jedes Teilproblem nur einmal lösen. Falls die Lösung eines Teilproblems nochmal benötigt wird, schlagen wir diese nach.

- Reduktion von exponentieller auf polynomielle Laufzeit.
- Dynamische Programmierung wird zusätzlichen Speicherplatz benutzen um Laufzeit einzusparen.

Rekursiver Top-Down-Ansatz mit Memoisation:

Idee – Rekursiver Top-Down-Ansatz mit Memoisation Speicherung der Lösungen der Teilprobleme

Laufzeit: $\Theta(n^2)$

MEMOIZED-CUT-ROD(p, n)

```

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

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

```

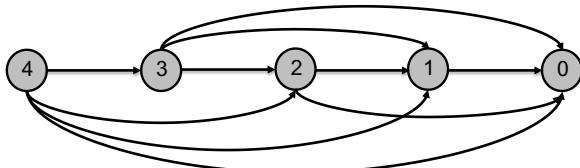
1 IF r[n] ≥ 0                                // Abfrage ob vorhanden
2   return r[n]
3 IF n == 0
4   q = 0
5 ELSE
6   q = -∞
7   FOR i = 1 to n
8     q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
9   r[n] = q                                    // Abspeichern
10  return q
```

Bottom-Up Ansatz:

- Laufzeit: $\Theta(n^2)$
- Sortieren der Teilprobleme nach ihrer Größe und lösen in dieser Reihenfolge
- Alle Teilprobleme kleiner als das momentane Problem sind bereits gelöst

BOTTOM-UP-CUT-ROD(p, n)	EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
<pre> 1 Let r[0...n] be a new array 2 r[0] = 0 3 FOR j = 1 TO n 4 q = -∞ 5 FOR i = 1 TO j 6 q = max(q, p[i] + r[j - i]) 7 r[j] = q 8 return r[n] </pre>	<pre> 1 Let r[0...n] and s[0...n] be new arrays 2 r[0] = 0, s[0] = 0 3 FOR j = 1 TO n 4 q = -∞ 5 FOR i = 1 TO j 6 IF q < p[i] + r[j-i] 7 q = p[i] + r[j - i] 8 s[j] = i 9 r[j] = q 10 return r and s </pre>

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

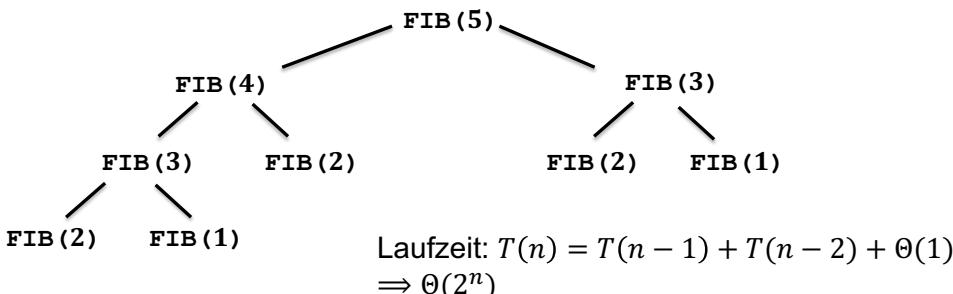


Fibonacci-Zahlen

- $F_1 = F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$

Naiver rekursiver Algorithmus:

FIB(n)
1 IF n ≤ 2
2 f = 1;
3 ELSE
4 f = FIB(n-1) + FIB(n-2);
5 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)

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

MEMOIZED-FIB-AUX(n, m)

```
1 IF m[n-1] != 0
2     return m[n-1];           // Auslesen von gespeicherten Werten
3 IF n <= 2
4     f = 1;
5 ELSE
6     f = MEMOIZED-FIB-AUX(n-1, m) + MEMOIZED-FIB-AUX(n-2, m);
7 m[n-1] = f;
8 return f;
```

Bottom-Up Algorithmus:

Hier wieder Berechnen aller Teilprobleme von unten beginnend

BOTTOM-UP-FIB(n)

```
1 Let m[0...n] be a new array
2 FOR i = 1 TO n
3     IF i <= 2
4         f = 1;
5     ELSE
6         f = m[i-1] + m[i-2];
7     m[i] = f;
8 return m[n];
```

7.2 Greedy-Algorithmus

Idee – Greedy-Algorithmus

- Trifft stets die Entscheidung, die in diesem Moment am besten erscheint
- Trifft **lokale** optimale Entscheidung (evtl. nicht global die Beste)

7.2.1 Aktivitäten-Auswahl-Problem

Definition – Aktivitäten-Auswahl-Problem

- 11 anstehende Aktivitäten $S = \{a_1, \dots, a_{11}\}$
- 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 Aktivitäten 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)
1 // s Anfangszeitenarray, f Endzeitenarray,
2 // k Index von Teilproblem, n Größe Anfangsproblem
3 m = k + 1;
4 WHILE m ≤ n and s[m] < f[k] // Suche nach erster Kompatibilität
5     m = m + 1;
6 IF m ≤ n
7     // Ausgabe des Elements und Berechnung weiterer Aktivitäten
8     return {am} ∪ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)
9 ELSE
10    return ∅
```

Iterativer Greedy-Algorithmus

Voraussetzung: Aktivitäten sind monoton steigend nach der Endzeit sortiert

Laufzeit: $\Theta(n)$

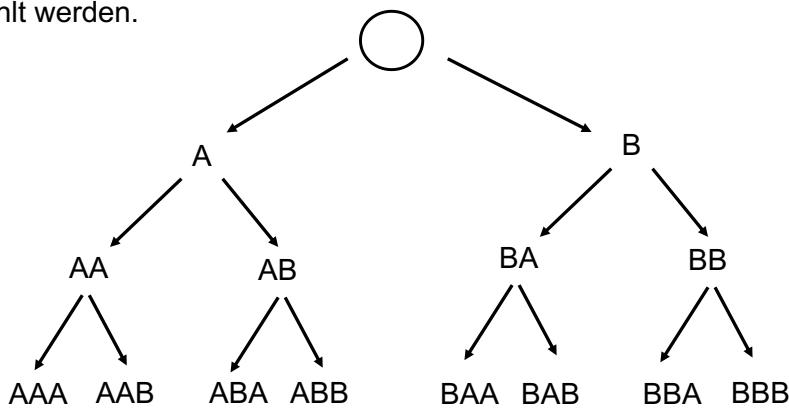
```
GREEDY-ACTIVITY-SELECTOR(s,f)
1 n = s.length;
2 A = {a1};
3 k = 1;           // Index des zuletzt hinzugefügten Elements in A
4 FOR m = 2 TO n
5     IF s[m] ≥ f[k]      // Findet zuerst endende Aktivität in Menge
6         A = A ∪ {am}; // Fügt diese hinzu, falls kompatibel
7         k = m;
8 return A;
```

7.3 Backtracking

Suchbaum - Baum der Möglichkeiten

Darstellung aller für ein Problem bestehenden Möglichkeiten

Problem: Aus den Buchstaben A, B soll dreimal nacheinander einer gewählt werden.



Der Suchraum ist die Menge aller für ein Problem bestehende Möglichkeiten.

Idee – Backtracking

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

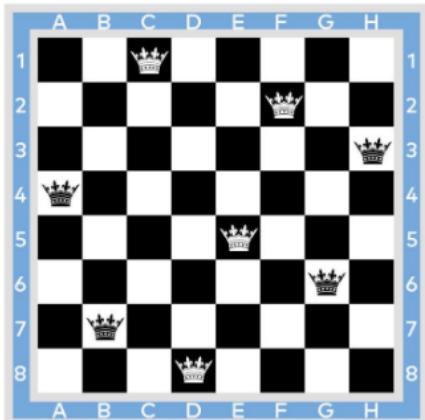
Allgemeiner Backtracking-Algorithmus

```

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

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)

Abbildung 42: Beispielhafte Darstellung des Damenproblems

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

```

1 IF r == n
2     return Q;
3 ELSE
4     FOR j = 0 TO n - 1 // Mögliche Positionierungen
5         legal = true;
6         FOR i = 0 TO r - 1 // Evaluation der mgl. Bedrohungen
7             IF (Q[i] == j) OR (Q[i] == j + r - i) OR (Q[i] == j - r + i)
8                 legal = false;
9         IF legal == true
10            Q[r] = j;
11            PLACE-QUEENS(Q, r + 1)

```

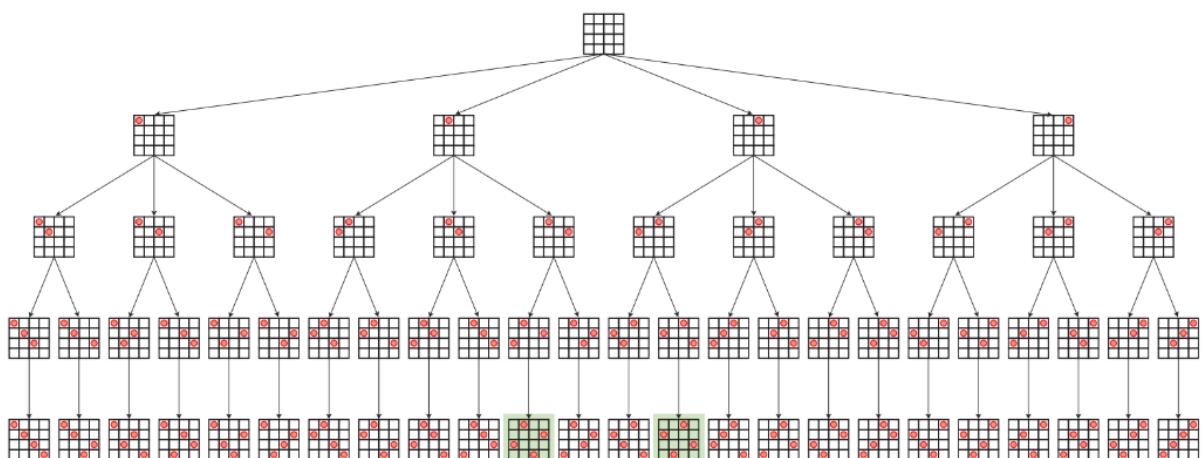
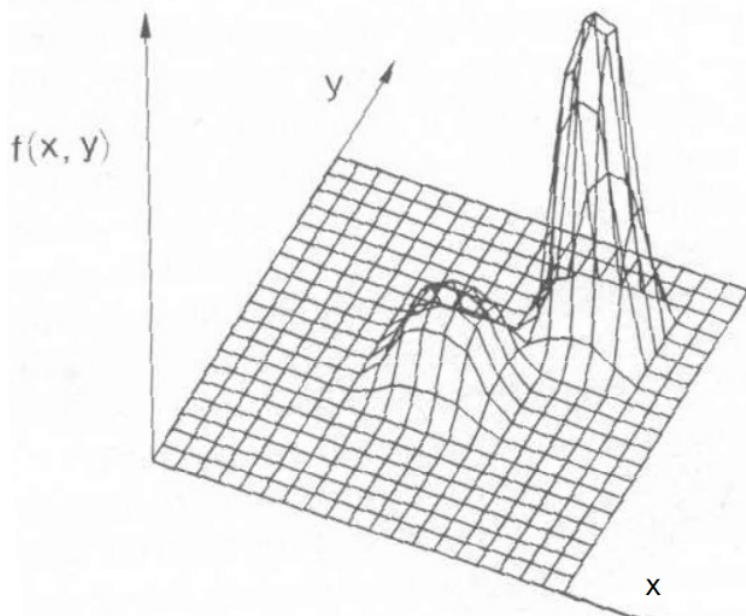


Abbildung 43: Mögliche Pfade von Place-Queens

7.4 Metaheuristiken

Optimierungsproblem



- Lösungsstrategien:
 - Exakte Methode
 - Approximationsmethode
 - Heuristische Methode
 - Einschränkungen
 - Antwortzeit
 - Problemgröße
- ⇒ exkludieren oft exakte Methoden

Abbildung 44: Beispiel Optimierungsproblem

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

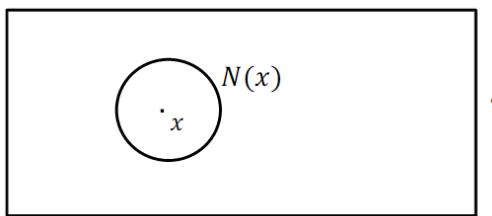
Ziel: Höchster Wert der Gegenstände im Rucksack

Beispiel:

	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

Abbildung 45: Beispielgegenstände für Rucksackproblem

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



Nachbarschaft:

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

Zufällige Suche

Idee – Zufällige Suche

- 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 \Rightarrow als neue **aktuelle** Lösung setzen
- Terminierung, falls keine weiteren Verbesserungen auffindbar oder Zeit vorbei

Code:

```
RANDOM-SEARCH
1 best <- irgendeine initiale zufällige Lösung
2 REPEAT
3     S <- zufällige Lösung // von "best" unabhängig
4     IF (Quality(S) > Quality(best)) THEN
5         best <- S
6 UNTIL best ist die ideale Lösung oder Zeit ist vorbei
7 return best
```

- Nachteile
- Potentiell lange Laufzeit
 - Laufzeit abhängig von der initialen Konfiguration

- Vorteile
- Algorithmus **kann** beim globalen Optimum terminieren

Bergsteigeralgorithmus

Idee – Bergsteigeralgorithmus

- 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 auffindbar oder Zeit vorbei

Code

HILL-CLIMBER

```
1 T <- Distribution von möglichen Zeitintervallen
2 S <- irgendeine initiale zufällige Lösung
3 best <- S
4 REPEAT
5   time <- zufälliger Zeitpunkt in der Zukunft aus T
6   REPEAT
7     wähle R aus der Nachbarschaft von S
8     IF Quality(R) > Quality(S) THEN
9       S <- R
10    UNTIL S ist ideale Lösung oder time ist erreicht oder totale Zeit erreicht
11    IF Quality(S) > Quality(best) THEN
12      best <- S
13    S <- irgendeine zufällige Lösung
14  UNTIL best ist die ideale Lösung oder totale Zeit erreicht
15  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 – Iterative lokale Suche

- 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

```
1 T <- Distribution von möglichen Zeitintervallen
2 S <- irgendeine initiale zufällige Lösung
3 H <- S      // Wahl des Homebasepunktes
4 best <- S
5 REPEAT
6     time <- zufälliger Zeitpunkt in der Zukunft aus T
7     REPEAT
8         wähle R aus der Nachbarschaft von S
9         IF Quality(R) > Quality(S) THEN
10            S <- R
11        UNTIL S ist ideale Lösung oder time ist erreicht oder totale Zeit erreicht
12        IF Quality(S) > Quality(best) THEN
13            best <- S
14        H <- NewHomeBase(H,S)
15        S <- Perturb(H)
16    UNTIL best ist die ideale Lösung oder totale Zeit erreicht
17    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 – Simulated Annealing

- 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

SIMULATED ANNEALING

```

1 t <- Temperatur, initial eine hohe Zahl
2 S <- irgendeine initiale zufällige Lösung
3 best <- S
4 REPEAT
5     wähle R aus der Nachbarschaft von S
6     IF Quality(R) > Quality(S) oder zufälliges
7         Z ∈ [0, 1] < e^{\frac{Quality(R) - Quality(S)}{t}} THEN
8         S <- R
9     dekrementiere t
10    IF Quality(S) > Quality(best) THEN
11        best <- S
12 UNTIL best ist die ideale Lösung oder Temperatur ≤ 0
13 return best

```

Tabu-Search

Idee – Tabu-Search

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

TABU-SEARCH

```

1 l <- maximale Größe der Tabu List
2 n <- Anzahl der zu betrachtenden Nachbarschaftslösungen
3 S <- irgendeine initiale zufällige Lösung
4 best <- S
5 L <- {} Tabu List der Länge l
6 Füge S in L ein
7 REPEAT
8     IF Length(L) > l THEN
9         Entferne ältestes Element aus L
10        wähle R aus Nachbarschaft von S
11        FOR n - 1 mal DO
12            Wähle W aus Nachbarschaft von S
13            IF W ∉ L und (Quality(W) > Quality(R)) oder R ∈ L) THEN
14                R <- W
15            IF R ∉ L THEN
16                S <- R
17                Füge R in L ein
18            IF Quality(S) > Quality(best) THEN
19                best <- S
20 UNTIL best ist die ideale Lösung oder totale Zeit erreicht
21 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 – Evolutionärer Algorithmus

- 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 Stichproben basierend auf vorherigen Resultaten

Abstrakter Code (Allgemeiner Breed und Join):

```
ABSTRACT-EVOLUTIONARY-ALGORITHM
1 P <- generiere initiale Population
2 best <- [] // leere Menge
3 REPEAT
4     AssesFitness(P)
5     FOR jedes individuelle  $P_i \in P$  DO
6         IF best = [] oder Fitness( $P_i$ ) > Fitness(best) THEN
7             best <-  $P_i$ 
8         P <- Join(P, Breed(P))
9     UNTIL best ist die ideale Lösung oder totale Zeit erreicht
10    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 ($O(n)$)

Idee – Evolutionsstrategie

- Generiere Population zufällig
- Beurteile Qualität jedes Individuums
- Lösche alle bis auf die μ besten Individuen
- Generiere $\frac{\lambda}{\mu}$ -viele Nachfahren pro bestes Individuum
- Join Funktion: Die Nachfahren ersetzen die Individuen

Algorithmus der Evolutionsstrategie

```
(μ, λ)-EVOLUTION-STRATEGY
1 μ <- Anzahl der Eltern (initiale Lösung)
2 λ <- Anzahl der Kinder
3 P <- {}
4 FOR λ-oft DO
5   P <- {neues zufälliges Individuum}
6 best <- □
7 REPEAT
8   FOR jedes individuelle  $P_i \in P$  DO
9     AssesFitness( $P_i$ )
10    IF best = □ oder Fitness( $P_i$ ) > Fitness(best) THEN
11      best <-  $P_i$ 
12    Q <- die  $\mu$  Individuen deren Fitness() am Größten ist
13    P <- {}
14    FOR jedes Element  $Q_j \in Q$  DO
15      FOR  $\frac{\lambda}{\mu}$ -oft DO
16        P <- P ∪ {MUTATE( $Q_j$ )}
17 UNTIL best ist die ideale Lösung oder totale Zeit erreicht
18 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 $x = 0$:

b_{k-1}	b_{k-2}	\dots	b_2	b_1	b_0
0	0	\dots	0	0	0

Inkrementieren eines Binärzählers

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

b_{k-1}	b_{k-2}	\dots	b_2	b_1	b_0
0	0	\dots	0	1	1

b_{k-1}	b_{k-2}	\dots	b_2	b_1	b_0
0	0	\dots	1	0	0

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

b_{k-1}	b_{k-2}	\dots	b_2	b_1	b_0
0	1	\dots	1	1	1

b_{k-1}	b_{k-2}	\dots	b_2	b_1	b_0
1	0	\dots	0	0	0

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:

b_4	b_3	b_2	b_1	b_0	Schrittosten	Gesamtkosten	b_4	b_3	b_2	b_1	b_0	Schrittosten	Gesamtkosten
0	0	0	0	0	0	0	0	0	1	0	1	+1	8
0	0	0	0	1	1	1	0	0	1	1	0	+1	10
0	0	0	1	0	2	3	0	0	1	1	1	+1	11
0	0	0	1	1	1	4	0	1	0	0	0	+1	15
0	0	1	0	0	3	7	0	1	0	0	1	+1	16
0	0	1	0	1	1	8	0	1	0	1	0	+1	18
b_4	b_3	b_2	b_1	b_0	Schrittosten	Gesamtkosten	b_4	b_3	b_2	b_1	b_0	Schrittosten	Gesamtkosten
0	1	0	1	0	+1	18	0	1	0	1	1	+1	19
0	1	0	1	1	1	19	0	1	1	0	0	+1	22
0	1	1	0	0	3	22	0	1	1	0	1	+1	23
0	1	1	0	1	1	23	0	1	1	1	0	+1	25
0	1	1	1	0	2	25	0	1	1	1	1	+1	26
0	1	1	1	1	1	26							

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: $\frac{2n}{n} = 2 \in O(1)$

- 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

Account Methode - Beispiel Binärzähler

Eigenschaften:

- Besteuerung einiger Operationen, so dass sie Kosten anderer Operationen mittragen
- Zuweisung von höheren Kosten (Amortisierte Kosten), als ihre tatsächlichen Kosten sind
- **Guthaben:** Differenz zwischen amortisierten und tatsächlichen Kosten
- Nutzung dieses Guthabens für Operationen bei denen amortisiert < tatsächlich gilt
- Guthaben darf nicht negativ werden:

Summe amortisierte Kosten > Summe tatsächliche Kosten

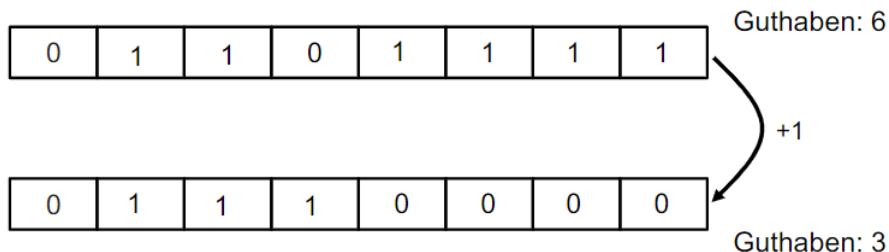
Wahl der Amortisierten Kosten - Binärzähler:

- Setzen eines Bits von 0 → 1 zahlt 2 Einheiten ein / Bezeichnung f_i
- Setzen eines Bits von 1 → 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 → 1: Zahlt 2 ein, kostet aber 1 → +1 Guthaben
- Setzen Bit 1 → 0: Zahlt 0 ein, kostet aber 1 → -1 Guthaben



Obere Schranken der Kosten:

- Guthaben auf dem Konto entspricht der Anzahl der auf 1 gesetzten Bits
- Kosten: $T(n) \sum_{i=1}^n t_i \leq v \sum_{i=1}^n a_i$, für ein konstantes v
- 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
→ $\phi(i)$ 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 \Rightarrow Manche nur in superpolynomieller Zeit lösbar
- Polynomielle Probleme: "einfach"
- Superpolynomielle Probleme: "hart"

Probleme die in
polynomieller
Zeit



einfach

super polynomieller
Zeit



lösbar sind, sind
hart

Abbildung 46: NP-Meme

Definition – 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, ...

Definition – 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: Sudoku, 3D-Matching,...
- Beispiel: Faktorisierungsproblem
 - Jede nicht-Primzahl kann eindeutig als Primzahlprodukt geschrieben werden
 - $35 = 5 \cdot 7$, $117 = 3 \cdot 3 \cdot 13$, ...
 - Faktorisieren auf klassischen Computern schwer
 - $n \xrightarrow{\text{schwer}} p, q$
 - $n, p, q \xrightarrow{\text{leicht}}$ ist $n = p \cdot q$?



- Rucksackproblem auch in polynomieller Laufzeit verifizierbar

Hamilton-Kreis-Problem

Definition – 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 \Rightarrow Matrix mit $n = m \times m$ Einträgen
 - $m!$ mögliche Permutationen der Knoten
 - $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$
 - \Rightarrow superpolynomielle Laufzeit (liegt **nie** in $O(n^k)$)
- **Allerdings:** Einfacher, wenn nur Beweis verifiziert werden muss
 - \Rightarrow Test, ob es sich um Permutation der Knoten handelt
 - \Rightarrow Test, ob alle angegebenen Kanten auf Kreis im Graphen existieren
 - \Rightarrow Verifikationsalgorithmus V mit quadratischer Laufzeit
- Verifikationsalgorithmus: $V(x, y) = 1/0$ (1, falls Kreis bzw. 0, falls nicht)
- Damit: Hamilton-Kreis $\in \text{NP}$

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

$$L \in P \longrightarrow L \in NP \longrightarrow P \subseteq NP$$

- 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 einfacher 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)

Definition – 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)

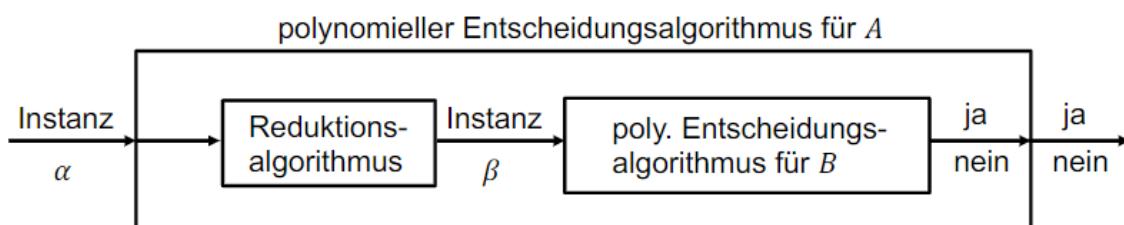
Definition – NP-Schwer Problem L ist **NP-schwer**, wenn $L' \leq_p L$ für alle $L' \in NP$

Definition – NP-Vollständig Problem L ist **NP-vollständig**, wenn L sowohl NP-schwer als auch in NP ist
z.B.: Hamilton-Kreis ist NP-vollständig

Reduktionen

Idee – Reduktion

- 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 polynomiale Zeit
 \Rightarrow Antworten sind gleich



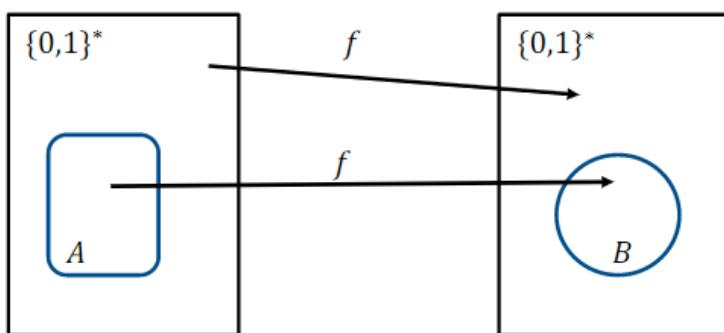
Beispiel:

- 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 Gleichungen
 - Lineare Gleichung $ax + b = 0 \Rightarrow x = \frac{-b}{a}$
 - Quadratische Gleichung $ax^2 + bx + c = 0 \Rightarrow x = \frac{-b}{a}, x = 0$
 - Quadratische Gleichung liefert also auch Lösung für lineare Gleichung

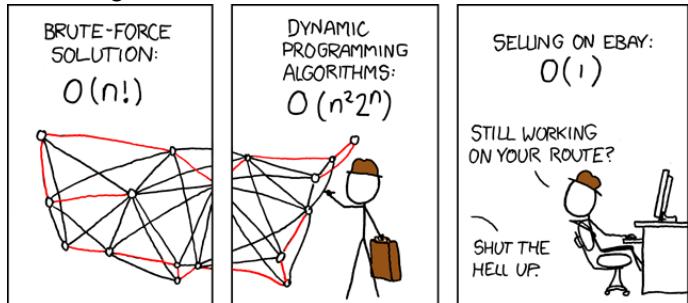
Formale Definition:

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

Illustration der Polynomialzeitreduktion:

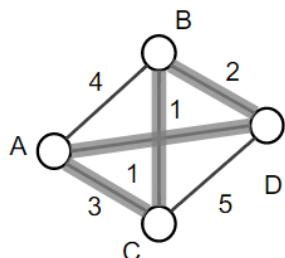


Travelling-Salesman Problem



Beschreibung:

- Reisender plant Rundreise durch mehrere Städte
- Start und Ziel ist eine vorgegebene Stadt
- Jede Stadt nur einmal besuchen
- Ziel: Minimale Reisekosten



Eine optimale Route mit Kosten 7 verläuft von $A \rightarrow D \rightarrow B \rightarrow C \rightarrow A$

Problem:

- Anzahl der verschiedenen 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

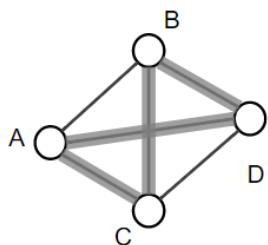
- Zeigen: TSP gehört zu NP und TSP ist NP-schwer

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

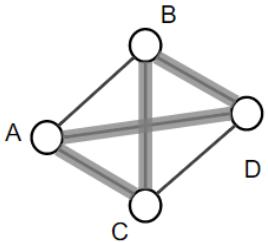
- Wir zeigen $HAM - KREIS \leq_p TSP$
- Start: Instanz von $HAM - KREIS$ mit $G = (V, E)$
- Konstruiere Instanz von TSP
 $\Rightarrow G' = (V, E')$ mit $E' = \{(i, j) : i, j \in V \text{ und } i \neq j\}$
- Definiere Kostenfunktion $c(i, j) = 0$, falls $(i, j) \in E$ / $c(i, j) = 1$, falls $(i, j) \notin E$
- 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.

Stichwortverzeichnis

Symbole

Ω -Notation	10
Θ -Notation	9
\mathcal{O} -Notation	10

A

Adjazenzmatrix	63
Advanced Data Structures	43
Advanced Designs.....	75
Algorithmen	3
Amortisierte Analyse.....	91
ancestor	33
Antisymmetrie	6
Array	5
Asymptotik	8
Asymptotische Komplexität	8
Asymptotische Notation	8
AVL-Bäume	49

B

B-Bäume	55
Backtracking.....	81
balanciert	49
Bellman-Ford	71
Bergsteigeralgorithmus	86
Binäre Bäume.....	33
Binäre Max-Heaps	53
Binäre Suchbäume	38
Bitcoin.....	25
Blatt	33
Bloom-Filter	61
Bottom-Up	77
Breadth-First Search (BFS)	65
Bubble Sort	14
buildHeap(H,A).....	54

C

call-by-reference	4
child	33
Combine	15
Conquer	15

D

Damenproblem	82
--------------------	----

Datenstrukturen	4
Depth-First Search(DFS).....	67
descendant	33
Determiniertheit	3
Determinismus	3
DFS-Wald	67
Dijkstra	71
Divide	15
Divide-And-Conquer	15
Dynamische Programmierung ..	75

E

Effizienz	3
Effizienz von Algorithmen	7
Eindeutigkeit	3
Elternknoten	33
Endlichkeit	3
Express-Liste	58
extract-max(H)	54

F

Fibonacci-Zahlen	77
FIFO	30
fixColorsAfterInsertion()	44
Fluss	72
Flussnetzwerk	72
Ford-Fulkerson	73

G

Geschwisterknoten	33
Gewichtete Graphen	64
Graph Algorithms	63
Graphen	63
Greedy	79
Grundlegende Datenstrukturen	25

H

Halbblatt	34
Hash-Funktion	60
• Kryptographisch	60
• universell	60
Hashtables	60
heapify(H,i)	54
HeapSort(H,A)	54
Heuristik	83
Höhe (Baum)	33

I

initBloom(X,BF,H)	61
inorder	34
Insertion Sort	12

K

Kindknoten	33
Komplexität	8
Komplexitätsklassen	11
Korrektheit	3
Kruskal	69
Kürzeste Wege	70

L

Laufzeit bei Rekursion	20
Laufzeitanalyse	7
leaf	33
LIFO	25
linkslastig	49

M

Mastertheorem	23
Maximaler Fluss	72
MaxSort	14
Merge Sort	16
Metaheuristiken	83
Minimale Spannbäume	69
MinSort	14

N

Nachkomme	33
NP	95

O

Optimierungsproblem	83
---------------------------	----

P

parent	33
Partition	18
Pfadfinder	64

Pivotelement	18	Totale Ordnung	6
postorder	34	Totalität	6
preorder	34	Transitivität	6
Prim	70	transplant	41, 46
Problem	3	Travelling-Salesman Problem ..	99
Probleminstanz	3	Traversieren von Bäumen	34
Pseudocode	4		
Q		V	
Queue	30	Verkettete Listen	28
Quicksort	18	Verschmelzen	57
R		Vorfahre	33
Randomized Data Structures ..	58		
rechtslastig	49	W	
Reduktion	98	Wurzel	33
Reflexivität	6		
Rekursionsbaum	21	Z	
Relax	70	Zig	52
Restkapazitätsgraph	73	Zig-Zag	51
root	33	Zig-Zig	52
Rot-Schwarz-Bäume	43	Zusammenhänge	64
rotateLeft()	44	• stark zusammenhängend ..	64
Rucksackproblem	84	• zusammenhängend ..	64
S		zyklisches Array	31
T			
Teilbaum	34		
Terminierung	3		
TopoSort	71		